7주차 실습과제 코드

2020142001 곽종근
제출일: 2024.05.07.

실습과제 Chap.3

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 데이터 불러오기
fold_dir = "C:\\Users\\user\\OneDrive - 한국공학대학교\\바탕 화면\\3학년
1학기\\머신러닝실습\\Machine-Learning\\logistic_regression_data.csv"
temp_data = pd.read_csv(fold_dir)
temp_data = temp_data.to_numpy()
#여기서는 헤더를 x

# 데이터 분리
x1 = temp_data[:, 1].reshape(-1,1)  # temp_data 1열을 저장
x2 = temp_data[:, 2].reshape(-1,1)  # temp_data 2열을 저장
y = temp_data[:, 3].reshape(-1,1)   # temp_data 3열을 y로 저장

# 더미 데이터 추가
total_data=np.hstack((x1,x2,y))
#위에서 분리한 데이터를 합침
new_x1 = total_data[:,0].reshape(-1,1)
new_x2 = total_data[:,1].reshape(-1,1)
new_y = total_data[:,2].reshape(-1,1)
# 더미 데이터를 선언
dummy_data = np.ones((len(total_data), 1))
#더미 데이터를 포함한 x값을 설정
x_with_dummy = np.hstack((new_x1, new_x2, dummy_data))

# 시그모이드 함수를 선언
def Sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Cee를 함수로 선언
def Cee(X, y, w):
    m = len(y)
    z = np.dot(X, w)
    p = Sigmoid(z)
    cost = (-1 / m) * np.sum(y * np.log(p) + (1 - y) * np.log(1 - p))
    return cost

# 예측값을 선언 이는 y^과 같음
```

```
def predict(X, w):
    z = np.dot(X, w)
    p = Sigmoid(z)
    predictions = np.where(p >= 0.5, 1, 0)
    return predictions

# 위에서 선언한 함수들을 이용해 경사하강법 진행
def gradient_descent(X, y, alpha, rp):
    # 가중치 초기값을 랜덤으로 선언
    w_ = np.random.rand(3, 1)
    w0_history, w1_history, w2_history, cee_history, accuracy_history = [], [], [], [], []
    # 훈련 세트와 테스트 세트에 대한 정확도 기록
    train_accuracy_history, test_accuracy_history = [], []
    for i in range(rp):
        z = np.dot(X, w_)
        p = Sigmoid(z)
        # axis를 선언해 축별로 계산되도록
        dif_cee = np.mean((p - y) * X, axis=0).reshape(-1, 1)
        w_ -= alpha * dif_cee

        w0_history.append(w_[0][0])
        w1_history.append(w_[1][0])
        w2_history.append(w_[2][0])

        cee = Cee(X, y, w_)
        cee_history.append(cee)

        predictions = predict(X, w_)
        accuracy = np.sum(predictions == y) / len(y)
        accuracy_history.append(accuracy)

        # 훈련 세트에 대한 정확도
        train_predictions = predict(train_x_with_dummy, w_)
        train_accuracy = np.sum(train_predictions == train_y) / len(train_y)
        train_accuracy_history.append(train_accuracy)

        # 테스트 데이터셋에 대한 정확도 계산
        test_predictions = predict(test_x_with_dummy, w_)
        test_accuracy = np.sum(test_predictions == test_y) / len(test_y)
        test_accuracy_history.append(test_accuracy)

    return w0_history, w1_history, w2_history, cee_history, accuracy_history,train_accuracy_history,
test_accuracy_history

def aug_data(data, train_ratio, test_ratio):
    # 데이터를 분할하는 것이므로 분할한 것들의 합이 1이 나와야 함
    assert train_ratio + test_ratio == 1
```

```
    # 데이터의 총 개수
    total_samples = len(total_data)

    # 각 세트의 크기 계산
    train_size = int(total_samples * train_ratio)

    # 데이터를 랜덤하게 섞음
    np.random.shuffle(total_data)

    # 데이터 분할
    train_set = total_data[:train_size]
    test_set = total_data[train_size:]

    return train_set, test_set
# 위에서 선언한 함수를 이용해 7:3으로 트레인과 테스트 분리
train_set, test_set = aug_data(total_data, 0.7, 0.3)

# 분리된 7에 대한 트레인 셋으로 더미데이터 포함한 새로운 값을 만들어준다
train_x_with_dummy = np.hstack((train_set[:, :2], np.ones((len(train_set), 1))))
train_y = train_set[:, 2].reshape(-1, 1)

# 분리된 3에 대한 테스트 셋으로 더미데이터 포함한 새로운 값을 만들어준다
test_x_with_dummy = np.hstack((test_set[:, :2], np.ones((len(test_set), 1))))
test_y = test_set[:, 2].reshape(-1, 1)

w0_history, w1_history, w2_history, cee_history, accuracy_history, train_accuracy_history,
test_accuracy_history = gradient_descent(train_x_with_dummy, train_y, 0.3, 4000)

# 가중치 변화 그래프
plt.figure(figsize=(8, 6))
plt.plot(w0_history, label='w0')
plt.plot(w1_history, label='w1')
plt.plot(w2_history, label='w2')
plt.xlabel('Iterations')
plt.ylabel('Weights')
plt.title('Training Data Change')
plt.grid()
plt.legend()
plt.show()

# 비용 함수 변화 그래프
plt.figure(figsize=(8, 6))
plt.plot(cee_history)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Training Data Cost Function')
```

```
plt.grid()
plt.show()

# 분류 정확도 변화 그래프 (훈련 세트와 테스트 세트)
plt.figure(figsize=(8, 6))
plt.plot(train_accuracy_history, label='Train Accuracy')
plt.plot(test_accuracy_history, label='Test Accuracy')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.title('Accuracy Train and Test')
plt.legend()
plt.grid()
plt.show()

# 테스트 데이터셋에 대한 최종 정확도 출력
print("Final Test Accuracy:", test_accuracy_history[-1])

# 결정 경계 계산해준다
# 각 세트의 최소 최대를 x축으로 하고 그에 따른 y값을 y축으로
x_values = np.linspace(np.min(train_set[:, 0]), np.max(train_set[:, 0]), 100)
y_values = (-w0_history[-1] / w1_history[-1]) * x_values - (w2_history[-1] / w1_history[-1])

# 트레인 셋에 대한 결정 경계 그래프
plt.figure(figsize=(8, 6))
# 클래스 0과 클래스 1을 다른 색상으로 표시
plt.scatter(train_set[train_set[:, 2] == 0, 0], train_set[train_set[:, 2] == 0, 1], color='blue', marker='o',
label='Train Class 0')
plt.scatter(train_set[train_set[:, 2] == 1, 0], train_set[train_set[:, 2] == 1, 1], color='red', marker='o',
label='Train Class 1')
plt.plot(x_values, y_values, color='black', label='Decision Boundary')
plt.xlabel('Attendance rate[%]')
plt.ylabel('Exam score')
plt.title('Training Data Decision Boundary')
plt.legend(title='Class', loc='upper left', labels=['0', '1','Decision Boundary'])
plt.grid()
plt.show()

# 테스트 셋에 대한 결정 경계 그래프
plt.figure(figsize=(8, 6))
# 클래스 0과 클래스 1을 다른 색상으로 표시
plt.scatter(test_set[test_set[:, 2] == 0, 0], test_set[test_set[:, 2] == 0, 1], color='blue', marker='o', label='Test
Class 0')
plt.scatter(test_set[test_set[:, 2] == 1, 0], test_set[test_set[:, 2] == 1, 1], color='red', marker='o', label='Test
Class 1')
plt.plot(x_values, y_values, color='black', label='Decision Boundary')
plt.xlabel('Attendance rate[%]')
plt.ylabel('Exam score')
```

```
plt.title('Test Data Decision Boundary')
plt.legend(title='Class', loc='upper left', labels=['0', '1','Decision Boundary'])
plt.grid()
plt.show()
```