1. K-means

i. Implementation:

For K-means Algorithm, the implementation steps are as following:

→ Initialization: randomly choose k points from dataset to be centroid of each k-th cluster.

```
20  def init_centroid(X, k):
21      """ return centroid in each clusters (there are k clusters) """
22      C = X[np.random.choice(X.shape[0], k, replace=False), :]
23      return C
```

→ Assign: assign each data point to its closest cluster. Here I use Euclidean distance to calculate distance from each data point to each cluster.

```
57          ### Assigning each value to its closest cluster ###
58          new_clusters = np.zeros(clusters.shape)
59          for i in range(len(X)):
60              distances = dist(X[i], C)
61              cluster = np.argmin(distances)
62              new_clusters[i] = cluster
63          clusters = new_clusters
```

```
15  def dist(a, b, ax=1):
16      """ calculate euclidean distance """
17      return np.linalg.norm(a - b, axis=ax)
```

→ Update: Store old centroid values and find new centroids by calculating mean of each cluster.

```
### Store the old centroid values ###
C_old = C.copy()
#### Find new centroids by mean of each cluster ###
for i in range(k):
    points = X[clusters == i]
    C[i] = np.mean(points, axis=0)
error = dist(C, C_old, None)
print(error)
```

→ Repeat the second and the third step until there's no difference between old centroids and new centroids.

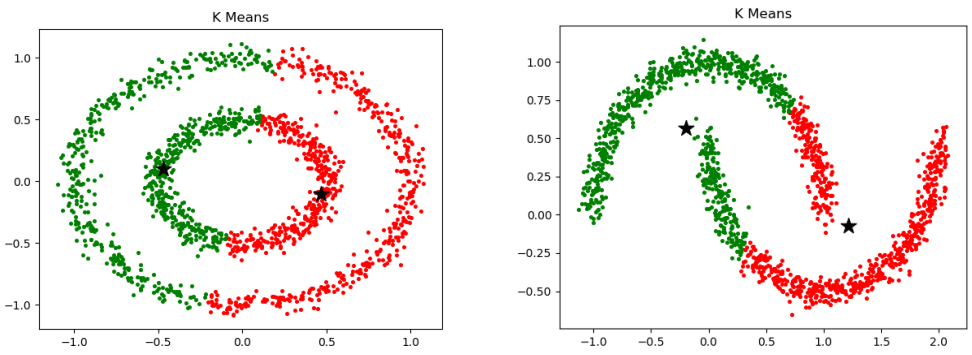
```
    ### Loop untill the error becomes zero ###
    while error != 0:
        ### Assigning each value to its closest cluster ###
        new_clusters = np.zeros(clusters.shape)
        for i in range(len(X)):
            distances = dist(X[i], C)
            cluster = np.argmin(distances)
            new_clusters[i] = cluster
        clusters = new_clusters
        ### Store the old centroid values ###
        C_old = C.copy()
        #### Find new centroids by mean of each cluster ###
        for i in range(k):
            points = X[clusters == i]
            C[i] = np.mean(points, axis=0)
        error = dist(C, C_old, None)
        print(error)
        ### Visualization ###
        visualization(X, k, clusters, C)
```

→ Visualization: For each iteration, visualization the clustering result by giving different colors for points in different clusters. Also, I use "star" to point out where the centroids are.

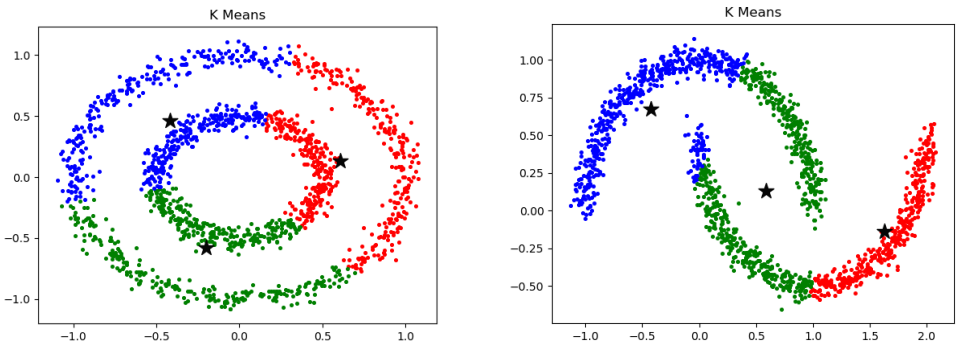
```python
def visualization(X, k, clusters, C):
    global fig_idx
    colors = ['r', 'g', 'b', 'y', 'c', 'm']
    global ax
    ax.clear()
    for i in range(k):
        points = X[clusters == i]
        ax.scatter(points[:, 0], points[:, 1], s=7, c=colors[i])
    ax.scatter(C[:, 0], C[:, 1], marker='*', s=200, c='#050505')
    ax.set_title('K Means (Gamma=30)')
    # plt.savefig('k_means_figures/moon/k_means_' + str(fig_idx) + '.png')
    fig_idx += 1
    plt.draw()
    plt.pause(0.3)
```

ii. Result:

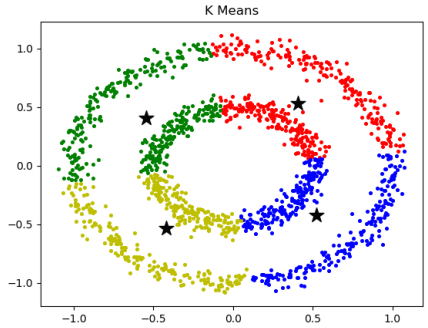
K=2: kmeans_circle.gif, kmeans_moon.gif

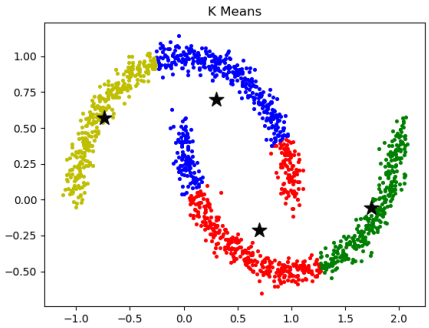


K=3: kmeans_3k_circle.gif, kmeans_3k_moon.gif



K=4: kmeans_4k_circle.gif, kmeans_4k_moon.gif





iii. K-means++:

I use K-means++ to do initialization. The implementation steps are as follow:

→ Randomly choose one point from dataset to be the first centroid.

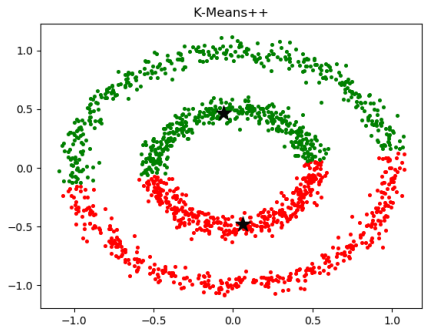
→ For each data point P, calculate the shortest distance from itself to each centroid, and store in array D (i.e. Di represents shortest distance from point i to each centroid)

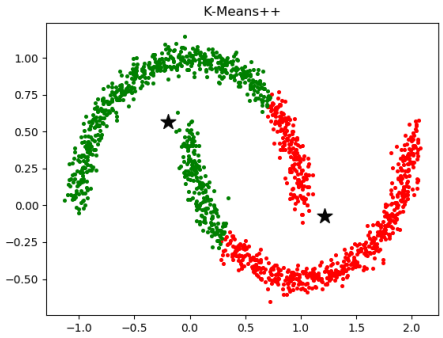
→ Use D to be the weight probability (Bigger weight for bigger D), randomly choose the next point using this weight until k centroids have been found.

```
    C = []
    ### randomly choose one to be the first centroid ###
    idx = np.random.randint(n)
    # C = np.append(C, X[idx])
    C.append(X[idx])
    ### Use shortest distance to be the weight and find next centroid ###
    while len(C) < k:
        D = []
        for i in range(n):
            d_to_centroids = np.array(
                [dist(X[i], C[j], None) for j in range(len(C))])
            D.append(np.min(d_to_centroids))
        D = np.array(D)
        D_weight = D / np.sum(D)
        next_C_idx = np.random.choice(n, p=D_weight)
        C.append(X[next_C_idx])
    C = np.array(C)
    return C
```

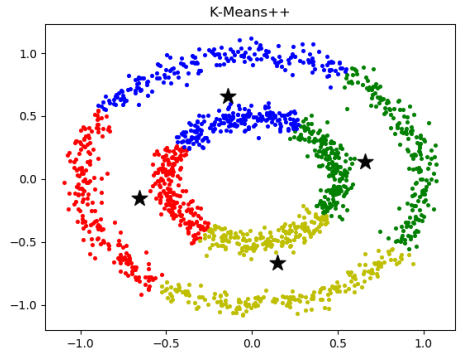
The results are shown below:

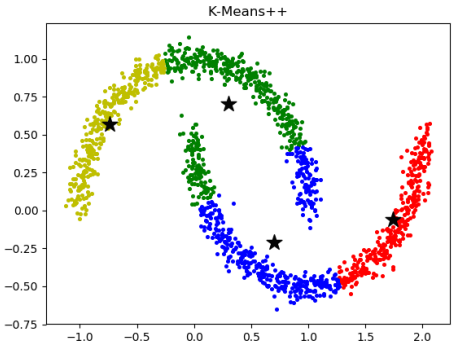
K=2: kmeans_plus_circle.gif, kmeans_plus_moon.gif





K=4: kmeans_plus_4k_circle.gif, kmeans_plus_4k_moon.gif





iv. Discussion:

- From these two dataset, the problem of K-means can be found

clearly. K-means only can detect clusters that are (roughly) linearly separable.

- Comparing normal K-means and K-means++, the difference can be found after trying clustering several times. In normal K-means, centroids choosing is totally random, so the situation that it finds dense centroids might be easily encountered. However, in K-means++, although the initialization of centroid is partially random, but since a weight is given when choosing centroids, the situation above can be avoided.

2. Kernel K-means

i. Implementation:

For kernel K-means, the implementation steps are similar to K-means. The only difference is the way to calculate distance – here RBF-Kernel is used.

The formula to calculate kernel distance is shown below:

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)\right\|$$
$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_{n}\alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q}\alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$

Firstly, precompute 3rd term (it doesn't need to loop with Xj).

```
98          ### Precompute kernel distance term 3 ###
99          ''' 1 / |Ck|^2 * sum(sum(A_kp*A_kq*K(Xp, Xq))) '''
100         term3 = np.zeros((k))
101         cluster_cnt = np.array([(clusters == i).sum() for i in range(k)])
102         for p in range(k):
103             result = 0
104             result = np.sum(
105                 RBF_kernel(X[clusters == p], X[clusters == p], gamma))
106             term3[p] = 1 / (cluster_cnt[p]**2) * result
```

Then, compute 1st and 2nd term for each data point Xj.

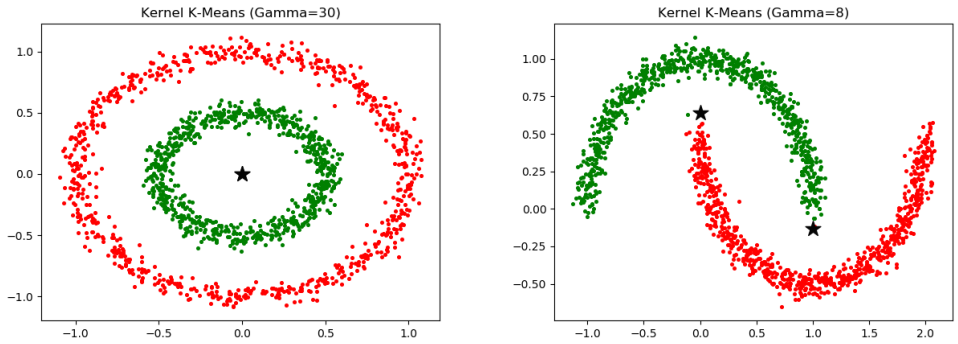
Combine these three terms to calculate kernel distance(term1-term2+term3).

```
38  def kernel_dist(Xj, X, clusters, k, term3, gamma):
39      """
40      Calculate Kernel Distance
41      K(Xj, Xj) - 2 / |Ck| * sum(A_kn*K(Xj, Xn)) + 1 / |Ck|^2 * sum(sum(A_kp*A_kq*K(Xp, Xq)))
42      where if the data point Xn is assigned to the k-th cluster, then A_kn = 1
43      """
44      ### kernel distance from Xj to each centroid ###
45      dist = np.zeros((k))
46
47      ### first term ###
48      term1 = RBF_kernel(Xj, Xj, gamma)
49      ### second term ###
50      cluster_cnt = np.array([(clusters == i).sum() for i in range(k)])
51      term2 = np.zeros((k))
52      for i in range(k):
53          term2[i] = 2 / (cluster_cnt[i]) * np.sum(
54              RBF_kernel(Xj, X[clusters == i], gamma))
55      return term1 - term2 + term3
```

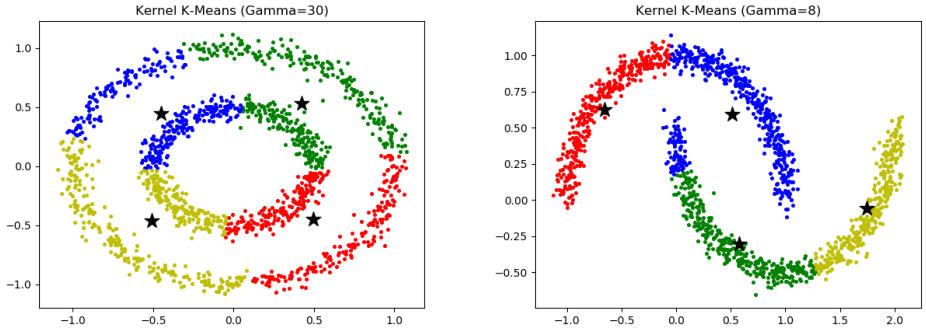
```
111                 distances = kernel_dist(X[i], X, clusters, k, term3, gamma)
```

ii. Result: (Here I only shows the best result of K-means++ initialization.)

K=2: kernel_circle.gif, kernel_moon.gif



K=4: kernel_4k_circle.gif, kernel_4k_moon.gif



iii. Discussion:

- Kernel K-means can solve K-means problem – it can do nonlinearly separation. However, parameter tuning is needed to get better result. I have tried many gamma-values and finally found the best gamma for those two datasets.
- Even with the help of K-means++, I still need to try many times to get the right centroids, but compared to normal K-means, it can save lots of time wasted on finding good centroids.

3. Spectral clustering

i. Implementation:

Here I use unnormalized spectral clustering. And the algorithm is as following:

Unnormalized spectral clustering

Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the unnormalized Laplacian L.
- **Compute the first k eigenvectors $u_1, \ldots, u_k$ of L.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j | \, y_j \in C_i\}$.

The implementation steps are as following:

→ Construct a similarity graph (which stores kernel distances between each points)

```
91  def similarity_graph(data):
92      gamma = 30
93      n = data.shape[0]  # 1500
94      G = RBF_kernel(data, data, gamma)
95      return G
```

→ Compute the unnormalized Laplacian L: L = D – W, where D denotes degree matrix. (Here, W is the similarity graph)

```
 98  def laplacian(W):
 99      n = G.shape[0]  # 1500
100
101      def degree_matrix():
102          # D = np.zeros((n, n))
103          # for i in range(n):
104          #     for j in range(n):
105          #         if (i != j):
106          #             D[i][j] += W[i][j]
107          return np.diag(np.sum(W, axis=1))
108
109      return degree_matrix() - W
```

→ Compute the first k eigenvalues u1, u2, …, uk for L.

```
112  def find_k_smallest_eigenvalues(L, K):
113      eigen_value, eigen_vector = np.linalg.eig(L)
114      sorting_index = np.argsort(eigen_value)
115      eigen_value = eigen_value[sorting_index]
116      eigen_vector = eigen_vector.T[sorting_index]
117      return eigen_value[1:k + 1], (eigen_vector[0:k])
```

→ Cluster the points (yi)i=1,...,n in Rk with the k-means algorithm into

clusters C1, . . . , Ck.

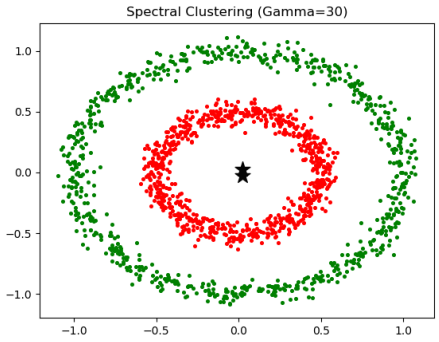
```
140      C = init_centroid(eigen_vector.T, k)
141      print(C)
142      k_means(data_point, eigen_vector.T, C, k)
```

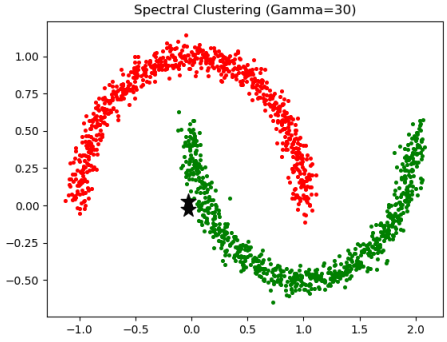
→ The main function is shown below.

```python
if __name__ == "__main__":
    k = 2
    data_point = loadData('moon.txt')
    plt.ion()
    """ Construct a similarity graph G """
    G = similarity_graph(data_point)
    # print(G[2][3])
    # print(G[3][2])
    print(G)
    """ Compute the unnormalized Laplacian L """
    L = laplacian(G)
    print(L)
    """ Compute the first k eigenvectors u1, . . . ,uk of L. """
    eigen_value, eigen_vector = find_k_smallest_eigenvalues(L, k)
    print('eigen_value:')
    print(eigen_value)
    print('eigen_vector:')
    print(eigen_vector[0])
    #print(L @ eigen_vector.T)
    """ Cluster the points (yi)i=1,...,n in Rk with the k-means algorithm into clusters C1, . . . , Cl
    C = init_centroid(eigen_vector.T, k)
    print(C)
    k_means(data_point, eigen_vector.T, C, k)
    plt.show()
```

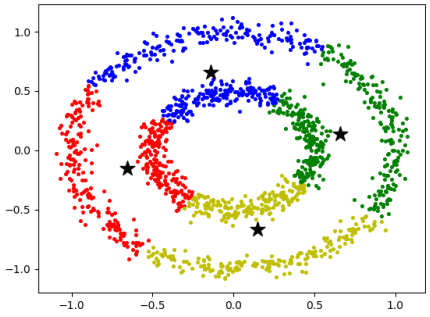
ii. Result:

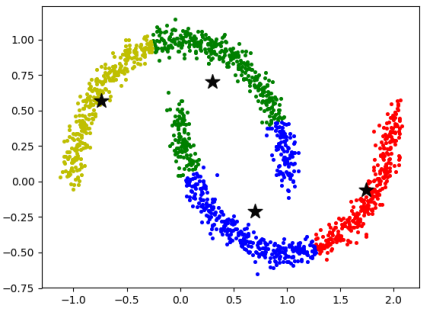
K=2: spectral_circle.gif, spectral_moon.gif





K=4: spectral_4k_circle.gif, spectral_4k_moon.gif





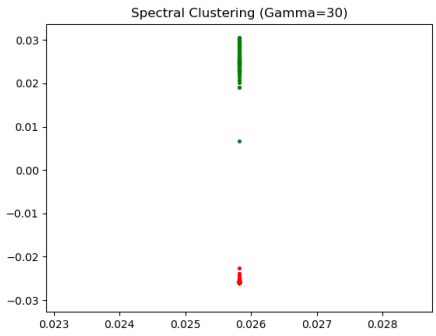
iii. Eigenspace:

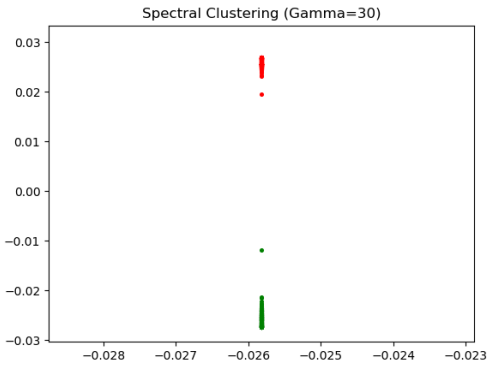
In eigenspace of graph Laplacian, data points within the same cluster do

have the same coordinates because they are indicator vectors on the partition.

The result is shown below:

(spectral_eigenspace_circle.gif, spectral_eigenspace_moon.gif)





iv. Discussion:

Spectral clustering is pretty fast. But tuning on parameter gamma is still needed. Here, I found that when gamma=30, spectral clustering gets the best result for these two datasets.

4. DBSCAN

i. Implementation:

The pseudo code is as below: (Reference: WIKI)

```
DBSCAN(D, eps, MinPts) {
   C = 0
   for each point P in dataset D {
      if P is visited
         continue next point
      mark P as visited
      NeighborPts = regionQuery(P, eps)
```

```
     if sizeof(NeighborPts) < MinPts
        mark P as NOISE
     else {
        C = next cluster
        expandCluster(P, NeighborPts, C, eps, MinPts)
     }
   }
}

expandCluster(P, NeighborPts, C, eps, MinPts) {
   add P to cluster C
   for each point P' in NeighborPts {
     if P' is not visited {
        mark P' as visited
        NeighborPts' = regionQuery(P', eps)
        if sizeof(NeighborPts') >= MinPts
           NeighborPts = NeighborPts joined with
NeighborPts'
     }
     if P' is not yet member of any cluster
        add P' to cluster C
   }
}

regionQuery(P, eps)
   return all points within P's eps-neighborhood
(including P)
```

I declare a class DBSCAN to do dbscan clustering.

The class has three part: main, neighbors, and expand_cluster.

For neighbors part, it find out all points within P's episilon-neighborhood

(including P) to be P's neighbors. Note that I use kernel distance here, so

"episilon-neighborhood" now is those points whose kernel distance is

larger than episilon.

```
51      def neighbors(self, point_idx, data):
52          """ return all points within P's eps-neighborhood (including P) """
53          # neighbors = []
54          neighbor_idx = []
55          for idx in range(data.shape[0]):
56              # if dist(data[idx], point) < self.eps:
57              if self.K[idx, point_idx] > self.eps:
58                  # neighbors.append(data[idx])
59                  neighbor_idx.append(idx)
60          # neighbors = np.array(neighbors)
61          neighbor_idx = np.array(neighbor_idx)
62          return neighbor_idx
```

For main part, it checks every data point P in the data set to find out whether it is a core point. If it's a core point, then merge all its neighbors into cluster a new cluster.

```
89      def dbscan_main(self, data):
90          cluster = -1
91          for data_idx in range(data.shape[0]):
92              if self.hasVisit[data_idx] == 1:  # has been visited
93                  continue
94              self.hasVisit[data_idx] = 1  # mark data_idx as visited
95              neighbor_idx = self.neighbors(data_idx, data)
96              if neighbor_idx.shape[0] < self.minPts:
97                  clusters[data_idx] = -2  #  it's noise!!
98              else:
99                  cluster += 1
100                 neighbor_idx = self.expand_cluster(data_idx, data,
101                                                    neighbor_idx, cluster)
102             print('clusters:')
103             print(self.clusters)
104             print(max(self.clusters))
105             visualization(data, max(self.clusters) + 1, self.clusters)
106         return self.clusters
```

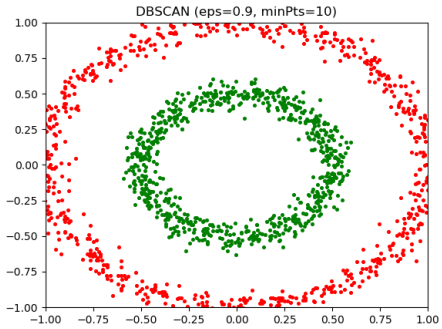
For expand_cluster part, in addition to put P into cluster C, we also need to expand P's neighbors, put them into cluster C, and check if they are core points or not. If P's neighbor is a core point, then put P's neighbor's neighbors into P's neighborhood.

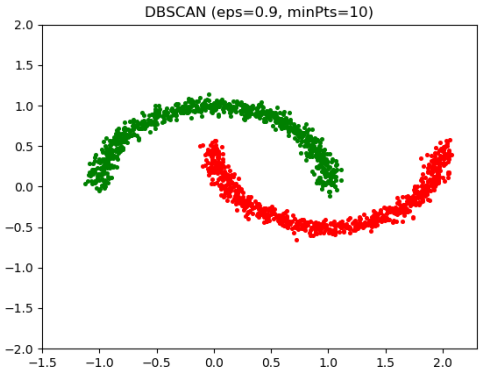
Note that I use **set** here so that when joining neighborhood, it will guarantee that no one is joined twice.

```
64      def expand_cluster(self, point_idx, data, neighbor_idx, C):
65          # add point to cluster C
66          self.clusters[point_idx] = C
67          neighbor_idx_set = set(neighbor_idx)
68          # changed = True
69          neighbor_idx = neighbor_idx.tolist()
70          for i, idx in enumerate(neighbor_idx):
71              print(i)
72
73              if self.hasVisit[idx] == 0:  # hasn't been visited
74                  self.hasVisit[idx] = 1  # mark it as visited
75                  new_neighbor_idx = self.neighbors(idx, data)
76                  if new_neighbor_idx.shape[0] >= self.minPts:
77                      first_encounter_neighbors = set(
78                          new_neighbor_idx).difference(neighbor_idx_set)
79
80                      neighbor_idx.extend(list(first_encounter_neighbors))
81
82                      neighbor_idx_set = neighbor_idx_set.union(
83                          set(new_neighbor_idx))
84                      #neighbor_idx = np.append(neighbor_idx, new_neighbor_idx)
85              if self.clusters[idx] == -1:  # hasn't belong to any cluster
86                  self.clusters[idx] = C
87          return np.array(neighbor_idx)
```

ii. Result:

dbscan_circle.gif, dbscan_moon.gif





iii. Discussion:

DBSCAN is fast and get good result for these two datasets. But the drawback is that we need to do more parameter tuning to get better result

comparing to the other three algorithm.