앞 단원에서 공부했던 공분산행렬, 고유값분해, 특이값 분해 등은 데이터차원을 효율적으로 줄일 수 있는 방법론에서도 적용되고 있다. 본 단원에서는 데이터의 차원을 줄여서 분석을 용이하게 만들어 주는 비지도학습의 하나인 주성분분석(PCA: Principal Component Analysis)에 대해서 알아보도록 하자.

참고로 여기서 말하는 비지도학습(unsupervised learning)이란, 특별한 정답지가 없는 상황에서 변수들끼리의 관계만을 분석하여 어떤 정보들이 데이터에 담겨 있는 지를 알아내는 방법론을 의미한다. 한편, 샘플링을 통해서 xi일 때, yi라는 정답을 이미 알려주고, xj일 때는 yj는 무엇인지를 알아내는 방법론을 지도학습(supervised learning)이라고 부른다.

문제1. 2차원의 좌표평면에 4개의 점 (3,4), (2,8), (6,9), (10,12)가 놓여 있다. 공간에 임의의 점 단위 벡터[1] e=(x,y)가 놓여 있을 때, 4개의 점을 e와의 내적(inner product)하여 이 값들을 C라고 하자. C는 4개의 값으로 되어 있는데, 이 값들의 평균과 분산을 구하고, 분산을 최대로 하는 x,y값을 구해 보아라.

위 4개의 점을 먼저 'sympy'를 이용해 행렬 모양으로 정리하고, C를 표현해 보면 아래와 같다.

```
from sympy import Matrix
D=Matrix(4,2,[3,4,2,8,6,9,10,12])
D
```

$$\begin{bmatrix} 3 & 4 \\ 2 & 8 \\ 6 & 9 \\ 10 & 12 \end{bmatrix}$$

```
from sympy import symbols
x,y=symbols('x y',real=True) # 실수 문자 x,y정의
C=D@Matrix(2,1,[x,y])
C
```

$$\begin{bmatrix} 3x+4y \\ 2x+8y \\ 6x+9y \\ 10x+12y \end{bmatrix}$$

벡터 C의 평균과 분산을 구하면 아래와 같다. 아래에서 벡터 C의 분산을 L이라고 정의해 두었다.

```
import numpy as np
np.mean(C) # C의 평균출력
```

$$\frac{21x}{4}+\frac{33y}{4}$$

```
# C의 분산을 계산하고 간단히 하기
L=np.var(C).simplify()
L
```

$$\frac{155x^2}{16}+\frac{115xy}{8}+\frac{131y^2}{16}$$

```
# 맞는지 다른 방법으로 분산을 구해보면...
E=np.mean(C) # 원래 평균
C2=C.applyfunc(lambda x:x**2) # 각 원소의 제곱
E2=np.mean(C2)
E2-E**2 # 분산구하기
```

$$\frac{(2x+8y)^2}{4}+\frac{(3x+4y)^2}{4}-\left(\frac{21x}{4}+\frac{33y}{4}\right)^2+\frac{(6x+9y)^2}{4}+\frac{(10x+12y)^2}{4}$$

```
_.simplify() # 위 셀의 결과를 간단히..
```

$$\frac{155x^2}{16}+\frac{115xy}{8}+\frac{131y^2}{16}$$

위에서 정의된 'sympy' 수식을 일반 함수로 전환하려면 아래와 같이 'lambdify()'를 사용하면 된다.

---

[1] 벡터의 크기(L2 norm)가 1인 벡터를 단위벡터라고 한다.

```
from sympy import lambdify
ff=lambdify((x,y),L,'numpy')
ff(1,0)
```

```
9.6875
```

'numpy' 함수로 정의된 ff()는 벡터C의 분산 L을 표현하고 있으며, ff()함수의 최대값을 구하기 위해서는 제약식이 존재하는 비선형함수의 최소값을 구하는 문제(NLP:Non-Linear Programming)를 풀어야 한다[2].

```
from scipy.optimize import minimize # 비선형함수의 최소값을 구하는 함수

def f(x): # 목적함수 정의
    return -ff(x[0],x[1]) #최대값을 구해야 하기 때문에 -ff()로 변환

def eqs(x): # 제약식 정의
    return x[0]**2+x[1]**2-1 # 단위벡터임을 표현

# 제약식이 방정식(eq)형태이며, 어떤 함수로 되어 있는지 정의
cond={'type':'eq','fun':eqs}

# 함수의 최소값을 구하여 res에 저장
res=minimize(f,[-1,0.5],constraints=cond)
res # 결과값 출력
```

```
message: Optimization terminated successfully
success: True
 status: 0
    fun: -16.16402477093814
      x: [-7.429e-01 -6.694e-01]
    nit: 10
    jac: [ 2.402e+01  2.164e+01]
   nfev: 31
   njev: 10
```

```
# 해가 제약식을 만족하는지 확인
res.x[0]**2+res.x[1]**2
```

```
1.000000017247445
```

```
# 최대값 구하기
ff(res.x[0],res.x[1])
```

```
16.16402477093814
```

조금 어려운 내용이긴 하지만, 위 문제를 수학적인 기호를 사용하여 다시 표현해 보면 전형적인 QP(Quadratic Programming) 문제가 된다. QP 문제는 대부분 수학적으로 깔끔한 해를 도출해 주는 경우가 많다.

$$\underset{m\times n}{X} = \left[\begin{pmatrix} X_1 \end{pmatrix} \quad \begin{pmatrix} X_2 \end{pmatrix} \quad \cdots \quad \begin{pmatrix} X_n \end{pmatrix}\right],\ \bar{X}_i\text{: } \begin{pmatrix} X_i \end{pmatrix}\text{의 평균}$$

$$X - \bar{X} = \left[\begin{pmatrix} X_1 - \overline{X_1} \end{pmatrix} \quad \begin{pmatrix} X_2 - \overline{X_2} \end{pmatrix} \quad \cdots \quad \begin{pmatrix} X_n - \overline{X_n} \end{pmatrix}\right]$$

$$C \equiv Xe$$

$$\mathrm{var}(C) = \frac{1}{n}\sum\ (Xe - \bar{X}e)^2 = \frac{1}{n}\sum\ \left((X - \bar{X})e\right)^2 = \frac{1}{n}e^T(X - \bar{X})^T(X - \bar{X})e$$

$$D_1 \equiv \frac{1}{n}(X - \bar{X})^T(X - \bar{X}) \ \rightarrow\ \mathrm{var}(C) = e^T D_1 e$$

[2] 부록의 최적화(optimization) 참조

$$\max \quad e^T D_1 e \qquad \text{s.t.} \quad \|e\| = 1$$

⇨ QP 문제

$L(e) = e^T D_1 e - \lambda([e]^2 - 1)$: Lagrange 함수

$$\frac{\partial L}{\partial \lambda} = [e]^2 - 1 = 0 \ , \frac{\partial L}{\partial \lambda} = 2D_1 e - 2\lambda e = 0$$

$\therefore D_1 e = \lambda e$=0 → e는 D_1의 고유벡터

앞에서 'scipy'의 최적화 해는 일종의 수치해석적인 해로써 컴퓨터의 반복적인 해찾기 알고리즘을 통해서 구한 것이다. 위의 QP 문제로 보면 최적해는 결국 D_1행렬의 고유벡터가 됨을 쉽게 알 수 있는데, 원래 문제의 행렬을 D_1문제로 바꾸어서 고유벡터를 구해보면 아래와 같다. 위 문제에서 D_1의 고유값 분해는 원래 행렬 X 입장에서 볼 때는 특이값분해(SVD)와 유사하다[3]. 아래 결과를 잘 보면, 첫번째 고유벡터값을 목적함수 값에 대입했을 때, 최대값은 해당 고유벡터의 고유값(eigenvalue)과 같음을 확인할 수 있다.

```
# 칼럼 평균을 구해서 쉽게 계산하도록 데이터프레임으로 변경
import pandas as pd
df=pd.DataFrame(data=np.array(D))
df
```

	0	1
0	3	4
1	2	8
2	6	9
3	10	12

```
# 칼럼평균을 구해서 각 칼럼에서 평균치를 뺀 후
# 다시 numpy 어레이로 변경
D1=np.array(df-df.mean(axis=0)).astype(float)
```

```
D1
```

```
array([[-2.25, -4.25],
       [-3.25, -0.25],
       [ 0.75,  0.75],
       [ 4.75,  3.75]])
```

```
C0=D1.T@D1/4
D,P=np.linalg.eig(C0)
D
```

```
array([16.16402449,  1.71097551])
```

```
P
```

```
array([[ 0.74289445, -0.66940857],
       [ 0.66940857,  0.74289445]])
```

```
P.T[0] # 첫번째 고유벡터
```

```
array([0.74289445, 0.66940857])
```

```
ff(_[0],_[1])
```

```
16.164024493143298
```

```
P.T[1] # 두번째 고유벡터
```

```
array([-0.66940857,  0.74289445])
```

```
ff(_[0],_[1])
```

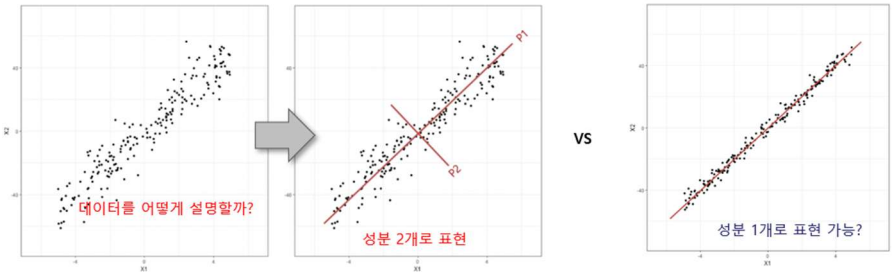
```
1.7109755068567027
```

위에서 분산을 최대로 하는 e를 'scipy'를 통해서도 찾아보았고, QP 포뮬레이션을 통해 고유벡터가 이에 해당한다는 사실까지 확인해 보았다. 그렇다면, 이러한 고유벡터가 가지는 의미는 무엇일까? X 데이터의 분산을 최대로 만드는 e 벡터는 사실 X 행렬에서 열(column)의 조합을 의미[4]한다. 예를 들어, 아래와 같이 무작위로 흩어져 있는 데이터를 좌표축 변환을 하여 P1, P2 축으로 바꾼다고 했을 때, P1은 원래 데이터의 분산을 최대로 하는 축이 되며, 자료를 가장 잘 설명한다고 볼

[3] X의 평균값 벡터가 0인 경우라고 하면 동일하다.

[4] A행렬의 왼쪽에 곱하는 벡터는 A행렬의 행(row)연산을 하게 되며, A행렬의 오른쪽에 곱하는 벡터는 A행렬의 열(column)연산을 하게 된다.

수 있다. 원래 데이터의 모든 정보를 다 표현하려면, P1, P2 모두가 필요하겠지만, 경우에 따라서는 P1하나만으로 데이터의 차원을 축소하겠다는 것이다. 이를 주성분분석(PCA; Principal Component Analysis)라고 부른다.



일반적으로 자료 X가 커다란 사각형 모양이라고 해 보도록 하자. 보통 자료의 행(row)은 관측치의 순번이 되고, 열(column)은 각 관측치에서 관찰된 여러가지 특성치들(features)로 볼 수 있다. 각 관측치에서 수없이 많은 특성치들이 있을 경우, PCA를 통해 특성치들의 특징을 가장 잘 표현할 수 있는 선형조합을 만들어서 feature의 수를 원하는 대로 줄일 수 있다는 것이 PCA 분석의 결과라고 볼 수 있다.

아래는 라이브러리 'scikit-learn' [5] 에 있는 PCA tool을 사용해 분석해 본 것이다. 매개변수 n_components=1로 하면 위 문제에서의 해와 동일함을 확인할 수 있다.

```
from sklearn.decomposition import PCA
# n_components=1로 하면 분산이 가장 큰 조합 하나만 출력
pca=PCA(n_components=1) # 객체 설정
pca.fit(D)
pca.components_ # 고유벡터 확인
```

```
array([[0.74289445, 0.66940857]])
```

```
pca=PCA(n_components=2) # 객체 설정
pca.fit(D)
pca.components_ # 고유벡터 확인
```

```
array([[ 0.74289445,  0.66940857],
       [-0.66940857,  0.74289445]])
```

특히, PCA에 있는 fit_tranform()이나 transform()을 활용하면, 원 좌표를 PCA에 의해 변환 조합까지 완료해 준다. 여기서 주의할 부분은 원 좌표들의 칼럼 평균을 뺀 후의 데이터를 변환해 주게 된다는 것이다. 다시한번, 예제를 통해서 살펴보면, 아래 D1은 D행렬에서 열평균을 빼서 새로이 만든 행렬이며, D의 PCA결과와 D1의 PCA결과는 모두 동일하다.

```
D=Matrix(4,2,[3,4,2,8,6,9,10,12])
D=np.array(D).astype(int)
D
```

```
array([[ 3,  4],
       [ 2,  8],
       [ 6,  9],
       [10, 12]])
```

```
df=pd.DataFrame(data=np.array(D))
df=df-df.mean()
D1=np.array(df)
D1 # D의 칼럼평균을 뺀 행렬
```

```
array([[-2.25, -4.25],
       [-3.25, -0.25],
       [ 0.75,  0.75],
       [ 4.75,  3.75]])
```

```
from sklearn.decomposition import PCA
# n_components=1로 하면 분산이 가장 큰 조합 하나만 출력
pca=PCA(n_components=1) # 객체 설정
pca.fit(D)
pca.components_ # 고유벡터 확인
```

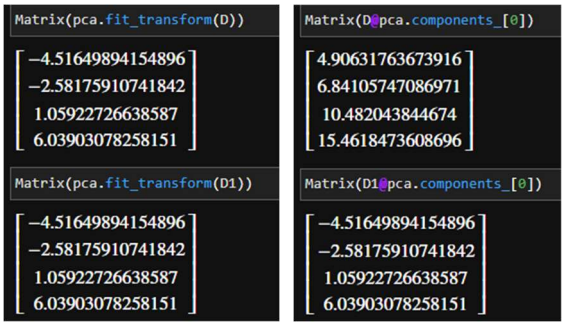
```
array([[0.74289445, 0.66940857]])
```

```
pca=PCA(n_components=1) # 객체 설정
pca.fit(D1)
pca.components_ # 고유벡터 확인
```

```
array([[0.74289445, 0.66940857]])
```

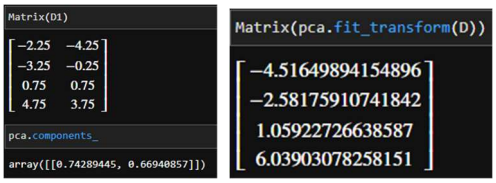
[5] conda install -c conda-forge scikit-learn 과 같이 설치하고, 약칭으로 sklearn으로 불러 온다.

좌표변환 작업을 위해 fit_transform()을 적용한 후의 값을 비교해 보면, D와 D1의 결과가 모두 같다는 것을 확인할 수 있다.



결국 PCA의 tool은 주어진 자료를 칼럼 평균이 0이 되도록 만든 후, 이를 바탕으로 PCA 분석을 하고, 이 값을 바탕으로 좌표변환을 한다고 기억하면 되겠다.

다시한번 정리해서 말하면, 아래 D1을 PCA(n_components=1)로 정리하게 되면, 첫번째 feature(첫번째 열)과 두번째 feature (두번째 열)을 pca.components_ 결과로 선형결합하여 새로 만든 열이 바로 fit_transform()의 결과물이며, 결과적으로 PCA를 통해 feature가 2개인 자료를 feature가 1개인 자료로 줄이게 된 것이다.



문제2. 파이썬에서는 AI와 관련된 여러가지 라이브러리들이 있는데, 일단 가장 쉽게 배울 수 있는 라이브러리 중의 하나가 scikit-learn이라는 패키지이다. 이 패키지는 머신러닝과 데이터분석에 관련된 여러가지 분석 도구뿐만 아니라, 학습을 할 수 있는 다양한 학습용 데이터들도 같이 들어있다. scikit-learn의 학습데이터 제공함수 중 load_iris()는 붓꽃관련 데이터셋을 불러오는 메서드이다. 붓꽃 데이터셋은 3가지 붓꽃 품종(setosa, versicolor, virginica)에 따라 꽃받침(sepal)의 길이와 너비, 꽃잎(petal)의 길이와 너비 데이터를 모아 두었다. 이 자료를 데이터프레임으로 불러와서 데이터의 총개수와 붓꽃 품종에 따라 몇 개의 개별 데이터들이 있는지 알아보아라.

먼저 load_iris를 실행하여 IR에 저장하고, IR 데이터 형식을 보면 아래와 같이 'sklearn'만의 데이터 유형을 가지고 있는 것 같아 보인다. 하지만, IR 데이터를 출력해 보면, 파이썬의 사전형(dictionary)으로 되어 있음을 확인할 수 있고, 각 key 값에 어떤 항목의 데이터들이 들어가 있는지 확인할 수 있다.

```
# 붓꽃 데이터 읽어들이기
from sklearn.datasets import load_iris
import numpy as np

IR=load_iris() # 데이터 읽어오기

type(IR) # IR데이터형식 확인

sklearn.utils._bunch.Bunch
```

```
IR

{'data': array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
```

IR['data']에는 각 붓꽃의 꽃받침(sepal)의 길이와 너비, 꽃잎(petal)의 길이와 너비 데이터들이 들어 있으며, IR['feature_names']에는 'data'칼럼 이름이 들어 있다. 이 데이터를 바탕으로 데이터프레임으로 읽어오면 아래와 같으며, 데이터의 총 개수는 150개임을 알 수 있다.

```
import pandas as pd
df=pd.DataFrame(data=IR['data'],columns=IR['feature_names'])
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

```
# 데이터 기본 통계량 보기
df.describe()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

IR['target']의 데이터를 데이터프레임으로 변경하고 value_counts()를 이용하여 항목 개수들의 분포를 보면 각 품종별로 50개씩임을 확인할 수 있다. 웹사이트에서 관련 정보를 검색해 보면, target의 0,1,2는 차례대로 setosa, versicolor, virginica의 품종을 대표한다고 한다.

```
# target에는 붓꽃의 품종명이 들어 있음.
df_Y=pd.DataFrame(IR['target'])
df_Y.value_counts() # 항목별로 몇 개씩 있는지 출력해줌

0    50
1    50
2    50
dtype: int64
```

문제3. 위 예제에서 만든 df에는 품종별로 꽃받침, 꽃잎의 특징(features) 데이터들이 저장되어 있다. 150행에 4개의 칼럼 데이터이다. 4개의 칼럼 데이터를 2개로 차원을 축소시켜 보아라.

'sklearn'의 PCA tool을 이용하면 쉽게 그 결과를 확인해 볼 수 있다.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # 4개의 feature를 2개로 줄이고자 하는 설정
pca.fit(df) # PCA 분석 실행
iris_pca = pca.transform(df) # 분석된 결과를 바탕으로 자료 변환수행
```

```
pca.components_ # 고유벡터 확인
```

```
array([[ 0.36138659, -0.08452251,  0.85667061,  0.3582892 ],
       [ 0.65658877,  0.73016143, -0.17337266, -0.07548102]])
```

```
iris_pca.shape # 변환된 결과 확인
```

```
(150, 2)
```

분석결과의 고유벡터 2개는 각 feature를 조합하여 새로운 feature로 만드는 가중치 벡터로 해석하면 된다. 위 분석에서 만들어진 새로운 데이터 iris_pca의 결과를 확인해 보도록 하자. 아래 df1은 칼럼 평균을 빼서 만들어진 새로운 데이터프레임이고, PCA분석의 결과로 만들어진 고유벡터로 본래 자료를 선형 재조합하면 아래와 같은 결과가 된다.

```
# 칼럼 평균을 빼서 새로운 데이터프레임으로
df1=df-df.mean()
```

```
# 첫행의 feature를 첫번째 고유벡터로 재조합
df1.iloc[0,:]@pca.components_[0]
```

```
-2.6841256259695365
```

```
# 첫행의 feature를 두번째 고유벡터로 재조합
df1.iloc[0,:]@pca.components_[1]
```

```
0.31939724658510055
```

위 분석의 결과값을 iris_pca[0]과 같은지 확인해 보도록 하자.

```
iris_pca[0]
```

```
array([-2.68412563,  0.31939725])
```

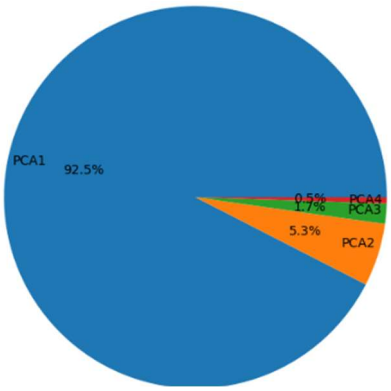
데이터의 손실 없이 PCA를 수행하려면 원래 feature의 수만큼 분석하면 된다.

```
# 모든 분산이 보존되도록 할 경우
pca1 = PCA(n_components=4) # 4개의 feature를 4개 그대로
pca1.fit(df) # PCA 분석 실행
pca1.explained_variance_ # 고유값 출력
```

```
array([4.22824171, 0.24267075, 0.0782095 , 0.02383509])
```

위 결과값을 원그래프를 이용하여 그려보면 아래와 같다. 첫번째 고유값이 차지하는 비율이 92.5%로 주성분 1개만으로도 자료 분산의 92.5%를 설명한다고 볼 수 있다.

```python
import matplotlib.pyplot as plt

labels=[f'PCA{i}' for i in range(1,5)]
plt.pie(pca1.explained_variance_,
        labels=labels,labeldistance=0.8,autopct='%1.1f%%')

# 차트를 정사각형으로 만들기
plt.axis('equal')

# 레이아웃 자동 조절
plt.tight_layout()

# 플롯 보여주기
plt.show()
```



'explained_variance_ratio_'를 사용하면 각 고유값이 차지하는 비율을 바로 출력할 수도 있으니 참고하자.

```python
# 비율로 표시
pca1.explained_variance_ratio_
```

```
array([0.92461872, 0.05306648, 0.01710261, 0.00521218])
```

결과적으로 2개의 주성분만으로 분석하게 될 경우, feature를 4개에서 2개로 줄이게 되면서 분산의 약 98% (=92.46%+5.3%)를 설명하게 되었다는 것으로 결론 내릴 수 있다.

문제4. SVD에서 사용했던 자유의 여신상(Statue of liberty) 사진을 사진을 바탕으로 PCA 분석을 수행해 보아라.

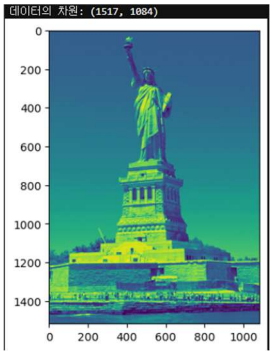
디렉토리에 저장된 jpg 파일을 읽어서 'numpy' array로 변환시켜 보면 아래와 같은 3차원 텐서(tensor) 형태를 확인할 수 있다.

```python
# 아래 파일 이름앞에 'r'을 붙이면 escape 문자(\,\n 등)를 해석하지 않게 됨
file=r'C:\Temp\liberty.jpg'
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
image=Image.open(file) # 파일열기
# 이미지를 NumPy 배열로 변환
image_array = np.array(image)
image_array.shape
```

```
(1517, 1084, 3)
```

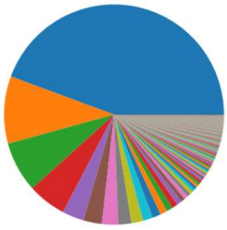
3차원 텐서는 바로 PCA 분석을 할 수가 없다. 세 번째 차원에 있는 정보가 (R,G,B)의 색 값의 채널을 나타내 주기 때문에, 이 중 하나를 선택하여 분석하면 된다. R채널만 추출하여 먼저 이미지를 출력해 보자.

```
# R 채널 분리
red_channel = np.array(image.getdata(band=0),int)

# numpy 어레이는 이미지 라이브러리와 행,열 역전
red_channel.shape=(image.size[1],image.size[0])

print('데이터의 차원:',red_channel.shape)
plt.imshow(red_channel)
plt.show() # 이미지만 출력
```



PCA 분석을 수행하면, 열에 있는 1084 개의 자료를 줄여볼 수가 있다. 몇 개 정도면 90% 가량의 분산을 설명할 수 있는지 찾아보도록 하자.

```
# PCA 분석
pca_red = PCA(n_components=1084)
red_pca = pca_red.fit_transform(red_channel)

# 고유값의 비율 출력
vars=pca_red.explained_variance_ratio_
plt.pie(vars) # 원그래프로 표현
plt.show()
```



원 그래프를 보면 대략 20개의 고유값이면 충분할 것으로 판단된다. 실제 수치를 합산해 보면 아래와 같이 확인해 볼 수 있다.

```
# 처음 20개 고유값의 비율 계산
np.sum(vars[:20])
```

```
0.881305553719532
```

PCA 분석의 결과 중 하나인 red_pca는 red_channel의 데이터를 PCA 변환시킨 데이터이다. 즉, 원래 데이터들을 선형 재조합한 결과라는 뜻이다. 실제로 plt.imshow(red_pca) 로 이미지를 출력해 보면 우리가 원하는 결과를 얻을 수가 없다. 이미지 데이터 압축 후의 결과를 제대로 그려 보기 위해서는 원래 데이터의 역변환 작업이 필요하다.

아래는 (4,2) 행렬의 PCA 분석을 하여 원본 데이터의 손실없이 변환시킨 후, inverse_transform()을 사용하여 원래 데이터가 복원되는지 확인해 본 것이다.

```
from sympy import Matrix
import numpy as np
A=np.array([[-2.25, -4.25],
       [-3.25, -0.25],
       [ 0.75,  0.75],
       [ 4.75,  3.75]])
```

```
Matrix(A)
```

$$\begin{bmatrix} -2.25 & -4.25 \\ -3.25 & -0.25 \\ 0.75 & 0.75 \\ 4.75 & 3.75 \end{bmatrix}$$

```
# 역변환 사용해 보기
pca=PCA(n_components=2) # 객체 설정
out=pca.fit_transform(A)
Matrix(pca.inverse_transform(out))
```

$$\begin{bmatrix} -2.25 & -4.25 \\ -3.25 & -0.250000000000001 \\ 0.75 & 0.75 \\ 4.75 & 3.75 \end{bmatrix}$$

만일, feature 수를 1개로 줄이고 역변환을 시키면 어떤 데이터가 나올까? 아래 데이터의 결과를 살펴보면 원래 A 행렬과는 엇비슷하지만 다른 데이터로 보인다.

```
# 역변환 사용해 보기
pca1=PCA(n_components=1) # feature의 개수가 줄어든다면...
out1=pca1.fit_transform(A)
Matrix(pca1.inverse_transform(out1))
```

$$\begin{bmatrix} -3.35528200005305 & -3.02338310326907 \\ -1.9179745138218 & -1.72825167526838 \\ 0.786894058177515 & 0.709055810962867 \\ 4.48636245569734 & 4.04257896757459 \end{bmatrix}$$

압축된 정보로 복원하여 만든 행렬을 A1이라고 하고, A와 A1의 SVD 분석 결과를 비교해 보면 아래와 같다. 두 행렬의 제1 주성분은 동일한 값이 되도록 보존되었음을 확인할 수 있다. 주성분의 역변환을 하여 원래 크기의 데이터를 복원하게 되면 데이터의 압축이라는 측면에서는 의미가 없어지지만, 주성분 분석 결과의 특이값을 보존하면서도 데이터의 분산을 줄인 데이터의 재구축이라고 보면 될 것 같다.

```
u,s,vt=np.linalg.svd(A) # 원래 행렬 A의 SVD
u,s,vt
```

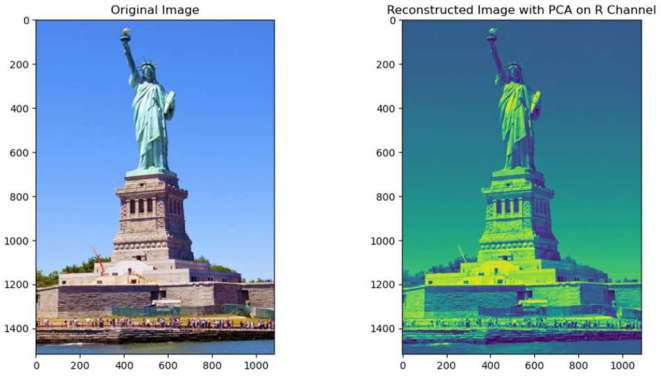
```
(array([[-0.56169061,  0.63114615,  0.11362528,  0.5227308 ],
        [-0.32107831, -0.76062286,  0.05387998,  0.56165695],
        [ 0.13172991, -0.02106751,  0.98988964, -0.04818609],
        [ 0.75103901,  0.15054422, -0.06561061,  0.63950925]]),
 array([8.04090156, 2.61608525]),
 array([[ 0.74289445,  0.66940857],
        [ 0.66940857, -0.74289445]]))
```

```
A1=pca1.inverse_transform(out1) # 수정된 행렬 A1
u1,s1,vt1=np.linalg.svd(A1) # A1의 SVD
u1,s1,vt1
```

```
(array([[-0.56169061, -0.62273321,  0.12889258,  0.5292388 ],
        [-0.32107831, -0.42612436,  0.01429015, -0.84564918],
        [ 0.13172991,  0.04031834,  0.98901296, -0.05361928],
        [ 0.75103901, -0.65497772, -0.0709638 ,  0.04368897]]),
 array([8.04090156e+00, 2.55583427e-16]),
 array([[ 0.74289445,  0.66940857],
        [ 0.66940857, -0.74289445]]))
```

다시 원래 작업으로 돌아가서 red_pca 변환의 역변환을 통해 원래의 픽셀 데이터를 복원시킨 후의 이미지를 그려보자. 원래의 이미지랑 비교해 보면 색감의 차이는 있지만 무슨 그림인지 거의 복원되는 것을 확인할 수 있다.

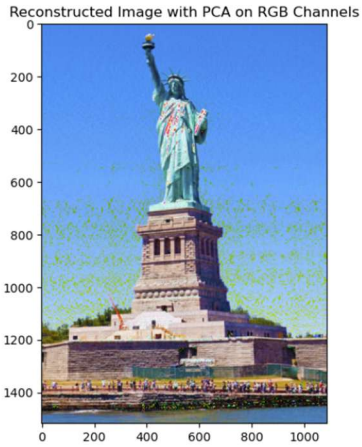
```
# PCA를 통해 변환된 데이터를 다시 복원
red_reconstructed = pca_red.inverse_transform(red_pca)

# 원래 이미지와 주성분을 이용해 재구성된 이미지 비교
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(image_array)
axes[0].set_title("Original Image")
axes[1].imshow(red_reconstructed.astype(np.uint8))
axes[1].set_title("Reconstructed Image with PCA on R Channel")

plt.show()
```



다음은 R 채널 뿐만 아니라, G,B 채널을 각각 PCA 분석한 후, RGB 채널 3색을 합하여 그려보도록 하자. R,G,B 각 채널의 데이터의 크기가 (1517,1084)인데, 주성분 100개만으로도 거의 유사한 이미지를 보일 수 있다는 것을 살펴볼 수 있다.

```
# 이미지를 NumPy 배열로 변환
image_array = np.array(image)

# R, G, B 채널 분리
red_channel = np.array(image.getdata(band=0),int)
red_channel.shape=(image.size[1],image.size[0])
green_channel = np.array(image.getdata(band=1),int)
green_channel.shape=(image.size[1],image.size[0])
blue_channel = np.array(image.getdata(band=2),int)
blue_channel.shape=(image.size[1],image.size[0])

# PCA 적용 (주성분 100개)
pca_red = PCA(n_components=100)
pca_green = PCA(n_components=100)
pca_blue = PCA(n_components=100)

red_pca = pca_red.fit_transform(red_channel)
green_pca = pca_green.fit_transform(green_channel)
blue_pca = pca_blue.fit_transform(blue_channel)
```

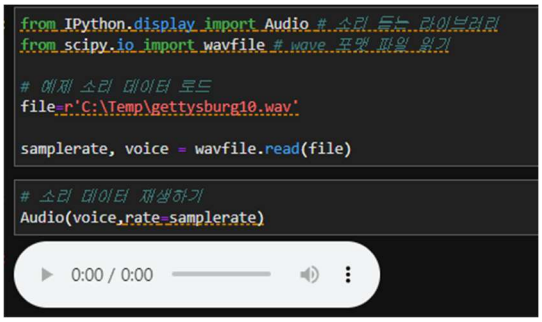
```
# PCA를 통해 변환된 데이터를 다시 복원
red_reconstructed = pca_red.inverse_transform(red_pca)
green_reconstructed = pca_green.inverse_transform(green_pca)
blue_reconstructed = pca_blue.inverse_transform(blue_pca)

# 이미지 재구성을 위해 shape 조정
red_reconstructed = red_reconstructed.reshape(image_array[:, :, 0].shape)
green_reconstructed = green_reconstructed.reshape(image_array[:, :, 1].shape)
blue_reconstructed = blue_reconstructed.reshape(image_array[:, :, 2].shape)

# 이미지 합치기
image_reconstructed = np.stack([red_reconstructed,
                                green_reconstructed,
                                blue_reconstructed], axis=-1)

# 원래 이미지와 주성분을 이용해 재구성된 이미지 비교
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(image_array)
axes[0].set_title("Original Image")
axes[1].imshow(image_reconstructed.astype(np.uint8))
axes[1].set_title("Reconstructed Image with PCA on RGB Channels")

plt.show()
```



문제5. 임의의 wav 파일을 읽어와서 PCA 분석을 수행하여 보아라.

구글 검색에서 무료 wav 파일을 선택하여 C:₩Temp₩gettysburg10.wav 이름으로 저장하였다. 이 wav 파일을 재생하기 위해서 scipy 라이브러리와 IPython을 아래와 같이 이용할 수 있다.

```
from IPython.display import Audio # 소리 듣는 라이브러리
from scipy.io import wavfile # wave 포맷 파일 읽기

# 예제 소리 데이터 로드
file=r'C:\Temp\gettysburg10.wav'

samplerate, voice = wavfile.read(file)

# 소리 데이터 재생하기
Audio(voice,rate=samplerate)
```

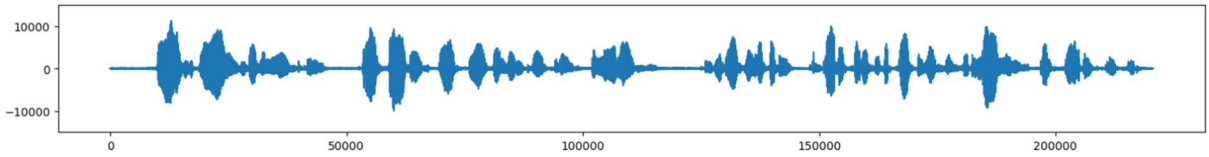


이 데이터를 'numpy' array로 변환하고 이를 그래프로 그려보면 아래와 같다.

```
import numpy as np
# 소리 데이터를 array 형태로 변환
voice_np=np.array(voice,np.float32)
voice_np.shape
```

```
(220568,)
```

```
# 소리를 그려보자~
plt.figure(figsize=(18,2))
plt.plot(voice_np)
plt.ylim((-15000,15000))
plt.show()
```



'voice'라는 데이터의 크기는 대략 220,000개로 시간에 따라 진폭이 표현되는 그래프로 보면 된다. 위 분석 코드 출력물 중 'samplerate'에는 초당 샘플링[6] 된 데이터의 개수를 나타낸다. 즉, 전체 데이터가 10초 짜리 소리이기 때문에 'samplerate' 는 대략 22,000이라고 보면 된다.

'voice'는 1차원 데이터로 PCA 분석에 사용할 수가 없다. 소리 데이터가 시간에 따른 진폭 데이터이기 때문에 적당한 크기('block_size')로 잘라서 2차원 데이터로 만들 필요가 있다[7]. 이 때, 지정된 'block_size' 나누어 떨어지도록 데이터를 변형할 필요가 있다.

```
block_size=516
# 소리 신호가 block_size 로 나누어 떨어질 수 있도록 0의 값을 추가함
samples = len(voice) # voice의 크기
# np.mod(a,b)는 a를 b로 나눈 나머지
# 아래 hanging에는 block_size에서 0으로 만들어야 하는 데이터 개수 계산
hanging = block_size - np.mod(samples, block_size)
# np.lib.pad()를 이용하여 voice 배열 뒷부분에 0을 추가함
padded = np.lib.pad(voice, (0, hanging), 'constant', constant_values=0)
```

```
#나누어 떨어지는지 검증
np.mod(len(padded),block_size)
```

```
0
```

이제 2차원 데이터로 변경하면 된다.

---

[6] 소리의 주파수는 1초 동안 소리의 주기가 반복되는 회수이고, 샘플레이트는 1초 동안 추출한 오디오 데이터 개수다.

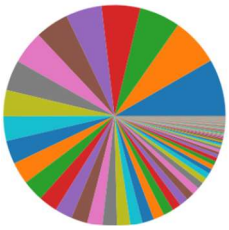
[7] 소리 데이터가 샘플링에 따라 정밀도가 달라지는 것에 착안하여 샘플링의 빈도를 바꾸는 PCA 분석이라고 이해할 수 있다.

```
# 데이터를 2차원으로 reshape
# a // b 는 a를 b로 나눈 몫
reshaped = padded.reshape((len(padded) // block_size, block_size))
reshaped.shape
```

```
(428, 516)
```

PCA 분석을 통해 분산의 분포를 먼저 살펴보면 아래와 같다. 분산이 큰 순서대로 처음 30개의 분산의 설명력을 살펴보면 대략 91% 정도가 된다.

```
# PCA 분석
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA(n_components=418) # 객체 지정
pca.fit(reshaped) # PCA 분석 수행
# 분산의 분포 보기
plt.pie(pca.explained_variance_ratio_)
plt.show()
```



```
# 설명된 분산의 % 계산
V=np.sum(pca.explained_variance_ratio_[:30])
print('보존된 데이터 분산: ', V)
```

```
보존된 데이터 분산:  0.9126292093764892
```

주성분 개수를 30개로 줄여서 데이터를 변환시킨 후, 다시 원복하여 소리의 어느 정도가 보존되었는지 확인해 보도록 하자. 잡음이 좀 들리기는 하지만, 상당 부분 식별이 가능하다.

```
# 소리 압축과 역변환
pca = PCA(n_components=30)
pca.fit(reshaped)

pca.fit(reshaped)
transformed = pca.transform(reshaped)
# 원래의 데이터 모양으로 역변환 시킴
reconstructed = pca.inverse_transform(transformed).reshape((len(padded)))
# 오디오 재생
Audio(reconstructed, rate=samplerate)
```

참고문헌

1. 주피터랩파일: PCA.ipynb
2. 캐글닷컴 코드: PCA Implementation
3. PCA를 이용한 오디오 압축
4. Sound File samples
5. 유투브 강의: 강완모 교수의 선형대수학