고급소프트웨어실습1 Week3 HW

컴퓨터공학 20172141 김미소

PCA reconstruction

Digits 데이터를 2차원, 3차원, 4차원, 32차원으로 PCA 한 결과를 reconstruction 했을 때 원래 데이터와의 MSE(Mean Square Error) 을 확인 및 복원데이터를 시각화하고 오차율이 다른 이유를 분석하여 보고서를 제출하시오.

```
def student_pca(X, n_components=2):
    pca_results = None
    # TODO -->
    # preprocessing
    tmp_X = X.copy()
    tmp_X -= tmp_X.mean(axis=0)
    cov_X = np.cov(tmp_X.T)

    # eigen val, vec
    eigen_value, eigen_vector = np.linalg.eig(cov_X)
    eigen_vector = eigen_vector[:, :n_components]
    pca_results = np.dot(tmp_X, eigen_vector)
    # <--
    return pca_results
```

우선, 처음에 PCA를 직접 구현할 때 비효율적으로 구현한 부분이 있어 수정하였다. 그리고 기존의 코드는 X 데이터에 손실을 일으키므로 tmp_X라는 변수에 카피하여 PCA결과를 구하였다.

```
def reconstruct(X, pca_X, n_components=2):
    reconstruction = None
    tmp_X = X.copy()
    tmp_X -= tmp_X.mean(axis=0)
    tmp_X_cov = np.cov(tmp_X.T)

    eigval, eigvec = np.linalg.eig(tmp_X_cov)

    eigvec = eigvec[:, :n_components]
    reconstruction = np.dot(pca_X, eigvec.T)
    reconstruction = reconstruction + X.mean(axis=0)

    return reconstruction
```

PCA 결과를 다시 복원할 수 있는 함수를 작성하였다.

X 데이터의 손실을 막기 위하여 tmp_X라는 변수에 X를 카피하여 사용하였다. 직접 구현한 PCA 함수에서 eigen value와 eigen vector를 따로 저장하지 않기 때문에 tmp_X를 이용하여 다시 구하였다.

PCA된 결과는 n*p 차원의 데이터와 p*k 차원의 eigen vector를 곱하여 얻어진 n*k 벡터이다. PCA 결과를 다시 복원하기 위해서는 PCA 데이터에 k*p 차원의 eigen vector(기존의 eigen vector의 전치 행렬)을 곱하고 원 데이터의 평균을 더해야한다.

이를 구현한 것이 위의 reconstruct 함수이다.

2차원 PCA reconstruction

```
# 2차원
pca_student_digits = student_pca(X, n_components=2)
pca_reconstructed = reconstruct(X, pca_student_digits, n_components=2)

# MSE 출력
mse = np.square(np.subtract(X, pca_reconstructed)).mean()
print(f'2차원: MSE Error: {mse}')

# 시각화
n_samples = pca_reconstructed.shape[0]
images = pca_reconstructed.reshape((n_samples, -1))

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image in zip(axes, images):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
```

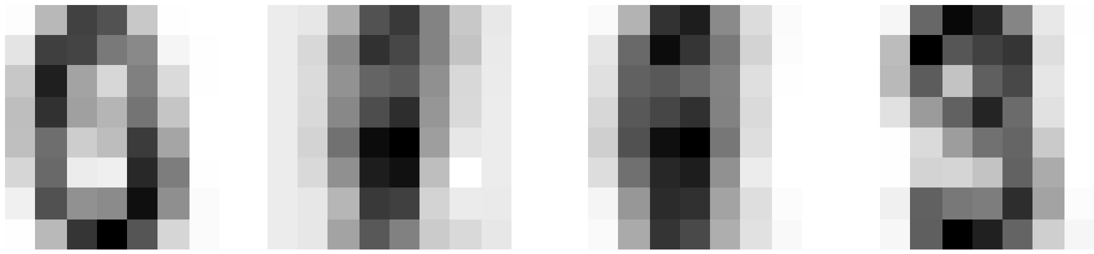
첫 번째 줄은 X를 2차원으로 PCA한 것이고 두 번째 줄은 PCA결과를 다시 복원한 것이다.

```
# numpy를 이용하여 MSE 구하기
mse = (np.square(A - B)).mean(axis=ax)
```

MSE는 A와 B 행렬이 있을 때 numpy를 이용하여 다음과 같이 구할 수 있다. 따라서 PCA reconstruction에 대한 MSE 또한 이 방식을 이용하여 구하였다.

시각화 방법은 파일에 구현되어있던 방법을 그대로 사용하였다.

2차원: MSE Error: 13.421012200761453



결과는 MSE Error = 13.421012200761453이 나왔으며 시각화 결과는 위와 같다.

3차원 PCA reconstruction

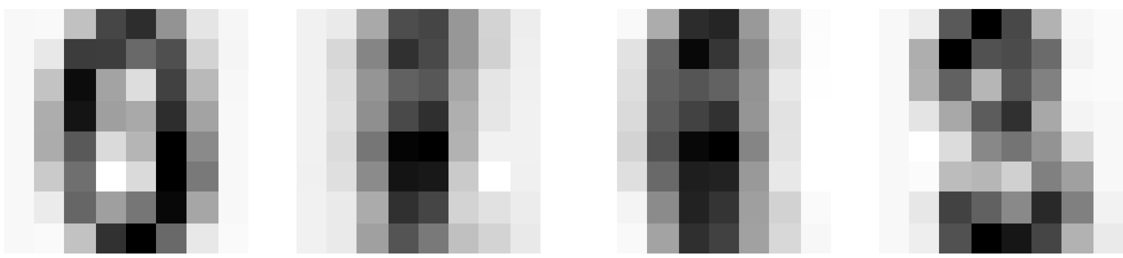
```
# 3차원
pca_student_digits = student_pca(X, n_components=3)
pca_reconstructed = reconstruct(X, pca_student_digits, n_components=3)

# MSE 출력
mse = np.square(np.subtract(X, pca_reconstructed)).mean()
print(f'3차원: MSE Error: {mse}')

# 시각화
n_samples = pca_reconstructed.shape[0]
images = pca_reconstructed.reshape((n_samples, -1))

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image in zip(axes, images):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
```

3차원 PCA reconstruction 또한 n_components를 3으로 바꾸고 동일하게 구현하였다.

PCA를 한 후 PCA 결과 데이터를 다시 reconstruct 함수를 이용하여 복원하였다.

그리고 MSE를 계산하여 출력하고 이미지 또한 출력하였다.

3차원: MSE Error: 11.206800697129161



결과는 MSE = 11.206800697129161이 나왔고 시각화 결과는 위와 같다.

4차원 PCA reconstruction

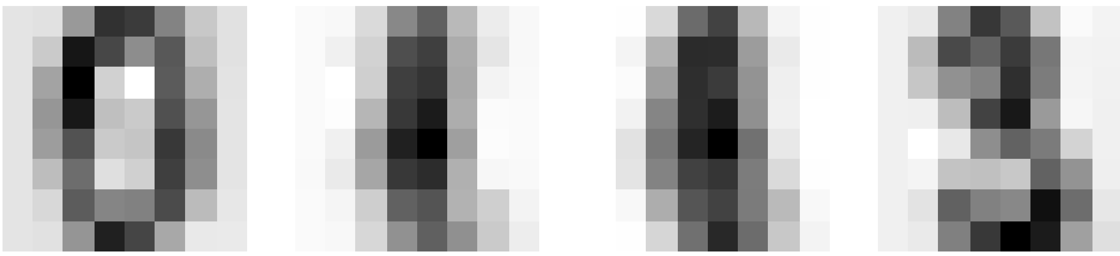
```
# 4차원
pca_student_digits = student_pca(X, n_components=4)
pca_reconstructed = reconstruct(X, pca_student_digits, n_components=4)

# MSE 출력
mse = np.square(np.subtract(X, pca_reconstructed)).mean()
print(f'4차원: MSE Error: {mse}')

# 시각화
n_samples = pca_reconstructed.shape[0]
images = pca_reconstructed.reshape((n_samples, -1))

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image in zip(axes, images):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
```

4차원 PCA reconstruction 또한 n_components를 4로 바꾸고 동일하게 구현하였다.

PCA를 한 후 PCA 결과 데이터를 다시 reconstruct 함수를 이용하여 복원하였다.

그리고 MSE를 계산하여 출력하고 이미지 또한 출력하였다.

```
4차원: MSE Error: 9.62798640712921
```



결과는 MSE = 9.62798640712921이 나왔고 시각화 결과는 위와 같다.

32차원 PCA reconstruction

```
# 32차원
pca_student_digits = student_pca(X, n_components=32)
pca_reconstructed = reconstruct(X, pca_student_digits, n_components=32)
```

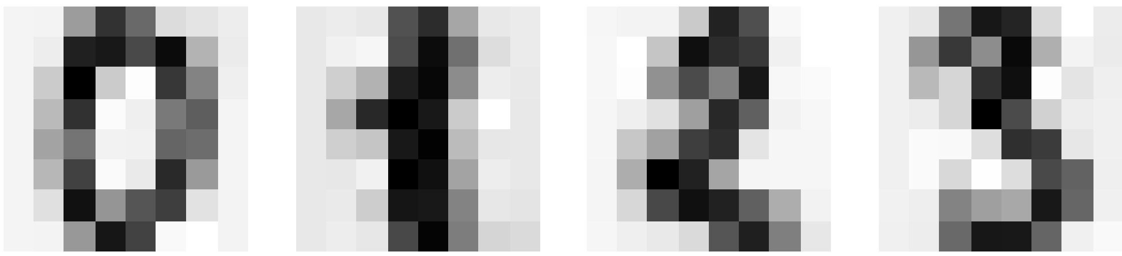
```
# MSE 출력
mse = np.square(np.subtract(X, pca_reconstructed)).mean()
print(f'32차원: MSE Error: {mse}')

# 시각화
n_samples = pca_reconstructed.shape[0]
images = pca_reconstructed.reshape((n_samples, -1))

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image in zip(axes, images):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
```

32차원 PCA reconstruction 또한 n_components를 32로 바꾸고 동일하게 구현하였다.

PCA를 한 후 PCA 결과 데이터를 다시 reconstruct 함수를 이용하여 복원하였다.

그리고 MSE를 계산하여 출력하고 이미지 또한 출력하였다.

```
32차원: MSE Error: 0.6316360146108383
```



결과는 MSE = 0.6316360146108383이 나왔고 시각화 결과는 위와 같다.

오차율이 다른 이유

PCA는 각각의 축이 주요 요소들을 표현하는 n차원 plane에 데이터를 맞추는 것으로 생각될 수 있다. 만약 몇 개의 축이 작으면, 축에 따른 변화 또한 작을 것이다. 데이터 집합의 표현에서 작은 축과 그에 해당하는 주요 요소들을 생략함으로써 우리는 상응하는 작은 정보만을 잃을 수 있다.

eigen vector가 eigen value가 높은 순으로 정렬되었을 때 첫 번째 eigen vector는 분산이 제일 큰 방향을 나타내고 두 번째 eigen vector는 분산이 두 번째로 큰 방향을 나타낼 것이다. 분산이 가장 큰 방향을 나타낸다는 것은 표본의 차이를 가장 잘 나타내는 성분이라는 것이다. 즉, 데이터에 대해서 가장 잘 설명할 수 있는 큰 정보이다. 즉, eigen vector에서 뒤쪽으로 갈 수록 노이즈성 정보를 많이 포함하고 있을 것이다. 따라서 차원이 높아질수록 데이

터를 더욱 잘 설명할 수 있는 축(eigen vector의 앞쪽 벡터들)을 보존하기 때문에 정보의 손실량이 줄어들 것이다.

p차원의 데이터 X가 있을 때 p차원의 eigen vector V가 있다고 하자. 이 때 복원을 위해 V'(V 전치행렬)을 곱하면 VV'는 실제로 차원 축소를 하지 않았으므로 Identity 행렬이 된다. 모든 축을 보존하였기 때문에 정보를 잃지도 않아 reconstruction이 완벽하게 구현된다.

따라서 차원이 작아질수록 손실되는 정보가 많기 때문에 오차율이 증가한다.

2차원 MSE Error = 13.421012200761453

3차원 MSE Error = 11.206800697129161

4차원 MSE Error = 9.62798640712921

32차원 MSE Error= 0.6316360146108383

차원이 증가할수록 MSE Error는 줄어드는 모습을 확인할 수 있다.