Lecture 10-2: Clustering Algorithm

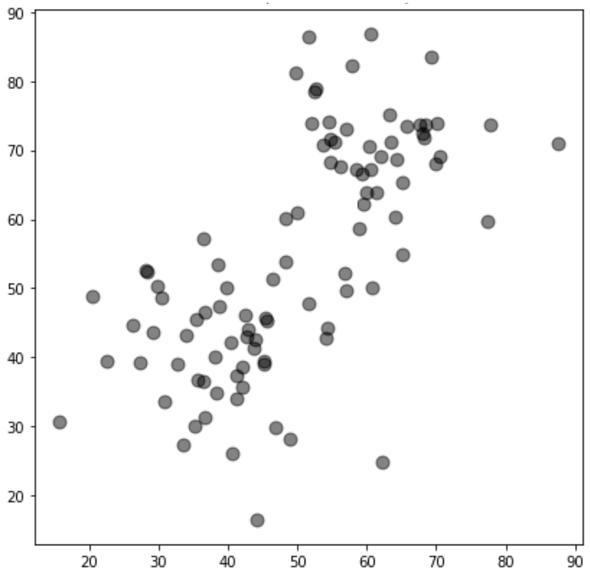
Dr. James Chen, Animal Data Scientist, School of Animal Sciences

2023 APSC-5984 SS: Agriculture Data Science

Applications

Generate labels

Unlabeled



Labeled

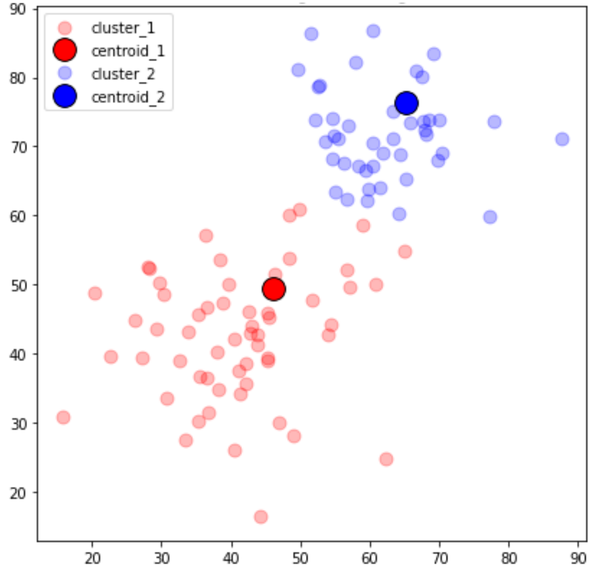
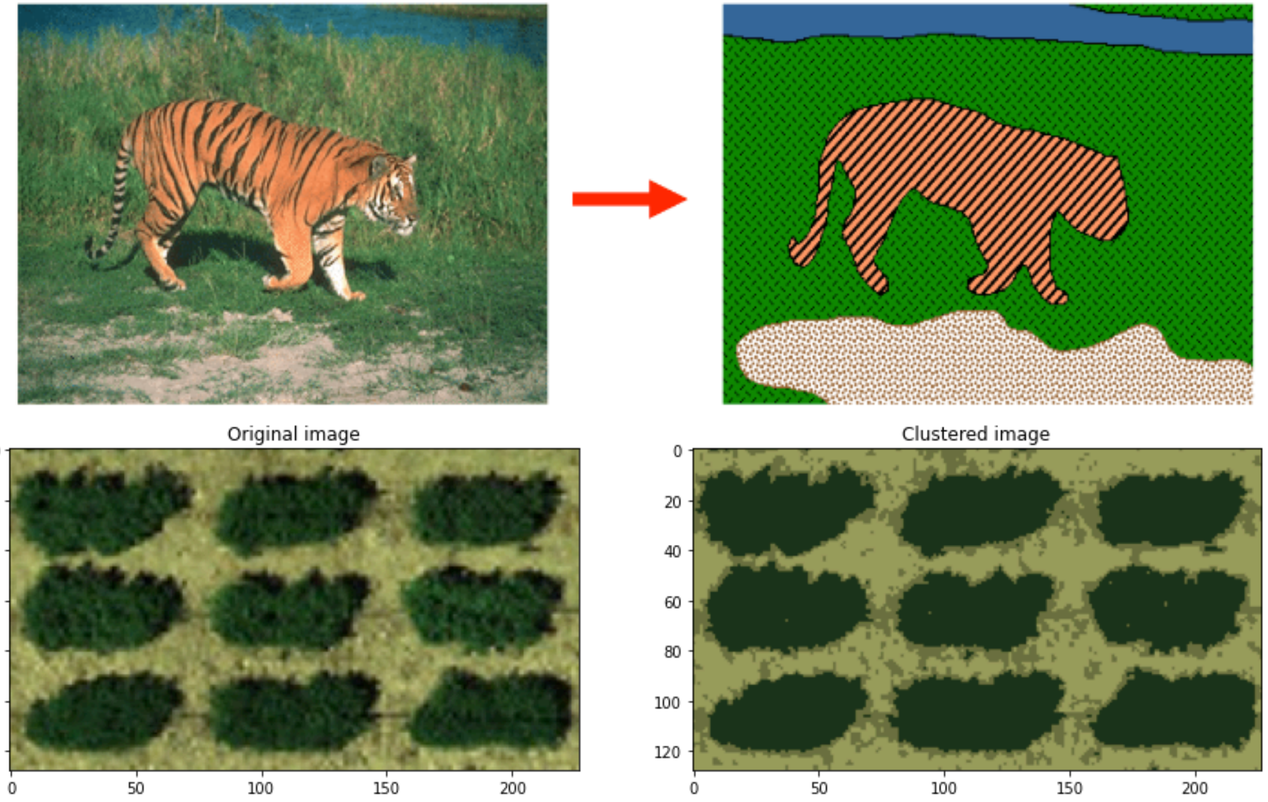


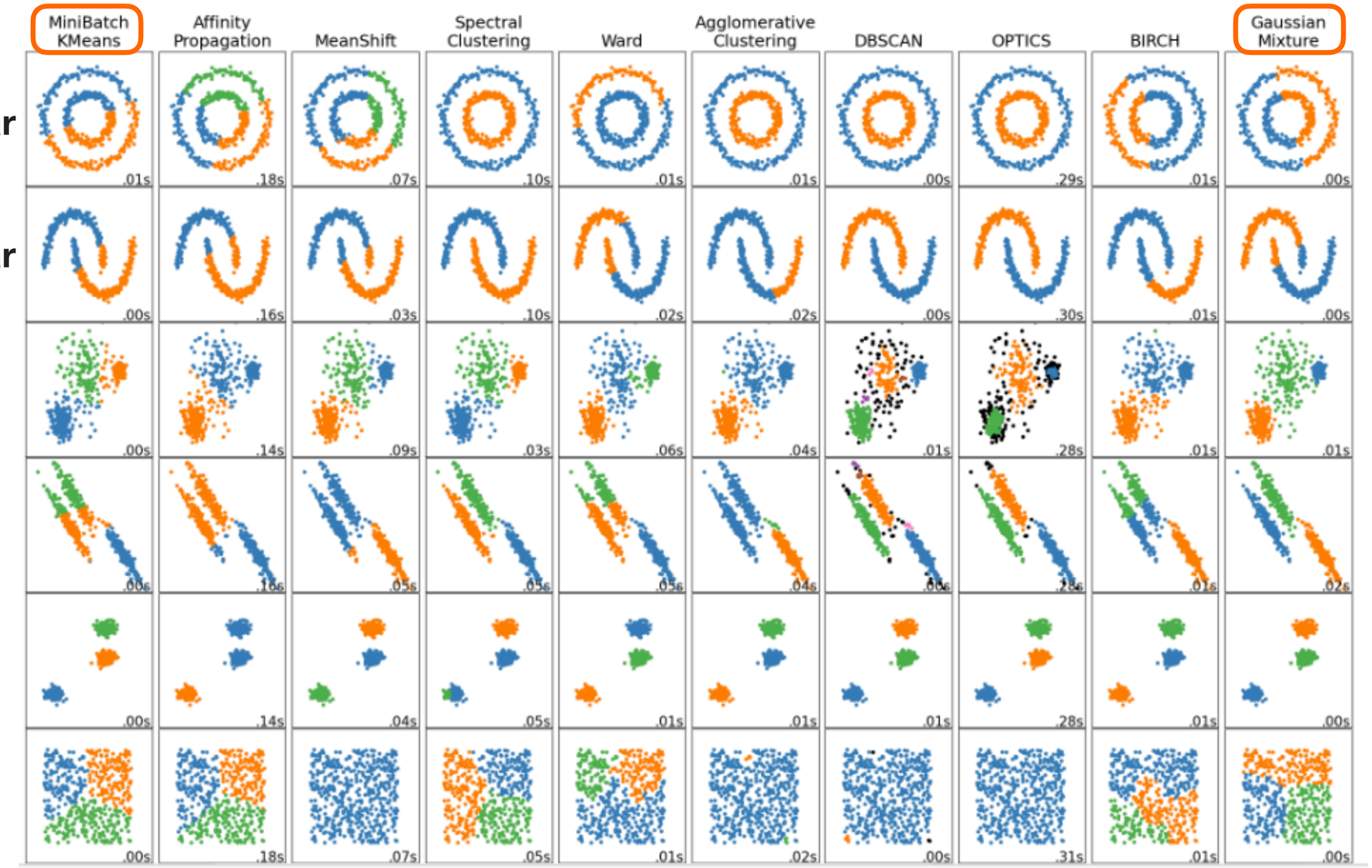
Image segmentation



Clustering Algorithm

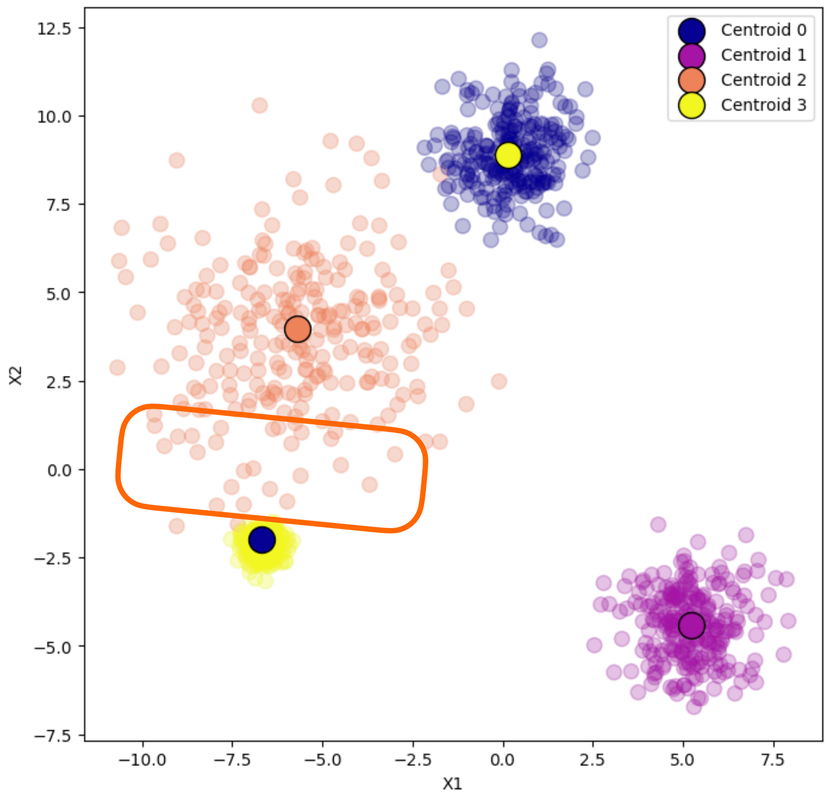
Algorithm

Problems

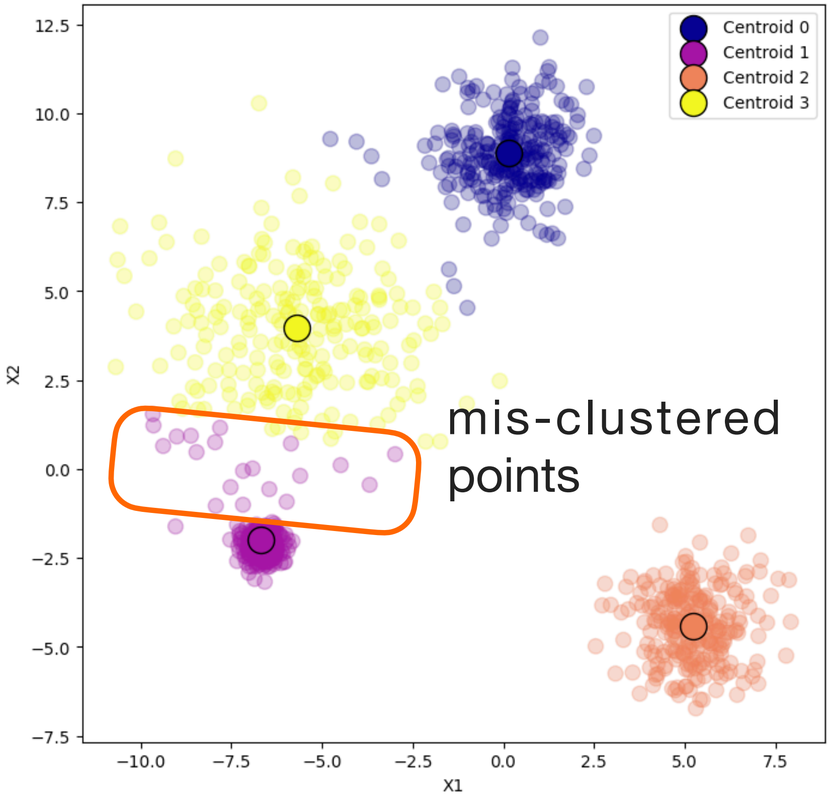


Clustering Algorithm

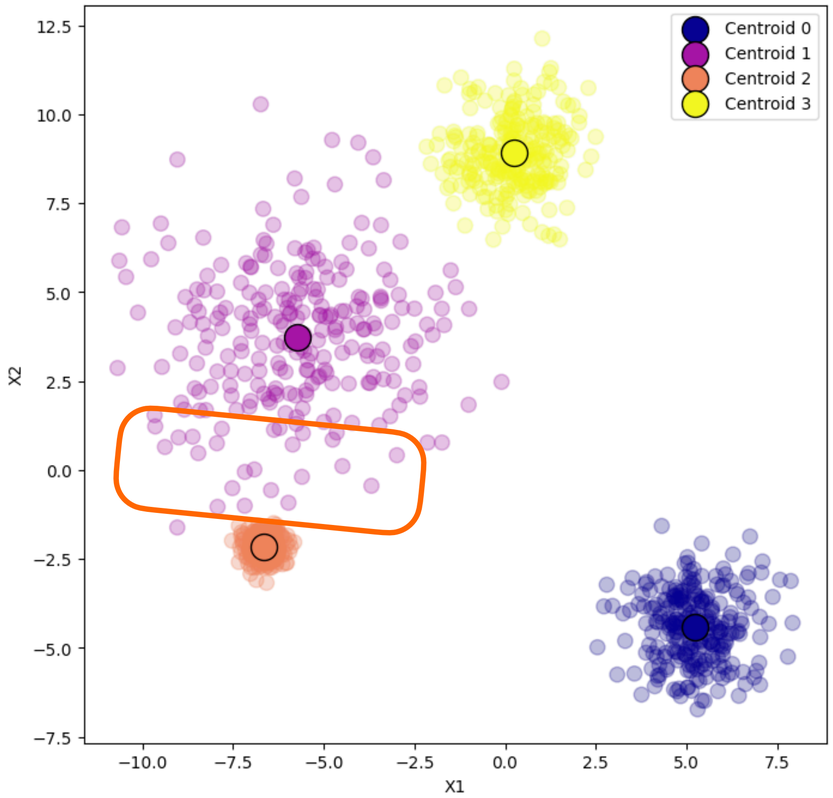
Ground Truth



K-Means



Gaussian Mixture Model



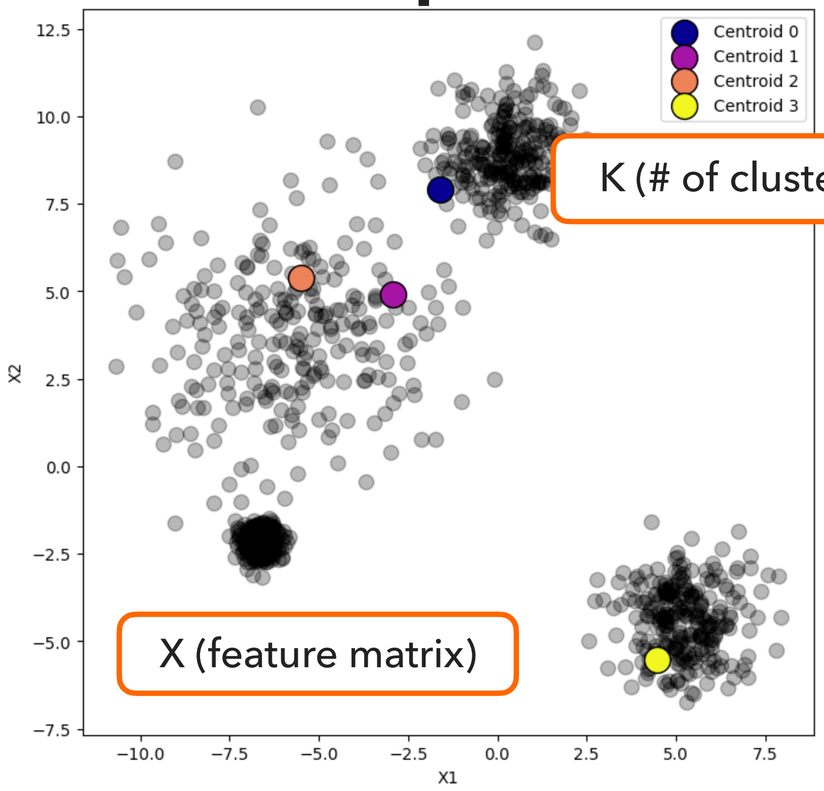
K-Means Clustering

K-means clustering algorithm

K-means clustering is a simple and popular clustering algorithm. It is a centroid-based algorithm, which means that it tries to find the center of each cluster. The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity. The results of the K-means clustering algorithm are:

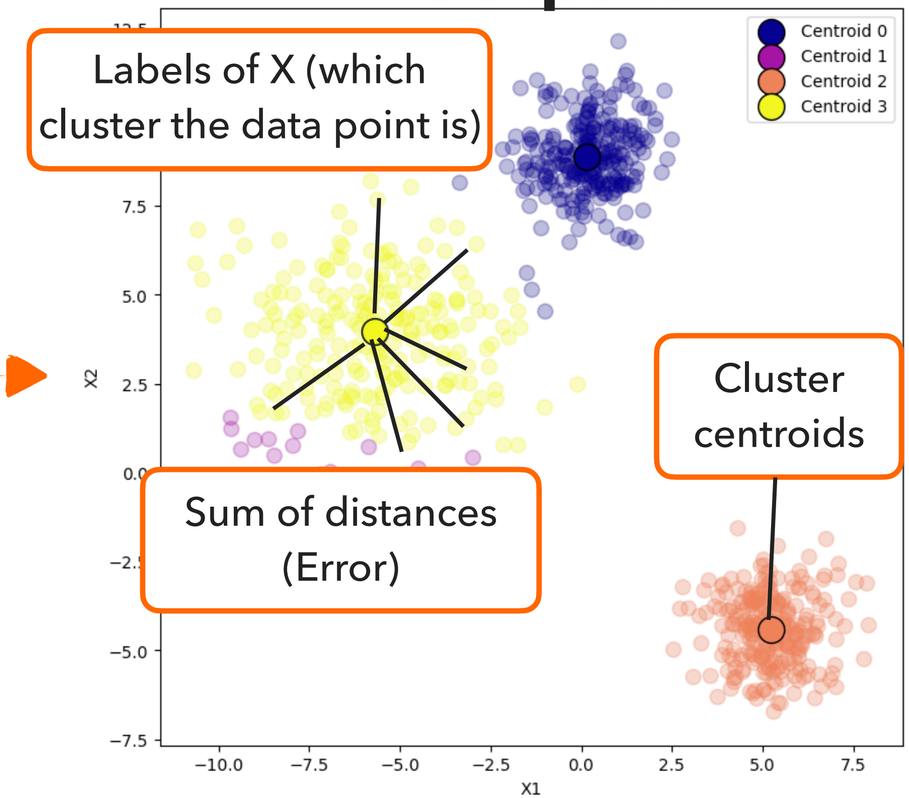
- The centroids of the K clusters, which can be used to label new data
- Labels for the training data (each data point is assigned to a single cluster)
- The sum of squared distances between data points and their cluster centroids (error)

Inputs



K-means clustering

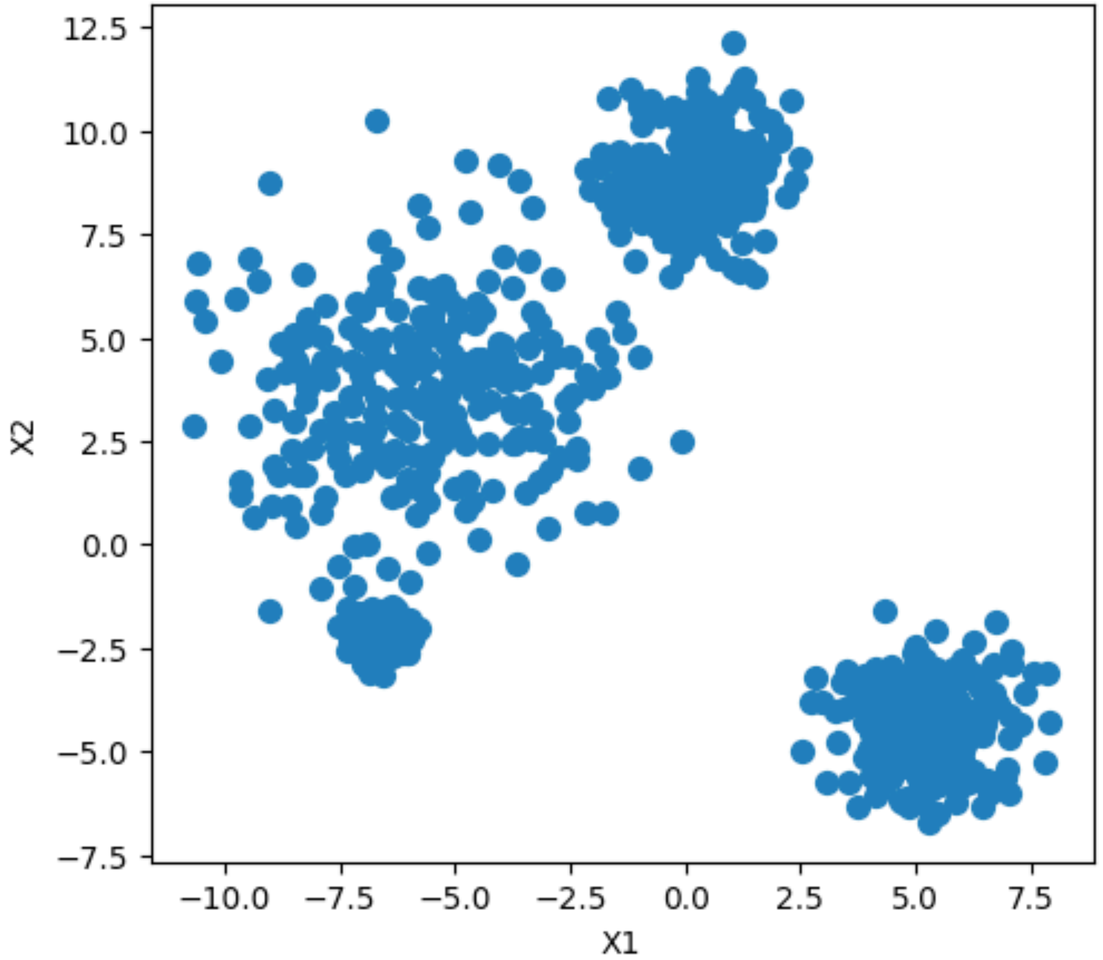
Outputs



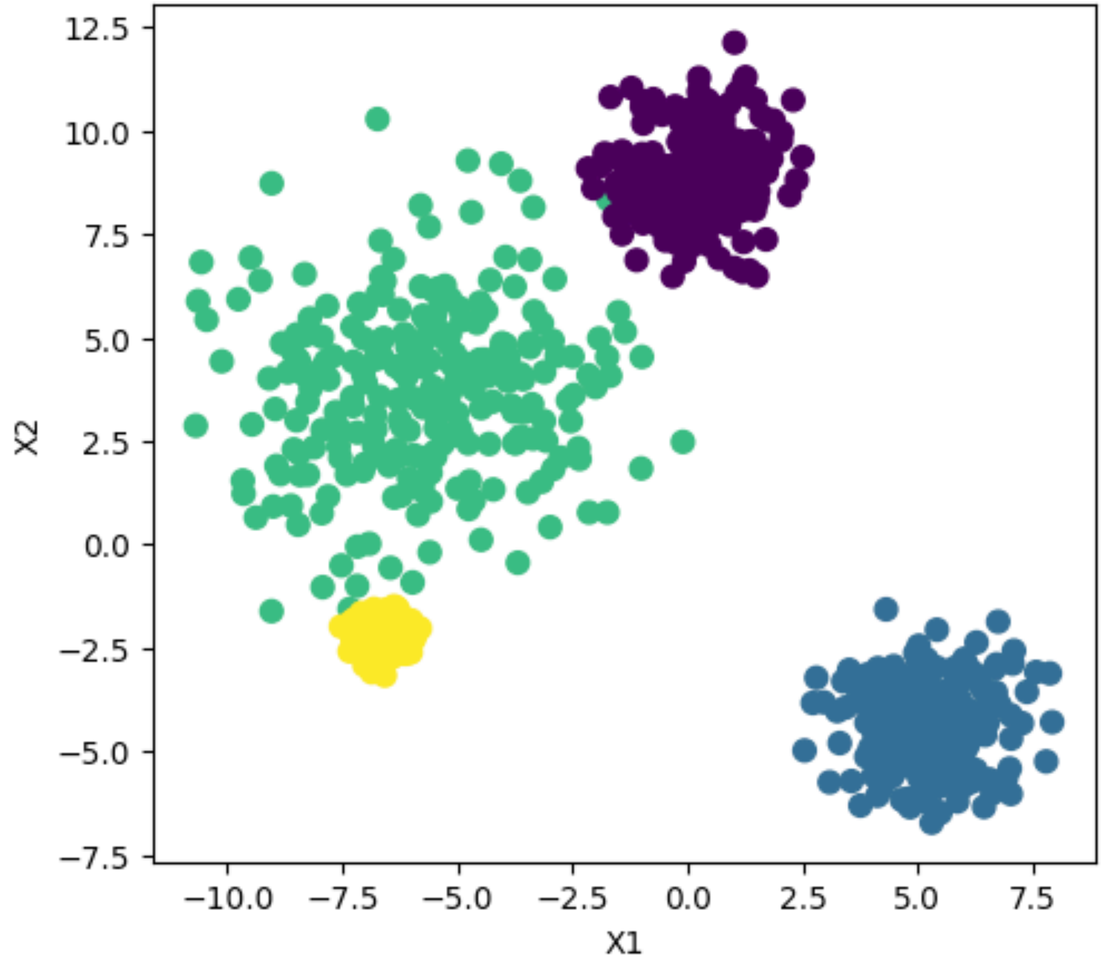
K-Means Clustering - Data Simulation

A dataset sampled from four clusters

This is what the algorithm sees



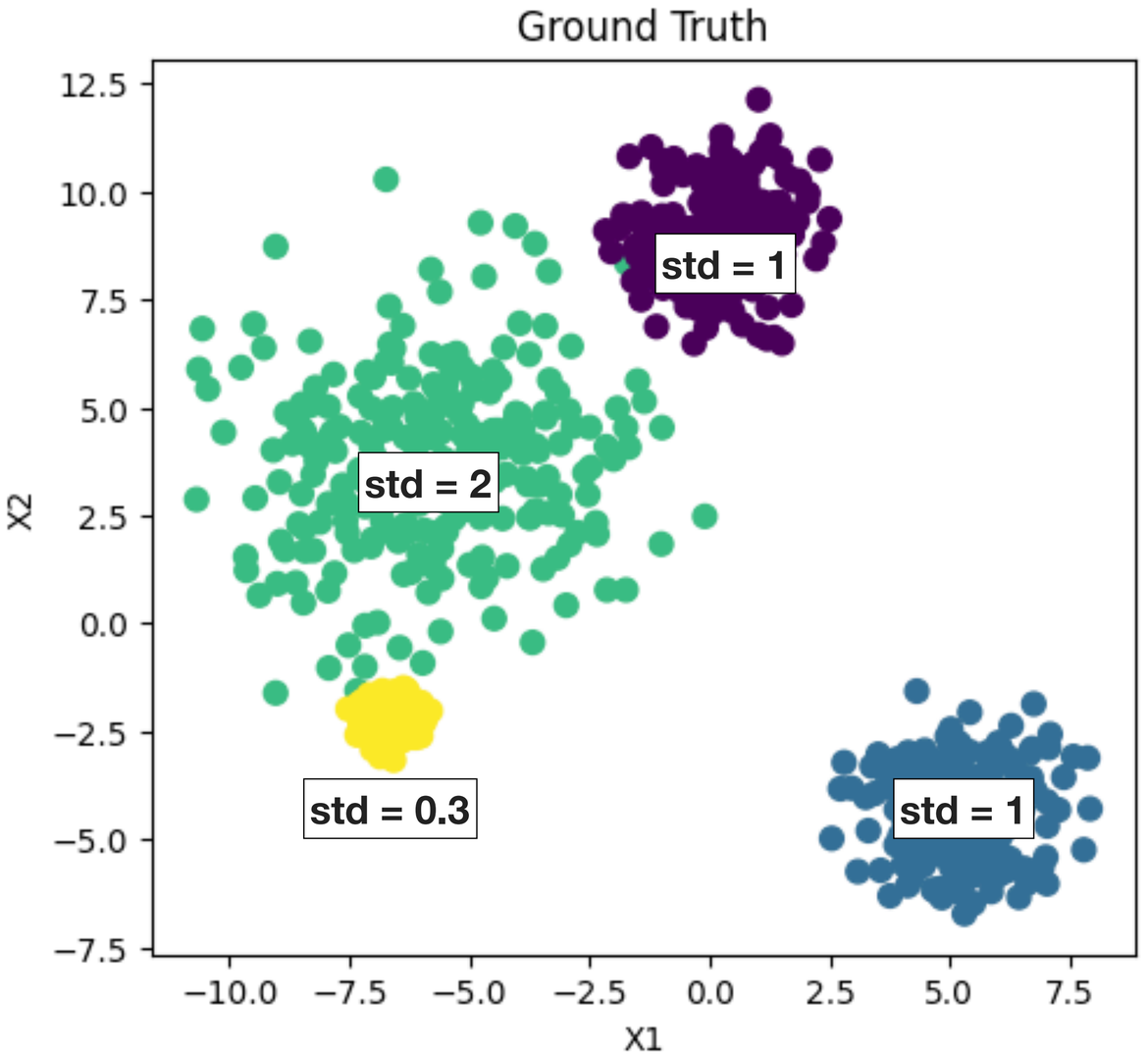
Learning target



K-Means Clustering - Data Simulation

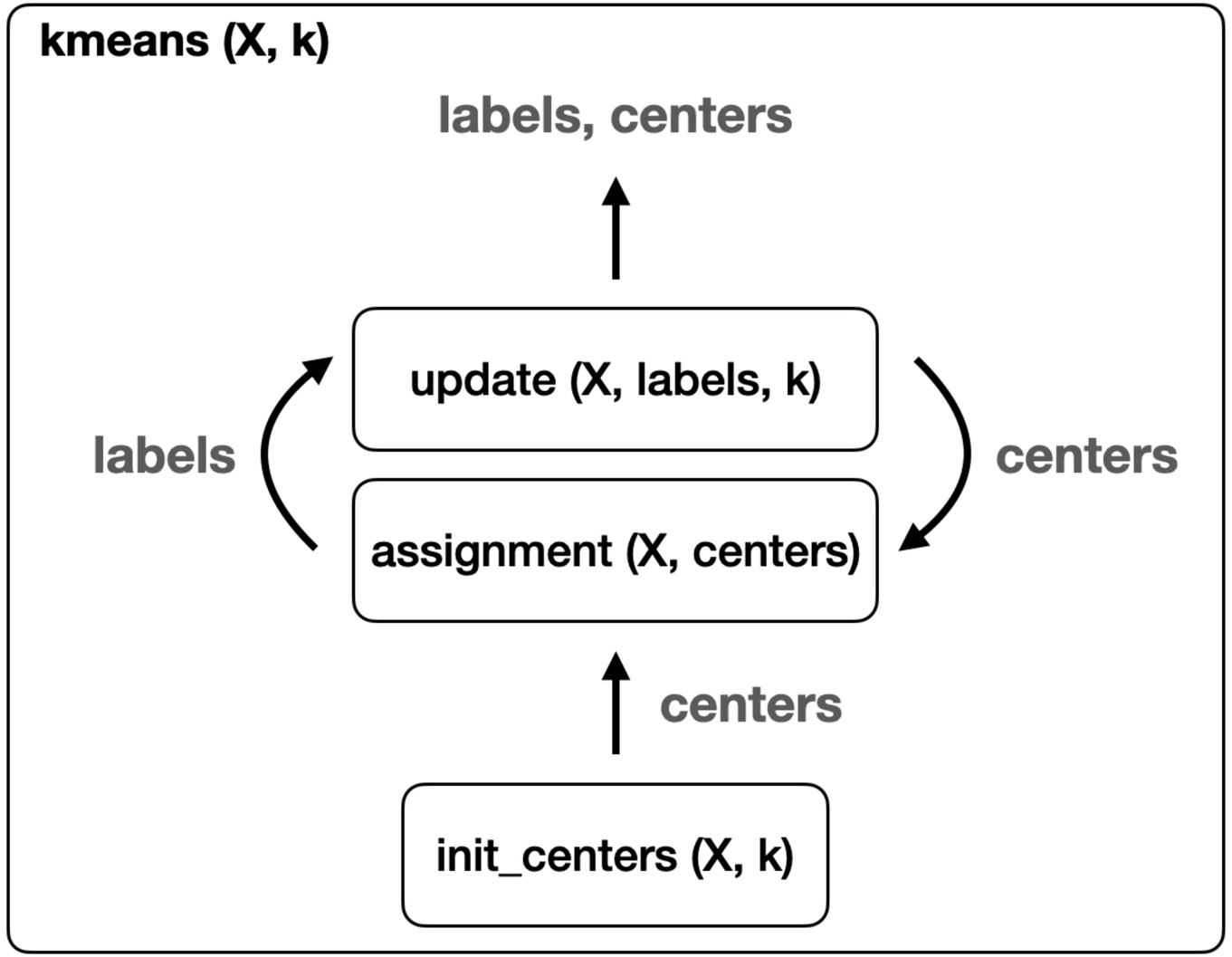
Python implementation

```
N = 1000 # 1000 samples
k = 4 # simulate 4 clusters
p = 2 # 2 features (x1 and x2)

X, y = make_blobs(
        n_samples=N,
        centers=k,
        n_features=p,
        cluster_std=[1, 1, 2, .3],
        random_state=23)
```



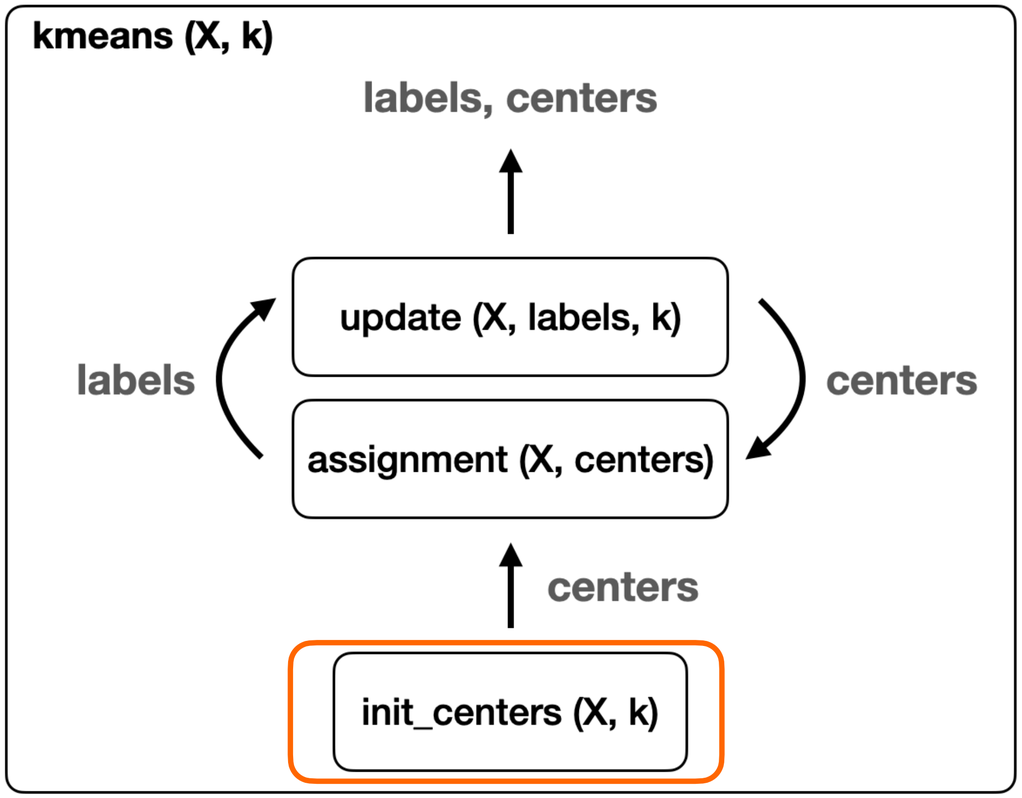
K-Means Clustering - Algorithm

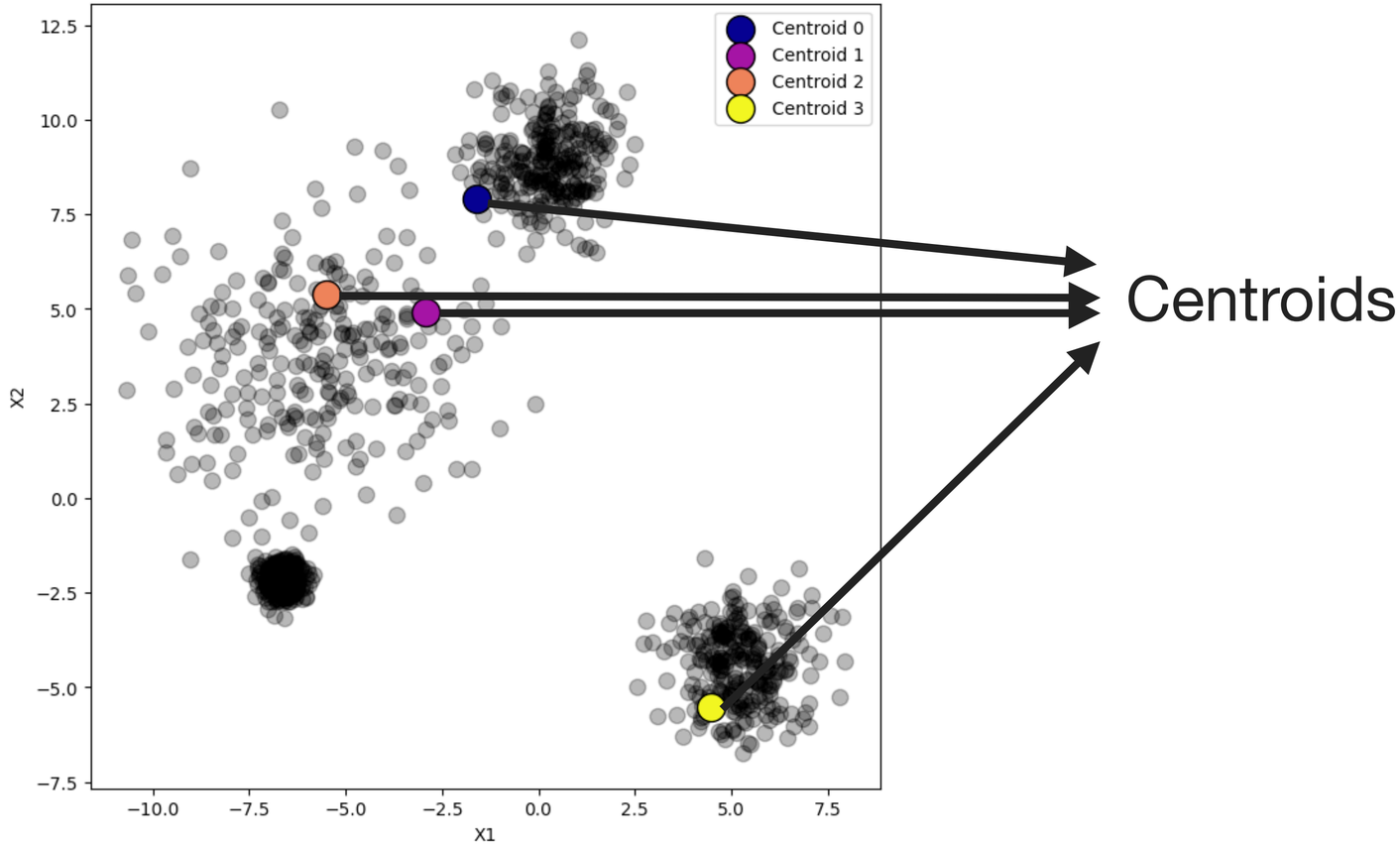
Clearly define the inputs and outputs of each step



K-Means Clustering - 1. Centroids

Randomly choose four data points as initial centroids



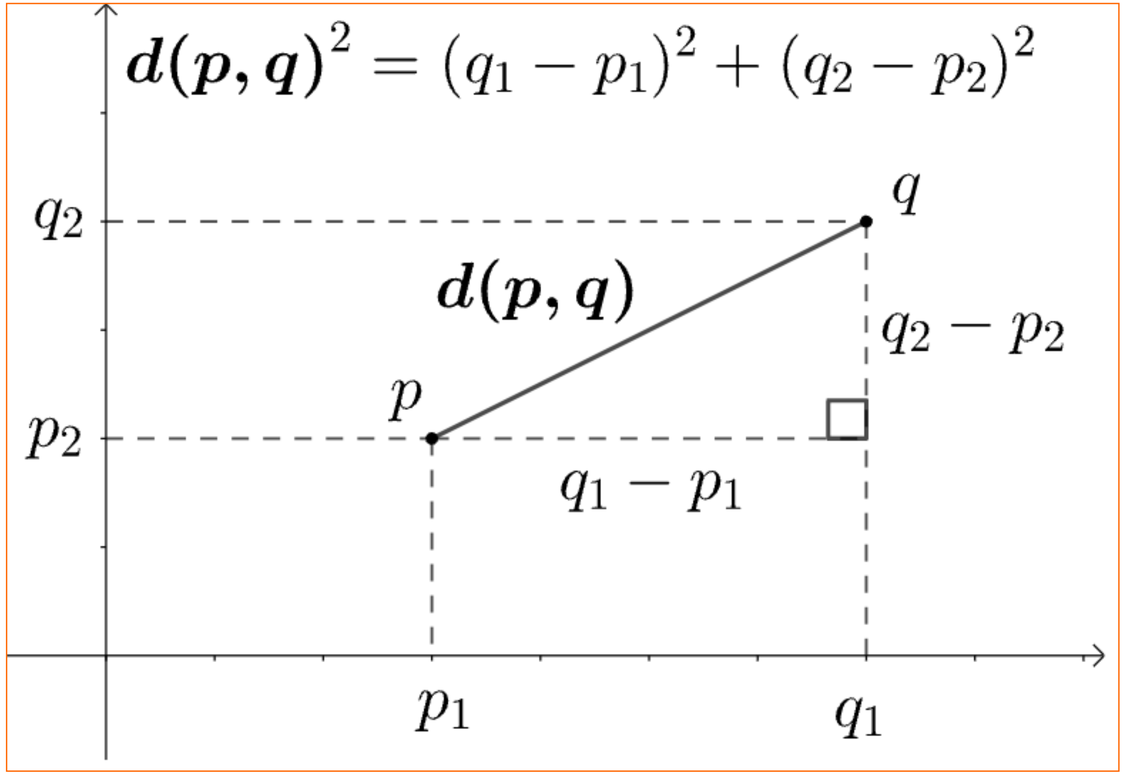


K-Means Clustering - Define Error

Euclidean Distance

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

It is a vector!



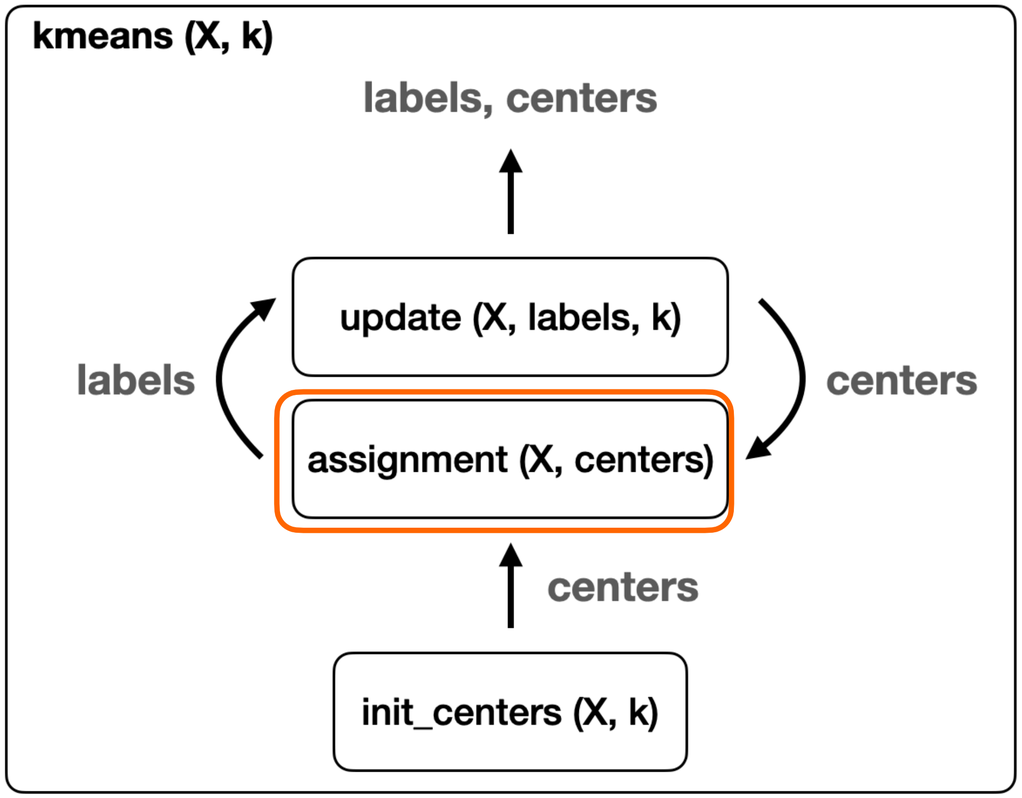

```
# define the evaluation function
def euclidean_distance(p1, p2):
    return np.sum((p1 - p2) ** 2) ** .5
```

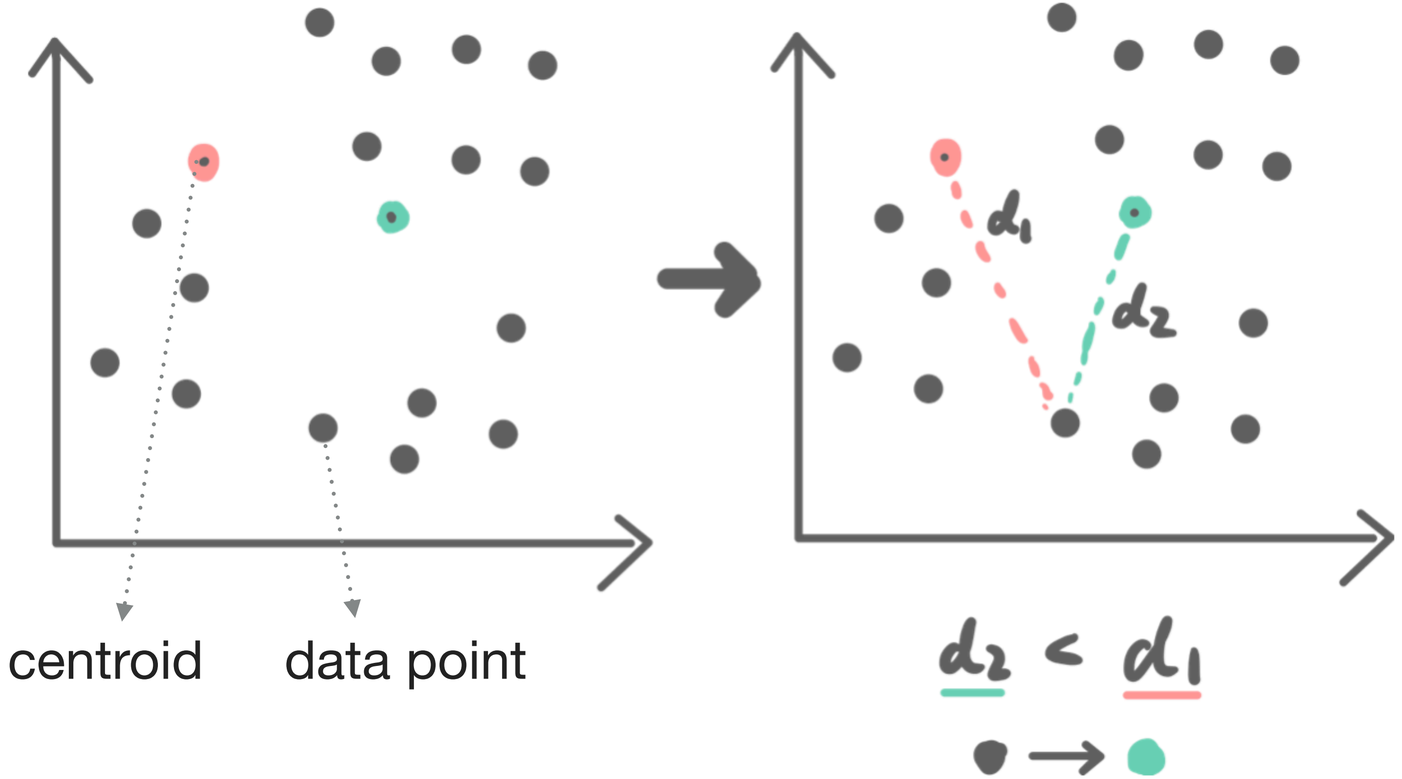
The error can be defined differently based on the research context. e.g., genetic distance.

https://upload.wikimedia.org/wikipedia/commons/5/55/Euclidean_distance_2d.svg

K-Means Clustering - 2. Assignment

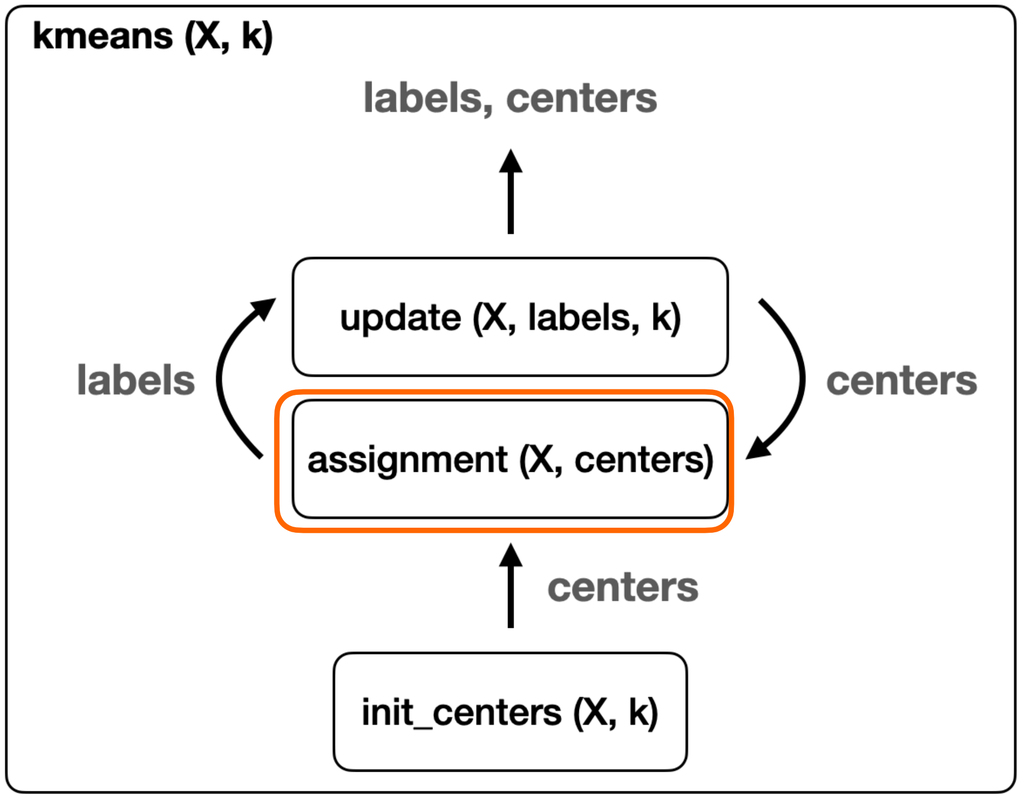
Assign data point to its nearest centroid





K-Means Clustering - 2. Assignment

Assign data point to its nearest centroid



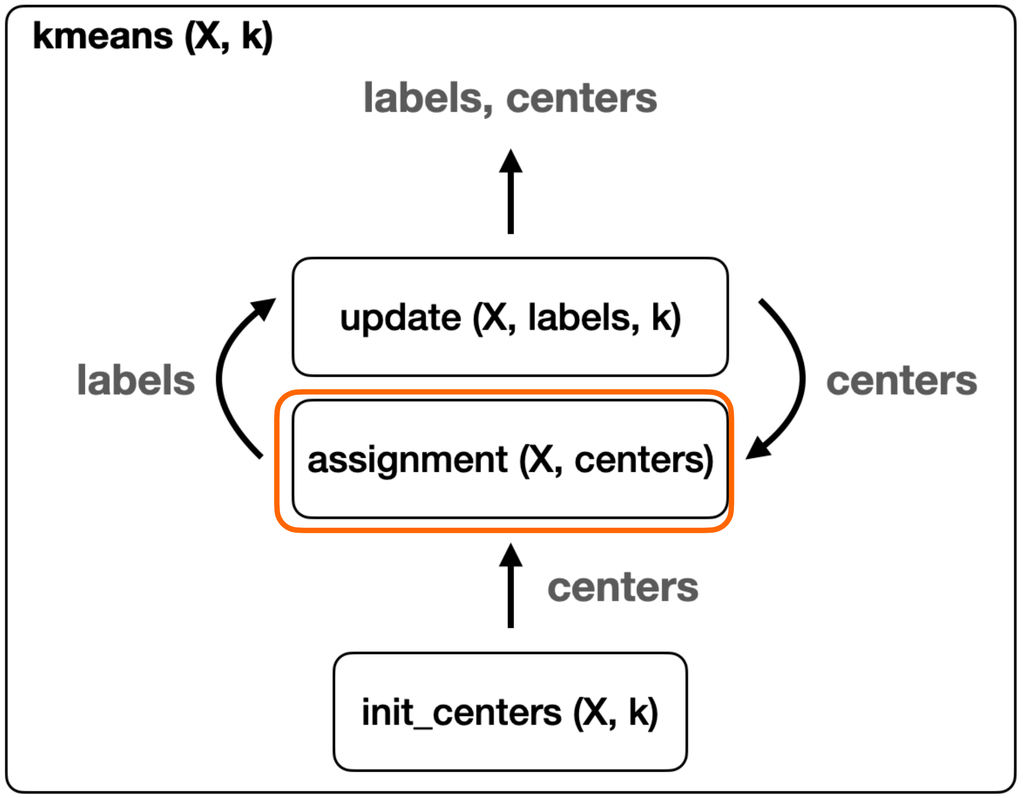

```python
# create an empty matrix to keep all evaluated distances
distances = np.zeros((N, k))

# iterate through each data point
for i, x in enumerate(X):
    # iterate through each cluster
    for j, center in enumerate(centroids):
        distances[i, j] = euclidean_distance(x, center)

# print the first ten distances
print("Distances:")
print(distances[:10])
print()

# this will return the position of the smallest values
labels = np.argmin(distances, axis=1)
```

Example: [0.3, **0.1**, 0.5, 0.3] -> 1

```python
# print the first ten labels
print("Assigned cluster: \n", labels[:10])
```

K-Means Clustering - 2. Assignment

Results of the first ten data points



Distances

N by 4 matrix

Clusters (4)

Data points (N)

```
[[12.41205425 10.91232208 13.17514998  2.95363661]
 [ 4.7322066   1.6436576   1.77190425 13.02772276]
 [ 3.99934548  7.19821409  8.58084985 16.79459887]
 [11.18048653  7.90594442  7.47292611 11.83475196]
 [ 4.79629454  2.87994802  0.2763631  15.0611299 ]
 [ 6.6041719   3.42226436  4.53330829 10.3926796 ]
 [ 5.30131932  2.6953981   0.96572457 14.15701745]
 [10.82429091  7.53976632  7.34280853 11.15212423]
 [11.55082143  8.27865243  7.80026454 11.9945368 ]
 [ 1.19337739  4.47383448  5.63989779 15.6629886 ]]
```

Labels

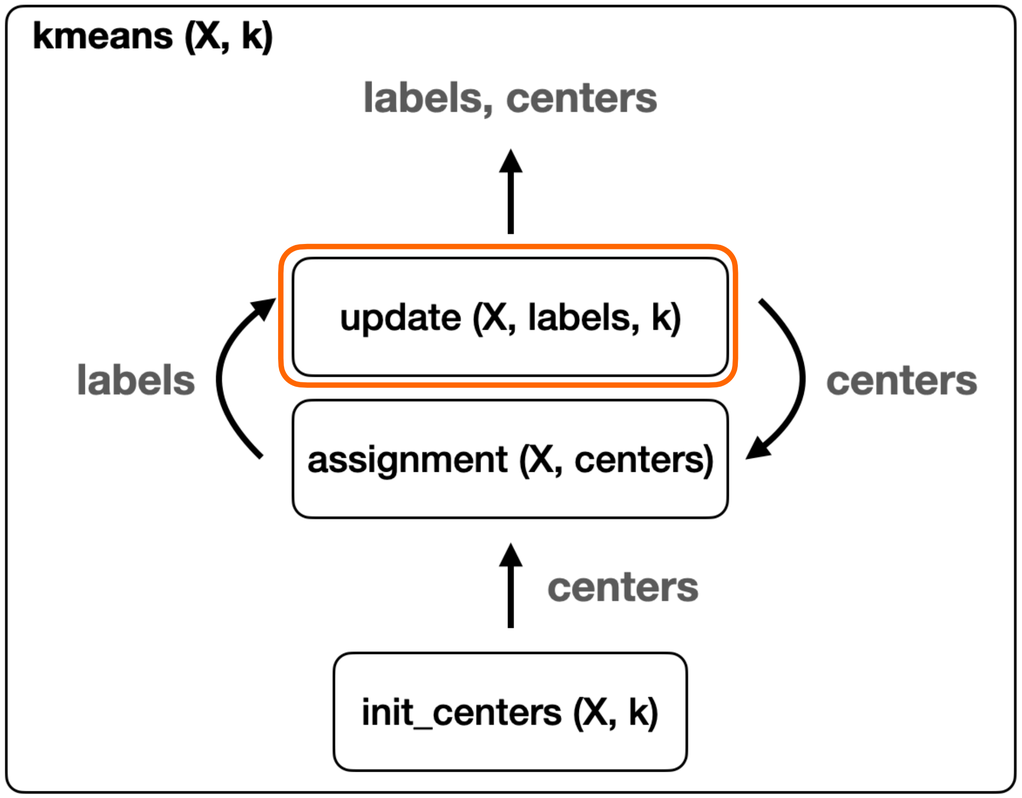
A vector with N elements

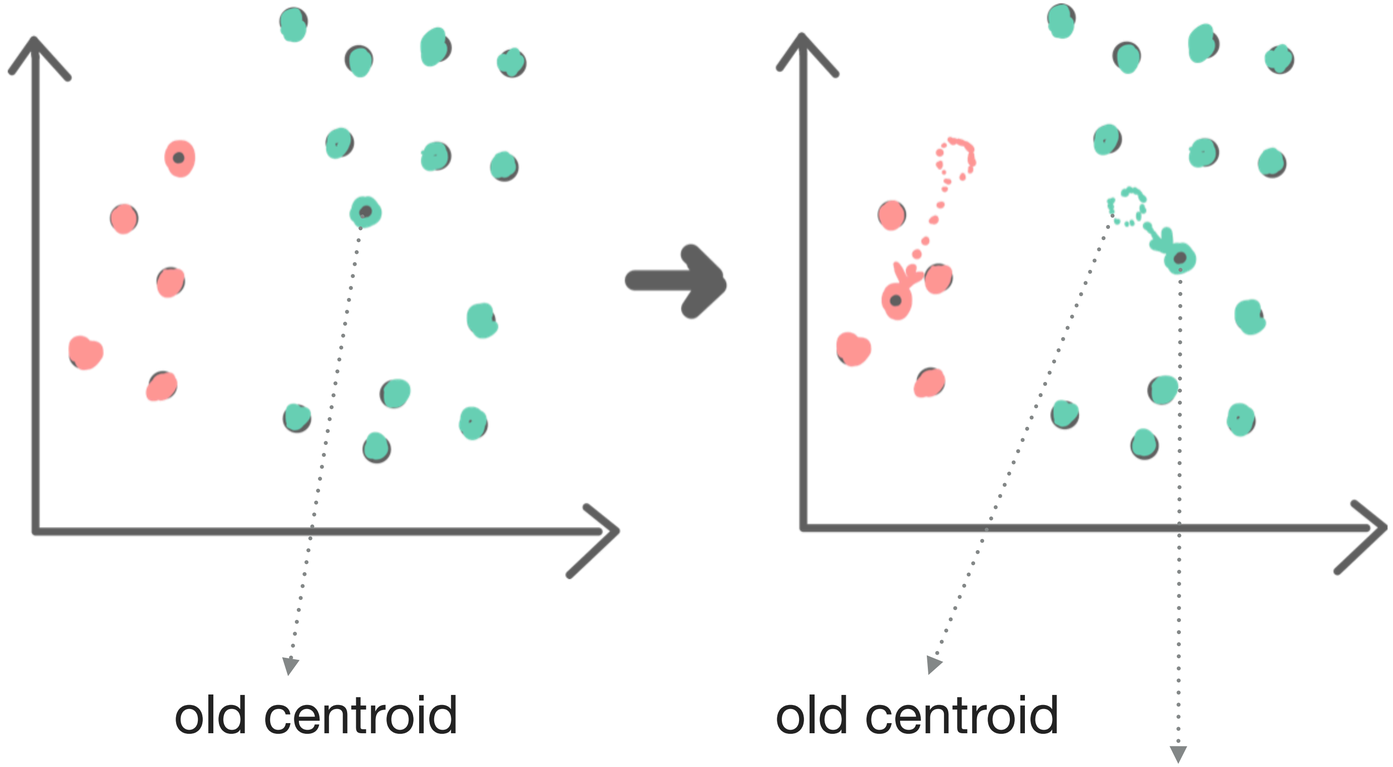
```
[3 1 0 2 2 1 2 2 2 0]
```

K-Means Clustering - 3. Update

In this stage, the centroids will be updated based on the mean of the assigned points.

This is why it's called k-"**means**" clustering

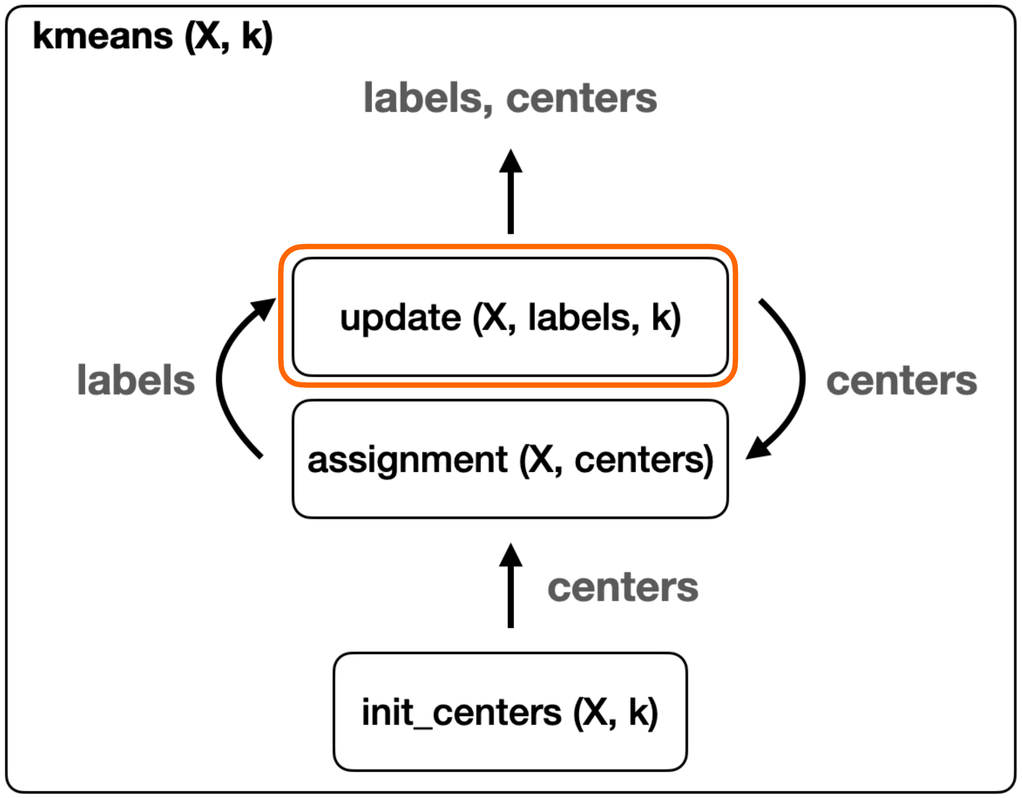




K-Means Clustering - 3. Update

In this stage, the centroids will be updated based on the mean of the assigned points.

This is why it's called k-"**means**" clustering




```
# create an empty matrix to keep the new centroids
new_centroids = np.zeros((k, 2))

# iterate over each cluster
for i in range(k):
    new_centroids[i] = X[labels == i].mean(axis=0)
```

```
New center:
 [[ 0.1696372   8.91666155]
 [-3.14397455  3.3749224 ]
 [-6.66036728  0.31953993]
 [ 5.22054811 -4.40112225]]
```

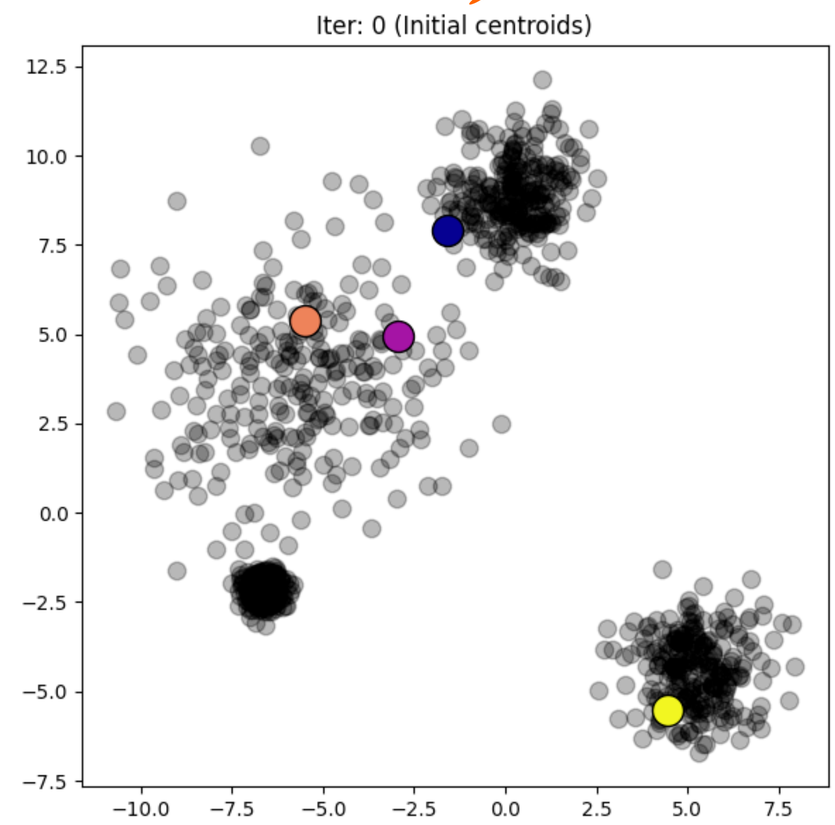
X-axis coordinate

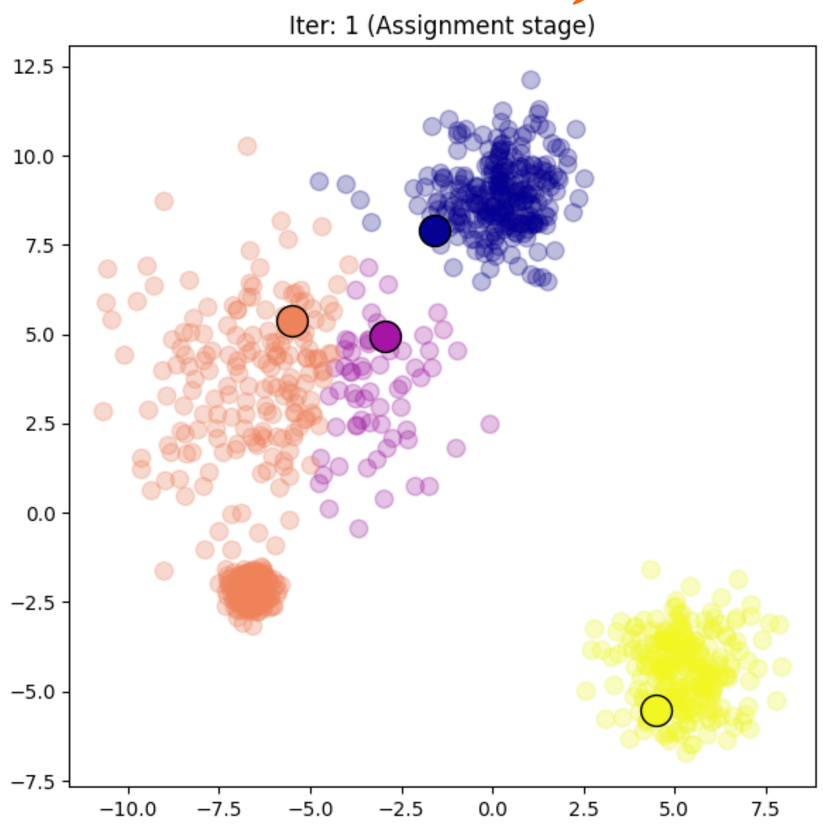
Y-axis coordinate

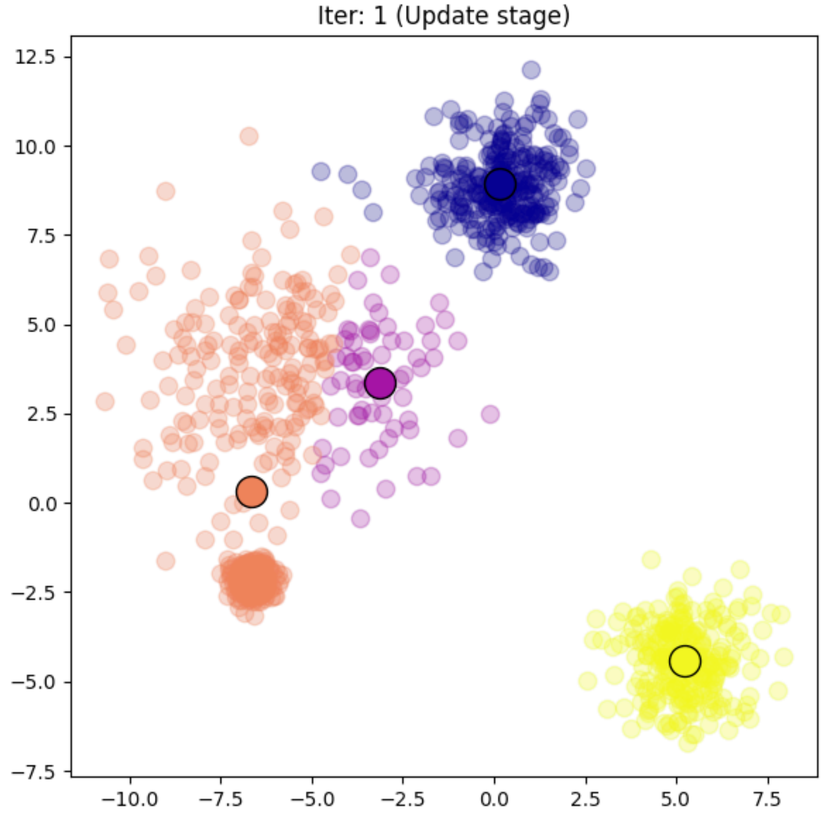
K-Means Clustering - Visualization

Assignment

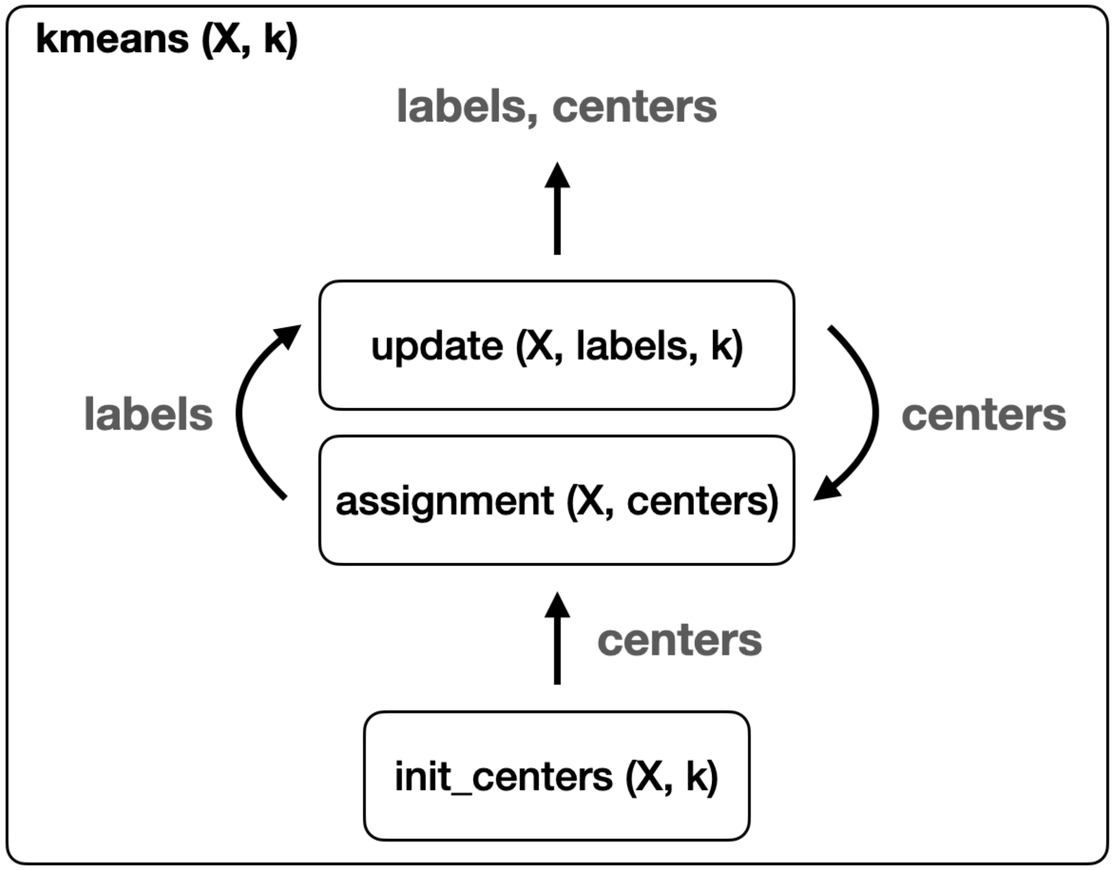
Update







K-Means Clustering - Complete Code

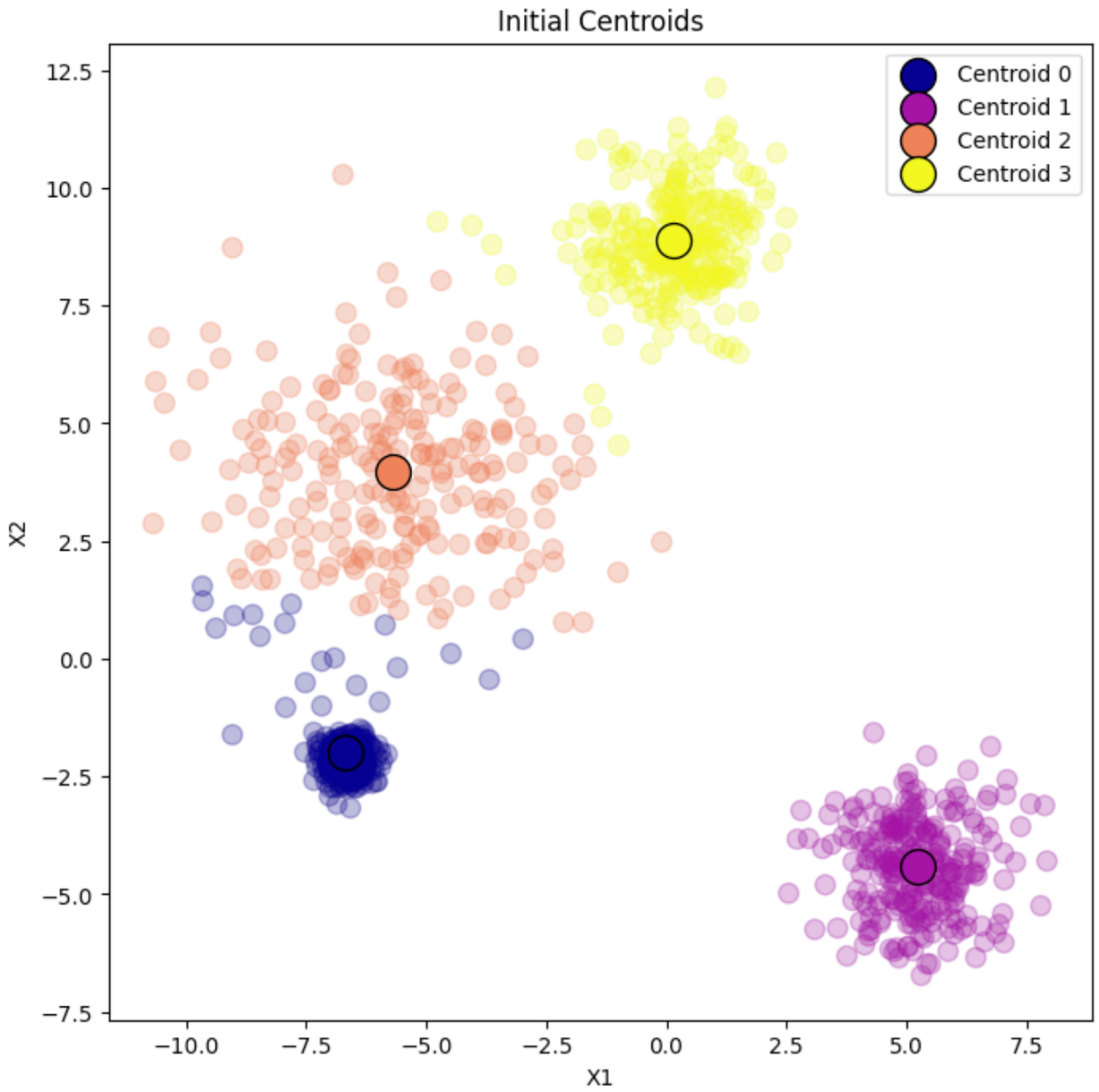



```python
def init_center(X, k):
    n = X.shape[0]
    return X[np.random.choice(n, k)]

def euclidean_distance(p1, p2):
    return np.sum((p1 - p2) ** 2) ** .5

def assignment(X, centroids):
    N, k = X.shape[0], centroids.shape[0]
    distances = np.zeros((N, k))
    for i, x in enumerate(X):
        for j, center in enumerate(centroids):
            distances[i, j] = euclidean_distance(x, center)
    return np.argmin(distances, axis=1)

def update(X, labels, k):
    N, p = X.shape
    centroids = np.zeros((k, p))
    for i in range(k):
        centroids[i] = X[labels == i].mean(axis=0)
    return centroids

def kmeans(X, k, max_iter=10):
    centroids = init_center(X, k)
    for i in range(max_iter):
        labels = assignment(X, centroids)
        centroids = update(X, labels, k)
    return centroids, labels
```

K-Means Clustering - Complete Code


```
centroids, labels = kmeans(X=X, k=4, max_iter=10)

print("Centroids: \n", centroids)
print("Labels: \n", labels)
```

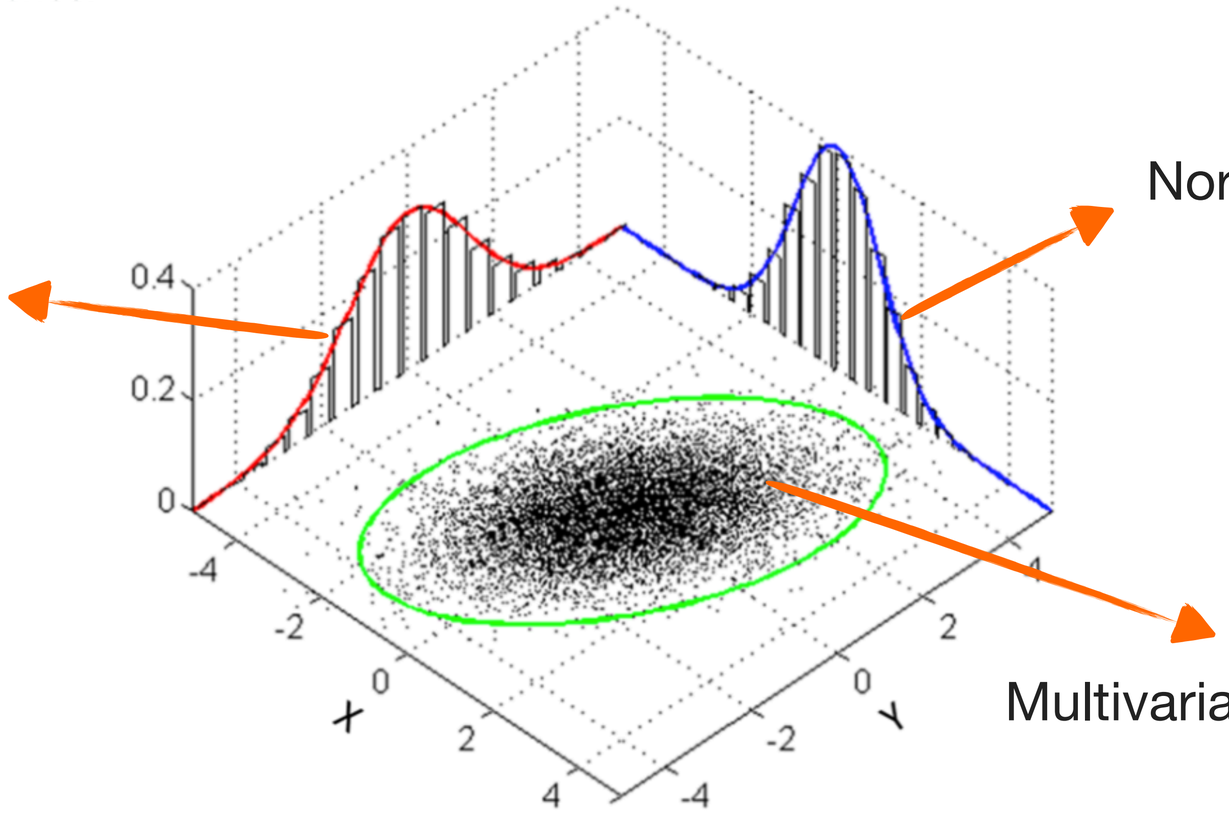
```
Output exceeds the size limit. Open the full output data in a text editor
Centroids:
 [[ 0.15275043  8.87221527]
 [-6.68493063 -1.97354619]
 [ 5.22054811 -4.40112225]
 [-5.69636389  3.96446855]]
Labels:
 [2 3 0 1 3 3 3 1 1 0 0 0 3 2 2 1 1 1 2 2 0 1 1 1 0 0 1 2 0 0 0 3 2 2 3 1 1
 1 3 1 0 2 2 1 3 3 0 2 3 1 1 1 3 1 0 1 0 0 1 2 0 0 1 3 1 3 2 2 2 1 3 0 0 2
 3 3 0 2 3 1 0 1 3 3 0 0 2 1 1 3 0 2 3 0 1 0 2 2 3 2 3 1 3 1 2 3 1 2 3 3 0
 2 3 2 3 3 1 3 1 2 0 1 2 1 2 1 0 3 0 2 1 3 0 1 3 2 2 1 3 1 2 0 0 1 1 2 2 3
 2 2 0 2 0 0 3 2 1 2 1 0 3 1 0 1 3 3 2 0 3 2 1 2 3 2 0 1 0 0 3 1 2 2 2 1 3
 1 0 3 3 2 2 1 2 1 2 3 3 0 3 2 2 0 1 1 0 0 3 0 3 0 0 3 0 1 0 0 3 1 0 2 1 1
 0 1 0 3 0 1 1 3 0 1 1 3 3 2 3 2 3 1 0 2 0 1 1 3 1 3 3 2 0 1 2 0 1 3 0 1 0
 3 2 3 1 0 2 2 1 0 0 3 1 3 0 1 2 3 0 2 2 3 1 2 2 0 2 3 2 1 0 0 2 0 0 2 3 3
 2 3 0 2 1 1 3 1 2 1 0 3 2 0 0 2 0 0 3 3 2 0 1 1 0 1 0 2 2 0 1 3 2 3 2 2 0
 3 0 2 0 0 3 0 0 3 0 3 3 1 0 0 2 1 1 2 0 1 1 2 2 0 2 1 3 2 0 3 2 3 3 2 2 0
 2 3 1 3 3 1 2 0 1 2 2 0 3 3 3 3 3 0 0 1 0 2 3 3 3 3 0 0 3 2 2 2 1 0 0 2 1
 2 0 2 3 3 0 1 1 1 0 2 1 1 3 3 2 3 3 2 1 2 1 1 1 1 0 0 1 1 2 2 1 2 3 3 2 2
 0 2 1 1 0 0 0 3 3 1 0 2 2 2 3 2 1 1 0 3 1 1 2 1 3 3 1 3 1 1 0 3 2 0 0 2 0
 0 1 0 1 0 2 2 1 1 1 2 1 2 1 0 0 2 3 0 1 0 3 1 0 2 2 0 2 2 3 2 2 0 1 0 3 3
 3 2 2 3 0 1 3 0 3 0 2 3 3 3 1 0 2 3 3 0 0 0 1 1 3 0 0 1 0 3 1 1 3 0 2 2 0
 3 3 2 1 0 2 2 1 0 1 3 1 1 0 3 3 0 1 1 0 1 3 3 1 3 0 1 1 0 1 1 3 2 1 3 0 0
 3 1 0 2 0 0 1 1 3 3 2 2 0 2 2 0 1 2 0 3 2 2 0 3 2 3 2 2 2 0 0 3 3 0 1 1 2
 0 1 1 0 2 3 1 1 1 1 2 0 2 3 0 3 2 1 3 3 0 2 2 3 0 0 3 1 0 1 0 1 3 3 1 3 3
 3 2 0 0 0 0 1 1 2 0 3 2 1 0 2 1 1 3 3 3 0 0 1 1 3 2 1 3 2 2 3 0 3 1 0 1 1
...
 2 2 2 3 0 3 2 1 1 0 0 1 0 0 1 3 1 1 1 3 0 1 1 0 2 2 1 1 2 0 1 1 0 2 3 3 2
 1 2 2 2 0 1 0 2 2 3 3 0 2 3 1 0 1 0 0 3 0 2 1 1 3 3 2 1 2 1 0 1 0 3 0 0 0
 2 1 1 1 0 0 1 0 2 1 3 1 2 3 1 1 0 1 2 1 0 2 1 1 0 2 2 2 0 0 0 0 3 3 2 2 0
 1]
```



Gaussian Mixture Model (GMM)

Gaussian mixture model (GMM) is another approach to cluster a given dataset. Unlike K-means clustering algorithm, which has no assumption on the cluster distribution, GMM assumes each cluster following a multivariate normal (MVN) distribution. This approach is useful when the observed data shows different variation across the studied dimensions (variates). Ones can implement GMM by Expectation Maximization (EM) algorithm that leverages Bayesian Theorm to maximize the posterior probability. The posterior probability in GMM is the chance the observed data is sampled from the estiamted MVN distributions.

In this section, we will walk through each step of EM algorithm, and we will then compare the results generated from K-Means and GMM clustering approaches.



GMM - Bayesian Theorem

Bayesian Theorm is a fundamental concept in statistics and machine learning. It is used to estimate the posterior probability based on the prior and the conditional probability. First, let's recap what Bayesian theorm is about:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Hypothesis (Distributions) — Estimated

Observed data — Fixed

Priors

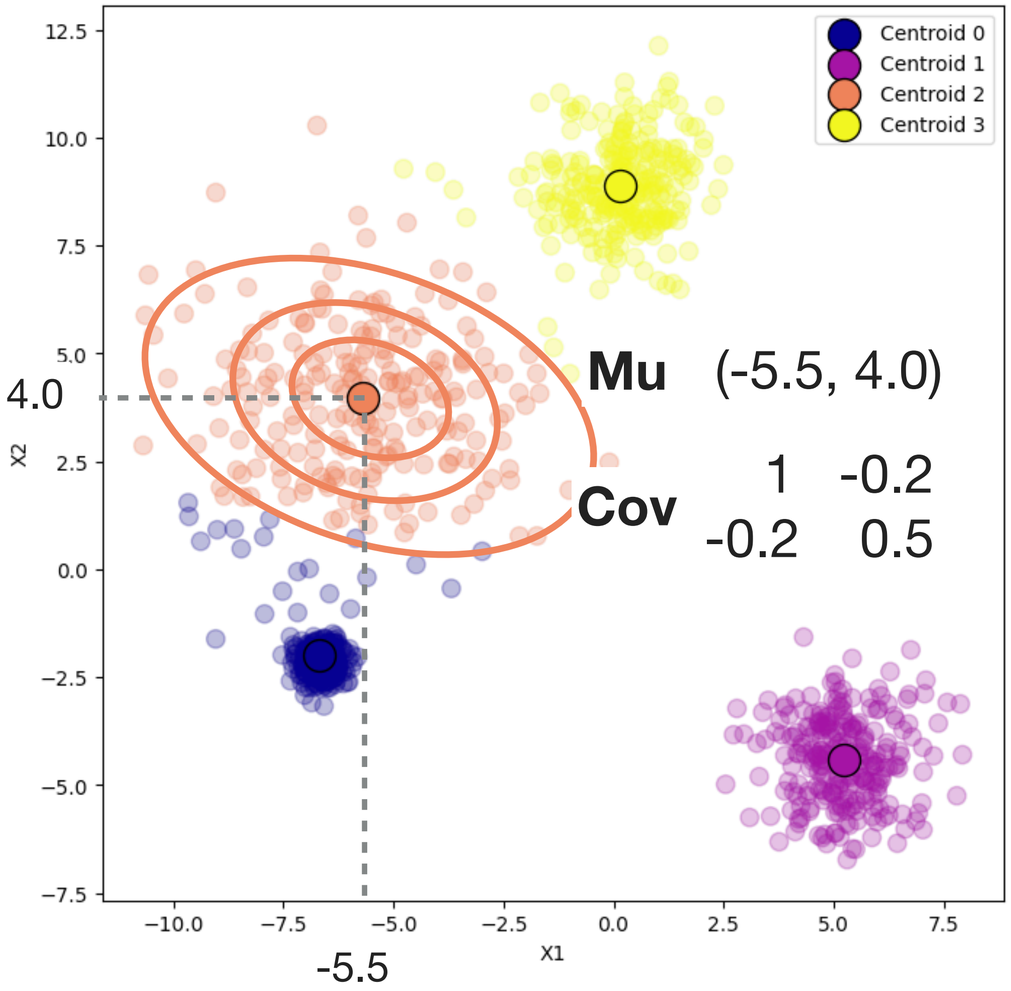
$$P(A) = P(c_j) = P(\mu_j, \Sigma_j) = \pi_j$$

Probability density function (PDF)

$$\begin{aligned} P(B \mid A) &= P(x_i \mid c_j) \\ &= P(x_i \mid \mu_j, \Sigma_j) \\ &= \frac{1}{\sqrt{det(2\pi\Sigma_j)}} \exp\left(-\frac{1}{2}(x_i - \mu_j)^T \Sigma_j^{-1}(x_i - \mu_j)\right) \end{aligned}$$

Likelihood of the observed data

$$P(B) = P(x_i) = \sum_{j=1}^{k} P(x_i \mid c_j)P(c_j)$$



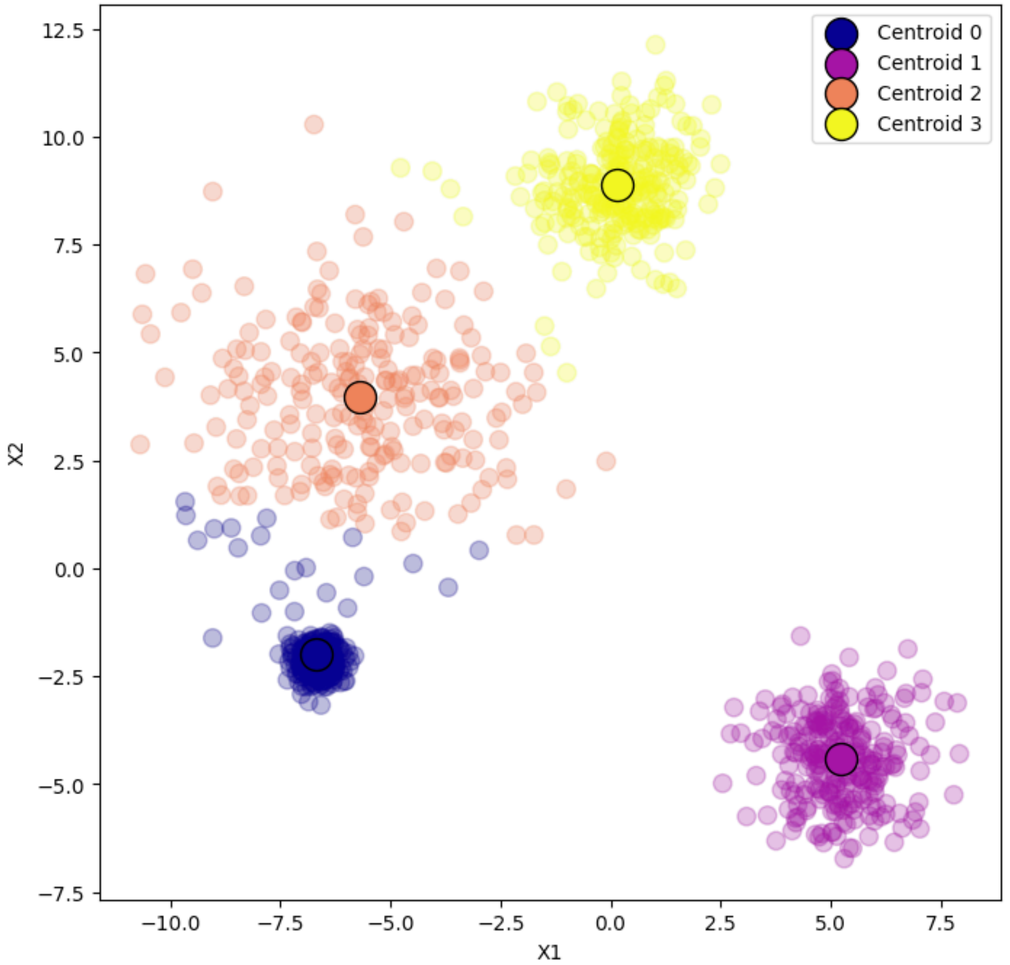
GMM - Priors

Priors

$$P(A) = P(c_j) = P(\mu_j, \Sigma_j) = \pi_j$$

	j	count	pi
	0	250	0.25
	1	200	0.2
	2	300	0.3
	3	250	0.25

We can assume a uniform pi among all clusters as priors



GMM - Probability Density Function

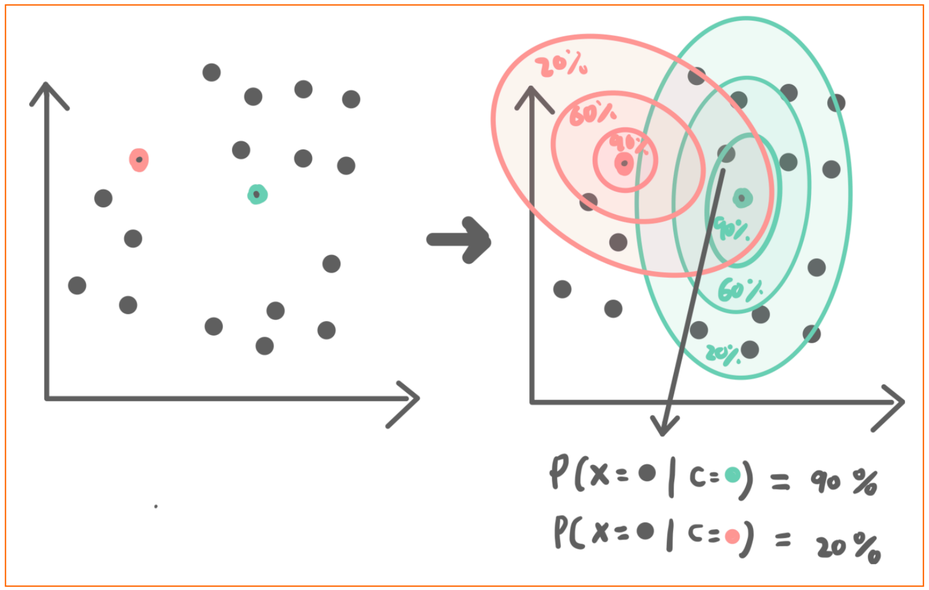
Probability density function (PDF)

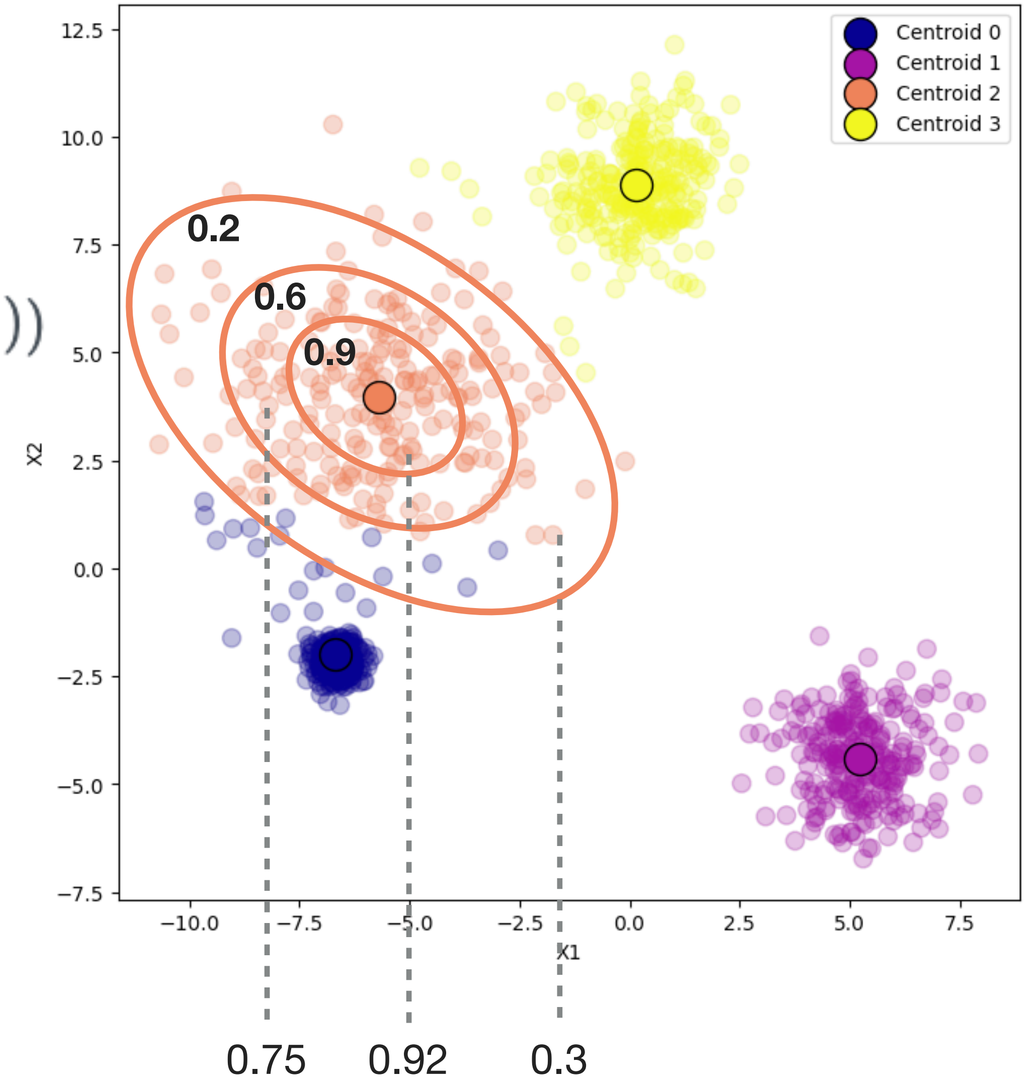
$$P(B \mid A) = P(x_i \mid c_j)$$
$$= P(x_i \mid \mu_j, \Sigma_j)$$
$$= \frac{1}{\sqrt{det(2\pi\Sigma_j)}} \exp\left(-\frac{1}{2}(x_i - \mu_j)^T \Sigma_j^{-1}(x_i - \mu_j)\right)$$

Observed data

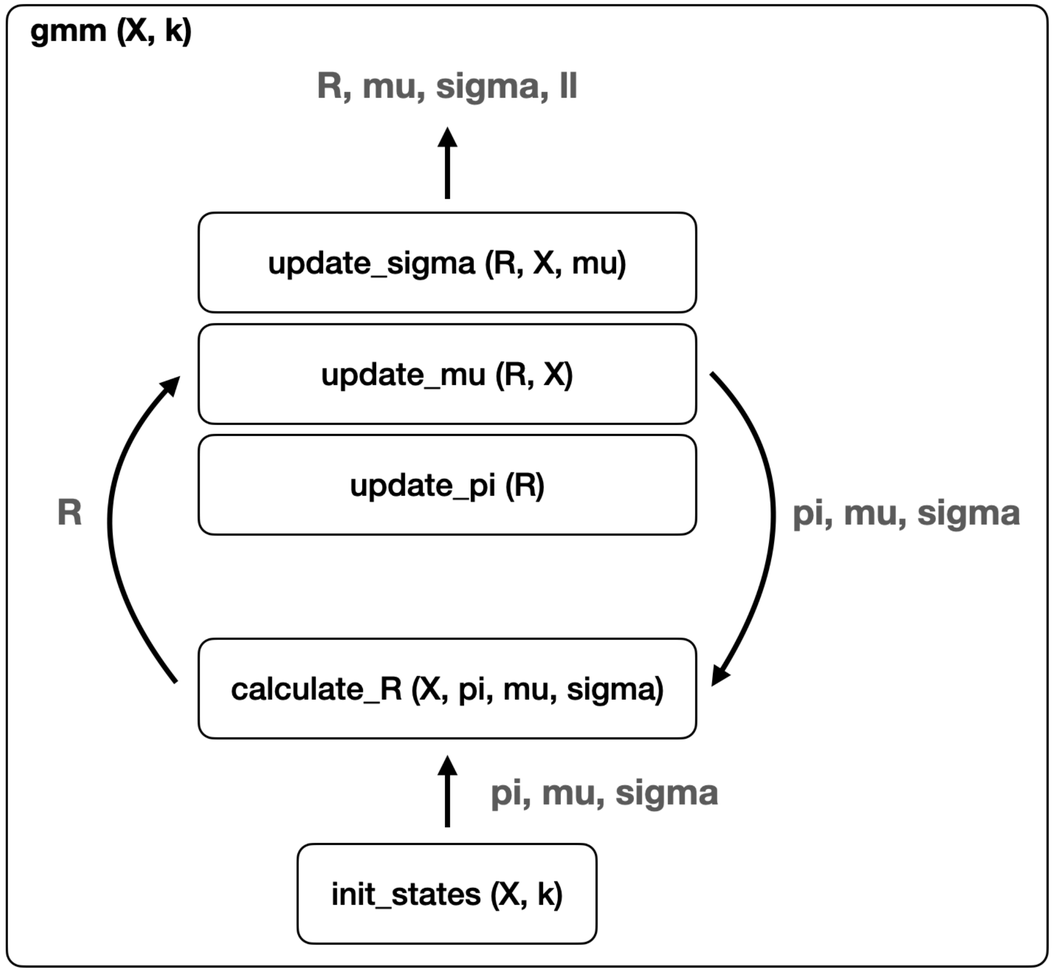
Hypothesis (Distributions)

How typically a data point x_i is sampled from the distribution c_j

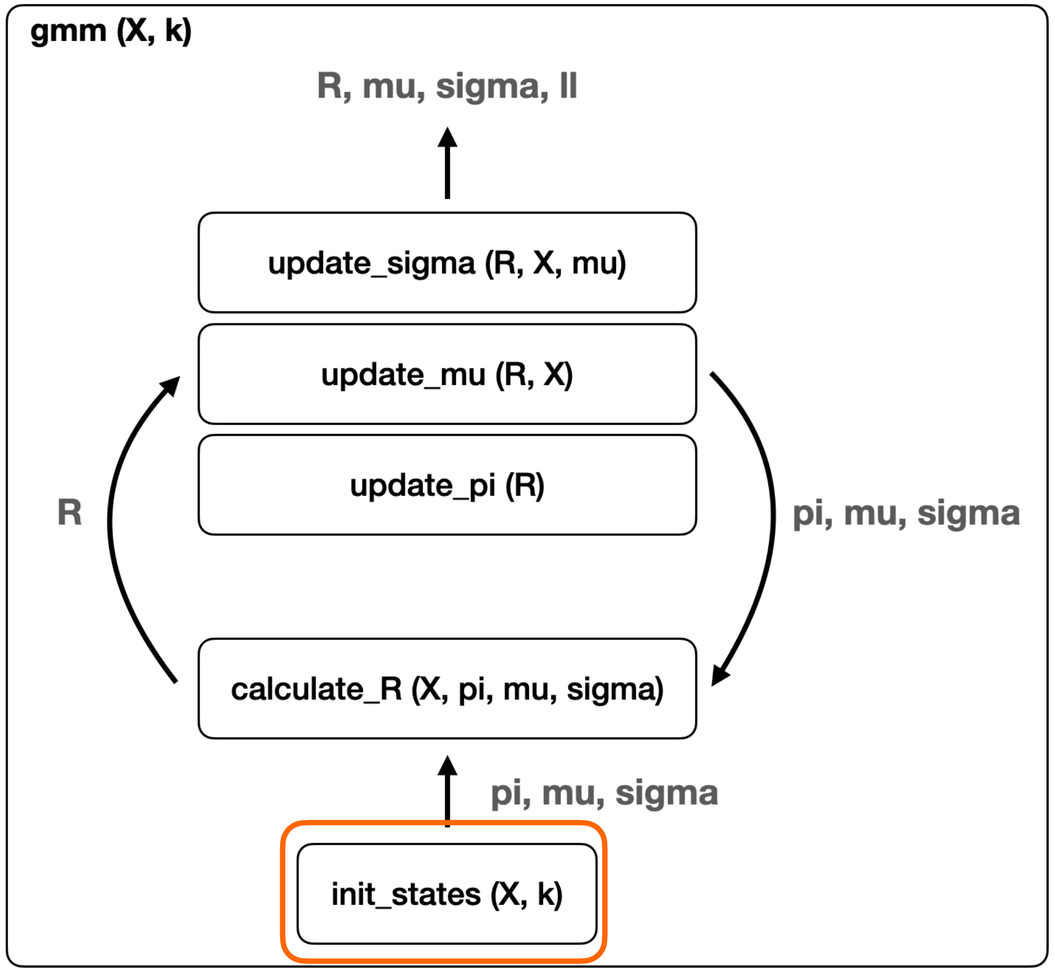




GMM - Algorithm



GMM - Initial Distributions



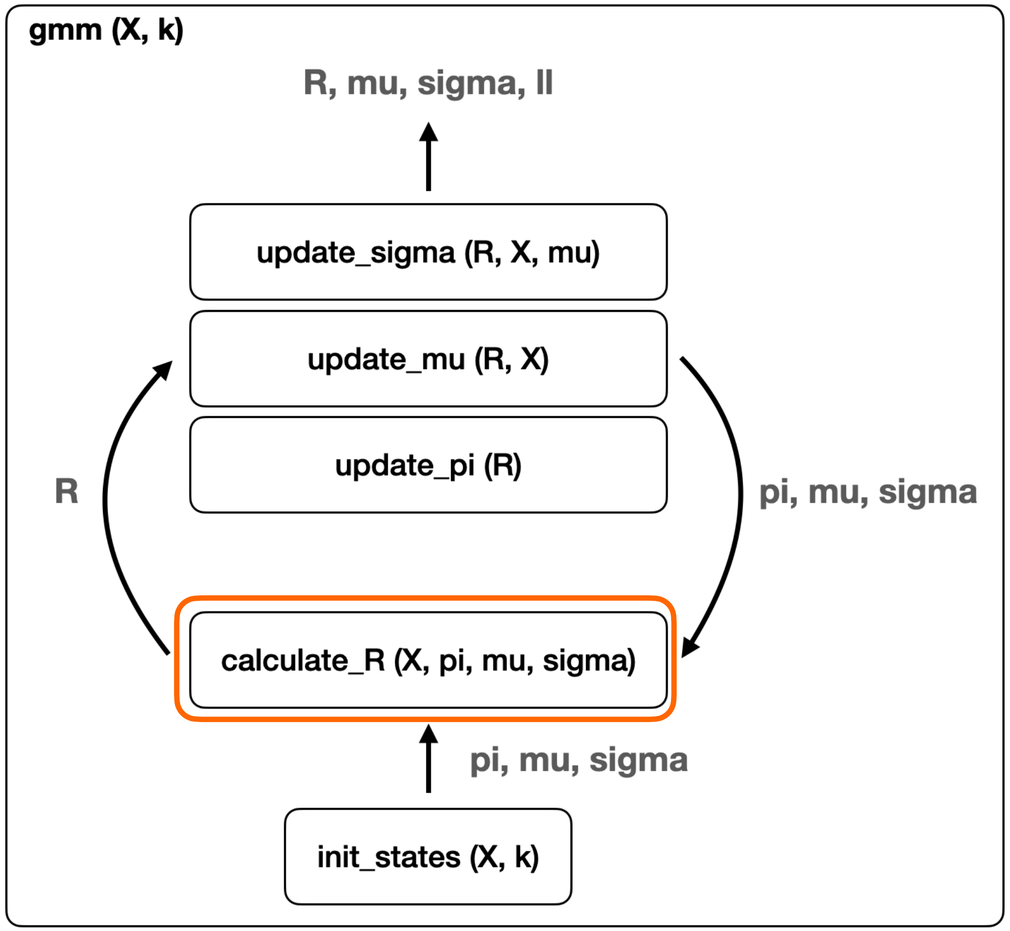
$$P(A) = P(c_j) = P(\mu_j, \Sigma_j) = \pi_j$$

where μ_j and Σ_j are the mean and (co)variance of the distribution c_j, respectively.

```
# parameters
N = 1000
k = 4

# randomly assign clusters (labels)
# use "quotients" as the initial labels
labels = np.random.random(N) // (1 / k)
pi     = [np.mean(labels == i) for i in range(k)]



# subset data by each cluster
ls_subdata = [X[labels == i]         for i       in range(k)]
ls_mu      = [np.mean(subdata, axis=0) for subdata in ls_subdata]
ls_sigma   = [np.cov(subdata.T)        for subdata in ls_subdata]
```

Obtain the **quotient**

GMM - Estimation Stage (E Stage)




```python
def calculate_R(X, pi, mu, sigma):
    """Calculate posterior probability, P(xi|cj)

    Parameters
    ----------
    X : array_like
        A NumPy array with a shape of (n, p),
        where n is the number of observations,
        and p is the data dimension.
    pi : array_like
        A NumPy array with a shape of (k,).
        Prior, P(A).
    mu : array_like
        A Numpy array with a shape of (k, p)
        The mean of the inspected MVN distribution
    sigma : array_like
        A Numpy array with a shape of (k, p, p)
        The covariance of the inspected MVN distribution

    Returns
    -------
    array_like, a NumPy array with a shape of (n, k),
    where n is the number of observations,
    and k is the number of clusters.
    """
    # create an empty R matrix
    n, k = len(X), len(mu)
    R = np.zeros((n, k))
    # numerator
    for i in range(k):
        R[:, i] = pi[i] * mvn(mean=mu[i],
                              cov=sigma[i]).pdf(X)
    # denominator
    R /= np.sum(R, axis=1)[:, None]
    return R
```

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

$$= P(c_j \mid x_i)$$

$$= \frac{P(x_i \mid c_j)P(c_j)}{\sum_{j=1}^{k} P(x_i \mid c_j)P(c_j)} = r_{ij}$$

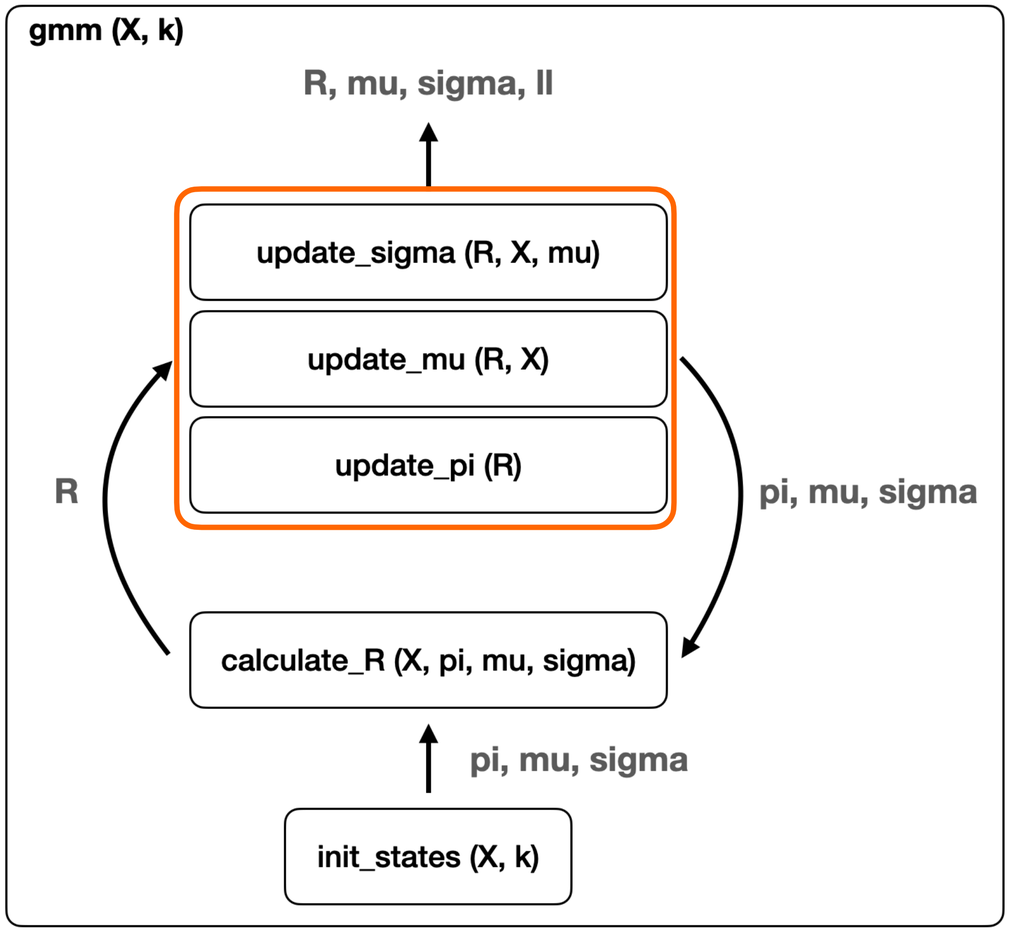
R matrix

4 (k)

1000 (n)

```
[[0.23168997 0.23742123 0.27997582 0.25091298]
 [0.25895036 0.24766292 0.22314003 0.27024669]
 [0.23025957 0.29411632 0.24118739 0.23443672]
 [0.26293503 0.24013226 0.21777755 0.27915516]
 [0.26281677 0.24685173 0.22169722 0.26863428]
 [0.25652262 0.24539485 0.2267431  0.27133943]
 [0.26220488 0.24519259 0.22066192 0.27194061]
 [0.26239167 0.24072164 0.22026067 0.27662601]
 [0.2630115  0.24018805 0.21725698 0.27954348]
 [0.24460083 0.27450037 0.23474062 0.24615818]]
```

GMM - Maximization Stage (M Stage)



R matrix

k

n

```
[[0.23168997 0.23742123 0.27997582 0.25091298]
 [0.25895036 0.24766292 0.22314003 0.27024669]
 [0.23025957 0.29411632 0.24118739 0.23443672]
 [0.26293503 0.24013226 0.21777755 0.27915516]
 [0.26281677 0.24685173 0.22169722 0.26863428]
 [0.25652262 0.24539485 0.2267431  0.27133943]
 [0.26220488 0.24519259 0.22066192 0.27194061]
 [0.26239167 0.24072164 0.22026067 0.27662601]
 [0.2630115  0.24018805 0.21725698 0.27954348]
 [0.24460083 0.27450037 0.23474062 0.24615818]]
```

```python
def update_pi(R):
    return R.mean(axis=0)

def update_mu(R, X):
    n, k = R.shape
    new_mu = [0] * k
    for j in range(k):
        num = np.sum(X * R[:, j][:, None], axis=0)
        den = np.sum(R[:, j])
        new_mu[j] = num / den
    return new_mu
```

$$P(c_j) = \frac{\sum_{i=1}^{n} r_{ij}}{\sum_{j=1}^{k}\sum_{i=1}^{n} r_{ij}}$$

$$\mu_j = \frac{\sum_{i=1}^{n} r_{ij} x_i}{\sum_{i=1}^{n} r_{ij}}$$

```python
def update_sigma(R, X, mu):
    n, k = R.shape
    new_sigma = [0] * k
    for j in range(k):
        # weighted mean squared devication
        weighted_MSD = [np.matmul((x - mu[j])[:, None], (x - mu[j])[:, None].T) * r for x, r in zip(X, R[:, j])]
        new_sigma[j] = np.sum(weighted_MSD, axis=0) / np.sum(R[:, j])
    return new_sigma
```

$$\Sigma_j = \frac{\sum_{i=1}^{n} r_{ij}(x_i - \mu_j)^T(x_i - \mu_j)}{\sum_{i=1}^{n} r_{ij}}$$

GMM - Maximum Likelihood Estimation

$$
\begin{aligned}
arg\max_{\theta} \ell(\theta) &= arg\max_{\theta} \ln \prod_{i=1}^{n} P(x_i \mid \theta) \\
&= arg\max_{\theta} \sum_{i=1}^{n} \ln P(x_i \mid \theta) \\
&= arg\max_{\theta} \sum_{i=1}^{n} \ln \sum_{j=1}^{k} P(x_i \mid c_j) P(c_j)
\end{aligned}
$$

where

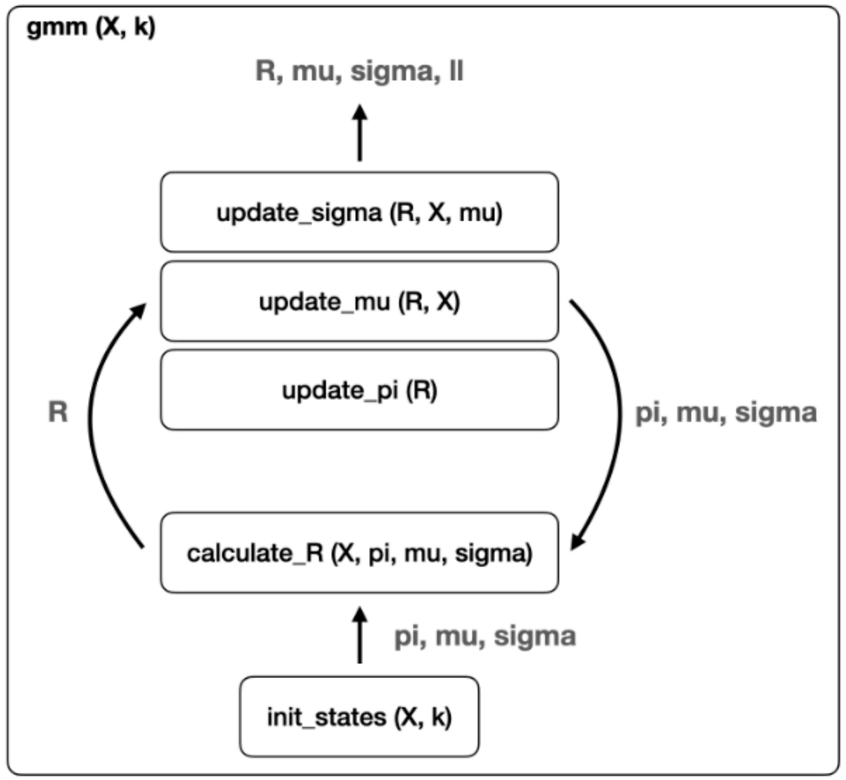
$$\theta = [c_1, ..., c_k, P(c_1), ..., P(c_k)]$$

```python
def get_ll(X, pi, mu, sigma):
    n = len(X)
    k = len(pi)
    mat_ll = np.zeros((n, k))
    for i in range(n):
        for j in range(k):
            mat_ll[i, j] = mvn(mu[j], sigma[j]).pdf(X[i]) * pi[j]
    ll = np.log(np.sum(mat_ll, axis=1)).sum()
    return ll
```

GMM - Put Things Together

We have defined everything we need in EM algorithm. Now, let's try to put code blocks into one single function. You will need:

- `init_states(X, k)` returns μ, Σ, and pi
- `calculate_R(X, pi, mu, sigma)` returns $P(cj|xi)$
- `update_pi(R)` returns $P(c_j)$
- `update_mu(R, X)` returns updated μ
- `update_sigma(R, X, mu)` returns updated Σ




```python
def gmm(X, k, max_iter=10):
    n, p = X.shape
    # initialize parameters
    pi = np.ones(k) / k
    mu = init_center(X, k)
    sigma = np.array([np.eye(p)] * k)
    # EM algorithm
    for i in range(max_iter):
        R = calculate_R(X, pi, mu, sigma)
        pi = update_pi(R)
        mu = update_mu(R, X)
        sigma = update_sigma(R, X, mu)
        ll = get_ll(X, pi, mu, sigma)
        print("Iteration: {}, log-likelihood: {}".format(i + 1, ll))
    # return
    return dict(labels=R.argmax(axis=1),
                mu=np.array(mu), sigma=sigma, ll=ll)
```

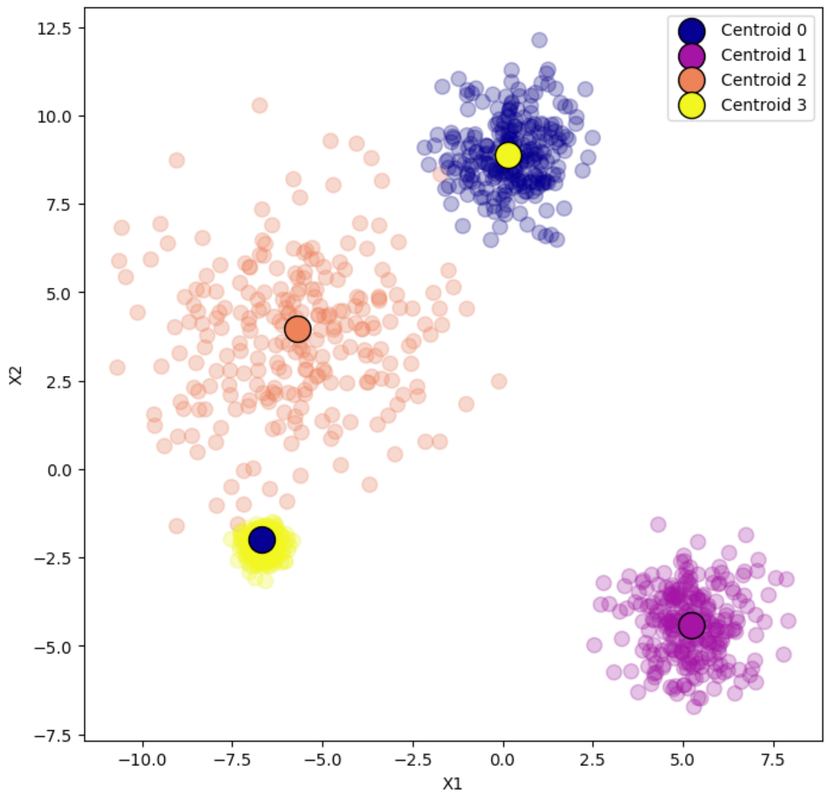
```
gmm_out = gmm(X, k=4, max_iter=20)
```

✓ 5.6s

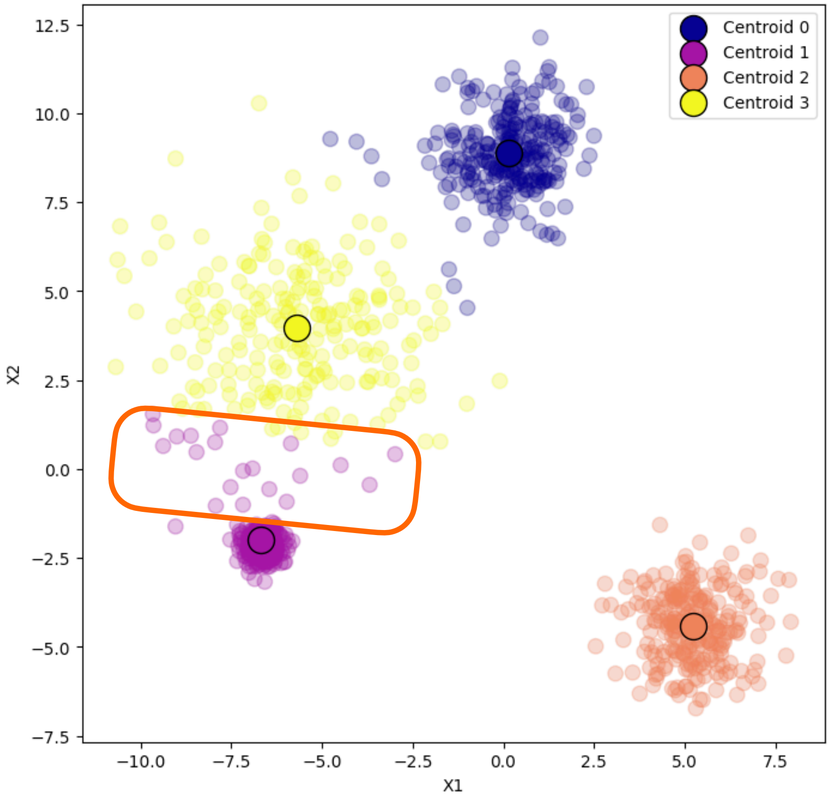
```
Iteration: 1, log-likelihood: -4211.813258643539
Iteration: 2, log-likelihood: -4029.298329541709
Iteration: 3, log-likelihood: -3956.0671587612655
Iteration: 4, log-likelihood: -3945.8428852763564
Iteration: 5, log-likelihood: -3945.414489316607
Iteration: 6, log-likelihood: -3945.4029279901506
Iteration: 7, log-likelihood: -3945.402506140723
Iteration: 8, log-likelihood: -3945.402489336918
Iteration: 9, log-likelihood: -3945.4024886528155
Iteration: 10, log-likelihood: -3945.4024886247853
```

Can You Explain This Figure Now?

Ground Truth



K-Means



Gaussian Mixture Model