1. (a) $J(w) = 0.5 \|y - Xw\|_2^2$

$J(w) = 0.5 \times (y - Xw)^2$

$= 0.5 \times (X^T X w^2 - 2X^T w y + y^2)$

$\nabla J(w) = X^T X w - X^T y$

$\because X^T X = H$ (Hessian matrix)

$\therefore \nabla J(w) = Hw - X^T y$ ※

(b) $H \succeq 0 \Rightarrow H$ is positive definite

$\therefore H^{-1}$ exist

To prove w^* is the local minimum

$\Rightarrow \nabla J(w^*) = Hw^* - X^T y = 0$

$\Rightarrow Hw^* = X^T y$

$\times H^{-1} \Rightarrow w^* = H^{-1} X^T y$

Recall $H = X^T X$

$\Rightarrow w^* = (X^T X)^{-1} X^T y$ ※

(c) $e_{t+1} = w_{t+1} - w^*$

$e_{t+1} = (w_t - \eta \nabla J(w_t)) - w^*$

$= w_t - \eta H w - \eta X^T y - w^*$

From 1.(b) $Hw^* = X^T y$

$= \underbrace{w_t - w^*}_{\| e_t} - \eta H w - \underbrace{\eta X^T y}_{\| \eta H w^*}$

$= e_t - \eta (Hw - Hw^*)$

$= e_t - \eta H e_t$ ※

(d) Since $e_{t+1} = e_t - \eta H e_t$

$\| e_{t+1} \| = \| e_t - \eta H e_t \|$

From triangular inequality $\| e_{t+1} \| \le \| e_t \| + \| -\eta H e_t \|$

$\Rightarrow \| e_{t+1} \| \le \| e_t \| \times \| I - \eta H \|$

$\le \| e_t \| \times \text{rate}_\eta$

For $t=0 \Rightarrow \| e_1 \| \le \| e_0 \| \times \text{rate}_\eta$

$t=1 \Rightarrow \| e_2 \| \le \| e_0 \| \times \text{rate}_\eta^2$

$\Rightarrow \| e_t \| \le \text{rate}_\eta^t \times \| e_0 \|_{e_2}$ ※

1-(e) $\|e_t\|_{l_2} = \|I - \eta H\|_{l_2}^t \|e_0\|_{l_2}$

Let λ is the eigenvalue of H

$\text{rate}_\eta = \|I - \eta H\|$ has the eigenvalue

$(1 - \eta\lambda_{min})$

$(1 - \eta\lambda_{max})$

$\Rightarrow \text{rate}_\eta = \|I - \eta H\|_{l_2} = \max(1 - \eta\lambda_{min}, \eta\lambda_{max} - 1)$ ※

(f) Case 1:

If $\text{rate}_\eta = 1 - \eta\lambda_{min}$

To minimize $\text{rate}_\eta \Rightarrow \eta^*$ should be $\frac{1}{\lambda_{min}}$

the corresponding convergence $\text{rate}_{\eta^*} = 0$

Case 2:

$\text{rate}_\eta = \eta\lambda_{max} - 1$

$\Rightarrow \eta^*$ should be $\frac{1}{\lambda_{max}}$

the corresponding convergence $\text{rate}_{\eta^*} = 0$ ※

2. (b)

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n}(w^T x_i + b - y_i)^2 + \lambda\|w\|_1$$

$$\partial_{w_j} J(w,b) = a_j w_j - c_j + \lambda \partial_{w_j}|w_j|$$

1st. $c_j > \lambda$, $w_j = soft\left(\frac{c_j}{a_j}; \frac{\lambda}{a_j}\right) = \frac{c_j - \lambda}{a_j}$

$$\partial_{w_j} J(w,b) = a_j\left(\frac{c_j-\lambda}{a_j}\right) - c_j + \lambda$$

$$= 0 \quad ※$$

2nd. $-\lambda \le c_j \le \lambda$, $w_j = 0$

$$\partial_{w_j} J(w,b) = -c_j + \lambda \cdot [-1,1] \quad \text{where} \quad c_j = [-\lambda, \lambda]$$

$$= 0 \quad ※$$

3rd. $c_j < -\lambda$, $w_j = soft\left(\frac{c_j}{a_j}; \frac{\lambda}{a_j}\right) = \frac{c_j + \lambda}{a_j}$

$$\partial_{w_j} J(w,b) = a_j\left(\frac{c_j+\lambda}{a_j}\right) - c_j + \lambda \times (-1)$$

$$= 0 \quad ※$$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)

Xtrain = np.load("housing_train_features.npy")
Xtest = np.load("housing_test_features.npy")
ytrain = np.load("housing_train_labels.npy")
ytest = np.load("housing_test_labels.npy")

feature_names = np.load("housing_feature_names.npy", allow_pickle=True)
print("First feature name: ", feature_names[0])
print("Lot frontage for first train sample:", Xtrain[0,0])
```

```
First feature name:  Lot.Frontage
Lot frontage for first train sample: 141.0
```

In [2]:
```
def normalize(x):
    u = np.mean(x, axis = 1).reshape(-1, 1)
    #u = np.tile(u, (1, x.shape[1]))

    var = np.var(x, axis = 1).reshape(-1, 1)
    norm_x = (x - u)/ var**0.5

    return norm_x, u, var
```

In [3]:
```
Xtrain_norm, u, var = normalize(Xtrain)
# since the numerical issues, the values are very close to 0
rounded_mean_values = np.round(np.mean(Xtrain_norm, axis = 1), decimals=10)
print(np.mean(Xtrain_norm, axis = 1))
print(np.var(Xtrain_norm, axis = 1))
```

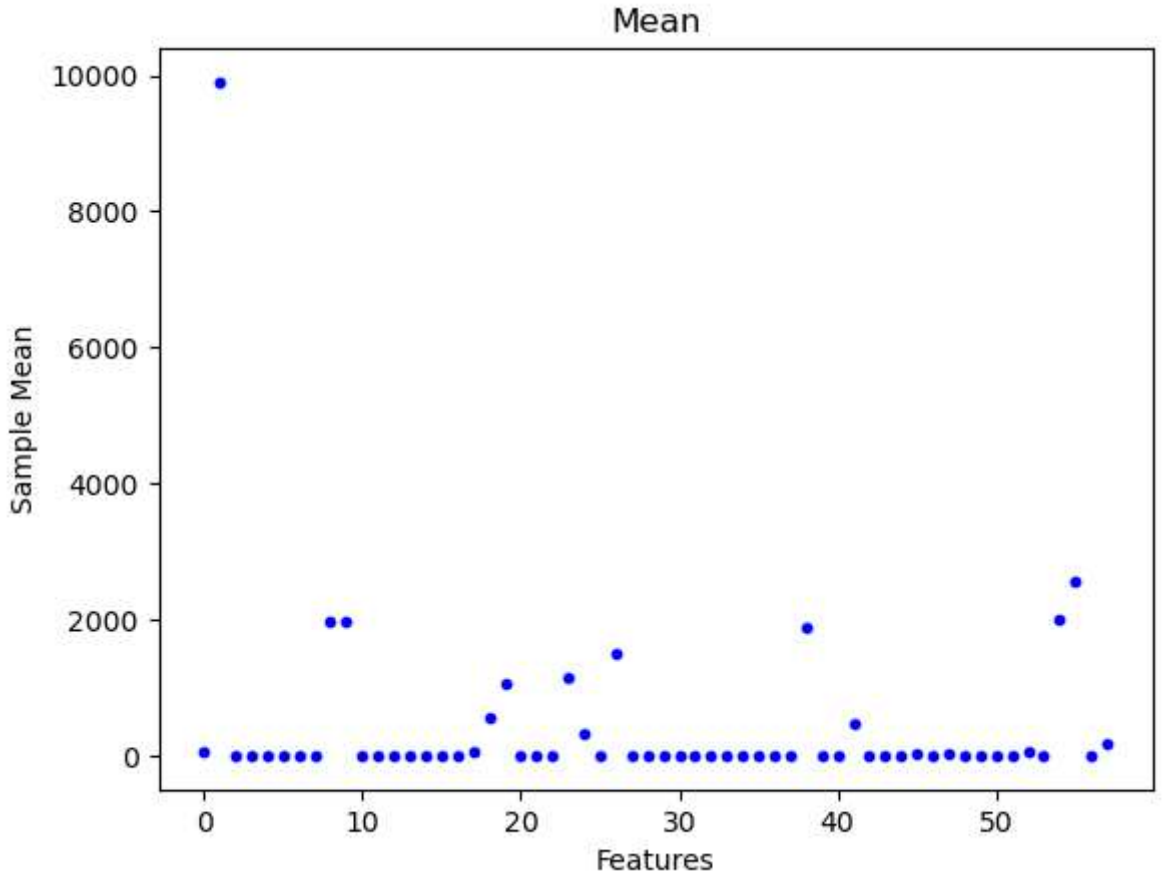
```
[-2.06945572e-16 -4.97379915e-17 -7.88702437e-16  7.46069873e-17
  9.76996262e-18  3.55271368e-18 -1.17239551e-16  2.80664381e-16
 -1.08002496e-15 -4.41602310e-15 -2.27373675e-16  1.35003120e-16
  4.97379915e-17  1.50990331e-17 -5.50670620e-17 -1.06581410e-17
  1.20792265e-16 -4.61852778e-17 -7.81597009e-17 -1.58095759e-16
  1.06581410e-16 -3.55271368e-18  1.03028697e-16 -1.27897692e-16
 -1.42108547e-17 -1.06581410e-17  3.19744231e-17  6.03961325e-17
  3.90798505e-17 -2.06057393e-16 -1.66977543e-16  2.04281037e-17
 -4.51194637e-16  8.17124146e-17  1.84741111e-16  1.63424829e-16
  6.92779167e-17  3.55271368e-18 -1.59872116e-16  2.66453526e-17
  3.55271368e-17 -6.03961325e-17  8.34887715e-17 -5.50670620e-17
 -1.31450406e-16  1.06581410e-17  1.59872116e-17 -6.21724894e-18
  0.00000000e+00 -3.90798505e-17  1.24344979e-17  3.73034936e-17
 -2.66453526e-17  5.68434189e-17 -2.75690581e-14 -7.46069873e-17
 -5.32907052e-17 -1.77635684e-18]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

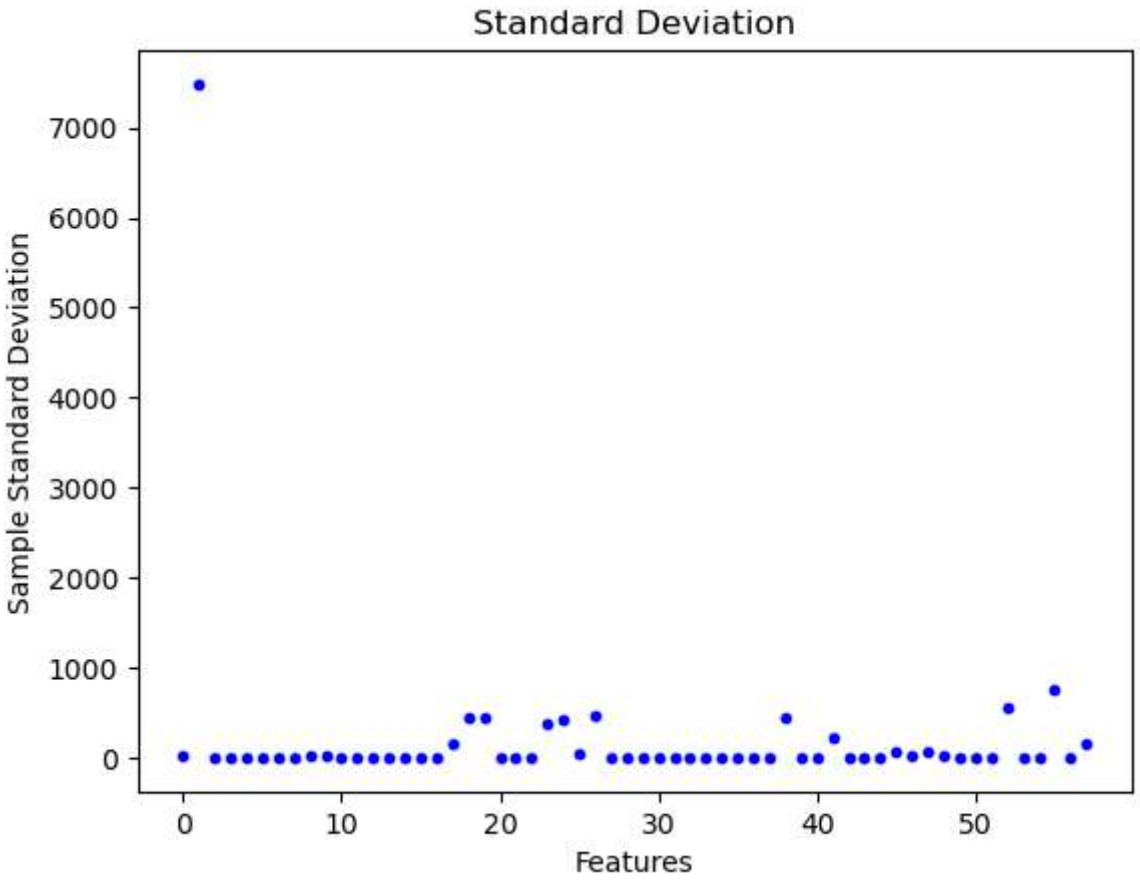
In [4]:
```
print(Xtrain.shape)
```

```
(58, 2000)
```

In [5]:
```
plt.figure(1)
plt.scatter(range(Xtrain.shape[0]), u[:, 0], marker='o', linestyle='-', color=
plt.xlabel("Features")
plt.ylabel("Sample Mean")
plt.title("Mean")

plt.figure(2)
plt.scatter(range(Xtrain.shape[0]), var[:, 0]**(0.5), marker='o', linestyle='-
plt.xlabel("Features")
plt.ylabel("Sample Standard Deviation")
plt.title("Standard Deviation")
```

Out[5]:
```
Text(0.5, 1.0, 'Standard Deviation')
```




Standard Deviation
Sample Standard Deviation
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
Features

In [6]:
```
def soft_threshold(c, lam, a):
    if c < -lam:
        return (c + lam) / a
    elif c > lam:
        return (c - lam) / a
    else:
        return 0
def CD_Lasso(x, y, lam = 100 / 2000):
    iterations = int(2900 / x.shape[0])
    w = np.ones((x.shape[0], 1)) # d * 1 = 58 * 1

    for i in range(iterations):
     # x is d * n
        for j in range(x.shape[0]):
            x0 = x[j].reshape(-1, 1) # n * 1

            x1 = np.delete(x, j, 0) # d-1 * n
            w1 = np.delete(w, j).reshape(-1, 1) # d-1 * 1
            y_pred = (np.transpose(x1).dot(w1)) # n * 1
            #y_pred = np.dot(np.transpose(x1) - np.mean(y), w1)
            #print(y_pred.shape)
            y = y.reshape(-1, 1)
            c = (2 * np.dot(x0.T, (y - y_pred - np.mean(y)))) / x.shape[1]
            #c = (2 * np.dot(x0.T, (y - y_pred))) / x.shape[1]
            #print(c)
            a = (2 * np.sum(np.power(x0, 2))) / x.shape[1]

            w[j, :] = soft_threshold(c, lam, a)
    return w
```

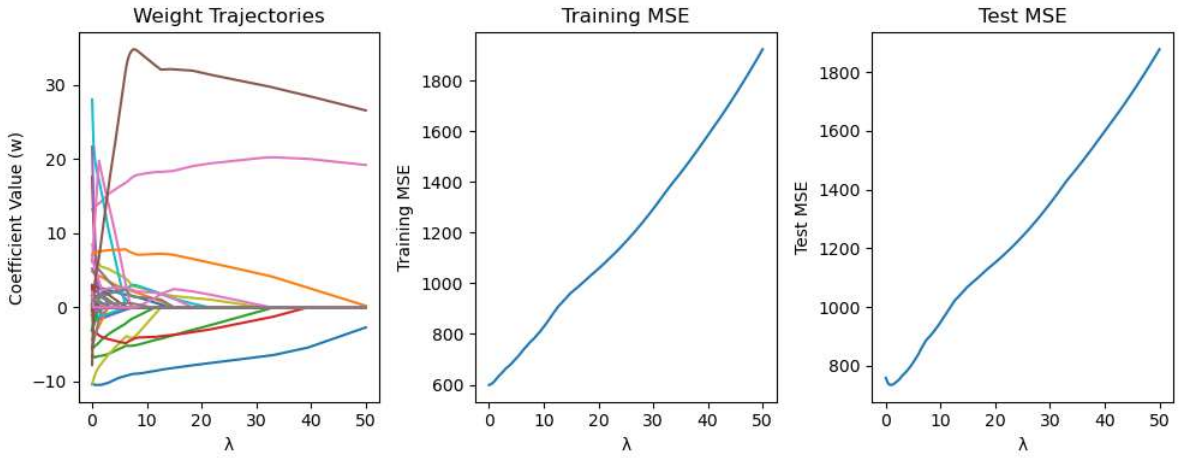
In [7]:
```
w_optimal = CD_Lasso(Xtrain_norm, ytrain)
```

In [8]:
```
# Compute test MSE
# normalize Xtest by the mean and std calculate on Xtrain
Xtest_norm = (Xtest - u) / var**0.5
#print(Xtest_norm.shape)
y_pred = np.transpose(Xtest_norm).dot(w_optimal) + np.mean(ytrain)
y_pred = y_pred.reshape(-1)
test_mse = np.mean(np.power(ytest - y_pred, 2))
print("Test MSE:", test_mse)
```

```
Test MSE: 755.0824383673374
```

In [9]:
```
num_samples = 200
lambda_values = np.linspace(0, 1e5 / Xtrain.shape[1], num_samples)

trajectory_w = np.zeros((num_samples, Xtrain.shape[0]))
train_mse_all = np.zeros(num_samples)
test_mse_all = np.zeros(num_samples)

for i, lambda_ in enumerate(lambda_values):

    w = CD_Lasso(Xtrain_norm, ytrain, lambda_)
    w = w.reshape(-1)
    # Store the coefficients (ignoring the offset b)
    #print(w)
    trajectory_w[i, :] = w

    y_pred1 = np.transpose(Xtrain_norm).dot(trajectory_w[i, :])
    train_mse_all[i] = np.mean((ytrain - y_pred1- np.mean(ytrain))**2)

    y_pred2 = np.transpose(Xtest_norm).dot(trajectory_w[i, :])
    test_mse_all[i] = np.mean((ytest - y_pred2 - np.mean(ytrain))**2)




#print(train_mse_all)
plt.figure(figsize=(10, 4))

# Weight trajectories plot
plt.subplot(1, 3, 1)
for i in range(Xtrain.shape[0]):
    plt.plot(lambda_values, trajectory_w[:, i], label=f'w_{i+1}')
plt.xlabel('λ')
plt.ylabel('Coefficient Value (w)')
#plt.legend()
plt.title('Weight Trajectories')

# Training MSE plot
plt.subplot(1, 3, 2)
plt.plot(lambda_values, train_mse_all)
plt.xlabel('λ')
plt.ylabel('Training MSE')
plt.title('Training MSE')

# Test MSE plot
plt.subplot(1, 3, 3)
plt.plot(lambda_values, test_mse_all)
plt.xlabel('λ')
plt.ylabel('Test MSE')
plt.title('Test MSE')

plt.tight_layout()
plt.show()

# Find the λ that minimizes training error and test error
best_lambda_train = lambda_values[np.argmin(train_mse_all)]
best_lambda_test = lambda_values[np.argmin(test_mse_all)]
```

```
print(f"λ that minimizes training error: {best_lambda_train}")
print(f"λ that minimizes test error: {best_lambda_test}")
```




```
λ that minimizes training error: 0.0
λ that minimizes test error: 1.0050251256281406
```

In [ ]:

3. $$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + \frac{C}{n}\left[(1-\alpha)\sum_{i:y_i=1}^{n}\xi_i + \alpha\sum_{i:y_i=-1}^{n}\xi_i\right]$$

$$\text{s.t. } y_i(w^T x_i + b) \geq 1-\xi_i, \quad \forall i$$

$$\xi_i \geq 0, \quad \forall i$$

It can be written as

$$\Rightarrow \min_{w,b,\xi} \frac{1}{2}\|w\|^2 + \frac{C}{n}\left[(1-\alpha)\sum_{i:y_i=1}^{n}\max(0, 1-(w^T x_i+b)) + \alpha\sum_{i:y_i=-1}^{n}\max(0, 1+(w^T x_i+b))\right]$$

$$\Rightarrow \min_{w,b,\xi} \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i=1}^{n}\left[(1-\alpha)\left(\frac{1+y_i}{2}\right)\times\max(0, 1-(w^T x_i+b)) - \alpha\left(\frac{-1+y_i}{2}\right)\times\max(0, 1+(w^T x_i+b))\right]$$

If $\lambda = \frac{1}{C}$

$$\Rightarrow \min_{w,b,\xi} \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\left[(1-\alpha)\left(\frac{1+y_i}{2}\right)\times\max(0, 1-(w^T x_i+b)) - \alpha\left(\frac{-1+y_i}{2}\right)\times\max(0, 1+(w^T x_i+b))\right]$$

$\lambda = \frac{1}{C}$ (regularization parameter) → Regularization term

This is the loss function which is similar to hinge loss ※

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

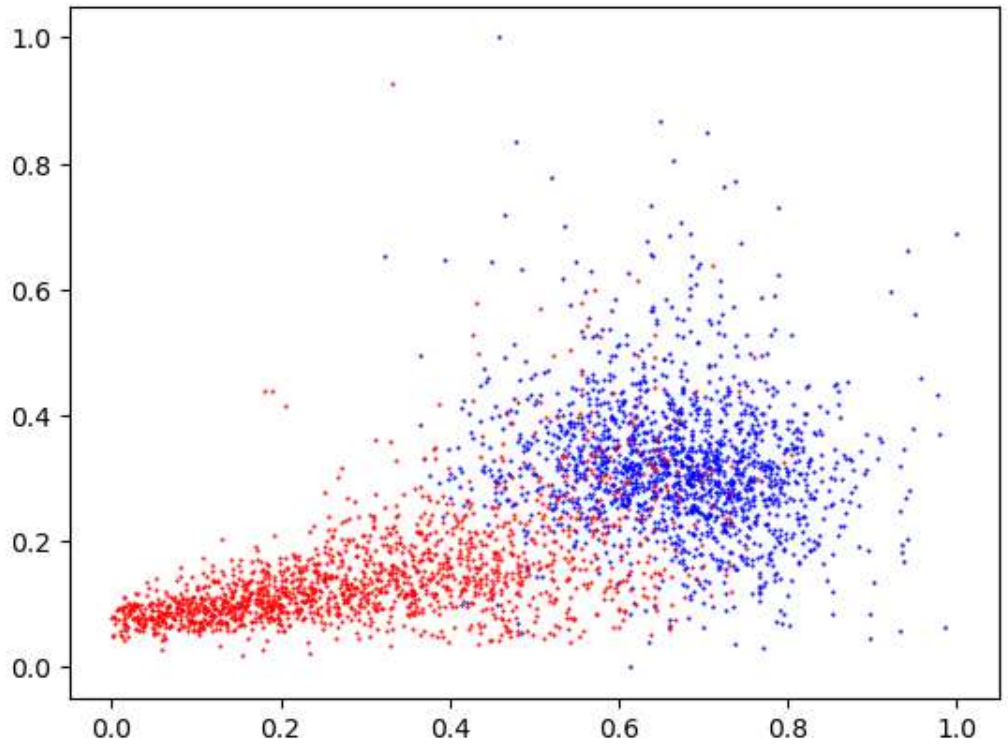
$$J_i(w,b) = \frac{1}{n}\left(L(y_i, w^T x_i + b) + \frac{\lambda}{2}\|w\|^2\right)$$

$$\nabla_i = \begin{bmatrix} \frac{\partial J}{\partial b} \\ \frac{\partial J}{\partial w} \end{bmatrix} = \begin{bmatrix} -y_i \cdot 1 \quad (\text{if } 1 - y_i(x_i w^T + b) > 0) \\ \lambda w - y_i \cdot x_i \quad (\text{if } 1 - y_i(x_i w^T + b) > 0) \end{bmatrix}$$

else

$$= \begin{bmatrix} 0 \\ \lambda w \end{bmatrix}$$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)
x = np.load("pulsar_features.npy")
y = np.load("pulsar_labels.npy")

negInd = y == -1
posInd = y == 1
plt.scatter(x[0, negInd[0, :]], x[1, negInd[0, :]], color='b', s=0.3)
plt.scatter(x[0, posInd[0, :]], x[1, posInd[0, :]], color='r', s=0.3)
plt.figure(1)
plt.show()
```



In [2]:
```
lambda_val = 0.001
max_iterations = 10

w = np.zeros(2)
b = 0.0

objective_values = []
```

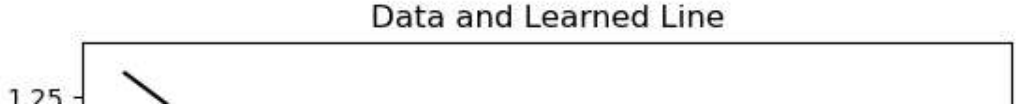
In [3]:
```
for iteration in range(1, max_iterations + 1): # don't want to divide by 0
    step_size = 100.0 / iteration
```

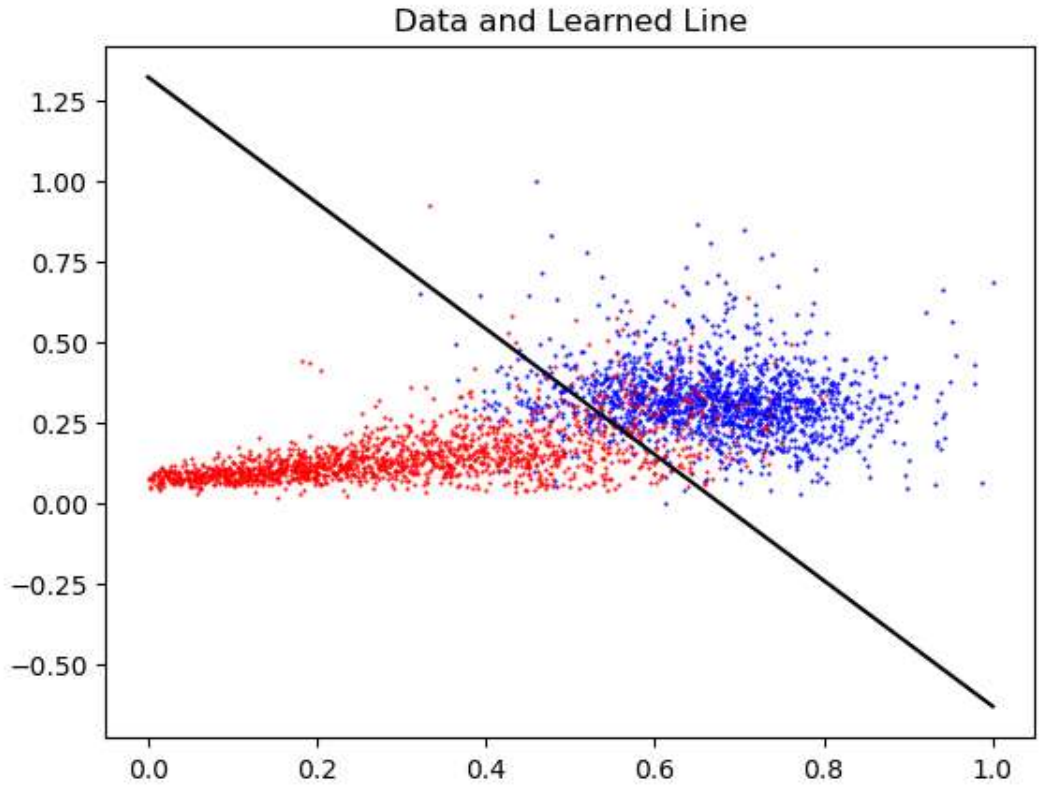
In [3]:
```
for iteration in range(1, max_iterations + 1): # don't want to divide by 0
    step_size = 100.0 / iteration

    # sum each subgradient
    subgrad_w = np.zeros(2)
    subgrad_b = 0.0
    for i in range(y.shape[1]):
        margin = y[:, i] * (np.dot(x[:, i], w) + b)
        if margin < 1:
            subgrad_w += -y[:, i] * x[:, i]
            subgrad_b += -y[:, i]

    # regularization term for bothe margin < 1 and margin >= 1
    subgrad_w /= y.shape[1]
    subgrad_b /= y.shape[1]
    subgrad_w += lambda_val * w

    w -= step_size * subgrad_w
    b -= step_size * subgrad_b


    hinge_loss = np.maximum(0, 1 - y * (x.T.dot(w) + b))
    objective_value = np.mean(hinge_loss) + 0.5 * lambda_val * (np.linalg.norm(w,
    objective_values.append(objective_value)
```

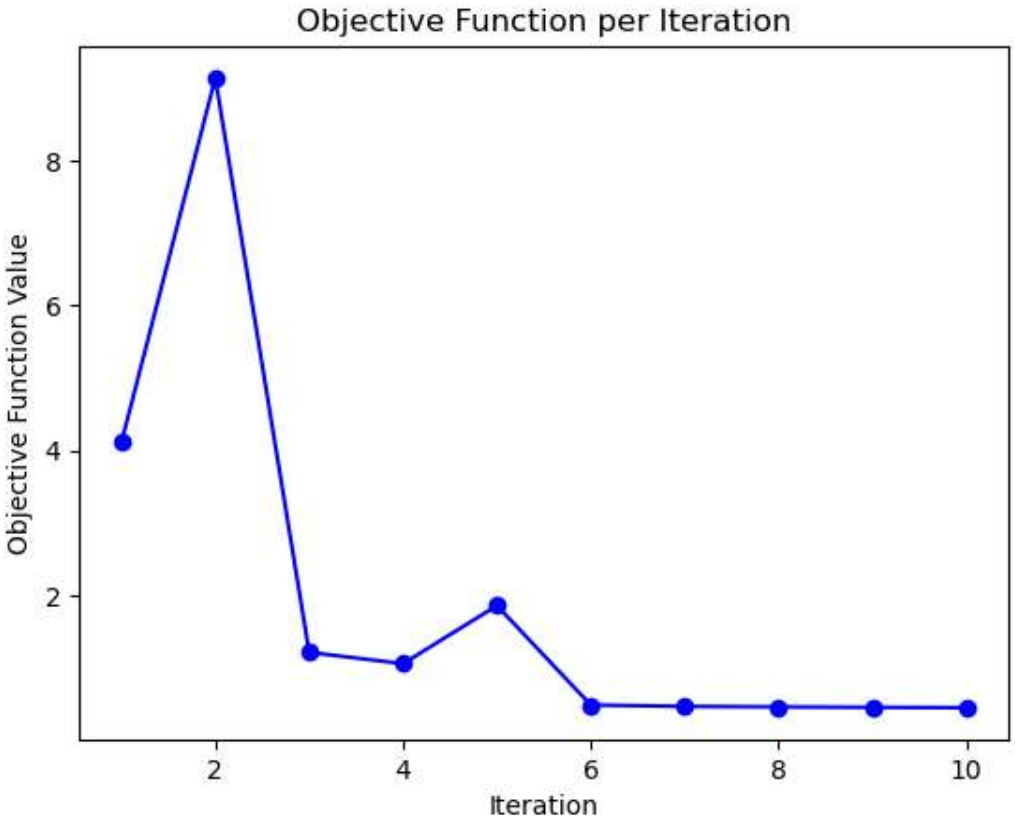
In [4]:
```
print("Learned hyperplane parameters:")
print("w =", w)
print("b =", b)
```

In [4]:
```
print("Learned hyperplane parameters:")
print("w =", w)
print("b =", b)


margin = np.min(abs((x.T.dot(w) + b) / np.linalg.norm(w, 2)))
print("The margin = ", margin)


min_objective_value = min(objective_values)
print("The minimun objective value =", min_objective_value)

# Plot the data and the learned line
plt.figure(1)
plt.scatter(x[0, negInd[0, :]], x[1, negInd[0, :]], color='b', s=0.3)
plt.scatter(x[0, posInd[0, :]], x[1, posInd[0, :]], color='r', s=0.3)
x_plot = np.linspace(0, 1, 2)
y_plot = ((-w[0]) * x_plot - b) / w[1]
plt.plot(x_plot, y_plot, 'k-')
plt.title("Data and Learned Line")

# Plot the objective function as a function of iteration number
plt.figure(2)
plt.plot(range(1, max_iterations + 1), objective_values, 'bo-')
plt.xlabel("Iteration")
plt.ylabel("Objective Function Value")
plt.title("Objective Function per Iteration")

plt.show()
```

```
Learned hyperplane parameters:
w = [-17.81627138  -9.11707611]
b = [12.0680196]
The margin =  4.713532755725594e-05
The minimun objective value = 0.4498841370611548
```







Part 2 stochastic gradient descent

In [5]:
```
lambda_val = 0.001
max_iterations = 10
```

In [5]:
```
lambda_val = 0.001
max_iterations = 10

w = np.zeros(2)
b = 0.0

objective_values = []

n_passes = 10
```

In [6]:
```
permutation = np.random.permutation(10)
print(permutation)
x1 = y[:, permutation]
x1.shape
```

```
[2 8 4 9 1 6 7 3 0 5]
```

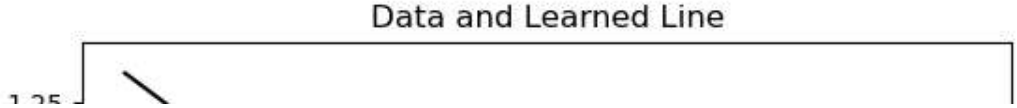
Out[6]:
```
(1, 10)
```

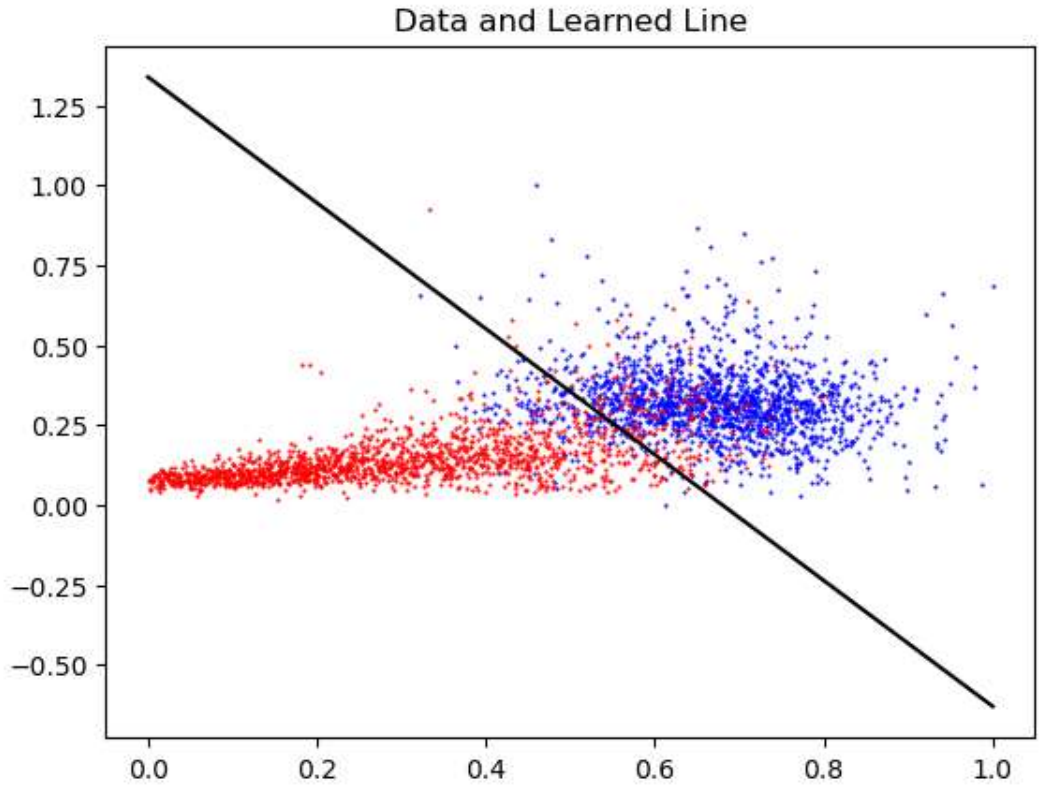
In [7]:
```
for iteration in range(0, max_iterations): # don't want to divide by 0
    permutation = np.random.permutation(y.shape[1])
    x_shuffle = x[:, permutation]
    y_shuffle = y[:, permutation]

    for j in range(y.shape[1]):
        step_size = 100.0 / (iteration * y.shape[1] + (j+1))
        subgrad_w = np.zeros(2)
        subgrad_b = 0.0
        # take 10 data per time, and do 10n times

        margin = y_shuffle[:, j] * (np.dot(x_shuffle[:, j], w) + b)
        if margin < 1:
            subgrad_w += -y_shuffle[:, j] * x_shuffle[:, j]
            subgrad_b += -y_shuffle[:, j]

    # regularization term for bothe margin < 1 and margin >= 1
        subgrad_w += lambda_val * w

        w -= step_size * subgrad_w
        b -= step_size * subgrad_b


    hinge_loss = np.maximum(0, 1 - y * (x.T.dot(w) + b))
    objective_value = np.mean(hinge_loss) + 0.5 * lambda_val * (np.linalg.norm(w,
    objective_values.append(objective_value)
```

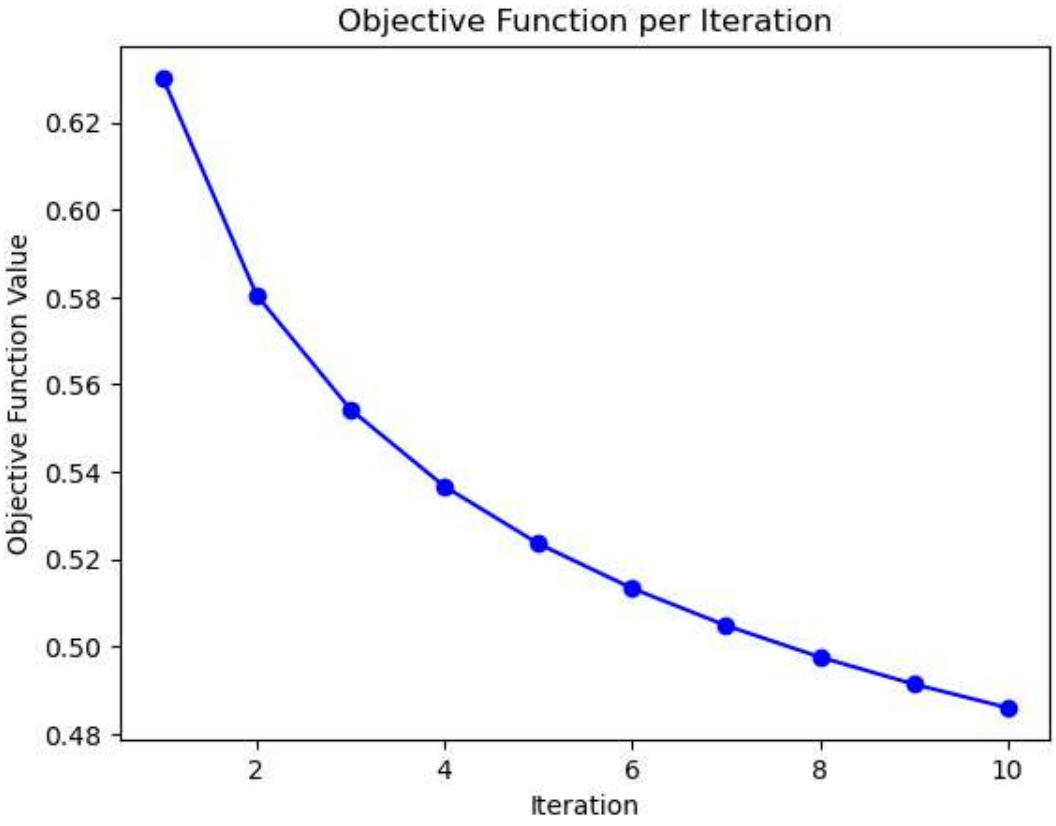
In [8]:
```
print("Learned hyperplane parameters:")
print("w =", w)
print("b =", b)
```

In [8]:
```
print("Learned hyperplane parameters:")
print("w =", w)
print("b =", b)


margin = np.min(abs((x.T.dot(w) + b) / np.linalg.norm(w, 2)))
print("The margin = ", margin)


min_objective_value = min(objective_values)
print("The minimun objective value =", min_objective_value)

# Plot the data and the learned line
plt.figure(1)
plt.scatter(x[0, negInd[0, :]], x[1, negInd[0, :]], color='b', s=0.3)
plt.scatter(x[0, posInd[0, :]], x[1, posInd[0, :]], color='r', s=0.3)
x_plot = np.linspace(0, 1, 2)
y_plot = ((-w[0]) * x_plot - b) / w[1]
plt.plot(x_plot, y_plot, 'k-')
plt.title("Data and Learned Line")

# Plot the objective function as a function of iteration number
plt.figure(2)
plt.plot(range(1, max_iterations + 1), objective_values, 'bo-')
plt.xlabel("Iteration")
plt.ylabel("Objective Function Value")
plt.title("Objective Function per Iteration")

plt.show()
```

```
Learned hyperplane parameters:
w = [-19.08784073  -9.69244475]
b = [12.97706403]
The margin =  8.239383798670525e-05
The minimun objective value = 0.4860044411801069
```




Data and Learned Line
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
0.0
0.2
0.4
0.6
0.8
1.0


Objective Function per Iteration
Objective Function Value
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0.48
2
4
6
8
10
Iteration

5. $K(u, v) = (u^T v + 1)^2$

$$= \left(\sum_{i=1}^{3}(u_i v_i + 1)\right)\left(\sum_{j=1}^{3}(u_j v_j + 1)\right)$$

$$= \sum_{i=1}^{3}\sum_{j=1}^{3} u_i v_i u_j v_j + 2\sum_{i=1}^{3} u_i v_i + 1$$

$$= \sum_{i=1}^{3}\sum_{j=1}^{3} u_i u_j v_i v_j + \sum_{i=1}^{3} \sqrt{2}u_i \sqrt{2}v_i + 1$$

since $u, v \in R^3$

$$\Rightarrow \underline{\Phi}(u) = [u_1^2, u_1u_2, u_1u_3, u_2^2, u_2u_3, u_3^2, \sqrt{2}u_1, \sqrt{2}u_2, \sqrt{2}u_3, 1]$$

$$\underline{\Phi}(v) = [v_1^2, v_1v_2, v_1v_3, v_2^2, v_2v_3, v_3^2, \sqrt{2}v_1, \sqrt{2}v_2, \sqrt{2}v_3, 1]$$

XX