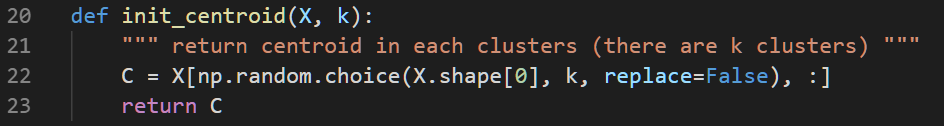
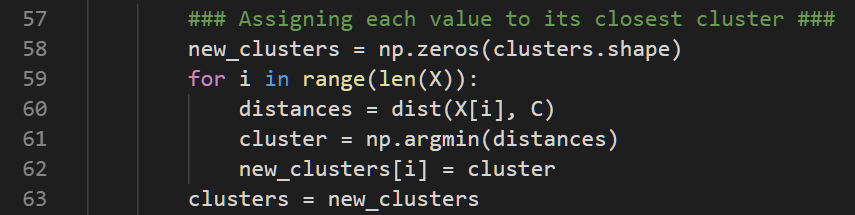
1. **K-means**
2. Implementation:

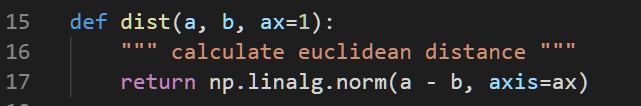
For K-means Algorithm, the implementation steps are as following:

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

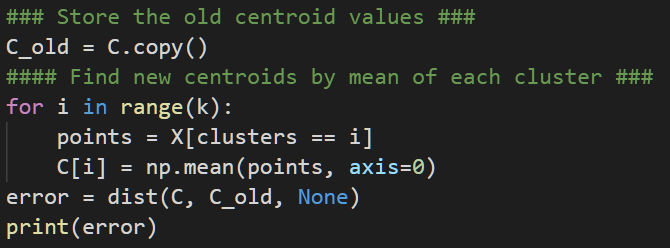


→ Assign: assign each data point to its closest cluster. Here I use Euclidean distance to calculate distance from each data point to each cluster.

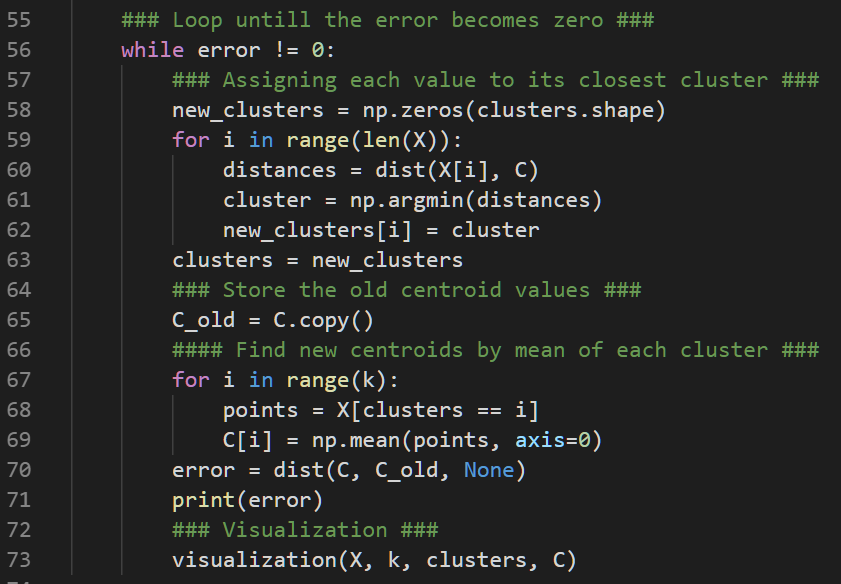




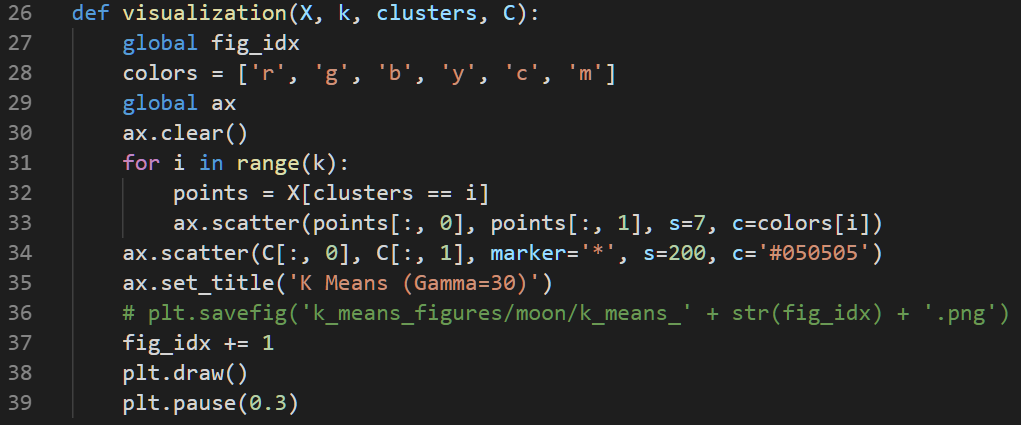
→ Update: Store old centroid values and find new centroids by calculating mean of each cluster.



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

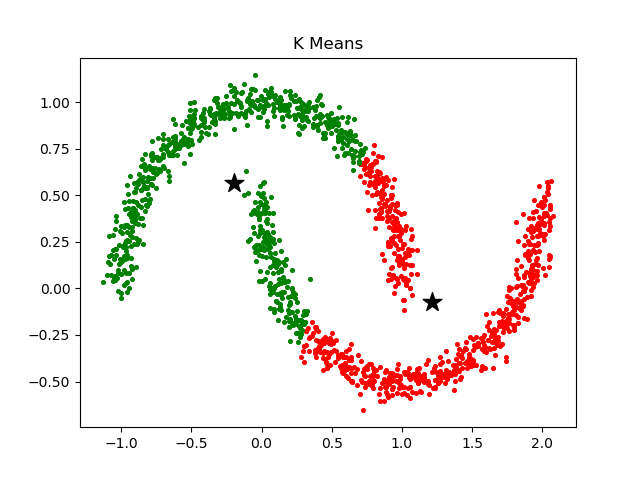
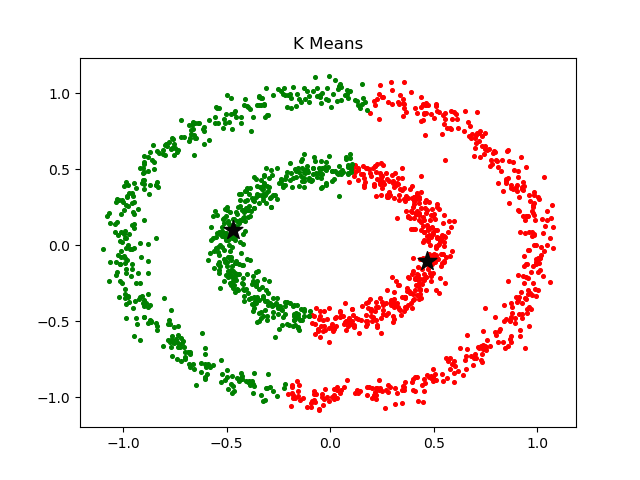


→ Visualization: For each iteration, visualization the clustering result by giving different colors for points in different clusters. Also, I use “star” to point out where the centroids are.

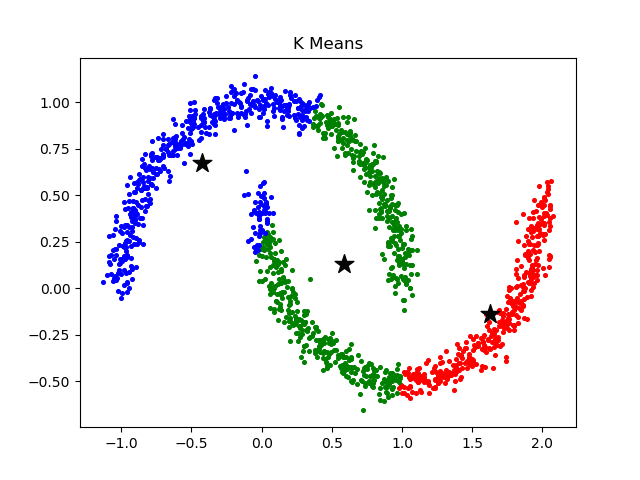
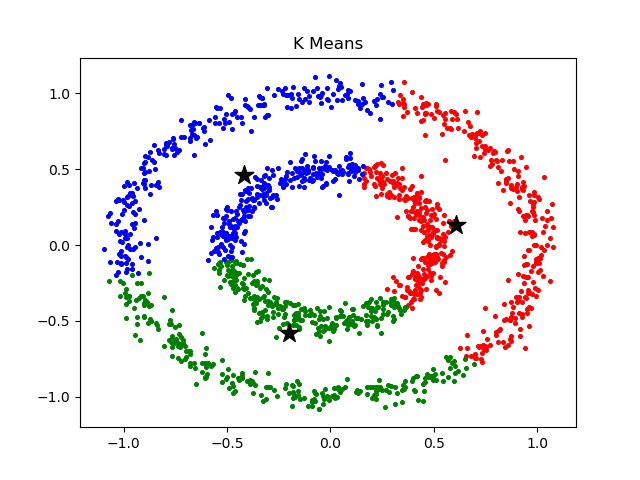


1. Result:

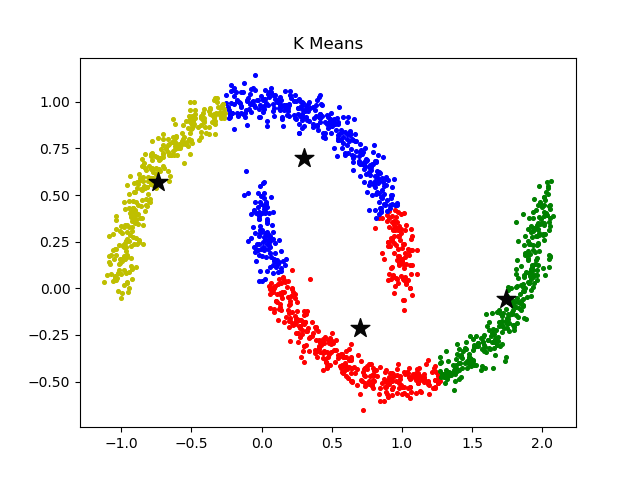
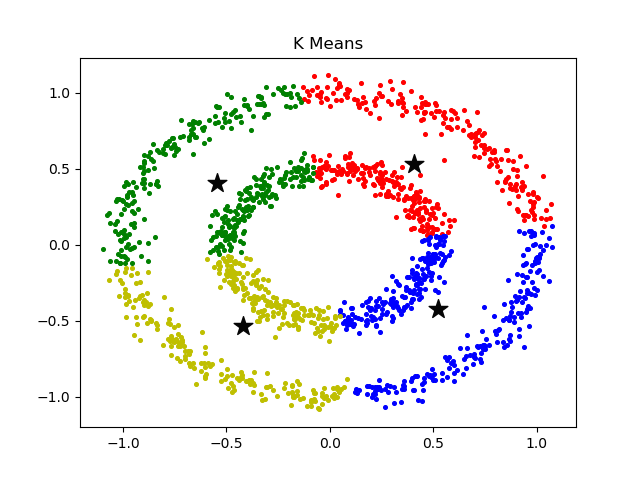
K=2: kmeans\_circle.gif, kmeans\_moon.gif



K=3: kmeans\_3k\_circle.gif, kmeans\_3k\_moon.gif



K=4: kmeans\_4k\_circle.gif, kmeans\_4k\_moon.gif



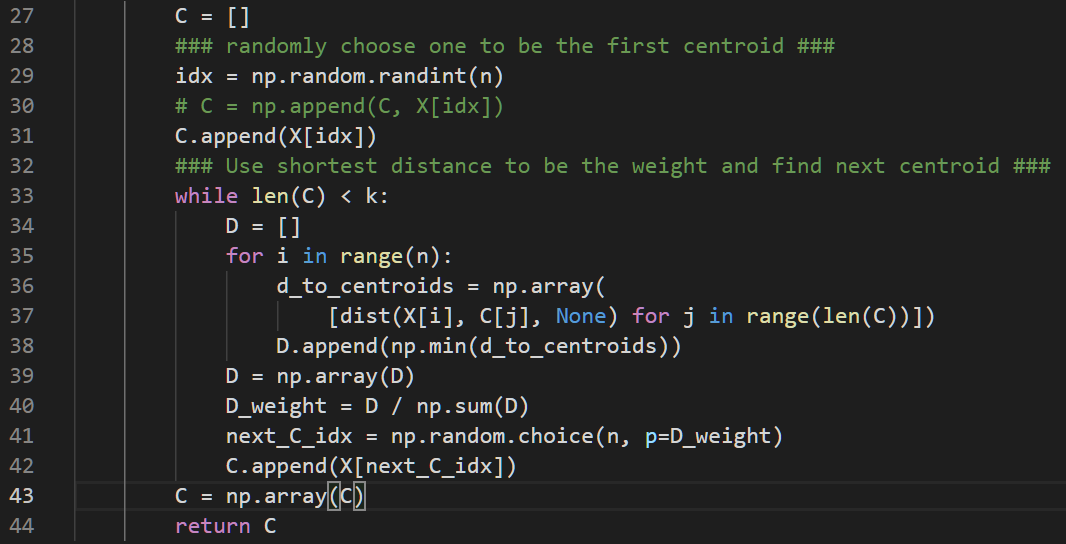
1. K-means++:

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

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

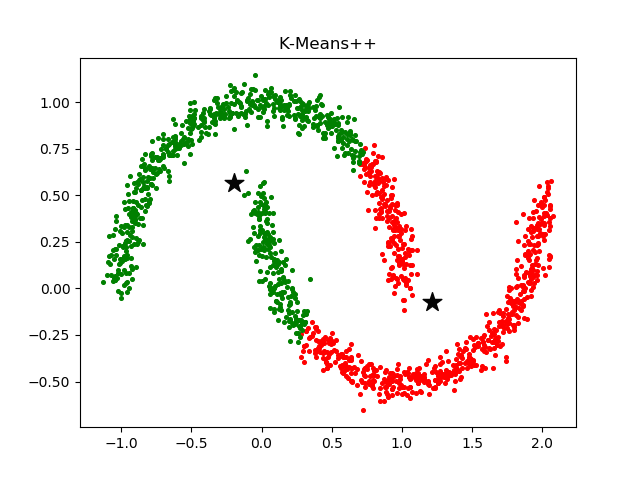
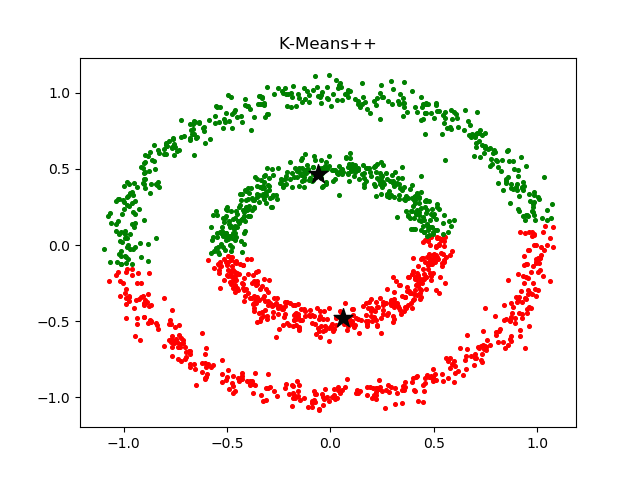
→ For each data point P, calculate the shortest distance from itself to each centroid, and store in array D (i.e. Di represents shortest distance from point i to each centroid)

→ 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.

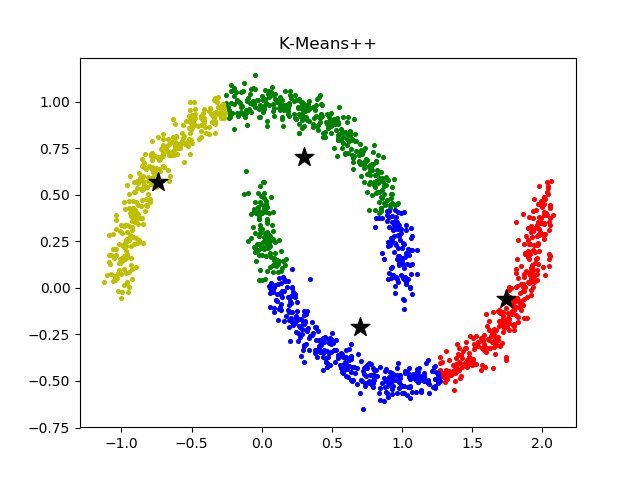
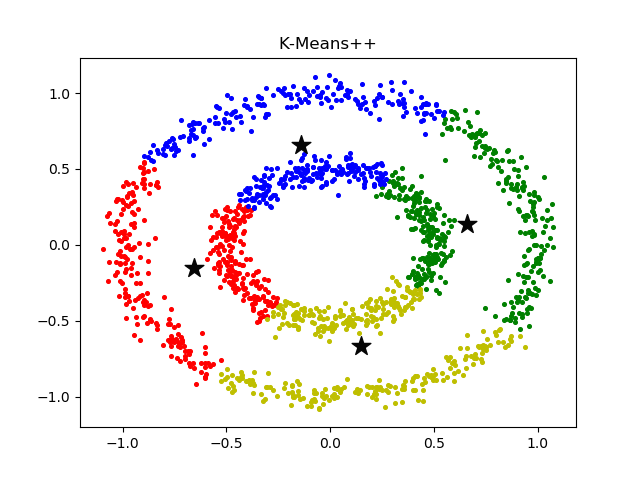


The results are shown below:

K=2: kmeans\_plus\_circle.gif, kmeans\_plus\_moon.gif



K=4: kmeans\_plus\_4k\_circle.gif, kmeans\_plus\_4k\_moon.gif



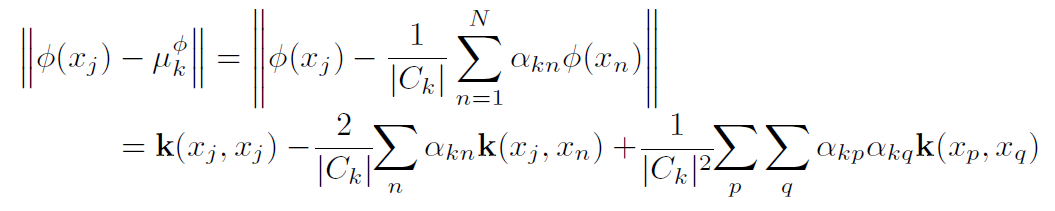
1. Discussion:

* 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.

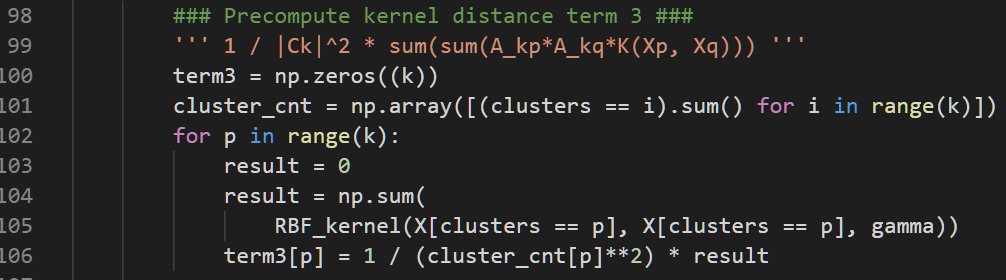
1. **Kernel K-means**
2. Implementation:

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:

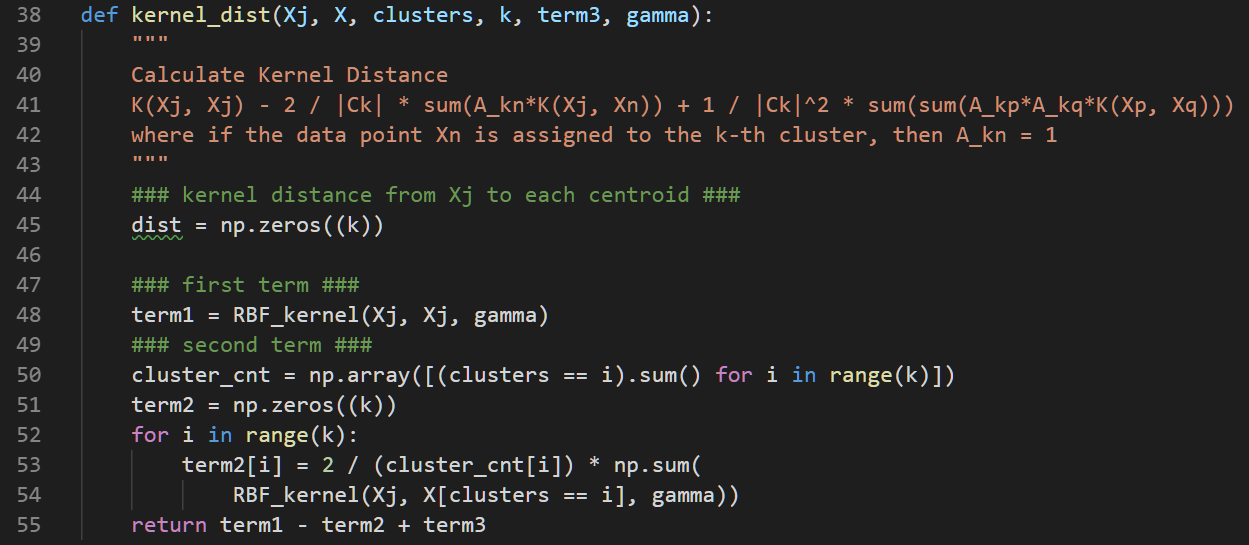


Firstly, precompute 3rd term (it doesn’t need to loop with Xj).



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

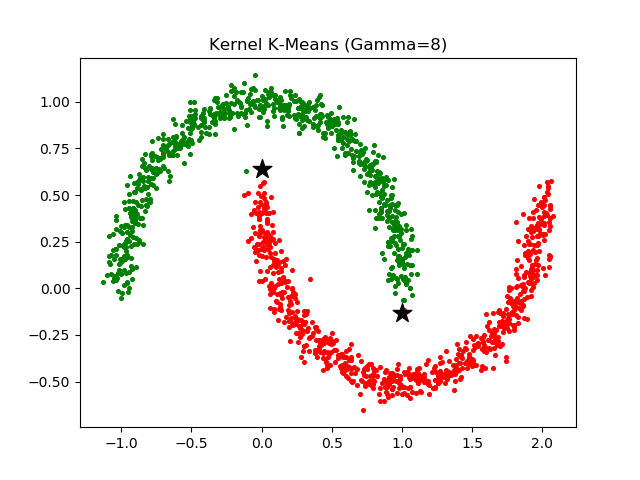
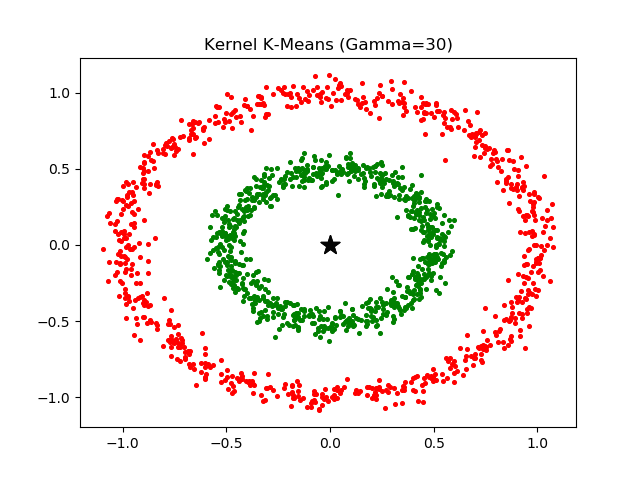
Combine these three terms to calculate kernel distance(term1-term2+term3).



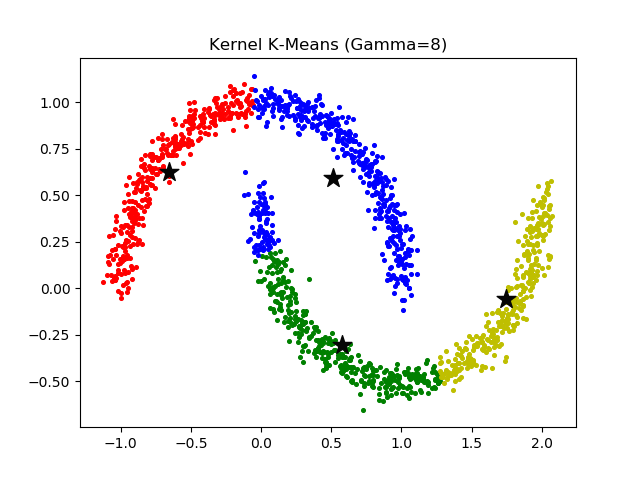
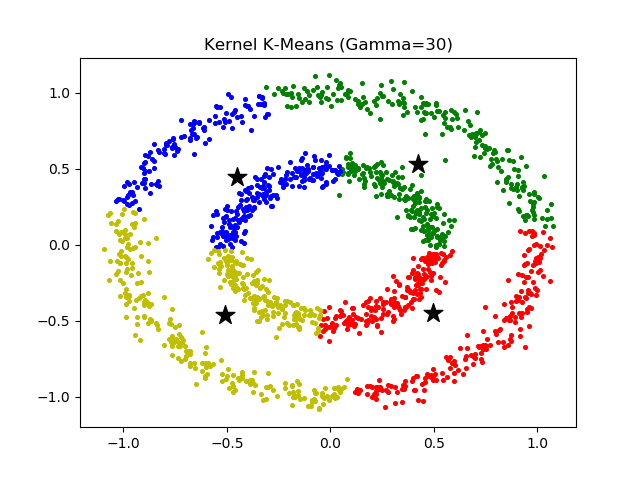


1. Result: (Here I only shows the best result of K-means++ initialization.)

K=2: kernel\_circle.gif, kernel\_moon.gif



K=4: kernel\_4k\_circle.gif, kernel\_4k\_moon.gif

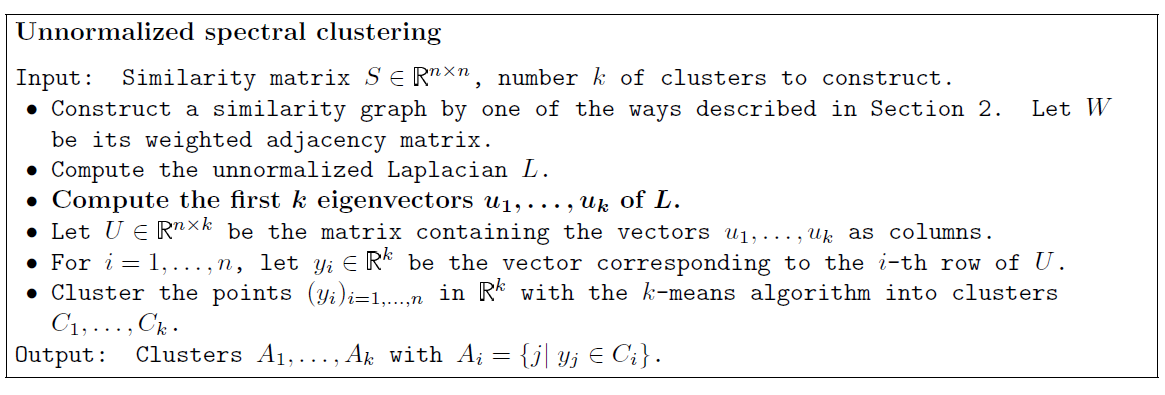


1. Discussion:

* 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.

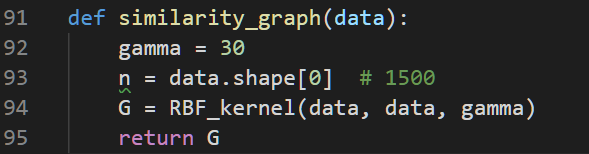
1. **Spectral clustering**
2. Implementation:

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

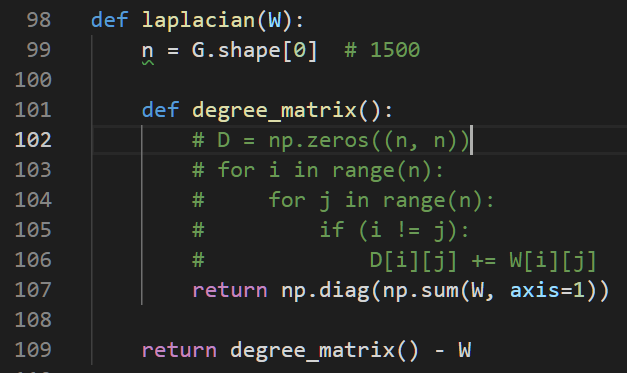


The implementation steps are as following:

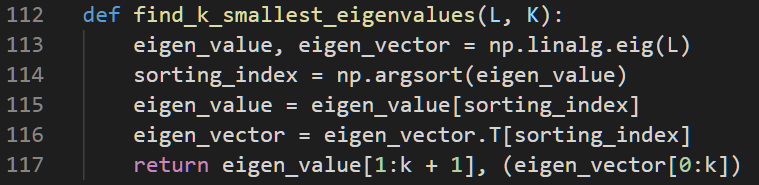
→ Construct a similarity graph (which stores kernel distances between each points)



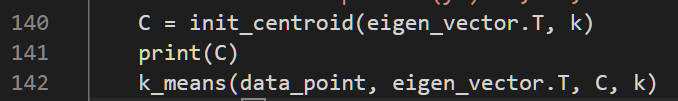
→ Compute the unnormalized Laplacian L: L = D – W, where D denotes degree matrix. (Here, W is the similarity graph)



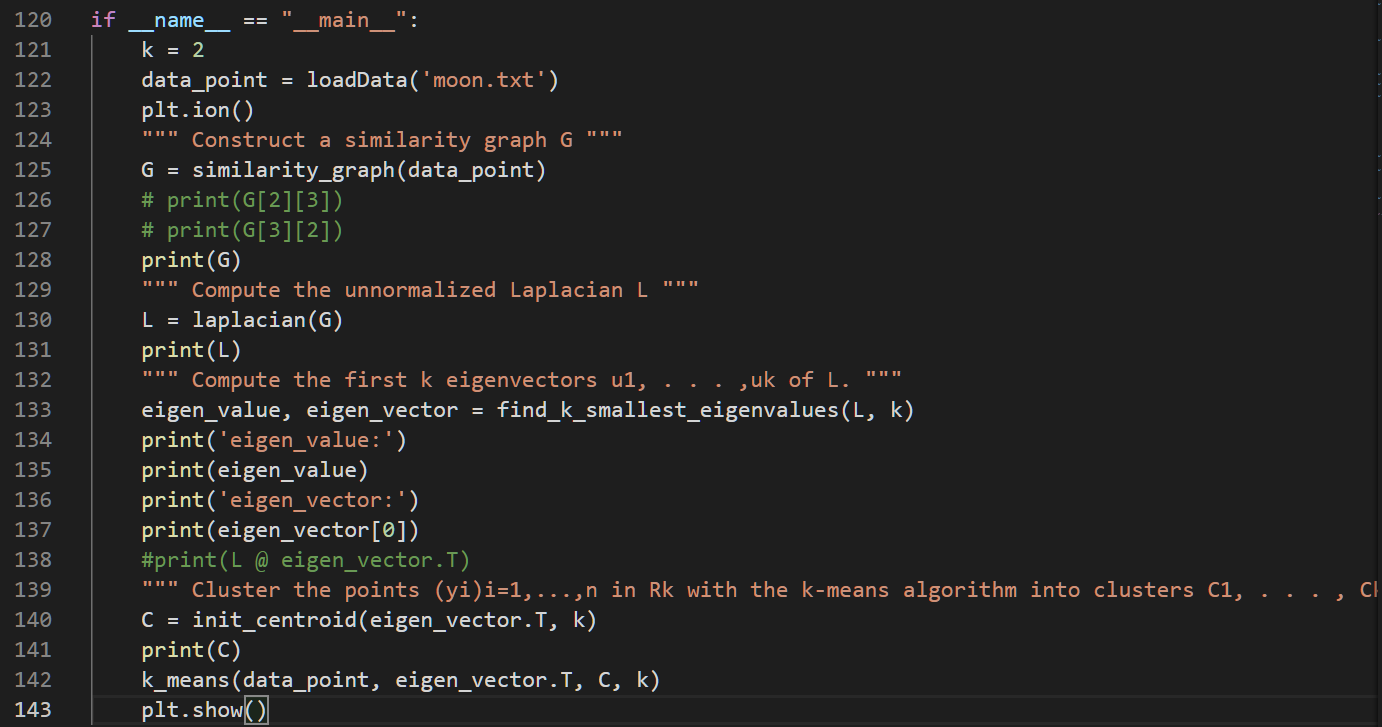
→ Compute the first k eigenvalues u1, u2, …, uk for L.



→ Cluster the points (yi)i=1,...,n in Rk with the k-means algorithm into clusters C1, . . . , Ck.

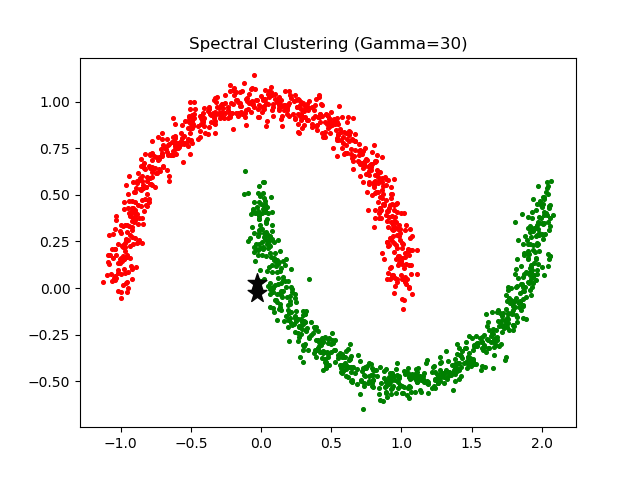
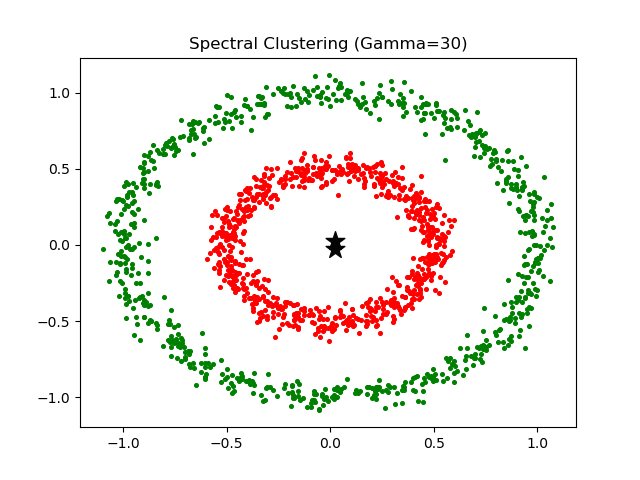


→ The main function is shown below.

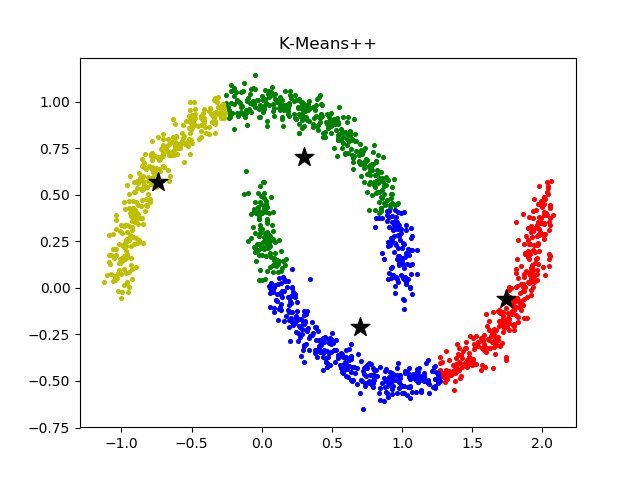
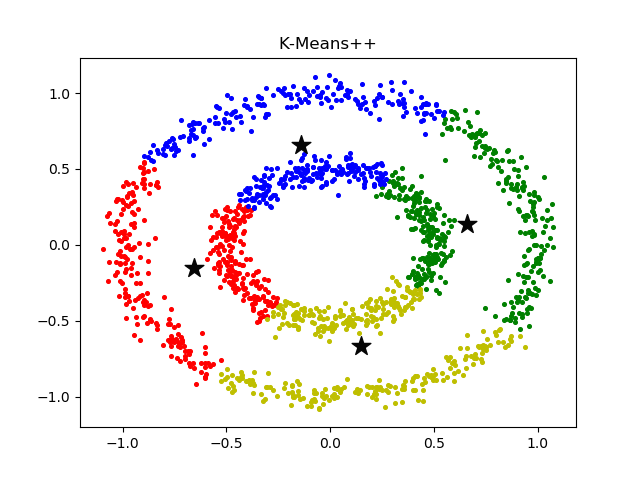


1. Result:

K=2: spectral\_circle.gif, spectral\_moon.gif



K=4: spectral\_4k\_circle.gif, spectral\_4k\_moon.gif

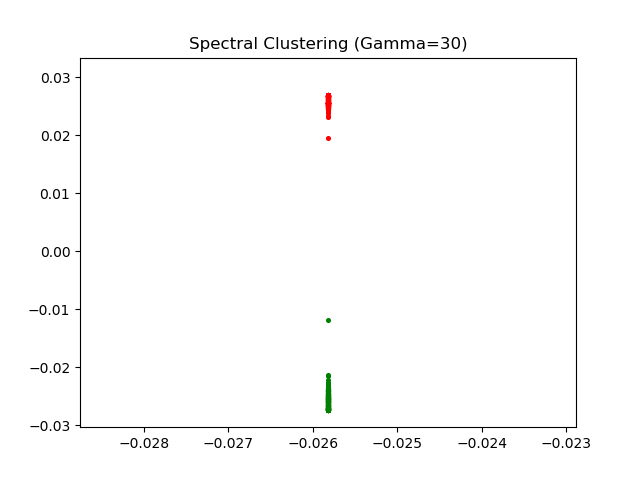
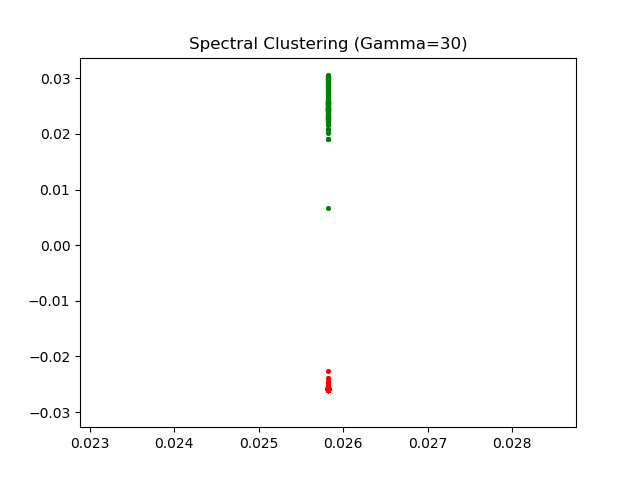


1. Eigenspace:

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)



1. 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.

1. **DBSCAN**
2. Implementation:

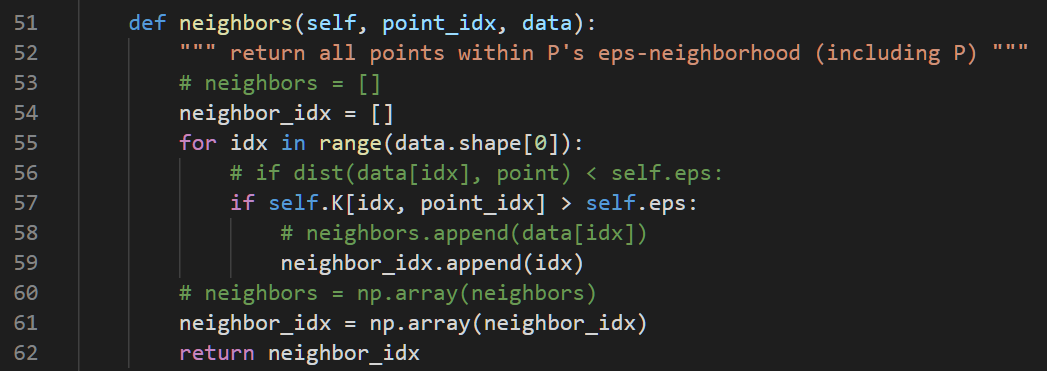
The pseudo code is as below: (Reference: [WIKI](https://zh.wikipedia.org/wiki/DBSCAN))

|  |
| --- |
| 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) |

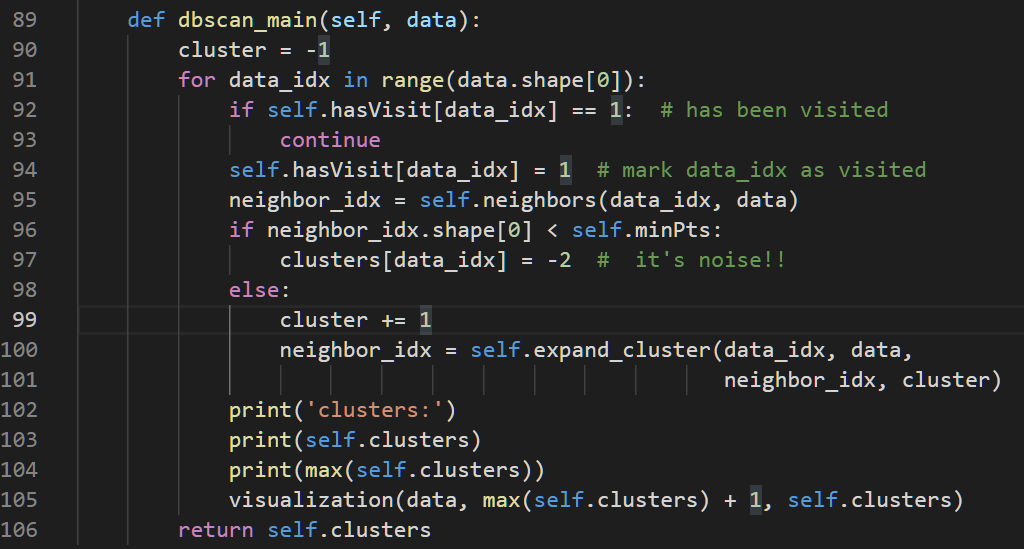
I declare a class DBSCAN to do dbscan clustering.

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.

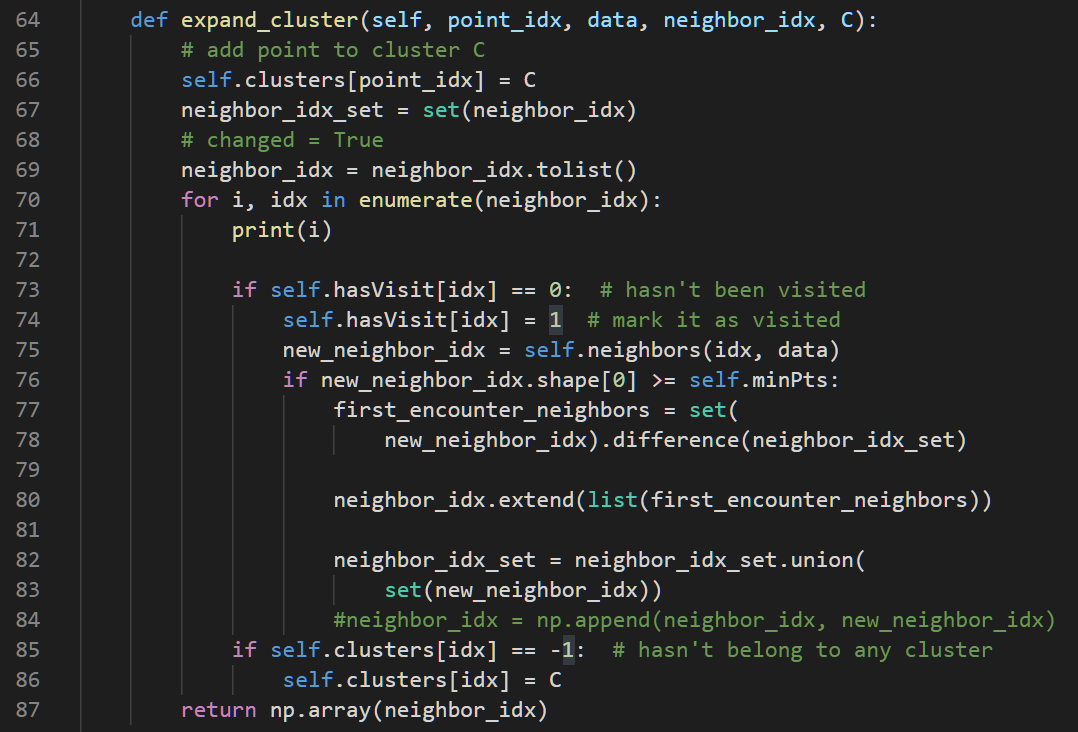


For main part, it checks every data point P in the data set to find out whether it is a core point. If it’s a core point, then merge all its neighbors into cluster a new cluster.



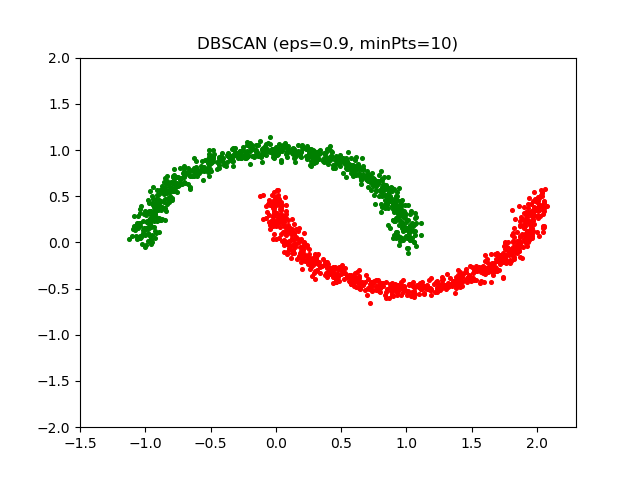
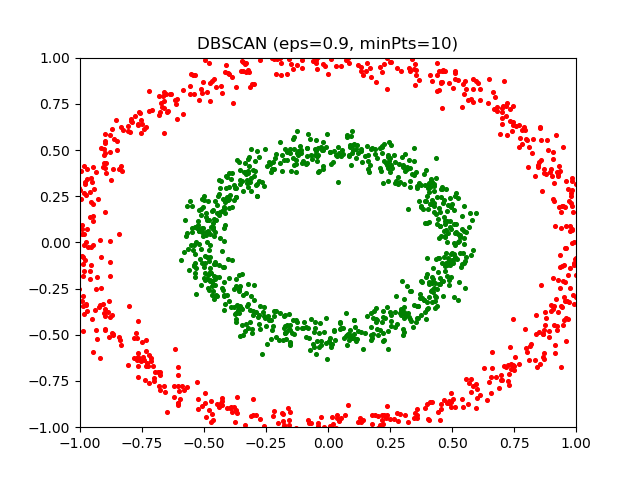
For expand\_cluster part, in addition to put P into cluster C, we also need to expand P’s neighbors, put them into cluster C, and check if they are core points or not. If P’s neighbor is a core point, then put P’s neighbor’s neighbors into P’s neighborhood.

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



1. Result:

dbscan\_circle.gif, dbscan\_moon.gif



1. 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.