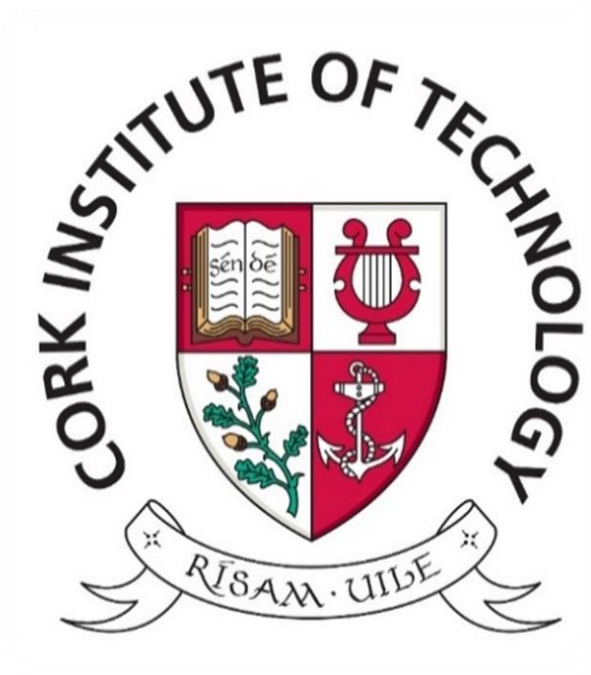
CORK INSTITUTE OF TECHNOLOGY



DEPARTMENT OF MATHEMATICS

MASTER OF SCIENCE IN DATA SCIENCE & ANALYTICS
2019-2020

PREDICTING HOUSING PRICES BY USING MACHINE LEARNING ALGORITHMS

by

SHIVAANI KATRAGADDA

(R00183214)

FOR THE MODULE COMP9060 – APPLIED MACHINE LEARNING

17TH MAY 2020, SUNDAY

SUPERVISOR: Dr. HAITHEM AFLI

Declaration of Authorship

I, Shivaani Katragadda, declare that the work presented in this assignment is my own. I confirm that:

- This work was done wholly or mainly while in candidature for the Masters' degree at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- I understand that my project documentation may be stored in the library at CIT and may be referenced by others in the future.

Signed: SHIVAANI KATRAGADDA

Date: 17TH MAY 2020

TABLE OF CONTENTS

ABSTRACT

House price forecasting is a big real estate problem. This research aims to extract useful knowledge from historical property market data. Machine learning methods are used to analyze actual property deals in order to discover appropriate models for home buyers and sellers. The high discrepancy between house prices in the most expensive and affordable suburbs has been revealed. Housing rates are affected by many variables. The Many existing models of home price forecasts usually are part of the so-called single predictor model, the prediction accuracy of which is not Ideal, and the over-fitting phenomenon often occurs because of noise throughout data. Machine learning is an essential approach to achieving practical and effective solutions for predicting the output for the given input. Various machine learning techniques including classification, regression, and clustering can be utilized to predict the prices of houses. Artificial and Deep Neural Networks (ANNs and DL), Support Vector Machines (SVMs), Ensemble learning (EL), linear and logistic regression, decision trees, Naive Bayes are some of the algorithms that can be used to implement prediction. The primary focus of this research is to use a machine learning technique for predicting housing prices using regression by tuning hyper-parameters with Scikit learn GridSearchCV. The algorithms used in this research are Linear Regression, Decision Tree, and Ensemble Learning _ Random Forest Regressor, among all the three Random Forest Regressor, looks very promising with the best performance. In this project, feature Engineering Encoding is taken as the research element where I have integrated various feature Encoding mechanisms and evaluating the impact of integrating them into my machine learning pipeline. This model helps to estimate houses price for the given set of characteristics. The good model helps homebuyers or real estate agents to make decisions.

Keywords: House price estimation, Housing Dataset, Feature Engineering Encoding, One-Hot Encoder, Ordinal Encoder, Random forest.

1 INTRODUCTION

Civilization development is the basis of house demand increasing day by day. Accurate forecasting house prices have always been a challenge for buyers, sellers, and even for the bankers. A lot of researchers have already worked to solve the complexities of house price prediction. There are many theories produced as a result of the research work contributed by the various researchers around the world.

Some of these theories assume that a particular's geographical position and community determine whether housing prices can increase or reduce, whereas some others thought that the socio-economic conditions play a primary role behind the increase in house prices. We all know that a house price is a number of some defined assortment, so it is naturally a function of regression to predict house prices.

In order to estimate the price of the house, a person usually attempts to find similar properties in his or her neighborhood and will attempt to predict the price of the house based on the data obtained. All of these indicate that the prediction of house prices is an emerging field of regression research that requires knowledge of machine learning. Housing price has many impacts [1][2] which includes time and space, house ages, climate conditions, transport. Existing models of predictions are generally single predictors, i.e. the prediction is applied to a single forecasting model. The Accuracy of this sort of model is unsatisfactory when there is Noise in the Datasets[3]. Some simple ensemble models, like Random forests, would undergo an over-fitting phenomenon If there's more noise in data

1.1 LITERATURE REVIEW

According to the various predictions obtained, the government and developers will determine whether or not to grow the real estate market in the respective areas. In this paper[4], support vector machines (SVM), least-square support vector machines (LSSVM), and partial least square (PLS) methods are used to forecast house prices. Previous history and experiments show that although there is significant non-linearity in the data collection, the outcome of the experiment also shows that SVM and LSSVM methods are superior to PLS in solving the non-linearity problems. The globally optimal solution can be found, and SVM (support vector machine) can accomplish the

forecasting effect by solving a quadratic programming problem. The algorithm's multiple computation efficiencies are compared by the calculation times correlated with similar algorithms.

The model included multiple regression approaches that are discussed and compared. Gaussian Regression Processes (GP) are selected for model due to its robust and probabilistic approach to the requirements for learning and sample selection. In London[5], the exploited data set structure for distributing computations for the small independent local models is used to manage different forms of the past property's large dataset. Final forecasts obtained are by the recombination of forecasts from existing local models. And by training these models and generating predictions on the application's server-side, and able to offload the output of computationally intensive tasks and concentrate on producing visualizations on the client-side. The findings indicate that the solution to the issue has been relatively successful and that it can generate different predictions that are competitive to other housing price-prediction models.

Structural and non-structural models with and without frameworks Use 10-variable linear structural general equilibrium model for forecasting the US[6] real house price index and its turning point in 2006. In a forecasting experiment for comparing to the dynamic stochastic general equilibrium model, using Bayesian approaches, there are also numerous analyses of Bayesian and classical time series models. In addition to the introduction of regular vector-auto regressive and Bayesian vector-auto regressive models, the knowledge information of either 10 or 120 quarterly series in a few of models to capture the fundamental impact is also included. Comparing the out-of-sample forecast performance of alternative models with the use of the average root mean square error for forecasts. The conclusion is that the small-scale Bayesian-shrinkage model (10 variables) out performs the other models, including the Bayesian-shrinkage large-scale model (120 variables). Finally, estimated model through 2005 is used to predict the turning point in 2006.

There are two major challenges forced to face the researchers. The biggest challenge is to identify the optimum number of features that will help predict the direction of house prices accurately. In this paper[7] it is mentiond that productivity growth in different residential building sectors has an impact on house prices growth. The model shows how housing prices can appear apparently trendy in which housing wealth rises faster than income over an extended period of time, then collapses and experiences an extended decline.

In his doctoral thesis, Lowrance[8] states that he considered the interior living room the most important element in deciding the costs of housing in his study. He also references the census tract's median income that determines the rates. The features such as floor area, lot size category, bathroom number, and bedroom number, uniform age, and garage size as features and uses linear regression techniques to forecast home prices.

2 RESEARCH

In this project, Feature Engineering Encoding is taken as the research element where I have integrated various feature encoding mechanisms and evaluating the impact of integrating them into my machine learning pipeline

2.1 What is Feature Engineering?

Feature engineering is the process of using data domain knowledge to produce features that make algorithms for machine learning work. Feature engineering is fundamental to machine learning and is both expensive and difficult. Automated feature learning can obviate the need for manual feature engineering.

Feature engineering is an informal topic but in applied machine learning, it is considered essential.

Coming up with features is difficult, time-consuming, requires expert knowledge. "Applied machine learning" is basically feature engineering.

— Andrew Ng, Machine Learning and AI via Brain simulations[9]

2.2 Why Feature Engineering is Important?

If feature engineering is performed correctly, the predictive ability of machine learning algorithms is improved by building features from raw data that help accelerate the process of machine learning. Feature Engineering is something of an art.

There are several steps to solve any Machine Learning problem that we need to follow

1. Collecting data.

2. Data Cleaning.

3. Feature Engineering.

4. Defining the model.

5.Training, testing, and predicting the output of the model.

All machine learning algorithms essentially use certain input data to construct outputs. These input data contain features, usually in the form of organized columns. To function properly, algorithms need features of a certain unique characteristic.

What are the Methods for Encoding? And Encoding Techniques

The data collection can include categorical variables in many specific Data Science practices. Typically, these variables are stored as text values. Machine learning is based on mathematical equations, if we hold categorical variables as they are, it will cause a problem. Many algorithms accept categorical values without further modification, but even in those situations, it is still a discussion point about whether or not to encode the variables. In this case, the algorithms which do not support categorical values are left with methodologies for encoding.

2.3 Encoding Methodologies:

2.3.1 Nominal Encoding:

The Order of data does not matter in Nominal Encoding, various techniques in Nominal Encoding are:

1.One Hot Encoding

2. Frequency Encoding:

3.Mean Encoding

2.3.2 Ordinal Encoding:

The Order of data matters in Ordinal Encoding, various techniques in Ordinal Encoding are:

1.Label Encoding

2.Target or Mean Guided Ordinal Encoding

One Hot Encoding: — In this method, we map each category to a vector containing 1 and 0 which denotes the presence of the feature. The number of vectors depends on which categories we prefer to keep. For features of high cardinality, this method produces many columns that significantly slow down the learning process. Between hot encoding and dummy encoding, there is a little confusion about what to use and when to use. They 're almost the same but one hot encoding generates the number of columns equal to the number of columns and dummy producing is less. In the testing process, this will eventually be done by the developers accordingly.

User	City
1	Roma
2	Madrid
1	Madrid
3	Istanbul
2	Istanbul
1	Istanbul
1	Roma

→

User	Istanbul	Madrid
1	0	0
2	0	1
1	0	1
3	1	0
2	1	0
1	1	0
1	0	0

ONE HOT ENCODING

Label Encoding: — Each category is given a value from 1 through N in this encoding (where N is the category number for the feature). It could look like (Car < Bus < Truck .... 0 < 1<2). After encoding, categories that have some relationships or similar to each other or lose some data.

CAT73
A
A
C
B
A
C
B

CAT73 label_encoded
1
1
3
2
1
3
2

LABEL ENCODING

Frequency Encoding: — It is a way of using the category frequency as labels. In some cases where the frequency is somewhat similar to the target variable, depending upon the requirement of the results, it allows the model to understand and assign the weight in direct and inverse proportion.

Feature Encoding

- Frequency Encoding
  - Encoding of categorical levels of feature to values between 0 and 1 based on their relative frequency

A	0.44 (4 out of 9)
B	0.33 (3 out of 9)
C	0.22 (2 out of 9)

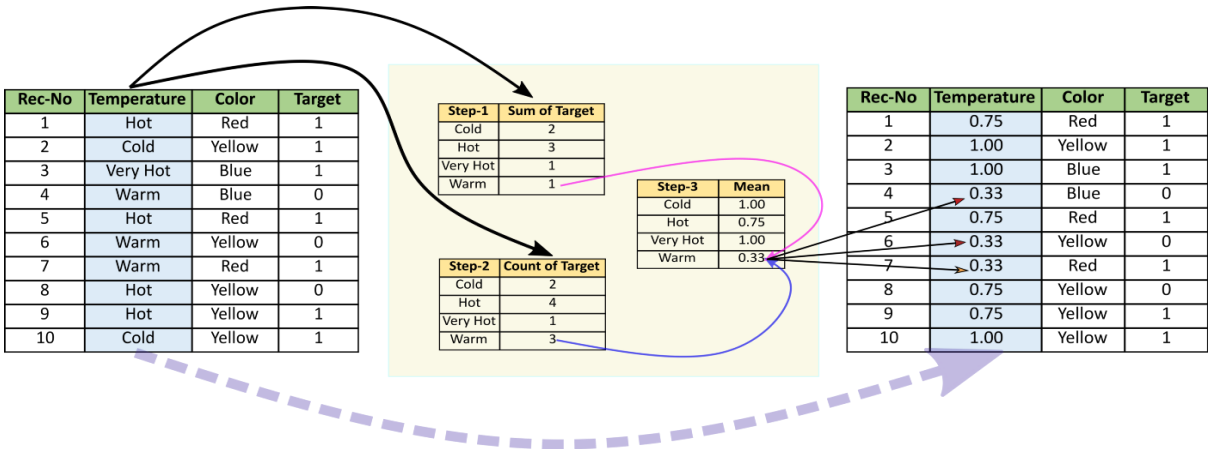
Feature	Encoded Feature
A	0.44
A	0.44
A	0.44
A	0.44
B	0.33
B	0.33
B	0.33
C	0.22
C	0.22

H_2O.ai

FREQUENCY ENCODING

Mean Encoding: - Mean encoding is identical to label encoding, except that the labels are directly correlated with the target. For example, the mean target encoding on a training data is determined for each group in the feature label, with the mean value of the target variable.

The benefits of the mean target encoding are that the amount of the data is not disturbed and results in better learning.



MEAN ENCODING

2.4 Encoding statistics:

Encoding variability[10] describes the encoding variation within a category, individually. When we think about the variation in one-hot encoding, the variability depends on the time of implementation at which it determines how many categories to take that will have enough impact on the target. Many methodologies for encoding also display a major variation that is defined at validation time.

In order to calculate the Encoding Distinguishability, we count the category labels and then calculate the standard label deviation. That shows the encoded categories distinguishability that they are either far apart or clustered within a region.

I used OrdinalEncoder and OneHotEncoder in this assignment. I used OrdinalEncoder for converting category into numerical as ml algos prefer to work with numbers, and OneHotEncoder for the categorical attribute.

3 METHODOLOGY

The main aim of this research is to apply Feature Engineering Encoding, perform hyper-parameter optimization and to predict the housing price by using regression.

Dataset collection:

The dataset I have chosen for this research project is "housing" dataset from the GitHub repository ageron/handson-ml[11] which was collected by Aurelien Geron. This is a regression problem where the aim is to predict the housing price. The data is related to the houses found in a given California district and some summary statistics about them based on the 1990 census data. The names of the features are longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value, ocean_proximity. Dataset contains the records of the 20640 rows. There is a total of 10 features out of which 9 are numerical and 1 is categorical. The target feature is the median_house_value.

The 8-Step in the Workflow are as follows:

1. Basic understanding of the dataset
2. Make sense of the data from a high level
   - data types (number, text, object, etc.)
   - continuous/discrete
   - basic stats (min, max, std, median, etc.) using boxplot
   - frequency via histogram
   - scales and distributions of different features
3. Create the training and test sets using proper sampling methods, e.g., random vs. stratified
4. Correlation analysis (attribute combinations)
5. Data cleaning (missing data, outliers)
6. Data transformation via pipelines (categorical text to number using one hot encoding, feature scaling via normalization/standardization, feature combinations)
7. Train and cross validate different models and select the most promising one (Linear Regression, Decision Tree, and Random Forest were tried in this tutorial)

8. Fine tune the model using trying different combinations of hyperparameters

STEP 1: Basic understanding of the dataset

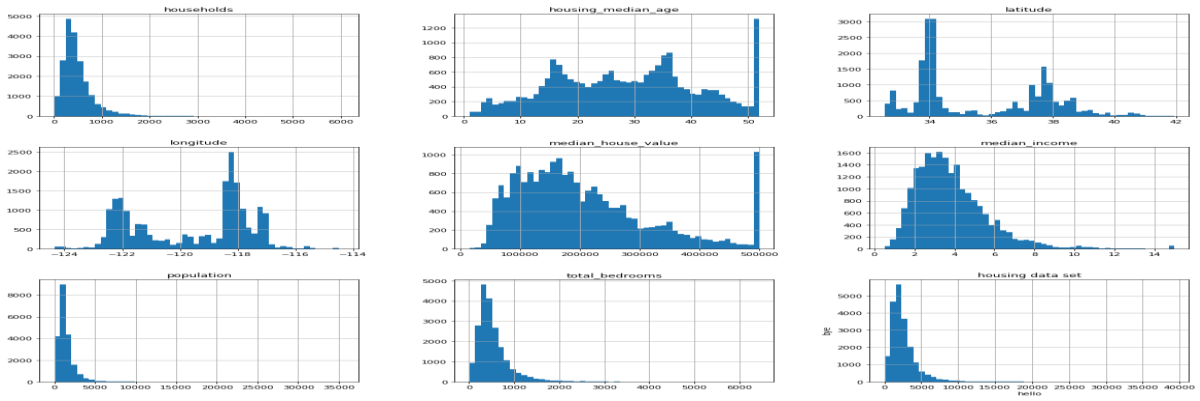
At first, we have to understand what is the given dataset how many observations and features it contains and whether is there any missing values present in the data.

- total observations: 20640
- total columns (features): 10
- data type of each feature: 9 numbers and 1 object
- meaning of each feature: it is very important to work with domain expert to fully understand each feature
- any null values (e.g., total_bedrooms is 20433, which indicates null)

STEP 2: Make sense of the data from a high level

What are the things observed from the basic statistics with visualizaitons

1. scan each column and see whether the data make sense at a high level.
    - longitude, latitude and house median age seem to be OK.
    - total rooms and total bedrooms are in hundreds and thousands - this does make sense given each row is a district
    - population seems to be OK.
    - median income is apparently.
    - median house value data is OK and this is the target variable, i.e., which we need to build a model to predict this value.
2. Feature scaling: It is observed that the features have very different scales, which we need to handle later



3. Distribution: from the histograms, we can tell many of them are skewed, i.e., having a long tail on one side or the other. In many cases, we need to transform the data so that they have more bell-shaped distributions.

STEP 3: Create the training and test sets using proper sampling methods, e.g., random vs. stratified

Random sampling

```
def split_train_test(data, test_ratio ):
    shuffled_indeces=np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indeces = shuffled_indeces[:test_set_size]
    train_indeces = shuffled_indeces[test_set_size:]
    return data.iloc[train_indeces], data.iloc[test_indeces]
train_set, test_set = split_train_test(housing, 0.2)
```

The above function will generate a different test set every time you run the program, which is undesirable. A few potential solutions:

1. save the test set and training set for the first run into csv files
2. set a fixed random number of generator seed so that the random numbers are always the same

1 and 2 won't work if new data is loaded. The book proposed a method of hashing the identifier of the data to get the same test set

About Sampling

In order to make sure that the test set is a good representation of the overall population. We may want to consider different sampling techniques:

- random sampling (what we used above): OK if the dataset is large enough
- stratified sampling: the population is divided into homogeneous subgroups called *strata* and the correct instances are sampled from each *stratum*

Strarified sampling

#CREATING STRATIFIED SPLIT BASED ON INCOME_CAT

from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size= 0.2, random_state=42)

for train_index, test_index in split.split(housing, housing['income_cat']):

strat_train_set = housing.loc[train_index]

strat_test_set = housing.loc[test_index]

We can compare the different sampling results for the test set by comparing them to the overall population distribution. As we can see, stratified sampling's distributions are much more similar to the overall distributions compared with random sampling.

After that I performed some visualization for the geographical data.

STEP 4: Correlation analysis (attribute combinations)

And then exploring the data to look for correlations between different attributes. correlation coefficient is between -1 and 1, representing negative and positive correlations. 0 means there is no liner correlation. Correlation is said to be linear if the **ratio of change** is constant, otherwise is non-linear. Visual inspection of data is very important,

which cannot be replaced by simply looking at the correlation coefficients.

Attribute Combinations

Sometime, the combinations of attributes are more meaningful and interesting in terms of solving the business problems

- rooms per household: total number of rooms per district is not useful but rooms per household may be interesting
- bedroom/total room ratio
- population per household

There are two findings after combining attributes:

1. rooms_per_household is slightly more correlated (0.146285) with house value than total_rooms (0.135097)
2. bedrooms_per_room is much more correlated (-0.259984) than total_rooms (0.135097) and total_bedrooms (0.047689): houses with lower bedroom/room ratio is more expensive: this sort of make sense, more expensive houses may have more offices, dens, playrooms, etc.

STEP 5: Data cleaning (missing data, outliers)

Typically, data need to be cleaned and transformed before trying different ML algorithms.

Missing data: There is one column i.e. total_bedrooms I replaced the missing values in that column by median value.

Outliers: I did not consider the outliers here as there are not significant and I did not remove them.

Separate the predictors (independent variables) and labels(target/dependent variables)

We want to create a clean training set first. So I drop target labels i.e median_house_value for the training set and made a copy of the target label vector. After that by using a computer, I found the median and removed non-numerical attributes for imputer by making a copy of the data frame and replaced computed median values in the data set. And then converted NumPy array to pandas data frame

STEP 6: Data transformation via pipelines (categorical text to number using one hot encoding, feature scaling via normalization/standardization, feature combinations)

categorical text to number using encoding

Most ML algorithms work with numbers better. Therefore, we often need to convert text attributes into numerical attributes. For ocean_proximity, we have two ways to handle this problem:

1. map each category to a number, such as "<1H OCEAN" is 0, "INLAND" is 1, 'NEAR OCEAN' is 4, etc. The problem with this solution is that ML algorithm may think 4 is greater than 0, which could cause problem.
2. To address the problem in 1, we can also create a binary variable for each attribute, which is called one-hot encoding (only one is 1 hot, all others are 0 cold)

Perform 1 and 2 and combine both

Feature Engineering Encoding Techniques:

Here I used OrdinalEncoder for converting category into numerical as ml algos prefer to work with numbers and performing OneHotEncoder to the categorical attribute

Custom Transformers

You may need to develop custom transformers - you can just write a simple function for that or if you want your transformer work with Scikit-Learn, you need to developed the transformers as a class.

Feature Scaling

Typically, ML algorithms don't perform well when the input numerical attributes have very different scales. For example, in this housing dataset (shown above), you can see median_income ranges from 0.4999 to 15 while total rooms are between 6 and 39320. Note that scaling the target values is typically not required.

Two major scaling methods (two different scalers):

- normalization (aka, min-max scaling): values are rescaled to between 0 and 1 using (value-min)/(max-min)
- standardization: values are rescaled to have unit variance: (value - mean)/variance

Normalization can be dramatically affected by outliner data while standardization handles outliers very well.

STEP 7: Train and cross validate different models and select the most promising one (Linear Regression, Decision Tree, and Random Forest were tried in this tutorial)

Select and Train a Model

I am going to train three model they are as follows

1. Linear Regression
2. Decision Tree Regressor
3. Random Forest Regressor

Random Forest is much better that the previous two models. Therefore choosing Random Forest for fine tuning the model for better performance.

Step 8: Fine tune the model using trying different combinations of hyperparameters

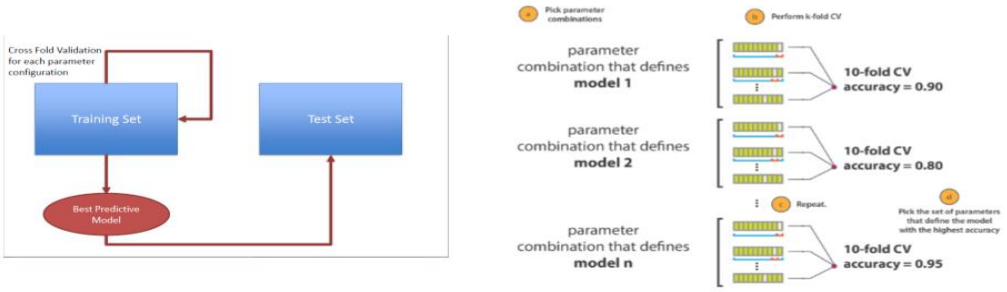
We chose Random Forest and Gradient Boosting for hyper-parameter tuning on the basis of best mean accuracy and mean f1 score on the training set evaluated using the 10-fold stratified cross validation. Hyper-parameters are the set of arguments within the class of each machine learning algorithm which are not directly learnt by the algorithm on its own. There is a defined search process for finding out the optimal values of these hyper-parameters which is as follows-:

1) A searching algorithm

2) A parameter space (range of values for the parameters)

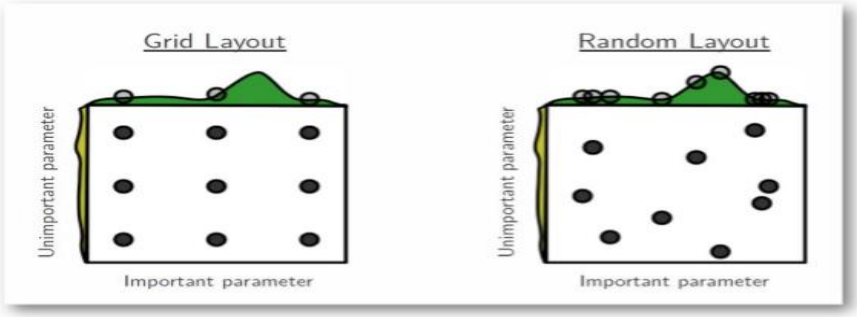
3) A method for searching the values

4) Cross validation for finding the best hyper-parameters.



We will only perform the hyper-parameter tuning on the training data and test data is kept aside for final evaluation. There would be n different parameter combinations (n different models) depending on the parameter space that the user has given. Each combination of parameter goes through the k-fold cross validation and we obtain the mean accuracy for that combination. The parameter combination which has the maximum (best) k-fold mean accuracy will be chosen as the optimal parameters for that machine learning algorithm.

There are mainly 2 techniques for hyper-parameter tuning- Grid Search CV and Randomized Search CV



Grid Search CV – In this approach, we try every combination of preset values given in the form of parameter grid which is a list of dictionaries. When the hyper-parameters are more, then the number of evaluation increases exponentially with each additional parameter.

Randomized Search CV – Random combinations of hyper-parameters are used to find the optimal parameters. It tries a random range of values given in the form of parameter distribution.

We have applied Randomized Search CV because of the following reasons.

1. Due to a smaller number of parameter combinations because of random selection, this is faster than Grid Search CV.

2. We are tuning the best algorithms for our dataset which are gradient boosting and random forest which has a lot of hyper-parameters so this Random Search CV is more suitable in this case.

3. Another reason for using this Random Search CV is that the magnitude of influence of each parameter in both of these algorithms varies. Performing tuning by using GridSearchCV and getting best combinations of parameters, best estimators, evaluation scores and analyzed the best model with their errors

4 EVALUATION

Now evaluating the results of the models.

1.Creating the train and test by using random sampling, this sampling is acceptable when the dataset is large enough

```
17]: #III] CREATING A TEST SET
     #FUNCTION TO SPLIT DATASET TO TRAIN AND TEST
     import numpy as np
     def split_train_test(data, test_ratio ):
         shuffled_indeces=np.random.permutation(len(data))
         test_set_size = int(len(data) * test_ratio)
         test_indeces = shuffled_indeces[:test_set_size]
         train_indeces = shuffled_indeces[test_set_size:]
         return data.iloc[train_indeces], data.iloc[test_indeces]

18]: train_set, test_set = split_train_test(housing, 0.2)

19]: len(train_set)

19]: 16512

20]: len(test_set)

20]: 4128
```

2.Creating the stratified split based on income_cat, in stratified sampling the population is divided into homogeneous subgroups called *strata* and the correct instances are sampled from each *stratum*. The length of the stratified sample training set.

```
35]: #CREATING STRATIFIED SPLIT BASED ON INCOME_CAT
     from sklearn.model_selection import StratifiedShuffleSplit
     split = StratifiedShuffleSplit(n_splits=1, test_size= 0.2, random_state=42)
     for train_index, test_index in split.split(housing, housing['income_cat']):
         strat_train_set = housing.loc[train_index]
         strat_test_set = housing.loc[test_index]

36]: print(strat_train_set)
     print(len(strat_train_set))
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	\
17606	-121.89	37.29	38.0	1568.0	351.0	
18632	-121.93	37.05	14.0	679.0	108.0	
14650	-117.20	32.77	31.0	1952.0	471.0	
3230	-119.61	36.31	25.0	1847.0	371.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	
...	...	...	...	...	...	
6563	-118.13	34.20	46.0	1271.0	236.0	
12053	-117.56	33.88	40.0	1196.0	294.0	
13908	-116.40	34.09	9.0	4855.0	872.0	
11159	-118.01	33.82	31.0	1960.0	380.0	
15775	-122.45	37.77	52.0	3095.0	682.0	

	population	households	median_income	median_house_value	\
17606	710.0	339.0	2.7042	286600.0	
18632	306.0	113.0	6.4214	340600.0	
14650	936.0	462.0	2.8621	196900.0	
3230	1460.0	353.0	1.8839	46300.0	
3555	4459.0	1463.0	3.0347	254500.0	
...	...	...	...	...	
6563	573.0	210.0	4.9312	240200.0	
12053	1052.0	258.0	2.0682	113000.0	
13908	2098.0	765.0	3.2723	97800.0	
11159	1356.0	356.0	4.0625	225900.0	

3.The stratified sample testing set.

```
In [37]: print(strat_test_set)
         print(len(strat_test_set))
```

```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
5241      -118.39     34.12                29.0       6447.0          1012.0
10970     -117.86     33.77                39.0       4159.0           655.0
20351     -119.05     34.21                27.0       4357.0           926.0
6568      -118.15     34.20                52.0       1786.0           306.0
13285     -117.68     34.07                32.0       1775.0           314.0
...           ...       ...                 ...          ...             ...
20519     -121.53     38.58                33.0       4988.0          1169.0
17430     -120.44     34.65                30.0       2265.0           512.0
4019      -118.49     34.18                31.0       3073.0           674.0
12107     -117.32     33.99                27.0       5464.0           850.0
2398      -118.91     36.79                19.0       1616.0           324.0

       population  households  median_income  median_house_value  \
5241       2184.0       960.0         8.2816            500001.0
10970      1669.0       651.0         4.6111            240300.0
20351      2110.0       876.0         3.0119            218200.0
6568       1018.0       322.0         4.1518            182100.0
13285      1067.0       302.0         4.0375            121300.0
...           ...         ...            ...                 ...
20519      2414.0      1075.0         1.9728             76400.0
17430      1402.0       471.0         1.9750            134000.0
4019       1486.0       684.0         4.8984            311700.0
12107      2400.0       836.0         4.7110            133500.0
2398        187.0        80.0         3.7857             78600.0

      ocean_proximity income_cat
5241        <1H OCEAN          5
10970       <1H OCEAN          4
20351       <1H OCEAN          3
6568            INLAND         3
13285           INLAND         3
```

4. Feature Engineering Encoding:

I used ordinal encoder to convert categorical data to numerical data and One Hot encoding to categorical variables.

```
In [62]: #converting category into numerical as ml algos prefer to work with numbers
         from sklearn.preprocessing import OrdinalEncoder
         ordinal_encoder = OrdinalEncoder()
         housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
         housing_cat_encoded[:10]
```

```
Out[62]: array([[0.],
                [0.],
                [4.],
                [1.],
                [0.],
                [1.],
                [0.],
                [1.],
                [0.],
                [0.]])
```

```
In [63]: #getting the list of categories
         ordinal_encoder.categories_
```

```
Out[63]: [array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
                dtype=object)]
```

```
In [64]: #performing onehot encoding the categorical attribute
         from sklearn.preprocessing import OneHotEncoder
         cat_encoder = OneHotEncoder()
         housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
         print(housing_cat_1hot)
         print(housing_cat_1hot.toarray())
         print(cat_encoder.categories_)
```

```
  (0, 0)        1.0
  (1, 0)        1.0
  (2, 4)        1.0
```

5.Custom Transfer

In [65]:
```
#custom transformation
from sklearn.base import BaseEstimator, TransformerMixin

rooms_xi, bedrooms_xi, population_xi, households_xi = 3,4,5,6
class CombinedAttributeAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self
    def transform(self, X, y=None):
        rooms_per_household=X[:, rooms_xi] / X[:,households_xi]
        population_per_household = X[:, population_xi] / X[:,households_xi]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_xi] / X[:,rooms_xi]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributeAdder(add_bedrooms_per_room = False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

In [66]:
```
housing_extra_attribs
```

Out[66]:
```
array([[-121.89, 37.29, 38.0, ..., '<1H OCEAN', 4.625368731563422,
        2.094395280235988],
       [-121.93, 37.05, 14.0, ..., '<1H OCEAN', 6.008849557522124,
        2.7079646017699117],
       [-117.2, 32.77, 31.0, ..., 'NEAR OCEAN', 4.225108225108225,
        2.0259740259740258],
       ...,
       [-116.4, 34.09, 9.0, ..., 'INLAND', 6.34640522875817,
        2.742483660130719],
       [-118.01, 33.82, 31.0, ..., '<1H OCEAN', 5.50561797752809,
        3.808988764044944],
       [-122.45, 37.77, 52.0, ..., 'NEAR BAY', 4.843505477308295,
```

6.Created pipeline, name/estimator pairs for pipeline steps foe each estimator except the last one must be transformers with fit_transform() method. Pipeline fit() calls each fit_transform() of each estimator and fit() for the last estimator.

In [67]:
```
#Transformation pipelines
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([('imputer', SimpleImputer(strategy = 'median')),
    ('attribs_adder', CombinedAttributeAdder()),
    ('std_scaler',StandardScaler()),])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

In [68]:
```
#pipelines to transform both numeric and categorical attributes
from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num)
cat_attribs = ['ocean_proximity']
full_pipeline = ColumnTransformer([
    ('num', num_pipeline, num_attribs),
    ('cat', OneHotEncoder(),cat_attribs),
])
housing_prepared = full_pipeline.fit_transform(housing)
```

In [69]:
```
housing_prepared
```

Out[69]:
```
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
         0.        ,  0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, ...,  0.        ,
         0.        ,  1.        ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.        ,
         0.        ,  0.        ],
```

7.Now building **linear regression** model which is imported from sklearn.linear_model, the predictions, actual labels, mean square error and root mean square error are clearly shown in the figure

```
In [70]: #VI] TRAINING THE ML MODEL
         #training a linear regression model
         from sklearn.linear_model import LinearRegression
         lin_reg = LinearRegression()
         lin_reg.fit(housing_prepared, housing_labels)

Out[70]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [71]: #trying linear regression on few instances
         some_data = housing.head()
         some_labels = housing_labels.head()
         some_data_prepared  = full_pipeline.transform(some_data)
         print("predictions:", lin_reg.predict(some_data_prepared))
         print("actual labels:",list(some_labels))

         predictions: [210644.60459286 317768.80697211 210956.43331178  59218.98886849
          189747.55849879]
         actual labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]

In [72]: #MEASURING  linear regression model using "root mean squared error (RMSE)" METRICS
         from sklearn.metrics import mean_squared_error
         housing_predictions = lin_reg.predict(housing_prepared)
         lin_mse = mean_squared_error(housing_labels,housing_predictions)
         lin_rmse = np.sqrt(lin_mse)
         print(lin_mse)
         print(lin_rmse)

         4709829587.971121
         68628.19819848923
```

1.Linear Regression

The predictions and actual labels I obtained are as follows

predictions: [210644.60459286 317768.80697211 210956.43331178 59218.98886849 189747.55849879]

actual labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]

The mean square error (MSE) is 4709829587.971121 and the RMSE (root-mean-square error) is 68628.19819848923

8.Now building the **decision tree regressor** which is imported from sklearn.tree, we can clearly see that the performance of the decision tree is very poor when compare to linear regression, so performing cross validation.

```
In [73]: #AS THERE IS HIGH ERROR,  LETS TRAIN OTHER REGRESSION MODEL FOR OUR DATA AND
         from sklearn.tree import DecisionTreeRegressor
         tree_reg = DecisionTreeRegressor()
         tree_reg.fit(housing_prepared,housing_labels)

Out[73]: DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                               max_leaf_nodes=None, min_impurity_decrease=0.0,
                               min_impurity_split=None, min_samples_leaf=1,
                               min_samples_split=2, min_weight_fraction_leaf=0.0,
                               presort=False, random_state=None, splitter='best')

In [74]: housing_predictions = tree_reg.predict(housing_prepared)
         tree_mse = mean_squared_error(housing_predictions, housing_labels)
         tree_rmse = np.sqrt(tree_mse)
         print(tree_mse)
         print(tree_rmse)

         0.0
         0.0

In [75]: #performing cross-validation on traiing set using k-folds cross validation from sklearn.model_selectidon
         from sklearn.model_selection import cross_val_score
         score = cross_val_score(tree_reg, housing_prepared, housing_labels, scoring = 'neg_mean_squared_error', cv = 10)
         tree_rmse_scores = np.sqrt(-score)

         print(tree_rmse_scores)

In [76]: print(score)
         print(tree_rmse_scores)

         [-4.78319881e+09 -4.62515605e+09 -5.27544464e+09 -5.07602779e+09
          -4.89659590e+09 -5.70410747e+09 -4.97705323e+09 -4.97306970e+09
          -5.85157184e+09 -4.77551515e+09]
```

2.Decision tree

The mean square error (MSE) is 0.0 and the RMSE (root-mean-square error) is 0.0
No error at all! This is a typical example of **overfitting**. You cannot use the same set of data for both training and validation. Cross-Validation (CV) can help with model validation. Scikit-Learn CV features expect a utility function (greater is better) than a cost function (lower is better).Therefore I performed cross-validation on training set using k-folds cross validation from sklearn.model_selection.

Now, Decision Tree Model's performance is actually worse than the Linear R egression Model: mean 70870.3426307563 and standard deviation 2828.90 593121599

```
[76]: print(score)
      print(tree_rmse_scores)

      [-4.78319881e+09 -4.62515605e+09 -5.27544464e+09 -5.07602779e+09
       -4.89659590e+09 -5.70410747e+09 -4.97705323e+09 -4.97306970e+09
       -5.85157184e+09 -4.77551515e+09]
      [69160.67387653 68008.49981984 72632.25619109 71246.24756975
       69975.68078319 75525.54183489 70548.23337021 70519.99503417
       76495.56743045 69105.10218639]

[77]: def display_scores(score):
          print('scores:', score)
          print('mean:', score.mean())
          print('standard deviation:', score.std())

      display_scores(tree_rmse_scores)

      scores: [69160.67387653 68008.49981984 72632.25619109 71246.24756975
       69975.68078319 75525.54183489 70548.23337021 70519.99503417
       76495.56743045 69105.10218639]
      mean: 71321.7798096505
      standard deviation: 2641.5492672920777
```

9.Building **Random forest Regressor** which is imported from sklearn.ensemble, we can see that the random forest is better when compare to decision tree and linear regression so considering this model to perform Hyper-parameter Optimization

```
In [78]: #TRAINING RANDOM FOREST REGRESSOR FOR MUCH MORE BETTER MODEL PERFORMANCE
         from sklearn.ensemble import RandomForestRegressor
         forest_reg = RandomForestRegressor()
         forest_reg.fit(housing_prepared, housing_labels)

         C:\Users\DELL\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:245: FutureWarning: The default value of n_estimators will
         change from 10 in version 0.20 to 100 in 0.22.
           "10 in version 0.20 to 100 in 0.22.", FutureWarning)

Out[78]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                               max_features='auto', max_leaf_nodes=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, n_estimators=10,
                               n_jobs=None, oob_score=False, random_state=None,
                               verbose=0, warm_start=False)

In [79]: #measuring the performance of the model
         from sklearn.model_selection import cross_val_score
         score = cross_val_score(forest_reg, housing_prepared, housing_labels, scoring = 'neg_mean_squared_error', cv = 10)
         forest_rmse_scores = np.sqrt(-score)

In [80]: display_scores(forest_rmse_scores)

         scores: [50972.29174586 49435.02637466 53205.7158712  54163.05623253
          52283.5871508  55700.93915854 49821.95682114 50279.16057801
          56444.23414768 52705.10287346]
         mean: 52501.10709538808
         standard deviation: 2301.905161492119
```

3.Random Forest

NOW TRAINING RANDOM FOREST REGRESSOR FOR MUCH MORE BETTER MODEL PERFORMANCE. The mean: 52971.90483666565 and standard deviation: 2050.000768490915

10.For Hyper-parameter Optimization considering random forest and tuning it by using GridSearchCV

```
In [81]: #fine-tuning the model by grid search
         from sklearn.model_selection import GridSearchCV
         param_grid = [
             {'n_estimators': [3,10,30], 'max_features': [2,4,6,8]},
             {'bootstrap': [False], 'n_estimators': [3,10], 'max_features': [2,3,4]}]
         forest_reg = RandomForestRegressor()
         grid_search = GridSearchCV(forest_reg, param_grid, cv=5, scoring = 'neg_mean_squared_error', return_train_score = True)
         trained_model = grid_search.fit(housing_prepared, housing_labels)

In [82]: #getting best combinations of parameters
         #getting the best estimstors
         #getting the evaluation scores
         print(grid_search.best_params_)
         print(grid_search.best_estimator_)
         cvres = grid_search.cv_results_
         for mean_score, params in zip(cvres['mean_test_score'], cvres['params']):
             print(np.sqrt(-mean_score), params)

         {'max_features': 6, 'n_estimators': 30}
         RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                               max_features=6, max_leaf_nodes=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, n_estimators=30,
                               n_jobs=None, oob_score=False, random_state=None,
                               verbose=0, warm_start=False)
         64026.58511450056 {'max_features': 2, 'n_estimators': 3}
         56021.57781868762 {'max_features': 2, 'n_estimators': 10}
         53045.4202132021 {'max_features': 2, 'n_estimators': 30}
         60655.62410276234 {'max_features': 4, 'n_estimators': 3}
         52714.40266458917 {'max_features': 4, 'n_estimators': 10}
         50578.13420541593 {'max_features': 4, 'n_estimators': 30}
         58900.35345327106 {'max_features': 6, 'n_estimators': 3}
         52214.82496185249 {'max_features': 6, 'n_estimators': 10}
         50083.40379784913 {'max_features': 6, 'n_estimators': 30}
```

11. Analyzing the best model with their errors.

```
In [84]: #analyzing the best model with their errors
         feature_importance = grid_search.best_estimator_.feature_importances_
         print(feature_importance)

         [7.39669221e-02 6.89529155e-02 4.29759351e-02 1.78746184e-02
          1.76309923e-02 1.86560939e-02 1.61914967e-02 3.54249123e-01
          5.91307003e-02 1.06916131e-01 6.51244958e-02 7.66421028e-03
          1.43204371e-01 9.14684239e-05 3.35942312e-03 4.01110266e-03]

In [85]: #displaying importance in ascending  order along with the attribute names
         extra_attribs = ['rooms_per_hhold', 'pop_per_hhold', 'bedrooms_per_room']
         cat_encoder = full_pipeline.named_transformers_['cat']
         cat_one_hot_attribs = list(cat_encoder.categories_[0])
         attributes = num_attribs + cat_one_hot_attribs + extra_attribs
         sorted(zip(feature_importance, attributes), reverse =True)

Out[85]: [(0.3542491232043371, 'median_income'),
          (0.14320437097989694, 'NEAR OCEAN'),
          (0.10691613118100486, 'INLAND'),
          (0.07396692214020702, 'longitude'),
          (0.06895291552504272, 'latitude'),
          (0.06512449579112246, 'ISLAND'),
          (0.059130700325351816, '<1H OCEAN'),
          (0.04297593513551952, 'housing_median_age'),
          (0.018656093867830972, 'population'),
          (0.01787461835747287, 'total_rooms'),
          (0.017630992338623398, 'total_bedrooms'),
          (0.0161914966610921, 'households'),
          (0.007664210284704937, 'NEAR BAY'),
          (0.004011102664908375, 'bedrooms_per_room'),
          (0.003359423119021552, 'pop_per_hhold'),
          (9.146842386332906e-05, 'rooms_per_hhold')]
```

12.Now evaluating the model on the test set and measuring its performance by computing 95% confidence interval for the generalization error.

```
In [86]: #evaluating model on the test set and measuaring its performance
         final_model = grid_search.best_estimator_
         x_test = strat_test_set.drop('median_house_value', axis = 1)
         y_test = strat_test_set['median_house_value'].copy()
         x_test_prepared = full_pipeline.transform(x_test)
         final_predictions = final_model.predict(x_test_prepared)
         final_mse = mean_squared_error(y_test, final_predictions)
         final_rmse = np.sqrt(final_mse)
         print(final_rmse)

         48613.70716067042

In [87]: #COMPUTING 95% CONFIDANCE INTERVAL FOR THIS GENERALISATION ERROR
         from scipy import stats
         confidance = 0.95
         squared_errors = (final_predictions - y_test) ** 2
         squared_errors

Out[87]: 5241     2.833836e+06
         10970    5.186247e+08
         20351    2.270044e+08
         6568     1.051273e+09
         13285    3.648100e+08
                      ...
         20519    5.303809e+08
         17430    3.466854e+09
         4019     4.796246e+06
         12107    1.600000e+09
         2398     3.386852e+09
         Name: median_house_value, Length: 4128, dtype: float64

In [88]: np.sqrt(stats.t.interval(confidance, len(squared_errors)-1, loc = squared_errors.mean(), scale = stats.sem(squared_errors)))
Out[88]: array([46599.79271341, 50547.44668995])
```

We can see that the performance of random forest regressor is best when compare to decision tree and linear regression.

5 CONCLUSION AND FUTURE WORK

Initially all the data pre-processing steps were applied to dataset to make it a valid input for various machine learning and optimization algorithms. Out of a range of machine learning algorithms, random forest was the top performing models among decision tree and linear regression that went under hyper-parameter optimization using Grid Search CV method. This model showed some improvement in terms of performance on the test set after parameter tuning. The dataset which I choose is related to regression and it contains 9 numerical columns and 1 categorical column. We then integrated various feature Engineering Encoding mechanisms and compared the performance on the test set with tuned random forest. The feature encoding techniques didn't show the significant improvement as the numbers of features are very less in our dataset. As there is only one categorical column in the dataset, we cannot see that much difference when compare with huge number of categorical columns this may be considered as disadvantage here as the dataset is not huge and contains only one categorical column.

Some of the possible areas of future work could be from either data-preparation phase or hyper-parameter tuning of the various models. Different techniques for handling the missing values like imputation can be tried to investigate on the performance metrics. Some more feature Engineering Encoding techniques could be explored if the dataset is huge with so many categorical columns. We could try random search CV technique and can try out the various combinations of hyper-parameters to improve the performance of our machine learning Regressor in predicting the housing price.

6 REFERENCES:

CITATIONS BY USING IEEE STYLE:

[1] B. Park and J. K. Bae, "Using machine learning algorithms for housing price prediction: The case of Fairfax County, Virginia housing data," *Expert Syst. Appl.*, vol. 42, no. 6, pp. 2928–2934, Apr. 2015, doi: 10.1016/j.eswa.2014.11.040.

[2] E. L. Glaeser and C. G. Nathanson, "An extrapolative model of house price dynamics," *J. Financ. Econ.*, vol. 126, no. 1, pp. 147–170, Oct. 2017, doi: 10.1016/j.jfineco.2017.06.012.

[3] U. Rajan, A. Seru, and V. Vig, "The failure of models that predict failure: Distance, incentives, and defaults," *J. Financ. Econ.*, vol. 115, no. 2, pp. 237–260, Feb. 2015, doi: 10.1016/j.jfineco.2014.09.012.

[4] "(PDF) Housing Value Forecasting Based on Machine Learning Methods," *ResearchGate*. https://www.researchgate.net/publication/270627564_Housing_Value_Forecasting_Based_on_Machine_Learning_Methods (accessed May 17, 2020).

[5] http://www.doc.ic.ac.uk/~mpd37/theses/2015_beng_aaron-ng.pdf

[6] R. Gupta, A. Kabundi, and S. M. Miller, "Forecasting the US real house price index: Structural and non-structural models with and without fundamentals," *Econ. Model.*, vol. 28, no. 4, pp. 2013–2021, Jul. 2011, doi: 10.1016/j.econmod.2011.04.005.

[7] J. A. Kahn, "What Drives Housing Prices?," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 1264048, Sep. 2008. doi: 10.2139/ssrn.1264048.

[8] https://cs.nyu.edu/media/publications/lowrance_roy.pdf

[9] H. Tripathi, "Different Type of Feature Engineering Encoding Techniques for Categorical Variable Encoding," *Medium*, Sep. 26, 2019. https://medium.com/analytics-vidhya/different-type-of-feature-engineering-encoding-techniques-for-categorical-variable-encoding-214363a016fb (accessed May 16, 2020).

REFERENCES:

[10] "Overview of Encoding Methodologies," *DataCamp Community*, Dec. 21, 2018. https://www.datacamp.com/community/tutorials/encoding-methodologies (accessed May 16, 2020).
[11] https://github.com/ageron/handson-ml