

HOUSE PRICE PREDICTION USING MACHINE LEARNING METHODS

A

Project Report

*Submitted in partial fulfilment of the
Requirements for the award of the Degree of*

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING

By

NAMSANI DEEPAK

1602-19-733-136

Under the guidance of

Dr. D. BASWARAJ

PROFESSOR, CSE DEPT



Department of Computer Science & Engineering

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Ibrahimbagh, Hyderabad-31

2023

Vasavi College of Engineering (Autonomous)
(Affiliated to Osmania University)
Hyderabad-500 031
Department of Computer Science & Engineering



DECLARATION BY THE CANDIDATE

I am **NAMSANI DEEPAK**, bearing hall ticket numbers and **1602-19-733-136**, hereby declare that the project report entitled **“HOUSE PRICE PREDICTION USING ML”** under the guidance of **Dr .B. BASWARAJ**, Professor, Department of Computer Science & Engineering, VCE, Hyderabad, is submitted in partial fulfilment of the requirement for the award of the degree of **BACHELOR OF ENGINEERING** in **COMPUTER SCIENCE & ENGINEERING**.

This is a record of bonafide work carried out by us and the results embodied in this project report have not been submitted to any other university or institute for the award of any other degree or diploma.

NAMSANI DEEPAK,
1602-19-733-136.

Vasavi College of Engineering (Autonomous)
(Affiliated to Osmania University)
Hyderabad-500 031
Department of Computer Science & Engineering



BONAFIDE CERTIFICATE

This is to certify that the project entitled “**HOUSE PRICE PREDICTION USING ML**” being submitted by **NAMSANI DEEPAK**, bearing hall ticket numbers **1602-19-733-136**, in partial fulfilment of the requirements for the award of the degree of **BACHELOR OF ENGINEERING IN COMPUTER SCIENCE & ENGINEERING** is a record of bonafide work carried out by them under my guidance.

Dr. B. Baswaraj,
Professor,
Internal guide

Dr.T.Adilakshmi,
Professor &HOD,
Dept. of CSE

Acknowledgement

We are thankful to the College Management for Encouraging us to do our Project titled **HOUSE PRICE PREDICTION USING ML**. We extend our heart-felt gratitude to our Professor, **Dr. M. SUNITHA REDDY** and our Internal Guide **Dr. B. Baswaraj** and Head of the Dept., **Dr. T. ADILAKSHMI** for their Invaluable Guidance and Support throughout our project. We extend our heartfelt gratitude to the Principal, **Dr. S. V. RAMANA**, Vasavi College of Engineering, Ibrahimbagh, for permitting us to undertake this Project. Their guidance was unforgettable, and their constructive suggestions helped us in finishing the project Effectively.

NAMSANI DEEPAK,
1602-19-733-136.

Abstract

This project aims to predict housing prices in Paris using machine learning techniques. The dataset used in this project is the Paris Housing dataset, which was obtained from Kaggle. The dataset contains information about the different attributes of houses such as the number of rooms, location, area, etc. In this project, we utilized various machine learning algorithms such as linear regression, XG Boost, and random forests to train and test our models. After thorough analysis, we found that the random forest and polynomial regression algorithms provides the best accuracy in predicting the housing prices. Our application allows users to input the attributes of the house they want to purchase and returns an estimated price. The accuracy of the model is approximately 99%, making it a reliable tool for predicting housing prices in Paris. This project can be extended to other cities and can provide an accurate estimate of the housing prices in those areas as well.

Table of Contents

Abstract	v
List of Tables	viii
List of Figures	ix
1. Introduction	1
1.1. Project Overview	1
1.2. Predictive analysis	2
1.3. Problem Definition	3
1.4. Objectives	4
1.5. About the datasets	5
2. Literature Survey	9
2.1. House Price Prediction using Machine Learning	9
2.2. Developing an application using Flask	10
2.3. Popular regression models for price prediction	11
3. System design	13
3.1. High Level Description	13
3.2. Process Flow	14
3.3. Assumptions	16
3.4. Software Requirements	17
3.5. Hardware Requirements	17
4. Implementation	18
4.1. Implementing Machine learning code on Jupyter notebook	18
4.2. Implementing individual prediction models	31
4.2.1 Linear Regression	31
4.2.2 KNN Regression	32
4.2.3 Random forest Regression	33
4.2.4 Polynomial Regression	34
4.2.5 MLP Regressor	35
4.2.6 XG Boost Regression	36

4.3.	Implementing the whole application	37
5.	Results and conclusions	41
5.1.	Testing the application	41
5.2.	Performance measures	43
5.3.	Conclusions	44
6.	Future work	45
7.	References	46

List of Tables

Table 5.2.1.1 : Data when models were performed on training data which is of 22730 entries 50

Table 5.2.2.1 : Data when models were trained on the 'ParisHousing' dataset which is the test dataset for the study 50

List of Figures

Figure 1.5.1	: Viewing train.csv	7
Figure 1.5.2	: Train.csv	7
Figure 1.5.3	: Viewing ParisHousing.csv	7
Figure 1.5.4	: ParisHousing.csv	8
Figure 3.2.1	: Process Flow	14
Figure 4.1.1	: Define required functions, import libraries and load the dataframe.	18
Figure 4.1.2	: Viewing the dataframe	19
Figure 4.1.3	: Table showing the correlation between attributes	19
Figure 4.1.4	: The correlation heatmap	20
Figure 4.1.5	: Using the corr_features functions	20
Figure 4.1.6	: We devised few derived features	21
Figure 4.1.7	: Function to detect outliers using inter-quartile range	21
Figure 4.1.8	: Studying the 'squareMeters'	22
Figure 4.1.9	: Studying the 'numberOfRooms' attribute	23
Figure 4.1.10	: The attributes 'hasYard' and 'hasPool'	23
Figure 4.1.11	: Studying the 'floors' attribute	24
Figure 4.1.12	: Studying the 'cityCode' attribute.	25
Figure 4.1.13	: Studying the 'numPreviousOwners' attribute.	25
Figure 4.1.14	: 'HouseAge' is another feature	26
Figure 4.1.15	: The attributes 'isNewBuilt' and 'hasStormProtector'	26
Figure 4.1.16	: Studying the 'basement' attribute	27

Figure 4.1.17 : Studying the ‘attic’ attribute	28
Figure 4.1.18 : Studying the ‘hasGuestRoom’ attribute	29
Figure 4.1.19 : Treating the ‘ActualArea’, ‘AreaPerHouse’, ‘AreaPerRoom’, and ‘Extra’ attributes for outliers	29
Figure 4.1.20 : Scaling the dataframe.	30
Figure 4.2.1.1 : Fitting the linear regression model	31
Figure 4.2.1.2 : Scatter plot for actual vs predicted values.	31
Figure 4.2.2.1 : Fitting the KNN regression model	32
Figure 4.2.2.2 : Scatter plot for actual vs predicted values.	32
Figure 4.2.3.1 : Fitting the Random forest regression model	33
Figure 4.2.3.2 : Scatter plot for actual vs predicted values.	33
Figure 4.2.4.1 : Fitting the polynomial regression model	34
Figure 4.2.5.1 : Fitting the MLP Regressor model	35
Figure 4.2.5.2 : Scatter plot for actual vs predicted values.	35
Figure 4.2.6.1 : Fitting the XG Boost model	36
Figure 4.2.6.2 : Scatter plot for actual vs predicted values.	36
Figure 4.3.1 : Using the XG Boost model in our application	37
Figure 4.3.2 : Flask application source code on Pycharm.	38
Figure 4.3.3 : Source code of the form written in HTML.	39
Figure 4.3.4 : Command prompt commands.	40
Figure 4.3.5 : The ‘http://127.0.0.1:5000’ webpage.	40
Figure 5.1.1 : One entry of the ‘ParisHousing’ dataset.	41
Figure 5.1.2 : Entering the values.	41

1. Introduction

1.1 Project Overview

In recent years, machine learning has become an essential tool for businesses and individuals to gain valuable insights and predictions from large datasets. One of the most significant applications of machine learning is in the field of real estate, where it is used to predict house prices accurately. In this project, we have developed a machine learning-based house price prediction model that can accurately predict the price of a house in Paris using the ParisHousing dataset available on Kaggle.

The goal of this project is to create an application that can predict the price of a house in Paris accurately. We achieved this by developing a machine learning model that takes into account several factors that influence the price of a house, such as the number of rooms, location, and area. Our model uses six different algorithms to predict the price of a house based on these factors.

The first step in building the model was to pre-process the data. We used Inter quartile range (IQR) to eliminate any outliers in the data. Outliers can significantly affect the accuracy of the model, so it's crucial to remove them. Once the outliers were removed, we scaled the data to ensure that all the variables were on the same scale. This step is essential because variables with different scales can influence the model in unpredictable ways.

The next step was to train six different models on the pre-processed data. We chose six different algorithms to ensure that we get the best possible results. The six algorithms we used are Linear Regression, K-nearest neighbour, Random Forest, Polynomial regression, Multilayer Perceptron and XGBoost. Each of these algorithms has its strengths and weaknesses, and by using all of them, we ensured that we get a well-rounded prediction.

Once the models were trained, we tested them on a test dataset to evaluate their performance. We used the Root Mean Squared Error (RMSE) to evaluate the performance of each model. The RMSE measures the difference between the predicted and actual values of the test data. A lower RMSE indicates better performance.

After evaluating the performance of each model, we selected the model with the lowest RMSE as our final model. This model was then deployed as a Flask application that takes input from the user and predicts the price of a house in Paris based on the input. The user can input the number of rooms, the location, and the area of the house, and the application will predict the price of the house.

Overall, this project demonstrates the power of machine learning in predicting house prices accurately. By using the ParisHousing dataset and pre-processing the data, scaling the variables, and training six different models, we were able to create a robust model that accurately predicts the price of a house in Paris. By deploying this model as a Flask application, we have created a tool that can be used by anyone to get an accurate estimate of the price of a house in Paris.

1.2. Predictive analysis

Price prediction is a complex process that requires a deep understanding of the underlying data and variables that influence price movements. To build accurate price prediction models, data scientists typically start by identifying the key factors that drive price movements in a particular market or industry. For example, in the stock market, factors such as earnings reports, interest rates, and geopolitical events can all influence the price of a particular stock.

Once the key factors have been identified, data scientists can use a variety of statistical and machine learning techniques to build predictive models. These models typically involve a combination of regression analysis, time series analysis, and machine learning algorithms such as neural networks and decision trees.

In addition to historical data, predictive models can also incorporate other relevant data sources such as news articles, social media sentiment, and macroeconomic indicators. By incorporating these additional data sources, predictive models can provide a more comprehensive and accurate view of future price movements.

The accuracy of price prediction models can be evaluated using a variety of metrics such as mean squared error, mean absolute error, and R-squared. These metrics provide a measure of how closely the predicted prices align with the actual prices over a given time period.

One of the main challenges in building accurate price prediction models is dealing with the inherent uncertainty and volatility of markets. Price movements can be influenced by a wide range of factors, many of which are unpredictable or difficult to quantify. As a result, price prediction models must be constantly updated and refined to account for new information and changing market conditions.

Despite these challenges, price prediction remains a valuable application of predictive analysis that can provide significant value to businesses and investors. By accurately predicting price movements, organizations can make more informed decisions about pricing, investment strategies, and supply chain management, leading to better financial outcomes and increased competitiveness in the marketplace.

1.3. Problem Definition

Predicting house prices accurately requires considering multiple factors that can influence the price of a property. These factors may include the location of the property, the size of the house, the number of rooms, the age of the property, the amenities available, and the overall condition of the property. All of these factors can play a crucial role in determining the price of a property. Hence, selecting the right set of attributes is essential in predicting house prices accurately.

To determine which attributes are most important, we can use various techniques such as data analysis, feature engineering, and statistical modeling. Data analysis can help us understand the relationship between each attribute and the target variable (price). We can use techniques such as

correlation analysis, scatter plots, and box plots to identify the most significant attributes that contribute to the price. Feature engineering can help us extract more information from the available attributes, and it can also help us create new features that are more relevant to the target variable. Statistical modeling can help us understand the relationship between the attributes and the target variable in a more systematic manner. We can use regression models, decision trees, and ensemble models to identify the most important attributes and to predict the price accurately.

In terms of the most suitable algorithm, there are various paradigms in machine learning that can be applied to predict house prices. Some of the commonly used algorithms are linear regression, decision trees, random forests, support vector regression, gradient boosting, and XGBoost. Each of these algorithms has its strengths and weaknesses, and the choice of algorithm depends on the characteristics of the data and the desired level of accuracy. For instance, linear regression is a simple and interpretable algorithm that works well for datasets with a small number of features, while decision trees and ensemble models are more complex and can handle large and complex datasets. Support vector regression is effective in handling noisy datasets, while gradient boosting and XGBoost are known for their high accuracy.

Overall, accurately predicting house prices requires a combination of selecting the right set of attributes and applying the most suitable algorithm. By leveraging the power of machine learning techniques such as data analysis, feature engineering, and statistical modeling, we can develop robust and accurate models that can help buyers, sellers, and real estate professionals make informed decisions.

1.4. Objectives

Develop an accurate and robust model: The primary objective is to develop a machine learning model that can accurately predict the price of a house based on the available attributes. The model should be able to handle the complexity of the data, and it should be robust enough to handle new data.

Improve the data quality: The quality of the data plays a crucial role in the accuracy of the model. Hence, it is essential to identify and eliminate any missing or incorrect data, perform feature engineering, and ensure that the data is appropriately scaled.

Select the right set of attributes: The choice of attributes can significantly impact the accuracy of the model. Hence, it is essential to identify the most significant attributes that contribute to the price and eliminate any redundant attributes.

Optimize the algorithm: There are various machine learning algorithms that can be applied to predict house prices. Hence, it is essential to select the most appropriate algorithm and optimize its parameters to achieve the desired level of accuracy.

Develop an intuitive user interface: Finally, it is crucial to develop an intuitive user interface that can help buyers, sellers, and real estate professionals understand the predicted prices easily. The interface should provide relevant information about the property, including the size, location, and amenities, and it should also allow users to input their preferences and obtain personalized predictions.

1.5. About the datasets

We used two datasets for our study.

1. train.csv : <https://www.kaggle.com/competitions/playground-series-s3e6>
2. ParisHousing.csv : <https://www.kaggle.com/datasets/mssmartypants/paris-housing-price-prediction>

train.csv: The train.csv dataset from the Kaggle Playground Series S3E6 competition contains information about various real estate properties in Paris. The dataset consists of 22,730 rows and 18 columns and composes of 16 features. The columns in the dataset include features such as the year of construction, floors, and total area of the property. Additionally, the dataset contains information on the location of the property. The target variable for this dataset is the property sale price, which we will be predicting. This variable is included in the dataset as the "price"

column. Overall, the train.csv dataset provides a rich set of features and information about real estate properties in Paris city, making it an interesting and challenging dataset for predictive analytics tasks such as regression analysis and machine learning.

ParisHousin.csv: The Paris Housing Price Prediction dataset available on Kaggle, provided by the user MSSmartypants, contains information about real estate properties in Paris, France. The dataset consists of 10,000 rows and 18 columns and composes 16 features out of which we use 15 for our study. The rest is same as train.csv dataset.

Both datasets have the same features, which are:

- *squareMeters

- *numberOfRooms

- *hasYard

- *hasPool

- *floors - number of floors

- *cityCode - zip code

- *cityPartRange - the higher the range, the more exclusive the neighbourhood is

- *numPrevOwners - number of prevoious owners

- *made - year

- *isNewBuilt

- *hasStormProtector

- *basement - basement square meters

- *attic - attic square meteres

- *garage - garage size

- *hasStorageRoom

- *hasGuestRoom - number of guest rooms

*price - predicted value

```
In [3]: original_data = pd.read_csv(r"D:\Deepak\major project\train.csv")
original_data_for_cor=original_data.drop(['price'],axis=1)
original_data.head()
```

Out[3]:

	id	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	at
0	0	34291	24	1	0	47	35693	2	1	2000	0	1	8	51
1	1	95145	60	0	1	60	34773	1	4	2000	0	1	729	44
2	2	92661	45	1	1	62	45457	4	8	2020	1	1	7473	89
3	3	97184	99	0	0	59	15113	1	1	2000	0	1	6424	85
4	4	61752	100	0	0	57	64245	8	4	2018	1	0	7151	27

Fig 1.5.1 : Viewing train.csv

```
In [3]: original_data = pd.read_csv(r"D:\Deepak\major project\train.csv")
original_data_for_cor=original_data.drop(['price'],axis=1)
original_data.head()
```

Out[3]:

	id	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	attic	garage	hasStorageRoom	hasGuestRoom	price
0	47	35693	2	1	2000	0	1	8	5196	369	0	3	3436795.2	
1	60	34773	1	4	2000	0	1	729	4496	277	0	6	9519958.0	
1	62	45457	4	8	2020	1	1	7473	8953	245	1	9	9276448.1	
0	59	15113	1	1	2000	0	1	6424	8522	256	1	9	9725732.2	
0	57	64245	8	4	2018	1	0	7151	2786	863	0	7	6181908.8	

Fig 1.5.2 : train.csv

```
In [2]: original_data = pd.read_csv(r"C:\Users\91995\Downloads\ParisHousing.csv")
original_data.head()
```

Out[2]:

	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	attic
0	75523	3	0	1	63	9373	3	8	2005	0	1	4313	9005
1	80771	39	1	1	98	39381	8	6	2015	1	0	3653	2436
2	55712	58	0	1	19	34457	6	8	2021	0	0	2937	8852
3	32316	47	0	0	6	27939	10	4	2012	0	1	659	7141
4	70429	19	1	1	90	38045	3	7	1990	1	0	8435	2429

Fig 1.5.3 : Viewing ParisHousing.csv

```
In [2]: original_data = pd.read_csv(r"C:\Users\91995\Downloads\ParisHousing.csv")
original_data.head()
```

```
Out[2]:
```

	id	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	attic	garage	hasStorageRoom	hasGuestRoom	price
1	63	9373	3	8	2005	0	1	4313	9005	956	0	7	7559081.5	
1	98	39381	8	6	2015	1	0	3653	2436	128	1	2	8085989.5	
1	19	34457	6	8	2021	0	0	2937	8852	135	1	9	5574642.1	
0	6	27939	10	4	2012	0	1	659	7141	359	0	3	3232561.2	
1	90	38045	3	7	1990	1	0	8435	2429	292	1	4	7055052.0	

Fig 1.5.4 : ParisHousing.csv

2. Literature Survey

2.1 House Price Prediction using Machine Learning

In recent years, house price prediction using machine learning has gained significant attention due to its potential applications in the real estate industry. Accurate house price prediction can help buyers and sellers make informed decisions, and it can also assist real estate professionals in setting the right prices for properties. Machine learning algorithms are well suited for this task as they can handle large and complex datasets, and they can learn the underlying patterns and relationships between the attributes and the target variable.

One of the most common datasets used for house price prediction is the Boston Housing Dataset, which contains information about 506 houses in Boston, Massachusetts. The dataset includes 14 attributes, including the crime rate, the average number of rooms per dwelling, and the distance to employment centers. Various machine learning algorithms such as linear regression, decision trees, and neural networks have been applied to this dataset, and the results have shown promising accuracy.

Another popular dataset is the California Housing Prices dataset, which contains information about housing prices in various districts in California. The dataset includes eight attributes, including the longitude, latitude, median age, and median income. Various machine learning algorithms such as support vector regression, random forests, and XGBoost have been applied to this dataset, and the results have shown high accuracy.

In addition to these datasets, other datasets such as the Melbourne Housing Market dataset and the New York City Airbnb dataset have also been used for house price prediction. These datasets contain information about housing prices in specific regions and can provide valuable insights into the factors that influence the price of a property in that region.

Overall, machine learning algorithms have shown promising results in predicting house prices, and the use of large and complex datasets can help improve the accuracy of the models. However, selecting the right set of attributes, eliminating outliers, and optimizing the algorithm's parameters are crucial steps in developing accurate models.

2.2 Developing an application using Flask

Flask is a lightweight web framework that is well suited for developing web applications using Python. Flask provides a range of features such as routing, templates, and session management, which can help developers build web applications quickly and efficiently. In recent years, Flask has become a popular choice for developing machine learning applications, as it can integrate seamlessly with popular machine learning libraries such as scikit-learn and TensorFlow.

To develop a house price prediction application using Flask, the first step is to preprocess the data and eliminate any outliers. One of the most commonly used techniques is the Inter Quartile Range (IQR), which helps identify any values that are significantly higher or lower than the median value. Once the outliers are removed, the data is scaled using techniques such as StandardScaler or MinMaxScaler, which help ensure that all attributes have a similar scale and prevent bias towards attributes with larger values.

Next, the data is passed through the machine learning models, which can include algorithms such as linear regression, decision trees, random forests, support vector regression, gradient boosting, and XGBoost. The accuracy and runtime of each model are evaluated, and the best performing model is selected.

Finally, the Flask application is developed, which includes a user interface that allows users to input the relevant attributes and obtain a predicted house price. The user interface can be designed using HTML, CSS, and JavaScript, and it can also include features such as data visualization and personalized recommendations.

In conclusion, developing a house price prediction application using Flask can provide valuable insights into the factors that influence the price of a property, and it can help buyers, sellers, and real estate professionals make informed decisions. By leveraging the power of machine learning algorithms and the simplicity of Flask, developers can build robust and efficient applications that can handle large and complex datasets.

2.3 Popular regression models for price prediction

2.3.1. Linear Regression: Linear regression is a simple and widely used regression model that predicts the value of a continuous variable based on one or more predictor variables. It presupposes that the predictor variables and the response variable have a linear relationship. Both simple and numerous linear regression issues can be solved using linear regression. Simple linear regression involves only one predictor variable, while multiple linear regression involves two or more predictor variables.

2.3.2. Polynomial Regression: Polynomial regression is an extension of linear regression that models the relationship between the response variable and the predictor variables as an n th degree polynomial. It can capture non-linear relationships between the variables. The non-linear correlations between the predictor factors and the response variable can be modelled using polynomial regression. However, as the degree of the polynomial increases, the model can become overfit and perform poorly on new data.

2.3.3. Random Forest Regression: Random forest regression is an ensemble regression model that combines multiple decision trees to predict the response variable. It works by randomly selecting a subset of features and data points for each tree and averaging their predictions. Random forest regression is often more accurate than a single decision tree and can handle high-dimensional datasets with many features.

2.3.4. Multi-Level Perceptron Regressor: MLP (Multi-Layer Perceptron) regressor is a type of neural network model that can perform regression tasks. It consists of multiple layers of interconnected neurons that can learn non-linear relationships between the input variables and

the response variable. MLP regressor is a flexible model that can be used for a wide range of regression problems, but it requires careful tuning of hyperparameters and can be computationally expensive for large datasets.

2.3.5. XG Boost Regression: XGBoost (Extreme Gradient Boosting) regression is an ensemble regression model that uses a gradient boosting algorithm to combine multiple weak learners to make accurate predictions. It works by iteratively adding decision trees that correct the errors of the previous trees, leading to improved accuracy. XGBoost regression is often used for large-scale regression problems with many features and can be highly accurate, but it also requires careful tuning of hyperparameters and can be computationally expensive.

Each of these regression models has its own strengths and weaknesses, and the choice of model depends on the specific problem at hand and the nature of the data. It is important to carefully evaluate the performance of each model on the given data and choose the model that provides the best balance between accuracy and computational efficiency.

3. System Design

3.1. High Level Description

1. Data Collection: In this step, the ParisHousing dataset is collected from Kaggle. The dataset contains information on various attributes of houses in Paris, such as the number of rooms, location, and price. The dataset is downloaded and stored in a local folder or a cloud-based storage service such as Amazon S3.

2. Data Preprocessing: The next step is to preprocess the data to prepare it for machine learning. This involves several steps, including data cleaning, handling missing values, encoding categorical variables, and scaling the data. Data cleaning involves removing any irrelevant or duplicated data points from the dataset. Handling missing values involves filling in missing data with the mean, median, or mode of the feature. Categorical variables are encoded using techniques such as one-hot encoding or label encoding. Scaling the data involves bringing all features onto the same scale by using techniques such as MinMaxScaler or StandardScaler.

In addition, the data is analyzed for outliers using methods such as the interquartile range (IQR). The IQR method involves identifying the middle 50% of the data points and removing any data points that fall outside of this range. This is done to improve the accuracy of the machine learning models.

3. Model Development: The preprocessed data is then passed to six different regression models, including linear, polynomial, K-nearest neighbor, random forest, MLP regressor, and XGBoost regression. Each model is trained on the preprocessed data using techniques such as cross-validation and hyperparameter tuning. The performance of each model is evaluated using metrics such as mean squared error (MSE), mean absolute error (MAE), and R-squared (R2) to determine the best model for predicting house prices.

4. Flask Application: Once the best model is identified, it is integrated into a Flask web application. The Flask application is built using Python and Flask, a micro web framework. The

Flask application consists of a server, routing logic, and HTML templates. The user can input the relevant attributes of the house, such as the number of rooms and location, and the application will use the trained model to predict the price of the house. The predicted price is then displayed on the web page.

5. Deployment: Finally, the application is deployed to a production environment where it can be accessed by users. This involves hosting the Flask application on a server, setting up a domain name, and configuring security measures to protect the user's data. The Flask application can be hosted on a cloud-based platform such as AWS Elastic Beanstalk or Google App Engine.

By following this system design, users can accurately predict the price of a house in Paris based on various attributes, helping them make informed decisions when buying or selling a property.

3.2. Process Flow

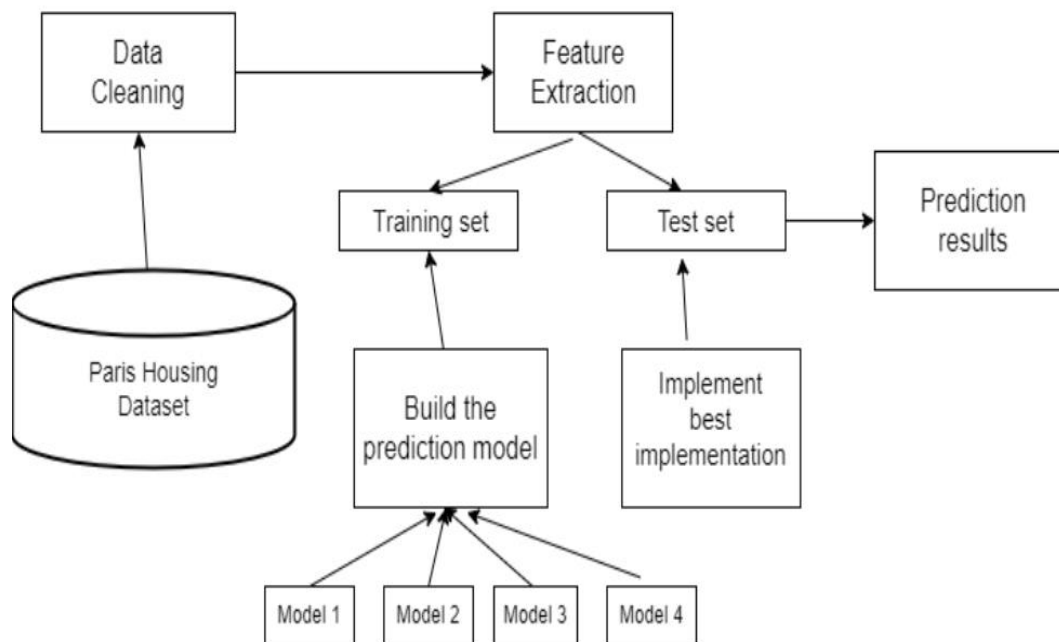


Fig 3.2.1: Process Flow

Here's a more detailed process flow for the house price prediction model:

Data Acquisition and Preprocessing: The first step in building a machine learning model is to acquire and preprocess the data. In this case, we used the ParisHousing dataset from Kaggle, which contains information on various attributes of houses in Paris such as the number of rooms, location, and other features. The data was preprocessed to remove any missing values, duplicates, or irrelevant features.

Outlier Detection and Removal: After the data is preprocessed, the next step is to detect and remove any outliers. In our model, we used the Interquartile Range (IQR) method to identify and eliminate outliers. This is important because outliers can have a significant impact on the accuracy of the model.

Feature Scaling: Once the outliers are removed, the data is scaled to ensure that all features are on a similar scale. This is done using the StandardScaler module from the Scikit-learn library.

Splitting the Data: The next step is to split the data into training and testing datasets. This is done to ensure that the model is trained on a subset of the data and tested on another subset. The training dataset is used to train the model, while the testing dataset is used to evaluate the model's accuracy.

Model Selection: The next step is to select the best regression model for the problem at hand. In your case, we used six different regression models: Linear Regression, Polynomial Regression, K-Nearest Neighbor Regression, Random Forest Regression, MLP Regressor, and XGBoost Regressor.

Model Training: Once the best model is selected, the next step is to train the model using the training dataset. This involves fitting the model to the training data and finding the optimal values for the model's parameters.

Model Evaluation: After the model is trained, the next step is to evaluate its performance on the testing dataset. This is done by predicting the house prices in the testing dataset and comparing them to the actual prices. The performance is measured using various metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-Squared (R^2).

Hyperparameter Tuning: After evaluating the model, the next step is to fine-tune the model's hyperparameters. This involves tweaking the model's parameters to improve its performance on the testing dataset. This step is important because it helps to prevent overfitting or underfitting of the model.

Model Deployment: Once the model is trained and tuned, the final step is to deploy the model into a production environment. In your case, you used Flask to build a web application and host it locally that allows users to input the attributes of a house and get a predicted price based on the model.

Overall, the process flow for the house price prediction model involves data acquisition and preprocessing, outlier detection and removal, feature scaling, data splitting, model selection, model training, model evaluation, hyperparameter tuning, and model deployment. By following this process flow, you were able to build an accurate and efficient model that can be used to predict house prices in Paris.

3.3. Assumptions

Assumption 1: The execution environment is consistent. Each time a model is run, the environment is the same.

Assumption 2: There are no processes running in the background which effect the running of the jupyter notebook.

Assumption 3: The attribute 'cityCode' does not contribute much to the prediction which led to not considering it as an attribute to pass to prediction models.

We personally oversaw that this attribute makes almost no difference in the predicted price. So, to reduce unwanted steps and increase the runtime, we did not consider this attribute as important and left it out of the selected features.

3.4. Software Requirements

Jupyter Notebook

OS

PyCharm for flask application

Browser

Command Prompt (as for windows OS)

3.5. Hardware Requirements

CPU: It is advised to use a multi-core processor with a minimum 2 GHz clock speed. By doing this, the server will be able to manage several requests at once.

RAM: Depending on the size of the database and the number of users who are concurrently visiting the website, a certain amount of RAM is needed. It is advised to have at least 4GB of RAM, but it is better to have 8GB or more.

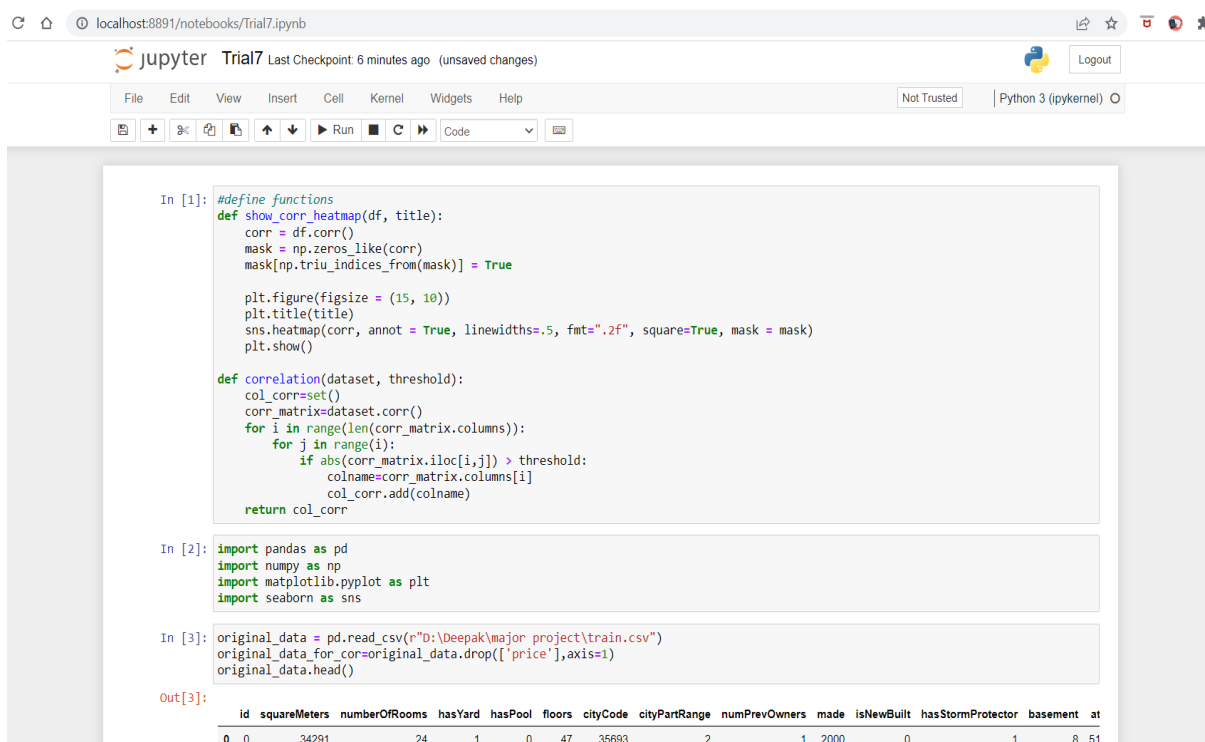
STORAGE: Depending on the size of the database, the size of any uploaded files, and the projected growth of the website, different amounts of storage are needed. Solid-state drives (SSDs) are suggested for faster access times.

NETWORK: To guarantee that customers may visit the website without any delay, the application should have a dependable and quick internet connection.

BACKUP: Backup systems are essential for preventing data loss due to server malfunctions or other problems. To store backups off-site, a backup solution that is external is advised.

4. Implementation

4.1. Implementing Machine learning code on Jupyter notebook



The screenshot shows a Jupyter Notebook interface with the following code:

```
In [1]: #define functions
def show_corr_heatmap(df, title):
    corr = df.corr()
    mask = np.zeros_like(corr)
    mask[np.triu_indices_from(mask)] = True

    plt.figure(figsize = (15, 10))
    plt.title(title)
    sns.heatmap(corr, annot = True, linewidths=.5, fmt=".2f", square=True, mask = mask)
    plt.show()

def correlation(dataset, threshold):
    col_corr=set()
    corr_matrix=dataset.corr()
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if abs(corr_matrix.iloc[i,j]) > threshold:
                colname=corr_matrix.columns[i]
                col_corr.add(colname)
    return col_corr

In [2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

In [3]: original_data = pd.read_csv(r"D:\Deepak\major project\train.csv")
original_data_for_cor=original_data.drop(['price'],axis=1)
original_data.head()

Out[3]:
```

	id	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	at
0	0	34291	24	1	0	47	35693	2	1	2000	0		1	8 51

Fig 4.1.1 : Define required functions, import libraries and load the dataframe. As you can see, there is a second dataframe ‘original_data_for_cor’ which we use for checking the correlation heatmap. The price attribute is dropped as it is not needed for the heatmap and would be somewhat disturbing the required heatmap which we need for feature selection and study.

```
In [3]: original_data = pd.read_csv(r"D:\Deepak\major project\train.csv")
original_data_for_cor=original_data.drop(['price'],axis=1)
original_data.head()

Out[3]:
```

	id	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	basement	at
0	0	34291	24	1	0	47	35693	2	1	2000	0	1	8	51
1	1	95145	60	0	1	60	34773	1	4	2000	0	1	729	44
2	2	92861	45	1	1	62	45457	4	8	2020	1	1	7473	86
3	3	97184	99	0	0	59	15113	1	1	2000	0	1	6424	85
4	4	61752	100	0	0	57	64245	8	4	2018	1	0	7151	27

Fig 4.1.2 : Viewing the dataframe. We are using the ParisHousing train dataframe from Kaggle.

It if of dimensions

```
In [4]: original_data_for_cor.corr()

Out[4]:
```

	id	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	isNewBuilt
id	1.000000	-0.001377	0.003697	-0.013849	-0.011452	-0.011556	0.007562	0.012823	-0.000870	0.004804	0.004701
squareMeters	-0.001377	1.000000	0.056755	-0.006512	-0.001615	0.017832	0.019843	0.002738	-0.000150	0.014827	0.010350
numberOfRooms	0.003697	0.056755	1.000000	-0.000989	0.000197	0.044159	-0.008527	0.012343	0.021984	0.004819	-0.008951
hasYard	-0.013849	-0.006512	-0.000989	1.000000	-0.068559	-0.010663	0.002786	-0.002071	-0.002494	-0.008165	0.001580
hasPool	-0.011452	-0.001615	0.000197	-0.068559	1.000000	-0.000490	0.008218	-0.002603	0.000512	-0.007610	0.032965
floors	-0.011556	0.017832	0.044159	-0.010663	-0.000490	1.000000	0.005899	0.000913	0.009200	0.004584	-0.005746
cityCode	0.007562	0.019843	-0.008527	0.002786	0.008218	0.005899	1.000000	0.009425	-0.003807	0.008044	-0.000551
cityPartRange	0.012823	0.002738	0.012343	-0.002071	-0.002603	0.000913	0.009425	1.000000	0.021518	0.005075	0.002154
numPrevOwners	-0.000870	-0.000150	0.021984	-0.002494	0.000512	0.009200	-0.003807	0.021518	1.000000	0.002074	-0.001010
made	0.004804	0.014827	0.004819	-0.008165	-0.007610	0.004584	0.008044	0.005075	0.002074	1.000000	-0.002023
isNewBuilt	0.004701	0.010350	-0.008951	0.001580	0.032965	-0.005746	-0.000551	0.002154	-0.001010	-0.002023	1.000000
hasStormProtector	0.000605	0.017886	0.011712	0.005141	0.014905	-0.002473	-0.003309	-0.000393	-0.013071	-0.001982	0.024039
basement	0.001734	-0.018948	0.023536	-0.011966	0.003251	0.007887	0.000686	0.000288	-0.005544	0.004274	-0.001705
attic	0.001661	-0.008333	0.027370	-0.003421	-0.001070	-0.006231	0.017507	0.010329	0.001098	0.010601	-0.000033
garage	0.010885	-0.063846	0.066873	-0.001047	0.007199	0.023794	0.007452	-0.001874	-0.004935	-0.006403	0.010843
hasStorageRoom	0.011835	0.006013	-0.007623	0.006715	0.016283	-0.007920	0.006606	0.003465	0.001031	-0.008561	0.027609
hasGuestRoom	0.001892	-0.005814	-0.015969	-0.007774	-0.006831	-0.020811	0.002911	0.018328	0.013139	-0.009767	0.010474

Fig 4.1.3 : Table showing the correlation between attributes.

```
In [5]: #display(show_corr_heatmap(original_data, "correlation heatmap"))
plt.figure(figsize=(15,10))
cor=original_data_for_cor.corr()
sns.heatmap(cor,annot=True,cmap=plt.cm.CMRmap_r)
plt.show

Out[5]: <function matplotlib.pyplot.show(close=None, block=None)>
```

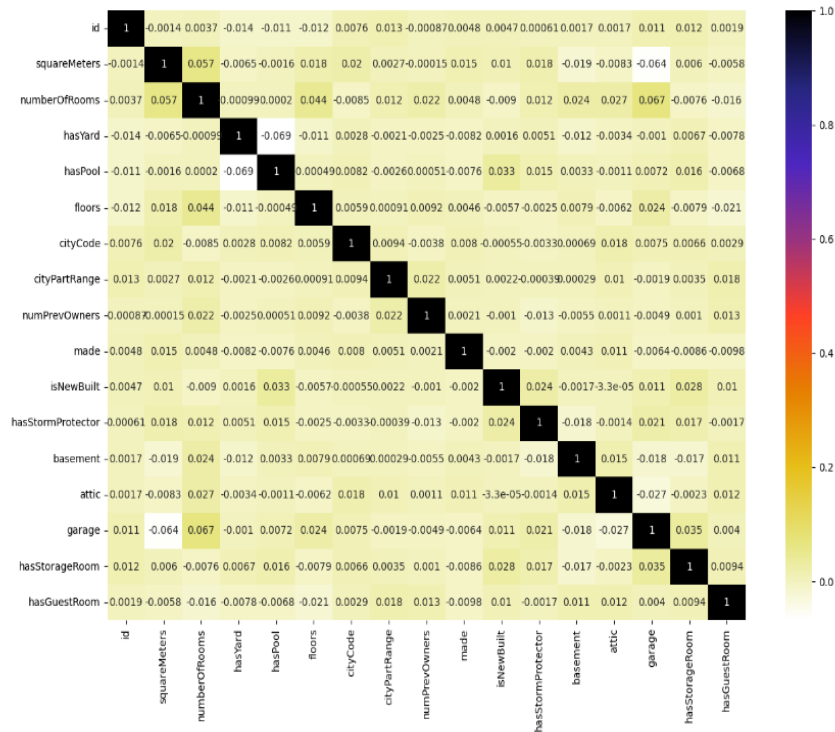


Fig 4.1.4 : The correlation heatmap which the Pearson correlation principle.

```
In [6]: corr_features=correlation(original_data_for_cor, 0.85)
len(set(corr_features))
```

Out[6]: 0

```
In [7]: corr_features
```

Out[7]: set()

Fig 4.1.5 : Using the corr_features functions we defined initially, we check if any features are highly correlated and eliminate them. As we can see, there are no such features.

```
In [8]: original_data["ActualArea"] = original_data['squareMeters'] - original_data['basement'] - original_data['attic'] - original_data['garage']
original_data["AreaPerHouse"] = original_data['squareMeters'] / original_data['numberOfRooms']
original_data["AreaPerRoom"] = original_data["ActualArea"] / original_data['numberOfRooms']
original_data["Extra"] = original_data["AreaPerHouse"] - original_data["AreaPerRoom"]
original_data.head()
```

Out[8]:

tRange	numPrevOwners	made	...	basement	attic	garage	hasStorageRoom	hasGuestRoom	price	ActualArea	AreaPerHouse	AreaPerRoom	Extra
2	1	2000	...	8	5196	369	0	3	3436795.2	28718	1428.791667	1196.583333	232.208333
1	4	2000	...	729	4496	277	0	6	9519958.0	89643	1585.750000	1494.050000	91.700000
4	8	2020	...	7473	8953	245	1	9	9276448.1	75990	2059.133333	1688.666667	370.466667
1	1	2000	...	6424	8522	256	1	9	9725732.2	81982	981.656566	828.101010	153.555556
8	4	2018	...	7151	2786	863	0	7	6181908.8	50952	617.520000	509.520000	108.000000

Fig 4.1.6 : We devised few derived features from the existing features. Which are then appended to the dataframe.

```
In [10]: def outliers(df, var):
Q1 = df[var].quantile(0.25)
Q3 = df[var].quantile(0.75)
IQR = Q3 - Q1
O1 = Q1 - (1.5*IQR)
O3 = Q3 + (1.5*IQR)
df.loc[(df[var] <= O1), var] = O1
df.loc[(df[var] >= O3), var] = O3
return df[var]
```

Fig 4.1.7 : Function to detect outliers using inter-quartile range.

```
In [11]: original_data['squareMeters'] = outliers(original_data, 'squareMeters')
original_data['squareMeters'].describe()
```

```
Out[11]: count    22730.000000
         mean     46325.635300
         std      29253.554766
         min       89.000000
         25%      20392.750000
         50%      44484.000000
         75%      71547.000000
         max     148278.375000
         Name: squareMeters, dtype: float64
```

```
In [12]: import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 7))
plt.boxplot(original_data.squareMeters)
plt.show()
```

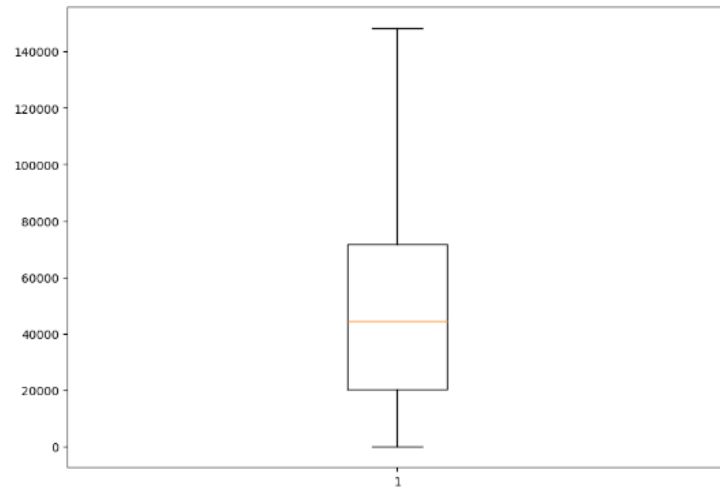


Fig 4.1.8 : Studying the 'squareMeters' attribute and checking and treating outliers.


```
In [13]: original_data['numberOfRooms'] = outliers(original_data, 'numberOfRooms')
original_data['numberOfRooms'].describe()
```

```
Out[13]: count    22730.000000
         mean      48.241091
         std       28.226428
         min        1.000000
         25%       25.000000
         50%       47.000000
         75%       75.000000
         max      100.000000
         Name: numberOfRooms, dtype: float64
```

```
In [14]: fig = plt.figure(figsize=(10, 7))
plt.boxplot(original_data.numberOfRooms)
plt.show()
```

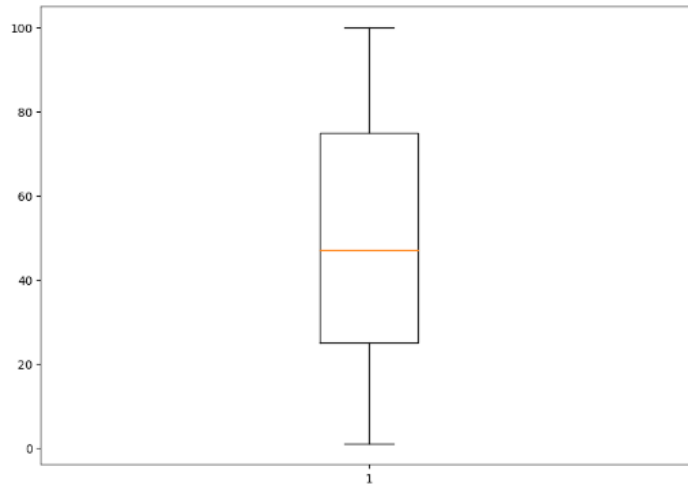


Fig 4.1.9 : Studying the 'numberOfRooms' attribute, checking and treating outliers.

```
In [15]: original_data['hasYard'].value_counts()
```

```
Out[15]: 0    11913
         1    10817
         Name: hasYard, dtype: int64
```

```
In [16]: original_data['hasPool'].value_counts()
```

```
Out[16]: 0    12439
         1    10291
         Name: hasPool, dtype: int64
```

Fig 4.1.10 : The attributes 'hasYard' and 'hasPool' have only two values 0 and 1. If the sum of entries having values 0 and 1 is equal to the size of the dataframe, we can conclude that there are no outliers. The sum is indeed equal to 22730 in both the cases.

```
In [17]: original_data['#floors'] = outliers(original_data, '#floors')
original_data['#floors'].describe()
```

```
Out[17]: count    22730.000000
         mean      47.047954
         std       26.905963
         min        1.000000
         25%       25.000000
         50%       45.000000
         75%       69.000000
         max      135.000000
         Name: floors, dtype: float64
```

```
In [18]: fig = plt.figure(figsize=(10, 7))
         plt.boxplot(original_data.floors)
         plt.show()
```

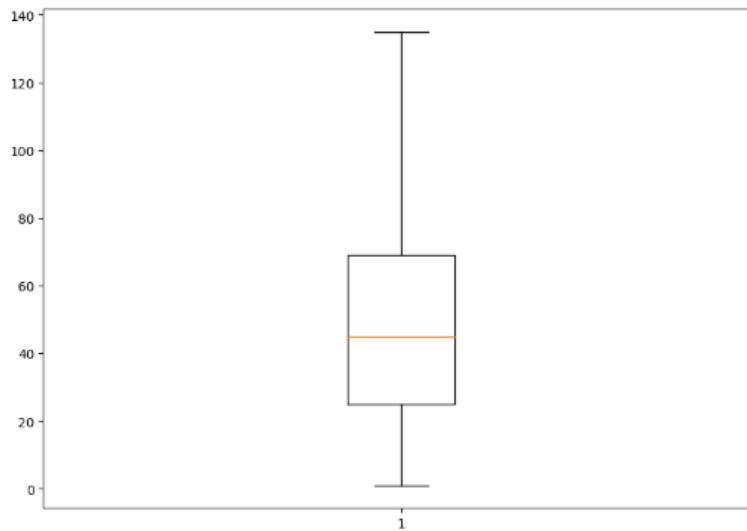


Fig 4.1.11 : Studying the 'floors' attribute, checking and treating outliers.

```
In [19]: original_data['cityCode'] = outliers(original_data, 'cityCode')
original_data['cityCode'].describe()
```

```
Out[19]: count    22730.000000
mean    49977.633744
std     29719.118134
min       3.000000
25%    22936.000000
50%    50414.000000
75%    76291.000000
max    156323.500000
Name: cityCode, dtype: float64
```

```
In [20]: original_data['cityCode'].value_counts()
```

```
Out[20]: 1906.0      69
42634.0     35
42626.0     35
3545.0      32
426.0       31
..
91673.0      1
72462.0      1
41026.0      1
30045.0      1
50813.0      1
Name: cityCode, Length: 7806, dtype: int64
```

Fig 4.1.12 : Studying the 'cityCode' attribute.

```
Name: cityCode, Length: 7806, dtype: int64
```

```
In [21]: original_data['numPrevOwners'].value_counts()
```

```
Out[21]: 5      2868
8      2639
9      2458
7      2437
4      2376
6      2363
3      2065
2      1985
1      1790
10     1749
Name: numPrevOwners, dtype: int64
```

```
In [22]: original_data['made'].value_counts()
```

```
Out[22]: 2000     3588
2003     1156
2014     1070
2015     1036
2007     1021
2008      981
2009      972
2019      956
2013      934
2018      927
2006      921
2004      915
2016      910
1996      822
2005      764
2017      741
1998      719
1993      651
1995      631
1994      628
2020      595
1997      592
1999      588
2010      152
2021      110
1990       77
2011       68
2001       59
1992       59
2012       51
1991       30
10000       5
2002        1
Name: made, dtype: int64
```

Fig 4.1.13 : Studying the 'numPreviousOwners' attribute.

```

name, money, us_type, area

In [23]: original_data['HouseAge'] = 2023 - original_data['made']
original_data['HouseAge'].value_counts()

Out[23]: 23    3588
        20    1156
         9    1070
         8    1036
        16    1021
        15     981
        14     972
         4     956
        10     934
         5     927
        17     921
        19     915
         7     910
        27     822
        18     764
         6     741
        25     719
        30     651
        28     631
        29     628
         3     595
        26     592
        24     588
        13     152
         2     110
        33      77
        12      68
        22      59
        31      59
        11      51
        32      30
       -7977      5
        21       1
Name: HouseAge, dtype: int64

```

Fig 4.1.14 : ‘HouseAge’ is another feature we derived from existing features. It is derived from the ‘made’ feature which represents the year in which the house was constructed. ‘HouseAge’ gives us the age of the house which is quite crucial while predicting house price.

```

In [24]: original_data['isNewBuilt'].value_counts()

Out[24]: 0    12093
        1    10637
        Name: isNewBuilt, dtype: int64

In [25]: original_data['hasStormProtector'].value_counts()

Out[25]: 0    12274
        1    10456
        Name: hasStormProtector, dtype: int64

```

Fig 4.1.15 : The attributes ‘isNewBuilt’ and ‘hasStormProtector’ have only two values 0 and 1. If the sum of entries having values 0 and 1 is equal to the size of the dataframe, we can conclude that there are no outliers. The sum is indeed equal to 22730 in both the cases.

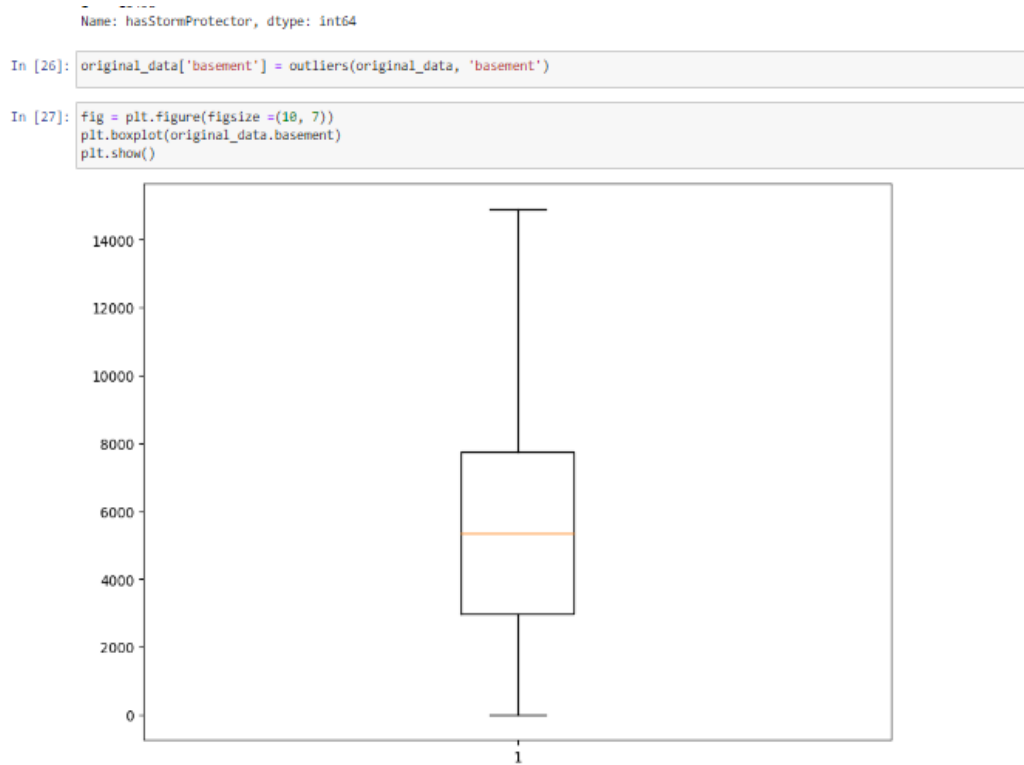


Fig 4.1.16 : Studying the 'basement' attribute, checking and treating outliers.

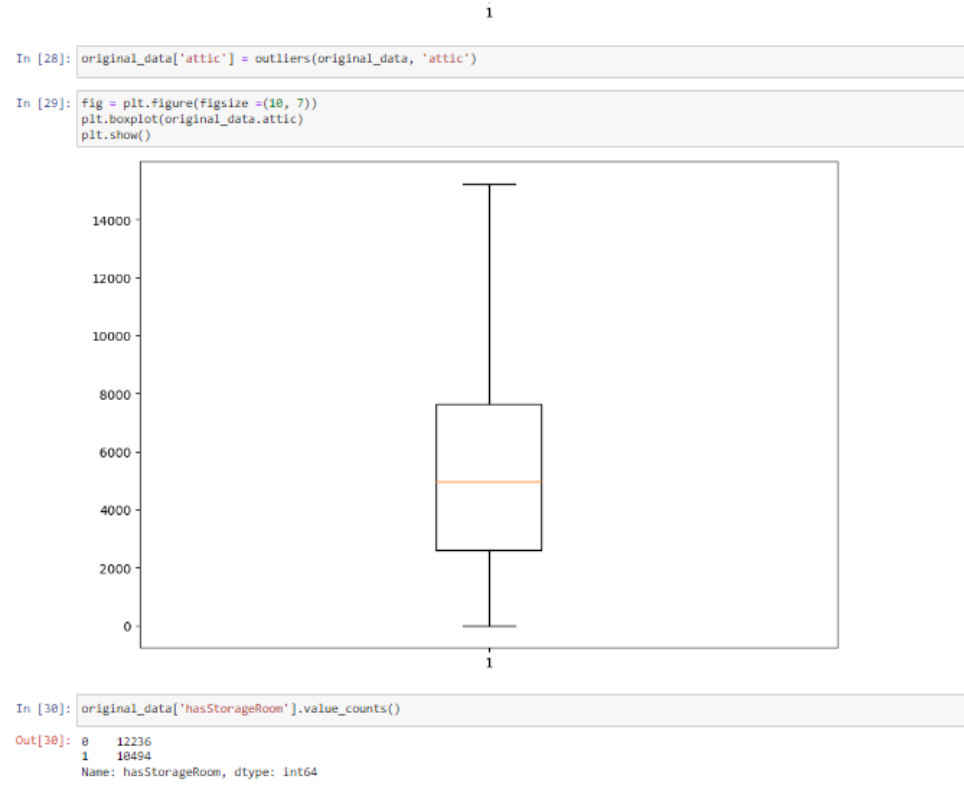


Fig 4.1.17 : Studying the 'attic' attribute, checking and treating outliers.

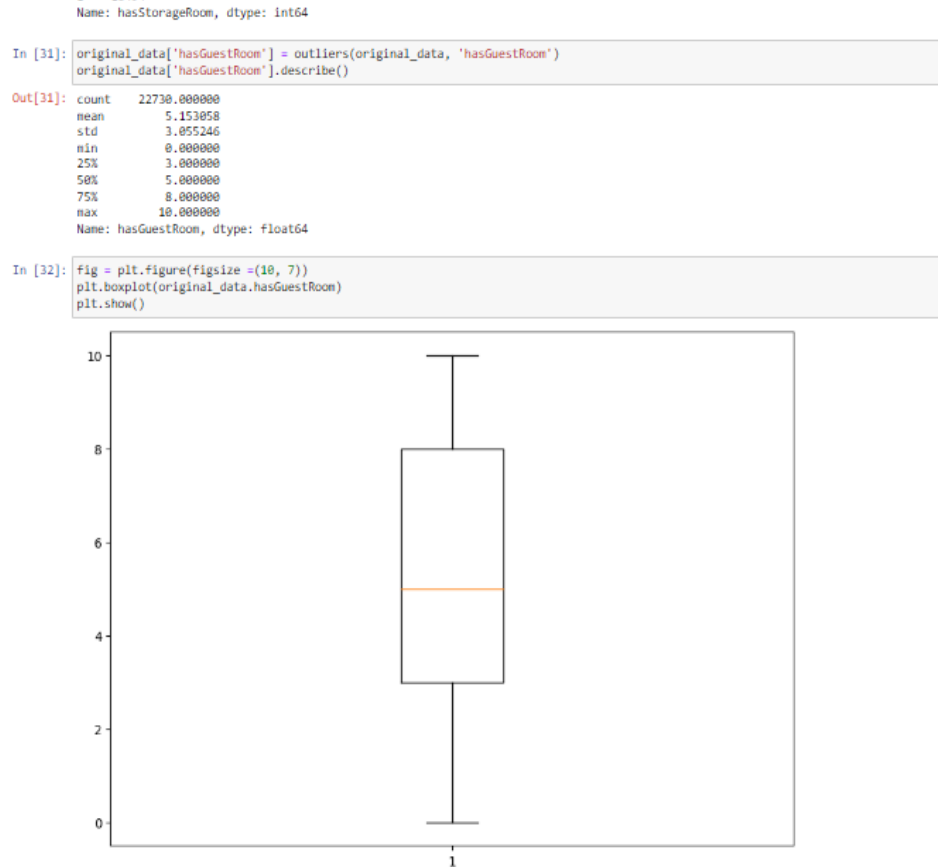


Fig 4.1.18 : Studying the ‘hasGuestRoom’ attribute, checking and treating outliers.



Fig 4.1.19 : Treating the ‘ActualArea’, ‘AreaPerHouse’, ‘AreaPerRoom’, and ‘Extra’ attributes for outliers.

In [38]:

```
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
d = scaler.fit_transform(original_data)
scaled_df = pd.DataFrame(d, columns=original_data.columns)
```

In [39]:

```
scaled_df
```

Out[39]:

	Id	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOwners	made	...	attic	garage
0	1.731975	-0.411400	-0.858827	1.049439	-0.909570	-0.001782	-0.480665	-1.308661	-1.703215	-0.067369	...	0.049157	-0.587516
1	1.731822	1.668672	0.416601	-0.952890	1.099421	0.481393	-0.511622	-1.673695	-0.597415	-0.067369	...	-0.191901	-0.922263
2	1.731670	1.583957	-0.114827	1.049439	1.099421	0.555727	-0.152115	-0.578594	0.876986	0.100947	...	1.342950	-1.036696
3	1.731517	1.738574	1.798316	-0.952890	-0.909570	0.444225	-1.173164	-1.673695	-1.703215	-0.067369	...	1.194527	-0.996672
4	1.731365	0.527345	1.833744	-0.952890	-0.909570	0.369891	0.480084	0.881541	-0.597415	0.084115	...	-0.780771	1.209925
...
22725	1.731365	0.324732	1.266887	1.049439	-0.909570	0.853066	-1.276871	-0.943627	1.614187	-0.067369	...	-1.469508	-0.674841
22726	1.731517	0.868117	1.408601	1.049439	-0.909570	0.072552	-0.901145	1.246575	1.245587	0.058868	...	-0.895101	0.816986
22727	1.731670	1.602109	-0.221113	1.049439	-0.909570	-0.299121	-1.394373	1.611609	-0.228814	0.050452	...	-0.337913	0.940676
22728	1.731822	0.665621	1.337744	1.049439	-0.909570	1.559244	-0.901145	-1.308661	1.614187	-0.067369	...	-0.874784	-0.434697
22729	1.731975	1.227856	-1.071399	1.049439	-0.909570	-0.336288	1.238617	-1.673695	1.245587	0.084115	...	-1.295602	0.151110

22730 rows x 23 columns

Fig 4.1.20 : Scaling the dataframe.

4.2. Implementing individual prediction models.

4.2.1. Linear Regression

```
In [40]: #model-1: Linear regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from time import time

In [44]: t0 = time()
X = scaled_df[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', 'isNew
y = scaled_df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,)
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print('run time: ', time() - t0)

run time: 0.01595783233642578

In [45]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.1464851714149713

In [46]: import sklearn.metrics as sm
print("r2 score:", (sm.r2_score(y_test, y_pred)))

r2 score: 0.9788065977971141
```

Fig 4.2.1.1 : Fitting the linear regression model using scikit learn library function.

```
In [76]: plt.scatter(y_test, y_test, color='blue', label='Actual Prices')
plt.scatter(y_test, y_pred, color='red', label='Predicted Prices')
plt.xlabel('House Prices')
plt.ylabel('House Prices')
plt.title('Actual vs Predicted House Prices | Linear regression')
plt.legend()
plt.show()
```



Fig 4.2.1.2 : Scatter plot for actual vs predicted values.

4.2.2. K-Nearest Neighbour Regression

```
In [48]: #model-2:k-nearest neighbours
from sklearn.neighbors import KNeighborsRegressor

In [49]: t0 = time()
X = scaled_df[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', 'isNew
y = scaled_df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
model = KNeighborsRegressor(n_neighbors=5)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print('run time: ', time() - t0)

run time: 1.613837718963623

In [50]: print("r2 score:", (sm.r2_score(y_test, y_pred)))

r2 score: 0.892845465705135

In [51]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.32739989414830874
```

Fig 4.2.2.1 : Fitting the KNN regression model using scikit learn library function.

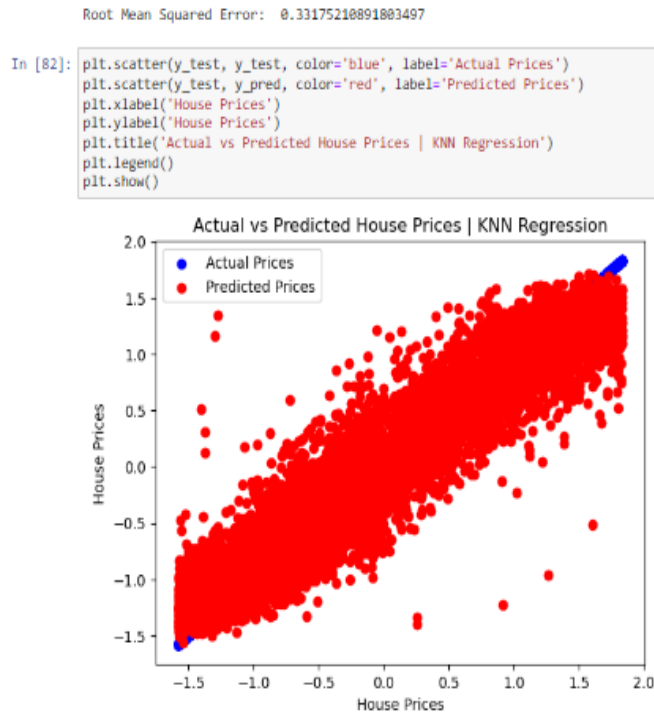


Fig 4.2.2.2 : Scatter plot for actual vs predicted values.

4.2.3. Random Forest Regression

```
In [53]: #model-3:Random forest regression
from sklearn.ensemble import RandomForestRegressor

In [54]: t0 = time()
X = scaled_df[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', 'isNew']]
y = scaled_df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print('run time: ', time() - t0)

run time: 6.806654453277588

In [55]: print("r2 score:", (sm.r2_score(y_test, y_pred)))

r2 score: 0.9941273704815792

In [56]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.07623369546411216
```

Fig 4.2.3.1 : Fitting the Random forest regression model using scikit learn library function.

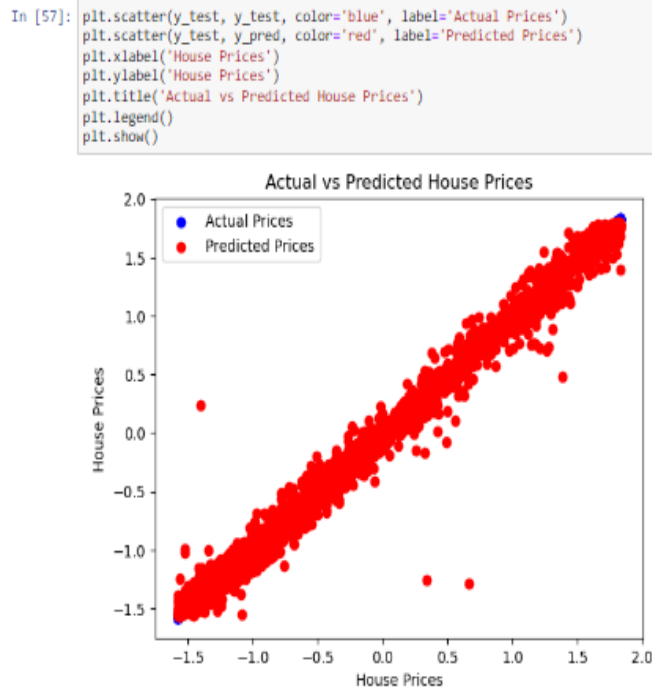


Fig 4.2.3.2 : Scatter plot for actual vs predicted values.

4.2.4. Polynomial Regression

```
In [58]: #model-4:polynomial regression
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

In [59]: t0 = time()
X = original_data[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', '1
y = original_data['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(X)
selected_model = LinearRegression()
selected_model.fit(X_poly, y)
y_pred = selected_model.predict(X_poly)
print('run time: ', time() - t0)

run time: 0.8641929626464844

In [60]: from sklearn.metrics import r2_score
r2 = r2_score(y, y_pred)
print('r2 score:', r2)

r2 score: 0.9874873806727597

In [75]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.1021091112612635
```

Fig 4.2.4.1 : Fitting the polynomial regression model using scikit learn library function.

4.2.5. Multi-Layer Perceptron Regressor

```
In [65]: #model-5:Multi layer perceptron regression
from sklearn.neural_network import MLPRegressor

In [66]: t0 = time()
X = scaled_df[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', 'isNew
y = scaled_df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
model = MLPRegressor(hidden_layer_sizes=(32, 16, 8), max_iter=1500, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print('run time: ', time() - t0)

run time: 2.9367098888288574

In [67]: score = model.score(X_test, y_test)
print('r2 score: {score}')

r2 score: 0.987967999266776

In [68]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print('Root Mean Squared Error: ', rmse)

Root Mean Squared Error: 0.10845361882101955
```

Fig 4.2.5.1 : Fitting the MLP Regressor model using scikit learn library function.

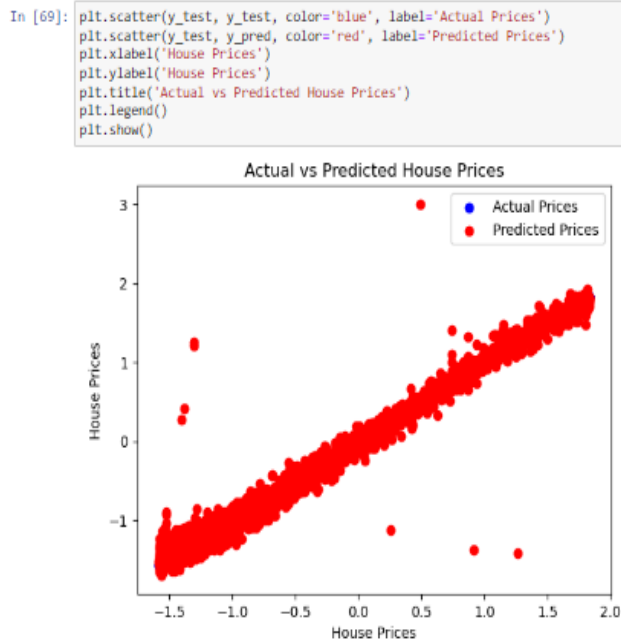


Fig 4.2.5.2 : Scatter plot for actual vs predicted values.

4.2.6. Extra Gradient Boost Regression

```
In [70]: #model-6: XG boost regression
import xgboost
from xgboost import XGBRegressor

In [71]: t0 = time()
X = scaled_df[['ActualArea', 'AreaPerRoom', 'Extra', 'HouseAge', 'hasYard', 'hasPool', 'floors', 'cityPartRange', 'numPrevOwners', 'isNew
y = scaled_df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
use_model = XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=3)
use_model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print('run time: ', time() - t0)

run time: 0.5684747695922852

In [72]: print("r2 score:", sm.r2_score(y_test, y_pred))

r2 score: 0.9895170392999603

In [73]: rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.1021091112612635
```

Fig 4.2.6.1 : Fitting the XG Boost model using scikit learn library function.

```
print("Root Mean Squared Error: ", rmse)

Root Mean Squared Error: 0.1021091112612635

In [74]: plt.scatter(y_test, y_test, color='blue', label='Actual Prices')
plt.scatter(y_test, y_pred, color='red', label='Predicted Prices')
plt.xlabel('House Prices')
plt.ylabel('House Prices')
plt.title('Actual vs Predicted House Prices')
plt.legend()
plt.show()

Actual vs Predicted House Prices

In [ ]: import pickle
pickle.dump(use_model, open('use_model_lr.pkl', 'wb'))
```

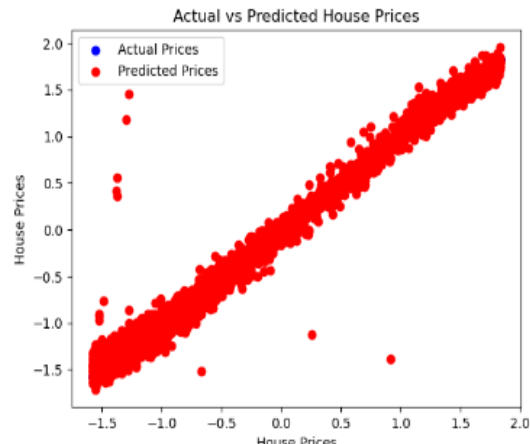


Fig 4.2.6.2 : Scatter plot for actual vs predicted values.

4.3. Implementing the whole application.

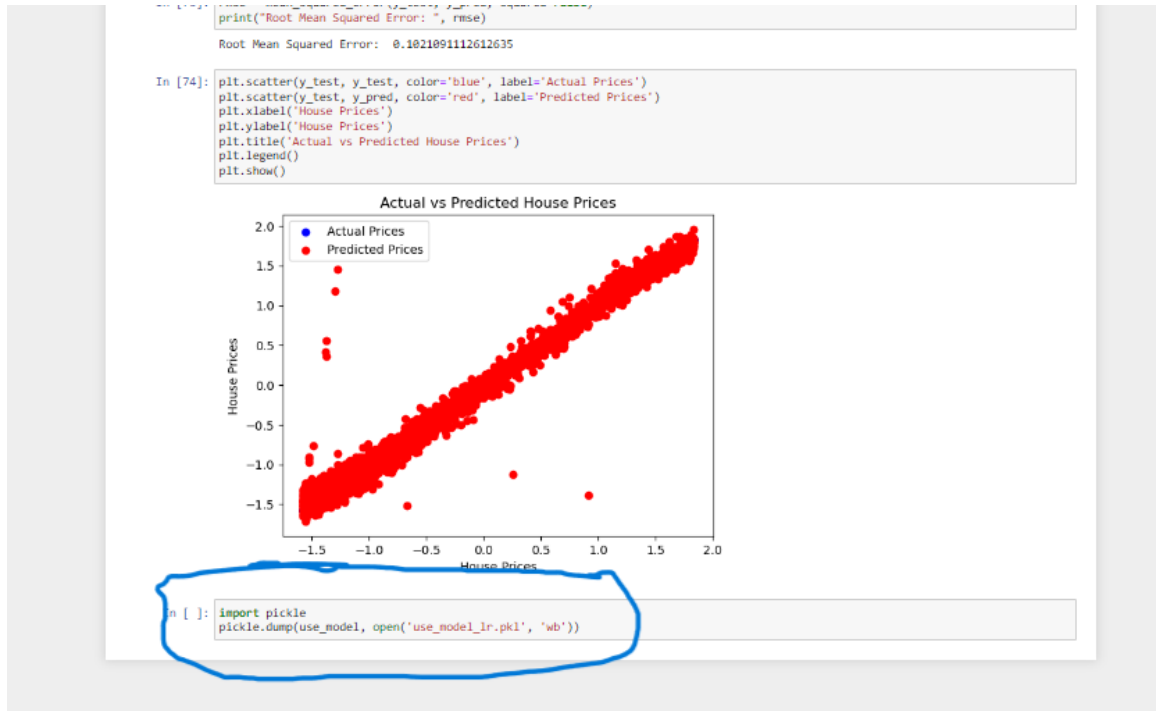
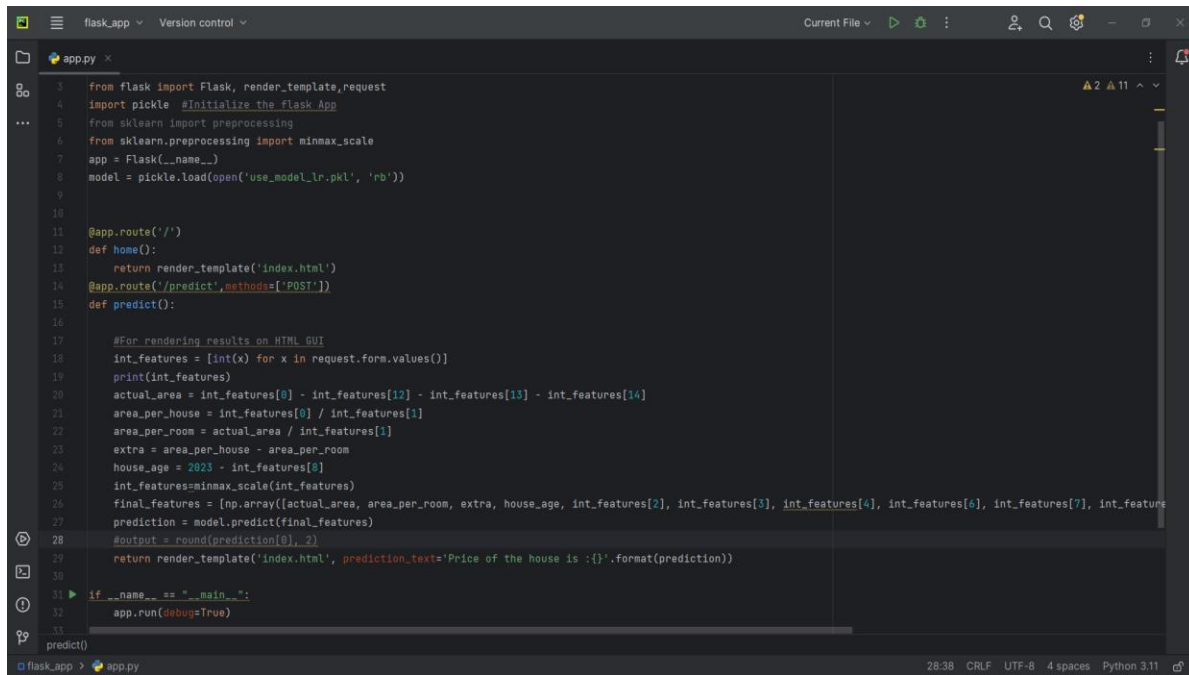


Fig 4.3.1 : Using the XG Boost model in our application.

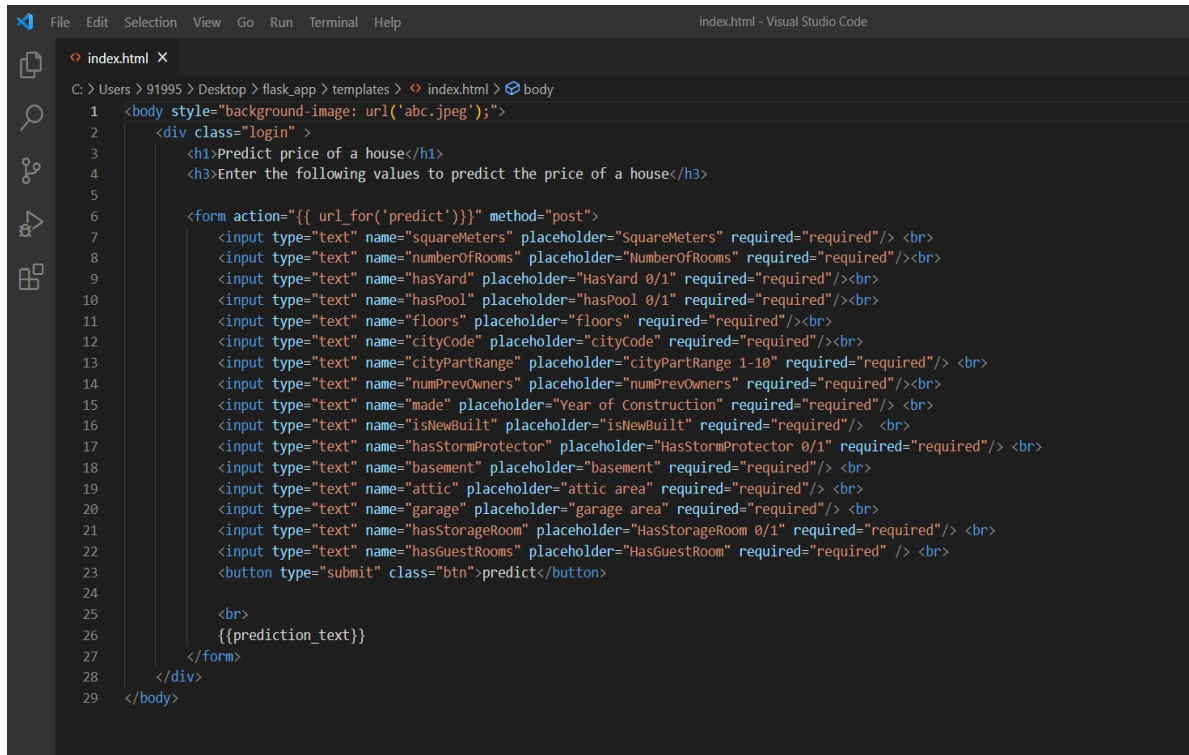
XG Boost comes out to be our best prediction model out of the 6 when compared in all aspects. We generate a pickle file which will be pasted in the flask application folder. This integrates the flask application to our XG Boost model.



```
1 from flask import Flask, render_template, request
2 import pickle #Initialize the Flask App
3 from sklearn import preprocessing
4 from sklearn.preprocessing import minmax_scale
5 app = Flask(__name__)
6 model = pickle.load(open('use_model_lr.pkl', 'rb'))
7
8
9
10
11 @app.route('/')
12 def home():
13     return render_template('index.html')
14 @app.route('/predict', methods=['POST'])
15 def predict():
16
17     #For rendering results on HTML GUI
18     int_features = [int(x) for x in request.form.values()]
19     print(int_features)
20     actual_area = int_features[0] - int_features[12] - int_features[13] - int_features[14]
21     area_per_house = int_features[0] / int_features[1]
22     area_per_room = actual_area / int_features[1]
23     extra = area_per_house - area_per_room
24     house_age = 2023 - int_features[5]
25     int_features=minmax_scale(int_features)
26     final_features = (np.array([actual_area, area_per_room, extra, house_age, int_features[2], int_features[3], int_features[4], int_features[6], int_features[7], int_features[8], int_features[9], int_features[10], int_features[11]]))
27     prediction = model.predict(final_features)
28     #output = round(prediction[0], 2)
29     return render_template('index.html', prediction_text='Price of the house is :{}'.format(prediction))
30
31
32 if __name__ == '__main__':
33     app.run(debug=True)
34
35 predict()
```

Fig 4.3.2 : Flask application source code on Pycharm.

We can either this code through Pycharm or through out command prompt. When running on command prompt, it is better to run it through a virtual environment. The flask reads the values given by user through 'index.html', devises new features as we defined, then scales them and passes them to our model to get a predicted price.



```
1 <body style="background-image: url('abc.jpeg');">
2   <div class="login">
3     <h1>Predict price of a house</h1>
4     <h3>Enter the following values to predict the price of a house</h3>
5
6     <form action="{{ url_for('predict')}}" method="post">
7       <input type="text" name="squareMeters" placeholder="SquareMeters" required="required"/> <br>
8       <input type="text" name="numberOfRooms" placeholder="NumberOfRooms" required="required"/> <br>
9       <input type="text" name="hasYard" placeholder="HasYard 0/1" required="required"/> <br>
10      <input type="text" name="hasPool" placeholder="hasPool 0/1" required="required"/> <br>
11      <input type="text" name="floors" placeholder="floors" required="required"/> <br>
12      <input type="text" name="cityCode" placeholder="cityCode" required="required"/> <br>
13      <input type="text" name="cityPartRange" placeholder="cityPartRange 1-10" required="required"/> <br>
14      <input type="text" name="numPrevOwners" placeholder="numPrevOwners" required="required"/> <br>
15      <input type="text" name="made" placeholder="Year of Construction" required="required"/> <br>
16      <input type="text" name="isNewBuilt" placeholder="isNewBuilt" required="required"/> <br>
17      <input type="text" name="hasStormProtector" placeholder="HasStormProtector 0/1" required="required"/> <br>
18      <input type="text" name="basement" placeholder="basement" required="required"/> <br>
19      <input type="text" name="attic" placeholder="attic area" required="required"/> <br>
20      <input type="text" name="garage" placeholder="garage area" required="required"/> <br>
21      <input type="text" name="hasStorageRoom" placeholder="HasStorageRoom 0/1" required="required"/> <br>
22      <input type="text" name="hasGuestRooms" placeholder="HasGuestRoom" required="required" /> <br>
23      <button type="submit" class="btn">predict</button>
24
25      <br>
26      {{prediction_text}}
27    </form>
28  </div>
29 </body>
```

Fig 4.3.3: Source code of the form written in HTML.

This form reads the values of the attributes given by the user to pass it to our XG Boost model. It is integrated to our flask application. The form passes the values to the flask application.

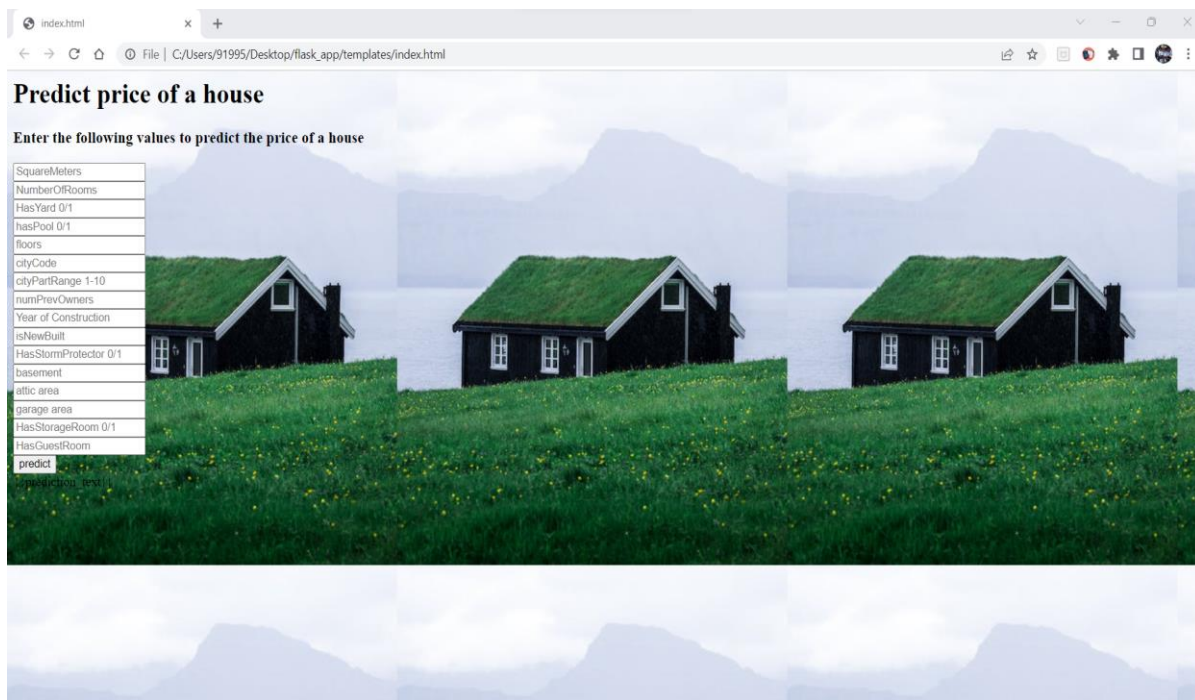
```
Command Prompt - python app.py
Microsoft Windows [Version 10.0.22000.1696]
(c) Microsoft Corporation. All rights reserved.

C:\Users\91995>cd Desktop
C:\Users\91995\Desktop>cd flask_app
C:\Users\91995\Desktop\flask_app>venv\Scripts\activate

(venv) C:\Users\91995\Desktop\flask_app>python app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 961-145-465
```

Fig 4.3.4 : Command prompt commands.

‘http://127.0.0.1:5000’ is our webpage which is being hosted locally.



Fig

4.3.5 : The ‘http://127.0.0.1:5000’ webpage.

5. Results and conclusions.

5.1. Testing the application

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	squareMet	numberOf	hasYard	hasPool	floors	cityCode	cityPartRa	numPrevO	made	isNewBuilt	hasStormP	basement	attic	garage	hasStorage	hasGuestR	price
2	75523	3	0	1	63	9373	3	8	2005	0	1	4313	9005	956	0	7	7559082

Fig 5.1.1 : One entry of the ‘ParisHousing’ dataset.

127.0.0.1:5000 x +

← → ↺ ⬆ 127.0.0.1:5000

Predict price of a house

Enter the following values to predict the price of a house

75523
3
0
1
63
9373
3
8
2005
0
1
4313
9005
956
0
7
predict

Fig 5.1.2 : Entering the values.

(Removed the background image so that it does not distract the displayed output.)

127.0.0.1:5000/predict

127.0.0.1:5000/predict

Predict price of a house

Enter the following values to predict the price of a house

SquareMeters
NumberOfRooms
HasYard 0/1
hasPool 0/1
floors
cityCode
cityPartRange 1-10
numPrevOwners
Year of Construction
isNewBuilt
HasStormProtector 0/1
basement
attic area
garage area
HasStorageRoom 0/1
HasGuestRoom
predict

Price of the house is :[7564230.21470546]

Fig 5.1.3 : Result

For the test run:

Actual value: 7559082.0

Predicted value: 7564230.2

5.2. Performance measures

5.2.1. Train data performance (22,730 entries dataframe)

Model	Execution time (average of 5 runs)	R2-score (average of 5 runs)	RMSE (average of 5 runs)
Linear Regression	0.28	0.9798	0.148
KNN Regression	1.0486	0.8892	0.334
Random forest	15.569	0.9916	0.081
Polynomial Regression	1.499	0.9874	0.102
MLP Regressor	2.195	0.9827	0.112
XG Boost	1.504	0.9885	0.109

Table 5.2.1.1---: Data when models were performed on training data which is of 22730 entries.

5.2.2. Test data performance (10,000 entries performance)

Model	Execution time (average of 5 runs)	R2-score (average of 5 runs)	RMSE (average of 5 runs)
Linear Regression	0.009	0.9864	0.160
KNN Regression	0.220	0.8916	0.389
Random forest	8.128	0.9947	0.090
Polynomial Regression	0.397	0.9948	0.113
MLP Regressor	1.520	0.9852	0.124
XG Boost	0.691	0.9935	0.120

Table 5.2.2.1---: Data when models were trained on the ‘ParisHousing’ dataset which is the test dataset for the study.

5.3. Conclusions

1. XG Boost is the best performing model out of the lot when we consider all the factors.
2. Random forest regression and polynomial regression give the best accuracy but they each have a drawback when compared to XG Boost.
3. Random forest regression takes a lot of time to execute. Rest of the models are way faster than random forest. When the sole priority is accuracy, we can prefer random forest regression. In our study, it gives an accuracy at about 99.1 in for train data and 99.4 for test data.
4. Polynomial regression on the other hand, does not give a good RMSE value. Ideal RMSE range is 0.2 to 0.7. Polynomial regression is giving an RMSE value of less than 0.1 in both test and train cases.
5. XG Boost performance is even better when the dataset size increases. It is given more data to train and hence performs better.
6. Linear Regression is also a good algorithm overall and is the fastest algorithm in our study. When the priority is speed, Linear Regression would be our first choice followed by Polynomial Regression.
7. KNN Regression gives the best RMSE value but does not give good accuracy.

6. Future work

1. We want to further our understanding to more algorithms such as Gradient Boost regression, Support Vector regression which are the new generation of regression models along with XG Boost.
2. Also, we want to study more complex data. When it comes to house price prediction, Boston housing and California house dataset from the Scikit learn library are popular datasets. But both these datasets have very less entries. Boston housing dataset has only 500 entries and has a smaller number of features. Same is the case with California house dataset. The ParisHousing dataset from Kaggle serves the purpose better. It has 10,000 entries and 15 features. Further, we hope future datasets come with more entries and more features to study.
3. Neural networks are known to handle complex data better than other algorithms. We want to test this when a dataset comes with good number of complex features.
4. We also want to further develop our application and host it on a cloud server. So, anyone can access it and can use it.
5. Not only house price, but these models can also be used to predict prices of other entities. A different subject may lead to different performance and results. Maybe a different algorithm might be a better fit for that subject.
6. We wish to deepen our knowledge in Machine Learning and apply more complex EDA and pre-processing to improve the quality of our data before passing it to the models. The RMSE values we got are a bit disappointing. We were only able to apply Inter Quartile Range to detect and treat outliers. Methods like Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) are becoming popular for dimensionality reduction. And there are different ways to detect and treat outliers such as Subspace Clustering and Auto encoder based methods which understand the data by studying it and identify the outliers which do not match the learned patterns.

7. References

- Predicting House Prices Using Linear Regression: An Application of Machine Learning and Artificial Intelligence Techniques by S. K. Singh and R. K. Tripathi (2020)
- House price prediction using multiple linear regression, support vector regression and artificial neural network by S. Haider, et al. (2019)
- A Comparative Study of Multiple Linear Regression and Support Vector Regression for House Price Prediction by M. Shetty and V. Bhat (2018)
- Predicting House Prices with k-Nearest Neighbors Regression by A. Chen and A. Zhang (2018)
- A Comparative Study of K-Nearest Neighbor and Artificial Neural Network in Predicting House Prices by R. Bhatia and S. Kumar (2020)
- House Price Prediction using Random Forest Regression by S. B. Olugbenga and O. M. Olaniyi (2020)
- Prediction of House Prices using Random Forest Regression by S. Lakshmi and S. Balamurugan (2021)
- House Price Prediction Using Polynomial Regression by S. Kumar and R. Bhatia (2020)
- House Price Prediction using Polynomial Regression by A. M. Ismail and S. F. Abdalla (2018)
- House Price Prediction Using MLP Regressor by N. P. Prasannakumar and S. Annapoorani (2020)
- House Price Prediction Using Multilayer Perceptron Regression by R. Malhotra, et al. (2021)

- House Price Prediction using XGBoost Algorithm by S. P. Patil, et al. (2021)
- XGBoost Regression for House Price Prediction by M. L. Mohamad, et al. (2019)
- Retail Sales Prediction Using Machine Learning Algorithms, by Dr. Bandaru Srinivasa Rao¹, Dr. Kamepalli Sujatha, Dr. Nannpaneni Chandra Sekhara Rao, Mr. T. Nagendra Kumar
- <https://www.analyticsvidhya.com/blog/2020/09/integrating-machine-learning-into-web-applications-with-flask/>
- https://www.w3schools.com/python/python_ml_scatterplot.asp
- <https://vitalflux.com/minmaxscaler-standardscaler-python-examples/>
- <https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02>
- <https://www.digitalocean.com/community/tutorials/standardscaler-function-in-python>
- <https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02>
- Principal Component Analysis (PCA) and Singular Value Decomposition (SVD)
- Harvard Business Review - Source: <https://hbr.org/>
- Predictive Analytics Times - Source: <https://www.predictiveanalyticsworld.com/patimes/>
- KDnuggets - Source: <https://www.kdnuggets.com/>
- Data Science Central - Source: <https://www.datasciencecentral.com/>
- Analytics Insight - Source: <https://www.analyticsinsight.net/>

- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
- <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html>
- <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html>
- <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html
- <https://xgboost.readthedocs.io/en/stable/>
- <https://scikit-learn.org/stable/modules/preprocessing.html>
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html