Table of Contents

LAB11 tutorial for Machine Learning K-Means

> The document description are designed by Jia Yanhong in 2022. Nov. 20th

---

1 Objective

- Understand K-means algorithm theory
- Implement the k-means algorithm from scratch in python
- Complete the LAB assignment and submit it to BB or sakai.

---

2 Introduction

K-means clustering is one of the simplest and popular **unsupervised** machine learning algorithms.

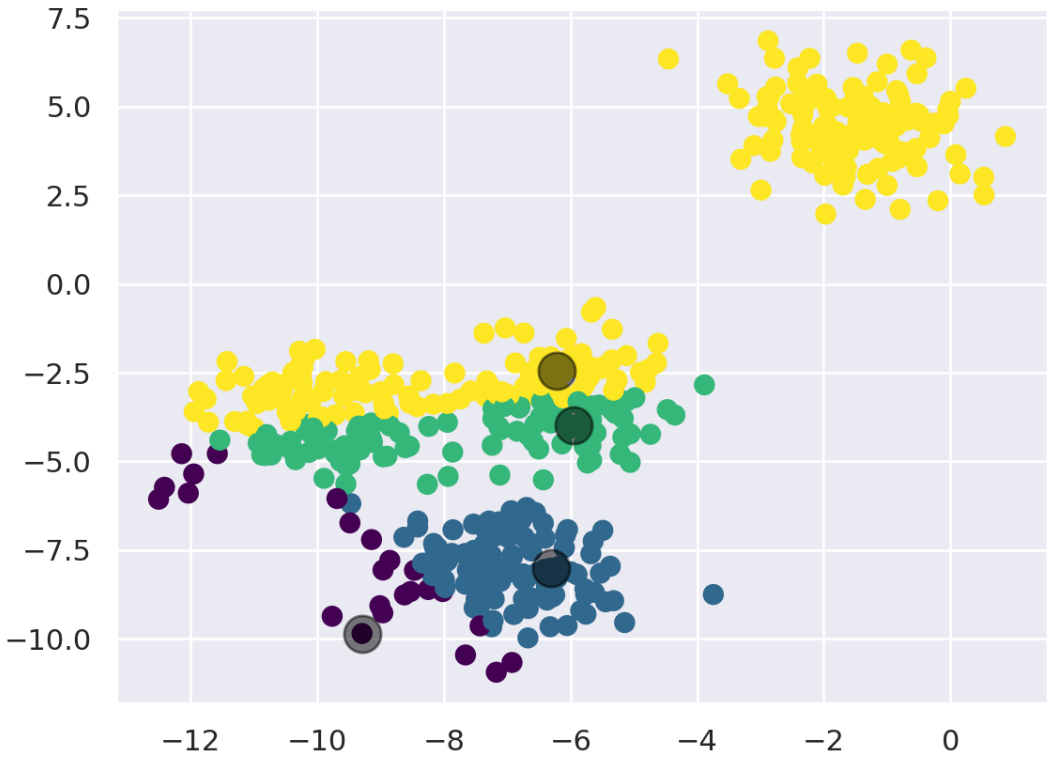
Its goal is to separate the data into K distinct non-overlapping subgroups (clusters) of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares.

$$\sum_{i=0}^{n} \min_{\mu_j \in C} \left(||x_i - \mu_j||^2 \right)$$

3 KMeans theory

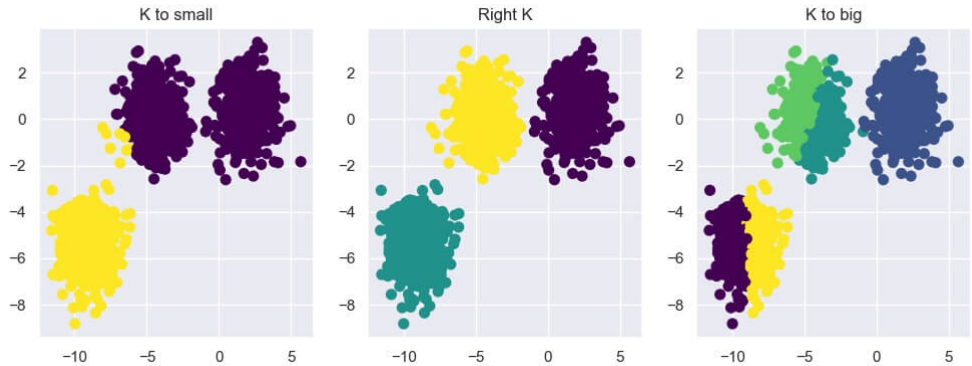
KMeans works as follows:

1. First, pick the number of clusters (For more info, check the "Choosing K" section).
2. Initialize the center points of the cluster (centroids) by shuffling the dataset and then selecting K data points for the centroids.
3. Assign data points to the cluster with the nearest centroid.
4. Recompute centroid position by taking the mean of all data points assigned to the cluster.
5. Repeat steps 3 and 4 for a set number of iterations or until the centroids aren't moving much between iterations anymore.



4 Choosing K

Choosing the right K value by hand can get quite tricky, especially if you're working with 3+ dimensional data. If you select a too small or big number for K, the result can be quite underwhelming.



In this section, I'll show you two methods commonly used to choose the right K value:

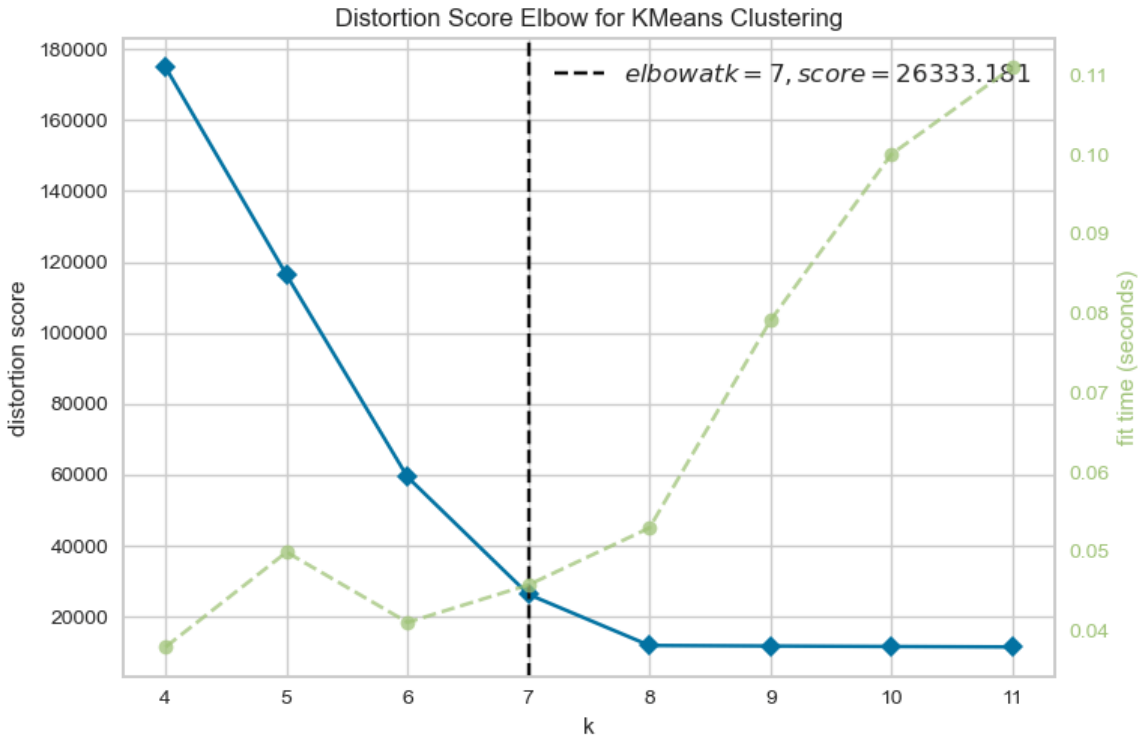
- The Elbow Method
- Silhouette Analysis

4.1 Elbow Method(手肘法)

The Elbow Method shows us what a good number for K is based on the sum of squared distances (SSE) between data points and their assigned clusters' centroid.

$$SSE = \sum_{j=0}^{C} \sum_{x_i \in C_j} \left(||x_i - \mu_j||^2 \right)$$

We pick k at the spot where the SSE starts to flatten out, which looks like an elbow. Below you can see an example created using Yellowbrick (https://www.scikit-yb.org/en/latest/api/cluster/elbow.html).



4.2 Silhouette Analysis(轮廓系数法)

The Silhouette Analysis can be used to study the separation distance between the resulting clusters. It displays a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation) and can thus be used to assess the number of clusters k.

The Silhouette Analysis is computed as follows:

- Compute the average distance between all data points in one cluster C_i

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j)$$

- For all data points i in cluster C_i compute the average distance to all points in another cluster C_k (where $C_k \neq C_i$)

$$b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i, j)$$

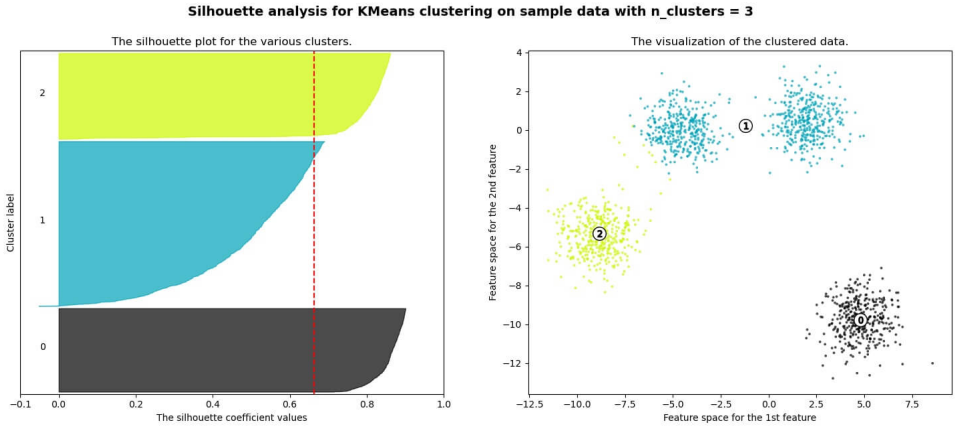
> The min is used, because we want to know the average distance to the closed cluster i is not a member of.

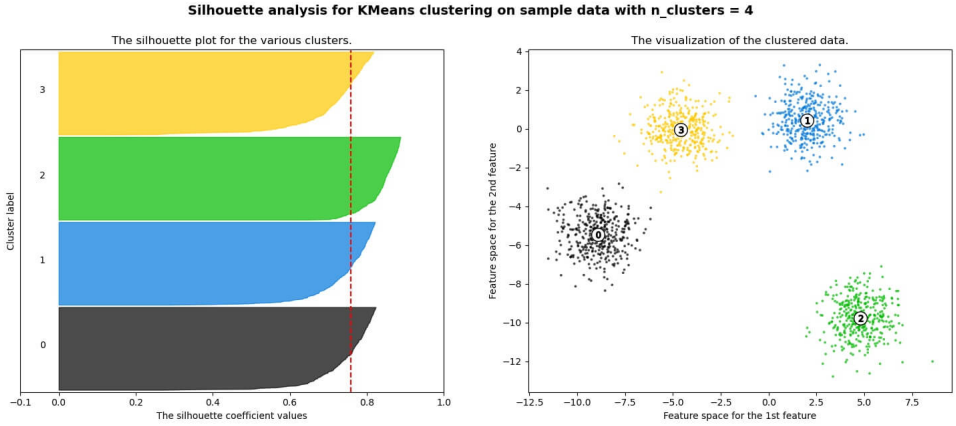
With a and b we can now calculate the silhouette coefficient:

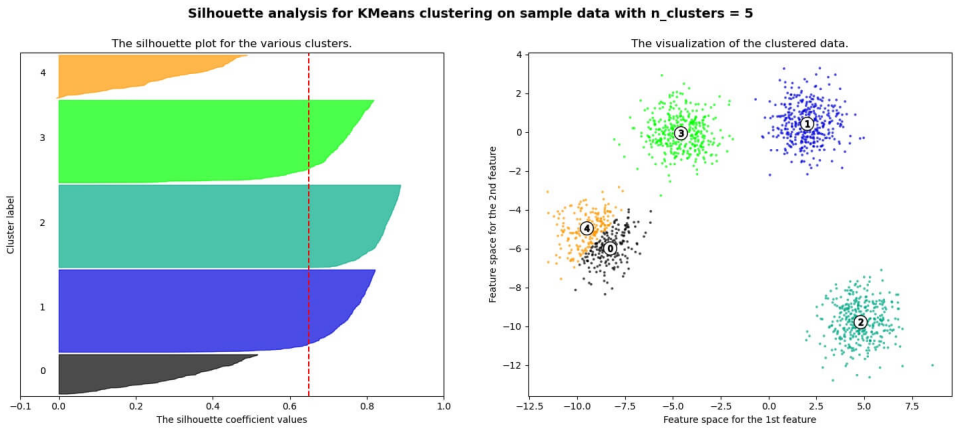
$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}, if|C_i| > 1$$

The coefficient can take values in the interval $[-1, 1]$. Zero means the sample is very close to the neighboring clusters. One means the sample is far away from the neighboring cluster, and negative one means the sample is probably assigned to the wrong cluster.

Below you can see an [example of silhouette analysis (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html)](https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html) using [Scikit Learn (https://scikit-learn.org/stable/index.html)](https://scikit-learn.org/stable/index.html):







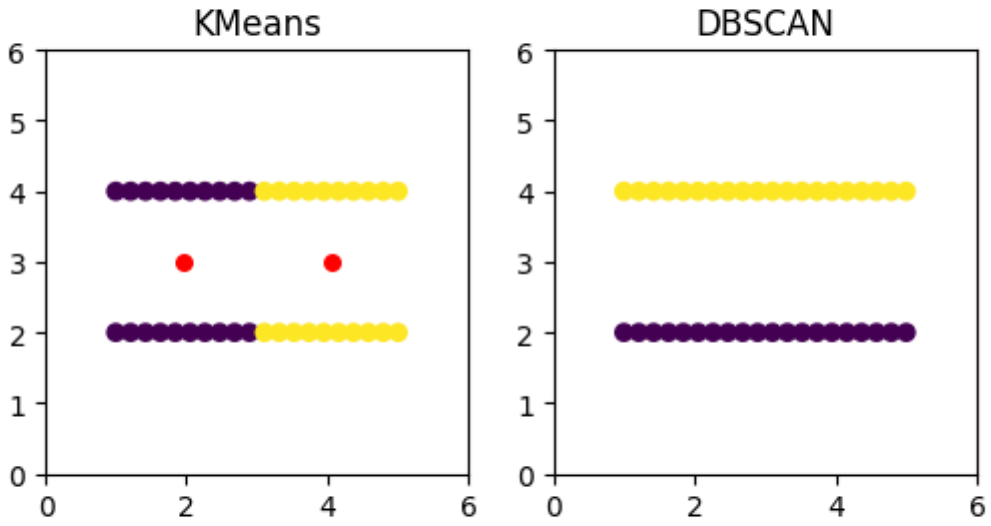
5 Advantages

KMeans is an easy-to-implement algorithm that is also quite fast with an average complexity of $O(k * n * T)$, where n is the number of samples, and T is the number of iteration.

6 Drawbacks

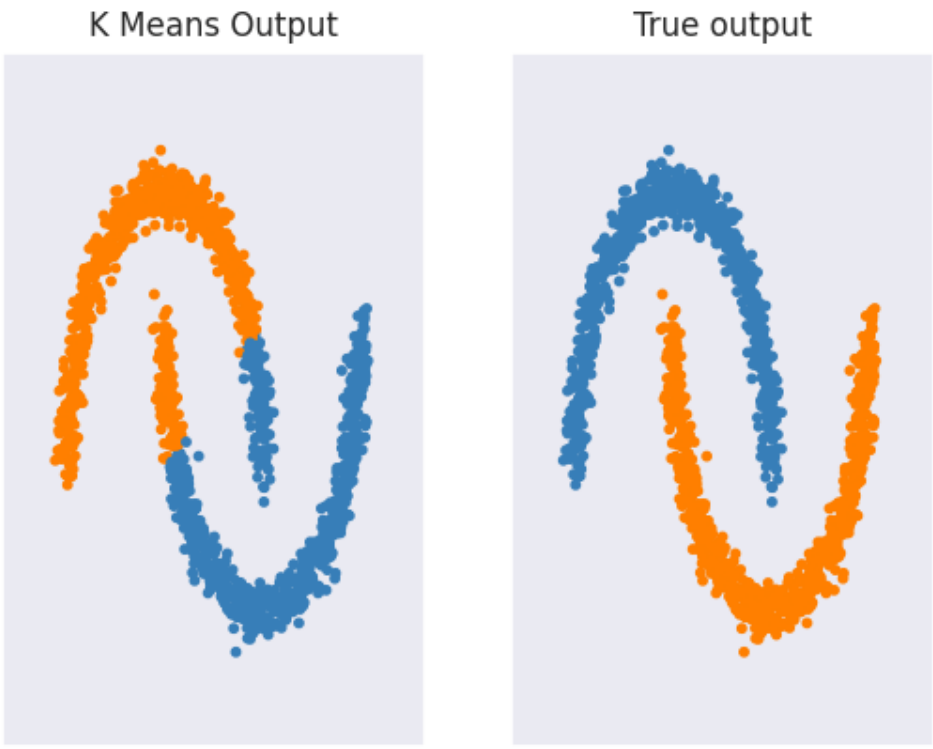
As mentioned above, KMeans makes use of the **sum-of-squares criterion**, which works well if the clusters have a spherical-like shape. It doesn't work well on many other types of data like complicated shapes, though. In this section, we'll go over a few cases where KMeans performs poorly.

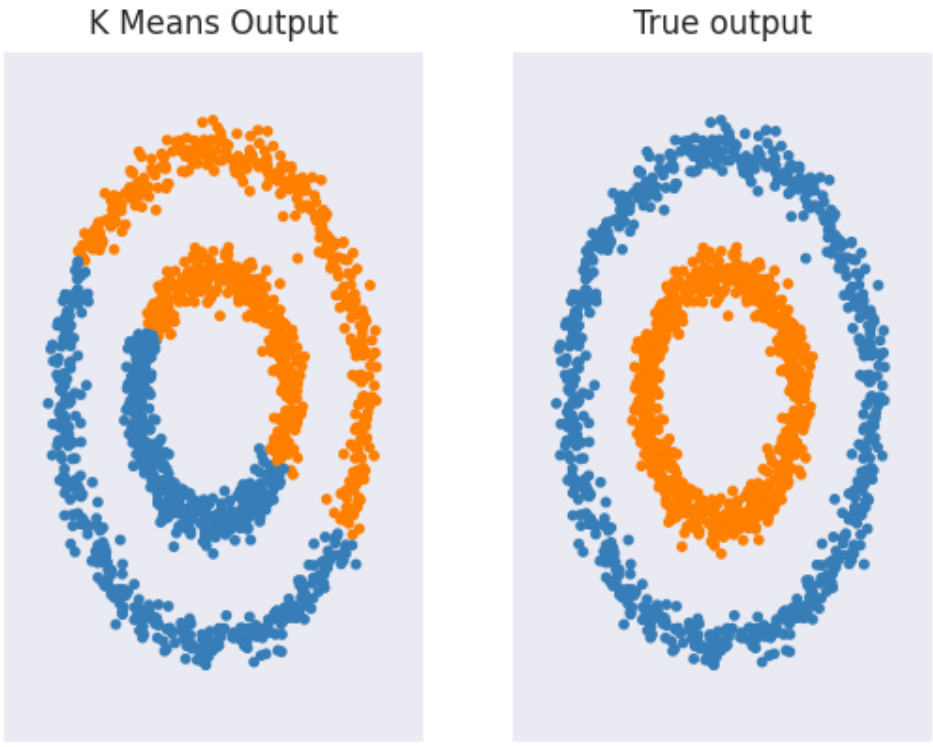
First, KMeans doesn't put data points that are far away from each other into the same cluster, even when they obviously should be because they underly some obvious structure like points on a line, for example.



In the image above, you can see that KMeans creates the clusters in between the two lines and therefore splits each line into one of two clusters rather than classifying each line as a cluster. On the right side, you can see the DBSCAN (Density-based spatial clustering of applications with noise) algorithm, which is able to separate the two lines without any issues.

Also, as mentioned at the start of the section KMeans performs poorly for complicated geometric shapes such as the moons and circles shown below.





Other clustering algorithms like Spectral Clustering, Agglomerative Clustering, or DBSCAN don't have any problems with such data. For a more in-depth analysis of how different clustering algorithms perform on different interesting 2d datasets, I recommend checking out 'Comparing different clustering algorithms on toy datasets' (https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html) from Scikit-Learn.

7 Other resources

- https://scikit-learn.org/stable/modules/clustering.html#k-means (https://scikit-learn.org/stable/modules/clustering.html#k-means)
- https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a (https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a)
- https://www.youtube.com/watch?v=4b5d3muPQmA (https://www.youtube.com/watch?v=4b5d3muPQmA)

8 LAB Assignment

Please finish the **Exercise** and answer **Questions**.

8.1 Exercise

In this lab, we will write a program to segment different objects in a video using *K-means* clustering. There are several steps:

- 1.Load video & extract frames
- 2.Implement *K-means* clustering
- 3.Write back to video

8.1.1 Import some libraries

In [3]:

```
import numpy as np
import cv2
import tqdm
import os
import sys
# color of different clusters
GBR = [[0, 0, 255],
       [0, 128, 255],
       [255, 0, 0],
       [128, 0, 128],
       [255, 0, 255]]

# path configuration
project_root = os.path.abspath('.')
output_path = os.path.join(project_root)
input_path = os.path.join(project_root)
if not os.path.exists(output_path):
    os.makedirs(output_path)
```

8.1.2 K-means

In this lab, you need to implement K-means, the rough procedure is:

1. **initialize centroids** of different classes

   In the simplest case, randomly choose centroids in original data
2. **calculate distances** between samples (pixels) and centroids

Since one sample (pixel) has 3 channels, you can calculate square sum of differences in each channel between it and centroids.

$$dist(S, C) = \sum_{i=1}^{3}(C_i - S_i)^2$$

$$\begin{cases} dist(S, C): \text{distance between a sample S and a centroid C} \\ C: \text{a centroid} \\ S: \text{a sample} \\ S_i: \text{the } i^{th} \text{ channel's value of S} \\ C_i: \text{the } i^{th} \text{ channel's value of C} \end{cases}$$

3. **classify** every samples

   A sample is belonging to the class whose centroid is closest to it among all centroids.

$$cls(S) = argmin(\sum_{i=1}^{3}(C_i^k - S_i)^2), k = 1, 2, \ldots, K$$

$$\begin{cases} cls(S): \text{class of a sample S} \\ K: \text{number of classes} \\ C^k: \text{centroid of } k^{th} \text{ class} \end{cases}$$

4. **update centroid**

   You can use mean of all samples in the same class to calculate new centroid.

$$C_i^k = \frac{1}{n^k}\sum_{n=1}^{n^k} S_{in}^k, \quad i = 1, 2, 3$$

$$\begin{cases} C_i^k: \text{the } i^{th} \text{channel's value of a centroid belonging to the } k^{th} \text{class} \\ n^k: \text{the number of samples in the } k^{th} \text{class} \\ S_{in}^k: \text{the } i^{th} \text{channel's value of a sample which is in the } k^{th} \text{class} \end{cases}$$

5. loop until classification result doesn't change

In addition, you may find there is code like this:

```
while ret:
    frame = np.float32(frame)
    h, w, c = frame.shape
    ...
```

Since if you don't converse the `dtype`, K-means hardly converges which means it will stuck into dead loop easily.

After you finish K-means, you will find the written video is hard to watch because **color** between adjacent frames **changes almost all the time**. Here, I want you to find a way to alleviate the situation yourself.

It isn't compulsory, you can try if you want.

In [4]:

```python
def kmeans(data: np.ndarray, n_cl: int):
    """
        K-means

    :param data:    original data
    :param n_cl:    number of classes
    :param seeds:   seeds
    :return:        new labels and new seeds
    """
    n_samples, channel = data.shape

    # TODO: firstly you should init centroids by a certain strategy
    centers = None

    old_labels = np.zeros((n_samples,))
    while True:
        # TODO: calc distance between samples and centroids
        distance = None
        # TODO: classify samples
        new_labels = old_labels

        # TODO: update centroids
        centers = centers

        if np.all(new_labels == old_labels):
            break
        old_labels = new_labels

    return old_labels
```

8.1.3 Load video and detect

We use `opencv` to read a video. Pay attention that data type of `frame` is `uint8`, not `int`; In this lab, frame has 3 channels. If you don't change `dtype` of frame into `unit8`, video you write will look strange which you can have a try.

In [5]:

```
def detect(video, n_cl=2):
    # load video, get number of frames and get shape of frame
    cap = cv2.VideoCapture(video)
    fps = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    size = (int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
            int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)))

    # instantiate a video writer
    video_writer = cv2.VideoWriter(os.path.join(output_path, "result_with_%dclz.mp4" % n_cl),
                                   cv2.VideoWriter_fourcc(*'mp4v'),
                                   (fps / 10),
                                   size,
                                   isColor=True)

    # initialize frame and seeds
    ret, frame = cap.read()


    print("Begin clustering with %d classes:" % n_cl)
    bar = tqdm.tqdm(total=fps)  # progress bar
    while ret:
        frame = np.float32(frame)
        h, w, c = frame.shape

        # k-means
        data = frame.reshape((h * w, c))
        labels = kmeans(data, n_cl=n_cl)

        # give different cluster different colors
        new_frame = np.zeros((h * w, c))
        # TODO: dye pixels with colors
        new_frame = new_frame.reshape((h, w, c)).astype("uint8")
        video_writer.write(new_frame)

        ret, frame = cap.read()
        bar.update()

    # release resources
    video_writer.release()
    cap.release()
    cv2.destroyAllWindows()


video_sample = os.path.join(input_path, "road_video.MOV")
detect(video_sample, n_cl=1)

```

```
Begin clustering with 1 classes:
100%|██████████| 35/35 [00:14<00:00,  2.35it/s]
```

8.1.4 Sample Result

8.2 Questions

1. What are the strengths of K-means; when does it perform well?
2. What are the weaknesses of K-means; when does it perform poorly?
3. What makes K-means a good candidate for the clustering problem, if you have enough knowledge about the data?