수치해석

HW04

담당교수 : 심동규 교수님

학 번 : 2021202058

성 명 : 송채영

1. Introduction

이번 과제는 Face Recognition, 얼굴 인식을 PCA(Principle Component Analysis)와 LDA(Linear Discriminant Analysis) 이 두가지 알고리즘을 활용해서 구현해보는 것이다. 모델을 훈련시켜 정확도를 비교해보고 eigen vector 를 변화시키며 차이점을 찾아본다. 추가적으로 distance-based classification(Euclidean distance, Cosine distance), ML-based classification(naïve-bayes classification, support vector machine)등의 분류 방법으로도 비교해본다.

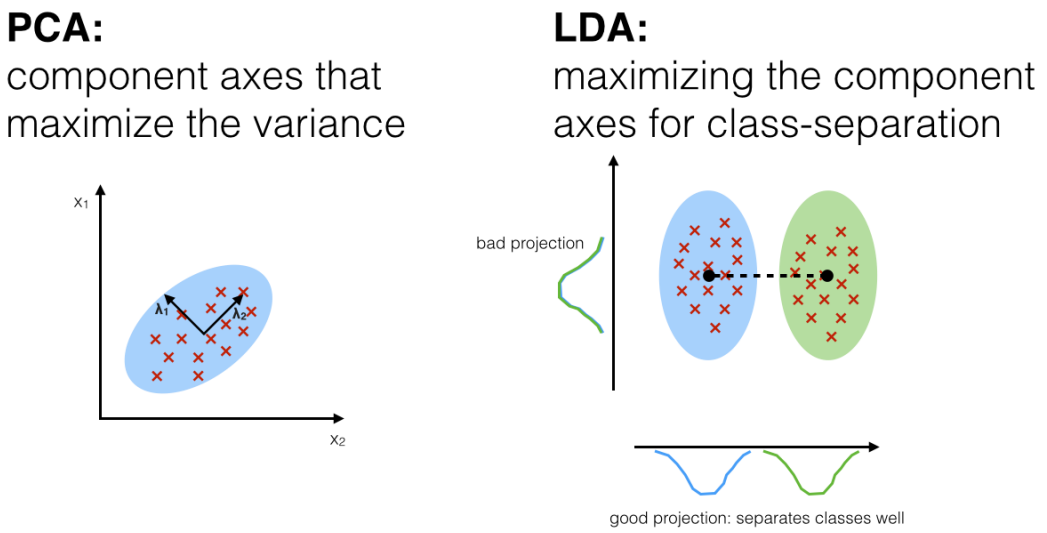
2. Face Recognition Method

Face Recognition, 얼굴 인식은 주어진 이미지나 비디오에서 얼굴을 감지하고 이를 식별하는 기술 중 하나이다. 그 중 Feature Representation 은 얼굴에서 중요한 정보를 추출해서 얼굴을 식별하기 위한 특징 벡터를 만드는 과정을 말한다. 이번 프로젝트에서는 두 가지 방법을 적용하는데 PCA 와 LDA 이다.

PCA 는 주성분 분석으로 데이터의 주성분을 찾은 후 그것을 이용해 차원을 축소하는 기법을 말한다. PCA 는 주어진 데이터의 분산을 최대화하는 방향으로 주성분을 찾아 차원을 줄이는데 사용된다. PCA 의 주요단계는 다음과 같다. 우선 데이터 정규화, Data Normalization 을 진행한다. PCA 를 적용하기 전에 데이터를 평균이 0 이 되도록 정규화하는 과정이다. 이후 공분산 행렬인 Covariance Matrix 를 계산한다. 정규화된 데이터를 이용해 공분산 행렬을 계산하고 이때 공분산은 두 변수 간의 관계를 나타내며 이 행렬은 데이터의 분산 및 상관관계를 나타낸다. 이어서 고유값 분해, Eigenvalue Decomposition 을 진행한다. 이 과정에서는 공분산 행렬을 고유값과 고유벡터로 분해하는 과정이다. 고유값은 새로운 축의 중요도를 나타내며 고유벡터는 해당 축의 방향을 나타낸다. 이어서 고유값 정렬, Eigenvalue sorting 을 한다. 이 과정에서는 고유값을 크기순으로 정렬해서 가장 큰 고유값에 대응하는 고유벡터가 가장 중요한 주성분이 된다. 마지막으로 주성분을 선택하는 selecting principle components 과정이다. 가장 중요한 주성분부터 차례로 선택하여 차원을 축소한다.

LDA 는 선형 판별 분석으로 클래스 간 분산과 클래스 내 분산의 비율을 최대화하여 데이터를 특징벡터로 투영하는 방법이다. LDA 는 클래스 간 분산은 최대화하고 클래스 내 분산은 최소화하여 클래스 간의 구분력을 높이는데 사용된다. LDA 의 주요단계는 다음과 같다. 우선 클래스 간 분산과 클래스 내 분산을 계산한다. 클래스 내 분산은 클래스 내의 데이터가 얼마나 흩어져 있는지를 나타내며 클래스 간 분산은 클래스 간의 차이를 나타낸다. 이어서 공분산 행렬의 역행렬을 계산한다. 이 과정은 LDA 에서 최적의 투영을

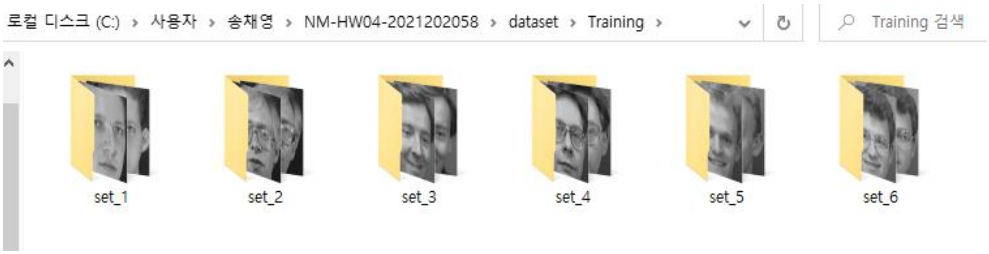
구하기 위해 필요하다. 다음으로 고유값 분해, Eigenvalue Decomposition 을 진행한다. 역행렬을 사용하여 고유값 분해를 수행하며 고유값은 LDA 에서 찾고자 하는 새로운 축의 중요도를 나타내고 고유벡터는 해당 축의 방향을 나타낸다. 다음으로 고유값 정렬을 진행한다. 고유값을 크기순으로 정렬하여 가장 큰 고유값에 해당하는 고유벡터를 선택하며 이는 클래스 간 구분력을 최대화하는 방향을 찾는 과정에 해당한다. 마지막으로 주성분을 선택한다. 가장 중요한 고유값에 대응하는 고유벡터를 사용하고 데이터를 새로운 축으로 투영한다.

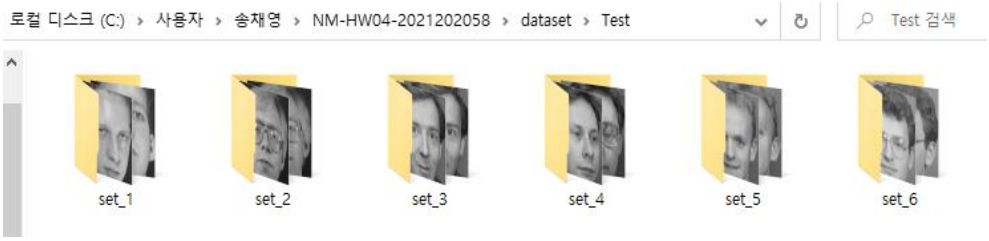


3. Experiments

- Image Classification

우선 Training data 와 Test data 이다. Training data 는 모델을 학습하는데 사용된 데이터이며 Test data 는 모델이 training 한 후 성능을 평가하는데 사용하였다. Test 는 1~12 까지의 사진들만 존재하였는데, class 분류를 위해 같은 사람끼리 같은 폴더에 분류해서 넣어주었다.





- PCA

우선 PCA 구현 방법에 대해 설명하겠다. 데이터의 평균을 계산하고 이를 데이터에서 빼줌으로써 데이터를 중앙화하였다. 이어서 공분산 행렬을 계산해주었다. Rowvar = false 는 각 열이 변수를 나타내도록 설정하였다. 이어서 고유값과 고유벡터를 계산한 후 고유값을 내림차순으로 정렬하였다. 상위 num_components 개의 고유값에 대응하는 고유벡터를 선택한 후 선택된 고유벡터로 데이터를 변환하도록 구현하였다.

```
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
])
```

이어서 기존의 이미지 크기는 시간이 너무 오래 걸려 빠르게 학습하기 위해 사진 크기를 64x64 로 조정하였다. 데이터셋은 ImageFolder 를 사용하여 루트 디렉토리를 지정해주었고 DataLoader 을 사용하여 training, test data set 을 불러왔다. Batch size 는 한번에 전체 데이터셋을 load 하도록 하였다.

```
train_dataloader = DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False)
```

```
Accuracy: 66.67%
```

```
train_dataloader = DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False)
```

```
Accuracy: 66.67%
```

첫 번째 case 는 shuffle = True 로 설정해주어 데이터를 섞은 경우고 두 번째 case 는 shuffle = False 로 설정하여 데이터를 섞지 않은 경우이다. 아래에서 살펴볼 예정이지만, eigen vector 의 개수는 1 이고 neighbors 의 개수가 1 일 때 가장 성능이 좋았는데 이 경우의 accuracy 를 뽑아 비교해보았다. 데이터의 순서가 중요하지 않다고 생각했기 때문에 사진을 섞어보았는데 애초의 데이터가 적기 때문에 accuracy 가 같게 나왔다고 생각한다. 따라서 이번 과제에서는 shuffle 의 여부가 accuracy 에 영향을 끼치지 않는다는 점을 알아 실험을 전부 shuffle = False 로 진행하였다.

```
# PCA 적용
X_train, y_train = next(iter(train_dataloader))
X_test, y_test = next(iter(test_dataloader))

num_components = 1  # Set the number of PCA components
X_train_pca = pca(X_train.view(X_train.size(0), -1).numpy(), num_components)
X_test_pca = pca(X_test.view(X_test.size(0), -1).numpy(), num_components)
```

이어서 구현한 PCA 를 적용하였다. 우선 1D 벡터로 데이터를 펼치고 구현한 pca 함수를 호출하여 데이터에 적용하였다. num_component 는 주성분의 수를 의미하며 과제의 조건에서 eigen vector 의 수를 변화시키며 accuracy 를 확인하라고 했으므로 해당 부분을 변화시켰다. 결과는 아래의 첨부하겠다.

다음으로 모델을 학습시키는데 K-Nearest Neighbors(KNN) 모델을 사용하였다. KNN 은 지도학습 알고리즘으로 데이터 포인트를 가장 가까운 이웃들과 비교해 분류 또는 회귀 문제를 해결한다. 이때 코드에서 n_neighbors 는 이웃의 수를 나타내며 예측할 때 가장 가까운 몇 개의 이웃을 고려할지 결정하는 역할을 한다. 이부분도 여러 test 를 해보며 적합한 개수를 찾아보았다. 결과는 아래의 첨부하겠다.

	1	2	3
1	66.67	58.33	58.33
2	41.67	50.00	50.00
3	33.33	33.33	33.33

위 표에서 가로축은 n_neighbors 에 해당하고 세로축은 num_components 에 해당한다. 결과적으로 eigen vector 가 1 이고 가장 가까운 1 개의 이웃을 고려할 때 accuracy 가 가장 좋게 나오는 것을 확인할 수 있었다. 1 부터 시작한 이유는, 처음 num_components 를 50 으로 설정하고 실행했을 때 accuracy 가 약 16 정도가 나왔기 때문에 확 내려서 1 부터 시작해보았다.

Scikit-learn 라이브러리를 사용해서 PCA 를 직접 구현하지 않고 테스트해보았는데 같은 코드를 실행해본 결과 아래 사진과 같이 accuracy 가 75 가 나오는 것을 확인할 수 있었다. 라이브러리를 사용했을 때는 위 표에서 수행하였던 경우로 그대로 실행해보았는데 정확도가 거의 약 75 가 나오는 것을 확인하였다.

```
In [94]: import os
import numpy as np
from PIL import Image
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
import torch
from torchvision import transforms, datasets
from torch.utils.data import DataLoader

# 데이터셋 가져오기
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
])

train_dataset = datasets.ImageFolder(root="dataset/Training", transform=transform)
test_dataset = datasets.ImageFolder(root="dataset/Test", transform=transform)

train_dataloader = DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False)

# PCA 적용
X_train, y_train = next(iter(train_dataloader))
X_test, y_test = next(iter(test_dataloader))

num_components = 1  # Set the number of PCA components

# Scikit-learn PCA 사용
pca_model = PCA(n_components=num_components)
X_train_pca = pca_model.fit_transform(X_train.view(X_train.size(0), -1).numpy())
X_test_pca = pca_model.transform(X_test.view(X_test.size(0), -1).numpy())

# KNN 모델 훈련
knn_model = KNeighborsClassifier(n_neighbors=1)
knn_model.fit(X_train_pca, y_train)

# 예측 및 정확도 계산
y_pred = knn_model.predict(X_test_pca)
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy: {accuracy * 100:.2f}%")

Accuracy: 75.00%
```

Visual studio, c++로도 구현해보았는데 이 경우에는 아래 사진과 같이 66.66 정도로 파이썬으로 구현했을 때와 비슷한 결과가 나오는 것을 확인할 수 있었다.

```
for (int i = 1; i <= 12; ++i)
{
    string filename = "dataset/Test/" + to_string(i) + ".bmp";
    Mat image = imread(filename, IMREAD_GRAYSCALE);

    if (image.empty())
    {
        cerr << "Could not open or find the image: " << filename << endl;
        return -1;
    }
```

```
Microsoft Visual Studio 디버그 콘솔
Accuracy: 66.6667%

C:₩Users₩송채영₩source₩repos₩NM-HW04-2021202058₩NM-HW04-2021202058₩x64₩Debug₩NM-HW04-2021202058.exe(프로세스 16196개)이(
가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

- LDA

우선 LDA 구현 방법에 대해 설명하겠다. 각 클래스에 대한 데이터 평균 벡터를 계산한 후 전체 평균 벡터를 계산하였다. 이후 클래스 간 산포 행렬과 클래스 내 산포 행렬의 고유값과 고유벡터를 계산한 후 고유값을 내림차순으로 정렬 후 상위 num_components 개의 고유벡터를 선택하였다. 마지막으로 선택된 고유벡터의 데이터를 변환하여 LDA 결과를 얻도록 구현하였다. LDA 의 경우 데이터 셋을 가져오고 KNN 모델을 훈련하고 정확도를 계산하는 코드는 PCA 와 동일하게 구현하였다. 구현한 LDA 함수를 적용하는 부분도 동일하게 데이터를 1D 벡터로 펼친 후 함수를 호출하여 LDA 를 적용해주었다. 이때 PCA 와의 결과를 비교하기 위해 num_components = 1, n_neighbors = 1 로 설정하여 실행하였다.

```
Accuracy: 16.67%
```

Accuracy 가 16.67 이 나왔는데 LDA 의 경우 클래스 수가 적거나 클래스 간의 차이가 미미한 경우 성능 향상이 미미할 수 있다는 단점이 있어, class 수가 적기 때문에 성능이 좋지 않게 나왔다고 생각한다.

라이브러리를 사용해서 accuracy 를 뽑아낸 결과 66.67 이 나왔지만 PCA 라이브러리를 활용해서 accuracy 를 뽑아낸 결과보다는 좋지 않게 나왔다는 것을 확인할 수 있었다.

```
Accuracy: 66.67%
```

4. Conclusion

이번 과제를 진행하면서 기본적인 것들을 구현하는 것이 더 힘들었던 것 같다. 처음에는 visual studio 에서 c++로 구현해보았는데, opencv 라이브러리를 활용하여 이미지를 불러오고 PCA 를 사용할 수 있었고 eigen 라이브러리를 통해 선형대수학에서 사용하는 계산들, 즉 PCA 를 구하는데 활용하는 계산들을 함수로 불러와 쓸 수 있었다. 하지만 이 과정에서 include 만 해서 쓸 수 있는 것이 아니라 사이트에서 해당 라이브러리를 다운 받은 후 프로젝트 속성에서 그 다운 받은 것을 넣어주어야 하는 과정이 따로 필요했다. 이 경우에는 구현하는 것에는 문제가 크게 없지만 제출 시 문제가 생길 것 같아 파이썬으로 다시 구현하였다. 데이터를 불러오는 부분에서 가장 시간이 많이 걸렸는데, 데이터가 저장되어 있는 경로를 제대로 인식을 못하는 것 같았다. 결국 인공지능 스터디를 하면서 배웠었던 ImageFolder 와 DataLoader 를 사용해서 데이터를 불러오는데 성공할 수 있었다. PCA 를 구현하는 과정은 인공지능 때도 한 번 했었고, 계산도 여러 번 해보아 개념적으로 이해가 됐기 때문에 오래 걸리지 않았다. 또한 KNN 모델도 인공지능 과제 때 사용했기 때문에 모델을 적용하는 부분도 어렵지 않았다. 결과가 라이브러리를

사용할 때 보다 좋게 나오지는 않았지만 생각했던 것 보다 좋은 결과가 나왔던 것 같다. 또한 eigen vector 의 개수에 따라 accuracy 의 차이가 꽤 크다는 점이 신기했던 것 같다. LDA 부분은 이해하는데 굉장히 오래 걸렸던 것 같다. 개념적으로는 이해가 됐지만 구현하면서 생각했던 것만큼 결과도 잘 나오지 않았고 size 문제도 계속 있었기 때문에 힘들었다. 비록 결과는 좋게 나오지 않았지만 LDA 에 대해 이해할 수 있었고, PCA 와 LDA 가 각각 상황에 따라 장단점이 있는 것 같다는 점도 느꼈다.