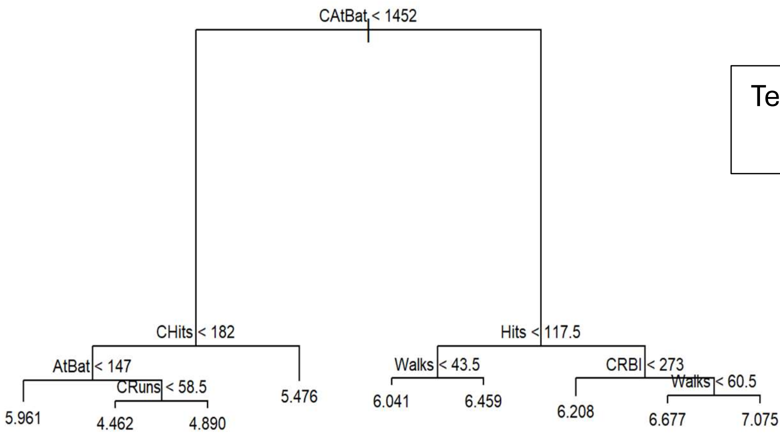
STAT 6340 Mini Project 6

Lakshmipriya Narayanan

SECTION 1: Observations and Answers

1

(a):



Test MSE using LOOCV (Decision tree) = 0.2545

Partitions of the predictor space for this decision tree with 9 terminal nodes where each terminal node's prediction is applied:

$R_1 = \{X|CAtBat < 1452, CHits < 182, AtBat < 147\}$
$R_2 = \{X|CAtBat < 1452, CHits < 182, AtBat \geq 147, CRuns < 58.5\}$
$R_3 = \{X|CAtBat < 1452, CHits < 182, AtBat \geq 147, CRuns \geq 58.5\}$
$R_4 = \{X|CAtBat < 1452, CHits \geq 182\}$
$R_5 = \{X|CAtBat \geq 1452, Hits < 117.5, Walks < 43.5\}$
$R_6 = \{X|CAtBat \geq 1452, Hits < 117.5, Walks \geq 43.5\}$
$R_7 = \{X|CAtBat \geq 1452, Hits \geq 117.5, CRBI < 273\}$
$R_8 = \{X|CAtBat \geq 1452, Hits \geq 117.5, CRBI < 273, Walks < 60.5\}$
$R_9 = \{X|CAtBat \geq 1452, Hits \geq 117.5, CRBI < 273, Walks \geq 60.5\}$

(b):

- Using LOOCV, we can conclude that pruning is not helpful because minimum deviance of 70.162 occurs at a pruned tree size of 9 terminal nodes which match our results in **(a)** above.
- **<u>Comparison</u>**: The best unpruned tree has a test MSE of 0.2545 and the best pruned tree has a test MSE of 0.2574. Since the unpruned tree has a lower MSE, it has performed better than the pruned tree indicating that pruning may have led to underfitting. Hence, pruning is not helpful.
- Estimated test MSE for the best pruned tree = 0.2574.
- Important predictors are CAtBat, AtBat, CRBI and Walks are the most important predictors because the first split is on CAtBat and it explains a significant amount of the deviance specifically from 207.2 to 36.22.

(c):

- Test MSE using LOOCV (Bagging) = 0.1775
- Important predictors are CAtBat, CRuns, CHits and, CRBI (observed from plot of important predictors since these predictors had longer bars and also high values for %IncMSE (Percentage Increase in MSE) and IncNodePurity (Increase in Node Purity).

(d):

- Test MSE using LOOCV (Random Forest) = 0.1799
- Important predictors are CAtBat, CRuns, CHits, CRBI and, CWalks (observed from plot of important predictors – same procedure as in **(b)** above.)

(e):

- Test MSE using LOOCV (Boosting) = 0.2212
- Important predictors are CAtBat, CRuns, CHits, and, CRBI (observed from plot in summary of boosting procedure.)

(f):

Question	Model	B	Given	Test MSE (LOOCV)
(a)	Decision Tree	--	Terminal nodes = 9	0.2545
(b)	Best Pruned tree	--	Terminal nodes = 9	0.2574
(c)	Bagging	1000	--	0.1775
(d)	Random forest	1000	m = $\frac{p}{3} = \frac{19}{3}$	0.1799
(e)	Boosting	1000	depth = 1, Shrinkage, λ = 0.01	0.2212

In the previous project, we recommended a ridge regression model. However, we recommend a model with **Bagging or Random Forest** because they have the lowest test MSE's. Also notice that in the previous project, every model's test MSE was around 0.40 – 0.42 and above all models have a test MSE in the range of 0.17 – 0.25. Therefore, tree models reduce test MSE and are preferred for this dataset.

----

2 **<u>(DONE IN PYTHON – R was tedious to work with)</u>**

(a) – (j): See summary table in (k) below.

(k):

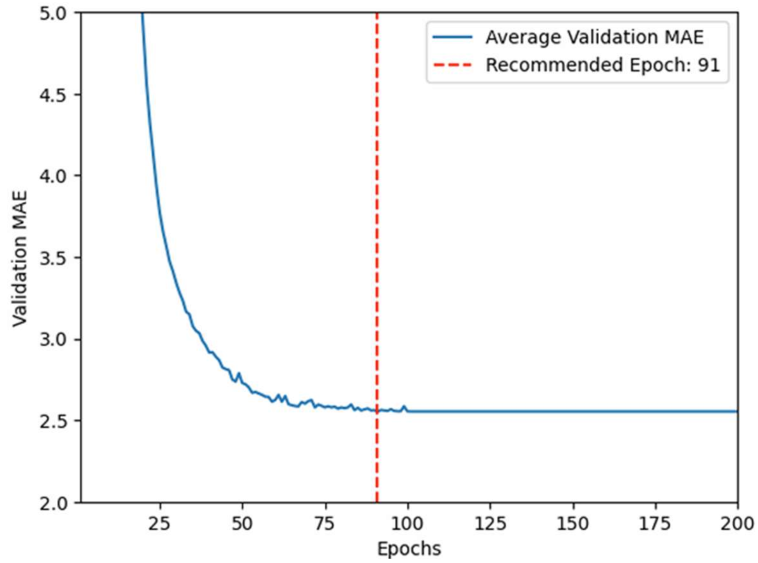
Question	Regularization/ Dropout	Number of Hidden Layers	Number of Hidden units in each layer	Number of epochs	Training error rate	Test error rate
(a)	Neither	1	512	5	0.863 %	2.03 %
(b)	Neither	1	512	10	0.153 %	1.82 %
(c)	Neither	1	256	5	1.182 %	2.23 %
(d)	Neither	1	256	10	0.240 %	1.92 %
(e)	Neither	2	512	5	0.763 %	2.11 %
(f)	Neither	2	512	10	0.178 %	1.67 %
(g)	Neither	2	256	5	0.885 %	2.51 %
(h)	Neither	2	256	10	0.750 %	2.52 %
(i)	L2 Reg. λ = 0.001	1	512	5	2.525 %	3.24 %
(j)	50 % Dropout	1	512	5	1.403 %	2.25 %

From the above table we can see that models that have more hidden layers, hidden units and epochs have the least training and test error rates (f). When we applied L2 regularization, both error rates were the highest, implying that regularization did not help; similarly for 50% dropout, the error rates are relatively higher. So, we recommend the model with **2 hidden layers with 512 hidden units in each layer and 10 epochs.**

----

3 <u>(DONE IN PYTHON – R was tedious to work with)</u>

(a):



- Since the plot indicates (after taking a closer look) a point where the validation MAE stops decreasing and starts to increase, early stopping is recommended.
- We suggest 91 epochs (red line).
- For model fit with recommended number of epochs, Validation MAE = 2.07 and corresponding test MAE = 2.9103.

(b) – (d): See summary table in (e) below.

(e):

Question	Regularization	Number of Hidden Layers	Number of Hidden units in each layer	Number of epochs	Validation MAE	Test MAE
(a)	No	2	64	91	2.070	2.910
(b)	No	1	128	91	2.446	2.622
(c)	L2	2	64	91	2.439	2.704
(d)	L2	1	128	91	2.507	2.885

From the above table, we can observe that all models perform well on validation data. Looking closely, we can attest that models with regularization perform comparatively well on validation data when compared to models with high complexity. Hence, we recommend the model with **L2 regularization with 2 hidden layers and 64 units in each layer.**

Section 2: Code

```r
#Load Required Libraries

library(ggplot2)
library(plyr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(caret)
```

```
## Loading required package: lattice
```

```r
library(ISLR)
library(pls)
```

```
##
## Attaching package: 'pls'

## The following object is masked from 'package:caret':
##
##     R2

## The following object is masked from 'package:stats':
##
##     loadings
```

```r
library(tree)
```

```
## Warning: package 'tree' was built under R version 4.4.2
```

```r
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.4.2

## randomForest 4.7-1.2

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 4.4.2

## Loaded gbm 2.2.2

## This version of gbm is no longer under development. Consider transitioning to gbm3, https://github.com/gbm-
```

```r
set.seed(1)
```

Problem 1:

```r
Hitters <- na.omit(Hitters)
players <- row.names(Hitters) #names of baseball players

#-------------------Dummy representation------------------------------------------

#Creating dummy variables for categorical variables: League, Division, and
# NewLeague, dropping the reference category to avoid multicollinearity issue
dummies_league <- model.matrix(~ League, data = Hitters)[, -1]
dummies_division <- model.matrix(~ Division, data = Hitters)[, -1]
dummies_newleague <- model.matrix(~ NewLeague, data = Hitters)[, -1]

#Combine the dummy variables with the original Hitters dataset excluding the
```

```r
# original categorical variables and use this for all analysis further.

Hitters_dummies <- cbind(Hitters[, !names(Hitters) %in%
                  c("League", "Division", "NewLeague")],dummies_league,
                  dummies_division, dummies_newleague)

# Inspect the new dataset #league and newleague N1 A0 division W1 E0
#str(Hitters_dummies)



#transformation of Salary variable to Log(Salary) as specified in the question
Hitters_dummies$salary_new <- log(Hitters_dummies$Salary)
str(Hitters_dummies)
```

```
## 'data.frame':    263 obs. of  21 variables:
##  $ AtBat            : int  315 479 496 321 594 185 298 323 401 574 ...
##  $ Hits             : int  81 130 141 87 169 37 73 81 92 159 ...
##  $ HmRun            : int  7 18 20 10 4 1 0 6 17 21 ...
##  $ Runs             : int  24 66 65 39 74 23 24 26 49 107 ...
##  $ RBI              : int  38 72 78 42 51 8 24 32 66 75 ...
##  $ Walks            : int  39 76 37 30 35 21 7 8 65 59 ...
##  $ Years            : int  14 3 11 2 11 2 3 2 13 10 ...
##  $ CAtBat           : int  3449 1624 5628 396 4408 214 509 341 5206 4631 ...
##  $ CHits            : int  835 457 1575 101 1133 42 108 86 1332 1300 ...
##  $ CHmRun           : int  69 63 225 12 19 1 0 6 253 90 ...
##  $ CRuns            : int  321 224 828 48 501 30 41 32 784 702 ...
##  $ CRBI             : int  414 266 838 46 336 9 37 34 890 504 ...
##  $ CWalks           : int  375 263 354 33 194 24 12 8 866 488 ...
##  $ PutOuts          : int  632 880 200 805 282 76 121 143 0 238 ...
##  $ Assists          : int  43 82 11 40 421 127 283 290 0 445 ...
##  $ Errors           : int  10 14 3 4 25 7 9 19 0 22 ...
##  $ Salary           : num  475 480 500 91.5 750 ...
##  $ dummies_league   : num  1 0 1 1 0 1 0 1 0 0 ...
##  $ dummies_division : num  1 1 0 0 1 0 1 1 0 0 ...
##  $ dummies_newleague: num  1 0 1 1 0 0 0 1 0 0 ...
##  $ salary_new       : num  6.16 6.17 6.21 4.52 6.62 ...
```

```r
ncol(Hitters_dummies)-1
```

```
## [1] 20
```

```r
ncol(Hitters)-1
```

```
## [1] 19
```

(a):

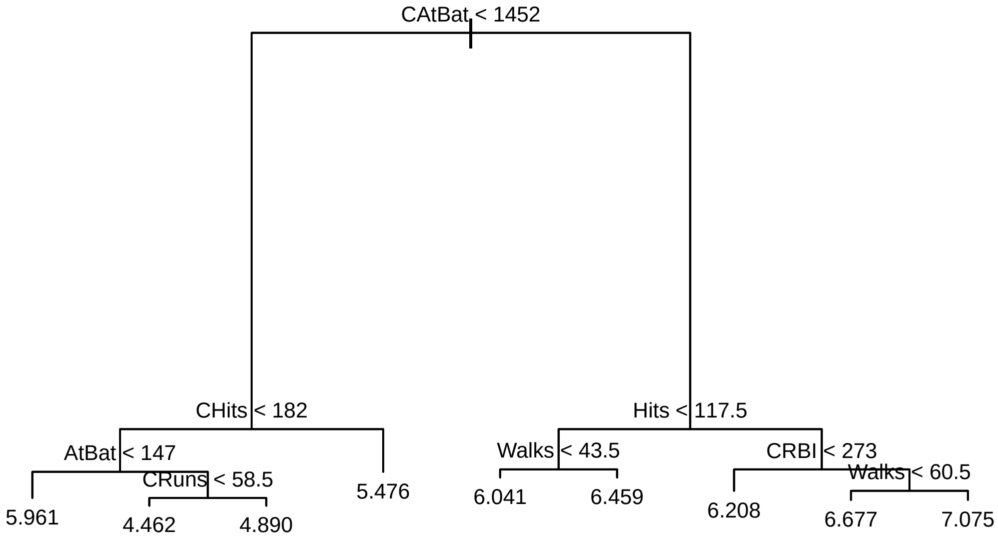
```r
#Fit the decision tree
tree_hitters <- tree(salary_new ~ AtBat + Hits + HmRun + Runs + RBI + Walks +
                  Years + CAtBat + CHits + CHmRun + CRuns + CRBI + CWalks +
                  PutOuts + Assists + Errors + dummies_division + dummies_league +
                  dummies_newleague, data = Hitters_dummies)

#Display tree and see summary to determine number of terminal nodes.
tree_hitters
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 263 207.200 5.927
##    2) CAtBat < 1452 103  36.220 5.093
##      4) CHits < 182 56  18.360 4.771
##        8) AtBat < 147 5   5.899 5.961 *
##        9) AtBat > 147 51   4.691 4.655
##         18) CRuns < 58.5 28   1.019 4.462 *
##         19) CRuns > 58.5 23   1.357 4.890 *
##      5) CHits > 182 47   5.165 5.476 *
##    3) CAtBat > 1452 160  53.080 6.464
##      6) Hits < 117.5 70  17.610 6.154
##       12) Walks < 43.5 51  12.700 6.041 *
##       13) Walks > 43.5 19   2.493 6.459 *
##      7) Hits > 117.5 90  23.490 6.706
##       14) CRBI < 273 20   4.418 6.208 *
##       15) CRBI > 273 70  12.700 6.848
##         30) Walks < 60.5 40   4.709 6.677 *
##         31) Walks > 60.5 30   5.275 7.075 *
```

```r
summary(tree_hitters)
```

```
##
## Regression tree:
## tree(formula = salary_new ~ AtBat + Hits + HmRun + Runs + RBI +
##     Walks + Years + CAtBat + CHits + CHmRun + CRuns + CRBI +
##     CWalks + PutOuts + Assists + Errors + dummies_division +
##     dummies_league + dummies_newleague, data = Hitters_dummies)
## Variables actually used in tree construction:
## [1] "CAtBat" "CHits"  "AtBat"  "CRuns"  "Hits"   "Walks"  "CRBI"
## Number of terminal nodes:  9
## Residual mean deviance:  0.1694 = 43.03 / 254
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.7080 -0.2213  0.0353  0.0000  0.2303  1.7020
```

```r
# Here, # of terminal nodes = 9. So we display the tree graphically.
plot(tree_hitters)
text(tree_hitters, pretty = 0, cex = 0.7)
```




```
# Test MSE -- Define a function to calculate test MSE
loocv_mse_tree <- function(data, formula) {
  n_obs_hitters <- nrow(data)
  testmse <- numeric(n_obs_hitters)

#Run the loop depending on the number of observations in the dataset. Each iteration leaves one observation ou
  for (i in 1:n_obs_hitters) {
    #LOOCV
    train_data_hitters <- data[-i, ]
    test_data_hitters <- data[i, , drop = FALSE]

    #fit a regression tree model and use it to predict response for the test data
    fit_model <- tree(formula, data = train_data_hitters)
    pred <- predict(fit_model, newdata = test_data_hitters)

    #MSE = square of difference between actual response of test data and prediction made above.
    testmse[i] <- (test_data_hitters$salary_new - pred)^2
  }

  #Mean of errors
  mean(testmse)
}

#Call the above defined function.
tree_mse <- loocv_mse_tree(Hitters_dummies, salary_new ~ AtBat + Hits + HmRun + Runs + RBI + Walks + Years + C
tree_mse
```

```
## [1] 0.2545162
```

(b):

```r
# Perform LOOCV to determine optimal tree size
pruned_hitters <- cv.tree(tree_hitters, K = nrow(Hitters_dummies))  # LOOCV
pruned_hitters
```

```
## $size
## [1] 9 8 7 6 5 4 3 2 1
##
## $dev
## [1]  69.13337  74.40305  73.31008  73.03341  82.88160  81.10150 107.16969
## [8] 105.40736 236.50768
##
## $k
## [1]       -Inf   2.314754   2.423858   2.713047   6.377474   7.769090  11.970263
## [8]  12.695982 117.857612
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```r
# Identify the optimal tree size based on minimum deviance
optimal_size <- pruned_hitters$size[which.min(pruned_hitters$dev)]
optimal_size
```

```
## [1] 9
```

```r
# Optimal size of pruned tree using LOOCV = 9 = number of terminal nodes.
#Therefore, pruning is not useful.

# Prune the tree to the optimal size
pruned_best_hitters <- prune.tree(tree_hitters, best = optimal_size)
pruned_best_hitters
```

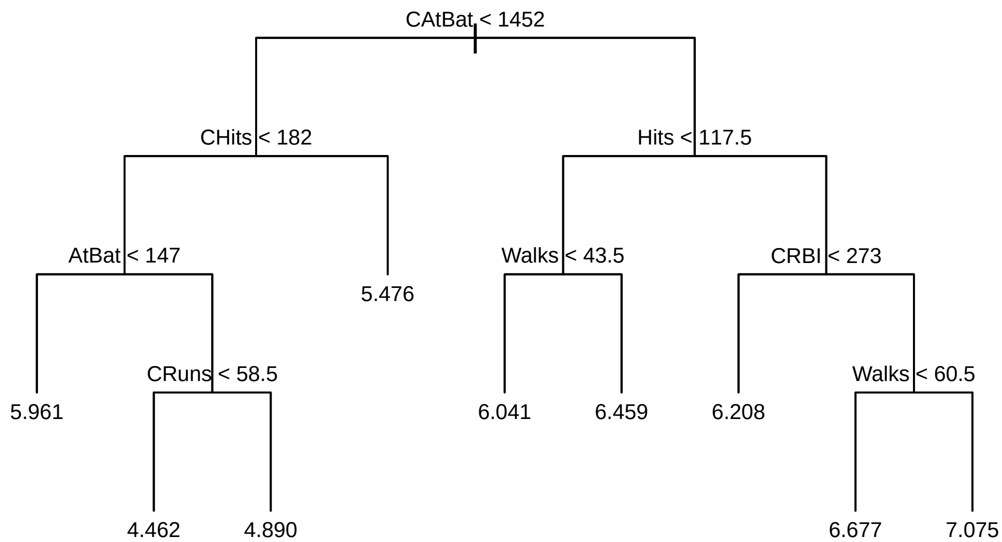
```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 263 207.200 5.927
##    2) CAtBat < 1452 103  36.220 5.093
##      4) CHits < 182 56  18.360 4.771
##        8) AtBat < 147 5   5.899 5.961 *
##        9) AtBat > 147 51   4.691 4.655
##         18) CRuns < 58.5 28   1.019 4.462 *
##         19) CRuns > 58.5 23   1.357 4.890 *
```

```
##      5) CHits > 182 47   5.165 5.476 *
##    3) CAtBat > 1452 160  53.080 6.464  
##      6) Hits < 117.5 70  17.610 6.154  
##       12) Walks < 43.5 51  12.700 6.041 *
##       13) Walks > 43.5 19   2.493 6.459 *
##      7) Hits > 117.5 90  23.490 6.706  
##       14) CRBI < 273 20   4.418 6.208 *
##       15) CRBI > 273 70  12.700 6.848  
##         30) Walks < 60.5 40   4.709 6.677 *
##         31) Walks > 60.5 30   5.275 7.075 *
```

```r
summary(pruned_best_hitters)
```

```
## 
## Regression tree:
## tree(formula = salary_new ~ AtBat + Hits + HmRun + Runs + RBI + 
##     Walks + Years + CAtBat + CHits + CHmRun + CRuns + CRBI + 
##     CWalks + PutOuts + Assists + Errors + dummies_division + 
##     dummies_league + dummies_newleague, data = Hitters_dummies)
## Variables actually used in tree construction:
## [1] "CAtBat" "CHits"  "AtBat"  "CRuns"  "Hits"   "Walks"  "CRBI"  
## Number of terminal nodes:  9 
## Residual mean deviance:  0.1694 = 43.03 / 254 
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## -1.7080 -0.2213  0.0353  0.0000  0.2303  1.7020
```

```r
# Plot the pruned tree and add labels
plot(pruned_best_hitters, type = "uniform")  # Nodes spaced uniformly
text(pruned_best_hitters, pretty = 0, cex = 0.7)
```




```r
# Function to compute LOOCV Test MSE for the pruned tree
loocv_mse_tree <- function(data, formula, optimal_size) {
  n_obs <- nrow(data)
  test_mse <- numeric(n_obs)

  for (i in 1:n_obs) {
    train_data <- data[-i, ]
    test_data <- data[i, , drop = FALSE]

    # Fit a regression tree on the training set
    fit_model <- tree(formula, data = train_data)

    # Prune the tree to the optimal size
    pruned_model <- prune.tree(fit_model, best = optimal_size)

    # Predict response for the test observation
    pred_pruned <- predict(pruned_model, newdata = test_data)

    # Compute MSE
    test_mse[i] <- (test_data$salary_new - pred_pruned)^2
  }
  mean(test_mse)
}

# Compute the LOOCV Test MSE for the pruned tree
pruned_mse <- loocv_mse_tree(Hitters_dummies, salary_new ~ AtBat + Hits +
```

```
            HmRun + Runs + RBI + Walks + Years + CAtBat + CHits + CHmRun +
            CRuns + CRBI + CWalks + PutOuts + Assists + Errors +
          dummies_division + dummies_league + dummies_newleague, optimal_size)
```

```
## Warning in prune.tree(fit_model, best = optimal_size): best is bigger than tree
## size
## Warning in prune.tree(fit_model, best = optimal_size): best is bigger than tree
## size
## Warning in prune.tree(fit_model, best = optimal_size): best is bigger than tree
## size
```

```
#Test MSE of pruned tree
pruned_mse
```

```
## [1] 0.2574206
```

(c):

```
bag_hitters <- randomForest(salary_new ~ AtBat + Hits + HmRun + Runs + RBI + Walks +
                  Years + CAtBat + CHits + CHmRun + CRuns + CRBI + CWalks +
                  PutOuts + Assists + Errors + dummies_division + dummies_league +
                  dummies_newleague, data = Hitters_dummies, mtry = 19 , ntree = 1000, importance = TRUE)
bag_hitters
```

```
##
## Call:
##  randomForest(formula = salary_new ~ AtBat + Hits + HmRun + Runs +      RBI + Walks + Years + CAtBat + CHit
##                Type of random forest: regression
##                      Number of trees: 1000
## No. of variables tried at each split: 19
##
##           Mean of squared residuals: 0.1863298
##                     % Var explained: 76.34
```

```
# Test MSE -- Define a function to calculate test MSE
loocv_mse_bagging <- function(data, formula, mtry, ntree) {
  n_obs_hitters <- nrow(data)
  testmse_bagging <- numeric(n_obs_hitters)

#Run the loop depending on the number of observations in the dataset. Each iteration leaves one observation ou
  for (i in 1:n_obs_hitters) {
    #LOOCV
    train_data_hitters <- data[-i, ]
    test_data_hitters <- data[i, , drop = FALSE]

    #fit a regression tree model and use it to predict response for the test data
    fit_model <-randomForest(formula, data = train_data_hitters)
```

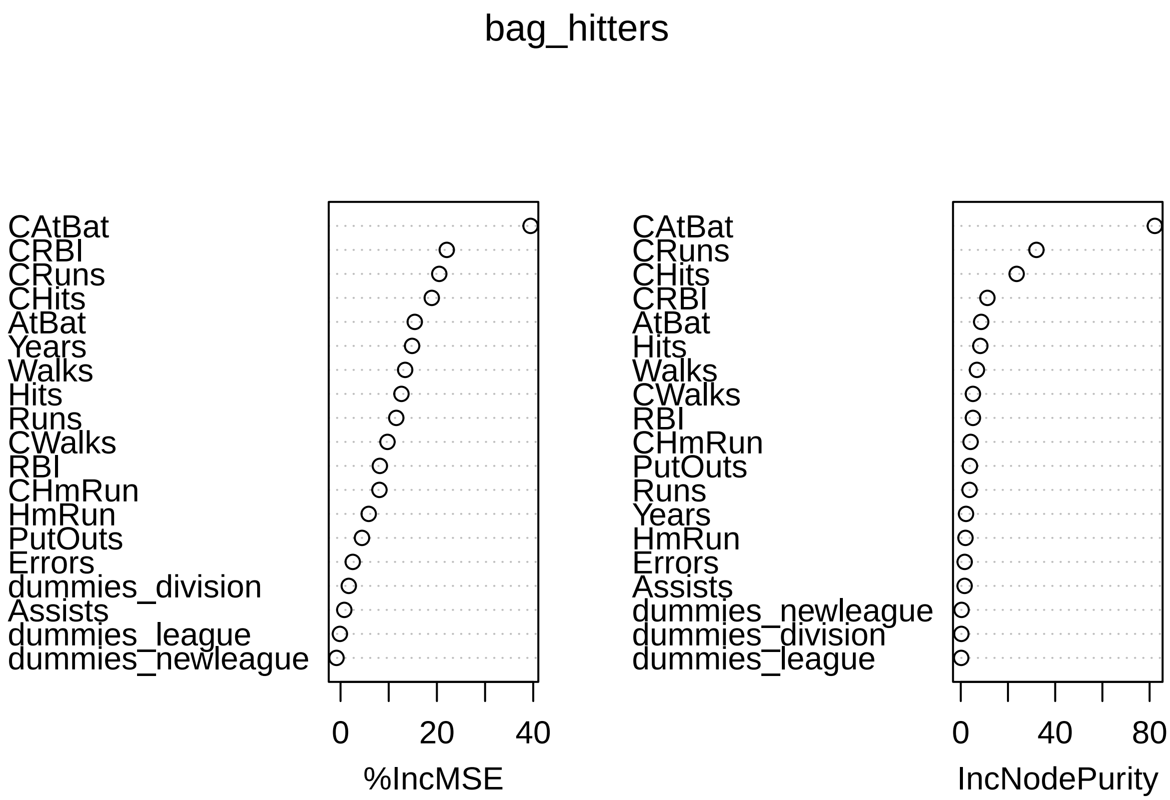
```r
    pred <- predict(fit_model, newdata = test_data_hitters)

    #MSE = square of difference between actual response of test data and prediction made above.
    testmse_bagging[i] <- (test_data_hitters$salary_new - pred)^2
  }

  #Mean of errors
  mean(testmse_bagging)
}

#Call the above defined function and calculate the test MSE.
bagging_mse <- loocv_mse_bagging(Hitters_dummies, salary_new ~ AtBat + Hits +
                                   HmRun + Runs + RBI + Walks + Years + CAtBat +
                                   CHits + CHmRun + CRuns + CRBI + CWalks +
                                   PutOuts + Assists + Errors + dummies_division +
                                   dummies_league + dummies_newleague,
                                 mtry = 19, ntree = 1000)
bagging_mse
```

```
## [1] 0.179624
```

```r
# Display the important predictors and plot them to see which ones are indeed important.
imp_pred_bagging <- bag_hitters$importance
imp_pred_bagging
```

```
##                         %IncMSE IncNodePurity
## AtBat              3.474895e-02     8.6552709
## Hits               4.428587e-02     8.2873751
## HmRun              8.040511e-03     1.9904871
## Runs               1.388398e-02     3.7296120
## RBI                1.129293e-02     5.1477443
## Walks              2.779032e-02     6.8492557
## Years              1.349102e-02     2.2209220
## CAtBat             3.388386e-01    82.2182390
## CHits              1.331028e-01    23.6745377
## CHmRun             1.121688e-02     4.1630332
## CRuns              1.733578e-01    32.0619000
## CRBI               1.684714e-01    11.2823323
## CWalks             2.025776e-02     5.1520616
## PutOuts            4.193860e-03     3.8861930
## Assists            4.869346e-04     1.6321109
## Errors             1.414031e-03     1.6558213
## dummies_division   3.630688e-04     0.2483158
## dummies_league    -2.517339e-05     0.1824245
## dummies_newleague -2.009446e-04     0.3445405
```

```r
varImpPlot(bag_hitters)
```



(d):

```r
randFor_hitters <- randomForest(salary_new ~ AtBat + Hits + HmRun + Runs + RBI + Walks +
                  Years + CAtBat + CHits + CHmRun + CRuns + CRBI + CWalks +
                  PutOuts + Assists + Errors + dummies_division + dummies_league +
                  dummies_newleague, data = Hitters_dummies, mtry = 19/3 , ntree = 1000, importance = TRUE)
randFor_hitters
```

```
##
## Call:
##  randomForest(formula = salary_new ~ AtBat + Hits + HmRun + Runs +      RBI + Walks + Years + CAtBat + CHit
##                Type of random forest: regression
##                      Number of trees: 1000
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 0.1794676
##                     % Var explained: 77.22
```

```r
# Test MSE -- Define a function to calculate test MSE
loocv_mse_randfor <- function(data, formula, mtry, ntree) {
  n_obs_hitters <- nrow(data)
  testmse_bagging <- numeric(n_obs_hitters)

#Run the loop depending on the number of observations in the dataset. Each iteration leaves one observation ou
  for (i in 1:n_obs_hitters) {
    #LOOCV
    train_data_hitters <- data[-i, ]
    test_data_hitters <- data[i, , drop = FALSE]
```

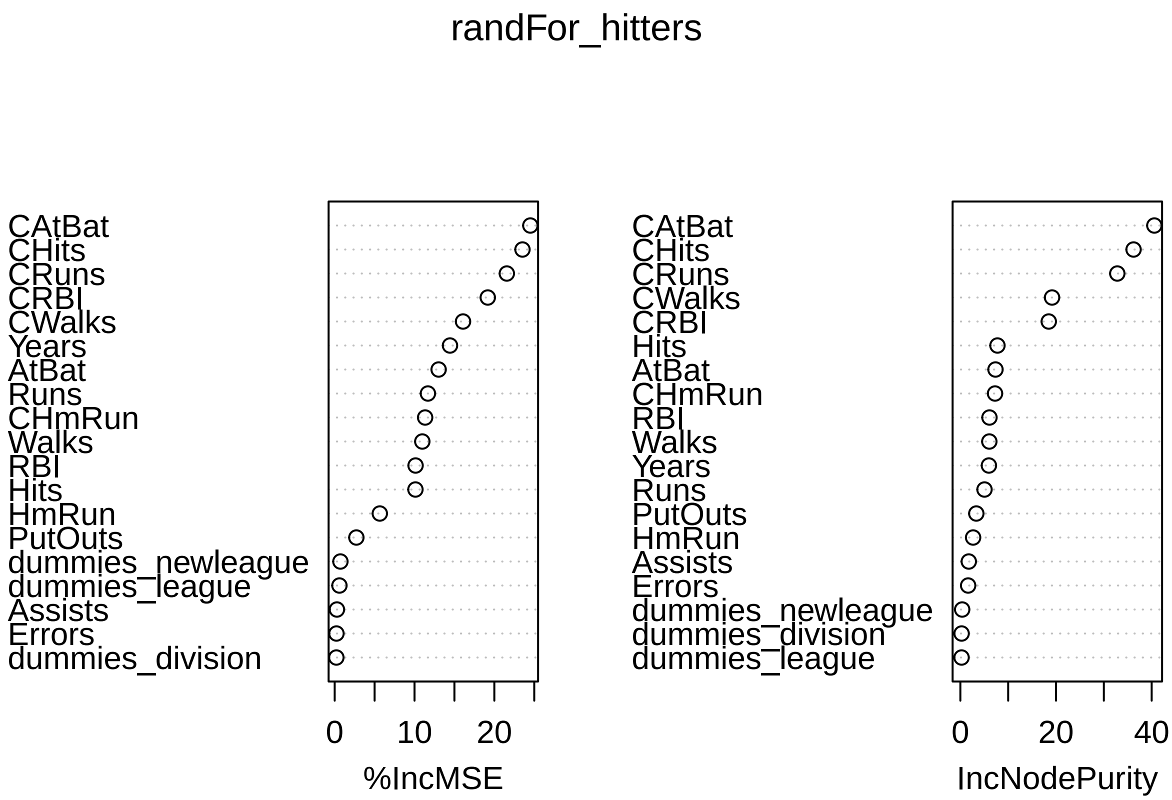
```r
    #fit a regression tree model and use it to predict response for the test data
    fit_model <-randomForest(formula, data = train_data_hitters)
    pred <- predict(fit_model, newdata = test_data_hitters)

    #MSE = square of difference between actual response of test data and prediction made above.
    testmse_bagging[i] <- (test_data_hitters$salary_new - pred)^2
  }

  #Mean of errors
  mean(testmse_bagging)
}

#Call the above defined function and calculate the test MSE.
randfor_mse <- loocv_mse_bagging(Hitters_dummies, salary_new ~ AtBat + Hits +
                                   HmRun + Runs + RBI + Walks + Years + CAtBat +
                                   CHits + CHmRun + CRuns + CRBI + CWalks +
                                   PutOuts + Assists + Errors + dummies_division +
                                   dummies_league + dummies_newleague,
                                 mtry = 19/3, ntree = 1000)
randfor_mse
```

```
## [1] 0.1803003
```

```r
# Display the important predictors and plot them to see which ones are indeed important.
imp_pred_randfor <- randFor_hitters$importance
imp_pred_randfor
```

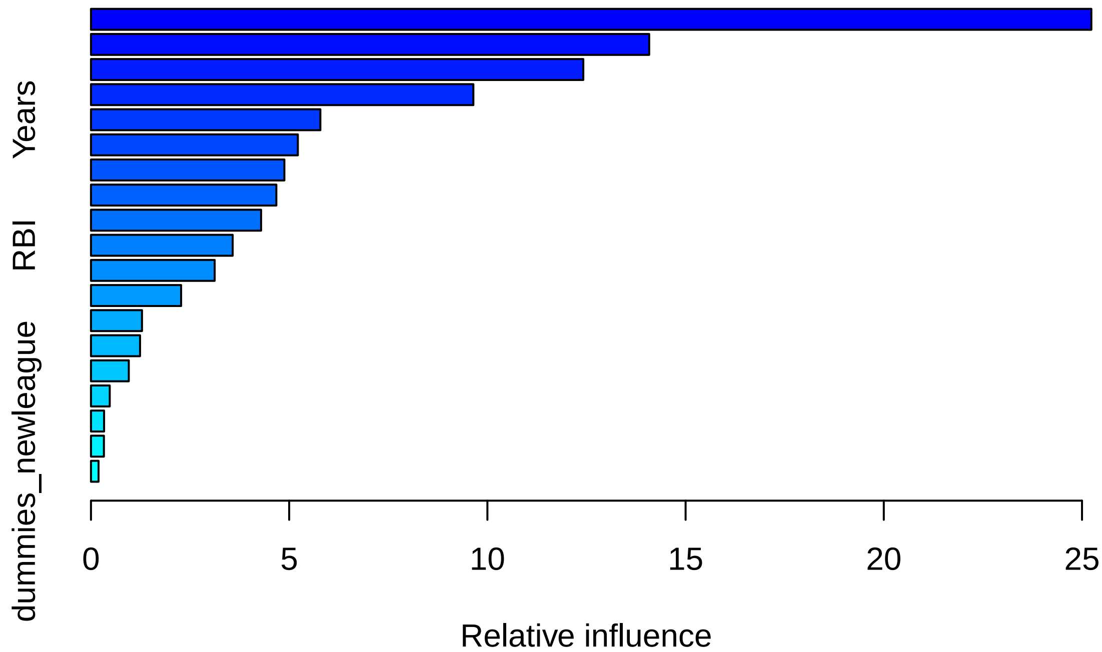
```
##                         %IncMSE IncNodePurity
## AtBat             0.0266907942     7.3472444
## Hits              0.0339833463     7.7605364
## HmRun             0.0064261457     2.6650434
## Runs              0.0192412811     5.0460871
## RBI               0.0164200874     6.0859582
## Walks             0.0186946055     6.0549881
## Years             0.0201961054     5.9659904
## CAtBat            0.1994801059    40.5591765
## CHits             0.1984872151    36.2134569
## CHmRun            0.0262860989     7.2520696
## CRuns             0.1782045506    32.8170731
## CRBI              0.1217553554    18.4843032
## CWalks            0.0638245648    19.1736615
## PutOuts           0.0023213960     3.3393015
## Assists           0.0001917311     1.7650371
## Errors            0.0001362157     1.6325434
## dummies_division  0.0000627936     0.2638157
## dummies_league    0.0001139990     0.2555396
## dummies_newleague 0.0002228722     0.3831599
```

```r
varImpPlot(randFor_hitters)
```



(e):

```r
boosting_hitters <- gbm(salary_new ~ AtBat + Hits + HmRun + Runs + RBI + Walks +
                  Years + CAtBat + CHits + CHmRun + CRuns + CRBI + CWalks +
                  PutOuts + Assists + Errors + dummies_division + dummies_league +
                  dummies_newleague, data = Hitters_dummies, distribution = "gaussian", n.trees = 1000, intera
summary(boosting_hitters)
```




```
##                                   var    rel.inf
## CAtBat                         CAtBat 25.2357621
## CRuns                           CRuns 14.0811757
## CRBI                             CRBI 12.4199630
## CHits                           CHits  9.6459641
## Years                           Years  5.7863266
## CWalks                         CWalks  5.2185461
## Hits                             Hits  4.8783876
## Walks                           Walks  4.6759233
## CHmRun                         CHmRun  4.2913092
## RBI                               RBI  3.5742745
## PutOuts                       PutOuts  3.1197659
## HmRun                           HmRun  2.2740726
## Runs                             Runs  1.2870313
## Errors                         Errors  1.2378445
## AtBat                           AtBat  0.9533915
## dummies_division   dummies_division  0.4732884
## Assists                       Assists  0.3304336
## dummies_league       dummies_league  0.3262714
## dummies_newleague dummies_newleague  0.1902685
```

```
# Test MSE -- Define a function to calculate test MSE
loocv_mse_boosting <- function(data, formula, n.trees, interaction.depth, shrinkage) {
  n_obs_hitters <- nrow(data)
  testmse_boosting <- numeric(n_obs_hitters)
```

```
#Run the loop depending on the number of observations in the dataset. Each iteration leaves one observation ou
  for (i in 1:n_obs_hitters) {
    #LOOCV
    train_data_hitters <- data[-i, ]
    test_data_hitters <- data[i, , drop = FALSE]

    #fit a regression tree model and use it to predict response for the test data
    fit_model <-gbm(formula, data = train_data_hitters, distribution = "gaussian", n.trees = n.trees, interact
    pred <- predict(fit_model, newdata = test_data_hitters, n.trees = n.trees)

    #MSE = square of difference between actual response of test data and prediction made above.
    testmse_boosting[i] <- (test_data_hitters$salary_new - pred)^2
  }

  #Mean of errors
  mean(testmse_boosting)
}

#Call the above defined function and calculate the test MSE.
boosting_mse <- loocv_mse_boosting(Hitters_dummies, salary_new ~ AtBat + Hits +
                                     HmRun + Runs + RBI + Walks + Years + CAtBat +
                                     CHits + CHmRun + CRuns + CRBI + CWalks +
                                     PutOuts + Assists + Errors + dummies_division +
                                     dummies_league + dummies_newleague,
                                   n.trees = 1000, interaction.depth = 1, shrinkage = 0.01)
boosting_mse
```

```
## [1] 0.2220895
```

Problem 2:

```
# Import required libraries
import numpy as np
import tensorflow as tf
import random
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.utils import to_categorical

# Load the MNIST dataset from keras.datasets and assign training and test data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Scale the images(in pixels to floating point) to a range of 0 to 1 by dividing
# by 255 since 0-255 is the initial range.
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# Convert categorical labels into binary matrix (one-hot encoding)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

(a):

```
# Build the neural network model with 1 hidden layer and 512 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Evaluate the model on the training data and test data
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate = 1 - test accuracy and training error rate = 1 - training accuracy.
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.8730 - loss: 0.4422 - val_accuracy: 0.9529 - val_loss: 0.1503
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9666 - loss: 0.1166 - val_accuracy: 0.9725 - val_loss: 0.0917
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.9787 - loss: 0.0731 - val_accuracy: 0.9703 - val_loss: 0.0950
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9835 - loss: 0.0539 - val_accuracy: 0.9762 - val_loss: 0.0749
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9893 - loss: 0.0374 - val_accuracy: 0.9797 - val_loss: 0.0690
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9915 - loss: 0.0290
Training Loss: 0.030, Training Accuracy: 0.991
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9778 - loss: 0.0786
Test Loss: 0.069, Test Accuracy: 0.980
Test Error Rate: 2.030
Training Error Rate: 0.863
```

(b):

```
# Build the neural network model with 1 hidden layer and 512 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 10 epochs
fit_model_result = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - accuracy: 0.8736 - loss: 0.4396 - val_accuracy: 0.9570 - val_loss: 0.1393
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 9ms/step - accuracy: 0.9665 - loss: 0.1176 - val_accuracy: 0.9742 - val_loss: 0.0839
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - accuracy: 0.9780 - loss: 0.0720 - val_accuracy: 0.9778 - val_loss: 0.0725
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9857 - loss: 0.0500 - val_accuracy: 0.9765 - val_loss: 0.0736
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9896 - loss: 0.0367 - val_accuracy: 0.9789 - val_loss: 0.0688
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 10ms/step - accuracy: 0.9920 - loss: 0.0284 - val_accuracy: 0.9808 - val_loss: 0.0590
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9943 - loss: 0.0210 - val_accuracy: 0.9811 - val_loss: 0.0637
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.9959 - loss: 0.0151 - val_accuracy: 0.9830 - val_loss: 0.0590
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9976 - loss: 0.0109 - val_accuracy: 0.9815 - val_loss: 0.0595
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.9979 - loss: 0.0091 - val_accuracy: 0.9818 - val_loss: 0.0649
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9986 - loss: 0.0061
Training Loss: 0.007, Training Accuracy: 0.998
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9775 - loss: 0.0824
Test Loss: 0.065, Test Accuracy: 0.982
Test Error Rate: 1.820
Training Error Rate: 0.153
```

(c):

```
# Build the neural network model with 1 hidden layer with 256 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")
```

```
train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.8632 - loss: 0.4916 - val_accuracy: 0.9552 - val_loss: 0.1533
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9599 - loss: 0.1413 - val_accuracy: 0.9661 - val_loss: 0.1104
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9741 - loss: 0.0913 - val_accuracy: 0.9714 - val_loss: 0.0936
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9793 - loss: 0.0690 - val_accuracy: 0.9768 - val_loss: 0.0773
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9857 - loss: 0.0501 - val_accuracy: 0.9777 - val_loss: 0.0746
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9881 - loss: 0.0414
Training Loss: 0.041, Training Accuracy: 0.988
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9739 - loss: 0.0883
Test Loss: 0.075, Test Accuracy: 0.978
Test Error Rate: 2.230
Training Error Rate: 1.182
```

(d):

```
# Build the neural network model with 1 hidden layer with 256 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 10 epochs
fit_model_result = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.8695 - loss: 0.4771 - val_accuracy: 0.9538 - val_loss: 0.1542
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.9603 - loss: 0.1384 - val_accuracy: 0.9670 - val_loss: 0.1090
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9736 - loss: 0.0895 - val_accuracy: 0.9739 - val_loss: 0.0842
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9799 - loss: 0.0668 - val_accuracy: 0.9744 - val_loss: 0.0808
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9851 - loss: 0.0489 - val_accuracy: 0.9774 - val_loss: 0.0706
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.9878 - loss: 0.0417 - val_accuracy: 0.9782 - val_loss: 0.0701
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9906 - loss: 0.0308 - val_accuracy: 0.9810 - val_loss: 0.0655
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9927 - loss: 0.0256 - val_accuracy: 0.9801 - val_loss: 0.0649
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.9946 - loss: 0.0196 - val_accuracy: 0.9768 - val_loss: 0.0725
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9963 - loss: 0.0155 - val_accuracy: 0.9808 - val_loss: 0.0641
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9974 - loss: 0.0108
Training Loss: 0.011, Training Accuracy: 0.998
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9772 - loss: 0.0760
Test Loss: 0.064, Test Accuracy: 0.981
Test Error Rate: 1.920
Training Error Rate: 0.240
```

(e):

```
# Build the neural network model with 2 hidden layers, each with 512 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.8727 - loss: 0.4150 - val_accuracy: 0.9696 - val_loss: 0.0978
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9743 - loss: 0.0841 - val_accuracy: 0.9752 - val_loss: 0.0775
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 18ms/step - accuracy: 0.9843 - loss: 0.0517 - val_accuracy: 0.9680 - val_loss: 0.1024
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 16ms/step - accuracy: 0.9882 - loss: 0.0367 - val_accuracy: 0.9801 - val_loss: 0.0681
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 17ms/step - accuracy: 0.9925 - loss: 0.0241 - val_accuracy: 0.9789 - val_loss: 0.0771
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9912 - loss: 0.0255
Training Loss: 0.023, Training Accuracy: 0.992
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9762 - loss: 0.0862
Test Loss: 0.077, Test Accuracy: 0.979
Test Error Rate: 2.110
Training Error Rate: 0.763
```

(f):

```
# Build the neural network model with 2 hidden layers, each with 512 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 10 epochs
fit_model_result = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.8690 - loss: 0.4222 - val_accuracy: 0.9705 - val_loss: 0.0973
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.9733 - loss: 0.0840 - val_accuracy: 0.9763 - val_loss: 0.0772
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 19ms/step - accuracy: 0.9829 - loss: 0.0523 - val_accuracy: 0.9759 - val_loss: 0.0820
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.9885 - loss: 0.0366 - val_accuracy: 0.9809 - val_loss: 0.0677
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.9918 - loss: 0.0253 - val_accuracy: 0.9817 - val_loss: 0.0755
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 20ms/step - accuracy: 0.9939 - loss: 0.0182 - val_accuracy: 0.9830 - val_loss: 0.0716
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9957 - loss: 0.0133 - val_accuracy: 0.9829 - val_loss: 0.0815
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9971 - loss: 0.0095 - val_accuracy: 0.9820 - val_loss: 0.0771
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 15ms/step - accuracy: 0.9972 - loss: 0.0090 - val_accuracy: 0.9815 - val_loss: 0.0796
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.9975 - loss: 0.0071 - val_accuracy: 0.9833 - val_loss: 0.0789
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9985 - loss: 0.0042
Training Loss: 0.005, Training Accuracy: 0.998
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9797 - loss: 0.0965
Test Loss: 0.079, Test Accuracy: 0.983
Test Error Rate: 1.670
Training Error Rate: 0.178
```

(g):

```
# Build the neural network model with 2 hidden layers, each with 256 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.8643 - loss: 0.4557 - val_accuracy: 0.9546 - val_loss: 0.1484
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9671 - loss: 0.1060 - val_accuracy: 0.9760 - val_loss: 0.0772
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9806 - loss: 0.0654 - val_accuracy: 0.9741 - val_loss: 0.0852
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.9854 - loss: 0.0477 - val_accuracy: 0.9799 - val_loss: 0.0688
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 8ms/step - accuracy: 0.9893 - loss: 0.0331 - val_accuracy: 0.9749 - val_loss: 0.0863
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9915 - loss: 0.0277
Training Loss: 0.028, Training Accuracy: 0.991
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9705 - loss: 0.1040
Test Loss: 0.086, Test Accuracy: 0.975
Test Error Rate: 2.510
Training Error Rate: 0.885
```

(h):

```
# Build the neural network model with 2 hidden layers, each with 512 units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 10 epochs
fit_model_result = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.8652 - loss: 0.4490 - val_accuracy: 0.9579 - val_loss: 0.1360
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9671 - loss: 0.1074 - val_accuracy: 0.9680 - val_loss: 0.1057
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9788 - loss: 0.0682 - val_accuracy: 0.9778 - val_loss: 0.0757
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.9852 - loss: 0.0477 - val_accuracy: 0.9807 - val_loss: 0.0711
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9897 - loss: 0.0341 - val_accuracy: 0.9776 - val_loss: 0.0838
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9922 - loss: 0.0245 - val_accuracy: 0.9802 - val_loss: 0.0714
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9938 - loss: 0.0190 - val_accuracy: 0.9817 - val_loss: 0.0649
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9957 - loss: 0.0141 - val_accuracy: 0.9810 - val_loss: 0.0673
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9958 - loss: 0.0126 - val_accuracy: 0.9814 - val_loss: 0.0719
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9972 - loss: 0.0089 - val_accuracy: 0.9748 - val_loss: 0.1118
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9913 - loss: 0.0282
Training Loss: 0.024, Training Accuracy: 0.993
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9739 - loss: 0.1127
Test Loss: 0.112, Test Accuracy: 0.975
Test Error Rate: 2.520
Training Error Rate: 0.750
```

(i):

```
from keras.regularizers import l2

# Build the neural network model with L2 weight regularization (λ = 0.001) and
# 1 hidden layer with 512 hidden units
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")
```

```
# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.8661 - loss: 0.8315 - val_accuracy: 0.9556 - val_loss: 0.2748
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.9565 - loss: 0.2573 - val_accuracy: 0.9612 - val_loss: 0.2110
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9668 - loss: 0.1940 - val_accuracy: 0.9636 - val_loss: 0.1968
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.9686 - loss: 0.1750 - val_accuracy: 0.9703 - val_loss: 0.1667
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9720 - loss: 0.1643 - val_accuracy: 0.9676 - val_loss: 0.1644
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9747 - loss: 0.1477
Training Loss: 0.148, Training Accuracy: 0.975
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9616 - loss: 0.1857
Test Loss: 0.164, Test Accuracy: 0.968
Test Error Rate: 3.240
Training Error Rate: 2.525
```

(j):

```
# Build the neural network model with 1 hidden layer with 512 units and 50% dropout
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model for 5 epochs
fit_model_result = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Model evaluation
train_loss, train_acc = model.evaluate(x_train, y_train)
print(f"Training Loss: {train_loss:.3f}, Training Accuracy: {train_acc:.3f}")

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.3f}, Test Accuracy: {test_acc:.3f}")

# Calculate test error rate and training error rate (same as in (a))
test_error_rate = (1 - test_acc)*100
print(f"Test Error Rate: {test_error_rate:.3f}")

train_error_rate = (1 - train_acc)*100
print(f"Training Error Rate: {train_error_rate:.3f}")
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.8484 - loss: 0.5073 - val_accuracy: 0.9554 - val_loss: 0.1531
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 15ms/step - accuracy: 0.9514 - loss: 0.1670 - val_accuracy: 0.9674 - val_loss: 0.1078
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9623 - loss: 0.1223 - val_accuracy: 0.9717 - val_loss: 0.0902
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9702 - loss: 0.0992 - val_accuracy: 0.9754 - val_loss: 0.0781
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9762 - loss: 0.0803 - val_accuracy: 0.9775 - val_loss: 0.0720
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9861 - loss: 0.0462
Training Loss: 0.047, Training Accuracy: 0.986
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9735 - loss: 0.0852
Test Loss: 0.072, Test Accuracy: 0.978
Test Error Rate: 2.250
Training Error Rate: 1.403
```

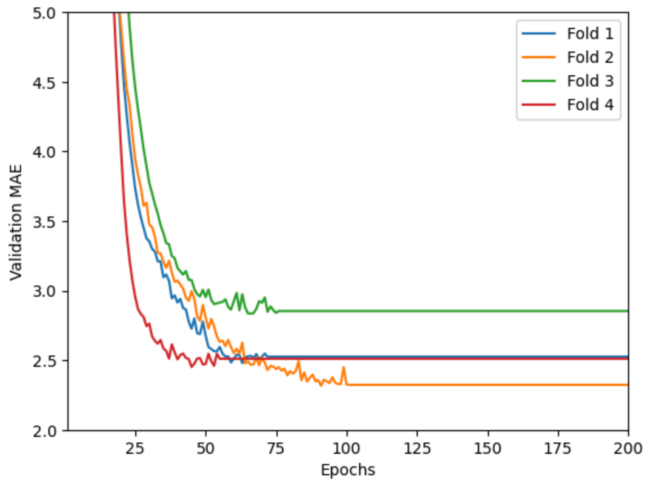
PROBLEM 3:

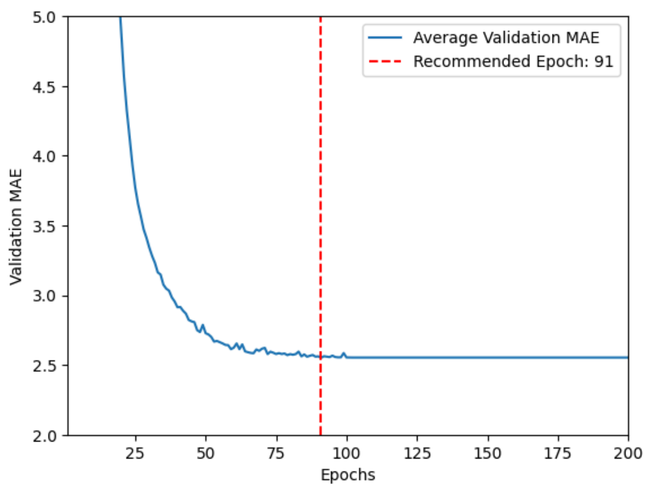
(a):

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import boston_housing
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.losses import MeanAbsoluteError
from keras.callbacks import EarlyStopping
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from keras.regularizers import l2

# Load the Boston Housing Price dataset
(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()

# Standardize the features using the mean and standard deviation from the training data
scaler = StandardScaler()
train_data = scaler.fit_transform(train_data)
test_data = scaler.transform(test_data)

# Define the model architecture with ReLU activation
def build_model():
    model = Sequential()
    # Add the first hidden layer with 64 units, ReLU activation, and L2 regularization
    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001), input_shape=(train_data.shape[1],)))
    # Add the second hidden layer with the same units, activation, and regularization
    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001)))
    # Add the output layer with a single unit
    model.add(Dense(1))
    # Compile the model using the RMSprop optimizer and use MAE as the loss function
    model.compile(optimizer=RMSprop(), loss='mse', metrics=['mae'])
    return model

# Perform 4-fold cross-validation as mentioned in the question
kfold = KFold(n_splits=4, shuffle=True, random_state=42)
num_epochs = 200
all_mae_histories = []

# Loop over the splits created by 4-fold CV to generate training and validation sets
for train_index, val_index in kfold.split(train_data):
    # Splitting the data (Just as in Python handout)
    partial_train_data, val_data = train_data[train_index], train_data[val_index]
    partial_train_targets, val_targets = train_targets[train_index], train_targets[val_index]

    # Build the model
    model = build_model()
    # Early stopping callback to determine recommended number of epochs
    early_stopping = EarlyStopping(monitor='val_mae', patience=10, restore_best_weights=True)
    # Train the model with above split data, 200 epochs and mini batch size = 64.
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=64, verbose=0,
                        callbacks=[early_stopping])
    # Get validation MAE from training history
    mae_history = history.history['val_mae']
    # Ensure mae_history has 200 epochs by padding with the last value
    mae_history += [mae_history[-1]] * (num_epochs - len(mae_history))
    all_mae_histories.append(mae_history)

# Calculate average MAE history for each epoch
average_mae_history = [np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]

# Plot all validation MAE's for 200 epochs with a smaller y-axis range
for i, mae_history in enumerate(all_mae_histories):
    plt.plot(range(1, len(mae_history) + 1), mae_history, label=f'Fold {i+1}')
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.xlim([1, num_epochs])
plt.ylim([2, 5])  # Set y-axis range to be smaller
plt.legend()
plt.show()

# Plot average validation MAE against epoch and add a line to determine recommended number of epochs with a smaller y-axis range
plt.plot(range(1, num_epochs + 1), average_mae_history, label='Average Validation MAE')
recommended_epochs = np.argmin(average_mae_history) + 1
plt.axvline(x=recommended_epochs, color='r', linestyle='--', label=f'Recommended Epoch: {recommended_epochs}')
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.xlim([1, num_epochs])
plt.ylim([2, 5])  # Set y-axis range to be smaller
plt.legend()
plt.show()
```

```
# Based on the plot, recommend early stopping and suggest the number of epochs
print(f"Recommended number of epochs: {recommended_epochs}")

# Fit a model with the suggested number of epochs and report its validation MAE
model = build_model()
history = model.fit(train_data, train_targets,
                    epochs=recommended_epochs, batch_size=64, verbose=0)

# Evaluate the model on the validation data and report its validation MAE
val_mse, val_mae = model.evaluate(val_data, val_targets)
print(f"Validation MAE: {val_mae: .4f}")

# Evaluate the model on the test data and report its TEST MAE
test_mse, test_mae = model.evaluate(test_data, test_targets)
print(f"Test MAE: {test_mae: .4f}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housing.npz
57026/57026 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When usi
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```






```
Recommended number of epochs: 91
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 6.5035 - mae: 1.9598
Validation MAE:  2.0700
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.7137 - mae: 2.7321
Test MAE:  2.9103
```

(b):

```
# Define the model architecture with 1 hidden layer with 128 units and 91 epochs and specified batch size
def build_model():
    model = Sequential()
    model.add(Dense(128, activation='relu', input_shape=(train_data.shape[1],)))
    model.add(Dense(1))
    model.compile(optimizer=RMSprop(), loss=MeanAbsoluteError(), metrics=['mae'])
    return model

# Build and fit the model
model = build_model()
history = model.fit(train_data, train_targets, epochs=91, batch_size=16, verbose=0, validation_split=0.2)

# Report the validation MAE
val_mae = history.history['val_mae'][-1]
```

```
print(f"Validation MAE: {val_mae}")

# Evaluate the model on the test data and report its TEST MAE
test_mse, test_mae = model.evaluate(test_data, test_targets)
print(f"Test MAE: {test_mae: .4f}")
```

```
Validation MAE: 2.446763038635254
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 2.3795 - mae: 2.3795
Test MAE:  2.6229
```

(c):

```
# Define the model architecture with L2 regularization with 2 hidden layers and 64 units in each layer
def build_model():
    model = Sequential()
    # First hidden layer with L2 regularization
    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001), input_shape=(train_data.shape[1],)))
    # Second hidden layer with L2 regularization
    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001)))
    # Output layer
    model.add(Dense(1))
    # Compile the model
    model.compile(optimizer=RMSprop(), loss='mse', metrics=['mae'])
    return model

# Build and fit the model
model = build_model()
history = model.fit(train_data, train_targets, epochs=91, batch_size=16, verbose=0, validation_split=0.2)

# Report the validation MAE
val_mae = history.history['val_mae'][-1]
print(f"Validation MAE: {val_mae:.4f}")

# Evaluate the model on the test data and report its test MAE
test_mse, test_mae = model.evaluate(test_data, test_targets, verbose=0)
print(f"Test MAE: {test_mae:.4f}")
```

```
Validation MAE: 2.4393
Test MAE: 2.7045
```

(d):

```
# Define the model architecture with 1 hidden layer (128 units) and L2 regularization
def build_model():
    model = Sequential()
    # Single hidden layer with 128 units and L2 regularization
    model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001), input_shape=(train_data.shape[1],)))
    # Output layer
    model.add(Dense(1))
    # Compile the model
    model.compile(optimizer=RMSprop(), loss='mse', metrics=['mae'])
    return model

# Build and fit the model
model = build_model()
history = model.fit(train_data, train_targets, epochs=91, batch_size=16, verbose=0, validation_split=0.2)

# Report the validation MAE
val_mae = history.history['val_mae'][-1]
print(f"Validation MAE: {val_mae:.4f}")

# Evaluate the model on the test data and report its test MAE
test_mse, test_mae = model.evaluate(test_data, test_targets, verbose=0)
print(f"Test MAE: {test_mae:.4f}")
```

```
Validation MAE: 2.5072
Test MAE: 2.8855
```