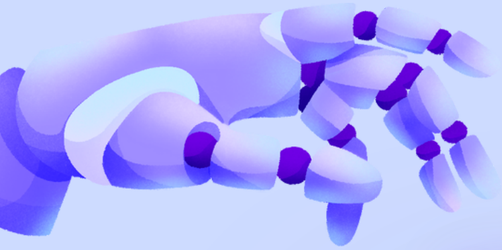
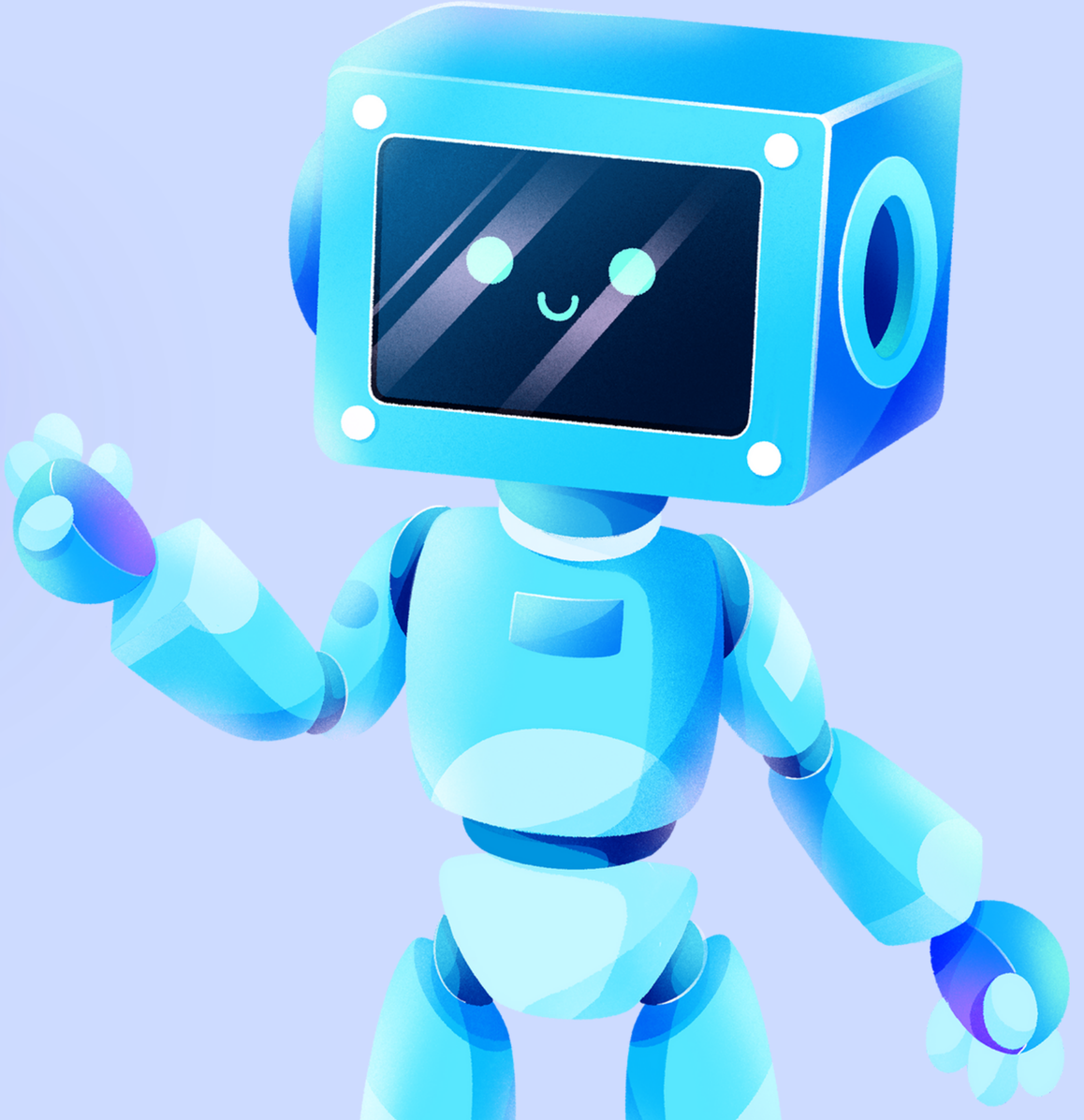
Machine Learning

PROJECT

Arjwan Adel - 2110826
Ruba Alsulami - 2110618
Ramah Alharbi - 2110173
Fay Redha - 2111992
Riof Alzahrani - 2111606





ALGORITHMS

K-mean

An algorithm for unsupervised machine learning is utilized to group data points according to their similarity. Its goal is to divide a given dataset into K clusters, each of which stands for a collection of related data points. The method assigns data points to the closest centroid or representative point, iteratively, then updates the centroids in light of the new clusters.

DBSCAN

Data points are grouped according to their density using a density-based clustering method. In contrast to K-means, DBSCAN may find clusters of any shape and does not require the number of clusters to be provided beforehand. It can handle datasets with different densities by defining clusters as dense regions of data points divided by sparser regions.

GENERATE KMEAN MODELS

```
class KMeans:
    def __init__(self, n_clusters, max_iter=300, random_state=None, tol=1e-4):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.random_state = random_state
        self.tol = tol
        self.centroids = None

    def _initialize_centroids(self, X):
        if self.random_state:
            np.random.seed(self.random_state)

        centroids = [X[np.random.choice(range(X.shape[0]))]]
        for _ in range(1, self.n_clusters):
            distances = np.min(cdist(X, centroids, 'euclidean'), axis=1)
            prob = distances / np.sum(distances)
            cumulative_prob = np.cumsum(prob)
            r = np.random.rand()
            i = np.searchsorted(cumulative_prob, r)
            centroids.append(X[i])
        return np.array(centroids)
```

This is a private method (_initialize_centroids) that is responsible for initializing the centroids of the clusters. It uses the k-means++ initialization, which aims to spread out the initial centroids. It starts by randomly selecting the first centroid, and then iteratively selects the next centroid based on a probability distribution that favors points farther away from existing centroids.

```
def fit(self, X):
    self.centroids = self._initialize_centroids(X)

    for _ in range(self.max_iter):
        distances = cdist(X, self.centroids, 'euclidean')
        labels = np.argmin(distances, axis=1)

        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(self.n_clusters)])
        diff = np.linalg.norm(new_centroids - self.centroids)
        if diff < self.tol:
            break

        self.centroids = new_centroids

    return labels

def predict(self, X):
    distances = cdist(X, self.centroids, 'euclidean')
    return np.argmin(distances, axis=1)
```

The fit method performs the actual clustering. It initializes the centroids using the previously defined method, and then iteratively assigns data points to the nearest centroid, updates the centroids based on the assigned points, and repeats until convergence (when the centroids stop changing significantly) or until reaching the maximum number of iterations. It returns the cluster labels assigned to each data point.

The predict method assigns new data points to the nearest cluster based on the centroids obtained during the fitting process. It calculates the distances between data points and centroids and assigns each point to the cluster with the closest centroid. It returns the predicted cluster labels for the input data X.

GENERATE DBSCAN MODELS

```python
class DBSCAN:
    # DBSCAN (Density-Based Spatial Clustering of Applications with Noise) implementation
    # Used to classify points as core, border, or noise in a given dataset

    def __init__(self, data, eps: float, min_samples: int):
        """
        Initialize DBSCAN object with the given parameters.

        Parameters:
        - data: The input dataset, where each row represents a data point with coordinates.
        - eps: The maximum distance between two samples for one to be considered as in the neighborhood of the other.
        - min_samples: The number of samples (or total weight) in a neighborhood for a point to be considered as a core point.

        Initializes internal variables and sets up the initial state of the dataset.
        """
        self.eps = eps
        self.min_samples = min_samples
        self.data = data
        self.dataSet1 = pd.DataFrame(self.data, columns=['x', 'y'])
        self.dataSet1["state"] = -1  # -1 represents unclassified
        self.dataSet1["Clusters"] = -1  # -1 represents unassigned cluster
        self.core = []  # List to store core points
        self.border = []  # List to store border points
        self.noise = []  # List to store noise points
        self.cluster = []

    def distance_matrix(self):
        """
        Create a distance matrix for the given dataset.

        Returns:
        - distance_of_matrix: The matrix where element [i, j] represents the distance between data points i and j.
        """
        n = len(self.data)
        distance_of_matrix = np.zeros((n, n))

        # Calculate Euclidean distance between each pair of points
        for i in range(n):
            for j in range(n):
                distance_of_matrix[i, j] = np.linalg.norm(self.data[i] - self.data[j])

        return distance_of_matrix
```

```
def neighbor_points(self):
    """
    Find all neighbor points for every point based on the epsilon value and minimum points in the circle .

    Returns:
    - Neighboring_points: list contains all neighbors of each point.
    """
    distances=self.distance_matrix()
    Neighboring_points=[]

    for i in range(len(distances)):
      for j in range(len(distances[0])):

        if(distances[i][j]<=self.eps):
            Neighboring_points.append([i,j])

    return Neighboring_points


def is_in_cluster(self,cluster,corPoint):
     """
     To find the index of the cluster that is the core neighbor of the point.

     Returns:
     -If the core neighbor is part of a cluster, return the index of the cluster.
     -If not, return -1 to indicate that no cluster was found.
     """
     for i in range(len(cluster)):
      for j in range(len(cluster[i])):
        if(cluster[i][j]==corPoint):
            return i

     return -1
```

The neighbor points function: to find all neighbor points for every point based on the epsilon value and minimum points in the circle,and will returns list contains all neighbors of each point.

The is_in_cluster function: To find the index of the cluster that is the core neighbor of the point, then it will return

-If the core neighbor is part of a cluster, return the index of the cluster.

-If not, return -1 to indicate that no cluster was found.

```
def density_reachable(self,cluster,C1,C2):
    """
    Merge clusters that have a density reachable connection between them.

    Returns:
    -After merging two clusters that are density reachable, return the new clusters .
    """

    for i in range(len(cluster[C2])):

        cluster[C1].append(cluster[C2][i])

    for i in range(len(cluster[C2])):
        cluster[C2].pop()

    return cluster
```

density_reachable function: Merge clusters that have a density reachable connection between them.And will return :

-After merging two clusters that are density reachable, return the new clusters .

```python
def predict(self,stateData,core,border):
        """
        Predict the clusters of the dataset based on the state of each point and its neighbors.

        Returns:
        -the final clusters of the dataset.
        """
        self.core=core
        self.border=border
        StateDate=stateData
        neighboring_points=self.neighbor_points()

        #Creating clusters for core points.
        for i in range(len(self.dataSet1)):

            #If there are no clusters created yet, then make the first cluster.
            if (StateDate[i]=='core' and len(self.cluster)==0):

               self.dataSet1["Clusters"][i]=0
               self.cluster.append([i])

            #if there is a clusters search if there's a neighbor point that has been assigned to a cluster.
            elif (StateDate[i]=='core' and len(self.cluster)!=0):


                for j in range(len(neighboring_points)):

                    if(neighboring_points[j][0]==i):

                       for z in range(len(self.core)):

                          if(neighboring_points[j][1]==self.core[z]):

                             find=self.is_in_cluster(self.cluster,self.core[z])

                             #If the neighboring point has already been assigned to a cluster, assign the current point to the same cluster.
                             if(find!=-1):
                               self.dataSet1["Clusters"][i]=find
                               self.cluster[find].append(i)
                               break
                #If there are no neighboring points assigned to a cluster, a new cluster will be created.
                if(self.dataSet1["Clusters"][i]==-1):
                   self.cluster.append([i])
                   self.dataSet1["Clusters"][i]=(len(self.cluster))+1

        #After assigning all the core points, assign all the border points to their respective cores.
        for i in range(len(self.dataSet1)):

                if (StateDate[i]=='border'):

                   for j in range(i,len(neighboring_points)):

                       if(neighboring_points[j][0]==i):

                          for z in range(len(self.core)):

                              if(neighboring_points[j][1]==self.core[z]):

                                 find=self.is_in_cluster(self.cluster,self.core[z])
                                 if(find!=-1):
                                       self.dataSet1["Clusters"][i]=find
                                       self.cluster[find].append(i)
                                       break
```

```python
#All clusters that are density-reachable with each other will be merged into one cluster.
for i in range(len(self.cluster)):
    for j in range(len(self.cluster)):
       if(i!=j):
         for itemC1 in range(len(self.cluster[i])):
            for itemC2 in range(len(self.cluster[j])):
               if(self.cluster[i][itemC1]==self.cluster[j][itemC2]):

                  self.cluster=self.density_reachable(self.cluster,i,j)
                  break

#to remove the empty indexes from the list because they have been transferred into another index.
while([]in self.cluster):
   self.cluster.remove([])
#Update the cluster of each point in dataSet1["Clusters"].
for i in range(len(self.cluster)):
    for j in range(len(self.cluster[i])):
         if(self.dataSet1["Clusters"][self.cluster[i][j]]!= i):
             self.dataSet1["Clusters"][self.cluster[i][j]]= i


return self.dataSet1["Clusters"]
```

predict function: to predict the clusters of the dataset based on the state of each point and its neighbors, then it will return the final clusters.

```python
def fit(self):
    """
    Determine the state of each point by analyzing the list of neighboring points.

    Returns:
    -The list of the points states and two array of the core and border points.
    """
    The_point=0
    Point_count=0
    neighboring_points=self.neighbor_points()

    #The first step is to determine the state of each core point based on the number of its neighbors.
    for i in range(len(neighboring_points)):

      if neighboring_points[i][0]==The_point :

         Point_count+=1

      elif (neighboring_points[i][0]!=The_point and Point_count>=self.min_samples):

         self.core.append(The_point)
         self.dataSet1["state"][The_point]='core'
         The_point+=1
         Point_count=1

      elif (neighboring_points[i][0]!=The_point and Point_count<=self.min_samples):
         The_point+=1
         Point_count=1


    #When a point has a neighbor core and its value is still -1, it is considered a border point.
    for i in range(len(self.dataSet1)):

       if(self.dataSet1["state"][i]==-1):

          for j in range(i,len(neighboring_points)):

              for z in range(len(self.core)):

                  if (neighboring_points[j][0]==i and neighboring_points[j][1]==self.core[z]):
                      self.border.append(i)
                      self.dataSet1["state"][i]='border'
                      break

                  if (neighboring_points[j][0]==i and neighboring_points[j][1]==self.core[z]):
                      self.border.append(i)
                      self.dataSet1["state"][i]='border'
                      break

    #Every state point that has a value of -1 is considered as noise.
    for i,n in enumerate(self.dataSet1["state"]):

        if n==-1:
           self.dataSet1["state"][i]='nois'


    return self.dataSet1["state"],self.core,self.border
```

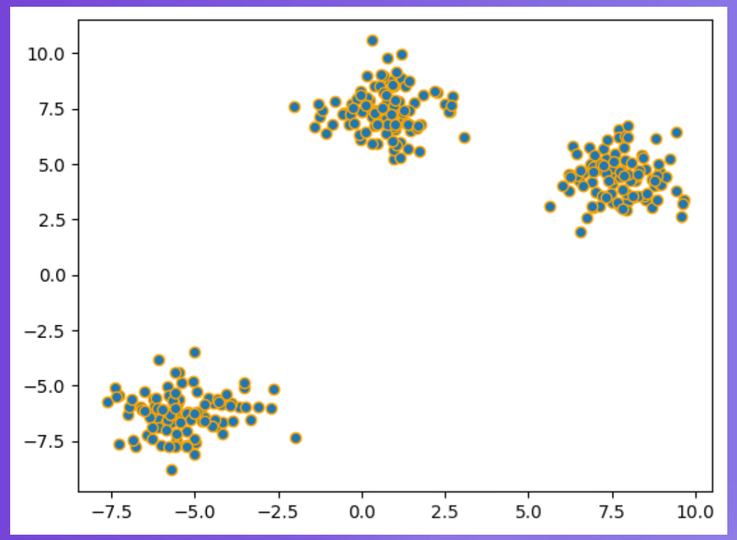
fit function: to determine the state of each point by analyzing the list of neighboring points.And it will return -The list of the points states and two array of the core and border points.On other hands every state point that has a value of -1 is considered as noise.

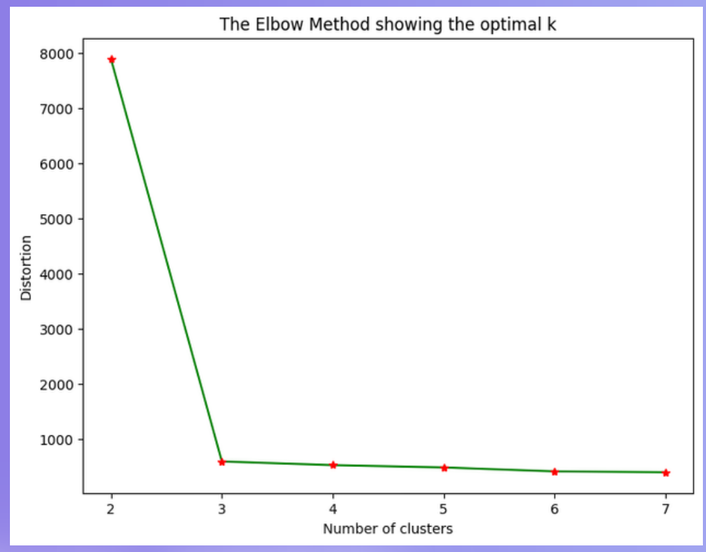
CALCULATE RANDOM STATE

STUDENT1 ID	STUDENT2 ID	STUDENT3 ID	STUDENT4 ID	STUDENT5 ID
2110618	2110826	2110173	2111992	2111606
2+1+1+0+6+1+8	2+1+1+0+8+2+6	2+1+1+0+1+7+3	2+1+1+1+9+9+2	2+1+1+1+6+0+6
19	20	15	25	17
=96				
RANDOM STATE = 96				

GENERATE DATASET1

```
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
```



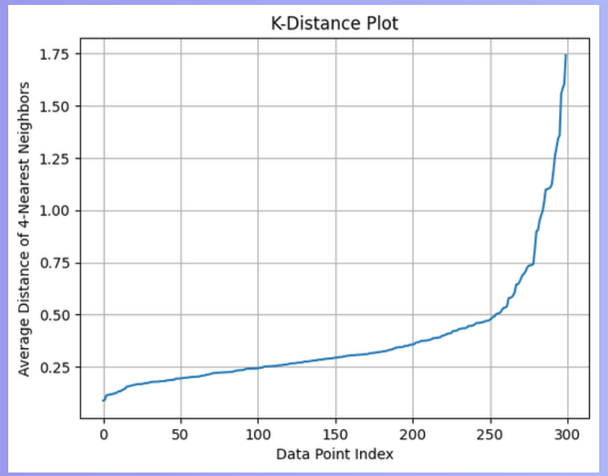


APPLY K-MEAN MODEL

```
kmeans = KMeans(n_clusters=3, random_state=random_state)
labels = kmeans.fit(X)

# Plot
plt.scatter(X[:, 0], X[:, 1], c=labels, edgecolor='orange',cmap = "cool" )
plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1],s = 60,  c='black', marker='x')
plt.title("K-Means Clustering on Blobs Dataset")
plt.show()
```

APPLY DBSCAN MODEL

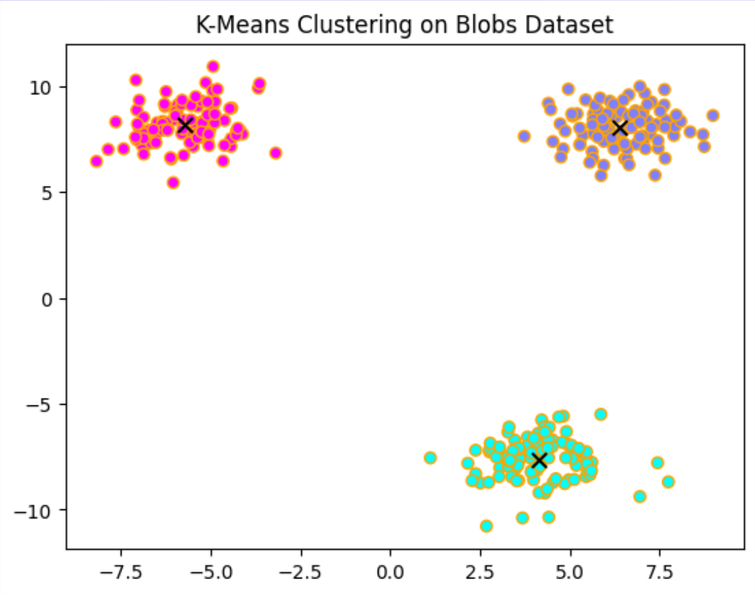
```python
# Calculate the k-distance for each data point
k = 4  # Set the value of k for the k-distance plot
neigh = NearestNeighbors(n_neighbors=k+1)
neigh.fit(X)
distances, indices = neigh.kneighbors(X)
avg_distances = np.mean(distances[:, 1:], axis=1)
sorted_avg_distances = np.sort(avg_distances, axis=0)

plt.plot(np.arange(len(X)), sorted_avg_distances)
plt.xlabel("Data Point Index")
plt.ylabel(f"Average Distance of {k}-Nearest Neighbors")
plt.title("K-Distance Plot")
plt.grid(True)
plt.show()
```

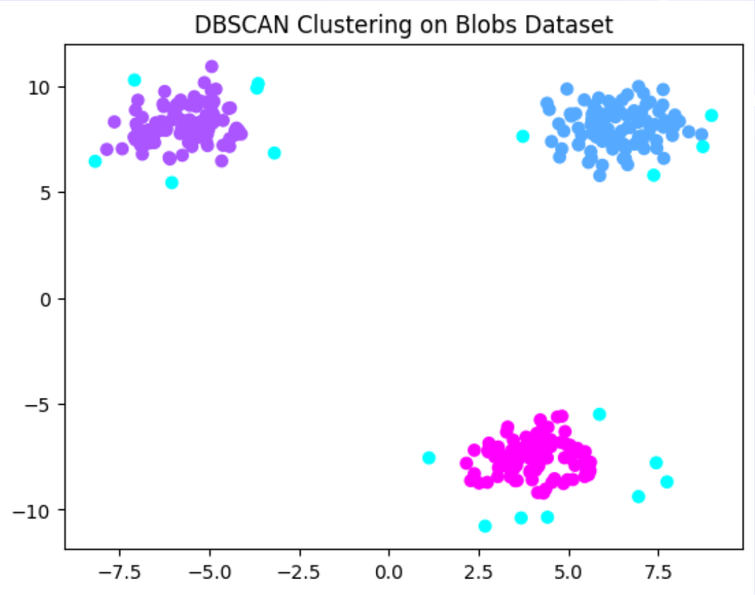


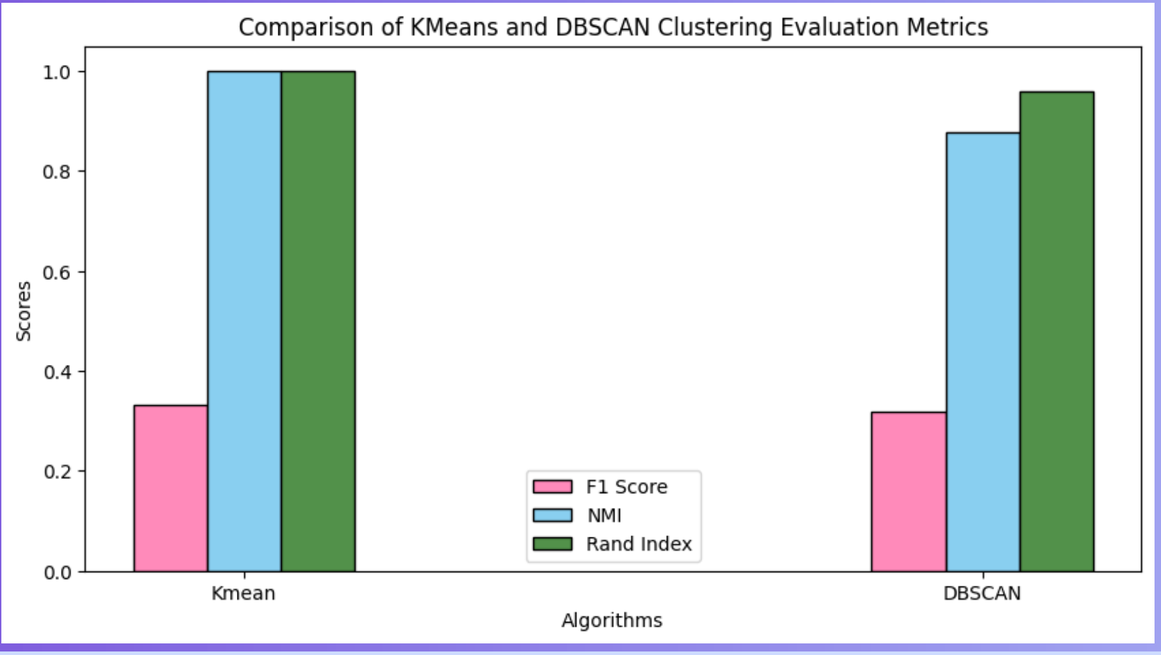

```python
#Implementing the DBSCAN model with dataset 1 after determining the best epsilon value
DB1=DBSCAN(X,0.8,4)
Data_State1,corePoint1,borderPoint1=DB1.fit()
DBLabel1=DB1.predict(Data_State1,corePoint1,borderPoint1)
```

```python
#Visualize the clusters after determining the best epsilon value
plt.scatter(X[:,0], X[:,1], c=DBLabel1, cmap='cool')
plt.title('DBSCAN Clustering on Blobs Dataset')
plt.show()
```

RESULTS

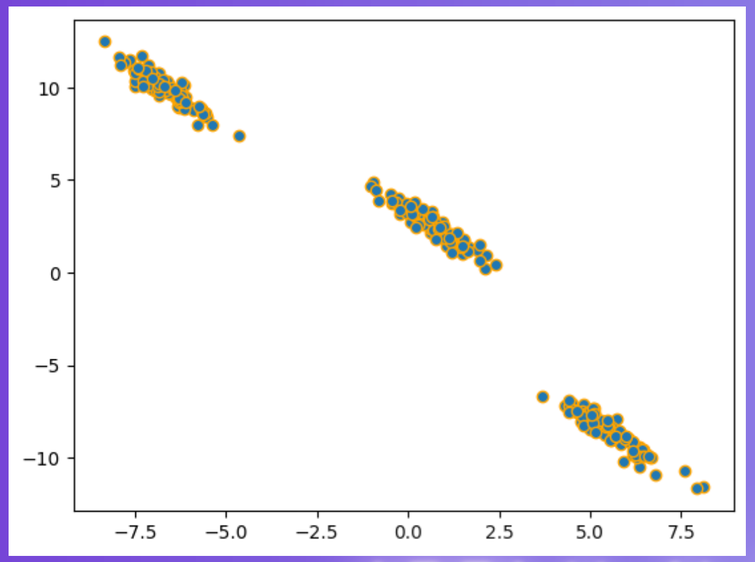

K-Means Clustering on Blobs Dataset
10
5
0
−5
−10
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

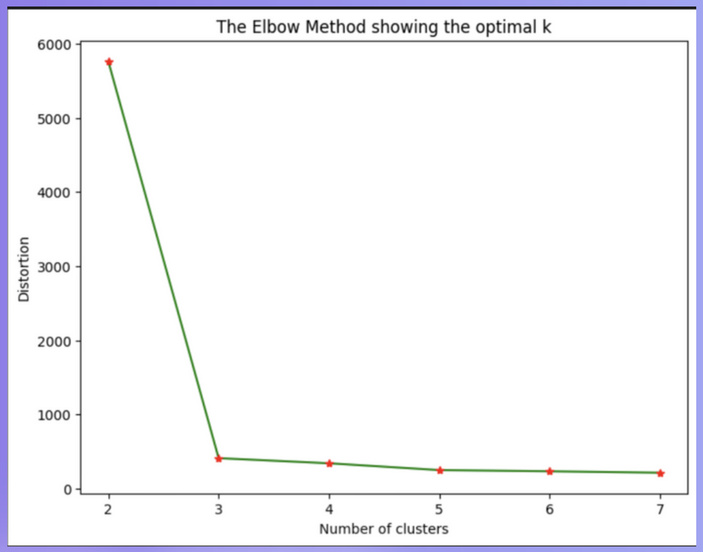

DBSCAN Clustering on Blobs Dataset
10
5
0
−5
−10
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5


Comparison of KMeans and DBSCAN Clustering Evaluation Metrics
Scores
1.0
0.8
0.6
0.4
0.2
0.0
F1 Score
NMI
Rand Index
Kmean
DBSCAN
Algorithms

GENERATE DATASET2

```
X_aniso, Y_aniso = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X_aniso, transformation)
```



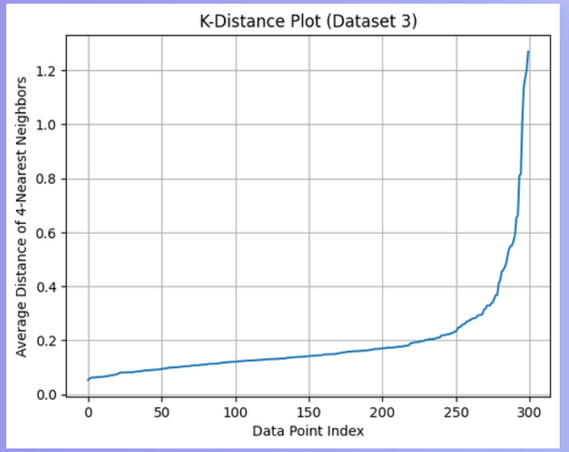


APPLY K-MEAN MODEL

```
kmeans_aniso = KMeans(n_clusters=3, random_state=random_state)
labels_aniso = kmeans_aniso.fit(X_aniso)

# Plot
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=labels_aniso, edgecolor='orange',cmap = "cool" )
plt.scatter(kmeans_aniso.centroids[:, 0], kmeans_aniso.centroids[:, 1], c='black', marker='x')
plt.title("K-Means Clustering on Anisotropic Dataset")
plt.show()
```

APPLY DBSCAN MODEL

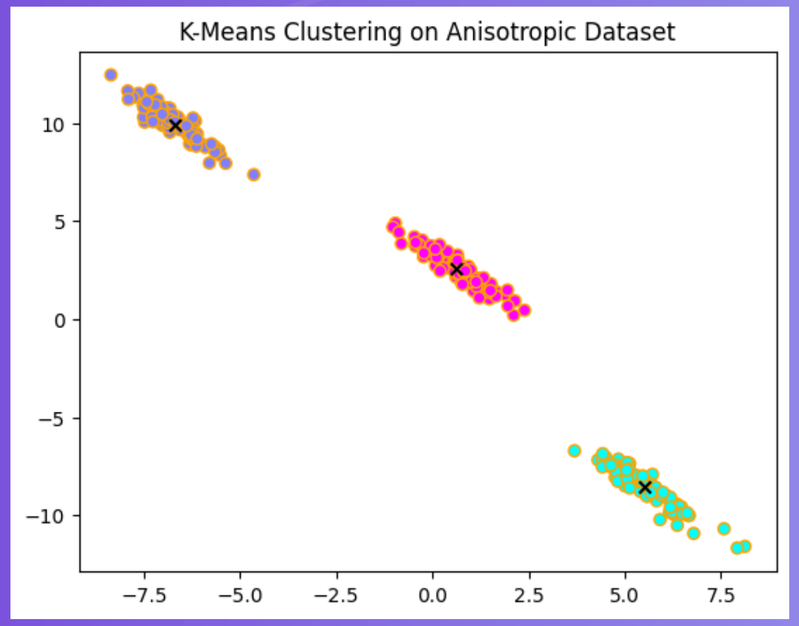
```
# Calculate k-distance graph
k = 4  # Set the value of k for the k-distance plot
neigh = NearestNeighbors(n_neighbors=k+1)
neigh.fit(X_aniso)
distances, _ = neigh.kneighbors(X_aniso)
avg_distances = np.mean(distances[:, 1:], axis=1)
sorted_avg_distances = np.sort(avg_distances, axis=0)

# Plot the k-distance graph
plt.plot(np.arange(len(X_aniso)), sorted_avg_distances)
plt.xlabel("Data Point Index")
plt.ylabel(f"Average Distance of {k}-Nearest Neighbors")
plt.title("K-Distance Plot (Dataset 3)")
plt.grid(True)
plt.show()
```

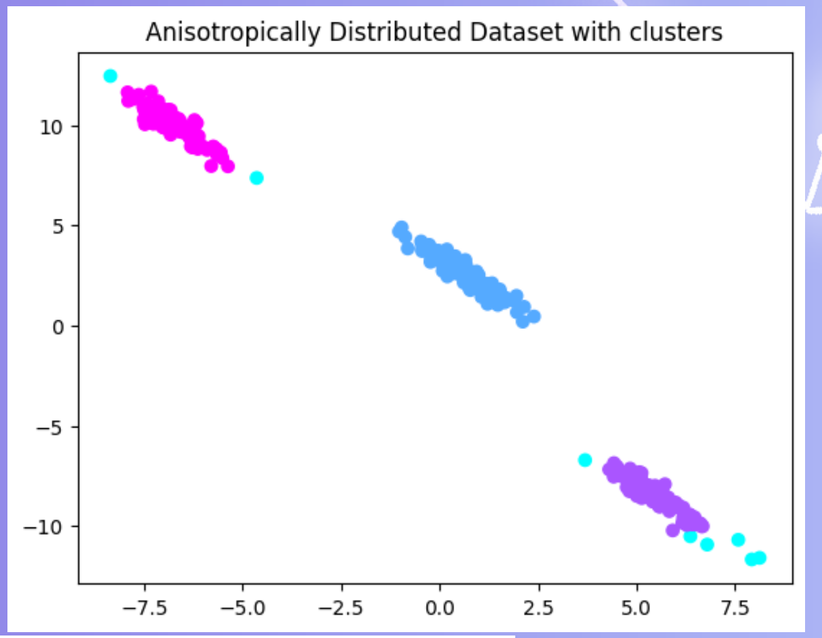


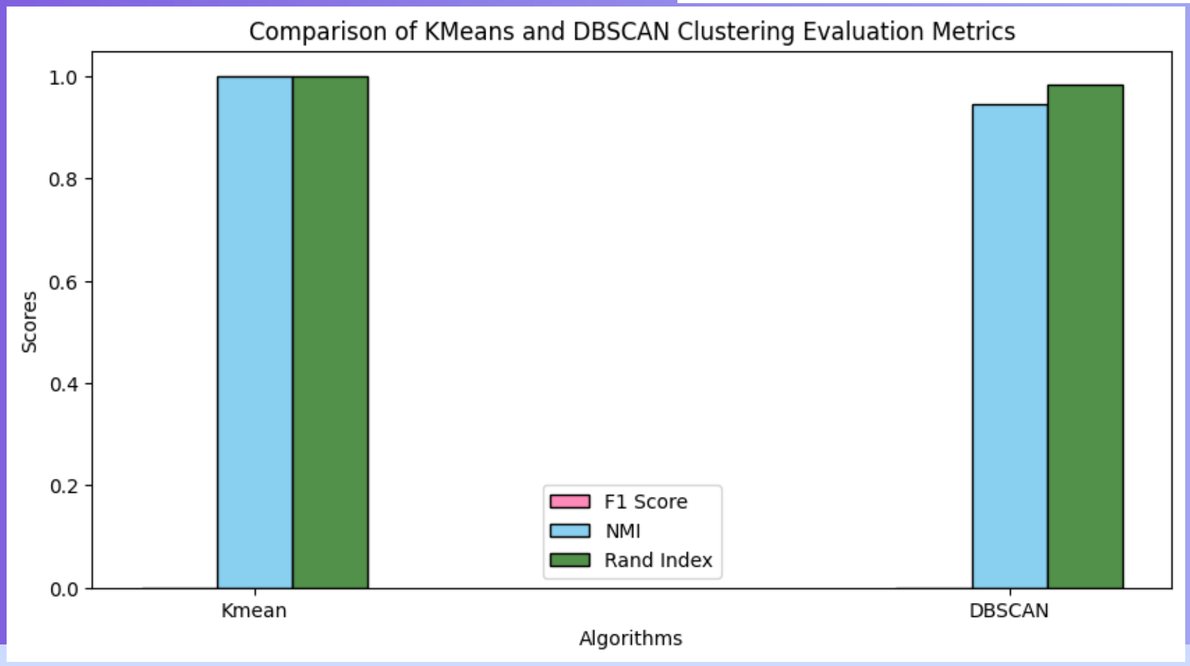

```
[24] DB3=DBSCAN(X_aniso,0.5,3)
     Data_State3,corePoint3,borderPoint3=DB3.fit()
     DBLabel3=DB3.predict(Data_State3,corePoint3,borderPoint3)
```

```
plt.scatter(X_aniso[:,0], X_aniso[:,1], c=DBLabel3, cmap='cool')
plt.title('Anisotropically Distributed Dataset with clusters')
plt.show()
```

RESULTS

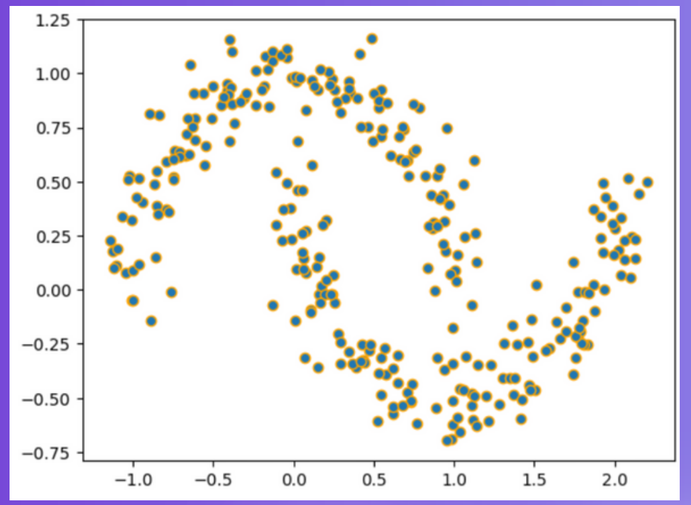

K-Means Clustering on Anisotropic Dataset
10
5
0
−5
−10
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

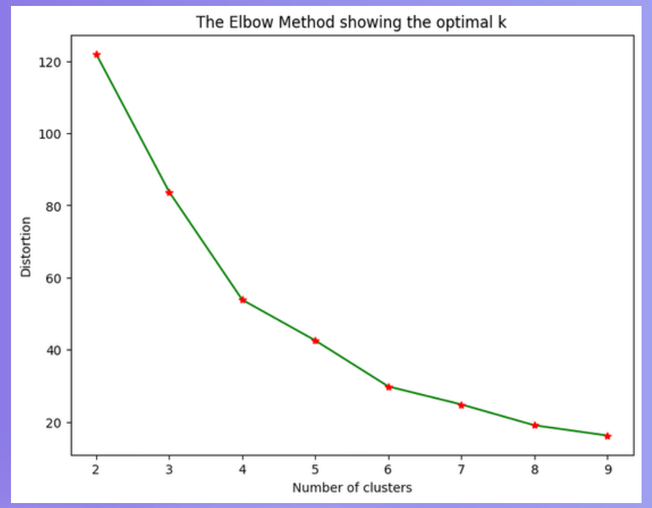

Anisotropically Distributed Dataset with clusters
10
5
0
−5
−10
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5


Comparison of KMeans and DBSCAN Clustering Evaluation Metrics
Scores
1.0
0.8
0.6
0.4
0.2
0.0
F1 Score
NMI
Rand Index
Kmean
DBSCAN
Algorithms

GENERATE DATASET3

```
] X_moons,Y_moons= datasets.make_moons(n_samples=n_samples, noise=0.1, random_state=random_state)
```



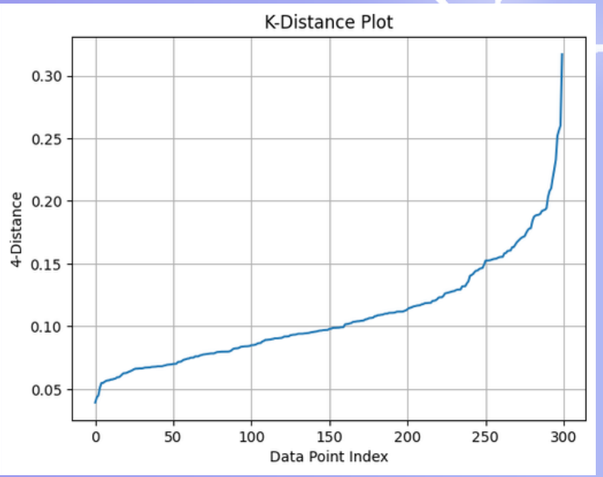


APPLY KMEANS MODEL

```
kmeans_moons = KMeans(n_clusters=3, random_state=random_state)
labels_moons = kmeans_moons.fit(X_moons)

# Plot
plt.scatter(X_moons[:, 0], X_moons[:, 1], c=labels_moons,edgecolor='orange',cmap = "cool" )
plt.scatter(kmeans_moons.centroids[:, 0], kmeans_moons.centroids[:, 1], c='black', marker='x')
plt.title("K-Means Clustering on Noisy Moons Dataset")
plt.show()
```

APPLY DBSCAN MODEL

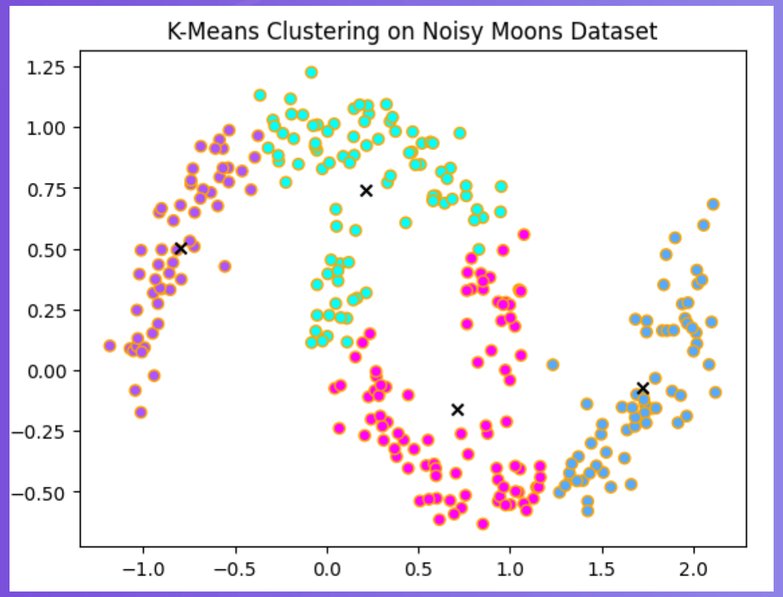
```
k = 4  # Set the value of k for the k-distance plot
neigh = NearestNeighbors(n_neighbors=k+1)
neigh.fit(X_moons)
distances, _ = neigh.kneighbors(X_moons)
k_distances = np.sort(distances[:, k], axis=0)

sorted_distances = np.sort(k_distances, axis=0)

plt.plot(np.arange(len(X_moons)), sorted_distances)
plt.xlabel("Data Point Index")
plt.ylabel(f"{k}-Distance")
plt.title("K-Distance Plot")
plt.grid(True)
plt.show()
```

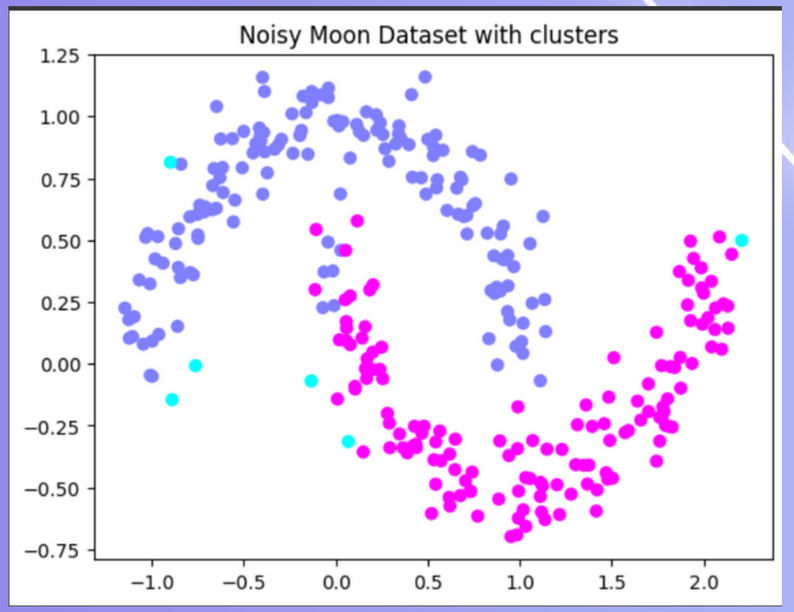


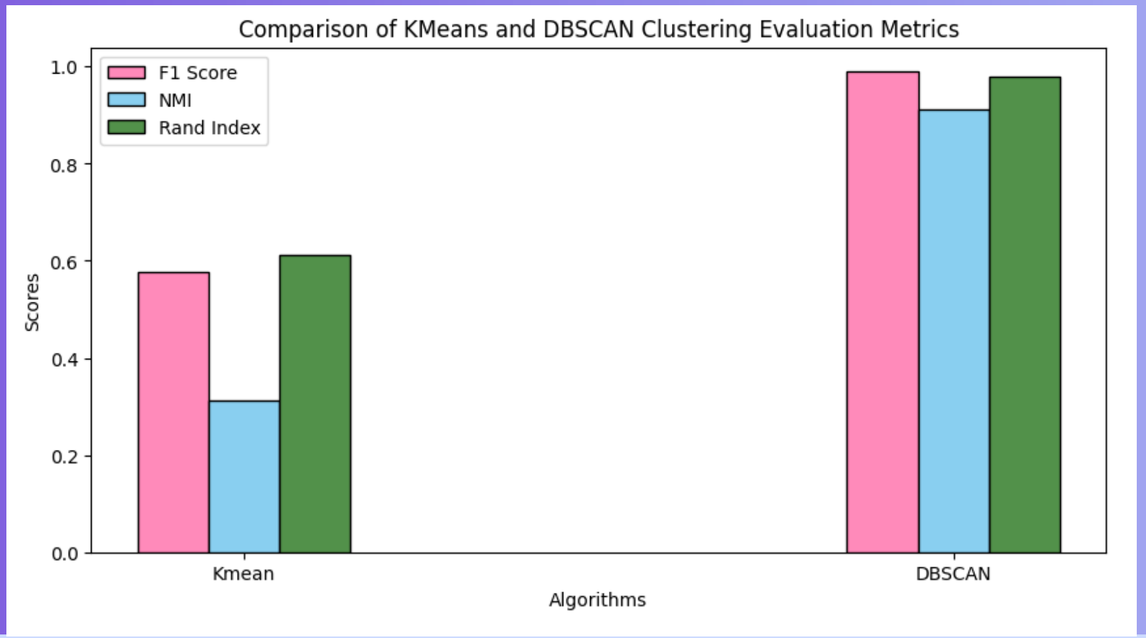

```
#Implementing the DBSCAN model after determining the epsilon value
DB2=DBSCAN(X_moons,0.21,9)
Data_State2,corePoint2,borderPoint2=DB2.fit()
DBLabel2=DB2.predict(Data_State2,corePoint2,borderPoint2)
```

```
#demonstrating the DBSCAN model after determining the epsilon value
plt.scatter(X_moons[:,0], X_moons[:,1], c=DBLabel2, cmap='cool')
plt.title('Noisy Moon Dataset with clusters ')
plt.show()
```

RESULTS

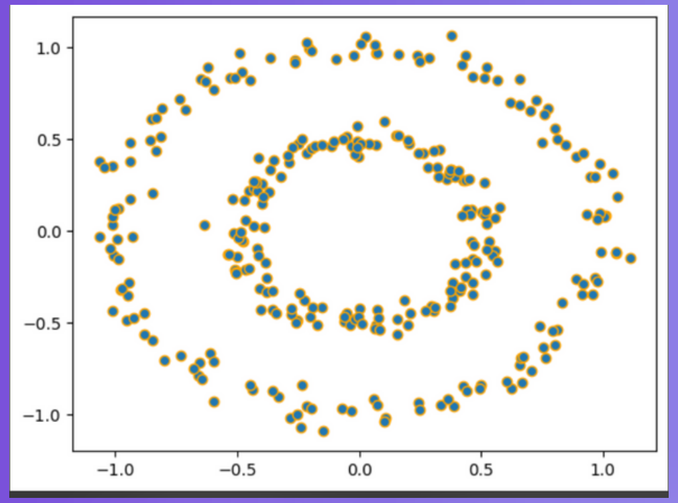

K-Means Clustering on Noisy Moons Dataset
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

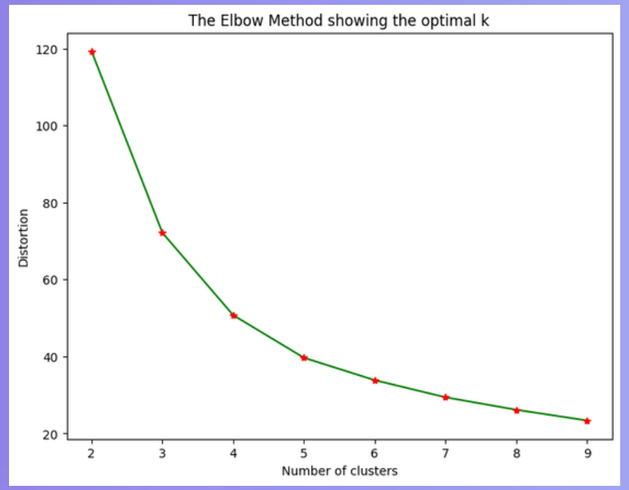

Noisy Moon Dataset with clusters
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


Comparison of KMeans and DBSCAN Clustering Evaluation Metrics
F1 Score
NMI
Rand Index
Scores
1.0
0.8
0.6
0.4
0.2
0.0
Kmean
DBSCAN
Algorithms

GENERATE DATASET4

```
[ ] X_circles, Y_circles = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05, random_state=random_state)
```



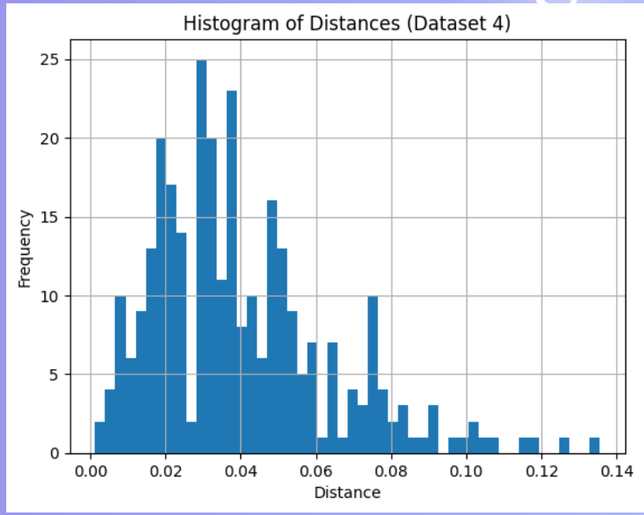


APPLY KMEANS MODEL

```
kmeans_circles = KMeans(n_clusters=3, random_state=random_state)
labels_circles = kmeans_circles.fit(X_circles)

# Plot
plt.scatter(X_circles[:, 0], X_circles[:, 1], c=labels_circles, edgecolor='orange',cmap = "cool" )
plt.scatter(kmeans_circles.centroids[:, 0], kmeans_circles.centroids[:, 1], c='black', marker='x')
plt.title("K-Means Clustering on Noisy Circles Dataset")
plt.show()
```

APPLY DBSCAN MODEL

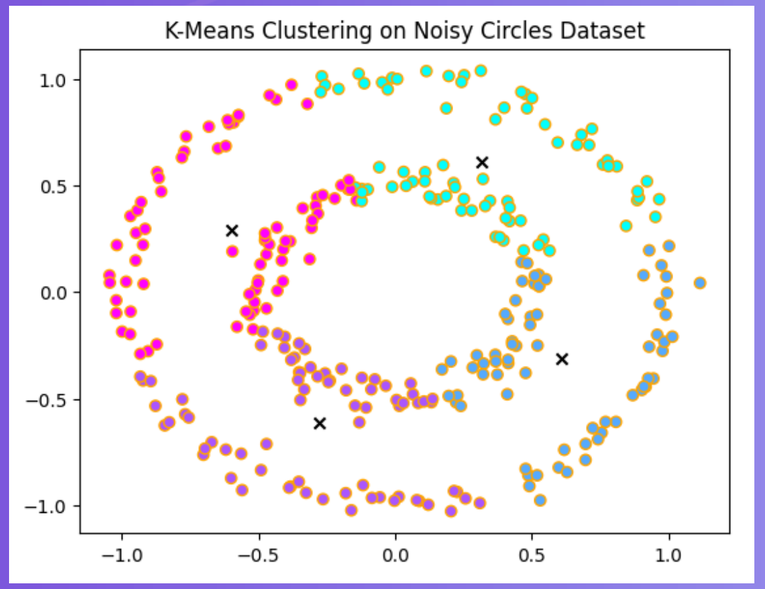
```python
# Calculate distances
neigh = NearestNeighbors(n_neighbors=2)
neigh.fit(X_circles)
distances, _ = neigh.kneighbors(X_circles)
distances = np.sort(distances[:, 1], axis=0)

# Plot histogram of distances
plt.hist(distances, bins=50)
plt.xlabel("Distance")
plt.ylabel("Frequency")
plt.title("Histogram of Distances (Dataset 4)")
plt.grid(True)
```

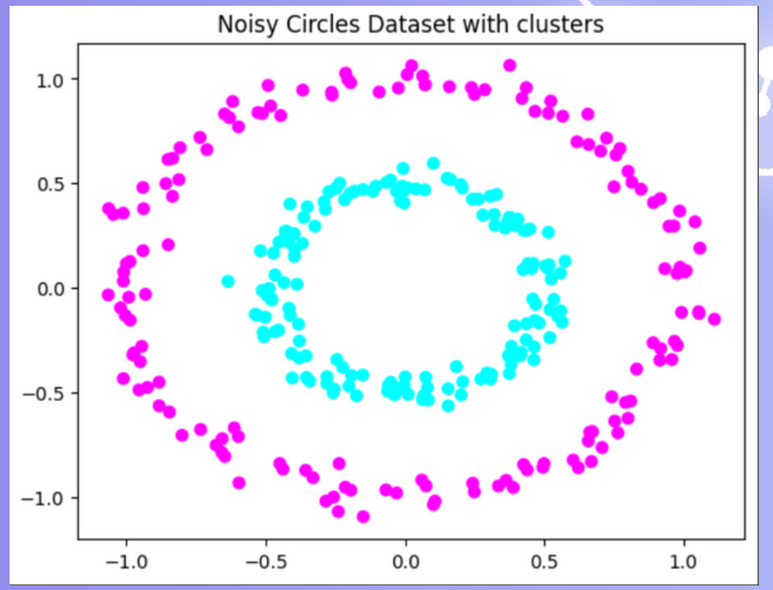


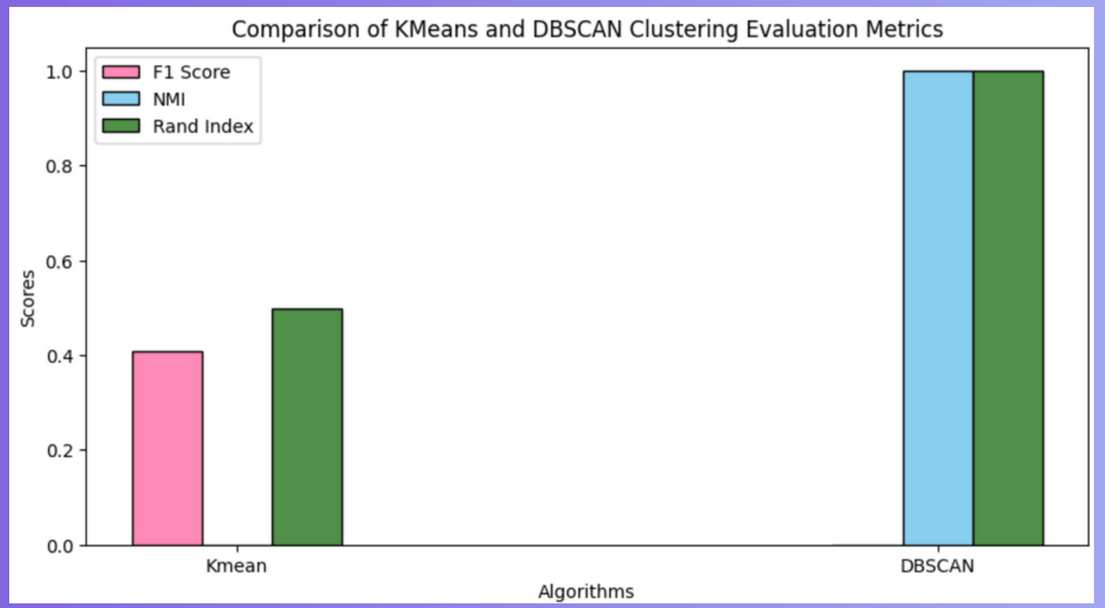

```python
#Implementing the DBSCAN model with dataset after determining the epsilon value
DB4=DBSCAN(X_circles,0.2,4)
Data_State4,corePoint4,borderPoint4=DB4.fit()
DBLabel4=DB4.predict(Data_State4,corePoint4,borderPoint4)
```

```python
#Visualize the clusters
plt.scatter(X_circles[:,0], X_circles[:,1], c=DBLabel4, cmap='cool')
plt.title('Noisy Circles Dataset with clusters')
plt.show()
```

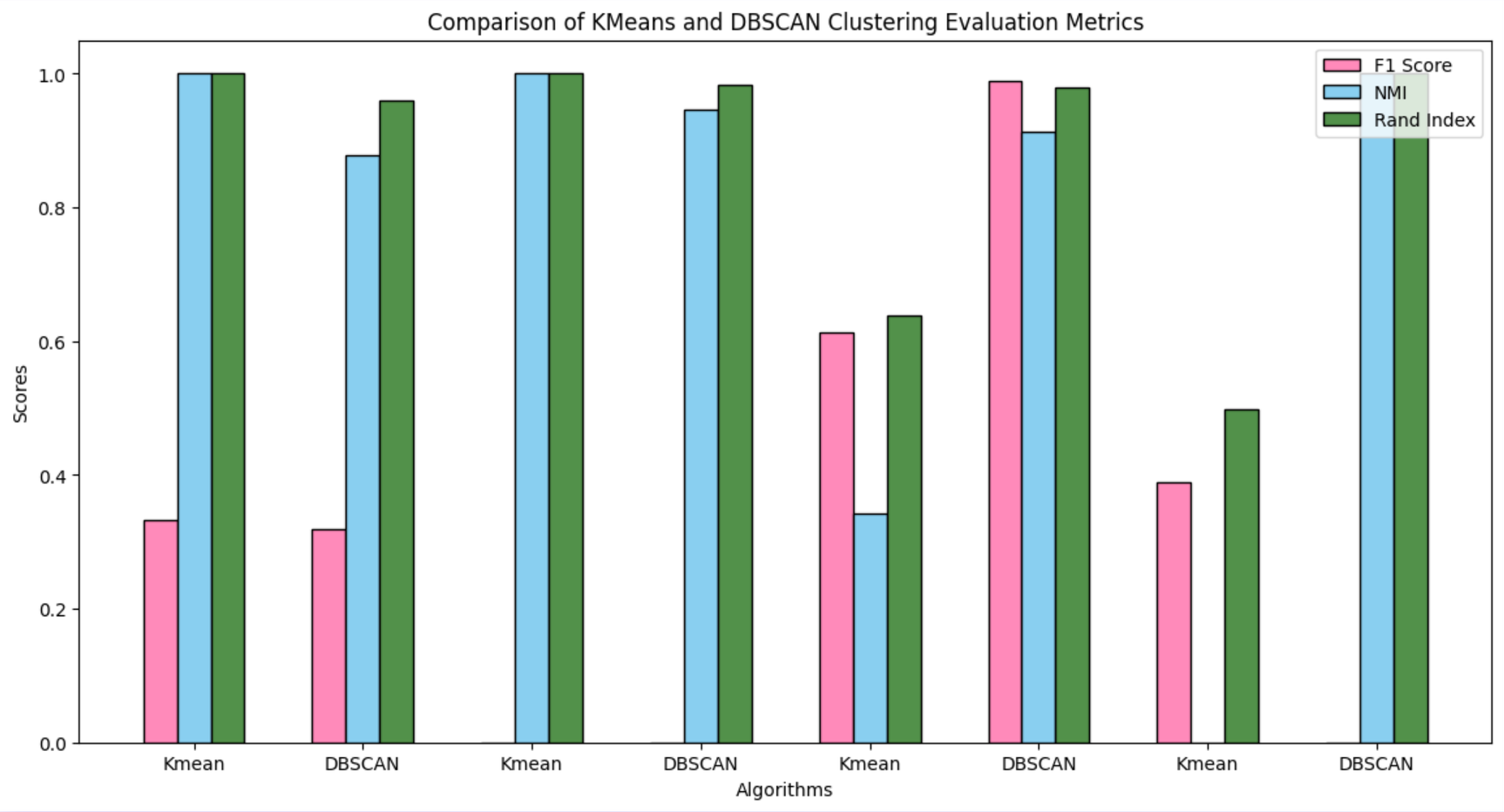
RESULTS


K-Means Clustering on Noisy Circles Dataset
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Noisy Circles Dataset with clusters
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Comparison of KMeans and DBSCAN Clustering Evaluation Metrics
F1 Score
NMI
Rand Index
Scores
1.0
0.8
0.6
0.4
0.2
0.0
Kmean
DBSCAN
Algorithms

SUMMARIZE THE MEASURES VALUES FOR ALL ALGORITHMS



THANK YOU!