Code Report

Assignment 1

Machine Learning Course

By: Katayoon Kobraei - 99222084

Exercise 5: Implement Linear Regression with Mean Absolute Error as the cost function from scratch. Compare your results with the Linear Regression module of Scikit-Learn.

As expected, the results from the Scikit-learn library were better and more accurate compared to our model. When our simple linear regression model was implemented, the following results were obtained:

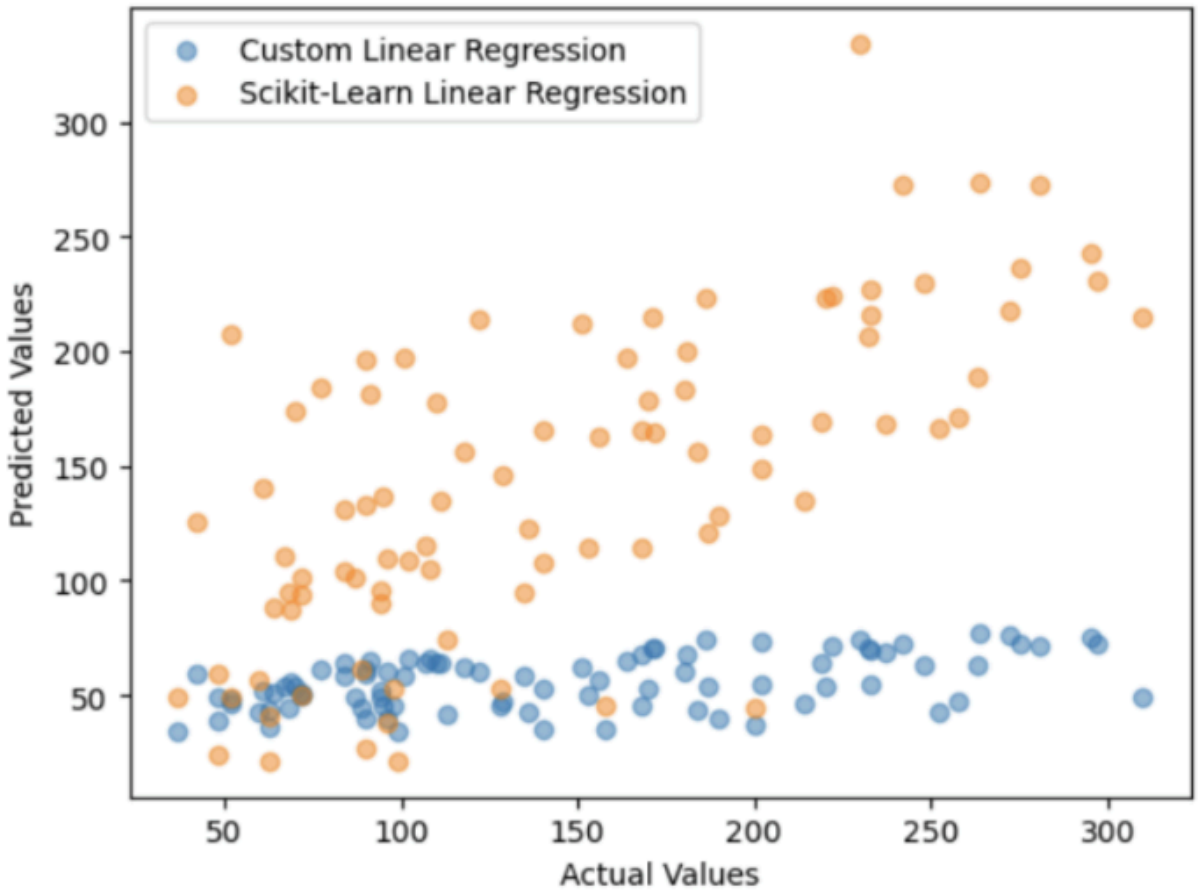
```
y_pred_model = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred_model)
print("Custom Linear Regression MAE:", mae)
```

Meanwhile, for the library's model, the values are as follows:

```
lr_sklearn = LinearRegression()
lr_sklearn.fit(X_train, y_train)
y_pred_sklearn = lr_sklearn.predict(X_test)
mae_sklearn = mean_absolute_error(y_test, y_pred_sklearn)
print("Scikit-Learn Linear Regression MAE:", mae_sklearn)
```

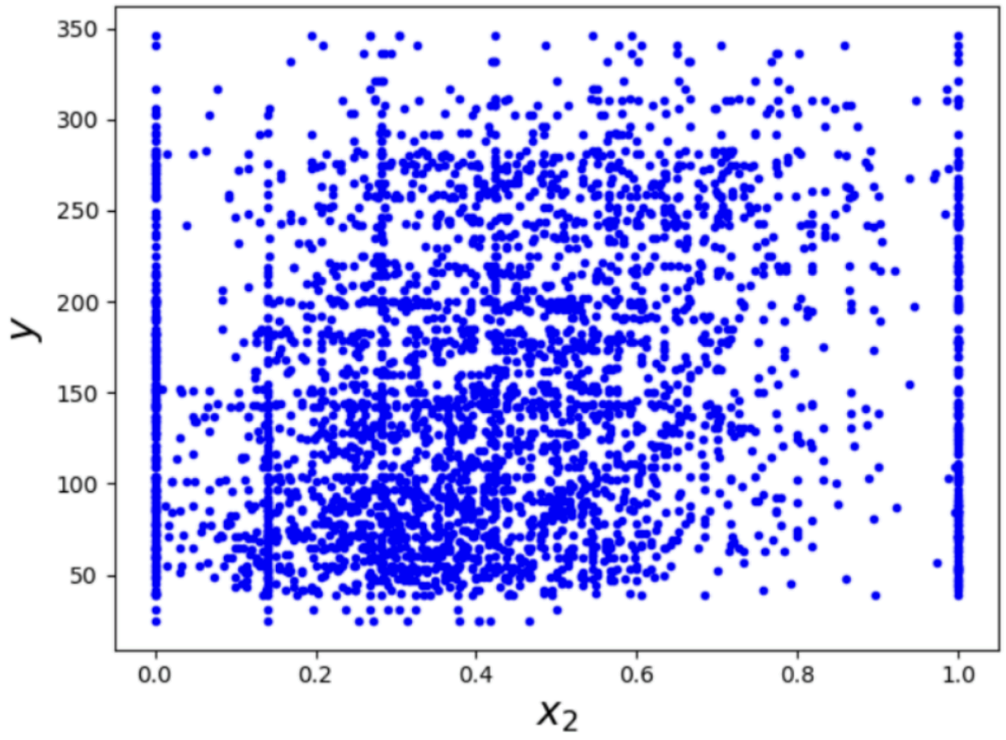
```
Scikit-Learn Linear Regression MAE: 43.48538185702934
```

In the end, if we compare these two models in a table, we see that the results of the Scikit-learn model are much more reasonable than our model, which makes sense. Our model is very simple, and the goal was to implement the most basic algorithm. It is possible that with feature engineering or choosing different loss functions, we could achieve a better model.

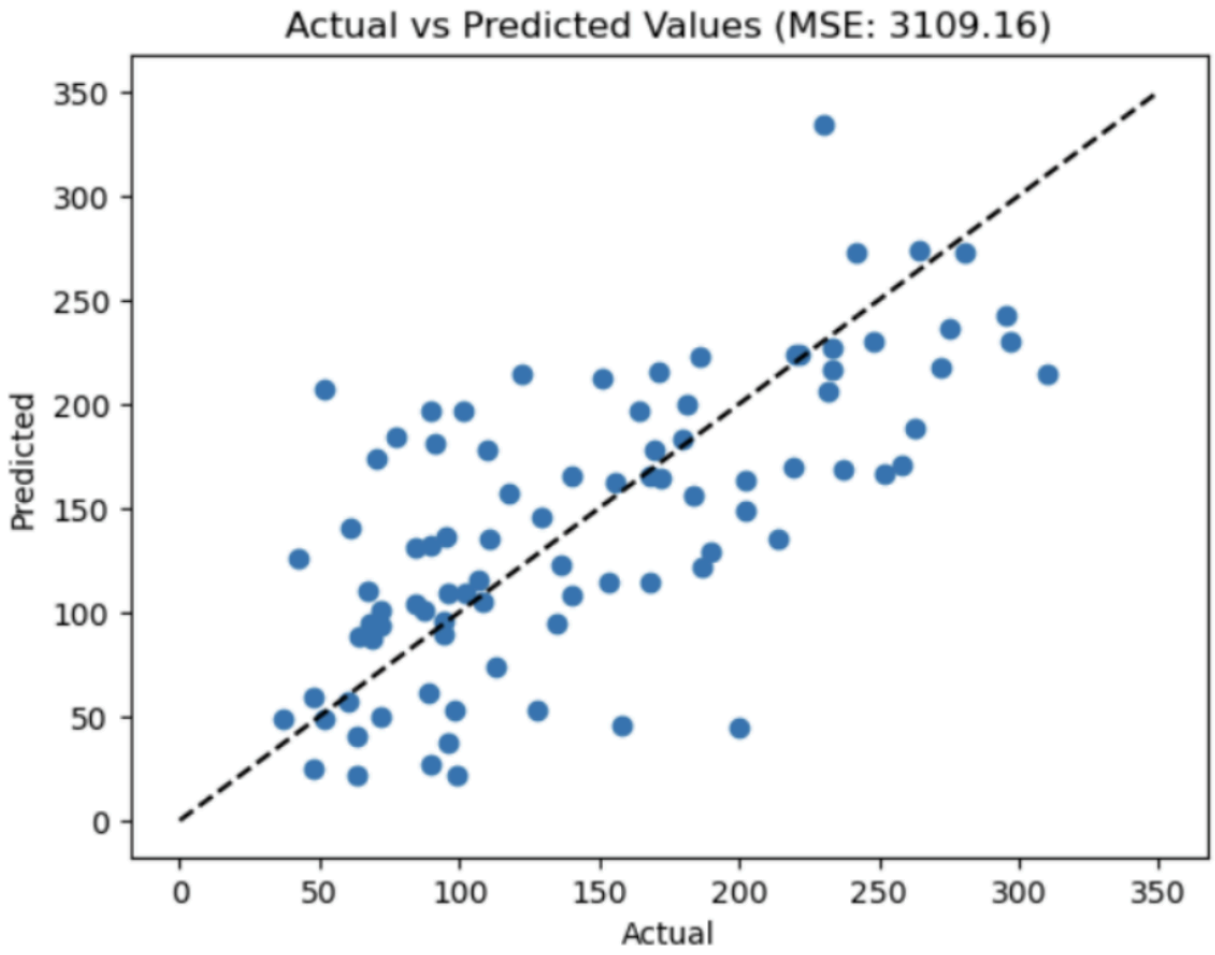

Custom Linear Regression
Scikit-Learn Linear Regression
Predicted Values
Actual Values
50
100
150
200
250
300

Exercise 6: Implement Linear Regression using the normal equation as the training algorithm from scratch.

In this problem, we try to achieve a better estimate using the normal equation. Generally, gradient descent is more popular and widely used than this technique, and it also has more popularity among data scientists. However, we can see that sometimes the normal equation technique can also provide satisfactory results.
If we look at the data and the estimated values:



we can see that our model has managed to make a fairly good estimate using this technique.


Actual vs Predicted Values (MSE: 3109.16)
Predicted
Actual
0
50
100
150
200
250
300
350

Exercise 10: Implement Forward and Backward Feature selection algorithms from scratch with MSE as the metric.

We know that one of the important techniques in machine learning is feature engineering. Two well-known algorithms for this task have been implemented. As expected, in the end, more important features are identified, while overfitting and underfitting are also avoided. In the forward function, we have all the features, and each time one feature is selected. This process continues until no more features can be removed.
Our algorithm works by returning the selected features in both the forward and backward stages and also calculates the "MSE" along the way:

testing

In [60]:
```
selected_features, best_mse = forward_feature_selection(X, y)
print("Forward features = ", selected_features)
print("MSE = ", best_mse)
```

```
Forward features =  [2, 8, 3, 4, 1, 5, 7, 9, 6, 0]
MSE =  2859.6903987680657
```

In [61]:
```
selected_features, best_mse = forward_feature_selection(X, y)
print("Forward features = ", selected_features)
print("MSE = ", best_mse)
```

```
Forward features =  [2, 8, 3, 4, 1, 5, 7, 9, 6, 0]
MSE =  2859.6903987680657
```

Exercise 12: In this part, you are going to work with the News Popularity Prediction dataset. You will implement a regression model using the Scikit-Learn package to predict the popularity of new articles (the number of times they will be shared online) based on about 60 features. You are expected:

- Perform exploratory data analysis on the dataset.
- Propose 5 different hypothesis tests related to the dataset. At least use 3 different tests.
- Try Ridge and Lasso regression.
- Use various scaling methods and report their effects.
- Add polynomial features and report their effect.
- Try using GridSearchCV with RandomizedSearchCV to tune your model's hyperparameters. (Extra Point)
- Apply the feature selection methods that you have implemented in the above sections.
- Get familiar with and implement the following loss functions from scratch and utilize them with a Linear Regression model and discuss their effect on the performance of the model. (Extra Point)
    - Absolute Error
    - Epsilon-sensetive error
    - Huber

In this section of the exercises, we aim to implement a regression model for the **Online News Popularity** dataset. This dataset is publicly available and is commonly used in machine learning and data mining research to predict the popularity of online news articles. The dataset includes various features related to news articles, such as the number of words, the number of social media shares, keywords, and more. The target variable in the dataset is typically the number of shares an article receives on social media, which can be used to predict the popularity or virality of news articles.

The Online News Popularity dataset is often used for tasks such as regression, classification, and feature selection. Typically, it is used to develop and evaluate machine learning models that predict the popularity of online news articles based on various features. In this process, we apply preprocessing steps to make the data more suitable for the task. Along the way, the model moves from a basic implementation to more advanced methods. Finally, the final result is printed, and we can compare it with more well-known models.

The goal of this exercise is to fully familiarize ourselves with preprocessing procedures and, ultimately, to find the best way to prepare the data for training. This includes using data visualization techniques, identifying the best training algorithm for this task, and exploring various methods to improve the results and accuracy. The exercise also involves the use of different loss functions to achieve the highest possible accuracy. In the following report, we will walk through the code implementation and explain how to achieve the problem's objective using these methods.

First, we import the required dataset.

importing data

```
In [4]: df = pd.read_csv("OnlineNewsPopularity.csv")
```

Next, we move on to data preprocessing. Data preprocessing is a crucial step in the data analysis process because it helps us understand the characteristics of the data, identify any data quality issues, and gain insights that can inform feature engineering, model selection, and model evaluation. We start by displaying the initial information to make any necessary changes before implementing the model.

```
In [5]: print("Dataset shape: ", df.shape)
        print("Columns: ", df.columns)
```

```
Dataset shape:  (39644, 61)
Columns:  Index(['url', ' timedelta', ' n_tokens_title', ' n_tokens_content',
       ' n_unique_tokens', ' n_non_stop_words', ' n_non_stop_unique_tokens',
       ' num_hrefs', ' num_self_hrefs', ' num_imgs', ' num_videos',
       ' average_token_length', ' num_keywords', ' data_channel_is_lifestyle',
       ' data_channel_is_entertainment', ' data_channel_is_bus',
       ' data_channel_is_socmed', ' data_channel_is_tech',
       ' data_channel_is_world', ' kw_min_min', ' kw_max_min', ' kw_avg_min',
       ' kw_min_max', ' kw_max_max', ' kw_avg_max', ' kw_min_avg',
       ' kw_max_avg', ' kw_avg_avg', ' self_reference_min_shares',
       ' self_reference_max_shares', ' self_reference_avg_sharess',
       ' weekday_is_monday', ' weekday_is_tuesday', ' weekday_is_wednesday',
       ' weekday_is_thursday', ' weekday_is_friday', ' weekday_is_saturday',
       ' weekday_is_sunday', ' is_weekend', ' LDA_00', ' LDA_01', ' LDA_02',
       ' LDA_03', ' LDA_04', ' global_subjectivity',
       ' global_sentiment_polarity', ' global_rate_positive_words',
       ' global_rate_negative_words', ' rate_positive_words',
       ' rate_negative_words', ' avg_positive_polarity',
       ' min_positive_polarity', ' max_positive_polarity',
       ' avg_negative_polarity', ' min_negative_polarity',
       ' max_negative_polarity', ' title_subjectivity',
       ' title_sentiment_polarity', ' abs_title_subjectivity',
       ' abs_title_sentiment_polarity', ' shares'],
      dtype='object')
```

Additionally, we can provide a summary of the feature types, their means, etc. Then, we use the `describe()` method to display summary statistics for the numerical columns in the dataset and check for missing values using the `isnull()` method.

```
In [6]: df.describe()
```

Out[6]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	num_self_hrefs	num_
count	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	39644.00
mean	354.530471	10.398749	546.514731	0.548216	0.996469	0.689175	10.883690	3.293638	4.54
std	214.163767	2.114037	471.107508	3.520708	5.231231	3.264816	11.332017	3.855141	8.30
min	8.000000	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	164.000000	9.000000	246.000000	0.470870	1.000000	0.625739	4.000000	1.000000	1.00
50%	339.000000	10.000000	409.000000	0.539226	1.000000	0.690476	8.000000	3.000000	1.00
75%	542.000000	12.000000	716.000000	0.608696	1.000000	0.754630	14.000000	4.000000	4.00
max	731.000000	23.000000	8474.000000	701.000000	1042.000000	650.000000	304.000000	116.000000	128.00

8 rows × 60 columns

find missing values

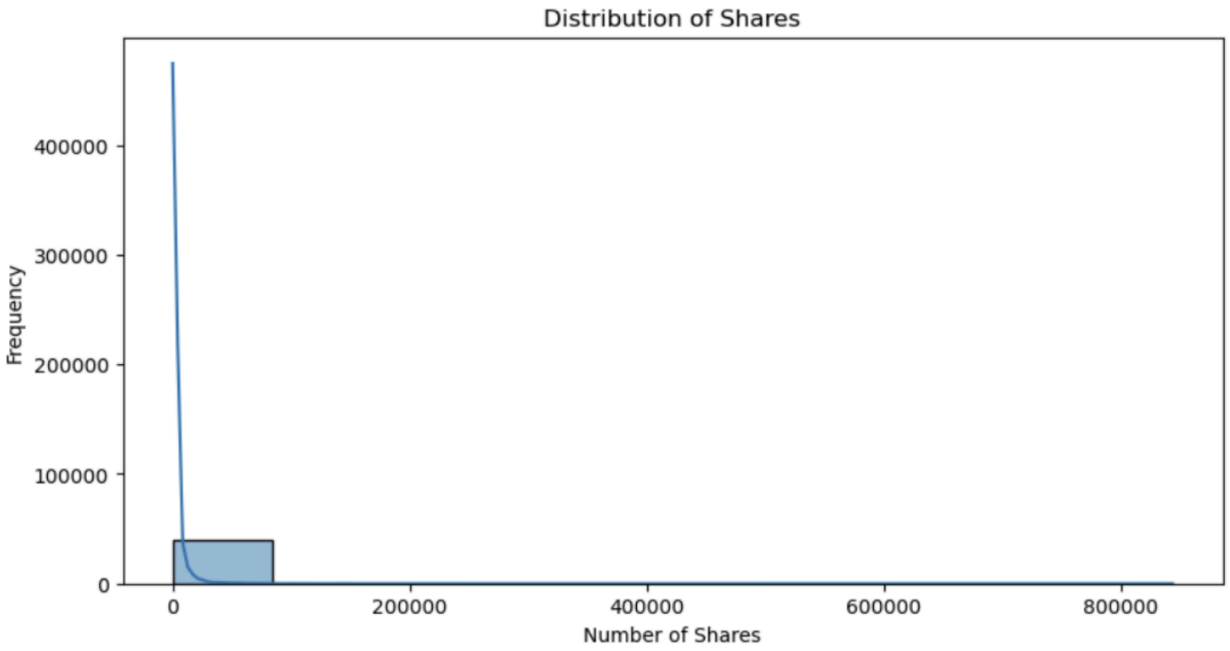
```
In [9]: print("Missing values: ")
        df.isnull().sum()
```

```
Missing values:

Out[9]: url                                0
         timedelta                         0
         n_tokens_title                    0
         n_tokens_content                  0
         n_unique_tokens                   0
                                          ..
         title_subjectivity                0
         title_sentiment_polarity          0
         abs_title_subjectivity            0
         abs_title_sentiment_polarity      0
         shares                            0
        Length: 61, dtype: int64
```

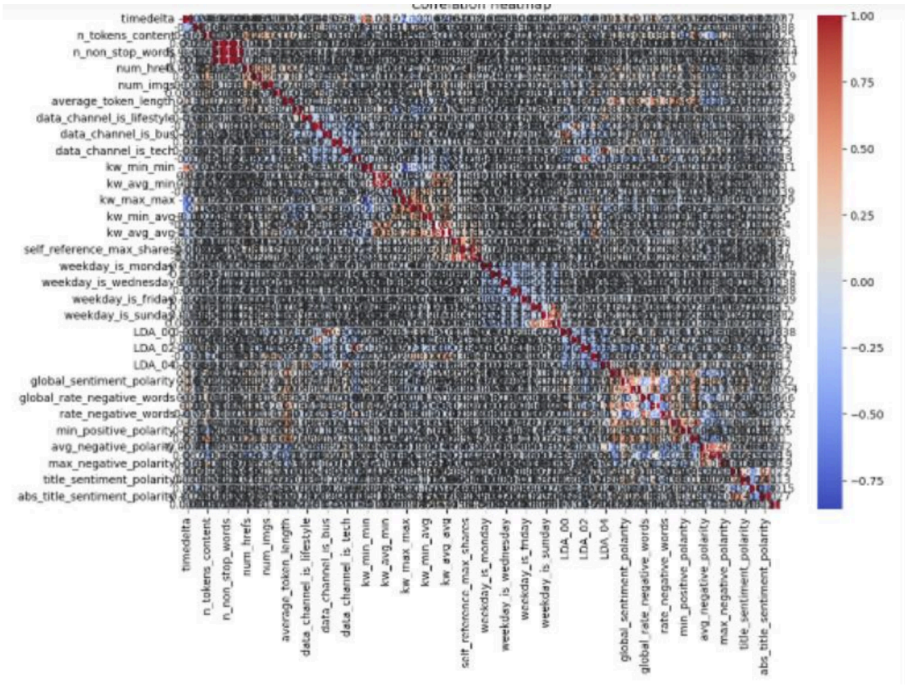
As we can see, our dataset has no missing values, so it is ready for preprocessing.

In the next step, we use data visualization techniques, such as histograms and heat maps, to gain insights into the distribution of the target variable (number of shares) and the correlations between features.

```
plt.figure(figsize=(10, 5))
sns.histplot(df[' shares'], bins=10, kde=True)
plt.xlabel('Number of Shares')
plt.ylabel('Frequency')
plt.title('Distribution of Shares')
plt.show()
```



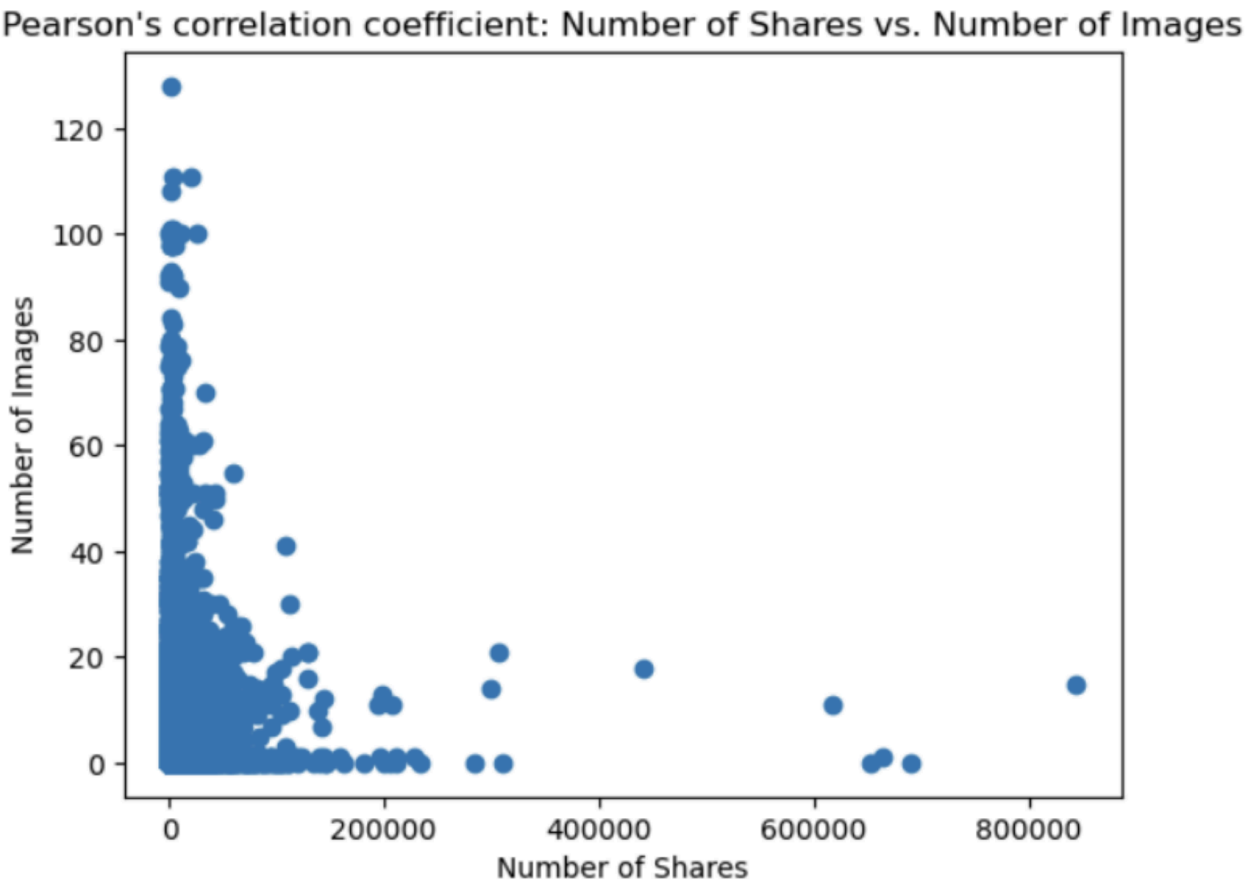

```
plt.figure(figsize=(12, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```



Now, it's time to conduct statistical hypothesis tests. Here, five tests have been performed using three different types of tests:

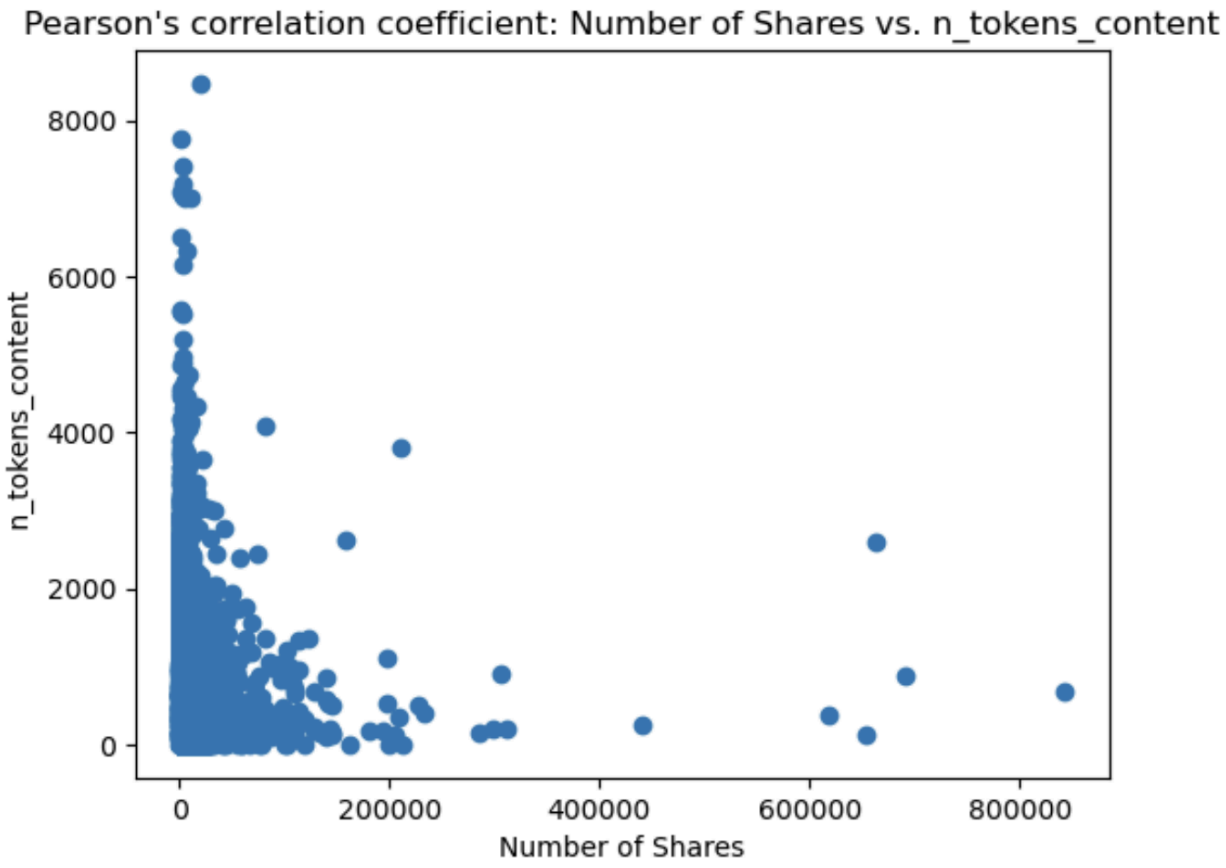
1. **Pearson's Correlation Coefficient**:
   This test is used to assess the strength and direction of the linear relationship between two numerical variables (e.g., the number of shares and the number of images). It helps determine whether a statistically significant correlation exists between two numerical variables. Here, Pearson's correlation coefficient can be used to measure the strength and direction of the linear relationship between two continuous variables (number of images and number of shares) and to determine if there is a significant correlation.

```
corr_coeff, p_value = stats.pearsonr(df[' shares'], df[' num_imgs'])

# Scatter plot of num_shares vs. num_images
plt.scatter(df[' shares'], df[' num_imgs'])
plt.xlabel('Number of Shares')
plt.ylabel('Number of Images')
plt.title("Pearson's correlation coefficient: Number of Shares vs. Number of Images")
plt.show()
```



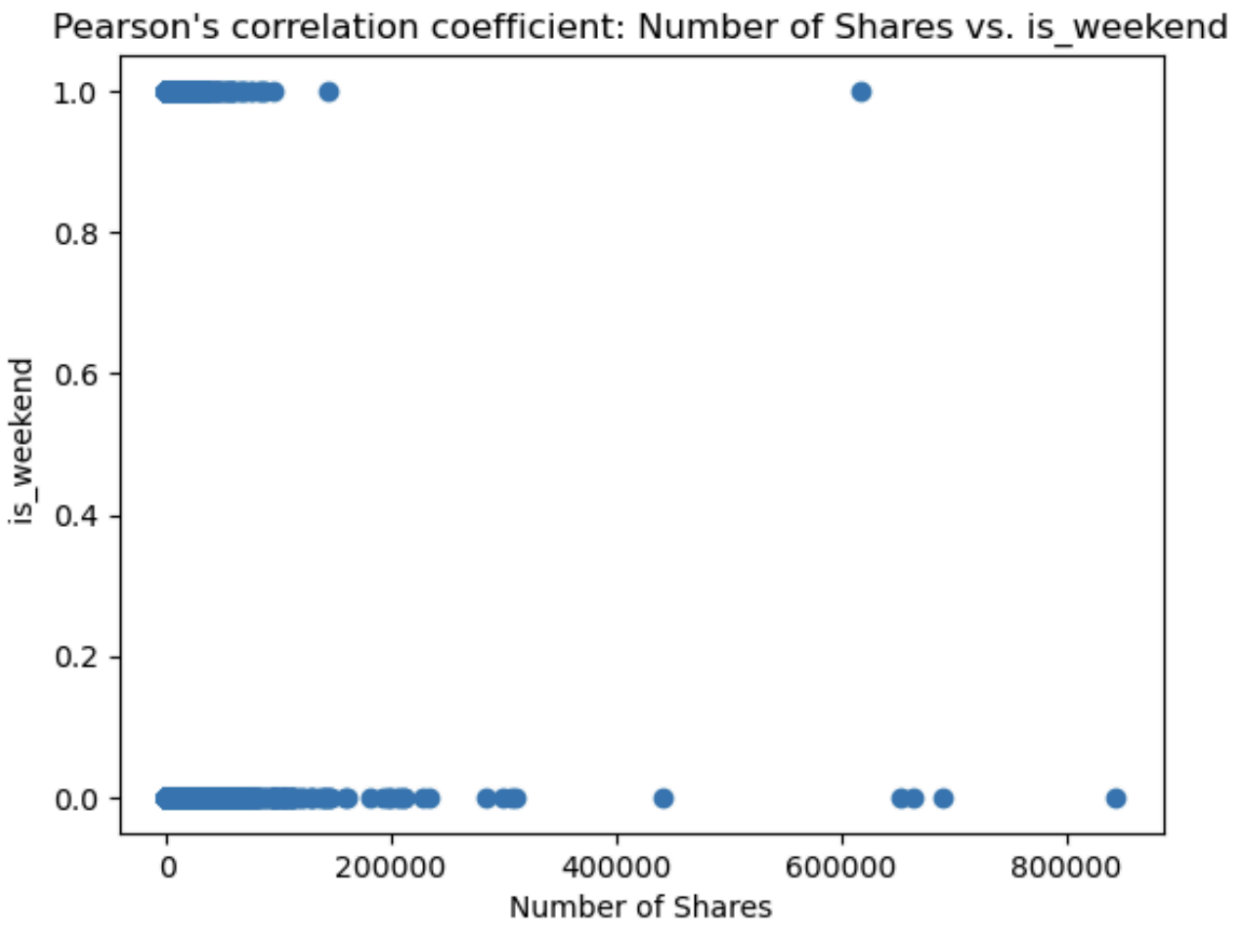
As we can see here, there is a high correlation, indicating that using fewer images can still result in a large number of shares.

```
Pearson's correlation coefficient:
Correlation coefficient: 0.002458984345090837
p-value: 0.6244249147493814
```



Additionally, we observe that the more tokens there are, the more shares will be generated.

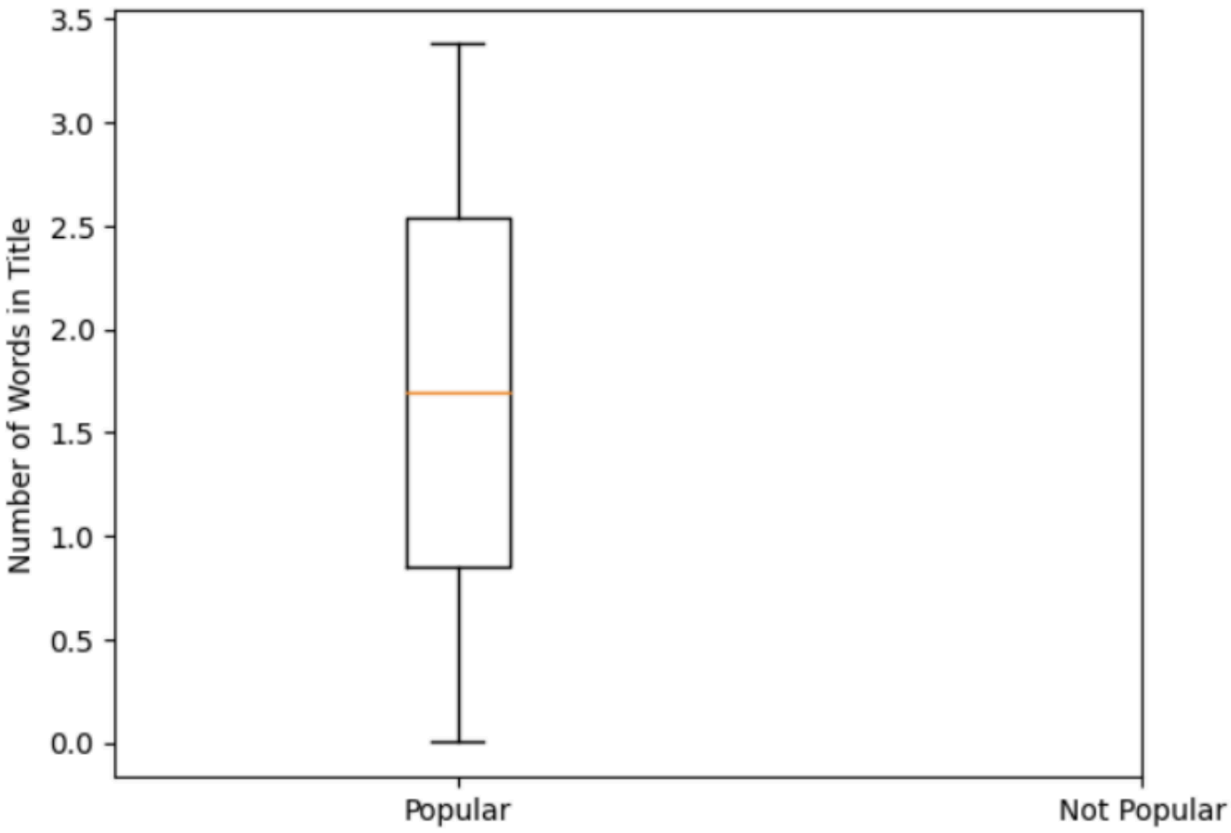
```
weekday_data = df[df[" is_weekend"] == 1][" shares"]
weekend_data = df[df[" is_weekend"] == 0][" shares"]

plt.scatter(df[' shares'], df[' is_weekend'])
plt.xlabel('Number of Shares')
plt.ylabel(' is_weekend')
plt.title("Pearson's correlation coefficient: Number of Shares vs. is_weekend")
plt.show()
```



2. **Independent t-test**:
   This test is used to compare the means of two independent groups to determine if there is a significant difference between them. It is commonly used when there are two distinct groups that are not related to each other, and we want to determine whether there is a statistically significant difference between the means of a specific variable in these two groups.

```
t_stat, p_value = stats.ttest_ind(weekday_data, weekend_data)
print("Independent t-test:")
print("t-statistic:", t_stat)
print("p-value:", p_value)

plt.boxplot([t_stat, p_value])
plt.xticks([1, 2], ['Popular', 'Not Popular'])
plt.ylabel('Number of Words in Title')
plt.show()
```

```
Independent t-test:
t-statistic: 3.3769109636398387
p-value: 0.00073375190865517 08
```

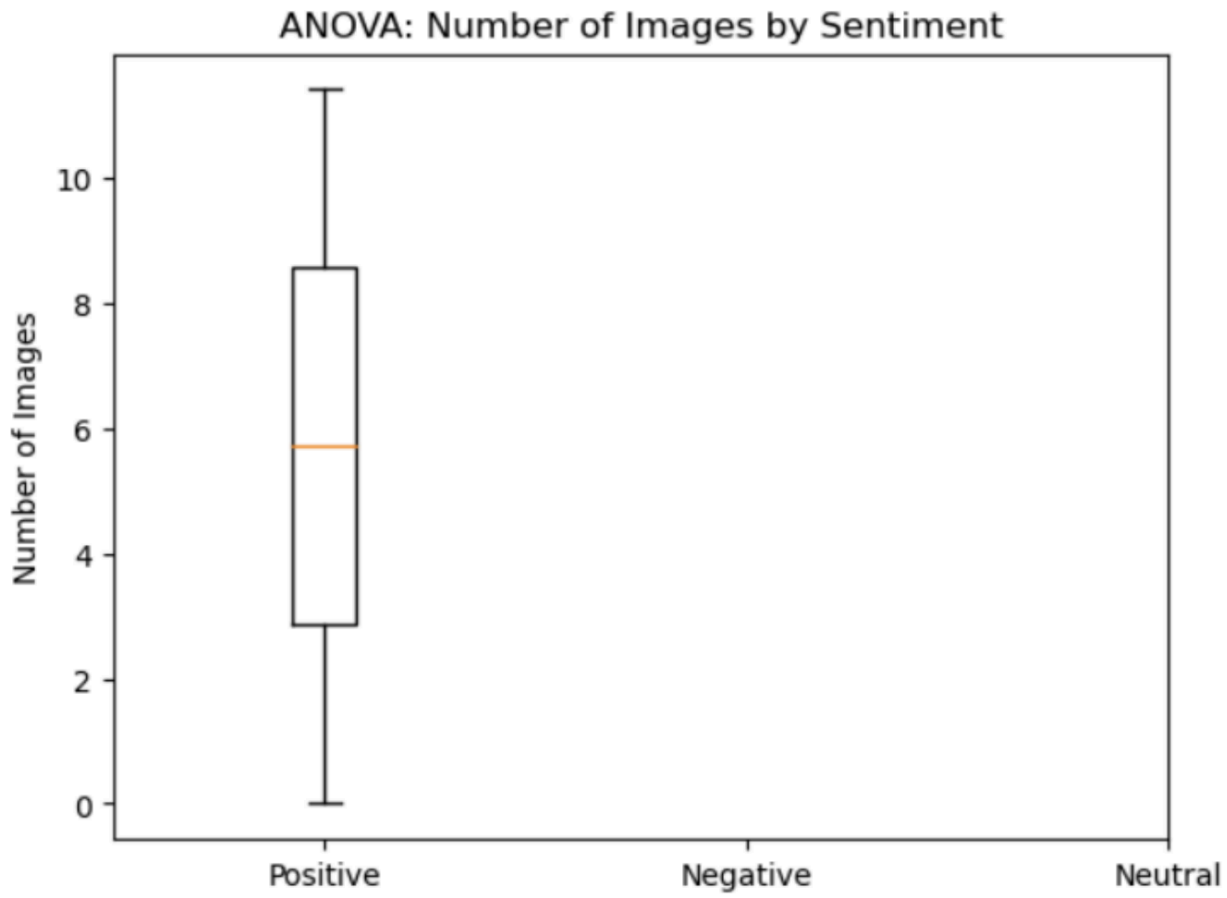


3. **One-way ANOVA**:
   This test is used to compare the means of a numerical variable across several groups (e.g., article categories or sentiments). It helps determine whether there is a statistically significant

difference in the means of the numerical variable across groups. In this case, it can be used to compare the means of more than two groups (article categories) and determine whether there is a significant difference.

```
f_stat, p_value = stats.f_oneway(*[df[df[" is_weekend"] == day][" shares"] for day in days_of_week])
print("One-way ANOVA:")
print("F-statistic:", f_stat)
print("p-value:", p_value)

# Plot boxplots for the three groups
plt.boxplot([f_stat, p_value])
plt.xticks([1, 2, 3], ['Positive', 'Negative', 'Neutral'])
plt.ylabel('Number of Images')
plt.title('ANOVA: Number of Images by Sentiment')
plt.show()
```

```
One-way ANOVA:
F-statistic: 11.403527656350944
p-value: 0.0007337519086632175
```



Next, we move on to the implementation of the main model. After importing the necessary libraries, loading the dataset, and extracting the features and target variable, we begin. Then, using a function, we split the data into training and test sets.

In the next step, we initialize a linear regression model, fit it to the training data using various algorithms, and use the prediction method to make predictions on the test set. Finally, we evaluate the model's performance using Mean Squared Error (MSE) and the R-squared score as evaluation metrics.

We need to first exclude the "URL" and "shares" (the target estimation) variables from the dataset, and then form the training and test sets.

Extract features and target variable

In [18]:
```
X = df.iloc[:, 2:-1]
y = df[' shares']
```

Split the data into training and testing sets

In [19]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Initialize and fit a linear regression model

In [20]:
```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

Out[20]:
```
LinearRegression()
```

Now, we will implement and test the model in a simple way without using any specific algorithm.

Initialize and fit a linear regression model

In [20]:
```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

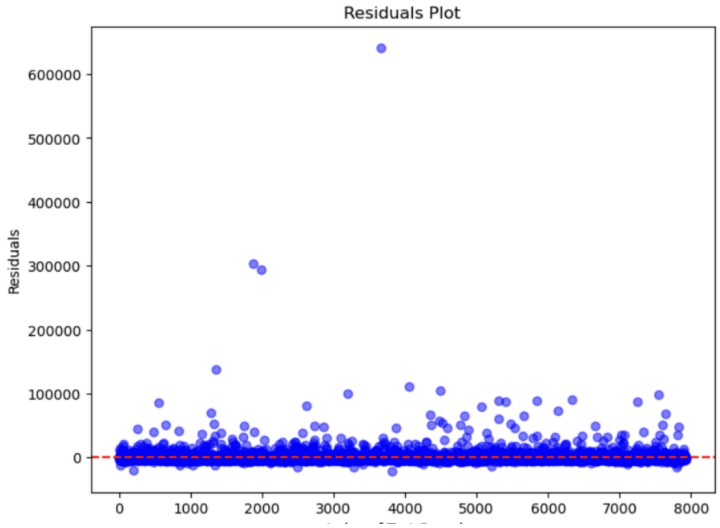
Out[20]:
```
LinearRegression()
```

Make predictions on the test set

In [21]:
```
y_pred = regressor.predict(X_test)
```

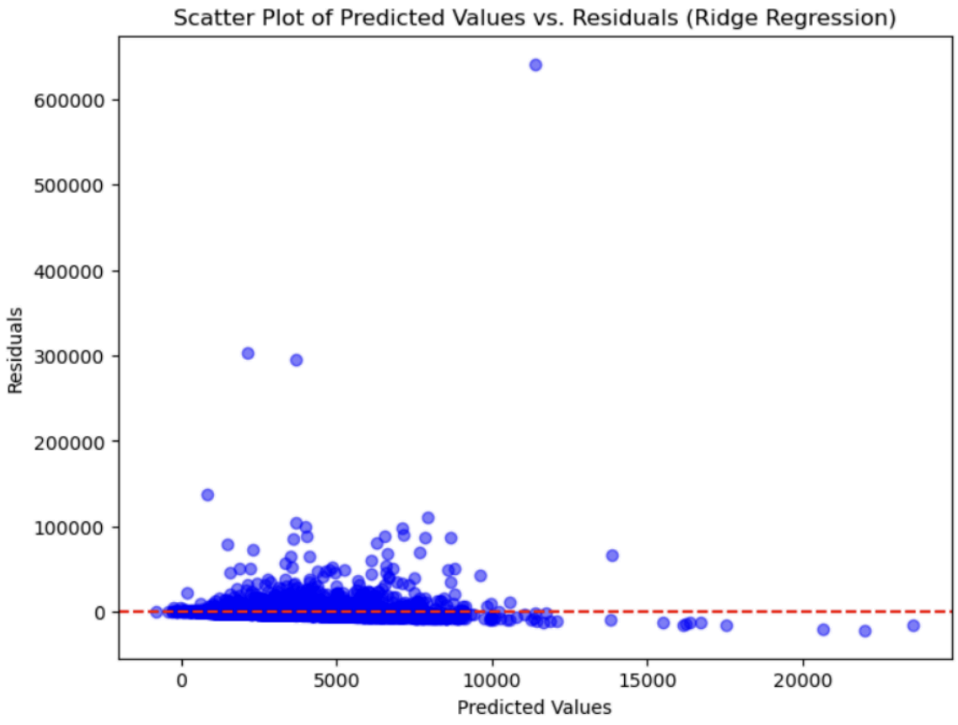
Plot residuals

In [22]:
```
y_residuals = y_test - y_pred

plt.figure(figsize=(8, 6))
plt.scatter(np.arange(len(y_residuals)), y_residuals, c='blue', alpha=0.5)
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Index of Test Samples')
plt.ylabel('Residuals')
plt.title('Residuals Plot')
plt.show()
```



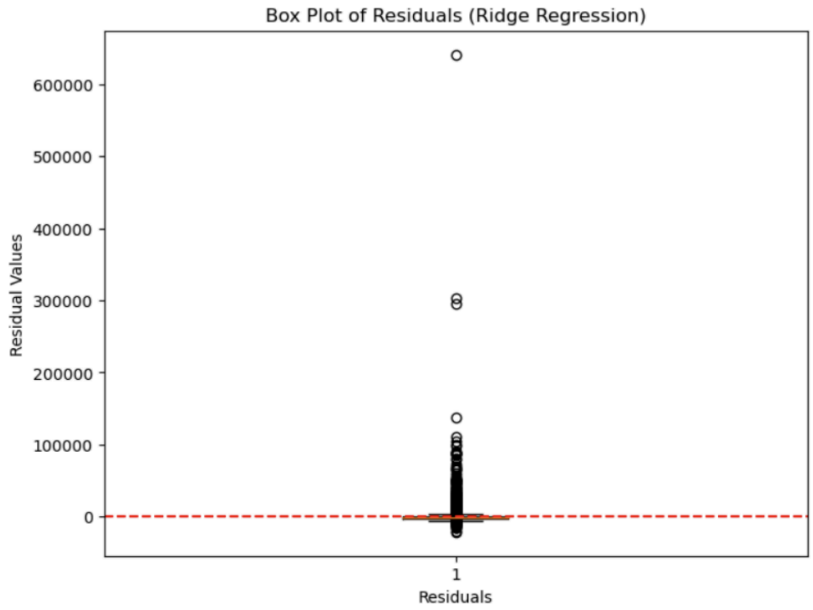
In the next section, we will try to implement the Ridge Regression algorithm. Alpha (α) is the strength of the regularization, where higher values lead to stronger regularization. Finally, we made predictions on the test data, calculated the Mean Squared Error (MSE), and computed the R-squared value to evaluate the model's performance.

If the residuals are evenly distributed and have similar variances across the range of predicted values, it indicates that the model's predictions are consistent across the entire range of predicted values. If the residuals are randomly scattered around the line, it suggests that the residuals are not systematically biased and do not show any specific pattern or trend. However, if the residuals show a detectable pattern or trend, such as fanning out or clustering toward one end of the plot, it may indicate heteroscedasticity, meaning that the variance of the residuals is not constant across the range of predicted values. This may suggest that the model's predictions are more accurate or less reliable in certain regions of the predicted values.



After this, we will implement the Lasso algorithm in the same way as Ridge.

```
In [34]: from sklearn.linear_model import Lasso

In [35]: X = df.iloc[:, 2:-1]
         y = df.iloc[:, -1]

In [36]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

In [37]: lasso = Lasso(alpha=1.0)

In [38]: lasso.fit(X_train, y_train)

C:\Users\katti\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:647: C
ot converge. You might want to increase the number of iterations, check the scale of the fea
risation. Duality gap: 1.981e+12, tolerance: 4.402e+08
  model = cd_fast.enet_coordinate_descent(

Out[38]: Lasso()

In [39]: # Predict on the testing data
         y_pred = lasso.predict(X_test)

In [41]: plt.figure(figsize=(8, 6))
         plt.boxplot(y_residuals)
         plt.axhline(y=0, color='red', linestyle='--')
         plt.xlabel('Residuals')
         plt.ylabel('Residual Values')
         plt.title('Box Plot of Residuals (Ridge Regression)')
```



At this stage, to achieve better results, we focus on examining the features, which includes scaling the features and applying polynomial transformations to the model. First, we need to test various methods for scaling the features and evaluate each one. Scaling input features often improves the performance of regression models. Here, we implement three feature scaling methods.

The first method scales the values between 0 and 1, which is a well-known technique. This method transforms the feature values into a common range, typically between 0 and 1. The effect of this method is that it prevents features with larger values from dominating the model during training.

Method1 : Min-Max scailing

In [97]:
```
scaler = MinMaxScaler()
X_train_minmax = scaler.fit_transform(X_train)
X_test_minmax = scaler.transform(X_test)
```

In [101]:
```
ridge.fit(X_train_minmax, y_train)
y_pred_minmax = ridge.predict(X_test_minmax)
mse_minmax = mean_squared_error(y_test, y_pred_minmax)
r2_minmax = r2_score(y_test, y_pred_minmax)
```

The next method is **standard scaling**, which scales the features to have a mean of zero and a unit variance. The effect of standard scaling is that it centers the feature values around zero and scales them to have similar variances, which can help prevent numerical instability and make the optimization process smoother.

Method2 : Standard Scaling

In [102]:
```
scaler = StandardScaler()
X_train_standard = scaler.fit_transform(X_train)
X_test_standard = scaler.transform(X_test)
```

In [103]:
```
ridge.fit(X_train_standard, y_train)
y_pred_standard = ridge.predict(X_test_standard)
mse_standard = mean_squared_error(y_test, y_pred_standard)
r2_standard = r2_score(y_test, y_pred_standard)
```

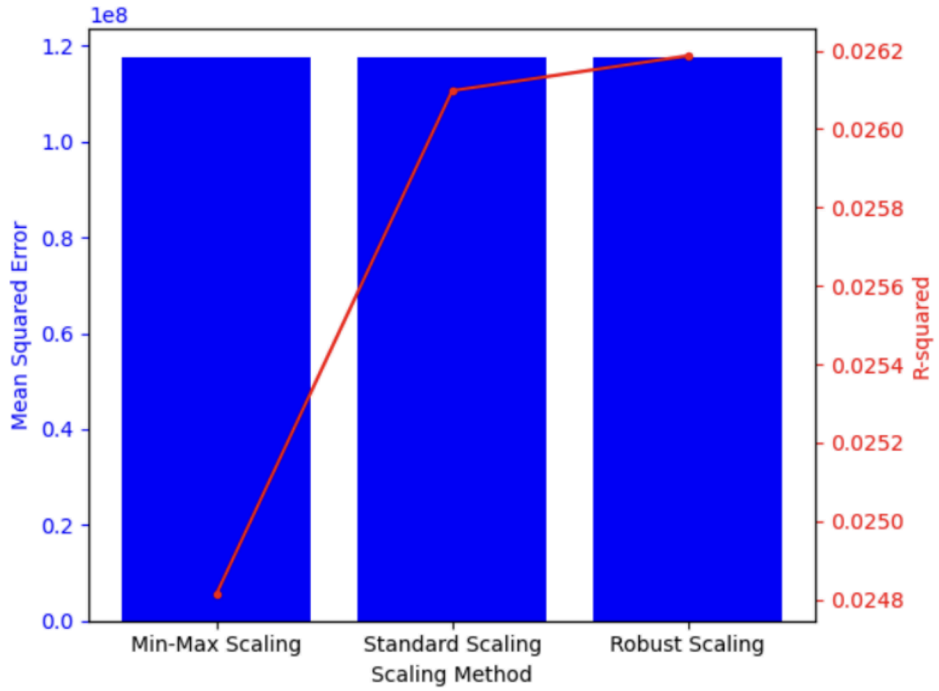
The final method is **robust scaling**, which scales the features using robust statistics that are less sensitive to outliers. It uses the formula (x - median) / IQR, where *x* is the original feature value, the median is the central tendency of the feature, and the IQR (Interquartile Range) is the range between the 25th and 75th percentiles. The effect of robust scaling is that it reduces the impact of outliers on the scaling process, making it more suitable for datasets with potential outlier values.

Method3 : Robust Scaling

In [89]:
```
scaler = RobustScaler()
X_train_robust = scaler.fit_transform(X_train)
X_test_robust = scaler.transform(X_test)
```

In [90]:
```
ridge.fit(X_train_robust, y_train)
y_pred_robust = ridge.predict(X_test_robust)
mse_robust = mean_squared_error(y_test, y_pred_robust)
r2_robust = r2_score(y_test, y_pred_robust)
```

In the chart, you can see that the Mean Squared Error (MSE) for min-max scaling is lower compared to the other methods, indicating better model performance in terms of fewer prediction errors. The R-squared value is also higher for min-max scaling, suggesting a better fit of the model to the data.



After this, we will focus on adding polynomial features. The effect of this is to allow the model to capture non-linear relationships between the features and the target variable. Higher degrees of polynomial features may lead to more complex models with increased ability to fit the training data, but they may also raise the risk of overfitting. Generally, lower degrees of polynomial features may be preferred to avoid overfitting, while higher degrees may be useful for capturing more complex patterns in the data. It is important to experiment with different degrees of polynomial features and choose one that best balances model complexity and performance for the specific dataset.

In the example presented above, using a degree of 5 for the polynomial features, the impact on model performance can be assessed by examining the Mean Squared Error (MSE), which is a measure of how well the model's predictions align with the actual target values.

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, PolynomialFeatures
```

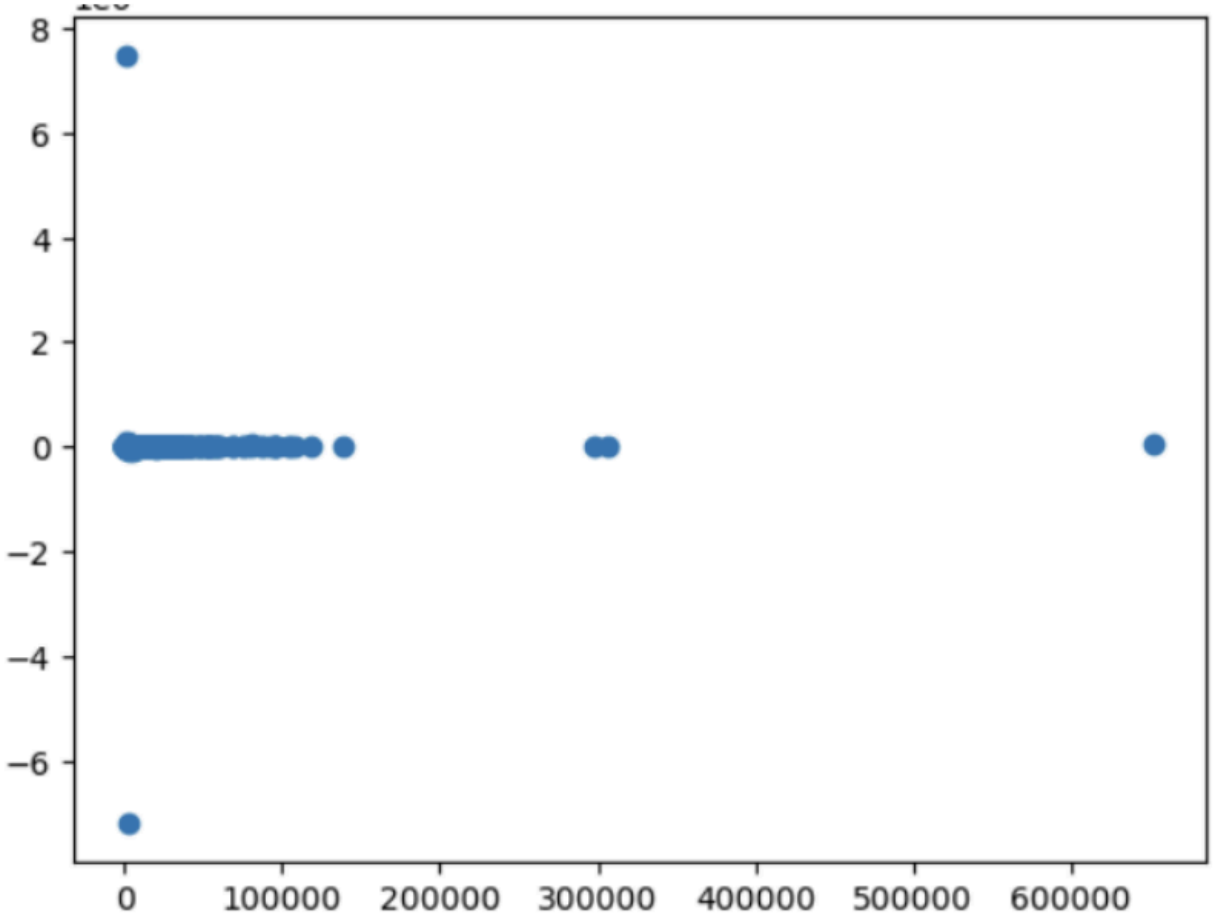
```
scaling_methods = {
    'Min-Max Scaling': MinMaxScaler(),
    'Standard Scaling': StandardScaler(),
    'Robust Scaling': RobustScaler()
}
```

```
polynomial_degrees = [1, 2, 3]
```

```
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
```

```
model = LinearRegression()
model.fit(X_train_poly, y_train)
y_pred = model.predict(X_test_poly)
mse = mean_squared_error(y_test, y_pred)
print('Mean Squared Error (MSE):', mse)
```

In the chart, lower values indicate better performance, as this means that the predicted values are closer to the actual values.



In this section of the exercise, we will implement GridSearchCV alongside RandomizedSearchCV. We will first use GridSearchCV to perform a grid search over a predefined hyperparameter grid for the main model. Then, we will use the best hyperparameters obtained from GridSearchCV to train a linear regression model with those hyperparameters. Similarly, we will use RandomizedSearchCV for execution.

In [121]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
```

In [122]:
```
model = LinearRegression()
```

Define hyperparameter grid for GridSearchCV

In [123]:
```
param_grid = {
    'fit_intercept': [True, False],
    'normalize': [True, False]
}
```

Define hyperparameter grid for GridSearchCV

In [124]:
```
param_grid = {
    'fit_intercept': [True, False],
    'normalize': [True, False]
}
```

Initialize the random forest regressor model

```
from sklearn.ensemble import RandomForestRegressor
```

```
rf_model = RandomForestRegressor()
```

Define hyperparameter distribution for RandomizedSearchCV

```
param_dist = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

random_search = RandomizedSearchCV(rf_model, param_distributions=param_dist, n_iter=10, cv=5, scoring='neg_mean_squared_error')
random_search.fit(X_train, y_train)

best_params = random_search.best_params_
best_rf_model = RandomForestRegressor(**best_params)
best_rf_model.fit(X_train, y_train)
y_pred = best_rf_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)
print("R-squared (R2) Score:", r2)
```

In this section, we aim to implement forward and backward feature selection methods. **Forward selection** is a common method in machine learning used to select a subset of features from a larger set based on their predictive performance. The main idea is to start with an empty set of selected features and iteratively add one feature at a time, choosing the feature that provides the best improvement in the model's performance.

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

selected_features = []

while len(selected_features) < X.shape[1]:
    best_feature = None
    best_mse = float('inf')

    # Iterate over features
    for feature in X.columns:
        if feature not in selected_features:
            features = list(selected_features) + [feature]
            X_train_selected = X_train[features]
            X_val_selected = X_val[features]
            model = LinearRegression()
            model.fit(X_train_selected, y_train)

            y_val_pred = model.predict(X_val_selected)
            mse = mean_squared_error(y_val, y_val_pred)

            # Update best feature if necessary
            if mse < best_mse:
                best_feature = feature
                best_mse = mse

    # Add the best feature to the set of selected features
    selected_features.append(best_feature)
    print(f'Selected Feature: {best_feature}, MSE: {best_mse}')
```

```python
# Choose final feature subset
final_feature_subset = selected_features[:-1]  # Remove last feature added as it resulted in the worst performance

X_train_final = X_train[final_feature_subset]
X_val_final = X_val[final_feature_subset]
model_final = LinearRegression()
model_final.fit(X_train_final, y_train)
y_val_pred_final = model_final.predict(X_val_final)
mse_final = mean_squared_error(y_val, y_val_pred_final)

X_test = ...
y_test = ...
X_test_final = X_test[final_feature_subset]
y_test_pred_final = model_final.predict(X_test_final)
mse_test_final = mean_squared_error(y_test, y_test_pred_final)

print(f'Final Feature Subset: {final_feature_subset}')
print(f'Validation MSE with Final Model: {mse_final}')
print(f'Test MSE with Final Model: {mse_test_final}')
```

We will have for selected features:

```
Selected Feature:  LDA_00, MSE: 117132801.34588273
Selected Feature:  num_videos, MSE: 117155814.46287051
Selected Feature:  self_reference_max_shares, MSE: 117180149.96496108
Selected Feature:  kw_max_max, MSE: 117207370.29463711
Selected Feature:  n_tokens_content, MSE: 117267470.26899552
Selected Feature:  LDA_03, MSE: 117340282.3776753
Selected Feature:  LDA_02, MSE: 117341110.86502957
Selected Feature:  data_channel_is_bus, MSE: 117421313.41471866
Selected Feature:  n_tokens_title, MSE: 117512637.27801216
Final Feature Subset: [' kw_avg_avg', ' self_reference_min_shares', ' kw_max_avg', ' kw_min_avg', ' avg_negative_polarity', ' k
w_min_min', ' data_channel_is_entertainment', ' num_imgs', ' average_token_length', ' weekday_is_monday', ' num_hrefs', ' num_s
elf_hrefs', ' data_channel_is_lifestyle', ' is_weekend', ' min_positive_polarity', ' kw_min_max', ' title_sentiment_polarity',
' self_reference_avg_sharess', ' data_channel_is_socmed', ' data_channel_is_world', ' global_rate_positive_words', ' data_chann
el_is_tech', ' kw_avg_max', ' LDA_04', ' weekday_is_tuesday', ' n_non_stop_unique_tokens', ' n_non_stop_words', ' max_positive_
polarity', ' num_keywords', ' kw_avg_min', ' LDA_01', ' global_rate_negative_words', ' rate_positive_words', ' rate_negative_wo
rds', ' n_unique_tokens', ' weekday_is_friday', ' title_subjectivity', ' abs_title_sentiment_polarity', ' avg_positive_polarit
y', ' weekday_is_saturday', ' weekday_is_sunday', ' kw_max_min', ' abs_title_subjectivity', ' min_negative_polarity', ' max_neg
ative_polarity', ' weekday_is_thursday', ' weekday_is_wednesday', ' global_subjectivity', ' global_sentiment_polarity', ' LDA_0
0', ' num_videos', ' self_reference_max_shares', ' kw_max_max', ' n_tokens_content', ' LDA_03', ' LDA_02', ' data_channel_is_bu
s']
```

After that, we proceed to implement the **backward method** similar to the previous method.

```
selected_features = set(X.columns)

# Backward feature selection
mse_vals = []
while len(selected_features) > 1:
    best_feature = None
    best_mse = float('inf')

    # Iterate over features
    for feature in selected_features:
        # Remove the feature from the set of selected features
        features = list(selected_features - set([feature]))
        X_train_selected = X_train[features]
        X_val_selected = X_val[features]
        model = LinearRegression()
        model.fit(X_train_selected, y_train)

        y_val_pred = model.predict(X_val_selected)
        mse = mean_squared_error(y_val, y_val_pred)

        if mse < best_mse:
            best_feature = feature
            best_mse = mse

    # Remove the best feature from the set of selected features
    selected_features.remove(best_feature)
    mse_vals.append(best_mse)
```

```
final_feature_subset = list(selected_features)
X_train_final = X_train[final_feature_subset]
X_val_final = X_val[final_feature_subset]
model_final = LinearRegression()
model_final.fit(X_train_final, y_train)
y_val_pred_final = model_final.predict(X_val_final)
mse_final = mean_squared_error(y_val, y_val_pred_final)

# Evaluate final model performance on the test set
X_test = ...  # Load and preprocess the test set
y_test = ...  # Load the target variable for the test set

print(f'Final Feature Subset: {final_feature_subset}')
print(f'Validation MSE with Final Model: {mse_final}')
```

In the final part of this exercise, we implement various loss functions. The first method is **Mean Absolute Error (MAE)**. In this example, we define a function that calculates the mean absolute error by taking the absolute difference between the actual target values and the predicted values, and then averaging the absolute differences. We use this function to compute the MAE of the linear regression model's predictions on the test set.

```
def absolute_error(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

model = LinearRegression()
model.fit(X_train, y_train)
y_val_pred = model.predict(X_val)

mae = absolute_error(y_val, y_val_pred)
print(f'Mean Absolute Error: {mae}')

mae_sklearn = mean_absolute_error(y_val, y_val_pred)
```

```
Mean Absolute Error: 2999.186075775858
```

Since applying the absolute error loss function with a linear regression model is similar to using the Mean Squared Error (MSE) loss function, model performance can be evaluated using similar metrics, such as R-squared score and Root Mean Squared Error (RMSE).

Next, we implement the **Epsilon-Sensitive Error** method. In this example, we define a function that calculates epsilon-sensitive error by taking the absolute difference between the actual target values and the predicted values, and then subtracting the epsilon value from the absolute differences. If the absolute difference is smaller than epsilon, the error is set to zero. Finally, we take the average of the resulting errors to compute the epsilon-sensitive error.

```python
def epsilon_sensitive_error(y_true, y_pred, epsilon=0.1):
    errors = np.maximum(0, np.abs(y_true - y_pred) - epsilon)
    return np.mean(errors)

model = LinearRegression()
model.fit(X_train, y_train)
y_val_pred = model.predict(X_val)

epsilon = 0.1 # Set the value of epsilon
ese = epsilon_sensitive_error(y_val, y_val_pred, epsilon)
print(f'Epsilon-Sensitive Error (epsilon={epsilon}): {ese}')

mae_sklearn = mean_absolute_error(y_val, y_val_pred)
print(f'Mean Absolute Error (from sklearn): {mae_sklearn}')
```

```
Epsilon-Sensitive Error (epsilon=0.1): 2999.08607577583
Mean Absolute Error (from sklearn): 2999.186075775861
```

The value of the epsilon-sensitive error depends on the epsilon value, which is a threshold. By adjusting the epsilon value, you can control how sensitive the error is to the differences between predicted and actual values. A larger epsilon makes the error more tolerant of small differences, while a smaller epsilon makes the error less tolerant, penalizing small differences more. It is important to choose an appropriate epsilon value based on the problem's range and the specific requirements of your prediction task. In this example, we used epsilon = 0.1, but you can experiment with different values to find the best one for your dataset.

Finally, the last function we implement is the **Huber Loss** method. In this section, we design a method for this function. The epsilon parameter specifies the threshold that transitions between squared loss and linear loss in the Huber loss function. Our function is defined from scratch to calculate the Huber loss, considering the delta value, which is equivalent to the epsilon parameter. The R-squared score is

computed, and RMSE is used to evaluate the performance of the model trained with the Huber loss function.

```
def huber_error(y_true, y_pred, delta=1.0):
    errors = np.abs(y_true - y_pred)
    mask = errors <= delta
    squared_errors = np.where(mask, 0.5 * np.square(errors), delta * (errors - 0.5 * delta))
    return np.mean(squared_errors)

model = LinearRegression()
model.fit(X_train, y_train)
y_val_pred = model.predict(X_val)

delta = 1.0 # Set the value of delta
huber = huber_error(y_val, y_val_pred, delta)
print(f'Huber Error (delta={delta}): {huber}')

mae_sklearn = mean_absolute_error(y_val, y_val_pred)
print(f'Mean Absolute Error (from sklearn): {mae_sklearn}')
```

```
Huber Error (delta=1.0): 2998.6861374500622
Mean Absolute Error (from sklearn): 2999.186075775861
```

The Huber loss introduces a threshold parameter called "delta" that determines the point at which the loss function transitions from behaving like MSE for small errors to behaving like MAE for larger errors. This makes the Huber loss less sensitive to outliers compared to MSE, as the squared term in MSE can significantly amplify the influence of outliers on the overall loss. In general, the Huber loss function is less sensitive to outliers than MSE. Outliers—data points that deviate significantly from the general trend—can disproportionately affect the model's performance when MSE is used as the loss function. Huber can mitigate this issue by assigning less weight to large errors, making the model more robust against outliers.