인공지능

[Assignment #1]

Class : **수2**

Professor : **최상호**

Student ID : **2020202031**

Name : **김재현**

- Introduction

본 프로젝트에서는 feature extraction을 하는 두 가지 방법을 구현하고 비교합니다. 데이터셋은 LFW(Labeled Faces in the World)를 사용합니다. 이 데이터셋은 53명의 얼굴을 담은 이미지를 포함합니다. 이 데이터셋을 이용해 PCA를 이용한 feature extraction을 진행하고, 그렇게 추출한 feature를 이용해 같은 KNN 모델을 이용해 두 feature extraction에 따른 결과 차이를 비교해봅니다.

이 실습에서는 LFW(Labeled Faces in the World), 얼굴 데이터셋을 이용합니다. 53명의 얼굴을 찍은 이미지를 담고 있습니다. 얼굴의 주성분들을 시각화 하여 확인해보고, PCA를 적용한 훈련 데이터로 학습시켜서 K-Nearest Neighbor(KNN) 알고리즘을 통해서 정확도를 측정해봅니다.

데이터셋은 sklearn.datasets 라이브러리의 fetch_lfw_people 메서드를 사용하여 추출된 1140개의 이미지입니다. fetch_lfw_people.images는 2D 이미지 1140개 즉, 3차원 데이터가 주어지고, fetch_lfw_people.data는 2D 이미지를 1D로 차원축소를 통해 1D 데이터 1140개 즉, 2차원 데이터가 주어집니다.

- Algorithm

PCA: 한국어로 주성분분석으로 불리는 PCA는 고차원의 데이터의 특징을 최대한 살리면서 더 간단하게 표현하기 위해 차원을 축소하는 통계적 기법으로 다음과 같은 과정을 거칩니다.

1. 데이터 표준화
2. 공분산 행렬 계산
3. 고유벡터와 고유값 계산
4. 주성분 선택
5. 새로운 좌표계로 데이터 변환

이번 과제에서는 데이터 표준화를 centering과 whitening으로 진행하고, 고유벡터와 고유값 계산은 gram_schmidt를 통한 qr분해를 반복해서 도출해냅니다.

KNN: K-NN은 K-Nearest Neighbor의 줄임말인데, 말 그대로 K개의 가까운 이웃의 속성에 따라 분류합니다. KNN 알고리즘은 이처럼 데이터를 가장 가까운 속성에 따라 분류하여 레이블링을 하는 알고리즘입니다.

KNN은 거리를 측정하여 가까운 속성을 판별해냅니다. 거리를 측정하는 방법에는 유클리드 거리, 맨해튼 거리 등 여러 방법이 존재하지만, 일반적으로는 유클리드 거리를 사용합니다.

유클리드 거리는 대각방향 거리를 고려한 두 점 사이의 최소거리를 의미하고, 맨해튼 거리는 수평, 수직 방향만 고려한 두 점 사이의 최소거리를 의미합니다.

KNN은 적절한 K값을 선택하지 않으면 과적합 또는 과소적합이 발생할 수 있습니다.

K를 너무 작은 수로 채택하면, 매우 가까운 소수의 이웃들에게 의존하여 결정하게 되므로, 노이즈에 민감하게 반응해 예측이 왜곡될 가능성이 커집니다.

K를 너무 큰 수로 채택하면, 더 많은 이웃을 고려하기 때문에, 모델은 전반적으로 데이터를 잘 일반화할 수 있습니다. 그러나 모델이 매우 단순해져서 복잡한 패턴을 제대로 학습하지 못하고, 예측 성능이 떨어질 수 있습니다.

또한 K 값을 홀수로 설정하는 것이 일반적입니다. 이는 다수결 방식에서 동률이 발생하지 않도록 하는 데 도움이 됩니다.

PCA를 구현합니다.

데이터셋의 공분산행렬을 구하고, 공분산행렬의 고유값과 고유벡터를 구합니다. 고유값의 크기가 큰 순서로 변환하고자 하는 차원 수만큼 고유벡터를 추출합니다.

데이터셋에 추출한 고유벡터를 행렬곱하면 pca_output이 생성됩니다.

```
class PCA:
    def __init__(self, n_components, whiten, random_state=41, gram_iter=100):
        self.n_components = n_components
        self.whiten = whiten
        self.gram_iter = gram_iter
```

n_coponents는 공분산행렬의 고유벡터에서 추출할 상위벡터 개수입니다.
whiten은 whitening을 적용할지 여부입니다. boolean 자료형입니다.
gram_iter은 gram_schmidt 알고리즘을 반복할 횟수입니다.

1) centering

centering은 데이터를 centering하는 메서드입니다. 데이터를 평균으로 뺄셈하면 centering 됩니다. numpy.mean(data, axis=0) 메서드를 통해 열벡터의 평균을 구하고 원본 데이터에서 -연산을 진행함으로써 centered_data를 도출합니다.

centering 메서드 TEST

[9]:
```
pca = PCA(200, False)
data = people.data
print("before centering:")
print(data)
print()
centered_data = pca.centering(people.data)
print("after centering:")
print(centered_data)
print()
centered_data_sum = np.sum(centered_data, axis=0)
print("sum of vectors:")
print(centered_data_sum)
```

```
before centering:
[[0.32026145 0.34771243 0.26013073 ... 0.4        0.5542484  0.82483655]
 [0.21045752 0.18954249 0.27189544 ... 0.9281046  0.89673203 0.86928105]
 [0.14379086 0.151634   0.16209151 ... 0.3869281  0.2784314  0.23137255]
 ...
 [0.8457516  0.83398694 0.8091503  ... 0.57254905 0.54509807 0.62614375]
 [0.37124184 0.4627451  0.52287585 ... 0.8679738  0.8392157  0.5124183 ]
 [0.13594772 0.3124183  0.427451   ... 0.09411765 0.14248367 0.08366013]]

after centering:
[[-0.03172213 -0.01628271 -0.12614125 ... -0.05471763  0.12844151
   0.4268923 ]
 [-0.14152606 -0.17445265 -0.11437654 ...  0.47338694  0.47092515
   0.4713368 ]
 [-0.20819272 -0.21236114 -0.22418047 ... -0.06778952 -0.1473755
  -0.16657169]
 ...
 [ 0.493768    0.4699918   0.4228783  ...  0.11783141  0.11929119
   0.22819951]
 [ 0.01925826  0.09874997  0.13660386 ...  0.41325617  0.41340882
   0.11447409]
 [-0.21603586 -0.05157682  0.04117903 ... -0.3606     -0.28332323
  -0.31428412]]

sum of vectors:
[ 0.00016786 -0.00019288  0.00027695 ... -0.00032485 -0.00022879
 -0.0002211 ]
```

centering을 적용하기 전과 후를 비교해 보고 centering 한 데이터의 합을 구해봅니다. 오차 때문에 centered_data의 합이 완전한 0은 아니지만 거의 0에 근접하는 것을 확인할 수 있습니다.

2) whitening

whitening(x) = (x – mean) / std 공식을 활용하여 구현합니다. x – mean은 centered_data이므로 whitened_data = centered_data / std 입니다.

whitening 메서드 TEST ¶

```
pca = PCA(200, False)
data = people.data
print("before whitening:")
print(data)
print()
whitened_data = pca.whitening(data)
print("after centering:")
print(whitened_data)
print()
whitened_data_sum = np.sum(whitened_data, axis=0)
print("sum of vectors:")
print(whitened_data_sum)
```

```
before whitening:
[[0.32026145 0.34771243 0.26013073 ... 0.4        0.5542484  0.82483655]
 [0.21045752 0.18954249 0.27189544 ... 0.9281046  0.89673203 0.86928105]
 [0.14379086 0.151634   0.16209151 ... 0.3869281  0.2784314  0.23137255]
 ...
 [0.8457516  0.83398694 0.8091503  ... 0.57254905 0.54509807 0.62614375]
 [0.37124184 0.4627451  0.52287585 ... 0.8679738  0.8392157  0.5124183 ]
 [0.13594772 0.3124183  0.427451   ... 0.09411765 0.14248367 0.08366013]]

after centering:
[[-0.18723463 -0.09794051 -0.7703593  ... -0.17709179  0.41925177
   1.3976971 ]
 [-0.8353342  -1.049333   -0.6985108  ...  1.5321012   1.5371681
   1.5432137 ]
 [-1.2288232  -1.2773527  -1.3690962  ... -0.21939854 -0.48105505
  -0.5453759 ]
 ...
 [ 2.9143841   2.8270016   2.582567   ...  0.38135743  0.38938376
   0.7471528 ]
 [ 0.1136687   0.5939812   0.83425564 ...  1.3374898   1.3494265
   0.374802  ]
 [-1.275116   -0.3102347   0.25148514 ... -1.1670699  -0.9248082
  -1.0290042 ]]

sum of vectors:
[ 0.00100362 -0.00117537  0.00169322 ... -0.00103641 -0.0007357
 -0.00071585]
```

whitening을 적용하기 전과 후를 비교해 보고 whitening한 데이터의 합을 구해봅니다. 오차 때문에 whitened_data의 합이 완전한 0은 아니지만 거의 0에 근접하는 것을 확인할 수 있습니다.

3) covariance

공분산행렬을 구하는 covariance 메서드입니다.

covariance matrix = zz^T/n 공식을 사용하여 구현했습니다.

covariance 메서드 TEST

```
pca = PCA(200, False)
data = people.data
print("PCA:")
print(pca.covariance(data))
print()
print("NUMPY: ")
print(np.cov(data, rowvar=False))
```

```
PCA:
[[ 0.02872987  0.02668983  0.02281423 ... -0.00139315 -0.00165067
  -0.00146528]
 [ 0.02668983  0.02766363  0.02523447 ... -0.00078209 -0.00096708
  -0.0011113 ]
 [ 0.02281423  0.02523447  0.02683547 ... -0.00056825 -0.00087729
  -0.00160631]
 ...
 [-0.00139315 -0.00078209 -0.00056825 ...  0.09555178  0.08638863
   0.07212054]
 [-0.00165067 -0.00096708 -0.00087729 ...  0.08638863  0.09393813
   0.08742227]
 [-0.00146528 -0.0011113  -0.00160631 ...  0.07212054  0.08742227
   0.09336673]]

NUMPY:
[[ 0.02872987  0.02668983  0.02281423 ... -0.00139315 -0.00165067
  -0.00146528]
 [ 0.02668983  0.02766364  0.02523447 ... -0.00078209 -0.00096708
  -0.0011113 ]
 [ 0.02281423  0.02523447  0.02683546 ... -0.00056825 -0.00087729
  -0.00160631]
 ...
 [-0.00139315 -0.00078209 -0.00056825 ...  0.09555179  0.08638861
   0.07212054]
 [-0.00165067 -0.00096708 -0.00087729 ...  0.08638861  0.09393812
   0.08742228]
 [-0.00146528 -0.0011113  -0.00160631 ...  0.07212054  0.08742228
   0.09336671]]
```

PCA class의 covariance 메서드와 np의 cov메서드가 동일한 결과를 출력하는 것을 확인할 수 있습니다.

4) gram_schmidt

QR분해를 하기 위해 gram_schmidt 알고리즘을 사용합니다.

$$u_k = v_k - \sum_{j=1}^{k-1} \mathrm{proj}_{u_j}(v_k)$$

위 공식을 활용하여 코드를 구현했습니다.

gram_schmidt 메서드 TEST ¶

```
A = np.array([[ 1.04716195, -0.18351747, -0.10934386, -0.00601054, -0.07288471],
                       [-0.18351747,  0.94827502,  0.00726028, -0.0830421 ,  0.08130832],
                       [-0.10934386,  0.00726028,  1.08134424,  0.0548622 , -0.03538415],
                       [-0.00601054, -0.0830421 ,  0.0548622 ,  0.91862992, -0.05356764],
                       [-0.07288471,  0.08130832, -0.03538415, -0.05356764,  1.08806177]])


pca = PCA(200, False)
Q, R = pca.gram_schmidt(A)
print("PCA")
print("Q:\n", Q)
print("R:\n", R)
print()

print("NUMPY")
Q, R = np.linalg.qr(A)
print("Q:\n", Q)
print("R:\n", R)
print()
```

```
PCA
Q:
 [[ 0.97753339  0.1720481   0.10439437  0.0138482   0.06113887]
 [-0.17131491  0.97808922  0.02037106  0.09368345 -0.06929004]
 [-0.1020733  -0.03106457  0.99230642 -0.05100842  0.03663518]
 [-0.00561088 -0.09353995  0.04571164  0.99329719  0.04989916]
 [-0.06803841  0.06345235 -0.04387759 -0.04231388  0.99379518]]
R:
 [[ 1.07122883e+00  1.12558858e-17 -2.23392779e-17  4.27399188e-18
  -9.44794220e-19]
 [ 2.44172212e-17  9.08625164e-01  9.08967993e-18  3.98729912e-17
  -5.08121850e-17]
 [-1.90171613e-17  8.16724616e-18  1.06581827e+00 -2.02573374e-17
   3.43961751e-17]
 [ 4.82707684e-18  4.50304903e-17 -2.23810211e-17  9.04077836e-01
   1.95240862e-17]
 [ 4.36403032e-18 -2.82753252e-17  2.97022610e-17  1.07566129e-17
   1.06725132e+00]]

NUMPY
Q:
 [[-0.97753339 -0.1720481  -0.10439437 -0.0138482   0.06113887]
 [ 0.17131491 -0.97808922 -0.02037106 -0.09368345 -0.06929004]
 [ 0.1020733   0.03106457 -0.99230642  0.05100842  0.03663518]
 [ 0.00561088  0.09353995 -0.04571164 -0.99329719  0.04989916]
 [ 0.06803841 -0.06345235  0.04387759  0.04231388  0.99379518]]
R:
 [[-1.07122883e+00  0.00000000e+00  2.77555756e-17 -3.46944695e-18
  -1.38777878e-17]
 [ 0.00000000e+00 -9.08625164e-01 -6.93889390e-18 -4.16333634e-17
   1.38777878e-17]
 [ 0.00000000e+00  0.00000000e+00 -1.06581827e+00  2.08166817e-17
  -2.08166817e-17]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 -9.04077836e-01
  -2.08166817e-17]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.06725132e+00]]
```

gram_schmidt를 통한 QR분해는 numpy의 QR 메서드를 통한 QR분해에 비해 정확성이 떨어지는 편입니다. 데이터의 크기가 작은 상황에서는 오차가 매우 미미하지만 데이터의 크기가 커진다면 오차가 커지게됩니다. 위 TEST에서는 5x5 행렬을 QR분해하여 numpy의 QR 메서드와 유사한 결과가 나오는 것을 확인할 수 있습니다. R의 i<j인 성분들은 0입니다. gram_schmidt를 통한 R의 i<j인 성분들 또한 e^-(k>17) 이므로 0에 수렴하는 것을 확인할 수 있습니다.

5) eig

gram_schmidt 알고리즘을 일정 횟수 반복하여 eigenvalues와 eigenvectors를 도출해냅니다.

eig 메서드 TEST

```
A = np.array([[ 1.04716195, -0.18351747, -0.10934386, -0.00601054, -0.07288471],
                       [-0.18351747,  0.94827502,  0.00726028, -0.0830421 ,  0.08130832],
                       [-0.10934386,  0.00726028,  1.08134424,  0.0548622 , -0.03538415],
                       [-0.00601054, -0.0830421 ,  0.0548622 ,  0.91862992, -0.05356764],
                       [-0.07288471,  0.08130832, -0.03538415, -0.05356764,  1.08806177]])

pca = PCA(200, False)
print("pca")
val, vec = pca.eig(A)
print("val: ", val)
print("vec: ", vec)

print("numpy")
val, vec = np.linalg.eigh(A)
print("val: ", val)
print("vec: ", vec)
```

```
pca
val:  [1.28179641 1.16066394 0.97370118 0.89681411 0.77049726]
vec:  [[ 0.66360279  0.21627328 -0.25265738 -0.39665271  0.54008155]
 [-0.5151154   0.1159494   0.42303722 -0.06991391  0.73305076]
 [-0.26369332 -0.74809918 -0.3738609  -0.4692758   0.10402735]
 [ 0.13404164 -0.32999035 -0.30936197  0.78583835  0.39986557]
 [-0.45473743  0.52080283 -0.7223067   0.00412072  0.01530819]]
numpy
val:  [0.77049726 0.89675976 0.97375554 1.16066394 1.28179641]
vec:  [[ 0.5400817  -0.38979931  0.26310731 -0.21628385 -0.66359935]
 [ 0.73305079 -0.08112912 -0.42103024 -0.1159412   0.51511725]
 [ 0.10402753 -0.45917653  0.38619767  0.74810337  0.26368141]
 [ 0.39986526  0.79378087  0.28837288  0.32998821 -0.13404689]
 [ 0.01530818  0.0233111   0.72194219 -0.5207956   0.45474572]]
```

eig 메서드에서 gram_schmidt메서드를 통한 QR 분해가 아닌 numpy의 qr메서드를 사용한다면 정확도가 높습니다.

eig 메서드 TEST

```
A = np.array([[ 1.04716195, -0.18351747, -0.10934386, -0.00601054, -0.07288471],
                       [-0.18351747,  0.94827502,  0.00726028, -0.0830421 ,  0.08130832],
                       [-0.10934386,  0.00726028,  1.08134424,  0.0548622 , -0.03538415],
                       [-0.00601054, -0.0830421 ,  0.0548622 ,  0.91862992, -0.05356764],
                       [-0.07288471,  0.08130832, -0.03538415, -0.05356764,  1.08806177]])

pca = PCA(200, False)
print("pca")
val, vec = pca.eig(A)
print("val: ", val)
print("vec: ", vec)

print("numpy")
val, vec = np.linalg.eigh(A)
print("val: ", val)
print("vec: ", vec)
```

```
pca
val:  [1.04716195 0.91611348 0.92200251 0.90735179 0.902919  ]
vec:  [[-0.61923945  0.57670058 -0.08190873  0.33057157  0.4098443 ]
 [-0.3577804   0.25306905  0.46540826 -0.74085479 -0.20610333]
 [ 0.39033199  0.0388916   0.69264682  0.09354104  0.5980124 ]
 [-0.50813577 -0.55097421  0.46251516  0.41248405 -0.23272749]
 [-0.27923944 -0.5461569  -0.288108   -0.40368751  0.61462847]]
numpy
val:  [0.77049726 0.89675976 0.97375554 1.16066394 1.28179641]
vec:  [[ 0.5400817  -0.38979931  0.26310731 -0.21628385 -0.66359935]
 [ 0.73305079 -0.08112912 -0.42103024 -0.1159412   0.51511725]
 [ 0.10402753 -0.45917653  0.38619767  0.74810337  0.26368141]
 [ 0.39986526  0.79378087  0.28837288  0.32998821 -0.13404689]
 [ 0.01530818  0.0233111   0.72194219 -0.5207956   0.45474572]]
```

하지만 gram_schmidt메서드를 통해 QR분해를 진행한다면, 데이터의 크기가 작은데도 불구하고 다음과 같이 정확도가 떨어지는 것을 확인할 수 있습니다. 정확도가 떨어지는 gram_schmidt를 gram_iter(100)만큼 반복하기 때문에, 매우 작은 오차도 쌓이면 매우 큰 오차가 된다는 것을 알 수 있습니다.

- Result

resize를 0.3, gram_iter를 100, n_components를 200으로 하여 PCA를 진행합니다. PCA를 통해 차원이 축소된 데이터로, KNN 알고리즘을 통해 값을 예측했습니다.

먼저 K=1일 때입니다.

```
Naive PCA: 0.526315789473684 2
Whitening PCA: 0.582456140350877 2
```

whitening을 거친 PCA가 f1_score가 더 높은 것을 확인할 수 있습니다.

다음은 K=3일 때입니다.

```
Naive PCA: 0.51929824561403 5
Whitening PCA: 0.571929824561403 5
```

두 PCA의 f1_score가 줄었지만 여전히 whitening을 거친 PCA가 더 높은 것을 확인할 수 있습니다.

다음은 K=5일 때입니다.

```
Naive PCA: 0.561403508771929 8
Whitening PCA: 0.6
```

whitening PCA f1_score가 0.6을 돌파하면서 여전히 더 높은 f1_score를 가지는 것을 확인할 수 있습니다.

다음은 K=7일 때입니다.

```
Naive PCA: 0.592982456140350 8
Whitening PCA: 0.575438596491228 1
```

K의 값으로 더 큰 수들을 대입해봐도 K=5일 때보다 더 큰 f1_score 값은 나오지 않았습니다. 따라서 저의 PCA에서 가장 적합한 K값은 5임을 알 수 있습니다.

resize를 0.3, gram_iter를 30, n_components를 200으로 하여 PCA를 진행합니다. PCA를 통해 차원이 축소된 데이터로, KNN 알고리즘을 통해 값을 예측했습니다.

- Consideration

gram_schmidt를 반복하여 고유벡터와 고유값을 구할 때, 단지 반복하고 마지막에 나오는 Q, R 값이 고유벡터와 고유값이라고 생각했습니다. 그러나 항등행렬은 초기 데이터의 기저벡터이며, 그렇기에 매 gram_schmidt를 통해 나오는 Q 값을 항등행렬에 누적곱해줘야 한다는 것을 깨달았습니다.

직접 K 값에 변화를 주면서 KNN 값 비교를 수행해보니, K 값은 너무 작지도, 너무 크지도 않은 적당한 값을 찾는 것이 중요함을 체감했습니다. 저의 경우엔 실제 데이터는 1000개 가까이 되지만, 적당한 K 값은 5였습니다.