**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** warnings

warnings**.**simplefilter(action**=**'ignore', category**=**FutureWarning)

pd**.**options**.**mode**.**chained\_assignment **=** **None**

In [2]:

housing **=** pd**.**read\_csv('C:\DataScienceForPublicGood\data\_science\_101\housing.csv')

In [3]:

housing**.**head() *# head() method returns top 5 rows from dataset, that way we can clearly observe dataset*

Out[3]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** | **ocean\_proximity** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 | 8.3252 | 452600.0 | NEAR BAY |
| **1** | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 358500.0 | NEAR BAY |
| **2** | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 | 7.2574 | 352100.0 | NEAR BAY |
| **3** | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 | 5.6431 | 341300.0 | NEAR BAY |
| **4** | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 | 3.8462 | 342200.0 | NEAR BAY |

In [4]:

housing**.**info() *# We use info() method For accessing information about dataset such as, properties of data, types of data and number of data.*

RangeIndex: 20640 entries, 0 to 20639

Data columns (total 10 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 longitude 20640 non-null float64

1 latitude 20640 non-null float64

2 housing\_median\_age 20640 non-null float64

3 total\_rooms 20640 non-null float64

4 total\_bedrooms 20433 non-null float64

5 population 20640 non-null float64

6 households 20640 non-null float64

7 median\_income 20640 non-null float64

8 median\_house\_value 20640 non-null float64

9 ocean\_proximity 20640 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

In [5]:

*# We plot data as histograms for visualization*

housing**.**hist(bins**=**50,figsize**=**(20,15))

plt**.**show()

In [6]:

housing**.**isnull()**.**sum() *# We must control that whether our data set contains any null values or not.*

Out[6]:

longitude 0

latitude 0

housing\_median\_age 0

total\_rooms 0

total\_bedrooms 207

population 0

households 0

median\_income 0

median\_house\_value 0

ocean\_proximity 0

dtype: int64

**Analyzing Null values**

We detected some null values in the total\_bedrooms column, what can we do about it? We can basically replace numeric values with null values using the fillna() method. But what if we want to try another method? In this project, I would like to present you an alternative way of handling null datas.  
When we observe the head of the dataset, we see some correlation between the total\_bedrooms column and the household column. If we can prove our statement we can replace missing values in total\_bedrooms column with the corresponded values in household column. Since we only observed a small part of our dataset, we cannot be completely sure of our statement.

**To better interpret our statement, let's vizualize the columns total\_bedrooms and houses columns.**

In [7]:

plt**.**subplots(figsize**=**(15, 7))

plt**.**title('Histogram Plot: Total Bedrooms')

total\_bedrooms **=** housing['total\_bedrooms']

plt**.**hist( total\_bedrooms, bins**=**500, alpha**=**0.8,

histtype**=**'bar', color**=**'steelblue',

edgecolor**=**'green')

plt**.**show()

In [8]:

plt**.**subplots(figsize**=**(15, 7))

plt**.**title('Histogram Plot: Households')

households **=** housing['households']

plt**.**hist( households, bins**=**500, alpha**=**0.8,

histtype**=**'bar', color**=**'blue',

edgecolor**=**'green')

plt**.**show()

In [9]:

plt**.**figure(figsize**=**(14, 6))

plt**.**rcParams['axes.grid'] **=** **False** *# For suppressing the depreciation error*

plt**.**title('Scatter Plot: Total Bedrooms and Households ')

N **=** households**.**size

colors **=** np**.**random**.**rand(N)

area **=** np**.**pi **\*** (20 **\*** np**.**random**.**rand(N))**\*\***2

plt**.**xlabel('Total Bedrooms')

plt**.**ylabel('Households')

plt**.**scatter(total\_bedrooms, households, s**=**area, c**=**colors, alpha**=**0.5, cmap**=**'Spectral')

plt**.**colorbar()

plt**.**show()

**Applying statistics for observing relationship between total bedrooms and households**

As we observed from scatter plot there is a positive trend between total bedrooms and households. So we are supposing a null hypothesis that total bedrooms and households are same attributes. In this case, we need to try to reject our null hypothesis and come up with an alternative hypothesis that assumes total bedrooms and households are not same attributes.

**We can represent this relationship like that;**

**tb** : Total bedrooms  
**hh** : House holds  
**Ho** : tb = hh  
**Ha** : tb != hh

**What are the steps for testing Null Hypothesis?**

* Specify a significance level α
* Calculate variance for total bedrooms and households
* Calculate standart deviation for total bedrooms and households
* Evaluate p-score using t-test
* Apply p-test, if p-score is greater than significance level accept null hypothesis, if not reject null hypothesis

**For testing our null hypothesis we select our significance level as 0.05**

In [10]:

*# Variance total\_bedrooms and households*

var\_total\_bedrooms **=** (np**.**square(total\_bedrooms **-** total\_bedrooms**.**mean()))**.**sum() **/** (housing**.**shape[0] **-** 1)

var\_households **=** (np**.**square(households **-** households**.**mean()))**.**sum() **/** (housing**.**shape[0] **-** 1)

print(f'Variance of; (total bedrooms: {var\_total\_bedrooms}, import scipy.stats as stats: {var\_households})')

Variance of; (total bedrooms: 175784.4754402569, import scipy.stats as stats: 146176.03990028054)

In [11]:

*# Standart deviation for total\_bedrooms and households*

S\_total\_bedrooms **=** np**.**sqrt(var\_total\_bedrooms)

S\_households **=** np**.**sqrt(var\_households)

print(f'Standart deviation of; (total bedrooms: {S\_total\_bedrooms}, households: {S\_households})')

Standart deviation of; (total bedrooms: 419.26659232552373, households: 382.32975283161073)

In [12]:

*# We can evaluate p score from t-test using scipy.stats library*

**import** scipy.stats **as** stats

tstat, pvalue **=** stats**.**ttest\_ind\_from\_stats(total\_bedrooms**.**mean(), S\_total\_bedrooms, total\_bedrooms**.**size, households**.**mean(), S\_households, households**.**size)

print(f't score: {tstat}, p score: {pvalue})')

t score: 9.705149779887096, p score: 3.0279724591955457e-22)

As a result, we can clearly see that our p value is lesser than significance level α (3.0279724591955457e-22 < 0.05).  
In that case we reject our null hypothesis. So we can say our statement is false but we should also check correlation between two columns.

**Correlation between two column.**

In [13]:

*# Covariance between total\_bedrooms and households*

cov **=** ((total\_bedrooms **-** total\_bedrooms**.**mean()) **\*** (households **-** households**.**mean()))**.**sum() **/** (housing**.**shape[0] **-** 1)

cov

Out[13]:

156246.54825893574

In [14]:

*# Correlation between total\_bedrooms and population*

corr **=** cov **/** (S\_total\_bedrooms **\*** S\_households)

corr

Out[14]:

0.974724937213194

In [15]:

*# Or in shorter way, we can use pandas correlation() method for calculating correlation between columns*

housing[['total\_bedrooms','households']]**.**corr()

Out[15]:

|  | **total\_bedrooms** | **households** |
| --- | --- | --- |
| **total\_bedrooms** | 1.000000 | 0.979728 |
| **households** | 0.979728 | 1.000000 |

So, there is a strong positive correlation exists between two columns. Eventough we rejected our null hypothesis, since there is a strong correlation exists, we can replace missing values in total bedrooms column with the corresponding values in household column.

**Handling null values**

* First we have to find indexes of the null values in total\_bedrooms column.
* Then we will get corresponding indexes in the household column.
* Finally we will replace null values with the values from household column.

In [16]:

*# Finding indexes of the null values in total\_bedrooms column and appending them in to a empty list*

null\_indexes **=** []

**for** (i, v) **in** housing["total\_bedrooms"]**.**iteritems():

**if** pd**.**isna(v):

null\_indexes**.**append(i)

print(null\_indexes)

[290, 341, 538, 563, 696, 738, 1097, 1350, 1456, 1493, 1606, 2028, 2115, 2301, 2323, 2334, 2351, 2412, 2420, 2578, 2608, 2647, 2826, 3024, 3328, 3354, 3376, 3482, 3485, 3529, 3721, 3778, 3912, 3921, 3958, 4043, 4046, 4186, 4279, 4309, 4391, 4447, 4496, 4591, 4600, 4629, 4667, 4691, 4738, 4743, 4744, 4767, 4852, 5059, 5216, 5222, 5236, 5654, 5665, 5678, 5723, 5751, 5990, 6052, 6068, 6220, 6241, 6253, 6298, 6421, 6541, 6590, 6814, 6835, 6962, 7097, 7113, 7168, 7191, 7228, 7316, 7330, 7547, 7654, 7668, 7763, 7806, 8337, 8383, 8530, 8915, 9149, 9571, 9620, 9622, 9814, 9845, 9877, 9942, 9970, 10033, 10216, 10236, 10385, 10389, 10428, 10495, 10761, 10885, 10915, 11096, 11311, 11351, 11441, 11449, 11512, 11741, 12101, 12414, 12570, 12809, 13015, 13069, 13311, 13332, 13336, 13597, 13656, 13706, 13925, 13932, 13933, 14015, 14152, 14173, 14307, 14331, 14386, 14462, 14521, 14641, 14930, 14970, 14986, 15030, 15060, 15118, 15137, 15397, 15479, 15607, 15663, 15890, 15975, 16025, 16038, 16104, 16105, 16330, 16757, 16879, 16880, 16885, 17041, 17198, 17202, 17639, 17825, 17840, 17923, 17928, 17973, 18177, 18246, 18261, 18332, 18346, 18466, 18786, 18873, 18914, 19060, 19071, 19122, 19150, 19252, 19332, 19391, 19402, 19485, 19559, 19607, 19638, 19766, 19818, 19833, 19890, 19932, 19959, 20046, 20069, 20125, 20267, 20268, 20372, 20460, 20484]

In [17]:

*# Accessing household columns values with null\_indexes from total\_bedrooms column*

values **=** []

**for** i **in** range(len(null\_indexes)):

values**.**append(housing["households"]**.**iloc[null\_indexes[i]])

print(values)

[218.0, 259.0, 1273.0, 146.0, 161.0, 557.0, 600.0, 1012.0, 540.0, 499.0, 626.0, 372.0, 1260.0, 928.0, 580.0, 224.0, 292.0, 243.0, 257.0, 669.0, 266.0, 91.0, 16.0, 584.0, 310.0, 902.0, 769.0, 623.0, 1200.0, 550.0, 737.0, 1492.0, 1280.0, 403.0, 677.0, 155.0, 528.0, 294.0, 312.0, 726.0, 422.0, 439.0, 290.0, 1179.0, 682.0, 1462.0, 439.0, 516.0, 187.0, 271.0, 475.0, 292.0, 245.0, 749.0, 231.0, 476.0, 302.0, 403.0, 771.0, 441.0, 339.0, 326.0, 530.0, 328.0, 727.0, 500.0, 497.0, 403.0, 1251.0, 474.0, 257.0, 155.0, 1012.0, 622.0, 357.0, 552.0, 221.0, 120.0, 452.0, 427.0, 1196.0, 212.0, 248.0, 629.0, 269.0, 130.0, 219.0, 270.0, 951.0, 287.0, 130.0, 1483.0, 132.0, 257.0, 112.0, 428.0, 393.0, 67.0, 72.0, 439.0, 437.0, 276.0, 423.0, 762.0, 541.0, 218.0, 522.0, 183.0, 622.0, 499.0, 351.0, 242.0, 1063.0, 996.0, 606.0, 512.0, 186.0, 1176.0, 202.0, 562.0, 281.0, 627.0, 375.0, 843.0, 611.0, 358.0, 107.0, 391.0, 532.0, 158.0, 375.0, 592.0, 201.0, 539.0, 390.0, 108.0, 499.0, 426.0, 268.0, 687.0, 246.0, 108.0, 677.0, 400.0, 499.0, 682.0, 231.0, 862.0, 801.0, 214.0, 152.0, 963.0, 89.0, 647.0, 406.0, 404.0, 328.0, 354.0, 254.0, 1537.0, 156.0, 715.0, 480.0, 757.0, 603.0, 395.0, 535.0, 1438.0, 364.0, 386.0, 686.0, 893.0, 275.0, 584.0, 194.0, 1093.0, 3589.0, 482.0, 181.0, 516.0, 738.0, 275.0, 501.0, 778.0, 397.0, 1405.0, 329.0, 106.0, 596.0, 255.0, 320.0, 83.0, 262.0, 494.0, 149.0, 293.0, 254.0, 516.0, 206.0, 359.0, 93.0, 347.0, 779.0, 762.0, 669.0, 814.0, 495.0]

In [18]:

*# Replacing null values in the total\_bedrooms column with the values from household column.*

index **=** 0

**for** (i, v) **in** housing["total\_bedrooms"]**.**iteritems():

**if** pd**.**isna(v):

housing["total\_bedrooms"]**.**iloc[i] **=** values[index]

index**+=**1

In [19]:

*# Checking if null values still exists*

housing**.**isnull()**.**sum()

Out[19]:

longitude 0

latitude 0

housing\_median\_age 0

total\_rooms 0

total\_bedrooms 0

population 0

households 0

median\_income 0

median\_house\_value 0

ocean\_proximity 0

dtype: int64

**Analyzing categorical values**

We have only 1 categorical feature in our dataset. To better analyzing it let's visualize our ocean proximity column first.

In [20]:

*# Barplot demonstration of the ocean\_proximity column*

**%matplotlib** inline

**import** seaborn **as** sns

**import** matplotlib.pyplot **as** plt

distance\_count **=** housing['ocean\_proximity']**.**value\_counts()

sns**.**set(style**=**"darkgrid")

sns**.**barplot(distance\_count**.**index, distance\_count**.**values, alpha**=**0.9)

plt**.**title('Frequency Distribution of Ocean Proximity Column')

plt**.**ylabel('Number of Occurrences', fontsize**=**12)

plt**.**xlabel('Distance to Ocean', fontsize**=**12)

plt**.**show()

In [21]:

*# Pie chart demonstration of the ocean\_proximity column*

labels **=** housing['ocean\_proximity']**.**astype('category')**.**cat**.**categories**.**tolist()

counts **=** housing['ocean\_proximity']**.**value\_counts()

sizes **=** [counts[var\_cat] **for** var\_cat **in** labels]

fig1, ax1 **=** plt**.**subplots()

ax1**.**pie(sizes, labels**=**labels, autopct**=**'%1.1f%%', shadow**=True**) *#autopct is show the % on plot*

ax1**.**axis('equal')

plt**.**show()

In [22]:

*# Number of values for each feature in ocean\_proximity columns*

print(housing['ocean\_proximity']**.**value\_counts())

<1H OCEAN 9136

INLAND 6551

NEAR OCEAN 2658

NEAR BAY 2290

ISLAND 5

Name: ocean\_proximity, dtype: int64

Since we have only 5 feature in ocean\_proximity column and these features are easy to interpret we can replace them with the desired numeric values. In this case, we will score them by their distances to the ocean.  
Our placement will look like this;

* INLAND < 1H TO OCEAN < NEAR\_BAY < NEAR\_OCEAN < ISLAND

**Handling categorical values**

We will create and dictionary which contains features and corresponding values that we hardcoded.

In [23]:

replace\_map **=** {'ocean\_proximity': {'<1H OCEAN': 2, 'INLAND': 1, 'NEAR OCEAN': 4, 'NEAR BAY': 3,

'ISLAND': 5}}

In [24]:

housing\_replace **=** housing**.**copy()

housing\_replace**.**replace(replace\_map, inplace**=True**)

housing\_replace**.**head()

Out[24]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** | **ocean\_proximity** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 | 8.3252 | 452600.0 | 3 |
| **1** | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 358500.0 | 3 |
| **2** | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 | 7.2574 | 352100.0 | 3 |
| **3** | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 | 5.6431 | 341300.0 | 3 |
| **4** | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 | 3.8462 | 342200.0 | 3 |

**Handling Outliers**

We detect outliers with using Z-score, our criteria is any data point whose Z-score falls out of 3rd standard deviation is an outlier.  
Steps for calculating outliers with using z-score;

* Calculate the Z-score using the formula (x-mean)/std.
* Define a threshold value of 3 and mark the datapoints whose absolute value of Z-score is greater than the threshold as outliers.

Once we find outliers, we have consider how to deal with them. There are several techniques for replacing outliers, but we will use the median imputation method. The reason we use the median imputation is that the median method is not affected by the presence of outliers like the mean method.

In [25]:

**def** detect\_outliers\_zscore(data):

outliers **=** []

thres **=** 3

mean **=** np**.**mean(data)

std **=** np**.**std(data)

*# print(mean, std)*

**for** i **in** data:

z\_score **=** (i**-**mean)**/**std

**if** (np**.**abs(z\_score) **>** thres):

outliers**.**append(i)

**return** outliers*# Driver code*

In [26]:

sample\_outliers **=** []

*# We got (len - 1) because we don't want to look for outliers in categorical values.*

**for** i **in** range(len(housing\_replace**.**count()) **-** 1):

sample\_outliers**.**append(detect\_outliers\_zscore(housing\_replace**.**iloc[:,i]))

In [27]:

**def** replace\_outliers(data, sample\_index):

median **=** np**.**median(data)

index **=** 0

**if** len(sample\_outliers[sample\_index]) **==** 0: *# returns if sample has no outlier*

**return**

**for** (i, v) **in** data**.**iteritems():

**if** sample\_outliers[sample\_index][index] **==** v:

data[i] **=** median

**if** index **<** len(sample\_outliers[sample\_index]) **-** 1:

index **+=**1

In [28]:

housing\_outliers **=** housing\_replace**.**copy() *# We copy our dataframe and replace outliers on this new dataframe.*

**for** i **in** range(len(sample\_outliers)):

replace\_outliers(housing\_outliers**.**iloc[:,i], i)

* We know that the mean of the dataset is affected by outliers, let's check the effect of outliers on our dataset. For this, let's take the average of the columns with and without outliers.

In [29]:

*# Mean of columns with and without outliers*

df **=** pd**.**DataFrame(housing\_replace**.**mean() , columns**=**['Mean with outliers'])

df['Mean without outliers'] **=** housing\_outliers**.**mean()

df

Out[29]:

|  | **Mean with outliers** | **Mean without outliers** |
| --- | --- | --- |
| **longitude** | -119.569704 | -119.569704 |
| **latitude** | 35.631861 | 35.631861 |
| **housing\_median\_age** | 28.639486 | 28.639486 |
| **total\_rooms** | 2635.763081 | 2432.444380 |
| **total\_bedrooms** | 537.591279 | 499.616231 |
| **population** | 1425.476744 | 1330.515068 |
| **households** | 499.539680 | 465.958285 |
| **median\_income** | 3.870671 | 3.735332 |
| **median\_house\_value** | 206855.816909 | 206855.816909 |
| **ocean\_proximity** | 2.051841 | 2.051841 |

* We know that the median of the dataset is not affected much by outliers, let's check the effect of outliers on our dataset. For this, let's take the median of the columns with and without outliers.

In [30]:

*# Median of columns with and without outliers*

df **=** pd**.**DataFrame(housing\_replace**.**median() , columns**=**['Median with outliers'])

df['Median without outliers'] **=** housing\_outliers**.**median()

df

Out[30]:

|  | **Median with outliers** | **Median without outliers** |
| --- | --- | --- |
| **longitude** | -118.4900 | -118.49000 |
| **latitude** | 34.2600 | 34.26000 |
| **housing\_median\_age** | 29.0000 | 29.00000 |
| **total\_rooms** | 2127.0000 | 2127.00000 |
| **total\_bedrooms** | 435.0000 | 435.00000 |
| **population** | 1166.0000 | 1166.00000 |
| **households** | 409.0000 | 409.00000 |
| **median\_income** | 3.5348 | 3.53475 |
| **median\_house\_value** | 179700.0000 | 179700.00000 |
| **ocean\_proximity** | 2.0000 | 2.00000 |

We could also detect outliers with using Interquartile range (IQR).  
Steps for applying IQR;

* Sort the data
* Calculate Q1 and Q3
* Find IQR (Q3 - Q1)
* Find the lower fence (Q1 - 1.5 \* IQR)
* Find the upper fence (Q3 + 1.5 \* IQR)

In [31]:

**def** detect\_outliers\_iqr(dataset):

outliers **=** []

dataset **=** sorted(dataset)

q1,q3 **=** np**.**percentile(dataset, [25,75])

iqr **=** q3 **-** q1

lower\_fence **=** q1 **-** (1.5 **\*** iqr)

upper\_fence **=** q3 **+** (1.5 **\*** iqr)

**for** elem **in** dataset:

**if** elem **<** lower\_fence **or** elem **>** upper\_fence:

outliers**.**append(elem)

**return** outliers

*# We can use box plots for visulazition of outliers*

**import** seaborn **as** sns

sns**.**boxplot(housing['median\_house\_value'])

Out[31]:

**Selecting Model**

In order to choose the right model, we must ask ourselves what is our purpose. Determining the problem and forming strategies accordingly are very important for choosing the right model. In our problem we are asked to estimate house prices in California. We can see that this is a Regression problem because our task is to approximate a continuous output variable from input variables (X) to a matching function (f).  
OK, we have identified the problem, so what do we do next? Since we define the problem we could search for different algorithms for our purpose. We will use Random Forest Reggression model in this project because results from researchs show that random forest reggession model gives the best results so far.

We will use sci-kit random forest reggressor as our model. I preffered not divide dataset into train, validation and test. Reason is, since we use random forest as our model using validation set is not mandatory. Also our dataset is small so, fine tuning hyper parameter in training set will not be big problem.

Steps for appliying sci-kit random forest reggressor;

* Rearrenge columns in the dataset for divide into attributes and label sets.
* Split dataset as test and train
* Scale the dataset (It is optional since we use random forest)
* Train model
* Calculate train and test score of the model
* Fine tune hyperparameters for acquiring better accuracy

In [32]:

*# Rearranging the order of data in dataset*

cols **=** ['longitude', 'latitude', 'housing\_median\_age','total\_rooms', 'total\_bedrooms',

'population', 'households', 'median\_income', 'ocean\_proximity', 'median\_house\_value' ]

dataset **=** housing\_outliers**.**copy()

dataset **=** dataset[cols]

dataset**.**head()

Out[32]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **ocean\_proximity** | **median\_house\_value** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 | 8.3252 | 3 | 452600.0 |
| **1** | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 3 | 358500.0 |
| **2** | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 | 7.2574 | 3 | 352100.0 |
| **3** | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 | 5.6431 | 3 | 341300.0 |
| **4** | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 | 3.8462 | 3 | 342200.0 |

In [33]:

*# Dividing dataset into attributes and label sets*

X **=** dataset**.**iloc[:, 0:9]**.**values

y **=** dataset**.**iloc[:, 9]**.**values

In [34]:

**from** sklearn.model\_selection **import** train\_test\_split

*# Split dataset as test and train*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size**=**0.2, random\_state**=**1)

In [35]:

**from** sklearn.preprocessing **import** StandardScaler

*# In our dataset each column has different scale of representation, this may cause problem in some algorithms*

*# For preventing that we scale our dataset*

sc **=** StandardScaler()

X\_train **=** sc**.**fit\_transform(X\_train)

X\_test **=** sc**.**transform(X\_test)

In [36]:

**import** time

**from** sklearn.ensemble **import** RandomForestRegressor

time\_start **=** time**.**time()

*# Training model*

regressor **=** RandomForestRegressor(n\_estimators**=**100, max\_features **=** 0.5)

regressor**.**fit(X\_train, y\_train)

y\_pred **=** regressor**.**predict(X\_test)

time\_end **=** time**.**time() *# Estimating running time of the model*

print(f'Run time : {time\_end **-** time\_start}')

Run time : 3.500152826309204

In [37]:

**from** sklearn **import** metrics

print('Training score: ', regressor**.**score(X\_train, y\_train))

print('Testing score: ', regressor**.**score(X\_test, y\_test))

print('Root Mean Squared Error:', np**.**sqrt(metrics**.**mean\_squared\_error(y\_test, y\_pred)))

Training score: 0.9750914650936254

Testing score: 0.8233739136464446

Root Mean Squared Error: 48133.02610517148