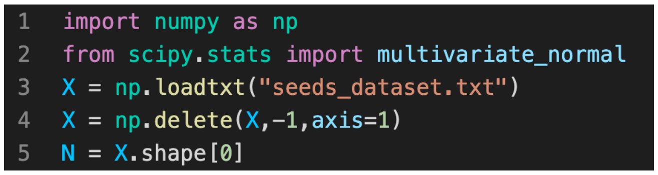
Programming Part

In this question, we want to implement two clustering algorithms, K-means and Gaussian Mixture Model (GMM), from scratch to cluster *UCI seed dataset*, which contains measurements of geometrical properties of kernels belonging to three different varieties of wheat, a glance of data set is as follows:

15.26	14.84	0.871	5.763	3.312	2.221	5.22	1
14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
14.29	14.09	0.905	5.291	3.337	2.699	4.825	1
13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
14.38	14.21	0.8951	5.386	3.312	2.462	4.956	1
14.69	14.49	0.8799	5.563	3.259	3.586	5.219	1

Figure 1. UCI Seed Dataset

There are 210 data in total, 7 attributes for each data, and the last column is the label. We first load the dataset into our program, where the last column is dropped since it is not of our clustering interests.

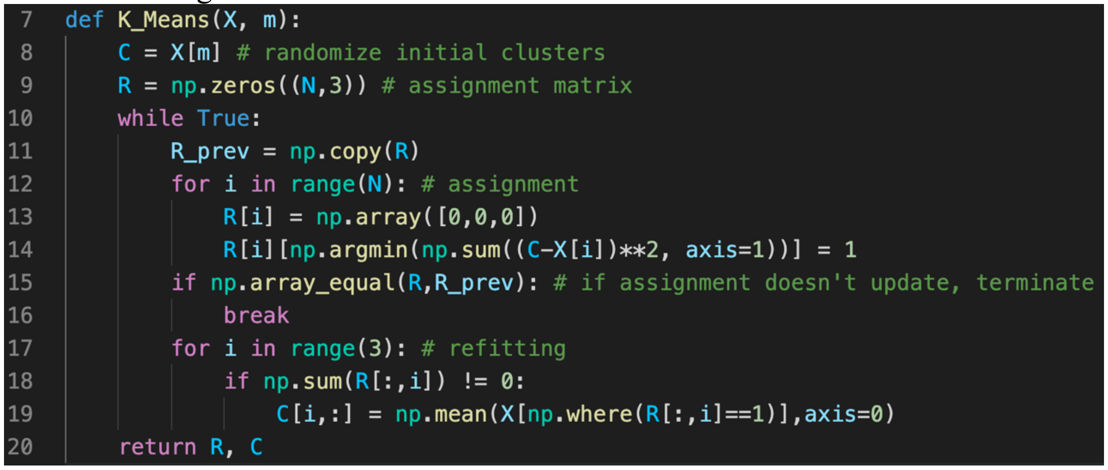


```
1  import numpy as np
2  from scipy.stats import multivariate_normal
3  X = np.loadtxt("seeds_dataset.txt")
4  X = np.delete(X,-1,axis=1)
5  N = X.shape[0]
```

Figure 2. Dataset Initialization

K-means Implementation: Line 7-20 implements the K-means algorithm as a function. Here, m is an array of random index of the dataset X with length three, so C represents a random sample from the original dataset X which we use as our initial clusters. R represents the assignment matrix, whose i^th^ column uses 0 and 1 to indicate which cluster the i^th^ sample belongs to.

We follow the Assignment and Refitting steps to iteratively update the assignment matrix R and the positions of the clusters C. After initialization, in each Assignment step, for each data point $x^{(i)}$, we calculate its distance to all three clusters and choose the cluster with the smallest distance as its assignment, say cluster k, and update R[i][k] = 1, and R[i][j] = 0 for j != k. After the assignment for all data points, we check if the assignment matrix R is the same as before i.e. R_prev. If so, no new assignment occurs and we terminate the algo. Otherwise, we come to Refitting step to update each three clusters as the mean vector of all data points assigned to it. The return is the assignment matrix R and the final clusters C.



```
7  def K_Means(X, m):
8      C = X[m] # randomize initial clusters
9      R = np.zeros((N,3)) # assignment matrix
10     while True:
11         R_prev = np.copy(R)
12         for i in range(N): # assignment
13             R[i] = np.array([0,0,0])
14             R[i][np.argmin(np.sum((C-X[i])**2, axis=1))] = 1
15         if np.array_equal(R,R_prev): # if assignment doesn't update, terminate
16             break
17         for i in range(3): # refitting
18             if np.sum(R[:,i]) != 0:
19                 C[i,:] = np.mean(X[np.where(R[:,i]==1)],axis=0)
20     return R, C
```

Figure 3. K-means Implementation from Scratch

GMM Implementation: Line 22 – 51 implements the GMM algo from scratch. As in K-means, we initialize the Gaussian mean vector as three random samples from the original dataset X. The covariance matrix is initialized to be three identity matrices. The weights are initialized to be random numbers from 0 to 1 such that they sum to 1. Gamma is the 210-by-3 posterior probability matrix, with Gamma[i][k] = P($z^{(i)}$=k| $x^{(i)}$, Mu, Sigma, Pi). Mu (stands for μ), Sigma (stands for Σ), Pi (stands for π) are the parameters that we need to optimize.

We follow the EM method. Line 31 – 36 stands for E-step, where we calculate the posterior probabilities P($z^{(i)}$=k| $x^{(i)}$, Mu, Sigma, Pi) for each k = 0,1,2 and each $x^{(i)}$ given the current Mu, Sigma, and Pi, and use the results to update the posterior probability matrix Gamma.

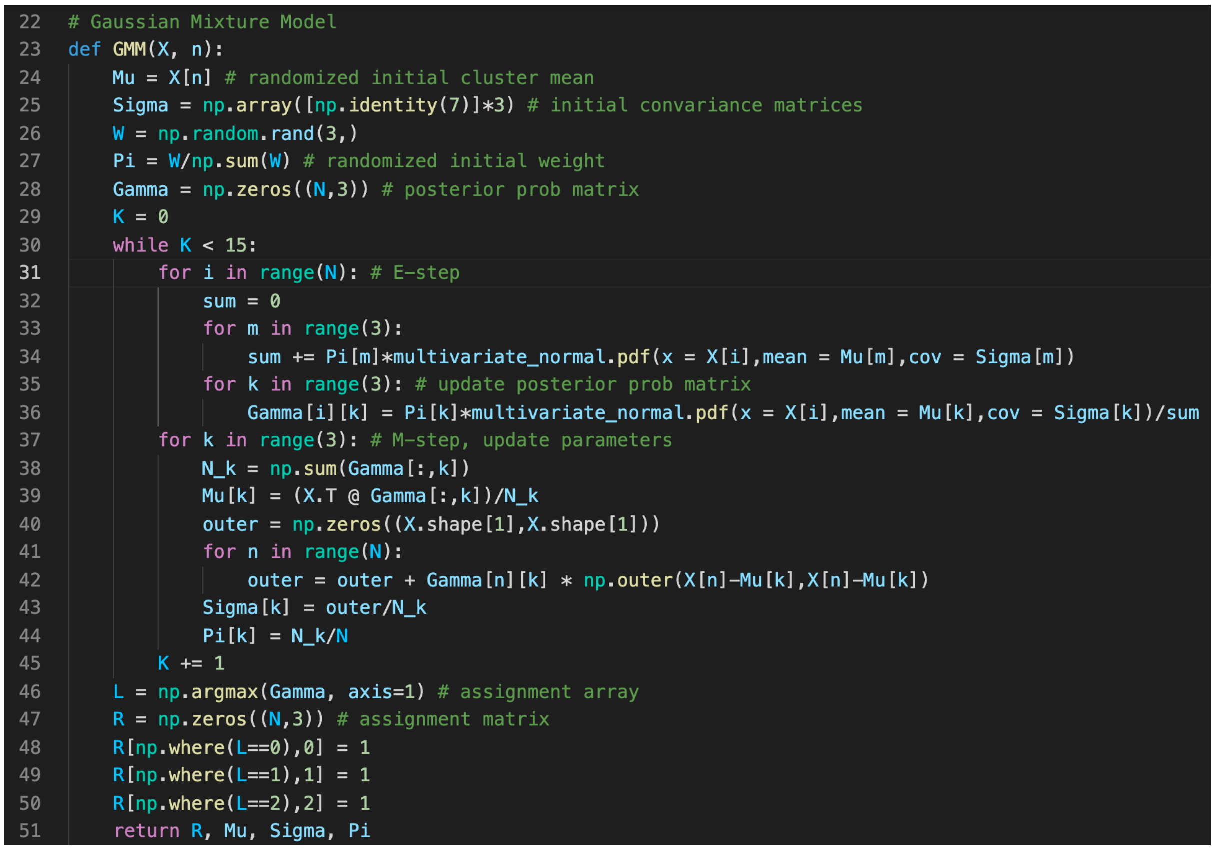
Line 37 – 44 stands for M-step, in which we update the parameters according to the formula derived in the lecture:

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}^{(n)}$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\left(\mathbf{x}^{(n)}-\boldsymbol{\mu}_k\right)\left(\mathbf{x}^{(n)}-\boldsymbol{\mu}_k\right)^{\top}$$

$$\pi_k = \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N}\gamma_k^{(n)}$$

Figure 4. Formulas in M-step for Updating Parameters



```
22  # Gaussian Mixture Model
23  def GMM(X, n):
24      Mu = X[n] # randomized initial cluster mean
25      Sigma = np.array([np.identity(7)]*3) # initial convariance matrices
26      W = np.random.rand(3,)
27      Pi = W/np.sum(W) # randomized initial weight
28      Gamma = np.zeros((N,3)) # posterior prob matrix
29      K = 0
30      while K < 15:
31          for i in range(N): # E-step
32              sum = 0
33              for m in range(3):
34                  sum += Pi[m]*multivariate_normal.pdf(x = X[i],mean = Mu[m],cov = Sigma[m])
35              for k in range(3): # update posterior prob matrix
36                  Gamma[i][k] = Pi[k]*multivariate_normal.pdf(x = X[i],mean = Mu[k],cov = Sigma[k])/sum
37          for k in range(3): # M-step, update parameters
38              N_k = np.sum(Gamma[:,k])
39              Mu[k] = (X.T @ Gamma[:,k])/N_k
40              outer = np.zeros((X.shape[1],X.shape[1]))
41              for n in range(N):
42                  outer = outer + Gamma[n][k] * np.outer(X[n]-Mu[k],X[n]-Mu[k])
43              Sigma[k] = outer/N_k
44              Pi[k] = N_k/N
45          K += 1
46      L = np.argmax(Gamma, axis=1) # assignment array
47      R = np.zeros((N,3)) # assignment matrix
48      R[np.where(L==0),0] = 1
49      R[np.where(L==1),1] = 1
50      R[np.where(L==2),2] = 1
51      return R, Mu, Sigma, Pi
```

Figure 5. GMM Implementation from Scratch

We terminate after 15 iterations for both E-step and M-step, and return the assignment matrix R defined analogously in the K-means algo.

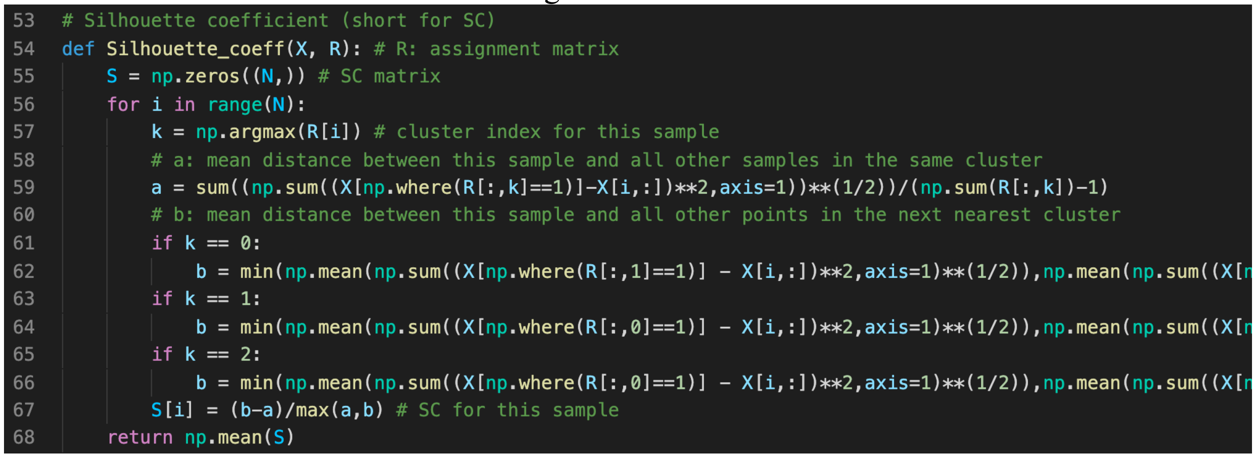
After the implementation for K-means and GMM as our clustering algos, we then implement two evaluation metrics, the Silhouette Coefficient and Rand Index from scratch, so that we could evaluate how the two clustering algos perform on the *UCI seed dataset*, and how similar the clustering results produced by these two algos are.

Silhouette Coefficient (short for SC) Implementation: Line 53 – 68 implements the SC coefficient. The SC coefficient is defined as:

$$s = \frac{b - a}{\max(a, b)}$$

Figure 6. Definition of Silhouette Coefficient

where a is the mean distance between a point and all other points in the same cluster, and b is the mean distance between a point and all other points in the next nearest cluster. Note that s is strictly between -1 and 1, and a larger s indicates a better clustering performance. We thereby calculate a in Line 59 and b in Line 61 – 66 for each sample and store the SC coefficient for this sample in S, our SC matrix. We return the mean of SC coefficients for all samples as the SC coefficient for the dataset X with clustering result R.



```
53  # Silhouette coefficient (short for SC)
54  def Silhouette_coeff(X, R): # R: assignment matrix
55      S = np.zeros((N,)) # SC matrix
56      for i in range(N):
57          k = np.argmax(R[i]) # cluster index for this sample
58          # a: mean distance between this sample and all other samples in the same cluster
59          a = sum((np.sum((X[np.where(R[:,k]==1)]-X[i,:])**2,axis=1))**(1/2))/(np.sum(R[:,k])-1)
60          # b: mean distance between this sample and all other points in the next nearest cluster
61          if k == 0:
62              b = min(np.mean(np.sum((X[np.where(R[:,1]==1)] - X[i,:])**2,axis=1)**(1/2)),np.mean(np.sum((X[
63          if k == 1:
64              b = min(np.mean(np.sum((X[np.where(R[:,0]==1)] - X[i,:])**2,axis=1)**(1/2)),np.mean(np.sum((X[
65          if k == 2:
66              b = min(np.mean(np.sum((X[np.where(R[:,0]==1)] - X[i,:])**2,axis=1)**(1/2)),np.mean(np.sum((X[
67          S[i] = (b-a)/max(a,b) # SC for this sample
68      return np.mean(S)
```

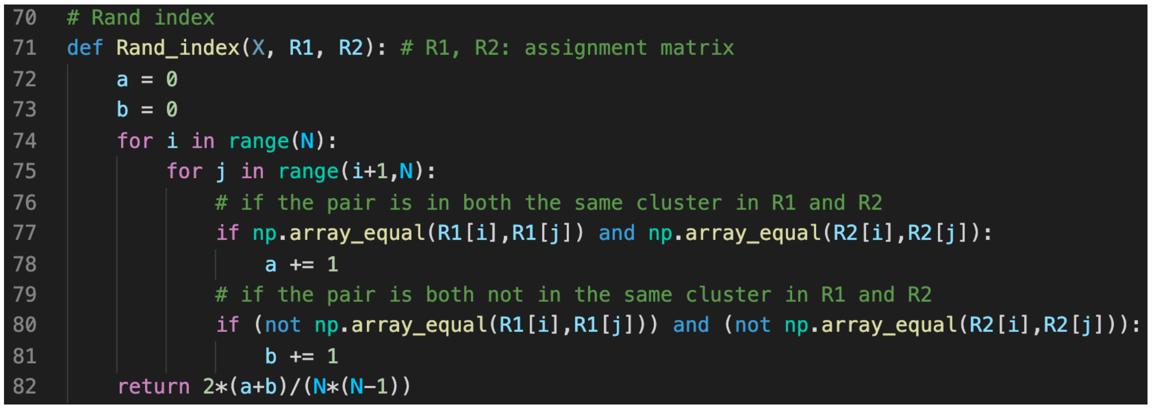
Figure 7. SC Implementation from Scratch

Rand Index (Short for RI) Implementation: Line 70 – 82 implements the Rand Index. Given a dataset X and two clustering results R_1 and R_2 of it, the Rand Index is defined to be:

$$\mathrm{RI} = \frac{a + b}{\frac{n(n-1)}{2}}$$

Figure 8. Definition of Rand Index

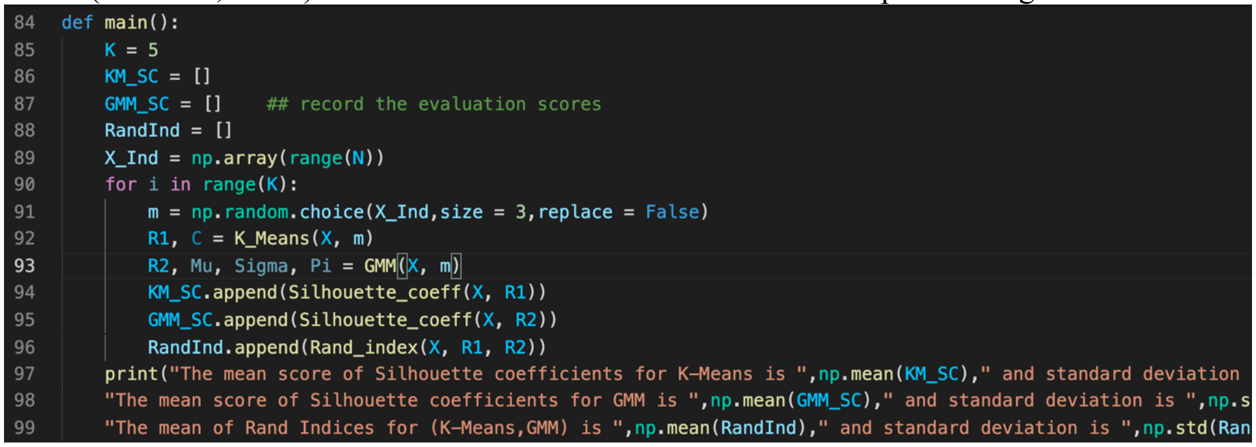
where a is the number of pairs of elements in X such that the pair is in the same cluster of R_1 and the same cluster of R_2, and b is the number of pairs of elements in X such that the pair is in the different clusters of R_1 and the different clusters of R_2. Accordingly, we loop through the (i,j) pair of elements in X for i from 0 to 209 and j from i+1 to 209 in our program and check whether the i^{th} row and the j^{th} row of the assignment matrix R_m are the same for m = 1,2. If this is true for both m = 1 and 2, we count 1 for a; if this is false for both m = 1 and 2, we count 1 for b. We return the Rand Index calculated by the formula in Figure 8 with n = 210.



```
70  # Rand index
71  def Rand_index(X, R1, R2): # R1, R2: assignment matrix
72      a = 0
73      b = 0
74      for i in range(N):
75          for j in range(i+1,N):
76              # if the pair is in both the same cluster in R1 and R2
77              if np.array_equal(R1[i],R1[j]) and np.array_equal(R2[i],R2[j]):
78                  a += 1
79              # if the pair is both not in the same cluster in R1 and R2
80              if (not np.array_equal(R1[i],R1[j])) and (not np.array_equal(R2[i],R2[j])):
81                  b += 1
82      return 2*(a+b)/(N*(N-1))
```

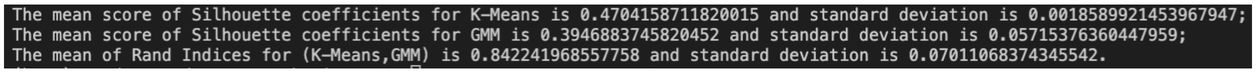
Figure 9. Rand Index Implementation from Scratch

The main() Function and the Output: In the main function from Line 84 – 99, we randomly initialize the index set with length 3 denoted as **m**, and choose the three samples in X according to this index set as our initial clusters for K-means and initial Gaussian centers for GMM. We then use the output assignment matrices to calculate the SC score for both algos and the RI for similarity of the output of these two algos. The three scores are recorded respectively in three lists. We repeat this step for K = 5 times, and output the mean score and standard deviation for those three lists, i.e., the mean SC and std SC for K-means with 5 random initializations, the mean SC and std SC for GMM with the same 5 random initializations, and the mean RI and std RI for (K-means,GMM) with those 5 random initializations. The output is in Figure 11.



```
84  def main():
85      K = 5
86      KM_SC = []
87      GMM_SC = []    ## record the evaluation scores
88      RandInd = []
89      X_Ind = np.array(range(N))
90      for i in range(K):
91          m = np.random.choice(X_Ind,size = 3,replace = False)
92          R1, C = K_Means(X, m)
93          R2, Mu, Sigma, Pi = GMM(X, m)
94          KM_SC.append(Silhouette_coeff(X, R1))
95          GMM_SC.append(Silhouette_coeff(X, R2))
96          RandInd.append(Rand_index(X, R1, R2))
97      print("The mean score of Silhouette coefficients for K-Means is ",np.mean(KM_SC)," and standard deviation
98      "The mean score of Silhouette coefficients for GMM is ",np.mean(GMM_SC)," and standard deviation is ",np.s
99      "The mean of Rand Indices for (K-Means,GMM) is ",np.mean(RandInd)," and standard deviation is ",np.std(Ran
```

Figure 10. The main() Function



```
The mean score of Silhouette coefficients for K-Means is 0.4704158711820015 and standard deviation is 0.0018589921453967947;
The mean score of Silhouette coefficients for GMM is 0.3946883745820452 and standard deviation is 0.05715376360447959;
The mean of Rand Indices for (K-Means,GMM) is 0.842241968557758 and standard deviation is 0.07011068374345542.
```

Figure 11. Output Mean and Std of SC/RI for K-means and GMM

Theoretically, SC is strictly between -1 and 1 and a larger SC implies a better clustering performance. Since both the mean SC for K-means and GMM are positive, with 0.47 for K-means and 0.39 for GMM, we conclude that both algos work fairly well in our *UCI seed dataset*. Moreover, K-means performs slightly better than GMM, which may result from the separability of the dataset. The std SC score for K-means is 0.0018, smaller than the std SC score for GMM which is 0.0571. This may result from our initialization method for the clusters, where we use random samples from the dataset itself, so K-means can be stable in performance in this case with a separable dataset. The mean of RI for these two algos is 0.842, meaning that the two algos provide similar clustering results in overall five iterations, and the std for RI is 0.07, indicating a similarity in each single iteration.