Linear Regression

Import Packages

In [22]:
```
# Import Library for Working with Tabular Data
import pandas as pd
# Import Library for Numerical Computing
import numpy as np
# Import Library for Data Visualization
import matplotlib.pyplot as plt
# Import Another Library for Data Visualizations
# This makes it easier to create beautiful data visualizations using matplotlib.
import seaborn as sns
```

In [23]:
```
# matplotlib visualizations will embed themselves
# directly in our Jupyter Notebook. This will make them easier to
# access and interpret.
%matplotlib inline
```

Explore Dataset

In [24]:
```
# Import Housing Dataset into Jupyter Notebook Under raw_data Variable
raw_data = pd.read_csv('Housing_Data.csv')
```

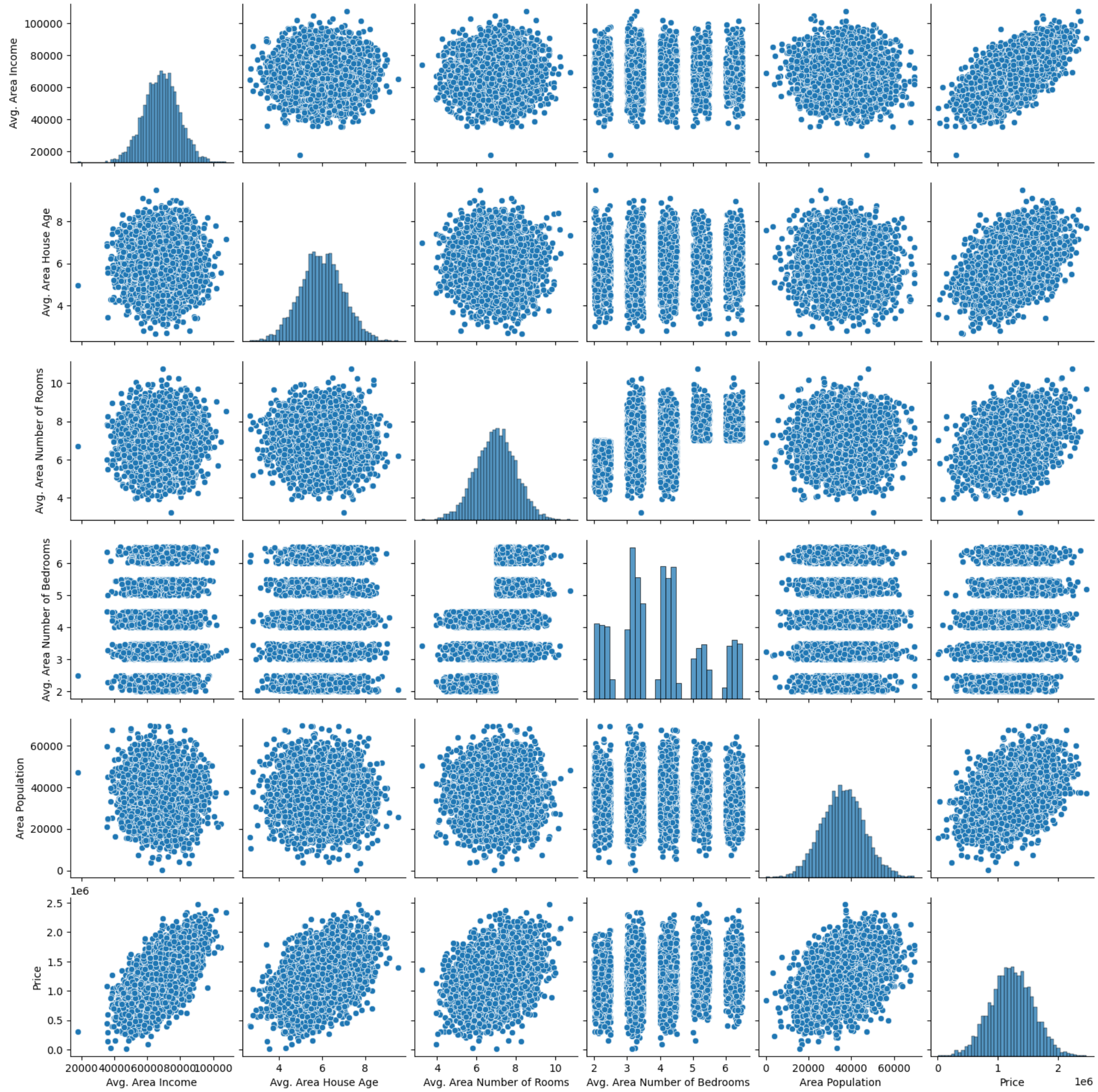
In [25]:
```
# Can use info method to get some high-level information about the dataset.
raw_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
 #   Column                        Non-Null Count  Dtype  
---  ------                        --------------  -----  
 0   Avg. Area Income              5000 non-null   float64
 1   Avg. Area House Age           5000 non-null   float64
 2   Avg. Area Number of Rooms     5000 non-null   float64
 3   Avg. Area Number of Bedrooms  5000 non-null   float64
 4   Area Population               5000 non-null   float64
 5   Price                         5000 non-null   float64
 6   Address                       5000 non-null   object 
dtypes: float64(6), object(1)
memory usage: 273.6+ KB
```

We can also use `seaborn` method `pairplot` to learn about this Dataset. This passes in the entire DataFrame as a parameter and provides a visual model of the dataset as opposed to above.

In [26]:
```
sns.pairplot(raw_data)
```

Out[26]: <seaborn.axisgrid.PairGrid at 0x24fc7161c30>



We can generate a list of the DataFrame's columns and will use all of these variables in x-array except for:

- Price (which we are trying to predict)
- Address (contains text)

In [27]:
```
display(raw_data.columns)
```

```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price', 'Address'],
      dtype='object')
```

Set Variables

In [28]:
```
# Create our x-array and assign it to a variable called x
x = raw_data[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms
```

```
    'Avg. Area Number of Bedrooms', 'Area Population']]
```

In [29]:
```
# Similarly, create our y-array and assign it to a variable y
y = raw_data['Price']
```

Linear Regression Model

We want the Test Data to be 30% of the Entire Dataset.

- `train_test_split` function returns a Python list of length 4, where the items are `x_train` ... `y_test` respectively.
  - List unpacking is used to assign the proper values to the correct variable names.

In [30]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

In [31]:
```
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size = 0.3)
```

In [32]:
```
model = LinearRegression().fit(x_train, y_train)
```

Linear Regression Equation

At this point model has been trained. We can now examine each of the model's coefficients.

In [33]:
```
print(model.coef_)
```

```
[2.14983304e+01 1.65238202e+05 1.19580159e+05 1.17101604e+03
 1.51382019e+01]
```

We can also view the coefficients by placing them in a DataFrame. This organizes the output with labels in a tablelike format.

In [34]:
```
pd.DataFrame(model.coef_, x.columns, columns = ['Coeff'])
```

Out[34]:

	Coeff
Avg. Area Income	21.498330
Avg. Area House Age	165238.201838
Avg. Area Number of Rooms	119580.158588
Avg. Area Number of Bedrooms	1171.016039
Area Population	15.138202

Coefficients quantify the impact of the value of the specified variable on the predicted variable.

It makes the assumption that all other variables are held constant. (i.e. Coefficient 15 for Area Population means that a 1-unit increase in the variable will result in a 15-unit increase in the predicted variable Price.)

In [35]:
```
# Can similarly see the intercept of the regression equation.
print(model.intercept_)
```

```
-2618888.0929693757
```

Make Predictions

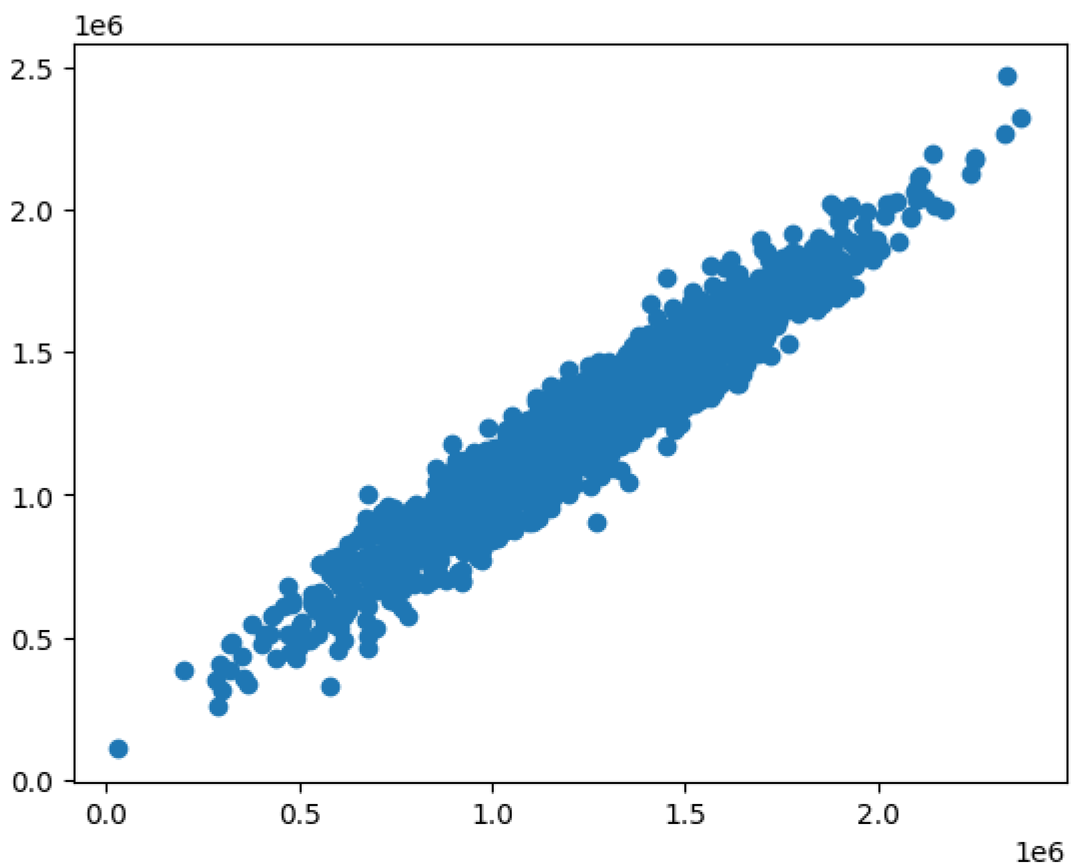
Call the predict method on the model variable to make predictions from a machine learning model using `scikit-learn`.
The predictions variable holds the predicted values of the features stored in `x_test`.

In [36]:
```
predictions = model.predict(x_test)
```

Plot the real values in `y_test` array against the predictions array using a matplotlib scatterplot.
A perfectly straight diagonal line in our scatterplot would indicate that our model perfectly predicted the `y_array` values.

In [37]:
```
plt.scatter(y_test, predictions)
```

Out[37]:
```
<matplotlib.collections.PathCollection at 0x24fca3965c0>
```

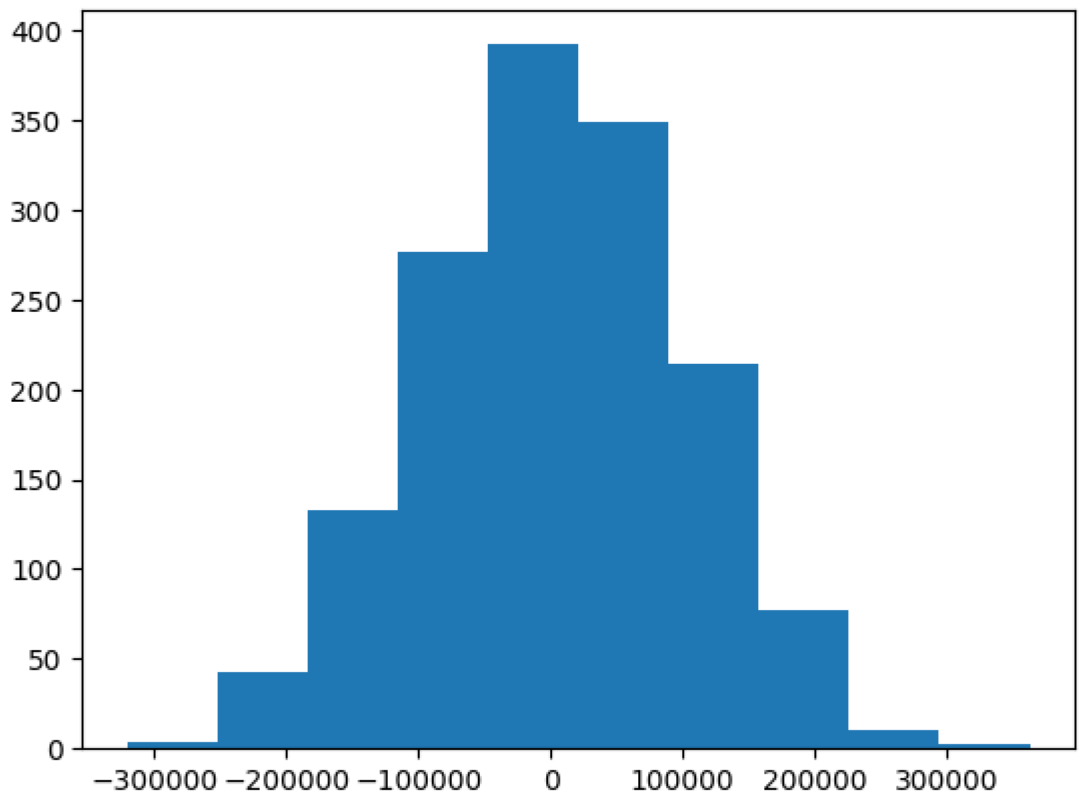


We can also plot residuals to visually assess the performance of our model. These are the difference between the actual `y_array` and the predicted `y_array` values.

If the residuals from our machine learning model appear to be normally distributed, this is a good sign that we have selected an appropriate model type (i.e. linear regression) to make predictions from our Dataset.

In [38]:
```
plt.hist(y_test - predictions)
```

Out[38]:
```
(array([  4.,  43., 133., 276., 392., 349., 214.,  77.,  10.,   2.]),
 array([-321019.02786777, -252588.96622566, -184158.90458354,
        -115728.84294143,  -47298.78129932,   21131.2803428 ,
          89561.34198491,  157991.40362702,  226421.46526914,
         294851.52691125,  363281.58855337]),
 <BarContainer object of 10 artists>)
```



Measure Performance

Three main performance metrics used for regression machine learning models:

- Mean Absolute Error
- Mean Squared Error
- Root Mean Squared Error

In [39]:
```
from sklearn import metrics
```

In [40]:
```
# Calculate Mean Absolute Error
mean_abs_error = metrics.mean_absolute_error(y_test, predictions)
print(mean_abs_error)
```

80396.69476637646

In [41]:
```
# Calculate Mean Squared Error
mean_squared_error = metrics.mean_squared_error(y_test, predictions)
display(mean_squared_error)
```

9970090400.776184

In [42]:
```
# Calculate Root Mean Squared Error (scikit-learn doesn't have built in method.)
rms_error = np.sqrt(metrics.mean_squared_error(y_test, predictions))
display(rms_error)
```

99850.34001332286