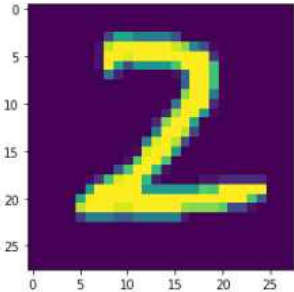


과제명	HW4	학번	201611182	이름	최동현
-----	-----	----	-----------	----	-----

보고서 내용	주요코드, 결과, 결과에 대한 분석을 기술한 보고서(jupyter notebook 활용)
--------	---

문제 번호	내용
데이터 세팅	<div>○ MNIST 데이터 불러오기</div> <pre> from keras.datasets import mnist import numpy as np import matplotlib.pyplot as plt import random (X_train, y_train), (X_test, y_test) = mnist.load_data() X_train = X_train.astype('float32') / 255. X_test = X_test.astype('float32') / 255. print (X_train.shape) X_train = X_train.reshape((len(X_train), np.prod(X_train.shape[1:]))) X_test = X_test.reshape((len(X_test), np.prod(X_test.shape[1:]))) print (X_train.shape) print (X_test.shape) (60000, 28, 28) (60000, 784) (10000, 784) </pre> <p>-데이터를 불러오고, shape한 결과</p>
	<div>○ 클래스별 100개 총 1000개 변수 만들기</div> <pre> #X for i in range(10): X_list[i] = random.sample(X_list[i], 100) new_X_train = np.concatenate(X_list, axis=0) print("숫자당 100개를 랜덤 추출하여 1000개 이미지 저장 :", new_X_train.shape) #y y_list = [] for i in range(10): y_list.append([i]*100) new_y_train = np.concatenate(y_list, axis=0) print("new_y_train :", new_y_train.shape) </pre> <p>숫자당 100개를 랜덤 추출하여 1000개 이미지 저장 : (1000, 784) new_y_train : (1000,)</p> <p>-new_X_train와 new_y_train 변수에 클래스 별로 100개씩 총 1000개 저장</p>
	<div>○ 숫자 2의 100 training image data 만들기</div>
	

```
#변수 저장
X_Digit_2 = X_list[2]
y_Digit_2 = y_list[2]

print(len(X_Digit_2))
print(len(y_Digit_2))
```

```
100
100
```

-X_Digit_2와 y_Digit_2 변수에 숫자 2 image data 100개 저장

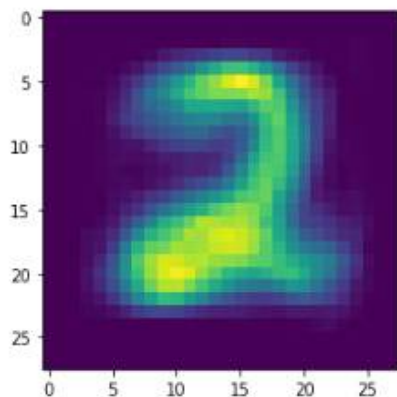
○ Plot the mean image for Digit-2-Space

```
#mean image
import numpy as np

mean_img_list = []
empty_list = []
mean = 0

for i in range(784):
    for k in range(100):
        empty_list.append(X_Digit_2[k][i])
        mean = np.mean(empty_list)
        mean_img_list.append(mean)
        empty_list = []

mean_img_list = np.reshape(mean_img_list,(28,28))
plt.imshow(mean_img_list)
plt.show()
```



1번

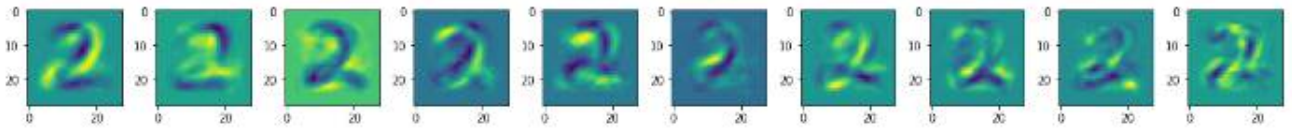
○ Plot the first 10 eigenvectors for Digit-2-Space(With PCA)

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
pca.fit(X_Digit_2)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    eigen_list = pca.components_[i]
    eigen_list = np.reshape(eigen_list,(28,28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')
|
fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors for Digit-2-Space(With PCA & 차원 복원)

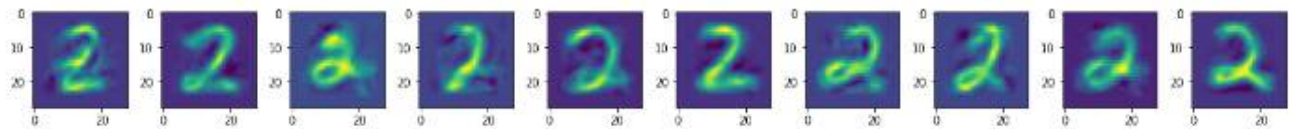
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_reduced = pca.fit_transform(X_Digit_2)
X_recovered = pca.inverse_transform(X_reduced)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    #eigen_list = pca.components_[i]
    eigen_list = np.reshape(X_recovered[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors for Digit-2-Space(With KernelPCA('linear') & 복원)

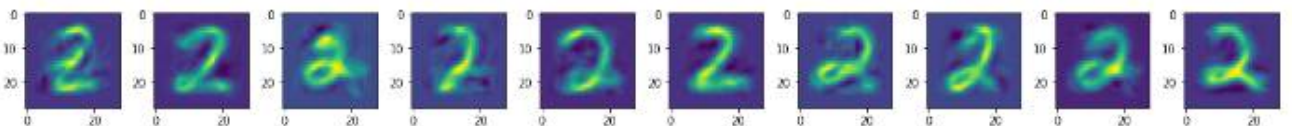
```
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=10, kernel='linear')
X_reduced_k = k_pca.fit_transform(X_Digit_2)
X_recovered_k = k_pca.inverse_transform(X_reduced_k)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    #eigen_list = k_pca.components_[i]
    eigen_list = np.reshape(X_recovered_k[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors for Digit-2-Space(With KernelPCA('rbf') & 복원)

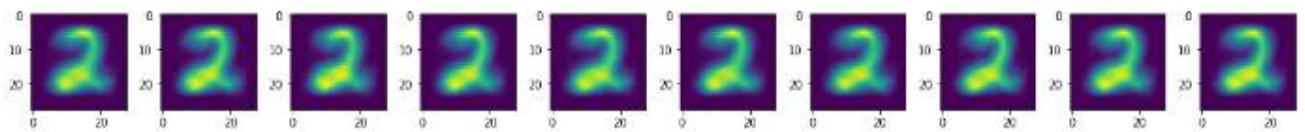
```
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=10, kernel='rbf')
X_reduced_k = k_pca.fit_transform(X_Digit_2)
X_recovered_k = k_pca.inverse_transform(X_reduced_k)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    #eigen_list = pca.components_[i]
    eigen_list = np.reshape(X_recovered_k[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



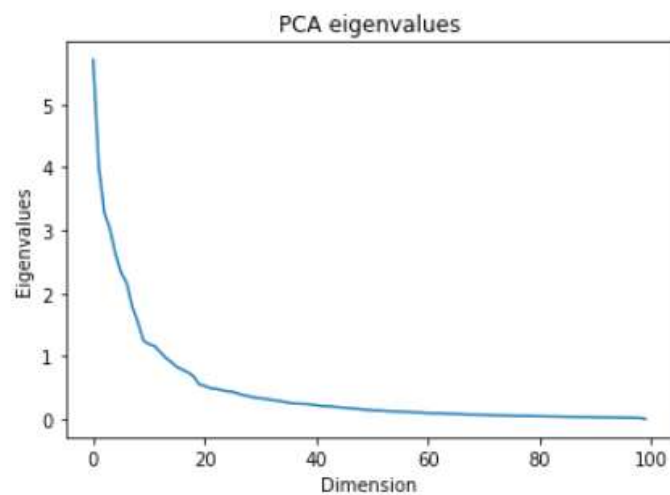
○ Plot the eigenvalues as a function of dimension (PCA)

```
#Plot the eigenvalues (in decreasing order) as a function of dimension.
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

pca = PCA(n_components=100)
pca.fit(X_Digit_2)

# 고유값 = Eigenvalue : 설명 정도
e_val = pca.explained_variance_
e_val.sort()
e_val=e_val[::-1]

plt.plot(e_val)
plt.title("PCA eigenvalues")
plt.xlabel('Dimension')
plt.ylabel('Eigenvalues')
plt.show()
```



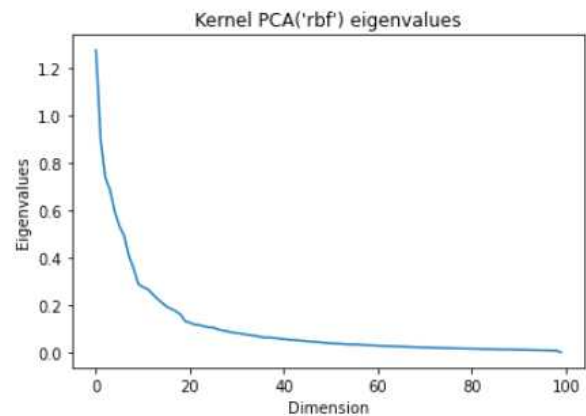
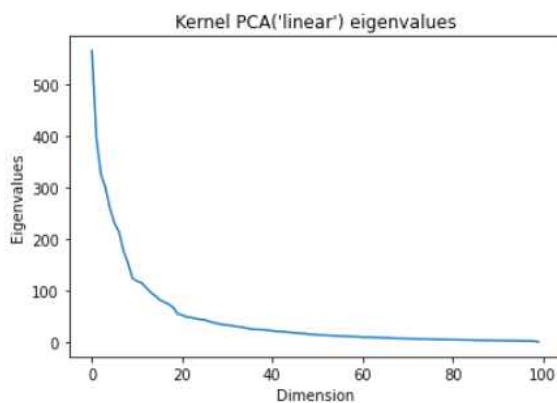
○ Plot the eigenvalues as a function of dimension (Kernel PCA)

```
#Plot the eigenvalues (in decreasing order) as a function of dimension.
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=100, kernel='linear')
k_pca.fit(X_Digit_2)

# 고유값 = Eigenvalue : 설명 정도
e_val = k_pca.eigenvalues_
e_val.sort()
e_val=e_val[::-1]

plt.plot(e_val)
plt.title("Kernel PCA('linear') eigenvalues")
plt.xlabel('Dimension')
plt.ylabel('Eigenvalues')
plt.show()
```



○ 결과 분석

- pca = PCA(n_components=10)으로 fit해서 10개의 고유벡터를 그려본 결과 눈으로 2라는 숫자는 확인할 수 있었지만 배경색과 섞여 다른 숫자도 같이 비교할 때, 쉽게 분석할 수 없었다.
- transform 함수를 활용하여 차원을 다시 복원시켜 주었다. Image를 plot해본 결과, mean image는 Kernel PCA('rbf')와 비슷했고, PCA와 Kernel PCA('linear')가 비슷한 결과를 도출했다. 글씨 자체는 PCA와 Kernel PCA('linear')가 더 구분하기 용이함을 확인하였다.
- eigenvalues를 내림차순으로 정렬하여 plot해본 결과, 비슷해 보이는 이미지와 달리 Eigenvalues값의 범위는 확연히 달랐다. 하지만 PCA, Kernel PCA('linear'), Kernel PCA('rbf') 모두 eigenvalues 값은 다르지만 ratio가 같아 그래프의 형태가 비슷했다. eigenvalue를 plot했을 때, 지수함수 형태의 극명한 값 차이를 확인할 수 있었다.

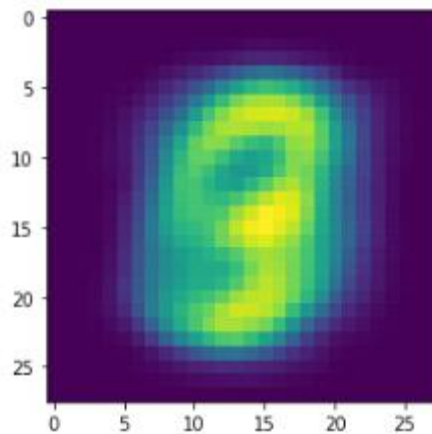
○ Plot the mean image for 1000 training images

```
#mean image
import numpy as np

mean_img_list = []
empty_list = []
mean = 0

for i in range(784):
    for k in range(1000):
        empty_list.append(new_X_train[k][i])
    mean = np.mean(empty_list)
    mean_img_list.append(mean)
    empty_list = []

mean_img_list = np.reshape(mean_img_list,(28,28))
plt.imshow(mean_img_list)
plt.show()
```

○ Plot the first 10 eigenvectors (With PCA)

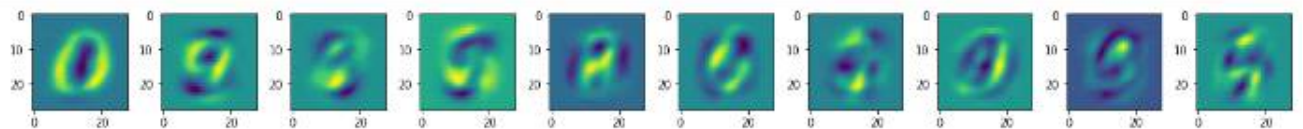
```
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
pca.fit(new_X_train)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    eigen_list = pca.components_[i]
    eigen_list = np.reshape(eigen_list, (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors (With PCA & 복원)

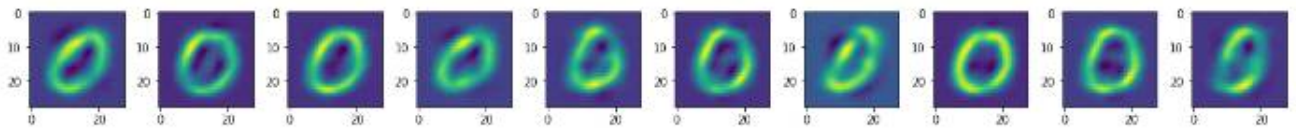
```
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_reduced = pca.fit_transform(new_X_train)
X_recovered = pca.inverse_transform(X_reduced)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    eigen_list = pca.components_[i]
    eigen_list = np.reshape(X_recovered[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors (With KernelPCA('linear')) & 복원

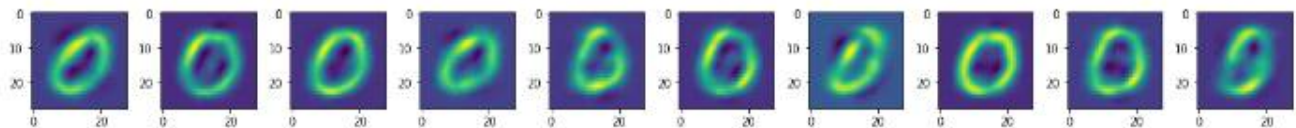
```
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=10, kernel='linear')
X_reduced_k = k_pca.fit_transform(new_X_train)
X_recovered_k = k_pca.inverse_transform(X_reduced_k)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    #eigen_list = pca.components_[i]
    eigen_list = np.reshape(X_recovered_k[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



○ Plot the first 10 eigenvectors (With KernelPCA('rbf')) & 복원

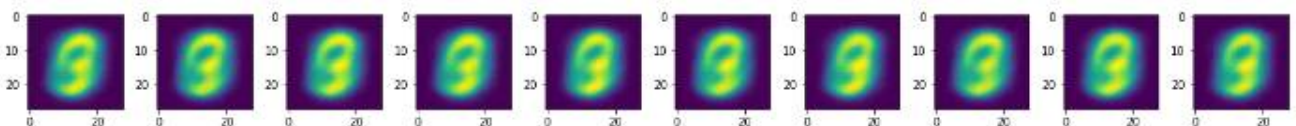
```
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=10, kernel='rbf')
X_reduced_k = k_pca.fit_transform(new_X_train)
X_recovered_k = k_pca.inverse_transform(X_reduced_k)

fig = plt.figure(figsize=(16, 16))
rows = 1; cols = 10

for i in range(10):
    #eigen_list = pca.components_[i]
    eigen_list = np.reshape(X_recovered_k[i], (28, 28))
    ax1 = fig.add_subplot(rows, cols, i+1)
    ax1.imshow(eigen_list, cmap='viridis')

fig.tight_layout()
plt.show()
```



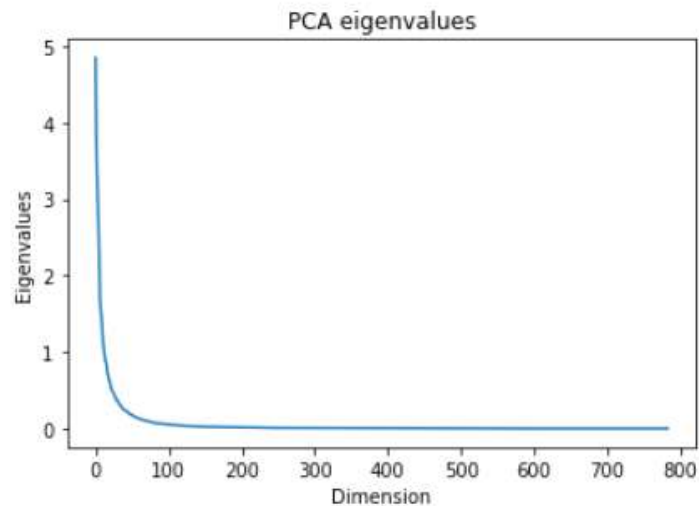
○ Plot the eigenvalues as a function of dimension (PCA)

```
#Plot the eigenvalues (in decreasing order) as a function of dimension.
from sklearn.decomposition import PCA

pca = PCA(n_components=784)
pca.fit(new_X_train)

# 고유값 = Eigenvalue : 설명 정도
e_val = pca.explained_variance_
e_val.sort()
e_val=e_val[::-1]

plt.plot(e_val)
plt.title("PCA eigenvalues")
plt.xlabel('Dimension')
plt.ylabel('Eigenvalues')
plt.show()
```



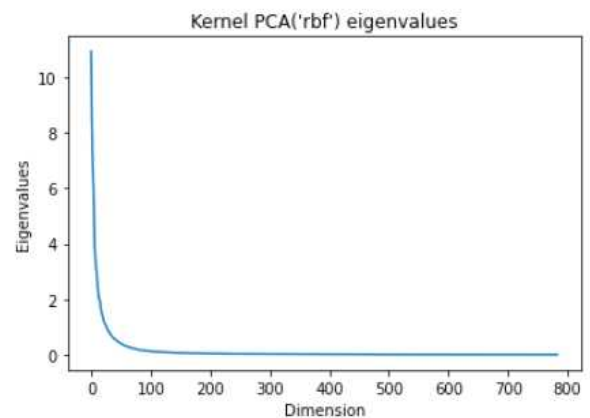
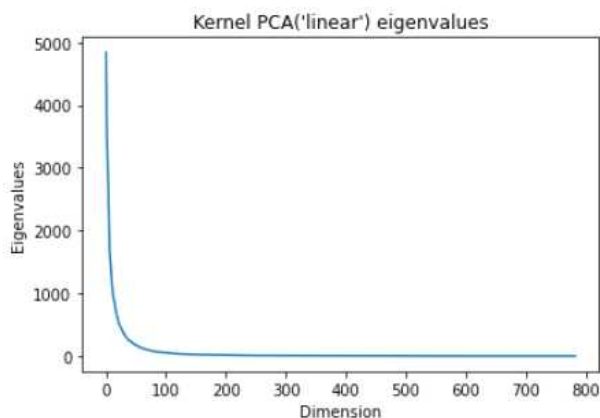
○ Plot the eigenvalues as a function of dimension (Kernel PCA)

```
#Plot the eigenvalues (in decreasing order) as a function of dimension.
from sklearn.decomposition import KernelPCA

k_pca = KernelPCA(n_components=784, kernel='linear')
k_pca.fit(new_X_train)

# 고유값 = Eigenvalue : 설명 정도
e_val = k_pca.eigenvalues_
e_val.sort()
e_val=e_val[::-1]

plt.plot(e_val)
plt.title("Kernel PCA('linear') eigenvalues")
plt.xlabel('Dimension')
plt.ylabel('Eigenvalues')
plt.show()
```



○ 결과 분석

- Image를 plot해본 결과, mean image는 Kernel PCA('rbf')와 비슷했고, PCA와 Kernel PCA('linear')가 비슷한 결과를 도출했다. 글씨 자체는 PCA와 Kernel PCA('linear')가 더 구분하기 용이함을 확인하였다. 0이라는 이미지가 쉽게 보였다. mean image는 Kernel PCA('rbf') 같은 경우 0과 3의 숫자가 겹쳐보이는 형태의 결과를 도출했다.

- eigenvalues를 내림차순으로 정렬하여 plot해본 결과, 비슷해 보이는 이미지와 달리 Eigenvalues값의 범위는 확연히 달랐다. 하지만 PCA, Kernel PCA('linear'), Kernel PCA('rbf') 모두 eigenvalues 값은 다르지만 ratio가 같아 그래프의 형태가 비슷했다. eigenvalue를 plot했을 때, eigenvalue값 끼리 극명한 값 차이를 확인할 수 있었다.

○ K-means clustering on the reduced features using the PCA

```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import rand_score
from sklearn.metrics.cluster import mutual_info_score

pca = PCA(n_components=10)
X_reduced = pca.fit_transform(new_X_train)

Kmeans = KMeans(n_clusters=10).fit_predict(X_reduced)

print("pca k-means clustering rand_score:", rand_score(new_y_train, Kmeans))
print("pca k-means clustering mutual_info_score:", mutual_info_score(new_y_train, Kmeans))

pca k-means clustering rand_score: 0.8862302302302302
pca k-means clustering mutual_info_score: 1.211053320889433
```

○ K-means clustering on the reduced features using the kernel PCA('linear')

```
k_pca = KernelPCA(n_components=10, kernel='linear')
X_reduced_k = k_pca.fit_transform(new_X_train)

Kmeans = KMeans(n_clusters=10).fit_predict(X_reduced_k)

print("k-pca('linear') k-means clustering rand_score:", rand_score(new_y_train, Kmeans))
print("k-pca('linear') k-means clustering mutual_info_score:", mutual_info_score(new_y_train, Kmeans))

k-pca('linear') k-means clustering rand_score: 0.8885745745745746
k-pca('linear') k-means clustering mutual_info_score: 1.2068145063458482
```

○ K-means clustering on the reduced features using the kernel PCA('rbf')

```
k_pca = KernelPCA(n_components=10, kernel='rbf')
X_reduced_k = k_pca.fit_transform(new_X_train)

Kmeans = KMeans(n_clusters=10).fit_predict(X_reduced_k)

print("k-pca('rbf')k-means clustering rand_score:", rand_score(new_y_train, Kmeans))
print("k-pca('rbf')k-means clustering mutual_info_score:", mutual_info_score(new_y_train, Kmeans))

k-pca('rbf')k-means clustering rand_score: 0.8787987987987989
k-pca('rbf')k-means clustering mutual_info_score: 1.1229970393757178
```

○ 결과 분석

- Rand index는 분류모형의 성능평가 지표인 정분류율(accuracy)과 비슷하다. Rand index는 Kernel

PCA("linear")에서 약 0.889정도로 가장 높게 나왔다. 하지만 가장 낮은 점수가 0.879이며, Rand index에서 PCA와 Kernel PCA의 차이가 크지 않았다.

- mutual_info_score 명령은 각 데이터에 대해서 X,Y 카테고리 값을 표시한 2차원 배열을 입력해야 한다. 사이킷런 패키지의 metrics 서브패키지는 이산 확률 변수의 상호정보량을 구하는 mutual_info_score 명령을 제공한다. mutual_info_score는 PCA에서 약 1.211정도로 가장 높게 나왔다. 앞선 1번과 2번의 결과에서 PCA와 Kernel PCA("linear")의 plot 결과가 비슷했고, mutual_info_score 또한 비슷하게 나왔음을 확인할 수 있다.

○ (과제3)1-NN classifier에 test data set을 적용하여 predict한 결과

```
k = Kmeans.reshape(10,-1)
Kmeans_X_means = []

for i in range(10):
    cnt = Counter(k[i])
    Kmeans_X_means.append([cnt.most_common(1)[0][0]])
    |
print(Kmeans_X_means)
#print(k)
#print(new_y_train)

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

#agg_X_means.reshape(1, -1)
knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(kmeans_centers, y_means)
knn_test = knn.predict(X_test)
#print(knn_test.shape)
score = classification_report(knn_test, y_test)
print(score)
```

```
[[9], [8], [0], [5], [7], [5], [1], [2], [6], [7]]
      precision    recall  f1-score   support

0         0.04        0.08        0.05         512
1         0.00        0.00        0.00        1010
2         0.02        0.01        0.01        1199
3         0.01        0.01        0.01        1619
4         0.00        0.00        0.00        1300
5         0.00        0.00        0.00         801
6         0.01        0.01        0.01         788
7         0.00        0.00        0.00         732
8         0.12        0.13        0.12         955
9         0.02        0.01        0.02        1084

 accuracy          0.02
 macro avg         0.02
 weighted avg      0.02
```

- 과제3에서 kmeans_centers를 1-nn에 fit하고, (10000, 784)의 shape을 가지는 test data set을 prediction하였다. 그 결과를 test data set label(이미지의 answer)와 비교한 결과, accuracy는 0.02였다.

○ 1-NN classifier (With PCA)

```
#pca 1-nn score

pca = PCA(n_components=10)
X_reduced = pca.fit_transform(new_X_train)
X_reduced_test = pca.fit_transform(X_test)

knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_reduced, new_y_train)
knn_test = knn.predict(X_reduced_test)
score = classification_report(knn_test, y_test)
print(score)
```

	precision	recall	f1-score	support
0	0.76	0.72	0.74	1036
1	0.16	0.53	0.24	340
2	0.27	0.27	0.27	1029
3	0.08	0.08	0.08	1014
4	0.03	0.03	0.03	959
5	0.48	0.37	0.41	1162
6	0.58	0.61	0.59	904
7	0.24	0.25	0.24	993
8	0.27	0.17	0.21	1559
9	0.02	0.02	0.02	1004
accuracy			0.28	10000
macro avg	0.29	0.30	0.28	10000
weighted avg	0.30	0.28	0.28	10000

○ 1-NN classifier (With kernel PCA('rbf'))

```
#k-pca 1-nn score

k_pca = KernelPCA(n_components=10, kernel='rbf')
X_reduced_k = k_pca.fit_transform(new_X_train)
X_reduced_k_test = k_pca.fit_transform(X_test)

knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_reduced_k, new_y_train)
knn_test = knn.predict(X_reduced_k_test)
score = classification_report(knn_test, y_test)
print(score)
```

	precision	recall	f1-score	support
0	0.70	0.58	0.63	1192
1	0.02	0.21	0.04	111
2	0.07	0.05	0.05	1489
3	0.07	0.08	0.08	952
4	0.01	0.01	0.01	722
5	0.32	0.27	0.29	1057
6	0.25	0.27	0.26	894
7	0.16	0.16	0.16	1061
8	0.23	0.16	0.19	1383
9	0.00	0.00	0.00	1139
accuracy			0.18	10000
macro avg	0.18	0.18	0.17	10000
weighted avg	0.21	0.18	0.19	10000

○ 결과 분석

- PCA의 정확도는 0.28이고, Kernel PCA("rbf")의 정확도는 0.18이었다. 과제3에서 나왔던 accuracy는 0.02였던 것에 비교했을 때, 정확도가 상당히 올라감을 확인할 수 있었다.

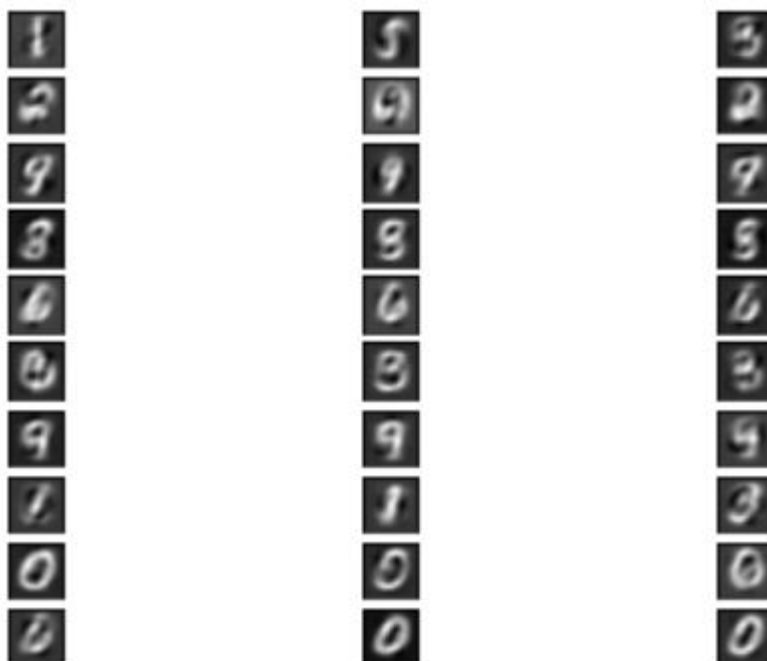
- Eigenvector가 10개를 가지는 PCA와 Kernel PCA("rbf")에 fit하여 test data를 적용했을 때, 서로 차원이 맞지 않는 문제가 발생했다. PCA는 차원을 줄이기 때문에 Test data 또한 이에 맞춰 코드를 짜서 정확도를 추출했다. 이러한 과정에서 정확도가 많이 낮아짐을 확인하였다. 가능하다면 10차원으로 줄여진 차원을 다시 784차원으로 늘리고 서로 비교하면 정확도가 올라갈 것 같다.

○ Visualize 3 correctly classified images



5번

○ Visualize 3 incorrectly classified images



○ 결과 분석

- 교수님이 첨부해주신 블로그 코드에서 viz_img함수를 활용하여 코드를 짰다.
- Mnist 데이터에는 총 10개의 손글씨 숫자로 이루어져 있다. 그중에서도 5번을 통해 패턴들을 파악해볼 수 있었다. 숫자를 쓰는 과정에서 일직선이 많으면, 1이나 7과 같이 분류되었다. 예를 들면, 숫자 6을 동그란 밑부분을 강조해서 쓰면 0이나 6으로 분류되는 반면, 6에서 윗부분 직선구간을 길게 쓰면, 1이나 7로 분류되는 것을 확인하였다.
- 또한 숫자 9에서 밑부분을 말아올리지 않고 일직선으로 적는데 이러한 형태가 숫자 7과 비슷한 패턴을 가져 9와 7을 같은 class로 분류하기도 하였다.
- 차원 축소 과정을 거치면 이미지의 해상도가 일부 낮아지면서 구분이 쉽지 않지만, eigenvector를 추출함으로써 숫자 class 간의 패턴을 파악할 수 있었다.