K-Means for Clustering

1. The main idea in K-Means

The idea for using K-Means is intuitive. It can be divided into four steps:

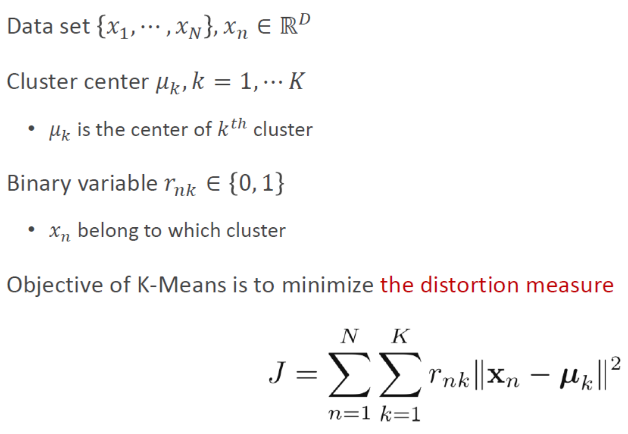
(1) Randomly select K center points.

(2) Each data point is assigned to one of the K centers.

(3) Re-compute the K centers by the mean of each group.

(4) Iterate step 2&3.

2. K-Means Definition



Data set $\{x_1, \cdots, x_N\}, x_n \in \mathbb{R}^D$

Cluster center $\mu_k, k = 1, \cdots K$

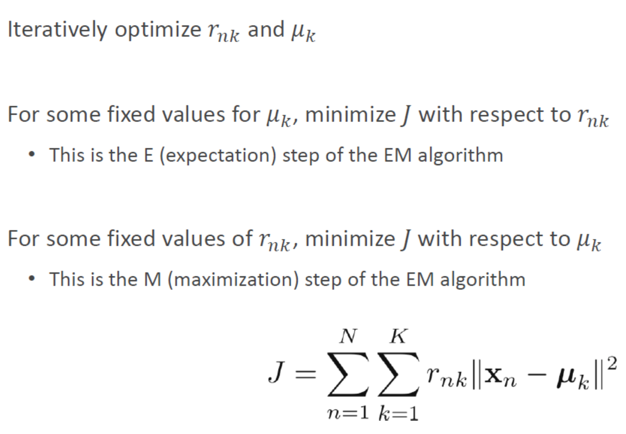
- μ_k is the center of k^{th} cluster

Binary variable $r_{nk} \in \{0, 1\}$

- x_n belong to which cluster

Objective of K-Means is to minimize the distortion measure

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$



Iteratively optimize r_{nk} and μ_k

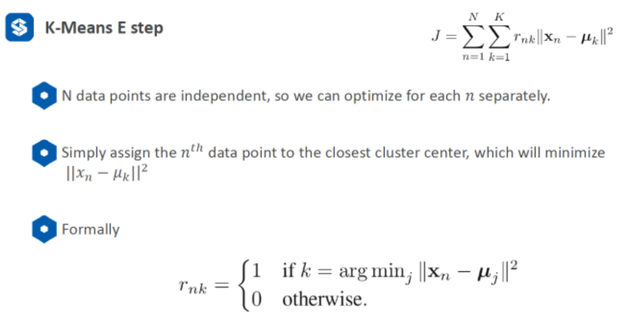
For some fixed values for μ_k, minimize J with respect to r_{nk}

- This is the E (expectation) step of the EM algorithm

For some fixed values of r_{nk}, minimize J with respect to μ_k

- This is the M (maximization) step of the EM algorithm

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

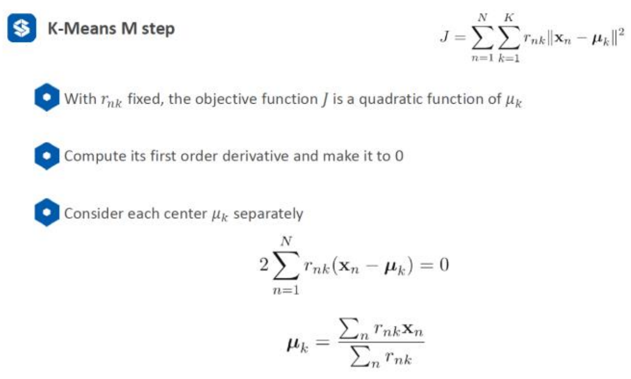


K-Means E step

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

- N data points are independent, so we can optimize for each n separately.
- Simply assign the n^{th} data point to the closest cluster center, which will minimize $||x_n - \mu_k||^2$
- Formally

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2 \\ 0 & \text{otherwise.} \end{cases}$$



K-Means M step

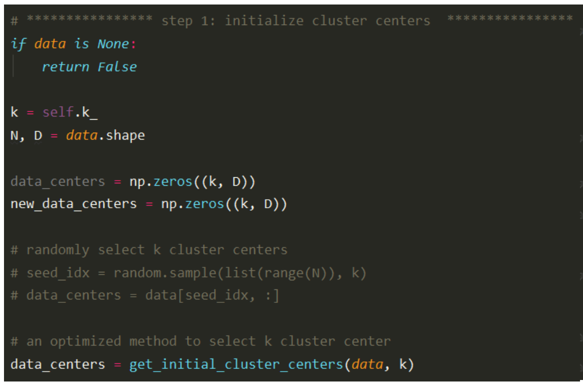
$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

- With r_{nk} fixed, the objective function J is a quadratic function of μ_k
- Compute its first order derivative and make it to 0
- Consider each center μ_k separately

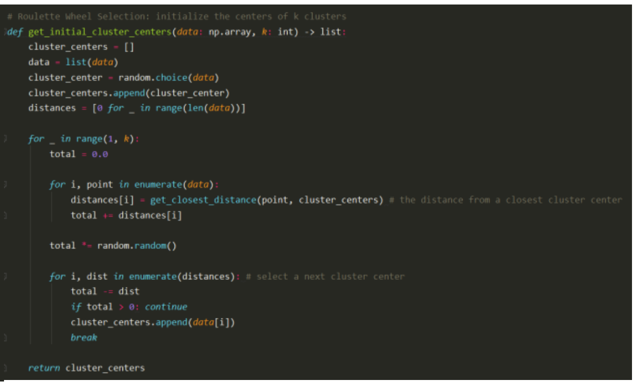
$$2 \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k) = 0$$

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk} \mathbf{x}_n}{\sum_n r_{nk}}$$

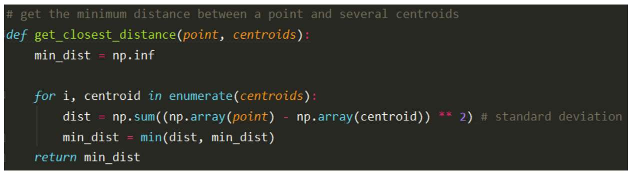
3. Initialization



```python
# ****************  step 1: initialize cluster centers  ****************
if data is None:
    return False

k = self.k_
N, D = data.shape

data_centers = np.zeros((k, D))
new_data_centers = np.zeros((k, D))

# randomly select k cluster centers
# seed_idx = random.sample(list(range(N)), k)
# data_centers = data[seed_idx, :]

# an optimized method to select k cluster center
data_centers = get_initial_cluster_centers(data, k)
```

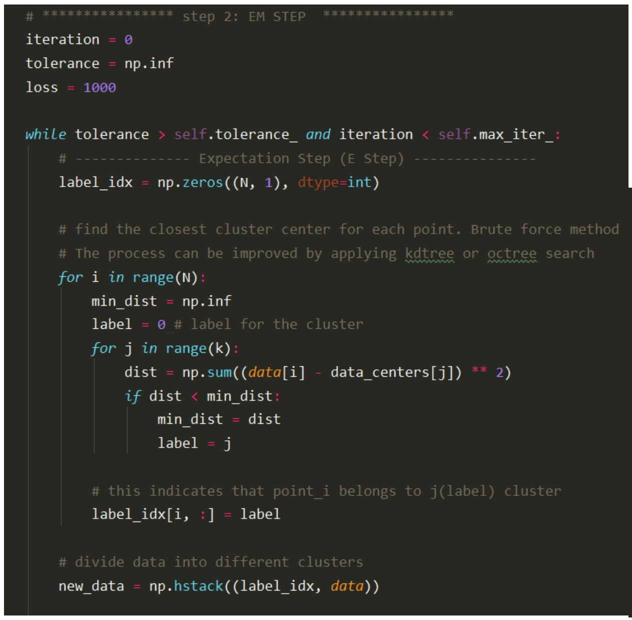


```python
# Roulette Wheel Selection: initialize the centers of k clusters
def get_initial_cluster_centers(data: np.array, k: int) -> list:
    cluster_centers = []
    data = list(data)
    cluster_center = random.choice(data)
    cluster_centers.append(cluster_center)
    distances = [0 for _ in range(len(data))]

    for _ in range(1, k):
        total = 0.0

        for i, point in enumerate(data):
            distances[i] = get_closest_distance(point, cluster_centers) # the distance from a closest cluster center
            total += distances[i]

        total *= random.random()

        for i, dist in enumerate(distances): # select a next cluster center
            total -= dist
            if total > 0: continue
            cluster_centers.append(data[i])
            break

    return cluster_centers
```

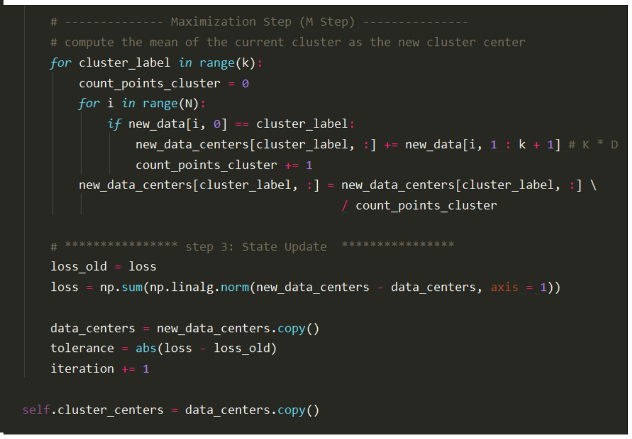


```python
# get the minimum distance between a point and several centroids
def get_closest_distance(point, centroids):
    min_dist = np.inf

    for i, centroid in enumerate(centroids):
        dist = np.sum((np.array(point) - np.array(centroid)) ** 2) # standard deviation
        min_dist = min(dist, min_dist)
    return min_dist
```

4. EM Steps



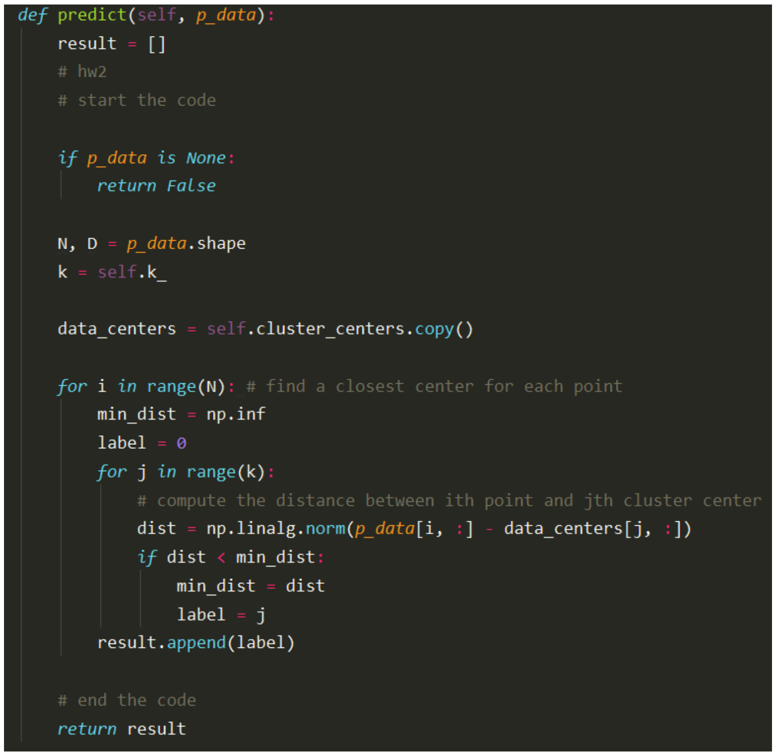
```python
# ****************  step 2: EM STEP  ****************
iteration = 0
tolerance = np.inf
loss = 1000

while tolerance > self.tolerance_ and iteration < self.max_iter_:
    # -------------- Expectation Step (E Step) --------------
    label_idx = np.zeros((N, 1), dtype=int)

    # find the closest cluster center for each point. Brute force method
    # The process can be improved by applying kdtree or octree search
    for i in range(N):
        min_dist = np.inf
        label = 0 # label for the cluster
        for j in range(k):
            dist = np.sum((data[i] - data_centers[j]) ** 2)
            if dist < min_dist:
                min_dist = dist
                label = j

        # this indicates that point_i belongs to j(label) cluster
        label_idx[i, :] = label

    # divide data into different clusters
    new_data = np.hstack((label_idx, data))
```



```python
    # -------------- Maximization Step (M Step) --------------
    # compute the mean of the current cluster as the new cluster center
    for cluster_label in range(k):
        count_points_cluster = 0
        for i in range(N):
            if new_data[i, 0] == cluster_label:
                new_data_centers[cluster_label, :] += new_data[i, 1 : k + 1] # K * D
                count_points_cluster += 1
        new_data_centers[cluster_label, :] = new_data_centers[cluster_label, :] \
                                             / count_points_cluster

    # ***************  step 3: State Update  ***************
    loss_old = loss
    loss = np.sum(np.linalg.norm(new_data_centers - data_centers, axis = 1))

    data_centers = new_data_centers.copy()
    tolerance = abs(loss - loss_old)
    iteration += 1

self.cluster_centers = data_centers.copy()
```

5. Predict

For each point, find out it nearest neighbor and corresponding cluster label, then output all the labels, which are the classified results.



```python
def predict(self, p_data):
    result = []
    # hw2
    # start the code

    if p_data is None:
        return False

    N, D = p_data.shape
    k = self.k_

    data_centers = self.cluster_centers.copy()

    for i in range(N): # find a closest center for each point
        min_dist = np.inf
        label = 0
        for j in range(k):
            # compute the distance between ith point and jth cluster center
            dist = np.linalg.norm(p_data[i, :] - data_centers[j, :])
            if dist < min_dist:
                min_dist = dist
                label = j
        result.append(label)

    # end the code
    return result
```

Gaussian Mixture Model (GMM) for Clustering

1. The main purpose in GMM

Compared to K-means, a cluster in GMM can be represented by a Gaussian Distribution. GMM tells the probability of a data point belonging to each cluster. How does this work? Suppose we have the posterior probability of each data point, which has a dimension of N x K (N is the number of points, K is the number of clusters). This means that for each point, we will have K probability values and each value indicates the probability to a specific cluster. For each single point, we can compare all K values and get the one with highest probability. The label corresponding to this value is the cluster that the data point belongs to. After we consider all the data points, we can get the corresponding cluster labels for these points, and hence the data points are classified as K clusters. Therefore, ***the most important part in GMM is to compute the posterior probability of the data points***. The steps can be divided as:

1. Initialize the means μ_k, covariances Σ_k and weights π_k
2. E-step. Evaluate the posterior $p(z_{nk} = 1|x_n)$, intuitively this is the probability of x_n being assigned to each of the K clusters.

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

3. M-Step. Estimate the parameters using MLE.
4. Evaluate the log likelihood, if converges, stop. Otherwise go back to E-step

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

$$\begin{aligned}
\boldsymbol{\mu}_k^{\text{new}} &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \\
\boldsymbol{\Sigma}_k^{\text{new}} &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \left(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}}\right) \left(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}}\right)^{\mathrm{T}} \\
\pi_k^{\text{new}} &= \frac{N_k}{N}
\end{aligned}$$

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

2. Parameters Update

For multivariate Gaussian Distribution with D-dimensional vector, it can be represented as

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\}$$

A Gaussian mixture distribution can be written as a linear combination of single Gaussians with weights:

$$p(x) = \sum_{k=1}^{K}\pi_k\mathcal{N}(x|\mu_k,\Sigma_k)$$

. Note: K is the number of data points, not the number of clusters.

The conditional probability P(z|x) is the clustering “label probability” we want, here is

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}, \qquad p(x) = \sum p(z)p(x|z)$$

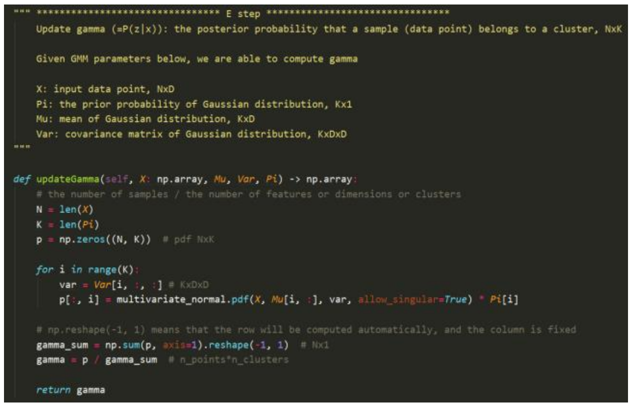
Therefore, the main purpose is to compute γ. There are other important parameters in GMM, such as π, μ, Σ, which will also be computed during this process.

(1) Compute γ

correction:p(z, x)

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{z}|\mathbf{x})}{p(\mathbf{x})} = \frac{p(\mathbf{z})p(\mathbf{x}|\mathbf{z})}{\sum_{j=1}^{K}p(\mathbf{x}|z_j)p(z_j)}$$

$$\gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) = \frac{p(z_k=1)p(\mathbf{x}|z_k=1)}{\sum_{j=1}^{K}p(z_j=1)p(\mathbf{x}|z_j=1)} = \frac{\pi_k\mathcal{N}(\mathbf{x}_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(\mathbf{x}_n|\mu_j,\Sigma_j)}$$



```
""" ******************************** E step ********************************
    Update gamma (=P(z|x)): the posterior probability that a sample (data point) belongs to a cluster, NxK

    Given GMM parameters below, we are able to compute gamma

    X: input data point, NxD
    Pi: the prior probability of Gaussian distribution, Kx1
    Mu: mean of Gaussian distribution, KxD
    Var: covariance matrix of Gaussian distribution, KxDxD
"""

def updateGamma(self, X: np.array, Mu, Var, Pi) -> np.array:
    # the number of samples / the number of features or dimensions or clusters
    N = len(X)
    K = len(Pi)
    p = np.zeros((N, K))  # pdf NxK

    for i in range(K):
        var = Var[i, :, :] # KxDxD
        p[:, i] = multivariate_normal.pdf(X, Mu[i, :], var, allow_singular=True) * Pi[i]

    # np.reshape(-1, 1) means that the row will be computed automatically, and the column is fixed
    gamma_sum = np.sum(p, axis=1).reshape(-1, 1)  # Nx1
    gamma = p / gamma_sum  # n_points*n_clusters

    return gamma
```

(2) Compute π

Solve π_k

Intuitively, π_k is the effective cluster member number over data set size.

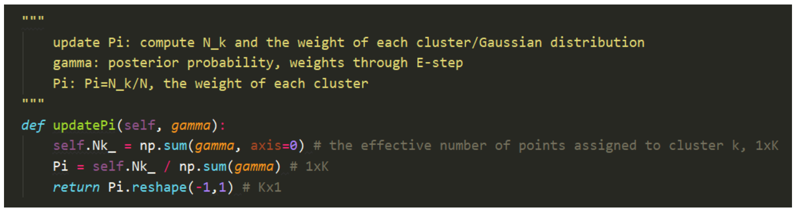
$$0 = \sum_{n=1}^{N}\frac{\mathcal{N}(\mathbf{x}_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(\mathbf{x}_n|\mu_k,\Sigma_k)} + \lambda$$

$$0 = \frac{1}{\pi_k}\sum_{n=1}^{N}\frac{\pi_k\mathcal{N}(\mathbf{x}_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(\mathbf{x}_n|\mu_k,\Sigma_k)} - N$$

$$0 = \frac{1}{\pi_k}\sum_{n=1}^{N}\gamma_{nk} - N$$

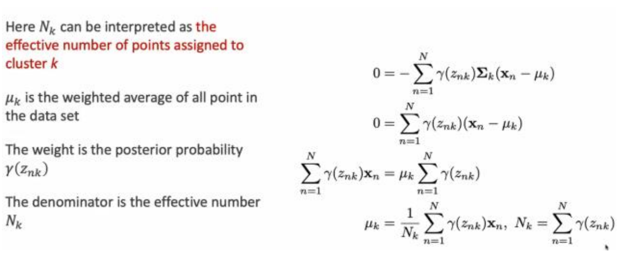
$$\pi_k = \frac{N_k}{N}$$

$$\boxed{\begin{gathered}\gamma(z_{nk}) = \frac{\pi_k\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)} \\ N_k = \sum_{n=1}^{N}\gamma(z_{nk})\end{gathered}}$$



```
"""
    update Pi: compute N_k and the weight of each cluster/Gaussian distribution
    gamma: posterior probability, weights through E-step
    Pi: Pi=N_k/N, the weight of each cluster
"""
def updatePi(self, gamma):
    self.Nk_ = np.sum(gamma, axis=0) # the effective number of points assigned to cluster k, 1xK
    Pi = self.Nk_ / np.sum(gamma) # 1xK
    return Pi.reshape(-1,1) # Kx1
```

(3) Compute μ



Here N_k can be interpreted as the effective number of points assigned to cluster *k*

μ_k is the weighted average of all point in the data set

The weight is the posterior probability $\gamma(z_{nk})$

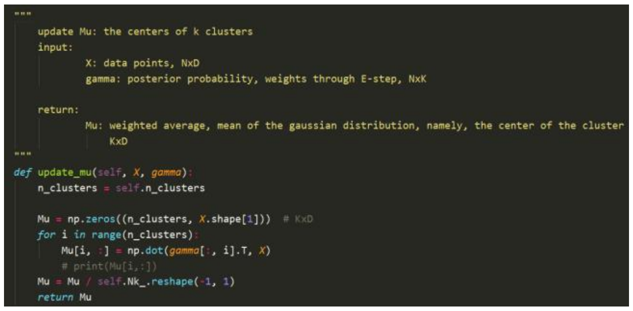
The denominator is the effective number N_k

$$0 = -\sum_{n=1}^{N} \gamma(z_{nk}) \boldsymbol{\Sigma}_k (\mathbf{x}_n - \mu_k)$$

$$0 = \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \mu_k)$$

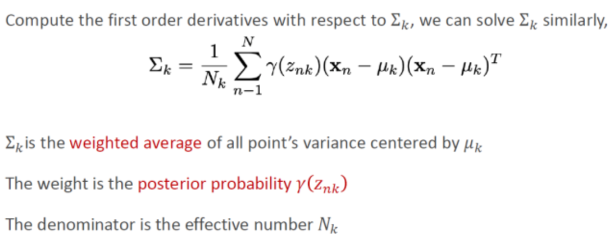
$$\sum_{n=1}^{N} \gamma(z_{nk})\mathbf{x}_n = \mu_k \sum_{n=1}^{N} \gamma(z_{nk})$$

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})\mathbf{x}_n, \; N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$



```
"""
    update Mu: the centers of k clusters
    input:
            X: data points, NxD
            gamma: posterior probability, weights through E-step, NxK

    return:
            Mu: weighted average, mean of the gaussian distribution, namely, the center of the cluster
                KxD
"""
def update_mu(self, X, gamma):
    n_clusters = self.n_clusters

    Mu = np.zeros((n_clusters, X.shape[1]))  # KxD
    for i in range(n_clusters):
        Mu[i, :] = np.dot(gamma[:, i].T, X)
        # print(Mu[i,:])
    Mu = Mu / self.Nk_.reshape(-1, 1)
    return Mu
```

(4) Compute Σ



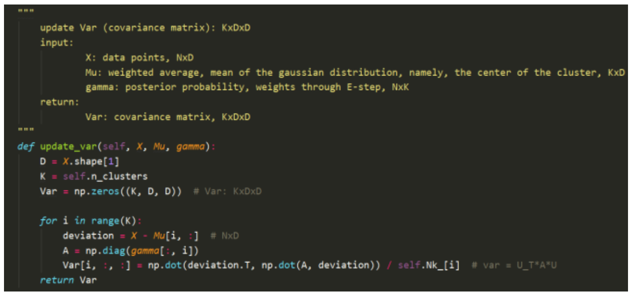
Compute the first order derivatives with respect to Σ_k, we can solve Σ_k similarly,

$$\Sigma_k = \frac{1}{N_k} \sum_{n-1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T$$

$\Sigma_k$is the weighted average of all point's variance centered by μ_k

The weight is the posterior probability $\gamma(z_{nk})$

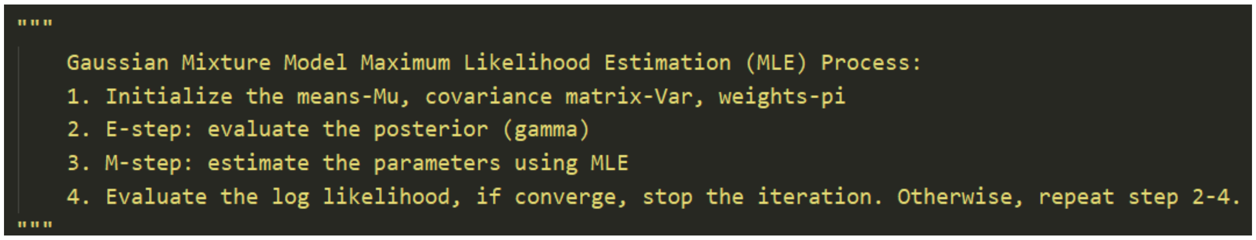
The denominator is the effective number N_k



```
"""
    update Var (covariance matrix): KxDxD
    input:
            X: data points, NxD
            Mu: weighted average, mean of the gaussian distribution, namely, the center of the cluster, KxD
            gamma: posterior probability, weights through E-step, NxK
    return:
            Var: covariance matrix, KxDxD
"""
def update_var(self, X, Mu, gamma):
    D = X.shape[1]
    K = self.n_clusters
    Var = np.zeros((K, D, D))  # Var: KxDxD

    for i in range(K):
        deviation = X - Mu[i, :]  # NxD
        A = np.diag(gamma[:, i])
        Var[i, :, :] = np.dot(deviation.T, np.dot(A, deviation)) / self.Nk_[i]  # var = U_T*A*U
    return Var
```

3. GMM Process

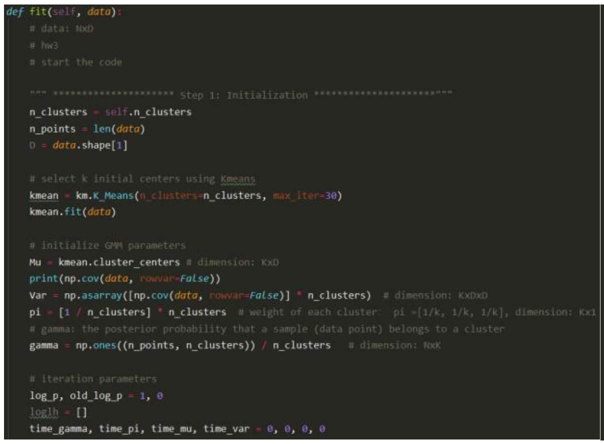
The process in GMM can be divided into four steps as follows.



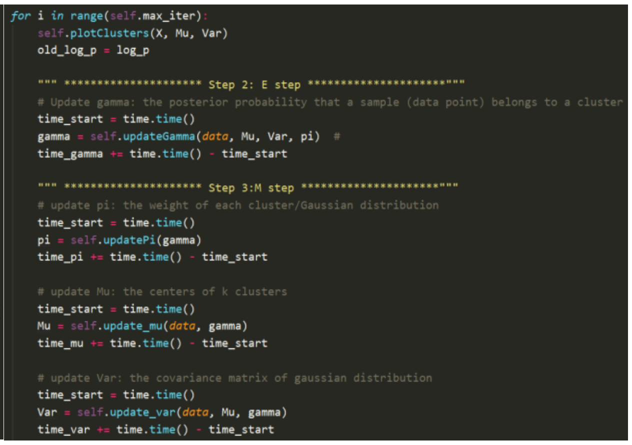
```
"""
    Gaussian Mixture Model Maximum Likelihood Estimation (MLE) Process:
    1. Initialize the means-Mu, covariance matrix-Var, weights-pi
    2. E-step: evaluate the posterior (gamma)
    3. M-step: estimate the parameters using MLE
    4. Evaluate the log likelihood, if converge, stop the iteration. Otherwise, repeat step 2-4.
"""
```

Step 1: Initialization. Note: K-means is used here to get the initial cluster centers.

Step 2-3: EM steps. Iteratively update the four parameters and record the time used for each function.

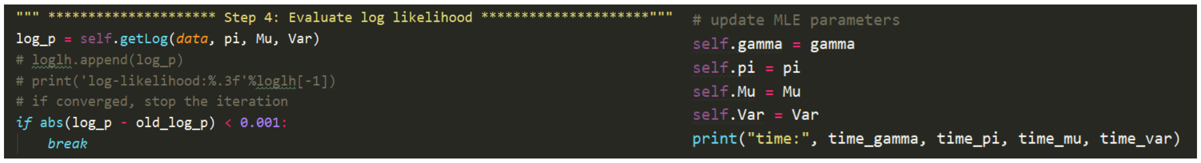


```
def fit(self, data):
    # data: NxD
    # hw3
    # start the code

    """ ******************** Step 1: Initialization *********************"""
    n_clusters = self.n_clusters
    n_points = len(data)
    D = data.shape[1]

    # select k initial centers using Kmeans
    kmean = km.K_Means(n_clusters=n_clusters, max_iter=30)
    kmean.fit(data)

    # initialize GMM parameters
    Mu = kmean.cluster_centers # dimension: KxD
    print(np.cov(data, rowvar=False))
    Var = np.asarray([np.cov(data, rowvar=False)] * n_clusters)  # dimension: KxDxD
    pi = [1 / n_clusters] * n_clusters  # weight of each cluster:  pi =[1/k, 1/k, 1/k], dimension: Kx1
    # gamma: the posterior probability that a sample (data point) belongs to a cluster
    gamma = np.ones((n_points, n_clusters)) / n_clusters   # dimension: NxK

    # iteration parameters
    log_p, old_log_p = 1, 0
    loglh = []
    time_gamma, time_pi, time_mu, time_var = 0, 0, 0, 0
```



```
for i in range(self.max_iter):
    self.plotClusters(X, Mu, Var)
    old_log_p = log_p

    """ ******************** Step 2: E step *********************"""
    # Update gamma: the posterior probability that a sample (data point) belongs to a cluster
    time_start = time.time()
    gamma = self.updateGamma(data, Mu, Var, pi)  #
    time_gamma += time.time() - time_start

    """ ******************** Step 3:M step *********************"""
    # update pi: the weight of each cluster/Gaussian distribution
    time_start = time.time()
    pi = self.updatePi(gamma)
    time_pi += time.time() - time_start

    # update Mu: the centers of k clusters
    time_start = time.time()
    Mu = self.update_mu(data, gamma)
    time_mu += time.time() - time_start

    # update Var: the covariance matrix of gaussian distribution
    time_start = time.time()
    Var = self.update_var(data, Mu, gamma)
    time_var += time.time() - time_start
```

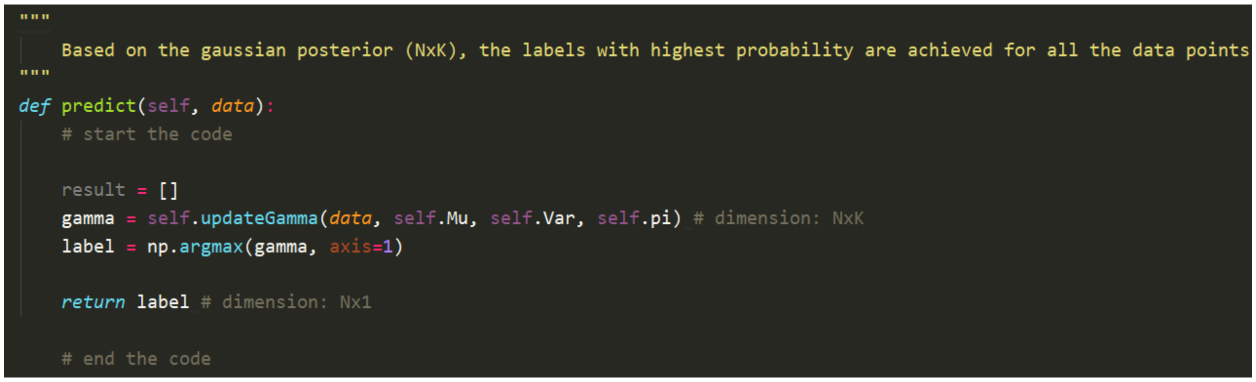
Step 4: Evaluate log likelihood. If the difference between current and previous log likelihood is smaller than 0.001, then we can consider the iteration as converged, and stop the update.



```
""" ******************** Step 4: Evaluate log likelihood *********************"""
log_p = self.getLog(data, pi, Mu, Var)
# loglh.append(log_p)
# print('log-likelihood:%.3f'%loglh[-1])
# if converged, stop the iteration
if abs(log_p - old_log_p) < 0.001:
    break
```

```
# update MLE parameters
self.gamma = gamma
self.pi = pi
self.Mu = Mu
self.Var = Var
print("time:", time_gamma, time_pi, time_mu, time_var)
```

After we fit GMM, we obtain the posterior probability of γ. However, how can we get the final classification results based on the posterior probability? Here it is.

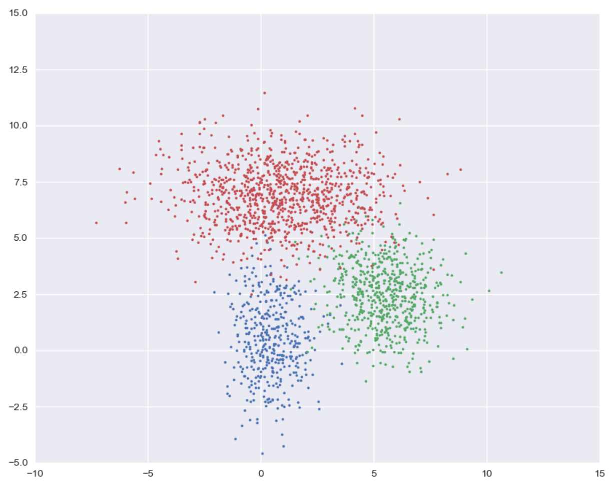


```
"""
    Based on the gaussian posterior (NxK), the labels with highest probability are achieved for all the data points
"""
def predict(self, data):
    # start the code

    result = []
    gamma = self.updateGamma(data, self.Mu, self.Var, self.pi) # dimension: NxK
    label = np.argmax(gamma, axis=1)

    return label # dimension: Nx1

    # end the code
```

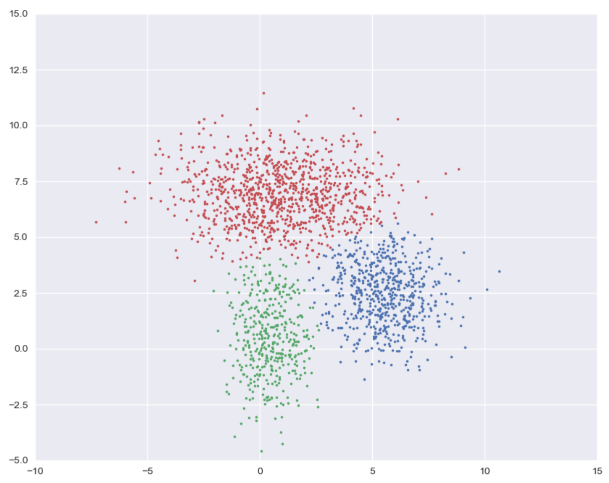
The posterior is used to output the label with highest probability for each data points, and finally output the labels for all the points. This is the classification result we want for clustering.

4. Simple Test for GMM

Here we use a simple example to demonstrate how GMM works for clustering. We use 2000 points to generate three clusters with 2D multivariate Gaussian distribution, which are 400, 1000, 600, respectively. Below shows the raw data points and the data points after GMM clustering. The result looks good.

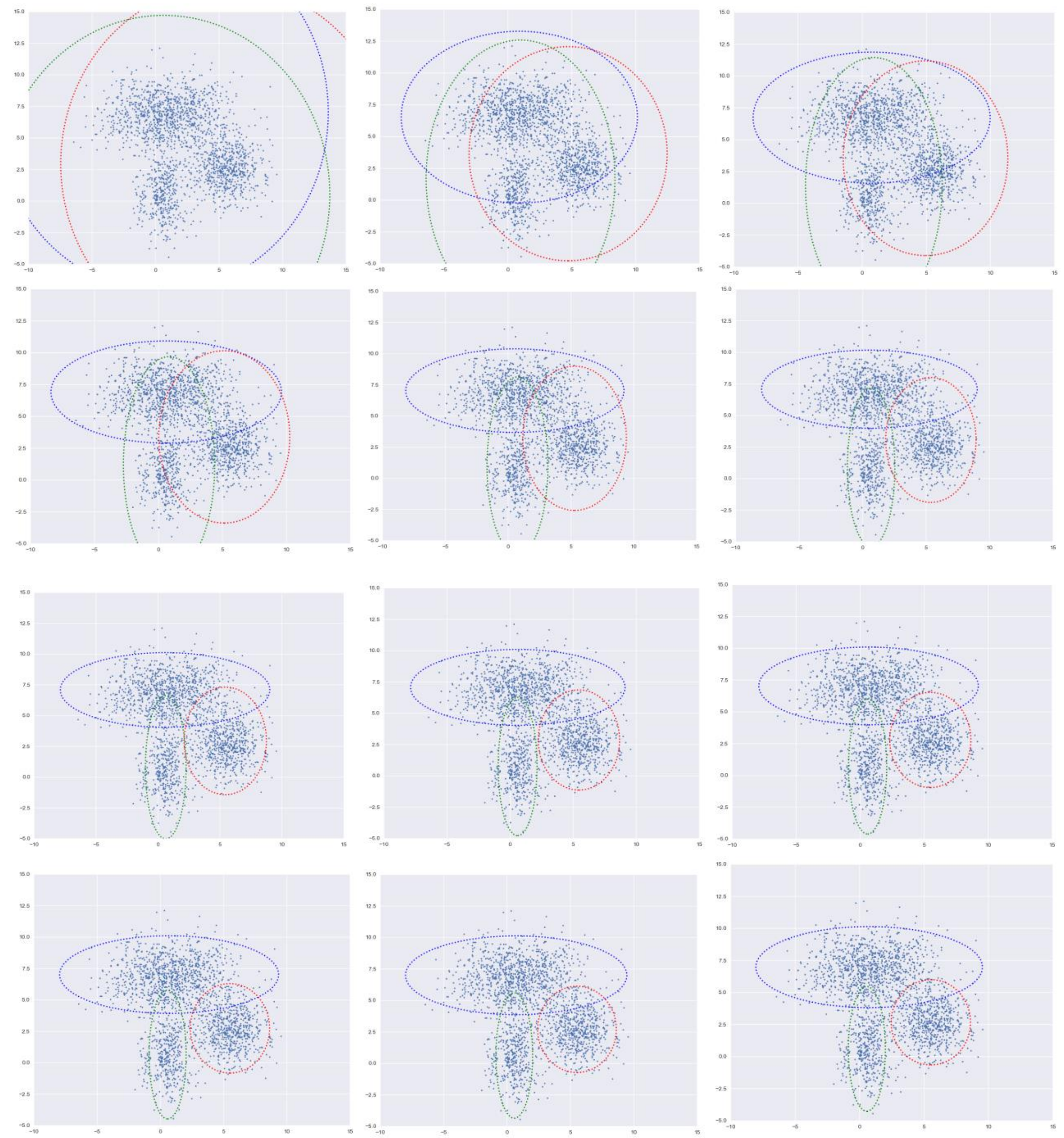


Raw data



Data after GMM

Below shows the details how GMM work during the clustering process.

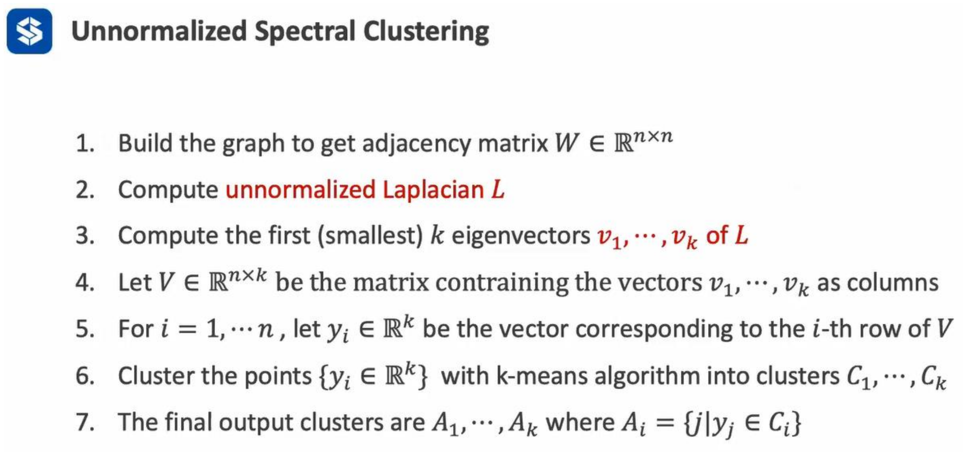


For the last few steps, the clustering becomes stable, which means the iteration tends to converge.

Spectral Clustering

GMM and K-Means are based on Euclidean distance, but they do not perform well for other metrics, i.e., connectivity. Thus, we use spectral clustering to deal with connectivity problems because it focuses on the connectivity between points in a graph (similar to connected blocks in a graph).

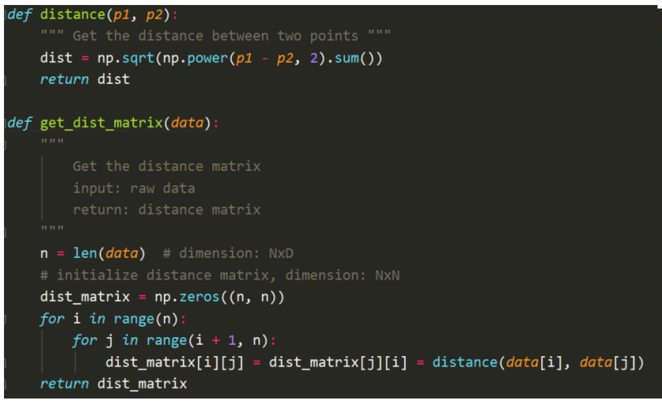
1. *Procedure for Spectral Clustering*



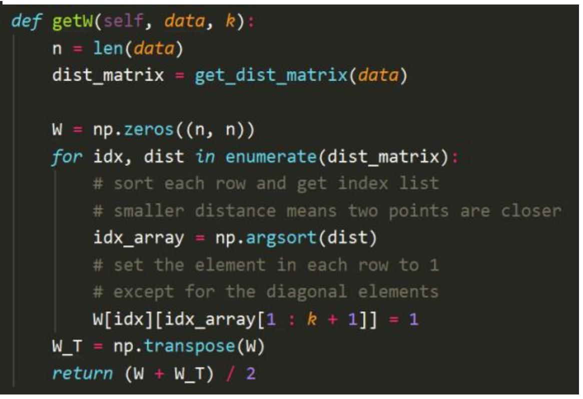
Unnormalized Spectral Clustering

1. Build the graph to get adjacency matrix $W \in \mathbb{R}^{n\times n}$
2. Compute unnormalized Laplacian L
3. Compute the first (smallest) k eigenvectors $v_1, \cdots, v_k$ of L
4. Let $V \in \mathbb{R}^{n\times k}$ be the matrix contraining the vectors $v_1, \cdots, v_k$ as columns
5. For $i = 1, \cdots n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of V
6. Cluster the points $\{y_i \in \mathbb{R}^k\}$ with k-means algorithm into clusters $C_1, \cdots, C_k$
7. The final output clusters are $A_1, \cdots, A_k$ where $A_i = \{j | y_j \in C_i\}$

2. *Compute Adjacent Matrix of Graph*

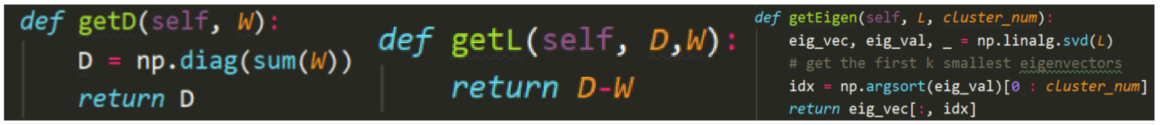


```python
def distance(p1, p2):
    """ Get the distance between two points """
    dist = np.sqrt(np.power(p1 - p2, 2).sum())
    return dist

def get_dist_matrix(data):
    """
        Get the distance matrix
        input: raw data
        return: distance matrix
    """
    n = len(data)  # dimension: NxD
    # initialize distance matrix, dimension: NxN
    dist_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(i + 1, n):
            dist_matrix[i][j] = dist_matrix[j][i] = distance(data[i], data[j])
    return dist_matrix
```



```python
def getW(self, data, k):
    n = len(data)
    dist_matrix = get_dist_matrix(data)

    W = np.zeros((n, n))
    for idx, dist in enumerate(dist_matrix):
        # sort each row and get index list
        # smaller distance means two points are closer
        idx_array = np.argsort(dist)
        # set the element in each row to 1
        # except for the diagonal elements
        W[idx][idx_array[1 : k + 1]] = 1
    W_T = np.transpose(W)
    return (W + W_T) / 2
```

3. *Compute Unnormalized Laplace Matrix and Corresponding Eigenvectors*

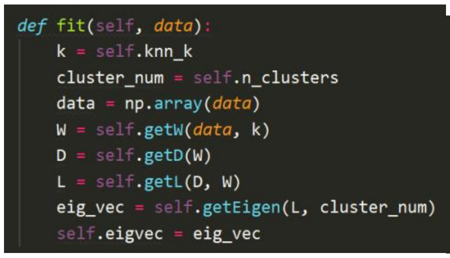


```python
def getD(self, W):
    D = np.diag(sum(W))
    return D
```

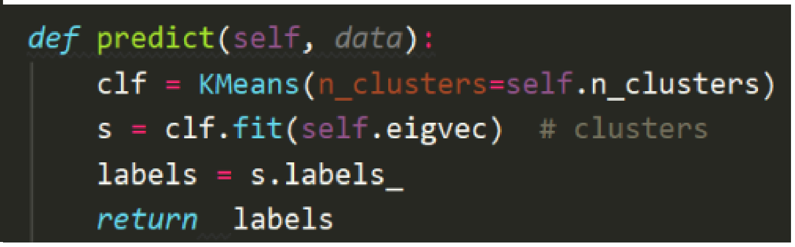
```python
def getL(self, D,W):
    return D-W
```

```python
def getEigen(self, L, cluster_num):
    eig_vec, eig_val, _ = np.linalg.svd(L)
    # get the first k smallest eigenvectors
    idx = np.argsort(eig_val)[0 : cluster_num]
    return eig_vec[:, idx]
```

4. **Fit the model and predict the labels**

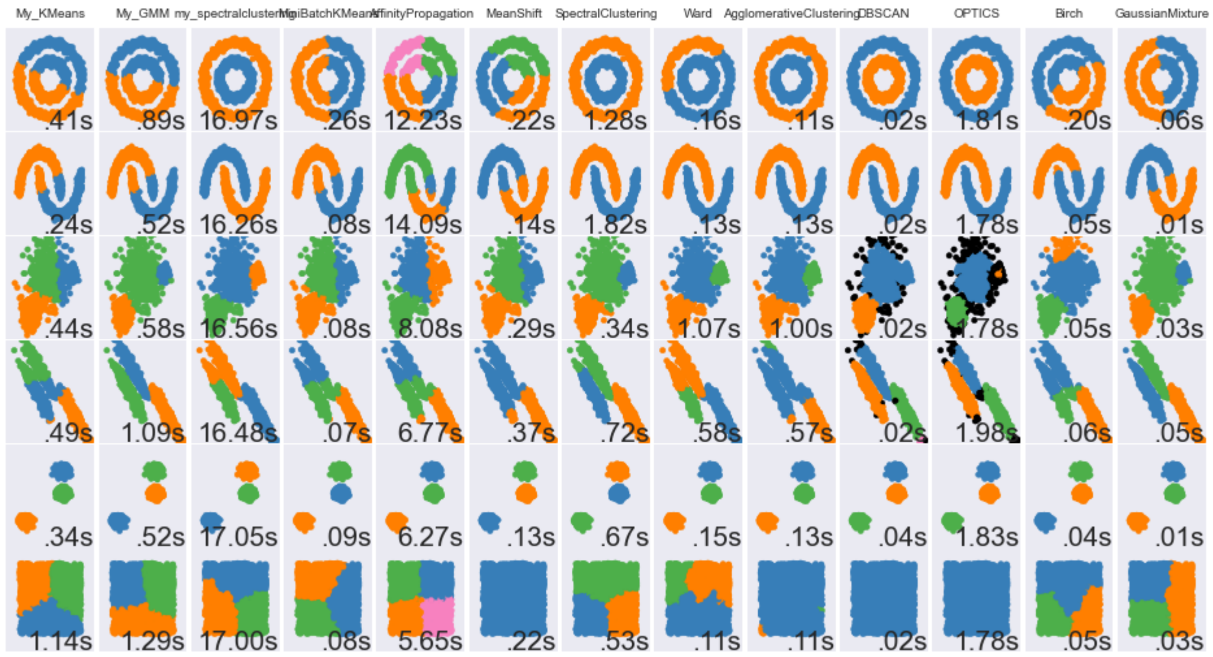


```python
def fit(self, data):
    k = self.knn_k
    cluster_num = self.n_clusters
    data = np.array(data)
    W = self.getW(data, k)
    D = self.getD(W)
    L = self.getL(D, W)
    eig_vec = self.getEigen(L, cluster_num)
    self.eigvec = eig_vec
```



```python
def predict(self, data):
    clf = KMeans(n_clusters=self.n_clusters)
    s = clf.fit(self.eigvec)  # clusters
    labels = s.labels_
    return  labels
```

Results Comparison by using self-developed clustering algorithms and default algorithm in Scikit-learn


My_KMeans
My_GMM
my_spectralclustering
MiniBatchKMeans
AffinityPropagation
MeanShift
SpectralClustering
Ward
AgglomerativeClustering
DBSCAN
OPTICS
Birch
GaussianMixture
.41s .89s 16.97s .26s 12.23s .22s 1.28s .16s .11s .02s 1.81s .20s .06s
.24s .52s 16.26s .08s 14.09s .14s 1.82s .13s .13s .02s 1.78s .05s .01s
.44s .58s 16.56s .08s 8.08s .29s .34s 1.07s 1.00s .02s 1.78s .05s .03s
.49s 1.09s 16.48s .07s 6.77s .37s .72s .58s .57s .02s 1.98s .06s .05s
.34s .52s 17.05s .09s 6.27s .13s .67s .15s .13s .04s 1.83s .04s .01s
1.14s 1.29s 17.00s .08s 5.65s .22s .53s .11s .11s .02s 1.78s .05s .03s