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 socio-economic 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 $n \times 1$ response vector
- X is the $n \times p$ design matrix
- β is the $p \times 1$ coefficient vector
- λ is the regularization parameter

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,

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{3}$$
$$(\lambda I + X^TXI_n)^{-1} = \lambda^{-1}I - \lambda^{-1}IX^T(I_n^{-1} + X^TX\lambda^{-1}I)^{-1}X\lambda^{-1}I \tag{4}$$
$$(\lambda I + X^TX)^{-1} = \lambda^{-1}I - \lambda^{-1}X^T(I_n^{-1} + X^TX\lambda^{-1})^{-1}X\lambda^{-1} \tag{5}$$

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

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

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 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:

$$P(\beta) = \alpha||\beta||_1 + (1-\alpha)||\beta||_2^2 \tag{7}$$

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:

$$L(\beta) = \frac{1}{2}||y - X\beta||_2^2 + \alpha||\beta||_1 + (1-\alpha)||\beta||_2^2 \tag{8}$$

2.2.1 Using the Proximal Gradient Method:

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

$$\nabla L(\beta) = -X^T(y - X\beta) + (1-\alpha)\beta \tag{9}$$

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

$$prox_{\alpha\lambda}(\beta) = sign(\beta)(|\beta| - \alpha\lambda)_+ \tag{10}$$

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:

$$\beta^{(k+1)} = prox_{\alpha\lambda}\left(\beta^{(k)} - t_k\nabla L(\beta^{(k)})\right) \tag{11}$$

2.2.2 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#, load_boston
from sklearn.feature_selection import RFE
from sklearn.linear_model import ElasticNet, ElasticNetCV, enet_path, Lasso, LassoCV, la
sso_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'` i
n 1.4. You can set `parser='auto'` to silence this warning. Therefore, an `ImportError` w
ill 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 AP
I doc for details.
  warn(
```

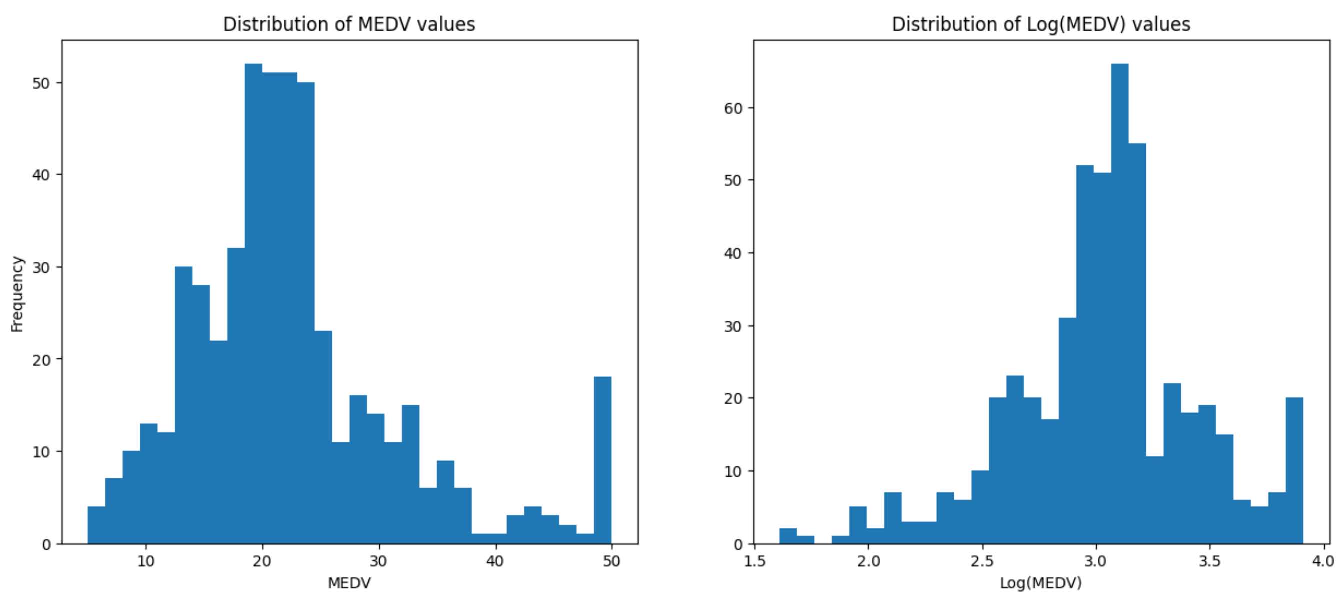
In [4]:

```
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

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:	Sat, 09 Sep 2023	**Prob (F-statistic):**	0.00
Time:	18:21:26	**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: 1.6147982729874112e-27
Best model with 2 features: (4, 12), RSS: 4.141519752410312e-28
Best model with 3 features: (0, 10, 12), RSS: 3.218552493301728e-28
Best model with 4 features: (1, 3, 7, 12), RSS: 3.7155348635909656e-28
Best model with 5 features: (1, 3, 4, 9, 12), RSS: 3.747089299799806e-28
Best model with 6 features: (1, 2, 3, 9, 10, 12), RSS: 4.1730741886191525e-28
Best model with 7 features: (0, 1, 3, 4, 9, 11, 12), RSS: 5.222259192563098e-28
Best model with 8 features: (0, 1, 5, 7, 9, 10, 11, 12), RSS: 6.137337842619472e-28
Best model with 9 features: (1, 2, 5, 6, 7, 8, 9, 10, 12), RSS: 6.571211340491028e-28
Best model with 10 features: (1, 2, 3, 4, 5, 6, 7, 9, 10, 12), RSS: 7.131302583197947e-28
Best model with 11 features: (0, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), RSS: 1.173036166063644(
e-27
Best model with 12 features: (0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12), RSS: 2.407603482734
28e-27
Best model with 13 features: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), RSS: 1.88379984l(
677762e-27
```

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=0.01, 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([ 9.96803041e-03, -0.00000000e+00,  5.64271796e-03, -0.00000000e+00,
         3.21128493e-03, -3.99302519e-02,  2.95273443e-02, -0.00000000e+00,
         1.50680665e-03,  5.16916918e-03,  0.00000000e+00, -7.89663751e-03,
         7.03524504e+00]),
 array([ 9.96803041e-03, -0.00000000e+00,  5.64271796e-03, -0.00000000e+00,
         3.21128493e-03, -3.99302519e-02,  2.95273443e-02, -0.00000000e+00,
         1.50680665e-03,  5.16916918e-03,  0.00000000e+00, -7.89663751e-03,
         7.03524504e+00]))
```

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

In [13]:

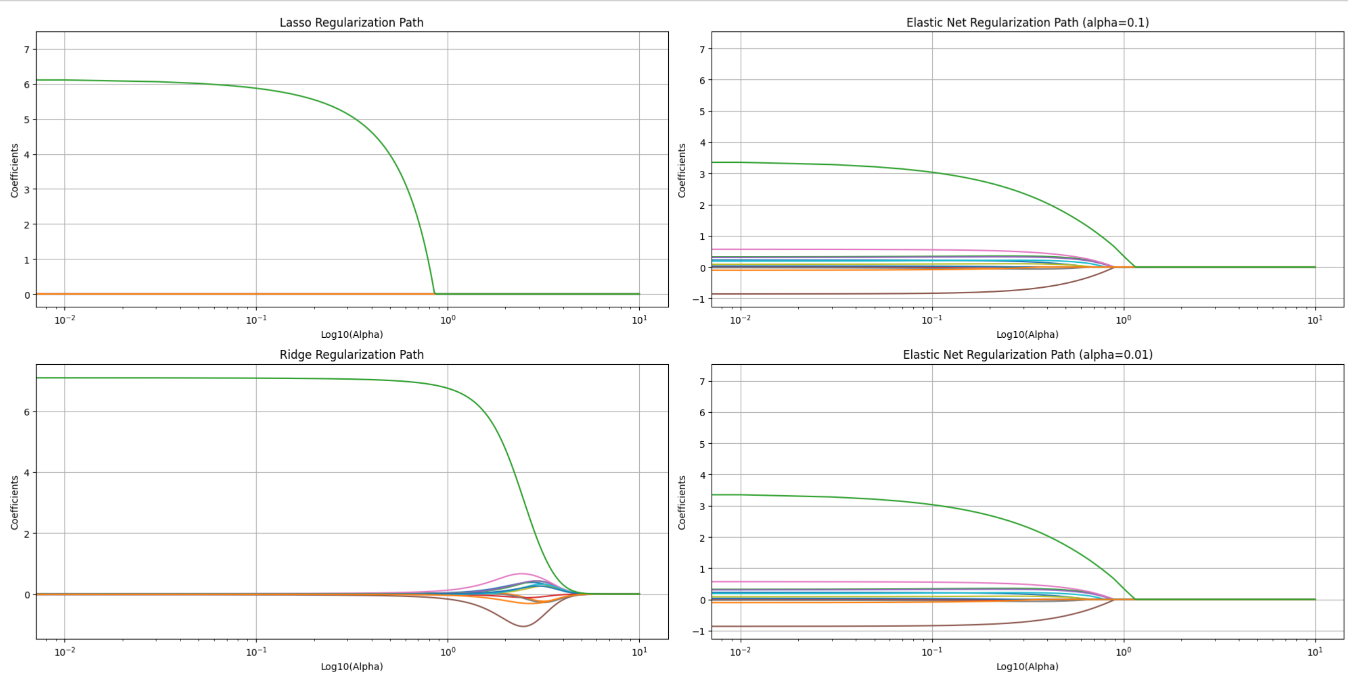
```
# alpha range (regularization strengths)
# alphas = np.logspace(-10, 10, 1000)

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',
 0.08564023621496919,
 'centered',
 0.07960314637583368,
 'ones',
 0.08760836893977858,
 'lr',
 0.08761743248985587,
 '\n',
 array([-0.00392898,  0.00091399, -0.00131138, ..., -0.00219029,
        -0.00243668,  0.00030878]),
 array([-0.00392898,  0.00091399, -0.00131138, ..., -0.00219029,
        -0.00243668,  0.00030878]),
 array([-0.00261934, -0.00011959,  0.00030385, ..., -0.00148804,
        -0.00183681,  0.00105631]))
```

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]:

```
(0.08564023621496919,
 array([0.90056032, 0.42855301, 0.25021507, 0.87797942]),
 array([0.53526879, 0.54253581, 0.53221042, 0.53649795]))
```

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:250
: UserWarning: Singular matrix in solving dual problem. Using least-squares solution inst
ead.
  warnings.warn(
/Users/gonz495/miniconda3/lib/python3.10/site-packages/sklearn/linear_model/_ridge.py:250
: UserWarning: Singular matrix in solving dual problem. Using least-squares solution inst
ead.
  warnings.warn(
```

Out[26]:

```
(0.08564023621496919, 0.0832484292389509, 0.0, 0.08324844899672901, 0.001)
```

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]:

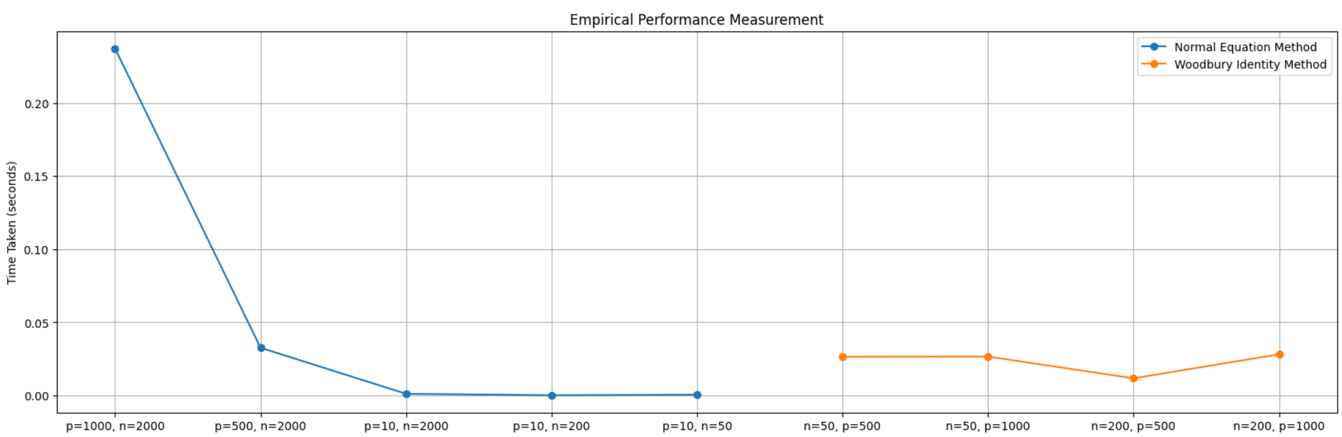
```
results, _ = evaluate_ridge_regression_efficiency(n_values_ridge, p_values_ridge)

normal_eq_xlabels = [res["label"] for res in results if res["method"] == "Normal Equation"]
normal_eq_times = [res["time_taken"] for res in results if res["method"] == "Normal Equation"]

woobdury_xlabels = [res["label"] for res in results if res["method"] == "Woodbury Identity"]
woodbury_times = [res["time_taken"] for res in results if res["method"] == "Woodbury Identity"]

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: 0.000526 seconds
For (n=50, p=500) using Woodbury Identity, time taken: 0.026406 seconds
For (n=50, p=1000) using Woodbury Identity, time taken: 0.026592 seconds
For (p=10, n=200) using Normal Equation, time taken: 0.000187 seconds
For (n=200, p=500) using Woodbury Identity, time taken: 0.011740 seconds
For (n=200, p=1000) using Woodbury Identity, time taken: 0.028130 seconds
For (p=10, n=2000) using Normal Equation, time taken: 0.001130 seconds
For (p=500, n=2000) using Normal Equation, time taken: 0.032608 seconds
For (p=1000, n=2000) using Normal Equation, time taken: 0.237188 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()
```



Lasso Regression Computation

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

method

In [33]:

```
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 [34]:

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

In [35]:

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

In [36]:

```
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 [37]:

```
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 [38]:

```
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: 3.090688804107309
```

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

In [39]:

```
# 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"]), r
esult["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", labe
l="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: 0.01682829290946102
Final Loss: 3.0906888463948654, Norm of Beta: 0.031286748388696606
SciKit-Learn Elastic Net Loss: 102.8887750994294, Norm of Beta: 0.3201252608418578
Difference in Loss: 99.79808625303453, Difference in Norm of Beta: 14.38457366931369

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.5: 0.0150249994351029
Final Loss: 3.0906888579490865, Norm of Beta: 0.03128760895315281
SciKit-Learn Elastic Net Loss: 102.88877515474135, Norm of Beta: 0.32012527649968936
Difference in Loss: 99.79808629679226, Difference in Norm of Beta: 14.384578052165162

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.5: 0.014124281830905564
Final Loss: 3.090688864350187, Norm of Beta: 0.03128803972851272
SciKit-Learn Elastic Net Loss: 102.88877515682665, Norm of Beta: 0.3201252642181919
Difference in Loss: 99.79808629247647, Difference in Norm of Beta: 14.384579350767668

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.5: 0.013224350979570438
Final Loss: 3.090688871167365, Norm of Beta: 0.03128847083438819
SciKit-Learn Elastic Net Loss: 102.88877516975515, Norm of Beta: 0.3201252736562084
Difference in Loss: 99.79808629858779, Difference in Norm of Beta: 14.384581617558245

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.5: 0.01142759189887546
Final Loss: 3.0906888860496835, Norm of Beta: 0.03128933403332638
SciKit-Learn Elastic Net Loss: 102.88877522368988, Norm of Beta: 0.3201252771838597
Difference in Loss: 99.7980863376402, Difference in Norm of Beta: 14.384585472363346

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.6: 0.019717137666938857
Final Loss: 3.090688829562088, Norm of Beta: 0.03128537420305891
SciKit-Learn Elastic Net Loss: 102.88877504807216, Norm of Beta: 0.32012526772568667
Difference in Loss: 99.79808621851008, Difference in Norm of Beta: 14.384568089694191

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.6: 0.017550168658278728
Final Loss: 3.0906888406578066, Norm of Beta: 0.03128640453142181
SciKit-Learn Elastic Net Loss: 102.88877510253538, Norm of Beta: 0.32012525683516796
Difference in Loss: 99.79808626187757, Difference in Norm of Beta: 14.384572018229358

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.6: 0.016467465330264414
Final Loss: 3.0906888471040745, Norm of Beta: 0.03128692039581223
SciKit-Learn Elastic Net Loss: 102.88877510080658, Norm of Beta: 0.3201252660464278
Difference in Loss: 99.79808625370251, Difference in Norm of Beta: 14.384574637719425
```

```
Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.6: 0.0153854756317249
Final Loss: 3.090688854149418, Norm of Beta: 0.03128743673498959
SciKit-Learn Elastic Net Loss: 102.88877512697223, Norm of Beta: 0.32012527526249673
Difference in Loss: 99.79808627282281, Difference in Norm of Beta: 14.38457725933263

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.6: 0.013224350979554116
Final Loss: 3.090688870037385, Norm of Beta: 0.03128847083437314
SciKit-Learn Elastic Net Loss: 102.88877516900565, Norm of Beta: 0.3201252755769596
Difference in Loss: 99.79808629896827, Difference in Norm of Beta: 14.38458170305042

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.8: 0.025503150112019257
Final Loss: 3.0906888086758246, Norm of Beta: 0.03128263496561336
SciKit-Learn Elastic Net Loss: 102.8887749631134, Norm of Beta: 0.32012526967528493
Difference in Loss: 99.79808615443758, Difference in Norm of Beta: 14.384556437218148

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.8: 0.0226090098153319
Final Loss: 3.0906888158624506, Norm of Beta: 0.03128400295850789
SciKit-Learn Elastic Net Loss: 102.88877521279466, Norm of Beta: 0.3201253196239209
Difference in Loss: 99.7980863969322, Difference in Norm of Beta: 14.384564523137382

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.8: 0.021162694939054347
Final Loss: 3.090688821051762, Norm of Beta: 0.031284688228634325
SciKit-Learn Elastic Net Loss: 102.8887750113666, Norm of Beta: 0.3201252602225368
Difference in Loss: 99.79808619031485, Difference in Norm of Beta: 14.384564816219996

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.8: 0.01971713766691742
Final Loss: 3.090688827303557, Norm of Beta: 0.0312853742030288
SciKit-Learn Elastic Net Loss: 102.88877504575328, Norm of Beta: 0.3201252702866886
Difference in Loss: 99.79808621844973, Difference in Norm of Beta: 14.38456820368382

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.8: 0.016828292909416898
Final Loss: 3.0906888430061192, Norm of Beta: 0.03128674838865144
SciKit-Learn Elastic Net Loss: 102.88877509656596, Norm of Beta: 0.3201252656437349
Difference in Loss: 99.79808625355984, Difference in Norm of Beta: 14.384573883044176

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=0.9: 0.028398568143196003
Final Loss: 3.090688804621678, Norm of Beta: 0.031281270449385876
SciKit-Learn Elastic Net Loss: 102.88877485568014, Norm of Beta: 0.3201252553490802
Difference in Loss: 99.79808605105846, Difference in Norm of Beta: 14.384549952260631

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=0.9: 0.02514129988803266
Final Loss: 3.09068880835356, Norm of Beta: 0.0312828057747401
SciKit-Learn Elastic Net Loss: 102.8887749704124, Norm of Beta: 0.3201252724074867
Difference in Loss: 99.79808616205884, Difference in Norm of Beta: 14.384557290701009

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=0.9: 0.023513252365016325
Final Loss: 3.0906888122419036, Norm of Beta: 0.03128357508815212
SciKit-Learn Elastic Net Loss: 102.8887750505039, Norm of Beta: 0.32012528078629005
Difference in Loss: 99.79808623826199, Difference in Norm of Beta: 14.384560960832816

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=0.9: 0.021885756089684485
Final Loss: 3.09068881747742, Norm of Beta: 0.03128434549367284
SciKit-Learn Elastic Net Loss: 102.88877522510433, Norm of Beta: 0.32012532271173405
Difference in Loss: 99.79808640762691, Difference in Norm of Beta: 14.384566128636653

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=0.9: 0.018633354481055412
Final Loss: 3.090688831985798, Norm of Beta: 0.03128588914236106
SciKit-Learn Elastic Net Loss: 102.88877509733602, Norm of Beta: 0.32012527862124407
Difference in Loss: 99.79808626535022, Difference in Norm of Beta: 14.384570780641097

Optimality Check with beta_init=zeros, alpha=8e-09, lambda=1.0: 0.031294876109196776
Final Loss: 3.0906888048285523, Norm of Beta: 0.03127990936847964
SciKit-Learn Elastic Net Loss: 102.88877489443792, Norm of Beta: 0.3201252417599258
Difference in Loss: 99.79808608960937, Difference in Norm of Beta: 14.384543515779734

Optimality Check with beta_init=zeros, alpha=7e-09, lambda=1.0: 0.027674617404392934
Final Loss: 3.090688804107309, Norm of Beta: 0.0312816112552798
SciKit-Learn Elastic Net Loss: 102.88877487362808, Norm of Beta: 0.32012525988218454
Difference in Loss: 99.79808606952076, Difference in Norm of Beta: 14.384551614402012

Optimality Check with beta_init=zeros, alpha=6.5e-09, lambda=1.0: 0.025865021114283192
```

```
Final Loss: 3.090688806243193, Norm of Beta: 0.031282464212068734
SciKit-Learn Elastic Net Loss: 102.88877495410176, Norm of Beta: 0.3201252693419505
Difference in Loss: 99.79808614785857, Difference in Norm of Beta: 14.384555690529968

Optimality Check with beta_init=zeros, alpha=6e-09, lambda=1.0: 0.024055883366628453
Final Loss: 3.090688810043559, Norm of Beta: 0.03128331852835967
SciKit-Learn Elastic Net Loss: 102.88877502068, Norm of Beta: 0.3201252786515918
Difference in Loss: 99.79808621063644, Difference in Norm of Beta: 14.38455976634392

Optimality Check with beta_init=zeros, alpha=5e-09, lambda=1.0: 0.02043989512345741
Final Loss: 3.0906888226328757, Norm of Beta: 0.03128503113767621
SciKit-Learn Elastic Net Loss: 102.8887750238553, Norm of Beta: 0.32012526675725134
Difference in Loss: 99.79808620122243, Difference in Norm of Beta: 14.384566576705316

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.5: 44.18182748910278
Final Loss: 1955.013314362749, Norm of Beta: 44.180579155410285
SciKit-Learn Elastic Net Loss: 102.8887750994294, Norm of Beta: 0.3201252608418578
Difference in Loss: 1852.1245392633195, Difference in Norm of Beta: 1975.9674033507567

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.5: 44.18184029259812
Final Loss: 1955.014292781831, Norm of Beta: 44.18059022805774
SciKit-Learn Elastic Net Loss: 102.88877515474135, Norm of Beta: 0.32012527649968936
Difference in Loss: 1852.1255176270897, Difference in Norm of Beta: 1975.9678985570115

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.5: 44.181846694946444
Final Loss: 1955.014781991364, Norm of Beta: 44.180595764380186
SciKit-Learn Elastic Net Loss: 102.88877515682665, Norm of Beta: 0.3201252642181919
Difference in Loss: 1852.1260068345375, Difference in Norm of Beta: 1975.9681461169841

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.5: 44.18185309769972
Final Loss: 1955.0152712010806, Norm of Beta: 44.18060130070392
SciKit-Learn Elastic Net Loss: 102.88877516975515, Norm of Beta: 0.3201252736562084
Difference in Loss: 1852.1264960313256, Difference in Norm of Beta: 1975.9683937215088

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.5: 44.18186590441305
Final Loss: 1955.0162496207213, Norm of Beta: 44.180612373351366
SciKit-Learn Elastic Net Loss: 102.88877522368988, Norm of Beta: 0.3201252771838597
Difference in Loss: 1852.1274743970314, Difference in Norm of Beta: 1975.9688889023355

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.6: 44.18180707719323
Final Loss: 1955.0117488462467, Norm of Beta: 44.180561438824576
SciKit-Learn Elastic Net Loss: 102.88877504807216, Norm of Beta: 0.32012526772568667
Difference in Loss: 1852.1229737981746, Difference in Norm of Beta: 1975.9666110718886

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.6: 44.1818224296971
Final Loss: 1955.0129229543895, Norm of Beta: 44.180574726044426
SciKit-Learn Elastic Net Loss: 102.88877510253538, Norm of Beta: 0.32012525683516796
Difference in Loss: 1852.124147851854, Difference in Norm of Beta: 1975.9672052541741

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.6: 44.18183010681903
Final Loss: 1955.0135100084576, Norm of Beta: 44.1805813696526
SciKit-Learn Elastic Net Loss: 102.88877510080658, Norm of Beta: 0.3201252660464278
Difference in Loss: 1852.1247349076511, Difference in Norm of Beta: 1975.967502378311

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.6: 44.18183778452041
Final Loss: 1955.014097062837, Norm of Beta: 44.18058801326316
SciKit-Learn Elastic Net Loss: 102.88877512697223, Norm of Beta: 0.32012527526249673
Difference in Loss: 1852.1253219358648, Difference in Norm of Beta: 1975.9677995024103

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.6: 44.181853141658145
Final Loss: 1955.015271171785, Norm of Beta: 44.18060130048301
SciKit-Learn Elastic Net Loss: 102.88877516900565, Norm of Beta: 0.3201252755769596
Difference in Loss: 1852.1264960027793, Difference in Norm of Beta: 1975.9683937114903

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.8: 44.18176626572779
Final Loss: 1955.0086178148476, Norm of Beta: 44.18052600564711
SciKit-Learn Elastic Net Loss: 102.8887749631134, Norm of Beta: 0.32012526967528493
Difference in Loss: 1852.1198428517341, Difference in Norm of Beta: 1975.9650264881902

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.8: 44.18178671335802
Final Loss: 1955.0101833008366, Norm of Beta: 44.180543722014306
SciKit-Learn Elastic Net Loss: 102.88877521279466, Norm of Beta: 0.3201253196239209
```

```
Difference in Loss: 1852.121408088042, Difference in Norm of Beta: 1975.9658188815204

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.8: 44.181796938719145
Final Loss: 1955.010966044155, Norm of Beta: 44.18055258019854
SciKit-Learn Elastic Net Loss: 102.8887750113666, Norm of Beta: 0.3201252602225368
Difference in Loss: 1852.1221910327886, Difference in Norm of Beta: 1975.9662148876482

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.8: 44.18180716510997
Final Loss: 1955.011748787652, Norm of Beta: 44.18056143838277
SciKit-Learn Elastic Net Loss: 102.88877504575328, Norm of Beta: 0.3201252702866886
Difference in Loss: 1852.122973741899, Difference in Norm of Beta: 1975.966611051922

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.8: 44.181827620977955
Final Loss: 1955.013314274859, Norm of Beta: 44.18057915474757
SciKit-Learn Elastic Net Loss: 102.88877509656596, Norm of Beta: 0.3201252656437349
Difference in Loss: 1852.1245391782932, Difference in Norm of Beta: 1975.9674033207327

Optimality Check with beta_init=ones, alpha=8e-09, lambda=0.9: 44.181745866172406
Final Loss: 1955.007052300373, Norm of Beta: 44.180508289060135
SciKit-Learn Elastic Net Loss: 102.88877485568014, Norm of Beta: 0.3201252553490802
Difference in Loss: 1852.118277444693, Difference in Norm of Beta: 1975.9642341664

Optimality Check with beta_init=ones, alpha=7e-09, lambda=0.9: 44.18176885992022
Final Loss: 1955.0088134749344, Norm of Beta: 44.18052821999987
SciKit-Learn Elastic Net Loss: 102.8887749704124, Norm of Beta: 0.3201252724074867
Difference in Loss: 1852.120038504522, Difference in Norm of Beta: 1975.9651255194078

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=0.9: 44.18178035874643
Final Loss: 1955.009694062548, Norm of Beta: 44.18053818546967
SciKit-Learn Elastic Net Loss: 102.8887750505039, Norm of Beta: 0.32012528078629005
Difference in Loss: 1852.1209190120442, Difference in Norm of Beta: 1975.9655711951848

Optimality Check with beta_init=ones, alpha=6e-09, lambda=0.9: 44.18179185887866
Final Loss: 1955.0105746504996, Norm of Beta: 44.18054815094074
SciKit-Learn Elastic Net Loss: 102.88877522510433, Norm of Beta: 0.32012532271173405
Difference in Loss: 1852.1217994253952, Difference in Norm of Beta: 1975.96601694979

Optimality Check with beta_init=ones, alpha=5e-09, lambda=0.9: 44.18181486305018
Final Loss: 1955.012335826969, Norm of Beta: 44.1805680818816
SciKit-Learn Elastic Net Loss: 102.88877509733602, Norm of Beta: 0.32012527862124407
Difference in Loss: 1852.123560729633, Difference in Norm of Beta: 1975.9669081665716

Optimality Check with beta_init=ones, alpha=8e-09, lambda=1.0: 44.181725470738414
Final Loss: 1955.0054867865126, Norm of Beta: 44.18049057247204
SciKit-Learn Elastic Net Loss: 102.88877489443792, Norm of Beta: 0.3201252417599258
Difference in Loss: 1852.1167118920748, Difference in Norm of Beta: 1975.963441849767

Optimality Check with beta_init=ones, alpha=7e-09, lambda=1.0: 44.18175100963349
Final Loss: 1955.0074436496777, Norm of Beta: 44.180512717986566
SciKit-Learn Elastic Net Loss: 102.88877487362808, Norm of Beta: 0.32012525988218454
Difference in Loss: 1852.1186687760496, Difference in Norm of Beta: 1975.9644322354382

Optimality Check with beta_init=ones, alpha=6.5e-09, lambda=1.0: 44.18176378149561
Final Loss: 1955.008422081524, Norm of Beta: 44.18052379074208
SciKit-Learn Elastic Net Loss: 102.88877495410176, Norm of Beta: 0.3201252693419505
Difference in Loss: 1852.1196471274222, Difference in Norm of Beta: 1975.9649274332066

Optimality Check with beta_init=ones, alpha=6e-09, lambda=1.0: 44.181776554964216
Final Loss: 1955.0094005137487, Norm of Beta: 44.180534863498714
SciKit-Learn Elastic Net Loss: 102.88877502068, Norm of Beta: 0.3201252786515918
Difference in Loss: 1852.1206254930687, Difference in Norm of Beta: 1975.9654226294963

Optimality Check with beta_init=ones, alpha=5e-09, lambda=1.0: 44.18180210673383
Final Loss: 1955.0113573792587, Norm of Beta: 44.180557009014514
SciKit-Learn Elastic Net Loss: 102.8887750238553, Norm of Beta: 0.32012526675725134
Difference in Loss: 1852.1225823554034, Difference in Norm of Beta: 1975.9664129559767

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.5: 25.90450086918147
Final Loss: 674.1066593254061, Norm of Beta: 25.903994650298955
SciKit-Learn Elastic Net Loss: 102.8887750994294, Norm of Beta: 0.3201252608418578
Difference in Loss: 571.2178842259767, Difference in Norm of Beta: 1158.6372876324049
```

```
Optimality Check with beta_init=random, alpha=7e-09, lambda=0.5: 25.90451081220629
Final Loss: 674.1071592253986, Norm of Beta: 25.90400430574713
SciKit-Learn Elastic Net Loss: 102.88877515474135, Norm of Beta: 0.32012527649968936
Difference in Loss: 571.2183840706573, Difference in Norm of Beta: 1158.6377194481076

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.5: 25.904515784236832
Final Loss: 674.10740917542, Norm of Beta: 25.90400913347023
SciKit-Learn Elastic Net Loss: 102.88877515682665, Norm of Beta: 0.320125264218919
Difference in Loss: 571.2186340185933, Difference in Norm of Beta: 1158.6379353129712

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.5: 25.904520756615174
Final Loss: 674.107659125575, Norm of Beta: 25.904013961194924
SciKit-Learn Elastic Net Loss: 102.88877516975515, Norm of Beta: 0.3201252736562084
Difference in Loss: 571.2188839558198, Difference in Norm of Beta: 1158.6381512224996

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.5: 25.904530702411538
Final Loss: 674.1081590261023, Norm of Beta: 25.904023616645567
SciKit-Learn Elastic Net Loss: 102.88877522368988, Norm of Beta: 0.3201252771838597
Difference in Loss: 571.2193838024124, Difference in Norm of Beta: 1158.6385830128947

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.6: 25.904485004463716
Final Loss: 674.1058589439556, Norm of Beta: 25.903979201378515
SciKit-Learn Elastic Net Loss: 102.88877504807216, Norm of Beta: 0.32012526772568667
Difference in Loss: 571.2170838958834, Difference in Norm of Beta: 1158.6365967849497

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.6: 25.90449692844297
Final Loss: 674.10645889116, Norm of Beta: 25.903990787938728
SciKit-Learn Elastic Net Loss: 102.88877510253538, Norm of Beta: 0.32012525683516796
Difference in Loss: 571.2176837886245, Difference in Norm of Beta: 1158.6371148983428

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.6: 25.904502891181732
Final Loss: 674.1067588649414, Norm of Beta: 25.903996581220174
SciKit-Learn Elastic Net Loss: 102.88877510080658, Norm of Beta: 0.3201252660464278
Difference in Loss: 571.2179837641348, Difference in Norm of Beta: 1158.6373739878447

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.6: 25.90450885441916
Final Loss: 674.1070588388133, Norm of Beta: 25.904002374501953
SciKit-Learn Elastic Net Loss: 102.88877512697223, Norm of Beta: 0.32012527526249673
Difference in Loss: 571.218283711841, Difference in Norm of Beta: 1158.6376330772343

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.6: 25.90452078238911
Final Loss: 674.1076587867713, Norm of Beta: 25.904013961065402
SciKit-Learn Elastic Net Loss: 102.88877516900565, Norm of Beta: 0.3201252755769596
Difference in Loss: 571.2188836177656, Difference in Norm of Beta: 1158.6381512170922

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.8: 25.90445328566857
Final Loss: 674.1042581829869, Norm of Beta: 25.90394830354477
SciKit-Learn Elastic Net Loss: 102.8887749631134, Norm of Beta: 0.32012526967528493
Difference in Loss: 571.2154832198735, Difference in Norm of Beta: 1158.6352150647879

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.8: 25.904469169065475
Final Loss: 674.1050582242876, Norm of Beta: 25.903963752329812
SciKit-Learn Elastic Net Loss: 102.88877521279466, Norm of Beta: 0.3201253196239209
Difference in Loss: 571.216283011493, Difference in Norm of Beta: 1158.635906029653

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.8: 25.904477112095204
Final Loss: 674.1054582452401, Norm of Beta: 25.903971476724397
SciKit-Learn Elastic Net Loss: 102.8887750113666, Norm of Beta: 0.3201252602225368
Difference in Loss: 571.2166832338735, Difference in Norm of Beta: 1158.6362513235688

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.8: 25.904485056011534
Final Loss: 674.1058582663482, Norm of Beta: 25.903979201119473
SciKit-Learn Elastic Net Loss: 102.88877504575328, Norm of Beta: 0.3201252702866886
Difference in Loss: 571.2170832205949, Difference in Norm of Beta: 1158.6365967738896

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.8: 25.904500946503244
Final Loss: 674.106658308995, Norm of Beta: 25.903994649910395
SciKit-Learn Elastic Net Loss: 102.88877509656596, Norm of Beta: 0.3201252656437349
Difference in Loss: 571.217883212429, Difference in Norm of Beta: 1158.6372876159944

Optimality Check with beta_init=random, alpha=8e-09, lambda=0.9: 25.904437431591013
Final Loss: 674.1034578034694, Norm of Beta: 25.903932854631485
```

```
SciKit-Learn Elastic Net Loss: 102.88877485568014, Norm of Beta: 0.3201252553490802
Difference in Loss: 571.2146829477892, Difference in Norm of Beta: 1158.6345241746114

Optimality Check with beta_init=random, alpha=7e-09, lambda=0.9: 25.904455293451377
Final Loss: 674.1043578916621, Norm of Beta: 25.90395023452946
SciKit-Learn Elastic Net Loss: 102.8887749704124, Norm of Beta: 0.3201252724074867
Difference in Loss: 571.2155829212496, Difference in Norm of Beta: 1158.6353014212666

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=0.9: 25.904464226063734
Final Loss: 674.1048079360091, Norm of Beta: 25.903958924478516
SciKit-Learn Elastic Net Loss: 102.8887750505039, Norm of Beta: 0.32012528078629005
Difference in Loss: 571.2160328855052, Difference in Norm of Beta: 1158.6356900439562

Optimality Check with beta_init=random, alpha=6e-09, lambda=0.9: 25.90447315979991
Final Loss: 674.1052579806393, Norm of Beta: 25.90396761442986
SciKit-Learn Elastic Net Loss: 102.88877522510433, Norm of Beta: 0.32012532271173405
Difference in Loss: 571.216482755535, Difference in Norm of Beta: 1158.6360787443102

Optimality Check with beta_init=random, alpha=5e-09, lambda=0.9: 25.904491030637992
Final Loss: 674.1061580704657, Norm of Beta: 25.90398499433393
SciKit-Learn Elastic Net Loss: 102.88877509733602, Norm of Beta: 0.32012527862124407
Difference in Loss: 571.2173829731296, Difference in Norm of Beta: 1158.6368558561373

Optimality Check with beta_init=random, alpha=8e-09, lambda=1.0: 25.90442158106237
Final Loss: 674.1026574246978, Norm of Beta: 25.903917405722545
SciKit-Learn Elastic Net Loss: 102.88877489443792, Norm of Beta: 0.3201252417599258
Difference in Loss: 571.2138825302599, Difference in Norm of Beta: 1158.6338332893813

Optimality Check with beta_init=random, alpha=7e-09, lambda=1.0: 25.904441420551567
Final Loss: 674.1036575594765, Norm of Beta: 25.903936716729902
SciKit-Learn Elastic Net Loss: 102.88877487362808, Norm of Beta: 0.32012525988218454
Difference in Loss: 571.2148826858484, Difference in Norm of Beta: 1158.6346968902562

Optimality Check with beta_init=random, alpha=6.5e-09, lambda=1.0: 25.904451342375655
Final Loss: 674.1041576273025, Norm of Beta: 25.903946372236124
SciKit-Learn Elastic Net Loss: 102.88877495410176, Norm of Beta: 0.3201252693419505
Difference in Loss: 571.2153826732007, Difference in Norm of Beta: 1158.6351286952818

Optimality Check with beta_init=random, alpha=6e-09, lambda=1.0: 25.904461265583627
Final Loss: 674.1046576953077, Norm of Beta: 25.903956027741874
SciKit-Learn Elastic Net Loss: 102.88877502068, Norm of Beta: 0.3201252786515918
Difference in Loss: 571.2158826746277, Difference in Norm of Beta: 1158.6355604989083

Optimality Check with beta_init=random, alpha=5e-09, lambda=1.0: 25.904481116159804
Final Loss: 674.1056578322632, Norm of Beta: 25.903975338759842
SciKit-Learn Elastic Net Loss: 102.8887750238553, Norm of Beta: 0.32012526675725134
Difference in Loss: 571.216882808408, Difference in Norm of Beta: 1158.6364240406178
```

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

In [40]:

```
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()
```