PROGRAMMING ASSIGNMENT

Submitted by:	**Gavali Deshabhakt Nagnath**
Roll No.:	**202CD005**
Department:	**Mathematics and Computational Sciences**
Specialization:	**Computational and Data Science**

Subject:	**Big Data Analytics**
Instructor:	**Dr. Pushparaj Shetty**
Submitted on:	**13/05/2021**

Q1. Regression in python

A. Multivariate Linear Regression:

The linear regression models are of the form **y = a + bx**. As the given problem can have more than one indepedent variables i.e. x_i's, the equation becoms

$$\mathbf{y = a_0 + a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n}$$

The following algorithm can be used for solving these type of problems.

Algorithm:

1. Import/define database
2. Take array of independent variables and dependent variable as input
3. create two variables corresponding to input array of independent variables:
   Here we can use shape of input array to create 2 – variables of which the **first one will be our sample size** (i.e. rows in input array of independet variables) and second one i.e. **columns will be our number of features.**

4. Now we can use gradient descent approach to find out the multipliers/coefficients of our regression equation
   the formulas for gradient descent approach would be as follows

   i. finding deriavative with respect to unknown coefficients
      dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))

   ii. Finding deravative with respect to unknown constant
      db = (1 / n_samples) * np.sum(y_predicted - y)

   iii. update coefficients and constant
      self.weights -= self.lr * dw
      self.bias -= self.lr * db

   *here **lr is learing rate** which can be defined in program.

5. At the end of 4th step we will have our coefficients and constant and using this we can simply predicted values as follows

   y_approximated = np.dot(X, self.weights) + self.bias

Program:

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

class LinearRegression:
    def __init__(self, learning_rate=0.001, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
```

```
def fit(self, X, y):
        n_samples, n_features = X.shape

        # init parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        # gradient descent
        for _ in range(self.n_iters):
                y_predicted = np.dot(X, self.weights) + self.bias
                # compute gradients
                dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
                db = (1 / n_samples) * np.sum(y_predicted - y)

                # update parameters
                self.weights -= self.lr * dw
                self.bias -= self.lr * db

def predict(self, X):
        y_approximated = np.dot(X, self.weights) + self.bias
        return y_approximated
        def score(self,y,y_pred):
        return np.mean((1-abs(y-y_pred)))

def visualize(self,x,y,y_pred):
        plt.scatter(x,y,color='r')
        plt.plot(x,y_pred,color='g')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Multivariate Linear Regression')
        plt.show()

if __name__ == '__main__':
        # Input Section
        df = pd.DataFrame({
                'x1' : [ 1.1, 1.3, 1.5, 2. , 2.2, 2.9, 3. , 3.2, 3.2, 3.7, 3.9, 4. , 4. , 4.1, 4.5, 4.9,
                        5.1, 5.3, 5.9, 6. , 6.8, 7.1, 7.9, 8.2, 8.7, 9. , 9.5, 9.6, 10.3, 10.5],
                'x2' : [ 2.1, 2.3, 1.4, 2.5 , 2.9, 2., 3. , 3.2, 3.2, 3.7, 3.1, 4.2 , 4.6 , 4.3, 4.1, 4.7,
                        5.2, 5.4, 5.0, 6. , 6.6, 7.1, 7.9, 8.2, 5.7, 9. , 9.5, 9.6, 10.3, 10.5],
                'y' : [ 1.1, 1.3, 1.5, 2. , 2.2, 2.9, 3. , 3.2, 3.2, 3.7, 3.9, 4. , 4. , 4.1, 4.5, 4.9, 5.1,
                        5.3, 5.9, 6. , 6.8, 7.1, 7.9, 8.2, 8.7, 9. , 9.5, 9.6, 10.3, 10.5]
                })
        # Linear Regression
        le = LinearRegression()
        x = df.iloc[:,:2].values
        y = df.y.values
        le.fit(x,y)

        y_pred = le.predict(x)
        for i in range(len(y)):
                print(f'{y[i]} {y_pred[i]:10.5f}')

        accuracy = le.score(y,y_pred)
        print('\nAccuracy = ',accuracy)

        # we cannot visualize resutls if number of independent variables are more than 1
        if (len(x))==1:
                le.visualize(x,y,y_pred)
```

Output:

1.1	1.53477
1.3	1.73530
1.5	1.50639
2.0	2.24194
2.2	2.52054
2.9	2.59776
3.0	3.04937
3.2	3.24990
3.2	3.24990
3.7	3.75121
3.9	3.63942
4.0	4.13008
4.0	4.28623
4.1	4.23034
4.5	4.39715
4.9	4.87628
5.1	5.19393
5.3	5.39445
5.9	5.60564
6.0	6.05725
6.8	6.78127
7.1	7.16014
7.9	7.96224
8.2	8.26303
8.7	7.59317
9.0	9.06513
9.5	9.56644
9.6	9.66671
10.3	10.36855
10.5	10.56907

Accuracy = 0.8312599035317786

B. Polynomial regression:

The equation of polynomial regression is as follows

$$\mathbf{y = a_0 + a_1 * x + a_2 * x^2 + ... + a_n * x^n}$$

This type of regression is useful when data is not linearly dependent. Here we get a curve fitting between our data points. Our ultimate goal in any regression problem is to minimize least squared error.

Let us consider a quadratic regression problem or polynomial regression problem with degree 2. For this problem our equation will be

$$\mathbf{y = a_0 + a_1 * x + a_2 * x^2}$$

Algorithm:
So, our least square error will be,

$$\mathbf{E = (1/N)*(\sum(y - y\_predict)^2)}$$
$$\mathbf{E = (1/N)*(\sum(y - (a_0 + a_1*x + a_2*x^2))^2)}$$

where,

N = Sample size/ number of data points
y = actual y (from database)
y_predict = y's predicted using computed coefficients and constant

Now we to get normal equations we'll differenting above equation with respect to unkown variables one by one.

differentiating w.r.t a_0:

$$dE/da_0 = -(2/N)*(\sum(y - (a_0 + a_1*x + a_2*x^2)))$$

simplifying above equation,

$$\sum y = n*a_0 + a_1*\sum x + a_2*\sum x^2 \quad \text{--- (1)}$$

similiarly by differenting with respect to a_1 and a_2 respectively we get,

$$\sum x*y = a_0*\sum x + a_1*\sum x^2 + a_2*\sum x^3 \quad \text{---- (2)}$$

$$\sum x^2*y = a_0*\sum x^2 + a_1*\sum x^3 + a_2*\sum x^4 \quad \text{---- (3)}$$

now we have 3 equationg with 3 unknows (i.e. a_0 , a_1 , a_2). We can solve above three equations using any method such as gauss elimiation or cramers rule, etc.
Writing aboe equation in matrix form,
Let A = Matrix of summations on RHS of above equation
b = Matrix of terms of LHS of above equation
X = Matrix of Unknows i.e. a_0 , a_1 , a_2

$$A = \begin{bmatrix} n & \sum x & \sum x^2 \\ \sum x & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{bmatrix} \qquad X = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \qquad b = \begin{bmatrix} \sum y \\ \sum x*y \\ \sum x^2*y \end{bmatrix}$$

Then,

$$X = A^{-1}*b$$

Thus, we'll get our unknow coefficients and constant.
We can generalize this for a n-degree polynomial like, $\mathbf{y = a_0 + a_1*x + a_2*x^2 + ... + a_n*x^n}$.
The matrices for n-degree polynomial would be,

$$A = \begin{bmatrix} n & \sum x & \dots & \sum x^{n} \\ \sum x & \sum x^{2} & \dots & \sum x^{(n+1)} \\ . & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ \sum x^{n} & \sum x^{(n+1)} & \dots & \sum x^{2n} \end{bmatrix} \qquad X = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ a_n \end{bmatrix} \qquad b = \begin{bmatrix} \sum y \\ \sum x*y \\ \sum x^{2}*y \\ . \\ . \\ . \\ \sum x^{n}*y \end{bmatrix}$$

Program:

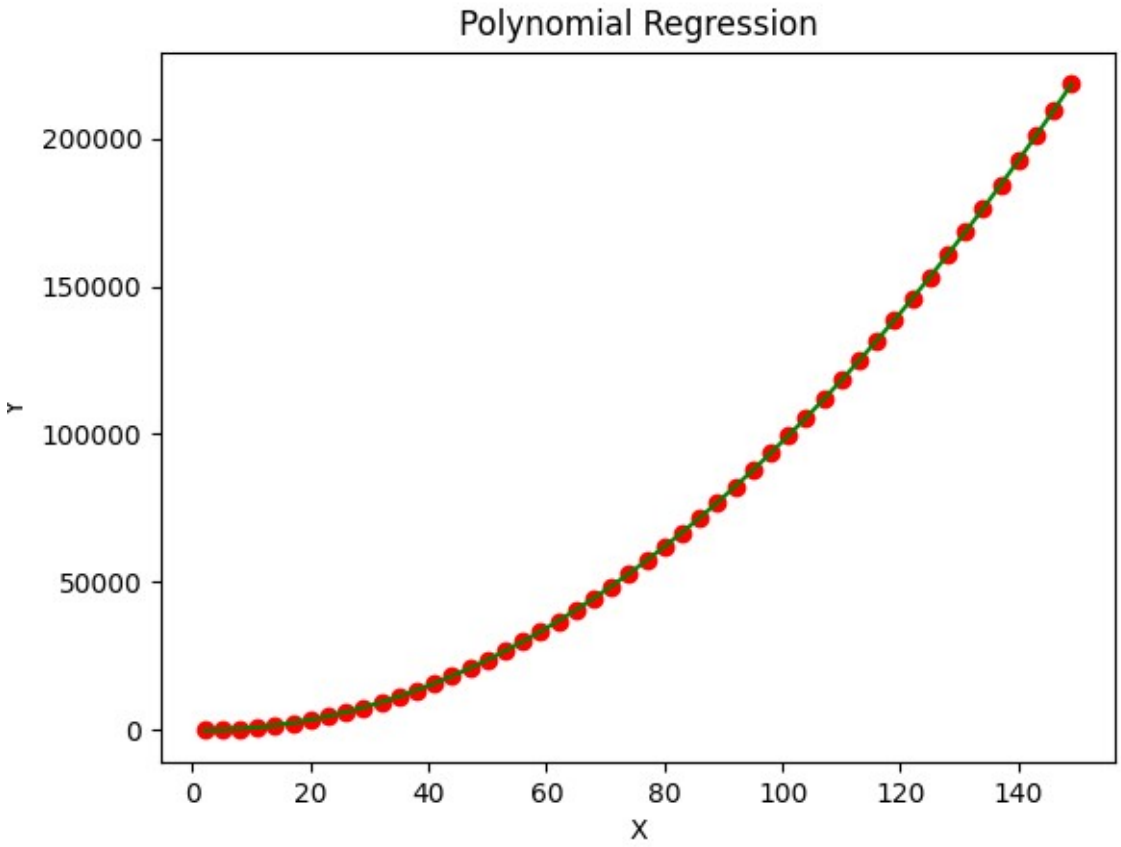
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

class PolynomialRegression:
        def __init__(self, learning_rate=0.001, n_iters=1000,degree=2):
                self.lr = learning_rate
                self.n_iters = n_iters
                self.coefficients = None
                self.degree = degree

        def fit(self, x, y):

                X_t = []
                y_t = []
                n = len(x)

                X_t.append(n)
                i = 1
                while(True):
                        X_t.append(sum((x**i)))
                        if(i/2 == self.degree):
                                break
                        i += 1
                y_t.append(sum(y))
                for i in range(1,self.degree+1):
                        y_t.append(sum((x**i)*y))
                A = np.zeros((self.degree+1,self.degree+1),dtype=float)
                b = np.zeros((self.degree+1,1),dtype=float)

                for i in range(self.degree+1):
                        k = i
                        for j in range(self.degree+1):
                                A[i,j] = X_t[k]
                                k +=1
                for i in range(self.degree+1):
                        b[i,0] = y_t[i]

                self.coefficients = np.matmul(np.linalg.inv(A),b)

        def predict(self, X):
                y_approximated = np.zeros(len(X))
```

```
            for i in range(0,self.degree+1):
                for j in range(len(X)):
                    y_approximated[j] += self.coefficients[i]*(X[j]**i)
            return y_approximated
        def score(self,y,y_pred):
            accuracy = 0
            for i in range(len(y)):
                accuracy += abs(1-(y[i][0]-y_pred[i]))
            return accuracy/len(y)
        def visualize(self,x,y,y_pred):
            plt.scatter(x,y,color='r')
            plt.plot(x,y_pred,color='g')
            plt.xlabel('X')
            plt.ylabel('Y')
            plt.title('Polynomial Regression')
            plt.show()


if __name__ == '__main__':

        # Input Section
        # Polynomial Regression

        df = pd.DataFrame({
            'x' : [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59,
                62, 65, 68, 71, 74, 77, 80, 83, 86, 89, 92, 95, 98, 101, 104, 107, 110,
                113, 116, 119, 122, 125, 128, 131, 134, 137, 140, 143, 146, 149],
            'y' : [5, 140, 455, 950, 1625, 2480, 3515, 4730, 6125, 7700, 9455, 11390,
                13505, 15800, 18275, 20930, 23765, 26780, 29975, 33350, 36905,
                40640, 44555, 48650, 52925, 57380, 62015, 66830, 71825, 77000,
                82355, 87890, 93605, 99500, 105575, 111830, 118265, 124880,
                131675, 138650, 145805, 153140, 160655, 168350, 176225, 184280,
                192515, 200930, 209525, 218300]
            })

        le = PolynomialRegression()

        x = df.x.values
        x = x.reshape(len(x),1)
        y = df.y.values
        y = y.reshape(len(y),1)

        le.fit(x,y)
        y_pred = le.predict(x)

        accuracy = le.score(y,y_pred)
        print('\nAccuracy = ',accuracy)

        le.visualize(x,y,y_pred)
```

Output:

5	5.00000
140	140.00000
455	455.00000
950	950.00000
1625	1625.00000
2480	2480.00000
3515	3515.00000
4730	4730.00000
6125	6125.00000
7700	7700.00000
9455	9455.00000
11390	11390.00000
13505	13505.00000
15800	15800.00000
18275	18275.00000
20930	20930.00000
23765	23765.00000
26780	26780.00000
29975	29975.00000
33350	33350.00000
36905	36905.00000
40640	40640.00000
44555	44555.00000
48650	48650.00000
52925	52925.00000
57380	57380.00000
62015	62015.00000
66830	66830.00000
71825	71825.00000
77000	77000.00000
82355	82355.00000
87890	87890.00000
93605	93605.00000
99500	99500.00000
105575	105575.00000
111830	111830.00000
118265	118265.00000
124880	124880.00000
131675	131675.00000
138650	138650.00000
145805	145805.00000
153140	153140.00000
160655	160655.00000
168350	168350.00000
176225	176225.00000
184280	184280.00000
192515	192515.00000
200930	200930.00000
209525	209525.00000
218300	218300.00000

Accuracy = 0.9999999999345164



Here the curve perfectly fits with actual data because I'd generated data using libre excel. This is not always the case. In most scenario we'll have some points off from the curve or in worst case there may not be any point on the curve. The main thing to note is that we are not drawing curve through given points, but we are drawing a curve which **best fits** given data points **i.e. minimizing least square error.**

2. K Means Clustering in python

Introduction:

Given a set of observations (x1, x2, ..., xn), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k (≤ n) sets S = {S1, S2, ..., Sk} so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

where μi is the mean of points in Si. This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2$$

The equivalence can be deduced from identity,

$$\sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \sum_{\mathbf{x} \neq \mathbf{y} \in S_i} (\mathbf{x} - \boldsymbol{\mu}_i)^T (\boldsymbol{\mu}_i - \mathbf{y})$$

Because the total variance is constant, this is equivalent to maximizing the sum of squared deviations between points in different clusters (between-cluster sum of squares, BCSS), which follows from the law of total variance.

Algorithm:

The most common algorithm uses an iterative refinement technique. Due to its ubiquity, it is often called **"the k-means algorithm"**; it is also referred to as **Lloyd's algorithm.** It is sometimes also referred to as **"naive k-means"**, because there exist much faster alternatives.

Given an initial set of k means $m_1^{(1)},...,m_k^{(1)}$ (see below), the algorithm proceeds by alternating between two steps:

Assignment step:

Assign each observation to the cluster with the nearest mean: that with the least squared Euclidean distance. (Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means.)

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\},$$

where each x_p is assigned to exactly one $S_{(t)}$, even if it could be assigned to two or more of them.

Update step:

Recalculate means (centroids) for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

The algorithm has converged when the assignments no longer change. The algorithm is not guaranteed to find the optimum.
The algorithm is often presented as assigning objects to the nearest cluster by distance. Using a different distance function other than (squared) Euclidean distance may prevent the algorithm from converging. Various modifications of k-means such as spherical k-means and k-medoids have been proposed to allow using other distance measures. Initialization of centroids is mostly done using random points.

Program:

```
import numpy as np
import pandas as pd
from matplotlib import cm
import matplotlib.pyplot as plt
import copy

class KMeansClustering:
    colormap = {}
    centroids = {}
    df = pd.DataFrame()
    labels = {}
    number_of_clusters = None
    lower_bound = None
    upper_bound = None

    # Constructor
    def __init__(self,K,dataframe,seed=None):
        if(seed!=None):
            np.random.seed(seed)

        self.number_of_clusters = K
        self.df = dataframe
        self.df['color'] = 'k'

        # Creating Color map using Set1 colormap format
        for i in range(self.number_of_clusters):
            color = cm.Set1(np.random.randint(0,10))
            self.colormap[i+1] = color
            self.labels[color] = 'Cluster ' + str(i+1)

        # offseting lower_bound and upper_bound by offset in order to increase range from fin
ding random centroids
        offset = 20

        # finding lower_bound and upper_bound from database
        self.lower_bound = min(self.df.iloc[:,0].min(),self.df.iloc[:,1].min()) - offset
        self.upper_bound = max(self.df.iloc[:,0].max(),self.df.iloc[:,1].max()) + offset
```

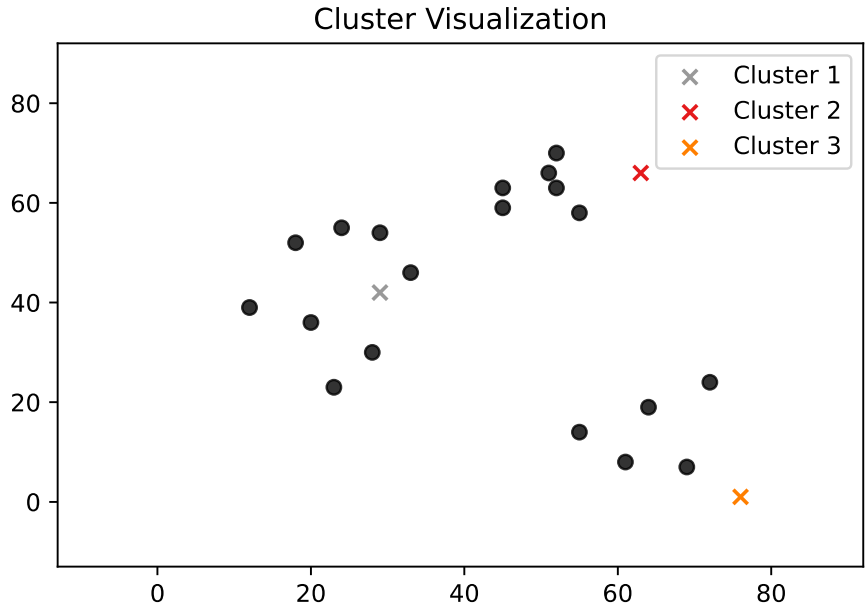
```
        # Generating random centroids
        for i in range(self.number_of_clusters):
            x = np.random.randint(self.lower_bound, self.upper_bound)
            y = np.random.randint(self.lower_bound, self.upper_bound)
            self.centroids[i+1] = [x,y]


    # Function for assigning clusters/centroids
    def assigning_clusters(self):
        for i in self.centroids.keys():

            # Finding distance of points from ith centroid
            self.df[f'distance_from_{i}'] = (np.sqrt(
                                    ((self.df.iloc[:,0] - self.centroids[i][0]) ** 2) +
                                    ((self.df.iloc[:,1] - self.centroids[i][1]) ** 2))
                                    )
        # Assigning Centroids to clusters
        centroid_distance_cols = [f'distance_from_{i}' for i in self.centroids.keys()]
        self.df['closest'] = self.df.loc[:, centroid_distance_cols].idxmin(axis=1).apply(lambda x:
int(x[-1]))
        self.df['color'] = self.df['closest'].map(lambda x: self.colormap[x])
        return self.df

    # Function for updating centroids
    def updatingCentroids(self):
        old_centroids = copy.deepcopy(self.centroids)

        for i in self.centroids.keys():
            self.centroids[i][0] = np.mean(self.df[self.df['closest'] == i].iloc[:,0])
            self.centroids[i][1] = np.mean(self.df[self.df['closest'] == i].iloc[:,1])

        return old_centroids,self.centroids

    # function for visualizing cluster
    def visualize(self):
        plt.scatter(self.df.iloc[:,0], self.df.iloc[:,1], color=self.df['color'], alpha=0.8)
        for i in self.centroids.keys():
            plt.scatter(*self.centroids[i], color=self.colormap[i], label = self.labels[self.colormap
[i]], marker='x')
        plt.xlim(self.lower_bound, self.upper_bound)
        plt.ylim(self.lower_bound, self.upper_bound)
        plt.legend()
        plt.title('Cluster Visualization')
        plt.show()

    # Plotting updated cluster
    def plot_updation(self,old_centroids,iteration):
        plt.figure(figsize=(5, 5))
        ax = plt.axes()

        # Plotting data points
        plt.scatter(self.df.iloc[:,0], self.df.iloc[:,1], color=self.df['color'], alpha=0.8)
```

```
        # Plotting centroids
        for i in self.centroids.keys():
            plt.scatter(*self.centroids[i], color=self.colormap[i],label=self.labels[self.colormap[i]
], marker='x')

        plt.xlim(self.lower_bound, self.upper_bound)
        plt.ylim(self.lower_bound, self.upper_bound)

        # Plotting arrow in direction of centroid movement
        for i in old_centroids.keys():
            old_x = old_centroids[i][0]
            old_y = old_centroids[i][1]

            scaling_difference_between_distance_of_old_and_new_centroids = 0.75

            dx = (self.centroids[i][0] - old_centroids[i][0]) * scaling_difference_between_distanc
e_of_old_and_new_centroids
            dy = (self.centroids[i][1] - old_centroids[i][1]) * scaling_difference_between_distanc
e_of_old_and_new_centroids
            ax.arrow(old_x, old_y, dx, dy, head_width=2, head_length=3, fc=self.colormap[i], e
c=self.colormap[i])
        plt.title(f'Iteration = {iteration}')
        plt.legend()
        plt.show()

    # Main Function For execution of Kmeans clustering
    def kmc_predict(self,plot=False):
        if(plot):
            self.visualize()
        self.df = self.assigning_clusters()
        iteration = 0
        while True:
            iteration += 1
            closest_centroids = self.df['closest'].copy(deep=True)
            self.df = self.assigning_clusters()

            old_centroids,self.centroids = self.updatingCentroids()
            if(plot):
                print(f'Iteration = {iteration}')
                self.plot_updation(old_centroids,iteration)

            if closest_centroids.equals(self.df['closest']):
                if plot:
                    self.visualize()
                return self.df['closest'].values

# Input Section
df = pd.DataFrame({
    'x': [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 23, 55, 61, 64, 69, 72],
    'y': [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63, 58, 23, 14, 8, 19, 7, 24]
})
```

```
seed = 200      # for generating random numbers
number_of_clusters = 3
show_graphs = True

knn = KMeansClustering(number_of_clusters,df,seed)
predicted_clusters = knn.kmc_predict(show_graphs)
print('Clusters ',predicted_clusters)
```

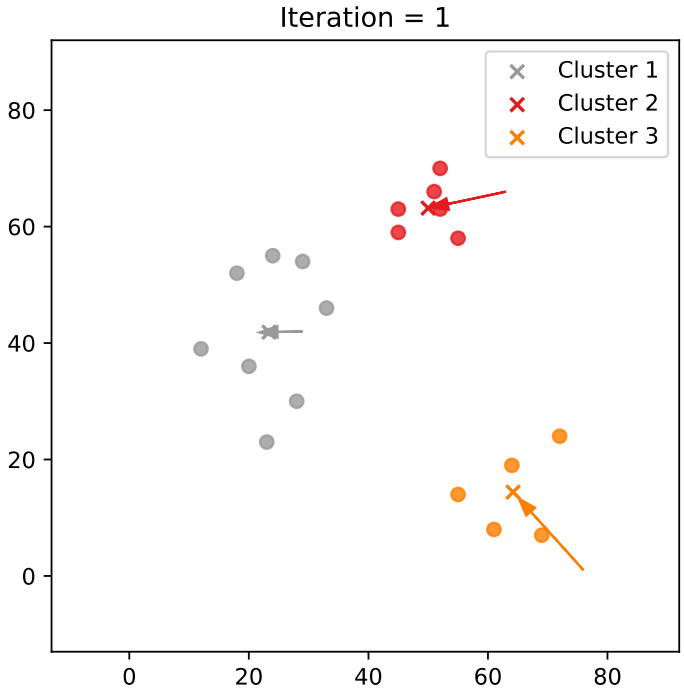
Output:

Visualizing Cluster Before Assignment

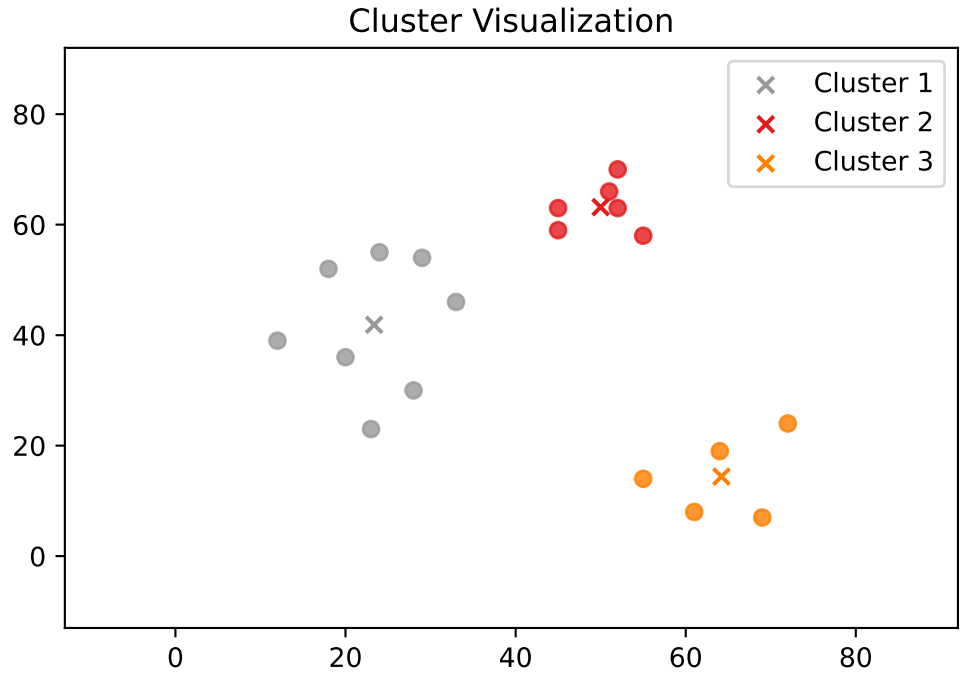


Iteration = 1

Arrow shows direction of cluster movement



#Clusters after applying clustering algoritm (we can see proper separation of 3 clusters from our database)



#This array shows clusters assigned to each row

Clusters [1 1 1 1 1 1 1 2 2 2 2 2 2 1 3 3 3 3 3]