DSCC 465 HW 02

Uzair Tahamid Siam

[Math Processing Error]

Importing

In [1]:

```
import numpy as np
import sklearn.datasets
import pandas as pd
from numpy.typing import ArrayLike
import matplotlib.pyplot as plt
```

Data Preprocessing

In [2]:

```
# importing california housing data from sklearn

dataset = sklearn.datasets.fetch_california_housing(as_frame=True).frame
```

In [3]:

```
x = dataset.iloc[:, :-1]
y = dataset.iloc[:, -1]
```

In [4]:

```
# Dividing into training and test sets

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=265)
```

In [5]:

```
# Normalizing the data i.e. feature scaling

from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.transform(x_test)
```

Q3

Gradient:

$$\nabla_\theta J = (2/m)\, X^T (X\theta - \vec{y})$$

$X =$ feature matrix

$\theta =$ weights

$m =$ number of data objects

$X\theta =$ prediction

$\vec{y} =$ targets

MSE function:

$$J(\vec{\theta}) = (1/2m)(X\theta - \vec{y})^2$$

In [6]:

```
def cost(theta: ArrayLike, X: ArrayLike, y: ArrayLike) -> int:
    '''
    Parameters
    ----------
    theta: numpy.typing.ArrayLike
        The weights that are to be optimized

    X: numpy.typing.ArrayLike
        Numpy Array of features

    y: numpy.typing.ArrayLike
        Numpy Array of targets

    Returns
    ----------
    mse: int
        Returns the mean-square-error between the predicted and actual target values
    '''
    m = len(y) # number of data
    y_pred = np.dot(X, theta) # calculating X*theta
    mse = 1/(2*m)*((y_pred - y)**2)   # calculating the error function

    return mse

def gradient(theta:ArrayLike, X:ArrayLike, y:ArrayLike)->ArrayLike:
    '''
    '''
    m = len(y) # number of data
    err = np.dot(X, theta) - y # error/residue term
    grad = 2/m * np.dot(X.T, err) # finding the gradient using the vector form

    return grad

def gradient_descent(X: ArrayLike, y:ArrayLike, number_of_steps: int = 1000, learning_rate
    '''
    Parameters
    ----------
    theta: numpy.typing.ArrayLike
        The weights that are to be optimized

    X: numpy.typing.ArrayLike
        Numpy Array of features

    y: numpy.typing.ArrayLike
        Numpy Array of targets

    number_of_steps: int
        Number of iterations for the gradient descent algorithm

    learning_rate: int
        Often called alpha. Used as a multiplier for the step-size

    Returns
    ----------
    theta: int
        Returns the optimized values of the weights initially given as a parameter
```

```
    '''

    y_reshaped = np.reshape(y, (len(y), 1)) # reshaping for calculation
    new_X = np.c_[np.ones((len(X), 1)), X] # appending a column of 1 to X for intercept
    theta = np.random.randn(len(X[0])+1, 1) # initializing the weights and also the interc
    m = len(y) # number of data
    cost_lst = [] # saving cost for plotting

    for i in range(number_of_steps): # initializing the iterations
        gradients = gradient(theta, new_X, y_reshaped) # finding the gradient using the in
        theta = theta - learning_rate * gradients # finding the new weights
        y_pred = np.dot(new_X, theta) # finding the new prediction value
        if plot:
            cost_value = cost(theta, new_X, y_reshaped) # finding the new cost if plotting
            # calculating the total cost
            total = 0
            for i in range(len(y)):
                total += cost_value[i][0]
                #Calculate the cost function for each iteration

                cost_lst.append(total)
    if plot:
        plt.plot(np.arange(1,number_of_steps),cost_lst[1:], color = 'red')
        plt.title('MSE Plot')
        plt.xlabel('Number of iterations')
        plt.ylabel('Cost')

    intercept_ = theta[0][0]

    return {'intercept':intercept_, 'weights':theta[1:].flatten()}
```

In [7]:

```
# If you wish to plot the cost please set plot as True in the argument

gradient_descent(x_train, y_train.values, number_of_steps= 1000, learning_rate= 0.01)
```

Out[7]:

```
{'intercept': 2.0660607482793427,
 'weights': array([ 0.81805816,  0.11062097, -0.23073109,  0.29083034, -0.0100911 ,
        -0.03901203, -0.95539921, -0.9218917 ])}
```

Interpretation:

The intercept represents the median value of the house if every feature had a value of 0.

If we are to assume that the data follows rules of linearity, then from our analysis we can make the following claims:

> 1) The median income has the highest positive trend with the median value of a house. This makes complete sense as if you make more money you are probably living in a high income area which also results in higher prices in housing.
>
> 2) The lattitude and longitudes have the most negative trend with the value. I am not sure what to make of this observation given my lack of knowledge in both the housing industry as well as what living at different longi/lattitude means.
>
> 3) The population and average occupation seem to have the lowest correlation to the house pricing. Kind of surprising to me as you would expect some sort of correlation between

> population in a region and how expensive it might be to get land/build housing ia highly populated region. But at least it does show slightly negative impact as expected.
>
> 4) The higher the age of the house the more expensive the house appears to be. This is truly surprising to me as I would expect newer houses to be more expensive.
>
> 5) The more *BEDROOMS* the higher the price but the more *TOTAL* rooms the lower the price. I suppose this kind of makes sense. If you have more bedrooms you do expect the house to be bigger but also if there's too many rooms that are not bedrooms people might not want to buy the house as a result decreasing the price.

Q4

In [11]:

```
# Using sklearn's Stochastic Gradient Descent Regression algorithm

from sklearn.linear_model import SGDRegressor

reg = SGDRegressor(loss='squared_error',random_state=265, max_iter=1000, alpha=0.01)

# training the data
reg.fit(x_train, y_train)

# getting the optimized weights
weights = reg.coef_
```

In [12]:

```
print(f"weights:\n{weights}")
print(f"intercept:\n{reg.intercept_}")
```

```
weights:
[ 0.8415846   0.12309233 -0.2766104   0.28230879 -0.01295436 -0.03037704
 -0.80087347 -0.76047679]
intercept:
[2.06235155]
```

Interpretation:

The behavior is very comparable to that seen in our analysis done in Q3. Of course this was to be expected as we are using similar techniques. The values for the weights do not completely align (i.e they are not 1:1) but the trends are all the same and so are the relative magnitudes of the weights. The slight decrepencies are likely to be a result of a suboptimized self-made gradient descent function compared to the more professionally developed sklearn module's regressor. But nonetheless given enough steps in our own gradient descent function, the two do get very close. (Feel free to run it for number_of_steps = 5000 and check for yourself)

Q5

$$cov(X) = E[(X - E[X])(X - E[X])^T]$$

In [22]:

```
def cov_like_np(X: ArrayLike) -> ArrayLike:
    '''
```

```
    Parameters
    ----------
    X: numpy.typing.ArrayLike
        Numpy Array for covariance calculation

    Returns
    ----------
    cov: numpy.typing.ArrayLike
        Numpy Array representing the covariance of the input matrix X
    '''
    X = np.array(X)
    EX = np.mean(X, axis=1).reshape(len(X), 1) # finding mean of each row in X and reshap
    diff = np.subtract(X, EX) # finding X - EX
    prod = np.dot(diff, diff.T)
    cov = prod/(len(X[0])-1)
    return cov
```

In [23]:

```
def cov_like_pd(X: ArrayLike, asFrame:bool = True) -> ArrayLike:
    '''
    Parameters
    ----------
    X: numpy.typing.ArrayLike
        Numpy Array for covariance calculation

    asFrame: bool
        User input for return type as a dataframe or numpy array

    Returns
    ----------
    cov: numpy.typing.ArrayLike
        Numpy Array representing the covariance of the input matrix X
    '''
    I = np.ones((len(X), len(X))) # identity matrix
    EX = np.dot(I, X)/(X.shape[0]) # finding the expectation of X by multiplying by I and
    diff = np.subtract(X,EX) # finding X - EX
    cov = np.dot(diff.T, diff)/(X.shape[0]-1)
    return pd.DataFrame.from_records(cov) if asFrame else cov
```

Pandas Like Covariance

In [24]:

```
cov_like_pd(x)
```

Out[24]:

	0	1	2	3	4	5	6	7
0	3.609323	-2.846140	1.536568	-0.055858	1.040098e+01	0.370289	-0.323860	-0.057765
1	-2.846140	158.396260	-4.772882	-0.463718	-4.222271e+03	1.724298	0.300346	-2.728244
2	1.536568	-4.772882	6.121533	0.993868	-2.023337e+02	-0.124689	0.562235	-0.136518
3	-0.055858	-0.463718	0.993868	0.224592	-3.552723e+01	-0.030424	0.070575	0.012670
4	10.400979	-4222.270582	-202.333712	-35.527225	1.282470e+06	821.712002	-263.137814	226.377839
5	0.370289	1.724298	-0.124689	-0.030424	8.217120e+02	107.870026	0.052492	0.051519
6	-0.323860	0.300346	0.562235	0.070575	-2.631378e+02	0.052492	4.562293	-3.957054
7	-0.057765	-2.728244	-0.136518	0.012670	2.263778e+02	0.051519	-3.957054	4.014139

In [25]:

```
pd.DataFrame.cov(pd.DataFrame.from_records(x))
```

Out[25]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Lo
MedInc	3.609323	-2.846140	1.536568	-0.055858	1.040098e+01	0.370289	-0.323860	-0.
HouseAge	-2.846140	158.396260	-4.772882	-0.463718	-4.222271e+03	1.724298	0.300346	-2.
AveRooms	1.536568	-4.772882	6.121533	0.993868	-2.023337e+02	-0.124689	0.562235	-0
AveBedrms	-0.055858	-0.463718	0.993868	0.224592	-3.552723e+01	-0.030424	0.070575	0.
Population	10.400979	-4222.270582	-202.333712	-35.527225	1.282470e+06	821.712002	-263.137814	226.
AveOccup	0.370289	1.724298	-0.124689	-0.030424	8.217120e+02	107.870026	0.052492	0
Latitude	-0.323860	0.300346	0.562235	0.070575	-2.631378e+02	0.052492	4.562293	-3.
Longitude	-0.057765	-2.728244	-0.136518	0.012670	2.263778e+02	0.051519	-3.957054	4

Numpy Like Covariance

In [28]:

```
cov_like_np(x)
```

Out[28]:

```
array([[ 15828.51059651, 100411.06909593,  22911.52253488, ...,
         43698.7069652 ,  32878.34650356,  59154.88863585],
       [100411.06909593, 726902.76879646, 152324.56887559, ...,
        307726.22523876, 227636.95255885, 422082.4403021 ],
       [ 22911.52253488, 152324.56887559,  33722.95864622, ...,
         65602.13612302,  49048.98091119,  89243.70502233],
       ...,
       [ 43698.7069652 , 307726.22523876,  65602.13612302, ...,
        131028.86321045,  97274.06321906, 179228.804142  ],
       [ 32878.34650356, 227636.95255885,  49048.98091119, ...,
         97274.06321906,  72373.786035  , 132831.76856219],
       [ 59154.88863585, 422082.4403021 ,  89243.70502233, ...,
        179228.804142  , 132831.76856219, 245479.08714416]])
```

In [27]:

```
np.cov(x)
```

Out[27]:

```
array([[ 15828.51059651, 100411.06909593,  22911.52253488, ...,
         43698.7069652 ,  32878.34650356,  59154.88863585],
       [100411.06909593, 726902.76879646, 152324.56887559, ...,
        307726.22523876, 227636.95255885, 422082.4403021 ],
       [ 22911.52253488, 152324.56887559,  33722.95864622, ...,
         65602.13612302,  49048.98091119,  89243.70502233],
       ...,
       [ 43698.7069652 , 307726.22523876,  65602.13612302, ...,
        131028.86321045,  97274.06321906, 179228.804142  ],
       [ 32878.34650356, 227636.95255885,  49048.98091119, ...,
         97274.06321906,  72373.786035  , 132831.76856219],
       [ 59154.88863585, 422082.4403021 ,  89243.70502233, ...,
        179228.804142  , 132831.76856219, 245479.08714416]])
```