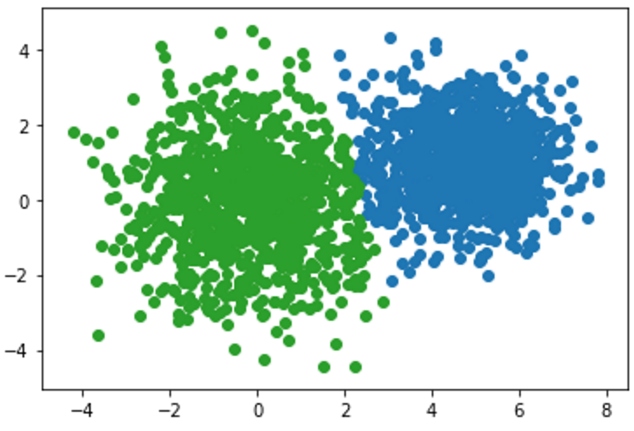
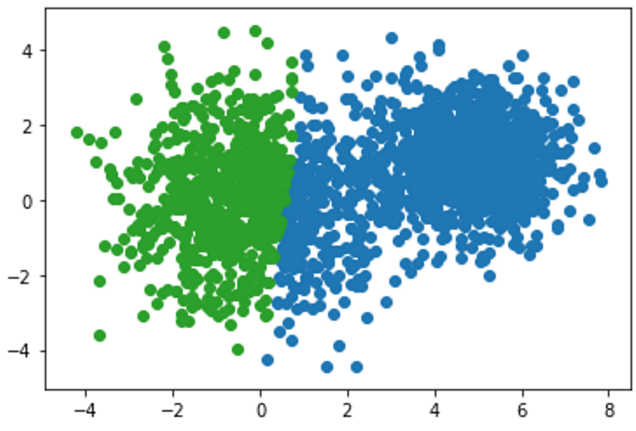
HW5 Zhonghao(Erik) Pan (zp3)

Sunday, November 24, 2019 7:11 PM

I. Write a basic implementation of Lloyd's algorithm for a large set of data in $\mathbf{R}^d$ (i.e., to find a Voronoi partition and a set of K centroids). Your algorithm should attempt to solve the classic *K-means* problem, for any user-selected positive integer value K.

- Assume the input data is given to you in a matrix $X \in \mathbf{R}^{N\times d}$, where each row in X corresponds to an observation of a d-dimensional point. That is, your inputs will be a user-provided matrix X and the number of clusters K.
- Your outputs should be (i) a matrix $Y \in \mathbf{R}^{K\times d}$, where row j contains the centroid of the j^{th} partition; (ii) a cluster index vector $C \in \{1, 2, \dots K\}^N$, where $C(i) = j$ indicates that the i^{th} row of X (or the i^{th} observation x_i) belongs to cluster j; and (iii) the final objective function value, i.e., the best distortion, or averaged distance value, D obtained.
- Convergence may be based on a norm-based comparison of the iterates of Y, i.e., $\|Y_{p+1} - Y_p\| < tol$, OR on a norm-based comparison of the distortion achieved $\|D_{p+1} - D_p\| < tol$. Choose tol to be (1) 1×10^{-5}, and (2) a different value of your choice, with your reasoning provided.

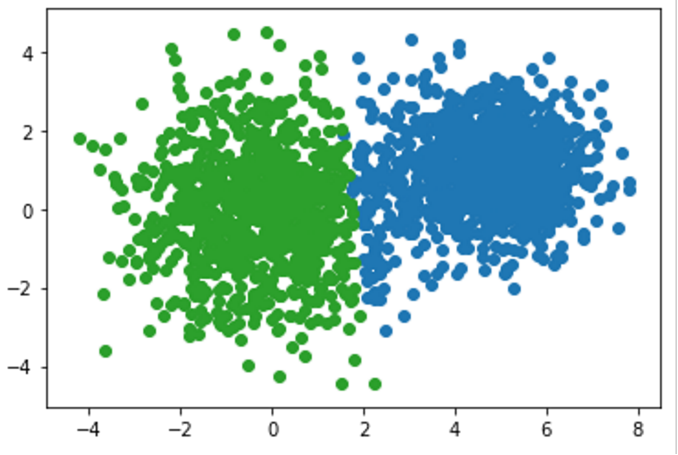
After observing the dataset, we will use k = 2, and let tol = 1e-5:

Lloyd's with good initialization	Lloyd's with bad initialization
	

Lloyd's produce acceptable result when run repeatedly. Even when the initialization is bad, the result is not as bad as that of greedy k centers.

Let tol = 1e-3:

Lloyd's with tol = 1e-3



With a smaller tol, the Lloyd's converge faster while not performing worse than a bad initialization of tol = 1e-5 above.

II.1 Write a basic implementation of the "GreedyKCenters" algorithm (described in the reading by S. Har-Peled, and discussed in class). Your algorithm should attempt to solve the classic *K-centers* problem, for any user-selected positive integer value K. The underlying distance function used in your algorithm should be the Euclidean distance, and your objective should be to *minimize* the *maximum* distance between any observation $x_i \in X$ and it's closest center $c_j \in Q$, i.e., to find Q giving

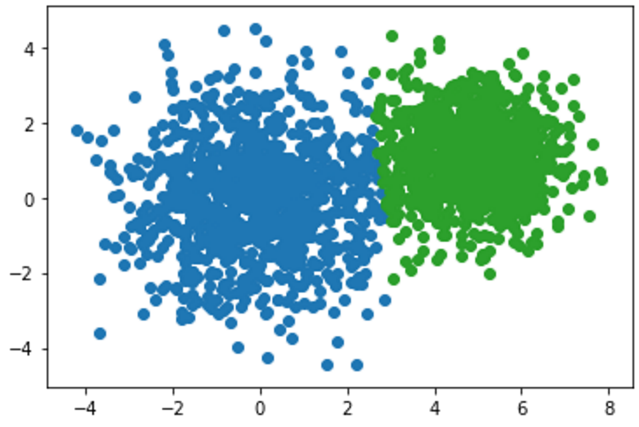
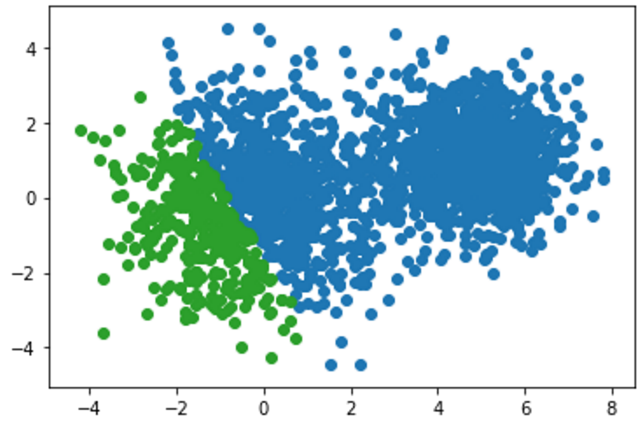
$$\min_{Q \subset X,\, |Q|=K} \left(\max_{x_i \in X} \left(\min_{c_j \in Q} \|x_i - c_j\|_2 \right) \right) \tag{1}$$

- You can again assume the input data is given to you as a matrix $X \in \mathbf{R}^{N \times d}$, and a positive integer K, as in **I.**

- Your output should be a matrix $Q \in \mathbf{R}^{K \times d}$ containing the final K d-dimensional centers, and the objective function value, i.e., the final $\max_{x_i \in X}(\min_{c_j \in Q} \|x_i - c_j\|_2)$ obtained.
- You do not need a convergence criteria for this algorithm.

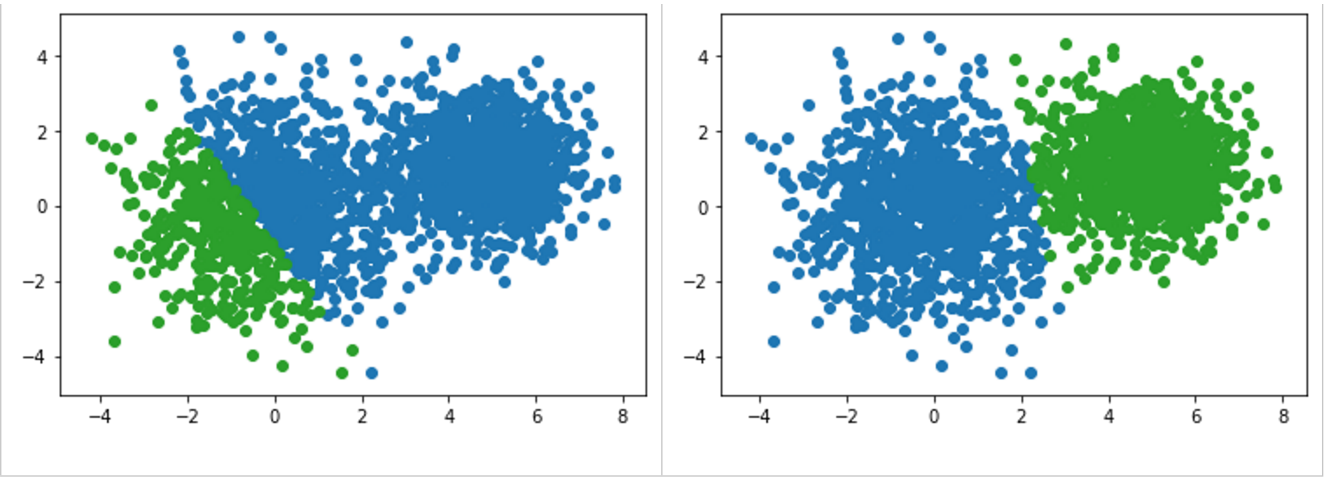
II.2 Write a basic implementation of the single-swap heuristic for which you try to improve the solution to the *K-centers* problem in **II.1** by a implementing a series of "swaps". If Q is your current set of centers, and you make a single swap, giving $Q_{new} = Q - \{c_j\} \cup \{o\}$, then you should replace Q with Q_{new} whenever the new objective value, that is the computed value for (1), is reduced by a factor of $(1 - \tau)$. When there is no swap that improves the solution by this factor, the local search stops. Let $\tau = 0.05$.

Let k = 2:

Greedy K Center with good initialization	Greedy K Center with bad initialization
	

Compare to Lloyd's, greedy heavily depends on a good initialization. Lloyd's will produce acceptable result most of the time, but greedy k center will rarely produce acceptable results.

Single Swap Before	Single Swap After



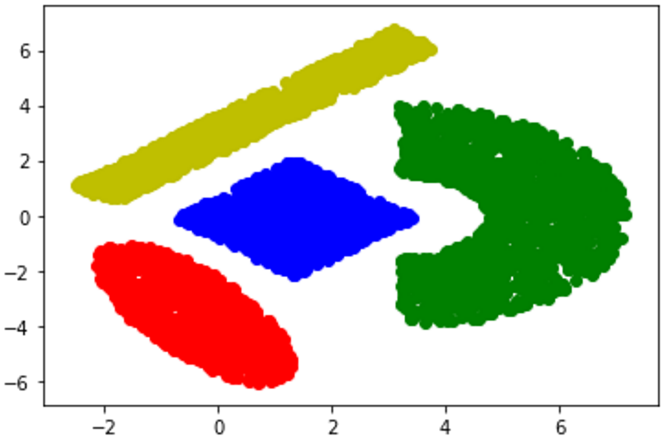
After applying single swap, the result is as good as that of running from a good initialization of greedy and Lloyd's.

III. Write an implementation of the Spectral Clustering algorithm, using either basic unnormalized clustering or normalized clustering (refer to the reading by Luxborg for details). Assume you are given a matrix of data $X \in \mathbf{R}^{N \times d}$, and you would like to identify some user-selected number of clusters, K. Your outputs should be:

- a weighted adjacency matrix, W, using the **Gaussian similarity function** based on the Euclidean distance (with parameter value σ of your choice but clearly stated) and a **k-nearest neighborhood structure** (where k is also your choice and clearly stated);
- a matrix U containing the first K eigenvectors of the Laplacian L (or generalized eigenvectors for the normalized case).;
- a cluster index vector $C \in \{1, 2, \ldots, K\}^N$, where $C(i) = j$ indicates that the ith row of U belongs to cluster j.

You should use your own *k-means/Lloyd's algorithm from Part I* to obtain the final cluster assignments, but you may want to also use an existing k-means algorithm (Matlab or Python, e.g.,) as well, depending on what you find in Part I.

Since Lloyd's is more robust than greedy K-center, we will use Lloyd's for spectral clustering. Let K = 4, sigma = 1, the result is shown below:



We can see the four shape is nicely captured by spectral clustering.

In [14]:

```
import numpy as np
import copy
import sys
import matplotlib.pyplot as plt

def Lloyd(k, N, d, m, dist, x, tol):
    def is_close(prev_m, m, tol):
        norm = 0
        for i in range(k):
            for j in range(d):
                norm += abs(m[i][j] - prev_m[i][j])
        return norm < tol

    prev_m = m
    obj_f = sum([sum([dist(x[i], m[j]) for i in range(N)]) for j in range(k)])
    t = 0
    c = [i for i in range(N)]
    while t == 0 or (is_close(prev_m, m, tol) and t < 1000):
        t += 1

        prev_m = copy.deepcopy(m)

        p = np.zeros([N,k])
        c_count = np.zeros(k)
        for i in range(N):
            assignment = np.argmin([dist(x[i],m[j]) for j in range(k)])
            p[i][assignment] = 1
            c[i] = assignment
            c_count[assignment] += 1

        m = np.array([1/c_count[j]*sum([x[i]*p[i][j] for i in range(N)]) for j
in range(k)])

    obj_f = sum([sum([dist(x[i], m[j]) for i in range(N)]) for j in range(k)])
    return (m, c, obj_f)
```

In [3]:

```
def GreedyKCenters(k, N, d, dist, x, c1):
    def two_norm(x, y):
        distance = 0
        for i in range(d):
            distance += (x[i] - y[i]) ** 2
        return distance ** 0.5
    def in_c (x, c):
        for i in c:
            if np.array_equal(x,i):
                return True
        return False
    c = [c1]
    while len(c) < k:
        distances = []
        for j in range(N):
            if not in_c (x[j], c):
                distances.append(sum([dist(i, x[j]) for i in c]))
            else:
                distances.append(0)

        xl = x[np.argmax(distances)]
        c.append(xl)
    obj_f = max([min([two_norm(i, j) for j in c]) for i in x])

    return (np.array(c), obj_f)
```

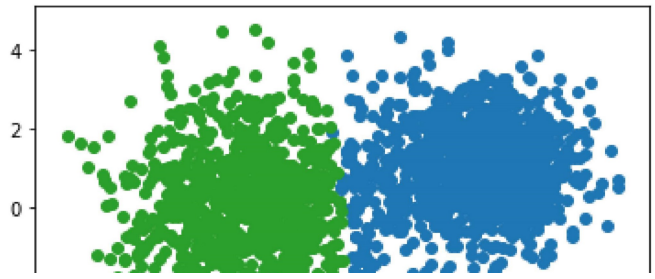
In [27]:

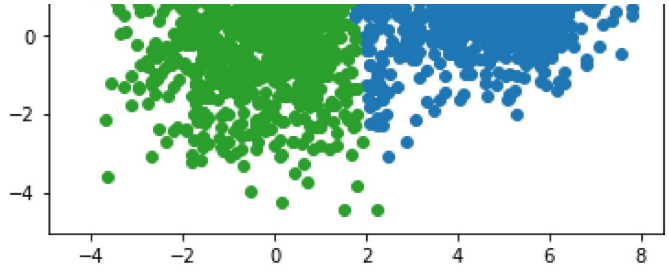
```
def SingleSwap(k, N, d, dist, x, c, obj_f, tau, tol):
    def two_norm(x, y):
        distance = 0
        for i in range(d):
            distance += (x[i] - y[i]) ** 2
        return distance ** 0.5
    def in_array (x, c):
        for i in c:
            if np.array_equal(x,i):
                return True
        return False

    p = np.zeros([N,k])
    for i in range(N):
        assignment = np.argmin([dist(x[i],c[j]) for j in range(k)])
        p[i][assignment] = 1
    cost = sum([sum([dist(x[i], c[j])*p[i][j] for i in range(N)]) for j in ran
ge(k)])

    t = 0
    while t < min(tol,N):
        t += 1

        x_exclude_c = [x[i] for i in range(N) if not in_array(x[i],c)]
        in_idx = np.random.choice([i for i in range(N - k)],1)[0]
        xi = x_exclude_c[in_idx]

        out_idx = np.random.choice([i for i in range(k)],1)[0]
        mj = c[out_idx]

        new_c = [xi]
        for i in range(k):
            if np.array_equal(c[i],mj):
                new_c.append(mj)

        p = np.zeros([N,k])
        for i in range(N):
            assignment = np.argmin([dist(x[i],new_c[j]) for j in range(k)])
            p[i][assignment] = 1
        new_cost = sum([sum([dist(x[i], new_c[j])*p[i][j] for i in range(N)])
for j in range(k)])

        if cost - 10**(1 - tau) > new_cost:
            c = new_c
            cost = new_cost
            t = 0

    obj_f = max([min([two_norm(i, j) for j in new_c]) for i in x])

    return (np.array(c), obj_f)
```

```
    return (np.array(c), obj_f)
```

In [28]:
```
import csv
import random


with open('C:/Users/ericp/Desktop/clustering.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    x = np.array(x).astype("float")

k = 2
N = len(x)
d = len(x[0])

def dist(p1,p2):
    distance = 0
    for i in range(len(p1)):
        distance += (p1[i] - p2[i])**2
    return distance
```

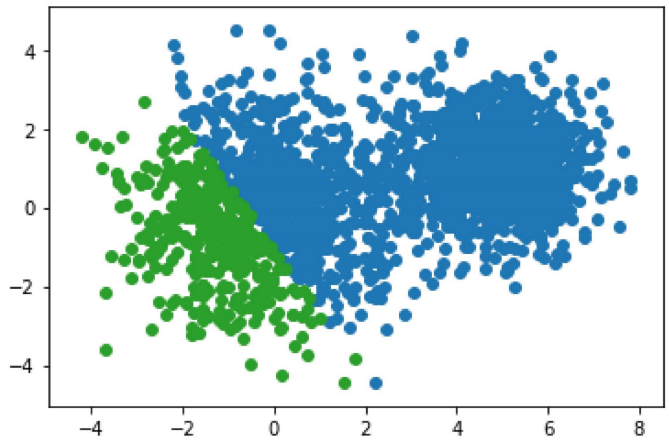
In [34]:
```
idxs = [i for i in range(N)]
random.shuffle(idxs)
idxs = idxs[:k]
m = []
for i in idxs:
    m.append(x[i])
m = np.array(m)

Lloyd_result = Lloyd(k, N, d, m, dist, x, 1e-3)

C = [np.argmin([dist(i, j) for j in Lloyd_result[0]]) for i in x]
fig, ax = plt.subplots()
colors = ['tab:blue', 'tab:green']
for i in range(k):
    for j in range(N):
        if C[j] == i:
            ax.scatter(x[j][0], x[j][1], c=colors[i])
plt.show()
```



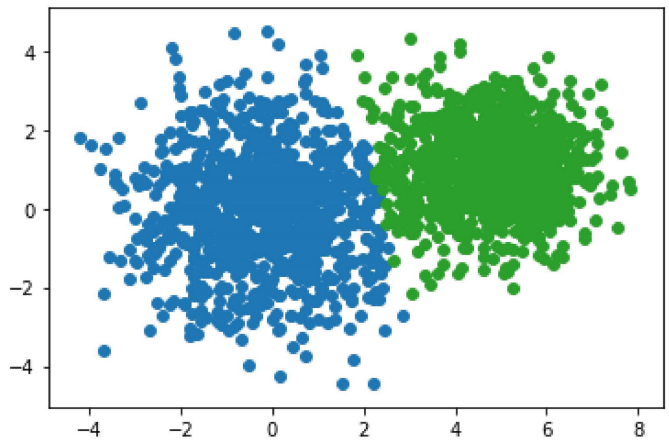


In [261]:

```
idxs = [i for i in range(N)]
random.shuffle(idxs)
idxs = idxs[:k]
m = []
for i in idxs:
    m.append(x[i])
m = np.array(m)

c1 = x[idxs[0]]
Greedyk_result = GreedyKCenters(k, N, d, dist, x, c1)

C = [np.argmin([dist(i, j) for j in Greedyk_result[0]]) for i in x]
fig, ax = plt.subplots()
colors = ['tab:blue', 'tab:green']
for i in range(k):
    for j in range(N):
        if C[j] == i:
            ax.scatter(x[j][0], x[j][1], c=colors[i])
plt.show()
```



In [292]:
```
SingleSwap_result = SingleSwap(k, N, d, dist, x, Greedyk_result[0], Greedyk_re
sult[1], 0.05, 100)
C = [np.argmin([dist(i, j) for j in SingleSwap_result[0]]) for i in x]
fig, ax = plt.subplots()
colors = ['tab:blue', 'tab:green']
for i in range(k):
    for j in range(N):
        if C[j] == i:
            ax.scatter(x[j][0], x[j][1], c=colors[i])
plt.show()
```



In [147]:

```
def Spectral(k, N, d, dist, x, sigma):
    def two_norm(x, y):
        distance = 0
        for i in range(len(x)):
            distance += (x[i] - y[i]) ** 2
        return distance ** 0.5

    W = np.zeros([N, N])
    for i in range(N):
        '''
        for j in range(N):
            W[i][j] = dist(x[i], x[j], sigma) #fully connected graph
        '''
        distances = sorted([(dist(x[i], x[j], sigma), j) for j in range(N)], k
ey=lambda x:x[0])[::-1][1:k+1]
        for point in distances:
            W[i][point[1]] = 1
            W[point[1]][i] = 1

    D = np.diag([sum([j for j in i]) for i in W])
    L = np.add(D, -W)

    eigenValues, eigenVectors = np.linalg.eig(L)

    idx = eigenValues.argsort()
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]

    U = eigenVectors[:,:k]
    y = np.array([i for i in U])

    idxs = [i for i in range(N)]
    random.shuffle(idxs)
```

```
    idxs = [i for i in range(N)]
    random.shuffle(idxs)
    idxs = idxs[:k]
    m = []
    for i in idxs:
        m.append(y[i])
    m = np.array(m)

    good_start_idx = [0,1,14,25]
    m = np.array([y[i] for i in good_start_idx])

    Lloyd_result = Lloyd(k, len(y), len(y[0]), m, two_norm, y, 1e-5)

    C = [np.argmin([two_norm(i, j) for j in Lloyd_result[0]]) for i in y]

    return (W, U, C)
```

In [148]:

```
import csv
import random

with open('C:/Users/ericp/Desktop/ShapedData.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    x = np.array(x).astype("float")

k = 4
N = len(x)
d = len(x[0])
sigma = 1

def dist(p1,p2,sigma):
    distance = 0
    for i in range(len(p1)):
        distance += (p1[i] - p2[i])**2
    return np.exp(-distance/(2*sigma**2))

Spectral_result = Spectral(k, N, d, dist, x, sigma)

fig, ax = plt.subplots()
colors = ['b','g','r','y']
for i in range(k):
    for j in range(N):
        if Spectral_result[2][j] == i:
            ax.scatter(x[j][0], x[j][1], c=colors[i])
plt.show()
```

```
        if Spectral_result[2][j] == i:
            ax.scatter(x[j][0], x[j][1], c=colors[i])
plt.show()
```