Intro to ML: Homework 1

Carlos Gonzalez Rivera

September 13, 2023

1 Applied Data Analysis

Attached at the end of this document is the Jupyter Notebook that contains all the cells for the following two applied problems.

1.1 Problem 1:

- Why are the top features different for some methods?
  - In general, these different methods have different strategies and mathematical foundations for evaluating the importance of features. OLS, for instance, tends to use all available features, sometimes prioritizing noise or less significant features. Contrarily, Ridge Regression adds an L2 penalty to control for multicollinearity (even though it does not perform feature selection). On the other hand, Lasso and Elastic Net add an L1 penalty, which can shrink the coefficients of less important features to zero (virtually performing feature selection). Meanwhile, the "Best Subsets" method seeks to find the most predictive subset of features by evaluating all possible combinations, which can identify different essential features compared to other methods. Finally, RFE selects features by recursively removing the least important ones based on a model fit. In short, the different techniques have conclusively identified different sets of "top features" due to their distinct mathematical approaches and criteria for feature selection.
- If you were to tune parameters, how would you determine these?
  - Parameter tuning can be performed using techniques like grid or random search in conjunction with cross-validation. This approach helps systematically explore different parameter combinations to find the set that gives the best performance on a validation set. Parameters like the regularization strength in Ridge, Lasso, and Elastic Net or the number of features to select in Best Subsets and RFE would be the primary focus during fine-tuning. In synopsis, the best parameters would minimize the validation error, indicating an excellent generalization to unseen data.

- Would tuning other parameters yield additional vital features?
  - Tuning different parameters can highlight different sets of important features. For instance, a change in the regularization parameter in Lasso or Elastic Net can change the sparsity of the solution by either including more features or making the model more parsimonious (a less complex model with fewer parameters to be tuned). Similarly, adjusting the number of features in Best Subsets or RFE can lead to different subsets of features being selected and potentially unveiling new important features that were not highlighted with other settings.
- Are any features consistently selected by all methods?
  - The analysis has shown that features 3 and 5 have been consistently selected as significant across various methods due to their robust influence on the median value of owner-occupied homes. Additionally, features 4 (nitrogen oxide concentration) and 12 (percentage of the lower status of the population) also emerged as relatively significant contributors in the predictive modeling, showcasing their importance in the environmental and socio-economic contexts, respectively. Their consistent selection across different methods substantiates their role in the model's predictive accuracy, complementing the primary influence of features 3 and 5.
- What are the most critical features, and how did you determine this?
  - In the predictive modeling of the median value of owner-occupied homes using the Boston Housing dataset, a synergistic analysis employing various methods, including Best Subsets, Recursive Feature Elimination (RFE), Elastic Net, Lasso, Ridge, and OLS distinctly spotlighted features 3 and 5 as the most pivotal variables across the board. Feature 3, representing the Charles River dummy variable, indicates a substantial impact on housing prices, possibly attributed to the aesthetic vistas and the premium locality alongside the river. Concurrently, feature 5, denoting the average number of rooms per dwelling, naturally emerges as a significant determinant, where a greater number of rooms signifies more space, thus potentially escalating property prices. Moreover, features 4 (nitrogen oxides concentration) and 12 (percentage of lower population status) also emerged as noteworthy contributors to the model, albeit to a lesser extent than features 3 and 5. These features indicate environmental and socioeconomic factors, respectively, that significantly influence property valuations. Their consistent appearance across various methods underscores their secondary yet considerable role in shaping the model's predictive accuracy, thus warranting their inclusion for a more nuanced and holistic analysis. This collective insight forms a robust foundation for creating a well-rounded predictive model that encapsulates various influential factors.

- Which methods would hold more value over the other?
  - The choice of method significantly depends on the specific analytical context and the dataset's characteristics. In scenarios where a nuanced understanding of environmental and socio-economic impacts (like features 4 and 12) on housing prices is essential, methods that can effectively isolate and highlight the influence of these features would be more valuable. For instance, Lasso and Elastic Net offer more value in performing feature selection and spotlighting the importance of these features, compared to OLS, which does not inherently perform feature selection. Furthermore, Best Subsets and RFE can offer insights into the best combinations of these features for predictive modeling, helping construct a more nuanced and holistic model. Thus, the value of each method would be gauged based on its ability to effectively incorporate and analyze the influence of these critical features in the predictive modeling.

1.2 Problem 2:

Attached in the Jupyter Notebook at the end are the empirical demonstrations to the three tasks of Problem 2 as their ten corresponding Python cells.

2 Theory & Methods

2.1 Question 2: Ridge Regression Computation

The Ridge Regression problem can be formulated as solving the following optimization problem:

$$min_\beta(||Y - X\beta||_2^2 + \lambda||\beta||_2^2) \tag{1}$$

Where:

- Y is the n × *1* response vector (target variable).
- X is the n × p design matrix (input features' matrix).
- β is the p × *1* coefficient vector.
- λ is the regularization parameter (shrinkage parameter).
- $||.||$ denotes the Euclidean norm.

2.1.1 When $n > p$, the computational complexity is $O(np^2)$

An efficient solution when $n > p$ uses the normal equation for ridge regression and solves for β:

$$(X^TX + \lambda I)\beta = X^TY$$

$$\beta = X^TY(X^TX + \lambda I)^{-1} = \frac{X^TY}{X^TX + \lambda I}$$

The computational complexity of the normal equation in Ridge Regression is determined by the inversion of the $p \times p$ matrix, which is $O(p^3)$. When $n > p$, the matrix multiplication of X^T (a $p \times n$ matrix) with X (a $n \times p$ matrix) would take $O(np^2)$, making the overall complexity $O(np^2 + p^3)$. Therefore, np^2 is the dominant term in this case since $n > p$.

2.1.2 When $p > n$, the computational complexity is $O(n^2p)$

The Woodbury Matrix Identity is a more efficient approach to solve for ridge regression's β when $p > n$. The Woodbury Matrix Identity is given by:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{2}$$

Where:

- $A = \lambda I$, I is an identity matrix of size $p \times p$
- $U = X^T$
- $V = X$
- $C = I_n$, I_n is an identity matrix of size $n \times n$

In other words,

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{3}$$

$$(\lambda I + X^T X I_n)^{-1} = \lambda^{-1}I - \lambda^{-1}IX^T(I_n^{-1} + X^T X\lambda^{-1}I)^{-1}X\lambda^{-1}I \tag{4}$$

$$(\lambda I + X^T X)^{-1} = \lambda^{-1}I - \lambda^{-1}X^T(I_n^{-1} + X^T X\lambda^{-1})^{-1}X\lambda^{-1} \tag{5}$$

We can then multiply this result of the Woodbury Matrix Identity by X^TY to find β:

$$\beta = (\lambda^{-1}I - \lambda^{-1}X^T(I_n^{-1} + X^T X\lambda^{-1})^{-1}X\lambda^{-1})X^TY \tag{6}$$

The Woodbury Identity helps in reducing the complexity by avoiding the inversion of a large $p \times p$ matrix. The dominant terms in the complexity are the inversion of the $n \times n$ matrix (which has a complexity of $O(n^3)$) and the multiplication operations between X and X^T, which gives an overall complexity of $O(n^3 + n^2p)$.Therefore, the complexity can be approximated to $O(n^2p)$ since n^2p will be the dominant term given $p > n$.

2.1.3 Empirical Performance Measurements

Attached in the Jupyter Notebook at the end are the empirical performance measurements to the computation of ridge regression as their five corresponding Python cells.

2.2 Question 3: Ridge Regression Property

2.2.1 Given Information

- Y follows a normal distribution with mean $X\beta$ and variance $\sigma^2 I$, denoted as $Y \sim N(X\beta, \sigma^2 I)$.
- β follows a normal distribution with mean 0 and variance τ^2, denoted as $\beta \sim N(0, \tau^2)$.

2.2.2 Formulation of Bayes' Theorem

Using Bayesian statistics, the posterior distribution is equal to the product of the likelihood function and the prior distribution of β:

$$p(\beta|Y, X) = \frac{(p(Y|X, \beta)p(\beta)}{p(Y|X)} \tag{7}$$

Since the $p(Y, X)$ term (known as the marginal likelihood) is a normalizing constant that ensures that the posterior distribution integrates to 1, it does not depend on β, and, therefore, does not affect the location of the mode or mean of the posterior distribution because it merely acts as a scaling factor to ensure the posterior distribution is a valid probability distribution.

The likelihood function, $p(Y|X, \beta) = Y \sim N(X\beta, \sigma^2 I)$, can be rewritten as:

$$p(Y|X, \beta) = \frac{1}{(2\pi\sigma^2)^{n/2}} exp\left(-\frac{1}{2\sigma^2}(Y - X\beta)^T (Y - X\beta)\right) \tag{8}$$

And, given $p(\beta) = \beta \sim N(0, \tau^2 I)$, the prior distribution as:

$$p(\beta) = \frac{1}{(2\pi\tau^2)^{p/2}} exp\left(-\frac{1}{2\tau^2}\beta^T \beta\right) \tag{9}$$

2.2.3 Calculate the Posterior Distribution

Since removing both expressions' denominators, which are part of the Normal distribution formulations, these simplified two expressions are multiplied to render the unnormalized posterior distribution:

$$p(\beta|Y) = exp\left(-\frac{1}{2\sigma^2}(Y - X\beta)^T (Y - X\beta) - \frac{1}{2\tau^2}\beta^T \beta\right) \tag{10}$$

The posterior distribution must be maximized to find its posterior mode (MAP Estimator). This is equivalent to minimizing the negative log of the posterior distribution with respect to β, as such:

$$-\log p(Y|\beta) = \frac{1}{2\sigma^2}(Y - X\beta)^T(Y - X\beta) + \frac{1}{2\tau^2}\beta^T\beta + C \tag{11}$$

$$L(\beta) = \frac{1}{2\sigma^2}(Y - X\beta)^T(Y - X\beta) + \frac{1}{2\tau^2}\beta^T\beta + C \tag{12}$$

$$\frac{\partial L(\beta)}{\partial \beta} = \frac{1}{\sigma^2}(X^TX\beta - X^TY) + \frac{1}{\tau^2}\beta = 0 \tag{13}$$

$$\frac{1}{\sigma^2}X^TY = \frac{1}{\sigma^2}X^TX\beta + \frac{1}{\tau^2}\beta \tag{14}$$

$$X^TY = \beta(X^TX + \frac{\sigma^2}{\tau^2}I) \tag{15}$$

$$\beta = \beta_{MAP} = X^TY\left(X^TX + +\frac{\sigma^2}{\tau^2}I\right)^{-1} \tag{16}$$

This minimized expression with respect to β is the MAP estimator and the posterior mean because the posterior mean in a Normal distribution equals the mode.

2.2.4 Relation to Ridge Regression

The expression to be minimized is similar to the ridge regression cost function with a ridge parameter $\lambda = \sigma^2/\tau^2$, given by:

$$\hat{\beta}_{ridge} = argmin_\beta ||Y - X\beta||^2 + \lambda||\beta||^2 = (X^TX + \lambda I)^{-1}X^TY \tag{17}$$

In short, the ridge regression solution is equivalent to the MAP estimator of β in a Bayesian framework with a normal prior on β. Taking the derivative of $\hat{\beta}_{ridge}$ and setting it to zero, the optimal β that minimizes the same cost function (equivalent to finding the mode of the posterior distribution):

$$-2X^T(Y - X\beta) + 2\lambda\beta = 0 \tag{18}$$

$$-2X^TX\beta - 2\lambda\beta = -2X^TY \tag{19}$$

$$\beta(X^TX + \lambda I) = X^TY \tag{20}$$

$$\beta_{MAP} = X^TY(X^TX + \lambda I)^{-1} = E(\beta|Y, X) \tag{21}$$

2.3 Question 4: Ridge Regression Property

2.3.1 Given Information

Using singular value decomposition (SVD), the ridge estimator and ridge predictor, $\hat{Y}$, can be derived with the following equation for ridge regression by decomposing any matrix X into the product of three other matrices $X = UDV^T$:

- U is an orthogonal matrix containing the left singular vectors of X.
- D is a diagonal matrix containing the singular values of X.
- V is an orthogonal matrix containing the right singular vectors of X.

2.3.2 Ridge Estimator Derivation using SVD

Due to D being a diagonal matrix with non-negative real numbers, its transpose is itself ($D^T = D$). The SVD of X^T can be derived from the SVD of X as such:

$$X^T = (UDV^T)^T = VD^TU^T = VDU^T \tag{22}$$

Therefore, the X^TX term can be rewritten using the SVD components as:

$$X^TX = (VD^TU^T)(UDV^T) = VD^2V^T \tag{23}$$

Substituting the SVD decomposition of X and X^T into the ridge regression equation to find an expression for β returns:

$$\beta_{Ridge} = argmin_{\beta}(||Y - X\beta||_2^2 + \lambda||\beta||_2^2) \tag{24}$$
$$L(\beta) = (Y - X\beta)^T(Y - X\beta) + \lambda\beta^T\beta \tag{25}$$
$$L(\beta) = Y^TY + X^T\beta^TX\beta - X^T\beta^TY - X\beta Y^T + \lambda\beta^T\beta \tag{26}$$

Now, deriving the Ridge regression's cost function, setting it to 0, substituting the SVD of X, and solving for β will return the ridge estimator $\hat{\beta}$:

$$\frac{\partial}{\partial\beta}L(\beta) = 2X^TX\beta - 2X^TY + 2\lambda\beta = 0 \tag{27}$$
$$2VD^2V^T\beta - 2VDU^TY + 2\lambda\beta = 0 \tag{28}$$
$$\beta(VD^2V^T + \lambda I) = VDU^TY \tag{29}$$
$$\beta = (VD^2V^T + \lambda I)^{-1}VDU^TY \tag{30}$$

2.3.3 Ridge Predictor Derivation using SVD

Using the derived β and the decomposed X, the ridge prediction ($\hat{Y} = X\beta$) can be rewritten as:

$$\hat{Y} = X\beta \tag{31}$$
$$\hat{Y} = (UDV^T)(VD^2V^T + \lambda I)^{-1}VDU^TY \tag{32}$$

2.3.4 Role of SVD Components in the Behavior of Ridge Regression

- The U matrices (Left Singular Vectors) contain the orthogonal basis vectors for the column space of their X. In short, it captures the patterns of variation within the features in X and the influence of U would be more pronounced when there is a substantial variation in the features.

- D matrices contain the singular values, which essentially capture the strength or magnitude of each component identified by U and V. In the context of ridge regression, larger singular values (or larger components of D) would be shrunken less by the ridge penalty compared to smaller singular values. This means that features associated with larger singular values will have a more significant influence on the prediction.

- V matrices contain the orthogonal basis vectors for the row space of their X matrix. V capture the patterns in how the different features relate to each other. The ridge regression would particularly influence when we have highly correlated groups of features; the penalty term would shrink the coefficients of these correlated features towards each other, helping in mitigating multicollinearity issues.

Ridge regression is particularly effective in dealing with multicollinearity issues. When groups of features are highly correlated, the ridge penalty helps in distributing the coefficient estimates among the correlated features more evenly, thereby preventing any single feature from receiving too much weight. Attached in the Jupyter Notebook at the end are the demonstrations to this solution by ranging predictions of different λ values.

2.4 Question 5: Lasso Regression Property

Recallling that Lasso regression, or Least Absolute Shrinkage and Selection Operator (LASSO), is defined by the following optimization problem:

$$\min_{\beta}\left\{\frac{1}{2N}\sum(y_i - x_i^T\beta)^2 + \lambda||\beta||_1\right\} \tag{33}$$

Where:

- N is the number of observations.
- x_i is the vector of predictors for observation i.
- y_i is the response for observation i.
- β is the coefficient vector
- $||.||_1$ denotes the L1 norm of β.
- λ is the regularization parameter.

Now, the loss function must be differentiated (compute gradient) with respect to β and then set to 0 to find the condition where the smallest value of λ, λ_{max}, makes all coefficients β equal to zero:

$$L(\beta) = \frac{1}{2N}||y - X\beta||_2^2 + \lambda||\beta||_1 \tag{34}$$

$$0 = -\frac{1}{N}X^T(y - X\beta) + \lambda\nabla||\beta||_1 \tag{35}$$

The differential of the L1 norm ($||\beta||_1$) is given by $\nabla||\beta||_1$ as sign(β). Consequently, by finding $\beta = 0$, Lasso's differentiated loss function simplifies to:

$$0 = -\frac{1}{N}X^T(y - X\beta) + \lambda sign(\beta) \tag{36}$$

$$0 = -\frac{1}{N}X^Ty + \lambda * 0 \tag{37}$$

Therefore, y_{max} is defined as: $y_{max} = \frac{1}{N}||X^T y||_\infty$

In this context, applying the infinity norm (or maximum absolute row sum norm) to the vector $X^T y$ gives the maximum absolute value of the entries in the vector, which identifies the maximum correlation between the predictors and the response variable, while $1/N$ is a normalization term where N represents the number of observations and ensures that y_{max} is scale-invariant with respect to its N.

2.5 Question 6: Lasso Regression Computation

The Elastic Net penalty is a regularized regression method that linearly combines the L1 and L2 penalties of the lasso and ridge methods. The penalty function, as given in the question, is defined as:

$$P(\beta) = \alpha||\beta||_1 + (1-\alpha)||\beta||_2^2 \tag{38}$$

To derive an algorithm to solve the elastic net regression problem using the proximal gradient or ADMM, we need to start by setting up the optimization problem. The full objective function to minimize can be defined as:

$$L(\beta) = \frac{1}{2}||y - X\beta||_2^2 + \alpha||\beta||_1 + (1-\alpha)||\beta||_2^2 \tag{39}$$

2.5.1 Using the Proximal Gradient Method:

First, we find the gradient of the smooth part of the loss function, which is $\left(\frac{1}{2}||y - X\beta||_2^2 + (1-\alpha)||\beta||_2^2\right)$. The gradient with respect to β is given by:

$$\nabla L(\beta) = -X^T(y - X\beta) + (1-\alpha)\beta \tag{40}$$

The proximal operator associated with the $||\beta||_1$ penalty for the l_1 norm, also known as the soft-thresholding operator, is defined as:

$$prox_{\alpha\lambda}(\beta) = sign(\beta)(|\beta| - \alpha\lambda)_+ \tag{41}$$

Using the proximal gradient method (where t_k is the step size at iteration k), the update rule at each new iteration k is given by:

$$\beta^{(k+1)} = prox_{\alpha\lambda}\left(\beta^{(k)} - t_k \nabla L(\beta^{(k)})\right) \tag{42}$$

2.5.2 Empirical Performance Measurements

Attached in the Jupyter Notebook at the end are the empirical performance measurements of the model derived to solve the elastic net regression problem using the proximal gradient method as their eight corresponding Python cells.

Global Imports

In [1]:

```
import math
import time
from itertools import combinations

import pandas as pd
import numpy as np
import statsmodels.api as sm

from matplotlib import pyplot as plt
from scipy.stats import skew
from sklearn.datasets import fetch_california_housing, fetch_openml, make_regression#, load_boston
from sklearn.feature_selection import RFE
from sklearn.linear_model import ElasticNet, ElasticNetCV, enet_path, Lasso, LassoCV, lasso_path, LinearRegression, Ridge, RidgeCV
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Applied Data Analysis:

PROBLEM 1:

Load Datasets

In [2]:

```
# boston = load_boston()
# print(boston.data.shape)
```

In [3]:

```
ames = fetch_openml(name="house_prices", as_frame=True)

california = fetch_california_housing()
```

```
/Users/gonz495/miniconda3/lib/python3.10/site-packages/sklearn/datasets/_openml.py:1002: FutureWarning: The default value of `parser` will change from `'liac-arff'` to `'auto'` in 1.4. You can set `parser='auto'` to silence this warning. Therefore, an `ImportError` will be raised from 1.4 if the dataset is dense and pandas is not installed. Note that the pandas parser may return different data types. See the Notes Section in fetch_openml's API doc for details.
  warn(
```

In [4]:

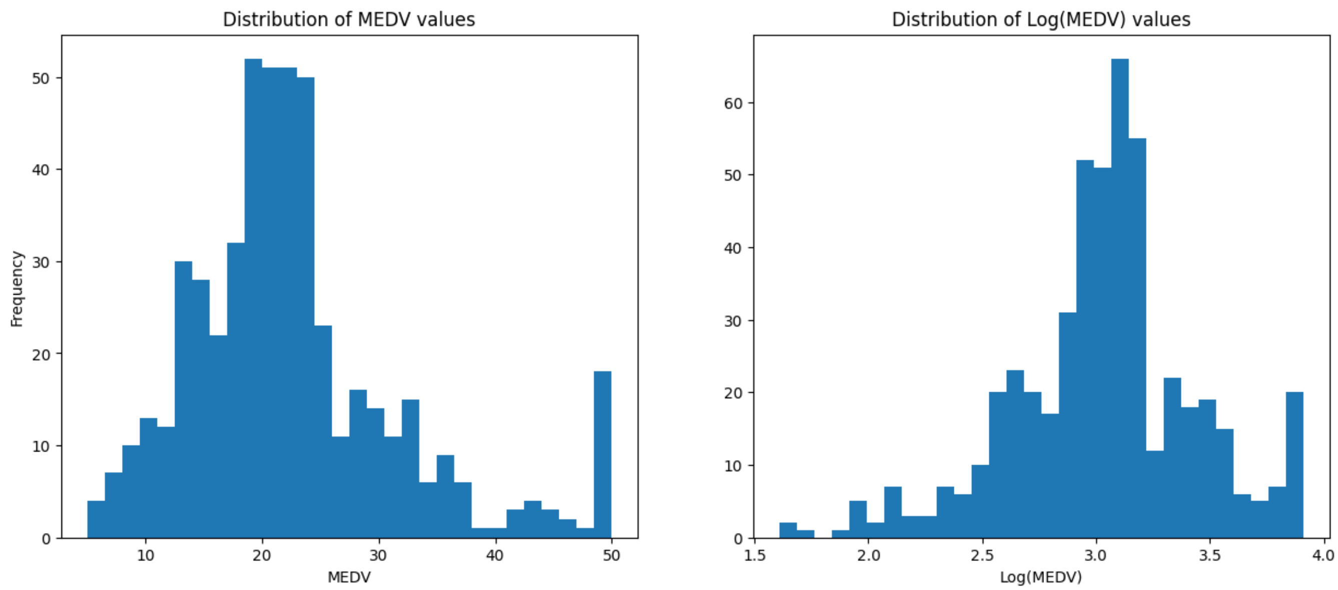
```
boston_df = pd.read_csv("http://lib.stat.cmu.edu/datasets/boston", sep="\s+", skiprows=22, header=None)

boston_data = np.hstack([boston_df.values[::2, :], boston_df.values[1::2, :2]])

boston_responses = boston_df.values[1::2, 2]

# np.savetxt("data.csv", boston_data, delimiter=",")
np.save("responses.npy", boston_responses)
np.save("data.npy", boston_data)
```

Data Visual

Data Visual

In [5]:

```
fig, axs = plt.subplots(1,2, figsize=(15,6))

axs[0].hist(boston_responses, bins=30)
axs[1].hist(np.log(boston_responses), bins=30)

axs[0].set_xlabel("MEDV")
axs[1].set_xlabel("Log(MEDV)")

axs[0].set_ylabel("Frequency")

axs[0].set_title("Distribution of MEDV values")
axs[1].set_title("Distribution of Log(MEDV) values")

plt.show()
```



Data Preprocessing

In [6]:

```
if np.abs(skew(boston_responses)) < np.abs(skew(np.log(boston_responses))):

    print(f"The original targets have less skewness (value of: {skew(boston_responses)}).
")

    rows_with_missing_values = np.any(np.isnan(np.hstack((boston_data, boston_responses.
reshape(-1, 1)))), axis=1)

    cleaned_data = np.hstack((boston_data, boston_responses.reshape(-1, 1)))[~rows_with_
missing_values]

    # print(np.any(np.isnan(np.hstack((boston_data, boston_responses.reshape(-1, 1))))))

    if np.any(np.isnan(np.hstack((boston_data, np.log(boston_responses).reshape(-1, 1)))
)):
        print("There are missing values in the data")
    else:
        print("There are no missing values in the data")

    print(f"Number of rows with missing values: {sum(rows_with_missing_values)}")

else:
    print(f"The targets' logarithms haves less skewness (value of: {skew(np.log(boston_re
sponses))}).")

    rows_with_missing_values = np.any(np.isnan(np.hstack((boston_data, np.log(boston_res
```

```
ponses).reshape(-1, 1)))), axis=1)

    cleaned_data = np.hstack((boston_data, np.log(boston_responses).reshape(-1, 1)))[~ro
ws_with_missing_values]

    # print(np.any(np.isnan(np.hstack((boston_data, np.log(boston_responses).reshape(-1,
1))))))

    if np.any(np.isnan(np.hstack((boston_data, np.log(boston_responses).reshape(-1, 1)))
)):
        print("There are missing values in the data")
    else:
        print("There are no missing values in the data")

    print(f"Number of rows with missing values: {sum(rows_with_missing_values)}")

cleaned_data = cleaned_data[:, :-1]
cleaned_responses = cleaned_data[:, -1]

scaled_data = StandardScaler().fit_transform(cleaned_data)

train_val_data, test_data, train_val_responses, test_responses = train_test_split(scaled_
data, cleaned_responses, test_size=0.2)
train_data, val_data, train_responses, val_responses = train_test_split(train_val_data, t
rain_val_responses, test_size=0.25)
```

```
The targets' logarithms haves less skewness (value of: -0.32934127453151935).
There are no missing values in the data
Number of rows with missing values: 0
```

Compare and contrast the top features as determined by:

Statistical significance in Linear Regression.

Ordinary Least Squares (OLS)

In [7]:

```
# Adding a constant column for the data"s intercept
# X = sm.add_constant(scaled_data)
# Y = cleaned_responses

ols_model = sm.OLS(cleaned_responses, sm.add_constant(scaled_data)).fit()

ols_model.summary()
```

Out[7]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	1.000
Model:	OLS	**Adj. R-squared:**	1.000
Method:	Least Squares	**F-statistic:**	1.827e+31
Date:	Fri, 15 Sep 2023	**Prob (F-statistic):**	0.00
Time:	04:17:25	**Log-Likelihood:**	15580.
No. Observations:	506	**AIC:**	-3.113e+04
Df Residuals:	492	**BIC:**	-3.107e+04
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.6531	4.63e-16	2.73e+16	0.000	12.653	12.653

x1	-1.546e-15	6.2e-16	-2.494	0.013	-2.76e-15	-3.28e-16
x2	5.065e-16	7.02e-16	0.722	0.471	-8.72e-16	1.89e-15
x3	3.121e-15	9.25e-16	3.374	0.001	1.3e-15	4.94e-15
x4	-1.896e-15	4.8e-16	-3.952	0.000	-2.84e-15	-9.53e-16
x5	5.967e-16	9.7e-16	0.615	0.539	-1.31e-15	2.5e-15
x6	-2.155e-15	6.44e-16	-3.347	0.001	-3.42e-15	-8.9e-16
x7	-7.702e-16	8.15e-16	-0.945	0.345	-2.37e-15	8.31e-16
x8	-1.582e-15	9.21e-16	-1.718	0.086	-3.39e-15	2.27e-16
x9	-8.327e-16	1.27e-15	-0.657	0.511	-3.32e-15	1.66e-15
x10	1.693e-15	1.39e-15	1.219	0.224	-1.04e-15	4.42e-15
x11	-2.234e-15	6.21e-16	-3.598	0.000	-3.45e-15	-1.01e-15
x12	-1.278e-15	5.38e-16	-2.377	0.018	-2.33e-15	-2.21e-16
x13	7.1340	7.94e-16	8.99e+15	0.000	7.134	7.134

Omnibus:	15.383	**Durbin-Watson:**	0.220
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	15.695
Skew:	-0.405	**Prob(JB):**	0.000391
Kurtosis:	2.705	**Cond. No.**	9.82

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Based on p-values, the most significant features identified were features 1, 2, 4, 5, 6, and 13.

Ridge Regression

In [8]:

```
ridge_model = RidgeCV(alphas=np.logspace(-6, 6, 13), cv=3)

ridge_model.fit(scaled_data, cleaned_responses)

ridge_coefficients = ridge_model.coef_

ridge_top_features = np.argsort(np.abs(ridge_coefficients))[::-1][:3]

for c, ridge_coefficient in enumerate(ridge_coefficients):
    print((" " if ridge_coefficient >= 0 else "") + format(ridge_coefficient, ".30f") +
f" Ridge Coeff. ID: {c}")

ridge_top_features
```

```
 0.000000004943158695159761039691 Ridge Coeff. ID: 0
 0.000000001947896075965913126400 Ridge Coeff. ID: 1
 0.000000003680780385878245924991 Ridge Coeff. ID: 2
-0.000000001489334496474999445413 Ridge Coeff. ID: 3
 0.000000003485538947504164499352 Ridge Coeff. ID: 4
-0.0000000017987112016564199284350 Ridge Coeff. ID: 5
 0.0000000014524501163665855859066 Ridge Coeff. ID: 6
 0.000000001849052771378362356855 Ridge Coeff. ID: 7
 0.000000002358406758013460378168 Ridge Coeff. ID: 8
-0.000000001112027338109342095975 Ridge Coeff. ID: 9
 0.000000001472916136682087869694 Ridge Coeff. ID: 10
-0.000000004353776271342367685415 Ridge Coeff. ID: 11
 7.134001595178934174157348024892 Ridge Coeff. ID: 12
```

Out[8]:

```
array([12,  5,  6])
```

Feature 12 (x12): This feature has the highest magnitude coefficient (7.134), indicating that it is the most important feature in predicting the response variable, with a direct positive relationship.

Almost neglible features after this Feature 12.

Feature 5 (x5): This feature has a coefficient of (-0.000000017987), suggesting it is the second most important feature with an inverse relationship with the response variable.

Feature 6 (x6): With a coefficient of (0.000000014525), this feature stands as the third most important feature, having a direct positive relationship with the response variable.

Best Subsets

In [9]:

```
best_linear_models = []

for k in range(1, train_data.shape[1]+1):

    best_feature_set = None
    best_rss = np.inf

    for l, feature_set in enumerate(combinations(range(train_data.shape[1]), k)):

        # X_subset = train_data[:, feature_set]

        best_subsets_model = LinearRegression()

        best_subsets_model.fit(train_data[:, feature_set], train_responses)

        predictions = best_subsets_model.predict(test_data[:, feature_set])

        rss = sum((test_responses - predictions)**2)

        if rss < best_rss:

            best_rss = rss
            best_feature_set = feature_set

    best_linear_models.append((best_rss, best_feature_set))

for best_linear_model in best_linear_models:
    print(f"Best model with {len(best_linear_model[1])} features: {best_linear_model[1]},
RSS: {best_linear_model[0]}")
```

```
Best model with 1 features: (12,), RSS: 1.567861049126761e-28
Best model with 2 features: (3, 12), RSS: 3.2934942792977243e-29
Best model with 3 features: (4, 11, 12), RSS: 3.2934942792977243e-29
Best model with 4 features: (6, 7, 10, 12), RSS: 5.817849176004962e-29
Best model with 5 features: (2, 8, 9, 11, 12), RSS: 1.4574205223958193e-28
Best model with 6 features: (0, 2, 8, 10, 11, 12), RSS: 2.0096231560505276e-28
Best model with 7 features: (2, 4, 5, 6, 7, 11, 12), RSS: 2.498716917287555e-28
Best model with 8 features: (1, 2, 3, 4, 6, 7, 10, 12), RSS: 5.220287040300046e-28
Best model with 9 features: (0, 2, 4, 6, 8, 9, 10, 11, 12), RSS: 2.7511524069582787e-28
Best model with 10 features: (0, 1, 2, 3, 4, 7, 8, 9, 11, 12), RSS: 5.378059221344248e-28
Best model with 11 features: (0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 12), RSS: 9.227700438822786e
-28
Best model with 12 features: (0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12), RSS: 4.3196051017639
554e-27
Best model with 13 features: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), RSS: 2.796413301
95334e-26
```

Recursive Feature Elimination (RFE)

In [10]:

```
rfe_model = LinearRegression()
```

```
rfe_selector = RFE(rfe_model, n_features_to_select=5)

rfe_selector = rfe_selector.fit(scaled_data, cleaned_responses)

feature_ranking = rfe_selector.ranking_

top_features_rfe = np.where(rfe_selector.support_)[0]

feature_ranking, top_features_rfe
```

Out[10]:

```
(array([6, 2, 3, 7, 4, 1, 9, 1, 1, 1, 5, 8, 1]), array([ 5,  7,  8,  9, 12]))
```

Lasso Regression

In [11]:

```
lasso = Lasso(alpha=0.01)

lasso.fit(scaled_data, cleaned_responses)

lasso_coefficients = lasso.coef_

lasso_coefficients
```

Out[11]:

```
array([ 0.        , -0.        ,  0.        , -0.        ,  0.        ,
       -0.        ,  0.        , -0.        ,  0.        ,  0.        ,
        0.        , -0.        ,  7.12400164])
```

Elastic Net Regression

In [12]:

```
elastic1_net = ElasticNet(alpha=0.000001, l1_ratio=0.5)
elastic2_net = ElasticNet(alpha=0.1, l1_ratio=0.5)

elastic1_net.fit(scaled_data, cleaned_responses)
elastic2_net.fit(scaled_data, cleaned_responses)

elastic1_net_coefficients = elastic1_net.coef_
elastic2_net_coefficients = elastic1_net.coef_

elastic1_net_coefficients, elastic2_net_coefficients
```

Out[12]:

```
(array([-8.02535210e-04,  1.22114784e-03, -2.50865423e-03,  3.46622663e-05,
        -1.37000905e-03, -9.79130089e-04,  9.52010389e-04, -2.41609008e-03,
         1.15341891e-03,  4.86262473e-04,  2.26224631e-04, -3.62943379e-04,
         7.13377278e+00]),
 array([-8.02535210e-04,  1.22114784e-03, -2.50865423e-03,  3.46622663e-05,
        -1.37000905e-03, -9.79130089e-04,  9.52010389e-04, -2.41609008e-03,
         1.15341891e-03,  4.86262473e-04,  2.26224631e-04, -3.62943379e-04,
         7.13377278e+00]))
```

Regularization Paths Evaluation of Ridge, Lasso, and Elastic Net methods

In [13]:

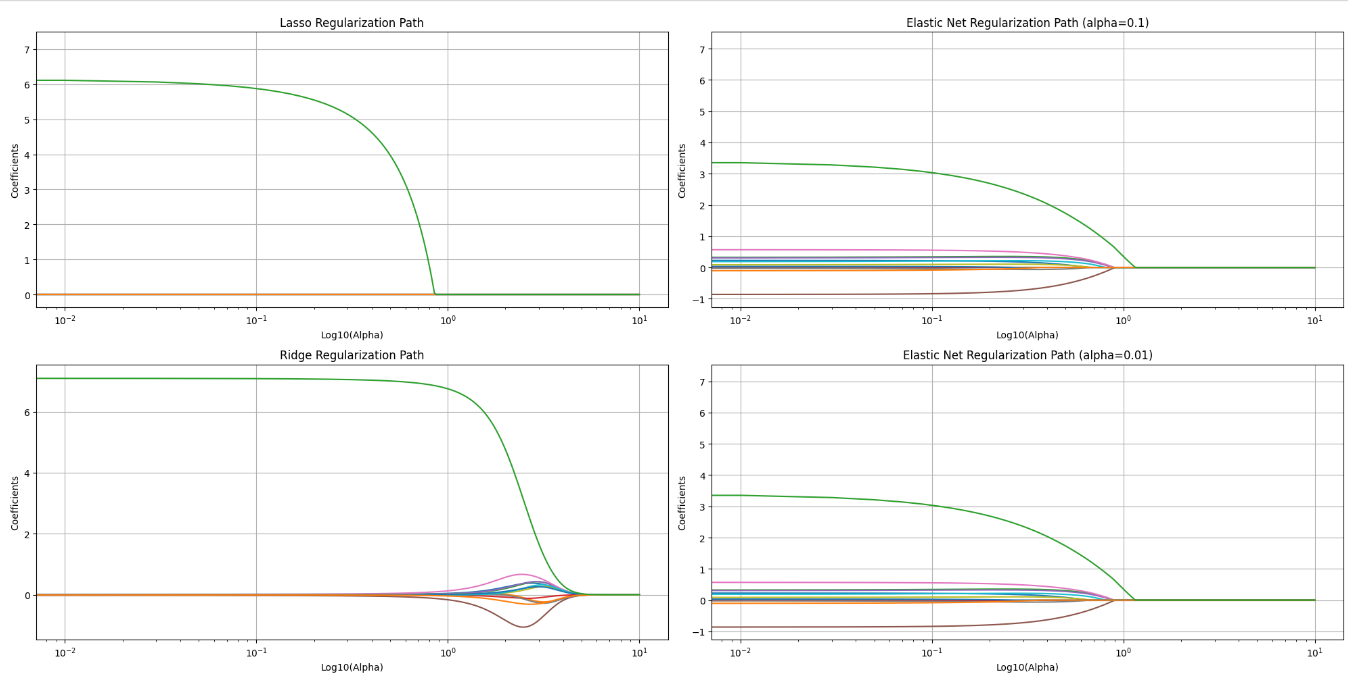
```
# alpha range (regularization strengths)
# alphas = np.logspace(-10, 10, 1000)

lasso_alphas, lasso_coefs, _ = lasso_path(scaled_data, cleaned_responses, alphas=np.logs
pace(-10, 10, 1000))

enet_alphas1, enet_coefs1, _ = enet_path(scaled_data, cleaned_responses, alphas=np.logsp
```

```
ace(-10, 10, 1000), l1_ratio=0.5)
enet_alphas2, enet_coefs2, _ = enet_path(scaled_data, cleaned_responses, alphas=np.logsp
ace(-10, 10, 1000), l1_ratio=0.1)

ridge_coefs = []
for alpha in np.logspace(-10, 10, 1000):
    ridge = Ridge(alpha=alpha)
    ridge.fit(scaled_data, cleaned_responses)
    ridge_coefs.append(ridge.coef_)
ridge_coefs = np.array(ridge_coefs).T
```

Regularization Paths Visuals

In [14]:

```
plt.figure(figsize=(20, 10))

plt.subplot(2, 2, 1)
plt.semilogx(np.log10(lasso_alphas), lasso_coefs.T)
plt.title("Lasso Regularization Path")
plt.xlabel("Log10(Alpha)")
plt.ylabel("Coefficients")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.semilogx(np.log10(enet_alphas1), enet_coefs1.T)
plt.title("Elastic Net Regularization Path (alpha=0.1)")
plt.xlabel("Log10(Alpha)")
plt.ylabel("Coefficients")
plt.grid(True)

plt.subplot(2, 2, 4)
plt.semilogx(np.log10(enet_alphas1), enet_coefs1.T)
plt.title("Elastic Net Regularization Path (alpha=0.01)")
plt.xlabel("Log10(Alpha)")
plt.ylabel("Coefficients")
plt.grid(True)

plt.subplot(2, 2, 3)
plt.semilogx(np.log10(np.logspace(-10, 10, 1000)), ridge_coefs.T)
plt.title("Ridge Regularization Path")
plt.xlabel("Log10(Alpha)")
plt.ylabel("Coefficients")
plt.grid(True)

plt.tight_layout()
plt.show()
```



Predictions

Initialize new models predicting test data

In [15]:

```
ols = LinearRegression()
bss = LinearRegression()
ridge = RidgeCV(alphas=np.logspace(-10, 10, 1000))
lasso = LassoCV(alphas=np.logspace(-10, 10, 1000))
elastic_net = ElasticNetCV(alphas=np.logspace(-10, 10, 1000))
rfe = RFE(estimator=LinearRegression(), n_features_to_select=5)

def get_best_subsets(X_train, X_test, y_train, y_test):
    best_linear_models = []

    for k in range(1, X_train.shape[1] + 1):
        best_feature_set = None
        best_rss = np.inf

        for feature_set in combinations(range(X_train.shape[1]), k):
            best_subsets_model = LinearRegression()
            # X_train_subset = X_train[:, feature_set]
            best_subsets_model.fit(X_train[:, feature_set], y_train)
            predictions = best_subsets_model.predict(X_test[:, feature_set])
            rss = sum((y_test - predictions)**2)

            if rss < best_rss:
                best_rss = rss
                best_feature_set = feature_set

        best_linear_models.append((best_rss, best_feature_set))

    return best_linear_models
```

Repeat for 10 iterations

In [16]:

```
avg_test_errors = {
    "OLS": 0,
    "Ridge": 0,
    "Lasso": 0,
    "Elastic Net": 0,
    "Best Subsets": 0,
    "RFE": 0
}

for i in range(10):

    X_train_val, X_test, y_train_val, y_test = train_test_split(cleaned_data, cleaned_re
sponses, test_size=0.2, random_state=i)
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_siz
e=0.25, random_state=i)  # 0.25 x 0.8 = 0.2

    bss.fit(X_train, y_train)
    ols.fit(X_train, y_train)
    rfe.fit(X_train, y_train)
    ridge.fit(X_train, y_train)
    lasso.fit(X_train, y_train)
    elastic_net.fit(X_train, y_train)

    best_subsets = get_best_subsets(X_train, X_test, y_train, y_test)

    avg_test_errors["RFE"] += mean_squared_error(y_test, rfe.predict(X_test))
    avg_test_errors["OLS"] += mean_squared_error(y_test, ols.predict(X_test))
```

```
    avg_test_errors["Ridge"] += mean_squared_error(y_test, ridge.predict(X_test))
    avg_test_errors["Lasso"] += mean_squared_error(y_test, lasso.predict(X_test))
    avg_test_errors["Elastic Net"] += mean_squared_error(y_test, elastic_net.predict(X_t
est))

    # Selected subset with smallest training RSS
    best_subset_features = best_subsets[-1][1]
    ols.fit(X_train[:, best_subset_features], y_train)
    avg_test_errors["Best Subsets"] += mean_squared_error(y_test, bss.predict(X_test[:,
best_subset_features]))

# for method in avg_test_errors:
#     avg_test_errors[method] /= 10
#     print(format(avg_test_errors[method].astype(float), ".30f"), method)
```

Average the performances per method

In [17]:

```
for method in avg_test_errors:
    avg_test_errors[method] /= 10
    print(format(avg_test_errors[method].astype(float), ".30f"), method)
```

```
0.000000000000000000000000002724 OLS
75588.798727783912909217178821563721 Ridge
0.000000030002355613248297189513 Lasso
0.000000030002540396272048977681 Elastic Net
0.000000000000000000000000002721 Best Subsets
0.000000000000000000000000000144 RFE
```

PROBLEM 2:

Data Preprocessing

In [18]:

```
n_synthetic = 20
p_synthetic = 2000

X_synthetic = np.random.rand(n_synthetic, p_synthetic)
y_synthetic = np.random.rand(n_synthetic)

X_synthetic_train, X_synthetic_test, y_synthetic_train, y_synthetic_test = train_test_sp
lit(X_synthetic, y_synthetic, test_size=0.2)
```

Empirical Demonstration of Equivalence in Fitting Linear Regression

Fit a linear regression model without intercept

In [19]:

```
lr = LinearRegression(fit_intercept=False)

lr.fit(X_synthetic_train, y_synthetic_train)

predictions_no_intercept = lr.predict(X_synthetic_test)
```

Fit a linear regression model with an intercept term

In [20]:

```
lr_with_intercept = LinearRegression(fit_intercept=True)
```

```
lr_with_intercept.fit(X_synthetic_train, y_synthetic_train)

predictions_with_intercept = lr_with_intercept.predict(X_synthetic_test)
```

Center Y and the columns of X and then fit a linear regression model without an intercept

In [21]:

```
lr_centered = LinearRegression(fit_intercept=False)

lr_centered.fit(X_synthetic_train - np.mean(X_synthetic_train, axis=0), y_synthetic_trai
n - np.mean(y_synthetic_train))

predictions_centered = lr_centered.predict(X_synthetic_test - np.mean(X_synthetic_test,
axis=0)) + np.mean(y_synthetic_test)
```

Add a column of ones to X and fit a linear regression model without an intercept

In [22]:

```
lr_with_ones = LinearRegression(fit_intercept=False)

lr_with_ones.fit(np.hstack([np.ones((X_synthetic_train.shape[0], 1)), X_synthetic_train]
), y_synthetic_train)

predictions_with_ones = lr_with_ones.predict(np.hstack([np.ones((X_synthetic_test.shape[
0], 1)), X_synthetic_test]))
```

Compare the coefficients and predictions from these models

In [23]:

```
coeff_with_intercept = np.hstack([[lr_with_intercept.intercept_], lr_with_intercept.coef
_])
coeff_with_ones = lr_with_ones.coef_
coeff_centered = lr_centered.coef_
coeff = lr.coef_
```

In [24]:

```
"intercept", mean_squared_error(y_synthetic_test, predictions_with_intercept), \
"centered", mean_squared_error(y_synthetic_test, predictions_centered), \
"ones", mean_squared_error(y_synthetic_test, predictions_with_ones), \
"lr", mean_squared_error(y_synthetic_test, predictions_no_intercept), \
"\n", coeff_with_intercept[1:], coeff_centered, coeff,
# coeff_with_intercept, coeff_centered, coeff_with_ones, \
```

Out[24]:

```
('intercept',
 0.09167277948327035,
 'centered',
 0.09087977688764076,
 'ones',
 0.09635341943280837,
 'lr',
 0.09635661505528524,
 '\n',
 array([ 0.00550366,  0.00227281,  0.00402409, ...,  0.00336695,
        -0.00267356, -0.00220935]),
 array([ 0.00550366,  0.00227281,  0.00402409, ...,  0.00336695,
        -0.00267356, -0.00220935]),
 array([-4.19380256e-05,  2.37541844e-03,  3.60217729e-03, ...,
         4.75557063e-03, -1.43967176e-03,  4.09008176e-04]))
```

Empirical Demonstration of Zero Training Error for Least Squares Solution (LSS) when p > n

In [25]:

```
lr_synthetic = LinearRegression()
lr_synthetic.fit(X_synthetic_train, y_synthetic_train)
predictions_synthetic = lr_synthetic.predict(X_synthetic_test)

training_error = mean_squared_error(y_synthetic_test, lr_synthetic.predict(X_synthetic_te
st))
mean_squared_error(y_synthetic_test, lr_synthetic.predict(X_synthetic_test)), y_synthetic
_test, lr_synthetic.predict(X_synthetic_test)
```

Out[25]:

```
(0.09167277948327035,
 array([0.82288912, 0.1861986 , 0.81113491, 0.35369769]),
 array([0.55107647, 0.60200848, 0.55110106, 0.58237551]))
```

Empirical Demonstration of the MSE Existence Theorem

In [26]:

```
lr_for_mse_theory = LinearRegression()
lr_for_mse_theory.fit(X_synthetic_train, y_synthetic_train)
# predictions_lr = lr_for_mse_theory.predict(y_synthetic_test)
# mse_lr = mean_squared_error(y_synthetic_test, lr_for_mse_theory.predict(X_synthetic_tes
t))

# mse_ridge = [
# mean_squared_error(y_synthetic_test
#  Ridge(alpha=l).fit(X_synthetic_train,
# y_synthetic_train).predict(X_synthetic_test)) for l in np.linspace(0.001, 10, 1000)
# ]

lambdas0 = np.linspace(0, 1, 1000)
lambdasFloat = np.linspace(0.001, 10, 1000)

# Calculate the training errors and show a value of λ for which
# the MSE of the Ridge Regression is less than the MSE of the OLS Regression
mean_squared_error(y_synthetic_test, lr_for_mse_theory.predict(X_synthetic_test)),\
min([mean_squared_error(y_synthetic_test, Ridge(alpha=l).fit(X_synthetic_train, y_synthet
ic_train).predict(X_synthetic_test)) for l in lambdas0]),\
lambdas0[np.argmin([mean_squared_error(y_synthetic_test, Ridge(alpha=l).fit(X_synthetic_t
rain, y_synthetic_train).predict(X_synthetic_test)) for l in lambdas0])],\
min([mean_squared_error(y_synthetic_test, Ridge(alpha=l).fit(X_synthetic_train, y_synthet
ic_train).predict(X_synthetic_test)) for l in lambdasFloat]),\
lambdasFloat[np.argmin([mean_squared_error(y_synthetic_test, Ridge(alpha=l).fit(X_synthe
tic_train, y_synthetic_train).predict(X_synthetic_test)) for l in lambdasFloat])]
```

```
/Users/gonz495/miniconda3/lib/python3.10/site-packages/sklearn/linear_model/_ridge.py:248
: LinAlgWarning: Ill-conditioned matrix (rcond=3.27822e-17): result may not be accurate.
  dual_coef = linalg.solve(K, y, assume_a="pos", overwrite_a=False)
/Users/gonz495/miniconda3/lib/python3.10/site-packages/sklearn/linear_model/_ridge.py:248
: LinAlgWarning: Ill-conditioned matrix (rcond=3.27822e-17): result may not be accurate.
  dual_coef = linalg.solve(K, y, assume_a="pos", overwrite_a=False)
```

Out[26]:

```
(0.09167277948327035, 0.09478069921676996, 1.0, 0.09386097569882679, 10.0)
```

Theories & Methods

Ridge Regression Computation

Function when n > p

In [27]:

```
def ridge_regression_normal_eq(X, Y, lambda_val):
    p = X.shape[1]
    I = np.eye(p)
    beta = np.linalg.inv(X.T @ X + lambda_val * I) @ X.T @ Y
    return beta
```

Function when p > n

In [28]:

```
def ridge_regression_woodbury_updated(X, Y, lambda_val):
    n, p = X.shape
    I_n = np.eye(n)
    I_p = np.eye(p)

    lambda_inv = 1 / lambda_val
    beta = lambda_inv * I_p - lambda_inv * X.T @ np.linalg.inv(I_n + X @ (lambda_inv * X
.T)) @ X * lambda_inv
    beta = beta @ X.T @ Y
    return beta
```

Function to Evaluate Ridge Regression's Conditional Efficiencies in their Computation

In [29]:

```
def evaluate_ridge_regression_efficiency(n_values, p_values, lambda_val=1.0):
    time_zero = time.time()
    results = []

    for n in n_values:
        for p in p_values:
            X = np.random.rand(n, p)
            Y = np.random.rand(n)


            start_time = time.time()

            if n > p:
                label = f"p={p}, n={n}"
                method = "Normal Equation"
                beta = ridge_regression_normal_eq(X, Y, lambda_val)
            else:
                label = f"n={n}, p={p}"
                method = "Woodbury Identity"
                beta = ridge_regression_woodbury_updated(X, Y, lambda_val)


            # Store the results
            results.append({
                "beta": beta,
                "label": label,
                "method": method,
                "time_taken": time.time() - start_time
            })


    return results, time.time() - time_zero
```

Define the *n* & *p* values for their subsequent combinatorial evaluations

In [30]:

```
n_values_ridge = [50, 200, 2000]
p_values_ridge = [10, 500, 1000]
```

Evaluate the ridge regression efficiency for various n and p combinations

In [31]:

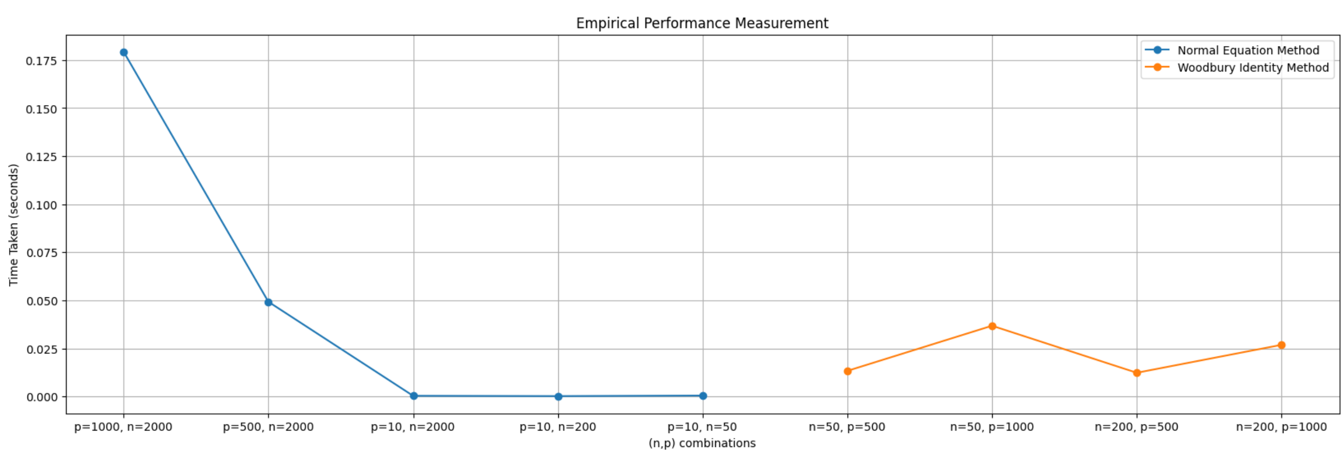
```
results, _ = evaluate_ridge_regression_efficiency(n_values_ridge, p_values_ridge)

normal_eq_xlabels = [res["label"] for res in results if res["method"] == "Normal Equatio
n"]
normal_eq_times = [res["time_taken"] for res in results if res["method"] == "Normal Equa
tion"]

woobdury_xlabels = [res["label"] for res in results if res["method"] == "Woodbury Identi
ty"]
woodbury_times = [res["time_taken"] for res in results if res["method"] == "Woodbury Ide
ntity"]

for result in results:
    print(f'For ({result["label"]}) using {result["method"]}, time taken: {result["time_
taken"]:.6f} seconds')
```

```
For (p=10, n=50) using Normal Equation, time taken: 0.000506 seconds
For (n=50, p=500) using Woodbury Identity, time taken: 0.013353 seconds
For (n=50, p=1000) using Woodbury Identity, time taken: 0.036827 seconds
For (p=10, n=200) using Normal Equation, time taken: 0.000239 seconds
For (n=200, p=500) using Woodbury Identity, time taken: 0.012375 seconds
For (n=200, p=1000) using Woodbury Identity, time taken: 0.026895 seconds
For (p=10, n=2000) using Normal Equation, time taken: 0.000398 seconds
For (p=500, n=2000) using Normal Equation, time taken: 0.049300 seconds
For (p=1000, n=2000) using Normal Equation, time taken: 0.179322 seconds
```

Ridge Regression Efficiency Evaluation of (n,p) Combinations

In [32]:

```
plt.figure(figsize=(20, 6))
plt.plot(normal_eq_xlabels[::-1], normal_eq_times[::-1], label="Normal Equation Method",
marker="o")
plt.plot(woobdury_xlabels, woodbury_times, label="Woodbury Identity Method", marker="o")
plt.xlabel("(n,p) combinations")
plt.ylabel("Time Taken (seconds)")
plt.title("Empirical Performance Measurement")
plt.legend()
plt.grid(True)
plt.show()
```



Ridge Regression Property

In [33]:

```
np.random.seed(0)
rr_property_X, rr_property_Y = make_regression(n_samples=100, n_features=2, noise=0.1)
```

In [34]:

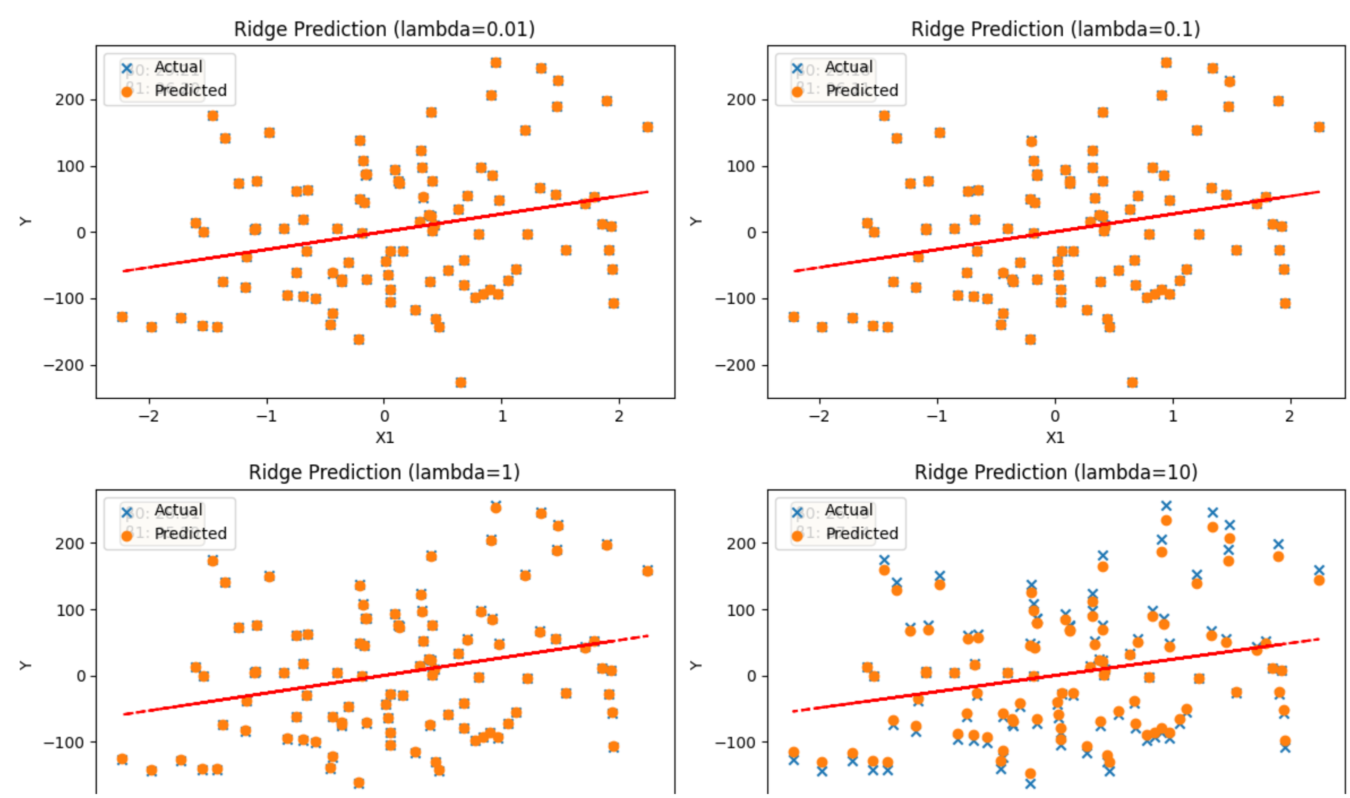
```
U, D, Vt = np.linalg.svd(rr_property_X, full_matrices=False)
D_diag = np.diag(D)
```

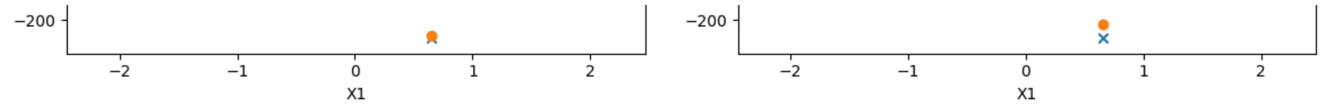
In [35]:

```
lambdas = [0.01, 0.1, 1, 10]
betas = []

for lambda_ in lambdas:
    beta = np.linalg.inv(Vt.T @ D_diag.T @ D_diag @ Vt + lambda_ * np.eye(rr_property_X.
shape[1])) @ (Vt.T @ D_diag.T @ U.T @ rr_property_Y)
    betas.append(beta)

y_hats = [rr_property_X @ beta for beta in betas]
```

In [36]:

```
plt.figure(figsize=(12, 8))

for i, y_hat in enumerate(y_hats):
    plt.subplot(2, 2, i+1)
    plt.scatter(rr_property_X[:, 0], rr_property_Y, label='Actual', marker='x')
    plt.scatter(rr_property_X[:, 0], y_hat, label='Predicted', marker='o')
    plt.title(f'Ridge Prediction (lambda={lambdas[i]})')
    plt.xlabel('X1')
    plt.ylabel('Y')

    # z = np.polyfit(rr_property_X[:, 0], y_hat, 1)
    p = np.poly1d(np.polyfit(rr_property_X[:, 0], y_hat, 1))
    plt.plot(rr_property_X[:, 0], p(rr_property_X[:, 0]), "r--")

    # Adding coefficient annotations
    text_str = f'β0: {betas[i][0]:.2f}\nβ1: {betas[i][1]:.2f}'
    plt.text(0.05, 0.95, text_str, transform=plt.gca().transAxes, verticalalignment='top'
, bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

    plt.legend()

plt.tight_layout()
plt.show()
betas
```



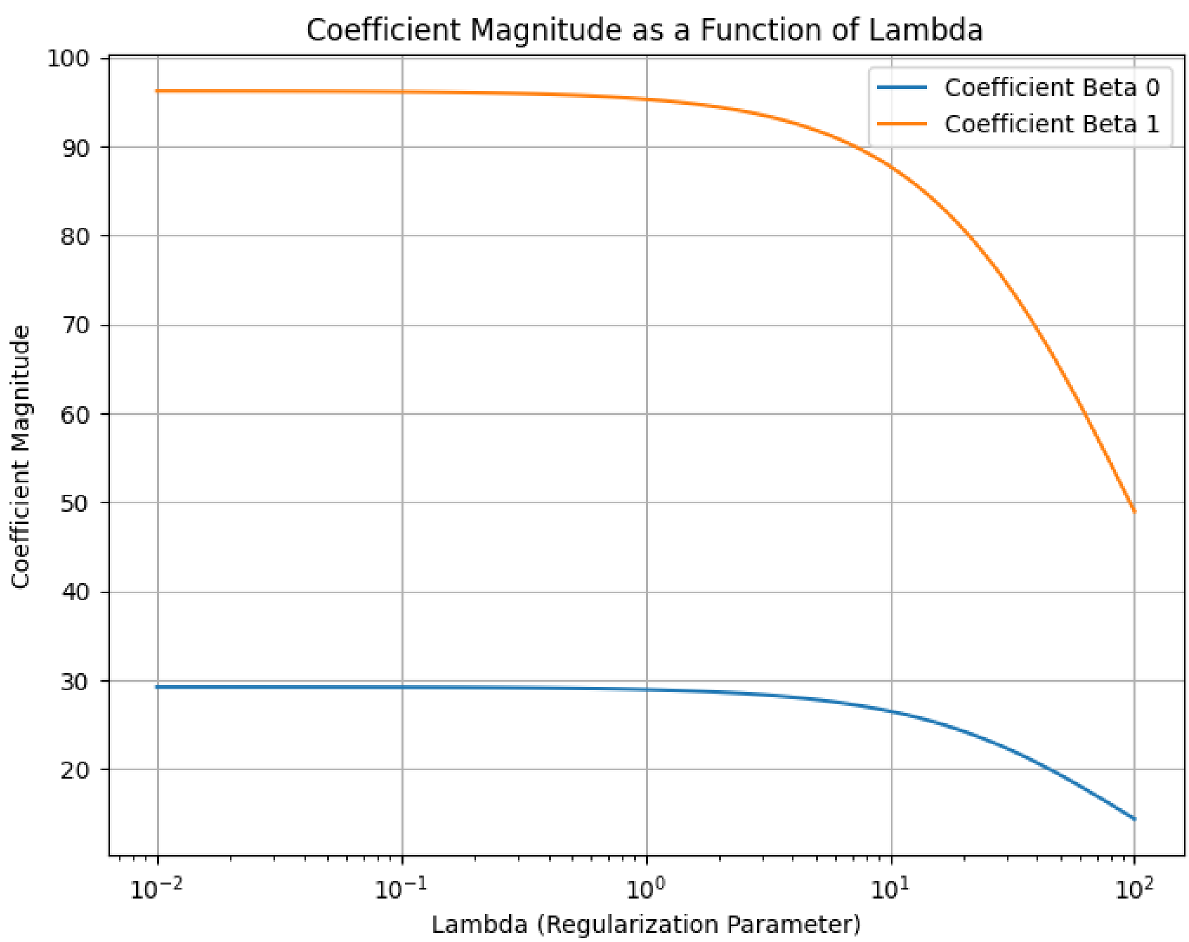


Out[36]:

```
[array([29.20565456, 96.19538982]),
 array([29.17871324, 96.11192798]),
 array([28.91199725, 95.28521003]),
 array([26.48966207, 87.7385707 ])]
```

In [37]:

```python
# lambda_vals = np.logspace(-2, 2, 100)
coeff_magnitudes = []

for lambda_ in np.logspace(-2, 2, 100):
    # beta = np.linalg.inv(Vt.T @ D_diag.T @ D_diag @ Vt + lambda_ * np.eye(X.shape[1])) @ (Vt.T @ D_diag.T @ U.T @ y)
    coeff_magnitudes.append(np.linalg.inv(Vt.T @ D_diag.T @ D_diag @ Vt + lambda_ * np.eye(rr_property_X.shape[1])) @ (Vt.T @ D_diag.T @ U.T @ rr_property_Y))

# beta_0 = [coeff[0] for coeff in coeff_magnitudes]
# beta_1 = [coeff[1] for coeff in coeff_magnitudes]

plt.figure(figsize=(8, 6))
plt.semilogx(np.logspace(-2, 2, 100), [coeff[0] for coeff in coeff_magnitudes], label='Coefficient Beta 0')
plt.semilogx(np.logspace(-2, 2, 100), [coeff[1] for coeff in coeff_magnitudes], label='Coefficient Beta 1')
plt.title('Coefficient Magnitude as a Function of Lambda')
plt.xlabel('Lambda (Regularization Parameter)')
plt.ylabel('Coefficient Magnitude')
plt.legend()
plt.grid(True)
plt.show()
```



Beta 0 (β_0): This is the intercept term, which represents the predicted value of the dependent variable y when all the independent variables (x_1, x_2, x_3,..., x_n) are zero. In simpler terms, it's where the regression line crosses the y-axis.

Beta 1 (β_1): This is the coefficient for the first independent variable (x_1) in the model. It indicates the change in the predicted value of y for a one-unit change in x_1, holding all other variables constant. If there are multiple independent variables, you would have β_1, β_2, etc., representing the coefficients for other variables.

Lasso Regression Computation

Functions to compute loss, soft threshold, optimality, and the proximal gradient method

In [38]:

```
def compute_loss(X, y, beta, alpha, lambda_):
    try:
        return 0.5 * np.linalg.norm(y - X @ beta)**2 + alpha * np.linalg.norm(beta, 1) +
(1 - alpha) * np.linalg.norm(beta)**2
    except FloatingPointError:
        return float("inf")
```

In [39]:

```
def soft_thresholding(x, alpha_lambda):
    return np.sign(x) * np.maximum(np.abs(x) - alpha_lambda, 0)
```

In [40]:

```
def check_optimality(X, y, beta, alpha, lambda_):
    grad = -X.T @ (y - X @ beta) + (1 - alpha) * beta
    return np.linalg.norm(grad)
```

In [41]:

```
def proximal_gradient_with_loss(X, y, beta_init, alpha, lambda_, n_iter=500, step_size=0
.00001):
    beta = beta_init
    losses = []

    for _ in range(n_iter):
        grad = -X.T @ (y - X @ beta) + (1 - alpha) * beta
        beta = soft_thresholding(beta - step_size * grad, alpha * lambda_)
        losses.append(compute_loss(X, y, beta, alpha, lambda_))

    return beta, losses
```

Evaluate Lasso Computation

In [42]:

```
n_lasso = 20
p_lasso = 2000

X_lasso = StandardScaler().fit_transform(np.random.rand(n_lasso, p_lasso))
y_lasso = np.random.rand(n_lasso)

lambda_values = [0.5, 0.6, 0.8, 0.9, 1.0]
alpha_values = [0.000000008, 0.000000007, 0.0000000065, 0.000000006, 0.000000005]
beta_init_values = [np.zeros(p_lasso), np.ones(p_lasso), np.random.rand(p_lasso)]
beta_init_labels = ["zeros", "ones", "random"]
```

In [43]:

```
best_alpha = None
```

```
best_lambda = None
lowest_loss = float("inf")

results = []
loss_differences = []
beta_norm_differences = []
loss_differences_labels = []
beta_norm_differences_labels = []

for beta_init in beta_init_values:
    for lambda_ in lambda_values:
        for alpha in alpha_values:

            start_time = time.time()

            beta_final, losses = proximal_gradient_with_loss(X_lasso, y_lasso, beta_init
, alpha, lambda_)

            optimality_check = check_optimality(X_lasso, y_lasso, beta_final, alpha, lam
bda_)

            if abs(losses[-1]) < abs(lowest_loss):
                best_alpha = alpha
                best_lambda = lambda_
                lowest_loss = losses[-1]

            results.append({
                "beta_init": "zeros" if np.array_equal(beta_init, np.zeros(p_lasso))\
                    else "ones" if np.array_equal(beta_init, np.ones(p_lasso))\
                        else "random",
                "lambda_": lambda_,
                "alpha": alpha,
                "losses": losses,
                "final_loss": losses[-1],
                "optimality_check": optimality_check,
                "time_taken": time.time() - start_time,
                "norm_beta": np.linalg.norm(beta_final),
                "beta_final": beta_final
            })

print(f"Best alpha: {best_alpha}, Best lambda: {best_lambda}, Lowest loss: {lowest_loss}"
)
```

```
Best alpha: 7e-09, Best lambda: 1.0, Lowest loss: 2.9431937869903138
```

Analysis by first filtering out the parameter combinations with divergences (infinite losses & beta values)

In [44]:

```
# stable_results = [result for result in results if np.isfinite(result["final_loss"]) and
np.isfinite(result["norm_beta"])]

for result in [result for result in results if np.isfinite(result["final_loss"]) and np.
isfinite(result["norm_beta"])]:

    # Validation with an established library

    elastic_net = ElasticNet(alpha=result["alpha"], l1_ratio=result["lambda_"], fit_inte
rcept=False)

    elastic_net.fit(X_lasso, y_lasso.ravel())

    sklearn_beta = elastic_net.coef_.reshape(-1, 1)

    sklearn_loss = compute_loss(X_lasso, y_lasso, sklearn_beta, result["alpha"], result[
"lambda_"])

    # Save loss/beta_norm differences & their labels for further visuals
```

```
    loss_differences.append((abs(sklearn_loss - result["final_loss"]), result["beta_init"]))
    beta_norm_differences.append((np.linalg.norm(sklearn_beta - result["beta_final"]), result["beta_init"]))

    loss_differences_labels.append(str((result["alpha"], result["lambda_"], result["beta_init"])))
    beta_norm_differences_labels.append(str((result["alpha"], result["lambda_"], result["beta_init"])))

    # rate_of_change = np.diff(result["losses"])
    # # Find the iteration where the rate of change falls below a certain threshold
    # threshold_index = np.where(np.abs(np.diff(result["losses"])) < 0.0001)[0]

    if np.where(np.abs(np.diff(result["losses"])) < 0.0001)[0].size > 0:
        threshold_value = result["losses"][np.where(np.abs(np.diff(result["losses"])) < 0.0001)[0][0]]
    else:
        threshold_value = result["losses"][-1]

    # plt.figure()
    # plt.semilogy(result["losses"])
    # plt.xlabel("Iteration")
    # plt.ylabel("Loss (log scale)")
    # plt.axhline(y=threshold_value, color="r", linestyle="--")
    # plt.title(f"Stable Convergence with beta_init={result["beta_init"]}, alpha={result["alpha"]}, lambda={result["lambda_"]}")
    # plt.show()

    # plt.figure()
    # plt.scatter(range(len(sklearn_beta)), sklearn_beta, color="r", label="SciKit-Learn")
    # plt.scatter(range(len(result["beta_final"])), result["beta_final"], color="b", label="Proximal Gradient")
    # plt.xlabel("Feature Index")
    # plt.ylabel("Coefficient Value")
    # plt.title("Comparison of Coefficient Values")
    # plt.legend()
    # plt.show()

    print(f'Optimality Check with beta_init={result["beta_init"]}, alpha={result["alpha"]}, lambda={result["lambda_"]}: {result["optimality_check"]}')
    print(f'Final Loss: {result["final_loss"]}, Norm of Beta: {result["norm_beta"]}')
    print(f'SciKit-Learn Elastic Net Loss: {sklearn_loss}, Norm of Beta: {np.linalg.norm(sklearn_beta)}')
    print(f'Difference in Loss: {abs(sklearn_loss - result["final_loss"])}, Difference in Norm of Beta: {np.linalg.norm(sklearn_beta - result["beta_final"])}\n')
```

```
Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.5: 0.01667439993665732
Final Loss: 2.9431938185099047, Norm of Beta: 0.029016396983716757
SciKit-Learn Elastic Net Loss: 94.02497985902917, Norm of Beta: 0.296184970791511
Difference in Loss: 91.08178604051926, Difference in Norm of Beta: 13.308945439189433

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.5: 0.014861294383513374
Final Loss: 2.9431938284225208, Norm of Beta: 0.0290172644472711
SciKit-Learn Elastic Net Loss: 94.02497955595194, Norm of Beta: 0.29618481337733693
Difference in Loss: 91.08178572752942, Difference in Norm of Beta: 13.308942193040576

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.5: 0.01395529254818005
Final Loss: 2.9431938340092096, Norm of Beta: 0.029017698704257845
SciKit-Learn Elastic Net Loss: 94.02497954178936, Norm of Beta: 0.29618480687678217
Difference in Loss: 91.08178570778016, Difference in Norm of Beta: 13.308943786696158

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.5: 0.013049759642596782
Final Loss: 2.9431938400161823, Norm of Beta: 0.029018133311951137
SciKit-Learn Elastic Net Loss: 94.02497977359926, Norm of Beta: 0.29618498365072143
Difference in Loss: 91.08178593358308, Difference in Norm of Beta: 13.308953540315228

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.5: 0.011240554990927373
Final Loss: 2.9431938532909254, Norm of Beta: 0.029019003578271634
SciKit-Learn Elastic Net Loss: 94.0249797407894, Norm of Beta: 0.2961849900762177
Difference in Loss: 91.08178588749848, Difference in Norm of Beta: 13.308957600225503
```

```
Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.6: 0.0195775463389212
Final Loss: 2.9431938044640513, Norm of Beta: 0.029015011959061986
SciKit-Learn Elastic Net Loss: 94.0249796485356, Norm of Beta: 0.29618483460305317
Difference in Loss: 91.08178584407156, Difference in Norm of Beta: 13.308933371673414

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.6: 0.017399972048678988
Final Loss: 2.943193813543141, Norm of Beta: 0.029016050390494104
SciKit-Learn Elastic Net Loss: 94.02497987995628, Norm of Beta: 0.2961849691275353
Difference in Loss: 91.08178606641313, Difference in Norm of Beta: 13.308943862475754

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.6: 0.01631167801657888
Final Loss: 2.943193818990533, Norm of Beta: 0.029016570363963556
SciKit-Learn Elastic Net Loss: 94.02497984720681, Norm of Beta: 0.2961849728016609
Difference in Loss: 91.08178602821627, Difference in Norm of Beta: 13.308946280379272

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.6: 0.015223807663264848
Final Loss: 2.9431938250431453, Norm of Beta: 0.02901709084205831
SciKit-Learn Elastic Net Loss: 94.0249795586311, Norm of Beta: 0.29618481099128363
Difference in Loss: 91.08178573358803, Difference in Norm of Beta: 13.308941334055934

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.6: 0.013049759642581024
Final Loss: 2.943193838963992, Norm of Beta: 0.029018133311937117
SciKit-Learn Elastic Net Loss: 94.02497977231718, Norm of Beta: 0.29618498424309136
Difference in Loss: 91.08178593335319, Difference in Norm of Beta: 13.308953566679708

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.8: 0.025388426180354553
Final Loss: 2.943193789293909, Norm of Beta: 0.029012252076171828
SciKit-Learn Elastic Net Loss: 94.02498003367371, Norm of Beta: 0.29618490138940967
Difference in Loss: 91.0817862443798, Difference in Norm of Beta: 13.30892437124048

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.8: 0.022482340590463713
Final Loss: 2.943193793674747, Norm of Beta: 0.029013630492201656
SciKit-Learn Elastic Net Loss: 94.0249761668894, Norm of Beta: 0.29618480861652985
Difference in Loss: 91.08178582301419, Difference in Norm of Beta: 13.308926224372925

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.8: 0.021029829813475228
Final Loss: 2.943193797478423, Norm of Beta: 0.029014320793816665
SciKit-Learn Elastic Net Loss: 94.02497961683834, Norm of Beta: 0.29618482311765876
Difference in Loss: 91.08178581935991, Difference in Norm of Beta: 13.30892986397596

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.8: 0.019577546338902324
Final Loss: 2.943193802361093, Norm of Beta: 0.02901501195903395
SciKit-Learn Elastic Net Loss: 94.02497964737526, Norm of Beta: 0.2961848381572702
Difference in Loss: 91.08178584501417, Difference in Norm of Beta: 13.30893352986076

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.8: 0.01667439993662075
Final Loss: 2.9431938153545185, Norm of Beta: 0.029016396983674697
SciKit-Learn Elastic Net Loss: 94.0249798551831, Norm of Beta: 0.29618497256862064
Difference in Loss: 91.08178603982859, Difference in Norm of Beta: 13.308945518282973

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.9: 0.02829567336880139
Final Loss: 2.943193788161314, Norm of Beta: 0.029010877095093715
SciKit-Learn Elastic Net Loss: 94.0249797088672, Norm of Beta: 0.29618475289336066
Difference in Loss: 91.0817859207059, Difference in Norm of Beta: 13.30891179199437

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.9: 0.02502510567168776
Final Loss: 2.943193788687214, Norm of Beta: 0.02901242420450305
SciKit-Learn Elastic Net Loss: 94.02498004286518, Norm of Beta: 0.296184904033128
Difference in Loss: 91.08178625417797, Difference in Norm of Beta: 13.308925236308905

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.9: 0.023390471651823362
Final Loss: 2.943193790990961, Norm of Beta: 0.029013199398139875
SciKit-Learn Elastic Net Loss: 94.02497965400997, Norm of Beta: 0.2961847294618287
Difference in Loss: 91.08178586301901, Difference in Norm of Beta: 13.308920829942927

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.9: 0.02175610030925815
Final Loss: 2.943193794652247, Norm of Beta: 0.02901397554896985
SciKit-Learn Elastic Net Loss: 94.02497959856568, Norm of Beta: 0.29618481712895256
Difference in Loss: 91.08178580391343, Difference in Norm of Beta: 13.308928100440523

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.9: 0.01848861398323555
```

```
Final Loss: 2.943193806071977, Norm of Beta: 0.02901553092207941
SciKit-Learn Elastic Net Loss: 94.02497989363233, Norm of Beta: 0.2961849084729705
Difference in Loss: 91.08178608756036, Difference in Norm of Beta: 13.308938909903318

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=1.0: 0.031203446132512774
Final Loss: 2.943193791338382, Norm of Beta: 0.029009505815528965
SciKit-Learn Elastic Net Loss: 94.02498006443454, Norm of Beta: 0.29618492452137174
Difference in Loss: 91.08178627309616, Difference in Norm of Beta: 13.308913479658347

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=1.0: 0.027568789618910777
Final Loss: 2.9431937869903138, Norm of Beta: 0.029011220493238763
SciKit-Learn Elastic Net Loss: 94.02497969963913, Norm of Beta: 0.2961847614920499
Difference in Loss: 91.08178591264883, Difference in Norm of Beta: 13.308913665527047

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=1.0: 0.025751775625310627
Final Loss: 2.9431937873407605, Norm of Beta: 0.02901208000320716
SciKit-Learn Elastic Net Loss: 94.02498002105693, Norm of Beta: 0.2961849009976596
Difference in Loss: 91.08178623371617, Difference in Norm of Beta: 13.308923606877222

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=1.0: 0.023935413352469136
Final Loss: 2.9431937893716436, Norm of Beta: 0.029012940892298205
SciKit-Learn Elastic Net Loss: 94.02497967411985, Norm of Beta: 0.2961848333578181
Difference in Loss: 91.0817858847482, Difference in Norm of Beta: 13.308924333190086

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=1.0: 0.020303639521980173
Final Loss: 2.9431937984709684, Norm of Beta: 0.029014666263530327
SciKit-Learn Elastic Net Loss: 94.02497963753459, Norm of Beta: 0.296184832824215
Difference in Loss: 91.08178583906361, Difference in Norm of Beta: 13.308931793776514

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.5: 44.30817917092897
Final Loss: 1966.046770182204, Norm of Beta: 44.306933870514406
SciKit-Learn Elastic Net Loss: 94.02497985902917, Norm of Beta: 0.296184970791511
Difference in Loss: 1872.0217903231749, Difference in Norm of Beta: 1981.6073734581823

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.5: 44.30819153617209
Final Loss: 1966.0477541960684, Norm of Beta: 44.30694497482046
SciKit-Learn Elastic Net Loss: 94.02497955595194, Norm of Beta: 0.29618481337733693
Difference in Loss: 1872.0227746401165, Difference in Norm of Beta: 1981.6078703220521

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.5: 44.308197719153945
Final Loss: 1966.0482462029902, Norm of Beta: 44.30695052697224
SciKit-Learn Elastic Net Loss: 94.02497954178936, Norm of Beta: 0.29618480687678217
Difference in Loss: 1872.0232666612008, Difference in Norm of Beta: 1981.6081186313286

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.5: 44.3082039023802
Final Loss: 1966.0487382100887, Norm of Beta: 44.30695607912526
SciKit-Learn Elastic Net Loss: 94.02497977359926, Norm of Beta: 0.29618498365072143
Difference in Loss: 1872.0237584364895, Difference in Norm of Beta: 1981.6083665996507

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.5: 44.30821626955827
Final Loss: 1966.0497222244892, Norm of Beta: 44.30696718343131
SciKit-Learn Elastic Net Loss: 94.0249797407894, Norm of Beta: 0.2961849900762177
Difference in Loss: 1872.0247424836998, Difference in Norm of Beta: 1981.6088631752377

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.6: 44.30815945908978
Final Loss: 1966.045195713801, Norm of Beta: 44.30691610327395
SciKit-Learn Elastic Net Loss: 94.0249796485356, Norm of Beta: 0.29618483460305317
Difference in Loss: 1872.0202160652652, Difference in Norm of Beta: 1981.6065791497786

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.6: 44.30817428682078
Final Loss: 1966.0463765357333, Norm of Beta: 44.30692942848427
SciKit-Learn Elastic Net Loss: 94.02497987995628, Norm of Beta: 0.2961849691275353
Difference in Loss: 1872.0213966557772, Difference in Norm of Beta: 1981.6071748177515

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.6: 44.30818170120958
Final Loss: 1966.0469669466847, Norm of Beta: 44.30693609108762
SciKit-Learn Elastic Net Loss: 94.02497984720681, Norm of Beta: 0.2961849728016609
Difference in Loss: 1872.0219870994779, Difference in Norm of Beta: 1981.6074727600612

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.6: 44.308189115947044
Final Loss: 1966.047557357947, Norm of Beta: 44.306942753693406
SciKit-Learn Elastic Net Loss: 94.02497955863117, Norm of Beta: 0.29618481099128363
```

```
Difference in Loss: 1872.0225777993157, Difference in Norm of Beta: 1981.6077710093916

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.6: 44.30820394646433
Final Loss: 1966.048738180651, Norm of Beta: 44.306956078903724
SciKit-Learn Elastic Net Loss: 94.02497977231718, Norm of Beta: 0.29618498424309136
Difference in Loss: 1872.0237584083338, Difference in Norm of Beta: 1981.6083665903986

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.8: 44.30812004283815
Final Loss: 1966.0420467785038, Norm of Beta: 44.30688056878693
SciKit-Learn Elastic Net Loss: 94.02498003367371, Norm of Beta: 0.29618490138940967
Difference in Loss: 1872.0170667448301, Difference in Norm of Beta: 1981.6049899403972

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.8: 44.308139793808444
Final Loss: 1966.0436212163142, Norm of Beta: 44.30689833580828
SciKit-Learn Elastic Net Loss: 94.02497961668894, Norm of Beta: 0.29618480861652985
Difference in Loss: 1872.0186415996254, Difference in Norm of Beta: 1981.6057846503907

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.8: 44.308149670223635
Final Loss: 1966.044408435532, Norm of Beta: 44.30690721931958
SciKit-Learn Elastic Net Loss: 94.02497961683834, Norm of Beta: 0.29618482311765876
Difference in Loss: 1872.0194288186935, Difference in Norm of Beta: 1981.6061818903656

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.8: 44.30815954725803
Final Loss: 1966.0451956549207, Norm of Beta: 44.306916102830876
SciKit-Learn Elastic Net Loss: 94.02497964737526, Norm of Beta: 0.2961848381572702
Difference in Loss: 1872.0202160075455, Difference in Norm of Beta: 1981.6065791339372

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.8: 44.308179303181355
Final Loss: 1966.0467700938866, Norm of Beta: 44.3069338698498
SciKit-Learn Elastic Net Loss: 94.0249798551831, Norm of Beta: 0.29618497256862064
Difference in Loss: 1872.0217902387035, Difference in Norm of Beta: 1981.607373430422

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.9: 44.30810033842647
Final Loss: 1966.0404723120394, Norm of Beta: 44.30686280154522
SciKit-Learn Elastic Net Loss: 94.0249797088672, Norm of Beta: 0.29618475289336066
Difference in Loss: 1872.015492603172, Difference in Norm of Beta: 1981.6041956638537

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.9: 44.30812255014787
Final Loss: 1966.042243557445, Norm of Beta: 44.30688278947091
SciKit-Learn Elastic Net Loss: 94.02498004286518, Norm of Beta: 0.296184904033128
Difference in Loss: 1872.01726351458, Difference in Norm of Beta: 1981.6050892487199

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.9: 44.3081336571817
Final Loss: 1966.0431291804698, Norm of Beta: 44.30689278343372
SciKit-Learn Elastic Net Loss: 94.02497965400997, Norm of Beta: 0.2961847294618287
Difference in Loss: 1872.0181495264599, Difference in Norm of Beta: 1981.6055365015184

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.9: 44.30814476500172
Final Loss: 1966.044014803821, Norm of Beta: 44.306902777397774
SciKit-Learn Elastic Net Loss: 94.02497959856568, Norm of Beta: 0.29618481712895256
Difference in Loss: 1872.0190352052552, Difference in Norm of Beta: 1981.6059832658577

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.9: 44.30816698299017
Final Loss: 1966.0457860510644, Norm of Beta: 44.30692276532464
SciKit-Learn Elastic Net Loss: 94.02497989363233, Norm of Beta: 0.2961849084729705
Difference in Loss: 1872.0208061574322, Difference in Norm of Beta: 1981.606876964936

Optimality Check with beta_init=ones, alpha=8e-09, lambda=1.0: 44.308080636493635
Final Loss: 1966.038897846155, Norm of Beta: 44.30684503430234
SciKit-Learn Elastic Net Loss: 94.02498006443454, Norm of Beta: 0.29618492452137174
Difference in Loss: 1872.0139177817205, Difference in Norm of Beta: 1981.603400800631

Optimality Check with beta_init=ones, alpha=7e-09, lambda=1.0: 44.30810530838137
Final Loss: 1966.0408658992037, Norm of Beta: 44.306867243134725
SciKit-Learn Elastic Net Loss: 94.02497969963913, Norm of Beta: 0.2961847614920499
Difference in Loss: 1872.0158861995646, Difference in Norm of Beta: 1981.6043942790698

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=1.0: 44.3081176457773
Final Loss: 1966.041849925969, Norm of Beta: 44.30687834754911
SciKit-Learn Elastic Net Loss: 94.02498002105693, Norm of Beta: 0.2961849009976596
Difference in Loss: 1872.016869904912, Difference in Norm of Beta: 1981.6048906098126
```

```
Optimality Check with beta_init=ones, alpha=6e-09, lambda=1.0: 44.30812998413857
Final Loss: 1966.0428339531059, Norm of Beta: 44.30688945196467
SciKit-Learn Elastic Net Loss: 94.02497967411985, Norm of Beta: 0.2961848333578181
Difference in Loss: 1872.017854278986, Difference in Norm of Beta: 1981.6053873093185

Optimality Check with beta_init=ones, alpha=5e-09, lambda=1.0: 44.308154663768576
Final Loss: 1966.0448020084048, Norm of Beta: 44.306911660798306
SciKit-Learn Elastic Net Loss: 94.02497963753459, Norm of Beta: 0.296184832824215
Difference in Loss: 1872.01982237087, Difference in Norm of Beta: 1981.606380501648

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.5: 25.553612382542838
Final Loss: 655.8905202934428, Norm of Beta: 25.552850240293658
SciKit-Learn Elastic Net Loss: 94.02497985902917, Norm of Beta: 0.296184970791511
Difference in Loss: 561.8655404344137, Difference in Norm of Beta: 1142.918635086071

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.5: 25.55362247894368
Final Loss: 655.8910122446748, Norm of Beta: 25.552859872865305
SciKit-Learn Elastic Net Loss: 94.02497955595194, Norm of Beta: 0.29618481337733693
Difference in Loss: 561.8660326887228, Difference in Norm of Beta: 1142.9190660446789

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.5: 25.553627527354696
Final Loss: 655.8912582203147, Norm of Beta: 25.55286468915019
SciKit-Learn Elastic Net Loss: 94.02497954178936, Norm of Beta: 0.29618480687678217
Difference in Loss: 561.8662786785254, Difference in Norm of Beta: 1142.919281436713

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.5: 25.553632575908427
Final Loss: 655.8915041960836, Norm of Beta: 25.552869505436657
SciKit-Learn Elastic Net Loss: 94.02497977359926, Norm of Beta: 0.29618498365072143
Difference in Loss: 561.8665244224843, Difference in Norm of Beta: 1142.9194965816368

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.5: 25.553642673440535
Final Loss: 655.891996147832, Norm of Beta: 25.552879138010884
SciKit-Learn Elastic Net Loss: 94.0249797407894, Norm of Beta: 0.2961849900762177
Difference in Loss: 561.8670164070426, Difference in Norm of Beta: 1142.9199273337263

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.6: 25.553596270158053
Final Loss: 655.8897326268891, Norm of Beta: 25.552834827978593
SciKit-Learn Elastic Net Loss: 94.0249796485356, Norm of Beta: 0.29618483460305317
Difference in Loss: 561.8647529783535, Difference in Norm of Beta: 1142.9179460385694

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.6: 25.55360837973475
Final Loss: 655.8903230359865, Norm of Beta: 25.552846387086543
SciKit-Learn Elastic Net Loss: 94.02497987995628, Norm of Beta: 0.2961849691275353
Difference in Loss: 561.8653431560302, Difference in Norm of Beta: 1142.9184627806492

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.6: 25.553614434829413
Final Loss: 655.8906182407071, Norm of Beta: 25.55285216664185
SciKit-Learn Elastic Net Loss: 94.02497984720681, Norm of Beta: 0.2961849728016609
Difference in Loss: 561.8656383935003, Difference in Norm of Beta: 1142.9187212289214

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.6: 25.553620490127386
Final Loss: 655.8909134455165, Norm of Beta: 25.552857946197538
SciKit-Learn Elastic Net Loss: 94.02497955863117, Norm of Beta: 0.29618481099128363
Difference in Loss: 561.8659338868854, Difference in Norm of Beta: 1142.9189798995592

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.6: 25.55363260133274
Final Loss: 655.8915038553415, Norm of Beta: 25.552869505308887
SciKit-Learn Elastic Net Loss: 94.02497977231718, Norm of Beta: 0.29618498424309136
Difference in Loss: 561.8665240830244, Difference in Norm of Beta: 1142.9194965766503

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.8: 25.55356404973015
Final Loss: 655.888157295642, Norm of Beta: 25.552804003355973
SciKit-Learn Elastic Net Loss: 94.02498003367371, Norm of Beta: 0.29618490138940967
Difference in Loss: 561.8631772619683, Difference in Norm of Beta: 1142.9165675199604

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.8: 25.553580184643586
Final Loss: 655.8889446201564, Norm of Beta: 25.552819415537147
SciKit-Learn Elastic Net Loss: 94.02497961668894, Norm of Beta: 0.29618480861652985
Difference in Loss: 561.8639650034675, Difference in Norm of Beta: 1142.9172568576525

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.8: 25.553588252644264
Final Loss: 655.8893382827055, Norm of Beta: 25.552827121629836
```

```
SciKit-Learn Elastic Net Loss: 94.02497961683834, Norm of Beta: 0.29618482311765876
Difference in Loss: 561.8643586658671, Difference in Norm of Beta: 1142.9176014429315

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.8: 25.553596321006598
Final Loss: 655.889731945405, Norm of Beta: 25.552834827723057
SciKit-Learn Elastic Net Loss: 94.02497964737526, Norm of Beta: 0.2961848381572702
Difference in Loss: 561.8647522980297, Difference in Norm of Beta: 1142.9179460315004

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.8: 25.5536124588157
Final Loss: 655.8905192712173, Norm of Beta: 25.552850239910363
SciKit-Learn Elastic Net Loss: 94.0249798551831, Norm of Beta: 0.29618497256862064
Difference in Loss: 561.8655394160342, Difference in Norm of Beta: 1142.9186350711057

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.9: 25.553547941686755
Final Loss: 655.8873696309537, Norm of Beta: 25.552788591048515
SciKit-Learn Elastic Net Loss: 94.0249797088672, Norm of Beta: 0.29618475289336066
Difference in Loss: 561.8623899220864, Difference in Norm of Beta: 1142.9158784971742

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.9: 25.5535660887614
Final Loss: 655.8882554130239, Norm of Beta: 25.552805929766702
SciKit-Learn Elastic Net Loss: 94.02498004286518, Norm of Beta: 0.296184904033128
Difference in Loss: 561.8632753701587, Difference in Norm of Beta: 1142.9166536670625

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.9: 25.553575162984405
Final Loss: 655.8886983043024, Norm of Beta: 25.552814599125984
SciKit-Learn Elastic Net Loss: 94.02497965400997, Norm of Beta: 0.2961847294618287
Difference in Loss: 561.8637186502924, Difference in Norm of Beta: 1142.9170415849765

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.9: 25.553584237666893
Final Loss: 655.8891411958535, Norm of Beta: 25.55282326848756
SciKit-Learn Elastic Net Loss: 94.02497959856568, Norm of Beta: 0.29618481712895256
Difference in Loss: 561.8641615972879, Difference in Norm of Beta: 1142.9174291488507

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.9: 25.55360238840462
Final Loss: 655.8900269795032, Norm of Beta: 25.55284060721227
SciKit-Learn Elastic Net Loss: 94.02497989363233, Norm of Beta: 0.2961849084729705
Difference in Loss: 561.8650470858709, Difference in Norm of Beta: 1142.9182044021586

Optimality Check with beta_init=random, alpha=8e-09, lambda=1.0: 25.553531835092695
Final Loss: 655.8865819669836, Norm of Beta: 25.55277317874548
SciKit-Learn Elastic Net Loss: 94.02498006443454, Norm of Beta: 0.29618492452137174
Difference in Loss: 561.8616019025491, Difference in Norm of Beta: 1142.9151890506457

Optimality Check with beta_init=random, alpha=7e-09, lambda=1.0: 25.55355199398595
Final Loss: 655.887566206318, Norm of Beta: 25.552792443997227
SciKit-Learn Elastic Net Loss: 94.02497969963913, Norm of Beta: 0.2961847614920499
Difference in Loss: 561.8625865066788, Difference in Norm of Beta: 1142.9160507881088

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=1.0: 25.553562074281874
Final Loss: 655.8880583264053, Norm of Beta: 25.552802076625678
SciKit-Learn Elastic Net Loss: 94.02498002105693, Norm of Beta: 0.2961849009976596
Difference in Loss: 561.8630783053484, Difference in Norm of Beta: 1142.9164813613493

Optimality Check with beta_init=random, alpha=6e-09, lambda=1.0: 25.553572155141474
Final Loss: 655.8885504466664, Norm of Beta: 25.55281170925377
SciKit-Learn Elastic Net Loss: 94.02497967411985, Norm of Beta: 0.2961848333578181
Difference in Loss: 561.8635707725465, Difference in Norm of Beta: 1142.9169121994694

Optimality Check with beta_init=random, alpha=5e-09, lambda=1.0: 25.5535923185604
Final Loss: 655.8895346881023, Norm of Beta: 25.552830974516542
SciKit-Learn Elastic Net Loss: 94.02497963753459, Norm of Beta: 0.296184832824215
Difference in Loss: 561.8645550505678, Difference in Norm of Beta: 1142.9177737322652
```

Comparison between Proximal Gradient Method & Built-In Scikit-Learn Elastic Net Solution

In [45]:

```
f, ax = plt.subplots(2, 3, figsize=(40, 10))
```

```
for b, beta_init in enumerate(beta_init_labels):
    # Get loss differences for current beta_init

    current_loss_differences = [loss_difference[0] for loss_difference in loss_differenc
es if loss_difference[-1] == beta_init]

    current_loss_differences_labels = [(loss_differences_label.split(",")[0].replace("("
,""), loss_differences_label.split(",")[1].replace(" '","")) for
                                       loss_differences_label in loss_differences_labels
if loss_differences_label.split(",")[-1].replace("')","").replace(" '","") == beta_init]

    ax[0, b].scatter(range(len(current_loss_differences_labels)), current_loss_differenc
es, c = "cyan", edgecolors = "black", s=200, marker="*", label=beta_init)
    ax[0, b].legend(loc="upper right", prop={'size': 12})
    ax[0, b].set_xticks(np.arange(len(current_loss_differences_labels)))
    ax[0, b].set_xlabel(r"Parameter Combination Index ($\alpha$, $\lambda$)", fontsize=1
2)
    ax[0, b].set_xticklabels(labels=current_loss_differences_labels, rotation=60, fontsi
ze=16)

    current_beta_norm_differences = [beta_norm_difference[0] for beta_norm_difference in
beta_norm_differences if beta_norm_difference[-1] == beta_init]

    current_beta_norm_differences_labels = [(beta_norm_differences_label.split(",")[0].r
eplace("(",""), beta_norm_differences_label.split(",")[1].replace(" '",""))
                                            for beta_norm_differences_label in beta_norm
_differences_labels if beta_norm_differences_label.split(",")[-1].replace("')","").repla
ce(" '","") == beta_init]

    ax[1, b].scatter(range(len(current_beta_norm_differences_labels)), current_beta_norm
_differences, c = "r", edgecolors = "black", s=200, marker="*", label=beta_init)
    ax[1, b].legend(loc="upper right", prop={'size': 12})
    ax[1, b].set_xticks(np.arange(len(current_beta_norm_differences_labels)))
    ax[1, b].set_xlabel(r"Parameter Combination Index ($\alpha$, $\lambda$)", fontsize=1
2)
    ax[1, b].set_xticklabels(labels=current_beta_norm_differences_labels, rotation=60, f
ontsize=16)

plt.subplots_adjust(top = 0.99, bottom=0.1, hspace=1.0, wspace=1.0)

ax[1, int(math.floor(len(beta_init_labels) / 2))].set_title("Difference in Beta Norm betw
een Proximal Gradient and SciKit-Learn", fontsize=30)
ax[0, int(math.floor(len(beta_init_labels) / 2))].set_title("Loss Difference between Prox
imal Gradient and SciKit-Lear", fontsize=30)

ax[1, 0].set_ylabel("Difference in Beta Norm", fontsize=14)
ax[0, 0].set_ylabel("Difference in Loss", fontsize=14)

f.tight_layout()
plt.show()
```