So, we now make a model for ourselves using all of the knowledge we have acquired.

Lets have a clear idea of what we should do to make a good model Steps:

- read the data
- understand the variables
- discart the unnecessary variables
- impute the missing values
- encode the categorical variables
- find the ideal leaf node number
- use crossvalidation for best mae result
- use our best model (XGBoost)
- reduce the error

We start by reading the data. We will be operating on melbourne housing data to predict the price of houses.

In [3]:
```
import pandas as pd
filepath = r"C:\Users\goura\Desktop\Data Science\melbourne_data_kaggle.csv"
data=pd.read_csv(filepath)
data.describe()
```

Out[3]:

	Rooms	Price	Distance	Postcode	Bedroom2	Bathro
count	13580.000000	1.358000e+04	13580.000000	13580.000000	13580.000000	13580.000
mean	2.937997	1.075684e+06	10.137776	3105.301915	2.914728	1.534
std	0.955748	6.393107e+05	5.868725	90.676964	0.965921	0.691
min	1.000000	8.500000e+04	0.000000	3000.000000	0.000000	0.000
25%	2.000000	6.500000e+05	6.100000	3044.000000	2.000000	1.000
50%	3.000000	9.030000e+05	9.200000	3084.000000	3.000000	1.000
75%	3.000000	1.330000e+06	13.000000	3148.000000	3.000000	2.000
max	10.000000	9.000000e+06	48.100000	3977.000000	20.000000	8.000

We can have a look at the table and get idea about relevant values.

In [4]:
```
data.shape
```

Out[4]:
```
(13580, 21)
```

We can see that we have a total of 13580 rows, 21 columns

In [5]:
```
data.columns
```

Out[5]:
```
Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',
       'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',
       'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude',
       'Longtitude', 'Regionname', 'Propertycount'],
      dtype='object')
```

We will set the prediction target to be Price

In [6]:
```
Y=data.Price
data=data.drop(['Price'], axis=1)
```

Now, we should be able to analyze which data to use for training our model. It contains a few steps:

- classifying categorical and numerical columns
- removing rest columns

In [7]:
```
catcols=[x for x in data.columns if data[x].nunique()<10 and data[x].dtype=='obj
catcols
```

Out[7]:
```
['Type', 'Method', 'Regionname']
```

The above code has tracked the categorical data columns in the DF

In [8]:
```
numcols=[x for x in data.columns if data[x].dtype in ['int64','float64']]
numcols
```

Out[8]:
```
['Rooms',
 'Distance',
 'Postcode',
 'Bedroom2',
 'Bathroom',
 'Car',
 'Landsize',
 'BuildingArea',
 'YearBuilt',
 'Lattitude',
 'Longtitude',
 'Propertycount']
```

The above code has tracked the numerical data columns in the DF

In [9]:
```
allcols=catcols+numcols
allcols
```

Out[9]:
```
['Type',
 'Method',
 'Regionname',
 'Rooms',
 'Distance',
 'Postcode',
 'Bedroom2',
 'Bathroom',
 'Car',
 'Landsize',
 'BuildingArea',
 'YearBuilt',
 'Lattitude',
 'Longtitude',
 'Propertycount']
```

So this is the set of columns we will be using to create our model.

In [10]:
```
len(allcols)
```

Out[10]: 15

We have now considered all columns we will consider for the model analysis. There can be further improvement. We will now observe any column for which predictor values may not be available.

There is a certain phenomenon called Data Leakage. This happens when the data we fit into the model for training is not available for future values.

There are not any data leakage columns here. So we just need to remove the Price column.

In [11]:
```
X=data[allcols]
X
```

Out[11]:

	Type	Method	Regionname	Rooms	Distance	Postcode	Bedroom2	Bathroom
0	h	S	Northern Metropolitan	2	2.5	3067	2	1
1	h	S	Northern Metropolitan	2	2.5	3067	2	1
2	h	SP	Northern Metropolitan	3	2.5	3067	3	2
3	h	PI	Northern Metropolitan	3	2.5	3067	3	2
4	h	VB	Northern Metropolitan	4	2.5	3067	3	1
...	...	...	...	...	...	...	...	...
13575	h	S	South-Eastern Metropolitan	4	16.7	3150	4	2
13576	h	SP	Western Metropolitan	3	6.8	3016	3	2
13577	h	S	Western Metropolitan	3	6.8	3016	3	2
13578	h	PI	Western Metropolitan	4	6.8	3016	4	1
13579	h	SP	Western Metropolitan	4	6.3	3013	4	1

13580 rows × 15 columns

Now we will split the data into training and testing data

In [12]:
```
from sklearn.model_selection import train_test_split
X_train,X_valid,Y_train,Y_valid = train_test_split(X,Y,random_state=0)
```

Now that we have our training and validation data, lets take care of the missing values in our training data

Lets create an imputer object

In [13]:
```
from sklearn.impute import SimpleImputer
imputer1 = SimpleImputer(strategy='mean')
imputer2 = SimpleImputer(strategy='most_frequent')
```

We have created two imputer objects. One for imputing numeric columns. Another for imputing categorical columns.

The reasons being:

- imputer1 imputes numeric data, so the mean/median value makes most sense.

- imputer2 imputes categorical data, so mean/median doesnt make much sence, so its safer to go with most frequent elements

In [14]:
```
X_tr = pd.DataFrame(imputer1.fit_transform(X_train[numcols]))
X_val = pd.DataFrame(imputer1.transform(X_valid[numcols]))
X_tr.columns=X_train[numcols].columns
X_val.columns=X_valid[numcols].columns
```

The numeric columns have been imputed. The column names restored as well.

In [15]:
```
X_tr2 = pd.DataFrame(imputer2.fit_transform(X_train[catcols]))
X_tr2.columns=X_train[catcols].columns
X_val2 = pd.DataFrame(imputer2.transform(X_valid[catcols]))
X_val2.columns=X_valid[catcols].columns
```

The categorical columns have been imputed. The column names restored as well.

In [16]:
```
X_train = pd.concat([X_tr, X_tr2], axis=1)
X_valid = pd.concat([X_val, X_val2], axis=1)
```

We now concatenate the two imputed datasets to form the overall training data. This includes both the Categorical as well as Numeric variables.

Now our dataset just needs encoding and it will be ready for fitting. Lets have a look at out current X_train

In [17]:
```
X_train
```

Out[17]:

	Rooms	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	BuildingArea
0	3.0	9.2	3104.0	3.0	2.0	2.0	368.0	177.000000
1	2.0	10.5	3081.0	2.0	1.0	2.0	586.0	80.000000
2	2.0	11.2	3145.0	2.0	1.0	1.0	348.0	154.655601
3	3.0	19.6	3076.0	3.0	1.0	1.0	521.0	154.655601
4	4.0	11.4	3163.0	3.0	2.0	2.0	687.0	237.000000
...	...	...	...	...	...	...	...	...
10180	3.0	5.2	3056.0	3.0	1.0	2.0	212.0	154.655601
10181	3.0	10.5	3081.0	3.0	1.0	1.0	748.0	101.000000
10182	4.0	6.7	3058.0	4.0	2.0	2.0	441.0	255.000000
10183	3.0	12.0	3073.0	3.0	1.0	1.0	606.0	154.655601
10184	4.0	6.4	3011.0	4.0	2.0	1.0	319.0	130.000000

10185 rows × 15 columns

We have now succesfully cleaned and arranged the data for analysing. But theres still a little to go. We can encome the categorical columns and convert them to numbers. This makes it easier for the model to analyse and deliver better predictions.

In [18]:
```
from sklearn.preprocessing import OrdinalEncoder
encoder=OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)
X_tr = X_train.copy()
X_val = X_valid.copy()
X_tr[catcols]=encoder.fit_transform(X_train[catcols])
X_val[catcols] = encoder.transform(X_valid[catcols])
X_tr
```

Out[18]:

	Rooms	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	BuildingArea
0	3.0	9.2	3104.0	3.0	2.0	2.0	368.0	177.000000
1	2.0	10.5	3081.0	2.0	1.0	2.0	586.0	80.000000
2	2.0	11.2	3145.0	2.0	1.0	1.0	348.0	154.655601
3	3.0	19.6	3076.0	3.0	1.0	1.0	521.0	154.655601
4	4.0	11.4	3163.0	3.0	2.0	2.0	687.0	237.000000
...	...	...	...	...	...	...	...	...
10180	3.0	5.2	3056.0	3.0	1.0	2.0	212.0	154.655601
10181	3.0	10.5	3081.0	3.0	1.0	1.0	748.0	101.000000
10182	4.0	6.7	3058.0	4.0	2.0	2.0	441.0	255.000000
10183	3.0	12.0	3073.0	3.0	1.0	1.0	606.0	154.655601
10184	4.0	6.4	3011.0	4.0	2.0	1.0	319.0	130.000000

10185 rows × 15 columns

We have now successfully encoded the data. It is now ready to be used for model fitting. From our study, the best model yet we have found is XGBoost. But we will try all three models.

Lets get the mae detecting function as well as an error detecting function

In [19]:
```
from sklearn.metrics import mean_absolute_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
```

In [81]:
```
def getmaecr(model,X_tr,Y_train):
    scores = -1 * cross_val_score(model,X_tr,Y_train,cv=200,scoring='neg_mean_ab
    return scores
```

We will also create an err function to keep track of the accuracy of predictions make by our model

In [21]:
```
avg_price = Y.mean()
def err(mae):
    return (mae/avg_price)*100
```

Lets use Desicion Tree Regressor first

In [20]:
```
my_model=DecisionTreeRegressor(random_state=0)
mae=getmaecr(my_model,X_tr,Y_train)
mae.min()
```

Out[20]:
```
225049.78841433482
```

So the MAE for DTR is 225000 approx. Lets find out the error

In [21]:
```
mae=mae.min()
error=err(mae)
error
```

Out[21]: 20.9215505474749

Our DTR model shows an accuracy of about 79% as of now. Lets try the RFR model to see if we get better results.

Now we move on to using the RFR model

In [22]:
```
from sklearn.ensemble import RandomForestRegressor
my_model=RandomForestRegressor(random_state=0)
mae=getmaecr(my_model,X_tr,Y_train)
mae=mae.min()
mae
```

Out[22]: 164718.5279056502

So the MAE for our RFR model is around 164700. Thats a significant drop from the DTR model. Lets have a look at the error.

In [23]:
```
error=err(mae)
error
```

Out[23]: 15.312909343149633

So the error is a little higher than 15%, which means our errors have decreased. by around 30%.

Upto what we did till now, we find that our accuracy is going opto 85%.

We will next step into complex usage of XGBoost and make leaf adjustments to try to further improve our model.

We need to import the XGR Note: XGB is not available in sklearn. We will have to import xgboost instead

In [28]:
```
from xgboost import XGBRegressor
my_model=XGBRegressor(random_state=0)
mae=getmaecr(my_model,X_tr,Y_train)
mae=mae.min()
mae
```

Out[28]: 159203.90064279578

So the MAE is around 159200. What might the error be?

In [29]:
```
error=err(mae)
error
```

Out[29]: 14.800246994772401

A 14.8% error! That would be 0.5% better than our RFR model's accuracy. There's still a lot we can do to improve our performance. Lets try to hit a 90% accuracy, we're currently at 85.2%.

What can we do?

- specify learning rate
- find ideal leaf node
- use early_stopping_rounds
- increase cv

lets try a few of those

In [52]:
```
my_model = XGBRegressor(n_estimators=500,random_state=0,learning_rate=0.1,n_jobs
mae=getmaecr(my_model,X_tr,Y_train)
mae=mae.min()
mae
```

Out[52]:
```
134175.49570235756
```

In [54]:
```
error=err(mae)
error
```

Out[54]:
```
12.473503909282368
```

It reduced our error by 2.5% We're making progress.

The main contributor behind this is the cv increase (5 -> 20)

What if we increase cv to 30?

In [63]:
```
mae=getmaecr(my_model,X_tr,Y_train)
mae=mae.min()
mae
```

Out[63]:
```
125178.4838679941
```

In [64]:
```
error=err(mae)
error
```

Out[64]:
```
11.637104820906794
```

While increasing the cv does work wonders, it does take a lot of time to run. As increasing the cv even once means XGR runs one more loop.

We will proceed with cv=100

In [82]:
```
mae=getmaecr(my_model,X_tr,Y_train)
mae=mae.min()
mae
```

Out[82]:
```
88723.74571078431
```

In [83]:
```
err(mae)
```

Out[83]:
```
8.248122976378893
```

Simply CV increase boosted our accuracy by 7% Since CV is directly proportional to the efficiency of the XGR.

Currently, our model stands at approx 92% accuracy.

So we have acheived our goal of pushing the accuracy to 90%

The model is ready.