# Programming Assignment 2: Clustering

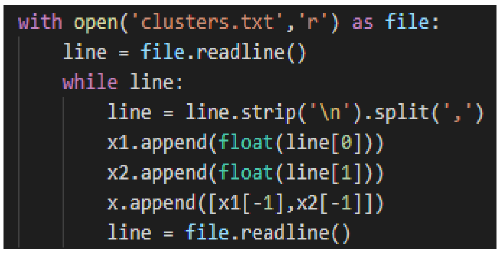
1. Group member

   a) Hao Yang, USCID: 5284267375. (Contributions: All)

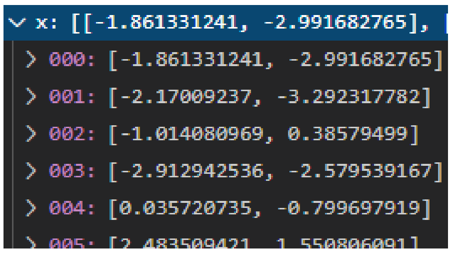
2. Part 1: Implementation

   1. Programming language: Python 3.7
   2. Libraries used: math, copy, random, numpy, matplotlib.pyplot
   3. Clustering algorithm: K-means algorithm & GMM.
   4. Script description:

      1. Data processing: read txt file, save points in a list of pairs. Save as like x = [[0,0], [1,1], [2,2], …, [$x_{n1}$, $x_{n2}$]].

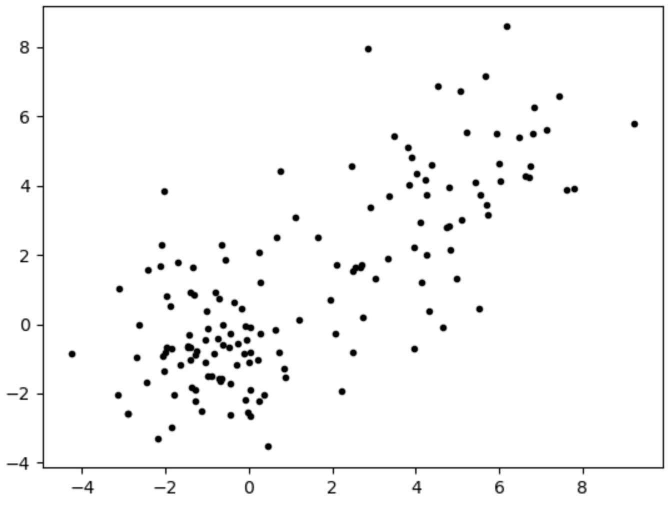


```
with open('clusters.txt','r') as file:
    line = file.readline()
    while line:
        line = line.strip('\n').split(',')
        x1.append(float(line[0]))
        x2.append(float(line[1]))
        x.append([x1[-1],x2[-1]])
        line = file.readline()
```



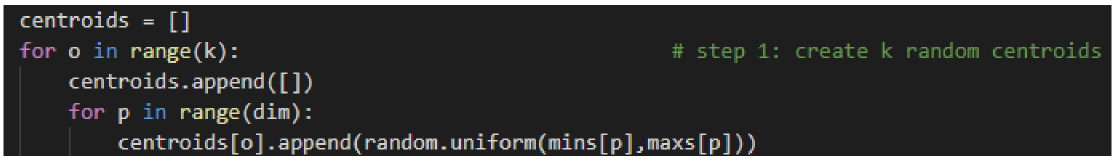
```
∨ x: [[-1.861331241, -2.991682765],
  > 000: [-1.861331241, -2.991682765]
  > 001: [-2.17009237, -3.292317782]
  > 002: [-1.014080969, 0.38579499]
  > 003: [-2.912942536, -2.579539167]
  > 004: [0.035720735, -0.799697919]
  > 005: [2.483509421, 1.550806091]
```

Data points distribution:



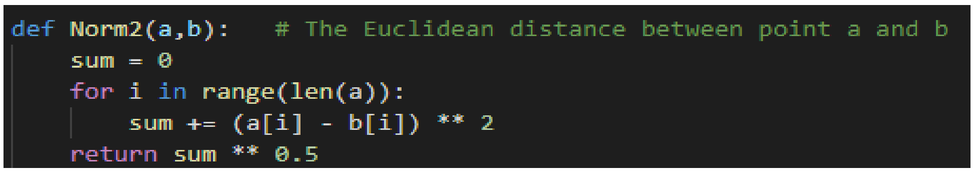
      2. K-means:

Step 1: create k random centroids. Firstly, get the range of all points. Then generate k random pairs in this range by random.uniform().



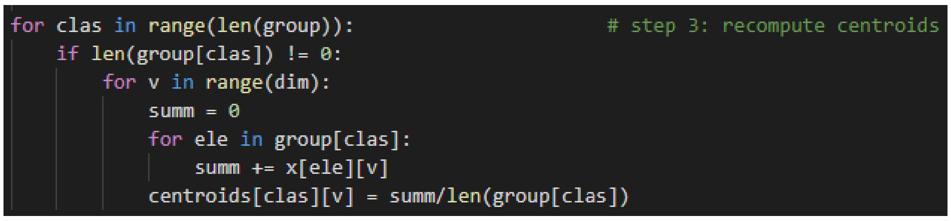
```
centroids = []
for o in range(k):                                    # step 1: create k random centroids
    centroids.append([])
    for p in range(dim):
        centroids[o].append(random.uniform(mins[p],maxs[p]))
```

Step 2: assign all points to closest centroids. Use Euclidean distance to compute distance.

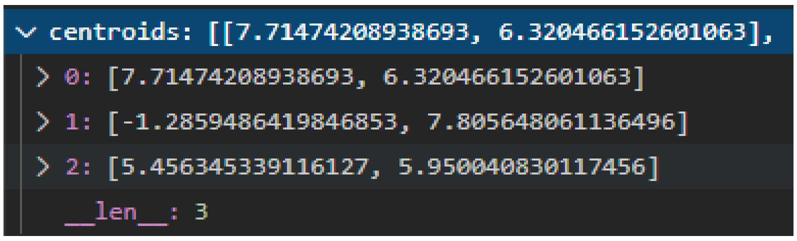


```
def Norm2(a,b):   # The Euclidean distance between point a and b
    sum = 0
    for i in range(len(a)):
        sum += (a[i] - b[i]) ** 2
    return sum ** 0.5
```

Step 3: recompute the new centroids by computing the center of points belong to that centroid.

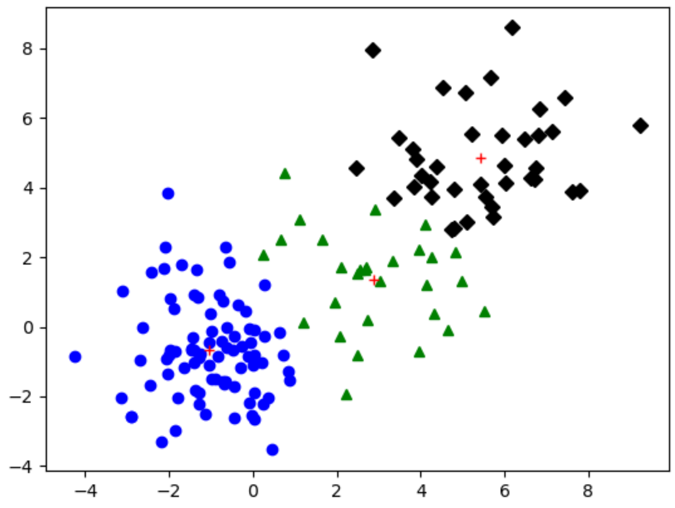


```
for clas in range(len(group)):                         # step 3: recompute centroids
    if len(group[clas]) != 0:
        for v in range(dim):
            summ = 0
            for ele in group[clas]:
                summ += x[ele][v]
            centroids[clas][v] = summ/len(group[clas])
```



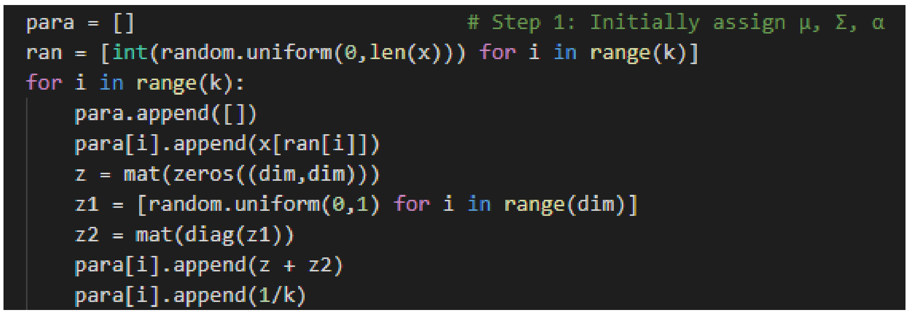
```
∨ centroids: [[7.71474208938693, 6.320466152601063],
  > 0: [7.71474208938693, 6.320466152601063]
  > 1: [-1.2859486419846853, 7.805648061136496]
  > 2: [5.456345339116127, 5.950040830117456]
    __len__: 3
```

Step 4: Repeat step 2 & step 3 until centroids don't change.



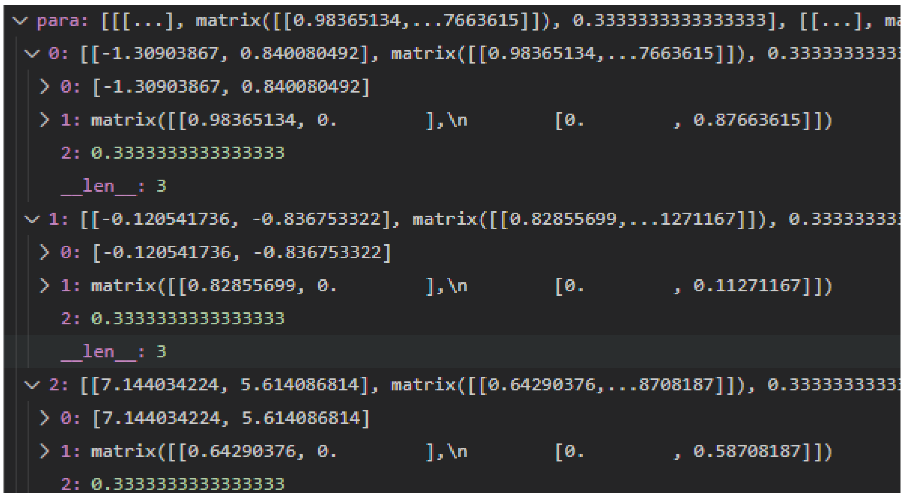
3. GMM:

Step 1: Randomly assign initial $\mu_c$, $\Sigma_c$, $\pi_c$. Assign $\mu_c$ as a random $x_i$ to avoid the initial $\mu$ locate too far away from data points. Assign $\Sigma_c$ as a small diagonal matrix. Assign $\pi_c$ evenly to all clusters.



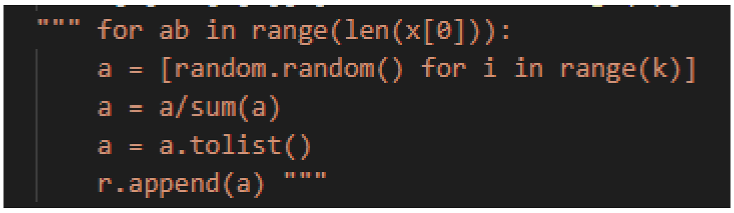
```
para = []                               # Step 1: Initially assign μ, Σ, α
ran = [int(random.uniform(0,len(x))) for i in range(k)]
for i in range(k):
    para.append([])
    para[i].append(x[ran[i]])
    z = mat(zeros((dim,dim)))
    z1 = [random.uniform(0,1) for i in range(dim)]
    z2 = mat(diag(z1))
    para[i].append(z + z2)
    para[i].append(1/k)
```

Got the initial parameter set (with $\mu_c$, $\Sigma_c$, $\pi_c$ for all k clusters.)



```
∨ para: [[[...], matrix([[0.98365134,...7663615]]), 0.3333333333333333], [[...], ma
  ∨ 0: [[-1.30903867, 0.840080492], matrix([[0.98365134,...7663615]]), 0.33333333333
    > 0: [-1.30903867, 0.840080492]
    > 1: matrix([[0.98365134, 0.        ],\n        [0.        , 0.87663615]])
      2: 0.3333333333333333
      __len__: 3
  ∨ 1: [[-0.120541736, -0.836753322], matrix([[0.82855699,...1271167]]), 0.333333333
    > 0: [-0.120541736, -0.836753322]
    > 1: matrix([[0.82855699, 0.        ],\n        [0.        , 0.11271167]])
      2: 0.3333333333333333
      __len__: 3
  ∨ 2: [[7.144034224, 5.614086814], matrix([[0.64290376,...8708187]]), 0.33333333333
    > 0: [7.144034224, 5.614086814]
    > 1: matrix([[0.64290376, 0.        ],\n        [0.        , 0.58708187]])
      2: 0.3333333333333333
```

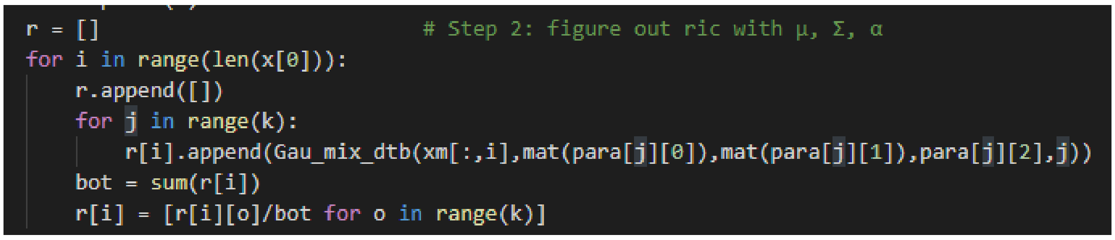
Or normally initialize $r_{ic}$ to start:



```
""" for ab in range(len(x[0])):
        a = [random.random() for i in range(k)]
        a = a/sum(a)
        a = a.tolist()
        r.append(a) """
```

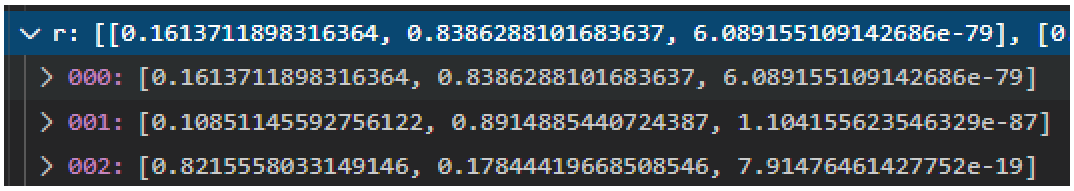
After multiple times of tests. I found out that starting with initialization of $\mu_c$, $\Sigma_c$, $\pi_c$ performs better than with that of $r_{ic}$, because the starting point will closer to the convergent result. So the report below will follow with initialization with $\mu_c$, $\Sigma_c$, $\pi_c$.

Step 2: Figure out new $r_{ic}$:



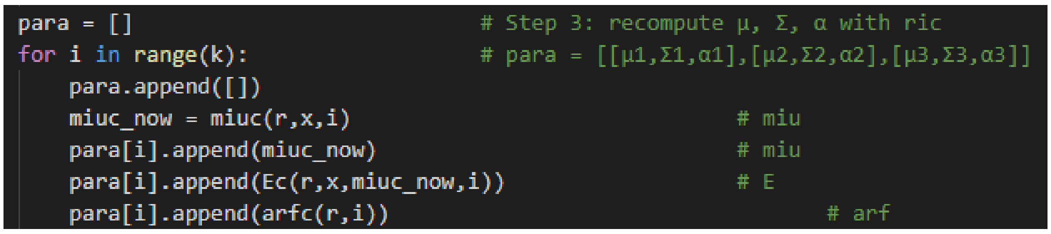
```
r = []                          # Step 2: figure out ric with μ, Σ, α
for i in range(len(x[0])):
    r.append([])
    for j in range(k):
        r[i].append(Gau_mix_dtb(xm[:,i],mat(para[j][0]),mat(para[j][1]),para[j][2],j))
    bot = sum(r[i])
    r[i] = [r[i][o]/bot for o in range(k)]
```

Result:



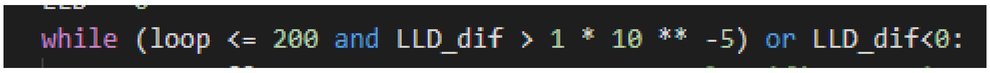
```
∨ r: [[0.1613711898316364, 0.8386288101683637, 6.089155109142686e-79], [0
  > 000: [0.1613711898316364, 0.8386288101683637, 6.089155109142686e-79]
  > 001: [0.10851145592756122, 0.8914885440724387, 1.104155623546329e-87]
  > 002: [0.8215558033149146, 0.17844419668508546, 7.91476461427752e-19]
```

Step 3: Recompute new $\mu_c$, $\Sigma_c$, $\pi_c$ with new $r_{ic}$.



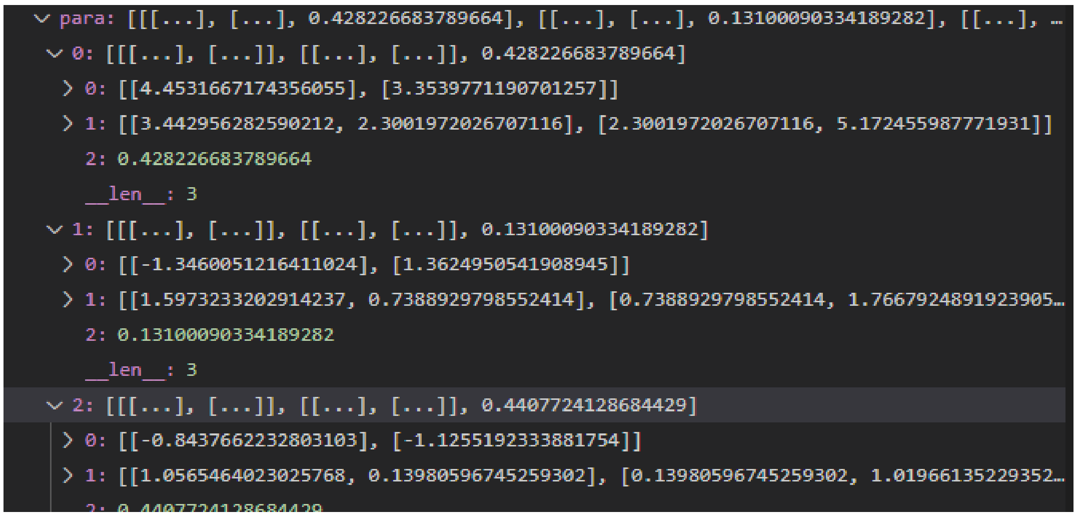
```
para = []                         # Step 3: recompute μ, Σ, α with ric
for i in range(k):                # para = [[μ1,Σ1,α1],[μ2,Σ2,α2],[μ3,Σ3,α3]]
    para.append([])
    miuc_now = miuc(r,x,i)                          # miu
    para[i].append(miuc_now)                        # miu
    para[i].append(Ec(r,x,miuc_now,i))              # E
    para[i].append(arfc(r,i))                             # arf
```

Iteratively do Step 2 and 3, and record the likelihood estimation function of each loop. Do until loop achieve a certain time or the LLD increase slowly.



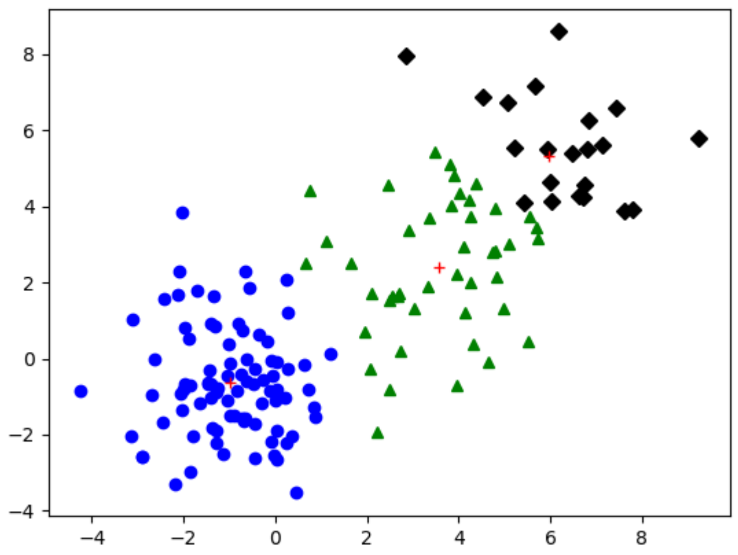
```
while (loop <= 200 and LLD_dif > 1 * 10 ** -5) or LLD_dif<0:
```

The result of $\mu_c$, $\Sigma_c$, $\pi_c$ set:



```
∨ para: [[[...], [...], 0.428226683789664], [[...], [...], 0.13100090334189282], [[...], …
  ∨ 0: [[[...], [...]], [[...], [...]], 0.428226683789664]
    > 0: [[4.4531667174356055], [3.3539771190701257]]
    > 1: [[3.442956282590212, 2.3001972026707116], [2.3001972026707116, 5.172455987771931]]
      2: 0.428226683789664
      __len__: 3
  ∨ 1: [[[...], [...]], [[...], [...]], 0.13100090334189282]
    > 0: [[-1.3460051216411024], [1.3624950541908945]]
    > 1: [[1.5973233202914237, 0.7388929798552414], [0.7388929798552414, 1.7667924891923905…
      2: 0.13100090334189282
      __len__: 3
  ∨ 2: [[[...], [...]], [[...], [...]], 0.4407724128684429]
    > 0: [[-0.8437662232803103], [-1.1255192333881754]]
    > 1: [[1.0565464023025768, 0.13980596745259302], [0.13980596745259302, 1.01966135229352…
      2: 0.4407724128684429
```

Result of clustering:



5. Optimizations

   1. K-means: Used k as the part of parameter of the function instead of 3, and compute the dimension of the dataset first instead of using 2 directly. These can help my function can use in general case with different k and different size of data instead of just can be used in this assignment.

   2. GMM:

      1. Start initialization with $\mu_c$, $\Sigma_c$, $\pi_c$ instead of $r_{ic}$ and initialize $\mu_c$ as a random $x_i$ instead of a random point to avoid a certain $\mu_c$ locates at far away of all data points to improve the speed of convergence.

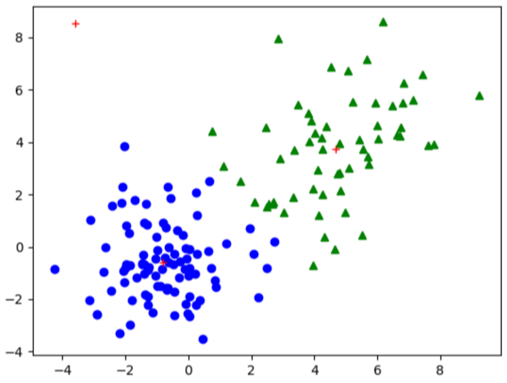
      2. Stop loop with not only times of loop but also value of LLD which means the extent of improvement is small, so that we can see it as it's around the convergent point and no need to run all loops set to save time.

6. Challenges faced

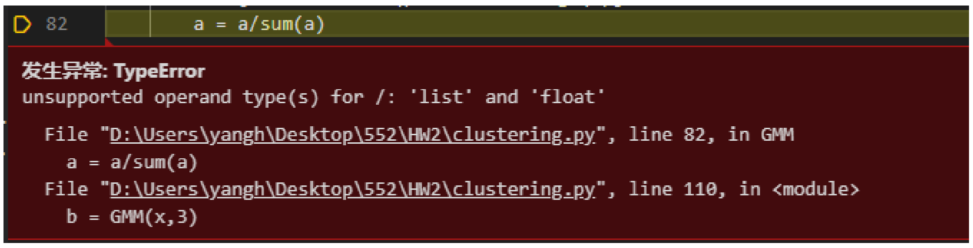
   1. K-means: Found out that because of the random pick of initial centroids. In

some case the points will just be assign to two centroids which means just be clustered into 2 group. Therefore, fixed it by setting if some groups are empty, restart the centroids initialization step to ensure points are assign into k groups.

Example:



2. GMM: When use calculation signs to deal with list of variable, python don't know whether we want to calculate the element of list or just merge the list. I found out that we can import * from numpy to solve this error.

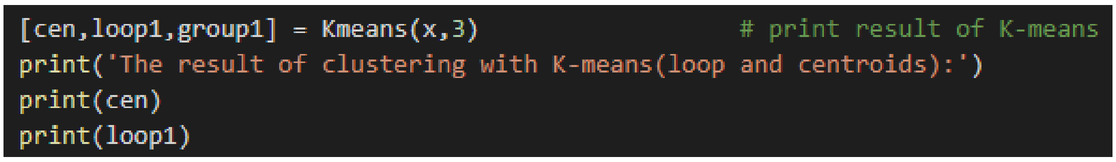


```
a = a/sum(a)
发生异常: TypeError
unsupported operand type(s) for /: 'list' and 'float'
  File "D:\Users\yangh\Desktop\552\HW2\clustering.py", line 82, in GMM
    a = a/sum(a)
  File "D:\Users\yangh\Desktop\552\HW2\clustering.py", line 110, in <module>
    b = GMM(x,3)
```

7. Result

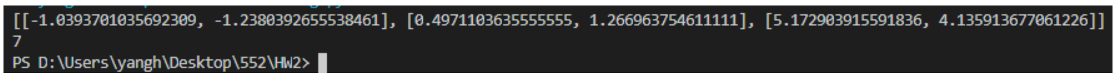
1. K-means:

Run Kmeans(dataset, # cluster) function with dataset x above:



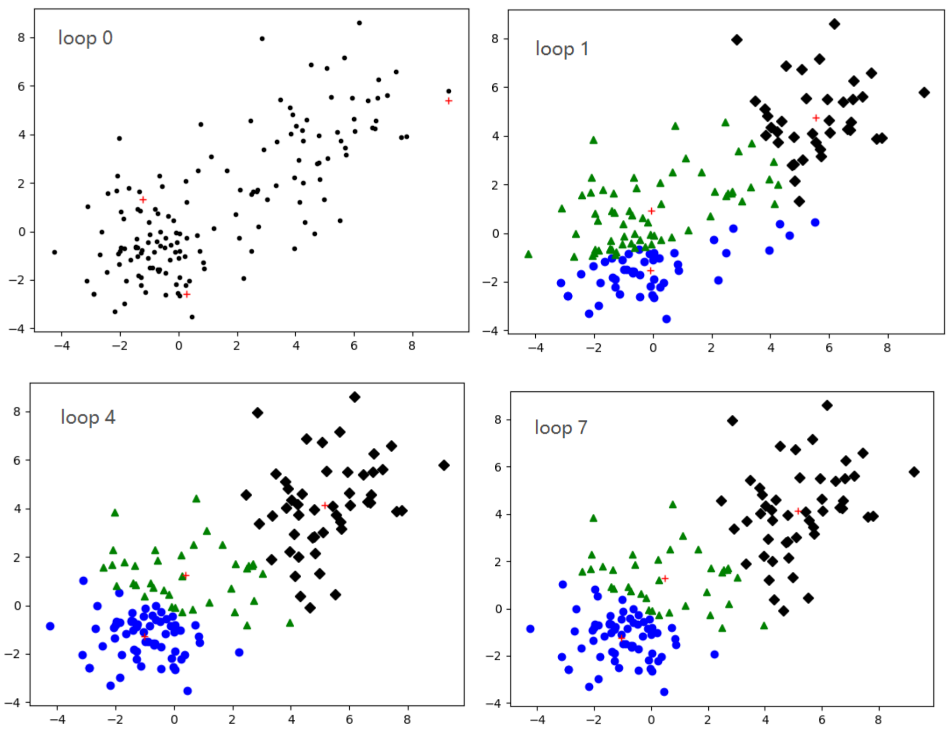
```
[cen,loop1,group1] = Kmeans(x,3)              # print result of K-means
print('The result of clustering with K-means(loop and centroids):')
print(cen)
print(loop1)
```

Print the result of centroids and times of loop:



```
[[-1.0393701035692309, -1.2380392655538461], [0.4971103635555555, 1.266963754611111], [5.172903915591836, 4.135913677061226]]
7
PS D:\Users\yangh\Desktop\552\HW2>
```

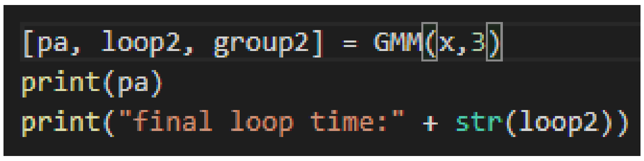
Loop:



P.S. the result of centroids and times of loop would be different based on the initial centroids choose. I've implemented 500 times of this function and there are about 9 different centroids output and the loop time are between 4 to 15.

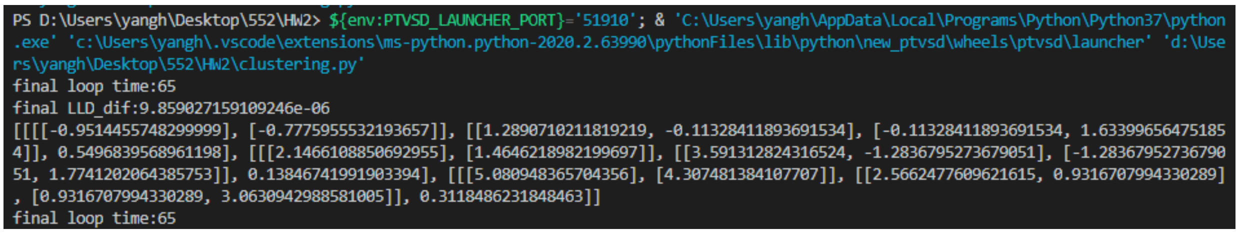
2. GMM:

Run GMM(dataset, # cluster) function with dataset x above:



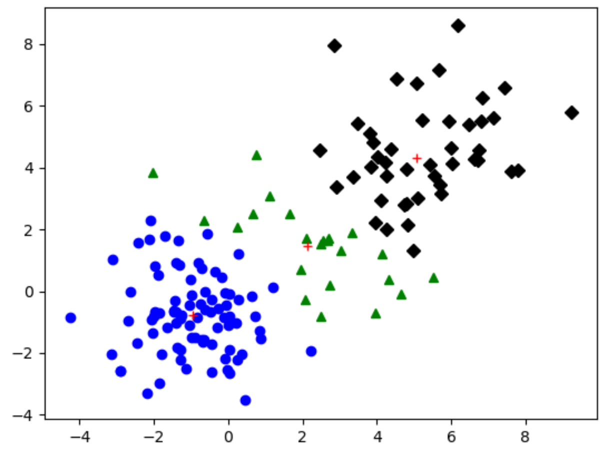
```
[pa, loop2, group2] = GMM(x,3)
print(pa)
print("final loop time:" + str(loop2))
```

Print the result of $\mu_c$, $\Sigma_c$, $\pi_c$ and times of loop:



```
PS D:\Users\yangh\Desktop\552\HW2> ${env:PTVSD_LAUNCHER_PORT}='51910'; & 'C:\Users\yangh\AppData\Local\Programs\Python\Python37\python
.exe' 'c:\Users\yangh\.vscode\extensions\ms-python.python-2020.2.63990\pythonFiles\lib\python\new_ptvsd\wheels\ptvsd\launcher' 'd:\Use
rs\yangh\Desktop\552\HW2\clustering.py'
final loop time:65
final LLD_dif:9.859027159109246e-06
[[[[-0.9514455748299999], [-0.7775955532193657]], [[1.2890710211819219, -0.11328411893691534], [-0.11328411893691534, 1.63399656475185
4]], 0.5496839568961198], [[[2.1466108850692955], [1.4646218982199697]], [[3.591312824316524, -1.2836795273679051], [-1.28367952736790
51, 1.7741202064385753]], 0.13846741991903394], [[[5.080948365704356], [4.307481384107707]], [[2.5662477609621615, 0.9316707994330289]
, [0.9316707994330289, 3.0630942988581005]], 0.31184862318484463]]
final loop time:65
```

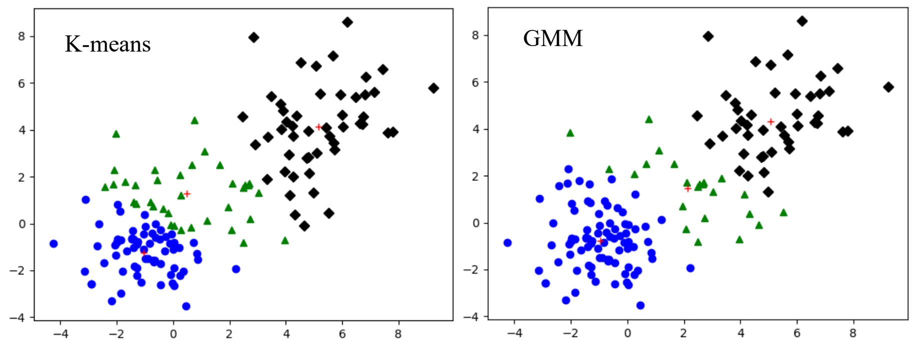
The final clustering image:



P.S. the result above is one of the possible results of clustering. There are three kinds of different clustering based on the initialization of parameter in my 30 times of tests. (Can only do less times of tests than k-means because GMM takes a much more time to implement.)

3. Comparison:

    1. The comparison of clustering is basically same.



    2. Loop time comparison: The number of loops of K-means is around 4 to 15. The number of loops of GMM is around 30 to 120. Both are based on the initial parameter pick.
    3. Other comparison: GMM need more calculation so takes more loops and more time, but it can deal with problems with overlapped cluster.

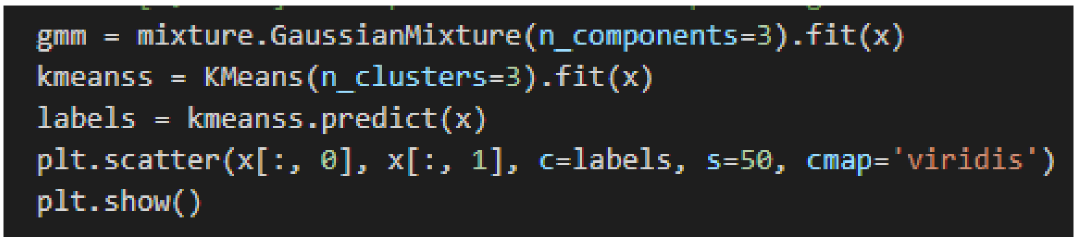
8. What I found:

I found out that in many article, they use the result of K-means(the centroids and calculate Σ with centroids as the mean) as the initial value of GMM. And I viewed many forums, many people said that this does improve the efficiency of GMM in not only speed but also can converge to a better clustering result. But I don't use the result of K-means as my initialization in GMM because I need to check if my algorithm works correctly rather than depended on "wonderful start points".

Therefore, I have many different result outputs from my implement. I just pick one which I think is the best to put in this report. :)

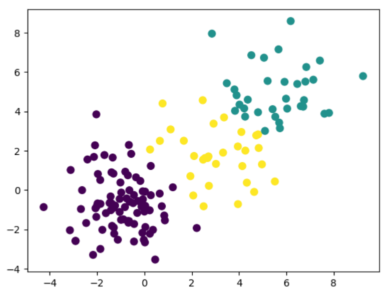
3. Part 2: Software familiarization

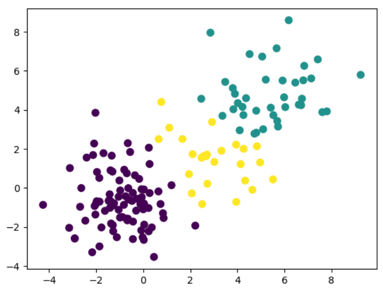
   1. Library function: sklearn.mixture.GaussianMixture and sklearn.cluster.KMeans
   2. How to use: use GaussianMixture(n_components=3).fit(x) and KMeans(n_clusters=3).fit(x) to implement GMM and Kmeans to the dataset x. Then use plt.scatter and plt.show the show the result of clustering.



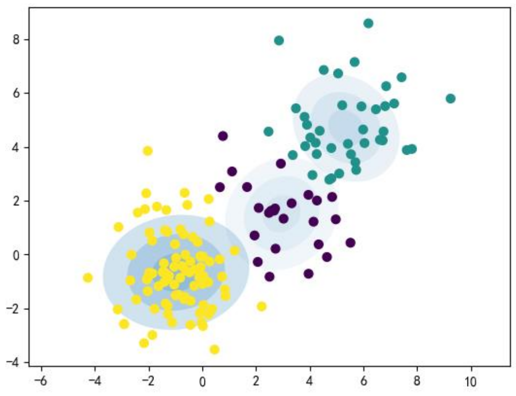
```
gmm = mixture.GaussianMixture(n_components=3).fit(x)
kmeanss = KMeans(n_clusters=3).fit(x)
labels = kmeanss.predict(x)
plt.scatter(x[:, 0], x[:, 1], c=labels, s=50, cmap='viridis')
plt.show()
```

KMeans: GMM:





3. Comparison: The algorithm of Kmeans and GMM in sklearn and in my implement is basically same. However, in sklearn, it has more parameter can choose. For example, sklearn.GaussianMixture has parameter of "covariance_type" to choose the type of $\Sigma$, and parameter of warm_start to run algorithm multiple times and use the result of last time as the initialization of next time. And it can show ellipse regions fitting the cluster shape like below:



These are what my implement doesn't have.

But in the case of this assignment, the result of sklearn and my implement is basically same.

4. Part 3: Applications

According to the Wikipedia of Clustering analysis, the algorithm of clustering is used on commercial, biology, and insurance field frequently. For example, it can be used on business to figure out the feature of clients based on their purchase behavior, or used on biology by analyzing the similarity and difference of construction of different organism.