Melbourn House Price Predition Model

- created by :
  - **Student of COEP technological University, PUNE**

    1. 732392024 -- AKSHAY GIDDE

Importing Necessory Libraries

In [1]:
```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
#Magic command -- no need to use plt.show() for every graph


#for ignoring warning.
import warnings
# Ignore specific UserWarning
warnings.filterwarnings("ignore", message="The figure layout has ch
```

Read csv data file and storing it in df

In [2]:
```
df = pd.read_csv('melb_data.csv')
```

- Let's check the shape of our datasets (number of rows, number of columns/features)

In [3]:
```
df.shape
```

Out[3]:
```
(13580, 21)
```

Understanding the data

In [4]:
```
#returning random 5 rows to get to know the data
df
```

Out[4]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distar
0	Abbotsford	85 Turner St	2	h	1480000	S	Biggin	03/12/16	
1	Abbotsford	25 Bloomburg St	2	h	1035000	S	Biggin	04/02/16	
2	Abbotsford	5 Charles St	3	h	1465000	SP	Biggin	04/03/17	

3	Abbotsford	40 Federation La	3	h	850000	PI	Biggin	04/03/17	
4	Abbotsford	55a Park St	4	h	1600000	VB	Nelson	04/06/16	
...	...	...	...	...	...	...	...	...	
13575	Wheelers Hill	12 Strada Cr	4	h	1245000	S	Barry	26/08/17	1
13576	Williamstown	77 Merrett Dr	3	h	1031000	SP	Williams	26/08/17	
13577	Williamstown	83 Power St	3	h	1170000	S	Raine	26/08/17	
13578	Williamstown	96 Verdon St	4	h	2500000	PI	Sweeney	26/08/17	
13579	Yarraville	6 Agnes St	4	h	1285000	SP	Village	26/08/17	

13580 rows × 21 columns

Understanding each column

1. Rooms: Number of rooms
2. Price: Price in dollars
3. Method: S - property sold; SP - property sold prior; PI - property passed in; VB - vendor bid; SA - sold after auction.
4. Type: h - house,cottage,villa, semi,terrace; u - unit, duplex; t - townhouse; dev site - development site.
5. SellerG: Real Estate Agent
6. Date: Date sold
7. Distance: Distance from CBD
8. Regionname: General Region (West, North West, North, North east …etc)
9. Propertycount: Number of properties that exist in the suburb.
10. Bedroom2 : Scraped # of Bedrooms (from different source)
11. Bathroom: Number of Bathrooms
12. Car: Number of carspots
13. Landsize: Land Size
14. BuildingArea: Building Size
15. CouncilArea: Governing council for the area

EDA - Exploratory Data Analysis

EDA - Exploratory Data Analysis

- cheecking 1st 5 rows of df

In [5]:
```
df.head()
```

Out[5]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Po
0	Abbotsford	85 Turner St	2	h	1480000	S	Biggin	03/12/16	2.5	
1	Abbotsford	25 Bloomburg St	2	h	1035000	S	Biggin	04/02/16	2.5	
2	Abbotsford	5 Charles St	3	h	1465000	SP	Biggin	04/03/17	2.5	
3	Abbotsford	40 Federation La	3	h	850000	PI	Biggin	04/03/17	2.5	
4	Abbotsford	55a Park St	4	h	1600000	VB	Nelson	04/06/16	2.5	

5 rows × 21 columns

- lets check the last 5 rows of df

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

Out[6]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distanc
13575	Wheelers Hill	12 Strada Cr	4	h	1245000	S	Barry	26/08/17	16.
13576	Williamstown	77 Merrett Dr	3	h	1031000	SP	Williams	26/08/17	6.
13577	Williamstown	83 Power St	3	h	1170000	S	Raine	26/08/17	6.
13578	Williamstown	96 Verdon St	4	h	2500000	PI	Sweeney	26/08/17	6.
13579	Yarraville	6 Agnes St	4	h	1285000	SP	Village	26/08/17	6.

5 rows × 21 columns

- Let's check the random ten number of data samples, Every time it will print the random five sample of records from original datasets. So we can easly understand the behaviour and what types of data type stored in particular features.

In [7]:
```
df.sample(10)
```

Out[7]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Dist
3798	Malvern	18 Soudan St	5	h	4240000	S	Marshall	19/11/16	
1522	Burwood	2 Bennett St	3	h	1342000	S	McGrath	10/12/16	
13046	Reservoir	25 Ramleh Rd	4	h	920000	S	Ray	19/08/17	
9950	Maribyrnong	29 Monash St	2	h	920000	VB	Biggin	24/06/17	
12538	Keilor East	14 Paul Av	3	h	1076500	S	Nelson	09/09/17	
11664	Epping	6 Shields St	3	h	545000	S	hockingstuart	22/07/17	
2840	Glen Iris	4/15 Rix St	2	u	720000	VB	Jellis	25/02/17	
8605	Elwood	23 Byron St	3	h	1520000	S	Chisholm	20/05/17	
2417	Essendon	1/139 Roberts St	2	u	667000	S	Nelson	17/09/16	
9284	Preston	145 Murray Rd	3	h	881000	S	Love	03/06/17	

10 rows × 21 columns

- lets check column wise information of our dataset

In [8]:
```
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  int64  
 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  int64  
 10  Bedroom2       13580 non-null  int64  
 11  Bathroom       13580 non-null  int64  
 12  Car            13518 non-null  float64
 13  Landsize       13580 non-null  int64  
 14  BuildingArea   7130 non-null   float64
```

From the above output we can see that :

1. There are 13580 unique rows
2. 21 column/features in the dataset i.e.[0 to 20]
3. Datatypes are of 3 types : object, float, int64
4. There 4 column which are having null values in it.
    - column with null values [BuildingArea , YearBuilt , CouncilArea , Car]
5. Memory usage is 2.2+ MB which is negligible and easily handled by machine.

Handling Null values

- lets check the number of null in each column

In [9]:
```
df.isnull().sum().sort_values(ascending = False)
```

Out[9]:
```
BuildingArea     6450
YearBuilt        5375
CouncilArea      1369
Car                62
Suburb              0
Bathroom            0
Regionname          0
Longtitude          0
Lattitude           0
Landsize            0
Bedroom2            0
```

```
Address          0
Postcode         0
Distance         0
Date             0
SellerG          0
Method           0
Price            0
Type             0
Rooms            0
Propertycount    0
dtype: int64
```

- Here, We convert the number of missing values into percentages. So, we can easly understand to how many percentage of missing values available.

In [10]:
```
# Gives the % of NULL values from the total NULL values down the co
((df.isnull().sum()*100)/df.isnull().count()).sort_values(ascending
```

Out[10]:
```
BuildingArea    47.496318
YearBuilt       39.580265
CouncilArea     10.081001
Car              0.456554
Suburb           0.000000
dtype: float64
```

- Here we can see [CouncilArea , Car] are the two columns which have null values percentage below or around 10 % so we can fill those values using statistical methods. Like mean, mode, median
- Here we can see [BuildingArea, YearBuilt] are the two colums where the NULL value percentage is very high which means close to half of the data is null. So, filling this makes no sense and it will also divert the model AND it will make wrong feeding to the model.
- So, we will drop thosee 2 columns with high percentage of NULL values.

In [11]:
```
# dropping  [ BuildingArea, YearBuilt ] columns
df.drop(columns = ['BuildingArea','YearBuilt',],inplace = True)
```

filling car column with the median of the car values

- Here mode = median != mean
- mean is prone to outliers so we are filling medain

In [12]:
```
#Filling car null with median
df['Car'].fillna(df['Car'].median(),inplace = True)
```

In [13]:
```
# Filling CouncilArea's NULL with its mode (most frequent value)
df['CouncilArea'].fillna(df['CouncilArea'].mode()[0],inplace = True
```

In [14]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13580 entries, 0 to 13579
Data columns (total 19 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Suburb         13580 non-null  object
 1   Address        13580 non-null  object
 2   Rooms          13580 non-null  int64
```

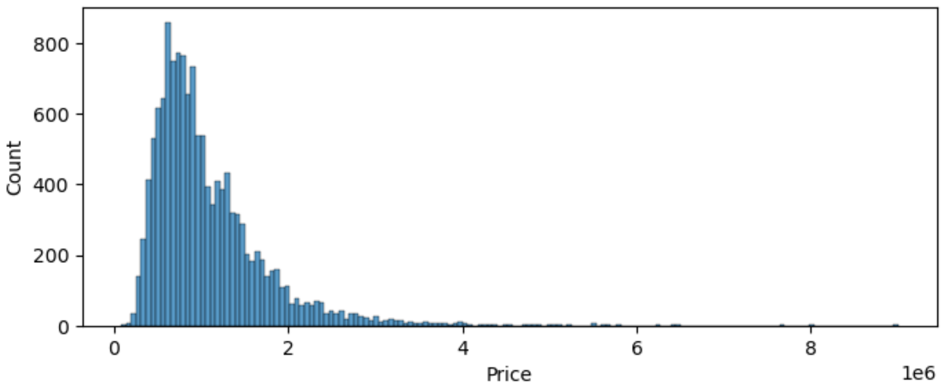
```
 3   Type          13580 non-null  object
 4   Price         13580 non-null  int64
 5   Method        13580 non-null  object
 6   SellerG       13580 non-null  object
 7   Date          13580 non-null  object
 8   Distance      13580 non-null  float64
 9   Postcode      13580 non-null  int64
 10  Bedroom2      13580 non-null  int64
 11  Bathroom      13580 non-null  int64
 12  Car           13580 non-null  float64
 13  Landsize      13580 non-null  int64
 14  CouncilArea   13580 non-null  object
 15  Lattitude     13580 non-null  float64
 16  Longtitude    13580 non-null  float64
 17  Regionname    13580 non-null  object
 18  Propertycount 13580 non-null  int64
dtypes: float64(4), int64(7), object(8)
memory usage: 2.0+ MB
```

Null values has been taken care of!!!!

Checking the distribution of target data ('Price')

In [15]:
```
#plotting a distribution graph of a 'Price'
plt.figure(figsize = (8,3))
sns.histplot(df['Price'])
```

Out[15]:
```
<Axes: xlabel='Price', ylabel='Count'>
```



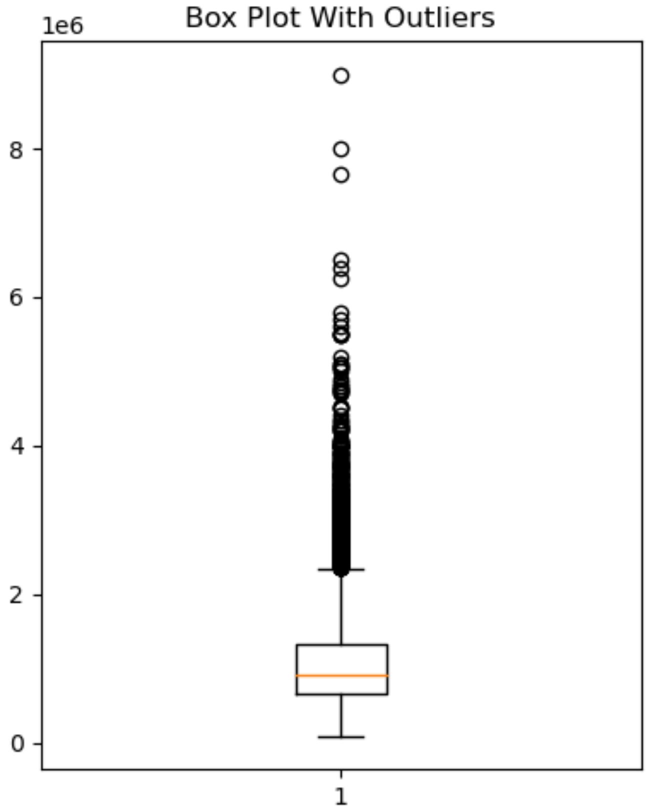
- here we clearly can see the data is not normally distributed
- Bell curve is clearly skewed towards right hand side
- **it means there are outliers in upper range i.e (4th quartile)**
- We have to take care of this

Handling Ouliers

- taking our target data price and will handle outliers on the basis of 'Price'
- There are sevral methods of Outlier Handling Like:
  1. Z-Score method
  2. IQR method (Interquartile range)
  3. Percentile method

Making suitable plot to identify outliers in price

In [16]:
```
plt.figure(figsize=(4,5))
plt.boxplot(df['Price'])
plt.title('Box Plot With Outliers')
plt.tight_layout()
```



- So, after ploting the boxplot for the Price column, we can see that there are some outliers present in the price column and from this we can assume that, there are outliers beyond value ~~~ 2.2

1. Z-Score Method

- so to handle the outlier and get the exact value level of lower and upper limit of the price we will use Z-scoe technique (method)
- We will take values upto 3 standard deviation from mean on either side to get upper and lower limit
  - For lower_limit add 3 std.dev. in the mean
  - For lower_limit less 3 std.dev. from mean

In [17]:
```
print('Price mean =',round(df['Price'].mean(),2))
print('Standard dev =',round(df['Price'].std(),2),'\n')

upper_limit = df['Price'].mean() + 3 * df['Price'].std()
lower_limit = df['Price'].mean() - 3 * df['Price'].std()


print(f'upper limit: {upper_limit}')
print(f'lower limit: {lower_limit}\n')

#Lets Find outliers based on upper and lower limits
print('total number of outliers =', len(df[(df['Price'] > upper_lim

# dropping the assigned ouliers
df.drop(index=df[(df['Price'] > upper_limit) | (df['Price'] < lower

print(f'Data rows without outlier = {len(df)}\n')
```

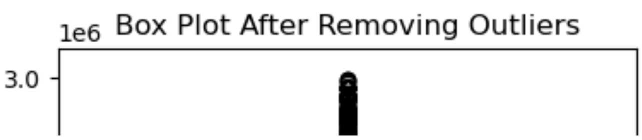
```
Price mean = 1075684.08
Standard dev = 639310.72

upper limit: 2993616.252343139
lower limit: -842248.0934329773

total number of outliers = 232
Data rows without outlier = 13348
```

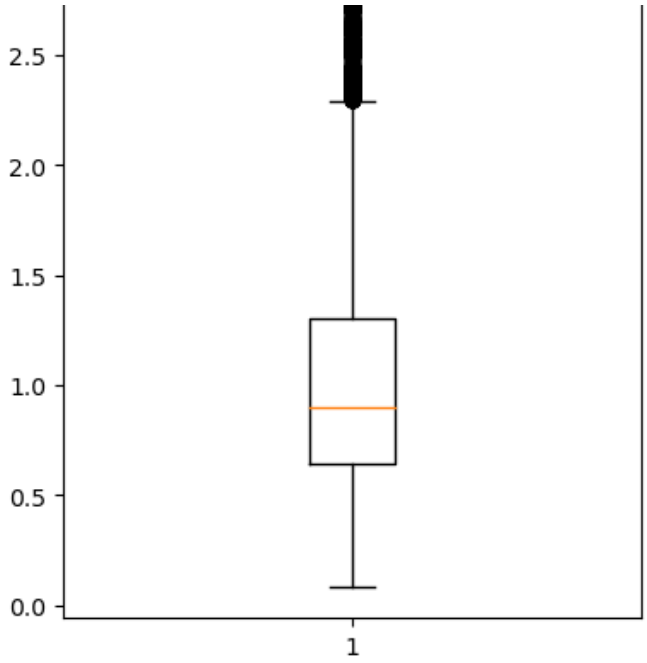
2. Inter Quartile Range Method

In [18]:
```
# #calculate q1 and q2
# q1 = df['Price'].quantile(0.25)
# q3 = df['Price'].quantile(0.75)

# #calculate iter quartile range
# iqr = q3-q1

# #calculating upper and lower limit
# upper_limit = ul = q3 + (1.5 * iqr)
# lower_limit = ll = q1 - (1.5 * iqr)


# print(f'Upper Limit : {ul}\nLower Limit : {ll}')

# #creating the index and dropping those rows which are outlier
# df.drop(index = df[df['Price']>= ul].index, inplace = True )
# df.drop(index = df[df['Price']<= ll].index, inplace = True )

# #printing the length of new df
# print('Rows count after Dropping Outliers :',len(df))
```

- **Plotting the boxplot again to see the inprovement ouliers**

In [19]:
```
plt.figure(figsize = (4,5))
plt.boxplot(df['Price'])
plt.title('Box Plot After Removing Outliers')
plt.tight_layout()
```



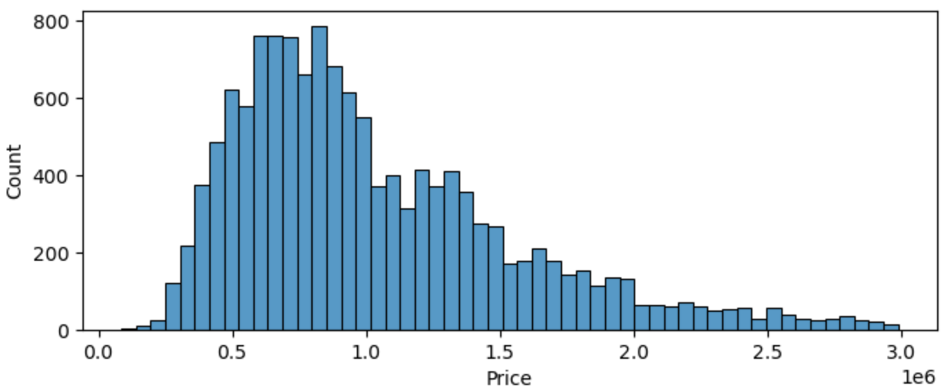


Distribution after oulier handling

- its somewhat normal now so we can proceed

In [20]:
```
plt.figure(figsize=(8,3))
sns.histplot(df['Price'])
```

Out[20]: <Axes: xlabel='Price', ylabel='Count'>



Outliers has been taken care of

Make categoriacal and numeric features (separate)

In [21]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 13348 entries, 0 to 13579
Data columns (total 19 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Suburb         13348 non-null  object 
 1   Address        13348 non-null  object 
 2   Rooms          13348 non-null  int64  
 3   Type           13348 non-null  object 
 4   Price          13348 non-null  int64  
 5   Method         13348 non-null  object 
 6   SellerG        13348 non-null  object 
 7   Date           13348 non-null  object 
 8   Distance       13348 non-null  float64
 9   Postcode       13348 non-null  int64  
 10  Bedroom2       13348 non-null  int64  
 11  Bathroom       13348 non-null  int64  
 12  Car            13348 non-null  float64
 13  Landsize       13348 non-null  int64  
 14  CouncilArea    13348 non-null  object 
 15  Lattitude      13348 non-null  float64
 16  Longtitude     13348 non-null  float64
 17  Regionname     13348 non-null  object 
 18  Propertycount  13348 non-null  int64  
dtypes: float64(4), int64(7), object(8)
memory usage: 2.0+ MB
```

In [22]:
```
categorical_features = df.select_dtypes('object')
numeric_features = df.select_dtypes(include = ('int64','float64'))
```

- Check the unique features in each of the categorical columns

In [23]:
```
for col in categorical_features:
    print(f'Unique values in {col} = {len(df[col].unique())}')
```

```
Unique values in Suburb = 314
```

```
Unique values in Address = 13151
Unique values in Type = 3
Unique values in Method = 5
Unique values in SellerG = 267
Unique values in Date = 58
Unique values in CouncilArea = 33
Unique values in Regionname = 8
```

- Each unique feature will become its own feature in the dataset, resulting in a high-dimensional feature space.
- This can make the model more complex and computationally intensive, especially if the address feature has a large number of unique values.
- to higher number of unique values(close to 99%)in 'Address' column it will heavly impact the complexity of the overall model.
- so we have to delete such columns. (like Address, Suburb, SellerG)

In [24]:
```
categorical_features.drop(columns = ['Suburb','Address','SellerG','
```

- Checking the columunar uniqueness again:

In [25]:
```
for col in categorical_features:
    print(f'Unique values in {col} = {len(df[col].unique())}')
```

```
Unique values in Type = 3
Unique values in Method = 5
Unique values in CouncilArea = 33
Unique values in Regionname = 8
```

Feature Modification

- **Here, We need to convert categorical values to numerical values**
- For that we use pandas inbuilt **.get_dummies** fucntion.
- **.get_dummies** do kind of one hot encoding which makes a new column for each unique feature and assign binnary to it

In [26]:
```
#take a look at categorical features
categorical_features
```

	Type	Method	CouncilArea	Regionname
0	h	S	Yarra	Northern Metropolitan
1	h	S	Yarra	Northern Metropolitan
2	h	SP	Yarra	Northern Metropolitan
3	h	PI	Yarra	Northern Metropolitan
4	h	VB	Yarra	Northern Metropolitan
...	...	...	...	...
13575	h	S	Moreland	South-Eastern Metropolitan
13576	h	SP	Moreland	Western Metropolitan
13577	h	S	Moreland	Western Metropolitan
13578	h	PI	Moreland	Western Metropolitan
13579	h	SP	Moreland	Western Metropolitan

13348 rows × 4 columns

- Encoding the actual data by using get dummies which is pandas default encoder
- creating new encoded data columns attaching those new columns to df and dropping original once

In [27]:
```
categorical_features = categorical_features.join(pd.get_dummies(cat
```

In [28]:
```
categorical_features.shape
```

Out[28]: (13348, 49)

In [29]:
```
numeric_features.shape
```

Out[29]: (13348, 11)

In [30]:
```
categorical_features
```

Out[30]:

	Type_h	Type_t	Type_u	Method_PI	Method_S	Method_SA	Method_SP	Method_VB
0	True	False	False	False	True	False	False	False
1	True	False	False	False	True	False	False	False
2	True	False	False	False	False	False	True	False
3	True	False	False	True	False	False	False	False

4	True	False	False	False	False	False	False	True
...	...	...	...	...	...	...	...	...
13575	True	False	False	False	True	False	False	False
13576	True	False	False	False	False	False	True	False
13577	True	False	False	False	True	False	False	False
13578	True	False	False	True	False	False	False	False

In [31]:
```
numeric_features
```

0	2	1480000	2.5	3067	2	1	1.0	202	-37.799
1	2	1035000	2.5	3067	2	1	0.0	156	-37.807
2	3	1465000	2.5	3067	3	2	0.0	134	-37.809
3	3	850000	2.5	3067	3	2	1.0	94	-37.796
4	4	1600000	2.5	3067	3	1	2.0	120	-37.807
...	...	...	...	...	...	...	...	...	
13575	4	1245000	16.7	3150	4	2	2.0	652	-37.905
13576	3	1031000	6.8	3016	3	2	2.0	333	-37.859
13577	3	1170000	6.8	3016	3	2	4.0	436	-37.852
13578	4	2500000	6.8	3016	4	1	5.0	866	-37.859
13579	4	1285000	6.3	3013	4	1	1.0	362	-37.811

13348 rows × 11 columns

Joining categorical_featues and numeric_features and storing them in one data (dataframe)

In [32]:
```
data = categorical_features.join(numeric_features)
```

Splitting data into training and target variables

- creating a list which has heading of each of the columns

In [33]:
```
features = list(data)
```

In [34]:
```
features.remove('Price')
```

- 'Price' is our target variable so, removing it from the list of features so that list of features can go to x whcih is training features

In [35]:
```
training_features = x = data[features]
```

In [36]:
```
target_features = y = data['Price']
```

Splitting data into Train Test Split

In [37]:
```
# importing train test split
from sklearn.model_selection import train_test_split
```

Splites the main data

- split data into training and validation data, for both features and target.
- The split is based on a random number generator.
- Supplying a numeric value to the random_state argument guarantees we get the same split even run this script.
- split test size will be of test = 25%

In [38]:
```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(x, y, test_size = 0
```

In [39]:
```
print("Total size: ", x.shape)
print("Train size: ", Xtrain.shape, Ytrain.shape)
print("Test size: ", Xtest.shape, Ytest.shape)
```

```
Total size:  (13348, 59)
Train size:  (11345, 59) (11345,)
Test size:  (2003, 59) (2003,)
```

Model Building

- As per the guidline of the project we will use SVR
- Although SVR is a classification based regressor

In [40]:
```
#importing the SVR
from sklearn.svm import SVR
```

In [41]:
```
# SVR model is storing in model variable
model = SVR()
```

In [42]:
```
# it will make sure each columns will have names assigned
model.feature_names_in_ = features
```

In [43]:
```
# Fit/Feed the train data to the model
model.fit(Xtrain,Ytrain)
```

Out[43]:
```
▾ SVR
SVR()
```

- To generates predictions for the test data based on the trained model, which are stored in the Ypred variable.

In [44]:
```
Ypred = model.predict(Xtest)
```

Accuracy testing

- this is a prediction algorithm so confusion matrix can't be used.
- so we will be using mean absolute error and R^2 error to evalute our model.

In [45]:
```
# Importing necessory library for testing
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
```

In [46]:
```
# mean squared error
mae = mean_absolute_error(Ytest,Ypred)
print(f'mean absolute error = {mae}\n')
```

```python
# R^2 score
r2 = r2_score(Ytest, Ypred)
print("R^2 Score:", r2)
```

```
mean absolute error = 411047.30953821767

R^2 Score: -0.08673428124577898
```

SVR is failing in this type of algorithm so we will use another regressor algorithms to predict the house prices

1. Decision Tree Regressor

In [47]:
```python
from sklearn.tree import DecisionTreeRegressor
```

In [48]:
```python
# #to get the best random state for model
# best_score = float('inf')  # Initialize with a high value
# best_random_state = None


# for random_state in range(100):  # Try different random states
#     model = DecisionTreeRegressor(random_state=random_state)
#     model.fit(Xtrain, Ytrain)
#     Ypred = model.predict(Xtest)
#     score = mean_absolute_error(Ytest, Ypred)  # Evaluate using m
#     if score < best_score:
#         best_score = score
#         best_random_state = random_state

# print("Best Random State:", best_random_state)
# print("Best Mean Squared Error:", best_score)
```

In [49]:
```python
dtm = DecisionTreeRegressor()
```

In [50]:
```python
dtm.fit(Xtrain, Ytrain)
```

Out[50]:
```
▾ DecisionTreeRegressor
DecisionTreeRegressor()
```

In [51]:
```python
dtm_Ypred = dtm.predict(Xtest)
```

In [52]:
```python
# mean squared error
mae = mean_absolute_error(Ytest,dtm_Ypred)
print(f'mean absolute error = {mae}\n')


# R^2 score
r2 = r2_score(Ytest, dtm_Ypred)
print('Accuracy = R^2 = ',round(r2*100,2),'%')
```

```
mean absolute error = 205358.96255616576

Accuracy = R^2 =  67.83 %
```

2. Linear Regression Model

In [53]:
```
from sklearn.linear_model import LinearRegression
```

In [54]:
```
lrm = LinearRegression()
```

In [55]:
```
lrm.fit(Xtrain,Ytrain)
```

Out[55]:
```
▾ LinearRegression
LinearRegression()
```

In [56]:
```
lrm_Ypred = lrm.predict(Xtest)
```

In [57]:
```
# mean squared error
mae = mean_absolute_error(Ytest,lrm_Ypred)
print(f'mean absolute error = {mae}\n')


# R^2 score
r2 = r2_score(Ytest, lrm_Ypred)
print('Accuracy = R^2 = ',round(r2*100,2),'%')
```

```
mean absolute error = 223098.24140755372

Accuracy = R^2 =  68.11 %
```

3. Rigid Regression

In [58]:
```
from sklearn.linear_model import Ridge
```

In [59]:
```
rr = Ridge()
```

In [60]:
```
rr.fit(Xtrain, Ytrain)
```

Out[60]:
```
▾ Ridge
Ridge()
```

In [61]:
```
rr_Ypred = rr.predict(Xtest)
```

In [62]:
```
# mean squared error
mae = mean_absolute_error(Ytest,rr_Ypred)
print(f'mean absolute error = {mae}\n')


# R^2 score
r2 = r2_score(Ytest,rr_Ypred)
print('Accuracy = R^2 = ',round(r2*100,2),'%')
```

```
mean absolute error = 222983.63655218072

Accuracy = R^2 =  68.13 %
```

- **From above observation we can see that all other algoriths are working fine and showing significat accuracy which is close to 70%**
- **here we can say SVR is not a best fit for this data prediction**

Hist - Gradient Boosting Regressor

- **Hist - Gradient Boosting Regressor will be the best for this problem lets check accuracy by using this.**

In [63]:
```
from sklearn.ensemble import HistGradientBoostingRegressor
```

In [64]:
```
hgbr = HistGradientBoostingRegressor()
```

In [65]:
```
hgbr.fit(Xtrain,Ytrain)
```

Out[65]:
```
▾ HistGradientBoostingRegressor
HistGradientBoostingRegressor()
```

In [66]:
```
hgbr_Ypred = hgbr.predict(Xtest)
```

In [67]:
```
# mean squared error
mae = mean_absolute_error(Ytest,hgbr_Ypred)
print(f'mean absolute error = {mae}\n')


# R^2 score
r2 = r2_score(Ytest,hgbr_Ypred)
print('Accuracy = R^2 = ',round(r2*100,2),'%')
```

```
mean absolute error = 147162.54183081223

Accuracy = R^2 =  84.06 %
```

Thanks