Managing Big Data: 2nd Assignment

December 1, 2022

Dimakopoulos Andreas (1067668)
Kountakis Marinos (1069158)
Mantzari Eleni (1067704)
Moustakas Andreas (1067656)
Fragkoulia Maria(1069868)

Supervisor: Tzagkarakis Emmanouil

School of Economics and Business Administration
Department of Economics
Master of Science:
"Applied Economics and Data Analysis"

Introduction

The purpose of the report is to get familiar with the methods of estimating coefficients of linear regression models both with the Least Squares (OLS) and with the Gradient Descent method and its various versions in the R and Python environments. First, we -wrote programs in R and Python that-estimated the coefficients of a linear least squares (OLS) and Batch Gradient Descent regression model using the Communities and Crime Data Set. Then we set the appropriate values for the learning parameter a and the number of iterations in the existing codes, in order to estimate the coefficients of the linear regression model using the Stochastic Gradient Descent method.

Moreover, we wrote a program in R that estimates by both OLS and Batch Gradient Descent the coefficients of a multiple linear regression model using the training dataset "HouseholdData". In the end, we wrote programs in R and Python who used 10-Fold Cross Validation in which the coefficients of a linear regression model were estimated by the method of Least Squares (OLS). Also, were estimated the Root Mean Squared Error (RMSE) of prediction. The set of data used were fires from areas of Portugal and we used 10-Fold Cross Validation twice, once with the entire data set and once with the "small fires" , i.e. only those observations where the value of the dependent variable (variable area) is less than 3.2 hectares (ie area <3.2).

Main assignment's tasks

Task 1

Task One is answered by the team member Andreas Dimakopoulos.

Task 2

R Code— Subtask i

```
#install.packages("readxl")
library ("readxl")

setwd("C:/Master/BD/Projects/Project2")

#Inserting the dataset
dta <- read.table("communities.data", fileEncoding="UTF-8",
sep=",", header=FALSE)

#Renaming the columns of the dataset
d1 <- read_excel("names.xlsx")
colnames(dta) <- d1$names

#Changing "?" missing values to be shown as NA
idx <- dta == "?"
is.na(dta) <- idx

#Droping NA values
data <- na.omit(dta)

#Exporting the dataset to use it as a csv file in python
 as well
#write.csv(data,"C:/Master/BD/Projects/Project2/data.csv",
row.names = FALSE)


str(data)
summary(data)

#OLS
model <- lm(ViolentCrimesPerPop ~ medIncome + whitePerCap +
blackPerCap + HispPerCap + NumUnderPov + PctUnemployed
 + HousVacant + MedRent + NumStreet, data= data)
```

```
summary (model)
print (model)
```

Python Code— Subtask i

```
import numpy as np
from sklearn.linear_model import LinearRegression

import pandas as pd
import matplotlib.pyplot as plt

#Inserting the dataset
dta = pd.read_csv("communities.data", header=None, sep=",")
print (dta.dtypes)
print (dta)

d1 = pd.read_excel("names.xlsx")
print (d1)

#Renaming the columns of the dataset
dta.columns = d1['names']
print (dta)

#Changing "?" missing values to be shown as NA
dta = dta.replace('?', np.nan)
print (dta)

#Droping NA values
data = dta.dropna()
print (data.dtypes)
print (data)

#OLS
independentVariables = data.loc[:, ['medIncome', '
                              whitePerCap', 'blackPerCap', '
                              HispPerCap', 'NumUnderPov', '
                              PctUnemployed', 'HousVacant',
                              'MedRent', 'NumStreet']]
dependentVariable = data.loc[:, 'ViolentCrimesPerPop']

lm = LinearRegression(normalize=False, fit_intercept=True)
model = lm.fit(independentVariables, dependentVariable)

print("Coefficients:", model.coef_)
# print("Note: Coefficients should be interpreted, based on
                               the order of variables in the
                               data.frame of independent
                              variables. This means that:")
```

```
# print("\t-Coefficient b1 (medIncome):", model.coef_[0])
# print("\t-Coefficient b2 (whitePerCap):", model.coef_[1])

print("\t-Intercept:", model.intercept_)

# print("You may also display the R-squared (the proportion
                              of variance explained):")

#Calculate R-squared
Rsquared = lm.score(independentVariables, dependentVariable
                             )

print("R-squared:", Rsquared)

from regressors import stats

print("\n====== Summary statistics ======\n")
stats.summary(model, independentVariables,
                             dependentVariable)
```

Python Code— Subtask ii

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

#
# Multiply two matrices i.e. mat1 * mat2
#
def matmultiply(mat1,mat2):

return( np.matmul(mat1, mat2) )

#
# Calculate current value of cost function J(\theta).
# indV: matrix of independent variables, first column must
                             be all 1s
# depV: matrix (dimensions nx1)of dependent variable i.e.
#
def calculateCost(indV, depV, thetas):
return( np.sum( ((matmultiply(indV, thetas) - depV)**2) / (
                             2*indV.shape[0]) ) )



#
```

```
# Batch gradient descent
#
# indV:matrix of independent variables, first column must
                             be all 1s
# depV: matrix (dimensions nx1)of dependent variable i.e.
# alpha: value of learning hyperparameter. Default (i.e. if
                              no argument provided)  0.01
# numIters: number of iterations. Default (i.e. if no
                             argument provided) 100
#
def batchGradientDescent(indV, depV, thetas, alpha = 0.01,
                             numIters = 200, verbose =
                             False):

calcThetas = thetas

# we store here the calculated values of J(\theta)
costHistory = pd.DataFrame( columns=["iter", "cost"])
m = len(depV)
#print (costHistory)

for i in range(0, numIters):
prediction=matmultiply(indV,calcThetas)
calcThetas=calcThetas-(1/m)*alpha*(matmultiply(indV.T,(
                             prediction-depV)))
print(">>>> Iteration", i, ")")
print("       Calculate thetas...", calcThetas)
c = calculateCost(indV, depV, calcThetas)
print("       Calculate cost fuction for new thetas...", c)
costHistory = costHistory.append({"iter": i, "cost": c},
                             ignore_index=True )


# Done. Return values
return calcThetas, costHistory

# Read the data
communities = pd.read_csv("communities.data", header=None,
                             sep=",", engine='python')

#That's our dependent variable
dependentVar = communities.iloc[:, 127]

# These are all our independent ones: 17,26,27,31,32,37,76,
                             90,95
communities = communities.iloc[:, [17,26,27,31,32,37,76,90,
                             95] ]

# Check to see if missing values are present.
```

```
communities = communities.replace(’?’, np.nan)
communities = communities.dropna()

# Add new column at the beginning representing the constant
                              term b0
communities.insert(0, ’b0’, 1)

# Add to a new variable to make the role of the data
                             clearer
independentVars = communities

# Initialize thetas with some random values.
# We’ll need (independentVars.shape[1]) theta values, one
                             for each independent variable.
iniThetas = []
for i in range(0, independentVars.shape[1]):
iniThetas.append( np.random.rand() )

initialThetas = np.array(iniThetas)

# Run BATCH gradient descent and return 2 values: I) the
                             vector of the estimated
                             coefficients (
                             estimatedCoefficients) and II)
                              the values of the
# cost function (costHistory)
estimatedCoefficients, costHistory = batchGradientDescent(
                             independentVars.to_numpy(),
                             dependentVar.to_numpy(),
                             initialThetas, 0.1)

# Display now the cost function to see if alpha and number
                             of iterations were appropriate
                             .
costHistory.plot.scatter(x="iter", y="cost", color=’red’)
plt.show()
```

Results of Task 2

For the first subtask of Task 2 we performed an OLS regression with R and Python. Before the estimation we omitted the missing values of the dataset, thus we dropped every observation with a "?" value. For our model we estimated the following relation:

$$ViolentCrimesPerPop = b_1 * medIncome + b_2 * whitePerCap + b_3 * blackPerCap + b_4 * HispPerCap + b_5 * NumUnderPov + b_6 * PctUnemployed + b_7 * HousVacant + b_8 * MedRent + b_9 * NumStreet + b_0$$

After the Ols regression we got the estimated coefficients:

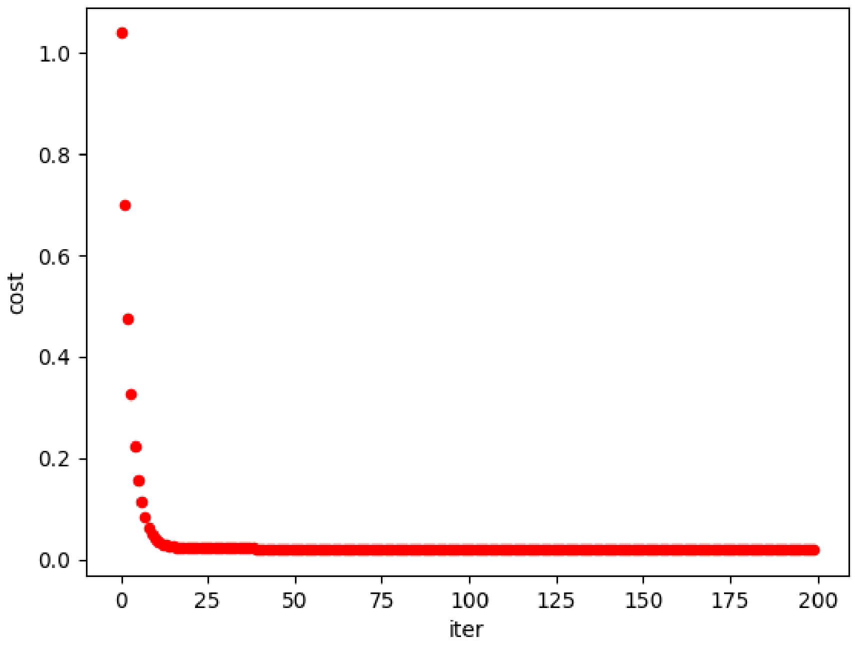
$$ViolentCrimesPerPop = -1.17 * medIncome + 0.62 * whitePerCap + 0.27 * blackPerCap - 0.01 * HispPerCap + 0.05 * NumUnderPov + 0.73 * PctUnemployed + 0.04 * HousVacant + 0.3 * MedRent + 0.21 * NumStreet - 0.04$$

We also performed a Batch Gradient Descent on the data and got the following model:

$$ViolentCrimesPerPop = -0.22 * medIncome + 0.008 * whitePerCap + 0.41 * blackPerCap + 0.18 * HispPerCap + 0.39 * NumUnderPov + 0.59 * PctUnemployed + 0.05 * HousVacant - 0.05 * MedRent + 0.49 * NumStreet - 0.11$$

It is evident that BGD does not produce the same results as OLS, even though they are relatively close. The observed difference occurs because BGD method approaches the coefficients with an iterative process and as a result the number of repetitions have an impact on the final estimates and the initialization values of the coefficients. With 200 iterations and a learning rate equal to 0.1 we get the graph below.

Figure 1: Cost plot in python



With the above calculated thetas, the cost function is minimized to 0.0196.

Task 3

i)Describe in detail what the program does.

Creates a data frame containing seven numerical variables, each of which receives four random values between 1 and 10. It builds an ordinary least squares regression model with Y as the dependent variable and

$$X_1, X_2, X_3, X_4, X_5 and X_6$$

as the independent variables. Finally, it calculates and displays the variable and constant term slope values.

ii)Describe the values of the calculated estimated coefficients. Which particular fact, in particular, do you believe some of the coefficient values that were produced are attributable to?

The estimated coefficients that arise are random in their values. However, there are some elements that go into their evaluation; for instance, the letter i dictates the lines that are generated. The constant term and the variables

$$X_1, X_2 and X_3$$

are hence the variables with values (4 lines). Finally, the scale of the predicted coefficients is determined by the values of min and max.

Task 4

i) Which independent variables and which dependent variables are mentioned in the article?

In the given paper (THE IMPACT OF JURY RACE IN CRIMINAL TRIALS), the dependent variables are consisted of “any guilty convictions” and “proportion guilty convictions”. As far as the independent variables are concerned, they are formed by “black defendant” , “any black in pool”, “defendant black*any black in pool”, “any black on seated jury”, “defendant black*any black on seated jury”.

ii) What is the form of the regression model(s) estimated in the study?

Given that their coefficients have a linear form, the model corresponds to linear regression.

iii) What method was used to estimate the coefficients of the models regression?

The study’s model was estimated with ordinary least squares regression method and was corrected for heteroscedasticity.

iv) Is the aim of the study to explain or predict the dependent variable?

The fundamental objective of this study is to explain the indirect process by which the racial makeup of the jury pool can influence the trial’s result. The aforementioned results of trials were explained by four models. “Any guilty convictions" and "proportion guilty convictions" were defined from the independent variables stated above.

Task 5

R code

```
library(ggplot2)
setwd("//Users//dhmako/Documents//Assignments␣in
Managing␣Big␣Data//Second␣Assignment")

graphics.off() ; rm(list = ls(all = TRUE)) ; cat("\014");

########## Linear Regression with OLS ##########

data <-read.csv("HouseholdData.csv", sep=",", header=T)

ols <- lm(FoodExpenditure ~ Income+FamilySize, data=data)

print(ols$coefficients)

########## Linear Regression with Gradient Descent ##########

calculateCost<-function(X, y, theta){
# Number of Observations
m <- length(y)
return( sum((X%*%theta- y)^2) / (2*m) )
} # calculateCost

# gradientDescent

gradientDescent<-function(X, y, theta, alpha=0.01, numIters=90)
{
m <- length(y)
costHistory <- rep(0, numIters)

for(i in 1:numIters){
theta <- theta - alpha*(1/m)*(t(X)%*%(X%*%theta - y))
costHistory[i]  <- calculateCost(X, y, theta)
}

gdResults<-list("coefficients"=theta, "costs"=costHistory)
return(gdResults)
}

y<- data[, "FoodExpenditure"]

x<- cbind( rep(1, 35), data[, "Income"], data[, "FamilySize"] )

initialThetas<-rep(runif(1), 3 )

gdOutput<-gradientDescent(x, y, initialThetas, 0.00000000003446, 1000)

# Show Coefficients
print(gdOutput$coefficients)

plot(gdOutput$costs, xlab="Interations", ylab="J(\theta)")
```

Results of Task 5— i) What degree of difference exists between the estimated coefficients of the ordinary least squares approach and those of the gradient descent?

First, we calculated the ideal coefficients using the ordinary least squares method. Subsequently, we used batch gradient descent method to observe the differences in the estimated coefficients between the two approaches. Regarding the results of the latter method, we note that the constant term and second coefficient values greatly differ from those obtained using the ordinary least squares method.

Table 1: Estimated coefficients

Method	theta0	theta1	theta2
OLS	1,182.10	0.12	325.71
Batch Gradient Descent	0.60	0.16	0.61

ii) What, in your opinion, accounts for the observed variance in the values of some coefficients calculated by the Gradient Descent method, if any, of the estimated coefficients?

In the outcomes of batch gradient descent, we observe that the constant term and coefficient of the second independent variable tend to receive values from the initial random "Thetas" (0-1). This is caused due to the size difference of the coefficients we are trying to estimate. The Batch Gradient Descent method uses an iterative procedure of the whole dataset and thus, because of the scale difference of the coefficients, some are approaching faster than others. In a dataset with this kind of issue a solution would be to normalize the numbers in the preprocessing of the data with the min-max method [0,1] or the z-score method [-3,3].

Task 6

R code

```
library(gdata)
library(readxl)
graphics.off() ; rm(list = ls(all = TRUE)) ; cat("\014");
data <- read.table("communities.data", fileEncoding="UTF-8", sep=","
, header=FALSE)
d1 <- read_excel("varnames.xlsx")
names <- d1$names
colnames(data) <- names

idx <- data == "?"
is.na(data) <- idx

df<-na.omit(data)

y<- df[, "ViolentCrimesPerPop"]

X<- cbind( rep(1, 123), df[, "medIncome"], df[,"whitePerCap"],
df[,"blackPerCap"], df[,"HispPerCap"], df[,"NumUnderPov"],
df[,"PctUnemployed"], df[,"HousVacant"], df[,"MedRent"],
df[,"NumStreet"] )

X <- as.matrix(X)
y <- as.matrix(y)

n_obs <- 123
learning_rate <- 0.13
batch_size <- 32
n_iter <- 10000
beta_hat_sgd <- matrix(0, nrow=n_iter, ncol=ncol(X))
beta_hat_gd <- matrix(0, nrow=n_iter, ncol=ncol(X))
beta_hat_norm_eq <- solve(t(X) %*% X) %*% t(X) %*% y

for(iter in seq_len(n_iter - 1)) {
## Keep it simple: sample a random subset of batch_size rows
 on every iteration row_idx <- sample(seq_len(n_obs), size=batch_size)
residuals <- y[row_idx] - X[row_idx, ] %*% beta_hat_sgd[iter,]
gradient <- -(2 / batch_size) * t(X[row_idx, ]) %*% residuals
beta_hat_sgd[iter + 1, ] <- beta_hat_sgd[iter, ] -learning_rate
* gradient
}

beta_hat_sgd[n_iter, ]

plot(beta_hat_sgd[, c(2, 3)], type="l", col="red",
xlab="beta_1", ylab="beta_2")
lines(beta_hat_sgd[, c(2, 3)], type="l", col="blue", lty=2)
points(x=beta_hat_norm_eq[2], y=beta_hat_norm_eq[3],
col="black", pch=4, cex=3)
```

Python Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

communities = pd.read_csv("communities.data", header=None,
                             sep=",", engine='python')

#Changing "?" missing values to be shown as NA
communities = communities.replace('?', np.nan)


#Droping NA values
communities = communities.dropna()
#print (communities)


y = communities.iloc[:, 127]
X = communities.iloc[:, [17,26,27,31,32,37,76,90,95] ]

#print(X)

communities.insert(0, 'b0', 1)
X_train, X_test, y_train, y_test = train_test_split(X,y,
                             test_size=0.30, random_state=
                             1551)

X_scale = MinMaxScaler().fit(X_train)
```

```
X_train_trans = X_scale.transform(X_train) # fit on
                             training set and transform the
                              data
X_train = pd.DataFrame(X_train_trans, columns = list(
                             X_train.columns)) # convert
                             matrix to data frame with
                             columns

y_scale = MinMaxScaler().fit(np.array(y_train).reshape(-1,
                             1))
y_train = y_scale.transform(np.array(y_train).reshape(-1, 1
                             ))

# Scale the test set using the X and y scalers
X_test_trans = X_scale.transform(X_test)
X_test = pd.DataFrame(X_test_trans, columns = list(X_test.
                             columns))
y_test = y_scale.transform(np.array(y_test).reshape(-1, 1))
y_test = y_test.flatten()




X_train = np.column_stack(([1]*X_train.shape[0], X_train))
                             # add a column with ones for
                             the bias value while
                             converting it into a matrix
m,n = X_train.shape
theta = np.array([1] * n) # initial theta
X = np.array(X_train) # convert X_train into a numpy matrix
y = y_train.flatten() # convert y into an array

alpha = 0.1 # alpha value
iteration = 100 # iterations
cost = [] # list to store cost values
theta_new = [] # list to store updates coeffient values

for i in range(0, iteration):
pred = np.matmul(X,theta) # Calculate predicted value
J = 1/2 * ((np.square(pred - y)).mean()) # Calculate cost
                             function

t_cols = 0 # iteration for theta values

# Update the theta values for all the features with the
                             gradient of the cost function
for t_cols in range(0,n):
t = round(theta[t_cols] - alpha/m * sum((pred-y)*X[:,t_cols
                             ]),4) # calculate new theta
```

```
                                  value
  theta_new.append(t) # save new theta values in a temporary
                                  array

  # update theta array
  theta = [] # empty the theta array
  theta = theta_new # assign new values of theta to array
  theta_new = [] # empty temporary array
  cost.append(J) # append cost function to the cost array

  plt.figure(figsize=(10,8))
  plt.plot(cost)
  plt.title('Cost Function')
  plt.xlabel('Iterations')
  plt.ylabel('Cost Function Value')
  None
  print(theta)
  plt.show()
```

Results of Task 6— i) The coefficients that have been estimated:

Table 2: Coefficients from stochastic gradient descent in R and python.

	theta0	th1	t2	th3	th4	th5	th6	th7	th8	th9
R	-0.03	-1.15	0.63	0.25	-0.01	0.05	0.74	0.03	0.27	0.20
Python	-0.36	0.21	0.12	0.46	0.11	0.37	0.70	0.28	0.18	0.59

ii) Plot of the cost function:

Figure 2. Cost plot in R

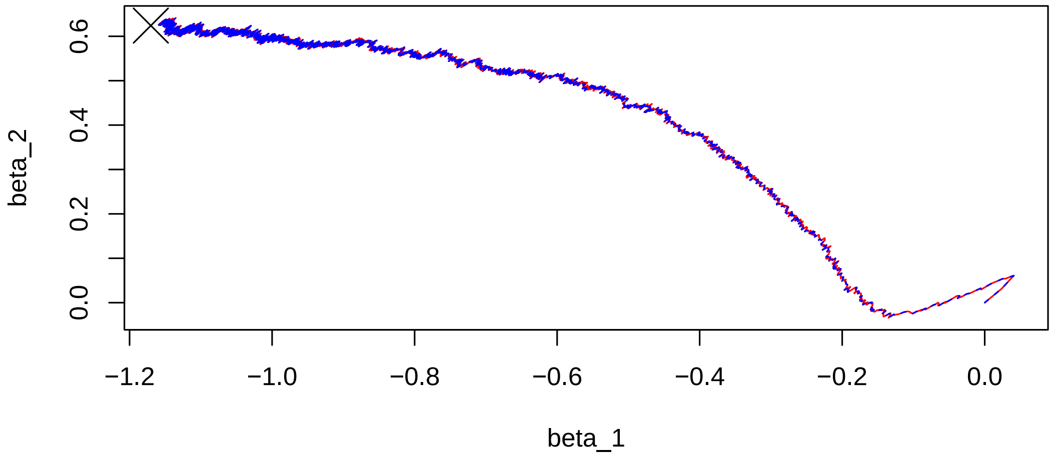
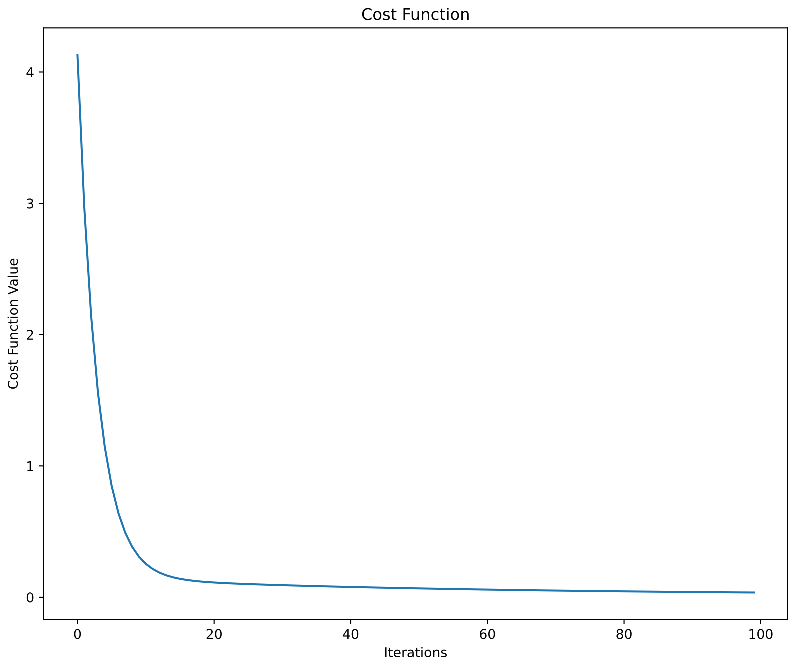


Figure 3. Cost plot in Python



We note that the results of the batch gradient descent method are more similar to the coefficients of the ordinary least squares approach than the

results of the stochastic gradient descent method.
Moreover, the stochastic's gradient descent "Thetas" are constnant while the "Thetas" from batch gradient descent variate with every execution.

Task 7

R Code

```
forestfiresData <- read.csv("C:\\Users\\30697\\Desktop\\ergtzagkara\\
forestfires.csv", sep=",", header=T)

forestfiresData<-na.omit(forestfiresData)

# forecast OLS

frml <- lm ( formula = area ~ temp + wind + rain,
data= forestfiresData)


# predictedValues: vector of values of the dependent variable
predicted by the model

predictedValues <- exp(predict(frml))
predictedValues <- as.vector (predictedValues)

# actualValues: vector of actual values of the dependent variable

actualValues <- forestfiresData$area

#Function that calculates and returns the Root Mean
#Squared Error-RMSE

calculateRMSE<-function(predictedValues, actualValues){
err<- sqrt( mean((actualValues - predictedValues)^2)  )
return( err )
}


#10-Fold Cross Validation

# Function Parameters:
# forestfiresData: the data set to be divided into
#testData and trainData
```

```
# frml: the linear regression model whose predictive accuracy
#will be assessed
# 10 : it's the parts which will be separated the original data set
k<-10
kFoldCrossValidation<-function(forestfiresData, frml, k){

# Randomly shuffle the observations of the dataset
dataset<-forestfiresData[sample(nrow(forestfiresData)),]

#Generate k in number parts of the data set with approximately
#equal number of observations in each segment.
folds <- cut(seq(1,nrow(dataset)), breaks=10, labels=FALSE)

RMSE<-vector()

#Iterative process where each of the 10 segments will be
#used sequentially as the control set for the regression
#model and all the rest as the training set.The process will
#terminate if all sections have been used as a check set.

for(i in 1:10){
# Define the control part for the current iteration
testIndexes <- which(folds==i,arr.ind=TRUE)
# Define control set of the model
testData <- dataset[testIndexes, ]
# train data, all other than the data used for control
trainData <- dataset[-testIndexes, ]
# Estimate coefficients ocandidate.linear.model<-lm( frml,
data = trainData)f the regression model using the training set
candidate.linear.model<-lm( frml, data = trainData)
# Calculate the values of the dependent variable predicted by the
#model for the current control set values
predicted<-predict(candidate.linear.model, testData)
# Calculate RMSE
error<-calculateRMSE(predicted, testData[, "area"])
RMSE<-c(RMSE, error)
}
# Return average value of the errors that occurred from
#all control parts
return( mean(RMSE) )
}

#RMSE
```

```
predictionModel<-vector()
predictionModel[1]<-"area~temp+wind+rain"
modelMeanRMSE<-vector()

for (k in 1:length(predictionModel)){
#10-fold cross-validation for the linear regression model k
modelErr<-kFoldCrossValidation(forestfiresData,
as.formula(predictionModel[k]), 10)

modelMeanRMSE<-c(modelMeanRMSE, modelErr)
print( sprintf("Linear␣regression␣model␣[%s]:␣prediction␣error
[%f]", predictionModel[k], modelErr ) )
modelMeanRMSE
}

#modelMeanRMSE

#Second question
kFoldCross32Validation<-function(forestfiresData, frml, k){

# Randomly shuffle the observations of the dataset
dataset <-  forestfiresData[ which(forestfiresData[,"area"] < 3.2),]

#Generate k in number parts of the data set with approximately
#equal number of observations in each segment.
folds <- cut(seq(1,nrow(dataset)), breaks=10, labels=FALSE)

RMSE<-vector()

#Iterative process where each of the 10 segments will be used
#sequentially as the control set for the regression
#model and all the rest as the training set.The process will
#terminate if all sections have been used as a check set.

for(i in 1:10){
# Define the control part for the current iteration
testIndexes <- which(folds==i,arr.ind=TRUE)
# Define control set of the model
testData <- dataset[testIndexes, ]
# train data, all other than the data used for control
trainData <- dataset[-testIndexes, ]
# Estimate coefficients of the regression model using the
#training set
candidate.linear.model<-lm( frml, data = trainData)
# Calculate the values of the dependent variable predicted by the
#model for the current control set values
predicted<-predict(candidate.linear.model, testData)
# Calculate RMSE
error<-calculateRMSE(predicted, testData[, "area"])
RMSE<-c(RMSE, error)
}
# Return average value of the errors that occurred from
#all control parts
return( mean(RMSE) )
}

#RMSE

predictionModel<-vector()
predictionModel[1]<-"area␣~␣temp␣+␣wind␣+␣rain"

modelMeanRMSE<-vector()
for (k in 1:length(predictionModel)){
# 10-fold cross-validation for the linear regression model k
modelErr<-kFoldCross32Validation(forestfiresData,
as.formula(predictionModel[k]), 10)

modelMeanRMSE<-c(modelMeanRMSE, modelErr)
print( sprintf("Linear␣regression␣model␣[%s]:
prediction␣error␣[%f]", predictionModel[k], modelErr ) )
}

modelMeanRMSE
#The model with the lowest mean squared error

bestModelIndex<-which( modelMeanRMSE == min(modelMeanRMSE) )

#The model with the smallest mean square error (the highest accuracy)

print( sprintf("Model␣with␣best␣accuracy␣was:
[%s]␣error:␣[%f]", predictionModel[bestModelIndex],
modelMeanRMSE[bestModelIndex]) )

#For the model with the lowest mean error, its
#coefficients are estimated
```

```
# considering the entire data set as the training set

final.linear.model<-lm( as.formula(predictionModel
[bestModelIndex]),data=forestfiresData )

final.linear.model
```

Python Code

```
from sklearn.linear_model import LinearRegression
from math import sqrt # We'll need sqrt()
import statistics # for mean()
from sklearn.metrics import mean_squared_error # for
                              mean_squared_error which
                              calculates
from sklearn.model_selection import KFold # import KFold
from sklearn.preprocessing import PolynomialFeatures
import numpy as np
import pandas as pd


# Read the data
forfiresData = pd.read_csv("forestfires.csv", header=0, sep
                              =",", engine='python')

# Randomly shuffle the data i.e. mix it up randomly.
# We do this in order to get different partitions during k-
                              fold cross validation between
# different executions of the program.
# This isn't required per se, but we do it for educational
                              purposes
forfiresData  = forfiresData .sample(frac=1).reset_index(
                              drop=True)

# We will execute k-fold cross validation to assess the
                              regression model with respect
                              to
# its accuracy to predict the value of the dependent
                              variable.
# The linear regression model that will be assessed is the
                              following:
# area = b1temp + b2wind + b3rain + b0
# We will assess its accurace in predicting the dependent
                              value using k-fold cross
                              validation
```

```
# First, we setup the k-fold cross validation object.
# We will do a 10-fold cross validation model (i.e. k=10) i
                             .e. the dataset will
# be split in 10 partitions with an approximately equal
                             number of observation in each
                             part.
# We use the KFold object from sklearn and initialize it
                             properly
kf = KFold(n_splits=10)


print("\nLinear regression model: area = b1temp + b2wind +
                             b3rain + b0\n")

# Create an empty array where we will store the calculated
                             RMSE values
# so that we may be able to
allRMSE = np.empty(shape=[0, 1])

# Just a variable to count at which tests we are
testNumber = 0

# Start now iterating over the partitioned dataset,
                             selecting each time a
                             different subset as the
                             testing set.
#
# .split(forfiresData) will split the dataset into 5 parts.
# Now we iterate over these parts. This iteration works as
                             follows:
# variables train_index and test_index will get the indexes
                              of the original dataset (
                             forfiresData)
# that will constitute the training- and testing-set
                             respectively.
# This for loop will be executed 10 times, equal to the
                             number of partitions.
for train_index, test_index in kf.split(forfiresData):

# Next test
testNumber += 1

# Use the current indexes train_index and test_index to get
                              the actual observations for
                             the
# training and testing of the model respectively.

# From the original, complete dataset get the data with
                             which we will use TRAIN our
```

```
                              model (aka the training set),
# i.e. estimate the coefficients. We get the rows
                              designated by train_index and
                              all their columns/variables
trainingData = forfiresData.iloc[train_index,:]

# From the original, complete dataset get the data with
                              which we will TEST our model,
                              i.e. estimate its
# accuracy. This is the "unknown" dataset.
# Note: we do know the value of the dependent variable area
                               for the testing set
# and hence we will be able to estimate the prediction
                              error/accuracy
testData = forfiresData.iloc[test_index,:]

# Use the training data to estimate the coefficients of the
                               multiple linear regression
                              model.
# The model we will estimate is: area = b1temp + b2wind +
                              b3rain + b0
lm = LinearRegression(normalize=False, fit_intercept=True)

# Since the linear regression model has as independent
                              variables temp, wind and rain,
# we get these variables from the training dataset.
# The method .fit() does the actual estimation of the
                              coefficients for the linear
                              regression
# model using OLS (Ordinary Least Squares).
# The first argument of the .fit() method are the values of
                               the independent variables -in
                               our case
# trainingData.loc[:,['temp','wind' , 'rain']] - and the
                              second
# argument are the values of the dependent variable, here
                              trainingData.loc[:,['area']].
estimatedModel = lm.fit(trainingData.loc[:,['temp','wind',
                              'rain']], trainingData.loc[:,[
                              'area']])

# Coefficients have been estimated. Take a look at them.
                              Not that it is important but
                              heck, why not.
# NOTE: the coefficients are returned (.coef_) as an array.
                               The two numbers you'll see
                              when
# printing .coef_  must be interpreted as follows: first
                              number is the coefficient
```

```
# for temp, second number the coefficient for wind and the
                             third number the coefficient
                             for rain. The estimated
                             constant term b0 can retrieved
# via the .intercept_ variable.
print(">>>Iteration ", testNumber, sep='')
print("\tEstimated coefficients:")
print("\t\tb1=", estimatedModel.coef_[0][0] , sep='')
print("\t\tb2=", estimatedModel.coef_[0][1] , sep='')
print("\t\tb3=", estimatedModel.coef_[0][2] , sep='')
print("\t\tb0=", estimatedModel.intercept_, sep='')


# Now, use the estimated model to predict the value for
                             area for the observations in
                             the testing set
# (i.e. the unknown data that was not used for estimating
                             the coefficients).
# For this, we'll use the model's .predict() method which
                             takes as argument the values
                             of the independent variables
# in the linear regression model.
# IMPORTANT NOTE: Since we gave to the .fit() method above
                             the the variables in a
                             specific order (temp, wind and
                              rain) we
# have to give the respective variables of the testing set
                             in the same order.
# The .predict() method will return a vector of predicted
                             values, one for each
                             observation in testData.loc
                             [:,['temp','wind','rain']].
# More specifically, the first value in the vector
                             predictedfirearea is the
                             predicted value for area for
                             the first row in
# testData.loc[:,['temp','wind','rain']], the second value
                             is the predicted value for
                             area for the second row
# testData.loc[:,['temp','wind','rain']] etc.
predictedfirearea = estimatedModel.predict(testData.loc[:,[
                             'temp','wind','rain']])

# Calculate the Root Mean Squared Error (RMSE) for this
                             testing set.
# We have the real values of the dependent variable from
                             the original dataset
# which is testData.loc[:,['area']] and the model's
                             predicted values in
```

```
                                 predictedfirearea
# IMPORTANT: Please note the following: the function
                                 mean_squared_error() calculate
                                  the MSE NOT the RMSE. In
# order to get the RMSE, we need to square the value
                                 returned by mean_squared_error
                                 (). See your notes on
# how MSE and RMSE differ.
RMSE = sqrt(mean_squared_error(testData.loc[:,['area']],
                                 predictedfirearea))

# Display the RMSE value
print("\t\tModel RMSE=", RMSE, sep='')

# Also, store the calculated RMSE value into an array, so
                                 that we can calculate the mean
                                  error after k-fold cross
# validation has been finished
allRMSE = np.append(allRMSE, RMSE)

# Here the for-loop ends and restarts again for a different
                                  training- and testing set.


# Ok. The iterations of k-fold cross validation have been
                                 done.
# We have now 5 values of RMSE, one for each testing set.
# We calculate the mean RMSE which gives a better estimate
                                 on how accurate the
                                 predictions
# of the linear regression model is for unknown data and
                                 thus
# a better estimate for the generalization error.
print("\n======================================")
print(" Final result: Mean RMSE of tests:",
statistics.mean(allRMSE), sep='' )
print("=======================================")


#second question


for3firesData =forfiresData.loc[forfiresData["area"]< 3.2 ]

forfiresData=for3firesData.sample(frac=1).reset_index
(drop=True)


kf = KFold(n_splits=10)
```

```
print("\nLinear regression model: area = b1temp + b2wind +
b3rain + b0\n")


allRMSE = np.empty(shape=[0, 1])


testNumber = 0


for train_index, test_index in kf.split(forfiresData):


testNumber += 1


trainingData = forfiresData.iloc[train_index,:]

testData = forfiresData.iloc[test_index,:]


lm = LinearRegression(normalize=False, fit_intercept=True)


estimatedModel = lm.fit(trainingData.loc[:,['temp','wind',
                              'rain']], trainingData.loc[:,[
                              'area']])


print(">>>Iteration ", testNumber, sep='')
print("\tEstimated coefficients:")
print("\t\tb1=", estimatedModel.coef_[0][0] , sep='')
print("\t\tb2=", estimatedModel.coef_[0][1] , sep='')
print("\t\tb3=", estimatedModel.coef_[0][2] , sep='')
print("\t\tb0=", estimatedModel.intercept_, sep='')


predictedfirearea = estimatedModel.predict(testData.loc[:,[
                              'temp','wind','rain']])


RMSE = sqrt(mean_squared_error(testData.loc[:,['area']],
                              predictedfirearea))


print("\t\tModel RMSE=", RMSE, sep='')
```

```
allRMSE = np.append(allRMSE, RMSE)

print("\n===================================")
print(" Final result: Mean RMSE of tests:",
statistics.mean(allRMSE), sep='' )
print("===================================")
```

Results of Task 7

Root Mean Square Error (RMSE) is one of the most popular measures to estimate the accuracy of our forecasting model's predicted values versus the actual or observed values while training the regression models. It measures the error in our predicted values when the target or response variable is a continuous number. Thus, RMSE is a standard deviation of prediction errors or residuals. It indicates how spread out the data is around the line of best fit. It is also an essential criterion in shortlisting the best performing model among different forecasting models that we have trained on one particular dataset. To do so, simply compare the RMSE values across all models and select the one with the lowest value on RMSE.

The prediction models we evaluated concerned a fire data set from regions of Portugal. The prediction model we used is:

$$area = b_1 temp + b_2 wind + b_3 + b_0$$

In the first case we used the whole dataset and we found that the RMSE was 47.13 (in R) and 46.90 (in Python) -deviations in errors are reasonable as the estimation was done in different programming languages.

In the second case we estimated again the coefficients of the same regression model, but this time we used the observations where the value of the dependent variable (variable area) is less than 3.2 hectares (i.e. area ¡ 3.2). The new RMSE is 0.79 (in R) and 0.92 (in Python) -deviations in errors are reasonable as the estimation was done in different programming languages.

Comparing the two RMSE we found that a better prediction is made in the second sample (i.e. where area <3.2) but we need to take into account that in the second sample most of the observations are zero which means that this model would indicate a perfect fit to the data. Therefore the model with the highest accuracy is the model that we used the entire data set.

Task 8

R Code

```r
# calculateSSR
# predictedValues: predicted values for the dependent variable
# actualValues: actual values of the dependent vairable

calculateSSR<-function(predictedValues, actualValues){
        err<- sum( (actualValues - predictedValues)^2  )
        return( err )
}

# createRandomDataFrame
# Parameters
# numVariables: number of variables of the dataframe
# numObservations: observations of the dataset
# minValue: the minimum value an observation can take
# maxValue: the maximum value an observation can take

createRandomDataFrame<-function( numVariables=15, numObservations=80,
minValue=10, maxValue=50){
        data <- replicate(numVariables, runif(numObservations,
        min=minValue, max=maxValue))
        return( as.data.frame(data) )
}

RSquared = data.frame(nVars=numeric(0), nObs=numeric(0),
RSquared=numeric(0))

#nVars: number of variables
for (nVars in 2:15) {
        # number of observations
        for (nObs in (nVars+1):80) {
        #
        # running an ols
        #
        cat( sprintf("Running␣OLS␣with:␣%d␣variables
␣␣␣␣␣␣␣␣and␣%d␣observations...", nVars, nObs))
        if (nObs*0.3 < 1){
                print('SKIPPING')
                next
        }
        if ( (nObs - nObs*0.3) < nVars){
                print('SKIPPING')
                next
        }
        df <- createRandomDataFrame(nVars, nObs)

        testDataPoints <- sample(nrow(df), nObs*0.3)
        trainDF<-df[-testDataPoints,]
        testDF<-df[testDataPoints,]

        estimation <- lm(V1 ~ ., data=trainDF)
        cat( sprintf("R-squared:␣%f\n", summary(estimation)$r.squared)
)
        RSquared[ nrow(RSquared)+1, ] <- c( nVars, nObs,
        summary(estimation)$r.squared)
}
}

cat(sprintf("Total␣of␣%d", nrow(RSquared[ which
(RSquared[,"RSquared"] >=0.80),])))

print( RSquared[ which(RSquared[,"RSquared"] >=0.80), ] )

# One of the cases where overfitting occurs is when we have 11
# variables and 18 observations, since the R squared
# of the OLS estimation is 0.9 (> 0.8)
# We will perform an ols for this dataset in order to see
# the overfitting via the generalization and training error.


df <- createRandomDataFrame(11, 18, 10, 50)
testDataPoints <- sample(nrow(df), 18*0.3)
trainDF<-df[-testDataPoints,]
testDF<-df[testDataPoints,]

#Ols
estimation <- lm(V1 ~ ., data=trainDF)
print( sprintf("R-Squared:␣%f", summary(estimation)$r.squared) )

# training error
predictedTrain<-predict(estimation, trainDF)
trainingError<-calculateSSR(predictedTrain, trainDF$V1)
print( sprintf("Train␣error=%.3f", trainingError))
```

```
# gerenalization error
predictedTest<-predict(estimation, testDF)
generalizationError<-calculateSSR(predictedTest, testDF$V1)
print( sprintf("Generalization␣error=%.3f", generalizationError))

# class difference: generalization error/ training error
class = generalizationError/trainingError
print (class)
```

Results of Task 8

For the last task we generated a random data frame to examine the overfitting state of a model. At first, we defined a function in order to calculate the Sum of Squares error (SSR) and a function for the creation of the dataframe. In order to find a combination of variables and observation where overfitting occurs we performed an automated process. If the R$\hat{2}$ of the model is higher than 0.80 then the model shall be overfitted.

It is evident that a model with 11 variables and 18 observations is one of the models that feature a large value of R$\hat{2}$ ($0.9 > 0.8$). We then proceeded to measure the training and the generalization error for the model, after the OLS estimation.

$$\text{Training error} = 261.403$$
$$\text{Generalization error} = 3490.817$$
$$\text{Their class difference} = 13.35418$$

When the training error is small, while the generalization error is large, the model is overfitted.