PROJECT 3 : BOSTON HOUSE PRICE PREDICTION – MACHINE LEARNING

1 . INTRODUCTION

In this project, we will develop and evaluate the performance and the predictive power of a model trained and tested on data collected from houses in Boston's suburbs .Once we get a good fit, we will use this model to predict the monetary value of a house located at the Boston's area.

A model like this would be very valuable for a real state agent who could make use of the information provided in a daily basis.

2 . WORKING

In this project, we will apply basic machine learning concepts on data collected for housing prices in the Boston, Massachusetts area to predict the selling price of a new home. We will first explore the data to obtain important features and descriptive statistics about the dataset. Next, we will properly split the data into testing and training subsets, and determine a suitable performance metric for this problem. We will then analyze performance graphs for a learning algorithm with varying parameters and training set sizes. This will enable us to pick the optimal model that best generalizes for unseen data. Finally, we will test this optimal model on a new sample and compare the predicted selling price to your statistics.

3 . ABOUT DATASET

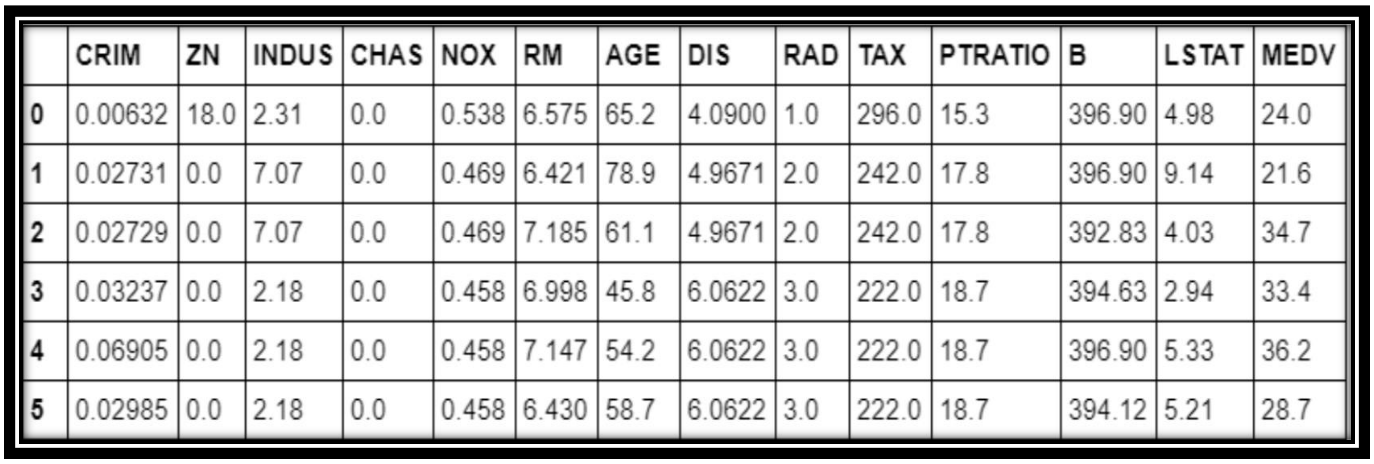
In this dataset, each row describes a boston town or suburb. There are 506 rows and 13 attributes (features) with a target column (price).

The problem that we are going to solve here is that given a set of features that describe a house in Boston, our machine learning model must predict the house price. To train our machine learning model with boston housing data, we will be using scikit-learn’s boston dataset.

Below is a brief description of each feature and the outcome in our dataset:

1. CRIM – per capita crime rate by town
2. ZN – proportion of residential land zoned for lots over 25,000 sq.ft
3. INDUS – proportion of non-retail business acres per town
4. CHAS – Charles River dummy variable (1 if tract bounds river; else 0)
5. NOX – nitric oxides concentration (parts per 10 million)
6. RM – average number of rooms per dwelling
7. AGE – proportion of owner-occupied units built prior to 1940
8. DIS – weighted distances to five Boston employment centres

9. RAD – index of accessibility to radial highways
10. TAX – full-value property-tax rate per $10,000
11. PT – pupil-teacher ratio by town
12. B – 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
13. LSTAT – % lower status of the population
14. MV – Median value of owner-occupied homes in $1000's



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21	28.7

For the purpose of the project the dataset has been preprocessed as follows:

- The essential features for the project are: 'RM', 'LSTAT', 'PTRATIO' and 'MEDV'. The remaining features have been excluded.
- 16 data points with a 'MEDV' value of 50.0 have been removed. As they likely contain censored or missing values.
- 1 data point with a 'RM' value of 8.78 it is considered an outlier and has been removed for the optimal performance of the model.

As our goal is to develop a model that has the capacity of predicting the value of houses, we will split the dataset into features and the target variable. And store them in features and prices variables, respectively

- The features ‘RM’, ‘LSTAT’ and ‘PTRATIO’, give us quantitative information about each datapoint. We will store them in *features*.
- The target variable, ‘MEDV’, will be the variable we seek to predict. We will store it in *prices*.

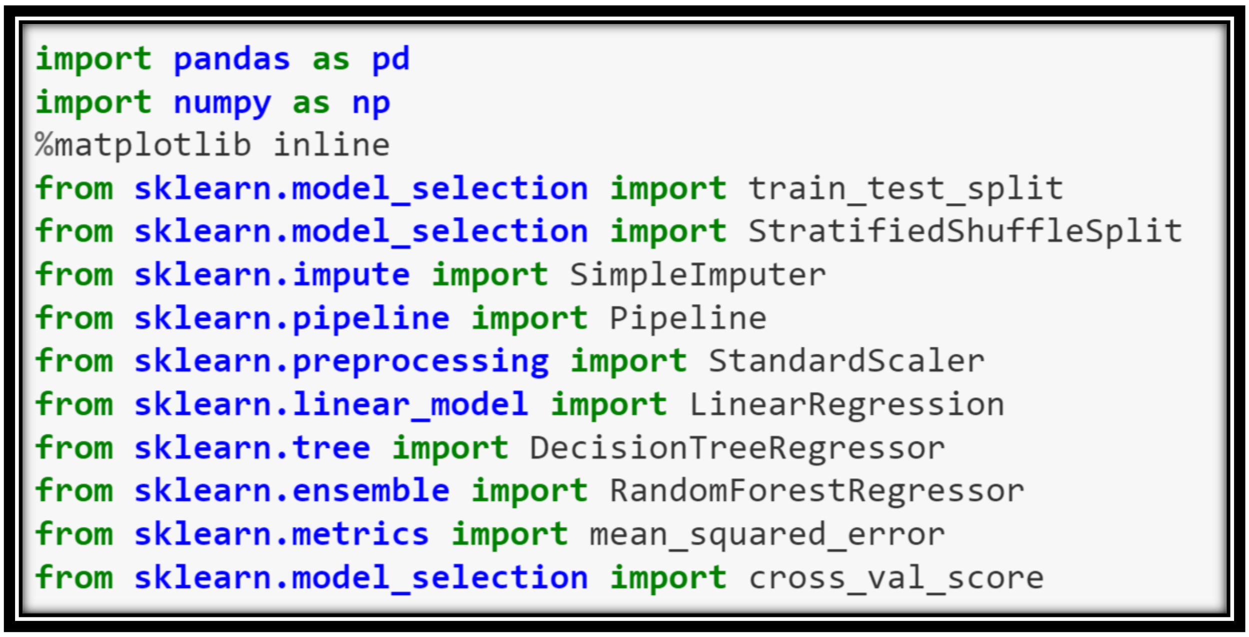
4 . CODE OF THE PROJECT

The GitHub repository of the project is :
https://github.com/Disha6/House-Price-Prediction

5 . EXPLANATION OF CODE

a . Importing libraries:

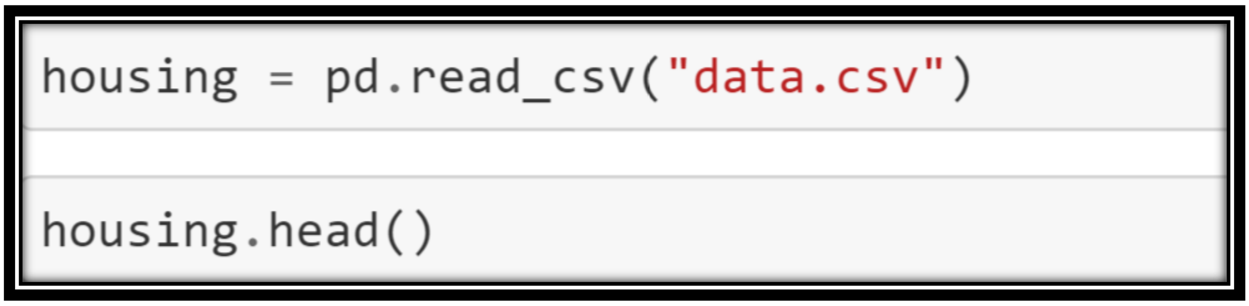
At the start of every notebook we must import necessary libraries in order to simply do Data Science.



```
import pandas as pd
import numpy as np
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
```

b . Reading the dataset:

In this section we read in the data and check the data we are working with. As we check the data we can clearly see the data’s dimension and it’s features.

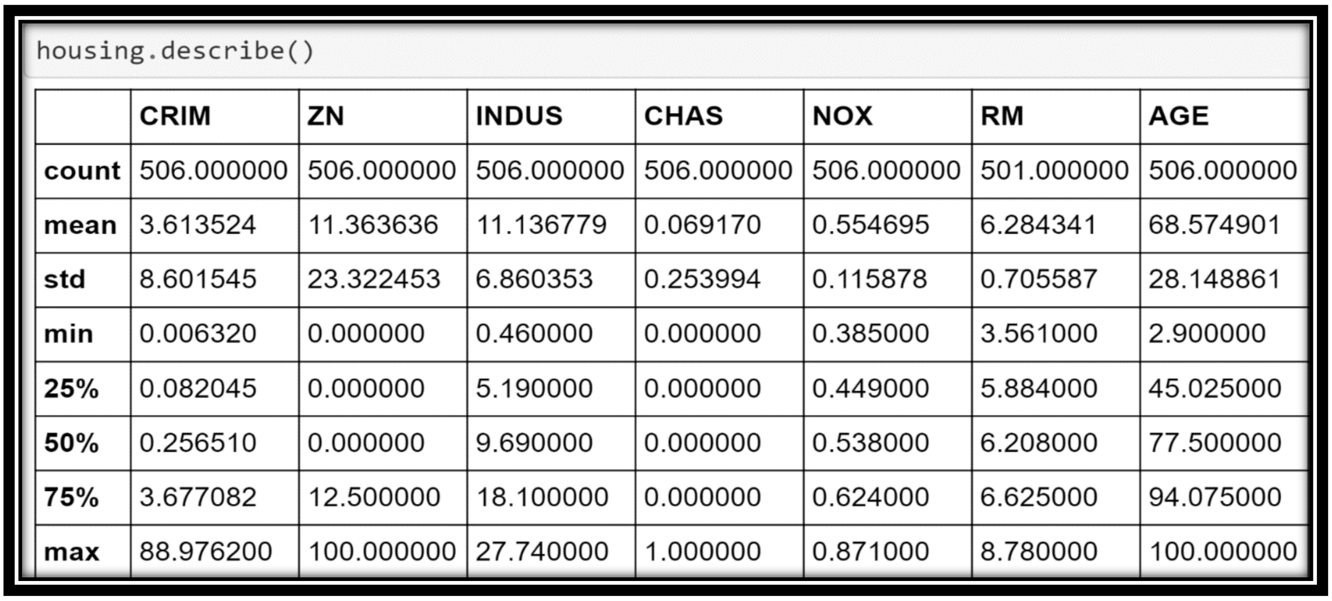


```
housing = pd.read_csv("data.csv")
```

```
housing.head()
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64
 4   NOX      506 non-null    float64
 5   RM       501 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(11), int64(3)
memory usage: 55.5 KB
```

Now we will use **describe()** which is used to view some basic statistical details like percentile, mean, std etc. of a data frame or a series of numeric values.

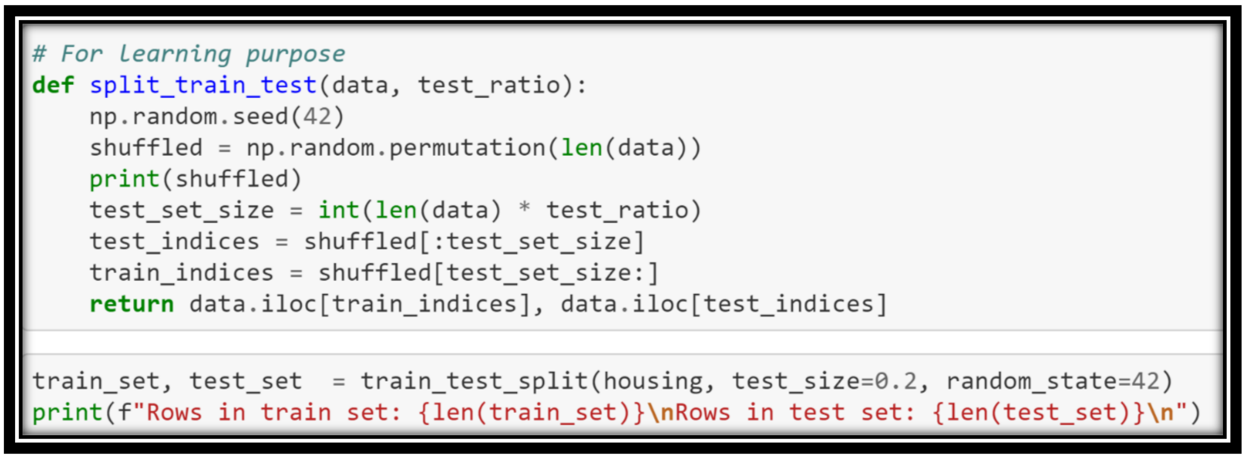


```
housing.describe()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	501.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284341	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.705587	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.884000	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208000	77.500000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.625000	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

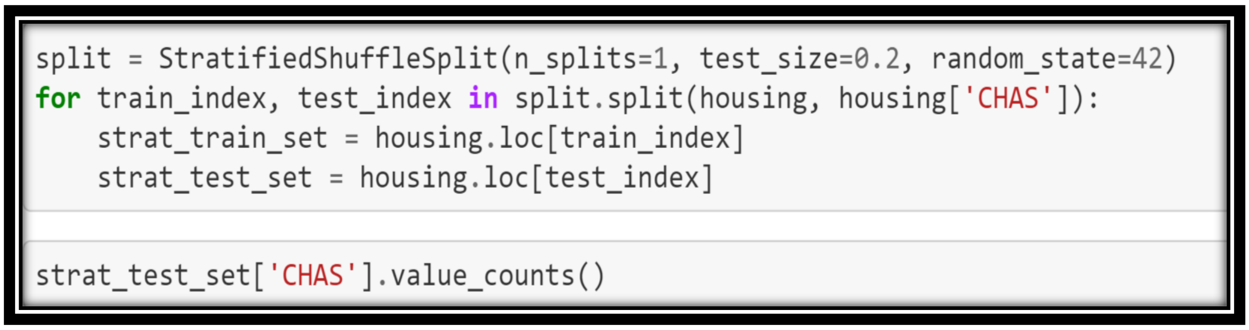
c . Train – Test data:

We will be working on train and test data by splitting it and checking the value counts of both train and test data.



```
# For learning purpose
def split_train_test(data, test_ratio):
    np.random.seed(42)
    shuffled = np.random.permutation(len(data))
    print(shuffled)
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled[:test_set_size]
    train_indices = shuffled[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

```
train_set, test_set  = train_test_split(housing, test_size=0.2, random_state=42)
print(f"Rows in train set: {len(train_set)}\nRows in test set: {len(test_set)}\n")
```



```
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing['CHAS']):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

```
strat_test_set['CHAS'].value_counts()
```

d . Correlation Matrix:

We are going to create now a correlation matrix to quantify and summarize the relationships between the variables.

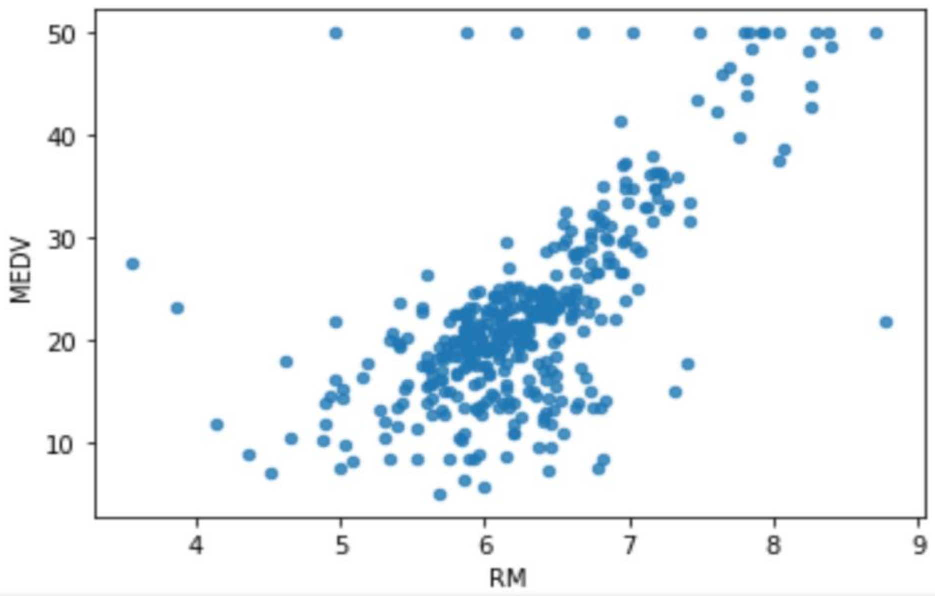
This correlation matrix is closely related with covariance matrix, in fact it is a rescaled version of the covariance matrix, computed from standardize features.

```
corr_matrix = housing.corr()
corr_matrix['MEDV'].sort_values(ascending=False)
```

Seeing it in scatter plot graph.

```
housing.plot(kind="scatter", x="RM", y="MEDV", alpha=0.8)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f62cb2579b0>
```

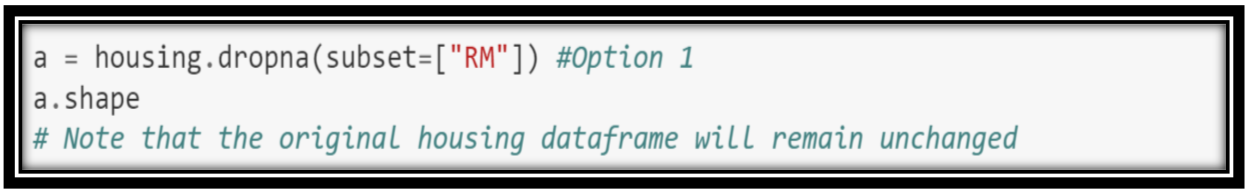


e . Missing attributes:

To take care of missing attributes, we have three options:

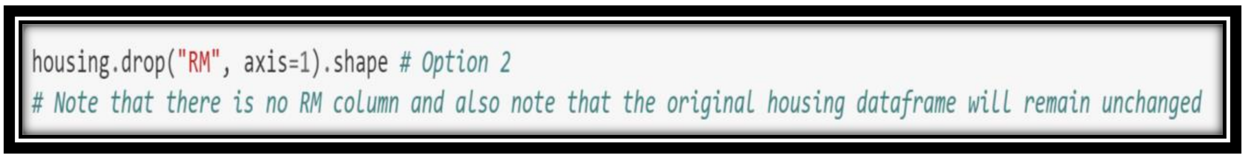
1. Get rid of the missing data points
2. Get rid of the whole attribute
3. Set the value to some value(0, mean or median)

Option 1: working on missing data points.



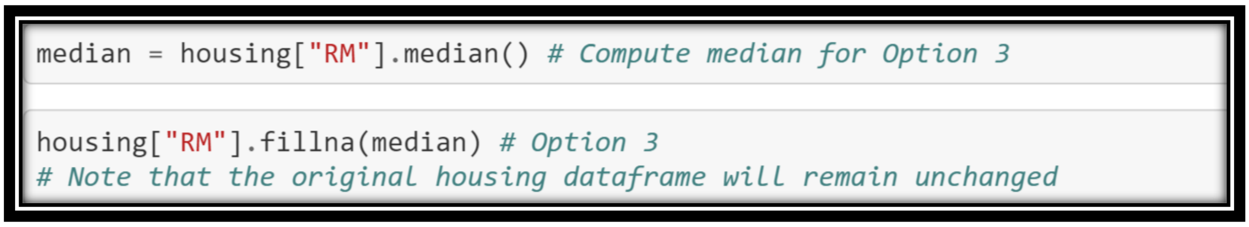
```
a = housing.dropna(subset=["RM"]) #Option 1
a.shape
# Note that the original housing dataframe will remain unchanged
```

Option 2: getting rid of that whole attribute.



```
housing.drop("RM", axis=1).shape # Option 2
# Note that there is no RM column and also note that the original housing dataframe will remain unchanged
```

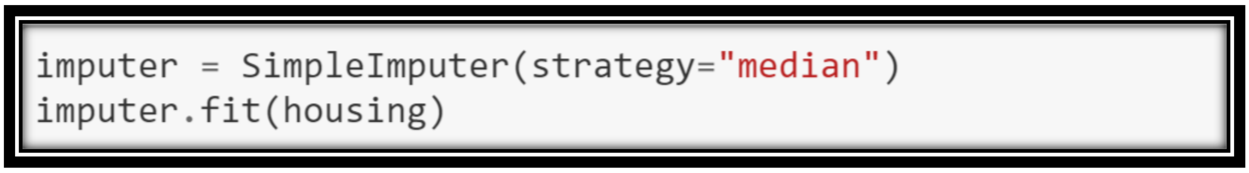
Option 3: filling the space with 0 , mean or median and in here we are filling it with median.



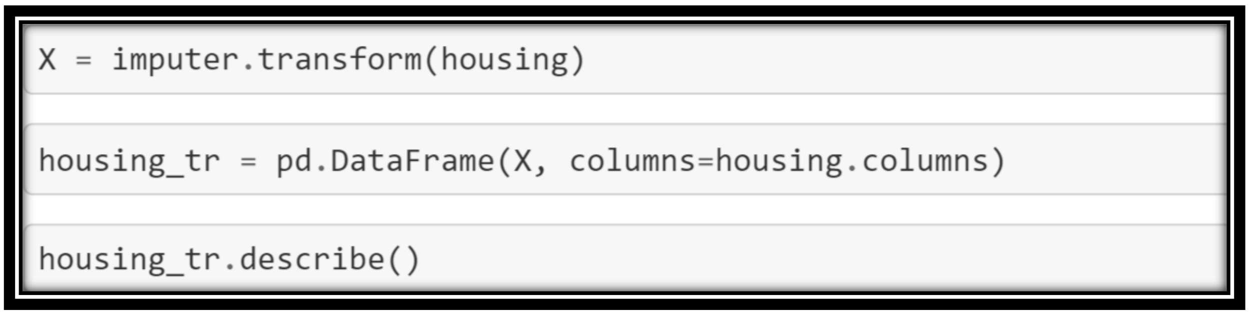
```
median = housing["RM"].median() # Compute median for Option 3

housing["RM"].fillna(median) # Option 3
# Note that the original housing dataframe will remain unchanged
```

f . Using Imputer:

Statistical Imputation for Missing Values in **Machine Learning**. This is called missing data imputation, or imputing for short. A popular approach for data imputation is to calculate a statistical value for each column (such as a mean) and replace all missing values for that column with the statistic.



```
imputer = SimpleImputer(strategy="median")
imputer.fit(housing)
```



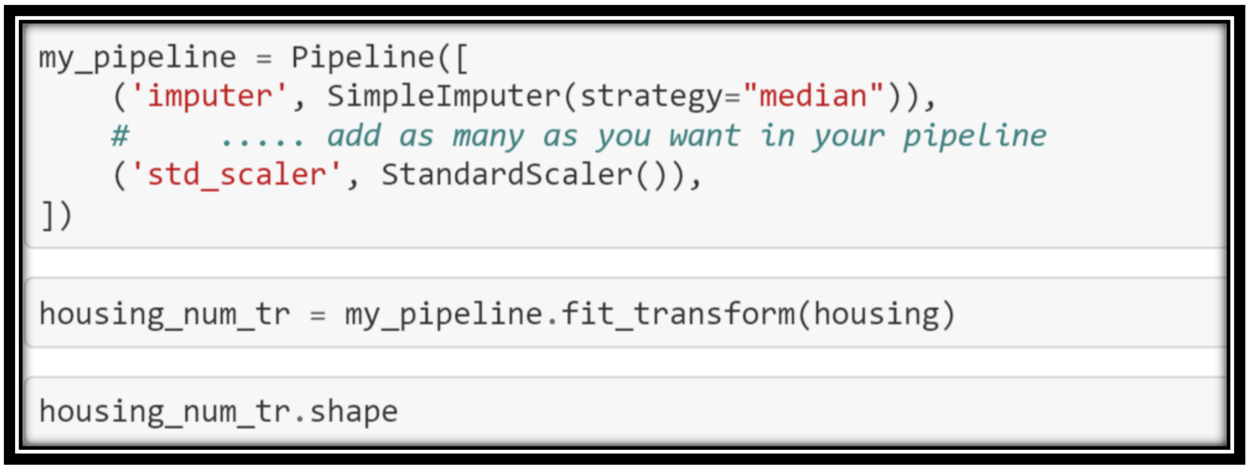
```
X = imputer.transform(housing)

housing_tr = pd.DataFrame(X, columns=housing.columns)

housing_tr.describe()
```

g . Feature Scaling and Pipelining:

Primarily, two types of feature scaling methods:

1. Min-max scaling (Normalization) (value - min)/(max - min) Sklearn provides a class called MinMaxScaler for this
2. Standardization (value - mean)/std Sklearn provides a class called StandardScaler for this.

For creating the pipeline and then checking the dimension of the dataset.



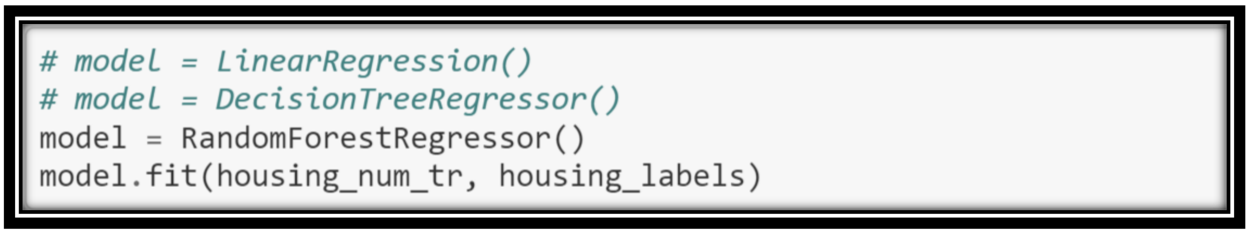
```python
my_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    #     ..... add as many as you want in your pipeline
    ('std_scaler', StandardScaler()),
])
```

```python
housing_num_tr = my_pipeline.fit_transform(housing)
```

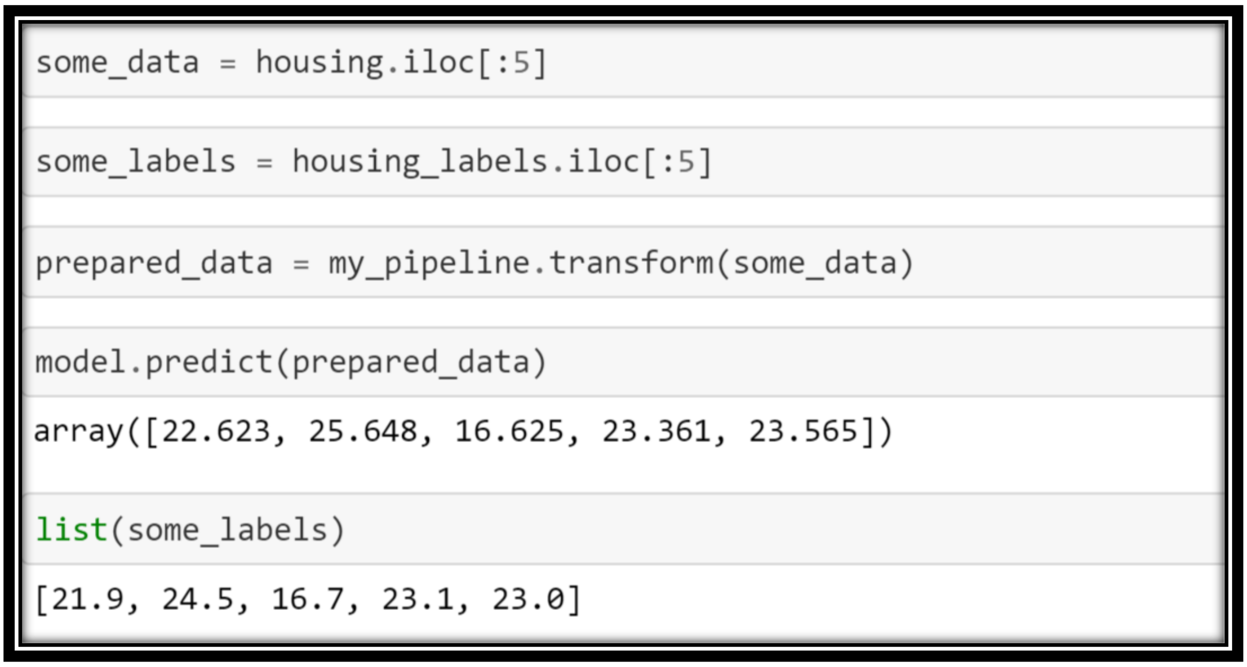
```python
housing_num_tr.shape
```

h . Selecting the Model :

Now it’s a very crucial time to check about various applications and select the model.



```
# model = LinearRegression()
# model = DecisionTreeRegressor()
model = RandomForestRegressor()
model.fit(housing_num_tr, housing_labels)
```



```
some_data = housing.iloc[:5]
```

```
some_labels = housing_labels.iloc[:5]
```

```
prepared_data = my_pipeline.transform(some_data)
```

```
model.predict(prepared_data)
```

```
array([22.623, 25.648, 16.625, 23.361, 23.565])
```

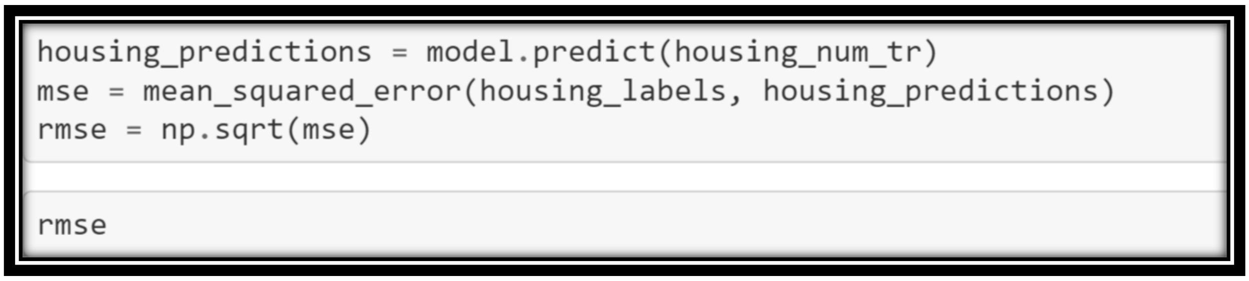
```
list(some_labels)
```

```
[21.9, 24.5, 16.7, 23.1, 23.0]
```

i . Evaluating Model’s Performance:

In this final section of the project, we will construct a model and make a prediction on the client’s feature set using an optimized model from fit_model.

When fitting it on a dataset all the possible combinations of parameter values are evaluated and the best combination is retained.



```
housing_predictions = model.predict(housing_num_tr)
mse = mean_squared_error(housing_labels, housing_predictions)
rmse = np.sqrt(mse)
```

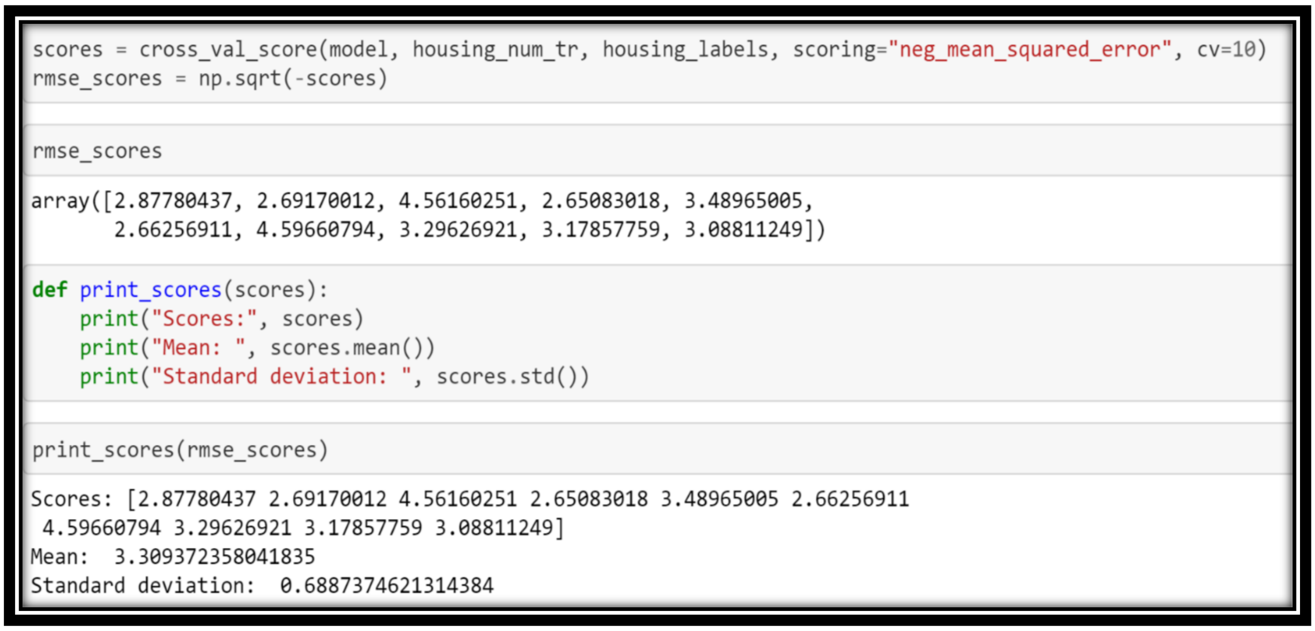
```
rmse
```

j . Cross – Validation:

K-fold cross-validation is a technique used for making sure that our model is well trained, without using the test set. It consist in splitting data into k partitions of equal size. For each partition i, we train the model on the remaining k-1 parameters and evaluate it on partition i. The final score is the average of the K scores obtained.

When evaluating different hyperparameters for estimators, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance.

To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.



```
scores = cross_val_score(model, housing_num_tr, housing_labels, scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```

```
rmse_scores
```

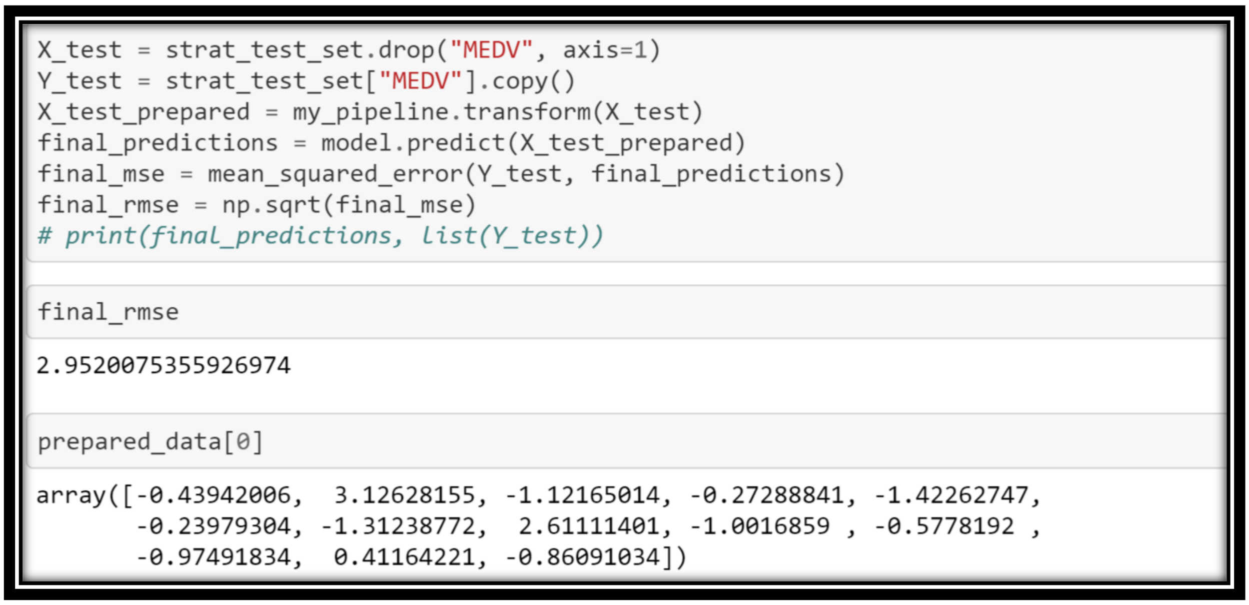
```
array([2.87780437, 2.69170012, 4.56160251, 2.65083018, 3.48965005,
       2.66256911, 4.59660794, 3.29626921, 3.17857759, 3.08811249])
```

```
def print_scores(scores):
    print("Scores:", scores)
    print("Mean: ", scores.mean())
    print("Standard deviation: ", scores.std())
```

```
print_scores(rmse_scores)
```

```
Scores: [2.87780437 2.69170012 4.56160251 2.65083018 3.48965005 2.66256911
 4.59660794 3.29626921 3.17857759 3.08811249]
Mean:  3.309372358041835
Standard deviation:  0.6887374621314384
```

k . Testing the model on Test set:



```
X_test = strat_test_set.drop("MEDV", axis=1)
Y_test = strat_test_set["MEDV"].copy()
X_test_prepared = my_pipeline.transform(X_test)
final_predictions = model.predict(X_test_prepared)
final_mse = mean_squared_error(Y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
# print(final_predictions, list(Y_test))
```

```
final_rmse
```

```
2.9520075355926974
```

```
prepared_data[0]
```

```
array([-0.43942006,  3.12628155, -1.12165014, -0.27288841, -1.42262747,
       -0.23979304, -1.31238772,  2.61111401, -1.0016859 , -0.5778192 ,
       -0.97491834,  0.41164221, -0.86091034])
```