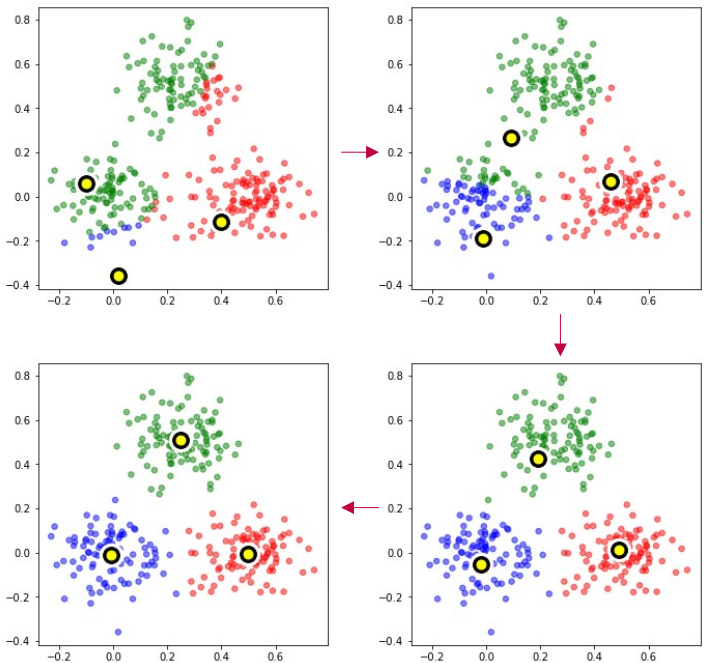


7. K-Means Clustering

Part 1: Basic algorithm for K-Means

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

1. K-Means clustering

[MXML-7-01]
- 1-1. Brief history of K-Means clustering
- 1-2. Training and Prediction process
- 1-3. Local Minimum problem
- 1-4. Loss function and optimization

[MXML-7-02]
- 1-5. Matrix broadcasting and distance matrix
- 1-6. Implement K-Means clustering from scratch
- 1-7. Implement K-Means using Scikit-Learn's KMeans and compare the results.

2. K-Means++

[MXML-7-03]
- 2-1. K-Means++ algorithm
- 2-2. Implement K-Means++ from scratch
- 2-3. Implement K-Means using Scikit-Learn's KMeans
  - 784-dimensional input space of vectorized MNIST
  - MNIST image clustering

3. Optimal number of clusters

[MXML-7-04]
- 3-1. Elbow method
- 3-2. Silhouette analysis: cohesion and separation
- 3-3. Calculating silhouette score
- 3-4. Finding the optimal K using silhouette score

MX-AI

▪ Brief history of K-Means Clustering

- K-means is an algorithm that clusters similar objects together. Although it is an old algorithm published in the 1950s, it is still widely used today.
- **History**: The term "k-means" was first used by James MacQueen in 1967, though the idea goes back to Hugo Steinhaus in 1956. The standard algorithm was first proposed by Stuart Lloyd of Bell Labs in 1957 as a technique for pulse-code modulation, although it was not published as a journal article until 1982. In 1965, Edward W. Forgy published essentially the same method, which is why it is sometimes referred to as the Lloyd–Forgy algorithm. (Source: Wikipedia)

SOME METHODS FOR
CLASSIFICATION AND ANALYSIS
OF MULTIVARIATE OBSERVATIONS

J. MACQUEEN
UNIVERSITY OF CALIFORNIA, Los ANGELES

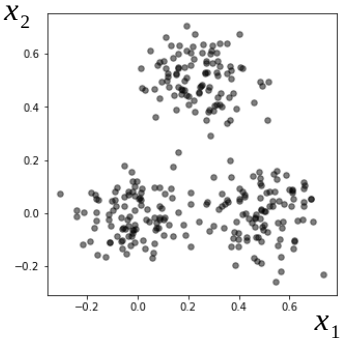
1. Introduction

The main purpose of this paper is to describe a process for partitioning an N-dimensional population into k sets on the basis of a sample. The process, which is called '**k-means,**' appears to give partitions which are reasonably efficient in the sense of within-class variance.

MX-AI

Training and Prediction process

- It seems reasonable to cluster the data points in the figure on the right into three groups.
- Because it is two-dimensional data, it is easy to cluster by just looking at it, but as the number of features increases, clustering becomes more difficult, so an algorithm is needed. K-Means is one of the clustering algorithms.
- Error cannot be measured in this type of data because it only has features and no target values or labels. Learning from such data is called unsupervised learning. K-Means is one of the unsupervised learning algorithms.
- The training process of K-Means is as follows, through which K centroids are determined.
- The prediction process uses the centroids. A test data point is predicted to belong to the cluster with the closest centroid to that point.



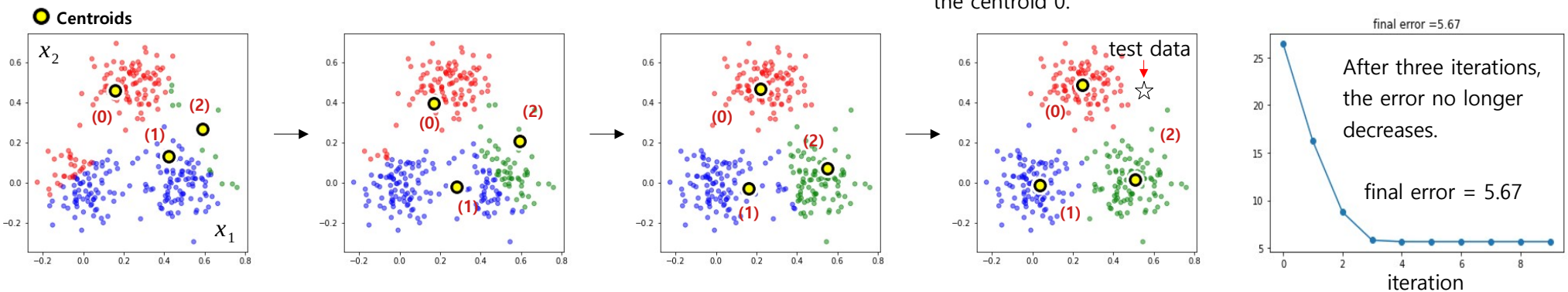
Step-1) K data points are randomly selected and used as K initial centroids. And assign each data point to the nearest centroid.

Step-2) Shift the centroid to the average coordinate of the data points assigned to that centroid and reassign each data point to the new centroid.

Step-3) Repeat step 2 until the centroids no longer shift. Each centroid gradually shifts towards the center of its cluster.

Step-4) Once training is complete, the centroids are used to predict which cluster a test data point belongs to. The test data point below is predicted to belong to cluster 0 because it is closest to the centroid 0.

The sum of the distances between each data point and its centroid can be used as a proxy for training error. The better the clustering, the smaller this error will be.

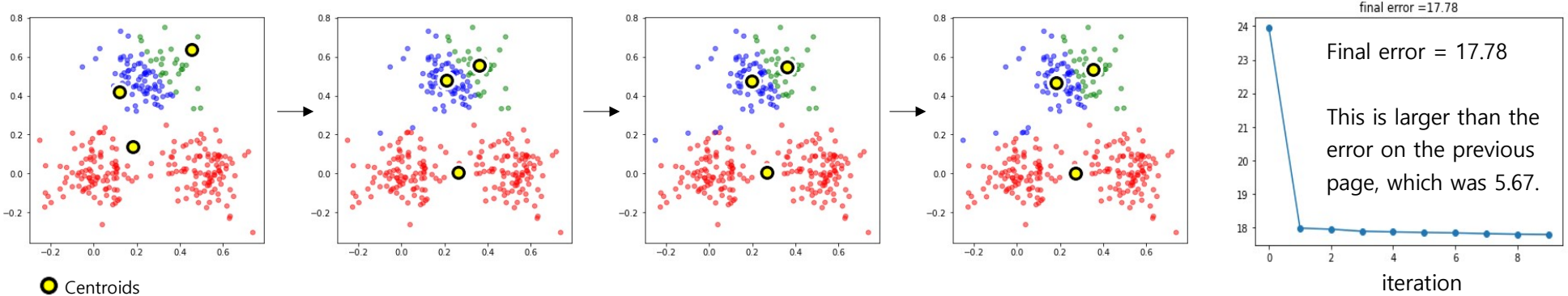


MX-AI

▪ Local minimum problem

- Depending on the location of the randomly set initial centroids, K-Means clustering may fail, as shown in the example below.
- Looking at the results below, you can see that the error decreased with one iteration, but did not decrease further after that. The final error is 17.78, which is larger than the error on the previous page of 5.67.
- The reason is that when minimizing the loss function, which is the objective function, it did not fall into the global minimum but fell into a local minimum. The loss function is covered on the next page.
- To solve this problem, you can try K-Means multiple times while varying the positions of the initial centroids and choose the result with the smallest error. This method is easy to implement, but has the disadvantage of being time consuming.
- Another way is to lower the probability of this happening by distributing the initial centroid positions appropriately. This is the K-Means++ algorithm and we will look at it in detail later.

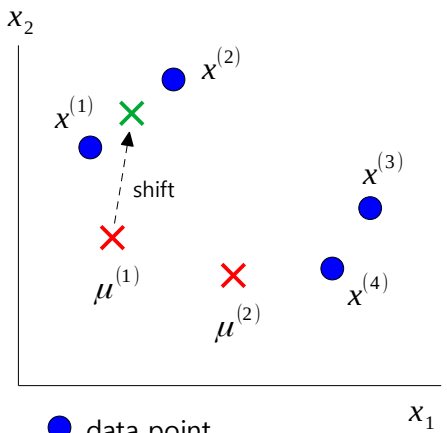
▪ Example of Local minimum



MX-AI

▪ Loss function and optimization

- K-Means is an algorithm that assigns data points to each centroid and then minimizes the sum of the distances from data points to their centroid.
- Step 1 assigns the data points to the nearest centroid, and step 2 minimizes the sum of the distances between each data point and the assigned centroid.
- If a is 0 or 1, it is called hard clustering, and if a is real value, it is called soft clustering. For example, in soft clustering, if a11 is 0.8 and a12 is 0.2, this means that the data point 1 has an 80% probability of being assigned to centroid 1 and a 20% probability of being assigned to centroid 2. In this video, we will only cover the hard clustering.



Loss function:

$$L = a_{11}\|x^{(1)}-\mu^{(1)}\|^2 + a_{12}\|x^{(1)}-\mu^{(2)}\|^2 + a_{21}\|x^{(2)}-\mu^{(1)}\|^2 + a_{22}\|x^{(2)}-\mu^{(2)}\|^2 + a_{31}\|x^{(3)}-\mu^{(1)}\|^2 + a_{32}\|x^{(3)}-\mu^{(2)}\|^2 + a_{41}\|x^{(4)}-\mu^{(1)}\|^2 + a_{42}\|x^{(4)}-\mu^{(2)}\|^2$$

$$L = \sum_{n=1}^{N}\sum_{k=1}^{K} a_{nk}\|x^{(n)}-\mu^{(k)}\|^2$$

1) step-1: Assign data points to centroids

$$\text{assignment} \begin{cases} a_{11}=1, & a_{12}=0 \\ a_{21}=1, & a_{22}=0 \\ a_{31}=0, & a_{32}=1 \\ a_{41}=0, & a_{42}=1 \end{cases}$$

2) step-2: Update centroids

$$\frac{\partial L}{\partial \mu^{(k)}} = -2\sum_{n=1}^{N} a_{nk}\|x^{(n)}-\mu^{(k)}\| = 0$$

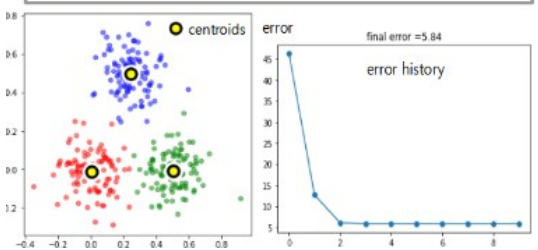
$$\sum_{n=1}^{N} a_{nk}\, x^{(n)} - \sum_{n=1}^{N} a_{nk}\,\mu^{(k)} = 0$$

$$\mu^{(k)} = \frac{\sum_{n=1}^{N} a_{nk}\, x^{(n)}}{\sum_{n=1}^{N} a_{nk}} = \frac{1}{n}\sum_{n=1}^{N} a_{nk}\, x^{(n)}$$

← This is where the k-th centroid will move to.

This is the average coordinate of the data points assigned to the k-th centroid. It is optimal to shift the k-th centroid to this location.

```python
# [MXML-7-02] 1.kmeans(basic).py
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import time

# Generate training data
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, center_box=(-1., 1.))

N = x.shape[0]  # the number of data points
K = 3           # the number of clusters
M = 10          # the number of iterations
L = 10  # the number of attempts to prevent local minimum problem

# Visualize training data
def plot_data(x):
    plt.figure(figsize=(5, 5))
    plt.scatter(x[:, 0], x[:, 1], s=30, c='black', alpha=0.5)
    plt.show()

# Visualize training data and clusters color-coded.
def plot_cluster(x, cluster, centroid):
    plt.figure(figsize=(5, 5))
    color = [['red', 'blue', 'green'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.show()
plot_data(x)
```



7. K-Means Clustering

Part 2: Implement K-Means from scratch

This video was produced in Korean, translated into English, and the audio was generated by AI (TTS).

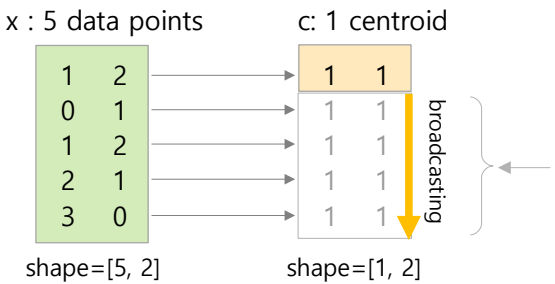
www.youtube.com/@meanxai

MX-AI

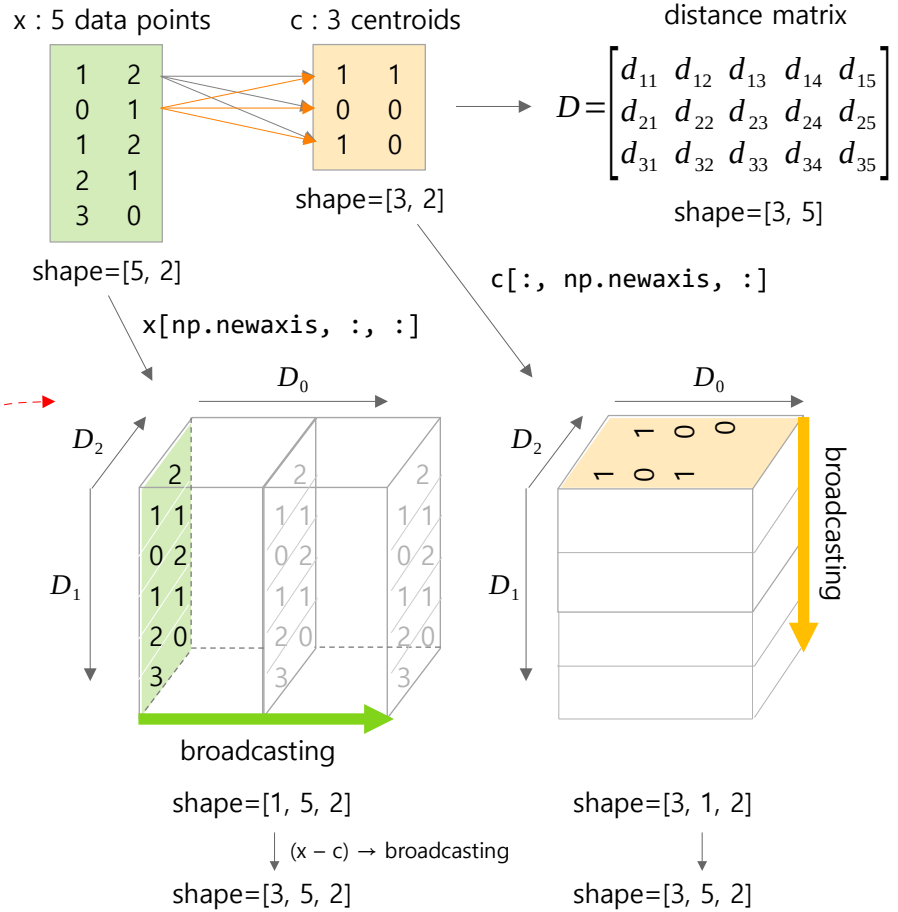
▪ Matrix broadcasting and distance matrix

- The distance matrix (D) between the training data points and the centroids can be found using multiple for-loops, but can easily be found using matrix broadcast.

▪ Broadcasting



Computing (x – c) performs broadcasting. shape=[5, 2]



- Python code for calculating the distance matrix (dist) between 5 training data points x and 3 centroids c in the figure on the right.

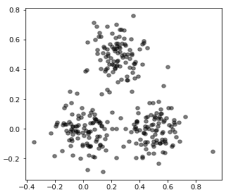
```
x_exp = x[np.newaxis, :, :]    # add a D0 axis. (1, 5, 2)
c_exp = c[:, np.newaxis, :]    # add a D1 axis. (3, 1, 2)

# Calculating the distances between the training data points and the
# centroids. The shape of dist = (3, 5)
dist = np.sqrt(np.sum(np.square(x_exp - c_exp), axis=2))
```

Distance between the first training data point (1,2) and the first centroid (1,1): $d = \sqrt{(1-1)^2+(2-1)^2}$

MX-AI

▪ Implement K-Means clustering from scratch

```
# [MXML-7-02] 1.kmeans(basic).py
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import time

# Generate training data
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, center_box=(-1., 1.))

N = x.shape[0]  # the number of data points
K = 3           # the number of clusters
M = 10          # the number of iterations
L = 10  # the number of attempts to prevent local minimum problem

# Visualize training data
def plot_data(x):
    plt.figure(figsize=(5, 5))
    plt.scatter(x[:, 0], x[:, 1], s=30, c='black', alpha=0.5)
    plt.show()

# Visualize training data and clusters color-coded.
def plot_cluster(x, cluster, centroid):
    plt.figure(figsize=(5, 5))
    color = [['red', 'blue', 'green'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.show()
plot_data(x)
```



```
f_error = [9999]      # final error history
f_centroids = None    # final centroids
f_assign = None       # final assignment

for l in range(L):    # Repeat L times, changing the initial
                      # positions of centroids.
    # Initialize the centroids. Randomly select K data points
    # and use them as the K initial centroids.
    idx = np.random.choice(np.arange(N), K)
    centroids = x[idx]
    error = []
    for m in range(M):
        # Calculate the distances between the training data points
        # and the centroids.
        x_exp = x[np.newaxis, :, :]    # add a D0 axis. (1, N, 2)
        c_exp = centroids[:, np.newaxis, :] # add a D1 axis.
                                            # (K, 1, 2)

        # create the distance matrix using matrix broadcasting.
        # The shape of dist = (K, N)
        dist = np.sqrt(np.sum(np.square(x_exp - c_exp), axis=2))

        # Assign each data point to the nearest centroid.
        # if assign = [0 1 2 1 0 2 0 1 ...]
        # The first data point is assigned to cluster 0,
        # and the second data point is assigned to cluster 1.
        assign = np.argmin(dist, axis=0)  # shape = (N,)

        # update centroids
        new_cent = []
        err = 0
```

MX-AI

- Implement K-Means clustering from scratch

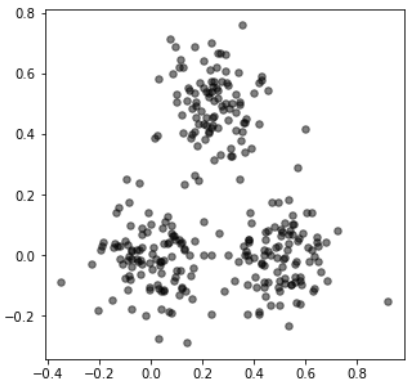
```
        for c in range(K):
            # Find the data points assigned to centroid c.
            idx = np.where(assign == c)
            x_idx = x[idx]

            # The error is measured as the sum of the squares of
            # the distances between data points and their centroid.
            err += np.sum(np.sum(np.square(x_idx - centroids[c]),
                                 axis=1))

            # Compute the average coordinates of the data points
            # assigned to this centroid.
            # And use that as new centroid.
            new_cent.append(np.mean(x_idx, axis=0))

        error.append(err)

        # To observe the centroid moving, set L=1
        # and run the code below.
        # plot_cluster(x, assign, centroids)
        # print("iteration:", m)
        # time.sleep(1)

        # Update centroids
        centroids = np.array(new_cent)

    # Among the L number of iterations, the one with the smallest
    # error is selected as the final result.
    if error[-1] < f_error[-1]:
        f_error = np.copy(error)
        f_centroids = np.copy(centroids)
        f_assign = np.copy(assign)
```

```
# Visualize the training result.
plot_cluster(x, f_assign, f_centroids)

# Visualize error history
plt.plot(f_error, 'o-')
plt.title('final error =' + str(np.round(error[-1], 2)))
plt.show()

# Check the cluster number for each data point.
import pandas as pd
df = pd.DataFrame({'x1': x[:,0], 'x2': x[:,1],
                   'cluster': f_assign})
print(df.head(10))
```

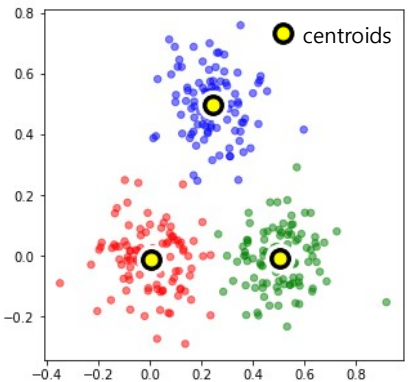
MX-AI

- Implement K-Means clustering from scratch
  - Results:

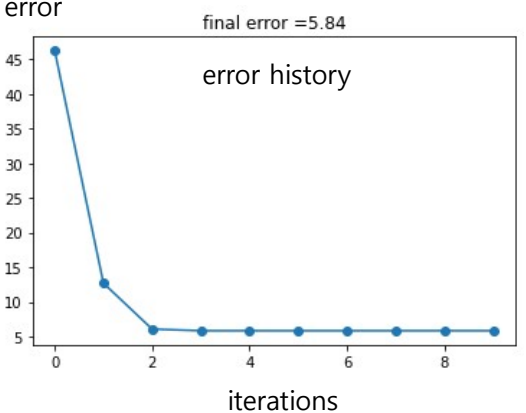
training data



training data and their centroids



error



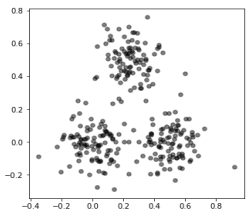
	x1	x2	cluster
0	0.418481	0.017363	2
1	0.142371	-0.089209	1
2	0.380852	-0.005970	2
3	-0.004757	-0.051350	1
4	0.522231	0.052283	2
5	0.342561	0.432163	0
6	0.056708	0.122125	1
7	-0.069264	0.036702	1
8	-0.241860	0.070617	1
9	0.173620	0.604274	0

- Cluster numbers may vary from run to run.
- The cluster number itself is not important.
- What's important is that the first, third, and fifth data points are in the same cluster, and the second, fourth, seventh, eighth, and ninth data points are in the same cluster.
- The data points belonging to the same cluster have the same properties.
- If your test data point falls into cluster 2, you can see that it is similar to the data points belonging to that cluster.
- Finding this out is the purpose of clustering.

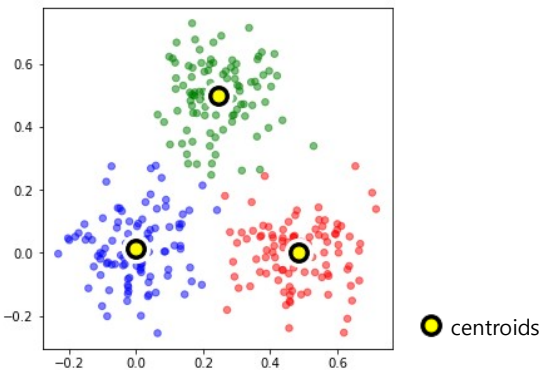
MX-AI

▪ Implement K-Means clustering using scikit-learn's KMeans

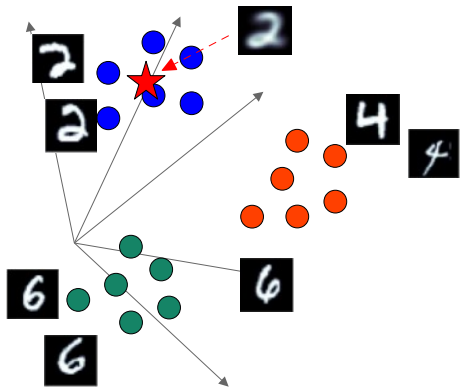
```
# [MXML-7-02] 2.sklearn(kmeans).py
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Generate training data
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, center_box=(-1., 1.))

K = 3           # the number of clusters
M = 10          # the number of iterations
L = 10      # the number of attempts to prevent local minimum problem.
model = KMeans(n_clusters = K,  # the number of clusters
               init='random',   # randomly initialize centroids
               max_iter=M,      # max iterations
               n_init = L)      # Number of times the k-means algorithm
                                # is run with different centroid seeds.
model.fit(x)

# Visualize training data and clusters color-coded.
def plot_cluster(x, cluster, centroid):
    plt.figure(figsize=(5, 5))
    color = [['red', 'blue', 'green'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.show()
```

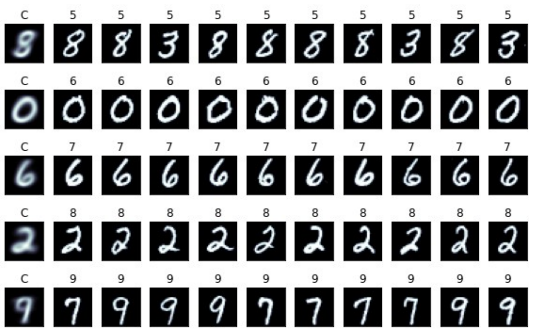


```
# Visualize the training result
plot_cluster(x, model.labels_, model.cluster_centers_)

# print the final error
# inertia_: Sum of squared distances of samples to their
# closest cluster center
print('error = {:.4f}'.format(model.inertia_))
```




```
final error = 5.7561

model.labels_ : Cluster number assigned to each data point.
model.cluster_centers_ : coordinates of final centroids
model.inertia_ : Sum of squared distances. This is the same
                 error defined in the previous video.
```





7. K-Means Clustering

Part 3: K-Means++ algorithm

This video was produced in Korean, translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ K-Means++

▪ K-means++ is an improvement on the standard K-means clustering algorithm. This was proposed by David Arthur and Sergei Vassilvitskii in 2006 to spread out the initial centroids. This reduces the local minimum problem and improves the overall performance of clustering.

k-means++: The Advantages of Careful Seeding

David Arthur Sergei Vassilvitskii

Abstract

The k-means method is a widely used clustering technique that seeks to minimize the average squared distance between points in the same cluster. Although it offers no accuracy guarantees, its simplicity and speed are very appealing in practice. By augmenting k-means with a very simple, randomized seeding technique, we obtain an algorithm that is Θ(log k)-competitive with the optimal clustering. Preliminary experiments show that our augmentation improves both the speed and the accuracy of k-means, often quite dramatically.

2 Preliminaries

2.2 The k-means++ algorithm The k-means algorithm begins with an arbitrary set of cluster centers. We propose a specific way of choosing these centers. At any given time, let D(x) denote the shortest distance from a data point x to the closest center we have already chosen. Then, we define the following algorithm, which we call k-means++.

1a. Choose an initial center c_1 uniformly at random from X .
1b. Choose the next center c_i, selecting $c_i = x' \in X$ with probability

$$\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$$

1c. Repeat Step 1b until we have chosen a total of k centers.

2-4. Proceed as with the standard k-means algorithm.

We call the weighting used in Step 1b simply "D^2 weighting".

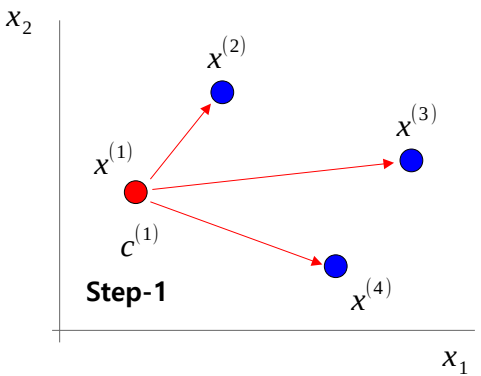
MX-AI

▪ K-Means++ algorithm

- Standard K-Means requires multiple attempts to solve the local minimum problem. This problem typically occurs when randomly set initial centroids are close to each other. Properly distributing the initial centroids can reduce the likelihood of this problem occurring.

- **K-Means++ algorithm**:
- At any given time, let D(x) denote the shortest distance from a data point x to the closest centroid we have already chosen.
  (1) Choose an initial centroid $c^{(1)}$ uniformly at random from X .
  (2) Choose the next centroid $c^{(i)}$, selecting $c^{(i)} = x' \in X$ with probability $\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$
  (3) Repeat Step (2) until we have chosen a total of k centroids.

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}]$$



$c^{(1)}$ →

i	x_1	x_2	$D(x)^2$
1	0.1	0.4	
2	0.4	0.6	
3	0.8	0.5	
4	0.7	0.2	

Step-2:

$$x' = [x^{(2)}, x^{(3)}, x^{(4)}]$$

$$D(x^{(2)})^2 = d(c^{(1)}, x^{(2)})^2 = (0.1-0.4)^2+(0.4-0.6)^2 = 0.13$$

$$D(x^{(3)})^2 = d(c^{(1)}, x^{(3)})^2 = (0.1-0.8)^2+(0.4-0.5)^2 = 0.5$$

$$D(x^{(4)})^2 = d(c^{(1)}, x^{(4)})^2 = (0.1-0.7)^2+(0.4-0.2)^2 = 0.4$$

probability that $x^{(3)}$ is chosen as $c^{(2)}$.

Probability density function: $\frac{D(x')^2}{\sum_{j=[2,3,4]} D(x^{(j)})^2} = \frac{1}{0.13+0.5+0.4} \times [0.13, 0.5, 0.4] = [0.126,\ 0.485,\ 0.388]$

probability that $x^{(4)}$ is chosen as $c^{(2)}$

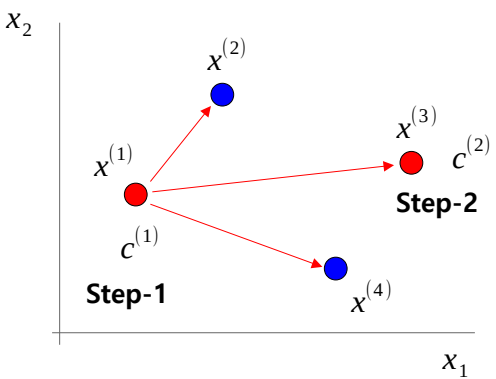
* $x^{(3)}$ is most likely to be chosen as $c^{(2)}$. → $x^{(3)}$ is furthest from $c^{(1)}$. → $c^{(1)}$ and $c^{(2)}$ are likely to be far away from each other. → The initial centroids are more likely to be spread out.

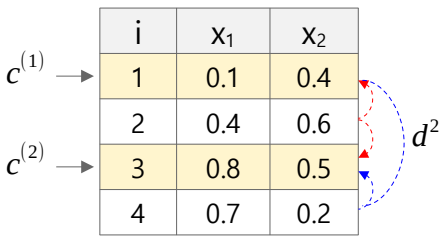
MX-AI

▪ K-Means++ algorithm

- At any given time, let D(x) denote the shortest distance from a data point x to the closest centroid we have already chosen.
  (1) Choose an initial centroid $c^{(1)}$ uniformly at random from X .
  (2) Choose the next centroid $c^{(i)}$, selecting $c^{(i)} = x' \in X$ with probability $\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$
  (3) Repeat Step (2) until we have chosen a total of k centroids.

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}]$$





i	x_1	x_2
1	0.1	0.4
2	0.4	0.6
3	0.8	0.5
4	0.7	0.2

Step-3:

$$x' = [x^{(2)}, x^{(4)}]$$

closest

$$D(x^{(2)})^2 = d(c^{(1)}, x^{(2)})^2 = (0.1-0.4)^2+(0.4-0.6)^2 = 0.13$$

$$D(x^{(2)})^2 = d(c^{(2)}, x^{(2)})^2 = (0.8-0.4)^2+(0.5-0.6)^2 = 0.17$$

$$D(x^{(4)})^2 = d(c^{(1)}, x^{(4)})^2 = (0.1-0.7)^2+(0.4-0.2)^2 = 0.4$$

$$D(x^{(4)})^2 = d(c^{(2)}, x^{(4)})^2 = (0.8-0.7)^2+(0.5-0.2)^2 = 0.1$$

probability that $x^{(2)}$ is chosen as $c^{(3)}$.

Probability density function: $\frac{D(x')^2}{\sum_{j=[2,4]} D(x^{(j)})^2} = \frac{1}{0.13+0.1} \times [0.13, 0.1] = [0.565,\ 0.435]$

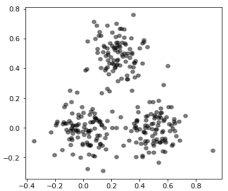
probability that $x^{(4)}$ is chosen as $c^{(3)}$.

* $x^{(1)}$, $x^{(2)}$, and $x^{(3)}$ are likely to be chosen as initial focuses.

MX-AI

- Implement K-Means++ algorithm from scratch

```
# [MXML-7-03] 3.kmeans(plus).py
# This code can be found at github.com/meanxai/machine_learning.
import numpy as np
import random as rd
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generate training data points.
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, centers_box=(-1., 1.))

N = x.shape[0]  # the number of data points
K = 3           # the number of clusters
M = 10          # the number of iterations

# Visualize the data points, x.
def plot_data(x):
    plt.figure(figsize=(5, 5))
    plt.scatter(x[:, 0], x[:, 1], s=30, c='black', alpha=0.5)
    plt.show()

# Visualize training data points and clusters color-coded.
def plot_cluster(x, cluster, centroid):
    plt.figure(figsize=(5, 5))
    color = [['red', 'blue', 'green'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.show()
plot_data(x)
```



```
# Generate initial centroids using the K-Means++ algorithm.
xp = x.copy()
centroids = []
density = np.ones(xp.shape[0]) / N
for c in range(K):
    # (1) Choose an initial centroid c(1) uniformly at random from X
    # (2) Choose the next centroid c(i), selecting c(i) = x' ∈ X with
    #     the probability density function.
    idx = rd.choices(np.arange(xp.shape[0]), weights=density, k=1)[0]
    centroids.append(xp[idx])
    xp = np.delete(xp, idx, axis=0)

    # Create a distance matrix between data points xp and the
    # centroids. Please refer to the video [MXML-7-02] for how to
    # create a distance matrix.
    x_exp = xp[np.newaxis, :, :]
    c_exp = np.array(centroids)[:, np.newaxis, :]
    dist = np.sqrt(np.sum(np.square(x_exp - c_exp), axis=2))

    # Find the centroid closest to each data point.
    assign = np.argmin(dist, axis=0)

    # Calculate D(x)
    # let D(x) denote the shortest distance from a data point x to
    # the closest centroid we have already chosen
    Dx = np.sum(np.square(xp - np.array(centroids)[assign]), axis=1)

    # Create a probability density function to select the next
    # centroid.
    density = Dx / np.sum(Dx)
centroids = np.array(centroids)
```

$$\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$$

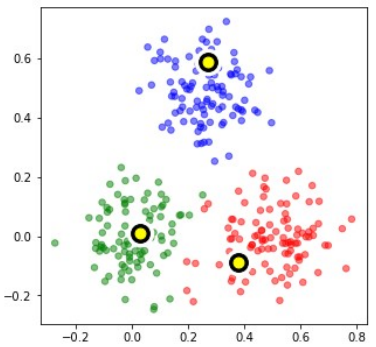
MX-AI

- Implement K-Means++ algorithm from scratch

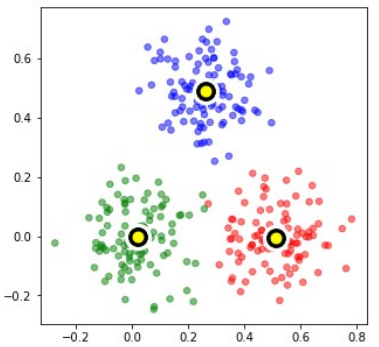
```
# Perform the K-Means using the centroids generated by K-Means++.
error = []
for m in range(M):
    # Calculate the distances between the training data points and the
    # centroids.
    x_exp = x[np.newaxis, :, :]
    c_exp = centroids[:, np.newaxis, :]
    dist = np.sqrt(np.sum(np.square(x_exp - c_exp), axis=2))

    # Assign each data point to the nearest centroid.
    assign = np.argmin(dist, axis=0)  # shape = (N,)

    # update centroids
    new_cent = []
    err = 0
    for c in range(K):
        # Find the data points assigned to centroid c.
        idx = np.where(assign == c)
        x_idx = x[idx]

        # To measure clustering performance, calculate the error.
        err += np.sum(np.sum(np.square(x_idx - centroids[c]), axis=1))

        # Compute the average coordinates of the data points
        # assigned to this centroid. And use that as new centroid.
        new_cent.append(np.mean(x_idx, axis=0))
    error.append(err)

    # Remove the if statement to see the centroids moving.
    if m == 0:
        plot_cluster(x, assign, centroids)

    # Update centroids
    centroids = np.array(new_cent)
```

```
# Visualize the training result.
plot_cluster(x, assign, centroids)

# Visualize error history
plt.plot(error, 'o-')
plt.title('final error =' + str(np.round(error[-1], 2)))
plt.show()
```

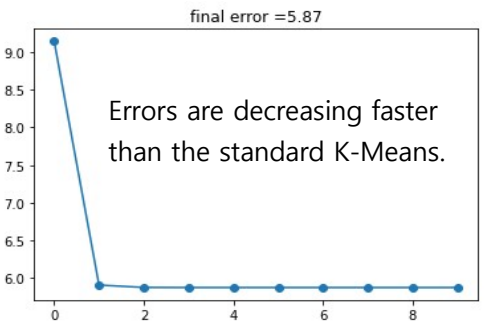
Initial centroids



Final centroids



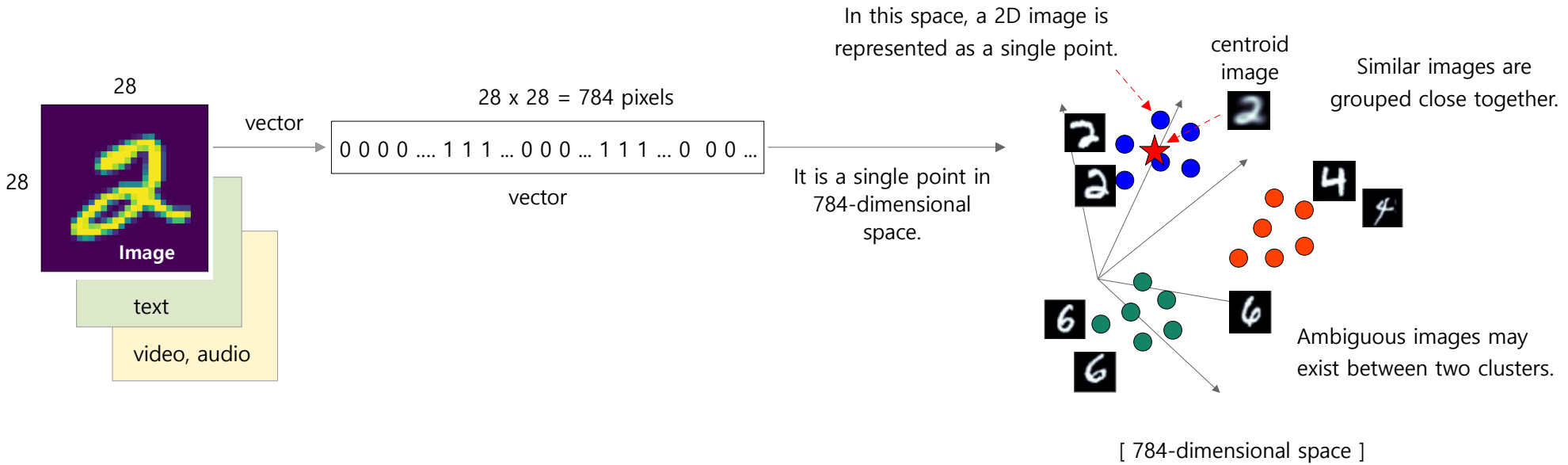
The initial centroids are spread out over the whole range of the data points. It doesn't always happen, but it's much more likely to happen.



MX-AI

- 784-dimensional input space of vectorized MNIST image

  - Data such as images, text, video, and audio can be represented as a single point in a high-dimensional space.
  - In the space, Similar images are grouped close together.



MX-AI

▪ Implement K-Means++ using sklearn's KMeans: MNIST clustering

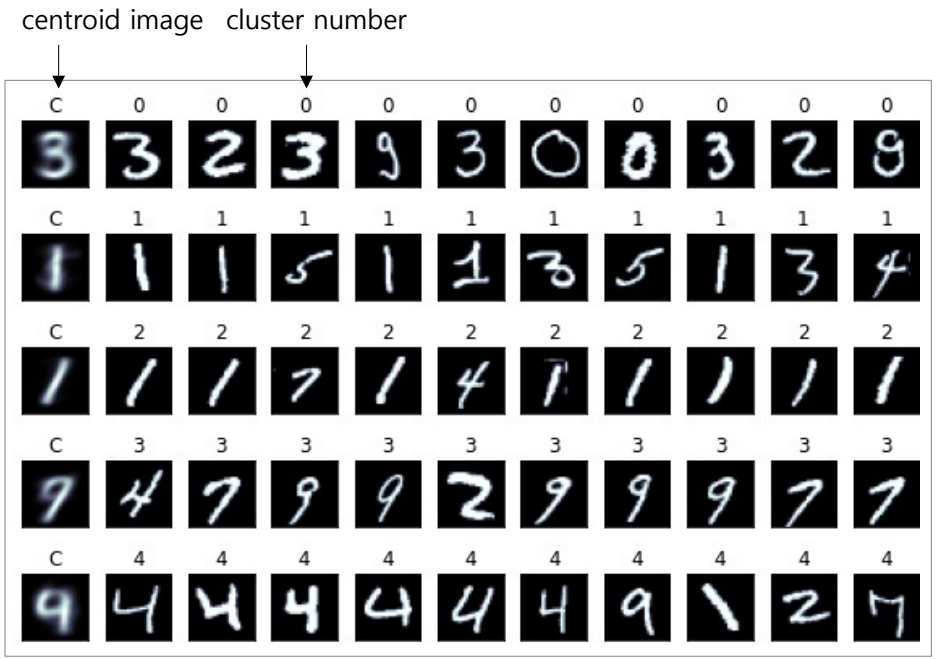
```
# [MXML-7-03] 4.sklearn(mnist).py
# MNIST clustering
# This code can be found at github.com/meanxai/machine_learning.
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import pickle

# from sklearn.datasets import fetch_openml
# mnist = fetch_openml('mnist_784')
# mnist.pkl is the saved mnist.
with open('data/mnist.pkl', 'rb') as f:
        mnist = pickle.load(f)

# Use only 10,000 data points and normalize them between 0 and 1
x = np.array(mnist['data'][:10000]) / 255.

# Cluster the data points into 10 groups using K-Means++.
model = KMeans(n_clusters=10,
               init='k-means++',   # default
               n_init=5,
               max_iter=50)

model.fit(x)
clust = model.predict(x)
centroids = model.cluster_centers_
```

```
# Check out the images for each cluster.
for k in np.unique(clust):
    # Find 10 images belonging to cluster k, and centroid image.
    idx = np.where(clust == k)[0]
    images = x[idx[:10]]
    centroid = centroids[k, :]

    # Find 10 images closest to each centroid image.
    # d = np.sqrt(np.sum((x[idx] - centroid)**2, axis=1))
    # nearest = np.argsort(d)[:10]
    # images = x[idx[nearest]]

    # display the centroid image
    f = plt.figure(figsize=(8, 2))
    image = centroid.reshape(28, 28)
    ax = f.add_subplot(1, 11, 1)
    ax.imshow(image, cmap=plt.cm.bone)
    ax.grid(False)
    ax.set_title("C")
    ax.xaxis.set_ticks([])
    ax.yaxis.set_ticks([])
    plt.tight_layout()

    # display 10 images belonging to the centroid
    for i in range(10):
        image = images[i].reshape(28,28)
        ax = f.add_subplot(1, 11, i + 2)
        ax.imshow(image, cmap=plt.cm.bone)
        ax.grid(False)
        ax.set_title(k)
        ax.xaxis.set_ticks([])
        ax.yaxis.set_ticks([])
        plt.tight_layout()
```

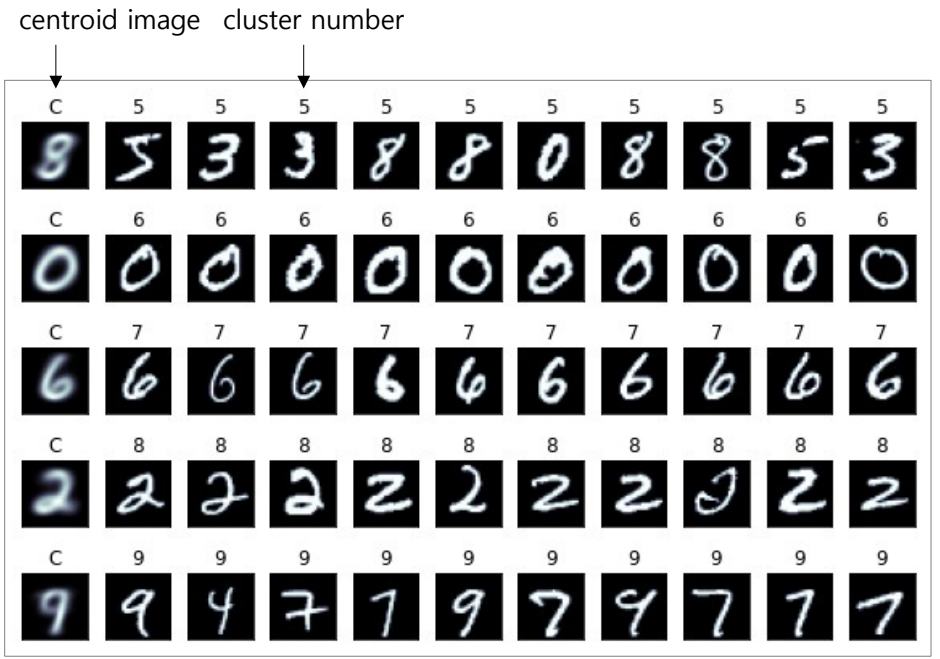
MX-AI

- Implement K-Means++ using sklearn's KMeans: MNIST clustering

Images belonging to each cluster



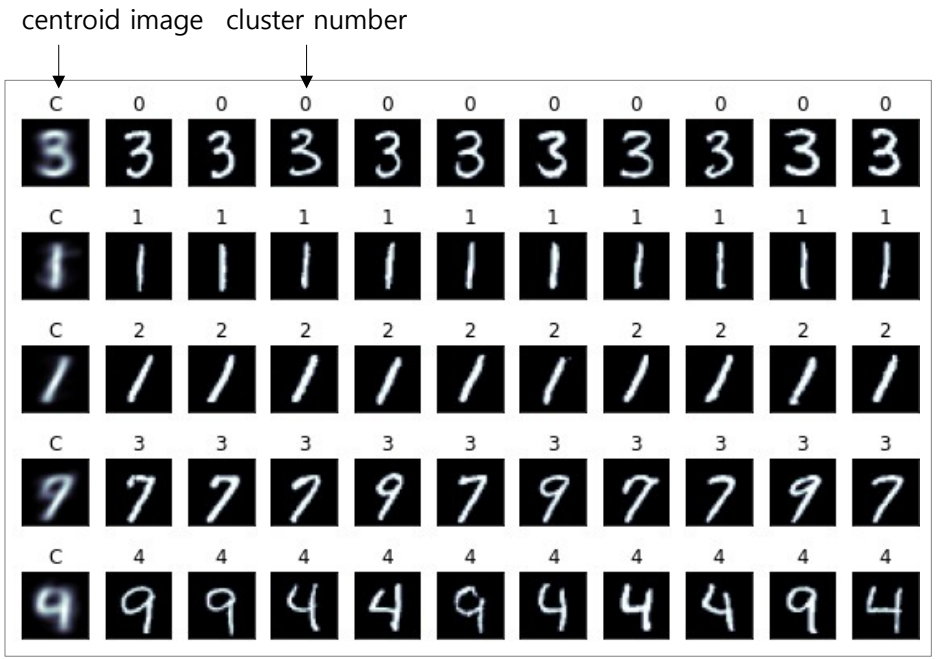
Images belonging to each cluster



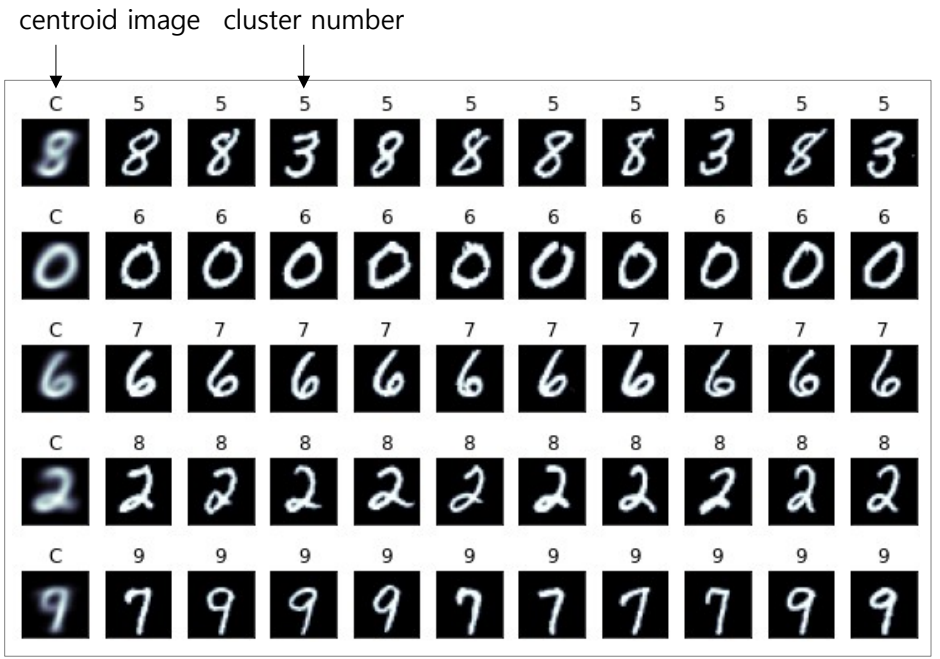
MX-AI

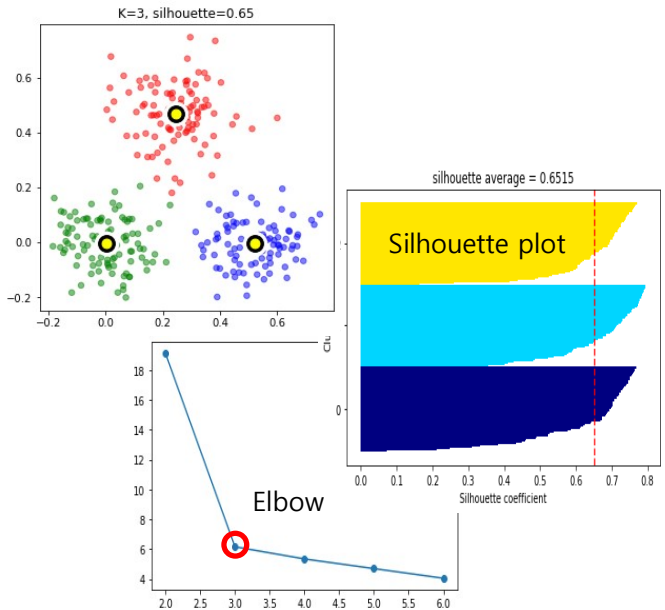
- Implement K-Means++ using sklearn's KMeans: MNIST clustering

Images closest to each centroid image



Images closest to each centroid image





7. K-Means Clustering

Part 4: Finding the optimal number of clusters

This video was produced in Korean, translated into English, and the audio was generated by AI (TTS).

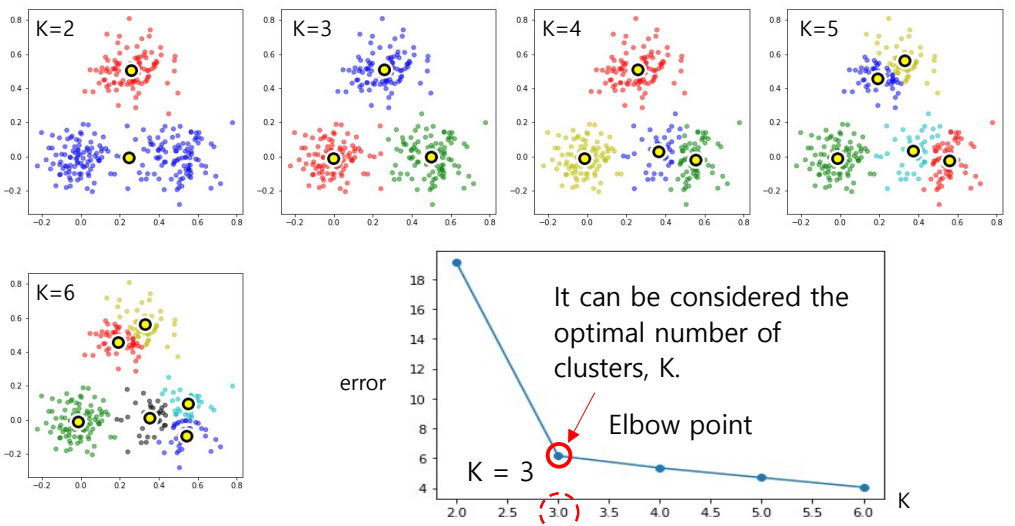
www.youtube.com/@meanxai

MX-AI

▪ Finding the optimal number of clusters – Elbow method

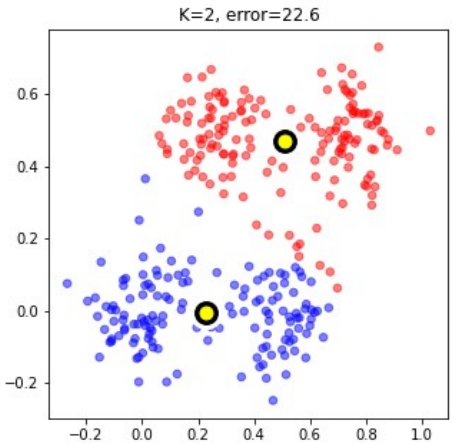
- The elbow method is a graphical method to find the optimal number of clusters, K in K-Means.
- Perform K-Means by varying K value and measure the errors. In the example below, if you set K small at first and gradually increase K, the error will gradually become smaller. At first the error decreases quickly, but later the error decreases slowly. In the error graph below, the point where the graph forms an elbow is likely to be the optimal K value.

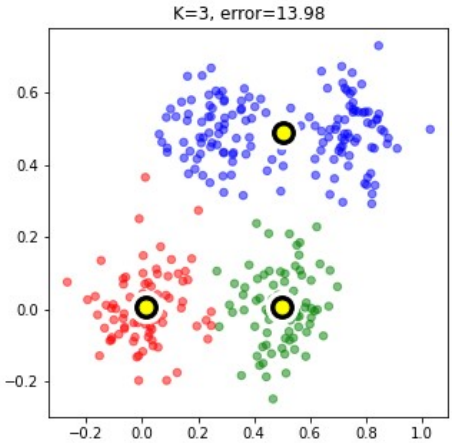
```
# [MXML-7-04] 5.sklearn(elbow).py
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Generate training data
cent3 = [[0., 0.],[0.25, 0.5],[0.5, 0.]]  # 3 clusters
cent4 = [[0., 0.],[0.25, 0.5],[0.5, 0.],[0.75, 0.5]] # 4 clusters
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=cent3, # 3 clusters
                  cluster_std=0.1, center_box=(-1., 1.))

# Color-code visualization of the training data and clusters.
def plot_cluster(x, cluster, centroid, title):
    plt.figure(figsize=(5, 5))
    color = [['r', 'b', 'g', 'y', 'c', 'k'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.title(title)
    plt.show()

inertia = []
for k in range(2, 7): # Measure the model's error by varying k.
    model = KMeans(n_clusters=k, n_init=5))
    model.fit(x)
    inertia.append(model.inertia_)
    # Visualize the clusters and centroids.
    title='K='+str(k) + ', error=' + str(np.round(inertia[-1], 2))
    plot_cluster(x, model.labels_, model.cluster_centers_, title)

# Observe the error change for varying K.
plt.plot(np.arange(2,7), inertia, 'o-')
plt.xlabel('K')
plt.ylabel('inertia')
plt.show()
```

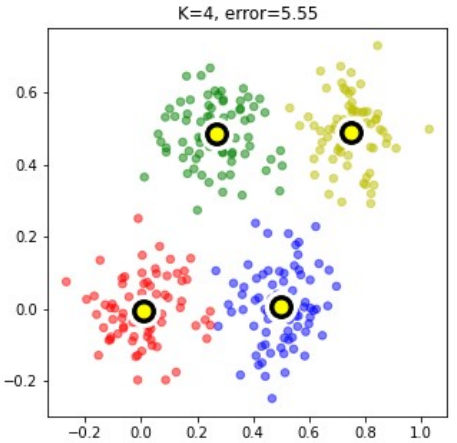


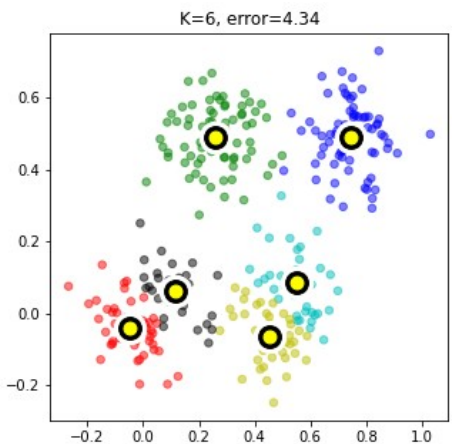
MX-AI

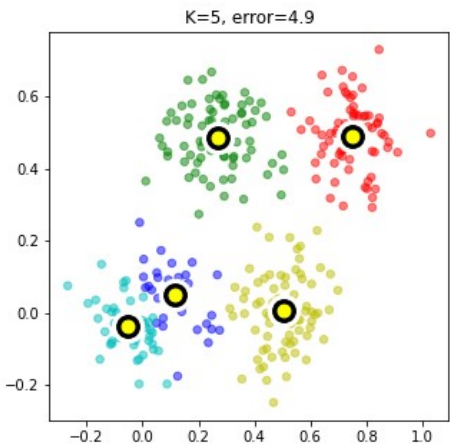
- Finding the optimal number of clusters – Elbow method

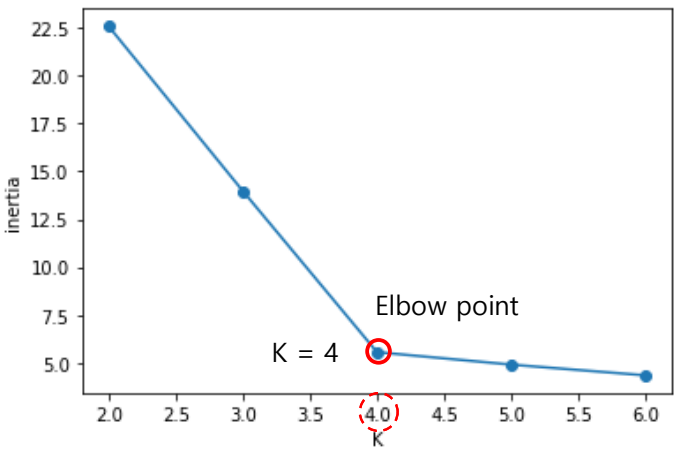










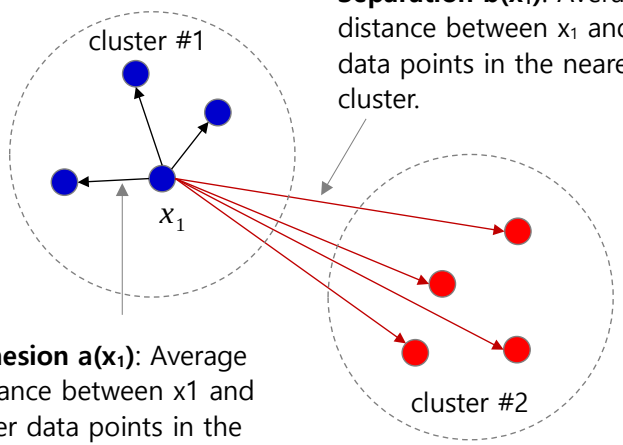


MX-AI

▪ Finding the optimal number of clusters – Silhouette method

- Silhouette method is a way to measure the clustering performance and can be used to determine the optimal number of clusters.
- Silhouette method uses a silhouette coefficient (s) which combines both cohesion (a) and separation (b).
- Cohesion is a measure of how closely the data points in a cluster are related to each other.
- Separation is a measure of how different or separated one cluster with another cluster.
- The higher the silhouette coefficient, the better the clustering performance.

▪ Good clustering



Separation b(x_1): Average distance between x_1 and the data points in the nearest cluster.

Cohesion a(x_1): Average distance between x1 and other data points in the same cluster.

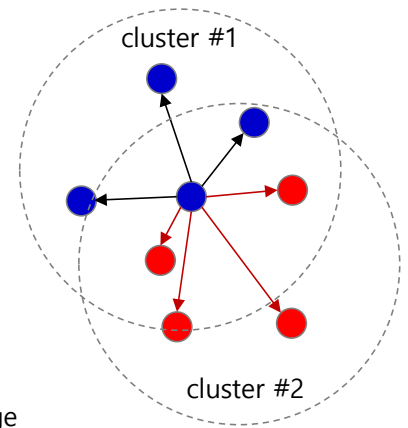
if b > a

$\frac{a}{b}$ → small

$1-\frac{a}{b}$ → large

The separation is large compared to cohesion.

▪ Bad clustering (Clusters overlap)



if b > a

$\frac{a}{b}$ → large

$1-\frac{a}{b}$ → small

The separation is small compared to cohesion.

▪ **Silhouette coefficient**

$$s(x_i)=1-\frac{a(x_i)}{b(x_i)}=\frac{b(x_i)-a(x_i)}{b(x_i)} \quad \text{(if b > a)}$$

$$s(x_i)=0 \quad \text{(if b = a)}$$

$$s(x_i)=\frac{a(x_i)}{b(x_i)}-1=\frac{a(x_i)-b(x_i)}{b(x_i)} \quad \text{(if b < a)}$$

$$(0 \le s(x_i) \le 1)$$

Or

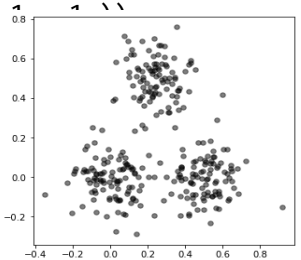
$$s(x_i)=\frac{b(x_i)-a(x_i)}{max(b(x_i),a(x_i))} \quad \leftarrow \text{normalizing}$$

$$(-1 \le s(x_i) \le 1)$$

MX-AI

▪ Code to calculate the silhouette score

```
# [MXML-7-04] 6.silhouette(score).py
# This code can be found at github.com/meanxai/machine_learning.
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import euclidean_distances as e_dist
import matplotlib.pyplot as plt

# Generate training data
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, center_box=( 1    1 ))

# training and prediction
K = 3
model = KMeans(n_clusters=K, n_init=5)
model.fit(x)
y_clust = model.predict(x)
centroids = model.cluster_centers_

# Find the data points that belong to each cluster.
# k_idx[i]: Indices of data points belonging to cluster i.
k_idx = []
for kk in range(K):
    k_idx.append(np.where(y_clust == kk)[0])

s_score1 = 0
for i in range(x.shape[0]):
    # Calculate the cohesion
    a_idx = k_idx[y_clust[i]]
    a = np.sum(e_dist(x[i].reshape(1, -1), x[a_idx])[0])
    a /= a_idx.shape[0] - 1    # Average except for x[i]
```




```
    # Calculate the separation
    # find the nearest centroid
    c_dist = e_dist(x[i].reshape(1, -1), centroids)[0]
    c_dist[y_clust[i]] = 9999    # Exclude the cluster that x[i]
                                 # belongs to.
    c_nearest = np.argmin(c_dist)

    # calculate b
    b_idx = k_idx[c_nearest]
    b = np.mean(e_dist(x[i].reshape(1, -1), x[b_idx])[0])

    # calculate silhouette coefficient
    s = (b - a) / np.max([b, a])
    s_score1 += s

s_score1 /= x.shape[0] # average silhouette coefficient
print('K = {}, silhouette score = {:.4f}'. format(K, s_score1))

# Compare the score with sklearn's silhouette_score() result.
from sklearn.metrics import silhouette_score

s_score2 = silhouette_score(x, y_clust, metric='euclidean')
print('K = {}, silhouette score (sklearn) = {:.4f}'\
      .format(K, s_score2))
```

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

```
Results:

K = 3, silhouette score = 0.6478
K = 3, silhouette score (sklearn) = 0.6478
```

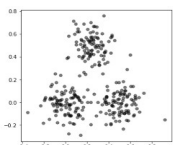
* Both results are identical.

* Reference: Sebastian Raschka, 2015, Python Machine Learning 2nd edition (p. 363)
Chap 11. Quantifying the quality of clustering via silhouette plots

MX-AI

▪ Code to find the optimal number of clusters using the Silhouette method

```
# [MXML-7-04] 7.silhouette(plot).py
# This code can be found at github.com/meanxai/machine_learning.
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from matplotlib import cm
from sklearn.metrics import silhouette_samples
import matplotlib.pyplot as plt

# Color-code visualization of the training data and clusters.
def plot_cluster(x, cluster, centroid, title):
    plt.figure(figsize=(5, 5))
    color = [['r', 'b', 'g', 'y', 'c', 'k'][a] for a in cluster]
    plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
    plt.scatter(centroid[:, 0], centroid[:, 1], s=500, c='white')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=250, c='black')
    plt.scatter(centroid[:, 0], centroid[:, 1], s=80, c='yellow')
    plt.title(title)
    plt.show()

# Generate training data
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.1, center_box=(-1., 1.))

# Training KMeans models by varying k,
# and calculating the Silhouette scores for each data point.
for k in range(2, 6):
    model = KMeans(n_clusters=k, n_init=5)
    model.fit(x)
    y_clust = model.labels_
    centroids = model.cluster_centers_
```



```
# Color-code visualization of the training data and clusters.
# And display the silhouette score in the title.
    silhouette_vals = silhouette_samples(x, y_clust)
    silhouette_avg = np.mean(silhouette_vals)
    title='K=' + str(k) + ', silhouette=' + \
          str(np.round(silhouette_avg, 2))
    plot_cluster(x, y_clust, centroids, title)

    # Bar plot of silhouette coefficients for all data points.
    cluster_labels = np.unique(y_clust)
    n_clusters = cluster_labels.shape[0]
    y_ax_lower, y_ax_upper = 0, 0
    yticks = []
    plt.figure(figsize=(6, 4))
    for i, c in enumerate(cluster_labels):
        c_silhouette_vals = silhouette_vals[y_clust == c]
        c_silhouette_vals.sort()
        y_ax_upper += len(c_silhouette_vals)
        color = cm.jet(float(i) / n_clusters)
        plt.barh(range(y_ax_lower, y_ax_upper), c_silhouette_vals,
                 height=1.0, edgecolor='none', color=color)

        yticks.append((y_ax_lower + y_ax_upper) / 2.)
        y_ax_lower += len(c_silhouette_vals)

    plt.axvline(silhouette_avg, color="red", linestyle="--")

    plt.yticks(yticks, cluster_labels)
    plt.ylabel('Cluster')
    plt.xlabel('Silhouette coefficient')
    plt.title('silhouette average = {:.4f}'.format(silhouette_avg))
    plt.tight_layout()
    plt.show()
```

MX-AI

▪ Code to find the optimal number of clusters using the Silhouette method: Results

- Do the silhouette plots of all clusters exceed the average silhouette score? If yes, this is likely optimal.
- Are there large variations in the size of your cluster plots? If yes, this is likely not optimal.
- Are the silhouette plots of the clusters uniform in thickness? If yes, this is likely optimal.
- **It is best when K is 3.** Neither the elbow nor the silhouette method are perfect, but they are very helpful in determining optimal K.