R E P O R T

딥러닝 Mini Project

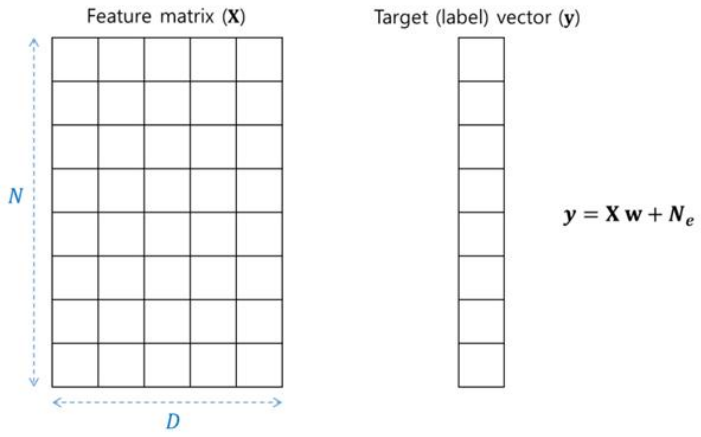
학과	항공전자정보공학부
학번	2021124195
이름	황서경
담당 교수	오병태 교수님
제출일	2025.04.27

1. Generate N linear regression samples, and its label vector using pre-determined parameters w. For analysis, use D-dimensional vectors with large number of samples (N ≥ 1000 , D ≥ 4).

$$\text{feature matrix}: \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ x_{31} & x_{32} & \cdots & x_{3D} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix}$$

$$\text{label vector}: \quad \mathbf{y} = [y_1, y_2, y_3, \cdots y_N]$$

$$\text{Parameter vector}: \quad \mathbf{w} = [w_1, w_2, \ldots, w_D]$$



1) Find the solution using the linear regression method, and measure its estimation error.

Numpy의 radom.randn()을 사용해 X matrix를 랜덤으로 생성시켜준 후 D의 크기에 맞춰 달라지겠지만 우선 w는 [1.5, 2, 3.5, 4]로 지정해주었다. 그에 따라 $y = Xw + N_e$ 공식에 맞춰 y vector를 구한다. 이때 noise 및 절편을 추가해서 약간의 오차가 생기도록 해주었다. 위 데이터를 이용해 w_star를 equation을 이용해 구하고 각 조건들을 다르게 했을 때의 w와 w_star를 이용해 MSE값이 변화하는 것을 비교해보면 아래 표와 같다.

- Shape of generated X : (1000, 4)
- Shape of generated y : (1000, 1)
- True weights used: [1.5, 2, 3.5, 4]
- True Weights : [[1.5,], [2], [3.5], [4]]

- Estimated Weights: [[1.49], [1.997], [3.51], [3.998]]

이외에 조건 변경해 시도해본 것

1	N	D	MSE
	1000	4	3.64
2	N	D	MSE
	1000	10	7.03
3	N	D	MSE
	5000	4	3.62

다양한 feature들을 변형시켜가며 estimation error를 확인해본 결과 noise의 값이 커질수록 MSE는 증가하는 것을 확인해볼 수 있었다. 또한 D의 값이 커질수록 정확한 weight 추정이 어려워지는 경향을 보였다. 그러나 충분한 데이터 수(N)을 가진다면, D의 값이 크더라도 낮은 오차가 구해졌다.

2) Find the solution using gradient descent method and analyze its performance by various adaptive learning rate schemes, batch size, and initialization methods, etc.

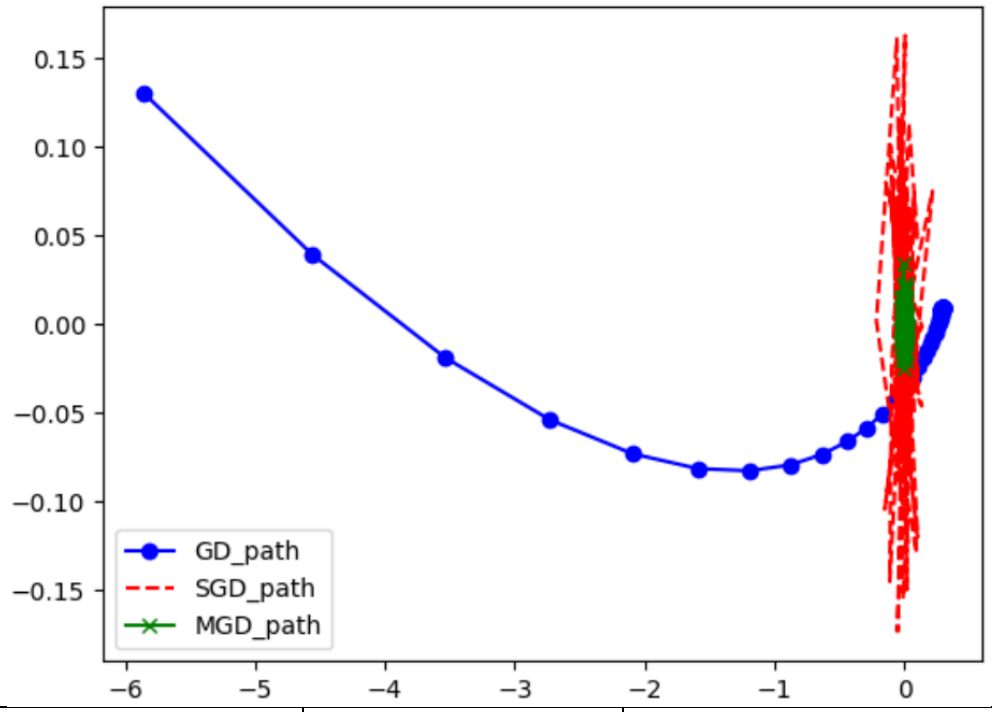
코드

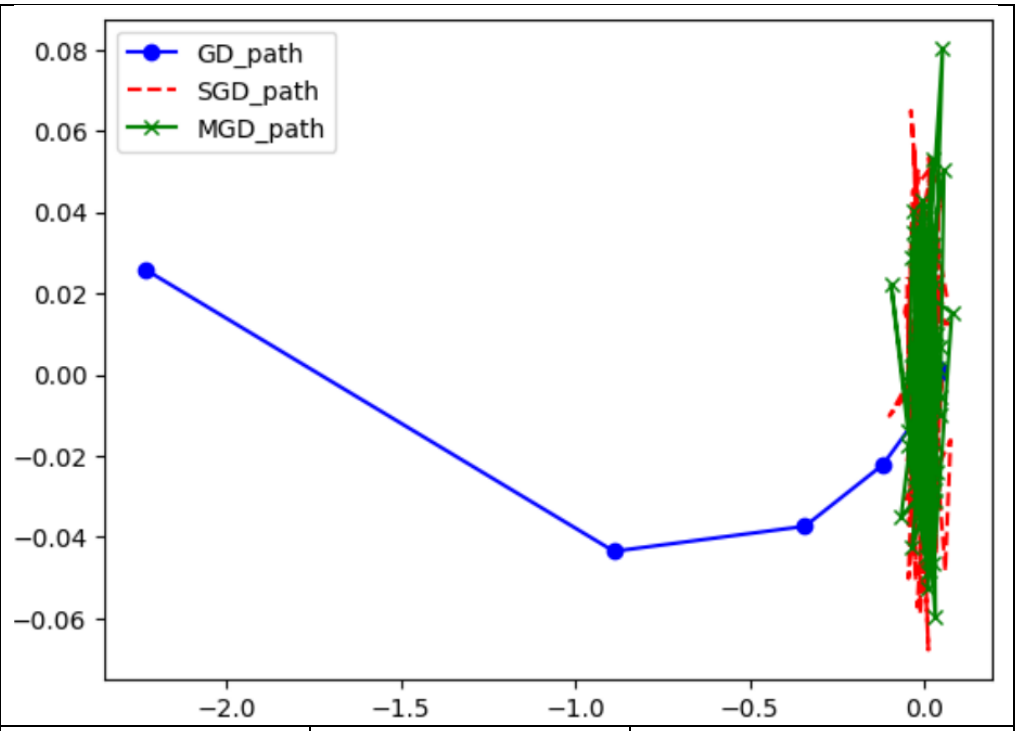
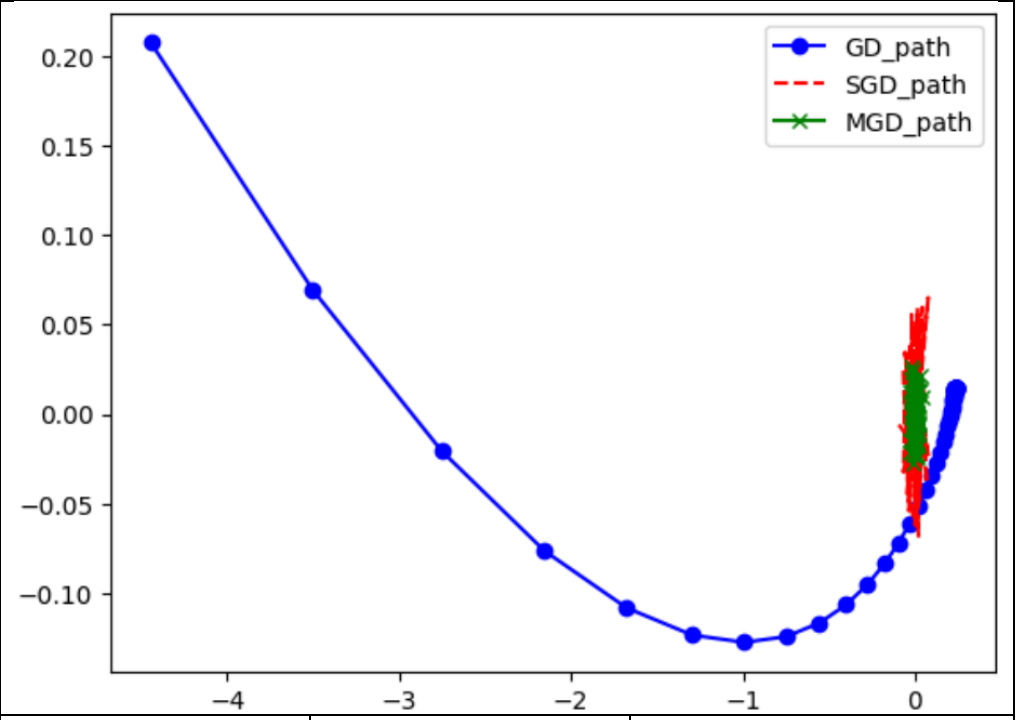
```
def gradient_descent_analysis(self, X, y, theta, eta):
      theta_path = []

      n_iterations = 100
      m = np.size(y)

      X_c = np.c_[np.ones((m,1)), X]

      for iteration in range(n_iterations):
        gradients = 2/m * X_c.T @ (X_c @ theta - y)
        theta = theta - eta * gradients
        theta_path.append(theta.flatten())

      return theta_path
```

```
theta_path_gd = np.array(analyzer.gradient_descent_analysis(X2, y2,
theta, eta=0.1))

pca = PCA(n_components=2)
theta_path_gd_2d = pca.fit_transform(theta_path_gd)
plt.plot(theta_path_gd_2d[:,0], theta_path_gd_2d[:,1],'b-o',
label='GD_path')
```

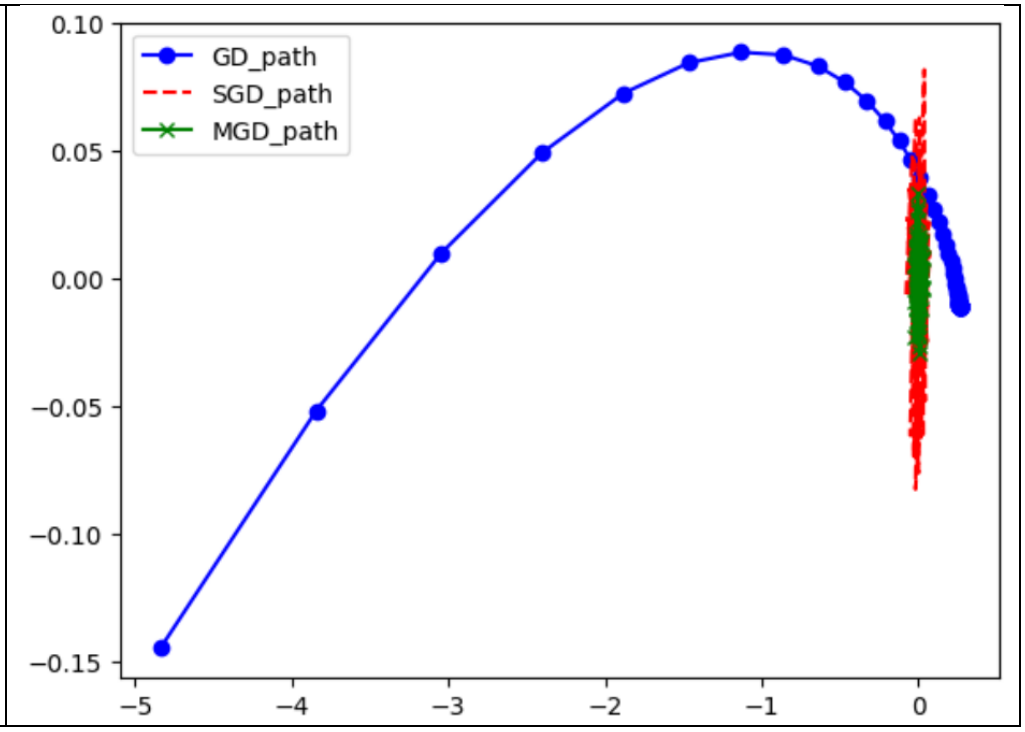
다음은 기존 gradient descent 코드를 살짝 변형한 것이다. 우선 기존의 코드에서 theta_path에 theta값을 넣어줄 때 flatten 함수를 사용해 1D vector 형태로 저장함으로써 2차원 이상일 때, PCA를 적용해 2D로 차원을 축소해서 그래프를 그릴 수 있도록 해주었다.

위와 똑같이 여러 batch size 변경(Full, Stochastic, mini-batch 함수도 유사하게 수정)하며 한번에 plot한 결과를 보면 아래 그래프와 같다.

<table>
<tr><td rowspan="5">1</td><td>Learning rate</td><td>Batch size</td><td>Initialization method</td></tr>
<tr><td rowspan="3">0.1</td><td>Full</td><td rowspan="3">Normal random initialization</td></tr>
<tr><td>stochastic</td></tr>
<tr><td>Mini-batch</td></tr>
<tr><td colspan="3"> 
</td></tr>
<tr><td rowspan="4">2</td><td>Learning rate</td><td>Batch size</td><td>Initialization method</td></tr>
<tr><td>0.3</td><td>Full</td><td rowspan="3">Normal random initialization</td></tr>
<tr><td>0.05</td><td>stochastic</td></tr>
<tr><td>0.3</td><td>Mini-batch</td></tr>
</table>

<table>
<tr><td></td><td> 
</td></tr>
<tr><td>3</td><td>
<table>
<tr><td>Learning rate</td><td>Batch size</td><td>Initialization method</td></tr>
<tr><td>0.1</td><td>Full</td><td rowspan="3">Zero initialization</td></tr>
<tr><td>0.05</td><td>stochastic</td></tr>
<tr><td>0.1</td><td>Mini-batch</td></tr>
</table>


</td></tr>
<tr><td>4</td><td>
<table>
<tr><td>Learning rate</td><td>Batch size</td><td>Initialization method</td></tr>
<tr><td>0.1</td><td>Full</td><td rowspan="3">Uniform random initialization</td></tr>
<tr><td>0.05</td><td>stochastic</td></tr>
<tr><td>0.1</td><td>Mini-batch</td></tr>
</table>
</td></tr>
</table>

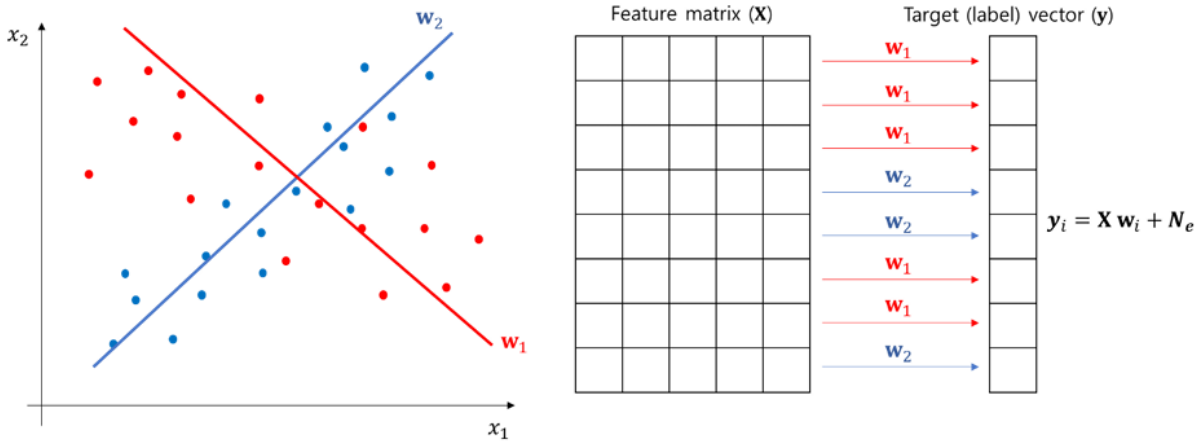


- Learning rate를 높게 하면 빠르게 수렴하는 경향을 보이지만 stochastic이나 mini-batch에서는 발산하는 확률이 높아지고, 낮은 learning rate는 더 안정적으로 수렴하는 양상을 보이는 것을 그래프를 통해 확인할 수 있다.

- Initialization method에 따르면, zero initialization은 모든 값이 0으로 초기화되면서 경사가 0으로 시작해 제대로 업데이트되지 않을 가능성도 있지만 위 3번 조건에서는 stochastic 방법에서 다른 시도에 비해 발산하는 영역이 가장 적게 나타났다. 또한, uniform random initialization은 학습은 더 빠를지언정 너무 큰 값으로 초기화되면 발산될 확률을 보인다.

- Full batch는 모든 데이터를 사용해서 정확하지만 시간이 오래 걸리고, stochastic은 빠르게 업데이트는 되지만 경로가 3가지 방법 중 가장 불규칙한 양상을 보인다. Mini-batch는 두 가지의 중간으로 상대적으로 균형 작인 수렴 양상을 보인다.

2. Repeat the problem #1 using two parameter vectors w_1, w_2, i.e., generated the label data using either w_1 or w_2 as the following figure. Here, the labeled data y_i are constructed using the parameter vector w_i with the probability p_i, and p_1 + p_2 = 1. Let p_1 = 90%.



두 개의 다른 모델 (w1, w2)를 위 그림과 같이 직교 형태처럼 선언하여 데이터를 생성하고, 주어진 p1에 따라 데이터 포인트들이 w1, w2에 의해 생성되도록 하는 class를 먼저 정의해주었다.

코드

```
def _generate_orthogonal_weights(self):
        # W1
        w1_true = np.random.randn(self.D + 1)

        # W2
        w2_initial = np.random.randn(self.D + 1)
        # proj_w1(w2_initial) = (w2_initial . w1_true) / ||w1_true||^2
* w1_true
        projection_on_w1 = (np.dot(w2_initial, w1_true) /
np.dot(w1_true, w1_true)) * w1_true
        w2_true = w2_initial - projection_on_w1

        self.w1_true = w1_true
        self.w2_true = w2_true
```

```
def _generate_labels(self, X):
        X_c = np.c_[np.ones(self.N), X]

        model_choice = np.random.rand(self.N)
        w1_mask = model_choice <= self.p1
        assignments = w1_mask.astype(int)

        y = np.zeros(self.N)
        y_pred_w1 = X_c @ self.w1_true
```

```
        y_pred_w2 = X_c @ self.w2_true

        y[w1_mask] = y_pred_w1[w1_mask]
        y[~w1_mask] = y_pred_w2[~w1_mask]

        noise = np.random.randn(self.N) * self.noise_level
        y = y + noise

        return y, assignments
```

위는 Class 내의 두 주요 함수이다.

- Generate_orthogonal_weights() 함수는 먼저 D+1 차원의 무작위 벡터 w1, w2를 생성한 후, $proj_{w1}(w2) = \frac{w2 \cdot w1}{\|w1\|^2}$w1 과 같은 공식을 적용해 무작위로 생성된 w1에서 빼 직교하도록 만들어주었다.

- Generate_labels(X) 함수는 주어진 행렬 X에 대해 두 개의 다른 모델(w1, w2)를 사용해 라벨을 생성하고, 각 데이터 포인트가 어느 모델에 의해 생성되었는지에 대한 정보를 반환하는 함수이다.

  우선 X 행렬에 1 값을 가진 열을 추가해준다. 이후 P1의 비율로 w1을, 나머지는 w2를 선택하도록 하기 위해 random으로 0~1 사이의 무작위 숫자를 생성해 p1(0.9)와 비교해 True 또는 False 값을 반환하는 mask를 정의해주었다. 위의 mask를 astype(int) 함수를 사용해 0 혹은 1로 바꿔서 assignments에 저장한다.

- 0으로 초기화한 배열 y를 생성한 뒤, w1 혹은 w2를 사용해 내적을 통해 예측값을 계산한 후 y에 선택한 데이터 포인트에 대한 예측값을 할당해준다.

1) Find the linear regression solution using y1 labeled data only, and measure its estimation error. (It will be the performance upper bound.)

코드

```
def find_regression_solution_y1(self):
        X_y1, y_y1 = self._get_y1_data()

        # Add a column of ones for the intercept
        X_y1_with_intercept = np.c_[np.ones(X_y1.shape[0]),
X_y1]

        # Calculate the normal equation solution
        XTX = X_y1_with_intercept.T @ X_y1_with_intercept
        XTy = X_y1_with_intercept.T @ y_y1
        w1_estimated = np.linalg.solve(XTX, XTy)
```

```
        return w1_estimated

def measure_weight_estimation_error(self, w_estimated, w_true):
        """
        Measures the L2 norm/Euclidean distance between the
estimated weights and a specified true weights vector.
        """
        error = np.linalg.norm(w_estimated - w_true)
        return error

def measure_prediction_error(self, w_estimated, X_data,
y_data):
        """
        Measures MSE of predictions made by an estimated weight
vector on given data.
        """
        # Add a column of ones for the intercept for prediction
        X_data_with_intercept = np.c_[np.ones(X_data.shape[0]),
X_data]

        y_predicted = X_data_with_intercept @ w_estimated
        mse = np.mean((y_data - y_predicted)**2)
        return mse



def analyze_y1_regression(self):
        print("\nAnalysis on y1 Data (Normal Equation -
Performance Upper Bound for w1) ")
        w1_estimated = self.find_regression_solution_y1()
        weight_estimation_error =
self.measure_weight_estimation_error(w1_estimated,
self.w1_true)

        X_y1, y_y1 = self._get_y1_data()
        prediction_error_mse =
self.measure_prediction_error(w1_estimated, X_y1, y_y1)


        print("Estimated w1 (using only y1 data):")
        if self.D + 1 > 10:
            print(np.round(w1_estimated[:5], 4), "...")
        else:
            print(np.round(w1_estimated, 4))

        print(f"\nWeight Estimation Error (L2 norm
||w1_estimated - w1_true||): {weight_estimation_error:.2f}")
```

```
        print(f"Prediction Error (MSE on y1 data):
{prediction_error_mse:.2f}")

        return w1_estimated, weight_estimation_error,
prediction_error_mse
```

- Find regression solution y1() 함수는 반환된 X 행렬과 레이블 데이터를 입력받아 X_y1 데이터에 1로 이루어진 열을 추가해 normal equation을 계산하는 방정식을 풀어 linear regression의 가중치 벡터를 구한다.
- Measure weight estimation error() 함수는 L2 norm으로 추정 가중치 값과 실제 값의 차이를 반환한다.
- Measure prediction error()에서는 추정된 가중치 벡터와 X, y 데이터를 사용해 MSE를 계산한다. 실제 y값과 y_predicted간의 MSE를 반환한다.
- Analyze_y1_regression 함수는 위의 세 함수를 조합해 y1 데이터에 대해 linear regression을 수행하고, 결과를 출력하는 함수이다. 먼저 find_regression_solution_y1 함수에서 구한 w1_estimated를 받아 가중치 추정 오류를 구한다. 이를 통해 예측 오류 MSE를 계산하고 출력한다.

출력 결과

```
Analysis on y1 Data (Normal Equation - Performance Upper Bound
for w1)
Estimated w1 (using only y1 data):
[ 2.2148 -1.6752 -0.7705  0.629   0.4045]

Weight Estimation Error (L2 norm ||w1_estimated - w1_true||):
0.03
Prediction Error (MSE on y1 data): 0.25
```

결과를 보면 weight estimation error가 0.03으로 매우 작아 w1을 거의 정확히 찾아냈다는 것을 의미한다. 또한 y1 데이터를 사용해 예측했을 때의 MSE는 0.25가 나온 것을 알 수 있다.

2) Find the solution using all data samples, and measure its estimation error.

코드

```
def find_regression_solution_all_data_normal_equation(self):
        # Use X_with_intercept already stored
        X_all_with_intercept = self.X_with_intercept
        y_all = self.y
```

```
        # Calculate the normal equation solution
        w_combined_estimated =
np.linalg.inv(X_all_with_intercept.T @ X_all_with_intercept) @
(X_all_with_intercept.T @ y_all)

        return w_combined_estimated


def analyze_all_data_regression_normal_equation(self):
        print("\n Analysis on All Data Samples (Normal Equation)
")
        w_combined_estimated =
self.find_regression_solution_all_data_normal_equation()
        prediction_error_mse =
self.measure_prediction_error(w_combined_estimated, self.X,
self.y)
        distance_to_w1_true =
self.measure_weight_estimation_error(w_combined_estimated,
self.w1_true)
        distance_to_w2_true =
self.measure_weight_estimation_error(w_combined_estimated,
self.w2_true)


        print("Estimated w (using all data - Normal Equation):")
        if self.D + 1 > 10:
            print(np.round(w_combined_estimated[:5], 4), "...")
        else:
            print(np.round(w_combined_estimated, 4))

        print(f"\nPrediction Error (MSE on all data):
{prediction_error_mse:.2f}")
        print(f"Distance of estimated w to w1_true (L2 norm):
{distance_to_w1_true:.2f}")
        print(f"Distance of estimated w to w2_true (L2 norm):
{distance_to_w2_true:.2f}")

        return w_combined_estimated, prediction_error_mse,
distance_to_w1_true, distance_to_w2_true
```

- Find_regression_solution_all_data_normal_equations 함수는 모든 데이터 X, y를 이용해 Normal equation 방식으로 linear regression solution을 구하는 함수로 가중치를 리턴한다.

- 위 함수에서 구한 w 값을 analyze_all_data_regression_normal_equation() 함수에서 분석하고 SE 계산과 함께 결과값들을 출력하는 함수이다.

  반환된 결과를 살펴보면 아래와 같다.

출력 결과

```
Analysis on All Data Samples (Normal Equation)
Estimated w (using all data - Normal Equation):
[ 1.9253 -1.4624 -0.8541  0.8053  0.1406]

Prediction Error (MSE on all data): 2.09
Distance of estimated w to w1_true (L2 norm): 0.51
Distance of estimated w to w2_true (L2 norm): 3.88
```

출력 결과를 확인해 보면 weight는 w1에 더 가깝고 w2와는 더 거리가 먼 것을 보아 w1 쪽 특성을 전체 데이터 중 더 많이 반영한 weight를 사용했다고 볼 수 있다. Y1만 썼을 때는 MSE가 0.25였는데 반해 전체 데이터를 다 쓰며 MSE가 2.09로 커지게 된 것을 확인해볼 수 있다. 왜냐하면 y1뿐 아니라 y2 데이터도 섞이며 w1과 w2 사이 중간쯤의 형태로 weight를 추정했기 때문으로 생각된다.

3) Use the gradient descent method and analyze its performance under various conditions.

Gradient descent의 condition에 따라 얼마나 잘 수렴하고 정확한 w를 찾는지를 분석하기 위해 아래와 같이 코드를 구성하였다.

코드

```
def analyze_gd_regression(self, learning_rate, n_iterations,
initial_w = None, plot_loss = True):
        print("\n Analysis on All Data Samples (Gradient
Descent) ")
        print(f"  Parameters: Learning Rate={learning_rate},
Iterations={n_iterations}")

        w_gd_estimated, loss_history =
self.find_regression_solution_gd(
            learning_rate=learning_rate,
            n_iterations=n_iterations,
            initial_w=initial_w
        )

        prediction_error_mse =
self.measure_prediction_error(w_gd_estimated, self.X, self.y)
        distance_to_w1_true =
self.measure_weight_estimation_error(w_gd_estimated,
self.w1_true)
        distance_to_w2_true =
self.measure_weight_estimation_error(w_gd_estimated,
self.w2_true)
```

```
        print("Estimated w (using all data - Gradient
Descent):")
        if self.D + 1 > 10:
            print(np.round(w_gd_estimated[:5], 4), "...")
        else:
            print(np.round(w_gd_estimated, 4))

        print(f"\nFinal Prediction Error (MSE on all data):
{prediction_error_mse:.2f}")
        print(f"Distance of estimated w to w1_true (L2 norm):
{distance_to_w1_true:.2f}")
        print(f"Distance of estimated w to w2_true (L2 norm):
{distance_to_w2_true:.2f}")

        if plot_loss:
            plt.figure(figsize=(10, 6))
            plt.plot(loss_history)
            plt.xlabel('Iteration')
            plt.ylabel('MSE Loss')
            plt.title(f'GD Loss Convergence (LR={learning_rate},
Iters={n_iterations})')
            plt.grid(True)
            plt.show()


        return w_gd_estimated, loss_history,
prediction_error_mse, distance_to_w1_true, distance_to_w2_true
```

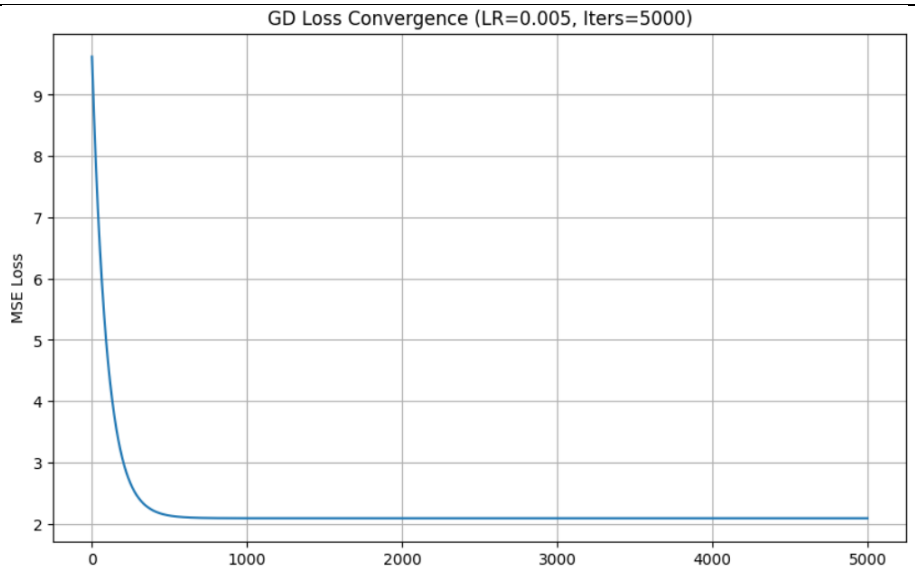
- Analyze_gd_regression 함수는 weight를 학습하고 MSE와 L2 norm 등을 통해 performance를 비교해볼 수 있는 함수이다.

  먼저 find_regression_solution_gd()를 통해 lr과 iteration수로 Gradient descent를 수행한다.

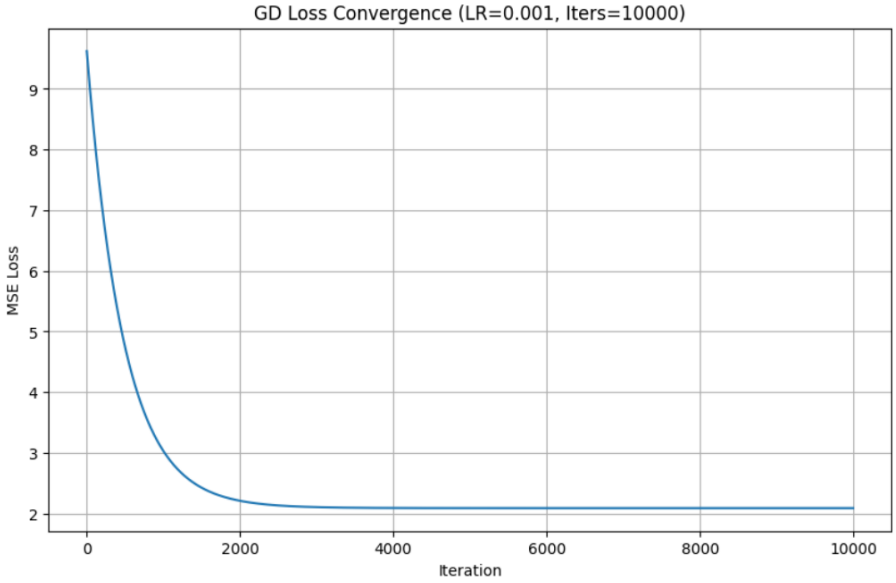
  이를 바탕으로 최종 weight으로 MSE를 계산하고 w1, w2의 L2 norm을 측정해 결과들을 출력한다.

출력 결과

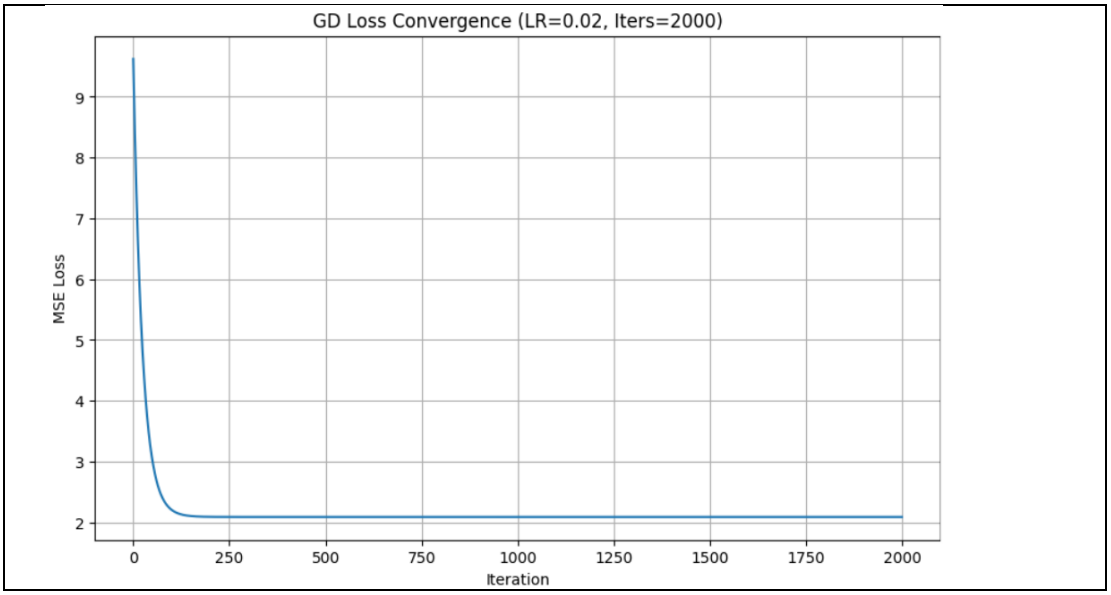
```
Analysis on All Data Samples (Gradient Descent)
  Parameters: Learning Rate=0.005, Iterations=5000
Estimated w (using all data - Gradient Descent):
[ 1.9253 -1.4624 -0.8541  0.8053  0.1406]

Final Prediction Error (MSE on all data): 2.09
Distance of estimated w to w1_true (L2 norm): 0.51
  Distance of estimated w to w2_true (L2 norm): 3.88
```




```
Analysis on All Data Samples (Gradient Descent)
  Parameters: Learning Rate=0.001, Iterations=10000
Estimated w (using all data - Gradient Descent):
[ 1.9252 -1.4624 -0.8541  0.8053  0.1406]

Final Prediction Error (MSE on all data): 2.09
Distance of estimated w to w1_true (L2 norm): 0.51
  Distance of estimated w to w2_true (L2 norm): 3.88
```




```
Analysis on All Data Samples (Gradient Descent)
  Parameters: Learning Rate=0.02, Iterations=2000
Estimated w (using all data - Gradient Descent):
[ 1.9253 -1.4624 -0.8541  0.8053  0.1406]

Final Prediction Error (MSE on all data): 2.09
Distance of estimated w to w1_true (L2 norm): 0.51
  Distance of estimated w to w2_true (L2 norm): 3.88
```



위 출력 결과를 살펴보면 learning rate=0.005, 5000번 반복한 경우 loss가 안정적으로 감소하며 적절한 성능을 얻을 수 있었다. 그에 반해 learning rate=0.001로 더 작을 경우, 수렴은 하나 매우 많은 반복 횟수가 필요했던 것을 확인할 수 있다. 학습이 0.02로 큰 경우, loss가 불안정하거나 발산해 학습이 제대로 이루어지지 않는 현상을 확인할 수 있다.

4) Analyze the performanc edegradation by outlier samples (y2) from 1)-3). Check the performance by controlling the number of outlier samples, i.e., by adjusting p1.

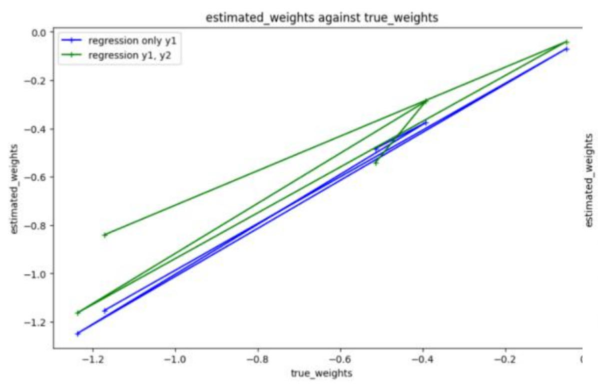
P1을 값을 낮춰가며 outlier가 성능에 어떠한 영향을 미치는지 분석해보는 문제이다. 따라서 각각의 출력 결과에 따라 비교해보면 아래 표와 같다.

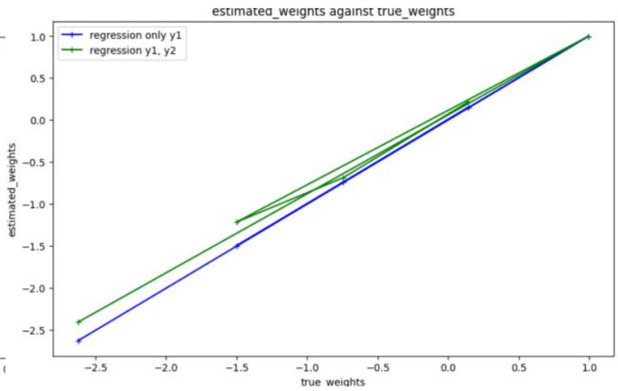
P1	Y1 weight error	Y1 MSE	All data weight error	All data MSE	Distance to w2
0.9	0.02	0.27	0.19	0.6	1.89
0.8	0.03	0.25	0.51	2.09	3.88
0.7	0.04	0.26	0.3	1.01	2.57
0.6	0.03	0.24	0.52	2.13	3.79
0.5	0.04	0.26	0.36	1.02	2.17

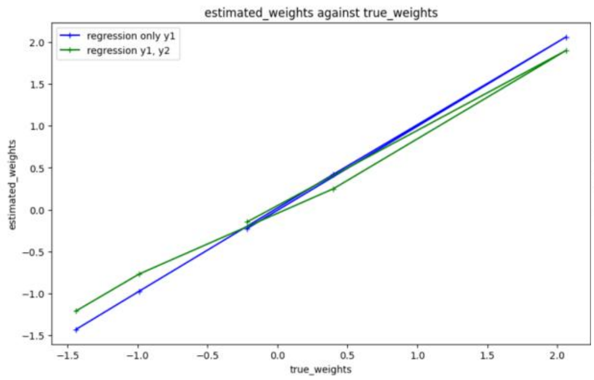
- Y1 데이터만을 사용하는 경우는 weight estimation error와 MSE도 모두 아주 작은 값을 보였다.
- 그러나 전체 데이터를 사용하는 경우, p1=0.9일 때까지는 w1의 값을 대체적으로 잘 찾는 편이었으나, p1=0.8부터 값들이 커지며 performance가 악화되는 것들을 볼 수 있다. 따라

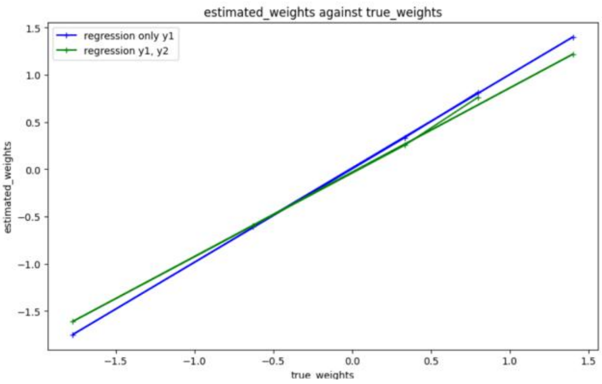
서 p1이 감소할수록 전체 데이터를 사용했을 때 w1 estimation 성능이 전반적으로 나빠진다는 것을 알 수 있다.

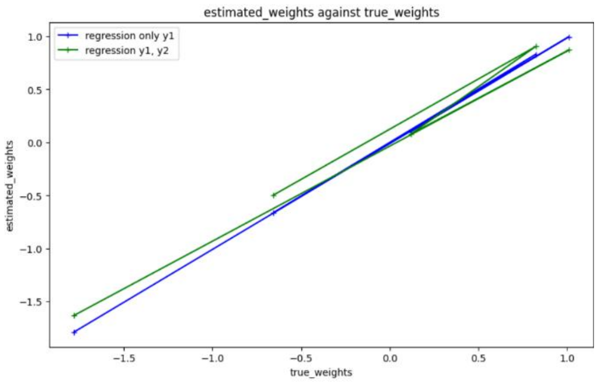
- 모든 결과에서 w1에 비해 w2와의 거리가 훨씬 컸는데 이는 주로 w1의 모델을 학습했다는 것을 의미한다. 그러나 p1이 작아질수록 w2와의 거리도 약간씩 작아지는데, 이는 데이터에 y2의 영향이 커졌기 때문이다.











위 그래프는 위에서부터 순차적으로

P1 = 0.5 p1 = 0.6

P1 = 0.7 p1 = 0.8

P1 = 0.9일 때의 결과 그래프이다.

파란 선은 y1 데이터만 사용했을 때의 weight 추정 결과이고 초록색은 y1, y2 모두 섞은 데이터를 사용했을 때의 결과이다. P1 값이 커질수록, 그리고 파란 선일수록 데이터가 크게 벗어난 값들이 줄어드는 것을 직관적으로 볼 수 있다.