04a_LAB_KMeansClustering

May 22, 2022

1 Machine Learning Foundation

1.1 Course 4, Part a: K-Means Clustering LAB

Purpose: The purpose of this lab exercise is to learn how to use an unsupervised learning algorithm, **K-means** using sklearn.

At the end of this lab you will be able to:

1. Run a K-means algorithm.
2. Understand what parameters are customizable for the algorithm.
3. Know how to use the inertia curve to determine the optimal number of clusters.

1.1.1 K-Means Overview

K-means is one of the most basic clustering algorithms. It relies on finding cluster centers to group data points based on minimizing the sum of squared errors between each datapoint and its cluster center.

[1]:
```
# Force no warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
warnings.filterwarnings('ignore')

# Setup and imports
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.utils import shuffle

# Sets backend to render higher res images
%config InlineBackend.figure_formats = ['retina']
```

[2]:
```
plt.rcParams['figure.figsize'] = [6,6]
sns.set_style("whitegrid")
sns.set_context("talk")
```

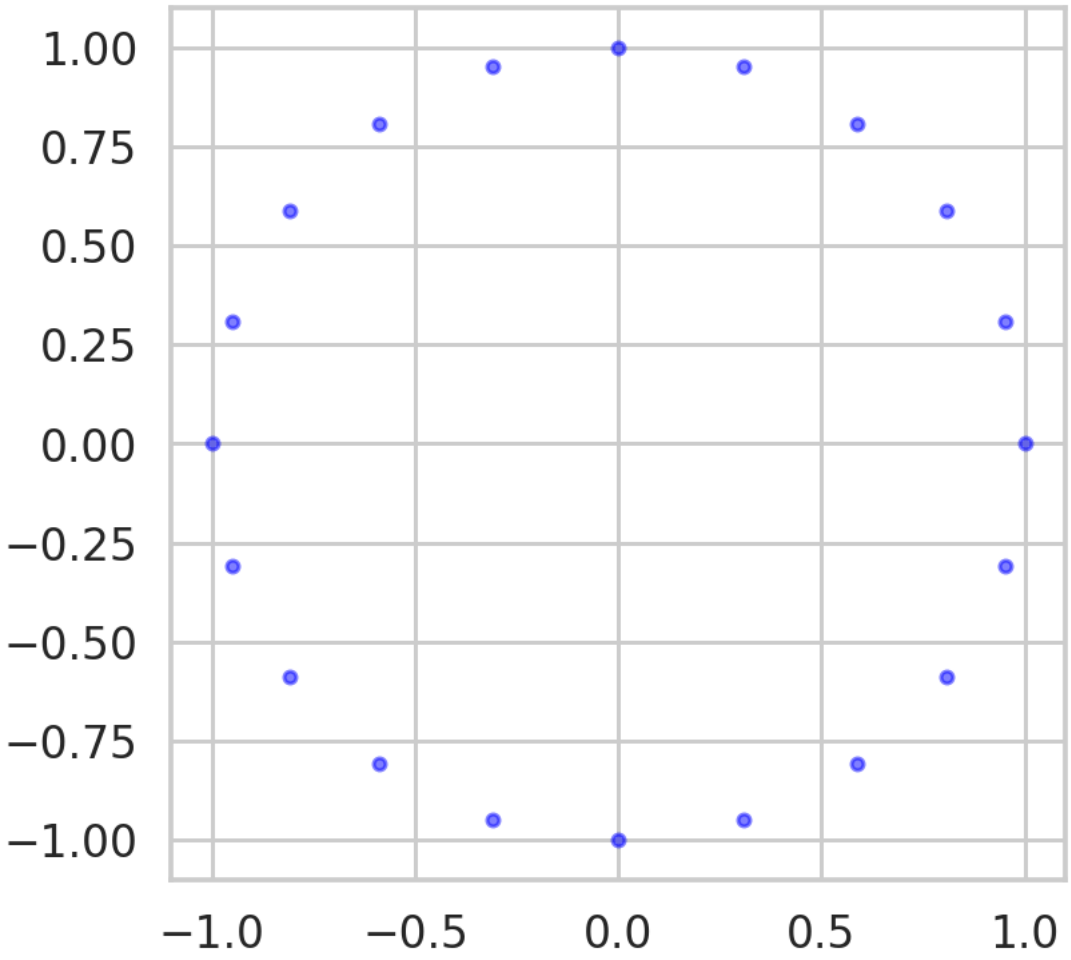
K-means clustering is one of the most simple clustering algorithms. One of the limitations is that it depends on the starting point of the clusters, and the number of clusters need to be defined beforehand.

1.1.2 Cluster starting points

Let's start by creating a simple dataset.

[3]:
```
# helper function that allows us to display data in 2 dimensions an highlights␣
 ↪the clusters
def display_cluster(X,km=[],num_clusters=0):
    color = 'brgcmyk'
    alpha = 0.5
    s = 20
    if num_clusters == 0:
        plt.scatter(X[:,0],X[:,1],c = color[0],alpha = alpha,s = s)
    else:
        for i in range(num_clusters):
            plt.scatter(X[km.labels_==i,0],X[km.labels_==i,1],c =␣
 ↪color[i],alpha = alpha,s=s)
            plt.scatter(km.cluster_centers_[i][0],km.cluster_centers_[i][1],c =␣
 ↪color[i], marker = 'x', s = 100)
```

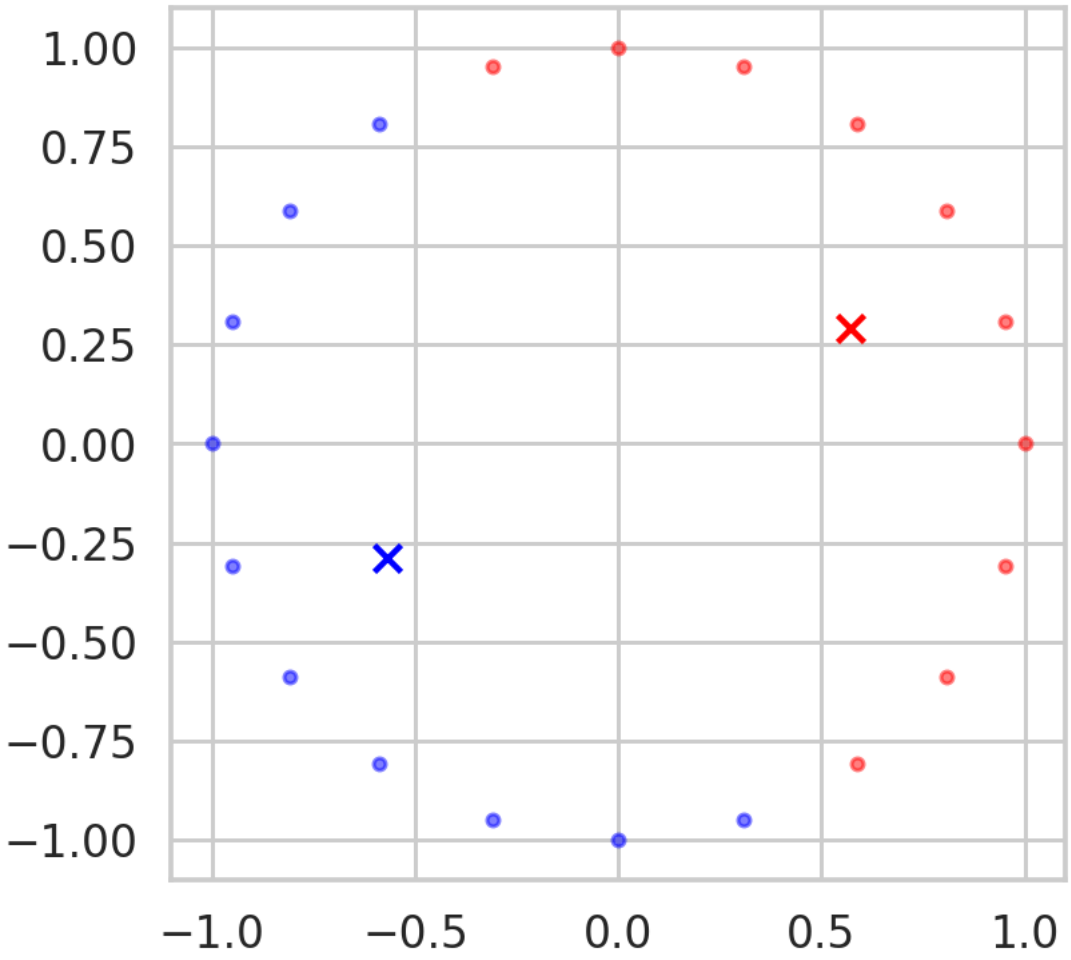
[5]:
```
angle = np.linspace(0,2*np.pi,20, endpoint = False)
X = np.append([np.cos(angle)],[np.sin(angle)],0).transpose()
display_cluster(X)
```



Let's now group this data into two clusters. We will use two different random states to initialize the algorithm. Settign a the **random state** variable is useful for testing and allows us to seed the randomness (so we get the same results each time).

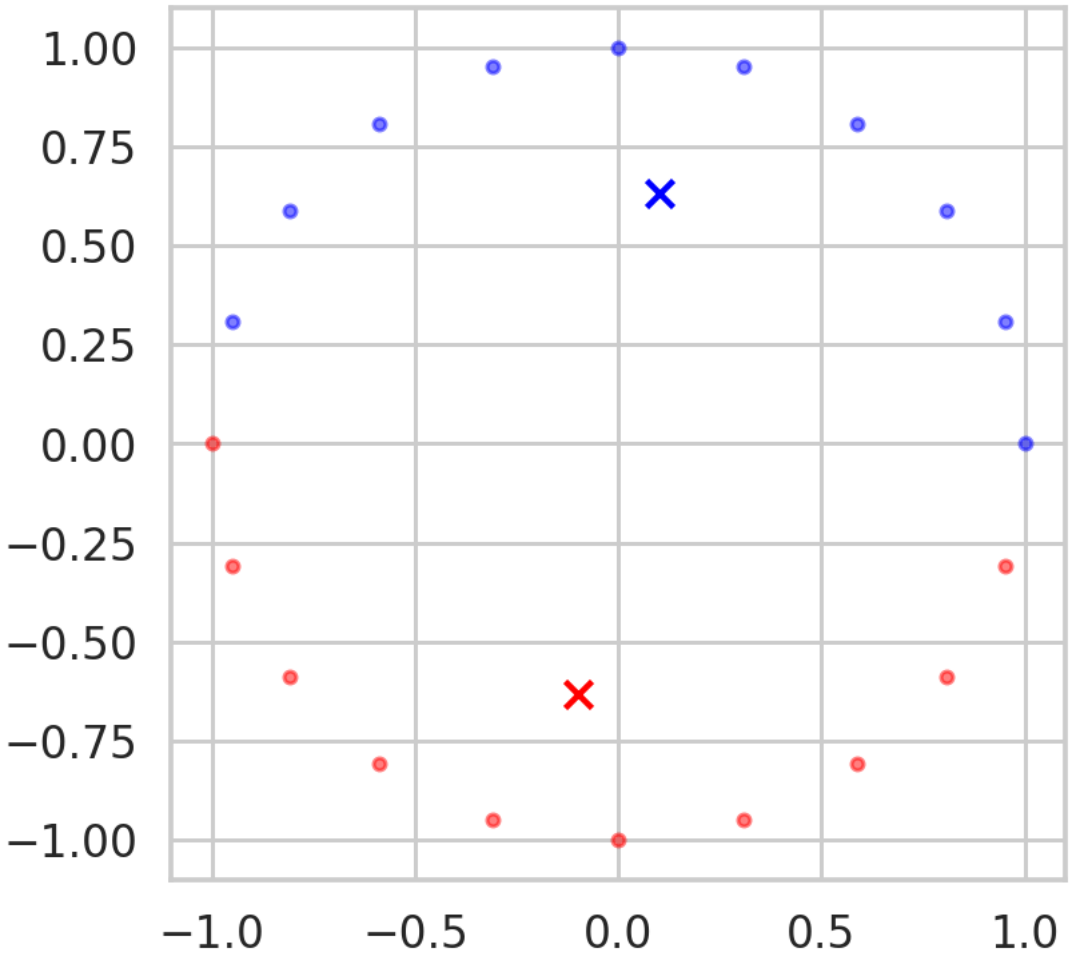
Clustering with a random state of 10:

[6]:
```
num_clusters = 2
km = KMeans(n_clusters=num_clusters,random_state=10,n_init=1) # n_init, number␣
 ↪of times the K-mean algorithm will run
km.fit(X)
display_cluster(X,km,num_clusters)
```



Clustering with a random state of 20:

[7]:
```
km = KMeans(n_clusters=num_clusters,random_state=20,n_init=1)
km.fit(X)
display_cluster(X,km,num_clusters)
```



1.2 Question:

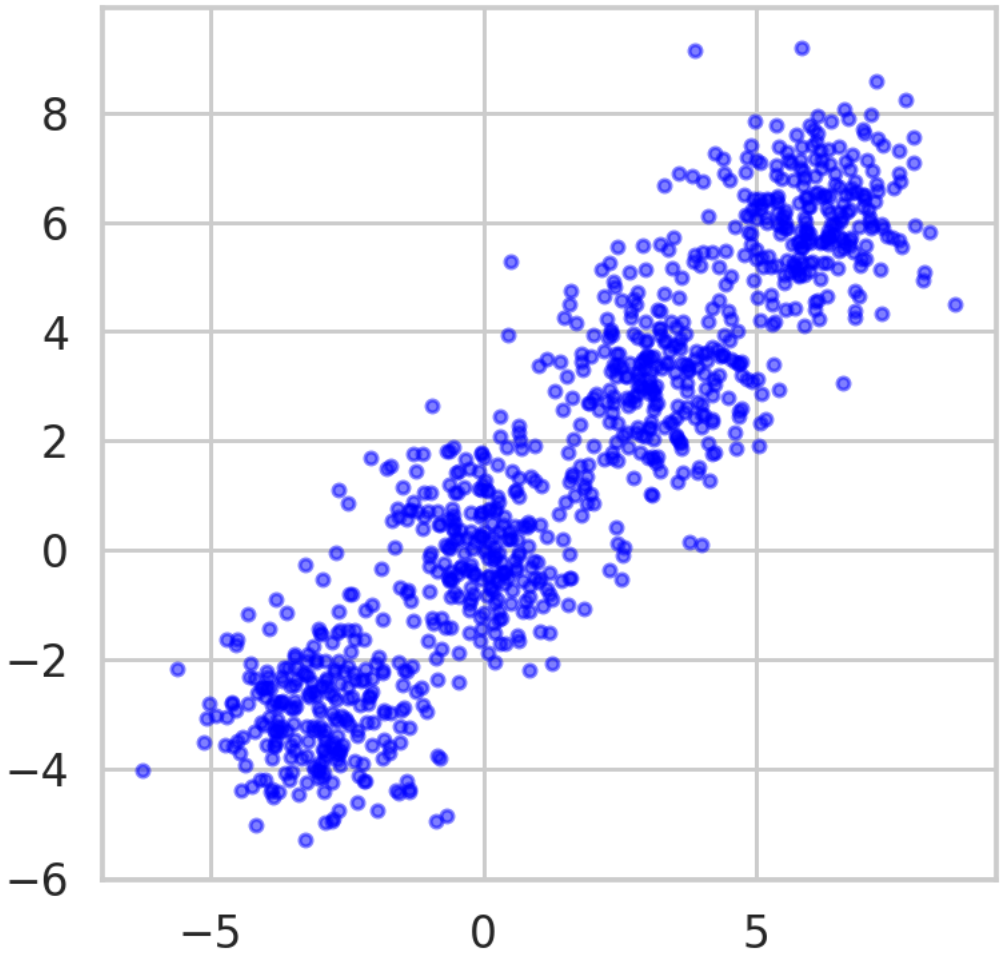
Why are the clusters different when we run the K-means twice?

It's because the starting points of the cluster centers have an impact on where the final clusters lie. The starting point of the clusters is controlled by the random state.

1.2.1 Determining optimium number of clusters

Let's create a new dataset that visually consists on a few clusters and try to group them.

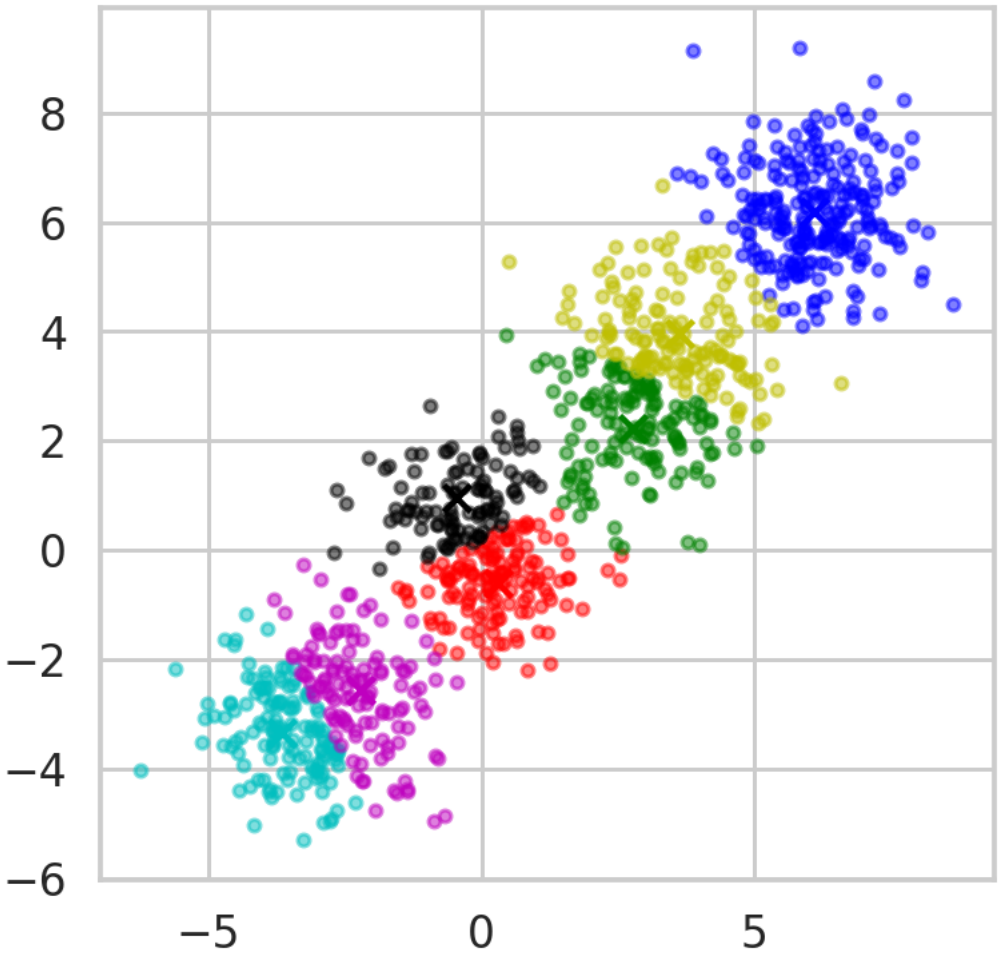
[8]:
```
n_samples = 1000
n_bins = 4
centers = [(-3, -3), (0, 0), (3, 3), (6, 6)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=1.0,
                  centers=centers, shuffle=False, random_state=42)
display_cluster(X)
```



How many clusters do you observe?

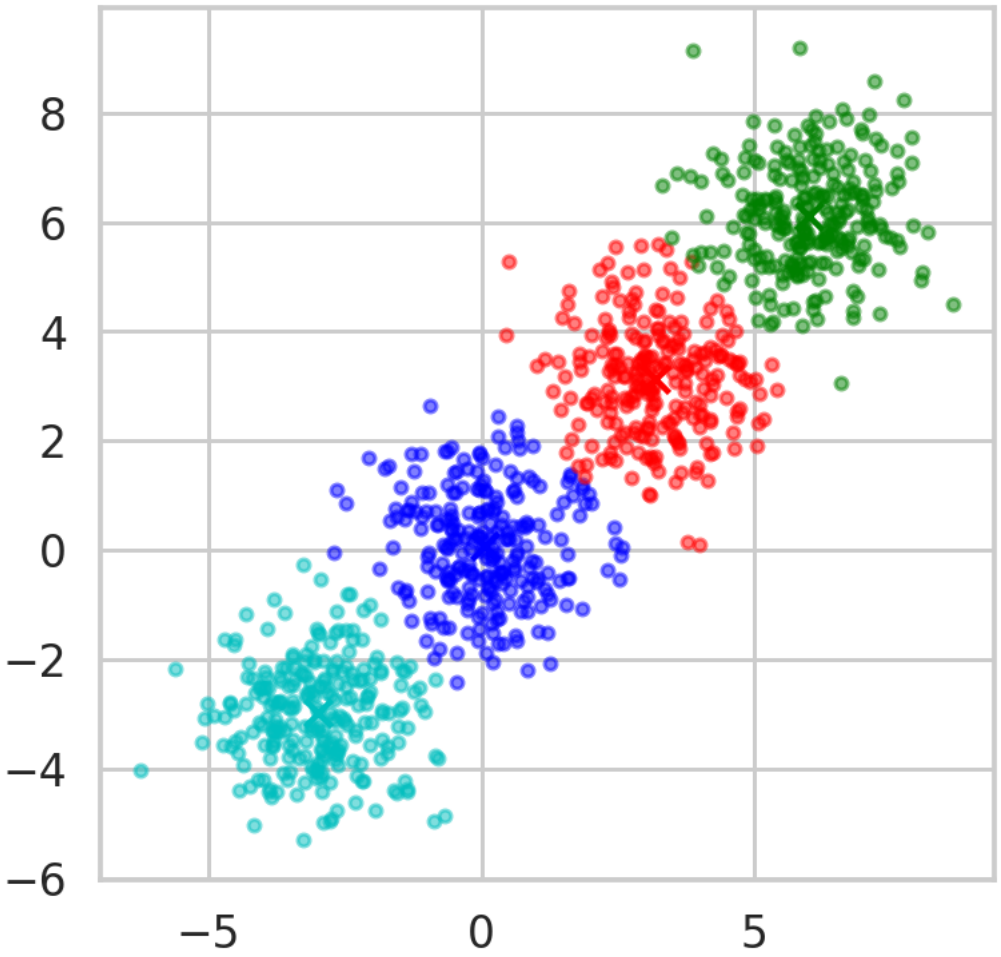
Let's run K-means with seven clusters.

[9]:
```
num_clusters = 7
km = KMeans(n_clusters=num_clusters)
km.fit(X)
display_cluster(X,km,num_clusters)
```



Now let's re-run the algorithm with four clusters.

[10]:
```
num_clusters = 4
km = KMeans(n_clusters=num_clusters)
km.fit(X)
display_cluster(X,km,num_clusters)
```



Should we use four or seven clusters?

- In this case it may be visually obvious that four clusters is better than seven.
- This is because we can easily view the data in two dimensional space.
- However, real world data usually has more than two dimensions.
- A dataset with a higher dimensional space is hard to visualize.
- A way of solving this is to plot the **inertia**

inertia: (sum of squared error between each point and its cluster center) as a function of the number of clusters.

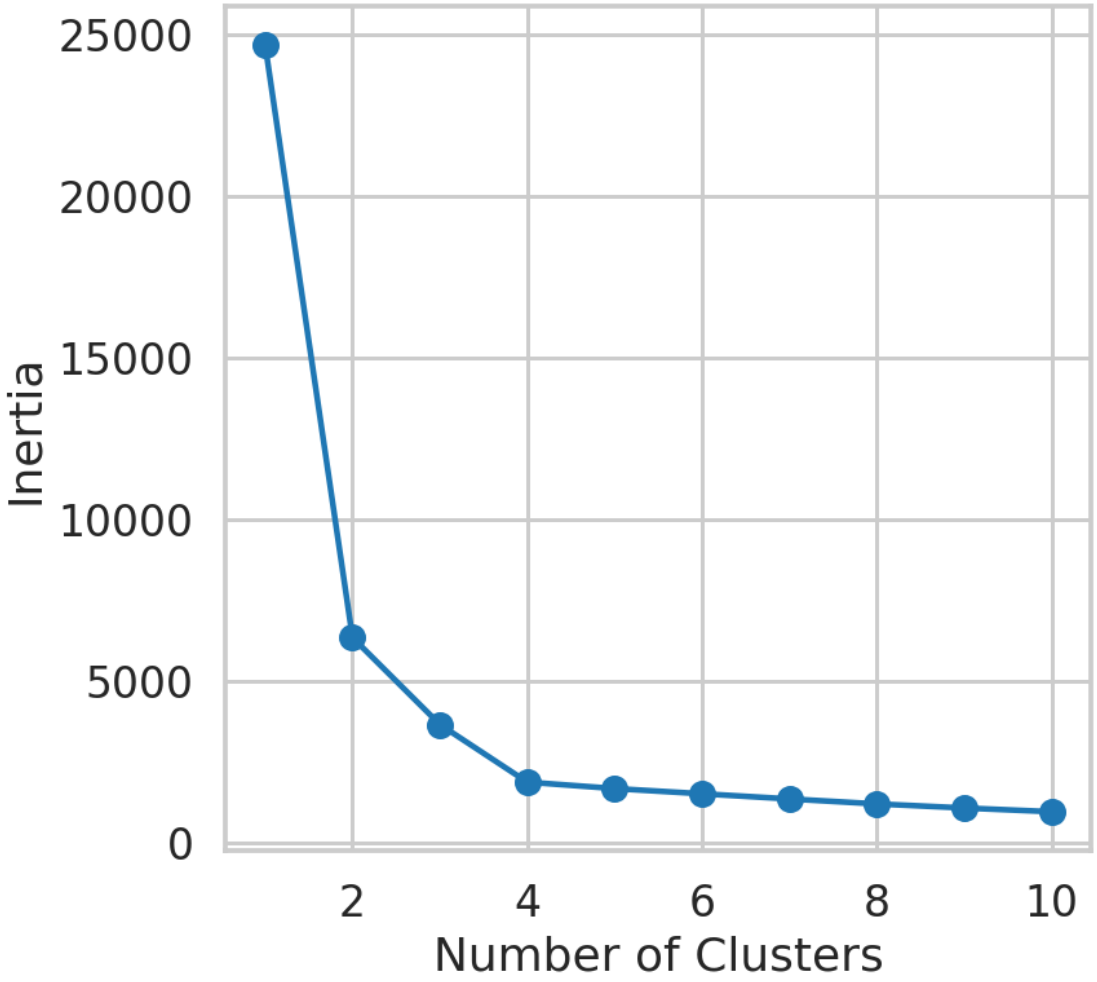
[11]:
```
km.inertia_
```

[11]:
```
1880.174402277563
```

1.2.2 Problem 1:

Write code that calculates the inertia for 1 to 10 clusters, and plot the inertia as a function of the number of clusters.

[12]:
```
### BEGIN SOLUTION
inertia = []
list_num_clusters = list(range(1,11))
for num_clusters in list_num_clusters:
    km = KMeans(n_clusters=num_clusters)
    km.fit(X)
    inertia.append(km.inertia_)

plt.plot(list_num_clusters,inertia)
plt.scatter(list_num_clusters,inertia)
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia');
### END SOLUTION
```



Where does the elbow of the curve occur?

What do you think the inertia would be if you have the same number of clusters and data points?

1.2.3 Clustering Colors from an Image

The next few exercises use an image of bell peppers. Let's start by loading it:

[13]:
```
img = plt.imread('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
↪cloud/IBM-ML0187EN-SkillsNetwork/labs/module%201/images/peppers.jpg',␣
↪format='jpeg')
plt.imshow(img)
plt.axis('off')
```

[13]:
```
(-0.5, 639.5, 479.5, -0.5)
```

[14]:
```
img.shape
```

[14]:
```
(480, 640, 3)
```

The image above has 480 pixels in height and 640 pixels in width. Each pixel has 3 values that represent how much red, green and blue it has. Below you can play with different combinations of RGB to create different colors. In total, you can create $256^3 = 16,777,216$ unique colors.

[15]:
```
# assign values for the RGB.  Each value should be between 0 and 255
R = 35
G = 95
B = 131
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off')
```

[15]:
```
(-0.5, 0.5, 0.5, -0.5)
```

First we will reshape the image into a table that has a pixel per row and each column represents the red, green and blue channel.

[16]:
```
img_flat = img.reshape(-1, 3)
img_flat[:5,:]
```

[16]:
```
array([[15, 18, 25],
       [26, 16, 24],
       [42, 15, 22],
       [65, 16, 22],
```

```
       [85, 14, 22]], dtype=uint8)
```

Since there are 480x640 pixels we get 307,200 rows!

[17]:
```
img_flat.shape
```

[17]:
```
(307200, 3)
```

Let's run K-means with 8 clusters.

[18]:
```
kmeans = KMeans(n_clusters=8, random_state=0).fit(img_flat)
```

Now let's replace each row with its closest cluster center.

[19]:
```
img_flat2 = img_flat.copy()

# loops for each cluster center
for i in np.unique(kmeans.labels_):
    img_flat2[kmeans.labels_==i,:] = kmeans.cluster_centers_[i]
```

We now need to reshape the the data from 307,200 x 3 to 480 x 640 x 3

[20]:
```
img2 = img_flat2.reshape(img.shape)
plt.imshow(img2)
plt.axis('off');
```

1.2.4 Problem 2:

Write a function that receives the image and number of clusters (k), and returns (1) the image quantized into k colors, and (2) the inertia.

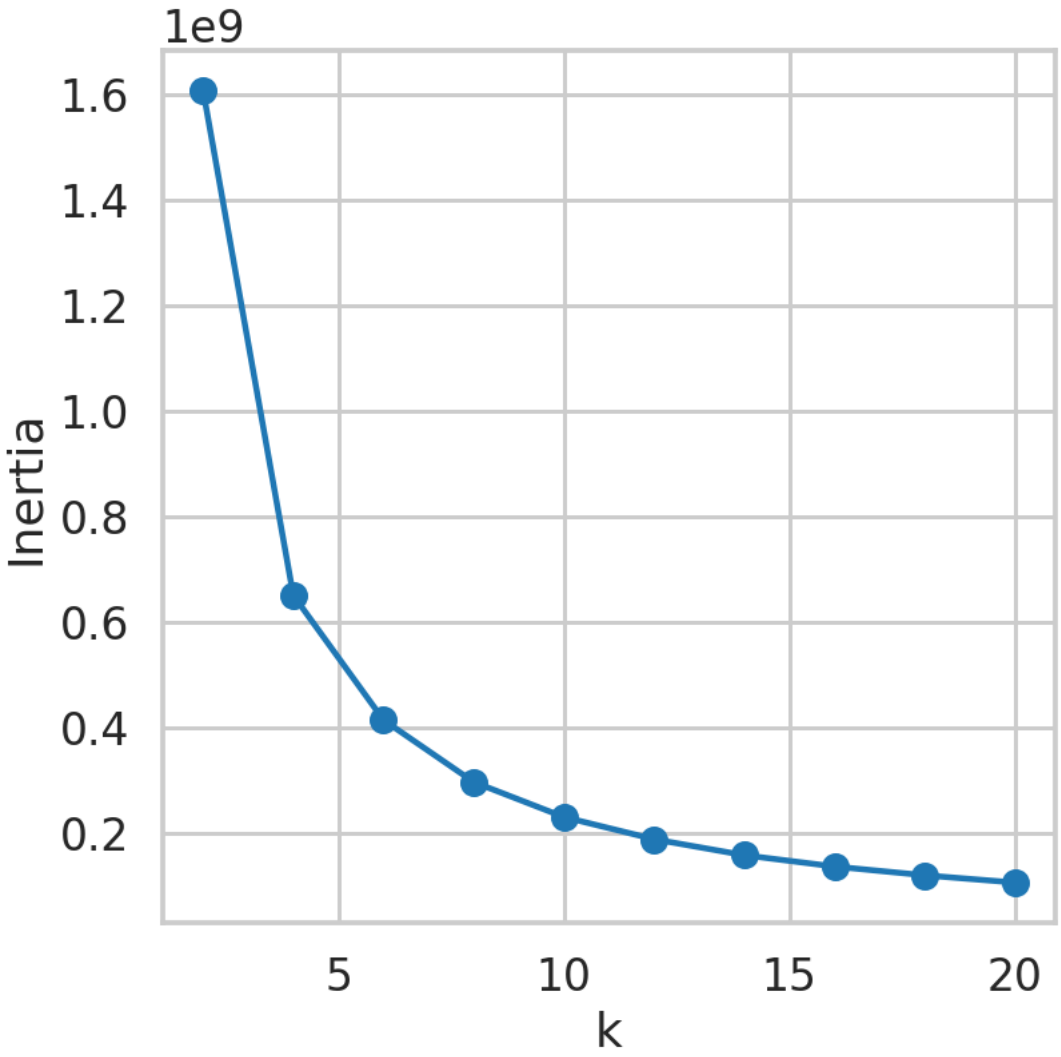
[21]:
```
### BEGIN SOLUTION
def image_cluster(img, k):
    img_flat = img.reshape(img.shape[0]*img.shape[1],3)
    kmeans = KMeans(n_clusters=k, random_state=0).fit(img_flat)
    img_flat2 = img_flat.copy()

    # loops for each cluster center
    for i in np.unique(kmeans.labels_):
        img_flat2[kmeans.labels_==i,:] = kmeans.cluster_centers_[i]

    img2 = img_flat2.reshape(img.shape)
    return img2, kmeans.inertia_
### END SOLUTION
```

1.2.5 Problem 3:

Call the function for k between 2 and 20, and draw an inertia curve. What is the optimum number of clusters?

[22]:
```
### BEGIN SOLUTION
k_vals = list(range(2,21,2))
img_list = []
inertia = []
for k in k_vals:
#    print(k)
    img2, ine = image_cluster(img,k)
    img_list.append(img2)
    inertia.append(ine)
```

[23]:
```
# Plot to find optimal number of clusters
plt.plot(k_vals,inertia)
plt.scatter(k_vals,inertia)
plt.xlabel('k')
plt.ylabel('Inertia');
### END SOLUTION
```



Sometimes, the elbow method does not yield a clear decision (for example, if the elbow is not clear and sharp, or is ambiguous). In such cases, alternatives such as the silhouette coefficient can be helpful.

1.2.6 Problem 4:

Plot in a grid all the images for the different k values.

[24]:
```
### BEGIN SOLUTION
plt.figure(figsize=[10,20])
for i in range(len(k_vals)):
    plt.subplot(5,2,i+1)
    plt.imshow(img_list[i])
    plt.title('k = '+ str(k_vals[i]))
    plt.axis('off');
```

```
### END SOLUTION
```

k = 2

k = 4

k = 6

k = 8

k = 10

k = 12

k = 14

k = 16

k = 18

k = 20

1.2.7 Machine Learning Foundation (C) 2020 IBM Corporation