

COGS 118A: Supervised Machine Learning Algorithms

Homework Assignment 4

1 (10 points) Conceptual Questions

(1.1)

- A. $f(x) + g(x)$ is always monotonically increasing
- D. $f(x^3)$ is always monotonically increasing

(1.2)

- A. Matrix inverse could be expensive for computing the closed form solution in least square estimation when the dataset consists of samples in high dimension spaces.

(1.3)

$$f(x) = x(10 - x) = -x^2 + 10x$$

$$\begin{aligned} f'(x) &= 0 \\ -2x + 10 &= 0 \\ x &= 5 \\ f(5) &= 5(10 - 5) = 25 \end{aligned}$$

- A. $\text{argmax}_x f(x) = 5$
- D. $\max_x f(x) = 25$

(1.4)

- B. Given (2) and (3), we can prove that (1) holds.

Counterexample: an equation with two minima with x_0 as the global minimum. In this case there could also be another minimum that would make $f(x)$ not convex.

(1.5)

False. Different models weigh outliers differently. To know which model is better, you could compare their performance on a test set.

2 (12 points) Convex

(2.a)

non-convex

(2.b)

convex

(2.c)

convex

(2.d)

non-convex

(2.e)

convex

(2.f)

non-convex

3 (7 points) Argmin and Argmax

$$\operatorname{argmin}_w G(w) = 15$$

$$\operatorname{argmax}_w G(w) = 25$$

$$w^* = \operatorname{argmin}_w [10 - 3 \times \ln(G(w))] =$$

$$\operatorname{argmin}_w [-\ln(G(w))] =$$

$$\operatorname{argmax}_w \ln(G(w)) =$$

$$\operatorname{argmax}_w G(w) = 25$$

$$w^* = 25$$

4 (6 points) Error Metrics

(4.1)

C. 1/4

(4.2)

B. 3/10

(4.3)

B. 9/10

5 (30 points) Coding: Hyper-plane Estimation

In [1]: `# Importing Important packages (nothing to add to this cell)`

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%config InlineBackend.figure_format = 'retina'
```

```
In [2]: # Loading the data (nothing to add to this cell)
```

```
X_and_Y = np.load('./hyperplane-estimation.npy')
X1 = X_and_Y[:, 0]      # Shape: (900,)
X2 = X_and_Y[:, 1]      # Shape: (900,)
Y = X_and_Y[:, 2]       # Shape: (900,)
print(X1.shape, X2.shape, Y.shape)
```

```
(900,) (900,) (900,)
```

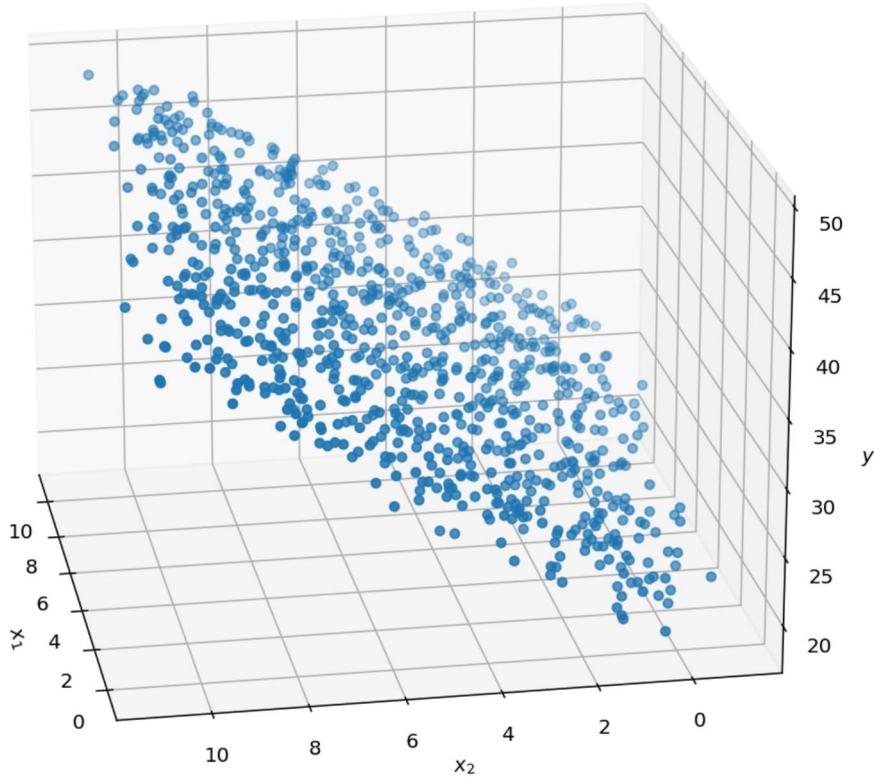
Original Data

```
In [3]: # Visualization of the original datapoints (nothing to add to this cell).
```

```
def vis(w0, w1, w2):
    draw_plane = (w0 is not None) and (w1 is not None) and (w2 is not None)
    if draw_plane:
        num = 30
        X_plane_range = np.linspace(0,10,num)
        X_plane_range = np.linspace(0,10,num)
        X1_plane, X2_plane = np.meshgrid(X_plane_range, X_plane_range)
        Y_plane = w0 + w1 * X1_plane + w2 * X2_plane

        fig = plt.figure(figsize = (7, 7))
        ax = Axes3D(fig, elev = 20, azim = 170)
        ax.scatter(X1, X2, Y)
        if draw_plane:
            ax.scatter(X1_plane, X2_plane, Y_plane)
            ax.set_xlabel('$x_1$')
            ax.set_ylabel('$x_2$')
            ax.set_zlabel('y')
        plt.show()

vis(None, None, None)
```



Hyperplane Estimation Using the Closed Form Solution

Assume data points are represented as matrices X and Y , please use the closed form solution to calculate the parameters W .

Hint: You may refer to HW3 Q4.

```
In [4]: # Estimating W, which defines the hyperplane (you need to add code to this cell as indicated)
# y = w0 + w1*x1 + w2*x2

X = np.matrix(np.hstack((np.ones((len(X1),1)),
                        X1.reshape(-1,1),
                        X2.reshape(-1,1)))) # X contains 1, X1 and X2.

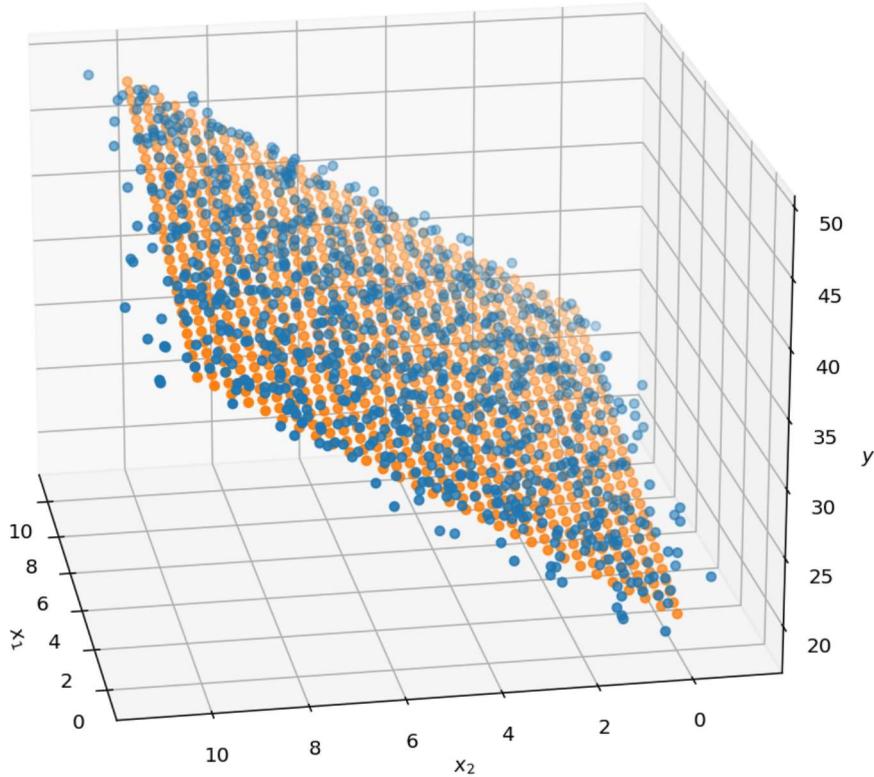
Y = Y
# Compute W using the closed form solution.
W = np.linalg.inv(X.T @ X) @ X.T @ Y

w0, w1, w2 = np.array(W).reshape(-1)
print('y = {:.2f} + {:.2f}*x1 + {:.2f}*x2'.format(w0, w1, w2))

y = 19.30 + 0.98*x1 + 1.94*x2
```

```
In [5]: # Visualization of the original datapoints and estimated plane (nothing to add to this cell)

vis(w0, w1, w2)
```



Hyperplane Estimation Using Gradient Descent

In this problem, we would like to use the gradient descent to calculate the parameters W for the hyperplane. If we have an error function (a.k.a objective function or loss function), then a typical gradient descent algorithm contains the following steps:

Step 1. Initialize the parameters W .

for i = 1 to #iterations:

- **Step 2.** Compute the gradient $\nabla \mathcal{L}(W) = \frac{\partial \mathcal{L}(W)}{\partial W}$.
- **Step 3.** Update the parameters $W \leftarrow \mathcal{L}(W) = W - \eta \frac{\partial \mathcal{L}(W)}{\partial W}$ where η is the learning rate.

```
In [6]: # Gradient of L(W) with respect to W (you need to add code to this cell as indicated below)
```

```
def grad_L_W(X, Y, W):
    return 2 * X.T @ X @ W - 2 * X.T @ Y
```

```
In [7]: # Estimating W, which defines the hyperplane (you need to add code to this cell as indicated below)
# y = w0 + w1*x1 + w2*x2
```

```
# Some settings.
```

```

Y = Y.reshape(-1, 1)
print(X.shape, Y.shape, W.shape)
iterations      = 20000
learning_rate   = 0.000001

# Gradient descent algorithm.
# Step 1. Initialize the parameters W.
W = np.matrix(np.zeros((3,1)))

for i in range(iterations):
    # Step 2. Calculate the gradient of L(W) w.r.t. W.
    grad = grad_L_W(X, Y, W)

    # Step 3. Update parameters W.
    W = W - learning_rate * grad

w0, w1, w2 = np.array(W).reshape(-1)
print('y = {:.2f} + {:.2f}*x1 + {:.2f}*x2'.format(w0, w1, w2))

```

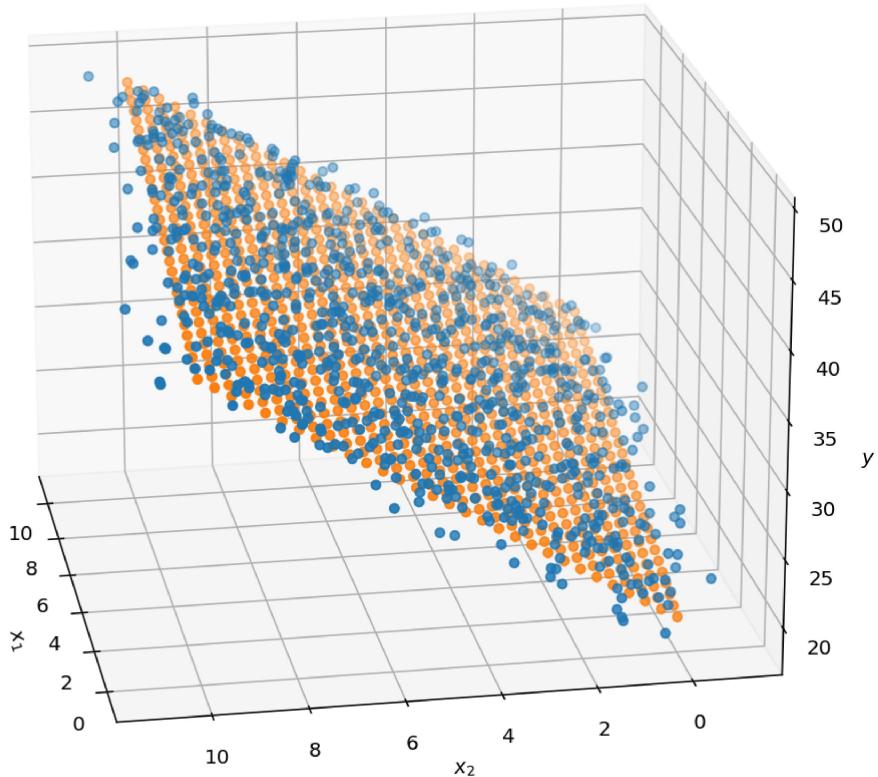
(900, 3) (900, 1) (1, 3)
 $y = 19.22 + 0.99*x1 + 1.95*x2$

In [8]: # of the original datapoints and estimated plane (nothing to add to this cell).

```

vis(w0, w1, w2)

```



6 (30 points) Coding: Parabola Estimation

```
In [9]: # Importing Important packages (nothing to add to this cell)
```

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
```

```
In [10]: # Loading the data (nothing to add to this cell)
```

```
X_and_Y = np.load('./parabola-estimation.npy')
old_X = X_and_Y[:, 0] # Shape: (300,)
Y = X_and_Y[:, 1] # Shape: (300,)
old_X.shape
```

```
Out[10]: (300,)
```

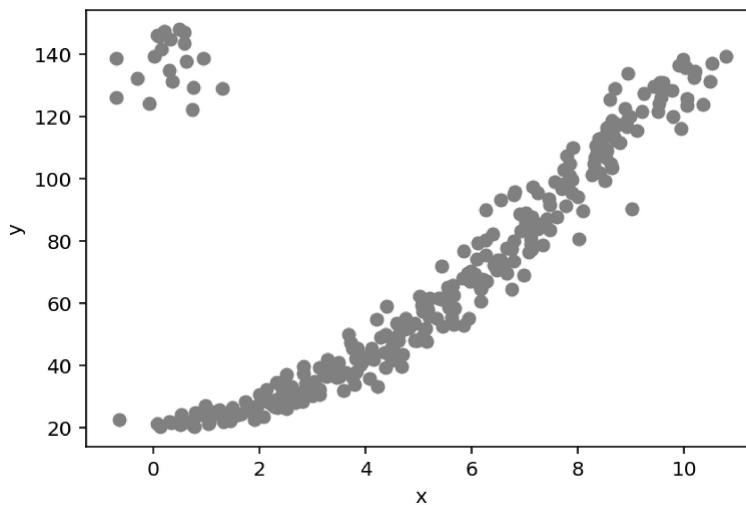
Original Data

```
In [11]: # Visualization of the original datapoints (nothing to add to this cell).
```

```
def vis(w0, w1, w2):
    draw_plane = (w0 is not None) and (w1 is not None) and (w2 is not None)
    if draw_plane:
        X_line = np.linspace(0,10,300)
        Y_line = w0 + w1 * X_line + w2 * (X_line**2)
        plt.plot(X_line, Y_line, color='orange')

    plt.scatter(old_X, Y, color='gray')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

vis(None, None, None)
```



(6.1) Squared L_2 Norm (5 points)

Assume data points are represented as matrices X and Y , please use the closed form solution to calculate the parameters W .

Hint: You may refer to HW3 Q4.

```
In [12]: # Estimating W, which defines the parabola estimation (you need to add code to this cell)
#  $y = w_0 + w_1*x + w_2*x^2$ 

X = np.matrix(np.hstack((np.ones((len(old_X),1)),
old_X.reshape(-1,1),
(old_X**2).reshape(-1,1)))))

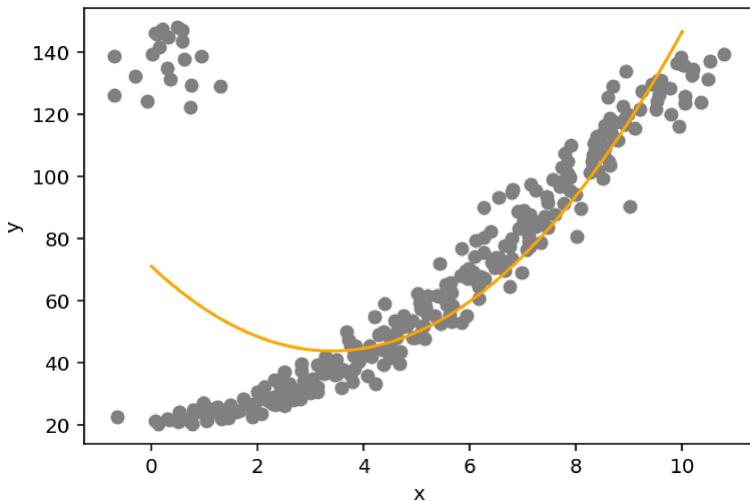
W = np.linalg.inv(X.T * X) @ X.T @ Y

w0, w1, w2 = np.array(W).reshape(-1)
print('y = {:.2f} + {:.2f}*x + {:.2f}*x^2'.format(w0, w1, w2))

y = 71.07 + -16.06*x + 2.36*x^2
```

```
In [13]: # Visualization of the original datapoints and estimated plane (nothing to add to this)

vis(w0, w1, w2)
```



(6.2) L_1 Norm (10 points)

In this problem, we would like to use the gradient descent to calculate the parameters W for the parabola. If we have a loss function $\mathcal{L}(W)$, then a typical gradient descent algorithm contains the following steps:

Step 1. Initialize the parameters W .

for i = 1 to #iterations:

- **Step 2.** Compute the gradient $\nabla \mathcal{L}(W) = \frac{\partial \mathcal{L}(W)}{\partial W}$.
- **Step 3.** Update the parameters $W \leftarrow \mathcal{L}(W) = W - \eta \frac{\partial \mathcal{L}(W)}{\partial W}$ where η is the learning rate.

Hint: You may refer to HW3 Q5.

```
In [14]: # Gradient of L(W) with respect to W (you need to add code to this cell as indicated below)

def grad_L_W(X, Y, W):
    return (np.sign(X @ W - Y).T @ X).T
```

```
In [15]: # Estimating W, which defines the hyperplane using gradient descent (you need to add code to this cell as indicated below)
```

```

#  $y = w_0 + w_1*x + w_2*x^2$ 

# Some settings.
Y = Y.reshape(-1, 1)
iterations      = 300000
learning_rate = 0.000001

# Gradient descent algorithm.
# Step 1. Initialize the parameters W.
W = np.array([[0], [0], [0]])

for i in range(iterations):
    # Step 2. Calculate the gradient of L(W) w.r.t. W.
    grad = grad_L_W(X, Y, W)

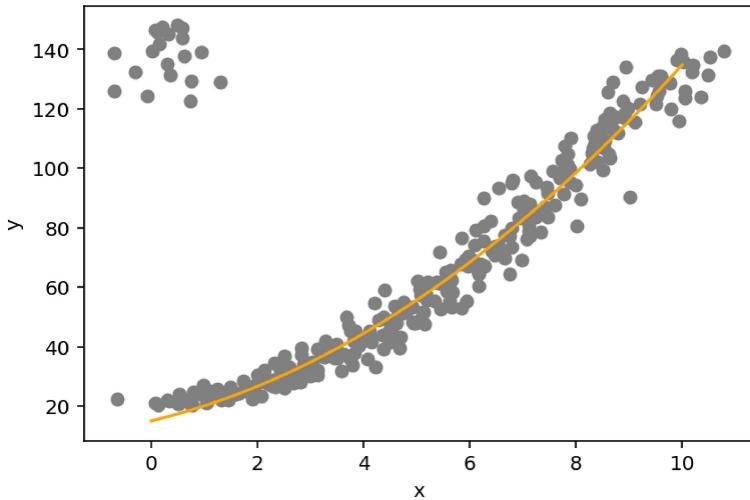
    # Step 3. Update parameters W.
    W = W - learning_rate * grad

# Print the parameters of the parabola.
w0, w1, w2 = np.array(W).reshape(-1)
print('y = {:.2f} + {:.2f}*x + {:.2f}*x^2'.format(w0, w1, w2))

# Visualization.
vis(w0, w1, w2)

```

$$y = 15.10 + 4.27*x + 0.77*x^2$$



(6.3) Squared L_2 Norm and L_1 Norm (12 points)

In this problem, we would like to use the gradient descent to calculate the parameters W for the parabola. The loss function $\mathcal{L}(W)$ now contains two parts: A squared L_2 norm and a L_1 norm. A coefficient α is used to control the ratio of these two norms:

$$\mathcal{L}(W) = \sum_{i=1}^n \left(\alpha(\mathbf{x}_i^T W - y_i)^2 + (1-\alpha)|\mathbf{x}_i^T W - y_i| \right) \quad (1)$$

$$= \alpha \|XW - Y\|_2^2 + (1-\alpha) \|XW - Y\|_1 \quad (2)$$

Note: It may take 2~3 mins to run the algorithm.

In [16]: # Gradient of $L(W)$ with respect to W (you need to add code to this cell as indicated by

```
def grad_L_W(X, Y, W, alpha):
    return alpha * (2 * X.T @ X @ W - 2 * X.T @ Y) + (1 - alpha) * ((np.sign(X @ W - Y)
```

In [17]:

```
# Function to use gradient descent to estimate parabola given a list of alphas
# (you need to add code to this cell as indicated below).
# Hint: For each alpha, you need to use gradient descent, hence, you need to write a lo
```

```
def parabola(alpha_list):
    # Some settings.
    global Y
    plt.scatter(old_X, Y, color='gray')
    Y = Y.reshape(-1, 1)
    iterations      = 300000
    learning_rate   = 0.000001

    # Loop over alpha(s).
    for alpha in alpha_list:

        # Gradient descent algorithm.
        # Step 1. Initialize the parameters W.
        W = np.array([[0], [0], [0]])

        ##### To be filled. #####
        for i in range(iterations):
            # Step 2. Calculate the gradient of L(W) w.r.t. W.
            grad = grad_L_W(X, Y, W, alpha)

            # Step 3. Update parameters W.
            ##### Write your code here #####
            W = W - learning_rate * grad

        # Print the parameters of the parabola.
        w0, w1, w2 = np.array(W).reshape(-1)
        print('When alpha = {}.'.format(alpha))
        print('y = {:.2f} + {:.2f}*x + {:.2f}*x^2'.format(w0, w1, w2))

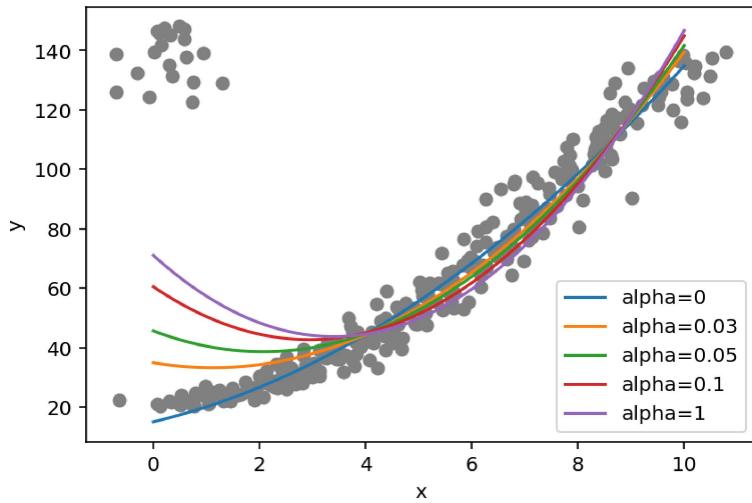
        # Visualization.
        X_line = np.linspace(0, 10, 300)
        Y_line = w0 + w1 * X_line + w2 * (X_line**2)
        plt.plot(X_line, Y_line, label='alpha={}'.format(alpha))

    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

In [18]:

```
# Estimate the parabolas given the list of alpha(s).
parabola(alpha_list=[0, 0.03, 0.05, 0.1, 1])
```

```
When alpha = 0,
y = 15.10 + 4.27*x + 0.77*x^2
When alpha = 0.03,
y = 34.97 + -3.02*x + 1.35*x^2
When alpha = 0.05,
y = 45.68 + -6.78*x + 1.64*x^2
When alpha = 0.1,
y = 60.53 + -12.10*x + 2.05*x^2
When alpha = 1,
y = 71.07 + -16.06*x + 2.36*x^2
```



(6.4) Comparison (3 points)

At $\alpha = 0$, we are using L_1 norm, which is the least influenced by outliers. At $\alpha = 1$, we are using L_2 norm, which is the most influenced by outliers since a large distance squared becomes much larger than a small distance squared. As alpha increases, outliers influence the model less.