Numerical Methods

HW04(Face Recognition)

담당교수 : 심동규 교수님

학 번 : 2019202050

성 명 : 이강현

Introduction

이번 과제는 image Classification에 대해 배우고 분류에 도움이 될 수 있는 feature representation 기법인 PCA,LDA를 구현해본다. 제공된 데이터셋에 PCA,LDA를 적용하고 분류 모델을 학습시켜 주성분 수에 따른 두 기법의 성능을 비교해보고 둘의 차이와 적용 의의에 대해 생각해본다. 추가적으로 다양한 분류모델을 사용하여 성능을 비교하여 해당 데이터셋에 어떤 기법이 더 유용한지에 대해 생각해본다.

Face Recognition Method

<PCA>

고차원의 데이터를 저차원의 데이터로 환원시키는 기법

2 차원으로 변환되게 PCA 를 적용한다면 고차원의 데이터를 저차원으로 변환하기 위해 직교 변환을 사용하고 가장 분산이 큰 축을 첫번째 주성분, 두 번째로 큰 축을 두번째 주성분으로 놓이도록 새로운 좌표계를 만들고 데이터를 선형 변환한다.

PCA 는 다음과 같은 순서로 진행된다.

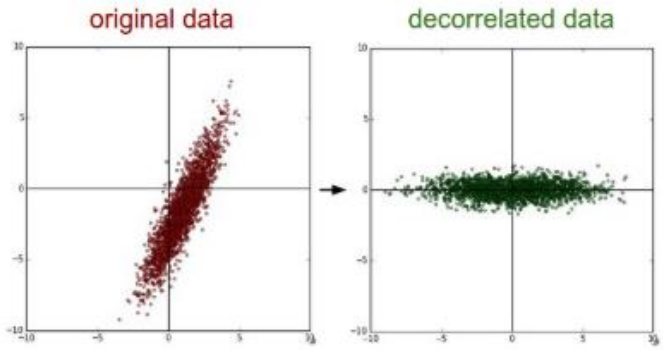
데이터에서 평균을 빼 데이터가 중앙에 위치하게 끔 하고 covariance 를 계산한다. Covariance matrix 에서 eigenvector 와 eigenvalue 를 계산하고 이렇게 구한 eigenvector 들을 eigenvalue 의 크기순으로 정렬하고 가장 큰 벡터들부터 새로운 좌표 축으로서의 역할을 한다. 이 좌표축에 데이터를 projection 하면 데이터를 더 적은 feature 로 표현 가능해진다.

<LDA>

주어진 데이터의 클래스간 분산은 최대화하고 클래스 내 분산은 최소화하여 데이터를 변환시키는 기법

LDA는 다음과 같은 순서로 진행된다.

데이터에서 평균을 빼 데이터를 표준화하고 클래스간 분산과 클래스내 분산을 계산한다. 클래스 내 분산 행렬의 역행렬과 클래스 간 분산 행렬을 곱하여 총 분산에 대한 비율이 가장 큰 특이벡터을 찾는다. 그 후 특이값 분해를 이용해 클래스간 분산행렬과 클래스내 분산행렬의 고유값과 고유벡터를 찾는다. 고유값의 크기순으로 정렬된 고유벡터에 대해 데이터를 projection하여 데이터를 더 적은 feature로 표현한다.



Experiments

실행환경은 window 10, python3.9.18, numpy 1.19.5, matplotlib 3.6.2, pytorch 2.1.2 환경에서 진행

먼저 데이터 전처리를 실시하였다.

데이터는 6개의 클래스로 이루어진 데이터이며 훈련데이터는 클래스 별로 8개, 테스트데이터는 클래스 별로 2개씩 이루어져 있다. 또 사진은 92x112의 크기이다. PCA와 LDA에 사진을 이용하기 위해 데이터를 sample x 1d 사진데이터 꼴의 shape로 만들어야 했다. 따라서 reshape을 진행하였는데 92x112의 크기와 RGB값까지 곱한다면 훈련데이터는 48 x 30912, 테스트데이터는 12 x 30912라는 shape을 가지게 된다. 이는 너무 큰 연산량을 요하기 때문에 너무 느렸기에 64x64로 사진데이터를 transform하여 로드하였다. 결과적으로 데이터셋은 아래와 같은 shape를 가지게 되었다.

```
<train data>
torch.Size([48, 3, 64, 64])

<test data>
torch.Size([12, 3, 64, 64])

<standardized_train_data>
(48, 12288)

<standardized_train_data_lda>
(48, 12288)
```

구현에 있어 전반적인 흐름은 PCA와 LDA를 알고리즘대로 그대로 구현하였다. 평균,공분산행렬,고유값분해와 같은 수학적인 연산은 numpy 라이브러리를 이용하였다. 한가지 이슈는 LDA를 구할 때 특이행렬이 만들어지지 않는 경우가 생겼는데 아래와 같다.

```
Traceback (most recent call last):
  File "C:\Users\dlrkd\OneDrive\3-2\수치해석\HW4\PCA_LDA.py", line 100, in <module>
    eigenvalues_lda, eigenvectors_lda = np.linalg.eigh(np.linalg.inv(class_scatter_matrix_lda).dot(classes_scatter_matrix_lda))
  File "<__array_function__ internals>", line 5, in inv
  File "C:\Users\dlrkd\anaconda3\envs\py39\lib\site-packages\numpy\linalg\linalg.py", line 546, in inv
    ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
  File "C:\Users\dlrkd\anaconda3\envs\py39\lib\site-packages\numpy\linalg\linalg.py", line 88, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

이는 클래스간 분산 행렬의 역행렬과 클래스 내 분산 행렬을 곱하는 과정에서 특이행렬이 만들어져 역행렬을 구할 수 없기 때문에 발생하는 오류였다. 따라서 작은 항을 추가하여 특이행렬이 만들어지지 않도록 하였다.

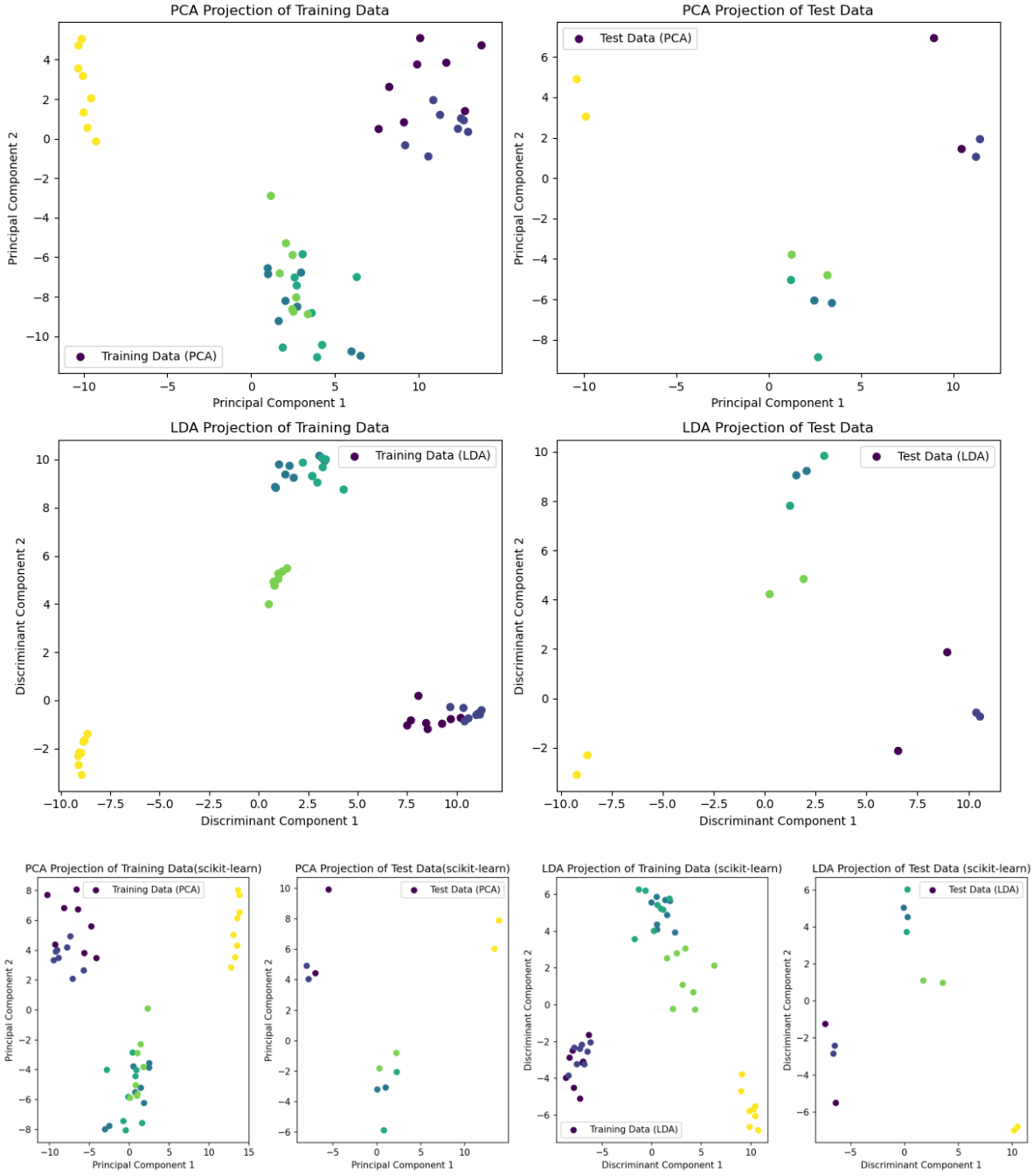
구현한 PCA,LDA와 라이브러리로 존재하는 PCA,LDA를 비교하여 구현유무를 파악하였다.

정확도는 로지스틱 회귀 모델을 이용하였고 이때 주성분의 개수는 2로 고정하였다.

성능과 산점도를 비교해본다.(상: 라이브러리, 하: 구현)

```
cumulative_explained_variance_ratio_threshold:0.95
num_components(PCA):2
num_components(LDA):2
Accuracy (PCA): 0.5833333333333334
Accuracy (LDA): 0.75
```

```
Logistic_Regression_Accuracy(PCA): 0.5833333333333334

Logistic_Regression_Accuracy(LDA): 0.8333333333333334
```

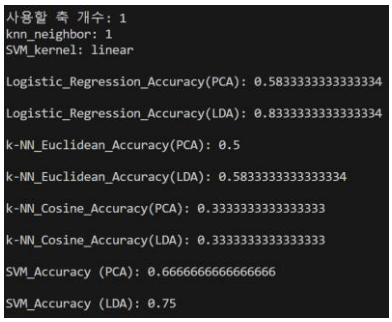


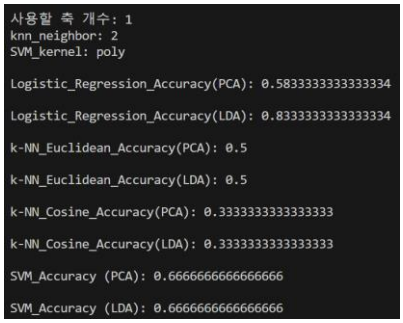
주성분이 2이고 로지스틱 회귀를 이용한 경우에는 구현한 것이 더 성능이 좋게 나왔다. 산점도를 확인해보니 PCA는 라이브러리와 첫번째 주성분 기준으로 반전되어 있는 것으로 보아 전체 샘플들의 분산이 가장 큰 곳으로 잘 뽑힌 것을 확인할 수 있다. LDA의 경우 오히려 더 클래스내 분산이 줄어들었고 첫번째 주성분의 분산정도는 -10부터 10정도로 비슷하지만 두번째 주성분의 분산에서 라이브러리와는 다른 성분을 뽑은 듯 보였다.

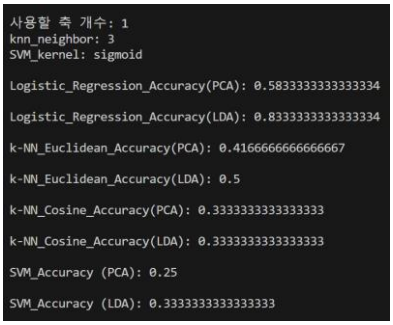
이 경우에는 클래스내 분산에 이점이 성능에 반영된 듯 보였다. 구현은 목적에 맞게 잘 되었다고 판단하고 이제부터는 PCA와 LDA간 주성분의 개수에 따른 성능비교와 다양한 분류모델을 활용하여 성능을 비교해보려고 한다.

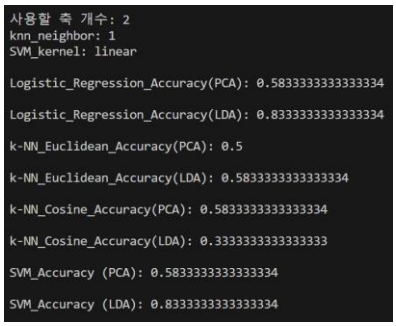
분류모델은 logistic Regression, knn(Euclidean),knn(cosine),SVM(linear,polynomial,sigmoid) 총 4개를 사용하여 성능을 측정하였다. 이에 축 개수를 1,2,3,4,28로 설정하여 비교하였다. 4와 28을 선택한 이유는 아래에서 설명한다.

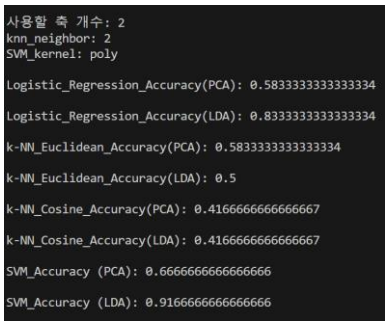
아래는 전체 성능 결과이다.

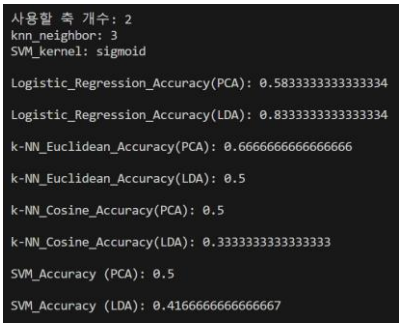


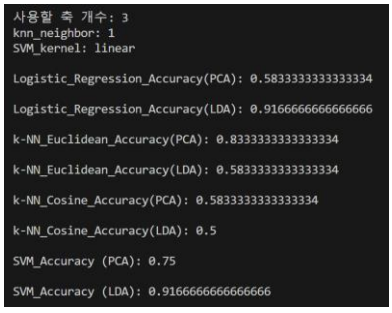


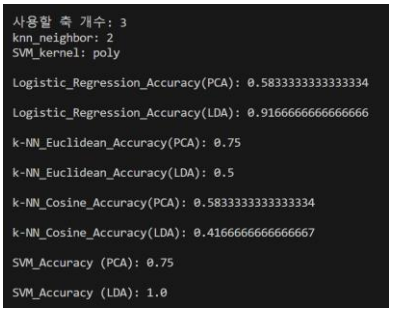


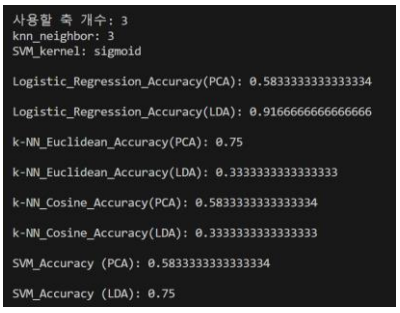


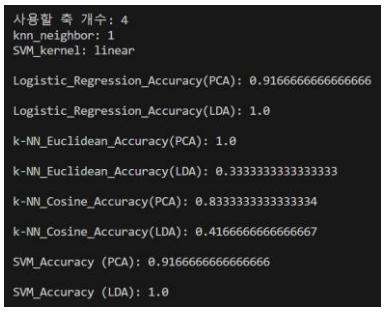


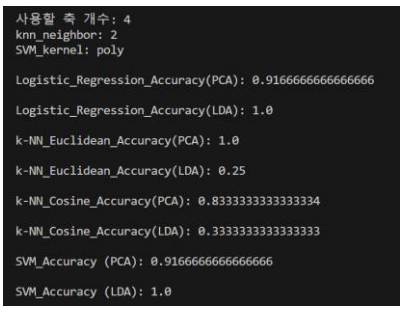


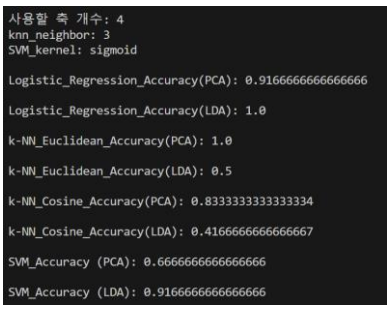










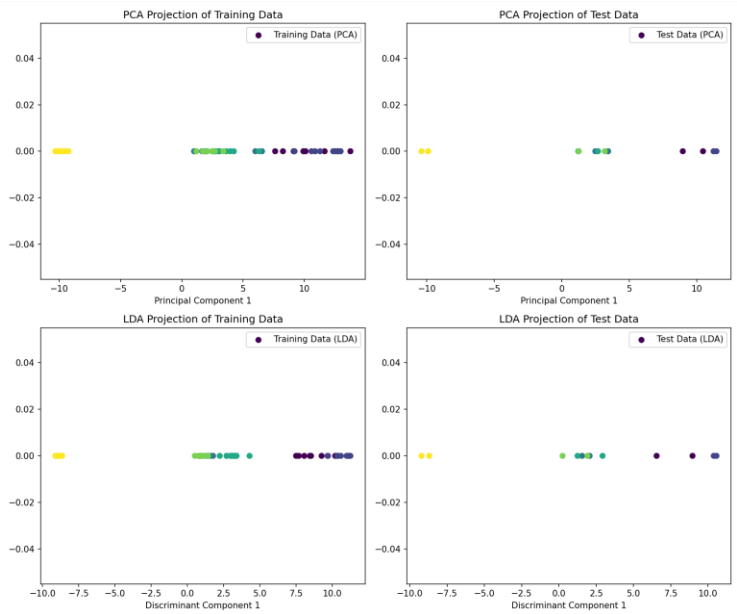


```
사용할 축 개수: 28
knn_neighbor: 1
SVM_kernel: linear

Logistic_Regression_Accuracy(PCA): 1.0

Logistic_Regression_Accuracy(LDA): 1.0

k-NN_Euclidean_Accuracy(PCA): 1.0

k-NN_Euclidean_Accuracy(LDA): 0.4166666666666667

k-NN_Cosine_Accuracy(PCA): 0.9166666666666666

k-NN_Cosine_Accuracy(LDA): 0.4166666666666667

SVM_Accuracy (PCA): 1.0

SVM_Accuracy (LDA): 1.0
```

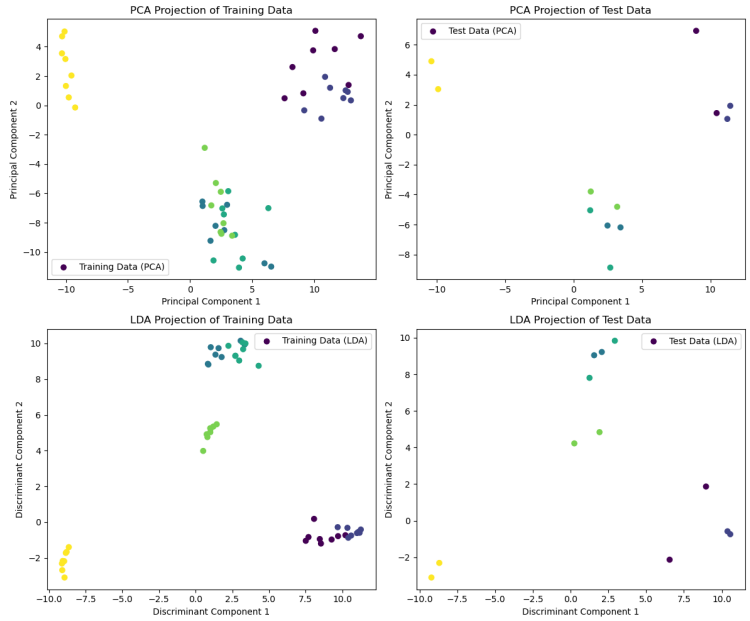
```
사용할 축 개수: 28
knn_neighbor: 2
SVM_kernel: poly

Logistic_Regression_Accuracy(PCA): 1.0

Logistic_Regression_Accuracy(LDA): 1.0

k-NN_Euclidean_Accuracy(PCA): 1.0

k-NN_Euclidean_Accuracy(LDA): 0.4166666666666667

k-NN_Cosine_Accuracy(PCA): 0.9166666666666666

k-NN_Cosine_Accuracy(LDA): 0.4166666666666667

SVM_Accuracy (PCA): 1.0

SVM_Accuracy (LDA): 1.0
```

```
사용할 축 개수: 28
knn_neighbor: 3
SVM_kernel: sigmoid

Logistic_Regression_Accuracy(PCA): 1.0

Logistic_Regression_Accuracy(LDA): 1.0

k-NN_Euclidean_Accuracy(PCA): 1.0

k-NN_Euclidean_Accuracy(LDA): 0.3333333333333333

k-NN_Cosine_Accuracy(PCA): 0.9166666666666666

k-NN_Cosine_Accuracy(LDA): 0.4166666666666667

SVM_Accuracy (PCA): 0.9166666666666666

SVM_Accuracy (LDA): 0.9166666666666666
```

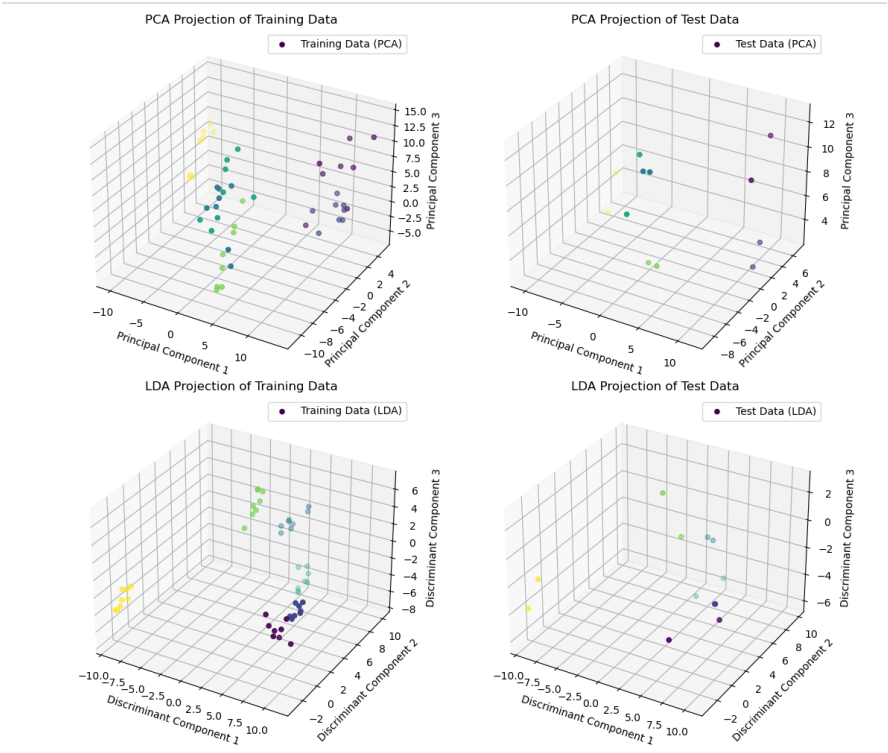
주성분이 1개일 때 plot



주성분이 2개일 때 plot



주성분이 3개일 때 plot



전체 성능 통계

model			neighbor = 1				neighbor = 2				neighbor = 3				linear		polynomial		sigmoid	
축	Logistic(PCA)	Logistic(LDA)	KNN(Euclidean_PCA)	KNN(Euclidean_LDA)	KNN(Cosine_PCA)	KNN(Cosine_LDA)	KNN(Euclidean_PCA)	KNN(Euclidean_LDA)	KNN(Cosine_PCA)	KNN(Cosine_LDA)	KNN(Euclidean_PCA)	KNN(Euclidean_LDA)	KNN(Cosine_PCA)	KNN(Cosine_LDA)	SVM(PCA)	SVM(LDA)	SVM(PCA)	SVM(LDA)	SVM(PCA)	SVM(LDA)
1	0.58	0.83	0.5	0.58	0.33	0.33	0.5	0.5	0.33	0.33	0.42	0.5	0.33	0.33	0.67	0.75	0.67	0.67	0.25	0.33
2	0.58	0.83	0.5	0.58	0.58	0.33	0.58	0.5	0.42	0.42	0.67	0.5	0.5	0.33	0.58	0.83	0.67	0.92	0.5	0.42
3	0.58	0.92	0.83	0.58	0.58	0.5	0.75	0.5	0.58	0.42	0.75	0.33	0.58	0.33	0.75	0.92	0.75	1	0.58	0.75
4	0.92	1	1	0.33	0.83	0.42	1	0.25	0.83	0.33	1	0.5	0.83	0.42	0.92	1	0.92	1	0.67	0.92
28	1	1	1	0.42	0.92	0.42	1	0.42	0.92	0.42	1	0.33	0.92	0.42	1	1	1	1	0.92	0.92
평균	0.732	0.916	0.766	0.498	0.648	0.4	0.766	0.434	0.616	0.384	0.768	0.432	0.632	0.366	0.784	0.9	0.802	0.918	0.584	0.668

축의 개수를 1,2,3,4,28로 한 이유는 1,2,3개는 시각화가 가능하다는 점 때문에 선정하였고 4는 LDA에서 28은 PCA에서 각각 데이터의 분포를 95퍼센트 설명할 수 있는 주성분의 개수이기 때문에 선정하였다.

PCA 평균	0.7098
LDA 평균	0.5916

전체 모델들의 분류 성능을 토대로 전체 평균을 내어보았을 때 PCA가 더 좋은 성능을 보였다.

클래스 별로 모여있는 LDA가 KNN에서 PCA보다 더 좋은 성능을 보일거라는 예상과는 달리 LDA는 KNN에서 PCA보다 항상 좋지 않은 성능을 보였고 그 이유는 LDA는 선형분리에 효과적이지만 사진 데이터는 비선형 데이터이기 때문에 혹은 KNN의 파라미터 의존성등 여러가지 이유 때문에 이러한 결과가 나온 것으로 예상할 수 있다. 또 KNN은 군집구조에 적합하고 선형적으로 잘 분리

되게끔 LDA를 적용한다 할지라도 고차원에서는 오히려 더 멀어지는 결과가 나올 수 있다는 예상이다. SVM과 logistic regression에서는 LDA가 항상 좋은 성능을 보였는데 그 이유는 먼저 logistic regression은 선형 분류 문제에 적합하게 끔 LDA가 데이터를 처리해주었다는 점에서 이점이 있었을 것으로 예상되고 SVM 또한 클래스 간 분리를 최대화한다는 LDA의 특성이 SVM에서 decision boundary를 더 명확하게 찾는데 도움을 주고 둘의 결합이 경계를 형성하는데 있어 더 강력하게 작용한 것이 아닐까 예상한다.

Conclusion

다양한 분류모델들을 이용해서 PCA와 LDA기법의 성능을 비교해보았다. 이미지와 같은 고차원데이터에는 CNN을 많이 적용해서 학습시키고 분류했는데 이러한 다양한 기법으로 차원을 축소시켜 연산량을 감소시키고 분류에도 좋은 결과를 보일 수 있다는 것을 알게 되었다. 클래스의 구분이 주 목적인 분류 문제에는 늘 LDA기법이 PCA보다 좋을 것으로 예상하였으나 분류모델이 무엇인지에 따라 오히려 PCA가 더 좋게 나오는 것을 확인하고 상황에 맞게 적절한 알고리즘을 사용하는 것이 좋겠다고 생각했다. 또한 이러한 여러가지 분류 모델을 적용해보면서 데이터셋의 특성을 예측할 수 있고 그러한 특성을 가지는 데이터셋에는 그에 맞는 특징추출 기법, 분류모델을 적용하는 것이 좋은 방법이라고 생각했다.