Final Project_2015-17231_박우정

November 28, 2019

[9]:
```
#1. k-means clustering

## initialization. k와 max_iteration I, r_nk
k=int(input("number of cluster?"))
max_iter=int(input("number of max_iteration?"))


##import library
import numpy as np
import matplotlib.pyplot as plt

##read data.csv
data1=np.loadtxt('data1.csv',delimiter=',')
data2=np.loadtxt('data2.csv',delimiter=',')
data3=np.loadtxt('data3.csv',delimiter=',') #모두 120 * 2 matrix
x1=data1[:,0]
y1=data1[:,1]
x2=data2[:,0]
y2=data2[:,1]
x3=data3[:,0]
y3=data3[:,1] #coordinates 별로 저장

plt.scatter(x1,y1,color='red')
plt.scatter(x2,y2,color='green')
plt.scatter(x3,y3,color='blue')
plt.title("All the scatter plots of data1(red), data2(green), and data3(blue)")
plt.show()
print("\n\n\n")

##declare functions
def dist(a,b):  #Euclidean norm
    return np.linalg.norm(b-a)

def init_centroids(data, k): #k를 받고, initial points로부터 k개의 중심 c1, c2,␣
 ↪c3, ..., ck를 리턴하는 함수
    centroids = data.copy()
    np.random.shuffle(centroids)
```

```
    return centroids[:k]

def closest_centroid(data, centroids): #Euclidean norm을 거리로 하는 distance를 주고, 그것이 최소가 되게 하는 중심좌표 index 반환.
    distances = np.sqrt(((data - centroids[:, np.newaxis])**2).sum(axis=2))
    #np.newaxis를 통해 각 data안의 성분들에 고정된 centroids가 모두 빼지도록 한다.
    #np.newaxis는 차원을 확장하는 함수. 가령, [1,2,3,4] -> [[1,2,3,4]] ->␣
 ↪[[1][2][3][4]] 이런 방식으로 확장.
    return np.argmin(distances, axis=0)
#closest_centroid(data3,centroids) #출력된 값은 c_0, c_1, ..., c_(k-1)의 index 0␣
 ↪1 2 ... k-1 중 하나이다. 가까운 index 반환

def new_centroids(data, closest, centroids): #위에서 구한 index들끼리 모아서 새로운␣
 ↪centroid를 구한다.
    return np.array([data[closest==k].mean(axis=0) for k in range(centroids.
 ↪shape[0])])

##data 1
plt.scatter(data1[:, 0], data1[:, 1])
centroids = init_centroids(data1, k)

for i in range(0, max_iter):
    closest = closest_centroid(data1, centroids)
    ncentroids = new_centroids(data1, closest, centroids)
    if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids, centroids)==0):
        print("The iteration number of data 1 is: ", i)
        break;
    centroids=new_centroids(data1, closest, centroids)

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 1 k-means clustering result with error=10^-3")
plt.show()

print("The coordinates of centroids of data1 is:\n", centroids)
print("The clusters of data1 is:\n", closest)
print("\n\n\n")
```

```
number of cluster?4
number of max_iteration?1000
```


```
The iteration number of data 1 is:  3
```


```
The coordinates of centroids of data1 is:
 [[  7.46629799  54.81537825]
 [-44.70775134 -38.82388583]
 [ 43.20121589 -39.51006513]
 [-26.23140434 -54.33709625]]
The clusters of data1 is:
 [2 1 1 3 0 1 1 2 3 1 1 2 1 2 2 1 1 3 3 1 1 1 1 3 1 1 1 1 3 2 0 1 3 1 0 1 3
 1 0 1 3 1 0 1 0 1 3 1 0 1 0 1 3 0 3 1 0 3 2 3 0 2 0 1 1 1 3 3 2 1 1 3 0 2
 1 1 0 2 0 1 1 2 0 1 3 1 0 1 1 0 3 0 1 3 1 1 0 0 2 1 1 2 2 1 2 1 1 1 2 3 1
 1 2 2 3 0 3 1 3 3]
```

[2]:
```
#1. k-means clustering

## initialization. k와 max_iteration I, r_nk
k=int(input("number of cluster?"))
max_iter=int(input("number of max_iteration?"))


##import library
import numpy as np
import matplotlib.pyplot as plt
```

```
##read data.csv
data1=np.loadtxt('data1.csv',delimiter=',')
data2=np.loadtxt('data2.csv',delimiter=',')
data3=np.loadtxt('data3.csv',delimiter=',') #모두 120 * 2 matrix
x1=data1[:,0]
y1=data1[:,1]
x2=data2[:,0]
y2=data2[:,1]
x3=data3[:,0]
y3=data3[:,1] #coordinates 별로 저장

plt.scatter(x1,y1,color='red')
plt.scatter(x2,y2,color='green')
plt.scatter(x3,y3,color='blue')
plt.title("All the scatter plots of data1(red), data2(green), and data3(blue)")
plt.show()
print("\n\n\n")

##declare functions
def dist(a,b):  #Euclidean norm
    return np.linalg.norm(b-a)

def init_centroids(data, k): #k를 받고, initial points로부터 k개의 중심 c1, c2,␣
↪c3, ..., ck를 리턴하는 함수
    centroids = data.copy()
    np.random.shuffle(centroids)
    return centroids[:k]

def closest_centroid(data, centroids): #Euclidean norm을 거리로 하는 distance를 주
고, 그것이 최소가 되게 하는 중심좌표 index 반환.
    distances = np.sqrt(((data - centroids[:, np.newaxis])**2).sum(axis=2))
    #np.newaxis를 통해 각 data안의 성분들에 고정된 centroids가 모두 빼지도록 한다.
    #np.newaxis는 차원을 확장하는 함수. 가령, [1,2,3,4] -> [[1,2,3,4]] ->␣
↪[[1][2][3][4]] 이런 방식으로 확장.
    return np.argmin(distances, axis=0)
#closest_centroid(data3,centroids) #출력된 값은 c_0, c_1, ..., c_(k-1)의 index 0␣
↪1 2 ... k-1 중 하나이다. 가까운 index 반환

def new_centroids(data, closest, centroids): #위에서 구한 index들끼리 모아서 새로운␣
↪centroid를 구한다.
    return np.array([data[closest==k].mean(axis=0) for k in range(centroids.
↪shape[0])])

##data 2
plt.scatter(data2[:, 0], data2[:, 1])
```

```
centroids = init_centroids(data2, k)

for i in range(0, max_iter):
    closest = closest_centroid(data2, centroids)
    ncentroids = new_centroids(data2, closest, centroids)
    if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids, centroids)==0):
        print("The iteration number of data 2 is: ", i)
        break;
    centroids=new_centroids(data2, closest, centroids)

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 2 k-means clustering result with error=10^-3")
plt.show()

print("The coordinates of centroids of data2 is:\n", centroids)
print("The clusters of data1 is:\n", closest)
print("\n\n\n")
```

```
number of cluster?6
number of max_iteration?1000
```

```
The iteration number of data 2 is:  1
```

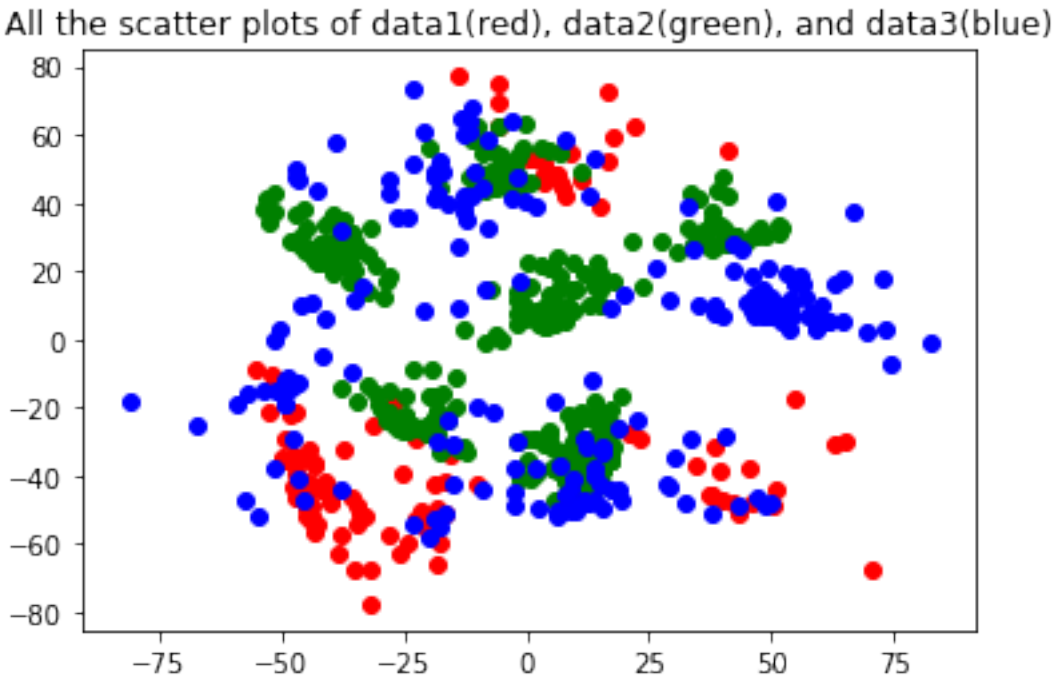

```
The coordinates of centroids of data2 is:
 [[-40.69362868  26.77287789]
 [ 40.29056533  31.42481471]
 [  5.23192998  10.58650307]
 [ -5.35294772  50.73166275]
 [ 12.12524292 -32.66803269]
 [-23.14794187 -22.33230429]]
```

[7]:
```
#1. k-means clustering

## initialization. k와 max_iteration I, r_nk
k=int(input("number of cluster?"))
max_iter=int(input("number of max_iteration?"))


##import library
import numpy as np
import matplotlib.pyplot as plt
```

```
##read data.csv
data1=np.loadtxt('data1.csv',delimiter=',')
data2=np.loadtxt('data2.csv',delimiter=',')
data3=np.loadtxt('data3.csv',delimiter=',') #모두 120 * 2 matrix
x1=data1[:,0]
y1=data1[:,1]
x2=data2[:,0]
y2=data2[:,1]
x3=data3[:,0]
y3=data3[:,1] #coordinates 별로 저장

plt.scatter(x1,y1,color='red')
plt.scatter(x2,y2,color='green')
plt.scatter(x3,y3,color='blue')
plt.title("All the scatter plots of data1(red), data2(green), and data3(blue)")
plt.show()
print("\n\n\n")

##declare functions
def dist(a,b):  #Euclidean norm
    return np.linalg.norm(b-a)

def init_centroids(data, k): #k를 받고, initial points로부터 k개의 중심 c1, c2,␣
 ↪c3, ..., ck를 리턴하는 함수
    centroids = data.copy()
    np.random.shuffle(centroids)
    return centroids[:k]

def closest_centroid(data, centroids): #Euclidean norm을 거리로 하는 distance를 주
고, 그것이 최소가 되게 하는 중심좌표 index 반환.
    distances = np.sqrt(((data - centroids[:, np.newaxis])**2).sum(axis=2))
    #np.newaxis를 통해 각 data안의 성분들에 고정된 centroids가 모두 빼지도록 한다.
    #np.newaxis는 차원을 확장하는 함수. 가령, [1,2,3,4] -> [[1,2,3,4]] ->␣
 ↪[[1][2][3][4]] 이런 방식으로 확장.
    return np.argmin(distances, axis=0)
#closest_centroid(data3,centroids) #출력된 값은 c_0, c_1, ..., c_(k-1)의 index 0␣
 ↪1 2 ... k-1 중 하나이다. 가까운 index 반환

def new_centroids(data, closest, centroids): #위에서 구한 index들끼리 모아서 새로운␣
 ↪centroid를 구한다.
    return np.array([data[closest==k].mean(axis=0) for k in range(centroids.
 ↪shape[0])])

##data 3
plt.scatter(data3[:, 0], data3[:, 1])
centroids = init_centroids(data3, k)
```

```
for i in range(0, max_iter):
    closest = closest_centroid(data3, centroids)
    ncentroids = new_centroids(data3, closest, centroids)
    if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids, centroids)==0):
        print("The iteration number of data3 is: ", i)
        break;
    centroids=new_centroids(data3, closest, centroids)

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 3 k-means clustering result with error=10^-3")
plt.show()

print("The coordinates of centroids of data3 is:\n", centroids)
print("The clusters of data1 is:\n", closest)
```

```
number of cluster?6
number of max_iteration?1000
```




```
The iteration number of data3 is:  6
```


```
The coordinates of centroids of data3 is:
 [[-50.95008522 -20.46132768]
 [-11.50396023  49.98153678]
 [ 51.14102688  11.88825836]
 [-31.02680348  25.77939495]
 [ 21.82375815 -40.63277086]
 [ -6.32328949 -40.99479766]]
```

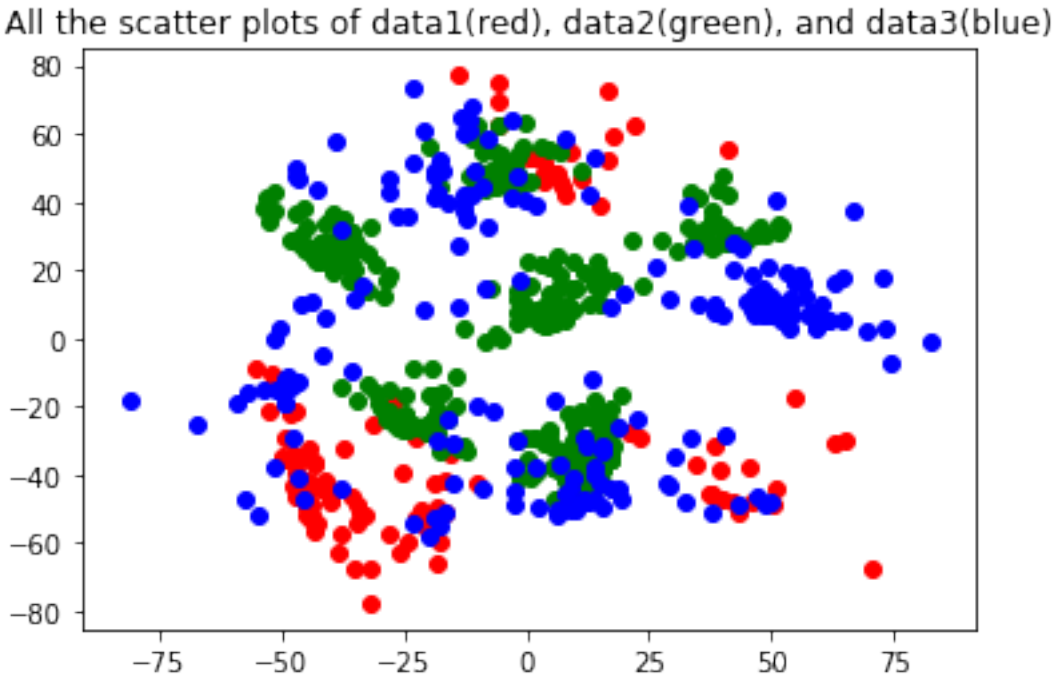
[11]:

```
#2. LBG clustering

## initialization. k와 max_iteration I, r_nk
epsilon=float(input("error?(10^-3 recommended)"))
max_iter=int(input("number of max_iteration?"))



##import library
import numpy as np
import matplotlib.pyplot as plt

##read data.csv
data1=np.loadtxt('data1.csv',delimiter=',')
data2=np.loadtxt('data2.csv',delimiter=',')
data3=np.loadtxt('data3.csv',delimiter=',') #모두 120 * 2 matrix
x1=data1[:,0]
```

```
y1=data1[:,1]
x2=data2[:,0]
y2=data2[:,1]
x3=data3[:,0]
y3=data3[:,1] #coordinates 별로 저장

plt.scatter(x1,y1,color='red')
plt.scatter(x2,y2,color='green')
plt.scatter(x3,y3,color='blue')
plt.title("All the scatter plots of data1(red), data2(green), and data3(blue)")
plt.show()
print("\n\n\n")

##declare functions
def dist(a,b):  #Euclidean norm
    return np.linalg.norm(b-a)

def init_centroids(data, k): #k를 받고, initial points로부터 k개의 중심 c1, c2,␣
 ↪c3, ..., ck를 리턴하는 함수
    centroids = data.copy()
    np.random.shuffle(centroids)
    return centroids[:k]

def closest_centroid(data, centroids): #Euclidean norm을 거리로 하는 distance를 주
고, 그것이 최소가 되게 하는 중심좌표 index 반환.
    distances = np.sqrt(((data - centroids[:, np.newaxis])**2).sum(axis=2))
    #np.newaxis를 통해 각 data안의 성분들에 고정된 centroids가 모두 빼지도록 한다.
    #np.newaxis는 차원을 확장하는 함수. 가령, [1,2,3k4] -> [[1,2,3,4]] ->␣
 ↪[[1][2][3][4]] 이런 방식으로 확장.
    return np.argmin(distances, axis=0)
#closest_centroid(data3,centroids) #출력된 값은 c_0, c_1, ..., c_(k-1)의 index 0␣
 ↪1 2 ... k-1 중 하나이다. 가까운 index 반환

def new_centroids(data, closest, centroids): #위에서 구한 index들끼리 모아서 새로운␣
 ↪centroid를 구한다.
    return np.array([data[closest==k].mean(axis=0) for k in range(centroids.
 ↪shape[0])])

##data 1
plt.scatter(data1[:, 0], data1[:, 1])
### start at k_1=1

k_1=1

centroids = np.array([[data1[:,0].mean(), data1[:,1].mean()]]) #초기 중심은 무게중
심.
```

```
distance = ((data1 - centroids[:, np.newaxis])**2).sum(axis=2) #120개 거리만을 적은 벡터. 가장 가까운 것을 찾자.
add_centroids=np.array([data1[distance.argmin(),]]) #가장 가까운 것.

for j in range(0, max_iter):
    centroids=np.array(np.append(centroids, add_centroids, axis=0)) #둘을 결합하여 중심 2개 집합 생성
    D_old = ((data1 - centroids[:, np.newaxis])**2).sum()
    for i in range(0, max_iter):
        closest = closest_centroid(data1, centroids)
        ncentroids = new_centroids(data1, closest, centroids)
        if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids,␣
 ↪centroids)==0):
            #print("The iteration number of data 1 is: ", i)
            break
        else:
            centroids=new_centroids(data1, closest, centroids)
    D_new = ((data1 - centroids[:, np.newaxis])**2).sum()
    error=abs(D_new-D_old)/D_new
    if((D_new-D_old)/D_new<epsilon or D_new==0 or k_1>=120):
        break
    else:
        k_1=k_1*2
        add_centroids=init_centroids(data1, k_1)
        continue

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 1 k-means clustering result with error=10^-3")
plt.show()

print("The optimized number of clusters is: \n", k_1*2)
print("The coordinates of centroids of data1 is:\n", centroids)
print("The clusters of data1 is:\n", closest)
print("\n\n\n")
```

```
error?(10^-3 recommended)0.001
number of max_iteration?100
```


All the scatter plots of data1(red), data2(green), and data3(blue)
80
60
40
20
0
−20
−40
−60
−80
−75
−50
−25
0
25
50
75


data 1 k-means clustering result with error=10^-3
80
60
40
20
0
−20
−40
−60
−80
−60
−40
−20
0
20
40
60

```
The optimized number of clusters is:
 8
The coordinates of centroids of data1 is:
 [[ 10.83772994  51.25822119]
 [-45.16020636 -43.1610801 ]
 [-34.80421915 -59.10501084]
 [ 43.20121589 -39.51006513]
 [-26.57143019 -29.60545592]
 [-18.64096929 -51.89151336]
 [-50.90584589 -17.90676474]
 [ -7.70514581  70.82258502]]
The clusters of data1 is:
 [3 1 1 2 7 1 6 3 2 1 1 3 4 3 3 1 1 5 4 1 1 1 1 2 1 1 1 1 2 3 7 1 5 6 0 1 2
 1 0 1 5 1 0 1 7 1 5 1 0 4 0 1 5 7 5 1 0 5 3 5 0 3 0 1 1 1 2 5 3 1 6 2 0 3
 4 1 0 3 0 1 6 3 0 1 5 1 0 6 1 0 2 0 6 2 6 1 0 0 3 1 4 3 3 2 3 1 1 1 3 2 1
 1 3 3 5 0 2 1 4 2]
```

[41]:
```
##data 2
plt.scatter(data2[:, 0], data2[:, 1])
### start at k_2=1

k_2=1

centroids = np.array([[data2[:,0].mean(), data2[:,1].mean()]]) #초기 중심은 무게중
심.
distance = ((data2 - centroids[:, np.newaxis])**2).sum(axis=2) #120개 거리만을 적
은 벡터. 가장 가까운 것을 찾자.
add_centroids=np.array([data2[distance.argmin(),]]) #가장 가까운 것.

for j in range(0, max_iter):
    centroids=np.array(np.append(centroids, add_centroids, axis=0)) #둘을 결합하
여 중심 2개 집합 생성
    D_old = ((data2 - centroids[:, np.newaxis])**2).sum()
    for i in range(0, max_iter):
        closest = closest_centroid(data2, centroids)
        ncentroids = new_centroids(data2, closest, centroids)
        if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids,␣
 ↪centroids)==0):
            #print("The iteration number of data 1 is: ", i)
            break
```

```
        else:
            centroids=new_centroids(data2, closest, centroids)
    D_new = ((data2 - centroids[:, np.newaxis])**2).sum()
    error=abs(D_new-D_old)/D_new
    if((D_new-D_old)/D_new<epsilon or D_new==0 or k_2>=120):
        break
    else:
        k_2=2*k_2
        add_centroids=init_centroids(data2, k_2)
        continue

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 2 k-means clustering result with error=10^-3")
plt.show()

print("The optimized number of clusters is: \n", k_2*2)
print("The coordinates of centroids of data2 is:\n", centroids)
print("The clusters of data2 is:\n", closest)
print("\n\n\n")
```


```
The optimized number of clusters is:
 8
The coordinates of centroids of data2 is:
 [[ 11.57463154  18.02447884]
 [ -5.35294772  50.73166275]
```

```
 [-40.69362868  26.77287789]
 [ 12.12524292 -32.66803269]
 [-23.14794187 -22.33230429]
 [  2.62094678   7.49833923]
 [ 42.91385454  29.73755941]
 [ 35.75965734  35.33087055]]
The clusters of data2 is:
 [1 3 0 6 5 7 4 2 3 1 3 4 2 3 2 0 3 3 1 1 7 6 5 6 3 4 0 3 1 6 2 6 3 2 7 3 6
 3 3 5 2 3 5 3 3 1 2 0 1 3 3 6 1 3 3 3 3 5 3 1 3 2 4 4 3 3 5 3 0 3 2 3 3 5
 1 3 3 6 6 5 7 4 7 5 3 3 2 2 1 2 0 7 5 4 3 6 1 2 3 5 2 1 6 2 5 2 4 6 3 0 3
 5 4 3 3 4 3 3 5 4 1 3 6 1 5 6 5 3 2 3 4 1 6 3 5 6 2 4 3 2 3 0 2 3 4 0 7 2
 5 3 2 3 2 3 1 6 4 4 5 6 3 3 4 2 4 2 2 1 3 3 0 2 0 4 7 5 3 1 2 1 1 5 3 5 2
 4 3 7 5 3 0 6 3 4 5 6 2 3 3 4 3 4 3 6 2 2 4 2 3 5 4 1 2 3 5 3 3 3 6 3 5 3
 3 2 1 4 5 5 2 3 3 3 5 3 3 1 5 4 2 5 5 5 6 0 0 4 2 3 2 1 0 3 1 5 2 4 5 5 3
 3 6 7 5 3 6 0 1 5 6 3 7 1 5 2 4 0 3 3 0 2 3 5 6 4 4 5 3 2 2 3 7 2 4 4 3 6
 4 3 3 3 1 4 5 6 1 5 1 3 2 1 2 2 4 2 1 1 2 4 2 3 4 3 3 1 4 2 2 4 0 4 6 5 2
 3 4 4 2 2 0 1 3 4 1 3 1 2 3 7 1 2 4 3 2 3 3 7 3 6 6 1 2 3 3 5 0 3 4 6 5 4
 5 2 3 4 3 5 2 0 2 5 2 7 0 3 0 1 2 4 2 3 3 3 3 3 5 5 2 1 3 1]
```

[43]:
```
##data 3
plt.scatter(data3[:, 0], data3[:, 1])
### start at k_3=1

k_3=1

centroids = np.array([[data3[:,0].mean(), data3[:,1].mean()]]) #초기 중심은 무게중
심.
distance = ((data3 - centroids[:, np.newaxis])**2).sum(axis=2) #120개 거리만을 적
은 벡터. 가장 가까운 것을 찾자.
add_centroids=np.array([data3[distance.argmin(),]]) #가장 가까운 것.

for j in range(0, max_iter):
    centroids=np.array(np.append(centroids, add_centroids, axis=0)) #둘을 결합하
여 중심 2개 집합 생성
    D_old = ((data3 - centroids[:, np.newaxis])**2).sum()
    for i in range(0, max_iter):
        closest = closest_centroid(data3, centroids)
        ncentroids = new_centroids(data3, closest, centroids)
        if(dist(ncentroids, centroids)<10^(-3) or dist(ncentroids,␣
 ↪centroids)==0):
            #print("The iteration number of data 1 is: ", i)
            break
        else:
```

```
            centroids=new_centroids(data3, closest, centroids)
    D_new = ((data3 - centroids[:, np.newaxis])**2).sum()
    error=abs(D_new-D_old)/D_new
    if((D_new-D_old)/D_new<epsilon or D_new==0 or k_3>=120):
        break
    else:
        k_3=2*k_3
        add_centroids=init_centroids(data3, k_3)
        continue

plt.scatter(centroids[:, 0], centroids[:, 1], c='r', s=100)
plt.title("data 3 k-means clustering result with error=10^-3")
plt.show()

print("The optimized number of clusters is: \n", k_3*2)
print("The coordinates of centroids of data3 is:\n", centroids)
print("The clusters of data3 is:\n", closest)
print("\n\n\n")
```


```
The optimized number of clusters is:
 4
The coordinates of centroids of data3 is:
 [[-48.4467347  -15.01285632]
 [ 51.14102688  11.88825836]
 [ 10.17670395 -40.78257506]
```

```
 [-15.8484895   45.67220216]]
The clusters of data3 is:
 [2 1 0 2 2 0 1 1 1 1 0 3 2 2 0 2 1 3 1 0 3 1 3 3 3 1 1 2 2 3 3 3 3 3 2 0 0
 0 0 2 3 1 3 1 2 1 1 0 2 3 2 2 1 1 1 1 2 1 0 2 2 0 3 1 2 2 1 1 0 1 2 2 2 2
 2 1 2 2 1 3 1 0 0 0 2 0 1 1 3 1 2 0 0 0 1 1 1 3 3 1 0 3 2 1 3 2 1 3 2 1 1
 0 1 3 2 3 2 1 1 2 3 1 0 1 1 3 2 2 2 2 1 3 2 0 1 2 2 2 2 1 3 2 0 1 3 0 2 2
 2 1 3 1 3 3 1 3 3 3 1 1 1 0 0 0 0 2 2 3 2 3 1 0 0 3 3 2 2 0 2 2 3 2 1 3 3
 1 1 3 2 0 3 2 2 3 3 1 1 1 1 1]
```

[14]:
```python
#3. SVM
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score

#keras libary에서 MNIST dataset을 다운받는다.
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#60000 by 28 by 28을 데이터처리를 위해 60000 by 28로 reshape한다.
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# normalize를 위해 나눗셈이 필요하고, 이를 위해서는 int형 대신 float 형을 사용.
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Scale the data to lie between -1 to 1(SVM ppt 참조)
x_train = x_train / 255.0*100 - 50
x_test = x_test / 255.0*100 - 50
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

#I one hot encoding step.
y_train =y_train % 2
y_test =y_test % 2

# PCA
pca = PCA(n_components=50)
x_train = pca.fit_transform(x_train)
x_test = pca.transform(x_test)

#매개변수 gamma와 C의 최적값을 찾기 위해 Gridsearchcv 이용
```

```
svm = SVC()
parameters = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4], 'C': [1, 10, 100,␣
 ↪1000]}]

print("Grid search")
grid = GridSearchCV(svm, parameters, verbose=3)
print("Grid.fit")
grid.fit(x_train[0:7000], y_train[0:7000]) #grid search learning the best␣
 ↪parameters
print("Grid done")

print (grid.best_params_)

#훈련 학습
print("training svm")
best_svm = grid.best_estimator_
best_svm.fit(x_train , y_train)
print("svm done")


print("Testing")
print("score: ", best_svm.score(x_test, y_test,))
```

```
60000 train samples
10000 test samples
Grid search
Grid.fit
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[CV] C=1, gamma=0.001, kernel=rbf ...

C:\Users\dnrlf\Anaconda3\lib\site-
packages\sklearn\model_selection\_split.py:1978: FutureWarning: The default
value of cv will change from 3 to 5 in version 0.22. Specify it explicitly to
silence this warning.
  warnings.warn(CV_WARNING, FutureWarning)
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[CV] ... C=1, gamma=0.001, kernel=rbf, score=0.509, total=   6.0s
[CV] C=1, gamma=0.001, kernel=rbf ...

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    5.9s remaining:    0.0s

[CV] ... C=1, gamma=0.001, kernel=rbf, score=0.508, total=   5.5s
[CV] C=1, gamma=0.001, kernel=rbf ...

[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:   11.4s remaining:    0.0s

[CV] ... C=1, gamma=0.001, kernel=rbf, score=0.508, total=   5.8s
[CV] C=1, gamma=0.0001, kernel=rbf ...
[CV] ... C=1, gamma=0.0001, kernel=rbf, score=0.592, total=   5.3s
```

```
[CV] C=1, gamma=0.0001, kernel=rbf ...
[CV] ... C=1, gamma=0.0001, kernel=rbf, score=0.596, total=   5.2s
[CV] C=1, gamma=0.0001, kernel=rbf ...
[CV] ... C=1, gamma=0.0001, kernel=rbf, score=0.594, total=   4.7s
[CV] C=10, gamma=0.001, kernel=rbf ...
[CV] ... C=10, gamma=0.001, kernel=rbf, score=0.509, total=   4.9s
[CV] C=10, gamma=0.001, kernel=rbf ...
[CV] ... C=10, gamma=0.001, kernel=rbf, score=0.508, total=   4.9s
[CV] C=10, gamma=0.001, kernel=rbf ...
[CV] ... C=10, gamma=0.001, kernel=rbf, score=0.508, total=   4.9s
[CV] C=10, gamma=0.0001, kernel=rbf ...
[CV] ... C=10, gamma=0.0001, kernel=rbf, score=0.597, total=   4.9s
[CV] C=10, gamma=0.0001, kernel=rbf ...
[CV] ... C=10, gamma=0.0001, kernel=rbf, score=0.604, total=   5.2s
[CV] C=10, gamma=0.0001, kernel=rbf ...
[CV] ... C=10, gamma=0.0001, kernel=rbf, score=0.600, total=   5.0s
[CV] C=100, gamma=0.001, kernel=rbf ...
[CV] ... C=100, gamma=0.001, kernel=rbf, score=0.509, total=   5.1s
[CV] C=100, gamma=0.001, kernel=rbf ...
[CV] ... C=100, gamma=0.001, kernel=rbf, score=0.508, total=   5.2s
[CV] C=100, gamma=0.001, kernel=rbf ...
[CV] ... C=100, gamma=0.001, kernel=rbf, score=0.508, total=   5.1s
[CV] C=100, gamma=0.0001, kernel=rbf ...
[CV] ... C=100, gamma=0.0001, kernel=rbf, score=0.597, total=   4.9s
[CV] C=100, gamma=0.0001, kernel=rbf ...
[CV] ... C=100, gamma=0.0001, kernel=rbf, score=0.604, total=   4.9s
[CV] C=100, gamma=0.0001, kernel=rbf ...
[CV] ... C=100, gamma=0.0001, kernel=rbf, score=0.600, total=   4.8s
[CV] C=1000, gamma=0.001, kernel=rbf ...
[CV] ... C=1000, gamma=0.001, kernel=rbf, score=0.509, total=   4.9s
[CV] C=1000, gamma=0.001, kernel=rbf ...
[CV] ... C=1000, gamma=0.001, kernel=rbf, score=0.508, total=   5.1s
[CV] C=1000, gamma=0.001, kernel=rbf ...
[CV] ... C=1000, gamma=0.001, kernel=rbf, score=0.508, total=   5.5s
[CV] C=1000, gamma=0.0001, kernel=rbf ...
[CV] ... C=1000, gamma=0.0001, kernel=rbf, score=0.597, total=   5.0s
[CV] C=1000, gamma=0.0001, kernel=rbf ...
[CV] ... C=1000, gamma=0.0001, kernel=rbf, score=0.604, total=   4.9s
[CV] C=1000, gamma=0.0001, kernel=rbf ...
[CV] ... C=1000, gamma=0.0001, kernel=rbf, score=0.600, total=   4.7s

[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:  2.0min finished

Grid done
{'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
training svm
svm done
Testing
score:  0.6283
```

[13]:
```
#4 MLP
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
import tensorflow.keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop

batch_size = 128 #batch : 나누어진 dataset이고 batch_size는 한 번의 batch마다 주는 데이터 샘플의 size를 의미한다.
num_classes = 10 #MNIST는 0부터 9까지 10개의 카테고리를 가지므로 10으로 둔다.
epochs = 20 #1 epoch : 신경망에서 전체 dataset 학습을 한 번 완료. 즉 각 퍼셉트론 간 w를 계산하는 것이 한 세션 완료.

(x_train, y_train), (x_test, y_test) = mnist.load_data() #데이터 download 하고 각각 항목에 저장.
#x_train, x_test에는 각각 60000 by 28 * 28 , 10000 by 28 * 28
#y_train, y_test에는 각각 60000, 100000 개의 label 저장.

#위에서 다운받은 MNIST data는 60000 by 28 by 28 이다. 처리를 위해 60000 by 784 by 1로 만들기 위해 다음을 이용한다.
x_train = x_train.reshape((x_train.shape[0], -1)) #-1은 차원을 줄이는 역할.
x_test = x_test.reshape((x_test.shape[0], -1))

#normalizing process
x_train = x_train.astype('float32') #int type이면 나누는데 오류가 발생하므로 float type으로 바꾸어준다.
x_test = x_test.astype('float32')
x_train /= 255 #MNIST matrix는 0~255사이의 값을 element로 가지므로 255로 나누어 normalize한다.
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# 신경망을 구현하기 위해 one-hot encoding하는 과정.
##one-hot encoding : data, label에 해당되는 index만 1로, 나머지는 0으로 단순화하는 자연어 처리과정.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

#이제 keras 내장함수를 이용해 relu activation function을 활용해 MLP를 구현.
#hidden layer=3개
model = Sequential()
```

```
model.add(Dense(512, activation='relu', input_shape=(784,))) #784개 입력해서 512
개 출력
model.add(Dropout(0.2)) #For fast modelling and prevent overfitting.dropout 0.2
는 randomly하게 연결된 뉴런들을 0.2만큼 drop out한다는 뜻이다.
model.add(Dense(512, activation='relu')) #512개 출력
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax')) #10개 출력

model.summary()

model.compile(loss='categorical_crossentropy', optimizer=RMSprop(),␣
 ↪metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,␣
 ↪validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0) #verbose=0 로깅 없음
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
60000 train samples
10000 test samples
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               401920
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 18s 302us/sample - loss: 0.2447 -
accuracy: 0.9237 - val_loss: 0.1264 - val_accuracy: 0.9603
Epoch 2/20
60000/60000 [==============================] - 16s 265us/sample - loss: 0.1030 -
accuracy: 0.9685 - val_loss: 0.0853 - val_accuracy: 0.9751
```

```
Epoch 3/20
60000/60000 [==============================] - 17s 288us/sample - loss: 0.0758 -
accuracy: 0.9778 - val_loss: 0.0833 - val_accuracy: 0.9776
Epoch 4/20
60000/60000 [==============================] - 17s 287us/sample - loss: 0.0609 -
accuracy: 0.9815 - val_loss: 0.0783 - val_accuracy: 0.9786
Epoch 5/20
60000/60000 [==============================] - 17s 280us/sample - loss: 0.0522 -
accuracy: 0.9841 - val_loss: 0.0727 - val_accuracy: 0.9810
Epoch 6/20
60000/60000 [==============================] - 17s 280us/sample - loss: 0.0450 -
accuracy: 0.9867 - val_loss: 0.0773 - val_accuracy: 0.9801
Epoch 7/20
60000/60000 [==============================] - 17s 280us/sample - loss: 0.0400 -
accuracy: 0.9880 - val_loss: 0.0803 - val_accuracy: 0.9812
Epoch 8/20
60000/60000 [==============================] - 17s 276us/sample - loss: 0.0353 -
accuracy: 0.9898 - val_loss: 0.0817 - val_accuracy: 0.9830
Epoch 9/20
60000/60000 [==============================] - 17s 285us/sample - loss: 0.0311 -
accuracy: 0.9909 - val_loss: 0.0857 - val_accuracy: 0.9822
Epoch 10/20
60000/60000 [==============================] - 17s 287us/sample - loss: 0.0279 -
accuracy: 0.9917 - val_loss: 0.0949 - val_accuracy: 0.9837
Epoch 11/20
60000/60000 [==============================] - 17s 284us/sample - loss: 0.0279 -
accuracy: 0.9919 - val_loss: 0.1037 - val_accuracy: 0.9822
Epoch 12/20
60000/60000 [==============================] - 17s 278us/sample - loss: 0.0239 -
accuracy: 0.9933 - val_loss: 0.0983 - val_accuracy: 0.9829
Epoch 13/20
60000/60000 [==============================] - 18s 294us/sample - loss: 0.0236 -
accuracy: 0.9931 - val_loss: 0.1016 - val_accuracy: 0.9832
Epoch 14/20
60000/60000 [==============================] - 16s 271us/sample - loss: 0.0202 -
accuracy: 0.9939 - val_loss: 0.1044 - val_accuracy: 0.9830
Epoch 15/20
60000/60000 [==============================] - 17s 282us/sample - loss: 0.0226 -
accuracy: 0.9939 - val_loss: 0.1041 - val_accuracy: 0.9838
Epoch 16/20
60000/60000 [==============================] - 18s 301us/sample - loss: 0.0207 -
accuracy: 0.9944 - val_loss: 0.1126 - val_accuracy: 0.9822
Epoch 17/20
60000/60000 [==============================] - 17s 276us/sample - loss: 0.0183 -
accuracy: 0.9950 - val_loss: 0.1079 - val_accuracy: 0.9830
Epoch 18/20
60000/60000 [==============================] - 17s 277us/sample - loss: 0.0193 -
accuracy: 0.9947 - val_loss: 0.1296 - val_accuracy: 0.9828
```

```
Epoch 19/20
60000/60000 [==============================] - 16s 266us/sample - loss: 0.0187 -
accuracy: 0.9949 - val_loss: 0.1311 - val_accuracy: 0.9835
Epoch 20/20
60000/60000 [==============================] - 13s 218us/sample - loss: 0.0206 -
accuracy: 0.9947 - val_loss: 0.1226 - val_accuracy: 0.9832
Test loss: 0.12259161651833374
Test accuracy: 0.9832
```

[ ]: