인공 신경망

(주)인피닉스 – 강호용 연구원

Infinyx

선형 분류-퍼셉트론 (Perceptron)

Infinyx
THINK LIFE SYNC AI

선형 분류 (Linear Classification) – 퍼셉트론 개념 소개

- 퍼셉트론이란 무엇인지, 어떤 문제를 해결할 수 있는지 ?

인공신경망(ANN)의 한 종류로, 이진 분류 문제를 해결하는 데 사용됩니다. 이진 분류는 두 가지 범주 중 하나에 데이터 포인트를 할당하는 작업입니다. 이론적으로 퍼셉트론은 하나 이상의 입력 값을 받아서 하나의 출력 값을 생성합니다.

초기 인공지능 분야의 발전을 이끈 프랑크 로젠블라트(Frank Rosenblatt)가 1957년에 개발했습니다. 이후, 퍼셉트론은 인공신경망의 핵심 구성 요소로 남았습니다.

머신 러닝에서 지도 학습(Supervised Learning)을 수행하는 알고리즘 중 하나입니다. 지도 학습은 입력 데이터와 해당 데이터의 정답(라벨)을 모두 제공하여 학습을 진행하는 방식입니다.

입력 데이터를 받아 가중치를 곱하고 편향을 더한 값을 출력으로 계산합니다. 출력 값은 미리 설정한 임계치를 기준으로 0 또는 1로 변환됩니다. 이러한 퍼셉트론은 선형 분류 문제에서 사용할 수 있습니다.

선형 분류 (Linear Classification) – 퍼셉트론 → 동작 원리

입력 신호를 받아 가중치와 곱한 값을 모두 더한 뒤, 그 값이 임계치(threshold)를 넘으면 1을 출력하고, 넘지 않으면 0을 출력하는 이진 분류 모델입니다. 즉, 입력 신호와 가중치의 곱의 합이 임계치를 넘으면 양성 클래스(1)에 속하고, 그렇지 않으면 음성 클래스(0)에 속하는 것으로 결정됩니다.

초기에는 가중치를 무작위로 초기화하고, 입력 데이터를 하나씩 주입하여 결과를 출력합니다. 그 결과를 바탕으로 오차를 계산하여 가중치를 조정합니다. 이 과정을 반복하면서 가중치를 최적화시키고, 입력 데이터를 정확하게 분류할 수 있도록 합니다. 이러한 학습 과정을 퍼셉트론 학습 규칙(Perceptron Learning Rule)이라고 합니다.

하나의 층으로 이루어진 단층 퍼셉트론과 여러 개의 층으로 이루어진 다층 퍼셉트론으로 나눌 수 있습니다. 단층 퍼셉트론은 선형 분리 가능한 문제에 대해서만 사용할 수 있으며, XOR과 같이 비선형 문제를 해결할 수 없습니다. 따라서 다층 퍼셉트론을 사용하여 비선형 분류 문제를 해결할 수 있습니다.

선형 분류 (Linear Classification) – 퍼셉트론

- 퍼셉트론 학습 알고리즘의 개념 설명

  퍼셉트론 학습 알고리즘은 지도 학습(Supervised Learning) 방법 중 하나로, 데이터의 특징(feature)과 레이블(label)이 주어졌을 때 이를 기반으로 모델의 가중치(weight)를 업데이트하여 최적의 모델을 학습하는 알고리즘입니다.

선형 분류 (Linear Classification) – 퍼셉트론

- 퍼셉트론 학습 알고리즘은 다음과 같은 단계로 이루어집니다.

단계 명칭	퍼셉트론 학습 알고리즘 단계 소개
초기화	✓ 가중치 w와 편향 b를 0 혹은 랜덤한 값으로 초기화합니다.
학습 데이터	✓ 학습에 필요한 데이터를 가져옵니다.
예측 값 계산	- 가져온 데이터의 특징과 가중치를 곱한 값을 더한 후, 편향을 더하여 예측값을 계산합니다. - 계산된 예측값이 0보다 크면 1, 0보다 작으면 -1로 분류합니다.
가중치 업데이트	- 계산된 예측값과 실제 레이블의 차이를 구합니다. - 이 차이를 이용하여 가중치와 편향을 업데이트합니다. - 예를 들어, 실제 레이블이 1이지만 예측값이 -1일 경우, 가중치 w와 편향 b를 다음과 같이 업데이트합니다.
모든 데이터에 대해 반복	✓ 위의 과정을 모든 데이터에 대해 반복합니다. ✓ 반복 횟수나 에포크(epoch)는 사용자가 지정합니다.
예측	✓ 학습된 모델을 이용하여 새로운 데이터의 레이블을 예측합니다.

 # PyTorch를 사용하여 단층 퍼셉트론을 학습하는 간단 실습

```
import torch

# 훈련 데이터
x = torch.Tensor([[0, 0], [0, 1], [1, 0], [1, 1]])
y = torch.Tensor([[0], [0], [0], [1]])

# 모델 초기화
w = torch.randn(2, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

# 하이퍼파라미터 설정
lr = 0.1
epochs = 10000
```

x와 y는 각각 입력값과 목표값을 나타냅니다.
>> 2차원 벡터를 사용하여 입력값을 표현합니다.

w는 가중치를 나타내는 변수입니다.
처음에는 무작위로 초기화됩니다.

lr은 학습률(learning rate)을 나타내며, 학습이 얼마나 빠르게 진행될지를 조절합니다.

PyTorch를 사용하여 단층 퍼셉트론을 학습하는 간단 실습

```
# 훈련
for epoch in range(epochs):
    # 순전파
    y_pred = torch.sigmoid(x.mm(w) + b)

    # 손실 함수 계산
    loss = torch.mean((y - y_pred)**2)

    # 기울기 계산 및 가중치 갱신
    loss.backward()
    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

        # 가중치 변화율 초기화
        w.grad.zero_()
        b.grad.zero_()

    if (epoch+1) % 1000 == 0:
        print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')
```

y_pred는 입력값에 대한 퍼셉트론의 출력값을 나타냅니다. 이 값은 입력값 x와 가중치 w의 내적을 계산하여 구합니다.

loss는 평균 제곱 오차(Mean Squared Error)를 나타내는 손실 함수입니다. 이 함수는 실제 출력값 y와 예측값 y_pred 간의 차이를 계산합니다.

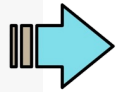
loss.backward()는 손실 함수를 미분하여 각 변수의 기울기(gradient)를 계산합니다.

w.grad는 w 변수의 기울기를 나타내며, lr * w.grad는 학습률을 곱한 값을 이용하여 가중치 w를 갱신합니다

이러한 과정을 여러 번 반복하면서 가중치 w가 최적의 값으로 수렴하게 됩니다. 이후에는 학습된 퍼셉트론을 사용하여 새로운 입력값에 대한 예측값을 계산할 수 있습니다.

 # PyTorch를 사용하여 단층 퍼셉트론을 학습하는 간단 실습

```python
# 모델 평가
with torch.no_grad():
    y_pred = torch.sigmoid(x.mm(w) + b)
    y_pred = torch.where(y_pred > 0.5, 1, 0)
    accuracy = (y_pred == y).float().mean()

print(f'Accuracy: {accuracy.item():.4f}')
```



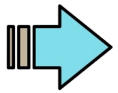
```
Epoch 1000/10000, Loss: 0.0471
Epoch 2000/10000, Loss: 0.0280
Epoch 3000/10000, Loss: 0.0193
Epoch 4000/10000, Loss: 0.0145
Epoch 5000/10000, Loss: 0.0116
Epoch 6000/10000, Loss: 0.0096
Epoch 7000/10000, Loss: 0.0081
Epoch 8000/10000, Loss: 0.0070
Epoch 9000/10000, Loss: 0.0062
Epoch 10000/10000, Loss: 0.0055
Accuracy: 1.0000
```

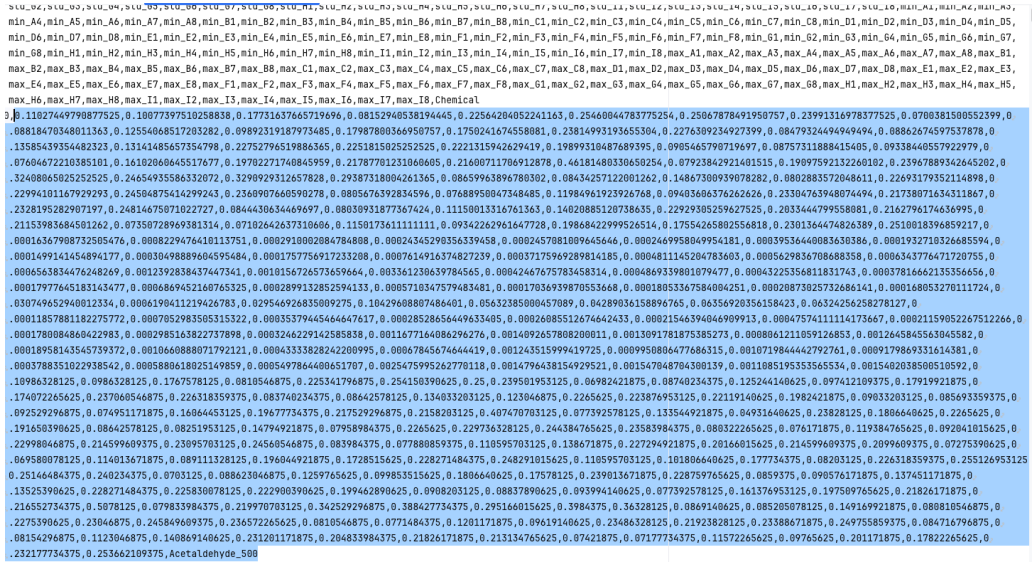
선형 분류 (Linear Classification) – 퍼셉트론

한계 / 필요성	설명
퍼셉트론 학습 알고리즘의 한계	단층 퍼셉트론은 선형 분리 문제만 해결할 수 있기 때문에, 비선형 분리 문제에 대해서는 제대로 동작하지 않습니다. 또한, 퍼셉트론 학습 알고리즘에서는 분류 오류를 줄이는 가중치 조정을 반복적으로 수행하면서 최적의 가중치 값을 찾아야 하는데, 이 과정에서 학습 데이터에 대해 수렴하지 않는 경우가 있습니다.
다중 층 퍼셉트론 필요성	다중 층 퍼셉트론(multilayer perceptron)은 여러 개의 은닉층(hidden layer)을 추가하여 비선형 문제에 대한 해결이 가능하도록 하였습니다. 은닉층을 추가하면서 입력 데이터를 비선형으로 변환하는 함수를 사용하여 다양한 패턴을 학습할 수 있습니다. 또한, 역전파 알고리즘을 이용하여 학습하면 더욱 정확한 분류 모델을 만들 수 있습니다. 따라서, 다중 층 퍼셉트론은 단층 퍼셉트론의 한계를 극복하고, 보다 복잡한 문제에 대한 해결이 가능하도록 하였습니다.

Infinyx

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현





이 데이터셋은 케미컬 감지 플랫폼에서 6개의 다른 위치에서 측정된 10가지 화학 가스 물질에 대한 시간 순서 기록입니다.

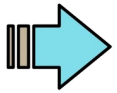
>> 목표 : 10가지 클래스의 화학 물질을 구분

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

- **데이터 소개**

```
data = df[[
    'mean_A1', 'mean_A2', 'mean_A3', 'mean_A4', 'mean_A5', 'mean_A6', 'mean_A7', 'mean_A8',
    'mean_B1', 'mean_B2', 'mean_B3', 'mean_B4', 'mean_B5', 'mean_B6', 'mean_B7', 'mean_B8',
    'mean_C1', 'mean_C2', 'mean_C3', 'mean_C4', 'mean_C5', 'mean_C6', 'mean_C7', 'mean_C8',
    'mean_D1', 'mean_D2', 'mean_D3', 'mean_D4', 'mean_D5', 'mean_D6', 'mean_D7', 'mean_D8',
    'mean_E1', 'mean_E2', 'mean_E3', 'mean_E4', 'mean_E5', 'mean_E6', 'mean_E7', 'mean_E8',
    'mean_F1', 'mean_F2', 'mean_F3', 'mean_F4', 'mean_F5', 'mean_F6', 'mean_F7', 'mean_F8',
    'mean_G1', 'mean_G2', 'mean_G3', 'mean_G4', 'mean_G5', 'mean_G6', 'mean_G7', 'mean_G8',
    'mean_H1', 'mean_H2', 'mean_H3', 'mean_H4', 'mean_H5', 'mean_H6', 'mean_H7', 'mean_H8',
    'mean_I1', 'mean_I2', 'mean_I3', 'mean_I4', 'mean_I5', 'mean_I6', 'mean_I7', 'mean_I8',
    'std_A1', 'std_A2', 'std_A3', 'std_A4', 'std_A5', 'std_A6', 'std_A7', 'std_A8',
    'std_B1', 'std_B2', 'std_B3', 'std_B4', 'std_B5', 'std_B6', 'std_B7', 'std_B8',
    'std_C1', 'std_C2', 'std_C3', 'std_C4', 'std_C5', 'std_C6', 'std_C7', 'std_C8',
    'std_D1', 'std_D2', 'std_D3', 'std_D4', 'std_D5', 'std_D6', 'std_D7', 'std_D8',
    'std_E1', 'std_E2', 'std_E3', 'std_E4', 'std_E5', 'std_E6', 'std_E7', 'std_E8',
    'std_F1', 'std_F2', 'std_F3', 'std_F4', 'std_F5', 'std_F6', 'std_F7', 'std_F8',
    'std_G1', 'std_G2', 'std_G3', 'std_G4', 'std_G5', 'std_G6', 'std_G7', 'std_G8',
    'std_H1', 'std_H2', 'std_H3', 'std_H4', 'std_H5', 'std_H6', 'std_H7', 'std_H8',
    'std_I1', 'std_I2', 'std_I3', 'std_I4', 'std_I5', 'std_I6', 'std_I7', 'std_I8',
    'min_A1', 'min_A2', 'min_A3', 'min_A4', 'min_A5', 'min_A6', 'min_A7', 'min_A8',
    'min_B1', 'min_B2', 'min_B3', 'min_B4', 'min_B5', 'min_B6', 'min_B7', 'min_B8',
    'min_C1', 'min_C2', 'min_C3', 'min_C4', 'min_C5', 'min_C6', 'min_C7', 'min_C8',
    'min_D1', 'min_D2', 'min_D3', 'min_D4', 'min_D5', 'min_D6', 'min_D7', 'min_D8',
    'min_E1', 'min_E2', 'min_E3', 'min_E4', 'min_E5', 'min_E6', 'min_E7', 'min_E8',
    'min_F1', 'min_F2', 'min_F3', 'min_F4', 'min_F5', 'min_F6', 'min_F7', 'min_F8',
    'min_G1', 'min_G2', 'min_G3', 'min_G4', 'min_G5', 'min_G6', 'min_G7', 'min_G8',
    'min_H1', 'min_H2', 'min_H3', 'min_H4', 'min_H5', 'min_H6', 'min_H7', 'min_H8',
    'min_I1', 'min_I2', 'min_I3', 'min_I4', 'min_I5', 'min_I6', 'min_I7', 'min_I8',
    'max_A1', 'max_A2', 'max_A3', 'max_A4', 'max_A5', 'max_A6', 'max_A7', 'max_A8',
    'max_B1', 'max_B2', 'max_B3', 'max_B4', 'max_B5', 'max_B6', 'max_B7', 'max_B8',
    'max_C1', 'max_C2', 'max_C3', 'max_C4', 'max_C5', 'max_C6', 'max_C7', 'max_C8',
    'max_D1', 'max_D2', 'max_D3', 'max_D4', 'max_D5', 'max_D6', 'max_D7', 'max_D8',
    'max_E1', 'max_E2', 'max_E3', 'max_E4', 'max_E5', 'max_E6', 'max_E7', 'max_E8',
    'max_F1', 'max_F2', 'max_F3', 'max_F4', 'max_F5', 'max_F6', 'max_F7', 'max_F8',
    'max_G1', 'max_G2', 'max_G3', 'max_G4', 'max_G5', 'max_G6', 'max_G7', 'max_G8',
    'max_H1', 'max_H2', 'max_H3', 'max_H4', 'max_H5', 'max_H6', 'max_H7', 'max_H8',
    'max_I1', 'max_I2', 'max_I3', 'max_I4', 'max_I5', 'max_I6', 'max_I7', 'max_I8'
]].values
```

케미컬 감지 플랫폼 6개의 다른 위치에서 10가지 화학 가스 물질 측정한 센서 값



```
"""
총 10개의 클래스를 가지고 있음
['Acetaldehyde_500' 'Acetone_2500' 'Ammonia_10000' 'Benzene_200'
 'Butanol_100' 'CO_1000' 'CO_4000' 'Ethylene_500' 'Methane_1000'
 'Methanol_200' 'Toluene_200']
"""
```

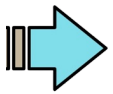
10가지 화학물질 명칭

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

```
# 퍼셉트론 01 실습 : 화학가스 물질 분류 문제
import torch
import pandas as pd
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# CSV 파일 로드
df = pd.read_csv('chemicals_in_wind_tunnel.csv')

# DataFrame의 컬럼 이름 추출
columns = df.columns.tolist()
columns_except_last = columns[:-1]
```

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

```
data = df[[
    'mean_A1', 'mean_A2', 'mean_A3', 'mean_A4', 'mean_A5', 'mean_A6', 'mean_A7', 'mean_A8',
    'mean_B1', 'mean_B2', 'mean_B3', 'mean_B4', 'mean_B5', 'mean_B6', 'mean_B7', 'mean_B8',
    'mean_C1', 'mean_C2', 'mean_C3', 'mean_C4', 'mean_C5', 'mean_C6', 'mean_C7', 'mean_C8',
    'mean_D1', 'mean_D2', 'mean_D3', 'mean_D4', 'mean_D5', 'mean_D6', 'mean_D7', 'mean_D8',
    'mean_E1', 'mean_E2', 'mean_E3', 'mean_E4', 'mean_E5', 'mean_E6', 'mean_E7', 'mean_E8',
    'mean_F1', 'mean_F2', 'mean_F3', 'mean_F4', 'mean_F5', 'mean_F6', 'mean_F7', 'mean_F8',
    'mean_G1', 'mean_G2', 'mean_G3', 'mean_G4', 'mean_G5', 'mean_G6', 'mean_G7', 'mean_G8',
    'mean_H1', 'mean_H2', 'mean_H3', 'mean_H4', 'mean_H5', 'mean_H6', 'mean_H7', 'mean_H8',
    'mean_I1', 'mean_I2', 'mean_I3', 'mean_I4', 'mean_I5', 'mean_I6', 'mean_I7', 'mean_I8',
    'std_A1', 'std_A2', 'std_A3', 'std_A4', 'std_A5', 'std_A6', 'std_A7', 'std_A8',
    'std_B1', 'std_B2', 'std_B3', 'std_B4', 'std_B5', 'std_B6', 'std_B7', 'std_B8',
    'std_C1', 'std_C2', 'std_C3', 'std_C4', 'std_C5', 'std_C6', 'std_C7', 'std_C8',
    'std_D1', 'std_D2', 'std_D3', 'std_D4', 'std_D5', 'std_D6', 'std_D7', 'std_D8',
    'std_E1', 'std_E2', 'std_E3', 'std_E4', 'std_E5', 'std_E6', 'std_E7', 'std_E8',
    'std_F1', 'std_F2', 'std_F3', 'std_F4', 'std_F5', 'std_F6', 'std_F7', 'std_F8',
    'std_G1', 'std_G2', 'std_G3', 'std_G4', 'std_G5', 'std_G6', 'std_G7', 'std_G8',
    'std_H1', 'std_H2', 'std_H3', 'std_H4', 'std_H5', 'std_H6', 'std_H7', 'std_H8',
    'std_I1', 'std_I2', 'std_I3', 'std_I4', 'std_I5', 'std_I6', 'std_I7', 'std_I8',
    'min_A1', 'min_A2', 'min_A3', 'min_A4', 'min_A5', 'min_A6', 'min_A7', 'min_A8',
    'min_B1', 'min_B2', 'min_B3', 'min_B4', 'min_B5', 'min_B6', 'min_B7', 'min_B8',
    'min_C1', 'min_C2', 'min_C3', 'min_C4', 'min_C5', 'min_C6', 'min_C7', 'min_C8',
    'min_D1', 'min_D2', 'min_D3', 'min_D4', 'min_D5', 'min_D6', 'min_D7', 'min_D8',
    'min_E1', 'min_E2', 'min_E3', 'min_E4', 'min_E5', 'min_E6', 'min_E7', 'min_E8',
    'min_F1', 'min_F2', 'min_F3', 'min_F4', 'min_F5', 'min_F6', 'min_F7', 'min_F8',
    'min_G1', 'min_G2', 'min_G3', 'min_G4', 'min_G5', 'min_G6', 'min_G7', 'min_G8',
    'min_H1', 'min_H2', 'min_H3', 'min_H4', 'min_H5', 'min_H6', 'min_H7', 'min_H8',
    'min_I1', 'min_I2', 'min_I3', 'min_I4', 'min_I5', 'min_I6', 'min_I7', 'min_I8',
    'max_A1', 'max_A2', 'max_A3', 'max_A4', 'max_A5', 'max_A6', 'max_A7', 'max_A8',
    'max_B1', 'max_B2', 'max_B3', 'max_B4', 'max_B5', 'max_B6', 'max_B7', 'max_B8',
    'max_C1', 'max_C2', 'max_C3', 'max_C4', 'max_C5', 'max_C6', 'max_C7', 'max_C8',
    'max_D1', 'max_D2', 'max_D3', 'max_D4', 'max_D5', 'max_D6', 'max_D7', 'max_D8',
    'max_E1', 'max_E2', 'max_E3', 'max_E4', 'max_E5', 'max_E6', 'max_E7', 'max_E8',
    'max_F1', 'max_F2', 'max_F3', 'max_F4', 'max_F5', 'max_F6', 'max_F7', 'max_F8',
    'max_G1', 'max_G2', 'max_G3', 'max_G4', 'max_G5', 'max_G6', 'max_G7', 'max_G8',
    'max_H1', 'max_H2', 'max_H3', 'max_H4', 'max_H5', 'max_H6', 'max_H7', 'max_H8',
    'max_I1', 'max_I2', 'max_I3', 'max_I4', 'max_I5', 'max_I6', 'max_I7', 'max_I8'
]].values
labels = df['Chemical']
```



케미컬 감지 플랫폼 6개의 다른 위치에서 10가지 화학 가스 물질 측정한 센서 값

최소 / 최대 / 표준편차 / 평균 값 이용

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

```
# 어떤 종류의 값이 있는지 확인 체크
unique_chemicals = df['Chemical'].unique()
"""
총 10개의 클래스를 가지고 있음
['Acetaldehyde_500' 'Acetone_2500' 'Ammonia_10000' 'Benzene_200'
 'Butanol_100' 'CO_1000' 'CO_4000' 'Ethylene_500' 'Methane_1000'
 'Methanol_200' 'Toluene_200']
"""

chemical_dict = {}
for i, chemical in enumerate(unique_chemicals):
    chemical_dict[chemical] = i

df['Chemical'] = df['Chemical'].map(chemical_dict)

labels = df['Chemical'].values
```

해당 컬럼에 어떤 값이 존재하는지 체크

해당 컬럼으로 dict 생성

라벨 : 숫자 형태로 추출

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

```
# 데이터 표준화
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# 데이터를 PyTorch Tensor로 변환
x = torch.tensor(data_scaled, dtype=torch.float32)
y = torch.tensor(labels, dtype=torch.long)

# 학습 및 테스트 데이터 분할
x_train, x_test, y_train, y_test = train_test_split( *arrays: x, y, test_size=0.2, random_state=42)
print("x_trina 데이터 >> ", len(x_train))
print("x_test 데이터 >> ", len(x_test))

# Dataset 및 DataLoader 생성
train_dataset = TensorDataset( *tensors: x_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = TensorDataset( *tensors: x_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

```
x_trina 데이터 >>  14336
x_test 데이터 >>  3585
```

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

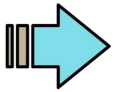
```
# 다층 퍼셉트론 모델 정의
2 usages  new *
class MLP(nn.Module):
    new *
    def __init__(self, input_size):
        super(MLP, self).__init__()
        self.linear1 = nn.Linear(input_size, out_features=512)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(in_features=512, out_features=256)
        self.linear3 = nn.Linear(in_features=256, out_features=128)
        self.linear4 = nn.Linear(in_features=128, out_features=64)
        self.linear5 = nn.Linear(in_features=64, out_features=34)
        self.linear6 = nn.Linear(in_features=34, out_features=11)

    new *
    def forward(self, x):
        out = self.relu(self.linear1(x))
        out = self.relu(self.linear2(out))
        out = self.relu(self.linear3(out))
        out = self.relu(self.linear4(out))
        out = self.relu(self.linear5(out))
        out = self.linear6(out)
        return out
```

6 레이어 생성

학습 클래스 개수

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

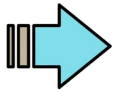
```python
# 모델 초기화 및 손실 함수, 옵티마이저 정의
model = MLP(input_size=x.shape[1])
criterion = nn.CrossEntropyLoss()  # 평균 제곱 오차 손실 함수
# optimizer = optim.Adam(model.parameters(), lr=0.001)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# 모델 학습
num_epochs = 100
for epoch in range(num_epochs):
    model.train()  # 모델을 학습 모드로 설정
    for inputs, targets in train_loader:
        optimizer.zero_grad()  # 기울기 초기화
        outputs = model(inputs)  # 모델 예측
        loss = criterion(outputs, targets)  # 손실 계산
        loss.backward()  # 역전파
        optimizer.step()  # 옵티마이저로 모델 파라미터 업데이트
    print(f"Epoch {epoch + 1}, Train Loss: {loss.item()}")
```



```
Epoch 11, Train Loss: 0.006323399022221565
Epoch 12, Train Loss: 0.004024043679237366
Epoch 13, Train Loss: 0.002476965542882681
Epoch 14, Train Loss: 0.028945447877049446
Epoch 15, Train Loss: 0.020429398864507675
Epoch 16, Train Loss: 0.024747375398874283
Epoch 17, Train Loss: 0.09392905980348587
Epoch 18, Train Loss: 0.0017761106137186289
Epoch 19, Train Loss: 0.0003051082894671708 3
Epoch 20, Train Loss: 0.013497118838131428
Epoch 21, Train Loss: 0.0002872652257792651 7
Epoch 22, Train Loss: 0.0379503034055233
```

다층 퍼셉트론(Multilayer Perceptron, MLP)을 PyTorch로 구현

```
# 모델 평가
model.eval()  # 모델을 평가 모드로 설정
total_correct = 0
total_samples = 0
with torch.no_grad():  # 그래디언트 계산 비활성화
    for inputs, targets in test_loader:
        outputs = model(inputs)  # 모델 예측
        _, predicted = torch.max(outputs, dim=1)  # 가장 높은 확률을 가진 클래스 선택
        total_samples += targets.size(0)  # 총 샘플 수 업데이트
        total_correct += (predicted == targets).sum().item()  # 맞춘 샘플 수 업데이트

accuracy = total_correct / total_samples  # 정확도 계산
print(f"Test Accuracy: {accuracy}")
```



Test Accuracy: 0.996094839609484

선형 분류-선형 판별 분석 (Linear Discriminant Analysis, LDA)

Infinyx
THINK LIFE SYNC AI

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

- 선형 판별 분석 이란 ?

분류(classification) 문제를 해결하는 데 사용되는 통계 기법 중 하나입니다.

LDA는 다차원 데이터를 축소시켜 분류를 수행합니다. 이를 위해 클래스 간 분산(between-class variance)과 클래스 내 분산(within-class variance)의 비율을 최대화하는 방식으로 차원을 축소시킵니다.

즉, LDA는 클래스 간의 차이를 최대화하고 클래스 내부의 차이를 최소화하는 방식으로 데이터를 분류합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

장점 / 단점	설명
선형 판별 분석 장점	차원 축소를 통해 데이터의 분산을 최대한 보존하면서 데이터의 차원을 축소시키기 때문에, 차원의 저주(curse of dimensionality) 문제를 완화할 수 있습니다. 분류 작업에서 과적합(overfitting) 문제를 예방할 수 있습니다.
선형 판별 분석 단점	LDA는 데이터가 정규분포(normal distribution)를 따른다는 가정이 필요합니다. 만약 이 가정이 맞지 않으면 LDA의 성능이 떨어질 수 있습니다. LDA는 이진 분류(binary classification)에만 적용 가능합니다. 즉, 다중 클래스(multiclass) 문제에 대해서는 LDA를 여러 번 적용해야 합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

용어 명칭	선형 판별 분석 알아야하는 용어 소개
데이터 분산	데이터의 흩어짐 정도를 나타내는 지표입니다. 표본 분산(Sample Variance)은 주어진 데이터 샘플들의 분산을 계산한 값으로, 표본 분산이 작으면 데이터가 모여있는 것이고, 크면 데이터가 흩어져 있는 것입니다.
공분산	두 개 이상의 확률 변수가 연관성을 가지며 분포하는지를 나타내는 지표입니다. 두 변수 X와 Y의 공분산은 X가 변할 때 Y도 변하는 정도를 나타내며, X와 Y가 서로 독립적이라면 공분산은 0이 됩니다. 또한, 공분산이 양수이면 X와 Y가 양의 상관관계를 가지며, 음수이면 음의 상관관계를 가지게 됩니다.
공분산 행렬	다변량 데이터의 공분산을 표현하는 행렬입니다. 대각선 방향에는 각 변수의 분산이 위치하고, 비대각 성분에는 두 변수 간의 공분산이 위치합니다. 이를 통해 다변량 데이터의 분산과 상관관계를 동시에 고려할 수 있습니다.
목적 함수	클래스 간 분산을 최대화하고 클래스 내 분산을 최소화하는 선형 변환을 찾는 것입니다. 즉, 클래스 간 거리는 멀리, 클래스 내 거리는 가깝게 하는 결정 경계(decision boundary)를 찾는 것이 LDA의 목적입니다. 이를 위해서 LDA는 고유값 분해(eigenvalue decomposition)를 사용하여 클래스 간 분산과 클래스 내 분산을 계산합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

- 파라미터 추정과 결정 경계 소개

선형 판별 분석에서는 클래스 간 분산(between-class scatter)과 클래스 내 분산(within-class scatter)을 최대화, 최소화하는 파라미터를 찾아 결정 경계(decision boundary)를 설정합니다.

파라미터 추정 과정에서는 클래스 간 분산과 클래스 내 분산을 계산합니다. 클래스 간 분산은 클래스들 간의 거리를 측정하는 것으로, 클래스들의 중심(평균) 간의 거리를 계산합니다. 클래스 내 분산은 각 클래스 내에서 데이터들이 서로 얼마나 가까이 분포해 있는지를 나타내는 것으로, 각 클래스의 분산을 모두 더한 값으로 계산합니다.

결정 경계는 파라미터 추정 과정에서 계산된 클래스 간 분산과 클래스 내 분산을 이용하여 설정됩니다. 클래스 간 분산이 크고 클래스 내 분산이 작을수록, 즉 클래스 간 거리가 멀고 클래스 내 데이터들이 서로 가까이 분포할수록 결정 경계는 잘 설정됩니다. 결정 경계는 일반적으로 직선 또는 초평면으로 표현됩니다.

 # scikit-learn 패키지를 통해 선형 판별 분석 실습 - 붓꽃 데이터 활용한 선형 판별 분석 실습

→ LinearDiscriminantAnalysis를 사용하여 2차원 데이터셋을 분류하고, 결정 경계를 시각화하는 코드입니다.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Load iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Fit LDA
lda = LinearDiscriminantAnalysis()
lda.fit(X, y)

# Calculate training accuracy
train_acc = lda.score(X, y)
```

load_iris() 함수를 사용하여 iris 데이터를 로드합니다.

X = iris.data: iris 데이터에서 feature 데이터를 가져와 X 변수에 할당합니다.
y = iris.target: iris 데이터에서 label 데이터를 가져와 y 변수에 할당합니다.

lda = LinearDiscriminantAnalysis(): LDA 모델 객체를 생성합니다.
lda.fit(X, y): X와 y 데이터를 사용하여 LDA 모델을 학습합니다.

LDA 모델의 score 함수는 모델을 사용하여 주어진 데이터에 대한 정확도를 계산합니다.

 # scikit-learn 패키지를 통해 선형 판별 분석 실습 - 붓꽃 데이터 활용한 선형 판별 분석 실습

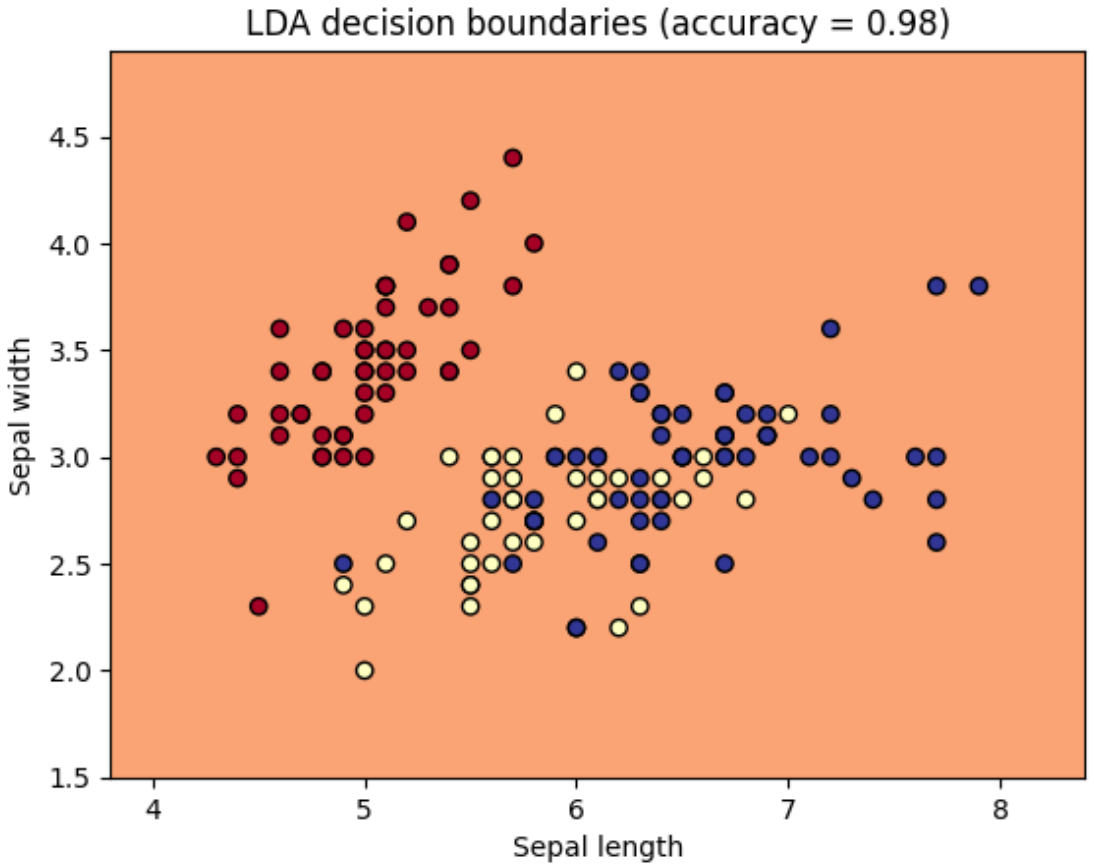
```
# Plot decision boundaries
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200), np.linspace(y_min, y_max, 200))
Z = lda.predict(np.c_[xx.ravel(), yy.ravel(), np.zeros(xx.ravel().shape), np.zeros(xx.ravel().shape)])
```

먼저, np.meshgrid 함수를 사용하여 x_min부터 x_max까지의 범위와 y_min부터 y_max까지의 범위를 200 등분한 좌표를 생성합니다. 이를 xx, yy 배열에 저장합니다.

다음으로 np.c_ 함수를 사용하여 xx와 yy 배열을 연결하여 2차원 평면상의 모든 좌표점을 구합니다.
이때, np.zeros 함수를 사용하여 나머지 두 개의 특성에 대한 값을 0으로 초기화합니다.

마지막으로, lda.predict 함수를 사용하여 각 좌표점의 클래스 레이블을 예측합니다. 이를 Z 배열에 저장합니다.
Z 배열의 각 값은 해당 좌표점이 속하는 클래스를 나타냅니다.

scikit-learn 패키지를 통해 선형 판별 분석 실습 - 붓꽃 데이터 활용한 선형 판별 분석 실습




```
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu, alpha=.8)

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu, edgecolor='k')
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title('LDA decision boundaries (accuracy = {:.2f})'.format(train_acc))
plt.show()
```

그래프 그림에서 빨간색, 파란색, 노란색은 각각 Setosa, Versicolor, Virginica 품종을 나타냅니다.

LDA 알고리즘은 클래스간의 분산이 최대화되도록 결정 경계를 그리므로, 결정 경계를 기준으로 한 색상이 속하는 클래스를 나타냅니다.

Sepal length는 붓꽃의 꽃받침(sepal)의 길이를 말합니다. 꽃받침의 길이는 붓꽃의 품종을 판별하는데 중요한 특성 중 하나입니다.
Sepal width (꽃받침 너비)는 꽃받침의 가장 넓은 부분을 재는 값으로, 위 그래프에서 y축에 해당합니다.
꽃받침 너비 가 크다는 것은 꽃받침이 넓다는 것을 의미합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

- 선형 판별 분석은 분류 문제에 널리 사용되며, 다양한 분야에서 활용됩니다.

분야 명칭	설명
의학	암 진단, 약물 치료 반응 예측 등 의학 분야에서 사용
금융	대출 기준 결정, 대출 상환 능력 예측 등 금융 분야에서 사용
이미지 분류	얼굴 인식, 디지털 이미지 분류 등의 문제에서 사용
소프트웨어 엔지니어링	결함 진단 및 결함 감지와 같은 소프트웨어 엔지니어링 문제에서 사용
제조 업체	제품 결함 감지, 제품 품질 관리 등에 사용
신뢰성 분석	신뢰성 분석에서 불량품 발견, 제품 신뢰성 예측 등에 사용

LDA는 분류 문제에 유용한 알고리즘 중 하나이며, 다른 분류 알고리즘과 함께 사용되기도 합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

LDA 한계점 소개
✓ 선형 결정 경계를 가정하기 때문에, 비선형적인 데이터에 대해서는 성능이 떨어질 수 있습니다.
✓ 이러한 경우에는 다른 비선형 분류 방법을 적용해야 합니다.
✓ 입력 변수들 간의 공분산이 동일하다는 가정을 전제로 합니다. 그러나 실제 데이터에서는 이러한 가정이 맞지 않을 수 있습니다.
✓ 이러한 경우에는 공분산이 서로 다른 경우를 고려하는 QDA(Quadratic Discriminant Analysis) 등의 분류 방법을 적용할 수 있습니다.
✓ 분류를 위한 정보를 직접적으로 추출하도록 설계되었습니다. 따라서, 회귀 문제 등 다른 종류의 문제에 대해서는 적용이 어렵습니다.
✓ 이러한 경우에는 선형 회귀 등 다른 방법을 사용해야 합니다.

선형 분류 (Linear Classification) – 선형 판별 분석 (Linear Discriminant Analysis, LDA)

- LDA의 한계를 극복하기 위해서는 다양한 개선 방법들이 제안되고 있습니다.

  ✓ 예를 들어, 입력 변수들 간의 공분산이 서로 다르더라도 LDA를 적용할 수 있는 Flexible Discriminant Analysis(FDA)가 있습니다.

  ✓ 비선형 데이터에 대해서는 커널 기반의 LDA 방법(Kernel LDA)이나, 심층 신경망(DNN) 등을 활용하는 방법들도 있습니다.

선형 분류
딥러닝과 선형 분류의 관계
어떤 분야에서 응용 되는가 ?

Infinyx
THINK LIFE SYNC AI

선형 분류 (Linear Classification) – 딥러닝과 선형 분류의 관계

딥러닝은 선형 분류의 일반화된 형태로 볼 수 있습니다. 딥러닝 모델은 일반적으로 여러 개의 선형 계층과 비선형 활성화 함수를 쌓아서 구성됩니다.

이때, 각 계층은 입력 데이터를 가중치와 편향을 곱하고 더하는 선형 변환(linear transformation)을 수행하고, 그 결과를 비선형 활성화 함수에 통과시켜 출력을 만듭니다. 이 과정은 여러 개의 선형 함수를 결합해서 비선형 분류 문제를 풀 수 있도록 합니다.

딥러닝에서 사용하는 대표적인 비선형 활성화 함수로는 시그모이드 함수, ReLU(Rectified Linear Unit) 함수, tanh 함수 등이 있습니다.

딥러닝은 선형 분류보다 더 복잡한 패턴을 학습할 수 있기 때문에, 대부분의 분류 문제에서 뛰어난 성능을 보입니다.

하지만, 딥러닝 모델은 학습에 필요한 계산량이 매우 많기 때문에, 학습이 느리고, 모델이 복잡해지면 과적합 문제가 발생할 수 있습니다.

따라서, 데이터가 적거나 단순한 분류 문제에서는 선형 분류기가 더 효과적일 수 있습니다.

선형 분류 (Linear Classification) – 이미지, 자연어 등 다양한 분야에서 응용 소개

이미지 분류, 자연어 처리 등 다양한 분야에서 응용됩니다.

예를 들어, 이미지 분류 분야에서는 다양한 CNN(Convolutional Neural Network) 구조가 제안되어 왔습니다. 하지만 CNN에서도 선형 분류가 매우 중요한 역할을 합니다.

보통 마지막 층에서 Fully Connected Layer(FC Layer)라는 선형 분류 계층이 사용되며, 이를 통해 이미지를 분류합니다. 또한, 선형 분류를 이용한 소프트맥스 회귀(softmax regression)도 자주 사용됩니다.

또한, 자연어 처리 분야에서도 선형 분류가 활용됩니다. 예를 들어, 텍스트 분류나 감성 분석 등에서는 선형 분류 알고리즘을 활용하여 문서를 분류합니다.

또한, 선형 분류를 이용한 로지스틱 회귀(logistic regression)도 많이 사용됩니다.

이러한 선형 분류의 기술은 현재도 지속적으로 연구되고 있으며, 머신러닝 및 딥러닝 분야에서 매우 중요한 역할을 하고 있습니다.


감사합니다.