Colab에서 Perceptron 구현과 pytorch MLP 사용 및 classification 성능 확인

– Linearly separable / 2d binary class ($x1$, $x2$, y)를 가지고 있는 서로 다른 data point class 별로 100개씩 생성하고 랜덤하게 70%는 train / 30%는 test 셋으로 분리 (10)

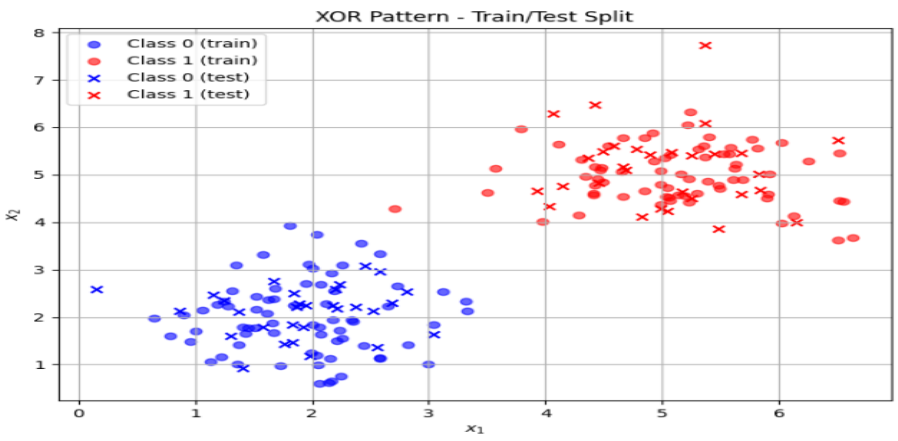
```
# 랜덤 시드 고정 (재현 가능하도록)
np.random.seed(42)

# 클래스 0: (x1, x2) around (2, 2)
class_0 = np.random.multivariate_normal(mean=[2, 2], cov=[[0.5, 0], [0, 0.5]], size=100)
labels_0 = np.zeros(100)

# 클래스 1: (x1, x2) around (5, 5)
class_1 = np.random.multivariate_normal(mean=[5, 5], cov=[[0.5, 0], [0, 0.5]], size=100)
labels_1 = np.ones(100)

# 데이터와 라벨 합치기
X = np.vstack((class_0, class_1))
y = np.hstack((labels_0, labels_1))

# train/test 셋 나누기 (stratify=y는 클래스 비율 유지 위해)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)
```

```
print(X_train.shape)    (140, 2)
print(X_test.shape)     (60, 2)
print(y_train.shape)    (140,)
print(y_test.shape)     (60,)
```

gpt사용해서 데이터 랜덤 생성 후 시각화 하여 데이터가 잘 생성되었는지 확인

– 2d visualization을 통하여 실제로 linearly separable한 data인지 확인 할 수 있는 scatter plot 생성 (10)

```
# 시각화
plt.figure(figsize=(8, 6))
plt.scatter(X_train[y_train==0][:, 0], X_train[y_train==0][:, 1], color='blue', label='Class 0 (train)', alpha=0.6)
plt.scatter(X_train[y_train==1][:, 0], X_train[y_train==1][:, 1], color='red', label='Class 1 (train)', alpha=0.6)
plt.scatter(X_test[y_test==0][:, 0], X_test[y_test==0][:, 1], color='blue', edgecolor='k', label='Class 0 (test)', marker='x')
plt.scatter(X_test[y_test==1][:, 0], X_test[y_test==1][:, 1], color='red', edgecolor='k', label='Class 1 (test)', marker='x')
plt.legend()
plt.title("XOR Pattern - Train/Test Split")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.grid(True)
plt.show()
```

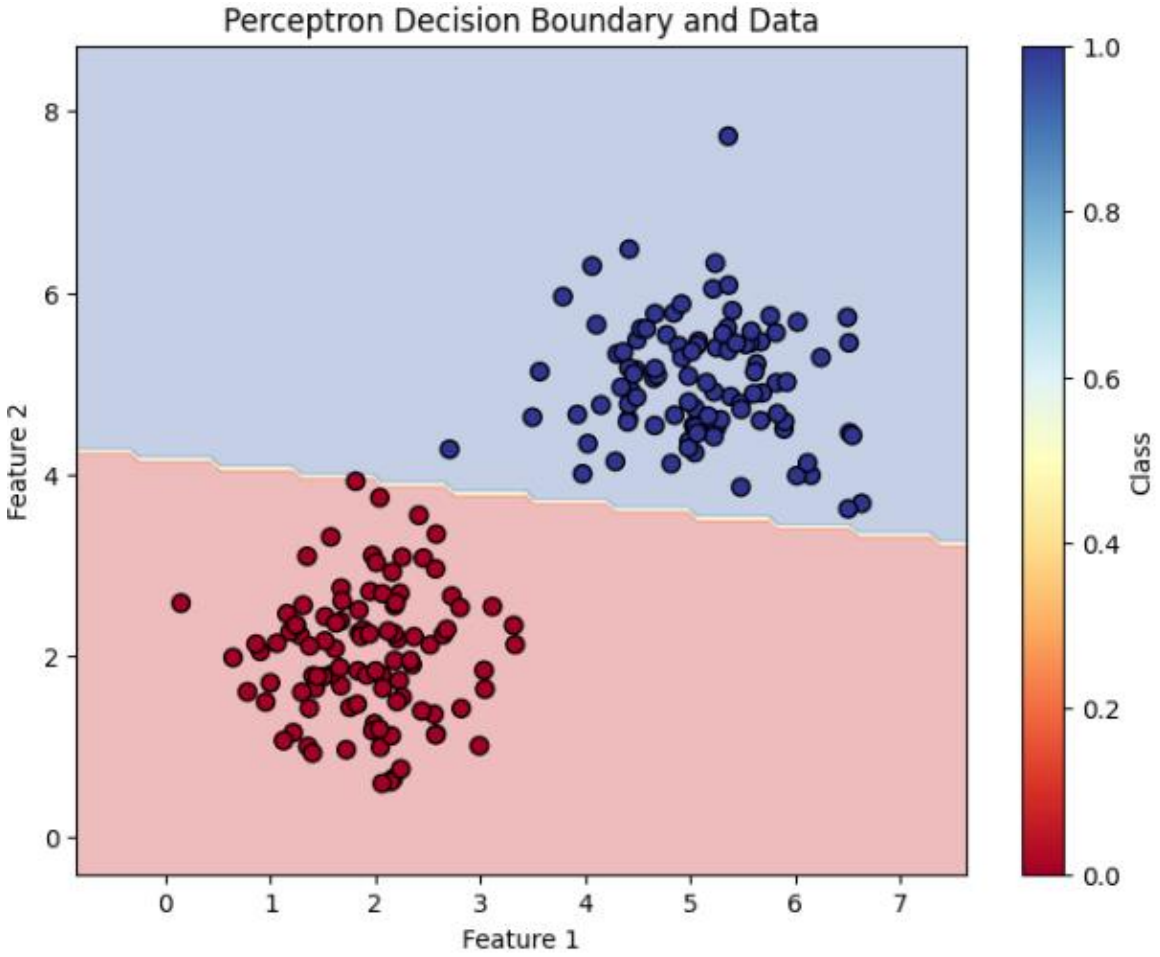


– training data를 사용하여 perceptron을 training 하고 train accuracy / test accuracy 작성 (15) -Perceptron을 사용해서 training해도 좋은 accuracy를 얻을 수 있었다.

```
# Perceptron 모델 구현
class Perceptron:
    def __init__(self, input_size, learning_rate=0.01, epochs=100):
        self.input_size = input_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(input_size)  # 초기 가중치
        self.bias = 0  # 초기 바이어스

    def activation(self, x):
        return 1 if x >= 0 else 0

    def forward(self, x):
        return self.activation(np.dot(x, self.weights) + self.bias)

    def train(self, X, y):
        for epoch in range(self.epochs):
            for i in range(len(X)):
                output = self.forward(X[i])
                error = y[i] - output  # 오차 계산

                # 가중치와 바이어스 업데이트
                self.weights += self.learning_rate * error * X[i]
                self.bias += self.learning_rate * error

            # Accuracy 출력
            if (epoch + 1) % 10 == 0:
                predictions = self.predict(X)
                accuracy = np.mean(predictions == y) * 100
                print(f'Epoch [{epoch + 1}/{self.epochs}], Accuracy: {accuracy:.2f}%')

    def predict(self, X):
        return np.array([self.forward(x) for x in X])
```

```
Epoch [10/100], Accuracy: 100.00%
Epoch [20/100], Accuracy: 100.00%
Epoch [30/100], Accuracy: 100.00%
Epoch [40/100], Accuracy: 100.00%
Epoch [50/100], Accuracy: 100.00%
Epoch [60/100], Accuracy: 100.00%
Epoch [70/100], Accuracy: 100.00%
Epoch [80/100], Accuracy: 100.00%
Epoch [90/100], Accuracy: 100.00%
Epoch [100/100], Accuracy: 100.00%
Train Accuracy: 100.00%
Test Accuracy: 100.00%
```

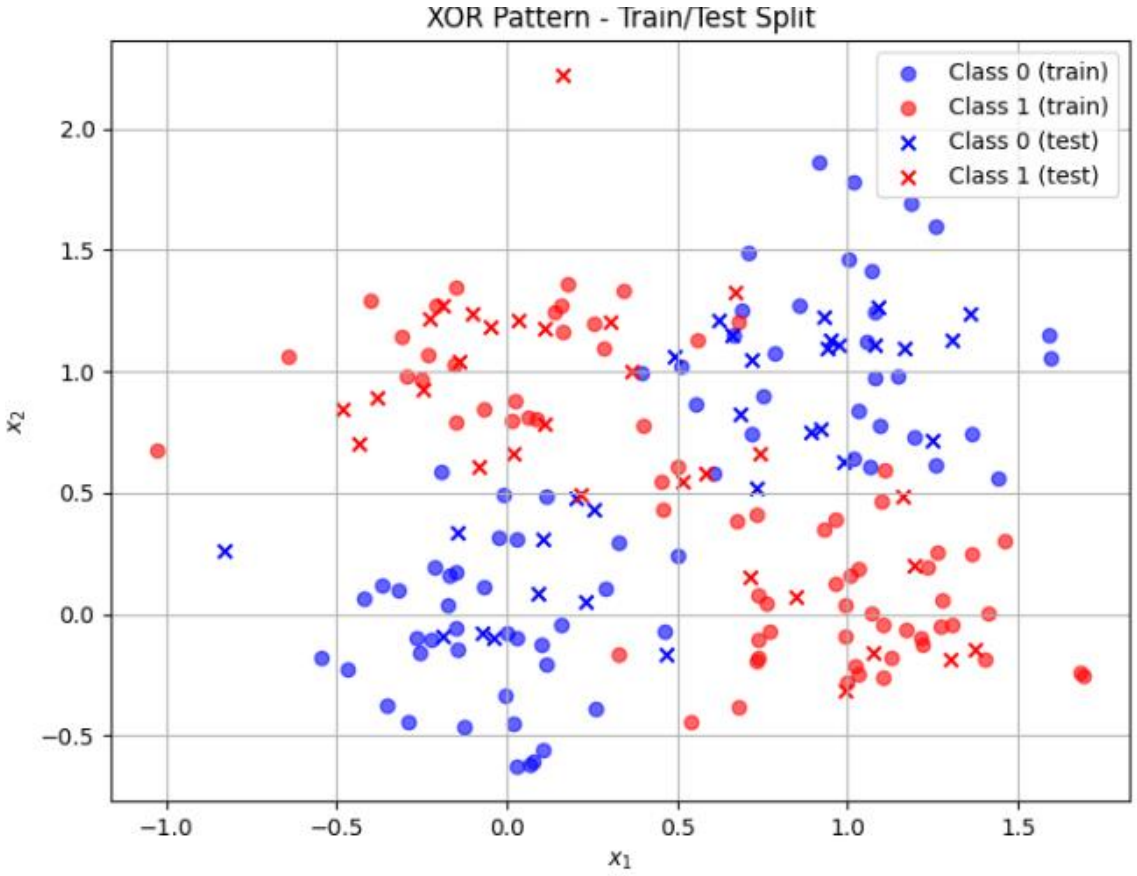
– perceptron의 decision boundary와 data를 2d상에서 확인 할 수 있도록 plot 생성 (10)
선형으로 구분할 수 있는 데이터이기 때문에 잘 구분된다. [그리는 코드 gpt사용]



– XOR 형태의 Nonlinearly separable 2d binary class를 가지고 있는 서로 다른 data point class 별로 100개씩 생성하고 랜덤하게 70%는 train / 30%는 test 셋으로 분리 (10)

```
# XOR 데이터 생성
np.random.seed(42)
class_0 = np.vstack((
    np.random.multivariate_normal([0, 0], [[0.1, 0], [0, 0.1]], 50),
    np.random.multivariate_normal([1, 1], [[0.1, 0], [0, 0.1]], 50)
))
class_1 = np.vstack((
    np.random.multivariate_normal([0, 1], [[0.1, 0], [0, 0.1]], 50),
    np.random.multivariate_normal([1, 0], [[0.1, 0], [0, 0.1]], 50)
))
X = np.vstack((class_0, class_1))
y = np.hstack((np.zeros(100), np.ones(100)))

# train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)
```

– 2d visualization을 통하여 실제로 linearly separable한 data인지 확인 할 수 있는 plot 생성 (10) - plot으로 확인해보니 확실히 선 1개로 구분할 수 없는 데이터임을 확인하였다



– training data를 사용하여 위에서 구현한 perceptron으로 training 하고 train / test accuracy 작성 (10)

```
Epoch [10/100], Accuracy: 50.00%
Epoch [20/100], Accuracy: 50.00%
Epoch [30/100], Accuracy: 50.00%
Epoch [40/100], Accuracy: 50.00%
Epoch [50/100], Accuracy: 65.00%
Epoch [60/100], Accuracy: 51.43%
Epoch [70/100], Accuracy: 50.00%
Epoch [80/100], Accuracy: 50.00%
Epoch [90/100], Accuracy: 55.71%
Epoch [100/100], Accuracy: 50.00%
[Perceptron]Train Accuracy: 50.00%
[Perceptron]Test Accuracy: 50.00%
```

만들어뒀던 Perceptron을 사용하였고 반복해도 50%로 고정이었다

– pytorch를 사용하여 Logistic regression / MLP를 구현하여 training data를 사용하여 training하고 train / test accuracy 작성 (15)

```python
# Logistic Regression
# ---------------------------------
class LogisticRegression(nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        return self.linear(x)  # BCEWithLogitsLoss를 쓸 것이므로 sigmoid 없음

lr_model = LogisticRegression()
criterion = nn.BCEWithLogitsLoss()
lr_optimizer = optim.SGD(lr_model.parameters(), lr=0.01)
```

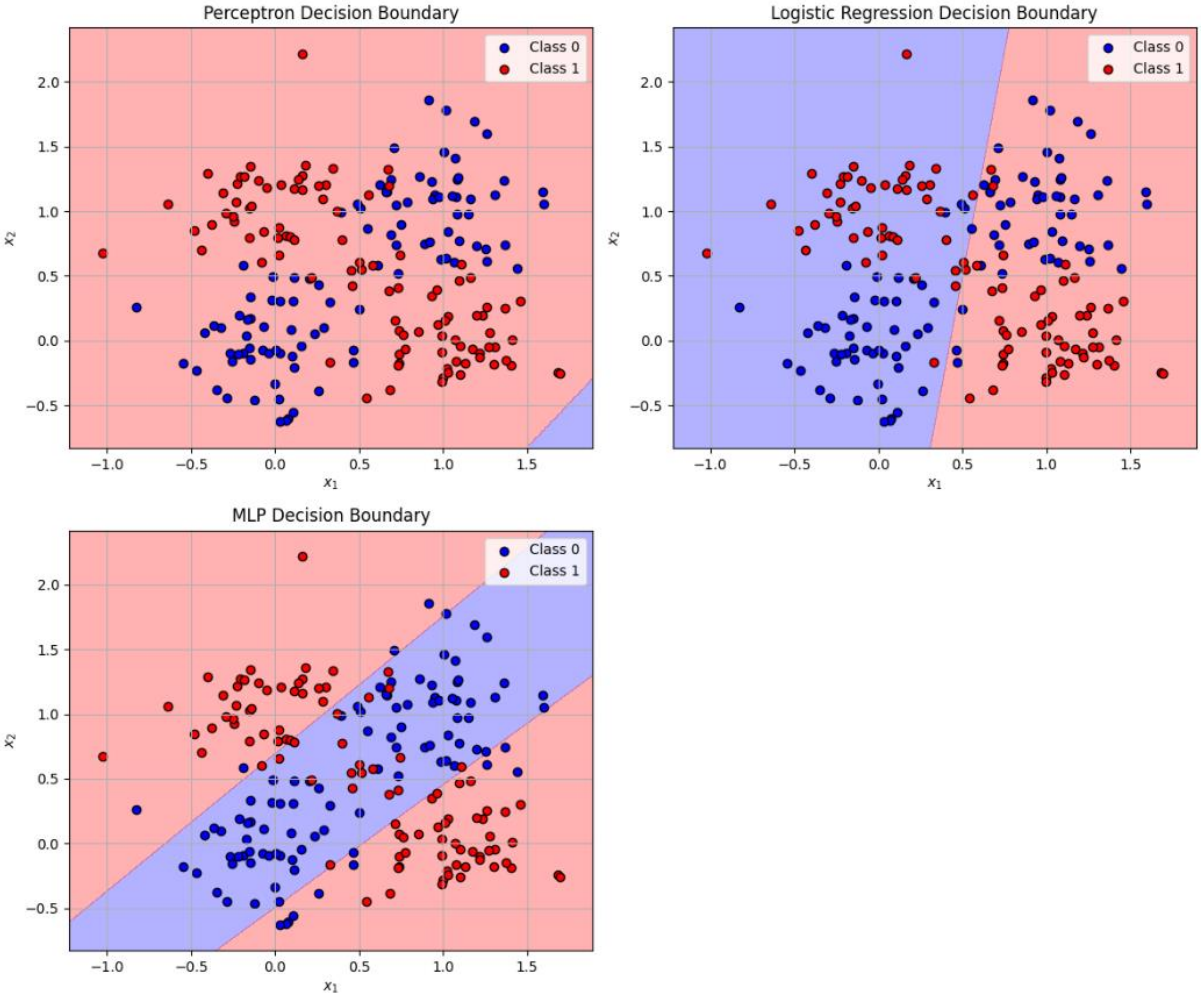
```
[LogReg] Epoch 100, Loss: 0.7125, Train Acc: 45.71%
[LogReg] Epoch 200, Loss: 0.7051, Train Acc: 43.57%
[LogReg] Epoch 300, Loss: 0.7001, Train Acc: 40.00%
[LogReg] Epoch 400, Loss: 0.6963, Train Acc: 27.86%
[LogReg] Epoch 500, Loss: 0.6934, Train Acc: 50.00%
[LogReg] Epoch 600, Loss: 0.6910, Train Acc: 52.86%
[LogReg] Epoch 700, Loss: 0.6889, Train Acc: 55.71%
[LogReg] Epoch 800, Loss: 0.6872, Train Acc: 57.14%
[LogReg] Epoch 900, Loss: 0.6857, Train Acc: 56.43%
[LogReg] Epoch 1000, Loss: 0.6845, Train Acc: 57.14%
[LogReg] Final Train Accuracy: 57.86%
[LogReg] Final Test Accuracy: 35.00%
```

```python
# ---------------------------------
# MLP
# ---------------------------------

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(2, 2)
        self.activation = nn.ReLU()  # 또는 nn.ReLU()
        self.output = nn.Linear(2, 1)

    def forward(self, x):
        x = self.activation(self.hidden(x))  # 은닉층에 비선형성 적용
        x = self.output(x)  # 그대로 출력
        return x  # BCEWithLogitsLoss와 함께 사용하므로 sigmoid 안 씀

mlp_model = MLP()
mlp_optimizer = optim.Adam(mlp_model.parameters(), lr=0.001)
mlp_criterion = nn.BCEWithLogitsLoss()
```

```
[MLP] Epoch 100, Loss: 0.6913, Train Acc: 50.00%
[MLP] Epoch 200, Loss: 0.6733, Train Acc: 50.00%
[MLP] Epoch 300, Loss: 0.6545, Train Acc: 50.00%
[MLP] Epoch 400, Loss: 0.6216, Train Acc: 66.43%
[MLP] Epoch 500, Loss: 0.5823, Train Acc: 75.00%
[MLP] Epoch 600, Loss: 0.5533, Train Acc: 77.86%
[MLP] Epoch 700, Loss: 0.5267, Train Acc: 80.71%
[MLP] Epoch 800, Loss: 0.5023, Train Acc: 82.14%
[MLP] Epoch 900, Loss: 0.4807, Train Acc: 84.29%
[MLP] Epoch 1000, Loss: 0.4617, Train Acc: 84.29%
[MLP] Final Train Accuracy: 84.29%
[MLP] Final Test Accuracy: 85.00%
```

– perceptron / logistic regression model의 decision boundary와 data를 2d상에서 확인 할 수 있도록 각각 plot 생성 (10)

– MLP 모델의 decision boundary와 data를 2d에서 확인 할 수 있도록 생성 (10)



– Logistic regression model이 non-linear function인 sigmoid를 사용하는데 왜 linear classifier라고 하는지 decision boundary 관점 에서 설명 (10)

Sigmoid는 비선형 함수로서, 선형으로는 분류하기 어려운 데이터에 비선형성을 부여하여

더 복잡한 결정 경계를 만들 수 있도록 돕는다. Logistic Regression에서도 sigmoid 함수를 사용하지만, 과제를 하면서 확인한 decision boundary는 명백하게 선형이었다.그 이유는로지스틱 결정 경계는 모델이 **클래스를 나누는 경계선**인데, sigmoid 함수에서는 보통 **출력값이 0.5인 지점**에서 클래스가 바뀌고 **결정 경계는 wTx+b=0**이라는 **선형 방정식**에 의해 정의된다.(왜냐하면 저 값이 sigmoid가 0.5가되는지점이어서)

sigmoid는 출력값을 0과 1 사이로 매핑하는 역할을 할 뿐, 결정 경계의 형태를 비선형으로 만들지는 않는다.