**Homework 2**

**Problem 1**

1. Produce a plot of ‘MEDV’ vs ‘RM’.

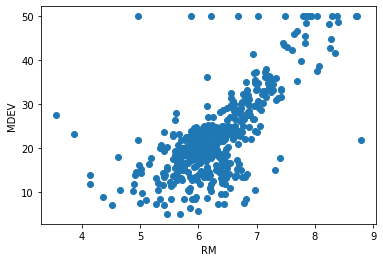
To produce the plot of MEDV vs RM I first imported the dataset from sklearns database. Once the database was loaded in I stored the predictors in a data frame (X) and I stored the response variable (MEDV) in a variable named y. Now to produce the plot I imported that matplotlib feature pyplot and used the below code to produce the corresponding graph.

ax = plt.gca()

ax.scatter(RM, y)

plt.xlabel('RM')

plt.ylabel('MDEV')



1. Use the Validation set approach to estimate the test error rate of the linear regression of ‘MEDV’ on ‘RM’

First step was to split my data set into a test set and a validation set. To do this I randomly took 80% of the observations from my data frame and stored them in a new data frame called train\_df. The remaining observations were stored in a data frame named test\_df. Once I had the data separated I fit a model using Linear Regression based on my training sets. I then used this fit to make a prediction based on my validation set. Once this was finished I took the mean square error of the model. See code below:

train\_df = df.sample(405, random\_state=1)

test\_df = df[~df.isin(train\_df)].dropna(how = 'all')

X\_train = train\_df['RM'].values.reshape(-1,1)

y\_train = train\_df['MDEV']

X\_test = test\_df['RM'].values.reshape(-1,1)

y\_test = test\_df['MDEV']

lm = skl\_lm.LinearRegression()

model = lm.fit(X\_train, y\_train)

pred = model.predict(X\_test)

from sklearn.metrics import mean\_squared\_error

MSE = mean\_squared\_error(y\_test, pred)

print("Linear Regression MSE: ", MSE)

Results:

Linear Regression MSE: 40.42405440528705

1. Use a 10-fold Cross-Validation approach to estimate the test error rate of the linear regression of ‘MEDV’ on ‘RM’

First I imported KFold and cross\_val\_score from sklearn.model\_selection. I then used the KFold command to specift a 10 fold cross-validation and stored that in cv\_k. Next I used the cross\_val\_score function to apply the 10 fold cross-validation as well as score each fold based on mean squared error. See code below:

cv\_k = KFold(n\_splits = 10, random\_state = 1, shuffle=True)

scores\_k = cross\_val\_score(model, X, y, scoring="neg\_mean\_squared\_error", cv = cv\_k, n\_jobs=1)

print("Using KFold: Folds: " + str(len(scores\_k)) +", MSE: " + str(np.mean(np.abs(scores\_k))) + ", STD: " + str(np.std(scores\_k)))

print("Cross-validated scores:", np.abs(scores\_k))

Results:

Using KFold: Folds: 10, MSE: 44.25914803119435, STD: 19.57431339508812

Cross-validated scores: [38.75302125 42.0122301 25.58254465 89.7091174 35.61028149 34.84372916

30.36242184 65.51945985 56.50779903 23.69087554]

1. Use a Leave-One-Out Cross-Validation approach to estimate the testerror rate of the linear regression of ‘MEDV’ on ‘RM’

First I imported LeaveOneOut from sklearn.model\_selection and stored the LeaveOneOut function into a variable called loo. I then stored and reshaped the predictor and target variables into data frames X and y. I set the number of splits to be equivalent to the number of observations of my data and performed KFold cross validation. See code below:

from sklearn.model\_selection import LeaveOneOut

loo = LeaveOneOut()

X = df['RM'].values.reshape(-1,1)

y = df['MDEV'].values.reshape(-1,1)

loo.get\_n\_splits(X)

cv\_loo = KFold(n\_splits = len(X), random\_state=None, shuffle=True)

scores\_loo = cross\_val\_score(model, X, y, scoring="neg\_mean\_squared\_error", cv=cv\_loo, n\_jobs=1)

print("Using LOOCV Folds: " + str(len(scores\_loo)), ", MSE: " + str(np.mean(np.abs(scores\_loo))) + ", STD: " +str(np.std(scores\_loo)))

Results:

Using LOOCV Folds: 506 , MSE: 44.216664193967986, STD: 119.35718726583212

1. 5. Compare your results between the 3 methods.

The MSE of linear regression was about 40, KFold MSE was about 44 and LOOCV MSE was about 44. Linear Regression seems to be the best way to model this data.

**Problem 2**

1. Make a histogram of the bootstrap samples of R for describing the bootstrap sampling distribution of R.

First I had to make a bootstrap function to take random observations of my data and evaluating the R correlation coefficient of those resampled observations. I did this by creating a nested loop in which the inner loop chose 10 random observations from the given data set and the outer loop evaluated the correlation coefficient of those resampled observations. The R values were then appended to a list called sample\_R which would be returned at the end of the loop. Once this is returned, I then created a histogram to show the distribution of R values produced. See code below:

def bt(df, B):

sample\_R=[]

data = []

for i in range(B):

for j in range(10):

x = np.random.choice(503, 1)

n = df.iloc[x,:]

data.append(dict(n))

y = pd.DataFrame(data)

z = y.astype(float)

corr = z.corr(method='pearson')

corr1 = corr.iloc[0,1]

sample\_R.append(corr1)

data=[]

return(sample\_R)

test = bt(df,50)

graph = pd.DataFrame(test)

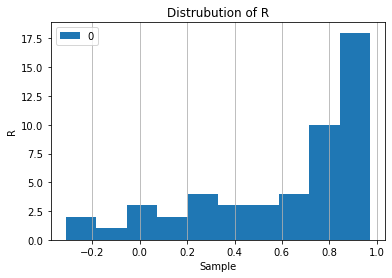
graph.plot.hist(grid = True,bins = 10, rwidth=1)

plt.title('Distrubution of R')

plt.xlabel('Sample')

plt.ylabel('R')

plt.grid(axis='y', alpha=0.75)



1. Provide a bootstrap estimate of R and an estimate of its standard error.

Once the results were stored in a data frame called test I used numpy functions to obtain the median, mean, and standard deviation. See code below:

print("Median estimate: ", np.median(test))

print("Average estimate: ", np.mean(test))

print("Standard Error estimate: ", np.std(test))

Results:

Median estimate: 0.7387406762471954

Average estimate: 0.6548054756074542

Standard Error estimate: 0.3085706048730679

1. Provide a 95% confidence interval for the correlation coefficient, R, of ‘MEDV’ and ‘RM’.

To create the confidence interval, I used the numpy function quantile and stored it in a variable named basic. See code below:

basic = np.quantile(test,(0.025,0.975))

print("95% confidence interval: ", basic)

Results:

95% confidence interval: [-0.1297047 0.96723753]

**Problem 3**

1. Use the bootstrap approach to obtain the coefficient estimates β0 and β1.

To answer this question I created a function called bt\_beta which included a nested loop. The inner loop randomly chose 10 observations from the given data and stored it in a list called data. The outer loop reformatted the data from a list to a dataframe as well as coercing the data type to type ‘float’. Then the data would be separated into two new arrays containing the predictor and response variables. Then a liner regression model would be created based on these two arrays. The intercept and coefficients were then evaluated and appended to two separate list one being sample\_beta and the other being sample\_intercept. However I needed the data to be stored in a data frame instead so it was then reformatted into two separate data frames but then recombined into a new data frame named ‘results’. This is what would be returned at the end. Once the function ended and the data frame returned I separated the data into two new data frames. One for intercepts and one for beta1. I then took the mean of each of these. See code below:

def bt\_beta(df,B):

data = []

sample\_beta = []

sample\_intercept = []

lm = skl\_lm.LinearRegression()

for i in range(B):

for j in range(10):

x = np.random.choice(503, 1)

n = df.iloc[x,:]

data.append(dict(n))

names = ['RM', 'MDEV']

frame = pd.DataFrame(data = data, columns = names)

flo = frame.astype(float)

X = np.array(flo['RM']).reshape(-1,1)

y = np.array(flo['MDEV']).reshape(-1,1)

n = lm.fit(X,y)

c = n.coef\_

inter = n.intercept\_

sample\_intercept.append(inter)

sample\_beta.append(c)

data = []

#result = {'Intercept': [sample\_intercept], 'Beta': [sample\_beta]}

int\_result = pd.DataFrame(sample\_intercept)

results= pd.DataFrame(int\_result)

beta\_result = np.array(sample\_beta).reshape(-1,1)

beta\_formatted = pd.DataFrame(beta\_result)

results['Beta'] = pd.DataFrame(beta\_formatted)

return(results)

X = pd.DataFrame(df['RM'])

y = pd.DataFrame(df['MDEV'])

test2 = bt\_beta(df, 50)

intercepts = test2.iloc[:,0]

betas = test2['Beta']

intercept\_result = np.mean(intercepts)

beta\_result = np.mean(betas)

print("Intercept Estimate: ", intercept\_result )

print("Beta Estimate", beta\_result)

Results:

Intercept Estimate: -36.56791401230322

Beta Estimate 9.352557823957902

1. Use the bootstrap approach to obtain estimates of the standard errors for β0 and β1.

I created a new function called bt\_beta\_SE which included a nested loop similar to the previous ones. However, this time I used a matrix for data to store my observations rather than a list due to type errors received. Once this error was avoided I followed the same procedure that I did for the beta estimate bootstrap function. See code below:

def bt\_beta\_SE(df, B):

data = np.zeros((B,2))

sample\_betaSE = []

sample\_intSE = []

for i in range(B):

for j in range(10):

x = np.random.choice(50, 1)

n = df.iloc[x,:]

data[i] = n

names = ['Intercept', 'Beta']

frame = pd.DataFrame(data = data, columns = names)

flo = frame.astype(float)

X = np.array(flo['Intercept']).reshape(-1,1)

y = np.array(flo['Beta']).reshape(-1,1)

int\_se = np.std(X)

beta\_se = np.std(y)

sample\_betaSE.append(beta\_se)

sample\_intSE.append(int\_se)

int\_result = np.array(sample\_intSE)

int\_formatted = pd.DataFrame(int\_result)

results= pd.DataFrame(int\_formatted)

beta\_result = np.array(sample\_betaSE)

beta\_formatted = pd.DataFrame(beta\_result)

results['Beta'] = pd.DataFrame(beta\_formatted)

return(results)

test3 = bt\_beta\_SE(test2, 50)

intercept\_se = test3.iloc[:,0]

beta\_se = test3['Beta']

intercept\_se\_avg = np.mean(intercept\_se)

beta\_se\_avg = np.mean(beta\_se)

print("Intercept SE Estimate: ", intercept\_se\_avg)

print("Beta SE Estimate", beta\_se\_avg)

Results:

Intercept SE Estimate: 20.942908415993674

Beta SE Estimate 4.401775508047705

1. Compare your results with the ones obtained using the linear regression commands from Python.

First I imported statsmodels.api for the linear regression commands. I created a new data frame called lr and stored the predictor variables in a column called x and the response variables in a column called y. I then used the regression commands to perform linear regression. The coefficient estimates were on par with my results, however the standard estimates from the regression commands were different. However, the output comes with a disclaimer that the standard error estimates could be inaccurate. See code below:

import statsmodels.api as sm

lr = pd.DataFrame()

lr['x'] = df['RM']

lr['y'] = df['MDEV']

lm = sm.OLS.from\_formula('y~x', lr)

result = lm.fit()

print(result.summary())

OLS Regression Results

==============================================================================

Dep. Variable: y R-squared: 0.484

Model: OLS Adj. R-squared: 0.483

Method: Least Squares F-statistic: 471.8

Date: Wed, 09 Jun 2021 Prob (F-statistic): 2.49e-74

Time: 18:35:09 Log-Likelihood: -1673.1

No. Observations: 506 AIC: 3350.

Df Residuals: 504 BIC: 3359.

Df Model: 1

Covariance Type: nonrobust

==============================================================================

coef std err t P>|t| [0.025 0.975]

------------------------------------------------------------------------------

Intercept -34.6706 2.650 -13.084 0.000 -39.877 -29.465

x 9.1021 0.419 21.722 0.000 8.279 9.925

==============================================================================

Omnibus: 102.585 Durbin-Watson: 0.684

Prob(Omnibus): 0.000 Jarque-Bera (JB): 612.449

Skew: 0.726 Prob(JB): 1.02e-133

Kurtosis: 8.190 Cond. No. 58.4

==============================================================================

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

1. Approximate a 95% confidence interval forβ1using the formula:

[ˆβ1−2SE(ˆβ1),ˆβ1+ 2SE(ˆβ1)]

I had the beta1 results stored in a variable called beta\_results and I had the standard error stored in a variable called beta\_se\_avg. I then used these to plug into the formula and stored the results in an array. See code below:

#confidence interval

lower\_bound = beta\_result - 2\*beta\_se\_avg

upper\_bound = beta\_result + 2\*beta\_se\_avg

CI = np.array([lower\_bound,upper\_bound])

print("Confidence Interval:", CI)

Results:

Confidence Interval: [ 0.54900681 18.15610884]