Machine Learning

Lab04 - K-Means Clustering (Unsupervised Learning)

Objectives:

- Learn how to use the scikit-learn's K-Means package.
- Learn how to implement our own K-Means class.

Material adapted from https://medium.com/machine-learning-algorithms-from-scratch/k-means-clustering-from-scratch-in-python-1675d38eee42 but I added and modifed quite a bit to it.

Version: 2024-11-19

This lab is by YP Wong <yp@ypwong.net>.

Import Libraries

```
#import libraries
import numpy as np
import pandas as pd
import random as rd

import matplotlib.pyplot as plt
%matplotlib inline
```

Experiment scikit-learn's K-means Package Using Generated Blobs Data

```
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

def get_sample_blobs_data():
  features, true_labels = make_blobs(
      n_samples = 200,
      centers = 3,
      cluster_std = 2.75,
      random_state = 42
  )

  scaler = StandardScaler()
  scaled_features = scaler.fit_transform(features)
  # print(type(scaled_features))
  # print(scaled_features.shape)
  return scaled_features
```

```
X = get_sample_blobs_data()

kmeans = KMeans(
    init = "k-means++",   # another option is "random"
    n_clusters = 3,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

kmeans.fit(X)

print(kmeans.inertia_)
print(kmeans.cluster_centers_)
print(kmeans.n_iter_)
print(kmeans.labels_)
```

```
74.57960106819853
[[ 1.19539276  0.13158148]
 [-0.91941183 -1.18551732]
 [-0.25813925  1.05589975]]
6
[0 2 1 1 1 1 2 1 0 1 0 0 0 0 2 1 2 1 0 1 0 0 1 2 1 2 2 1 2 0 0 0 1 1 2
2 1
 2 1 2 0 2 1 0 1 0 0 1 0 2 1 2 1 2 2 2 1 0 2 0 1 2 1 1 1 1 2 2 1 2 2 1
2 0
 0 0 0 2 0 2 2 0 1 1 1 1 1 2 0 1 0 2 2 2 0 1 2 0 0 2 1 1 2 1 2 0 1 0 0
1 0
 0 2 1 2 1 1 2 2 2 1 0 2 1 1 0 2 2 0 2 0 1 2 1 1 0 0 0 2 0 2 2 1 0 0 2
0 1
 1 0 2 1 0 1 0 1 1 2 0 0 2 0 0 1 2 0 0 2 1 0 1 2 0 1 2 0 2 2 2 0 2 0 1
1 1
 2 0 0 0 2 2 0 1 1 2 1 2 2 0 0]
```

Find Optimum Number of Clusters Using Elbow Method

The idea is that we want a small within-cluster sums of squares (WCSS), but that the WCSS tends to decrease toward 0 as we increase k (the WCSS is 0 when k is equal to the number of data points in the dataset, because then each data point is its own cluster, and there is no error between it and the center of its cluster). So our goal is to choose a small value of k that still has a low WCSS, and the elbow usually represents where we start to have diminishing returns by increasing k.

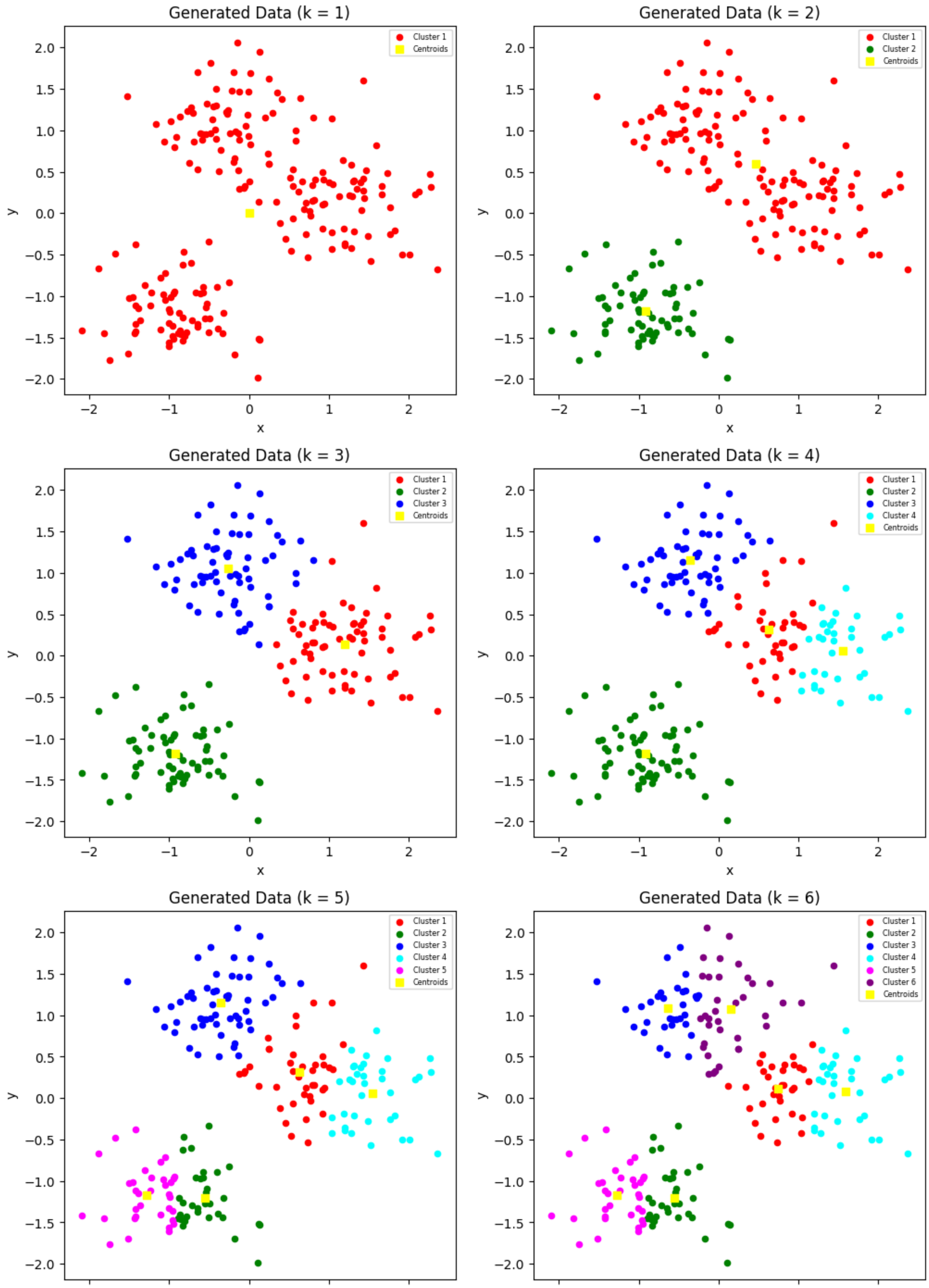
To identify the elbow point programmatically:

- kneed: https://github.com/arvkevi/kneed

```
#!pip install kneed
#from kneed import KneeLocator
#installed, no need repeat
```

```
import matplotlib.pyplot as plt
%matplotlib inline

# Visualising the clusters
default_colors = ["red", "green", "blue", "cyan", "magenta",
          "purple", "beige", "brown", "pink", "orange",
          "yellow", "gray", "black"]

def plot_inertia(WCSS_array, k_max,
                 x_label = "Number of Clusters",
                 y_label = "Within-Cluster Sums of Squares (WCSS)",
                 title = "Elbow Method to Determine Optimum Number of
Clusters"
                ):
  k = WCSS_array.size
  K_array = np.arange(1, k_max + 1, 1)
  plt.xlim(1, k_max)
  plt.plot(K_array[:k], WCSS_array)
  plt.title(title)
  plt.xlabel(x_label)
  plt.ylabel(y_label)


def get_elbow(WCSS_array, k_max):
  K_array = np.arange(1, k_max + 1, 1)
  kl = KneeLocator(K_array, WCSS_array, S = 1.0,
                   curve = "convex", direction = "decreasing")
  return kl.elbow

def plot_clusters(n_clusters, centroids, X, labels,
                  title, x_label, y_label,
                  colors = default_colors):
  for i in range(n_clusters):
    plt.scatter(X[labels == i, 0], X[labels == i, 1],
                s = 20, c = colors[i], label = f"Cluster {i+1}")

  plt.scatter(centroids[:, 0], centroids[:, 1],
                s = 40, c = 'yellow', marker = 's', label =
'Centroids')

  plt.title(title)
  plt.xlabel(x_label)
  plt.ylabel(y_label)
  plt.legend(fontsize = "xx-small")

def get_inertia_list(X, k_min, k_max, KMeans_model,
                     title, x_label, y_label,
                     visualize = False):

  # WCSS_array = np.array([])   # also works
```

```
  WCSS_array = []   # Within-Cluster Sum of Square i.e. Inertia

  if visualize:
    figure = plt.figure(figsize=(12, 30))

  for k in range(k_min, k_max + 1):

    kmeans = KMeans_model(
        init = "k-means++",   # another option is "random"
        n_clusters = k,
        n_init = "auto",
        max_iter = 100,
        random_state = 42
    )

    kmeans.fit(X)
    labels = kmeans.predict(X)
    WCSS_array = np.append(WCSS_array, kmeans.inertia_)

    if visualize:
      figure.add_subplot((k_max - 1) // 2 + 1, 2, k - k_min + 1)
      plot_clusters(k, kmeans.cluster_centers_, X, labels,
                    title + f" (k = {k})",
                    x_label, y_label,
                    colors = default_colors)

  return WCSS_array

WCSS_array = get_inertia_list(X, k_min = 1, k_max = 10, KMeans_model =
KMeans,
                              title = "Generated Data",
                              x_label = "x", y_label = "y",
                              visualize = True)
```


Generated Data (k = 1)
Cluster 1
Centroids
Generated Data (k = 2)
Cluster 1
Cluster 2
Centroids
Generated Data (k = 3)
Cluster 1
Cluster 2
Cluster 3
Centroids
Generated Data (k = 4)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Centroids
Generated Data (k = 5)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Centroids
Generated Data (k = 6)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
Centroids
x
y

```
plot_inertia(WCSS_array, k_max = 10)
plt.show()

n_clusters = get_elbow(WCSS_array, k_max = 10)
print("n_clusters = elbow =", n_clusters)

kmeans = KMeans(
    init = "k-means++",   # another option is "random"
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

kmeans.fit(X)
labels = kmeans.predict(X)

plot_clusters(n_clusters, kmeans.cluster_centers_, X, labels,
              title = "Generated Data",
              x_label = "x", y_label = "y",
              colors = default_colors)
plt.show()
```

Test the Model

```
sample_test = np.array([[1, 0.5], [-1, -1.5]])
print(sample_test)
print(sample_test.shape)

labels = kmeans.predict(sample_test)
print(labels)
```

Experiment scikit-learn's K-means package Using Real-World Data

Read the Data

Data source: https://www.superdatascience.com/

```
# Uncomment the below if you need to read data from your Google Drive
# Change the notebook_path to where you run the Jupyter Notebook from.

from google.colab import drive
import os

drive.mount('/content/drive')

notebook_path =
r"/content/drive/MyDrive/Classroom/_ML2425T3(2430)/__ML2425T3(2430)_SH
```

```
ARED__/Labs/Lab04_KMeans"
os.chdir(notebook_path)
!pwd

import pandas as pd

dataset = pd.read_csv('Mall_Customers.csv')

print(type(dataset))
print(dataset.shape)

dataset.describe()

X = dataset.iloc[:, [3, 4]].values  # all rows, column 3 and 4
print(type(X))
print(X.shape)

m = X.shape[0]  # number of training examples (number of rows)
n = X.shape[1]  # number of features (number of columns)

print(m)
print(n)

n_clusters = 5    # number of clusters
```

Find Optimum Number of Clusters Using Elbow Method

```
WCSS_array = get_inertia_list(X, k_min = 1, k_max = 10, KMeans_model =
KMeans,
                              title = "Clusters of customers",
                              x_label = "Annual Income (k$)",
                              y_label = "Spending Score (1-100)",
                              visualize = True)

plot_inertia(WCSS_array, k_max = 10)
plt.show()

n_clusters = get_elbow(WCSS_array, k_max = 10)
print("n_clusters = elbow =", n_clusters)

kmeans = KMeans(
    init = "k-means++",   # another option is "random"
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

kmeans.fit(X)
labels = kmeans.predict(X)
```

```
plot_clusters(n_clusters, kmeans.cluster_centers_, X, labels,
              title = "Clusters of customers",
              x_label = "Annual Income (k$)",
              y_label = "Spending Score (1-100)",
              colors = default_colors)
plt.show()
```

Test the Model

```
sample_test = np.array([[70, 60], [100, 60]])
print(sample_test)
print(sample_test.shape)

labels = kmeans.predict(sample_test)
print(labels)
```

Implement Our Own K-means Class From Scratch

```
import numpy as np
import random as rd

class MyKMeans:

  def __init__(self,
               n_clusters = 8,
               max_iter = 300,
               init = "ramdom",
               tol = 1e-4,
               random_state = None,
               n_init = "auto"):

    self.n_clusters = n_clusters
    self.max_iter = max_iter
    self.init = init
    self.tol = tol
    self.random_state = random_state
    if n_init == "auto":
      self.n_init = 1
    else:
      self.n_init = n_init

  # randomly initialize the centroids
  def __init_centroids_random(X, n_clusters, random_state = None):

    if random_state != None:
      rd.seed(random_state)

    n_points = X.shape[0]

    # centroids = np.array([]).reshape(0, 2)   # also works
```

```
    centroids = np.empty( (0, 2), dtype = float )
    for i in range(n_clusters):
      rand = rd.randint(0, n_points - 1)
      centroids = np.append(centroids, [X[rand]], axis = 0)  # add
another centroid

    return centroids

  # initialize the centroids using K-Means++ method
  def __init_centroids_kmeanspp(X, n_clusters, random_state = None):

    if random_state != None:
      rd.seed(random_state)

    n_points = X.shape[0]

    rand = rd.randint(0, n_points - 1)
    centroids = np.array([X[rand]])  # start with 1 random centroid

    # start with 1 centroid,
    #   each iteration adds one more centroid to list centroids
    for k in range(1, n_clusters):

      # dists = np.array([])   # also works
      dists = []
      # For each point, compute the distance to
      #   the nearest centroid amoung the k centroids so far
      for x in [x for x in X if x not in centroids]:
        dist_to_centroids = np.sum((x - centroids) ** 2)
        dists = np.append(dists, np.min(dist_to_centroids))

      probs = dists / np.sum(dists)
      cummulative_probs = np.cumsum(probs)

      # Randomly select a point as centroid
      #   with probability proportion to the distance
      #   of that point to the nearest so-far-selected centroid.
      #   Meaning, we want to select the next centroid
      #   to be as far as possible to the rest of the selected
centroids
      rand = rd.random()
      i = 0
      for j, prob in enumerate(cummulative_probs):
        if prob >= rand:
          i = j
          break

      centroids = np.append(centroids, [X[i]], axis = 0)  # add
another centroid
```

```
    return centroids

  def init_centroids(X, n_clusters, random_state, init = "random"):
    if init == "k-means++":
      centroids = MyKMeans.__init_centroids_kmeanspp(X, n_clusters,
random_state)
    else:
      centroids = MyKMeans.__init_centroids_random(X, n_clusters,
random_state)

    return centroids

  def assign_labels(X, centroids):

    n_points = X.shape[0]
    n_clusters = centroids.shape[0]

    dists = np.zeros( (n_points, n_clusters) )

    for k in range(n_clusters):
      centroid = centroids[k, :]
      dists[:, k] = np.sum( (X - centroid)**2, axis = 1 )
    labels = np.argmin(dists, axis = 1)

    return labels

  def initialize(self, X):

    self.cluster_centers_ = MyKMeans.init_centroids(X,
                                                    self.n_clusters,
                                                    self.random_state,
                                                    self.init)

    self.labels_ = MyKMeans.assign_labels(X, self.cluster_centers_)

    return self

  def recompute_centroids(self, X):

    n_points = X.shape[0]

    # Adjust the centroids

    # For k = 1, ..., n_clusters,
    #   initialize empty list Y[k] for cluster k,
    #   to store points for cluster k
    Y = {}
    for k in range(self.n_clusters):
      # Y[k] = np.array([]).reshape(0, X.shape[1])   # also works
      Y[k] = np.empty( (0, X.shape[1]), dtype = float )
```

```
    # For each point, if the label is k then,
    #   add the point to the list of points Y[k]
    for i in range(n_points):
      k = self.labels_[i]
      Y[k] = np.append(Y[k], X[i].reshape(1,-1), axis = 0)

    # for k in range(self.n_clusters):
    #   print(f"size Y{k} =" + str(Y[k].size))

    # Compute the new centroid for each cluster
    for k in range(self.n_clusters):
      self.cluster_centers_[k, :] = np.mean(Y[k], axis = 0)

    # Within-Cluster Sum of Square
    wcss = 0
    for k in range(self.n_clusters):
      wcss += np.sum((Y[k] - self.cluster_centers_[k, :]) ** 2)
    self.inertia_ = wcss

    self.labels_ = MyKMeans.assign_labels(X, self.cluster_centers_)

    return self

  def fit(self, X, visualize = False):

    self.initialize(X)

    self.n_iter_ = 0

    if visualize == True:

      plot_clusters(self.n_clusters, self.cluster_centers_, X,
self.labels_,
                    title = f"k = {self.n_clusters} (Iteration =
{self.n_iter_})",
                    x_label = "x",
                    y_label = "y",
                    colors = default_colors)
      plt.show()

    # Compute euclidian distances and assign clusters
    for n in range(self.max_iter):

      centroids_previous = np.copy(self.cluster_centers_)

      self.recompute_centroids(X)

      self.n_iter_ += 1

      if visualize == True:
        plot_clusters(self.n_clusters, self.cluster_centers_, X,
```

```
self.labels_,
                      title = f"k = {self.n_clusters} (Iteration =
{self.n_iter_})",
                      x_label = "x",
                      y_label = "y",
                      colors = default_colors)
        plt.show()

      diff_sq = (self.cluster_centers_ - centroids_previous) ** 2
      diff = np.sqrt(np.sum( np.sum(diff_sq, axis = 1) ))
      if diff < self.tol:
        break

    return self

  def predict(self, X):
    return MyKMeans.assign_labels(X, self.cluster_centers_)

  def fit_predict(self, X):
    self.fit(X)
    return MyKMeans.assign_labels(X, self.cluster_centers_)
```

Test Our Own K-means Class Using Generated Blobs Data

```
import matplotlib.pyplot as plt
%matplotlib inline

X = get_sample_blobs_data()

n_clusters = 3

mykmeans = MyKMeans(
    init = "k-means++",   # another option is "random"
    # init = "random",
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

mykmeans.initialize(X)

n_iter = 5

for i in range(n_iter):
  print(mykmeans.cluster_centers_.shape)
  print(mykmeans.cluster_centers_)
  print(mykmeans.labels_)

  plot_clusters(n_clusters, mykmeans.cluster_centers_, X,
```

```
mykmeans.labels_,
                title = "Generated Data",
                x_label = "x",
                y_label = "y",
                colors = default_colors)

  mykmeans.recompute_centroids(X)

  plt.show()

n_clusters = 3

mykmeans = MyKMeans(
    init = "k-means++",   # another option is "random"
    # init = "random",
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

mykmeans.fit(X, visualize = True)
print("Iteration =", mykmeans.n_iter_)

print(mykmeans.inertia_)
print(mykmeans.cluster_centers_)
print(mykmeans.n_iter_)
print(mykmeans.labels_)

WCSS_array = get_inertia_list(X, k_min = 1, k_max = 10, KMeans_model =
MyKMeans,
                              title = "",
                              x_label = "x",
                              y_label = "y",
                              visualize = True)

plot_inertia(WCSS_array, k_max = 10)
plt.show()

n_clusters = get_elbow(WCSS_array, k_max = 10)
print("n_clusters = elbow =", n_clusters)

mykmeans = MyKMeans(
    init = "k-means++",   # another option is "random"
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

mykmeans.fit(X)
```

```
labels = mykmeans.predict(X)
print("Iteration =", mykmeans.n_iter_)

plot_clusters(n_clusters, mykmeans.cluster_centers_, X, labels,
              title = "Generated Data",
              x_label = "x",
              y_label = "y",
              colors = default_colors)
plt.show()
```

Test the Model

```
sample_test = np.array([[1, 0.5], [-1, -1.5]])
print(sample_test)
print(sample_test.shape)

labels = mykmeans.predict(sample_test)
print(labels)
```

Test Our Own K-means Class Using Real-World Data

```
dataset = pd.read_csv('Mall_Customers.csv')

X = dataset.iloc[:, [3, 4]].values  # all rows, column 3 and 4
print(type(X))
print(X.shape)

n_clusters = 5

mykmeans = MyKMeans(
    init = "k-means++",   # another option is "random"
    # init = "random",
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

mykmeans.fit(X, visualize = True)
print("Iteration =", mykmeans.n_iter_)

print(mykmeans.inertia_)
print(mykmeans.cluster_centers_)
print(mykmeans.n_iter_)
print(mykmeans.labels_)

WCSS_array = get_inertia_list(X, k_min = 1, k_max = 10, KMeans_model =
MyKMeans,
                              title = "Customers",
                              x_label = "Annual Income (k$)",
```

```
                              y_label = "Spending Score (1-100)",
                              visualize = True)

plot_inertia(WCSS_array, k_max = 10)
plt.show()

n_clusters = get_elbow(WCSS_array, k_max = 10)
print("n_clusters = elbow =", n_clusters)

mykmeans = MyKMeans(
    init = "k-means++",   # another option is "random"
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

mykmeans.fit(X)
labels = mykmeans.predict(X)

plot_clusters(n_clusters, mykmeans.cluster_centers_, X, labels,
              title = "Clusters of customers",
              x_label = "Annual Income (k$)",
              y_label = "Spending Score (1-100)",
              colors = default_colors)
plt.show()
```

Test the Model

```
sample_test = np.array([[70, 60], [100, 60]])
print(sample_test)
print(sample_test.shape)

labels = mykmeans.predict(sample_test)
print(labels)
```

END.

Some Related Experiments - Coding Strategies of Assigning Labels Based on Minimal Distance

```
import pandas as pd
import numpy as np
import random as rd
from sklearn.cluster import KMeans

dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values  # all rows, column 3 and 4
```

```
print(X.shape)

n_clusters = 5

kmeans = KMeans(
    init="k-means++",   # another option is "random"
    n_clusters = n_clusters,
    n_init = "auto",
    max_iter = 100,
    random_state = 42
)

kmeans.fit(X)
centroids = kmeans.cluster_centers_
```

Not-So-Efficient Way - Nested Loops Iterate Rows

- The algorithm is the easiest to understand.
- No `numpy` broadcasting used.
- No `argmin` function used.

```
def assign_labels_nested_loops_iterate_rows(X, centroids):
  # print(X.shape)
  # print(centroids.shape)

  m = X.shape[0]
  n = X.shape[1]
  n_clusters = centroids.shape[0]

  C = np.zeros(m, dtype = int)

  for row in range(m):

    point = X[row, :]

    for k in range(n_clusters):

      centroid = centroids[k, :]

      # dist = np.sum( (point - centroid)**2 )  # slower
      dist = 0
      for j in range(n):
        dist += (point[j] - centroid[j])**2

      if (k == 0):
        dist_smallest = dist
        dist_smallest_index = 0
      else:
        if (dist < dist_smallest):
          dist_smallest = dist
          dist_smallest_index = k
```

```
    C[row] = dist_smallest_index + 1

  return C

C = assign_labels_nested_loops_iterate_rows(X, centroids)
print(C.shape)
print(C)
```

Not-So-Efficient Way - Nested Loops Iterate Centroids

- `argmin` function used once.

```
def assign_labels_nested_loops_iterate_centroids(X, centroids):
  # print(X.shape)
  # print(centroids.shape)

  n_points = X.shape[0]
  n_clusters = centroids.shape[0]

  dists = np.zeros( (n_points, n_clusters) )

  for k in range(n_clusters):

    centroid = centroids[k, :]

    for row in range(n_points):

      point = X[row, :]

      dist = 0
      for j in range(X.shape[1]):
        dist += (point[j] - centroid[j])**2

      dists[row, k] = dist

  # print(dists.shape)
  # print(dists)

  C = np.argmin(dists, axis = 1) + 1

  return C

C = assign_labels_nested_loops_iterate_centroids(X, centroids)
print(C.shape)
print(C)
```

Not-So-Efficient Way - One Loop Iterate Rows

- `argmin` function used `m` times.
- Slowest.

```
def assign_labels_one_loop_iterate_rows(X, centroids):
  # print(X.shape)
  # print(centroids.shape)

  n_points = X.shape[0]
  C = np.zeros(n_points, dtype = int)

  for row in range(n_points):

    point = X[row, :]

    dists = np.sum( (point - centroids)**2, axis = 1 )    # broadcast
used
    C[row] = np.argmin(dists) + 1

  return C

C = assign_labels_one_loop_iterate_rows(X, centroids)
print(C.shape)
print(C)
```

Most Efficient Way - One Loop Iterate Centroids

- `argmin` function used once.
- Fastest as it is most Pythonic way.

```
def assign_labels_one_loop_iterate_centroids(X, centroids):
  # print(X.shape)
  # print(centroids.shape)

  n_points = X.shape[0]
  n_clusters = centroids.shape[0]

  dists = np.zeros( (n_points, n_clusters) )

  for k in range(n_clusters):

    centroid = centroids[k, :]

    # dist = np.sum( (X - centroid)**2, axis = 1 )    # slower
    # dists = np.c_[dists, dist]
    dists[:, k] = np.sum( (X - centroid)**2, axis = 1 )

  # print(dists.shape)
  # print(dists)

  C = np.argmin(dists, axis = 1) + 1

  return C

C = assign_labels_one_loop_iterate_centroids(X, centroids)
```

```
print(C.shape)
print(C)
```

Performance of Coding Methods

```
# Source: https://stackoverflow.com/questions/37743843/python-why-use-numpy-r-instead-of-concatenate
!pip install perfplot

import numpy as np
import perfplot

b = perfplot.bench(
    setup = lambda n: (X, centroids),
    kernels = [
        assign_labels_nested_loops_iterate_rows,
        assign_labels_nested_loops_iterate_centroids,
        assign_labels_one_loop_iterate_rows,
        assign_labels_one_loop_iterate_centroids
    ],
    labels = ["nested_loops_iterate_rows",
              "nested_loops_iterate_centroids",
              "one_loop_iterate_rows",
              "one_loop_iterate_centroids"
              ],
    n_range = range(10),
    xlabel = "itr",
)
b.save("out.png")
b.show()
```

Tips: Randomly select a number with probability proportion to the number

```
np.random.seed(0)

def random_select(nums):
  probs = nums / np.sum(nums)
  cummulative_probs = np.cumsum(probs)
  rand = rd.random()
  i = 0
  for j, prob in enumerate(cummulative_probs):
    if prob >= rand:
      i = j
      break
  return i, nums[i]

nums = [10, 20, 5, 40, 25]
```

```
counts = np.zeros(len(nums))

round = 10000
for i in range(round):
  index, num = random_select(nums)
  counts[index] += 1

for i in range(len(nums)):
  print(counts[i] / np.sum(counts))
```

Tips: Append arrays of shape (1, 2) to an empty array

```
import numpy as np

array1 = np.empty((0, 2))   # start with empty array

array2 = np.array([[1, 2]])
array3 = np.array([[3, 4]])
array4 = np.array([[5, 6]])

array1 = np.append(array1, array2, axis = 0)
array1 = np.append(array1, array3, axis = 0)
array1 = np.append(array1, array4, axis = 0)

print(array1, array1.shape)
```

Tips: Filter one array with another array

```
import numpy as np
Z = np.array([ [ 1,  2,  3],
               [ 4,  5,  6],
               [ 7,  8,  9],
               [10, 11, 12],
               [13, 14, 15] ])

labels = np.array([0, 1, 1, 0, 1])

print(labels == 0)
print(Z[labels == 0])
print(Z[labels == 0, 0])
print(Z[labels == 0, 1])

print(labels == 1)
print(Z[labels == 1])
print(Z[labels == 1, 0])
print(Z[labels == 1, 1])
```

Tips: Class as a input to a function

```
class TestClass1:

  def __init__(self, data):
    self.data = data

  def test(self, data):
    self.data += data

  def show(self):
    print("data class TestClass1 =", self.data)

class TestClass2:

  def __init__(self, data):
    self.data = data

  def test(self, data):
    self.data -= data

  def show(self):
    print("data class TestClass2 =", self.data)

def test_fun(MyTestClass):
  c = MyTestClass(456)
  c.test(321)
  c.show()

test_fun(TestClass1)

test_fun(TestClass2)
```

Tips: Choose a number x from list X but not in list Y

This is used in `MyKMeans.__init_centroids_kmeanspp()` function

```
X = [2, 5, 27, 13, 86, 18, 22, 43]
Y = [2, 13, 18]

for x in [x for x in X if x not in Y]:
  print(x)
```