인공지능

[Project1]

Class : **[수 2]**

Professor : **[최상호 교수님]**

Student ID : **[2020202060]**

Name : **[홍왕기]**

Introduction

이 프로젝트는 얼굴 인식 기술의 성능을 향상시키기 위해 주성분 분석 및 KNN 알고리즘을 활용한 데이터 분석을 시행하는 것이다. 얼굴인식은 다양한 응용 프로그램에서 중요한 기술로 자리잡고 있으며 보안 시스템, 사용자 인증, 개인화된 서비스 등에서 사용된다. 하지만 높은 차원의 데이터는 컴퓨팅 자원 소모가 크기 때문에 차원 축소 기술이 필요한 것이다.

이번 프로젝트 에서는 PCA 를 사용하여 데이터의 차원을 효과적으로 축소하고 KNN 분류를 통해 얼굴 인식 성능 평가를 수행한다. 특히 PCA 과정에서 Whitening 을 적용했을 때와 적용하지 않았을 때의 성능 차이를 분석하여 Whitening 이 어떤 영향을 끼치는지 알아본다. 즉 PCA 를 통해 데이터의 차원을 축소하여 특징을 추출한다. 이를 통해 데이터의 분포를 시각적으로 이해하고 데이터의 복잡성을 줄이는 것을 목표로 한다. 이와 더불어 Whitening 적용 여부와 KNN 기법을 사용하여 성능을 평가한다.

이 프로젝트에 사용된 데이터 set 은 LFW 데이터 SET 으로 fetch_lfw_people 함수를 통해 로드된다. 샘플 수는 1140 개의 얼굴 이미지가 포함되어 있고, 각 샘플은 2914 개의 특성으로 표현된다. 각 특성은 이미지에 픽셀 값에 해당된다. 데이터 set 에는 각 샘플에 대한 레이블 정보가 포함되어 있어, 얼굴 인식 알고리즘의 성능을 평가하기 위해 사용될 것이다.

결론적으로 이 프로젝트의 최종 목표는 PCA 와 KNN 을 조합하여 얼굴 인식 기술의 성능을 개선하고, 데이터 전처리 기법인 Whitening 의 효과를 검증하는 것이다. 이를 통해 더 높은 정확도를 달성하고, 다양한 응용 프로그램에 적용 가능한 기술을 배우는 것이다.

알고리즘

이 프로젝트에서 전체 흐름은 다음과 같다.

1. **데이터 로드**: LFW 데이터셋을 로드하여 얼굴 이미지와 레이블을 준비한다.
2. **PCA 적용**

   (1)데이터의 차원을 축소하여 주요 특징을 추출한다.

   (2)PCA 는 데이터 분산이 가장 큰 방향으로 축을 정렬하고 그 방향에 따라 데이터를 변환한다.

   (3)Whitening 여부에 따라 데이터의 분포를 정규화 한다.

3. **KNN 학습:** 차원 축소된 데이터를 기반으로 KNN 알고리즘을 학습시킨다.

   (1)KNN 은 새로운 데이터 포인트가 주어졌을 때 훈련 데이터에서 가장 가까운 K 개의 이웃을 찾고, 이웃의 레이블 기반으로 예측을 수행한다.

4. **성능 평가**: 테스트 데이터를 사용하여 결과를 비교한다.

이를 통해 알고리즘은 큰 틀로 PCA 와 KNN 을 설명하고 코드에 대한 세부 알고리즘을 설명할 것이다.

먼저 **주성분 분석인 PCA 알고리즘이다.**

PCA 는 고차원 데이터를 저차원으로 축소하여 데이터의 주요 특징을 추출하는 차원 축소 기법이다. 이는 다음과 같이 동작한다.

먼저 각 변수의 평균을 빼서 데이터를 평균인 0 인 상태로 만든 후, 중심화 된 공분산 행렬을 계산하여 데이터의 분산과 공분산 관계를 파악한다. 공분산이 0 보다 크면 두 변수에 대한 한변수가 증가한다면 다른 변수도 증가하는 것이고, 0 보다 작으면 한변수가 증가하면 다른 변수는 감소하는 것이다. 만약 공분산이 0 이라면 두 변수의 상관관계는 없는 것인 것이다.

다음으로 공분산 행렬의 고유값과 고유벡터를 계산하여 데이터의 주성분을 찾는다. 이 과정은 아래서 자세히 설명할 것이다. 고유값은 각 주성분의 중요도를 나타내며 고유벡터는 데이터의 변화를 설명하는 방향을 줄 것이다.

이렇게 고유값의 크기 순서에 따라 상위 n 개의 주성분을 선택하여 차원을 축소한다. 선택한 주성분으로 원본 데이터를 변환하여 차원 축소된 데이터를 생성하는 것이다. 이때 선택적으로 Whitening 기법을 적용하여 데이터의 분포를 정규화하고, PCA 후 데이터를 스케일링 하여 각 주성분의 중요도를 동등하게 만들 수 있다.

다음은 KNN 알고리즘이다.

이 알고리즘은 주어진 데이터에 대해 가장 가까운 K 개의 이웃을 찾고 이웃의 레이블을 기반으로 예측을 수행하는 알고리즘이다.

KNN 은 훈련 단계에서 모델을 학습하는 것이 아니라, 훈련 데이터를 저장한다. 이 과정에서 데이터에 대한 별도의 가정이 없는 것이다. 새로운 데이터 포인트가 주어지면, 훈련 데이터와의 거리를 계산한다. 일반적으로는 유클리드 거리 또는 멘하탄 거리를 사용한다. K 개의 가장 가까운 이웃을 선택하여 이웃의 레이블 기반으로 예측한다. 이때 K 값이 작을수록 모델이 훈련 데이터에 민감해지고 K 의 값이 클수록 모델이 더 일반화된다. 이는 선택된 K 개의 이웃의 레이블 중 가장 많이 나타나는 레이블을 새로운 데이터 포인트의 예측 레이블로 할당하는 알고리즘이다.

PCA 알고리즘 세부 설명

__init__ function:

이 함수는 pca 객체를 초기화 하여 주성분, whiten 여부, 반복 수 등을 설정하는 함수이다.

covariance(self, data):

이 함수는 입력된 데이터의 공분산 행렬을 계산하는 함수이다.

먼저 함수는 mean 함수를 통해서 각 변수의 평균을 계산한다. 이 평균은 나중에 데이터 중심화를 하기 위해서 사용되는 것이다. 데이터 중심화는 원본 데이터에서 평균을 빼는 과정을 통해 이뤄지며, 이는 모든 변수의 평균이 0 이 되도록 한다. 이를 저장하여 공분산 행렬을 계산을 시작한다.

$$\text{Covariance matrix } \Sigma = \frac{1}{n} zz^T, \text{ which } z = x - \mu$$

공분산 행렬은 중심화 된 z 행렬을 기반으로 위와 같은 알고리즘으로 도출되기 때문에 np.dot(centered_data.T, centered_data)를 표본이기에 n-1 로 나누면 공분산 행렬이 도출된다. 이를 통해서 차원 축소를 하기 위해 공분산 행렬을 구한 것이다.

gram_schmidt(self, A):

이 함수는 주어진 행렬 A 에 대해서 Gram_Schmidt 과정을 수행하여 행렬 A 를 직교 백터 Q 와 상삼각 행렬 R 로 분해하는 함수이다. 받은 A 행렬을 통해서 행과 열 수를 얻고 직교벡터 Q 와 상 삼각 행렬 R 을 초기화 시켜야 한다. 이때 행렬 곱 특성상 Q 는 N x M 이고 R 은 M x M 크기를 가진다. 이 다음 각 열 벡터 v 를 A 의 j 번째 열로 설정한다. 직교화 과정은 이전에

계산된 Q 의 열 벡터를 사용하여 진행된다. 이때 각 q 에 대해 v 의 q 방향 성분을 제거하여 v 를 수정한다. 이는 v 에서 q 방향으로의 성분을 뺀 결과로 직교화를 수행하는 것이다.

정규화 과정에서는 v 의 길이를 계산하여 0 이 아닌 경우에만 Q 의 j 번째 열에 정규화 된 벡터를 저장한다. 만약 길이가 0 에 가까운 경우 해당 열에는 0 벡터를 할당한다. 이는 선형 종속적 벡터에 대한 처리를 위해 필요한 것이다. 이렇게 직교 행렬 Q 를 구했다면 R=Q^t * A 로 계산되므로 R 을 도출한다. 위와 같은 알고리즘으로 수행된다면 최종적으로 직교 기저벡터 Q 와 상삼각 행렬 R 을 구하여 PCA 기법 작업에 활용될 것이다.

eig(self, data):

이 함수는 공분산을 구하는 함수와 Gram_Schmidt 함수를 활용하여 고윳값과 고유벡터를 계산하는 역할을 한다. 이 알고리즘은 입력 데이터에 공분산 함수를 호출하여 공분산 행렬을 계산한다. 공분산 행렬은 데이터에 변동성을 나타나기에 PCA 기법에 기초가 될 것이다. 계산된 공분산 행렬을 복사하여 행렬을 만든 후, Q total 변수를 단위 행렬로 초기화 하여 반복 계산을 통해 고유벡터를 추출할 것이다.

주어진 반복 횟수로 Gram_Schmidt 함수를 호출하여 직교 행렬 Q 와 상 삼각행렬 R 을 계산한다. 이때 QR 알고리즘이 도입되는데 A 를 R*Q 로 업데이트하여 QR 분해를 반복한다. 각 직교행렬에 Q 를 Qtotal 에 지속해서 곱하여 업데이트 한다. 이렇게 반복한다면 대각 성분을 제외한 행렬이 0 으로 수렴할 것이다. 이때 수렴 조건은 tol 로 대각 성분 외에 최대값이 1E-6 을 넘지 않는다면 수렴된 것으로 판단해 반복을 종료시키며 불필요한 반복을 뺄 것이다. 모든 반복을 완료하였다면 A 행렬에 대각 성분은 고유값이 될 것이고 Q total 은 고유 벡터가 될 것이다. 추후에 고윳값이 가장 높은, 즉 분산이 가장 큰 요소를 주성분으로 선택하므로 정렬을 시켜주면 간편할 것이다. 이와 같은 상황으로 PCA 의 주요 과정인 QR 분해를 사용해 고유값과 고유 벡터를 얻을 수 잇는 것이다.

whitening(self, data, eigvalues=None):

이 함수는 데이터의 Whitening 처리를 수행하는 함수로 입력 데이터의 분포를 변환하여 평균이 0 이고 분산이 1 인 정규 분포를 따르도록 하는 함수이다. 이 함수로 인해서 데이터 구조를 보다 쉽게 분석할 수 있도록 도와준다. 매개변수로는 Whitening 처리를 적용할 입력 데이터가 들어오고 고유값이 들어온다.

고유값이 제공되지 않은 경우는 함수 입력 데이터를 평균 0 이 되도록 중심화 하고 특성의 표준편차로 나누어 표준화 한다. 표준화된 데이터는 각 특성이 평균이 0 이고 분산이 1 인 정규 분포를 따를 것이다. 만약 이 함수가 호출될 때 고유값이 제공된 경우에는 이 값을 사용해 Whitening 처리를 수행한다. 입력 데이터는 고유값의 제곱근으로 나누어 스케일링 한다면

PCA 에서 각 주성분의 분산을 균등하게 만들 수 있을 것이다. 이런 알고리즘으로 구현한다면 분석의 효율성을 높이고 데이터 패턴을 더 잘 인식할 수 있게 될 것이다.

fit(self, data):

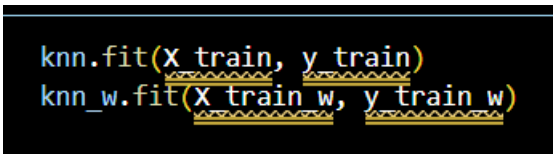
이 함수는 PCA 를 수행하여 입력 데이터를 변환하는 주 함수이다. 이 함수는 PCA 의 핵심 단계인 데이터 전처리, 고유값 및 고유벡터 계산 그리고 최종적인 데이터를 변환한다. 이 함수는 다음과 같은 알고리즘을 따른다.

먼저 Whitening 이 true 인 경우에 Whitening 을 적용한다. 이 과정에서 데이터의 평균을 0 으로 만들고 표준편차로 나누어 데이터를 정규화한다. 만약 적용되지 않는 경우라면 평균만 제거하여 데이터의 분포를 중앙에 정렬하여 공분산을 계산할 때 정확성을 높인다. 이 행렬을 eig 함수 매개변수로 넘겨주어 고유값과 고유 벡터를 계산한다. Eig 함수에서 고유 값의 크기에 따른 고유 벡터가 정렬되어 있음으로 n 개의 주성분을 선택할 것이다. 선택한 고유벡터를 사용하여 입력 데이터를 변환하고 이 과정에서 데이터는 고유벡터 공간으로 투영되어 차원 축소를 진행한다. 이는 다음과 같이 이루어진다. pca_output = np.dot(pca_output, selected_vectors) 이는 선택한 고유벡터를 사용하여 입력 데이터를 변환하는 과정으로 이로 인해 입력 데이터는 고유벡터 공간으로 투영되며 차원이 축소될 것이다. 이는 PCA 기법중 핵심 기능이다. PCA Out data 를 마지막으로 Whitening 이 ture 이라면 해당 함수를 호출하여 데이터 특성이 더욱 균일하게 조정되도록 호출한다. 위 과정을 거친다면 최종적으로 PCA 처리된 데이터 PCA_OUTPUT 이 도출될 것이다. 이 데이터는 차원이 축소되었으며 고유벡터 방향으로 재구성된 형태일 것이다.

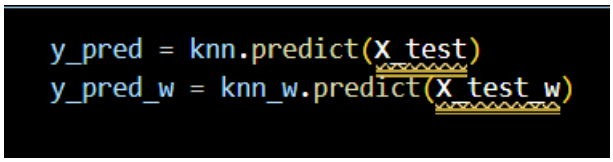
KNN 알고리즘 세부 설명

KNN 알고리즘은 위에 설명과 같이 기계 학습 방법이므로 데이터 포인트의 거리를 기반으로 클래스를 예측한다.

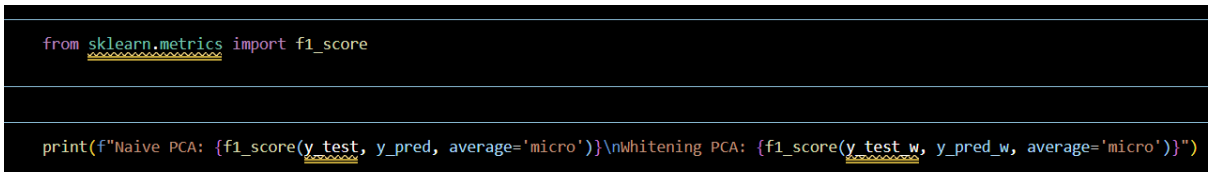
KNeighborsClassifier 클래스를 사용하여 KNN 객체를 생성한다. N_neighbors 를 1 로 설정하였기에 테스트 샘플의 분류는 가장 가까운 1 개의 이웃을 기반으로 한다는 것이다.



위는 위에서 구현한 fit 메서드를 호출하여 KNN 모델을 훈련시키는 것이다. X_train 은 훈련 데이터의 피처를, y_train 은 해당 데이터의 레이블을 나타낸다. X_train_w 와 y_train_w 는 Whitening PCA 를 적용한 후의 훈련 데이터와 레이블이다.



위는 Predict 메서드를 통해 예측을 수행하여 각 변수에 대해 할당한다.



이렇게 알고리즘을 모두 수행했다면 위와 같이 평가를 하는 것이다. F1_score 함수를 사용해 모델의 성능을 평가하고 모델의 성능을 한눈에 파악할 수 있게 해준다. 평균을 micro 로 설정하면 각 클래스의 총 TP, FP, FN 을 계산하여 전체 성능을 평가 할 수 있다.

결론 적으로 KNN 알고리즘을 통해 나이브 PCA 와 Whitening PCA 를 적용한 데이터에 대한 분류 성능을 평가할 수 있을 것이다.

결과화면

공분산 함수 테스트

먼저 구현된 Cov 함수와 내장되어있는 Cov 함수를 통해 정확히 구현하였는지 테스트를 다음과 같이 진행하였다.

```
#공분산 함수와 직접 구현한 공분산 함수와 비교 하는 o

# 데이터셋 가져오기
people = fetch_lfw_people(min_faces_per_person=100, resize=0.5)

# 데이터 행렬 출력
data = people.data  # 데이터 행렬


# PCA 객체 생성
pca = PCA(0, False)

data = np.random.randn(5, 10)
print("data shape:", data.shape)  # 데이터의 형태 출력

# 직접 구현한 공분산 행렬 계산
my_cov_matrix = pca.covariance(data)

# NumPy의 np.cov를 사용하여 공분산 행렬 계산 (행 방향으로 계산)
np_cov_matrix = np.cov(data, rowvar=False)  # rowvar=False는 각 열이 변수임을 의미

# 공분산 행렬 출력
#print("My Covariance Matrix:\n", my_cov_matrix)
#print("NumPy Covariance Matrix:\n", np_cov_matrix)

# 두 행렬의 차이 확인
difference = my_cov_matrix - np_cov_matrix
print("Difference:\n", difference)
✓ 0.0s
```

```
data shape: (5, 10)
Difference:
 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

위 테스트를 확인한다면 직접 구현한 covariance 함수와 내장되어 cov 함수에 차를 계산하였더니 모두 0 임을 볼 수 있다. 이는 공분산 행렬을 정확하게 구현하였음을 볼 수 있다.

Gram_schmidt 함수 테스트

```
A = np.random.rand(3,2)
# PCA 클래스의 인스턴스 생성
pca = PCA(0,whiten=False)
AB=pca.covariance(A)

# Gram-Schmidt 과정 실행
Q,R= pca.gram_schmidt(AB)

print("직접 구현한 Q:")
print(Q)

# 직교성 확인
print("\nQ^T * Q:")
print(np.dot(Q.T, Q))

print("R 값")
print(R)

RQ,RR=np.linalg.qr(AB)
print("정답 Q행렬 ")
print(RQ)

print("정답 R행렬")
print(RR)
```

위 함수를 확인하면 랜덤 숫자로 3x2 행렬을 만들어 직접 구현한 gram_schmidt 와 내장되어있는 qr 함수와 비교하는 함수이다. 이 결과는 다음과 같다.

```
직접 구현한 Q:
[[ 0.99997405  0.00720349]
 [-0.00720349  0.99997405]]

Q^T * Q:
[[ 1.00000000e+00 -7.92040929e-18]
 [-7.92040929e-18  1.00000000e+00]]
R 값
[[ 3.49910226e-02 -2.82455542e-04]
 [-2.55322464e-19  4.21887608e-03]]
정답 Q행렬
[[-0.99997405  0.00720349]
 [ 0.00720349  0.99997405]]
정답 R행렬
[[-0.03499102  0.00028246]
 [ 0.          0.00421888]]
```

qr 함수사용

위를 확인한다면 직접 구현한 q 와 r 이 모두 내장되어 있는 함수를 사용하였을 때와 같음을 보아 잘 구현됨을 확인할 수 있다. 이때 고유벡터는 방향이므로 부호는 상관이 없음을 볼 수 있다.

위를 확인한다면 gram_schmidt 함수를 호출하여 생성된 직교행렬 Q 와 상삼각 행렬 R 이 잘 구현되어 있는지 확인하는 것이다. 직교성, 정규화, 재구성을 파악하여 모두 통과돼 잘 구현됨을 확인할 수 있다.

eig 함수 테스트

```
data = np.array([[2.5000, 2.4000],
                 [0.5000, 0.7000],
                 [2.2000, 2.9000],
                 [1.9000, 2.2000],
                 [3.1000, 3.0000],
                 [2.3000, 2.7000],
                 [2.0000, 1.6000],
                 [1.0000, 1.1000],
                 [1.5000, 1.6000],
                 [1.1000, 0.9000]])
dataC=data-np.mean(data,axis=0)

# PCA 객체 생성
pca = PCA(0, False)
covv = pca.covariance(dataC)

# 고유값 및 고유벡터 계산
eigvalues, eigvectors = pca.eig(dataC)
print("고유값:", eigvalues)
#print("고유 벡터:\n", eigvectors)

# 고유값이 올바른지 확인 (양수여야 함)
assert np.all(eigvalues >= 0), "고유값이 0보다 작거나 같습니다."

# 고유벡터의 형상 확인
#assert eigvectors.shape == (data.shape[1], data.shape[1]), "고유벡터의 형상이 잘못되었습니다."

# 대각 성분 (고유값)이 실제 공분산 행렬의 고유값과 일치하는지 검증
eigen_decomp = np.linalg.eig(covv)
expected_eigvalues = np.sort(eigen_decomp[0])[::-1]  # 내림차순 정렬
expected_eigvectors = eigen_decomp[1][:, np.argsort(eigen_decomp[0])[::-1]]  # 고유값 정렬에 따라 고유벡터 정렬

print("고유값 정답:", expected_eigvalues)

# 허용 오차를 조정하여 assert 문 적용
assert np.allclose(eigvalues, expected_eigvalues, atol=1e-4), "고유값이 예상과 일치하지 않습니다."

#print("고유 벡터 정답:\n", expected_eigvectors)

# 고유벡터 비교: 부호를 무시하고 비교
for i in range(expected_eigvectors.shape[1]):
    comparison_result = np.allclose(eigvectors[:, i], expected_eigvectors[:, i], atol=1e-4) or np.allclose(eigvectors[:, i], -expected_eigvectors[:, i], atol=1e-5)
    assert comparison_result, f"{i}번째 고유벡터가 예상과 일치하지 않습니다."
```

이는 위와 같은 함수로 테스트하였다. 직접 만든 eig 함수를 호출하여 고유벡터와 고유값을 구하고 파이썬 안에 내장 되어있는 eig 함수를 통하여 값을 비교하는 코드인 것이다. 이를 실행해 보면 다음과 같다.

```
고유값: [1.28402771 0.0490834 ]
고유 벡터:
 [[ 0.67787346 -0.7351786 ]
 [ 0.7351786   0.67787346]]
고유값 정답: [1.28402771 0.0490834 ]
고유 벡터 정답:
 [[-0.6778734  -0.73517866]
 [-0.73517866  0.6778734 ]]
테스트가 성공적으로 완료되었습니다!
```

위를 확인한다면 실제 고유값 고유벡터가 구현한 eig 함수로 얻어낸 고유값 고유벡터가 같음을 확인할 수 있다.

Fit 함수 테스트

이를 테스트하기 위해서 다음과 같이 코드를 구현하였다.

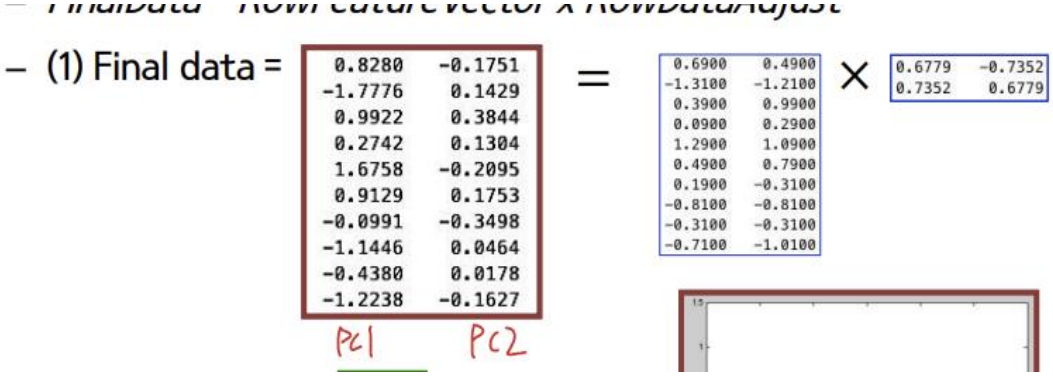
```
# 테스트 데이터 생성 (2차원 데이터)
data = np.array([[2.5000, 2.4000],
                 [0.5000, 0.7000],
                 [2.2000, 2.9000],
                 [1.9000, 2.2000],
                 [3.1000, 3.0000],
                 [2.3000, 2.7000],
                 [2.0000, 1.6000],
                 [1.0000, 1.1000],
                 [1.5000, 1.6000],
                 [1.1000, 0.9000]])


# PCA 클래스 인스턴스 생성 (n_components=1로 설정, Whitening은 적용하지 않음)
pca = PCA(n_components=3, whiten= False)
dataC=data-np.mean(data, axis=0)
# PCA fit 메서드 실행 (차원 축소 수행)
eigvalues, eigvectors = pca.eig(dataC)
pca_output = pca.fit(data)
print("고유값" ,eigvalues)
print("고유벡터 ",eigvectors)
# 결과 출력
print("원본 데이터:")
print(data)
print("\nPCA 변환된 데이터 (1차원):")
print(pca_output)
```

이는 강의자료에 있는 데이터를 활용한 것이다. fit 함수를 통해 얻어낸 고유값, 고유벡터, 원본데이터, 변환된 데이터를 뽑아본다면 다음과 같다.

```
QR 분해 반복: 2/100 완료, 대각 성분 외 최대값: 0.0019570911219078265
QR 분해 반복: 3/100 완료, 대각 성분 외 최대값: 7.481219002244574e-05
QR 분해 반복: 4/100 완료, 대각 성분 외 최대값: 2.85977985288437e-06
5번째 반복에서 수렴. 대각 성분 외 최대값: 1.0931829126737184e-07
고유값 [1.28402771 0.0490834 ]
고유벡터 [[ 0.67787346 -0.7351786 ]
 [ 0.7351786   0.67787346]]
원본 데이터:
[[2.5 2.4]
 [0.5 0.7]
 [2.2 2.9]
 [1.9 2.2]
 [3.1 3. ]
 [2.3 2.7]
 [2.  1.6]
 [1.  1.1]
 [1.5 1.6]
 [1.1 0.9]]

PCA 변환된 데이터 (1차원):
[[ 0.8279702  -0.17511523]
 [-1.77758034  0.14285707]
 [ 0.99219746  0.38437508]
 [ 0.2742104   0.13041723]
 [ 1.67580144 -0.20949831]
 [ 0.91294909  0.17528252]
 [-0.09910941 -0.34982471]
 [-1.14457217  0.04641716]
 [-0.43804614  0.01776459]
 [-1.22382054 -0.1626754 ]]
```



위를 확인한다면 강의자료에서 뽑아낸 최종 데이터, 고유값, 고유벡터가 테스트한 fit 함수에 결과와 같음을 확인할 수 있다.

최종 결과:

```
QR 분해 반복: 80/100 완료, 대각 성분 외 최대값: 0.008189333106270404
QR 분해 반복: 81/100 완료, 대각 성분 외 최대값: 0.008014280880769496
QR 분해 반복: 82/100 완료, 대각 성분 외 최대값: 0.007992418627096117
QR 분해 반복: 83/100 완료, 대각 성분 외 최대값: 0.00795667510803997
QR 분해 반복: 84/100 완료, 대각 성분 외 최대값: 0.007907370646605742
QR 분해 반복: 85/100 완료, 대각 성분 외 최대값: 0.007844939764147959
QR 분해 반복: 86/100 완료, 대각 성분 외 최대값: 0.007769921897492444
QR 분해 반복: 87/100 완료, 대각 성분 외 최대값: 0.007682950228385462
QR 분해 반복: 88/100 완료, 대각 성분 외 최대값: 0.007584739036084406
QR 분해 반복: 89/100 완료, 대각 성분 외 최대값: 0.007476070018301503
QR 분해 반복: 90/100 완료, 대각 성분 외 최대값: 0.007357778035843196
QR 분해 반복: 91/100 완료, 대각 성분 외 최대값: 0.007230736723513422 5
QR 분해 반복: 92/100 완료, 대각 성분 외 최대값: 0.007095844377074713
QR 분해 반복: 93/100 완료, 대각 성분 외 최대값: 0.006954010477316595
QR 분해 반복: 94/100 완료, 대각 성분 외 최대값: 0.006806143152237448
QR 분해 반복: 95/100 완료, 대각 성분 외 최대값: 0.006653137811840089
QR 분해 반복: 96/100 완료, 대각 성분 외 최대값: 0.00649586712165467
QR 분해 반복: 97/100 완료, 대각 성분 외 최대값: 0.006335172414836213
QR 분해 반복: 98/100 완료, 대각 성분 외 최대값: 0.006171856581745684 5
QR 분해 반복: 99/100 완료, 대각 성분 외 최대값: 0.006006678422624785
QR 분해 반복: 100/100 완료, 대각 성분 외 최대값: 0.005840348404730323
```

위를 확인하면 Whitening 기법을 사용하지 않은 결과 반복을 하면 할수록 대각 성분 외에 최대값이 작아지는 것을 확인할 수 있다. 이는 고유값을 추출할 때 대각 성분 외는 거의 0 으로 수렴해야 하므로 100 번에 반복을 선택하였다.

100 번을 모두 수행한 결과 다음과 같은 결과가 나온다.

```
Naive PCA: 0.5894736842105263
Whitening PCA: 0.6701754385964912
```

위를 확인하면 실제 값과 예측한 값을 F1 SCORE 를 통해 비교해봤을 때 Naïve PCA 는 0.59 정도 나오는 것을 확인할 수 있다. 이는 높을 수 록 좋은 성능을 인 것이다. 반면에 Whitening 기법을 도입하였을 때 0.67 로 높아진 것을 확인할 수 있다. 이는 데이터의 스케일이 통일되어 더 나은 예측 성능을 보인 것이다. 이 결과는 Whitening 기법은 PCA 향상을 위해 필수적인 기법이라고 볼 수 있을 것이다.

개선한 부분:

```python
def whitening(self, data, eigvalues=None):
    """
    Whitening 처리
    :param data: 입력 데이터
    :param eigvalues: 고유값 (Whitening 후 처리에 사용)
    :return: Whitening 처리된 데이터
    """
    if eigvalues is None:
        # PCA 전에 데이터를 표준화 (평균을 0으로, 표준편차로 나누기)
        data = data - np.mean(data, axis=0)  # 평균을 0으로
        data = data / np.std(data, axis=0)   # 표준편차로 나누기
        return data
    else:
        # PCA 후 고유값을 사용한 Whitening (고유값으로 스케일링)
        return data / np.sqrt(eigvalues[:self.n_components] + 1e-5)
```

Whitening 과정에서 마지막에 고유값으로 스케일링 하는 것은 PCA 결과의 품질을 향상시키는 중요한 단계다. 초기 Whitening 단계에서는 데이터의 평균을 0 으로 하고 표준편차로 나누어 모든 특성이 동일한 중요성을 갖도록 한다. 이후 PCA 를 통해 고유값을 얻으면, 각 주성분은 데이터의 분산을 설명하는 정도를 나타낸다. 고유값을 사용하여 각 주성분을 스케일링 함으로써, 데이터의 분산을 더 많이 설명하는 주성분이 더 큰 가중치를 갖도록 한다. 즉, 중요한 주성분이 더 강조되고 덜 중요한 주성분은 상대적으로 줄어들게 되는 것이다. 이러한 처리로 모델의 성능이 향상될 수 있을 것이라 생각되기에 개선하였다.

또한, 고유값이 작아 스케일링 된 주성분은 데이터에서 차지하는 비중이 줄어들어 노이즈의 영향을 감소시킬 것이다. 이로 인해 모델이 실제 유용한 정보를 더 잘 학습할 수 있을 것이다. 따라서 PCA 후 고유값으로 스케일링하는 것은 데이터의 본질적인 특성을 잘 반영하고, 더 나은 예측 성능을 확보할 것이다.

```
Naive PCA: 0.5929824561403508
Whitening PCA: 0.6210526315789474
```

위를 확인하였을 때 스케일링을 적용하지 않은 것은 0.62 임을 볼 수 있지만 적용함으로 써

```
Naive PCA: 0.5894736842105263
Whitening PCA: 0.6701754385964912
```

더 Whitening 정확도가 더 좋아진 것을 확인할 수 있다.

마지막으로 QR 분해를 수행할 때 대각 성분을 제외한 성분들이 점점 반복할 수 록 0 에 가까워지는 형태를 위에서 확인할 수 있었다. 하지만 프로그램을 실행하는 시간상 100 번 이상은 테스트를 할 수 없었기에 반복 수를 더 늘린다면 개선될 것이라고 생각했다.

고찰

이번 PCA 및 Whitening 프로젝트를 진행하면서 여러 가지 중요한 경험을 쌓을 수 있었다. 특히 데이터 분석의 복잡성과 이를 해결하기 위한 다양한 접근 방식에 대해 깊이 이해하게 되었던 경험이다.

먼저 PCA 구현 과정에서의 어려움이 있었다. 고차원 데이터를 효과적으로 다루는 것은 쉽지 않았고, 특히 고유벡터와 고유값을 계산하는 과정에서 수치적 불안정성이 발생하는 경우가 많았다. 이를 해결하기 위해 수정된 오차 범위를 계산하면서 특정 조건을 주는 Gram-Schmidt 방법을 도입했지만, 여전히 직교 벡터를 구하는 데 어려움이 있었다. 이러한 과정을 통해 수치 해석 알고리즘의 중요성을 느끼게 되었고, 데이터 분석에서의 정확성이 얼마나 중요한지를 느꼈다. 다음으론 Whitening 과정에서의 시행착오도 있었다. 초기에는 평균을 제거하고 표준편차로 나누는 방법만을 고려했으나, 고유값으로 데이터를 정규화하는 방법이 더 효과적이라는 것을 공부하고 알게 되었다. 이 과정에서 데이터 전처리의 중요성을 느끼는 순간이었다. 마지막으로 이론과 실제의 괴리를 느꼈다. 학문적으로 배운 내용은 이해했지만, 실제로 구현하면서 발생하는 다양한 문제를 해결하는 것은 별개의 과정이라는 것을 깊이 느꼈다. 이론과 실제를 연결하는 데에는 많은 연습과 경험이 필요하다는 것을 깨닫았다.

결론적으로, 이번 프로젝트는 데이터 분석의 복잡성을 체험하고, 문제 해결 능력을 키울 수 있는 경험이었다. 앞으로의 프로젝트에서도 이러한 경험을 바탕으로 더욱 깊이 있는 분석을 수행할 수 있도록 노력해야 한다고 생각했다.

Reference

인공지능 강의자료

김상목 교수님 선형대수 강의자료

[머신러닝] 이미지 분석 : PCA + knn : 네이버 블로그 (naver.com)