In [1]:
```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a v
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

```
/kaggle/input/random-linear-regression/train.csv
/kaggle/input/random-linear-regression/test.csv
```

Simple Class Pipeline for Linear Regression

1. What's It For? This class is like a swiss army knife for data science projects. It helps you load your data, clean it up, and get it ready for some serious number-crunching.
2. Loading Data: Just give it the path to your CSV file, and it'll load your data into a pandas DataFrame. Super easy!
3. Fixing Missing Data: Got gaps in your data? No problem! This class spots where you're missing info and fills in those gaps.
4. Seeing Your Data: It's got cool functions to visually show you what's up with your data – both before and after you clean it up.
5. Outlier Handling: Sometimes data can be weird (outliers, anyone?). This class can smartly find and handle those odd bits.
6. Transforming Data: If your numbers are all over the place, it can calm them down with log transformation, making your data more model-friendly.
7. Training Models: You can use it to train a Linear Regression model and even check which features are super important with Lasso.
8. Visual Fun: It lets you create neat plots to compare actual vs. predicted values and to see how your model learns.
9. Baseline Comparison: Wanna see how good your fancy model is? Compare it with a basic model to see the difference.
10. Main Pipeline: All of this comes together in the main_pipeline function, which runs the whole show from start to finish.

In short, this class is like having a data science buddy that takes a lot of the grunt work off your plate, letting you focus on the cool stuff and simple to customize! 🚀 📊

Visit my 15 learning project in my GitHub here https://github.com/light-87/Kaggle-First-15

In [2]:
```python
import pandas as pd

# Paths to the CSV files
train_file_path = '/kaggle/input/random-linear-regression/train.csv'
test_file_path = '/kaggle/input/random-linear-regression/test.csv'

# Read the CSV files
train_df = pd.read_csv(train_file_path)
test_df = pd.read_csv(test_file_path)

# Merge the dataframes
combined_df = pd.concat([train_df, test_df])

# Save the combined dataframe to a new CSV file
output_file_path = '/kaggle/working/combined_data.csv'
combined_df.to_csv(output_file_path, index=False)

print(f"Combined CSV saved to: {output_file_path}")
```

```
Combined CSV saved to: /kaggle/working/combined_data.csv
```

In [4]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, LassoCV
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
from sklearn.model_selection import ShuffleSplit
from yellowbrick.regressor import PredictionError
from yellowbrick.model_selection import LearningCurve
from sklearn.decomposition import PCA
import seaborn as sns
from sklearn.metrics import mean_absolute_error

np.random.seed(1)

class DataScienceProject:
    def __init__(self):
        pass

    def load_data(self, file_path):
        # Load data using pandas
        data = pd.read_csv(file_path)
```

```
        return data

    def report_missing_values(self, df ):
        # Calculate the number of missing values per column
        missing_values = df.isnull().sum()
        missing_report = pd.DataFrame(missing_values, columns=['missing_values'])
        missing_report = missing_report[missing_report['missing_values'] > 0]

        # Suggest imputation values
        imputation_values = {}
        for column in missing_report.index:
            if df[column].dtype in ['int64', 'float64']:
                skewness = df[column].skew()
                if abs(skewness) > 0.5:
                    imputation_value = df[column].median()
                    imputation_values[column] = ('median', imputation_value)
                else:
                    imputation_value = df[column].mean()
                    imputation_values[column] = ('mean', imputation_value)
            else:
                imputation_value = df[column].mode()[0]
                imputation_values[column] = ('mode', imputation_value)

        return imputation_values

    def apply_imputations(self, df, imputation_values):
        for column, (strategy, value) in imputation_values.items():
            if strategy in ['mean', 'median', 'mode']:
                df[column].fillna(value, inplace=True)
        # print("after imp", df.isna().sum())
        return df


    def visualize_data(self, data, title_suffix=''):
        """
        Visualizes distributions of numerical and categorical features in the dataset.

        Args:
        data (DataFrame): The dataset to visualize.
        title_suffix (str): A suffix for the plot title to distinguish between original and preprocessed data.
        """
        numerical_cols = data.select_dtypes(include=['int64', 'float64']).columns
        categorical_cols = data.select_dtypes(include=['object']).columns

        # Plot for numerical features
```

```
    for col in numerical_cols:
        plt.figure(figsize=(8, 4))
        sns.histplot(data[col], kde=True)
        plt.title(f'Distribution of {col} {title_suffix}')
        plt.xlabel(col)
        plt.ylabel('Frequency')
        plt.show()

    # Plot for categorical features
    for col in categorical_cols:
        plt.figure(figsize=(8, 4))
        sns.countplot(x=col, data=data)
        plt.title(f'Distribution of {col} {title_suffix}')
        plt.xlabel(col)
        plt.ylabel('Count')
        plt.show()

def select_data_within_iqr(self, df, iqr_factor=1.5):
    # Select only numerical columns for IQR calculation
    numerical_cols = df.select_dtypes(include=['int64', 'float64']).columns

    # Calculate IQR for numerical columns
    Q1 = df[numerical_cols].quantile(0.25)
    Q3 = df[numerical_cols].quantile(0.75)
    IQR = Q3 - Q1

    # Determine bounds for outlier detection
    lower_bound = Q1 - (iqr_factor * IQR)
    upper_bound = Q3 + (iqr_factor * IQR)

    # Create a filter for rows to keep
    filter_rows = ((df[numerical_cols] >= lower_bound) & (df[numerical_cols] <= upper_bound)).all(axis=1)

    # Apply this filter to the entire DataFrame
    filtered_df = df[filter_rows]

    return filtered_df

def apply_log_transformation(self, df, target_column,skew_threshold=0.5):
    """
    Applies log transformation to highly skewed columns.

    Args:
    df (DataFrame): The dataframe containing the data.
    skew_threshold (float): The threshold to identify highly skewed columns.
```

```
        Returns:
        DataFrame: The dataframe with log-transformed columns.
        """
        for column in df.select_dtypes(include=['float64', 'int64']):
            if df[column].skew() > skew_threshold and column != target_column :
                df['Log_' + column] = np.log1p(df[column])

        # print(df.columns)
        # print("after log", df.isna().sum())
        return df

    def encode_categorical_columns(self, df):
        """
        Encodes categorical columns using one-hot encoding and removes the original columns.

        Args:
        df (DataFrame): The dataframe to process.

        Returns:
        DataFrame: The dataframe with categorical columns one-hot encoded.
        """
        categorical_cols = df.select_dtypes(include=['object']).columns

        for col in categorical_cols:
            # Apply one-hot encoding to each categorical column
            dummies = pd.get_dummies(df[col], prefix=col)
            df = pd.concat([df, dummies], axis=1)

            # Drop the original categorical column
            df.drop(col, axis=1, inplace=True)

        return df

    def preprocess_data(self, data, target_column):
        # Report and apply imputations, and handle outliers
        imputation_values = self.report_missing_values(data)
        data = self.apply_imputations(data, imputation_values)
        data = self.encode_categorical_columns(data)
        data = self.apply_log_transformation(data, target_column)
        data = self.select_data_within_iqr(data)
        return data

    def train_model(self, training_data, target_column):
        # Modify to return X_train, X_test, y_train, y_test for Lasso analysis
```

```
        X = training_data.drop(target_column, axis=1)
        y = training_data[target_column]
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        model = LinearRegression()
        model.fit(X_train, y_train)
        return model, X_train, X_test, y_train, y_test

    def plot_actual_vs_predicted(self, y_test, y_predict, model_type):
        """
        Plots the actual vs predicted values.

        Args:
        y_test (array-like): The true values of the target variable.
        y_predict (array-like): The predicted values by the model.
        model_type (str): Type of the model ('Baseline' or 'Main').
        """
        plt.figure(figsize=(16, 6))
        plt.title(f"{model_type} Model: Actual vs Predicted Values")
        x_points = list(range(len(y_test)))
        plt.plot(x_points, y_test, label='Actual Values', marker='o')
        plt.plot(x_points, y_predict, label='Predicted Values', marker='x')
        plt.xlabel('Data Points')
        plt.ylabel('Target Variable')
        plt.legend()
        plt.show()


    def plot_learning_curve_and_prediction_error(self, model, X_train, X_test, y_train, y_test, model_type):
        """
        Plots the learning curve and prediction error using Yellowbrick.

        Args:
        model: The trained model.
        X_train, X_test, y_train, y_test: Training and testing data.
        model_type (str): Type of the model ('Baseline' or 'Main').
        """
        # Learning Curve
        plt.figure(figsize=(10, 6))
        lc_viz = LearningCurve(model, cv=5, scoring='r2', n_jobs=4, train_sizes=np.linspace(0.1, 1.0, 10))
        lc_viz.fit(X_train, y_train)
        lc_viz.set_title(f"{model_type} Model: Learning Curve")
        lc_viz.show()

        # Prediction Error
```

```
        plt.figure(figsize=(10, 6))
        pe_viz = PredictionError(model)
        pe_viz.fit(X_train, y_train)
        pe_viz.score(X_test, y_test)
        pe_viz.set_title(f"{model_type} Model: Prediction Error")
        pe_viz.show()


    def feature_importance_lasso(self, X_train, y_train, X_test, y_test):
        # Create and fit the LassoCV model
        lasso = LassoCV(cv=5, random_state=0)
        lasso.fit(X_train, y_train)

        best_alpha = lasso.alpha_
        lasso_coef = lasso.coef_
        y_pred = lasso.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        # Plotting feature importances
        plt.figure(figsize=(10, 6))
        feature_importance = pd.Series(lasso_coef, index=X_train.columns).sort_values()
        feature_importance.plot(kind='barh')
        plt.title('Feature Importances from Lasso Model')
        plt.xlabel('Importance')
        plt.ylabel('Features')
        plt.show()

        print(f"Best alpha: {best_alpha}")
        print(f"MSE: {mse}")
        print(f"R-squared: {r2:.2f}")

    def make_prediction(self, model, new_data):
        # Make predictions using the trained model
        prediction = model.predict(new_data)
        return prediction

    def train_and_evaluate(self, X_train, X_test, y_train, y_test):
        """
        Train a linear regression model and evaluate it.
        """
        model = LinearRegression()
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
```

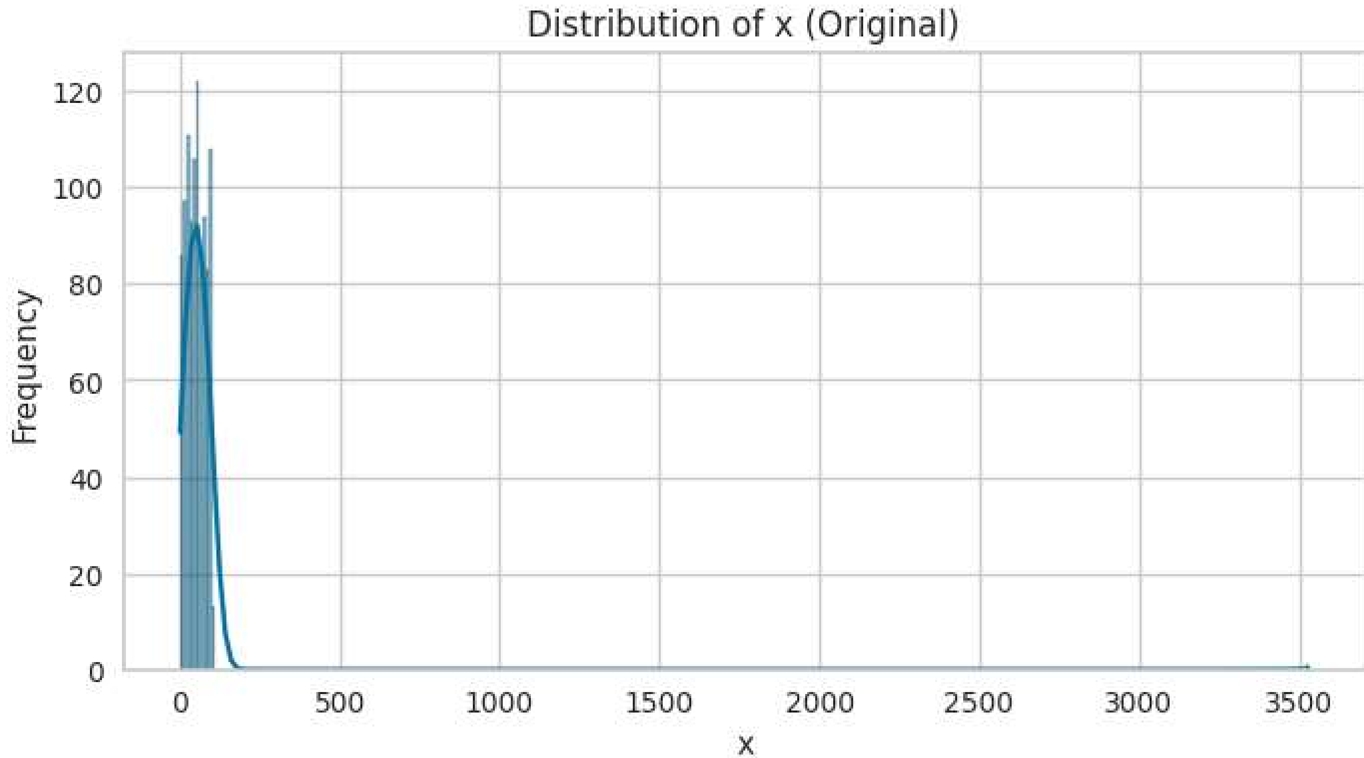
```
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)
        mae = mean_absolute_error(y_test, y_pred)
        return model, mse, r2, mae

    def main_pipeline(self, file_path, target_column):
        print("===== Data Science Project Pipeline =====")
        print("[Step 1] Loading and Preprocessing Data")
        data = self.load_data(file_path)

        # Visualize original data
        self.visualize_data(data, title_suffix='(Original)')

        preprocessed_data = self.preprocess_data(data, target_column)

        # Visualize preprocessed data
        self.visualize_data(preprocessed_data, title_suffix='(Preprocessed)')

        # Baseline Model
        print("\n[Step 2] Training and Evaluating Baseline Model")
        mse_baseline, r2_baseline, baseline_model, X_train_baseline, X_test_baseline, y_train_baseline, y_test_baseline
        y_pred_baseline = baseline_model.predict(X_test_baseline)
        self.plot_actual_vs_predicted(y_test_baseline, y_pred_baseline, "Baseline")
        self.plot_learning_curve_and_prediction_error(baseline_model, X_train_baseline, X_test_baseline, y_train_baseli

        # Main Model
        print("\n[Step 3] Training and Evaluating Main Model")
        model, X_train, X_test, y_train, y_test = self.train_model(preprocessed_data, target_column)
        _, mse_main, r2_main, mae = self.train_and_evaluate(X_train, X_test, y_train, y_test)
        y_predict_main = model.predict(X_test)
        self.plot_actual_vs_predicted(y_test, y_predict_main, "Main")
        self.plot_learning_curve_and_prediction_error(model, X_train, X_test, y_train, y_test, "Main")

        print("\n[Step 4] Performing Lasso Feature Importance Analysis")
        self.feature_importance_lasso(X_train, y_train, X_test, y_test)

        print("\n===== Model Comparison Results =====")
        print("Baseline Model:")
        print("  - MSE: {:.3f}".format(mse_baseline))
        print("  - R-squared: {:.3f}".format(r2_baseline))
        print("  - MAE: {:.3f}".format(mae))
        print("Main Model:")
        print("  - MSE: {:.3f}".format(mse_main))
        print("  - R-squared: {:.3f}".format(r2_main))
        print("  - MAE: {:.3f}".format(mae_baseline))
```

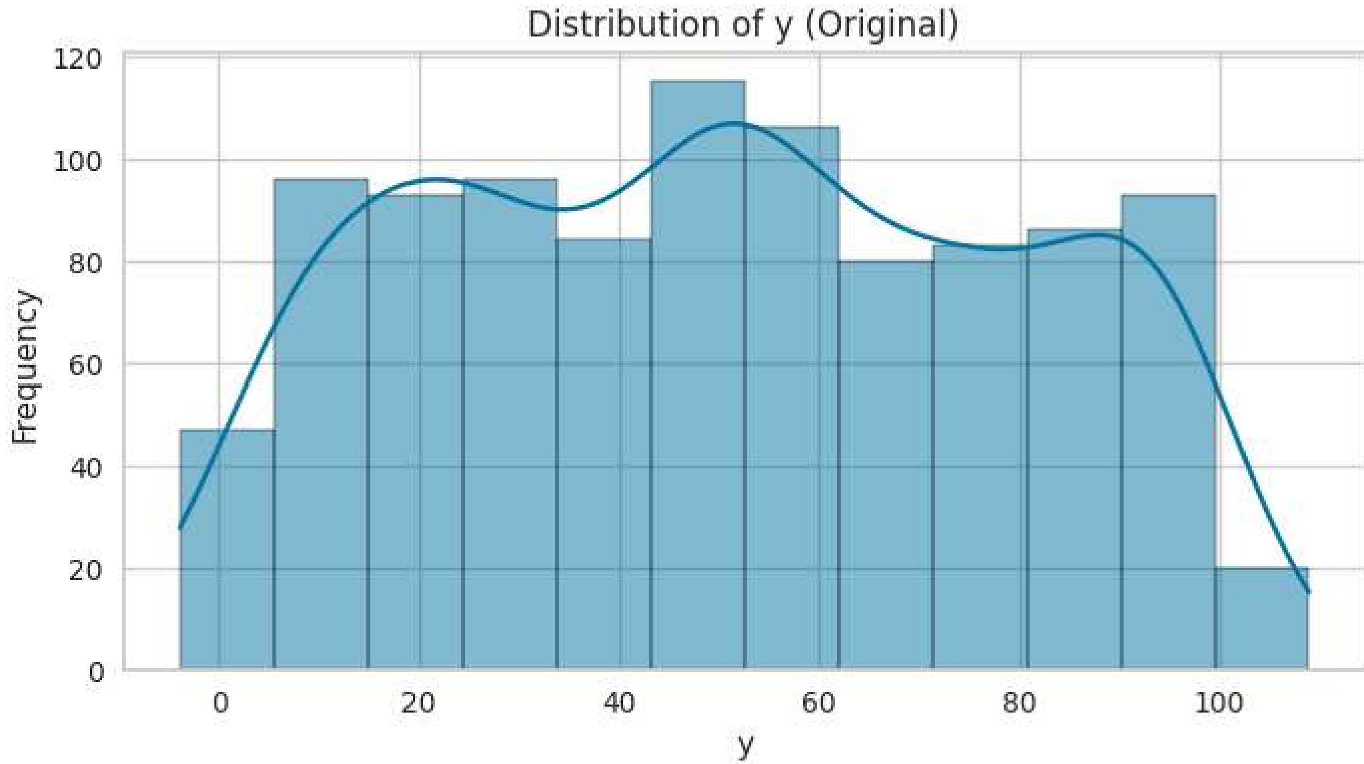
```python
        print("====================================")

        return preprocessed_data

    def compare_with_baseline(self, data, target_column):
        # Updated to return the model and train/test splits
        baseline_data = data.dropna()
        X_baseline = baseline_data.drop(target_column, axis=1)
        X_baseline = self.encode_categorical_columns(X_baseline)
        y_baseline = baseline_data[target_column]
        X_train_baseline, X_test_baseline, y_train_baseline, y_test_baseline = train_test_split(X_baseline, y_baseline,
        baseline_model, mse_baseline, r2_baseline, mae_baseline = self.train_and_evaluate(X_train_baseline, X_test_base
        return mse_baseline, r2_baseline, baseline_model, X_train_baseline, X_test_baseline, y_train_baseline, y_test_ba
```

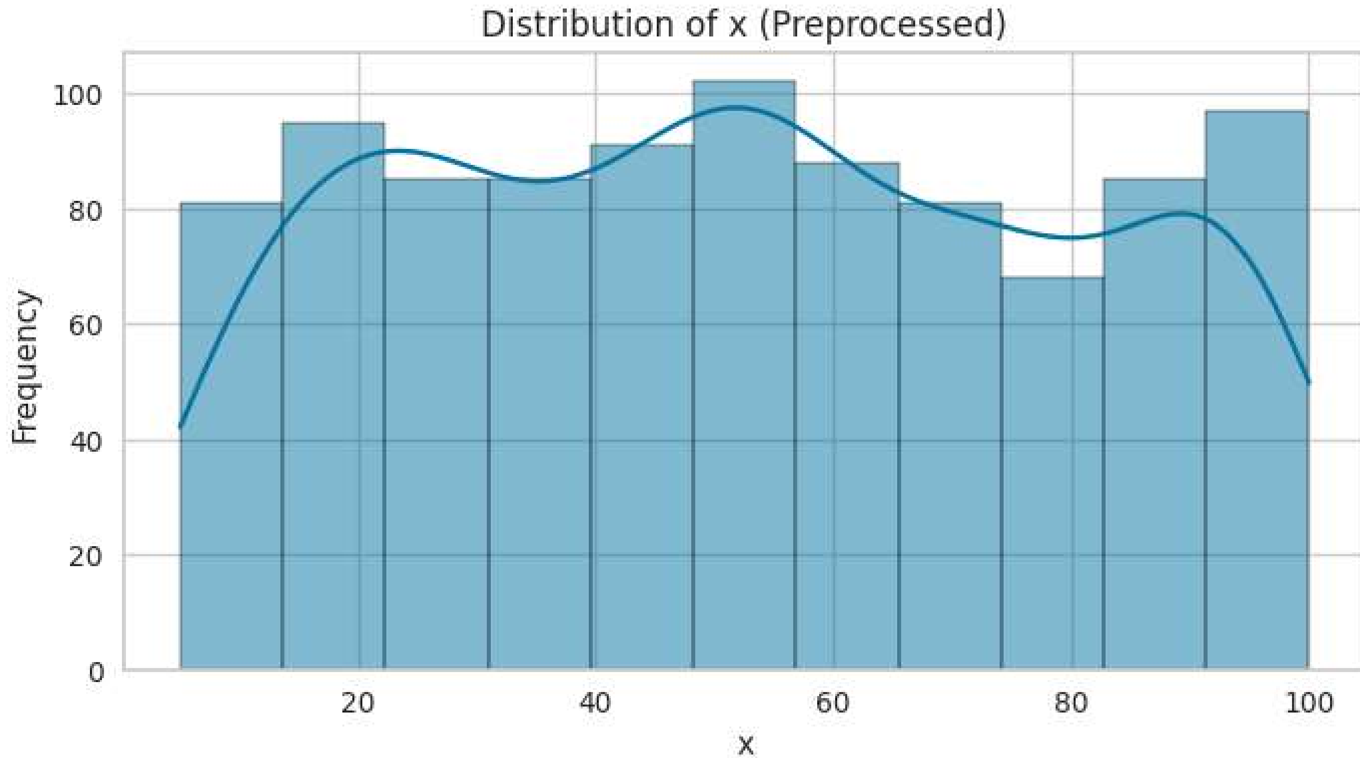
In [5]:

```python
# Create an instance of the DataScienceProject class
dsp = DataScienceProject()

# Run the main pipeline
prediction = dsp.main_pipeline('/kaggle/working/combined_data.csv', 'y')
```

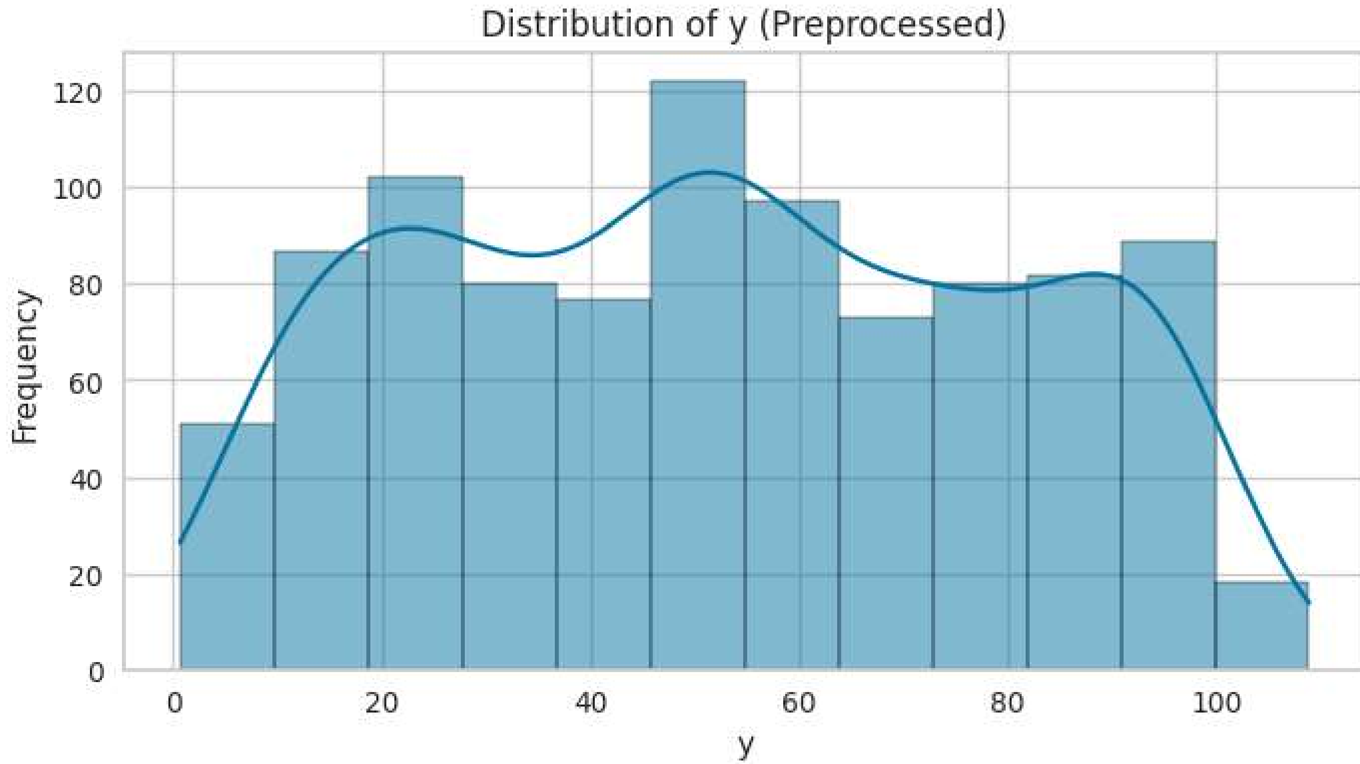
```
===== Data Science Project Pipeline =====
[Step 1] Loading and Preprocessing Data
```

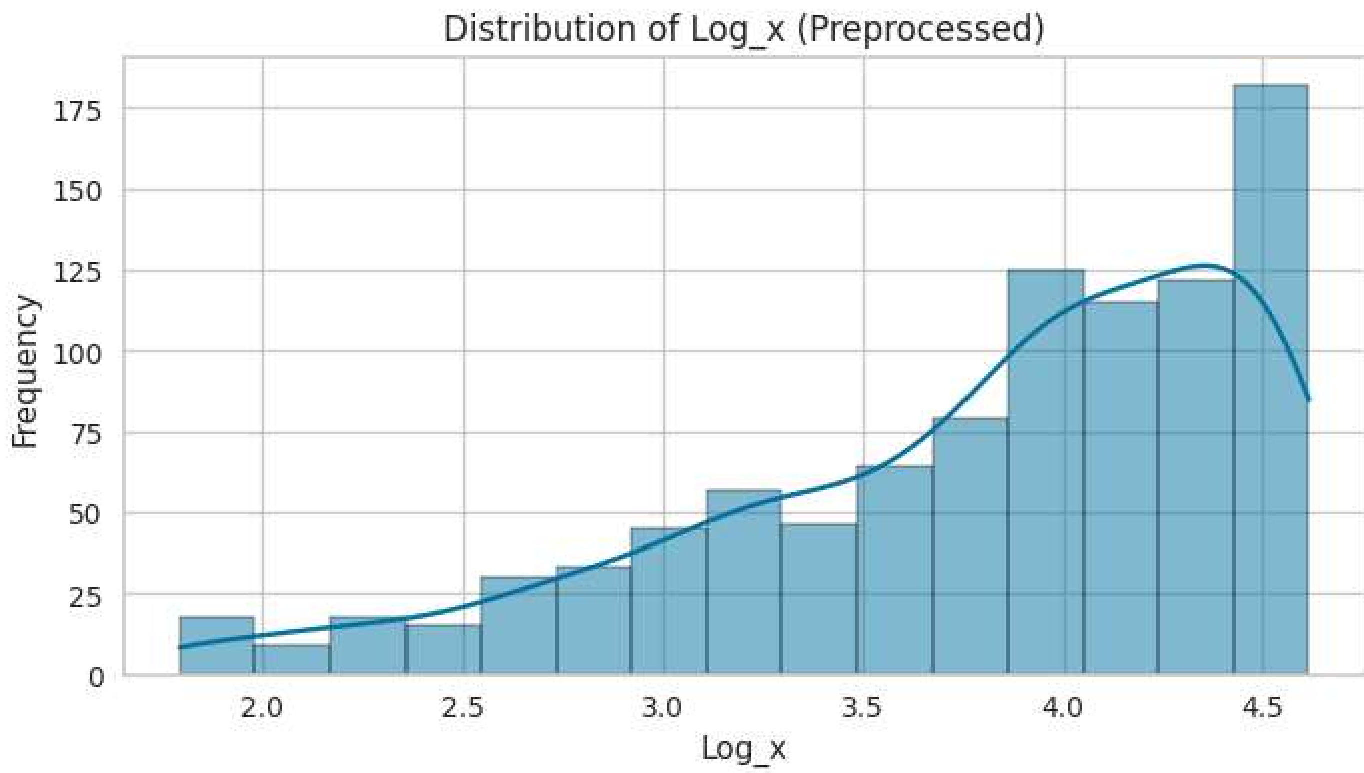

Distribution of x (Original)
Frequency
120
100
80
60
40
20
0
0
500
1000
1500
2000
2500
3000
3500
x


Distribution of y (Original)
Frequency
y
0
20
40
60
80
100
120
0
20
40
60
80
100

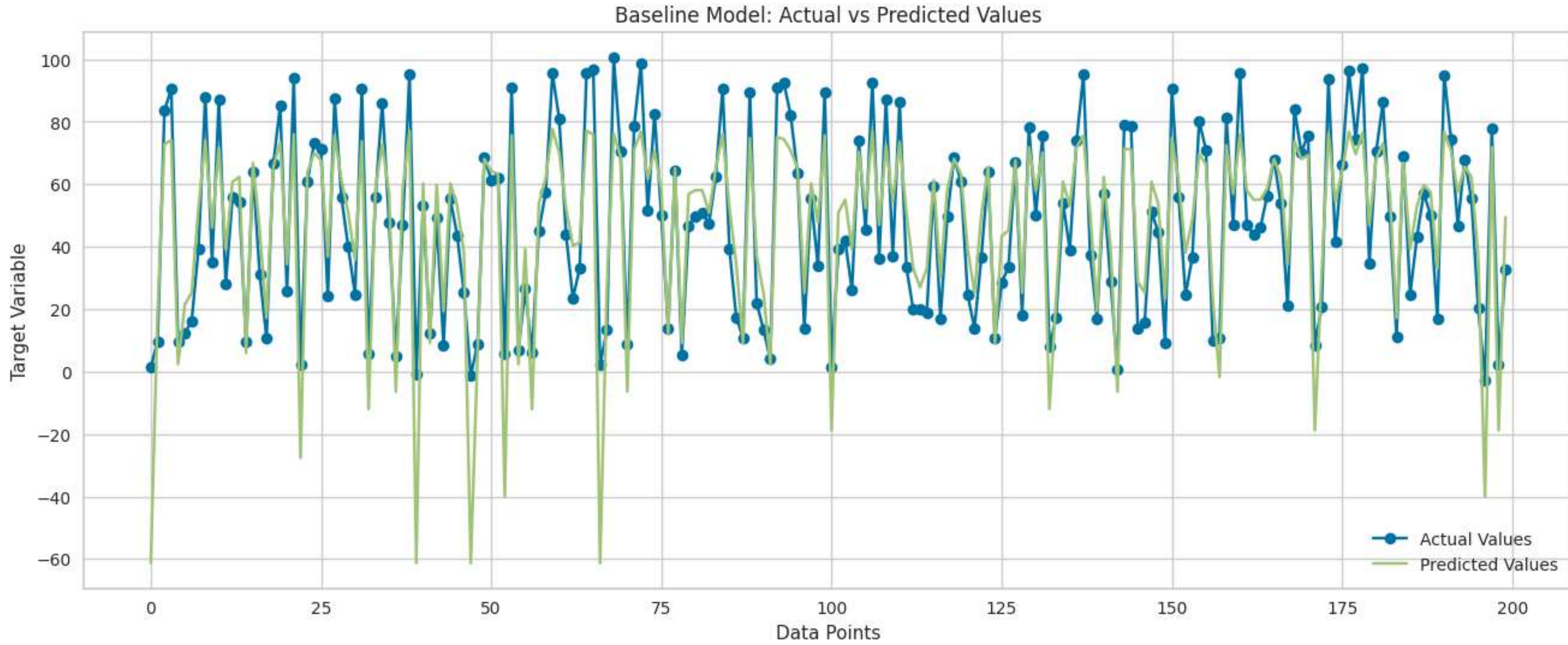


Distribution of x (Preprocessed)
Frequency
0
20
40
60
80
100
20
40
60
80
100
x

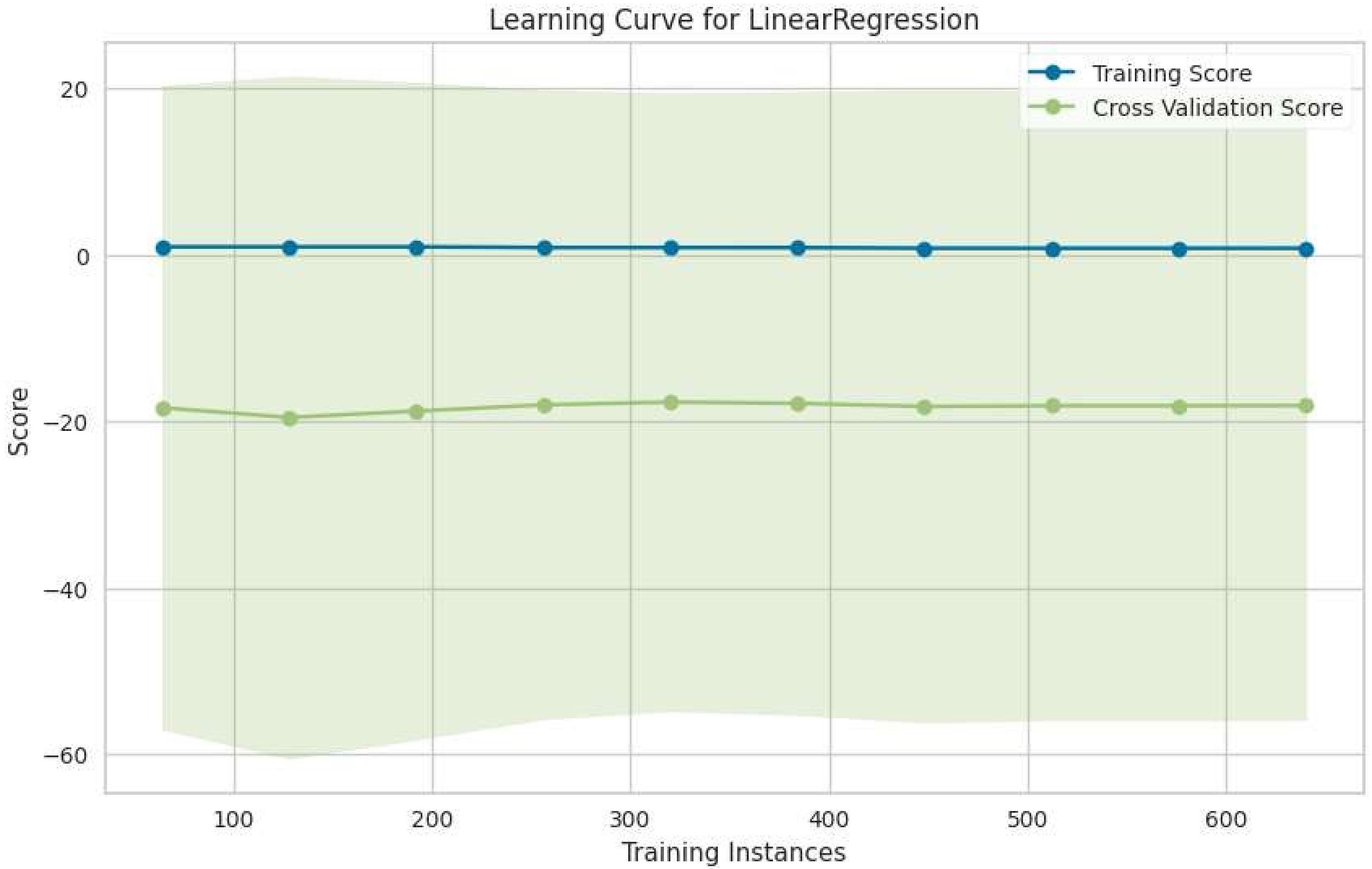

Distribution of y (Preprocessed)
Frequency
0
20
40
60
80
100
120
0
20
40
60
80
100
y



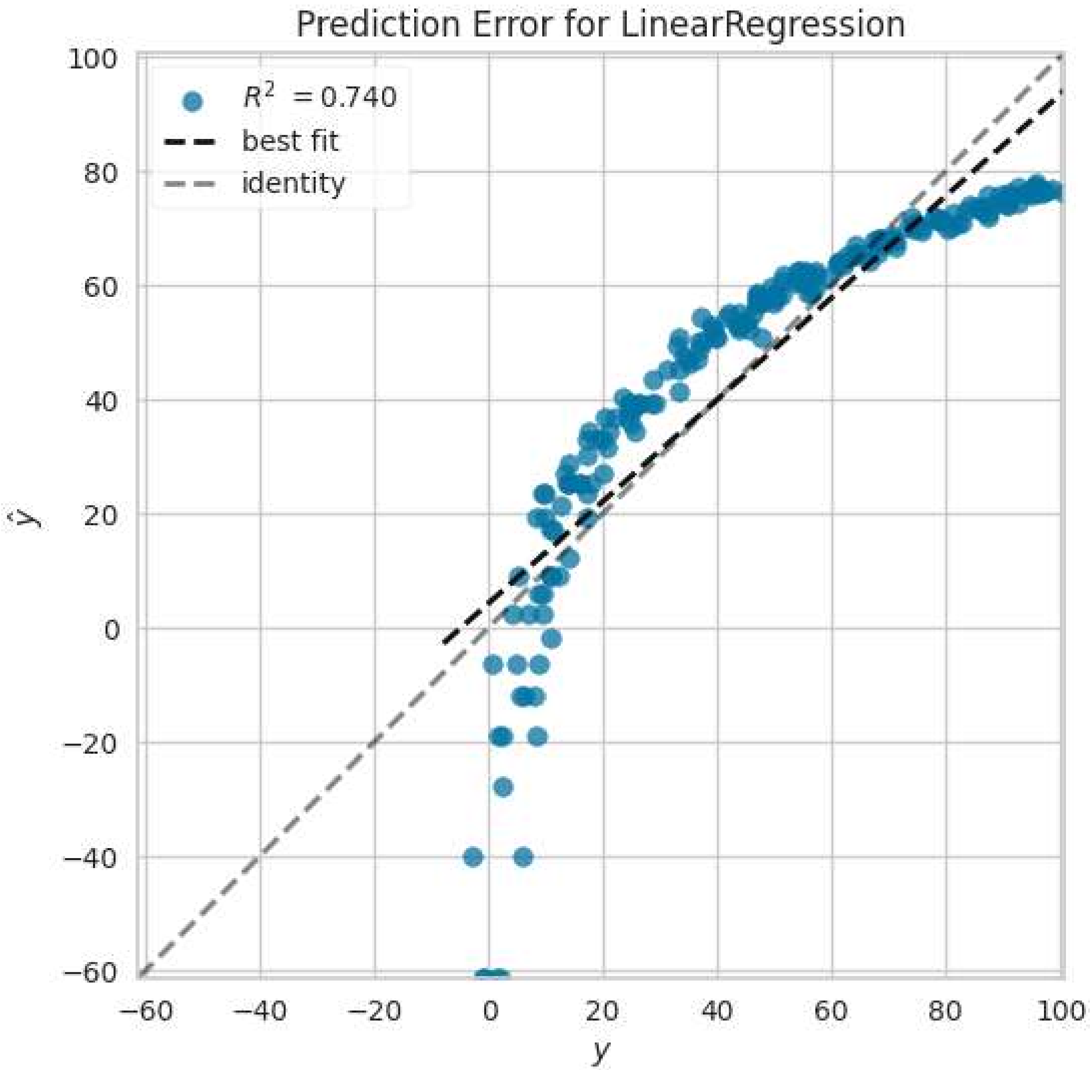
[Step 2] Training and Evaluating Baseline Model



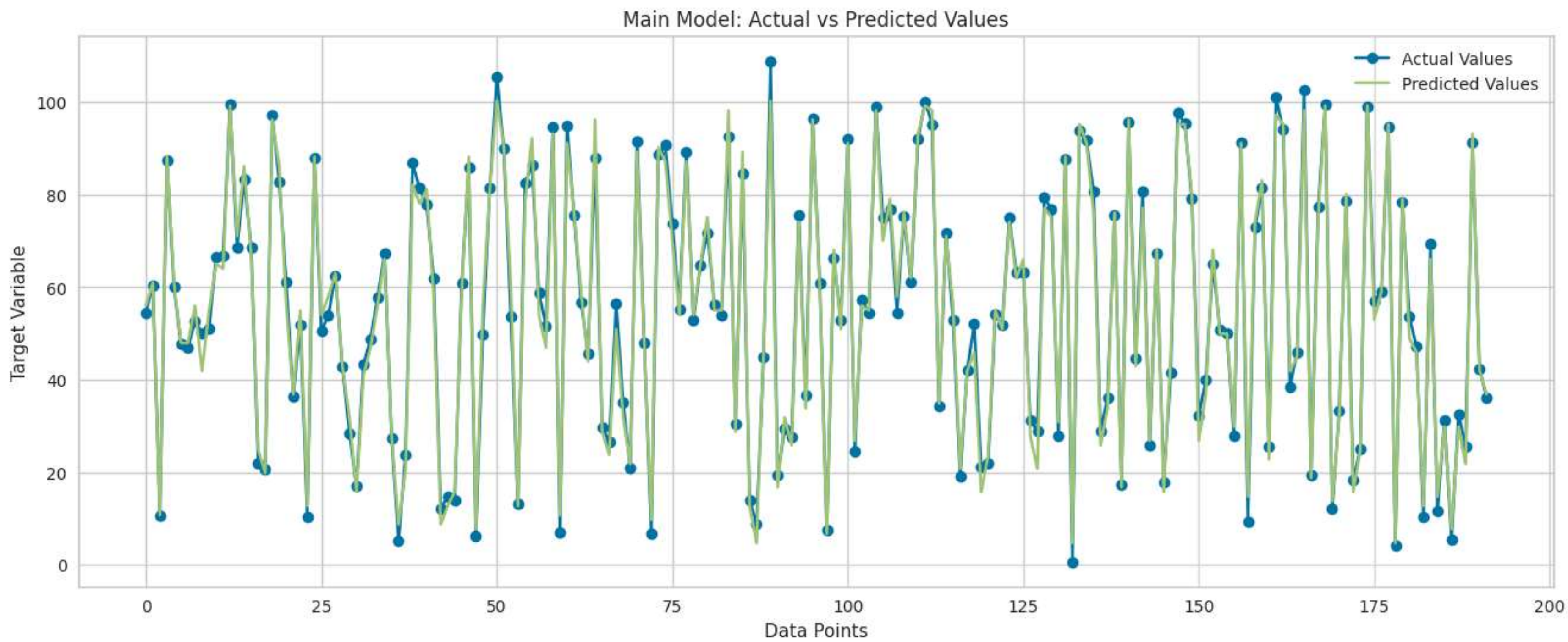
Baseline Model: Actual vs Predicted Values
Target Variable
Data Points
Actual Values
Predicted Values
100
80
60
40
20
0
−20
−40
−60
0
25
50
75
100
125
150
175
200

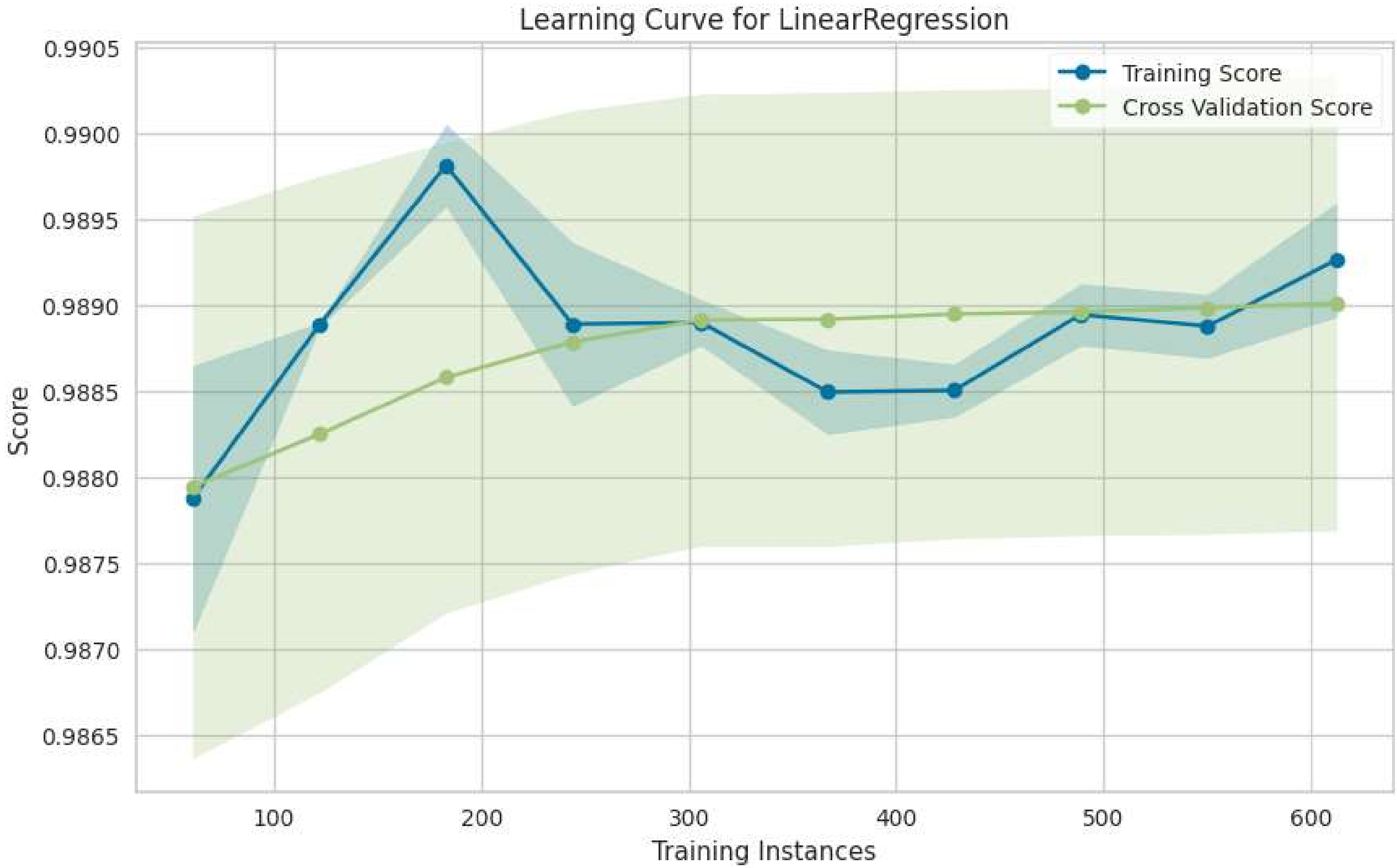



```
/opt/conda/lib/python3.10/site-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but Line
arRegression was fitted with feature names
  warnings.warn(
```

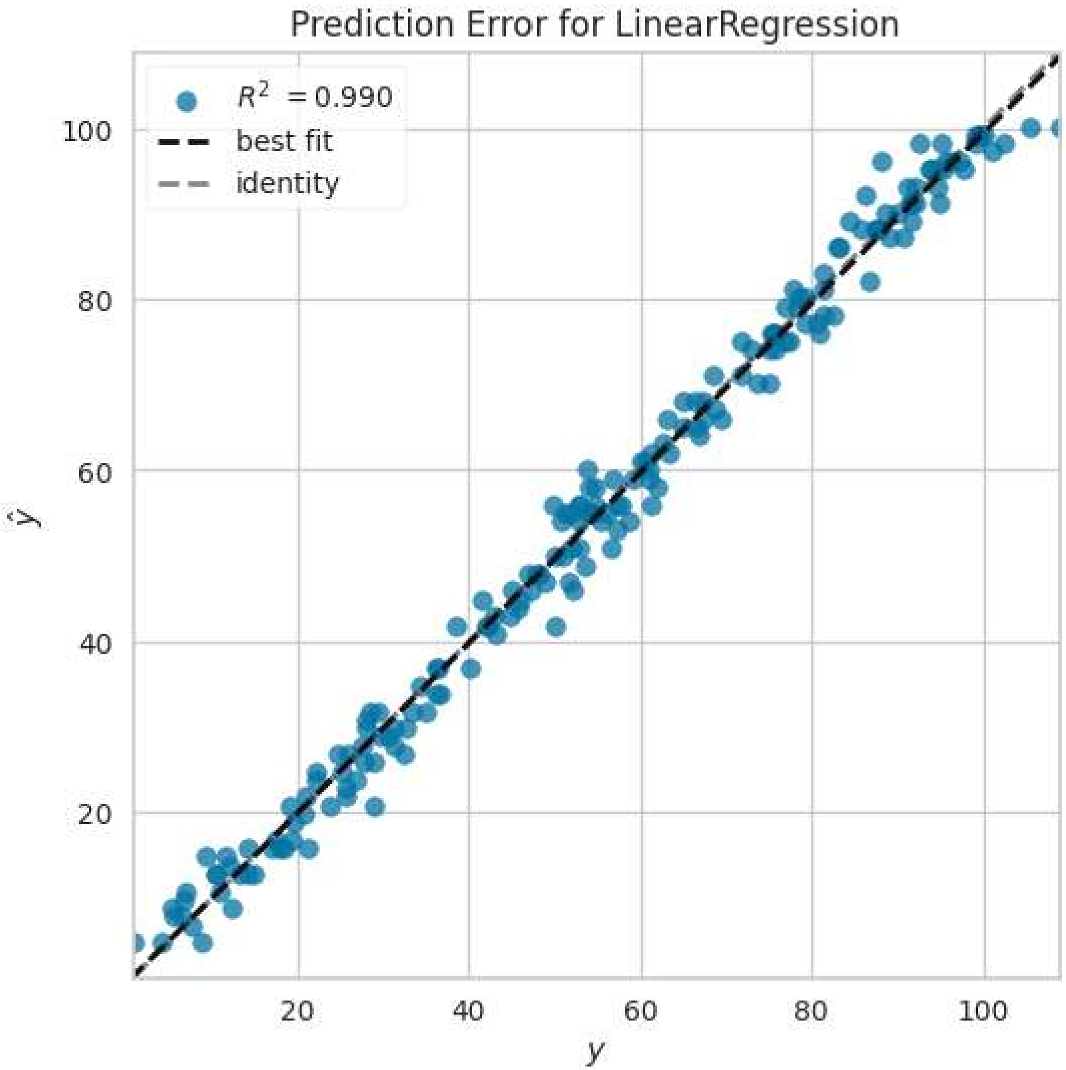



```
[Step 3] Training and Evaluating Main Model
```

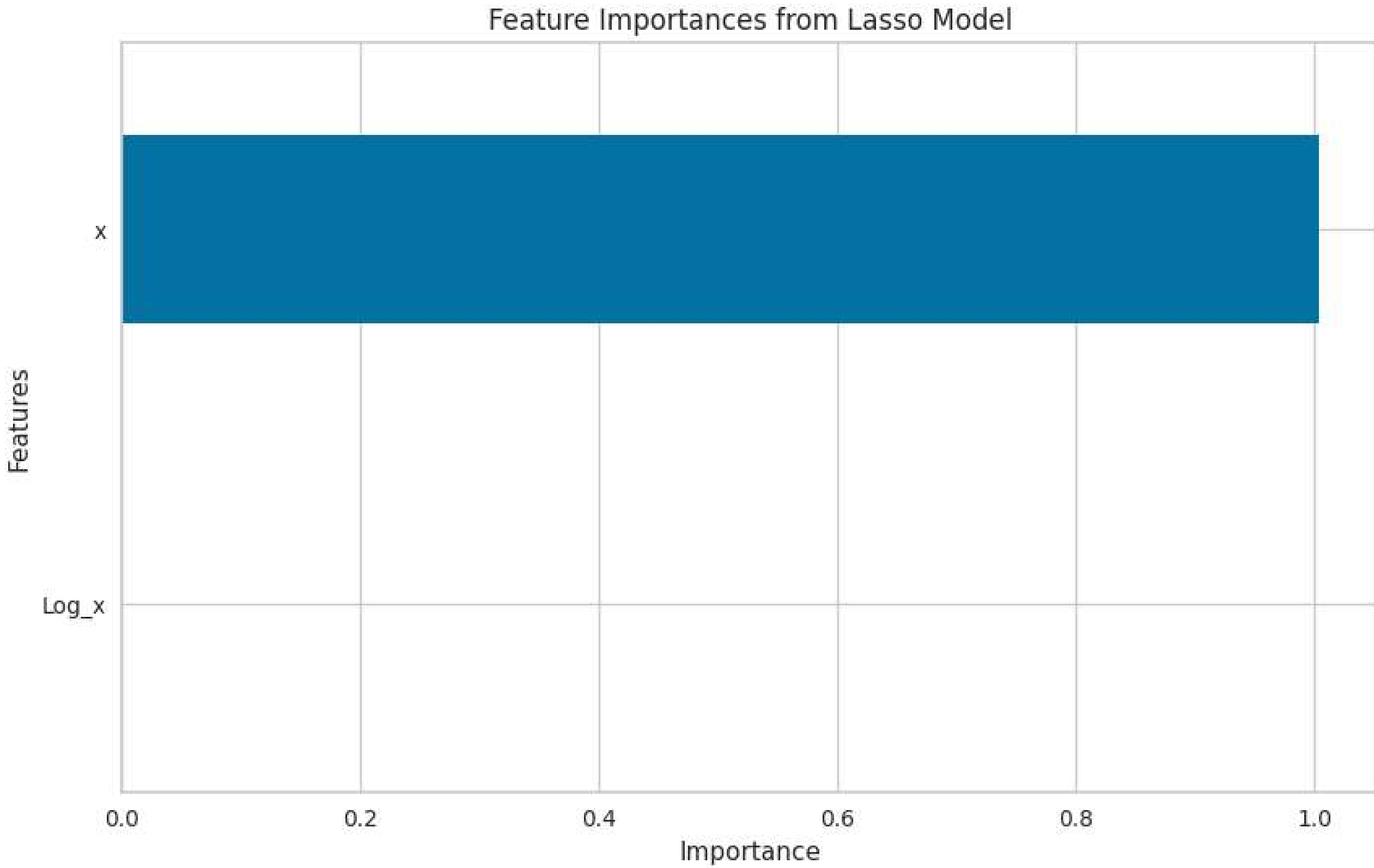

Main Model: Actual vs Predicted Values
Actual Values
Predicted Values
Target Variable
Data Points
0
20
40
60
80
100
0
25
50
75
100
125
150
175
200




```
/opt/conda/lib/python3.10/site-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but Line
arRegression was fitted with feature names
  warnings.warn(
```




```
[Step 4] Performing Lasso Feature Importance Analysis
```


Feature Importances from Lasso Model
x
Log_x
Features
0.0
0.2
0.4
0.6
0.8
1.0
Importance

```
Best alpha: 0.7556266916484009
MSE: 8.213646409156498
R-squared: 0.99

===== Model Comparison Results =====
Baseline Model:
  - MSE: 225.968
  - R-squared: 0.740
  - MAE: 2.266
Main Model:
  - MSE: 8.219
  - R-squared: 0.990
  - MAE: 11.491
====================================
```

In [ ]: