4

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LassoCV
from sklearn.metrics import precision_recall_curve
```

In [2]:
```
n = 600  # Samples
d = 1200  # Features
k = 120  # Relevant features
sigma = 1.0
```

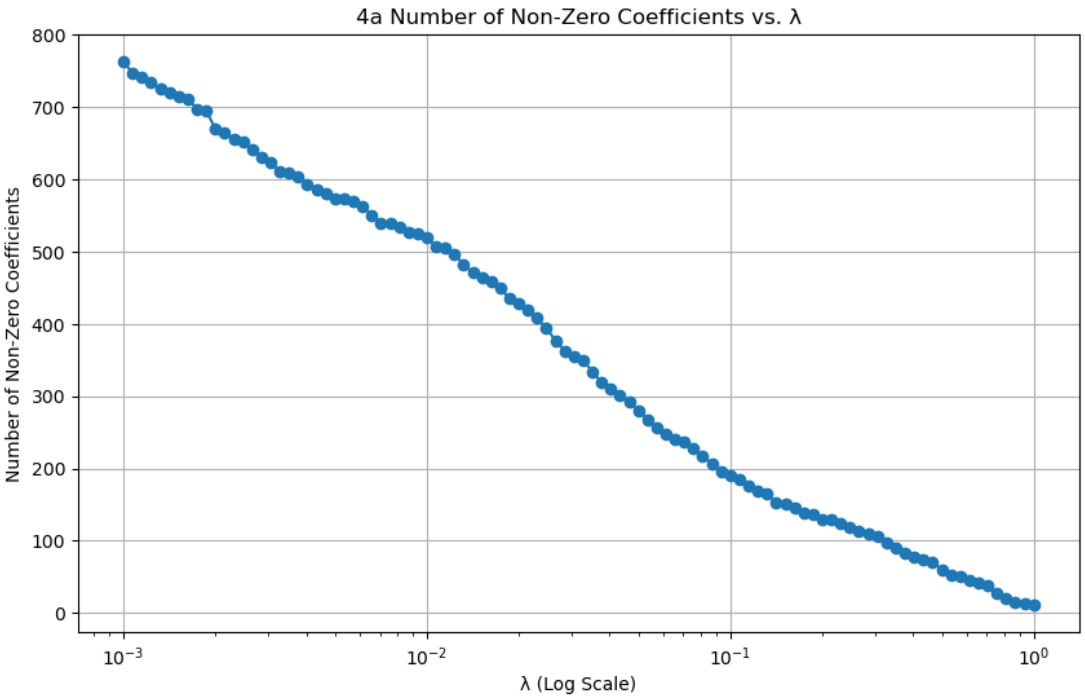
In [3]:
```
# Generate data
np.random.seed(42)
X = np.random.normal(0, 1, size=(n, d))
w_true = np.zeros(d)
for j in range(k):
    w_true[j] = (j + 1) / k
epsilon = np.random.normal(0, sigma, n)
y = X @ w_true + epsilon
```

In [8]:
```
# λ values range
alphas = np.logspace(0, -3, 100)
```

4a Number of Non-Zero Coefficients vs. λ

In [9]:
```
non_zero_counts = []

# Calculate the number of non-zeros coefficients
for alpha in alphas:
    lasso = LassoCV(alphas=[alpha], cv=5,  tol=1e-3)
    lasso.fit(X, y)
    w_hat = lasso.coef_

    non_zero_count = np.sum(w_hat != 0)
    non_zero_counts.append(non_zero_count)
```

In [10]:
```
# 4a Plot the Number of Non-Zero Coefficients vs. λ
plt.figure(figsize=(10, 6))
plt.plot(alphas, non_zero_counts, marker='o', linestyle='-')
plt.xlabel('λ (Log Scale)')
plt.ylabel('Number of Non-Zero Coefficients')
plt.xscale('log')
plt.title('4a Number of Non-Zero Coefficients vs. λ')
plt.grid(True)
```

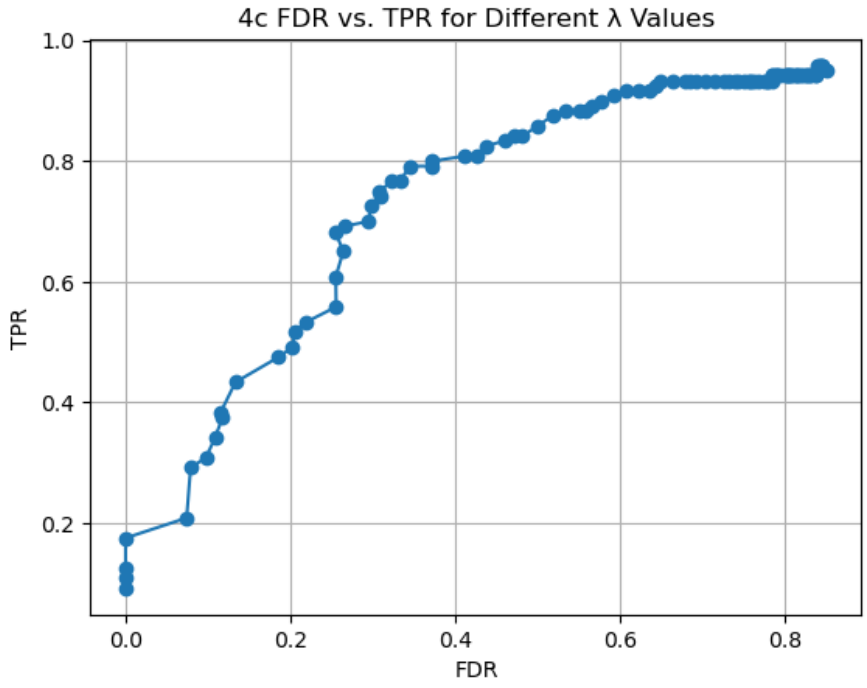

4a Number of Non-Zero Coefficients vs. λ
Number of Non-Zero Coefficients
λ (Log Scale)
800
700
600
500
400
300
200
100
0
10^{-3}
10^{-2}
10^{-1}
10^{0}

4c FDR vs. TPR

In [11]:
```
fdr_values = []
tpr_values = []
```

In [12]:
```
for alpha in alphas:
    lasso = LassoCV(alphas=[alpha], cv=5, tol=1e-3)
    lasso.fit(X, y)
    w_hat = lasso.coef_

    # Calculate FDR and TPR
    selected_features = np.where(w_hat != 0)[0]
    true_positives = np.intersect1d(selected_features, np.arange(k))
    false_positives = np.setdiff1d(selected_features, np.arange(k))

    if len(false_positives) > 0:
        fdr = len(false_positives) / len(selected_features)
    else:
        fdr = 0.0

    if len(true_positives) > 0:
        tpr = len(true_positives) / k
    else:
        tpr = 0.0

    fdr_values.append(fdr)
    tpr_values.append(tpr)
```

In [13]:
```
# 4c Plot FDR vs. TPR for each alpha
plt.plot(fdr_values, tpr_values, marker='o', linestyle='-')
plt.xlabel('FDR')
plt.ylabel('TPR')
plt.title('4c FDR vs. TPR for Different λ Values')
plt.grid(True)
```



5

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
```

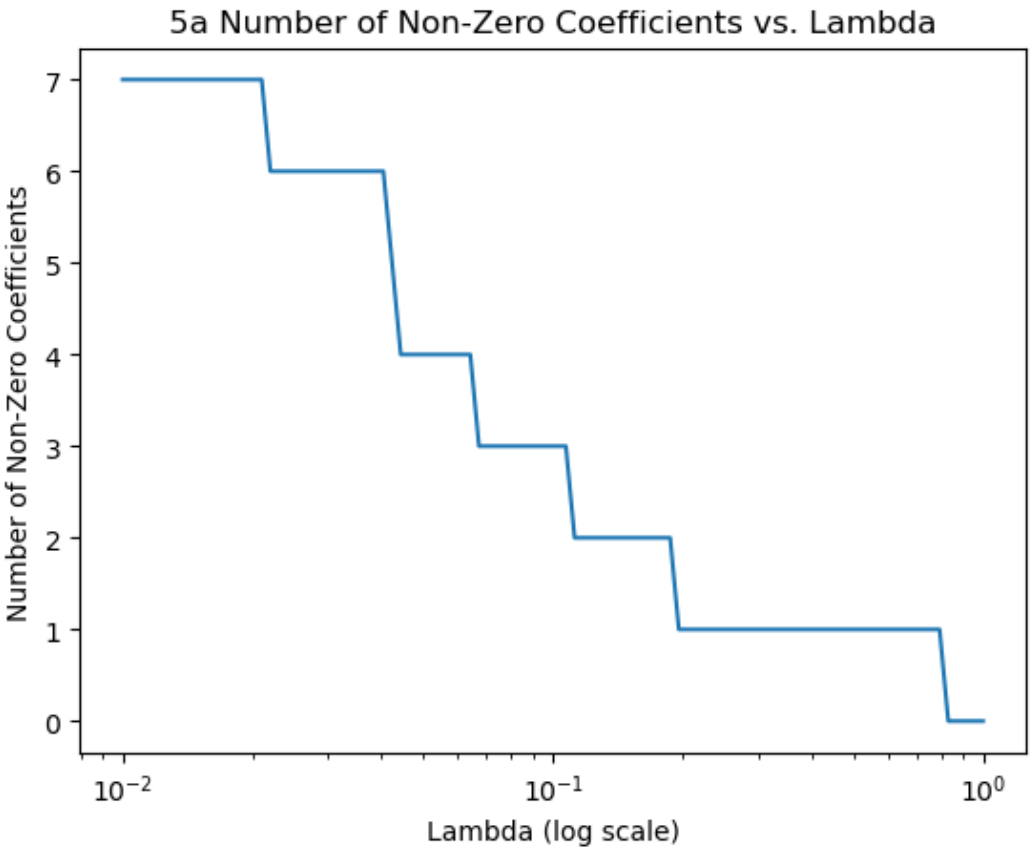
In [2]:
```
# Load the California Housing dataset
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
X = housing.data
y = housing.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

5a Number of Non-Zero Coefficients vs. Lambda

In [3]:
```
# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

lambda_values = np.logspace(0, -2, 100)
nonzero_coeffs = []

# Initialize weights
w = np.zeros(X_train.shape[1])
```

In [4]:
```
for alpha in lambda_values:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train)
    w = lasso.coef_
    nonzero_coeffs.append(np.sum(w != 0))
```

In [6]:
```
# 5a Plot the number of nonzero coefficients vs. lambda
plt.figure()
plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Number of Non-Zero Coefficients')
plt.plot(lambda_values, nonzero_coeffs)
plt.title('5a Number of Non-Zero Coefficients vs. Lambda')
```

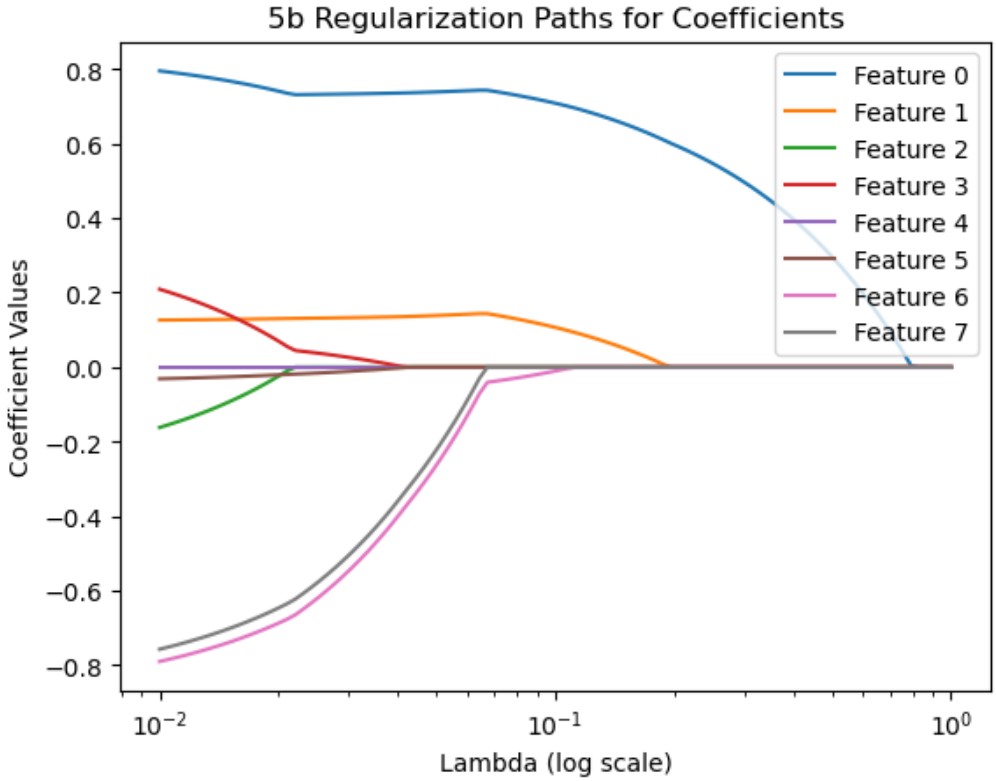

5a Number of Non-Zero Coefficients vs. Lambda
Number of Non-Zero Coefficients
0
1
2
3
4
5
6
7
10^{-2}
10^{-1}
10^{0}
Lambda (log scale)

5b Regularization Paths for Coefficients

In [8]:
```
coefficients = []

for alpha in lambda_values:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train)
    w = lasso.coef_
    coefficients.append(w)
```

In [13]:
```
# 5b Plot the regularization paths for all input variables
plt.figure()
plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Coefficient Values')

for i in range(X_train.shape[1]):
    plt.plot(lambda_values, [w[i] for w in coefficients], label=f'Feature {i}')
plt.legend(loc='upper right')
plt.title('5b Regularization Paths for Coefficients')
```

Out[13]: Text(0.5, 1.0, '5b Regularization Paths for Coefficients')

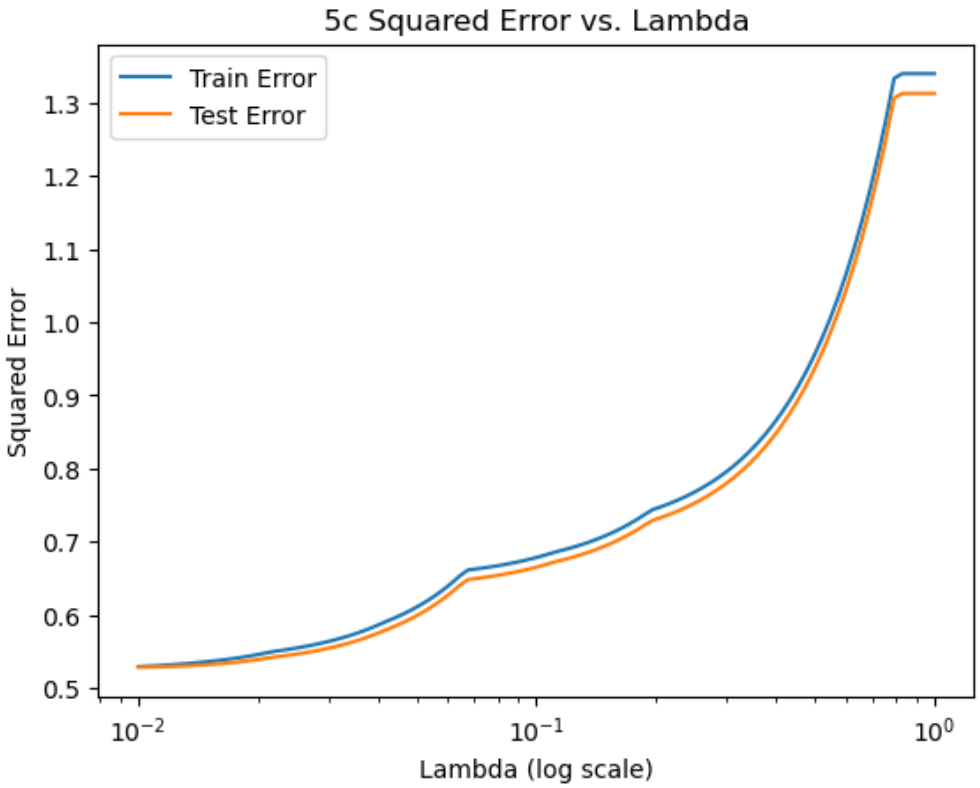


5c Squared Error vs. Lambda

In [11]:
```
from sklearn.metrics import mean_squared_error

train_errors = []
test_errors = []

for alpha in lambda_values:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train)
    y_train_pred = lasso.predict(X_train)
    y_test_pred = lasso.predict(X_test)
    train_errors.append(mean_squared_error(y_train, y_train_pred))
    test_errors.append(mean_squared_error(y_test, y_test_pred))
```

In [14]:
```
# 5c Plot the squared error on training and test data vs. lambda
plt.figure()
plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Squared Error')
plt.plot(lambda_values, train_errors, label='Train Error')
plt.plot(lambda_values, test_errors, label='Test Error')
plt.legend()
plt.title('5c Squared Error vs. Lambda')
```

Out[14]:
```
Text(0.5, 1.0, '5c Squared Error vs. Lambda')
```



In [17]:
```
# Define a range of lambda values)
lambda_values = np.logspace(np.log10(0.5), np.log10(1e4), 100)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

average_errors = []
standard_errors = []
```

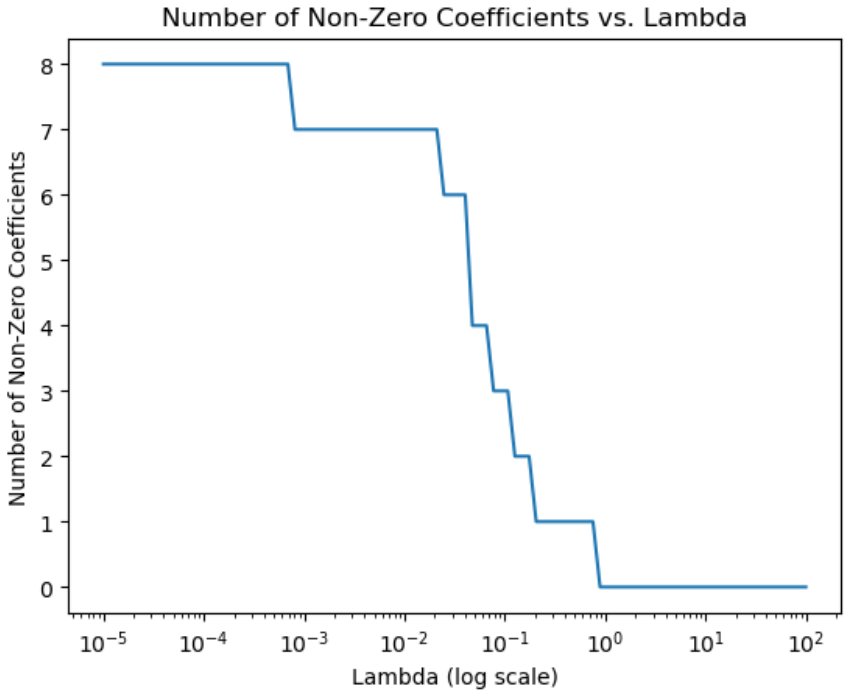
In [18]:
```
for alpha in lambda_values:
    errors = []
    for train_index, val_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train[train_index], X_train[val_index]
        y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

        lasso = Lasso(alpha=alpha)
        lasso.fit(X_train_fold, y_train_fold)
        y_val_pred = lasso.predict(X_val_fold)
        error = mean_squared_error(y_val_fold, y_val_pred)
        errors.append(error)

    average_error = np.mean(errors)
    standard_error = statistics.stdev(errors)
    average_errors.append(average_error)
    standard_errors.append(standard_error)
```

5d Squared Error vs. Lambda

```
In [24]: from sklearn.linear_model import Lasso

         # Fit the Lasso model with lambda = 100
         lasso = Lasso(alpha=100)
         lasso.fit(X_train, y_train)

         coefficients_100 = lasso.coef_

         # Find the feature with the largest (most positive) Lasso coefficient
         max_positive_feature = np.argmax(coefficients_100)
         max_positive_coefficient = coefficients_100[max_positive_feature]

         # Find the feature with the most negative Lasso coefficient
         max_negative_feature = np.argmin(coefficients_100)
         max_negative_coefficient = coefficients_100[max_negative_feature]

         print(f"Feature with the largest positive coefficient: Feature {max_positive_feature}, Coefficient: {max_positive_coefficient}")
         print(f"Feature with the most negative coefficient: Feature {max_negative_feature}, Coefficient: {max_negative_coefficient}")

Feature with the largest positive coefficient: Feature 0, Coefficient: 0.0
Feature with the most negative coefficient: Feature 0, Coefficient: 0.0
```

Why such weights?

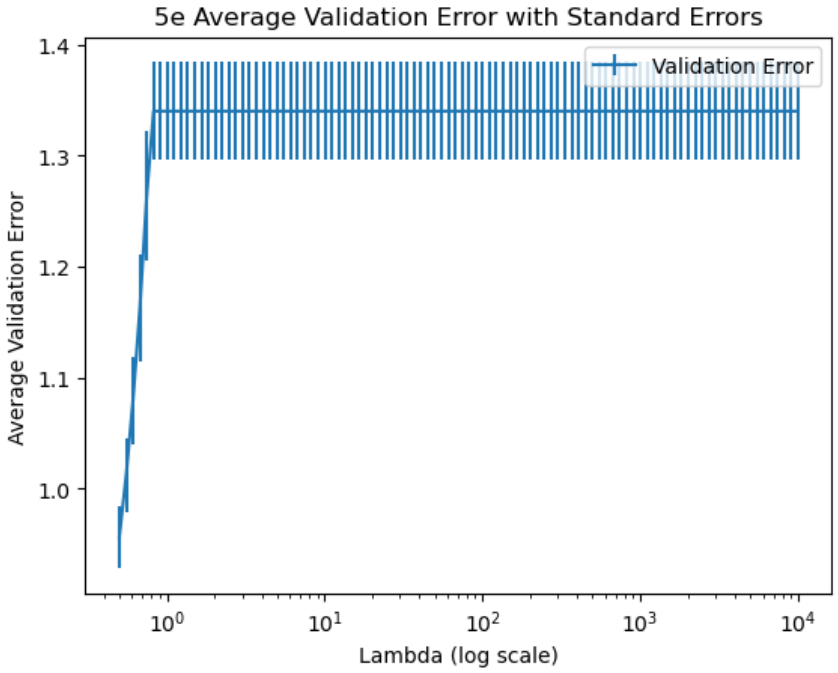
I am obtaining zero coefficients even for a relatively large value of λ such as 100, it suggests that the chosen λ is too large for the dataset, resulting in excessive regularization and effectively eliminating most features. In the context of the California Housing dataset, the Lasso regression may be forcing the coefficients of most features to zero due to the high penalty imposed by the large λ value. This could indicate that the model is not able to find strong linear relationships between the features and the target variable under such strong regularization. In this case, a smaller value of λ will allow the model to retain more features.



5e Plot the average validation error with standard errors and Optimal Lambda

In [22]:
```
# 5e Plot the average validation error with standard errors
plt.figure()
plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Average Validation Error')
plt.errorbar(lambda_values, average_errors, yerr=standard_errors, label='Validation Error')
plt.legend()
plt.title('5e Average Validation Error with Standard Errors')
```

Out[22]:
```
Text(0.5, 1.0, '5e Average Validation Error with Standard Errors')
```



In [21]:
```
# Optimal lambda
optimal_lambda = lambda_values[np.argmin(average_errors)]
print(f"Optimal lambda according to cross-validation: {optimal_lambda}")
```

```
Optimal lambda according to cross-validation: 0.5
```

6

```
# Gradient function
def gradient(w, b, X, y, lam):
    m = len(y)
    z = b + X.dot(w)
    h = sigmoid(z)
    dw = (1 / m) * X.T.dot(h - y) + (lam / m) * w
    db = (1 / m) * np.sum(h - y)
    return dw, db

# Gradient Descent function
def gradient_descent(X, y, alpha, lam, num_iter):
    w = np.zeros(X.shape[1])
    b = 0

    J_history = np.zeros(num_iter)
    iter_array = np.arange(1, num_iter + 1)

    for i in range(num_iter):
        # Compute the cost and gradient
        J = cost_function(w, b, X, y, lam)
        dw, db = gradient(w, b, X, y, lam)

        w = w - alpha * dw
        b = b - alpha * db

        # Store the cost in the history array
        J_history[i] = J

        # Print the costs every 100 iterations
        if (i + 1) % 100 == 0:
            print(f"Cost at iteration {i + 1}: {J}")

    return w, b, J_history, iter_array
```

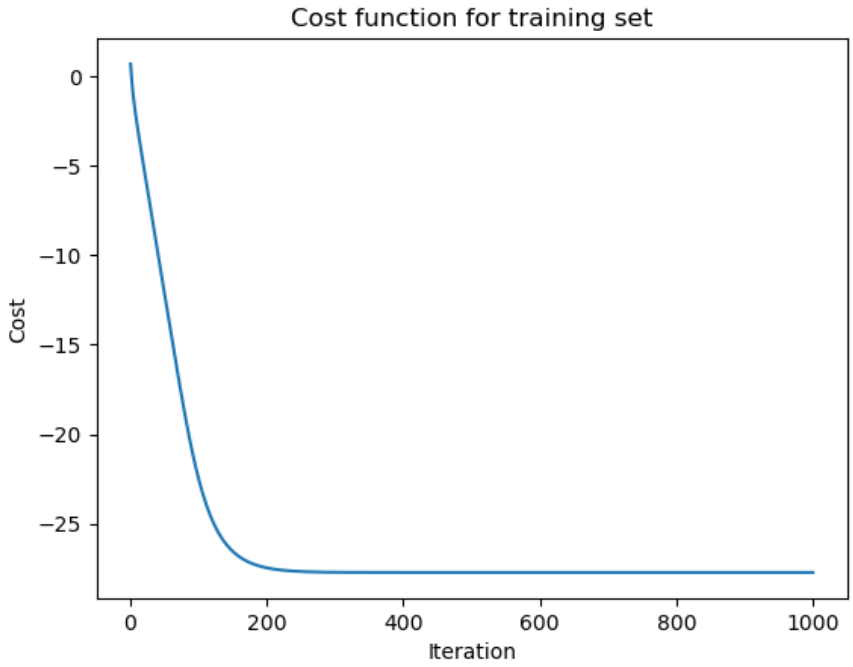
6b Gradient Descent

In [16]:
```
# Load the MNIST dataset
mnist_train = datasets.MNIST(root='./data', train=True, download=True)
mnist_test = datasets.MNIST(root='./data', train=False, download=True)

X_train = mnist_train.data.numpy()
y_train = mnist_train.targets.numpy()
X_test = mnist_test.data.numpy()
y_test = mnist_test.targets.numpy()

X_train = X_train.reshape(X_train.shape[0], -1) / 255.0
X_test = X_test.reshape(X_test.shape[0], -1) / 255.0

# Convert the labels to binary values: 1 for 6 and -1 for 9
y_train = np.where(y_train == 6, 1, -1)
y_test = np.where(y_test == 6, 1, -1)

X_train = X_train[(y_train == 1) | (y_train == -1)]
y_train = y_train[(y_train == 1) | (y_train == -1)]
X_test = X_test[(y_test == 1) | (y_test == -1)]
y_test = y_test[(y_test == 1) | (y_test == -1)]
```

In [30]:
```
alpha = 0.01
lam = 0.01
num_iter = 1000

# Run gradient descent
w, b, J_history_train, iter_array_train = gradient_descent(X_train, y_train, alpha, lam, num_iter)
```

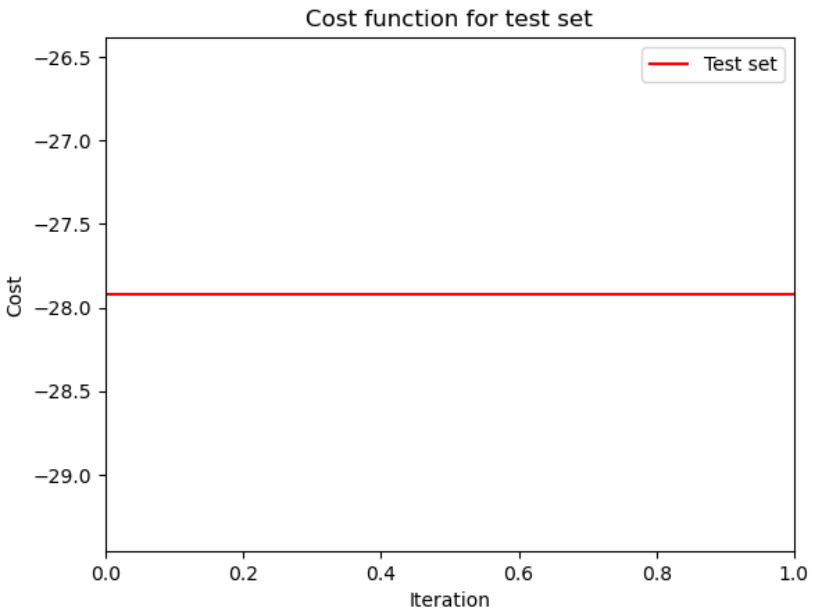
In [36]:
```
# Plot the cost function as a function of the iteration number for the training set
plt.plot(iter_array_train, J_history_train)
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Cost function for training set")
```

Out[36]:
```
Text(0.5, 1.0, 'Cost function for training set')
```



In [37]:
```
# Compute the cost function
J_history_test = cost_function(w, b, X_test, y_test, lam)

# Plot a horizontal line for the cost function for test
plt.axhline(y=J_history_test, color='r', label='Test set')
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Cost function for test set")
plt.legend()
```

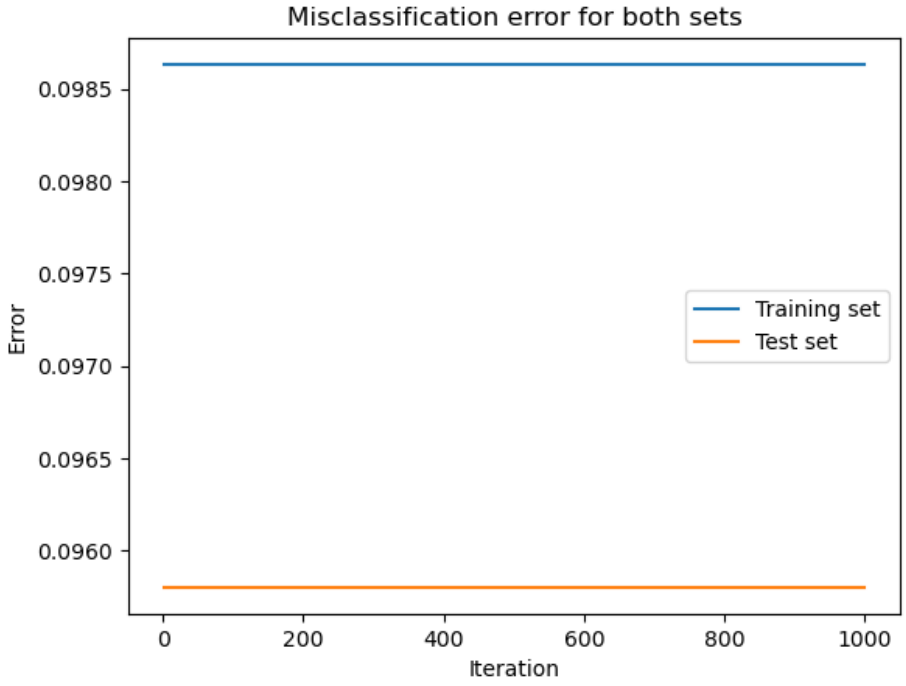
Out[37]:
```
<matplotlib.legend.Legend at 0x2225953ae50>
```



In [38]:
```
# Define a function to compute the misclassification error
def misclassification_error(w, b, X, y):
    # Predict the labels using the sign function
    y_pred = np.sign(b + X.dot(w))
    errors = np.sum(y_pred != y)
    return errors / len(y)
```

In [39]:
```
# Compute the misclassification error for the training set
error_train = misclassification_error(w, b, X_train, y_train)

# Compute the misclassification error for the test set
error_test = misclassification_error(w, b, X_test, y_test)

# Plot the misclassification error as a function of the iteration number for both sets
plt.plot(iter_array_train, np.repeat(error_train, num_iter), label='Training set')
plt.plot(iter_array_train, np.repeat(error_test, num_iter), label='Test set')
plt.xlabel("Iteration")
plt.ylabel("Error")
plt.title("Misclassification error for both sets")
plt.legend()
```

Out[39]:
```
<matplotlib.legend.Legend at 0x2225989f2d0>
```

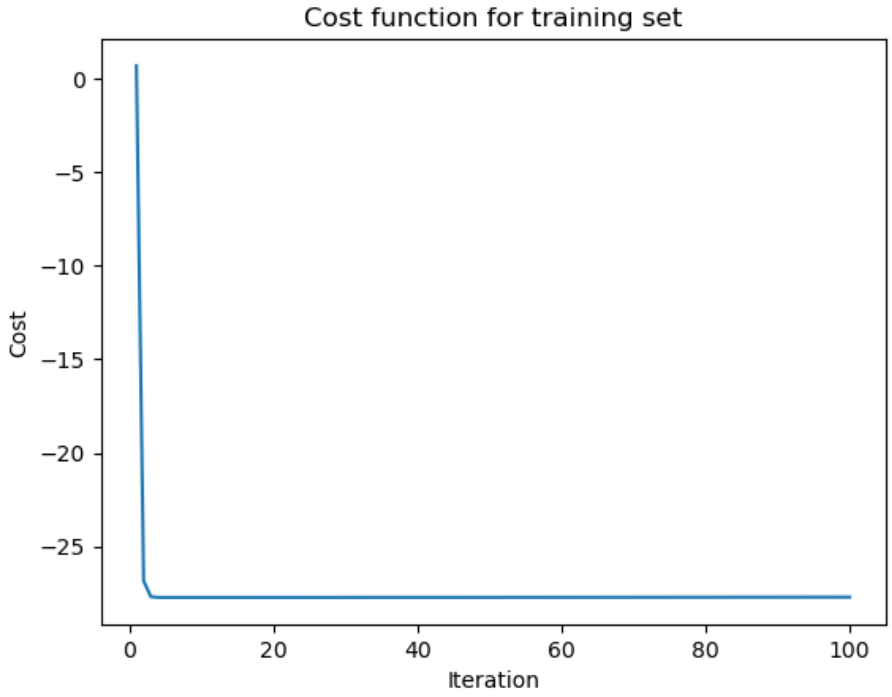


6c Stochastic Gradient Descent

In [24]:
```python
# Define the stochastic gradient descent function
def stochastic_gradient_descent(X, y, alpha, lam, num_iter):
    # Initialize w and b to zeros
    w = np.zeros(X.shape[1])
    b = 0

    # Initialize the cost history and iteration arrays
    J_history = np.zeros(num_iter)
    iter_array = np.arange(1, num_iter + 1)

    for i in range(num_iter):
        j = np.random.choice(np.arange(0, len(y)))
        Xj = X[j].reshape(1, -1)
        yj = y[j]

        # Compute the cost and gradient using the selected data point
        J = cost_function(w, b, Xj, yj, lam)
        dw, db = gradient(w, b, Xj, yj, lam)

        # Update w and b
        w = w - alpha * dw
        b = b - alpha * db

        # Store the cost in the history array
        J_history[i] = J

        # Print the cost every 100 iterations
        if (i + 1) % 100 == 0:
            print(f"Cost at iteration {i + 1}: {J}")

    # Return the final values of w, b, and the cost history and iteration arrays
    return w, b, J_history, iter_array
```

In [57]:
```python
plt.plot(iter_array_train, J_history_train)
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Cost function for training set")
```

Out[57]:
```
Text(0.5, 1.0, 'Cost function for training set')
```



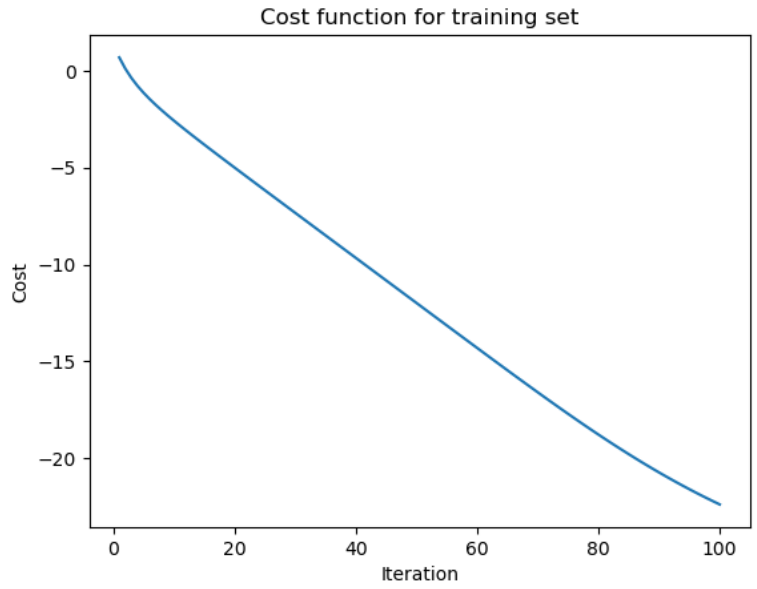
6d Stochastic Mini-Batch Gradient Descent

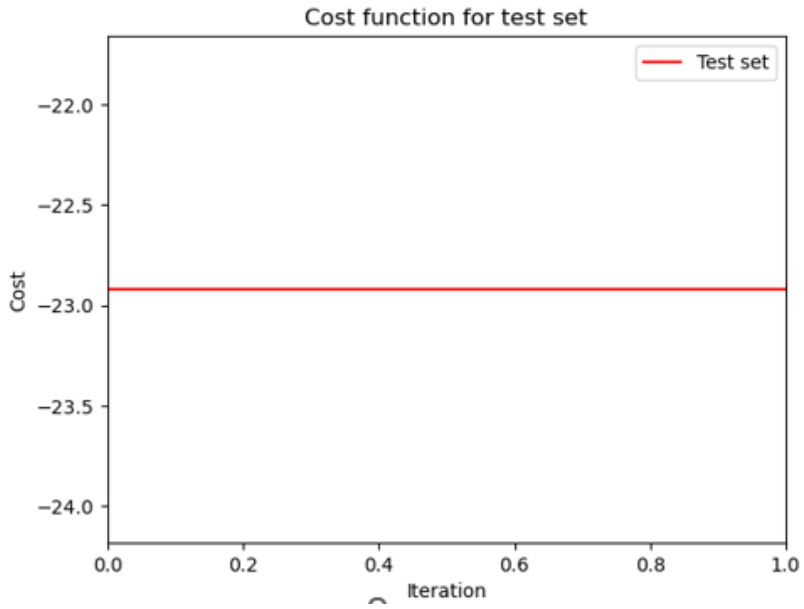
In [41]:
```
alpha = 0.01
lam = 0.01
num_iter = 100

# Run gradient descent
w, b, J_history_train, iter_array_train = gradient_descent(X_train, y_train, alpha, la
```

```
Cost at iteration 100: -22.3978583873724
```

In [42]:
```
# Plot the cost function as a function of the iteration number for the training set
plt.plot(iter_array_train, J_history_train)
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Cost function for training set")
```

Out[42]:
```
Text(0.5, 1.0, 'Cost function for training set')
```





Out[44]: <matplotlib.legend.Legend at 0x22259d47e90>