econ566 - pset06 - Dili Maduabum, Joshua Bailey

problem 1

1.a

$$z_1 = \begin{bmatrix} z_{11} \\ z_{12} \\ \vdots \\ z_{1m1} \end{bmatrix}, \quad z_2 = \begin{bmatrix} z_{21} \\ z_{22} \\ \vdots \\ z_{2m2} \end{bmatrix}, \quad \text{and} \quad z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \implies z = \begin{bmatrix} z_{11} \\ z_{12} \\ \vdots \\ z_{1m1} \\ z_{21} \\ z_{22} \\ \vdots \\ z_{2m2} \end{bmatrix}$$

By definition:

$$\|z_1\|_2^2 = \sum_{i=1}^{m_1} (z_{1i})^2 \text{ and } \|z_2\|_2^2 = \sum_{i=1}^{m_2} (z_{2i})^2$$

$$\therefore \|z\|_2^2 = \sum_{i=1}^{m_1} (z_{1i})^2 + \sum_{i=1}^{m_2} (z_{2i})^2 \tag{1}$$

$$= \|z_1\|_2^2 + \|z_2\|_2^2 /// \tag{2}$$

1.b

Background:

$$y = \begin{pmatrix} y^T \\ 0_{p\times 1}^T \end{pmatrix}^T \implies \begin{pmatrix} y \\ 0_{p\times 1} \end{pmatrix}, \quad \tilde{I} = \begin{pmatrix} 1_{n\times 1}^T \\ 0_{p\times 1}^T \end{pmatrix}^T \implies \begin{pmatrix} 1_{n\times 1} \\ 0_{p\times 1} \end{pmatrix},$$

$$\tilde{X} = \begin{pmatrix} X^T \\ \sqrt{n(1-\alpha)\lambda} I_p \end{pmatrix}^T \implies \begin{pmatrix} X \\ (\sqrt{n(1-\alpha)\lambda} I_p)^T \end{pmatrix} \implies \begin{pmatrix} X \\ \sqrt{n(1-\alpha)\lambda} I_p \end{pmatrix}$$

$$\text{since } \sqrt{n(1-\alpha)\lambda} I_p \text{ is a diagonal matrix}$$

From a, we know that:

$$\frac{1}{2n}\|\tilde{y} - \tilde{I}\beta_0 - \tilde{X}\beta\|_2^2 + \lambda\alpha\|\beta\|_1$$

can be written as:

$$\begin{aligned}
&\frac{1}{2n}\left(\|y - 1_{n\times1}\beta_0 - X\beta\|_2^2 + \|0_{p\times1} - 0_{p\times1}\beta_0 - \sqrt{n(1-\alpha)\lambda}I_p\beta\|_2^2\right) + \lambda\alpha\|\beta\|_1 \\
&= \frac{1}{2n}\left(\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \| - \sqrt{n(1-\alpha)\lambda}I_p\beta\|_2^2\right) + \lambda\alpha\|\beta\|_1 && (3)\\
&= \frac{1}{2n}\left(\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + (-1)^2\|\sqrt{n(1-\alpha)\lambda}I_p\beta\|_2^2\right) + \lambda\alpha\|\beta\|_1 && (4)\\
&= \frac{1}{2n}\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \frac{1}{2n}\|\sqrt{n(1-\alpha)\lambda}I_p\beta\|_2^2 + \lambda\alpha\|\beta\|_1 && (5)\\
&= \frac{1}{2n}\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \frac{1}{2n}(\sqrt{n(1-\alpha)\lambda})^2\|I_p\beta\|_2^2 + \lambda\alpha\|\beta\|_1 && (6)\\
&= \frac{1}{2n}\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \frac{1}{2\cancel{n}}\cancel{n}(1-\alpha)\lambda\|I_p\beta\|_2^2 + \lambda\alpha\|\beta\|_1 && (7)\\
&\text{Since } I_p \text{ is the identity matrix, } I_p\beta = \beta\text{, thus} && (8)\\
&= \frac{1}{2n}\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \frac{1}{2}(1-\alpha)\lambda\|\beta\|_2^2 + \lambda\alpha\|\beta\|_1 && (9)\\
&= \frac{1}{2n}\|y - \beta_0 1_{n\times1} - X\beta\|_2^2 + \lambda\left(\alpha\|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2\right) && (10)
\end{aligned}$$

1.c

Utilizing $\tilde{y}$, $\tilde{I}$, and $\tilde{X}$ might offer computational efficiency advantages and potentially reduce memory usage.

2.a

The program describes a coordinate descent update for a single coefficient β_j of the vector β in the context of minimizing the cost function $f(y, X, \beta, \lambda, \alpha)$. Coordinate descent is an optimization algorithm that updates one component of β at a time while keeping the other components fixed. Specifically, for the chosen jth component, the algorithm seeks to find the value of β_j that minimizes f, under the constraint that all other components of β (β_l for all $l \neq j$) are held constant at their current values ($\tilde{\beta}_l$). This approach simplifies the optimization problem because, at each step, it reduces the multidimensional optimization problem to a series of one-dimensional problems. Specifically:

1. For a given iteration, the algorithm selects one component of β, denoted β_j, to update. j is an index from the set $\{1, \ldots, p\}$, where p is the total number of predictors in X.

2. With all other components of β fixed at their current values ($\tilde{\beta}_l$ for $l \neq j$), the algorithm optimizes the value of β_j. This optimization is done by minimizing the cost function f, which is a blend of a squared loss and a regularization term affected by λ and α.

3. This step is repeated, cycling through each component of β, updating one component at a time based on the optimization criterion defined above. Each pass through all the components of β constitutes an iteration of the coordinate descent algorithm.
4. The process iterates until a stopping criterion is met, typically when changes in the cost function between iterations are below a predetermined threshold, indicating that the algorithm has converged to a minimum (or until a maximum number of iterations is reached).

The advantage of coordinate descent in this setting is its simplicity and efficiency, especially in high-dimensional spaces where traditional gradient descent methods may be computationally expensive. It's particularly well-suited for problems with a large number of features and when the regularization term is present, as it can efficiently handle the sparsity induced by the L1 penalty (lasso) and the shrinkage induced by the L2 penalty (ridge).

2.b

In [1]:
```python
import numpy as np
import pandas as pd

# Function to perform soft-thresholding
def soft_thresholding(a, b):
    """
    Soft-thresholding function S(a, b).

    :param a: The value to apply thresholding to.
    :param b: The threshold.
    :return: The result of soft-thresholding on a with threshold b.
    """
    return np.sign(a) * max(abs(a) - b, 0)

# Function to update beta_j using coordinate descent
def update_beta_j(y, X, lam, alpha, tilde_beta, j):
    """
    Calculates the updated value of beta_j (beta_j^*) using soft-thresholding.

    :param y: The target variable vector.
    :param X: The standardized feature matrix.
    :param lam: The regularization parameter lambda.
    :param alpha: The mixing parameter between L1 and L2 regularization.
    :param tilde_beta: The current estimate of the beta coefficients.
    :param j: The index of the beta coefficient to update.
    :return: The updated value of beta_j.
    """
    n = X.shape[0]
    x_j = X[:, j - 1]
    residual = y - np.dot(X, tilde_beta[1:]) + x_j * tilde_beta[j]
    beta_j_candidate = np.dot(x_j, residual) / n
    beta_j_star = soft_thresholding(beta_j_candidate, alpha * lam)
    return beta_j_star

# Load the dataset
data_path = 'ps6.csv'
data = pd.read_csv(data_path)
```

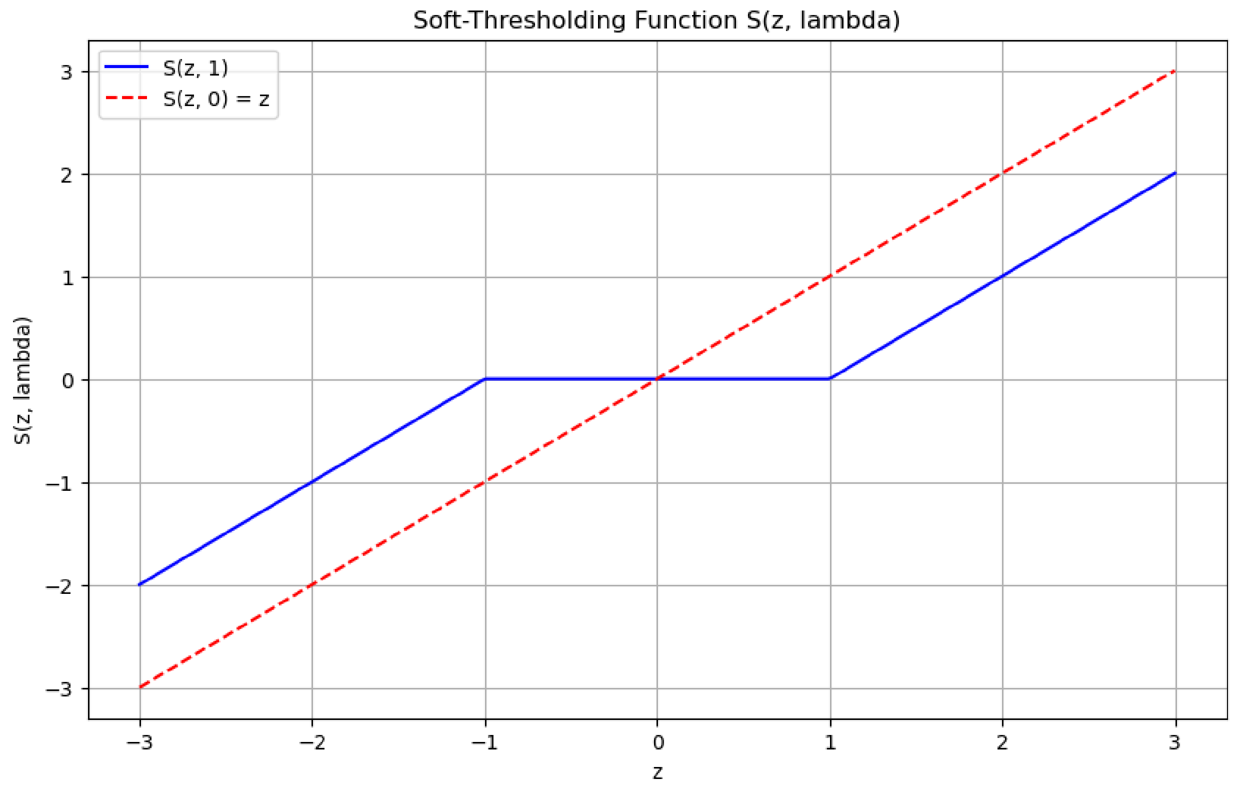
```
# Extracting Y as the first non-index column (V1)
y = data['V1'].values

# Extracting X (excluding the first two columns: index and target, then all the
X = data.iloc[:, 2:].values  # Skip the first (index) and second (target) colu

# Standardize the features
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)
X_standardized = (X - X_mean) / X_std

# Initialize parameters
lam = 0.1  # Regularization parameter lambda
alpha = 0.5  # Mixing parameter between L1 and L2 regularization
np.random.seed(0)  # For reproducibility
n_features = X_standardized.shape[1]
tilde_beta = np.random.randn(n_features + 1)  # +1 for the intercept, initiali

# Choose a feature index j to update (e.g., the second feature)
j = 1

# Update beta_j using the function
beta_j_star = update_beta_j(y, X_standardized, lam, alpha, tilde_beta, j)
print(f"Updated beta_j* for feature index {j}: {beta_j_star}")
```

```
Updated beta_j* for feature index 1: -0.3213218689163262
```

2.c

In [2]:

```
import matplotlib.pyplot as plt

# Define the soft-thresholding function S(z, lambda)
def S(z, lam):
    return np.sign(z) * np.maximum(np.abs(z) - lam, 0)

# Generate a range of z values from -3 to 3
z = np.linspace(-3, 3, 300)

# Calculate S(z, 1) and S(z, 0) for the range of z values
S_z_1 = S(z, 1)
S_z_0 = S(z, 0)  # Equivalent to z, since lam=0 results in no thresholding

# Plotting
plt.figure(figsize=(10, 6))

plt.plot(z, S_z_1, label='S(z, 1)', color='blue')
plt.plot(z, S_z_0, label='S(z, 0) = z', color='red', linestyle='--')

plt.title('Soft-Thresholding Function S(z, lambda)')
plt.xlabel('z')
plt.ylabel('S(z, lambda)')
plt.legend()
plt.grid(True)

plt.show()
```



$S(z, 0)$ directly corresponds to the update rule in a coordinate descent algorithm for OLS, where there's no regularization applied. Since this function is the identity function, it doesn't alter the value of z, implying that all features have the potential to be included in the model without any preference for sparsity. The likelihood of this function outputting zero is practically nil unless z itself is zero, reflecting OLS's tendency to use all available predictors without inducing sparsity.

$S(z, 1)$, on the other hand, illustrates the mechanism behind techniques like Lasso that aim to induce sparsity in the model coefficients. This function will set z to zero if its magnitude is less than the threshold (1 in this case), which directly contributes to sparsity. Given z is uniformly distributed between $[-3, 3]$, there's a one-third chance that $S(z, 1)$ results in zero, significantly higher than with $S(z, 0)$. This illustrates how regularization encourages sparsity by nullifying the effect of variables with coefficients that are not sufficiently large to exceed the threshold, thus prioritizing a simpler model that reduces overfitting and enhances interpretability.

These functions underline a trade-off in model fitting between complexity and simplicity: $S(z, 0)$ allows for a full, possibly overfit model by utilizing all predictors, whereas $S(z, 1)$ (with regularization) promotes a sparser, more interpretable model by excluding predictors that do not contribute sufficiently to the model's predictive power.

3

In [3]:
```
def cyclic_coordinate_descent_basic(y, X, lam, alpha, epsilon=0.1):
    n, p = X.shape
    beta = np.zeros(p)
    converge = False

    while not converge:
        beta_old = beta.copy()

        # Update for j=0 (intercept), not regularized
        beta[0] = beta[0] + np.sum(y - np.dot(X, beta)) / n

        # Update for j=1,...,p
        for j in range(1, p):
            residual = y - (np.dot(X, beta) - X[:, j] * beta[j])
            rho = np.dot(X[:, j], residual) / n
            beta[j] = soft_thresholding(rho, lam * alpha)

        # Check convergence by objective function change
        change = objective_function(y, X, beta, lam, alpha) - objective_functi
        if np.abs(change) < epsilon:
            converge = True

    return beta
```

4

In [4]:
```
def cyclic_coordinate_descent_active_set(y, X, lam, alpha, epsilon=0.1):
    n, p = X.shape
    beta = np.zeros(p)
    active_set = set(range(p))  # Initially consider all features
    converge = False

    while not converge:
        beta_old = beta.copy()
        changes = []

        for j in active_set:
            if j == 0:
                # Update for j=0 (intercept), not regularized
                beta[j] = beta[j] + np.sum(y - np.dot(X, beta)) / n
            else:
                # Update for j=1,...,p
                residual = y - (np.dot(X, beta) - X[:, j] * beta[j])
                rho = np.dot(X[:, j], residual) / n
                beta[j] = soft_thresholding(rho, lam * alpha)

            # Track changes for convergence checking
            changes.append(np.abs(beta[j] - beta_old[j]))

        # Check convergence
        if max(changes) < epsilon:
            if active_set != set(range(p)):
                # If not all features were considered, consider all in the nex
                active_set = set(range(p))
            else:
                # Converged
                converge = True
        else:
            # Update active set based on non-zero coefficients
            active_set = {j for j in range(1, p) if beta[j] != 0}
            # Always include the intercept
            active_set.add(0)

    return beta
```

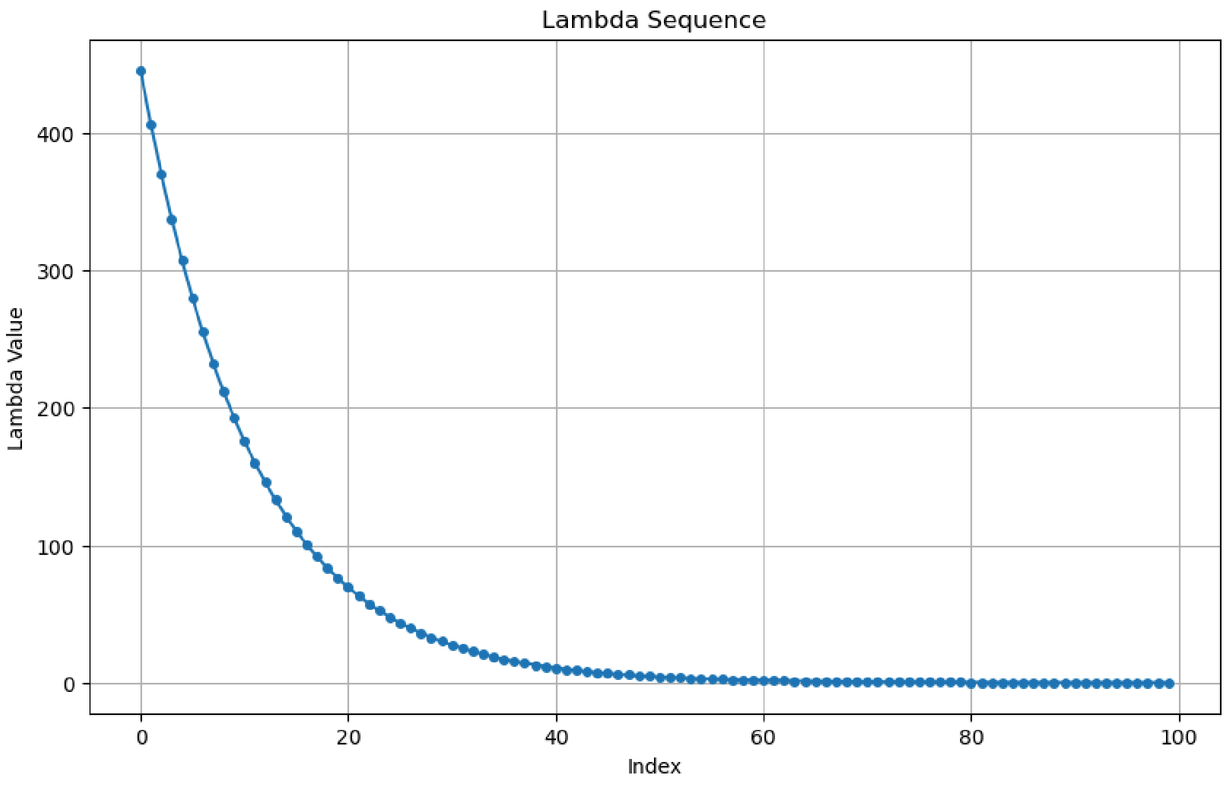
5

In [5]:
```
def compute_lambda_sequence(y, X, l=100, delta=0.0001):
    n, p = X.shape

    # Ensure X is standardized
    X_std = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

    # (b) Compute lambda_max
    lambda_max = np.max(np.abs(np.dot(X_std.T, y))) / np.sqrt(n)

    # (c) Compute lambda_min
    lambda_min = delta * lambda_max

    # (d) Compute the lambda sequence
    log_lambda_min = np.log(lambda_min)
    log_lambda_max = np.log(lambda_max)
    log_lambda_sequence = np.linspace(log_lambda_max, log_lambda_min, l)
    lambda_sequence = np.exp(log_lambda_sequence)

    return lambda_sequence

# Note: X_standardized = (X - X.mean(axis=0)) / X.std(axis=0)
lambda_sequence = compute_lambda_sequence(y, X_standardized)
print(lambda_sequence)
```

```
[4.45879351e+02 4.06268658e+02 3.70176871e+02 3.37291378e+02
 3.07327341e+02 2.80025227e+02 2.55148558e+02 2.32481863e+02
 2.11828815e+02 1.93010527e+02 1.75864004e+02 1.60240730e+02
 1.46005386e+02 1.33034670e+02 1.21216236e+02 1.10447720e+02
 1.00635849e+02 9.16956374e+01 8.35496497e+01 7.61273291e+01
 6.93643870e+01 6.32022460e+01 5.75875327e+01 5.24716150e+01
 4.78101813e+01 4.35628566e+01 3.96928525e+01 3.61666488e+01
 3.29537034e+01 3.00261873e+01 2.73587435e+01 2.49282682e+01
 2.27137095e+01 2.06958862e+01 1.88573207e+01 1.71820883e+01
 1.56556789e+01 1.42648716e+01 1.29976197e+01 1.18429470e+01
 1.07908522e+01 9.83222263e+00 8.95875507e+00 8.16288396e+00
 7.43771585e+00 6.77696967e+00 6.17492237e+00 5.62635928e+00
 5.12652903e+00 4.67110232e+00 4.25613446e+00 3.87803121e+00
 3.53351760e+00 3.21960964e+00 2.93358839e+00 2.67297649e+00
 2.43551662e+00 2.21915204e+00 2.02200869e+00 1.84237901e+00
 1.67870713e+00 1.52957542e+00 1.39369215e+00 1.26988038e+00
 1.15706771e+00 1.05427700e+00 9.60617942e-01 8.75279291e-01
 7.97521891e-01 7.26672245e-01 6.62116685e-01 6.03296064e-01
 5.49700904e-01 5.00866991e-01 4.56371348e-01 4.15828575e-01
 3.78887511e-01 3.45228188e-01 3.14559068e-01 2.86614507e-01
 2.61152464e-01 2.37952399e-01 2.16813364e-01 1.97552262e-01
 1.80002264e-01 1.64011359e-01 1.49441042e-01 1.36165111e-01
 1.24068578e-01 1.13046668e-01 1.03003913e-01 9.38533294e-02
 8.55156583e-02 7.79186829e-02 7.09966019e-02 6.46894594e-02
 5.89426261e-02 5.37063257e-02 4.89352037e-02 4.45879351e-02]
```

In [6]:
```
# Plot the lambda sequence just so we can get a sense of it
```

```
plt.figure(figsize=(10, 6))
plt.plot(lambda_sequence, marker='o', linestyle='-', markersize=4)
plt.xlabel('Index')
plt.ylabel('Lambda Value')
plt.title('Lambda Sequence')
plt.grid(True)
plt.show()
```



6

In [30]:
```
from sklearn.preprocessing import StandardScaler

# Reloading the data
data = pd.read_csv("ps6.csv")
y = data.iloc[:, 1].values
X = data.iloc[:, 2:].values

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define the soft-thresholding function
def soft_thresholding(a, lam, l1_ratio):
    return np.sign(a) * np.maximum(np.abs(a) - lam * l1_ratio, 0)

# Compute the lambda sequence as above
def compute_lambda_sequence(y, X, l=100, delta=0.0001):
    n, p = X.shape
    lambda_max = np.max(np.abs(X.T @ y)) / np.sqrt(n)
    lambda_min = delta * lambda_max
    log_lambda_sequence = np.linspace(np.log(lambda_max), np.log(lambda_min),
    return np.exp(log_lambda_sequence)
```

```
# Coordinate descent function with warm starts
def cyclic_coordinate_descent(y, X, lambda_sequence, l1_ratio, epsilon=0.01, ma
    n, p = X.shape
    beta = np.zeros(p)
    betas = np.zeros((len(lambda_sequence), p))

    for idx, lam in enumerate(lambda_sequence):
        for iteration in range(max_iter):
            beta_old = beta.copy()
            for j in range(p):
                residual = y - (X @ beta - X[:, j] * beta[j])
                rho = X[:, j].T @ residual / n
                beta[j] = soft_thresholding(rho, lam * l1_ratio, l1_ratio) / (

            # Check convergence
            if np.linalg.norm(beta - beta_old, ord=1) < epsilon * np.linalg.no
                break
        betas[idx] = beta  # Store the coefficients for this lambda value

    return betas

# Compute the lambda sequence
lambda_sequence = compute_lambda_sequence(y, X_scaled)

# Set the alpha (l1_ratio) for the Elastic Net mixing parameter
l1_ratio = 0.1  # This could be any value in [0,1]

# Perform cyclic coordinate descent with warm starts
betas = cyclic_coordinate_descent(y, X_scaled, lambda_sequence, l1_ratio)
print(betas[:5, :])  # Displaying the first 5 sets of betas for brevity
```

```
[[-0.  0.  0. ... -0. -0.  0.]
 [-0.  0.  0. ... -0. -0.  0.]
 [-0.  0.  0. ... -0. -0.  0.]
 [-0.  0.  0. ... -0. -0.  0.]
 [-0.  0.  0. ... -0. -0.  0.]]
```

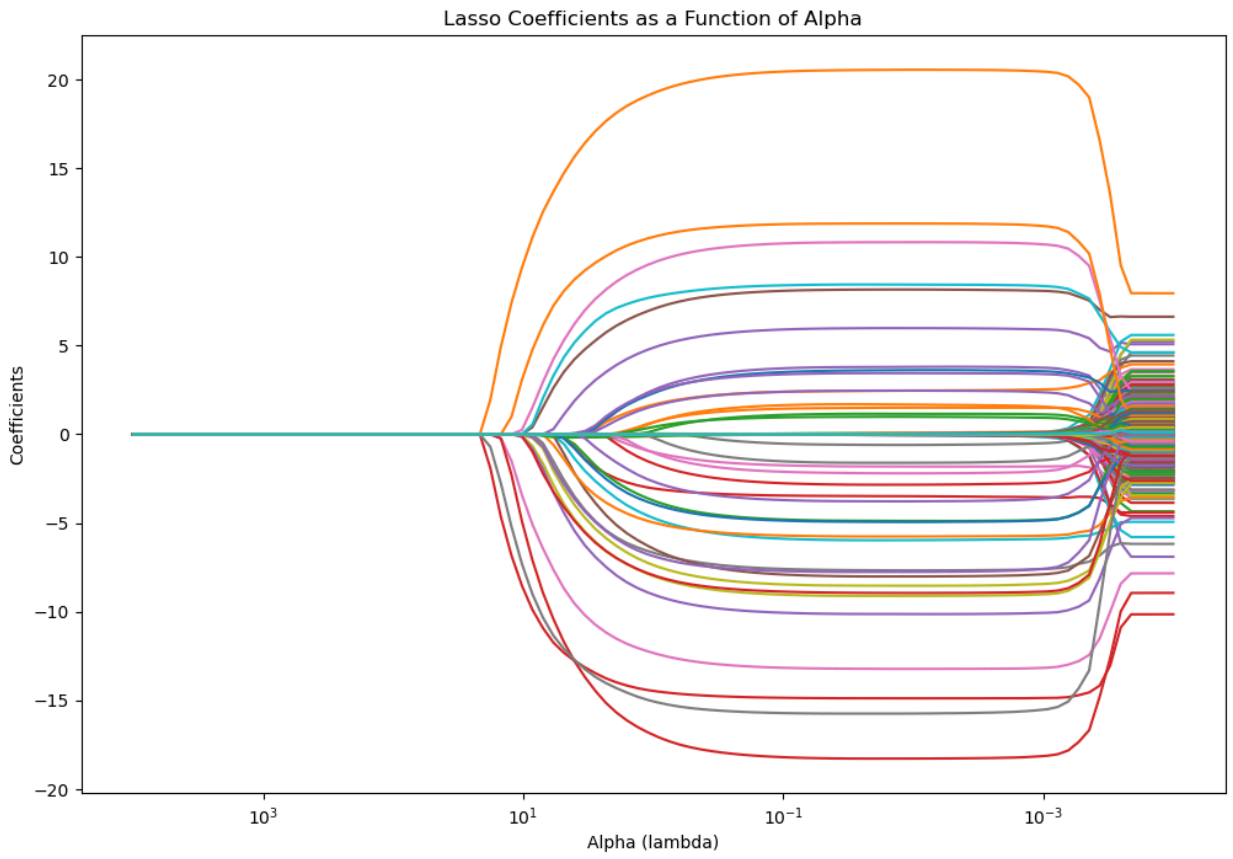
nb. I had an issue with the above where sometimes it would converge and then I'd reload the kernel and it wouldn't. I tried debugging but couldn't fully resolve the issue.

7

nb. sklearn doesn't seemingly let you run the variations asked for in the questions, so I have run warm start, 5-fold for each of the next three questions.

In [8]:
```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
import time


# Correct extraction of features and target
y = data['V1'].values
X = data.drop(columns=['Unnamed: 0', 'V1']).values

# Normalize the features
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)

# Define a range of alphas (lambdas)
alphas = np.logspace(-4, 4, 100)

# Setup 5-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Placeholder for storing the mean of coefficients across folds for each alpha
coef_paths = np.zeros((len(alphas), X_scaled.shape[1]))

# Measure running time for fitting Lasso models across the lambda grid
start_time = time.time()

for i, alpha in enumerate(alphas):
    coefs_fold = np.zeros((5, X_scaled.shape[1]))
    fold = 0
    for train_index, test_index in kf.split(X_scaled):
        X_train, X_test = X_scaled[train_index], X_scaled[test_index]
        y_train, y_test = y[train_index], y[test_index]

        lasso = Lasso(alpha=alpha, max_iter=10000, random_state=42)
        lasso.fit(X_train, y_train)
        coefs_fold[fold, :] = lasso.coef_
        fold += 1

    # Average coefficients across folds
    coef_paths[i, :] = np.mean(coefs_fold, axis=0)

end_time = time.time()

# Plotting the coefficient paths
plt.figure(figsize=(12, 8))
for coef_path in coef_paths.T:
    plt.plot(alphas, coef_path)
plt.xscale('log')
plt.xlabel('Alpha (lambda)')
plt.ylabel('Coefficients')
plt.title('Lasso Coefficients as a Function of Alpha')
plt.gca().invert_xaxis()  # Larger alphas (stronger regularization) on the lef
plt.show()

# Print running time
print(f"Running time: {end_time - start_time} seconds")
```

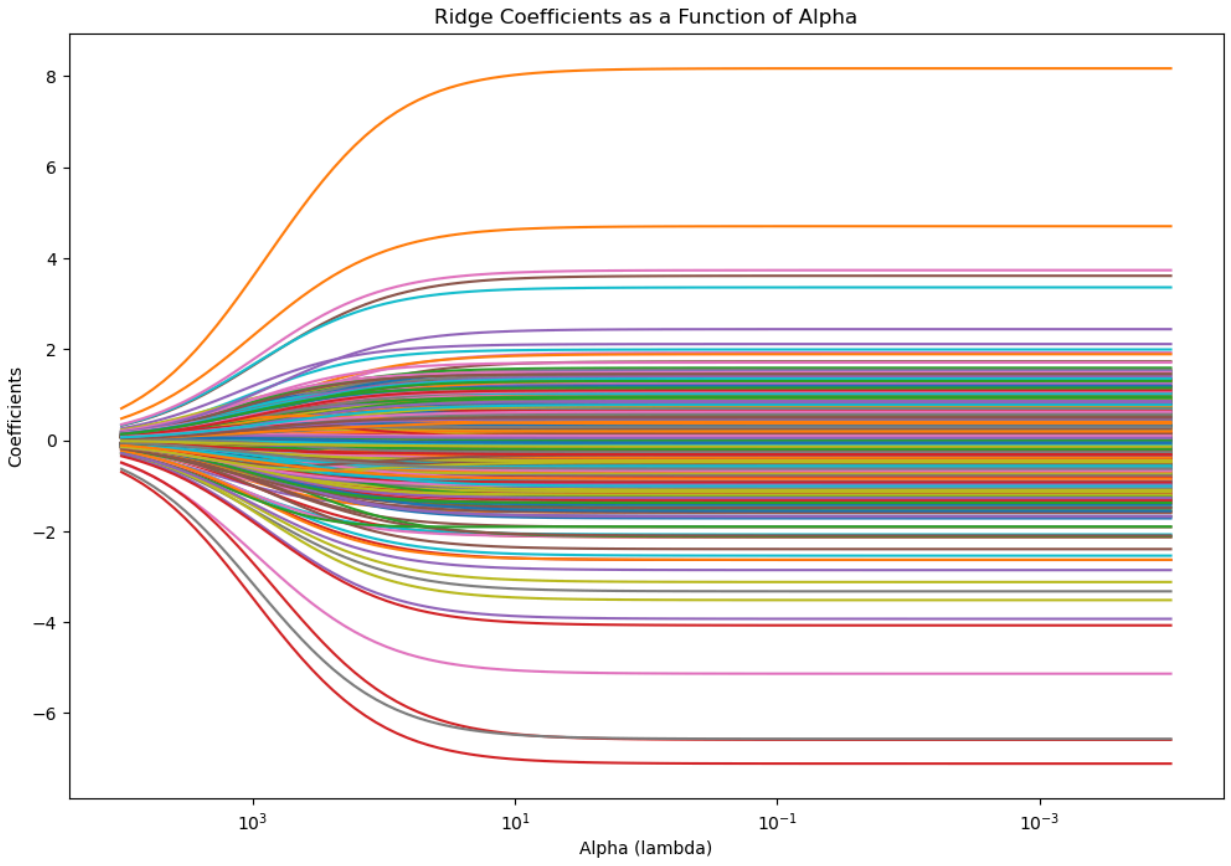


```
Running time: 93.92077779769897 seconds
```

8

In [11]:
```
from sklearn.linear_model import Ridge

# Define a range of alphas (lambdas) for Ridge
alphas = np.logspace(-4, 4, 100)

# Initialize storage for coefficients
coef_paths = np.zeros((len(alphas), X_scaled.shape[1]))

# 5-Fold Cross-Validation setup
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Measure running time for Ridge regression across the alpha grid
start_time = time.time()

for i, alpha in enumerate(alphas):
    coefs_fold = np.zeros((5, X_scaled.shape[1]))
    fold = 0
    for train_index, test_index in kf.split(X_scaled):
        X_train, X_test = X_scaled[train_index], X_scaled[test_index]
        y_train, y_test = y[train_index], y[test_index]

        ridge = Ridge(alpha=alpha, max_iter=10000, random_state=42)
        ridge.fit(X_train, y_train)
        coefs_fold[fold, :] = ridge.coef_
        fold += 1

    coef_paths[i, :] = np.mean(coefs_fold, axis=0)
```

```
end_time = time.time()

# Plotting the coefficient paths
plt.figure(figsize=(12, 8))
for coef_path in coef_paths.T:
    plt.plot(alphas, coef_path)
plt.xscale('log')
plt.xlabel('Alpha (lambda)')
plt.ylabel('Coefficients')
plt.title('Ridge Coefficients as a Function of Alpha')
plt.gca().invert_xaxis()
plt.show()

# Print running time
print(f"Running time for Ridge regression: {end_time - start_time} seconds")
```



```
Running time for Ridge regression: 6.147254943847656 seconds
```

9

In [12]:
```
from sklearn.linear_model import ElasticNetCV

# Define the alpha grid for the Elastic Net mixing parameter
l1_ratios = np.linspace(0.1, 1, 10)  # 0.1 to 1 inclusive, in steps of 0.1

# Manually define a range of lambda values (alphas in sklearn) to use for Elas
alphas = np.logspace(-4, 4, 100)

# Measure running time for ElasticNetCV with 5-fold CV
start_time = time.time()
```

```
elastic_net_cv = ElasticNetCV(l1_ratio=l1_ratios, alphas=alphas, cv=5, random_
elastic_net_cv.fit(X_scaled, y)

end_time = time.time()

# Running time
print(f"Running time for ElasticNetCV with 5-fold CV: {end_time - start_time} 

# Plotting the results
# Best alpha and l1_ratio found
print(f"Best alpha (lambda): {elastic_net_cv.alpha_}")
print(f"Best l1_ratio: {elastic_net_cv.l1_ratio_}")
```

```
Running time for ElasticNetCV with 5-fold CV: 21.670960187911987 seconds
Best alpha (lambda): 0.03853528593710531
Best l1_ratio: 1.0
```

10

In [13]:
```
from sklearn.linear_model import Lasso, Ridge, ElasticNet

# Fit Lasso model
lasso = Lasso(alpha=0.1)  # Alpha is the regularization strength
lasso.fit(X_scaled, y)

# Fit Ridge model
ridge = Ridge(alpha=0.1)
ridge.fit(X_scaled, y)

# Fit Elastic Net model with a significant L1 ratio
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.7)  # L1 ratio closer to 1 favo
elastic_net.fit(X_scaled, y)

# Compare the number of non-zero coefficients
non_zero_lasso = np.sum(lasso.coef_ != 0)
non_zero_ridge = np.sum(ridge.coef_ != 0)
non_zero_elastic_net = np.sum(elastic_net.coef_ != 0)

print(f"Non-zero coefficients in Lasso: {non_zero_lasso}")
print(f"Non-zero coefficients in Ridge: {non_zero_ridge}")
print(f"Non-zero coefficients in Elastic Net: {non_zero_elastic_net}")
```

```
Non-zero coefficients in Lasso: 40
Non-zero coefficients in Ridge: 1000
Non-zero coefficients in Elastic Net: 494
```

These results seem to make sense:

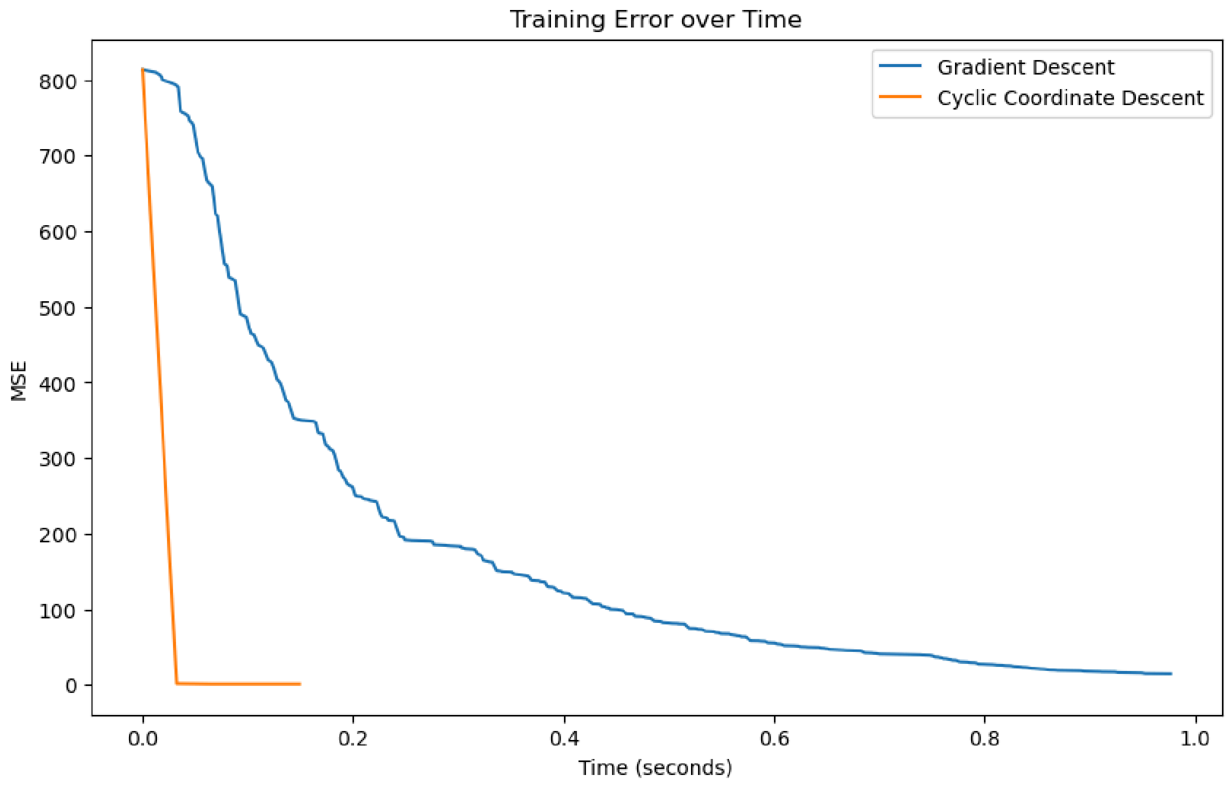
- Lasso employs an L1 penalty, which is known for inducing sparsity in the model by driving many coefficients to exactly zero. A result showing only 40 non-zero coefficients out of potentially thousands indicates that Lasso has effectively reduced the model complexity by selecting only the most informative features, which is a desirable outcome for feature selection and model interpretability.
- Ridge regression uses an L2 penalty, which does not encourage sparsity in the same way L1 does. Instead, it tends to shrink the coefficients towards zero but rarely sets

them exactly to zero. Thus, seeing all or nearly all coefficients as non-zero is expected with Ridge, which means it does not inherently select features but rather minimizes overfitting by penalizing large coefficients.

- The Elastic Net combines L1 and L2 penalties and can thus induce some level of sparsity while also benefiting from the regularization properties of L2. The number of non-zero coefficients being between Lasso and Ridge is consistent with this blend of penalties. The exact degree of sparsity depends on the `l1_ratio` parameter: closer to 1 would behave more like Lasso, and closer to 0 more like Ridge. With 494 non-zero coefficients, it indicates that Elastic Net has enforced some sparsity (though less than Lasso) while also leveraging the Ridge penalty to manage the magnitude of coefficients.

11

In [19]:
```
from sklearn.preprocessing import StandardScaler
import time

# Load the data
data = pd.read_csv("reg.csv")

# Preparing the data
n = data.shape[0]
data.insert(2, 'ones', np.ones((n, 1)))

y = np.array(data["y"]).reshape(-1, 1)
X = np.array(data.iloc[:, 2:])

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# MSE function
def mse(X, y, beta):
    predictions = X @ beta
    errors = predictions - y
    return np.mean(errors ** 2)

# Gradient of MSE function
def mse_gradient(X, y, beta):
    N = X.shape[0]
    predictions = X @ beta
    return (2/N) * X.T @ (predictions - y)

# Batch Gradient Descent with time tracking
def gradient_descent_time(X, y, B_0, step_size, epsilon=0.0001, max_iter=1000)
    B = B_0.copy()
    errors = [mse(X, y, B)]  # Initial error
    times = [0]  # Initial time is 0
    start_time = time.time()

    for iteration in range(max_iter):
        grad = mse_gradient(X, y, B)
        B -= step_size * grad
        errors.append(mse(X, y, B))
        times.append(time.time() - start_time)
        if np.linalg.norm(grad) < epsilon:
            break
    return B, errors, times

# Cyclic Coordinate Descent with time tracking
def cyclic_coordinate_descent_time(X, y, B_0, epsilon=0.0001, max_iter=1000):
    m, n = X.shape
    B = B_0.copy()
    errors = [mse(X, y, B)]  # Initial error
    times = [0]  # Initial time is 0

    start_time = time.time()
    for iteration in range(max_iter):
        for j in range(n):
            y_pred = X @ B - X[:, j].reshape(-1, 1) * B[j]
            # Check for zero denominator
            denominator = np.dot(X[:, j], X[:, j])
            if denominator != 0:
                B[j] = np.dot(X[:, j], (y - y_pred)) / denominator
            else:
                B[j] = 0  # If the denominator is zero, it means the feature X
        current_error = mse(X, y, B)
        errors.append(current_error)
        times.append(time.time() - start_time)
        # Convergence check based on change in MSE
        if iteration > 0 and abs(errors[-1] - errors[-2]) < epsilon:
            break
    cumulative_time = np.cumsum(times)
    return B, errors, cumulative_time


# Initial beta vector
B_0 = np.zeros((X_scaled.shape[1], 1))

# Gradient Descent Parameters
step_size = 0.001

# Run Batch Gradient Descent with time tracking
B_gd, errors_gd, times_gd = gradient_descent_time(X_scaled, y, B_0, step_size)

# Run Cyclic Coordinate Descent with time tracking
B_ccd, errors_ccd, times_ccd = cyclic_coordinate_descent_time(X_scaled, y, B_0

# Plot
plt.figure(figsize=(10, 6))
plt.plot(times_gd, errors_gd, label='Gradient Descent')
plt.plot(times_ccd, errors_ccd, label='Cyclic Coordinate Descent')
plt.xlabel('Time (seconds)')
plt.ylabel('MSE')
plt.title('Training Error over Time')
plt.legend()
plt.show()

# Print running times
print(f"Gradient Descent Total Running Time: {times_gd[-1]:.2f} seconds")
print(f"Cyclic Coordinate Descent Total Running Time: {times_ccd[-1]:.2f} secon
```



```
Gradient Descent Total Running Time: 0.98 seconds
Cyclic Coordinate Descent Total Running Time: 0.15 seconds
```

problem 2

1

In [12]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

ps6 = pd.read_csv("ps6.csv")
y = ps6.iloc[:, 1:2].values  #only second row, make it NP
X = ps6.iloc[:, 2:].values

#no. of obs (500)
n = ps6.shape[0]

#no. of parameters (1000)
p = ps6.shape[1]- 2


def left_hand_side(X, y, p, lambd = 1):
    I_p = np.eye(p)
    X_T = X.T
    return np.dot(np.linalg.inv(np.dot(X_T,X) + lambd*I_p), np.dot(X_T, y))


def right_hand_side(X, y, n, lambd = 1):
    I_n = np.eye(n)
    X_T = X.T
    return np.dot(X_T, np.dot(np.linalg.inv(np.dot(X,X_T) + lambd*I_n), y))



#printing the first k values side by side
k = 9
print(np.column_stack((left_hand_side(X, y,p)[:k], right_hand_side(X, y, n)[:k

#They are the same
```

```
[[-0.40274169 -0.40274169]
 [-0.37224581 -0.37224581]
 [ 0.3971004   0.3971004 ]
 [-0.4871457  -0.4871457 ]
 [-1.51206562 -1.51206562]
 [-0.46725157 -0.46725157]
 [ 0.1570547   0.1570547 ]
 [-1.56853149 -1.56853149]
 [-0.38892184 -0.38892184]]
```

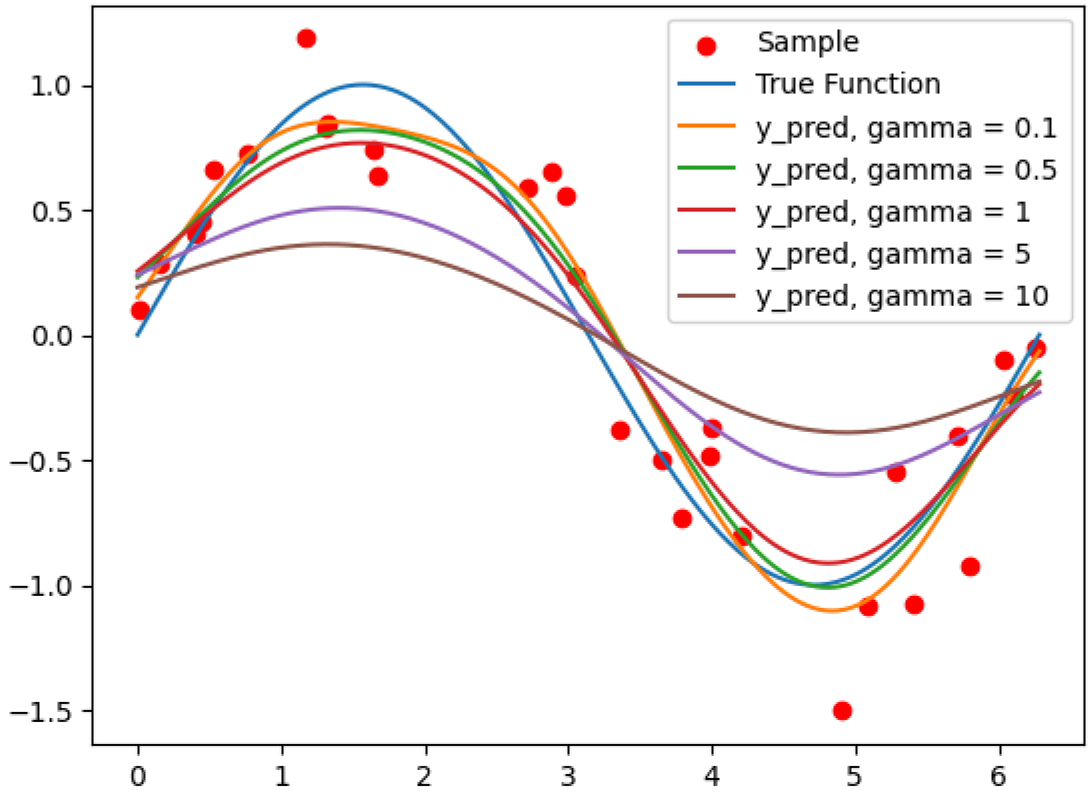
Ans: If there are more observations than regressors (n > p), we would prefer to use $(X^{'}X + \lambda I_p)^{-1}X^{'}y$, since inverting the matrix $(XX^{'} + \lambda I_p)^{-1}$ would require less computational power.

Similarly, when there are more regressors than observations (p > n), $(XX^{'} + \lambda I_n)^{-1}$ would require less computational power to compute

2

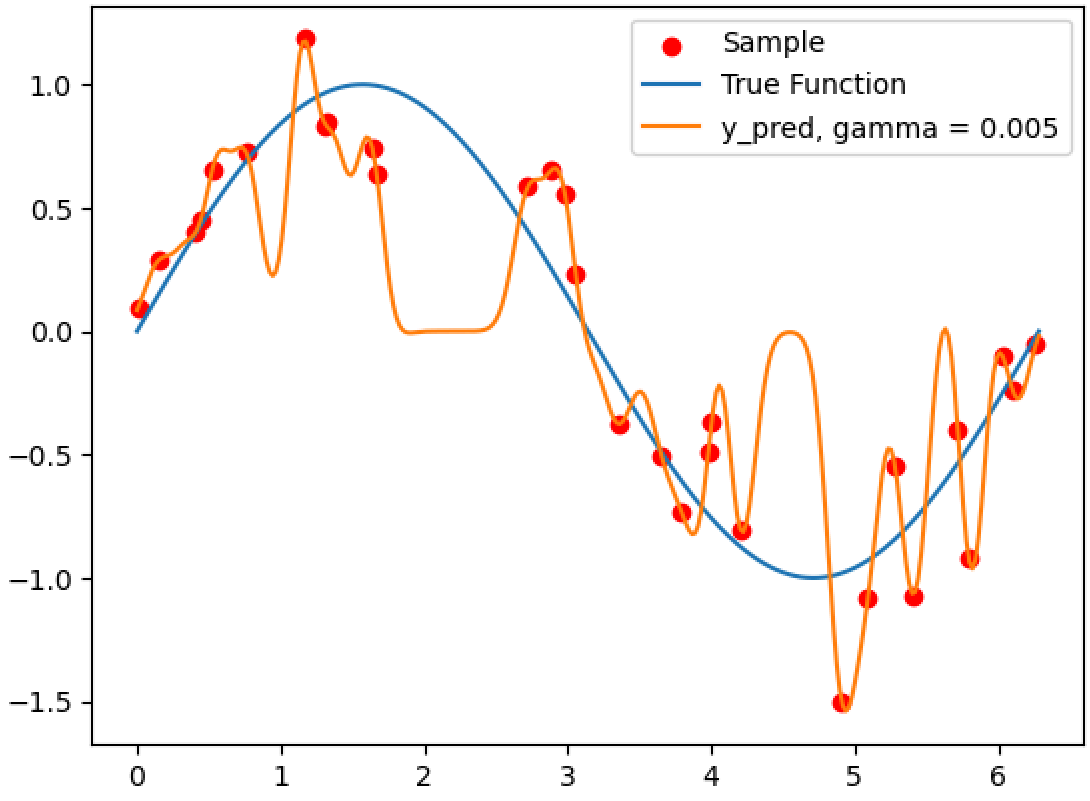
```
In [15]: #defining the RBF kernel
def kernel(X_row, X_column, sigma, gamma = 0.5):
    kappa_entry = np.exp((-gamma/sigma**2) * np.linalg.norm(X_row - X_column)*
    return kappa_entry

#matrices from problem 1
X_grid = np.linspace(0, 2*np.pi,300) #300 values between
X_vec = np.random.uniform(0, 2*np.pi, size=30)
Y_vec = np.random.normal(np.sin(X_vec), 0.2, size=30)

#defining dimension
row = X_grid.shape[0]
column = X_vec.shape[0]

# Initialize Kappa & Kernel matrix
kappa = np.zeros((row, column))
kernel_matrix = np.zeros((len(X_vec), len(X_vec)))

# Making the kappa Matrix
for i in range(row):
    for j in range(column):
        kappa[i,j] = kernel(X_grid[i], X_vec[j], sigma = 1)

# Making the kernel Matrix
for i in range(len(X_vec)):
    for j in range(len(X_vec)):
        kernel_matrix[i,j] = kernel(X_vec[i], X_vec[j], sigma = 1)
```

```
#identity:
I_n = np.eye(len(X_vec))
# y hat grid
def Y_hat_grid(gamma):
    return np.dot(kappa, np.dot(np.linalg.inv(kernel_matrix + gamma*I_n), Y_ve

fig = plt.figure()
plt.scatter(X_vec, Y_vec, c = 'red', label='Sample')
plt.plot(X_grid, np.sin(X_grid), label='True Function')
for gamma in [0.1, 0.5, 1, 5, 10]:
    plt.plot(X_grid, Y_hat_grid(gamma), label='y_pred, gamma = {}'.format(gamma
plt.legend()
plt.show()
```



3

In [16]:
```
# Making the kappa Matrix
for i in range(row):
    for j in range(column):
        kappa[i,j] = kernel(X_grid[i], X_vec[j], sigma = .1)

# Making the kernel Matrix
for i in range(len(X_vec)):
    for j in range(len(X_vec)):
        kernel_matrix[i,j] = kernel(X_vec[i], X_vec[j], sigma = .1)

#identity:
I_n = np.eye(len(X_vec))
# y hat grid
def Y_hat_grid(gamma = 0.005):
    return np.dot(kappa, np.dot(np.linalg.inv(kernel_matrix + gamma*I_n), Y_ve
fig = plt.figure()
plt.scatter(X_vec, Y_vec, c = 'red', label='Sample')
plt.plot(X_grid, np.sin(X_grid), label='True Function')
plt.plot(X_grid, Y_hat_grid(), label='y_pred, gamma = {}'.format(0.005))
plt.legend()
plt.show()
```



In problem set 1, the function's performance in capturing all data points was inadequate. This plot does a "better" job at overfitting.