EE2211 Tutorial 11

Dr Feng LIN

Q1

The K-means clustering method uses the target labels for calculating the distances from the cluster centroids for clustering.

a) True

b) False

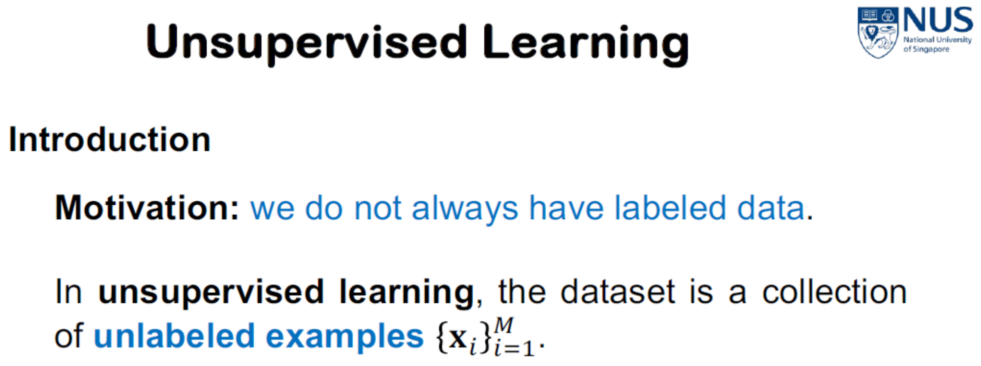
Q1

The K-means clustering method uses the target labels for calculating the distances from the cluster centroids for clustering.

a) True

b) False

Ans: b) because target labels are not available in clustering.



Unsupervised Learning

Introduction

Motivation: we do not always have labeled data.

In **unsupervised learning**, the dataset is a collection of **unlabeled examples** $\{\mathbf{x}_i\}_{i=1}^{M}$.

Q2

The fuzzy C-means algorithm groups the data items such that an item can exist in multiple clusters.

a) True

b) False

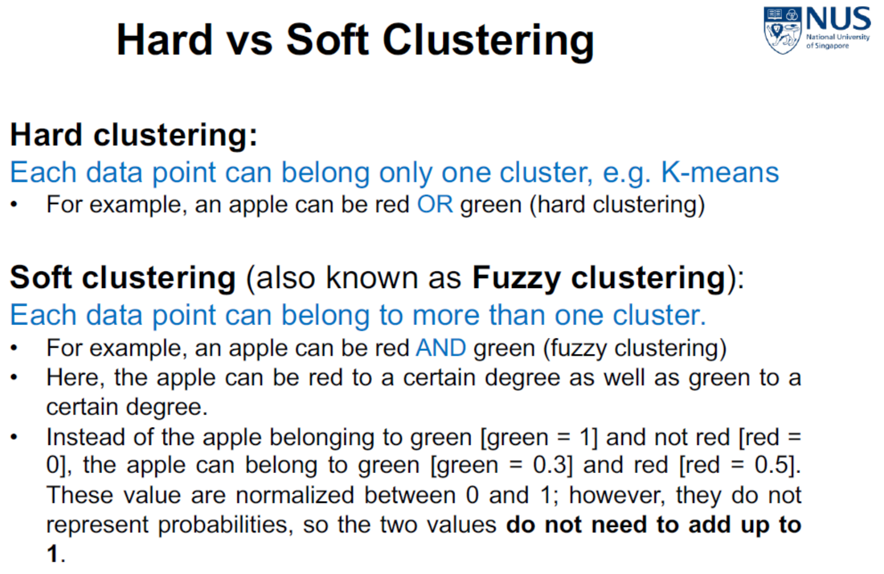
Q2

The fuzzy C-means algorithm groups the data items such that an item can exist in multiple clusters.

a) True

b) False

Ans: a)



Hard vs Soft Clustering

Hard clustering:

Each data point can belong only one cluster, e.g. K-means

- For example, an apple can be red OR green (hard clustering)

Soft clustering (also known as **Fuzzy clustering**):

Each data point can belong to more than one cluster.

- For example, an apple can be red AND green (fuzzy clustering)
- Here, the apple can be red to a certain degree as well as green to a certain degree.
- Instead of the apple belonging to green [green = 1] and not red [red = 0], the apple can belong to green [green = 0.3] and red [red = 0.5]. These value are normalized between 0 and 1; however, they do not represent probabilities, so the two values **do not need to add up to 1**.

Q3

How can you prevent a clustering algorithm from getting stuck in bad local optima?

a) Set the same seed value for each run

b) Use the bottom ranked samples for initialization

c) Use the top ranked samples for initialization

d) All of the above

e) None of the above

Q3

How can you prevent a clustering algorithm from getting stuck in bad local optima?

Random

a) Set the same seed value for each run

b) Use the bottom ranked samples for initialization

c) Use the top ranked samples for initialization

d) All of the above

e) None of the above

Ans: e).

Q4

Consider the following data points: $\boldsymbol{x} = \begin{bmatrix}1\\1\end{bmatrix}$, $\boldsymbol{y} = \begin{bmatrix}0\\1\end{bmatrix}$ and $\boldsymbol{z} = \begin{bmatrix}0\\0\end{bmatrix}$. The k-means algorithm is initialized with centers at $\boldsymbol{x}$ and $\boldsymbol{y}$. Upon convergence, the two centres will be at

a) $\boldsymbol{x}$ and $\boldsymbol{z}$

b) $\boldsymbol{x}$ and $\boldsymbol{y}$

c) $\boldsymbol{y}$ and the midpoint of $\boldsymbol{y}$ and $\boldsymbol{z}$

d) $\boldsymbol{z}$ and the midpoint of $\boldsymbol{x}$ and $\boldsymbol{y}$

e) None of the above

Q4

Consider the following data points: $\boldsymbol{x} = \begin{bmatrix}1\\1\end{bmatrix}$, $\boldsymbol{y} = \begin{bmatrix}0\\1\end{bmatrix}$ and $\boldsymbol{z} = \begin{bmatrix}0\\0\end{bmatrix}$. The k-means algorithm is initialized with centers at $\boldsymbol{x}$ and $\boldsymbol{y}$. Upon convergence, the two centres will be at

a) $\boldsymbol{x}$ and $\boldsymbol{z}$

b) $\boldsymbol{x}$ and $\boldsymbol{y}$

c) $\boldsymbol{y}$ and the midpoint of $\boldsymbol{y}$ and $\boldsymbol{z}$

d) $\boldsymbol{z}$ and the midpoint of $\boldsymbol{x}$ and $\boldsymbol{y}$

e) None of the above

Ans: e). The converged centers should be x and the midpoint of y and z.

Q4

```python
import numpy as np
# Data points
x = np.array([1, 1])
y = np.array([0, 1])
z = np.array([0, 0])
data_points = np.array([x, y, z])
# Initial centers
centers = np.array([x, y])
```

Load data and initialize centers.

Q4

```
def k_means (data_points, centers, n_clusters, max_iterations=100 , tol=1e-4):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np.zeros ((n_clusters, data_points.shape[1]))
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis=0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels
```

Step 1: Assign Data Points to Closest Centroid

- **np.linalg.norm(…)**: Calculates the Euclidean distance between each data point and each centroid.
    - **data_points[:, np.newaxis]-centers** expands data_points to align it with centers, so we can calculate distances between each point and all centroids.
    - **np.linalg.norm(..., axis=2):** Computes the Euclidean norm (distance) across each data point-centroid pair along axis 2.
- **np.argmin(..., axis=1)**: Finds the index (cluster) of the closest centroid for each data point, resulting in an array labels where each element is the assigned cluster for a data point.

```python
# Assign each data point to the closest centroid
labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
```

3×2

data_points

```
[[1 1]
 [0 1]
 [0 0]]
```

$3 \times 1 \times 2$

data_points[:, np.newaxis]

```
[[[1 1]]

 [[0 1]]

 [[0 0]]]
```

2×2

centers

```
[[1 1]
 [0 1]]
```

```
[[[ 1 1]              [[[ 1 1]
  [ 1 1]]               [ 0 1]]

 [[0  1]                [[1 1]
  [ 0  1]]       −       [ 0  1]]

 [[0 0]                 [[1 1]
  [ 0 0]]]               [ 0 1]]]
```

data_points[:, np.newaxis]-centers

```
[[[ 0  0]
  [ 1  0]]

 [[-1  0]
  [ 0  0]]

 [[-1 -1]
  [ 0 -1]]]
```

```python
# Assign each data point to the closest centroid
labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
```

data_points[:, np.newaxis]-centers

```
[[[ 0  0]
  [ 1  0]]

 [[-1  0]
  [ 0  0]]

 [[-1 -1]
  [ 0 -1]]]
```

np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2)

```
[[0.         1.        ]
 [1.         0.        ]
 [1.41421356 1.        ]]
```

labels

```
[0 1 1]
```

Q4

```
def k_means (data_points, centers, n_clusters, max_iterations=100 , tol=1e-4):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np.zeros ((n_clusters, data_points.shape[1]))
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis=0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels
```

Step 2: Update Centroids Based on New Cluster Assignments

- **new_centers = np.zeros(...):** Initializes an empty array to store the updated centroids.
- **for i in range(n_clusters):** Iterates over each cluster index.
- **data_points[labels == i].mean(axis=0):** Computes the mean (centroid) of all data points assigned to cluster i and assigns this mean as the new centroid for cluster i.

Q4

```python
def k_means (data_points, centers, n_clusters, max_iterations=100 , tol=1e-4):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np.zeros ((n_clusters, data_points.shape[1]))
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis=0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels
```

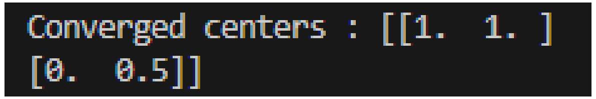
Step 3: Check for Convergence

- **np.linalg.norm(new_centers - centers):** Computes the Euclidean distance between the old centroids (centers) and the updated centroids (new_centers).
- **if ... < tol:** If the total movement of centroids is less than the tolerance tol, the algorithm stops early, assuming centroids have converged.
- **centers = new_centers:** If not converged, updates the old centroids to the new centroids for the next iteration.

Q4

Run K-means function

```
centers , labels = k_means(data_points, centers, n_clusters =2)
print (" Converged centers :", centers )
```



```
Converged centers : [[1.  1. ]
[0.  0.5]]
```

Q4 Simplified code: T11_Q4_simplified.py

```python
from sklearn.cluster import KMeans
import numpy as np

# input data
x = np.array([[1,1], [0,1], [0,0]])

init_centroids = np.array([[1,1], [0,1]])
kmeans = KMeans(n_clusters=2, init=init_centroids, random_state=0, n_init=1)
kmeans.fit(x)
print(f"prediction: \n {kmeans.labels_} \n")
print(f"centers: \n {kmeans.cluster_centers_} \n")
```

```
prediction:
 [0 1 1]

centers:
 [[1.  1. ]
 [0.  0.5]]
```

kmeans = KMeans(n_clusters=2, init=init_centroids, random_state=0, n_init=1)

- n_clusters=2: Group data into 2 clusters.
- init=init_centroids: Use the manually specified starting points.
- random_state=0: Ensure reproducible results.
- n_init=1: Run the algorithm **once only**, using the given centroids (not multiple random restarts).

kmeans.fit(x) runs the actual clustering algorithm.

kmeans.labels_ indicates the cluster assignment for each point.

kmeans.cluster_centers_ returns the final positions of the two cluster centers after convergence.

Q5

Consider the following 8 data points: $\mathbf{x}_1 = \begin{bmatrix}0\\0\end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix}0\\1\end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix}1\\1\end{bmatrix}$, $\mathbf{x}_4 = \begin{bmatrix}1\\0\end{bmatrix}$, $\mathbf{x}_5 = \begin{bmatrix}3\\0\end{bmatrix}$, $\mathbf{x}_6 = \begin{bmatrix}3\\1\end{bmatrix}$, $\mathbf{x}_7 = \begin{bmatrix}4\\0\end{bmatrix}$, and $\mathbf{x}_8 = \begin{bmatrix}4\\1\end{bmatrix}$, The k-means algorithm is initialized with centers at $\begin{bmatrix}0\\0\end{bmatrix}$ and $\begin{bmatrix}3\\0\end{bmatrix}$. The first center after convergence is $\mathbf{c}_1 = \begin{bmatrix}0.5\\0.5\end{bmatrix}$. The second centre after convergence is $\mathbf{c}_2 = \begin{bmatrix}blank\\blank\end{bmatrix}$ (up to 1 decimal place)

Q5

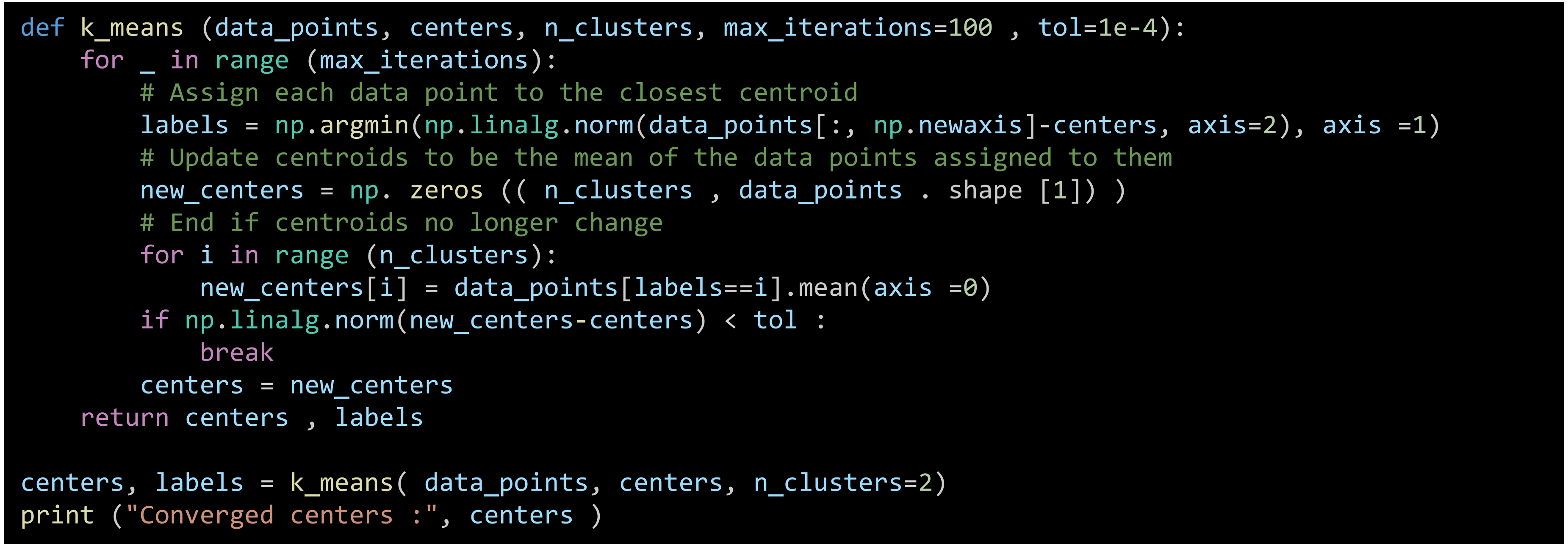
Consider the following 8 data points: $\mathbf{x}_1 = \begin{bmatrix}0\\0\end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix}0\\1\end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix}1\\1\end{bmatrix}$, $\mathbf{x}_4 = \begin{bmatrix}1\\0\end{bmatrix}$, $\mathbf{x}_5 = \begin{bmatrix}3\\0\end{bmatrix}$, $\mathbf{x}_6 = \begin{bmatrix}3\\1\end{bmatrix}$, $\mathbf{x}_7 = \begin{bmatrix}4\\0\end{bmatrix}$, and $\mathbf{x}_8 = \begin{bmatrix}4\\1\end{bmatrix}$, The k-means algorithm is initialized with centers at $\begin{bmatrix}0\\0\end{bmatrix}$ and $\begin{bmatrix}3\\0\end{bmatrix}$. The first center after convergence is $\mathbf{c}_1 = \begin{bmatrix}0.5\\0.5\end{bmatrix}$. The second centre after convergence is $\mathbf{c}_2 = \begin{bmatrix}blank\\blank\end{bmatrix}$ (up to 1 decimal place)

Answer: blank1 = 3.5, blank2 = 0.5.

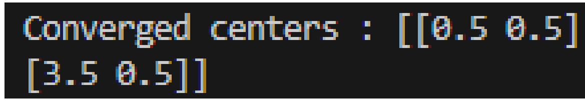
Q5

```
import numpy as np

# Data points
x1 = np.array([0, 0])
x2 = np.array([0, 1])
x3 = np.array([1, 1])
x4 = np.array([1, 0])
x5 = np.array([3, 0])
x6 = np.array([3, 1])
x7 = np.array([4, 0])
x8 = np.array([4, 1])

data_points = np.array ([x1, x2, x3, x4, x5, x6, x7, x8 ])

# Initial centers
c1_init = np. array ([0, 0])
c2_init = np. array ([3, 0])

centers = np. array ([c1_init, c2_init])
```

Q5



```
def k_means (data_points, centers, n_clusters, max_iterations=100 , tol=1e-4):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np. zeros (( n_clusters , data_points . shape [1]) )
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis =0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels

centers, labels = k_means( data_points, centers, n_clusters=2)
print ("Converged centers :", centers )
```



```
Converged centers : [[0.5 0.5]
 [3.5 0.5]]
```

Q5 Simplified code: T11_Q5_simplified.py

```python
from sklearn.cluster import KMeans
import numpy as np

# input data
x = np.array([[0,0], [0,1], [1,1], [1,0], [3,0], [3,1], [4,0], [4,1]])

init_centroids = np.array([[0,0], [3,0]])

kmeans = KMeans(n_clusters=2, init=init_centroids, random_state=0, n_init=1)
kmeans.fit(x)
print(f"prediction: \n {kmeans.labels_} \n")
print(f"centers: \n {kmeans.cluster_centers_} \n")
```

```
prediction:
 [0 0 0 0 1 1 1 1]

centers:
 [[0.5 0.5]
 [3.5 0.5]]
```

Q6 (K-means Implementation on 2D data)

Generate three clusters of data using the following codes.

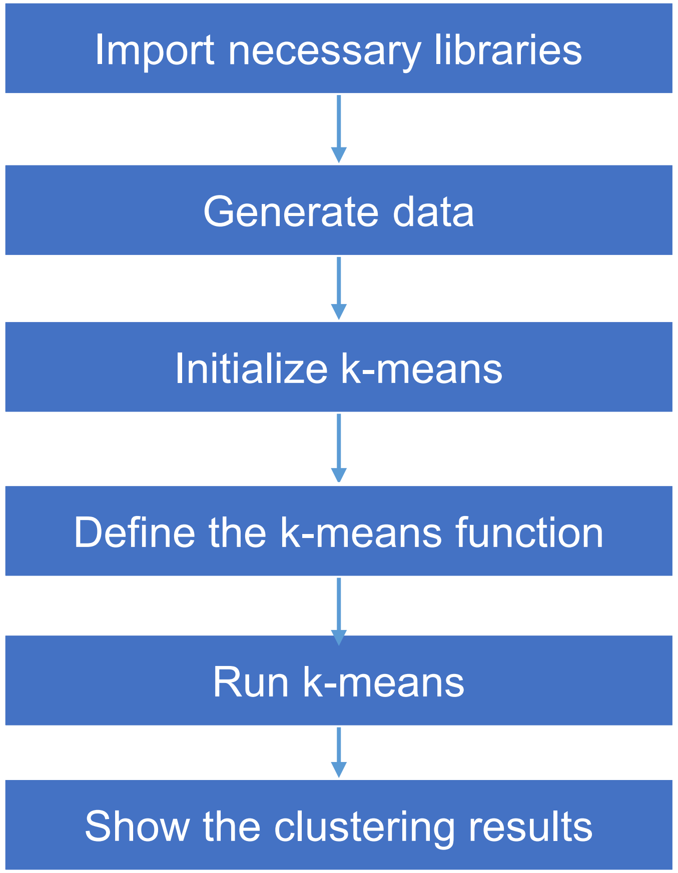
```
## Import necessary libraries
import random as rd
import numpy as np # linear algebra
from matplotlib import pyplot as plt
## Generate data
## Set three centers, the model should predict similar results
center_1 = np.array([2,2])
center_2 = np.array([4,4])
center_3 = np.array([6,1])
## Generate random data and center it to the three centers
data_1 = np.random.randn(200, 2) + center_1
data_2 = np.random.randn(200,2) + center_2
data_3 = np.random.randn(200,2) + center_3
data = np.concatenate((data_1, data_2, data_3), axis = 0)
plt.scatter(data[:,0], data[:,1], s=7)
```

(i) Implement the Naïve K-means (the basic/standard algorithm shown in lecture) clustering algorithm to find the 3 cluster centroids. Classify the data based on the three centroids found and illustrate the results using a plot (e.g., mark the 3 clusters of data points using different colours).

(ii) Change the number of clusters K to 5 and classify the data points again with a plot illustration.

Q6

Block diagram


Import necessary libraries
Generate data
Initialize k-means
Define the k-means function
Run k-means
Show the clustering results

Q6

```python
##--- Import necessary libraries ---#
import random as rd
import numpy as np # linear algebra
from matplotlib import pyplot as plt

##-- Generate data ---#
## Set three centers, the model should predict similar results
center_1 = np.array([2,2])
center_2 = np.array([4,4])
center_3 = np.array([6,1])
k = 3
## Generate random data and center it to the three centers
data_1 = np.random.randn(200, 2) + center_1
data_2 = np.random.randn(200,2) + center_2
data_3 = np.random.randn(200,2) + center_3
data = np.concatenate((data_1, data_2, data_3), axis = 0)
```

Q6

```
##--- Initialize k-means ---##
# Initialize the number of clusters
k = 3

# Forgy Initialize the centroids
centers = data[np.random.choice(len(data), k, replace=False)]
```

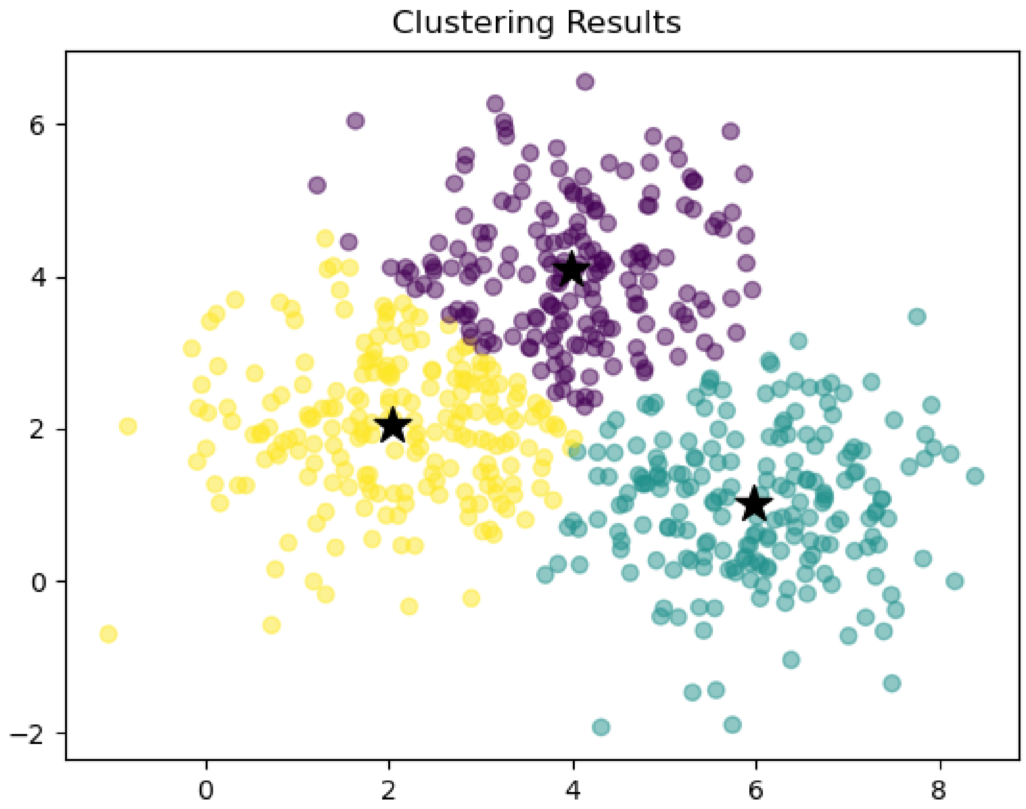
Q6

```python
##--- Define the k-means function ---#
def k_means (data_points, centers, n_clusters, max_iterations=100 , tol=1e-4):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2), axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np.zeros ((n_clusters,data_points.shape[1]) )
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis =0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels

##--- Run k-means ---##
centers, labels = k_means(data, centers, n_clusters =k)
print ("Converged centers :", centers )
```

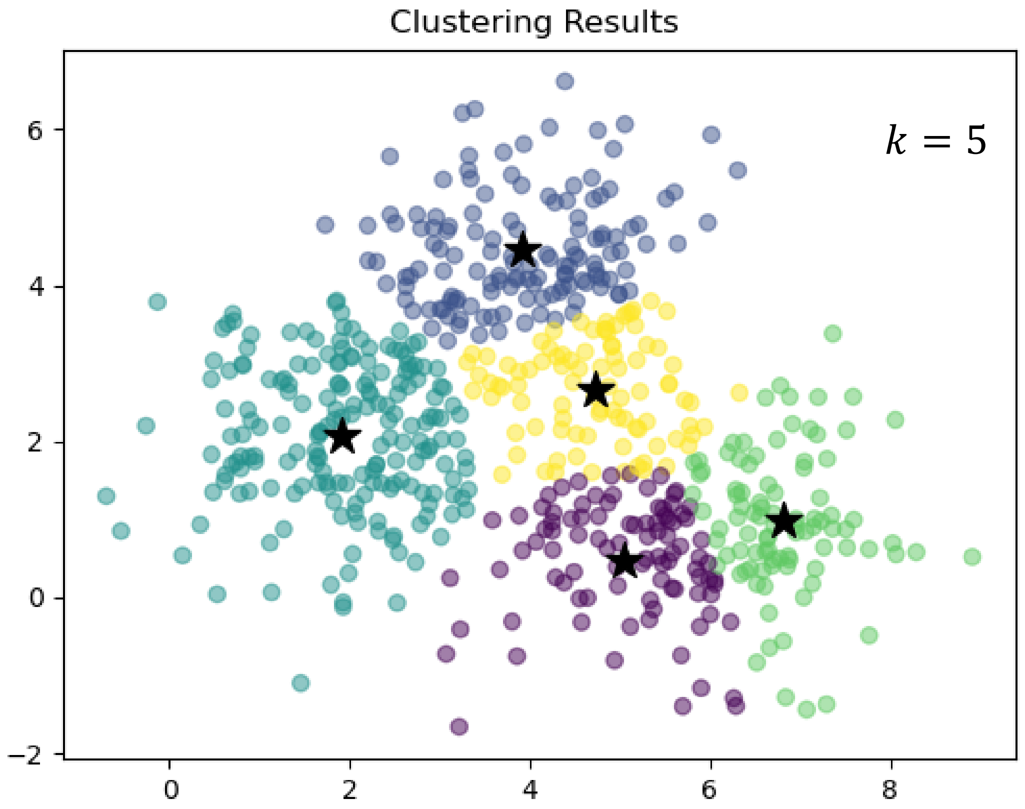
Q6

```python
###--- Show The Clustering Results ---#
plt.title('Clustering Results ')
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap ='viridis', alpha=0.5)
plt.scatter(centers[:, 0], centers [:, 1], marker ='*', s=200 , c='k')
plt.show()
```

Q6



Q6



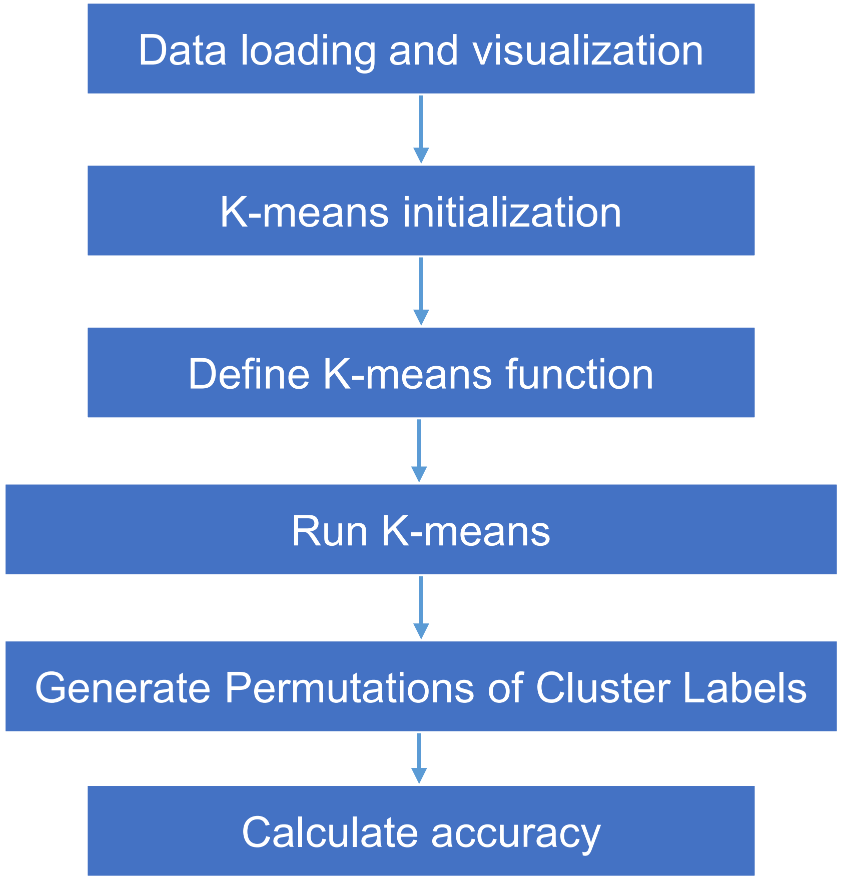
Q7

(K-means Classification of iris data, 4D input features)

Load the iris data “`from sklearn.datasets import load_iris`”. Assume that the class labels are not given. Use the Naïve K-means clustering algorithm to group all the data based on K=3. How accurate is the result of clustering comparing with the known labels?

Q7

Block diagram



Q7

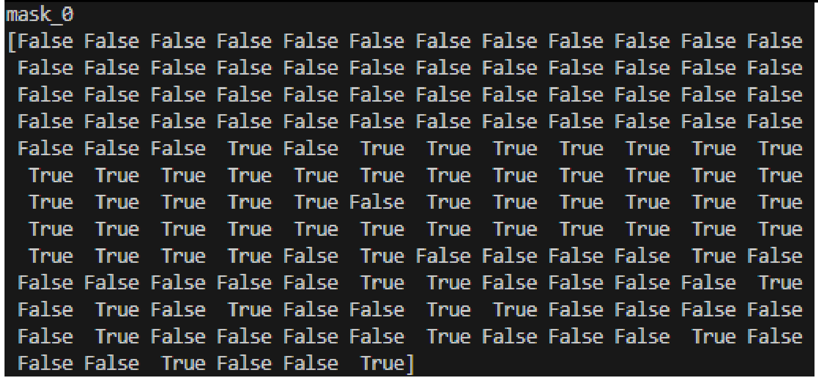
```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
import numpy as np

# load the iris dataset
iris = load_iris ()

# get the data and the true labels
data = iris.data
y_true = iris.target

##--- Initialize k-means ---##
# Initialize the number of clusters
k = 3

# Forgy Initialize the centroids
centers = data[np.random.choice(len(data), k, replace=False)]
```

Q7

```
##--- Define the k-means function ---#
def k_means (data_points, centers, n_clusters, max_iterations=1000, tol=1e-6):
    for _ in range (max_iterations):
        # Assign each data point to the closest centroid
        labels = np.argmin(np.linalg.norm(data_points[:, np.newaxis]-centers, axis=2),
axis =1)
        # Update centroids to be the mean of the data points assigned to them
        new_centers = np. zeros ((n_clusters,data_points.shape[1]) )
        # End if centroids no longer change
        for i in range (n_clusters):
            new_centers[i] = data_points[labels==i].mean(axis =0)
        if np.linalg.norm(new_centers-centers) < tol :
            break
        centers = new_centers
    return centers , labels

##--- Run k-means ---##
centers, y_pred = k_means(data, centers, n_clusters =k)
```

Q7

```python
# create a mask that selects elements where the value is 0, 1, 2
mask_0 = ( y_pred == 0)
mask_1 = ( y_pred == 1)
mask_2 = ( y_pred == 2)
```



```
mask_0
[False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False  True False  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True False  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True False  True False False False False  True False
 False False False False False  True  True False False False False  True
 False  True False  True False False  True  True False False False False
 False  True False False False False  True False False False  True False
 False False  True False False  True]
```

- These lines create Boolean masks for each of the three cluster labels in y_pred:
  - mask_0 selects the indices where y_pred is 0.
  - mask_1 selects the indices where y_pred is 1.
  - mask_2 selects the indices where y_pred is 2.
- These masks are used to map predicted clusters to potential class labels.

Q7

The next set of code creates six different mappings of the three cluster labels (0, 1, and 2) to three class labels (e.g., Setosa, Versicolor, and Virginica).

```
y_pred0=y_pred.copy()
y_pred0[mask_0] = 0
y_pred0[mask_1] = 1
y_pred0[mask_2] = 2
```

- Creates a copy of y_pred named y_pred0 and assigns cluster label 0 to class 0, cluster label 1 to class 1, and cluster label 2 to class 2.

```
y_pred1=y_pred.copy ()
y_pred1[mask_0] = 0
y_pred1[mask_1] = 2
y_pred1[mask_2] = 1
```

- Creates y_pred1 with cluster label 0 mapped to class 0, cluster label 1 to class 2, and cluster label 2 to class 1.

Q7

```
y_pred2 = y_pred.copy ()
y_pred2[mask_0] = 1
y_pred2[mask_1] = 0
y_pred2[mask_2] = 2

y_pred3 = y_pred.copy ()
y_pred3[mask_0] = 1
y_pred3[mask_1] = 2
y_pred3[mask_2] = 0

y_pred4=y_pred.copy ()
y_pred4[mask_0 ] = 2
y_pred4[mask_1 ] = 0
y_pred4[mask_2 ] = 1

y_pred5 = y_pred.copy ()
y_pred5[mask_0] = 2
y_pred5[mask_1] = 1
y_pred5[mask_2] = 0
```

- Each of these permutations creates another mapping by reassigning the three clusters to different classes.
- Each y_predX variable represents a different way of aligning the three cluster labels to the three class labels.

Q7

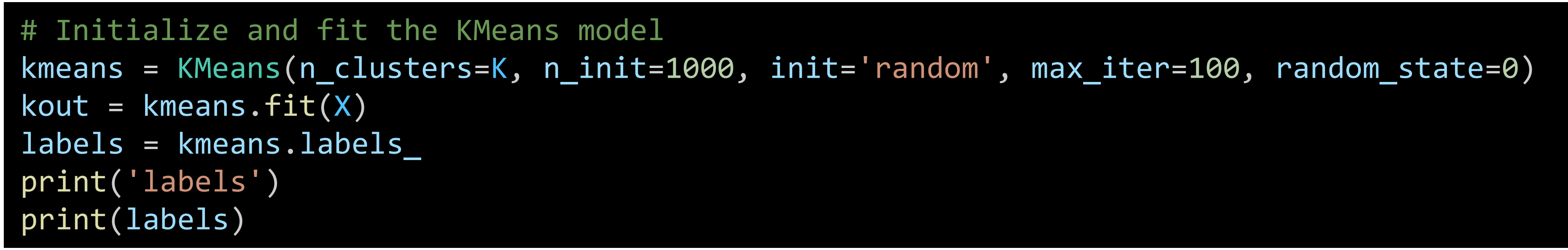
```
# calculate the accuracy of the clustering
accuracy = 0.0
for pred in [y_pred0, y_pred1, y_pred2, y_pred3, y_pred4, y_pred5 ]:
    accuracy = max ([ accuracy_score ( y_true , pred ), accuracy ])

print ("Accuracy of clustering : {:.2f}".format(accuracy))
```

```
Accuracy of clustering : 0.89
```

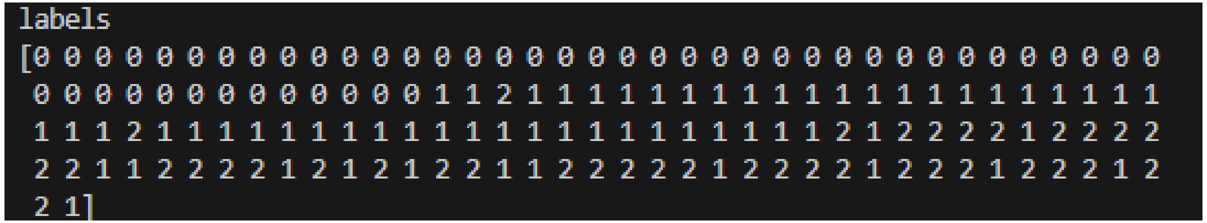
This loop calculates the accuracy of each permutation (y_pred0 to y_pred5) by comparing it to the true labels (y_true).

- accuracy_score(y_true, pred): Computes the accuracy between the true labels and the current permutation.
- accuracy = max([accuracy_score(y_true, pred), accuracy]): Updates accuracy with the maximum value from all permutations, ensuring that accuracy holds the highest achievable clustering accuracy after the loop.

Q7 T11_Q7_simplified.py



```python
# Initialize and fit the KMeans model
kmeans = KMeans(n_clusters=K, n_init=1000, init='random', max_iter=100, random_state=0)
kout = kmeans.fit(X)
labels = kmeans.labels_
print('labels')
print(labels)
```

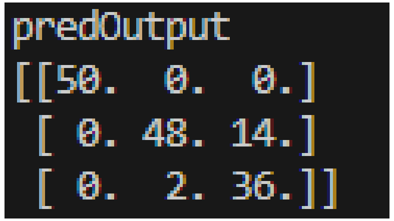


```
labels
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2
 2 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 1 2 2 2 1 2
 2 1]
```

- kmeans = KMeans(...): Initializes the KMeans object with three clusters (K=3), 1000 different initializations (n_init=1000) to ensure optimal centroid positioning, a random initialization strategy, a maximum of 100 iterations per run, and a fixed random seed (random_state=0).
- kout = kmeans.fit(X): Fits the KMeans model to the data X, determining the cluster each data point belongs to, stored in kmeans.labels_.

Q7 T11_Q7_simplified.py

```
# Calibrate each centroid to a corresponding target class
predOutput = np.zeros((K, 3))  # Count of each true label in each cluster
for jj, cluster_label in enumerate(labels):
    predOutput[cluster_label, y[jj]] += 1
print('predOutput')
print(predOutput)
```



```
predOutput
[[50.  0.  0.]
 [ 0. 48. 14.]
 [ 0.  2. 36.]]
```

- **predOutput = np.zeros((K, 3)):** Initializes a matrix to track the true class counts for each cluster.
- **for jj, cluster_label in enumerate(labels):** Loops through each data point to increment the count of each true label (y[jj]) within its assigned cluster.

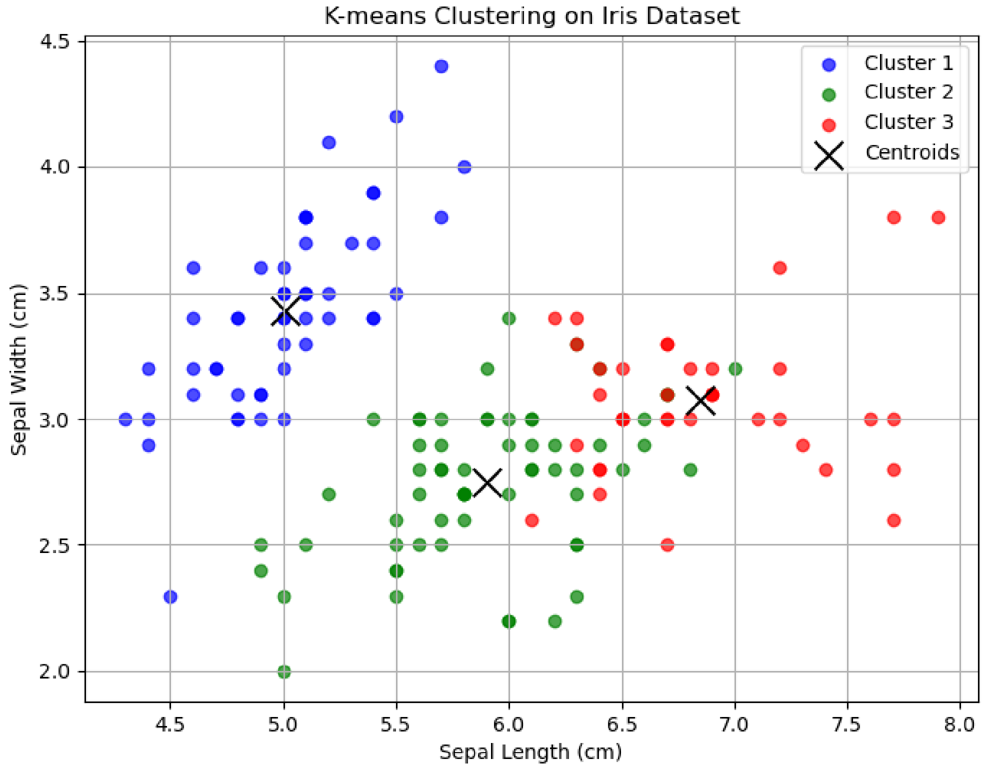
Q7 T11_Q7_simplified.py

```
# Assign each cluster the most common target class
predOutputIdx = np.argmax(predOutput, axis=1)
print('predOutputIdx')
print(predOutputIdx)

# Generate predictions based on calibrated centroids
y_pred = np.array([predOutputIdx[label] for label in labels])

# Calculate and print accuracy
accuracy = accuracy_score(y, y_pred) * 100
print('The K-means method accuracy is:', accuracy, '%')
```

```
predOutputIdx
[0 1 2]
The K-means method accuracy is: 89.33333333333333 %
```

- **np.argmax(predOutput, axis=1)** finds the index of the maximum value along each row (i.e., across columns) for each cluster. This gives us the most common class for each cluster.
- **predOutputIdx[label]** finds the target class associated with its assigned cluster for each data point.
- **accuracy_score(y, y_pred)** calculates the accuracy of the predicted labels (y_pred) against the true labels (y).

Q7 T11_Q7_simplified.py



THANK YOU