Homework 2

CAP 5610 - Machine Learning

Suma Marri

March 09, 2022

Problem 1: Linear Regression Model

In [1]:

```
# Import necessary packages to the jupyter notebook
# Implement a Linear Regression model using both Normal Equation Method and SGD
import pandas as pd
import numpy as np
from pandas import read_csv

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import accuracy_score

# read and load the csv data file
filename = "Dataset/AMZN.csv"
data = read_csv ( filename )

# Get the Adjusted Close Price
data_select = data [['Adj Close']]

# converting the dataset to a numpy array
values = data_select.values
```

In [2]:

```
from pandas import DataFrame
from pandas import concat

"""
Frame a time series as a supervised learning dataset .
Arguments :
data : Sequence of observations as a list or NumPy array .
n_in : Number of lag observations as input (X).
n_out : Number of observations as output (y).
dropnan : Boolean whether or not to drop rows with NaN values .
Returns :
Pandas DataFrame of series framed for supervised learning .
"""

def series_to_supervised ( data , n_in =1 , n_out =1 , dropnan = True ):
    n_vars = 1 if type ( data ) is list else data . shape [1]
    df = DataFrame ( data )
    cols , names = list () , list ()
    # input sequence (t-n, ... t-1)
    for i in range ( n_in , 0 , -1 ):
        cols . append ( df . shift ( i ) )
        names += [('var %d(t-%d)' % ( j+1 , i ) ) for j in range ( n_vars )]
```

```
    # forecast sequence (t, t+1, ... t+n)
    for i in range (0 , n_out ):
        cols.append ( df . shift (-i ) )
        if i == 0:
            names += [('var%d(t)' % ( j+1 ) ) for j in range ( n_vars )]
        else :
            names += [('var%d(t+%d)' % ( j+1 , i ) ) for j in range ( n_vars )]
    # put it all together
    agg = concat ( cols , axis =1 )
    agg.columns = names
    # drop rows with NaN values
    if dropnan :
        agg.dropna( inplace = True )
    return agg
```

(a)

Use the Python function named series_to_supervised() that takes a univariate or multivariate time series and frames it as a supervised learning dataset.

In [3]:
```
series_to_supervised(data_select, n_in=10, n_out=1, dropnan=True)
```

Out[3]:

	var 1(t-10)	var 1(t-9)	var 1(t-8)	var 1(t-7)	var 1(t-6)	var 1(t-5)	var 1(t-4)	va
10	1.958333	1.729167	1.708333	1.635417	1.427083	1.395833	1.500000	1.
11	1.729167	1.708333	1.635417	1.427083	1.395833	1.500000	1.583333	1.
12	1.708333	1.635417	1.427083	1.395833	1.500000	1.583333	1.531250	1.
13	1.635417	1.427083	1.395833	1.500000	1.583333	1.531250	1.505208	1.
14	1.427083	1.395833	1.500000	1.583333	1.531250	1.505208	1.500000	1.
...	...	...	...	...	...	...	...	
5753	1676.609985	1785.000000	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.
5754	1785.000000	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.
5755	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.
5756	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.839966	1955.
5757	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.839966	1955.489990	1900.

5748 rows × 11 columns

In [4]:
```
supervised_data = series_to_supervised(data_select, n_in=10, n_out=1, dropnan=True)
supervised_data
```

Out[4]:

	var 1(t-10)	var 1(t-9)	var 1(t-8)	var 1(t-7)	var 1(t-6)	var 1(t-5)	var 1(t-4)	va
10	1.958333	1.729167	1.708333	1.635417	1.427083	1.395833	1.500000	1.
11	1.729167	1.708333	1.635417	1.427083	1.395833	1.500000	1.583333	1.
12	1.708333	1.635417	1.427083	1.395833	1.500000	1.583333	1.531250	1.

	var 1(t-10)	var 1(t-9)	var 1(t-8)	var 1(t-7)	var 1(t-6)	var 1(t-5)	var 1(t-4)	va
13	1.635417	1.427083	1.395833	1.500000	1.583333	1.531250	1.505208	1.
14	1.427083	1.395833	1.500000	1.583333	1.531250	1.505208	1.500000	1.
...	...	...	...	...	...	...	...	
5753	1676.609985	1785.000000	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.
5754	1785.000000	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.
5755	1689.150024	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.
5756	1807.839966	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.839966	1955.
5757	1830.000000	1880.930054	1846.089966	1902.829956	1940.099976	1885.839966	1955.489990	1900.

5748 rows × 11 columns

(b)

Use MinMaxScaler to scale your data

In [5]:
```
scaler = MinMaxScaler()
supervised_data = scaler.fit_transform(supervised_data)
```

(c)

Use the Normal Equation Method to find the linear regression coefficients (w). To perform this you may want to take the following steps first: Split your data to X and Y by taking the columns var1(t-10),...,var(t-1) as your 10 features in X, and take the last column var1(t) as your target (Y). Expand your matrix X with a bias vector of ones as the first column (to accomplish this, you may want to use the numpy operations np.ones , np.reshape and np.append). Use the train test split with 'random state=1' to split your data to 70% training, and 30% test data. Solve the Normal Equation Method in (2) to find the coefficients w.

In [6]:
```
Y = supervised_data[:,-1]
X = supervised_data[:,0:-1]
```

In [7]:
```
print(X.shape)
print(Y.shape)
```

```
(5748, 10)
(5748,)
```

In [8]:
```
def normalEquation(X, Y):
    m = int(np.size(X[:, 1]))
    # This is the feature / parameter (2x2) vector that will
    # contain my minimized values
    theta = []

    # create a bias_vector to add to my newly created X vector
    bias_vector = np.ones((m, 1))
```

```
    # combine these two vectors together to get a (m, 2) matrix
    X = np.append(bias_vector, X, axis=1)
    # Normal Equation:
    # theta = inv(X^T * X) * X^T * y

    # For convenience create a new, tranposed X matrix
    X_transpose = np.transpose(X)

    # Calculating theta
    theta = np.linalg.inv(X_transpose.dot(X))
    theta = theta.dot(X_transpose)
    theta = theta.dot(Y)

    return theta

p = normalEquation(X, Y)

print(p)
```

```
[ 7.58927888e-05 -5.67708993e-02  2.09842519e-01 -1.65870717e-01
  2.89934644e-02 -6.11953566e-02  7.25451627e-02 -4.27946096e-02
  4.97008003e-03  7.34127990e-02  9.37474158e-01]
```

In [9]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.3,
random_state=1)
coef = normalEquation(X_train, Y_train)
coef
```

Out[9]:
```
array([ 1.26111054e-04, -4.83348220e-02,  2.24558809e-01, -1.52228741e-01,
        3.28901068e-02, -1.08811996e-01,  7.17528590e-02, -2.60760209e-02,
       -1.48358090e-03,  8.01298345e-02,  9.28047368e-01])
```

(d)

Make a prediction on your test set using the linear regression function f(x) = wT x, and use both the mean square error and coefficient of determination R2 to measure the performance of your prediction model. For this use fucntions mean squared error and r2 score from sklearn library

In [10]:
```
def predict(row, coefficients):
    yhat = coefficients[0] # is bias
    for i in range(len(row)):
        yhat = yhat + coefficients[i + 1] * row[i] # b+ W * x(inputs - row)
    return yhat
```

In [11]:
```
Y_pred = np.array(predict(X_test[0,:], coef))
for i in range(X_test.shape[0]-1):
    Y_pred = np.append(Y_pred, predict(X_test[i+1,:], coef))
```

In [12]:
```
print(mean_squared_error(Y_pred, Y_test))
```

```
2.4148595608723037e-05
```

In [13]:
```
print(r2_score(Y_pred, Y_test))
```

```
0.9995582657234232
```

(e)

Next, find the coefficients w using gradient descent algorithm and monitor how your error changes in each epoch; You can create a function coefficients sgd similar to what we did in our Lab Session 7. Note that you may have to make some minor changes to this part of the code (coefficients sgd for linear regression, in lab session 7), due to the additional bias term 1 in your matrix X. For this part, use learning rate 0.01, and number of epochs (iterations) 200.

In [14]:
```
def coefficients_sgd(X_train, Y_train, l_rate, n_epoch): #l-rate is learning rate
  #initializing all coefficients to zero
  coef = [0.0 for i in range(len(X_train[0])+1)]
  for epoch in range(n_epoch):
    sum_error = 0 # loss
    for i in range(X_train.shape[0]):
      # calculating the prediction using current coeeficients
      yhat = predict(X_train[i,:], coef)
      # calculating error
      error = yhat - Y_train[i] #yhat is prediction, Y_train is ground truth,
      sum_error += error**2 # error square, because loss cannot be negative, or we want
      #stochastic gradient descent
      coef[0] = coef[0] - l_rate * error
      for j in range(len(coef)-1):
        coef[j + 1] = coef[j + 1] - l_rate * error * X_train[i,j]

    print( ' >epoch=%d, lrate=%.3f, error=%.3f ' % (epoch, l_rate, sum_error))
  #returning the list of coefficients
  return coef
```

In [15]:
```
l_rate = 0.01
n_epoch = 200
coef = coefficients_sgd(X_train, Y_train, l_rate, n_epoch)
```

```
>epoch=0, lrate=0.010, error=5.666
>epoch=1, lrate=0.010, error=0.453
>epoch=2, lrate=0.010, error=0.447
>epoch=3, lrate=0.010, error=0.442
>epoch=4, lrate=0.010, error=0.437
>epoch=5, lrate=0.010, error=0.432
>epoch=6, lrate=0.010, error=0.427
>epoch=7, lrate=0.010, error=0.422
>epoch=8, lrate=0.010, error=0.417
>epoch=9, lrate=0.010, error=0.413
>epoch=10, lrate=0.010, error=0.408
>epoch=11, lrate=0.010, error=0.404
>epoch=12, lrate=0.010, error=0.399
>epoch=13, lrate=0.010, error=0.395
>epoch=14, lrate=0.010, error=0.391
>epoch=15, lrate=0.010, error=0.387
>epoch=16, lrate=0.010, error=0.383
>epoch=17, lrate=0.010, error=0.379
>epoch=18, lrate=0.010, error=0.375
>epoch=19, lrate=0.010, error=0.371
>epoch=20, lrate=0.010, error=0.367
>epoch=21, lrate=0.010, error=0.364
>epoch=22, lrate=0.010, error=0.360
```

```
>epoch=23, lrate=0.010, error=0.357
>epoch=24, lrate=0.010, error=0.353
>epoch=25, lrate=0.010, error=0.350
>epoch=26, lrate=0.010, error=0.347
>epoch=27, lrate=0.010, error=0.343
>epoch=28, lrate=0.010, error=0.340
>epoch=29, lrate=0.010, error=0.337
>epoch=30, lrate=0.010, error=0.334
>epoch=31, lrate=0.010, error=0.331
>epoch=32, lrate=0.010, error=0.328
>epoch=33, lrate=0.010, error=0.326
>epoch=34, lrate=0.010, error=0.323
>epoch=35, lrate=0.010, error=0.320
>epoch=36, lrate=0.010, error=0.317
>epoch=37, lrate=0.010, error=0.315
>epoch=38, lrate=0.010, error=0.312
>epoch=39, lrate=0.010, error=0.310
>epoch=40, lrate=0.010, error=0.307
>epoch=41, lrate=0.010, error=0.305
>epoch=42, lrate=0.010, error=0.302
>epoch=43, lrate=0.010, error=0.300
>epoch=44, lrate=0.010, error=0.298
>epoch=45, lrate=0.010, error=0.296
>epoch=46, lrate=0.010, error=0.293
>epoch=47, lrate=0.010, error=0.291
>epoch=48, lrate=0.010, error=0.289
>epoch=49, lrate=0.010, error=0.287
>epoch=50, lrate=0.010, error=0.285
>epoch=51, lrate=0.010, error=0.283
>epoch=52, lrate=0.010, error=0.281
>epoch=53, lrate=0.010, error=0.279
>epoch=54, lrate=0.010, error=0.277
>epoch=55, lrate=0.010, error=0.275
>epoch=56, lrate=0.010, error=0.274
>epoch=57, lrate=0.010, error=0.272
>epoch=58, lrate=0.010, error=0.270
>epoch=59, lrate=0.010, error=0.268
>epoch=60, lrate=0.010, error=0.267
>epoch=61, lrate=0.010, error=0.265
>epoch=62, lrate=0.010, error=0.264
>epoch=63, lrate=0.010, error=0.262
>epoch=64, lrate=0.010, error=0.260
>epoch=65, lrate=0.010, error=0.259
>epoch=66, lrate=0.010, error=0.257
>epoch=67, lrate=0.010, error=0.256
>epoch=68, lrate=0.010, error=0.255
>epoch=69, lrate=0.010, error=0.253
>epoch=70, lrate=0.010, error=0.252
>epoch=71, lrate=0.010, error=0.250
>epoch=72, lrate=0.010, error=0.249
>epoch=73, lrate=0.010, error=0.248
>epoch=74, lrate=0.010, error=0.246
>epoch=75, lrate=0.010, error=0.245
>epoch=76, lrate=0.010, error=0.244
>epoch=77, lrate=0.010, error=0.243
>epoch=78, lrate=0.010, error=0.242
>epoch=79, lrate=0.010, error=0.240
>epoch=80, lrate=0.010, error=0.239
>epoch=81, lrate=0.010, error=0.238
>epoch=82, lrate=0.010, error=0.237
```

```
>epoch=83, lrate=0.010, error=0.236
>epoch=84, lrate=0.010, error=0.235
>epoch=85, lrate=0.010, error=0.234
>epoch=86, lrate=0.010, error=0.233
>epoch=87, lrate=0.010, error=0.232
>epoch=88, lrate=0.010, error=0.231
>epoch=89, lrate=0.010, error=0.230
>epoch=90, lrate=0.010, error=0.229
>epoch=91, lrate=0.010, error=0.228
>epoch=92, lrate=0.010, error=0.227
>epoch=93, lrate=0.010, error=0.226
>epoch=94, lrate=0.010, error=0.225
>epoch=95, lrate=0.010, error=0.224
>epoch=96, lrate=0.010, error=0.223
>epoch=97, lrate=0.010, error=0.223
>epoch=98, lrate=0.010, error=0.222
>epoch=99, lrate=0.010, error=0.221
>epoch=100, lrate=0.010, error=0.220
>epoch=101, lrate=0.010, error=0.219
>epoch=102, lrate=0.010, error=0.218
>epoch=103, lrate=0.010, error=0.218
>epoch=104, lrate=0.010, error=0.217
>epoch=105, lrate=0.010, error=0.216
>epoch=106, lrate=0.010, error=0.215
>epoch=107, lrate=0.010, error=0.215
>epoch=108, lrate=0.010, error=0.214
>epoch=109, lrate=0.010, error=0.213
>epoch=110, lrate=0.010, error=0.213
>epoch=111, lrate=0.010, error=0.212
>epoch=112, lrate=0.010, error=0.211
>epoch=113, lrate=0.010, error=0.211
>epoch=114, lrate=0.010, error=0.210
>epoch=115, lrate=0.010, error=0.209
>epoch=116, lrate=0.010, error=0.209
>epoch=117, lrate=0.010, error=0.208
>epoch=118, lrate=0.010, error=0.208
>epoch=119, lrate=0.010, error=0.207
>epoch=120, lrate=0.010, error=0.206
>epoch=121, lrate=0.010, error=0.206
>epoch=122, lrate=0.010, error=0.205
>epoch=123, lrate=0.010, error=0.205
>epoch=124, lrate=0.010, error=0.204
>epoch=125, lrate=0.010, error=0.204
>epoch=126, lrate=0.010, error=0.203
>epoch=127, lrate=0.010, error=0.203
>epoch=128, lrate=0.010, error=0.202
>epoch=129, lrate=0.010, error=0.201
>epoch=130, lrate=0.010, error=0.201
>epoch=131, lrate=0.010, error=0.200
>epoch=132, lrate=0.010, error=0.200
>epoch=133, lrate=0.010, error=0.200
>epoch=134, lrate=0.010, error=0.199
>epoch=135, lrate=0.010, error=0.199
>epoch=136, lrate=0.010, error=0.198
>epoch=137, lrate=0.010, error=0.198
>epoch=138, lrate=0.010, error=0.197
>epoch=139, lrate=0.010, error=0.197
>epoch=140, lrate=0.010, error=0.196
>epoch=141, lrate=0.010, error=0.196
>epoch=142, lrate=0.010, error=0.195
```

```
>epoch=143, lrate=0.010, error=0.195
>epoch=144, lrate=0.010, error=0.195
>epoch=145, lrate=0.010, error=0.194
>epoch=146, lrate=0.010, error=0.194
>epoch=147, lrate=0.010, error=0.193
>epoch=148, lrate=0.010, error=0.193
>epoch=149, lrate=0.010, error=0.193
>epoch=150, lrate=0.010, error=0.192
>epoch=151, lrate=0.010, error=0.192
>epoch=152, lrate=0.010, error=0.192
>epoch=153, lrate=0.010, error=0.191
>epoch=154, lrate=0.010, error=0.191
>epoch=155, lrate=0.010, error=0.190
>epoch=156, lrate=0.010, error=0.190
>epoch=157, lrate=0.010, error=0.190
>epoch=158, lrate=0.010, error=0.189
>epoch=159, lrate=0.010, error=0.189
>epoch=160, lrate=0.010, error=0.189
>epoch=161, lrate=0.010, error=0.188
>epoch=162, lrate=0.010, error=0.188
>epoch=163, lrate=0.010, error=0.188
>epoch=164, lrate=0.010, error=0.187
>epoch=165, lrate=0.010, error=0.187
>epoch=166, lrate=0.010, error=0.187
>epoch=167, lrate=0.010, error=0.187
>epoch=168, lrate=0.010, error=0.186
>epoch=169, lrate=0.010, error=0.186
>epoch=170, lrate=0.010, error=0.186
>epoch=171, lrate=0.010, error=0.185
>epoch=172, lrate=0.010, error=0.185
>epoch=173, lrate=0.010, error=0.185
>epoch=174, lrate=0.010, error=0.184
>epoch=175, lrate=0.010, error=0.184
>epoch=176, lrate=0.010, error=0.184
>epoch=177, lrate=0.010, error=0.184
>epoch=178, lrate=0.010, error=0.183
>epoch=179, lrate=0.010, error=0.183
>epoch=180, lrate=0.010, error=0.183
>epoch=181, lrate=0.010, error=0.183
>epoch=182, lrate=0.010, error=0.182
>epoch=183, lrate=0.010, error=0.182
>epoch=184, lrate=0.010, error=0.182
>epoch=185, lrate=0.010, error=0.182
>epoch=186, lrate=0.010, error=0.181
>epoch=187, lrate=0.010, error=0.181
>epoch=188, lrate=0.010, error=0.181
>epoch=189, lrate=0.010, error=0.181
>epoch=190, lrate=0.010, error=0.180
>epoch=191, lrate=0.010, error=0.180
>epoch=192, lrate=0.010, error=0.180
>epoch=193, lrate=0.010, error=0.180
>epoch=194, lrate=0.010, error=0.179
>epoch=195, lrate=0.010, error=0.179
>epoch=196, lrate=0.010, error=0.179
>epoch=197, lrate=0.010, error=0.179
>epoch=198, lrate=0.010, error=0.179
>epoch=199, lrate=0.010, error=0.178
```

(f)

Make a prediction using the coefficients you found from SGD algorithm in previous step (Y prediction sgd = X test.dot(coef sgd)); Use both the mean square error and coefficient of determination R2 to measure the performance of your predictions; compare the results with your prediction performance in part d where you used the coefficients found from Normal Equation Method. Which method gives you better results?

In [16]:
```
Y_pred = np.array(predict(X_test[0,:], coef))
for i in range(X_test.shape[0]-1):
  Y_pred = np.append(Y_pred, predict(X_test[i+1,:], coef))
```

In [17]:
```
print(mean_squared_error(Y_pred, Y_test))
```

```
3.5368255527927726e-05
```

In [18]:
```
print(r2_score(Y_pred, Y_test))
```

```
0.9993455884170656
```

For R2 score in the normal equation, we get 0.9995582657234232 and the R2 score in SGD is 0.9993455884170656. The MSE for the normal equation is 2.4148595608723037e-05 and the MSE for SGD is 3.5368255527927726e-05.

If we comapre the results, there is a very little differnece between Normal Equation and SGD. However, taking that minimal difference into consideration, Normal Equation perform better than SGD.

Problem 2

Create a Perceptron model with an optimal value of hyperparameter α (learning rate of SGD)

In [19]:
```
# Import necessary packages to the Jupyter notebook
# Implement a Perceptron algorithm with an optimal value of learning rate
import pandas as pd
import numpy as np
from pandas import read_csv

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import Perceptron

# read and load the csv data file
filename = "Dataset/sonar.all-data.csv"
dataframe = read_csv (filename)

# converting the dataset to a numpy array
array = dataframe . values

# separate array into input and output components
X = array [:,:-1]
Y = array [:,-1]
```

(a)

Split your data into train and test portions with 'test size = 0.3' and 'random state = 3' . Define your learning model to be Perceptron. Use RepeatedStratifiedKFold with 'n splits=10', 'n repeats=5', and 'random state=1' as your model evaluation method.

In [20]:
```
model = Perceptron()
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=3
model.fit(X_train, Y_train)
```

Out[20]:
```
Perceptron()
```

In [21]:
```
# define model
model = Perceptron()
# define model evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=5, random_state=1)
```

(b)

Use GridSearchCV to perform a gird search on the parameter of Perceptron algorithm (learning rate α in SGD), consider values for α as [0.0001, 0.001, 0.01, 0.1]. For your GridSearch, use data only from your training sets (X-train, Y_train).

In [22]:
```
# define grid
grid = dict()
grid['alpha'] = [0.0001, 0.001, 0.01, 0.1]
```

In [23]:
```
# define search
search = GridSearchCV(model, grid, scoring='accuracy', cv=cv, n_jobs=-1)
# perform the search
results = search.fit(X_train, Y_train)
```

(c)

Report the best score and the best value of the parameter in your search.

In [24]:
```
# summarize
print('Mean Accuracy: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)
```

```
Mean Accuracy: 0.664
Config: {'alpha': 0.0001}
```

(d)

Create a Perceptron model which takes as an argument the best value of parameter you found in the previous step, and use this model to make predictions on your test set; Report the accuracy.

In [25]:
```
clf = Perceptron(alpha=0.0001)
results = clf.fit(X_train, Y_train)
results.score(X_train, Y_train)
```

Out[25]:
```
0.7172413793103448
```

If you see in part c, we used the attribute best*score* on GridSearchCV to find the mean accuracy or the mean cross-validated score of the best estimator, which was about 0.664. We also used the best*params* attribute to find the parameter setting that gave the best results on the hold out data, which happened to be 0.0001 in this example. Then, when we used the score() method on the Perceptron model. The score() method returns the mean accuracy on the given test data and labels. We got a mean accuracy of about 0.712, which is higher than the mean accuracy of the GridSearch.

Problem 3: Create a KNN model with an optimal value of hyperparameter K (the number of nearest neighbors)

In [26]:
```
# import necessary packages to the Jupyter notebook
# Create a KNN model with the best parameter K
import pandas as pd
import numpy as np
import matplotlib . pyplot as plt

from pandas import read_csv
from sklearn . model_selection import train_test_split
from sklearn . metrics import accuracy_score
from sklearn . neighbors import KNeighborsClassifier

# read and load the csv data file
filename = "Dataset/sonar.all-data.csv"
dataframe = read_csv (filename)

# converting the dataset to a numpy array
array = dataframe . values

# separate array into input and output components
X = array [:,:-1]
Y = array [:,-1]
```

(a)

Split the data into train and test sets with 'test_size = 0.3', and 'random_state = 5'. Create a KNN model with parameter 'n_neighbor' varying from 1 to 30 (see the code from Lab Session 6).

In [27]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state =
```

In [28]:
```
scores = {}
for k in range(1,30):
  knn = KNeighborsClassifier(n_neighbors=k)
  knn.fit(X_train, Y_train)
  y_pred = knn.predict(X_test)
  scores[k] = accuracy_score(y_pred, Y_test)
```

In [29]:
```
scores
```

Out[29]:
```
{1: 0.7777777777777778,
 2: 0.7142857142857143,
 3: 0.7301587301587301,
 4: 0.7142857142857143,
 5: 0.746031746031746,
```

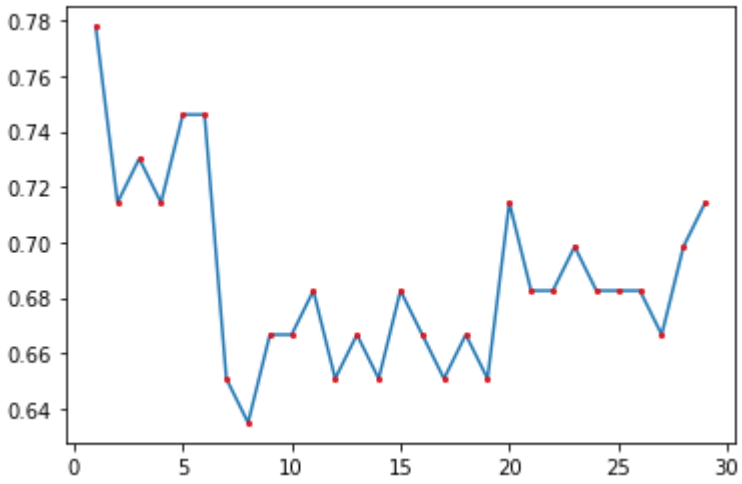
```
 6: 0.746031746031746,
 7: 0.6507936507936508,
 8: 0.6349206349206349,
 9: 0.6666666666666666,
 10: 0.6666666666666666,
 11: 0.6825396825396826,
 12: 0.6507936507936508,
 13: 0.6666666666666666,
 14: 0.6507936507936508,
 15: 0.6825396825396826,
 16: 0.6666666666666666,
 17: 0.6507936507936508,
 18: 0.6666666666666666,
 19: 0.6507936507936508,
 20: 0.7142857142857143,
 21: 0.6825396825396826,
 22: 0.6825396825396826,
 23: 0.6984126984126984,
 24: 0.6825396825396826,
 25: 0.6825396825396826,
 26: 0.6825396825396826,
 27: 0.6666666666666666,
 28: 0.6984126984126984,
 29: 0.7142857142857143}
```

(b)

Plot the accuracy of the KNN model in terms of the number of nearest neighbor k varying from 1 to 30. Choose and report the best value for k.

In [30]:
```
plt.plot(list(scores.keys()),list(scores.values()),marker="o", markersize=2, markeredge
```

Out[30]:
```
[<matplotlib.lines.Line2D at 0x1aad288b5b0>]
```



After running the KNeighborsClassifier with diffrent number of neighbors (1 -30), we can see that the best value of k is 1. If you see the list of accuracy classification scores and the line graph, you can see that k=1 has the highest accuracy. Then 5 and 6 would be the next best values for k.

(c)

Create a new KNN model with the best values of nearest neighbors that you found in previous step, and perform prediction on your test set. Report the accuracy of the model.

In [31]:
```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, Y_train)
knn.score(X_train, Y_train)
```

Out[31]:
```
1.0
```

In [32]:
```
y_pred = knn.predict(X_test)
score = accuracy_score(y_pred, Y_test)
score
```

Out[32]:
```
0.7777777777777778
```

In [33]:
```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, Y_train)
knn.score(X_train, Y_train)
```

Out[33]:
```
0.8344827586206897
```

In [34]:
```
y_pred = knn.predict(X_test)
score = accuracy_score(y_pred, Y_test)
score
```

Out[34]:
```
0.746031746031746
```

In [35]:
```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, Y_train)
knn.score(X_train, Y_train)
```

Out[35]:
```
0.8275862068965517
```

In [36]:
```
y_pred = knn.predict(X_test)
score = accuracy_score(y_pred, Y_test)
score
```

Out[36]:
```
0.746031746031746
```

When I take the KNeighborsClassifier and use the score method, I get the mean accuracy of the data. However, I found this unreliable, because it is checking if it is an exact match of X_train to get high accuracy. That is why the accuracy_score(y_pred, Y_test) was a better test. I choosed the 3 highest accuracy's (k = 1, 5, & 6). k = 1 was the highest and 5/6 tied for 2nd.

In [ ]:
```

```