Спочатку я просто зчитав дані з csv файла у датафрейм та вивів його.

In [36]:
```
# dataset: https://www.kaggle.com/datasets/dansbecker/melbourne-housing-snapshot
df = pd.read_csv('./data/melb_data.csv')
df.head()
```

Out[36]:

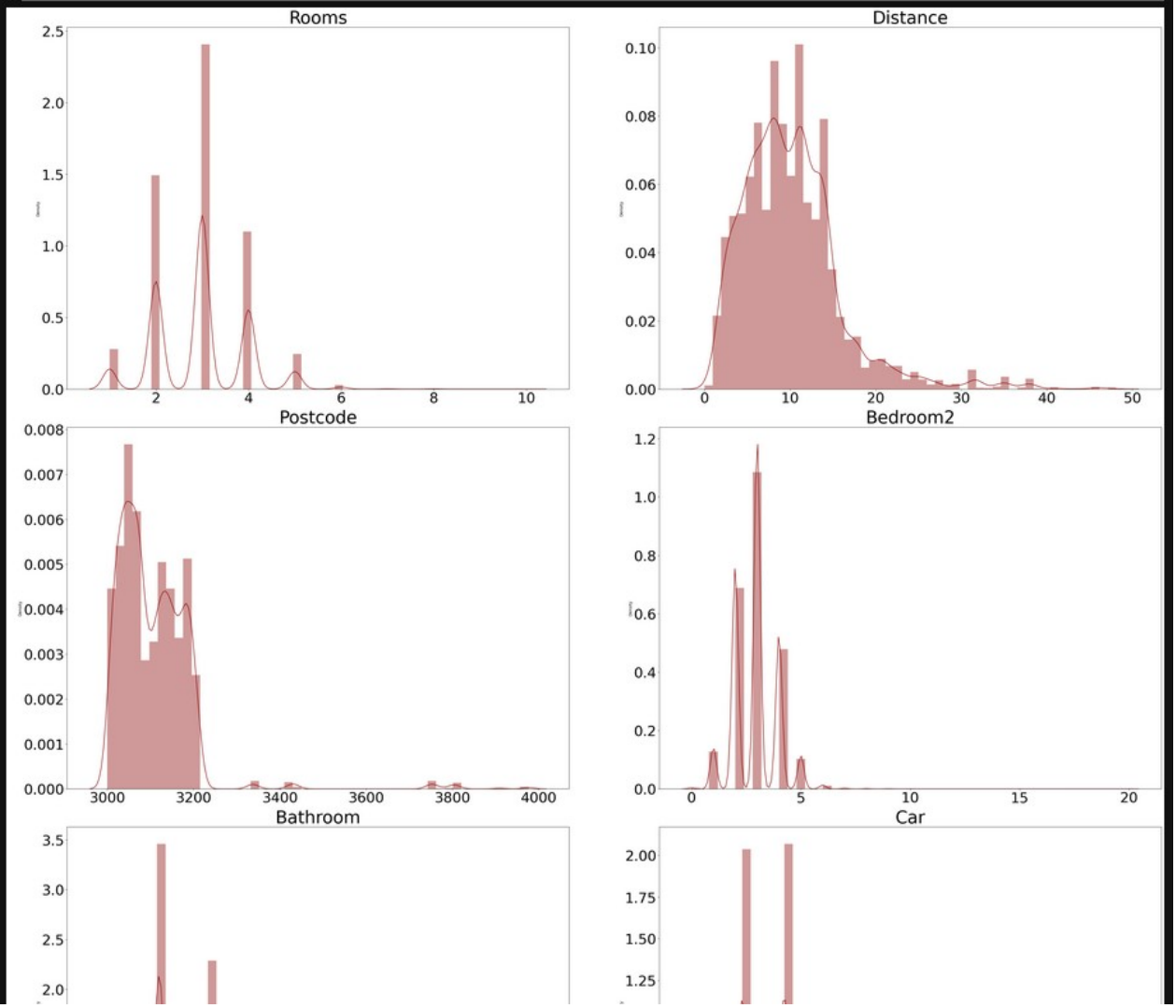
	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize	Bu
0	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0	
1	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0	
2	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0	
3	Abbotsford	40 Federation La	3	h	850000.0	PI	Biggin	4/03/2017	2.5	3067.0	...	2.0	1.0	94.0	
4	Abbotsford	55a Park St	4	h	1600000.0	VB	Nelson	4/06/2016	2.5	3067.0	...	1.0	2.0	120.0	

5 rows × 21 columns

Далі я намалював distplot для усіх числових колонок, щоб подивитися на розподіли, зрозуміти чи потрібно видаляти викиди чині.

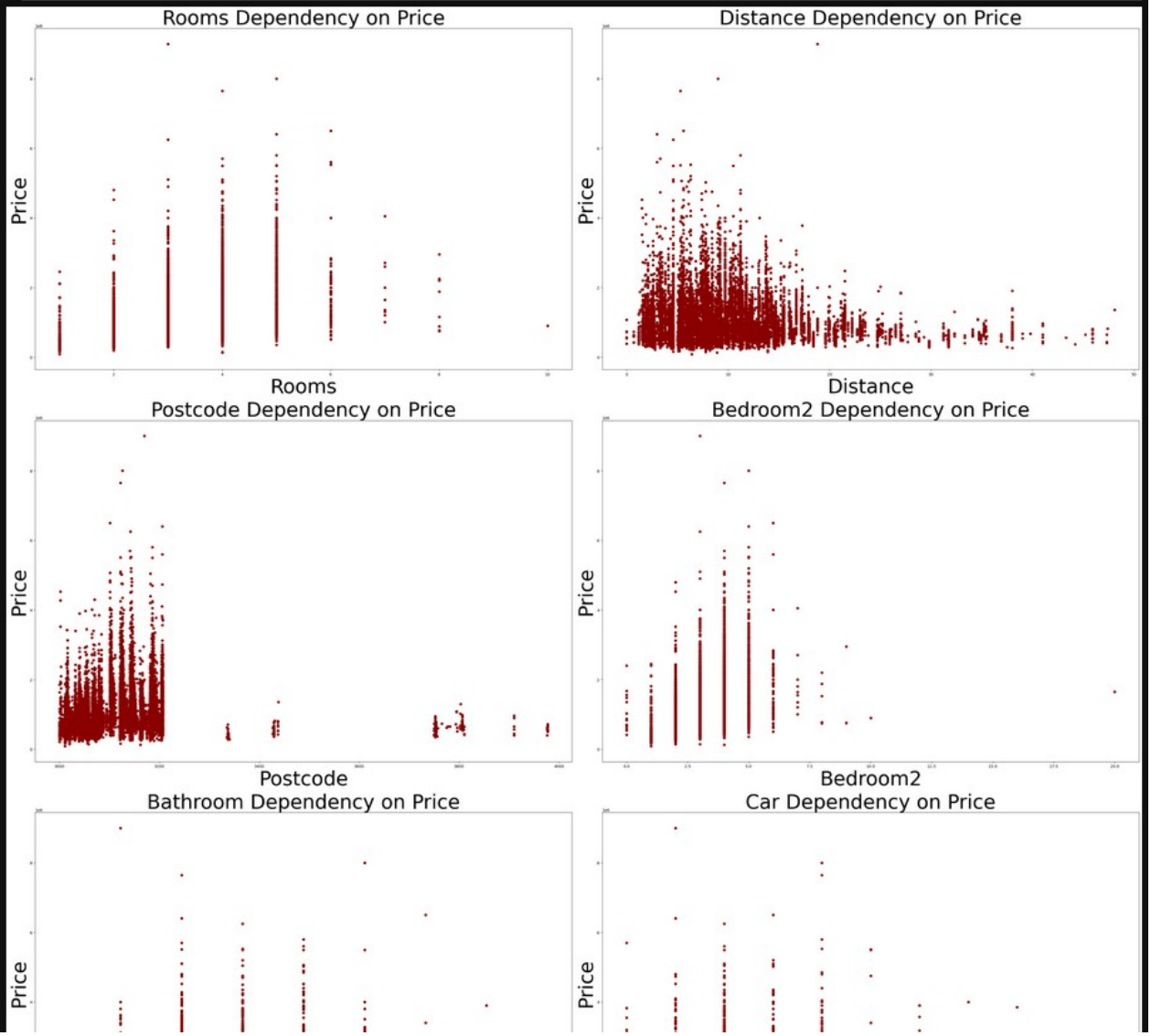
In [38]:

```
# Select numeric columns from the dataframe df:
numeric_columns = df.select_dtypes(include=['int', 'float'])
distplot(df, numeric_columns)
```

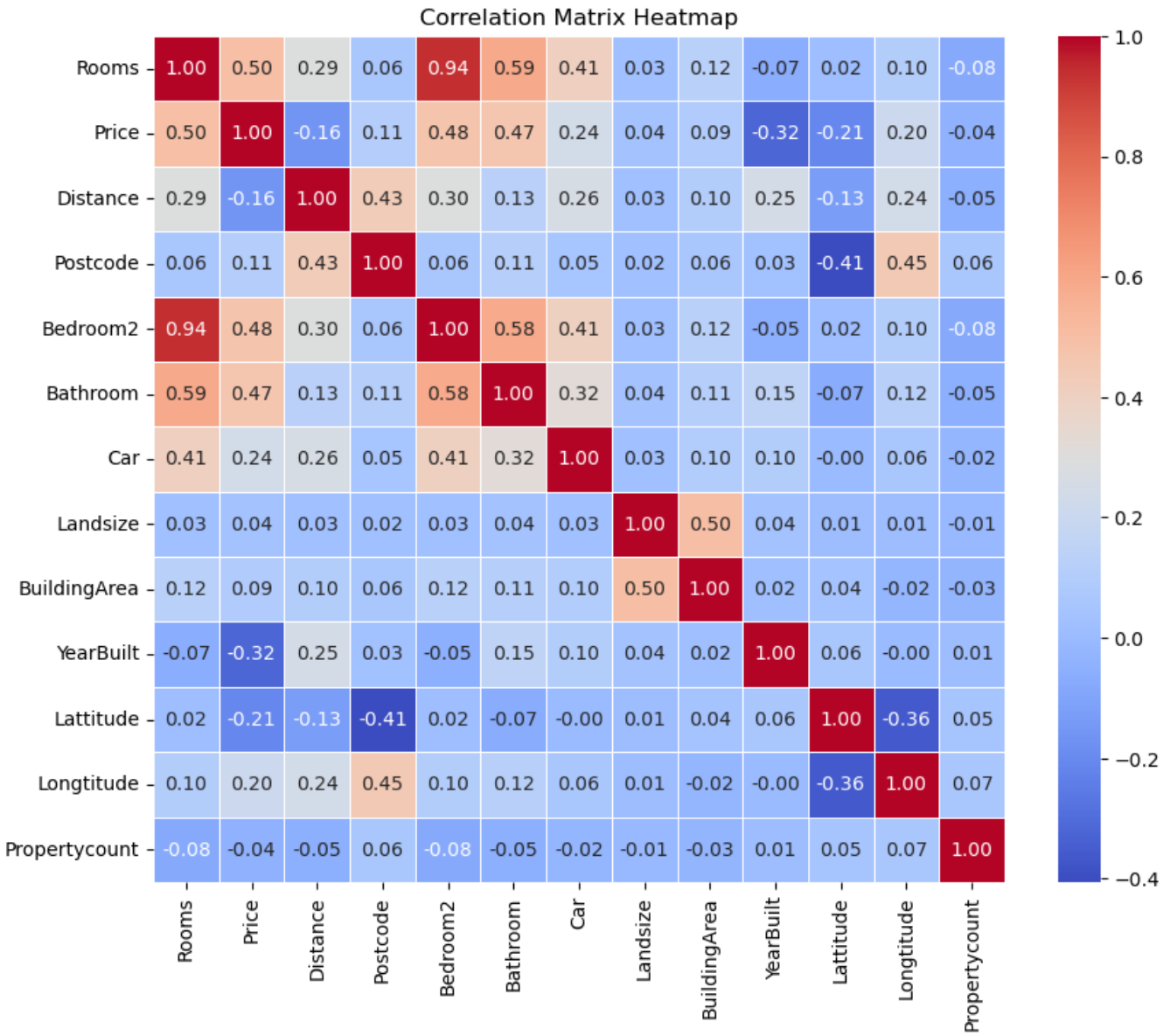


А також scatterplot-и, щоб подивитися, як числові стовбчики залежать від таргетної змінної.

```
In [39]: scatterplot(df, numeric_columns)
```



Також вивів кореляційну матрицю, щоб мати уявлення про змінні, які корелюють з тергетною змінною



Як можемо бачити, тут немає прямо вираженої кореляції з таргетом, але з стовбцем Price, непогано корелюють такі фічі: Rooms, Bedroom2, Bathroom та YearBuilt.
Я вивів інформацію про датафрейм, щоб подивитися чи є в ньому пропущені значення.

```
In [37]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13580 entries, 0 to 13579
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Suburb         13580 non-null  object
 1   Address        13580 non-null  object
 2   Rooms          13580 non-null  int64
 3   Type           13580 non-null  object
 4   Price          13580 non-null  float64
 5   Method         13580 non-null  object
 6   SellerG        13580 non-null  object
 7   Date           13580 non-null  object
 8   Distance       13580 non-null  float64
 9   Postcode       13580 non-null  float64
 10  Bedroom2       13580 non-null  float64
 11  Bathroom       13580 non-null  float64
 12  Car            13518 non-null  float64
 13  Landsize       13580 non-null  float64
 14  BuildingArea   7130 non-null   float64
 15  YearBuilt      8205 non-null   float64
 16  CouncilArea    12211 non-null  object
 17  Lattitude      13580 non-null  float64
 18  Longtitude     13580 non-null  float64
 19  Regionname     13580 non-null  object
 20  Propertycount  13580 non-null  float64
dtypes: float64(12), int64(1), object(8)
memory usage: 2.2+ MB
```

Після цього я заповнюю ці пропущенні значення, в колонці Car, YearBuilt та CouncilArea я заміняю їх на медіани, оскільки це категоріальні колонки, а в колонці BuildingArea заміняю на середнє.

```
In [42]:
# Fill missing values in the 'Car' column with the median of the column
df['Car'].fillna(df['Car'].median(), inplace=True)

# Fill missing values in the 'BuildingArea' column with the mean of the column
df['BuildingArea'].fillna(df['BuildingArea'].mean(), inplace=True)

# Fill missing values in the 'YearBuilt' column with the median of the column
df['YearBuilt'].fillna(df['YearBuilt'].median(), inplace=True)

# Fill missing values in the 'CouncilArea' column with the most common value (mode)
df['CouncilArea'].fillna(df['CouncilArea'].value_counts()[0], inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13580 entries, 0 to 13579
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Suburb         13580 non-null  object
 1   Address        13580 non-null  object
 2   Rooms          13580 non-null  int64
 3   Type           13580 non-null  object
 4   Price          13580 non-null  float64
 5   Method         13580 non-null  object
 6   SellerG        13580 non-null  object
 7   Date           13580 non-null  object
 8   Distance       13580 non-null  float64
 9   Postcode       13580 non-null  float64
 10  Bedroom2       13580 non-null  float64
 11  Bathroom       13580 non-null  float64
 12  Car            13580 non-null  float64
 13  Landsize       13580 non-null  float64
 14  BuildingArea   13580 non-null  float64
 15  YearBuilt      13580 non-null  float64
 16  CouncilArea    13580 non-null  object
 17  Lattitude      13580 non-null  float64
 18  Longtitude     13580 non-null  float64
 19  Regionname     13580 non-null  object
 20  Propertycount  13580 non-null  float64
dtypes: float64(12), int64(1), object(8)
memory usage: 2.2+ MB
```

Після цього я вирішив витягнути фічі з колонки Date(дата продажу будинку у форматі DD/MM/YYYY), а також почистити колонку Address, яка має формат №будику вулиця.

In [44]:

```
# Remove numeric and alphanumeric characters from the 'Address' column using regular expressions
df['Address'] = df['Address'].str.replace(r'[0-9]+[a-zA-Z]*', '', regex=True).str.strip()

# Convert the 'Date' column to a datetime format with the specified format
df['Date'] = pd.to_datetime(df['Date'], format='%d/%m/%Y')

# Extract the year from the 'Date' column and create a new 'Year' column
df['Year'] = df['Date'].dt.year

# Extract the month from the 'Date' column and create a new 'Month' column
df['Month'] = df['Date'].dt.month

# Extract the weekend information from the Date column and create a new 'Is weekend' column
df['Is weekend'] = df['Date'].apply(is_weekend_or_weekday)

# Drop the original 'Date' column from the DataFrame
df = df.drop('Date', axis=1)
```

З колонки Address я видалив номери будинків, за допомогою регулярного виразу і залишив тільки назви вулиць. А з колонки Date я повитягував рік продажу, місяць продажу, а також чи був проданий будинок у вихідний день чи ні.
Далі я видалив з датафрейму колонки Postcode, Lattitude, Longtitude, Suburb оскільки точність регресії з цими колонками була зовсім погана.

In [45]:

```
# Check if the 'Postcode' column exists in the DataFrame
if 'Postcode' in df.columns:
    # If it exists, drop the 'Postcode' column from the DataFrame
    df.drop('Postcode', axis=1, inplace=True)

# Check if the 'Lattitude' column exists in the DataFrame
if 'Lattitude' in df.columns:
    # If it exists, drop the 'Lattitude' column from the DataFrame
    df.drop('Lattitude', axis=1, inplace=True)

# Check if the 'Longtitude' column exists in the DataFrame
if 'Longtitude' in df.columns:
    # If it exists, drop the 'Longtitude' column from the DataFrame
    df.drop('Longtitude', axis=1, inplace=True)

# Check if the 'Suburb' column exists in the DataFrame
if 'Suburb' in df.columns:
    # If it exists, drop the 'Suburb' column from the DataFrame
    df.drop('Suburb', axis=1, inplace=True)
```

Далі я використав One-Hot для того, щоб закодувати категоріальні стовбці.

In [ ]:

```
# Define a list of categorical columns to be one-hot encoded
categorical_cols = ['Address', 'Type', 'Method', 'CouncilArea', 'SellerG', 'Regionname', 'YearBuilt', 'Year',
                    'Month', 'Is weekend']

# Create a new DataFrame 'df_encoded' by applying one-hot encoding to the selected categorical columns
encoded_df = pd.get_dummies(df[categorical_cols].astype(str), columns=categorical_cols)

# Concatenate the one-hot encoded columns with the original DataFrame, dropping the original categorical columns
# This creates a new DataFrame 'encoded_df' with one-hot encoded categorical variables and the remaining numerica
encoded_df = pd.concat([df.drop(categorical_cols, axis=1), encoded_df], axis=1).astype(int)
encoded_df.head()
```

Out[46]:

	Rooms	Price	Distance	Bedroom2	Bathroom	Car	Landsize	BuildingArea	Propertycount	Address_/ Abbotsford Gr	...	Month_2	Month_3
0	2	1480000	2	2	1	1	202	151	4019	0	...	0	0
1	2	1035000	2	2	1	0	156	79	4019	0	...	1	0
2	3	1465000	2	3	2	0	134	150	4019	0	...	0	1
3	3	850000	2	3	2	1	94	151	4019	0	...	0	1
4	4	1600000	2	3	1	2	120	142	4019	0	...	0	0

5 rows × 6725 columns

Після цього я засплітив дані на навчальну та тестову вибірку.

In [48]:

```
# Split the DataFrame into features (X) and the target variable (y)
X = encoded_df.drop('Price', axis=1)
y = encoded_df['Price']

# Split the data into training and testing sets using train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Ну і після цього я вже треную лінійну регресію, лінійну регресію з L2 регуляризацією(Ridge), а також xgboost.Regressor в основі якого лінійні регресії, а також вимірюю результати кожного методу на навчальній вибірці. Найкращі результати показала лінійна регресія з L2 регуляризацією, але я не зміг виконати GridSearch для xgboost.Regressor оскільки, для цього потрібні обчислювальні ресурси, яких у мене немає.

In [49]:

```
# Create a Linear Regression model
model = LinearRegression()

# Train (fit) the model using the training data
model.fit(X_train, y_train)

# Use the trained model to make predictions on the train data
y_pred_train = model.predict(X_train)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_train = r2_score(y_train, y_pred_train)

# Print the best model performance
print(f"Train: R-squared (R^2) Score: {r2_train}")
print(f"Train: Mean Squared Error (MSE): {mse_train}")

# Use the trained model to make predictions on the test data
y_pred_test = model.predict(X_test)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_test = mean_squared_error(y_test, y_pred_test)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_test = r2_score(y_test, y_pred_test)

# Print the best model performance
print(f"Test: R-squared (R^2) Score: {r2_test}")
print(f"Test: Mean Squared Error (MSE): {mse_test}")
```

```
Train: R-squared (R^2) Score: 0.8821540153615146
Train: Mean Squared Error (MSE): 48499996026.22382
Test: R-squared (R^2) Score: 0.6137605952202531
Test: Mean Squared Error (MSE): 153419642558.5499
```

In [50]:

```python
# Create a Ridge model
model = Ridge(alpha=4)

# Train (fit) the model using the training data
model.fit(X_train, y_train)

# Use the trained model to make predictions on the train data
y_pred_train = model.predict(X_train)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_train = r2_score(y_train, y_pred_train)

# Print the best model performance
print(f"Train: R-squared (R^2) Score: {r2_train}")
print(f"Train: Mean Squared Error (MSE): {mse_train}")

# Use the trained model to make predictions on the test data
y_pred_test = model.predict(X_test)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_test = mean_squared_error(y_test, y_pred_test)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_test = r2_score(y_test, y_pred_test)

# Print the best model performance
print(f"Test: R-squared (R^2) Score: {r2_test}")
print(f"Test: Mean Squared Error (MSE): {mse_test}")
```

```
Train: R-squared (R^2) Score: 0.7681830543024877
Train: Mean Squared Error (MSE): 95405210280.44421
Test: R-squared (R^2) Score: 0.6857348822926175
Test: Mean Squared Error (MSE): 124830458598.03333
```

In [51]:

```python
# Create a XGBRegressor model
model = xgb.XGBRegressor(booster='gblinear', eval_metric='rmse', reg_lambda=0.1, n_estimators=1000)

# Train (fit) the model using the training data
model.fit(X_train, y_train)

# Use the trained model to make predictions on the train data
y_pred_train = model.predict(X_train)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_train = r2_score(y_train, y_pred_train)

# Print the best model performance
print(f"Train: R-squared (R^2) Score: {r2_train}")
print(f"Train: Mean Squared Error (MSE): {mse_train}")

# Use the trained model to make predictions on the test data
y_pred_test = model.predict(X_test)

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_test = mean_squared_error(y_test, y_pred_test)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_test = r2_score(y_test, y_pred_test)

# Print the best model performance
print(f"Test: R-squared (R^2) Score: {r2_test}")
print(f"Test: Mean Squared Error (MSE): {mse_test}")
```

```
Train: R-squared (R^2) Score: 0.5922277808718404
Train: Mean Squared Error (MSE): 167820321311.66556
Test: R-squared (R^2) Score: 0.6175070037967498
Test: Mean Squared Error (MSE): 151931517169.03354
```

І на останок я натренував кастомну лінійну регресію не на всіх даних, а тільки на частині, оскільки це знову ж таки вимагає обчислювальних ресурсів, яких в мене немає і порівняв результати зі вбудованою регресію. Результати майже однакові, похибка може бути через якісь округлення, які робить вбудована регресія.

In [52]:

```python
# Create a CustomLinearRegression model
model = CustomLinearRegression()

# Train (fit) the model using the training data
model.fit(X_train[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea',
                   'Propertycount']], y_train)

# Use the trained model to make predictions on the train data
y_pred_train = model.predict(X_train[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize',
                                      'BuildingArea', 'Propertycount']])

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_train = r2_score(y_train, y_pred_train)

# Print the best model performance
print(f"Train: R-squared (R^2) Score: {r2_train}")
print(f"Train: Mean Squared Error (MSE): {mse_train}")

# Use the trained model to make predictions on the test data
y_pred_test = model.predict(X_test[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea',
                   'Propertycount']])

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_test = mean_squared_error(y_test, y_pred_test)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_test = r2_score(y_test, y_pred_test)

# Print the best model performance
print(f"Test: R-squared (R^2) Score: {r2_test}")
print(f"Test: Mean Squared Error (MSE): {mse_test}")
```

```
Train: R-squared (R^2) Score: 0.3896325299473309
Train: Mean Squared Error (MSE): 251199223825.08786
Test: R-squared (R^2) Score: 0.40671787357459444
Test: Mean Squared Error (MSE): 235659879976.42792
```

In [53]:

```python
# Create a Linear Regression model
model = LinearRegression()

# Train (fit) the model using the training data
model.fit(X_train[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea',
                   'Propertycount']], y_train)

# Use the trained model to make predictions on the train data
y_pred_train = model.predict(X_train[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize',
                                      'BuildingArea', 'Propertycount']])

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_train = r2_score(y_train, y_pred_train)

# Print the best model performance
print(f"Train: R-squared (R^2) Score: {r2_train}")
print(f"Train: Mean Squared Error (MSE): {mse_train}")

# Use the trained model to make predictions on the test data
y_pred_test = model.predict(X_test[['Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea
                                    'Propertycount']])

# Calculate the Mean Squared Error (MSE) to assess the model's performance
mse_test = mean_squared_error(y_test, y_pred_test)

# Calculate the R-squared (R^2) score to assess how well the model fits the data
r2_test = r2_score(y_test, y_pred_test)

# Print the best model performance
print(f"Test: R-squared (R^2) Score: {r2_test}")
print(f"Test: Mean Squared Error (MSE): {mse_test}")
```

```
Train: R-squared (R^2) Score: 0.3896325299473309
Train: Mean Squared Error (MSE): 251199223825.08786
Test: R-squared (R^2) Score: 0.40671787357455924
Test: Mean Squared Error (MSE): 235659879976.44193
```

Також ось реалізація кастомної лінійної регресії, я просто використав метод найменших квадратів.

In [32]:

```python
class CustomLinearRegression():
    def fit(self, X, y):
        """
        Fit linear model.

        Argument:
        X -- {array-like, sparse matrix} of shape (n_samples, n_features)
            Training data.

        y -- array-like of shape (n_samples,) or (n_samples, n_targets)
            Target values. Will be cast to X's dtype if necessary.
        """
        # Convert X to a NumPy array
        Z = np.array(X)

        # Add a column of ones to represent the intercept term
        ones_column = np.ones((Z.shape[0], 1))
        Z = np.column_stack((ones_column, Z))

        # Calculate the least squares estimation coefficients
        self.least_squares_estimation = np.dot(np.dot(np.linalg.inv(np.dot(Z.T, Z)), Z.T), y)

    def predict(self, X):
        """
        Predict using the linear model.

        Argument:
        X -- array-like or sparse matrix, shape (n_samples, n_features)
            Samples.

        Returns:
        prediction -- array, shape (n_samples,)
            Returns predicted values.
        """
        # Convert X to a NumPy array
        X = np.array(X)

        # Calculate predictions using the linear model equation
        prediction = self.least_squares_estimation[0] + np.dot(self.least_squares_estimation[1:], X.T)

        return prediction
```