EECS 553 HW4

Lingqi Huang

September 2024

1 Problem 1

Part(a): Notice that the loss function $J(w, b)$ can be written as

$$J(w,b) = \sum_{i=1}^{n} \left(\frac{1}{n} (L(y_i, w^T x_i + b) + \frac{\lambda}{2} ||w||^2) \right)$$

Thus, we observe that

$$J_i(w,b) = \frac{1}{n} \left(L(y_i, w^T x_i + b) + \frac{\lambda}{2} ||w||^2 \right) = \frac{1}{n} \left(\max\{0, 1 - y_i(w^T x_i + b)\} + \frac{\lambda}{2} ||w||^2 \right)$$

Now we set $\tilde{x}_i = [1, x_{i1}, \dots, x_{id}]^T$, $\theta = [b, w^T]^T$, we conclude that our J_i can be re-written as

$$J_i(\theta) = \frac{1}{n} \left(\max\{0, 1 - y_i \theta^T \tilde{x}_i\} + \frac{\lambda}{2} ||w||^2 \right)$$

Now for convenience, we only discuss two cases, which are $y_i \theta^T \tilde{x}_i < 1$, and $y_i \theta^T \tilde{x}_i \geq 1$. The last case make senses because the hinge loss function is non-differentiable as point 0, and so we let the subgradient to be 0 which is the right derivative of the hinge loss if the hinge loss is 0, ie, $1 - y_i \theta^T \tilde{x}_i = 0$. Thus, we can easily show that

$$u_i = \nabla J_i(\theta) = \begin{cases} \frac{1}{n} \left(-y_i \tilde{x}_i + \lambda \begin{bmatrix} 0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} \right) & \text{if } y_i \theta^T \tilde{x}_i < 1 \\ \frac{\lambda}{n} \begin{bmatrix} 0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} & \text{if } y_i \theta^T \tilde{x}_i \geq 1 \end{cases}$$

by the fact that hinge loss is convex and chain rule.

Part(b): By the code in the attachment, we find that the estimated parameters are $w_1 = -17.816, w_2 = -9.117, b = 12.06$, the margin is $\frac{1}{||w||} = 0.04997$, and the minimum achieved value of the objective function is 0.4498. You can check the diagrams in the next page.

Part(c): By the code in the attachment, we find that the estimated parameters are $w_1 = -5.82, w_2 = -4.41, b = 4.005$, the margin is 0.13683, and the minimum achieved value of the objective function is 0.25827. We can see that the applying SGD algorithm could converges much faster than using subgradient method.

Pictures in 1(b)

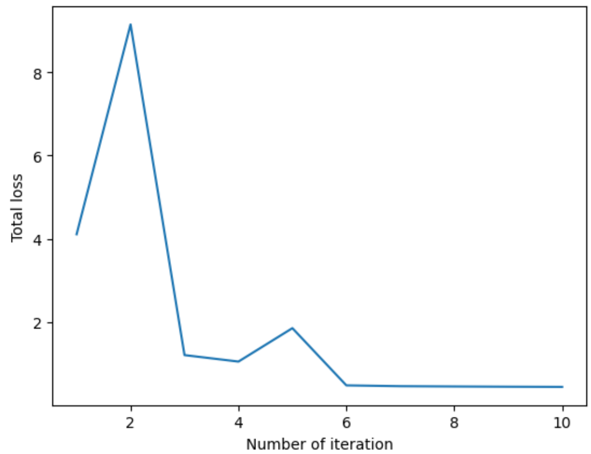


Figure 1: The loss function value VS Number of iteration using subgradient method

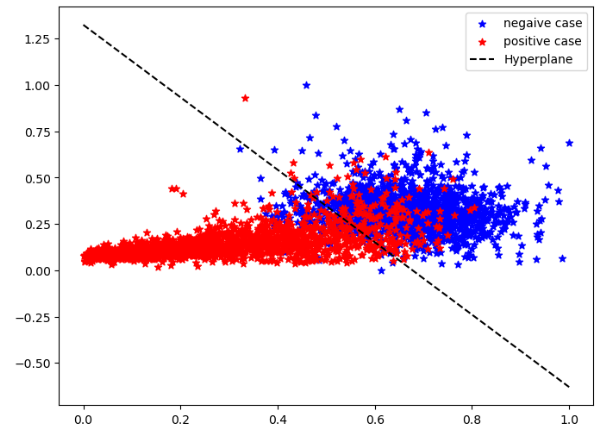


Figure 2: Visualization of data and the learned line using subgradient method

Pictures in 1(c)

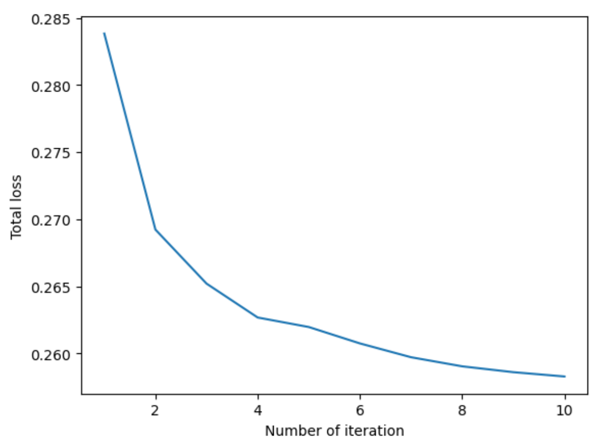


Figure 3: The loss function value VS Number of iteration using stochastic gradient descent method

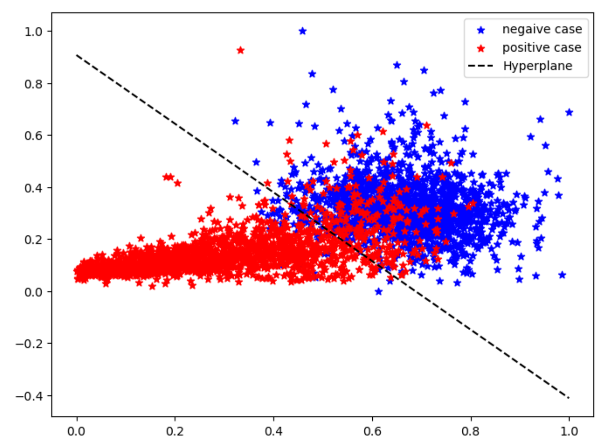


Figure 4: Visualization of data and the learned line using stochastic gradient descent method

2 Problem 2

Part(a): Notice that we can write $g(w_j)$ at iteration t as

$$g(w_j) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \sum_{k=1}^{d} w_k^{(t)} x_{ik} - b^{(t)})^2 + \lambda ||w||_1$$

Now we define function $h(x)$ such that

$$h(x) = \begin{cases} -1 & \text{if } x < 0 \\ [-1, 1] & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Thus, we find that the sub-differential could be written as

$$\partial g(w_j) = \frac{2}{n}\sum_{i=1}^{n}(-y_i x_{ij} + x_{ij}\sum_{k=1}^{d} w_k^{(t)} x_{ik} + b^{(t)} x_{ij}) + \lambda h(w_j)$$

Notice that we can observe that

$$\frac{2}{n}\sum_{i=1}^{n} x_{ij}\left(\sum_{k=1}^{d} w_k^{(t)} x_{ik}\right) = \left(\frac{2}{n}\sum_{i=1}^{n} x_{ij}^2\right) w_j + \frac{2}{n}\sum_{i=1}^{n} x_{ij}\left(\sum_{\substack{k=1\\k\neq j}}^{d} w_k^{(t)} x_{ik}\right)$$

where the second term could be written as the

$$\frac{2}{n}\sum_{i=1}^{n} x_{ij}\left(\sum_{\substack{k=1\\k\neq j}}^{d} w_k^{(t)} x_{ik}\right) = \frac{2}{n}\sum_{i=1}^{n} x_{ij}(w_{-j}^{(t)})^T x_{i,-j}$$

where $w_{-j}^{(t)}, x_{i,-j}$ are the notation defined in the problem. Thus, we get that

$$\partial g(w_j) = \left(\frac{2}{n}\sum_{i=1}^{n} x_{ij}^2\right) w_j - \frac{2}{n}\sum_{i=1}^{n} x_{ij}(y_i - (w_{-j}^{(t)})^T x_{i,-j} - b^{(t)}) + \lambda h(w)$$

Now if we define

$$a_j^{(t)} = \frac{2}{n}\sum_{i=1}^{n} x_{ij}^2, \qquad c_j^{(t)} = \frac{2}{n}\sum_{i=1}^{n} x_{ij}(y_i - (w_{-j}^{(t)})^T x_{i,-j} - b^{(t)})$$

and combine the definition of $h(x)$, we finally conclude that

$$\partial g(w_j) = \begin{cases} a_j^{(t)} w_j - c_j^{(t)} - \lambda, & w_j < 0 \\ [a_j^{(t)} w_j - c_j^{(t)} - \lambda, a_j^{(t)} w_j - c_j^{(t)} + \lambda], & w_j = 0 \\ a_j^{(t)} w_j - c_j^{(t)} + \lambda, & w_j > 0 \end{cases}$$

This completes the proof.

Part(b): Let's first consider the case of $c_j^{(t)} < -\lambda$, which means $w_j = \frac{c_j^{(t)}+\lambda}{a_j^{(t)}} < 0$, and this is the unique w_j such that $0 \in \partial g(w_j)$. when $c^{(t)} \in [-\lambda, \lambda]$, we automatically have $w_j = 0$ that only the second condition can be satisfied. For the similar reason, when $c^{(t)} > \lambda$, we have the unique w_j that satisfies the third condition, where $w_j = \frac{c_j - \lambda}{a_j^{(t)}} > 0$. Thus, we conclude that the optimal value of w_j is given by the soft-threshoding formula, that

$$w_j = soft\left(\frac{c_j^{(t)}}{a_j^{(t)}}, \frac{\lambda}{a_j^{(t)}}\right)$$

where

$$soft(\alpha, \beta) = \begin{cases} \alpha - \beta, & \alpha > \beta \\ 0, & \alpha \in [-\beta, \beta] \\ \alpha + \beta, & \alpha < -\beta \end{cases}$$

This completes the proof.

3 Problem 3

Part(a): We have verified that rows of thus sphered $\mathbb{X}$ training data matrix have zero sample mean and unit variance. You can see the picture below as well as code in the end of this pdf.

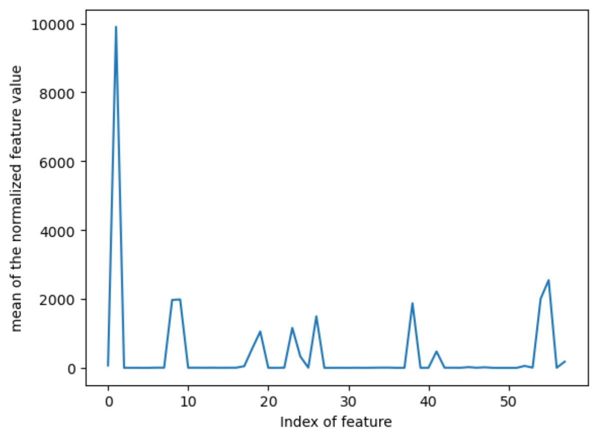


Figure 5: feature index VS Mean of feature

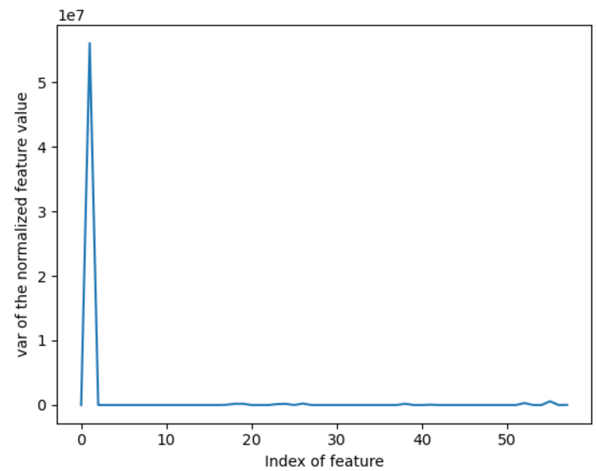


Figure 6: feature index VS Var of feature

Part(b): Finally we find that the Mean squared error is about 754.79, and there are 4 entries are 0 inside of the final parameter ω.You can see the code at the end of the pdf.

4 Problem 4

Part(a): We denote $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$, and so we notice that

$$k(\mathbf{u}, \mathbf{v}) = (\mathbf{u}^T\mathbf{v})^3 = \left(\sum_{i=1}^{d} u_i v_i\right)^3 = \sum_{k_1+k_2+\cdots+k_d=3} \frac{3!}{k_1!k_2!\cdots k_d!} \prod_{i=1}^{d} (u_i v_i)^{k_i}$$

Notice that the last term could be written as an inner product, where the vector of $\Phi(\mathbf{u}), \Phi(\mathbf{v})$ has dimension of $\binom{d+2}{d-1}$ by 1. Now we can find a way to order the element of $\Phi(\mathbf{u})$(same for $\Phi(\mathbf{v})$), that is we first consider all cases of $k_1 = 3, k_2 = 3, \ldots, k_d = 3$, and then all cases of $k_2 = 2, k_2 = 2, \ldots, k_d = 2$. Thus, we can find a order function $\phi(k_1, k_2, \ldots, k_d)$ ($\phi : \mathbb{R}^d \to \mathbb{R}^{\binom{d+2}{d-1}}$) mapping to the position of the inner product vector $\Phi(\mathbf{u})$where $\forall 1 \leq i \leq d$, $0 \leq k_i \leq 3$, and $\sum_{i=1}^{d} k_i = 3$. Thus, given the order function ϕ, we can $k(\mathbf{u}, \mathbf{v})$ as

$$k(\mathbf{u}, \mathbf{v}) = \left\langle \begin{bmatrix} \vdots \\ \sqrt{\frac{3!}{k_1!k_2!\cdots k_d!}} \prod_{i=1}^{d} u_i^{k_i} \\ \vdots \end{bmatrix}_{\phi}, \begin{bmatrix} \vdots \\ \sqrt{\frac{3!}{k_1!k_2!\cdots k_d!}} \prod_{i=1}^{d} v_i^{k_i} \\ \vdots \end{bmatrix}_{\phi} \right\rangle$$

This completes the proof.

Part(b): Suppose that k is an inner product kernel, then $\exists \Phi$, $\forall \mathbf{x_i}, \mathbf{x_j}, k(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_j, \mathbf{x}_i) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ by definition of inner product, so k is symmetric. Now notice that the matrix of k with dimension of $n \times n$ can be rewritten as

$$k = \begin{bmatrix} \Phi(\mathbf{x}_1)^T \\ \Phi(\mathbf{x}_2)^T \\ \vdots \\ \Phi(\mathbf{x}_n)^T \end{bmatrix} \cdot [\Phi(\mathbf{x}_1), \Phi(\mathbf{x_2}), \ldots, \Phi(\mathbf{x_n})]$$

Now denote

$$\varphi = \begin{bmatrix} \Phi(\mathbf{x}_1)^T \\ \Phi(\mathbf{x}_2)^T \\ \vdots \\ \Phi(\mathbf{x}_n)^T \end{bmatrix}$$

Then for any vector $z \in \mathbb{R}^{n\times 1}$, we must have

$$z^T \cdot k \cdot z = z^T \varphi \varphi^T z = \langle \varphi^T z, \varphi^T z \rangle \geq 0$$

Then by definition of semi-positive matrix, we conclude that k is a symmetric, positive definite kernel. This completes the proof.

Part(c):By lecture notes, we know that

$$\hat{b} = \bar{y} - \hat{w}^T \bar{\mathbf{x}}$$

where

$$\hat{w}^T = \tilde{y}^T (\tilde{G} + n\lambda I)^{-1} \tilde{\mathbf{X}}$$

$$\tilde{y} = \begin{bmatrix} y_1 - \bar{y} \\ y_2 - \bar{y} \\ \vdots \\ y_n - \bar{y} \end{bmatrix}$$

$$\tilde{\mathbf{X}} = \begin{bmatrix} \mathbf{x}_1 - \bar{\mathbf{x}} \\ \mathbf{x}_2 - \bar{\mathbf{x}} \\ \vdots \\ \mathbf{x}_d - \bar{\mathbf{x}} \end{bmatrix}$$

And $\hat{G}$ is the matrix that presented in the lecture slide, which is a SPD kernel. Thus, we can rewrite $\hat{b}$ as

$$\hat{b} = \bar{y} - \hat{y}^T (\hat{G} + n\lambda I)^{-1} \tilde{\mathbf{X}} \bar{\mathbf{x}}$$

Now we can write $\tilde{\mathbf{X}}\bar{\mathbf{x}}$ as $\tilde{g}(\tilde{\mathbf{x}})$ where

$$\tilde{g}(\tilde{\mathbf{x}}) = \tilde{\mathbf{X}}\bar{\mathbf{x}} = \begin{bmatrix} \langle \tilde{\mathbf{x}}_1, \bar{\mathbf{x}} \rangle \\ \vdots \\ \langle \tilde{\mathbf{x}}_1, \bar{\mathbf{x}} \rangle \end{bmatrix}$$

And we have that

$$\langle \tilde{\mathbf{x}}_i, \bar{\mathbf{x}} \rangle = \frac{1}{n} \sum_{j=1}^{n} \langle \mathbf{x}_i, \mathbf{x}_j \rangle - \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \langle \mathbf{x}_i, \mathbf{x}_j \rangle$$

Thus, we have shown that the offset b can also be evaluated using the kernel.

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Problem 2

In [3]:
```
### Import the data
X_1 = np.load('pulsar_features.npy')
y_1 = np.load('pulsar_labels.npy')

### Notice we first need to transpose the X and y
X = X_1.T
y = y_1.T
```

In [4]:
```
# Add one's to the X so that we can estimate b.
X  = np.concatenate((np.ones(X.shape[0]).reshape(-1,1), X), axis = 1)
```

Part(b)

In [5]:
```
theta = np.zeros(3)
```

In [6]:
```
### Define the loss function
def loss_function(X, y, theta, lamb):
    loss = 0
    w = theta.copy()[1:]

    for i in range(X.shape[0]):
        loss += (np.max([0, 1 - (y[i] * (np.dot(theta, X[i]))).item(0)]) + lamb

    loss = loss / X.shape[0]

    return loss
```

In [7]:
```
loss_function(X, y, theta, 0.001)
```

Out[7]:
```
1.0
```

In [8]:
```
### Define the subgradient function
def sub_gradient(X, y, theta, lamb):
    subgradient = 0
    w = theta.copy()
    w[0] = 0

    for i in range(X.shape[0]):
        if y[i] * np.dot(theta, X[i]) < 1:
            subgradient += (1 / X.shape[0]) * (-y[i] * X[i] + lamb * w)
        if y[i] * np.dot(theta, X[i]) >= 1:
            subgradient += (1 / X.shape[0]) * lamb * w

    return subgradient
```

In [9]:
```
sub_gradient(X, y, theta, 0.02)
```

Out[9]: array([4.22838847e-18, 1.80522359e-01, 8.74244260e-02])

In [10]:
```
### Start the subgradient method
def sub_grad_method(X, y, theta, lamb, iteration):
    loss_list = []

    for i in range(iteration):
        step_size = 100 / (i + 1)
        theta = theta - step_size * sub_gradient(X, y, theta, lamb)
        loss_list.append(loss_function(X, y, theta, lamb))

    theta_final = theta.copy()

    return theta_final, loss_list
```

In [11]:
```
result_subgrad = sub_grad_method(X, y, theta, 0.001, 10)
```

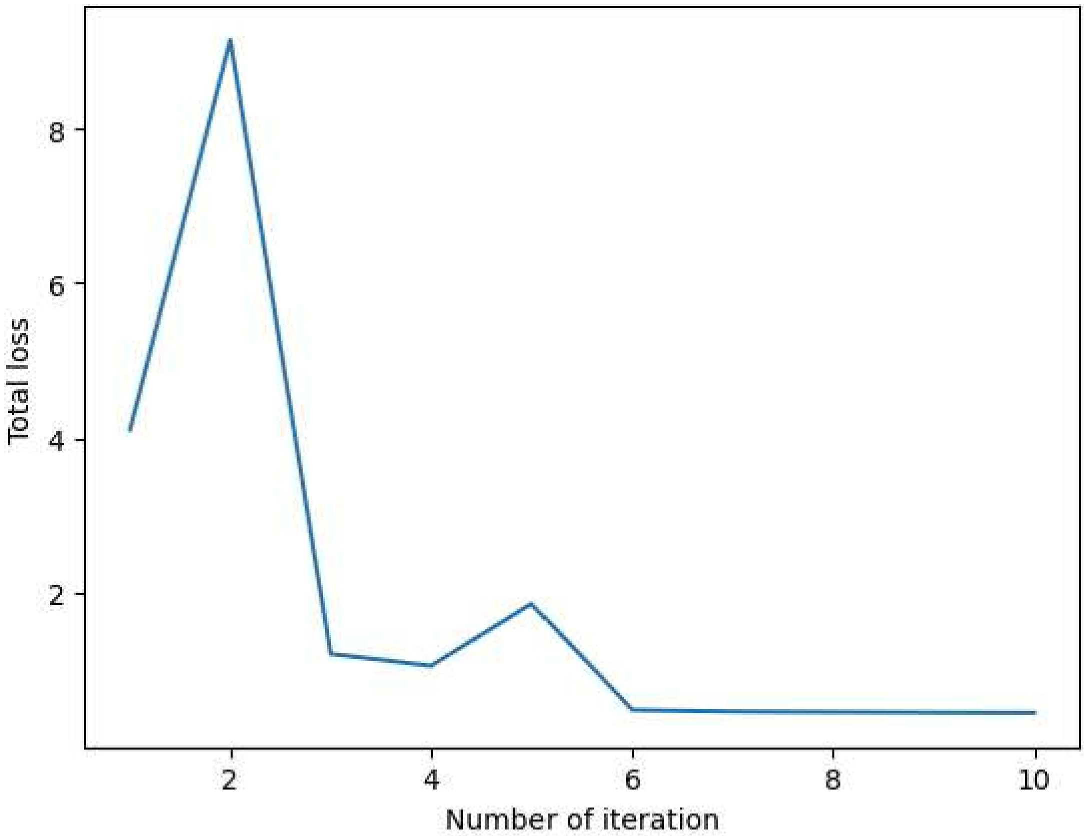
In [12]:
```
loss_list = result_subgrad[1]
parameters = result_subgrad[0]
```

In [13]:
```
### Get the paramters of the hyperplane
parameters
```

Out[13]: array([12.0680196 , -17.81627138, -9.11707611])

In [14]:
```
## Draw the picture of loss functions versus number of iteration
plt.plot(list(range(1, 11)), loss_list)
plt.xlabel('Number of iteration')
plt.ylabel('Total loss')
```

Out[14]: Text(0, 0.5, 'Total loss')



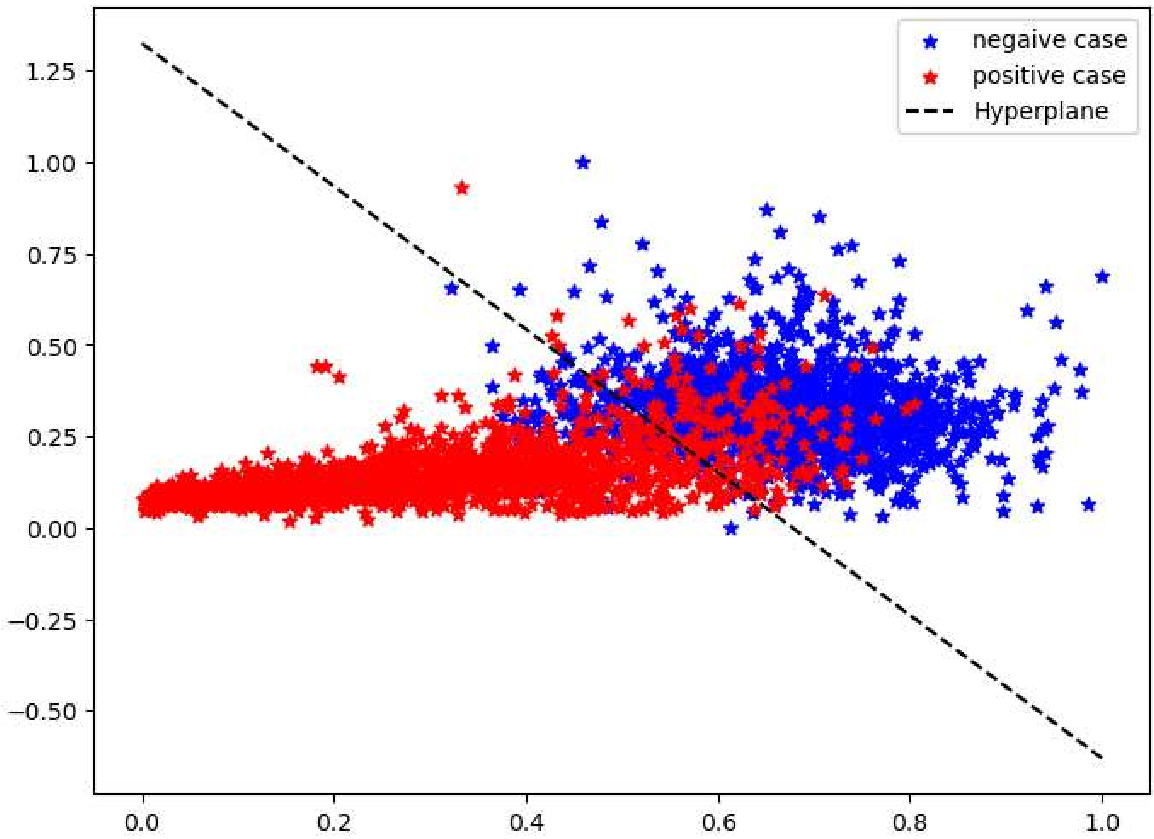
In [15]:
```
## put all columns into a dataframe so that we can visulize them
data_p1 = pd.DataFrame({
    'X_1': X_1.T[:, 0].tolist(),
    'X_2': X_1.T[:, 1].tolist(),
    'y'  : y_1.T.ravel().tolist()
})
```

In [16]:
```
### Get subdata for y = -1 and y = 1
subdata_p1_neg = data_p1[data_p1['y'] == -1]
subdata_p1_pos = data_p1[data_p1['y'] == 1]
```

In [17]:
```
### define the line of the hyperplane with derived parameters
X1 = np.linspace(0, 1, 200)
X2 = (parameters[0] + parameters[1] * X1)/(-parameters[2])
```

In [18]:
```
### Start visulization
### Negative case meaning label of y = -1. Positive case meaning label of y = 1
plt.figure(figsize = (8, 6))
plt.scatter(x = 'X_1', y = 'X_2', color = 'blue', marker = '*', data = subdata_p
plt.scatter(x = 'X_1', y = 'X_2', color = 'red', marker = '*', data = subdata_p1
### Add the line of hyperplane
plt.plot(X1, X2, linestyle = '--', color = 'black', label = 'Hyperplane')
plt.legend()
```

Out[18]: <matplotlib.legend.Legend at 0x1721594b050>



In [19]:
```
## Get the minimum achived value of the objective function
print('The mininum achived value of the objective function is:', loss_list[-1])
```

The mininum achived value of the objective function is: 0.44988413706113156

In [20]:
```
## Get the margin
print('The margin is:', 1/np.linalg.norm(parameters[1:]))
```

```
The margin is: 0.049966246537370425
```

Part(c)

In [21]:
```
### Define the function of subgradient for a single point
def single_subgradient(X, y, theta, lamb, row_dimension):
    w = theta.copy()
    w[0] = 0
    if y * (np.dot(theta, X)) < 1:
        return 1/row_dimension * (-y * X + lamb * w)

    if y * (np.dot(theta, X)) >= 1:
        return lamb/row_dimension * w
```

In [22]:
```
### Define the SGD function
def stochastic_grad_method(X, y, theta, lamb, iteration):
    np.random.seed(0)
    loss = []
    for j in range(iteration):
        step_size = 100 / (j + 1)
        for i in np.random.permutation(X.shape[0]):
            theta = theta - step_size * single_subgradient(X[i], y[i], theta, la

        theta_outer = theta.copy()
        loss.append(loss_function(X, y, theta_outer, lamb))

    return theta_outer, loss
```

In [23]:
```
sgd_result = stochastic_grad_method(X, y, theta, 0.001, 10)
```

In [24]:
```
### Extract the loss and the final parameter
sgd_loss = sgd_result[1]
sgd_parameter = sgd_result[0]
```

In [25]:
```
print("The minimum achieved value of the objective function is:", sgd_loss[-1])
```

```
The minimum achieved value of the objective function is: 0.2582782419707577
```

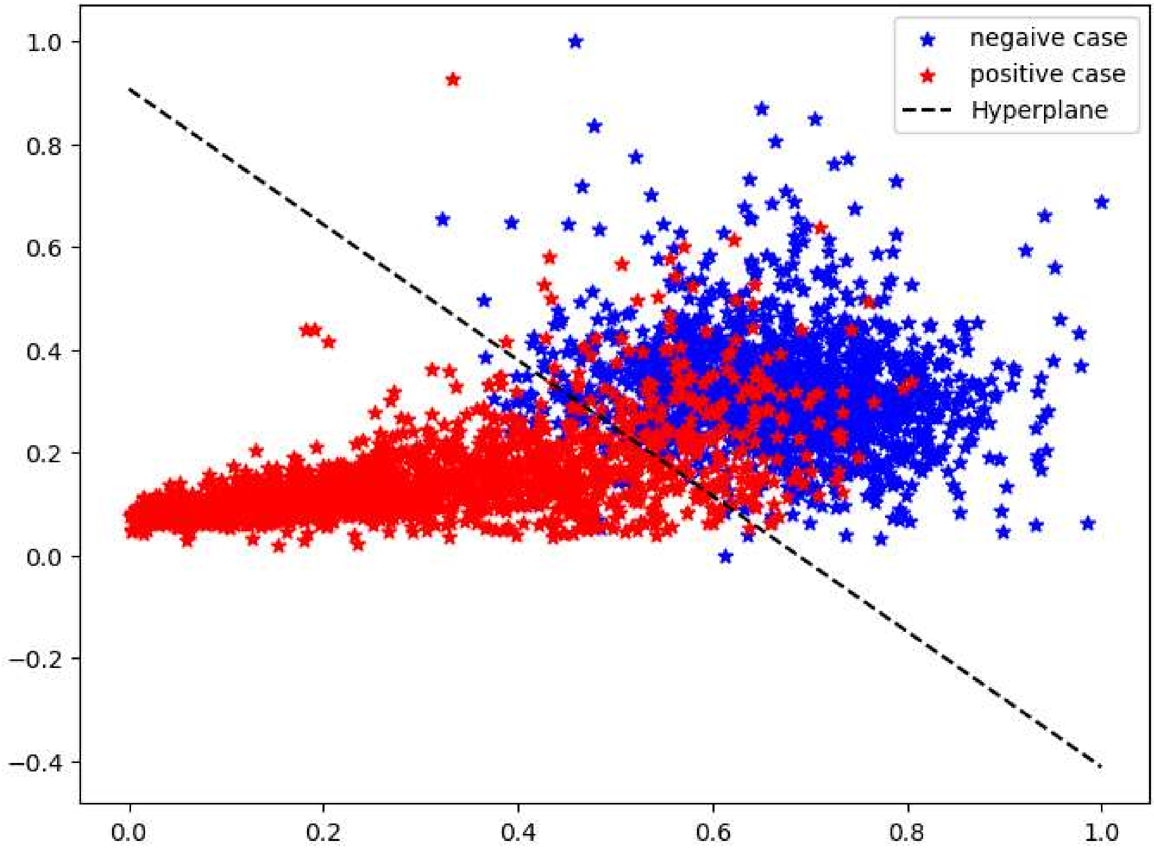
In [26]:
```
### show the parameters of SGD method
sgd_parameter
```

Out[26]:
```
array([ 4.00515219, -5.82463117, -4.41417027])
```

In [27]:
```
### define the line of the hyperplane with derived parameters
X1_sgd = np.linspace(0, 1, 200)
X2_sgd = (sgd_parameter[0] + sgd_parameter[1] * X1_sgd)/(-sgd_parameter[2])
```

In [28]:
```
### Start visulization
### Negative case meaning label of y = -1. Positive case meaning label of y = 1
plt.figure(figsize = (8, 6))
plt.scatter(x = 'X_1', y = 'X_2', color = 'blue', marker = '*', data = subdata_p
plt.scatter(x = 'X_1', y = 'X_2', color = 'red', marker = '*', data = subdata_p1
### Add the line of hyperplane
plt.plot(X1_sgd, X2_sgd, linestyle = '--', color = 'black', label = 'Hyperplane'
plt.legend()
```

Out[28]:
```
<matplotlib.legend.Legend at 0x17215a47a10>
```

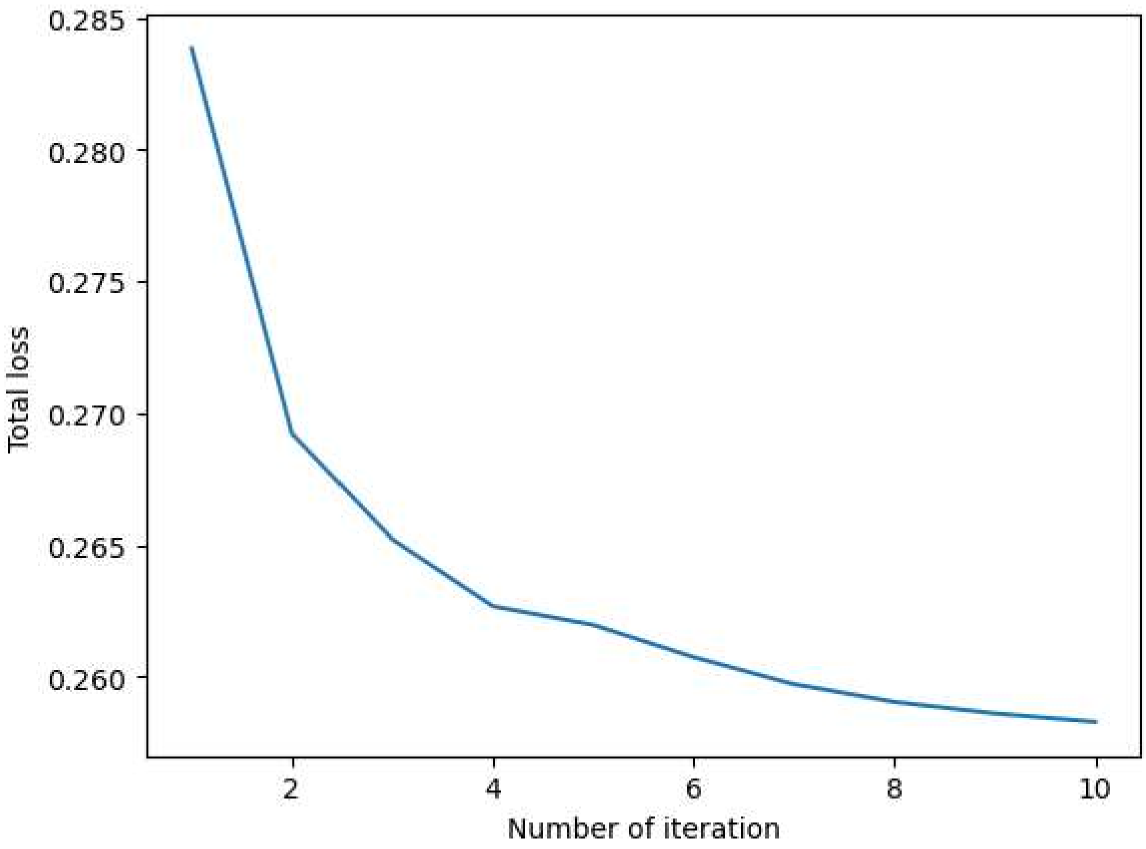


In [29]:
```
## Get the margin
print('The margin is:', 1/np.linalg.norm(sgd_parameter[1:]))
```

```
The margin is: 0.13683075407918574
```

In [30]:
```
plt.plot(list(range(1, 11)), sgd_loss)
plt.xlabel('Number of iteration')
plt.ylabel('Total loss')
```

Out[30]:
```
Text(0, 0.5, 'Total loss')
```



It seems that the SGD method converges much faster than the subgradient method.

Problem 3

Part(a)

In [31]:
```
X_train = np.load('housing_train_features.npy')
y_train = np.load('housing_train_labels.npy')

X_test = np.load('housing_test_features.npy')
y_test = np.load('housing_test_labels.npy')

X_train = X_train.T
X_test = X_test.T
```

In [32]:
```
np.sum(X_test[:, 4]) == 0
```

Out[32]: True

In [33]:
```
## Normalization, first find mean and standard deviation for train data
train_covariates_mean = np.mean(X_train, axis = 0)
train_deviation_std = np.std(X_train, axis = 0)

normalized_train = (X_train - train_covariates_mean) / train_deviation_std
```

In [34]:
```
## Normalization, first find mean and standard deviation for test data
test_covariates_mean = np.mean(X_test, axis = 0)
test_deviation_std = np.std(X_test, axis = 0)
```

```
normalized_test = (X_test - test_covariates_mean) / test_deviation_std
```

```
C:\Users\16343\AppData\Local\Temp\ipykernel_71648\2538276479.py:5: RuntimeWarnin
g: invalid value encountered in divide
  normalized_test = (X_test - test_covariates_mean) / test_deviation_std
```

In [35]:
```
normalized_test = np.nan_to_num(normalized_test, nan = 0)
```

In [36]:
```
## Verify the mean and variance of normalized train data
mean_normalized = np.mean(normalized_train, axis = 0)
var_normalized = np.var(normalized_train, axis = 0)
```

In [37]:
```
mean_normalized[:5]
```

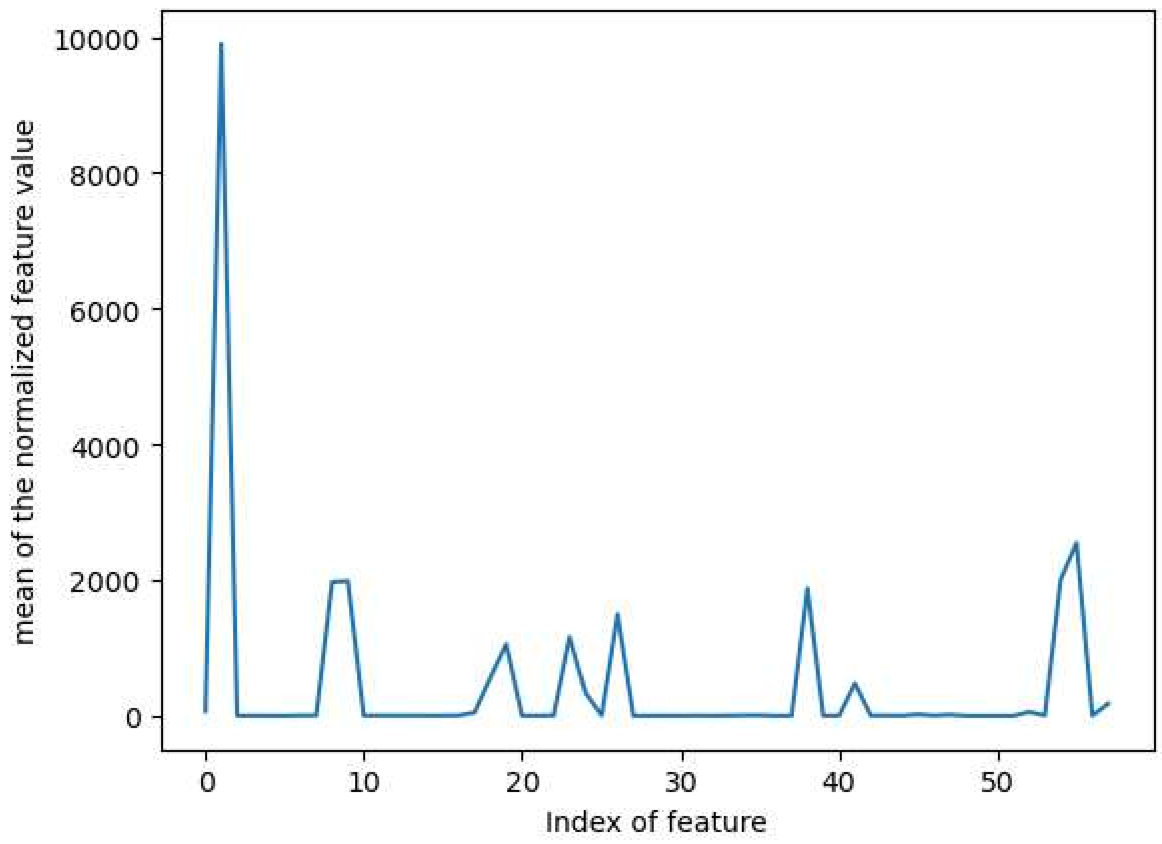
Out[37]:
```
array([-2.06945572e-16, -4.97379915e-17, -7.88702437e-16,  7.46069873e-17,
        9.76996262e-18])
```

In [38]:
```
var_normalized[:5]
```

Out[38]:
```
array([1., 1., 1., 1., 1.])
```

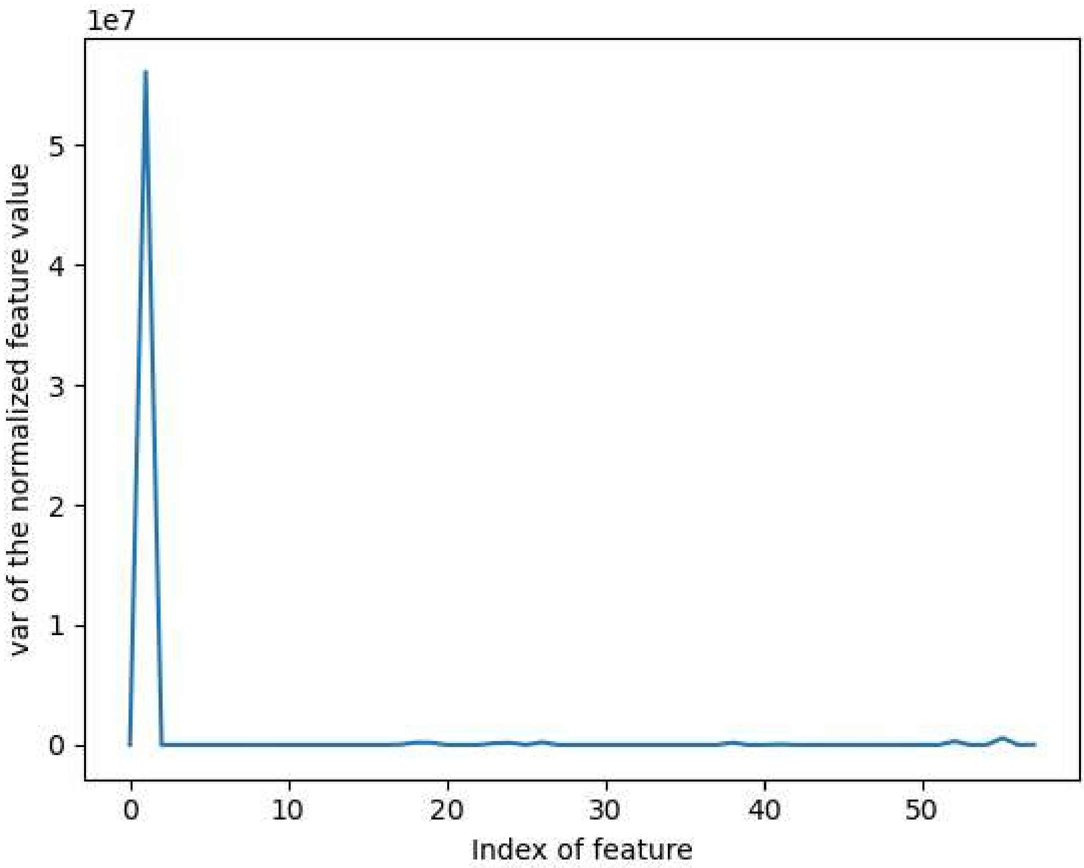
In [39]:
```
### Get the plot of mean and variance
plt.plot(np.arange(np.shape(mean_normalized)[0]), np.mean(X_train, axis = 0))
plt.xlabel('Index of feature')
plt.ylabel('mean of the normalized feature value')
```

Out[39]:
```
Text(0, 0.5, 'mean of the normalized feature value')
```



In [40]:
```
plt.plot(np.arange(np.shape(mean_normalized)[0]), np.var(X_train, axis = 0))
plt.xlabel('Index of feature')
plt.ylabel('var of the normalized feature value')
```

Out[40]: Text(0, 0.5, 'var of the normalized feature value')



Part(b)

In [444…

```
### define the dimension of the normalized training data
n, d = normalized_train.shape
```

In [447…

```
## define the function of computing alpha
def computing_alpha(X, column_number):
    alpha = 2 * (np.dot(X[:, column_number], X[:, column_number]))/n
    return alpha
```

In [463…

```
## define the function of computing c at each iteration
def computing_c(X, y, column_number, w, b):
    sum = 0
    w_inner = w.copy()
    w_inner[column_number] = 0

    for i in range(n):
        sum += X[i][column_number] * (y[i] - np.dot(w_inner, X[i]) - b)

    sum = 2 * sum / n

    return sum
```

In [464…

```
## define the soft function
def soft_function(a, b):
    if a > b:
        return a - b
    if a < -b:
        return a + b
```

```
    else:
        return 0
```

In [465…

```
### initialize w and b
w = np.ones(d)
b = np.ones(1)
lamb = 100 / n
iteration = 2900
```

In [466…

```
## Start the Coordinate Descent algorithm
def CD_method(X, y, w, b, lamb, iteration):
    for i in range(iteration):
        b = np.mean(y) - np.dot(w, np.mean(X, axis = 0))
        for j in range(d):
            ## Compute c and a inside of the loop
            c = computing_c(X, y, j, w, b)
            a = computing_alpha(X, j)
            ## computing soft value and update w_j
            soft_value = soft_function(c / a, lamb / a)
            w[j] = soft_value

    return w, b
```

In [467…

```
result = CD_method(normalized_train, y_train, w, np.mean(y_train), lamb, iterati
```

In [468…

```
final_w = result[0]
final_b = result[1]
```

In [469…

```
result
```

Out[469…

```
(array([ 2.99575334e+00,  4.79116557e+00,  2.45387945e+00,  2.94461786e-01,
        -4.09224509e-01, -5.63272845e-02,  1.32144814e+01,  6.29681834e+00,
         8.33256016e+00,  1.66730553e+00, -1.03933462e+01,  1.50871954e+00,
        -6.58841800e+00, -6.19700809e-01, -1.66831097e+00,  1.34224205e+00,
        -8.12517801e-01, -2.81963635e+00, -1.02007601e+01,  1.15785740e+01,
        -1.31149714e+00,  5.58964888e-01, -9.85282822e-01,  0.00000000e+00,
         2.06308197e+00, -1.09270225e-01,  1.02309765e+01,  8.80246342e-01,
        -6.34243842e-01, -1.70020031e+00,  3.14609340e-01, -4.22716224e+00,
        -5.54398388e+00, -3.12550025e+00,  5.12569530e+00,  2.94668204e+00,
         6.39078873e+00, -5.23618774e+00, -1.22609006e+00, -9.68079116e-01,
         1.34086435e+00,  7.09206669e+00, -3.05421368e+00,  2.38817421e+00,
         0.00000000e+00,  1.95892192e-02, -6.44737471e-01,  1.58019326e+00,
         1.23972670e+00,  0.00000000e+00,  4.95102062e-01,  2.05324605e-02,
        -5.30400179e-02, -4.05013091e-01, -1.01171167e+00,  2.06917139e+01,
         0.00000000e+00,  9.74767678e-01]),
 181.678874)
```

In [470…

```
### count how many zeros in the final w
zero_count = final_w.shape[0] - np.count_nonzero(final_w)
print("The number of zeros in w is:", zero_count)
```

```
The number of zeros in w is: 4
```

In [471…

```
### find the MSE of this model using test set
def calculate_MSE(X, y, w, b):
    MSE = 0
    for i in range(X.shape[0]):
        MSE += (np.dot(w, X[i]) + b - y[i]) ** 2
```

```
    MSE = MSE / X.shape[0]
    return MSE
```

In [472…

```
calculate_MSE(normalized_test, y_test, final_w, final_b)
```

Out[472…

```
754.7914449031186
```