**Machine Learning fundamentals Homework 2 : Hard/Soft *k*-means**

20213359 LEE HYO JEONG

**CODE:**

https://colab.research.google.com/drive/1-XxGVuR4AWqrHomhLjEXb0MK1OOMxTxf?usp=sharing

*\*Full code is also available at the end of this document*

**1. Artificial datasets.**

**A. Draw an artificial dataset consisting of two features and two clusters under the following constraints and explain what each constraint means.**

**A-1. Draw an artificial dataset**

**i, ii. “Balanced” and “spherical” & “Imbalanced” and “spherical”**

Description (Sampled from gaussian distribution)

Fig 1 : number of data = 200, centre=(0, 0),(2, 2), radius=3, gaussian noise=0.1

Fig 2 : number of data = 200, 1000, centre=(0, 0),(2, 2), radius=3 gaussian noise =0.1

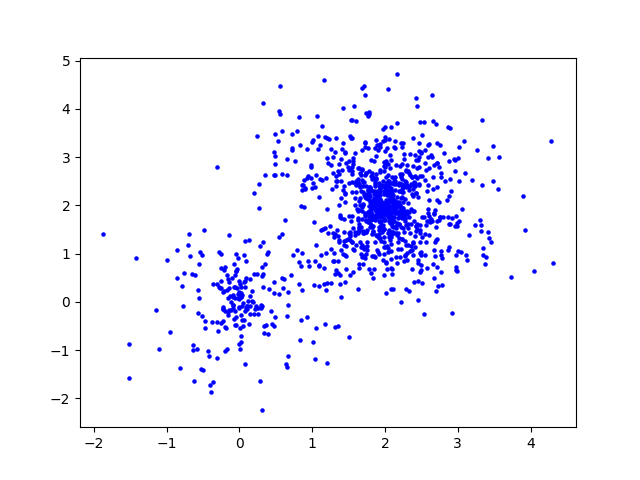
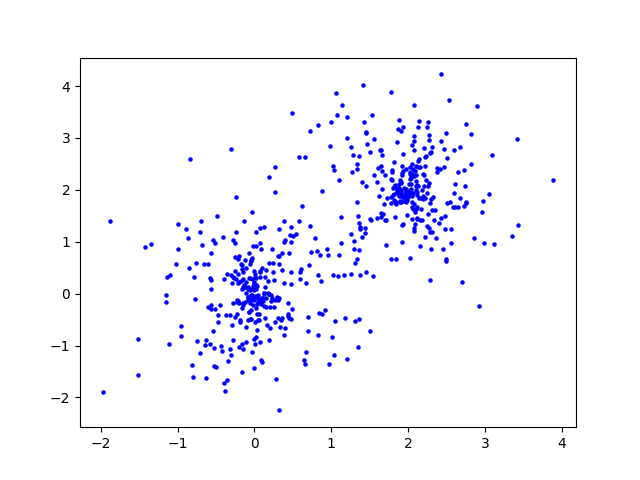


Fig 2. Imbalanced, spherical

Fig 1. Balanced, spherical

텍스트이(가) 표시된 사진

자동 생성된 설명

Fig 3. Spherical dataset generating code snippet

**iii, iv “Balanced” and “non-spherical” & “Imbalanced” and “non-spherical variance”**

Description

Fig 4 : number of data=200, centre=(3, 5),(2, 1), w=4, 1, noise=0.5, range=4

Fig 5 : number of data=100,300 centre=(0, 0), r=4, 1 noise=0.2

Fig 6 : number of data = 1000, 3000, mean=[2, 3, 1], [11, 15, 12],

variance=[1, 1.5, 0.8], [1.7, 1.5, 2], weights=[0.3, 0.4, 0.3], [0.3, 0.1, 0.6]

]

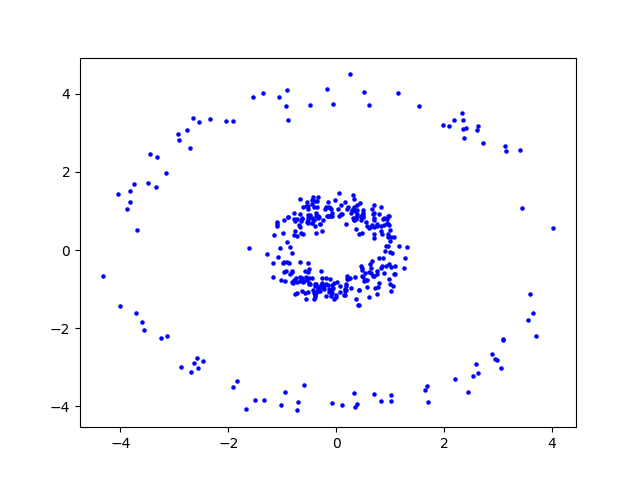
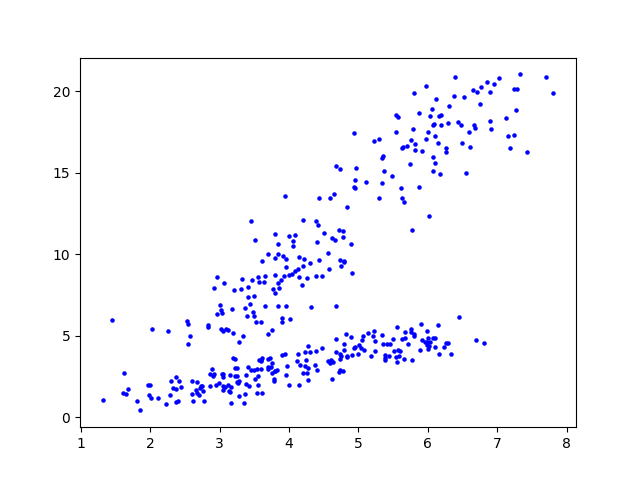


Fig 5. Imbalanced, non-spherical 2

Fig 4. Balanced, non-spherical

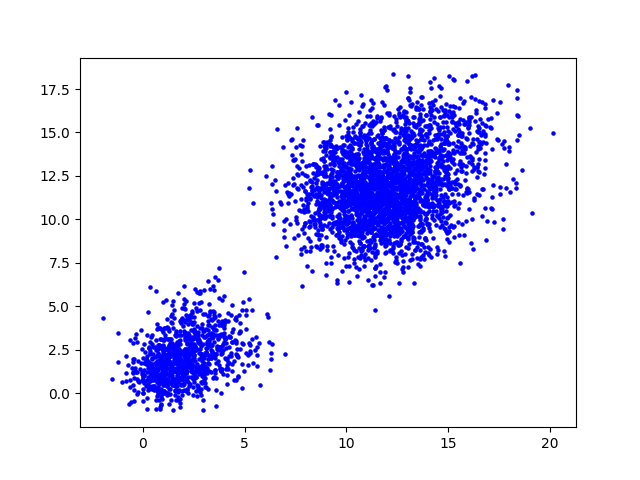


Fig 6. Imbalanced, non-spherical variance

텍스트이(가) 표시된 사진

자동 생성된 설명

Fig 6. Non-spherical dataset generating code snippet

텍스트이(가) 표시된 사진

자동 생성된 설명

Fig 7. Non-spherical dataset generating code snippet 2

텍스트이(가) 표시된 사진

자동 생성된 설명

Fig 8. Non-spherical variance dataset generating code snippet (using mixed gaussian)

**A-2. Explain what each constrain means.**

**i)** **Spherical**: Loosely speaking, we can say some data is spherical when the data has sphere-like shape when you plot it. But more exact explanation of spherical data could be found in definition of isotropy. General definition of Isotropy is uniformity in all orientations. And In the fields of machine learning, a probability distribution over vectors is said to be in isotropic position if its covariance matrix is proportionate to the identity matrix. That means, if we rotate the coordinate system with an orthogonal rotation matrix, covariance matrix will stay the same. This is why isotropy and spherical is related in meaning. Rotating sphere doesn’t change the sphere’s shape.

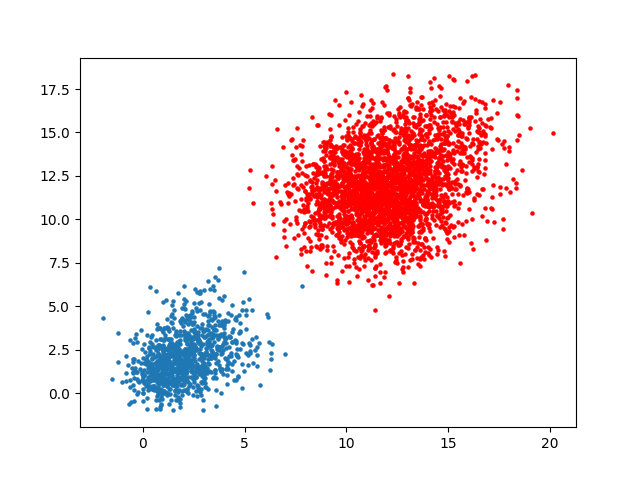
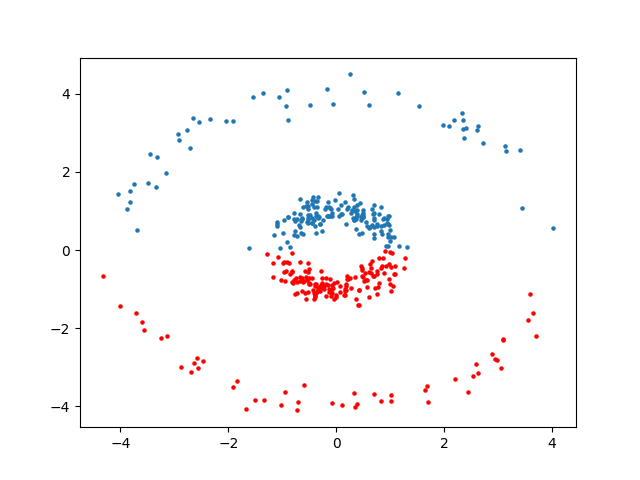
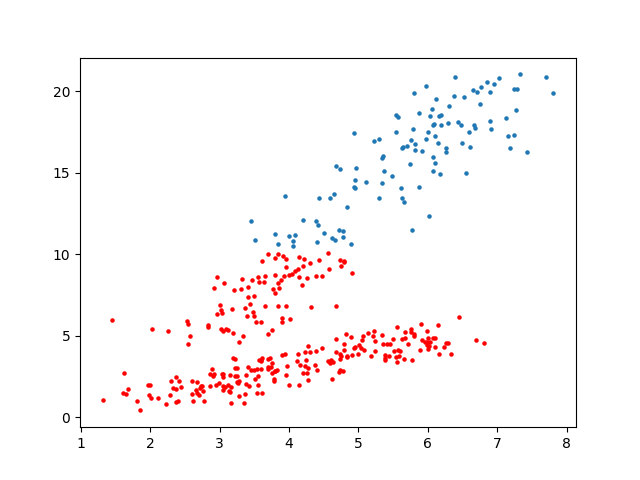
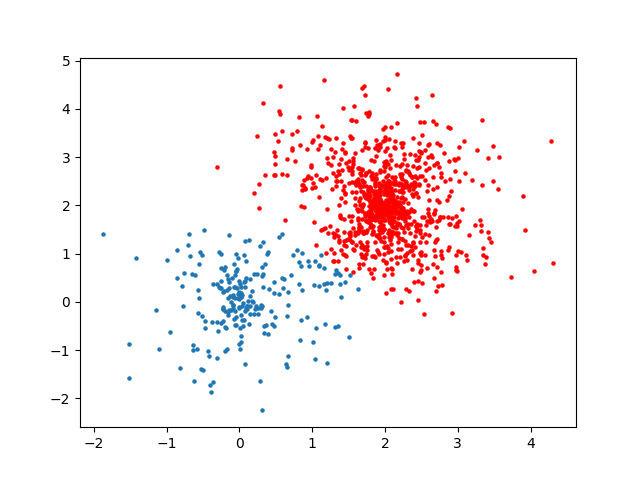
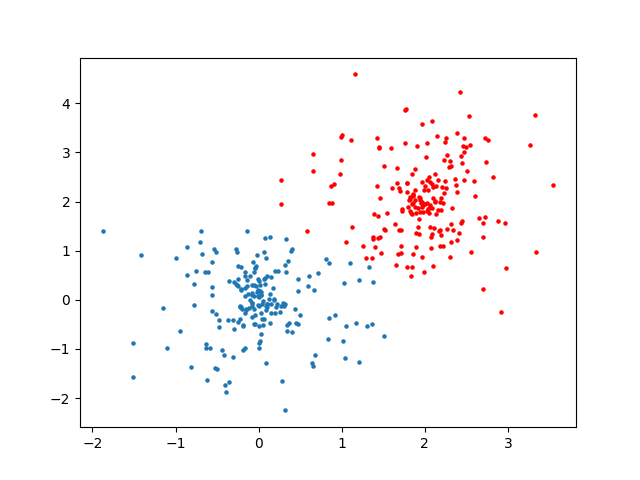
**ii) none-spherical**: Example of none-spherical shapes could be donut-shape, spiral and curves. K-means clustering may not perform well on these types of datasets because K-means assume the variance of the distribution of given data is spherical.

**iii) non-spherical variance**: spherical or Isotropic variance is refer to covariance matrices that has same standard deviation as diagonal elements. But non-spherical variance is not. One example of data that has none-spherical variance would be data sampled from mixed gaussian distribution. Since it is made by more then two different gaussian distribution, its variance is not fixed as one value which lead to be non-spherical.

**iv) imbalance**: We say some data is imbalanced when it’s number of observations per class is not equally distributed. In general machine learning fields, it can lead model to fail to correctly infer on classes that has few samples. In the case of multi-label classification problem, dataset tends to be very sparse and can lead the model to learn from false/negative samples, not positive ones. K-means clustering often fail on imbalanced data but its cause roots from limit of its kernel function, calculating the mean. K-means algorithm assigns datapoints to the closest centroid and moving the centroid to the mean of the updated corresponding datapoints. So, the centroids tends to moving towards where datapoints are denser which is the majority class; K-means clustering fails to represent the class that has fewer samples.

**B. Run your own hard/soft k-means clustering algorithm on artificial datasets you made.**

**B-a Hard k-means**



def K\_means(data, k):

assinged = np.zeros(data.shape[0])

cluster\_means = np.zeros((k, 2))

cluster\_datapoint\_num = np.zeros((k)) - 1

converge = 0

'''init the centroids'''

for i in range(0,k):

cluster\_means[i][0] = np.random.normal()

cluster\_means[i][1] = np.random.normal()\*2

while converge != k:

for i in range(0, data.shape[0]):

print(data.shape)

centroids = np.array([-1, np.amax(data) \* 1000000])

for j in range(0, k):

mean = np.sqrt((data[i][0]-cluster\_means[j][0])\*\*2 + (data[i][1]-cluster\_means[j][1])\*\*2)

if mean < centroids[1]:

centroids[0] = j

centroids[1] = mean

assinged[i] = centroids[0]

for i in range(0, k):

tmp = cluster\_datapoint\_num[i]

ith\_cluster\_idx = np.where(assinged == i)

cluster\_datapoint\_num[i] = len(ith\_cluster\_idx[0])

if tmp == cluster\_datapoint\_num[i]:

converge += 1

x\_sum = 0

y\_sum = 0

for j in ith\_cluster\_idx[0]:

x\_sum += data[j][0]

y\_sum += data[j][1]

x\_sum /= cluster\_datapoint\_num[i]

y\_sum /= cluster\_datapoint\_num[i]

cluster\_means[i][0] = x\_sum

cluster\_means[i][1] = y\_sum

if converge == k:

return assinged

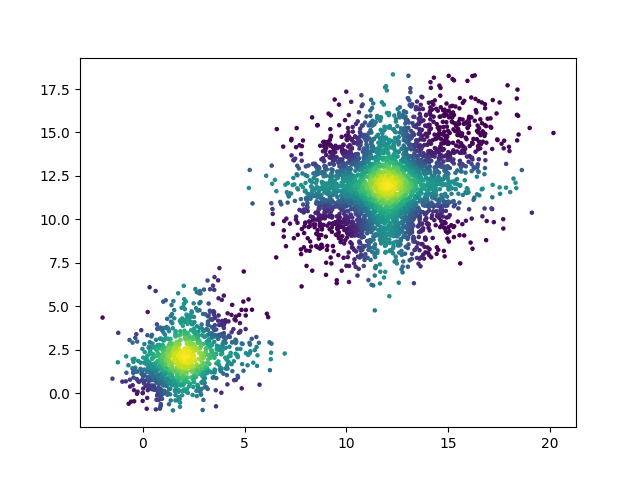
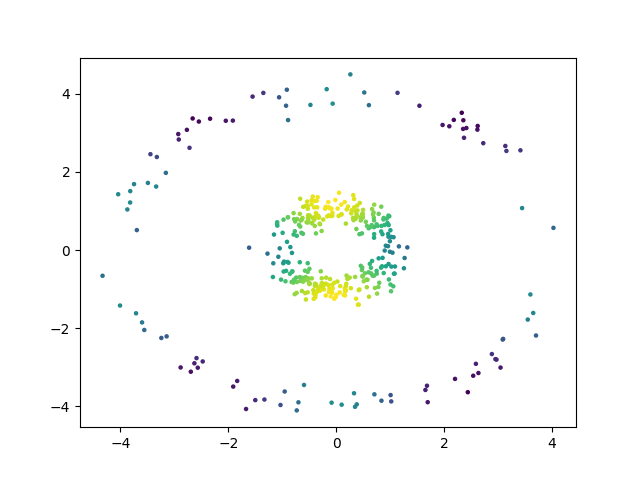
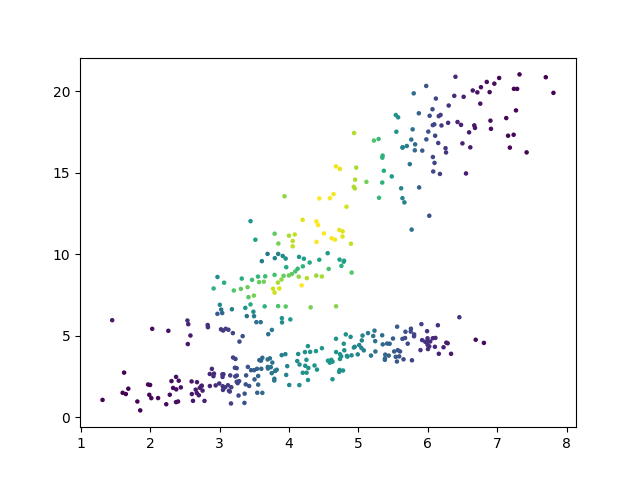
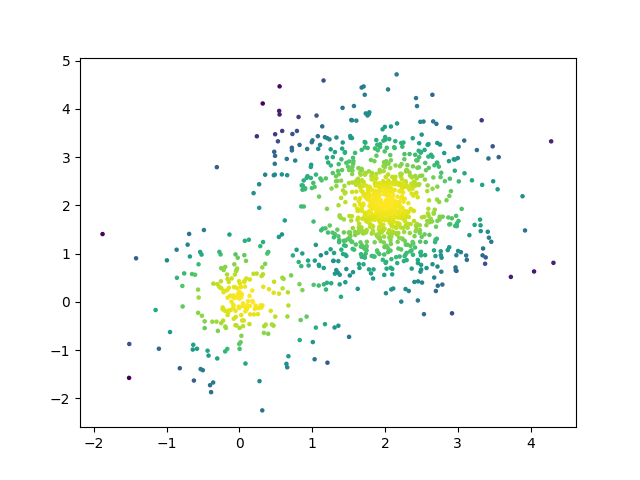
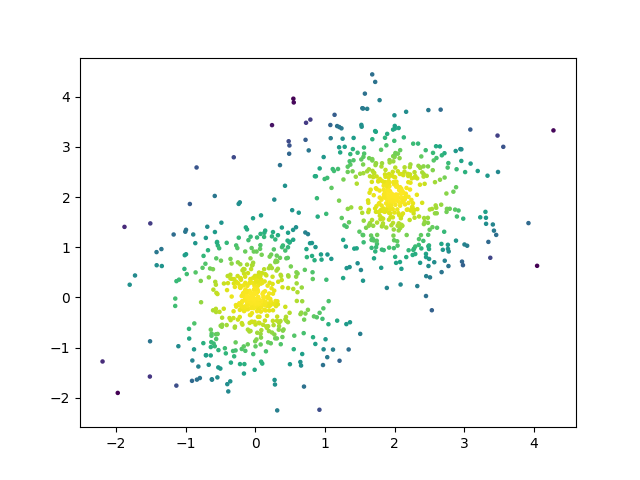
else:

converge = 0

**B-a.2 C. Describe your findings from the results of B in relation to k-means assumptions.**

In the results of hard k-means, I pre-assumed that second(Imbalanced spherical) and last(Imbalanced non-spherical variance) will not be that good. I found that hard k-means has robustness at a certain level even though it is simple algorithm. And even for the imbalanced and non-spherical variance cases, If the mean is distinguishable enough, results can be good.

**B-b Soft k-means**

****

**B-b.2 C. Describe your findings from the results of B in relation to k-means assumptions.**

Notable result can be found in 4th result. In the case of using hard k-means algorithm which we cannot figure out the possibility score, clustering fails. But using soft k-means, I think clustering from the donut-shaped data can be done much more nicely then hard k-means if we choose the right threshold of responsibility scores and separate the data again.

def K\_means\_soft(data, k):

assinged = np.zeros(data.shape[0])

cluster\_means = np.zeros((k, data.shape[1]))

cluster\_datapoint\_num = np.zeros((k)) - 1

responsibility = np.zeros((data.shape[0], data.shape[1]))

converge = 0

beta = 0.6

check = 0

'''init the centroids'''

for i in range(0,k):

cluster\_means[i][0] = np.random.normal()

cluster\_means[i][1] = np.random.normal()

while converge != k:

check += 1

for i in range(0, data.shape[0]):

centroids = np.array([-1, np.amax(data) \* 1000000])

responsibility\_tmp = np.zeros((k, data.shape[1]))

for j in range(0, k):

mean = np.sqrt((data[i][0]-cluster\_means[j][0])\*\*2 + (data[i][1]-cluster\_means[j][1])\*\*2)

responsibility\_tmp[j] = np.exp(-beta \* ((data[i]-cluster\_means[j])\*\*2))

# print(i, j, k)

# print(cluster\_means[j])

if mean < centroids[1]:

centroids[0] = j

centroids[1] = mean

assinged[i] = centroids[0]

assigned\_centroid = assinged[i]

#

responsibility[i] = responsibility\_tmp[int(assigned\_centroid)]

for i in range(0, k):

print('what')

tmp = cluster\_datapoint\_num[i]

ith\_cluster\_idx = np.where(assinged == i)

cluster\_datapoint\_num[i] = len(ith\_cluster\_idx[0])

if tmp == cluster\_datapoint\_num[i]:

converge += 1

x\_sum = 0

y\_sum = 0

for j in ith\_cluster\_idx[0]:

x\_sum += data[j][0]

y\_sum += data[j][1]

x\_sum /= cluster\_datapoint\_num[i]

y\_sum /= cluster\_datapoint\_num[i]

print(cluster\_datapoint\_num[i])

print(x\_sum, y\_sum)

cluster\_means[i][0] = x\_sum

cluster\_means[i][1] = y\_sum

if converge == k:

return assinged, responsibility

else:

converge = 0

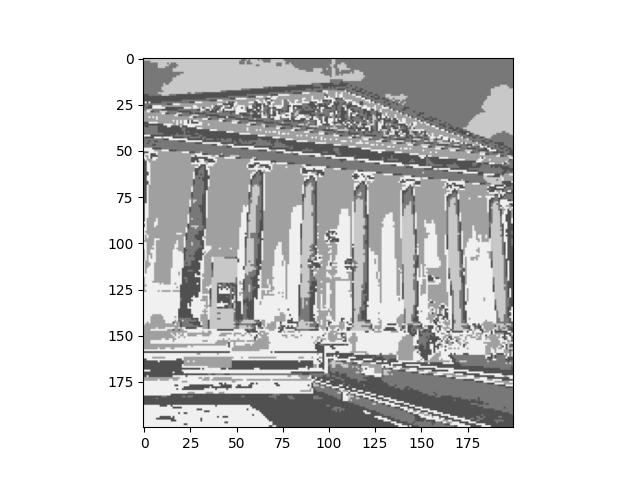
**2. Real datasets. (image segmentation)**

**1. Hard K-means : k = 5**

텍스트이(가) 표시된 사진

자동 생성된 설명하늘, 실외이(가) 표시된 사진

자동 생성된 설명

하늘, 건물, 실외, 청사이(가) 표시된 사진

자동 생성된 설명

**Full code**

import numpy as np

import matplotlib.pyplot as plt

import cv2

np.random.seed(10)

class Spherical\_shape\_like():

    def \_\_init\_\_(self, feature\_num=2, data\_num=300, centre=(0, 0), r=(2, 4), noise=0.01, fill=False):

        def dumy\_dataset(feature\_num=None):

            return np.zeros((1, feature\_num))

        self.fill = fill

        self.noise = noise

        self.feature\_num = feature\_num

        self.data\_num = data\_num

        self.dataset = dumy\_dataset(feature\_num)

        self.centre = centre

        self.r = r

    def \_\_len\_\_(self):

        return self.dataset.shape[0] - 1

    def make(self):

        if type(self.r) == tuple:

            len\_r = len(self.r)

        else:

            len\_r = 1

            self.r = [self.r] #For convenience

        if self.fill == False:

            for i in range(0, (int(self.data\_num/len\_r))):

                for r in self.r:

                    ''' r^2 = sqrt(x^2 + y^2)'''

                    x = np.random.uniform(low=-r, high=r)

                    y\_sqr = r \* r - x \* x

                    if i % 2 == 0:

                        y = np.sqrt(y\_sqr)

                    else :

                        y = -np.sqrt(y\_sqr)

                    '''Adding gaussian noise and move the center of the data'''

                    x += (np.random.normal() \* self.noise) + self.centre[0]

                    y += (np.random.normal() \* self.noise) + self.centre[1]

                    self.dataset = np.append(self.dataset, np.array([[x, y]]), axis=0)

        else:

            for i in range(0, (int(self.data\_num/len\_r))):

                for r in self.r:

                    r\_fill = r - np.random.normal(loc=r, scale=1)

                    x = np.random.uniform(low=-r\_fill, high=r\_fill)

                    y\_sqr = r\_fill \* r\_fill - x \* x

                    if i % 2 == 0:

                        y = np.sqrt(y\_sqr)

                    else:

                        y = -np.sqrt(y\_sqr)

                    '''Adding gaussian noise and move the center of the data'''

                    x += (np.random.normal() \* self.noise) + self.centre[0]

                    y += (np.random.normal() \* self.noise) + self.centre[1]

                    self.dataset = np.append(self.dataset, np.array([[x, y]]), axis=0)

    def get\_dataset(self):

        return self.dataset[1:] #Excluding redundant array

class Linear\_dataset():

    def \_\_init\_\_(self, feature\_num=2, data\_num=300, centre=(0, 0), w=(2, 4), noise=0.01, range=(3, 5)):

        def dumy\_dataset(feature\_num=None):

            return np.zeros((1, feature\_num))

        self.feature\_num = feature\_num

        self.data\_num = data\_num

        self.centre = centre

        self.weight = w

        self.noise = noise

        self.dataset = dumy\_dataset(feature\_num)

        self.range = range

        self.variance = None

        self.mean = None

    def make(self):

        if type(self.weight) == tuple:

            len\_weight = len(self.weight)

            for i in range(0, (int(self.data\_num/len\_weight))):

                for w in self.weight:

                    '''first cluster'''

                    x = np.random.uniform(low=0, high=self.range[0])

                    y = x \* self.weight[0]

                    '''Adding gaussian noise and move the center of the data'''

                    x += (np.random.normal() \* self.noise) + self.centre[0]

                    y += (np.random.normal() \* self.noise) + self.centre[1]

                    self.dataset = np.append(self.dataset, np.array([[x, y]]), axis=0)

                    '''second cluster'''

                    x = np.random.uniform(low=0, high=self.range[1])

                    y = x \* self.weight[1]

                    '''Adding gaussian noise and move the center of the data'''

                    x += (np.random.normal() \* self.noise) + self.centre[0]

                    y += (np.random.normal() \* self.noise) + self.centre[1]

                    self.dataset = np.append(self.dataset, np.array([[x, y]]), axis=0)

        else:

            for i in range(0, (int(self.data\_num))):

                '''first cluster'''

                x = np.random.uniform(low=0, high=self.range)

                y = x \* self.weight

                '''Adding gaussian noise and move the center of the data'''

                x += (np.random.normal() \* self.noise) + self.centre[0]

                y += (np.random.normal() \* self.noise) + self.centre[1]

                self.dataset = np.append(self.dataset, np.array([[x, y]]), axis=0)

    def get\_dataset(self):

        return self.dataset[1:] #Excluding redundant array

class Mixture\_gaussian\_blob():

    def \_\_init\_\_(self, mean=None, variance=None, weights=None, data\_num=None, feature\_num=None):

        self.mean = mean

        self.variance = variance

        self.data\_num = data\_num

        self.feature\_num = feature\_num

        self.weights = weights

    def make(self):

        data\_num = self.data\_num

        weights = self.weights

        centre\_n = len(self.mean)

        mixed\_gaussian = np.zeros((1, 2))

        for i in range(centre\_n):

            random = (np.random.normal(self.mean[i], self.variance[i], (int(weights[i] \* data\_num), self.feature\_num)))

            mixed\_gaussian = np.append(mixed\_gaussian, random, axis=0)

        return mixed\_gaussian

def K\_means(data, k):

    print(data.shape)

    assinged = np.zeros(data.shape[0])

    cluster\_means = np.zeros((k, data.shape[1]))

    cluster\_datapoint\_num = np.zeros((k)) - 1

    converge = 0

    check = 0

    '''init the centroids'''

    for i in range(0,k):

        for j in range(data.shape[1]):

            # cluster\_means[i][0] = np.random.normal()

            # cluster\_means[i][1] = np.random.normal()\*2

            cluster\_means[i][j] = np.random.uniform(low=0, high=255)

    while converge != k:

        check += 1

        for i in range(0, data.shape[0]):

            centroids = np.array([-1, np.amax(data) \* 1000000])

            for j in range(0, k):

                mean = np.sqrt(np.sum(data[i]-cluster\_means[j])\*\*2)

                if mean < centroids[1]:

                    centroids[0] = j

                    centroids[1] = mean

            assinged[i] = centroids[0]

            print(assinged)

        for i in range(0, k):

            tmp = cluster\_datapoint\_num[i]

            ith\_cluster\_idx = np.where(assinged == i)

            cluster\_datapoint\_num[i] = len(ith\_cluster\_idx[0])

            if tmp == cluster\_datapoint\_num[i]:

                converge += 1

            # x\_sum = 0

            # y\_sum = 0

            sum = np.zeros((1, data.shape[1]))

            for j in ith\_cluster\_idx[0]:

                # x\_sum += data[j][0]

                # y\_sum += data[j][1]

                sum += data[j]

            # x\_sum /= cluster\_datapoint\_num[i]

            # y\_sum /= cluster\_datapoint\_num[i]

            sum /= cluster\_datapoint\_num[i]

            # cluster\_means[i][0] = x\_sum

            # cluster\_means[i][1] = y\_sum

            cluster\_means[i] = sum

        if converge == k:

            return assinged

        else:

            converge = 0

def K\_means\_soft(data, k):

    assinged = np.zeros(data.shape[0])

    cluster\_means = np.zeros((k, data.shape[1]))

    cluster\_datapoint\_num = np.zeros((k)) - 1

    responsibility = np.zeros((data.shape[0], data.shape[1]))

    converge = 0

    beta = 0.6

    check = 0

    '''init the centroids'''

    for i in range(0,k):

        cluster\_means[i][0] = np.random.normal()

        cluster\_means[i][1] = np.random.normal()

    while converge != k:

        check += 1

        for i in range(0, data.shape[0]):

            centroids = np.array([-1, np.amax(data) \* 1000000])

            responsibility\_tmp = np.zeros((k, data.shape[1]))

            for j in range(0, k):

                mean = np.sqrt((data[i][0]-cluster\_means[j][0])\*\*2 + (data[i][1]-cluster\_means[j][1])\*\*2)

                responsibility\_tmp[j] = np.exp(-beta \* ((data[i]-cluster\_means[j])\*\*2))

                # print(i, j, k)

                # print(cluster\_means[j])

                if mean < centroids[1]:

                    centroids[0] = j

                    centroids[1] = mean

            assinged[i] = centroids[0]

            assigned\_centroid = assinged[i]

            #

            responsibility[i] = responsibility\_tmp[int(assigned\_centroid)]

        for i in range(0, k):

            print('what')

            tmp = cluster\_datapoint\_num[i]

            ith\_cluster\_idx = np.where(assinged == i)

            cluster\_datapoint\_num[i] = len(ith\_cluster\_idx[0])

            if tmp == cluster\_datapoint\_num[i]:

                converge += 1

            x\_sum = 0

            y\_sum = 0

            for j in ith\_cluster\_idx[0]:

                x\_sum += data[j][0]

                y\_sum += data[j][1]

            x\_sum /= cluster\_datapoint\_num[i]

            y\_sum /= cluster\_datapoint\_num[i]

            cluster\_means[i][0] = x\_sum

            cluster\_means[i][1] = y\_sum

        if converge == k:

            return assinged, responsibility

        else:

            converge = 0

'''Balanced spherical data'''

DatasetA = Spherical\_shape\_like(data\_num= 500, centre=(0, 0), r=3, noise=0.1, fill=True)

DatasetA.make()

DataA = DatasetA.get\_dataset()

DatasetB = Spherical\_shape\_like(data\_num= 500, centre=(2, 2), r=3, noise=0.1, fill=True)

DatasetB.make()

DataB = DatasetB.get\_dataset()

plt.scatter(DataA[:, :1], DataA[:, 1:2], s=5)

plt.scatter(DataB[:, :1], DataB[:, 1:2], s=5)

plt.show()

Dataset\_combined = DataA

Dataset\_combined = np.append(Dataset\_combined, DataB, axis=0)

np.random.shuffle(Dataset\_combined)

asign = K\_means(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], s=5, color='red')

plt.scatter(C2[:, :1], C2[:, 1:2], s=5)

plt.show()

cluster, res = K\_means\_soft(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

r1, r2 = res[asign], res[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], c=np.sum(r1, axis=1), s=5)

plt.scatter(C2[:, :1], C2[:, 1:2], c=np.sum(r2, axis=1), s=5)

plt.show()

'''ImBalanced spherical data'''

DatasetA = Spherical\_shape\_like(data\_num= 200, centre=(0, 0), r=3, noise=0.1, fill=True)

DatasetA.make()

DataA = DatasetA.get\_dataset()

DatasetB = Spherical\_shape\_like(data\_num= 1000, centre=(2, 2), r=3, noise=0.1, fill=True)

DatasetB.make()

DataB = DatasetB.get\_dataset()

plt.scatter(DataA[:, :1], DataA[:, 1:2], s=5)

plt.scatter(DataB[:, :1], DataB[:, 1:2], s=5)

plt.show()

Dataset\_combined = DataA

Dataset\_combined = np.append(Dataset\_combined, DataB, axis=0)

np.random.shuffle(Dataset\_combined)

asign = K\_means(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], s=5, color='red')

plt.scatter(C2[:, :1], C2[:, 1:2], s=5)

plt.show()

cluster, res = K\_means\_soft(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

r1, r2 = res[asign], res[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], c=np.sum(r1, axis=1), s=5)

plt.scatter(C2[:, :1], C2[:, 1:2], c=np.sum(r2, axis=1), s=5)

plt.show()

'''Balanced non-spherical data'''

DatasetA = Linear\_dataset(data\_num= 200, centre=(3, 5), w=4, noise=0.5, range=4)

DatasetA.make()

DataA = DatasetA.get\_dataset()

DatasetB = Linear\_dataset(data\_num= 200, centre=(2, 1), w=1, noise=0.5, range=4)

DatasetB.make()

DataB = DatasetB.get\_dataset()

plt.scatter(DataA[:, :1], DataA[:, 1:2], s=5, color='blue')

plt.scatter(DataB[:, :1], DataB[:, 1:2], s=5, color='blue')

plt.show()

Dataset\_combined = DataA

Dataset\_combined = np.append(Dataset\_combined, DataB, axis=0)

np.random.shuffle(Dataset\_combined)

asign = K\_means(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], s=5, color='red')

plt.scatter(C2[:, :1], C2[:, 1:2], s=5)

plt.show()

cluster, res = K\_means\_soft(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

r1, r2 = res[asign], res[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], c=np.sum(r1, axis=1), s=5)

plt.scatter(C2[:, :1], C2[:, 1:2], c=np.sum(r2, axis=1), s=5)

plt.show()

'''Balanced non-spherical data2'''

DatasetA = Spherical\_shape\_like(data\_num= 100, centre=(0, 0), r=4, noise=0.2)

DatasetA.make()

DataA = DatasetA.get\_dataset()

DatasetB = Spherical\_shape\_like(data\_num= 300, centre=(0, 0), r=1, noise=0.2)

DatasetB.make()

DataB = DatasetB.get\_dataset()

plt.scatter(DataA[:, :1], DataA[:, 1:2], s=5, color='blue')

plt.scatter(DataB[:, :1], DataB[:, 1:2], s=5, color='blue')

plt.show()

Dataset\_combined = DataA

Dataset\_combined = np.append(Dataset\_combined, DataB, axis=0)

np.random.shuffle(Dataset\_combined)

asign = K\_means(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], s=5, color='red')

plt.scatter(C2[:, :1], C2[:, 1:2], s=5)

plt.show()

cluster, res = K\_means\_soft(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

r1, r2 = res[asign], res[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], c=np.sum(r1, axis=1), s=5)

plt.scatter(C2[:, :1], C2[:, 1:2], c=np.sum(r2, axis=1), s=5)

plt.show()

'''Balanced non-spherical data variance'''

DatasetA = Mixture\_gaussian\_blob([2, 3, 1], [1, 1.5, 0.8], [0.3, 0.4, 0.3], 1000, 2)

DataA = DatasetA.make()

DatasetB = Mixture\_gaussian\_blob([11, 15, 12], [1.7, 1.5, 2], [0.3, 0.1, 0.6], 3000, 2)

DataB = DatasetB.make()

plt.scatter(DataA[:, :1], DataA[:, 1:2], s=5, color='blue')

plt.scatter(DataB[:, :1], DataB[:, 1:2], s=5, color='blue')

plt.show()

Dataset\_combined = DataA

Dataset\_combined = np.append(Dataset\_combined, DataB, axis=0)

np.random.shuffle(Dataset\_combined)

asign = K\_means(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], s=5, color='red')

plt.scatter(C2[:, :1], C2[:, 1:2], s=5)

plt.show()

cluster, res = K\_means\_soft(Dataset\_combined, 2)

asign = asign.astype(bool)

C1, C2 = Dataset\_combined[asign], Dataset\_combined[~asign]

r1, r2 = res[asign], res[~asign]

plt.scatter(C1[:, :1], C1[:, 1:2], c=np.sum(r1, axis=1), s=5)

plt.scatter(C2[:, :1], C2[:, 1:2], c=np.sum(r2, axis=1), s=5)

plt.show()

'''real data'''

image = cv2.imread('musium.jpg')

plt.show()

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

image = image.reshape((40000, 3))

asign = K\_means(image, 5)

image[np.where(asign == 0)] = 80

image[np.where(asign == 1)] = 120

image[np.where(asign == 2)] = 160

image[np.where(asign == 3)] = 200

image[np.where(asign == 4)] = 240

image = image.reshape((200, 200, 3))

plt.imshow(image)

plt.show()