India Real Estate Appraisal using machine learning

A Preprint

Agata Kopyt
Faculty of Mathematics and Information Science
Warsaw University of Technology
Warsaw, Poland

Zuzanna Kotlińska
Faculty of Mathematics and Information Science
Warsaw University of Technology
Warsaw, Poland

Szymon Matuszewski
Faculty of Mathematics and Information Science
Warsaw University of Technology
Warsaw, Poland

Patryk Słowakiewicz
Faculty of Mathematics and Information Science
Warsaw University of Technology
Warsaw, Poland

May 11, 2022

Keywords Machine Learning · Real Estate Appraisal · India · Tree-Based Models

1 Data

The data we have analysed originates from https://www.kaggle.com/datasets/ruchi798/housing-prices-in-metropolitan-areas-of-india and concerns properties in six Indian cities – Bangalore, Chennai, Delhi, Kolkata, Hyderabad and Mumbai. It contains information about their prices, as well as other factors (location, area, number of bedrooms, among others) that affect them. The data set (all six data frames combined) consists of 32963 rows and 41 columns.

Firstly, we checked some basic information – the type of our data and the number of NaN values. The majority of the variables is numeric type, which is certainly conductive to further analysis. At first glance, we have encountered about 7000 of NaN values in all columns except for Price, Area, Location, No. of Bedrooms and Resale which have no NaNs. However, in the dataset description there is an information that such values are also marked as '9' so taking that into consideration it turns out that the majority of columns have as many as 22870 missing data. For continuous variables such as Price, Area and Location there are 4924, 2452 and 1776 unique records.

We looked at what the distributions of the continuous variables contained in our dataset, such as Price and Area, look like. As they appear to be exponential they may need to be transposed.

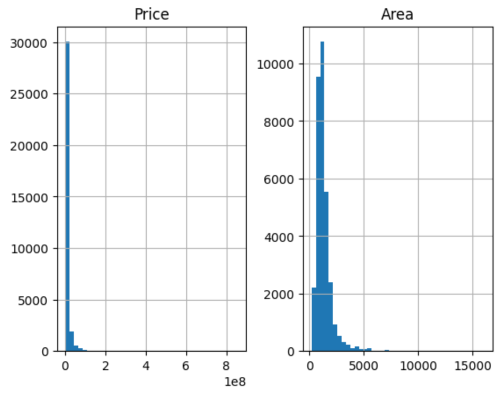


Figure 1: Price and Area distribution

We have also checked Cramer V correlation between our variables (firstly we have grouped Price and Area into categories and created PriceGroup and AreaGroup variables) and searched for strong correlations (>0.75) so we could drop redundant columns. There is of course a really strong correlation (nearly 1.00) between Location and City - but it is no surprise as every city has generally its unique district names. Of course in our analysis we want to concentrate mainly on price so we have checked correlations for PriceGroup but there don't seem to be any. Finally, we have checked some most important variable distributions and interesting relations between columns. Here are our conclusions:

- Top property locations in India are Noida (Delhi), New Town (Kolkata) and Khargar (Mumbai). It's worth noticing that Noida is a whole city (in our dataset Delhi is a union territory), so that's probably why it has the biggest number of properties comparing to other locations which are mainly just districts.
- The most common size of the property is about 1600 sq feet (150 sq metres).
- Most of the properties in India have 24/7 security.
- The biggest (and generally most expensive) properties are far from public facilities like shopping mall or hospital (as they are probably located on the suburbs, where plots are bigger).
- The majority of properties have no gas connection.
- Indias's most expensive areas are Sunder Nagar and Marsarovar garden (both in Delhi) and MG Road (Mumbai). The leader is Sunder Nagar with average house prices over 800000000.0 INR (10M EUR!). This small city is called the ultimate address of the uber-rich in Delhi.

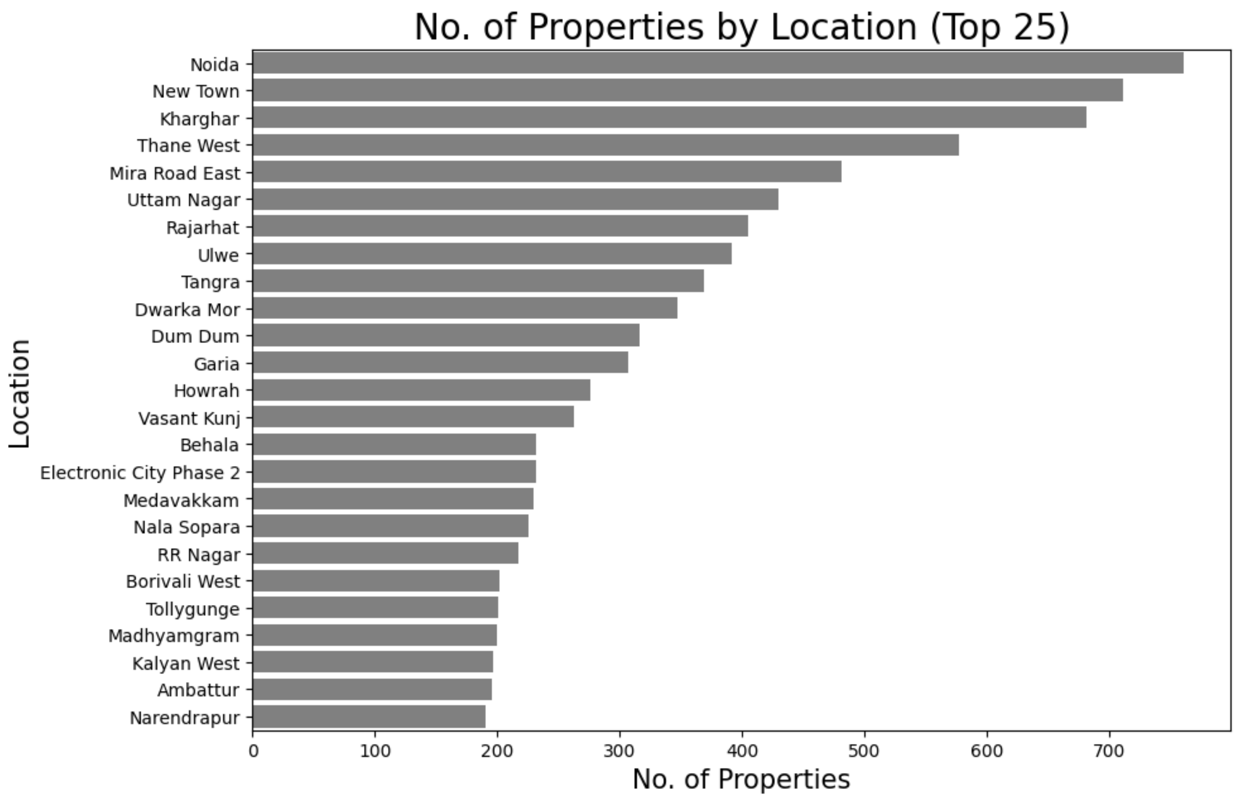


Figure 2: Top property Locations in India

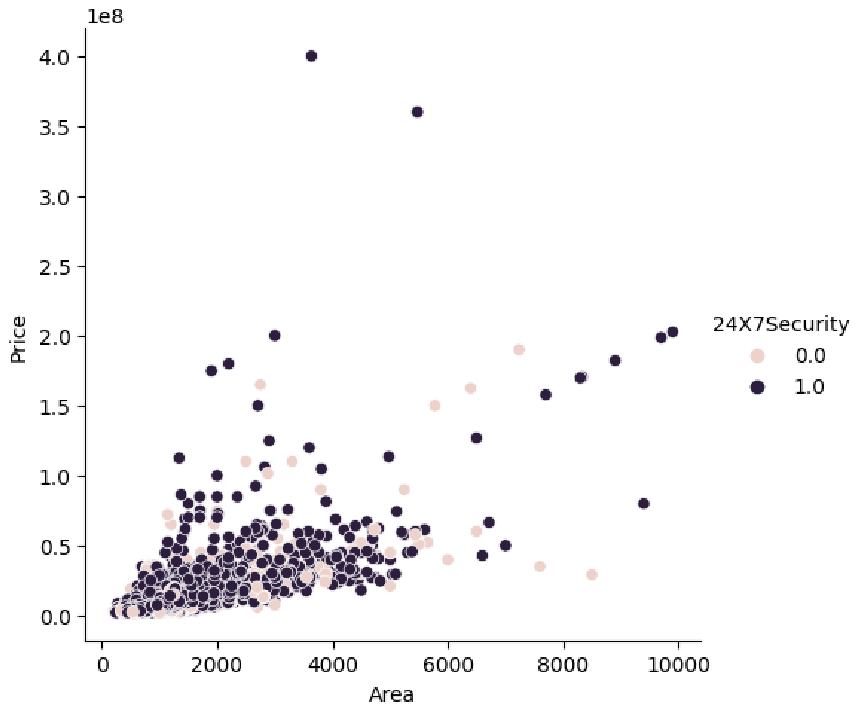


Figure 3: 24x7 security

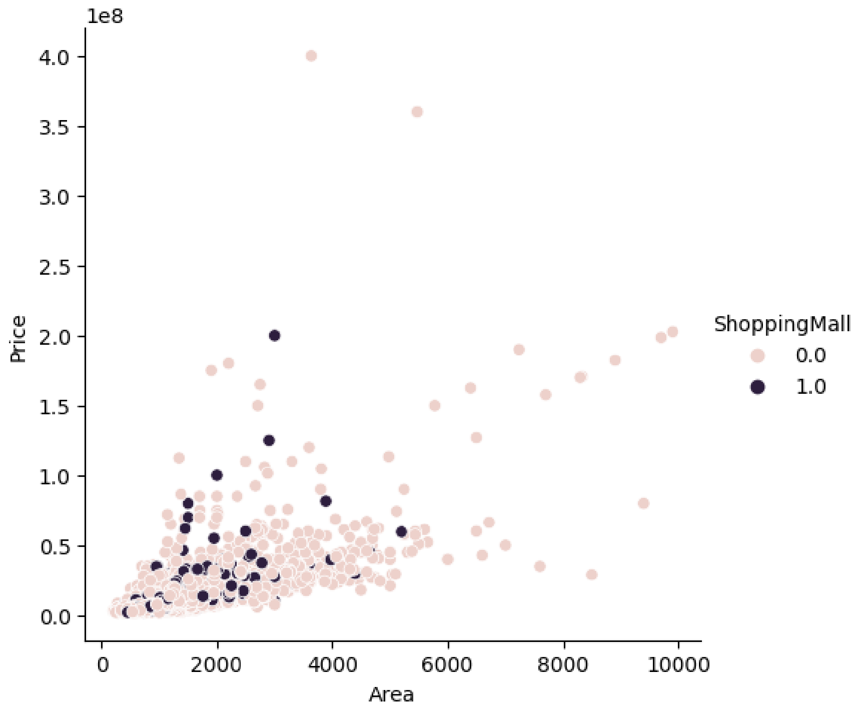


Figure 4: Shopping mall access

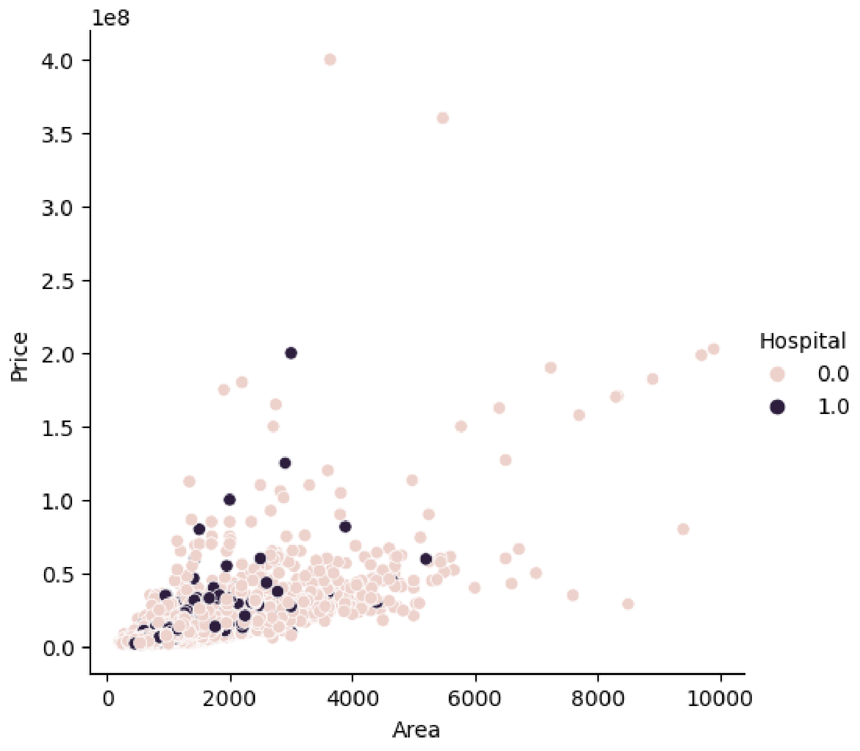


Figure 5: Hospital access

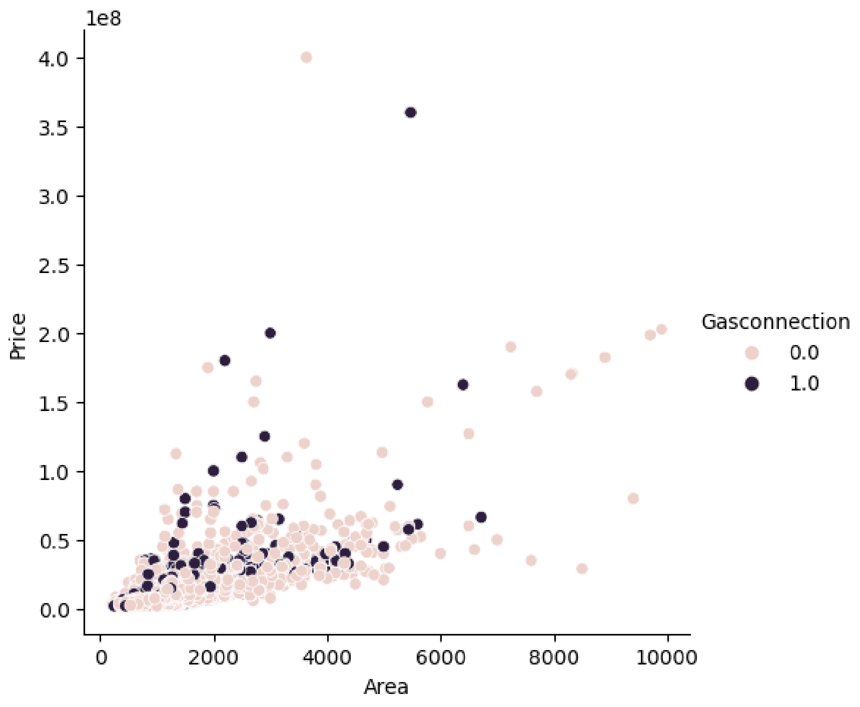


Figure 6: Gas connection

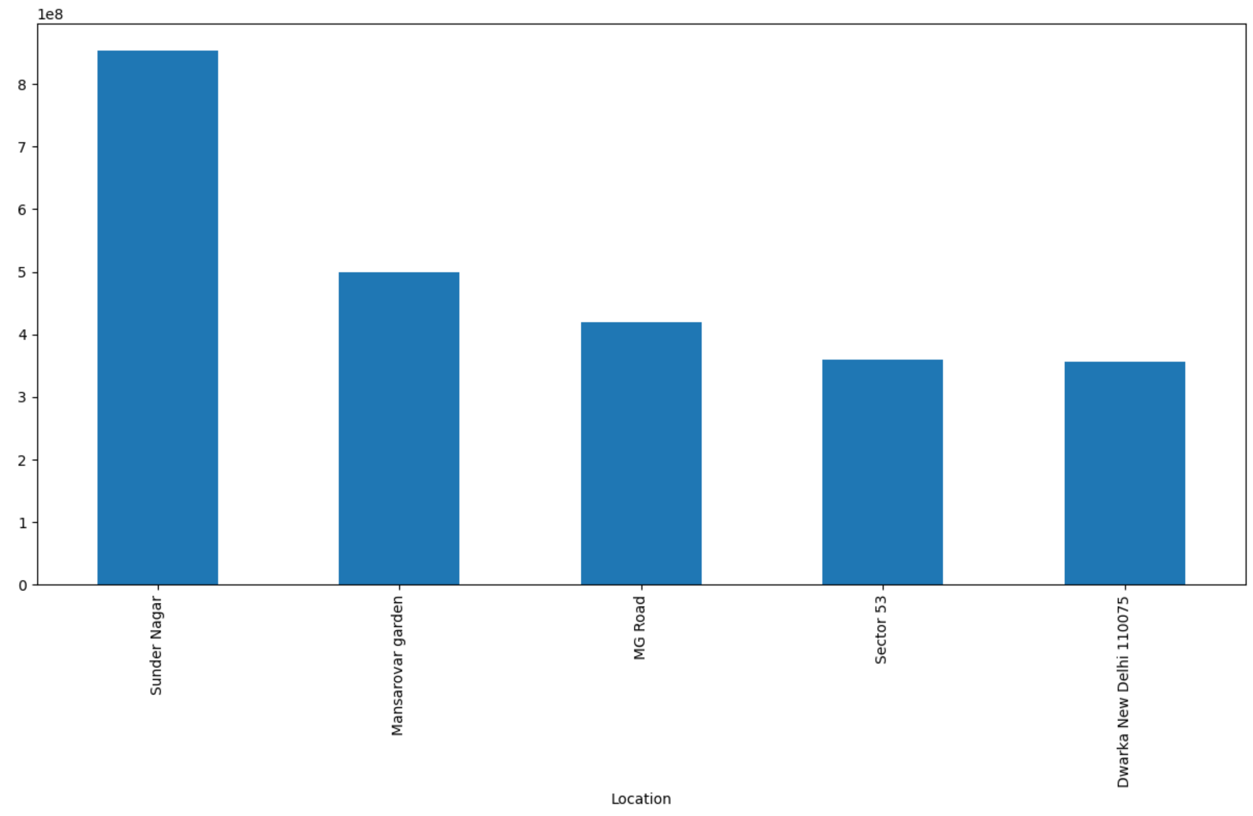


Figure 7: India's most expensive areas

For business purposes we decided to create our models only for one of six data frames described earlier - Bangalore. As housing prices and standards in every Indian district differ, the best solution is to create a separate prediction model for each of India's metropolitan cities. The model we are going to create for Bangalore will work for other data sets as well.

Therefore, below we presented some highlights of Bangalore exploratory data analysis.

Bangalore data set consists of 6207 rows and 40 columns. Apart from columns such as Price, Area, Location, No. of Bedrooms and Resale there are 4256 NaN values for every other variable.

- Top property locations in Bangalore are Electronic City Phase 2, RR Nagar and Begur with over 150 properties located in each of them.
- The most common size of the property is 1445 sq feet (approx 133 sq metres). Properties over 3960 sq feet (huge - 365 sq metres) and below 500 sq feet are rare to find.
- The majority of houses costs less than 15000000.0 INR (178k EUR).
- It goes without saying that the bigger the area, the higher the price and number of bedrooms. Surprisingly, there happen to be properties with big footage and the highest number of bedrooms that cost less than average. On the other hand, there are also expensive, small houses with one or two bedrooms.
- Properties located close to sports facilities, gymnasium or jogging track tend to be more expensive. There are not many properties near school or shopping mall but if there are any, they are usually smaller (which implies lower price).
- Bangalore's most expensive areas are Richmond Town, Bannerghatta Road Jigani and Kalkere with average property prices higher than 55000000.0 INR (653k EUR).

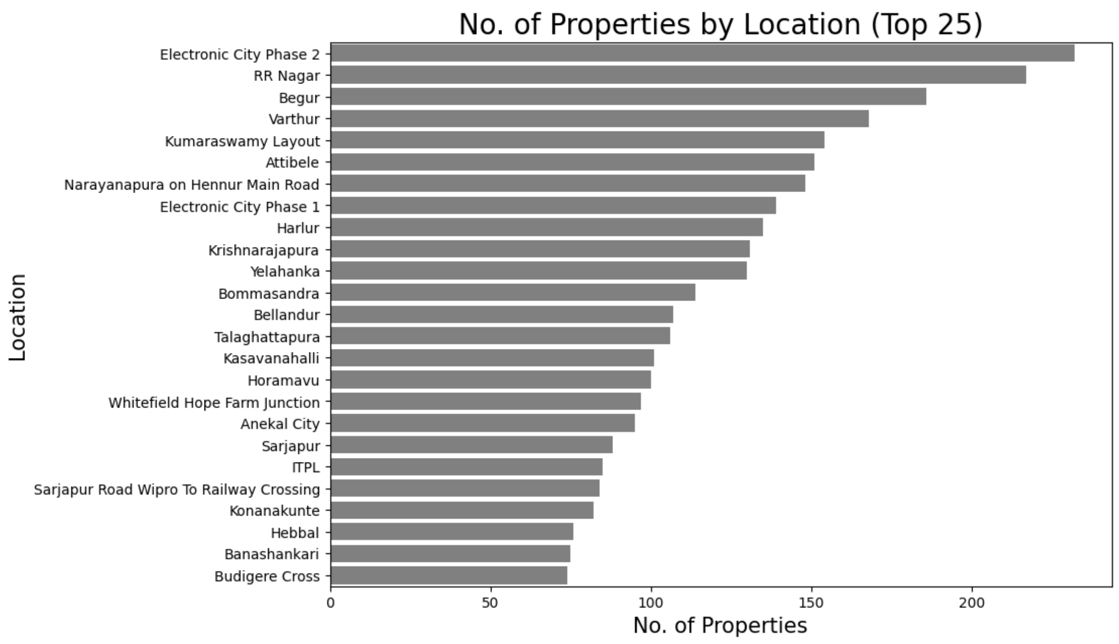


Figure 8: Top property Locations in Bangalore

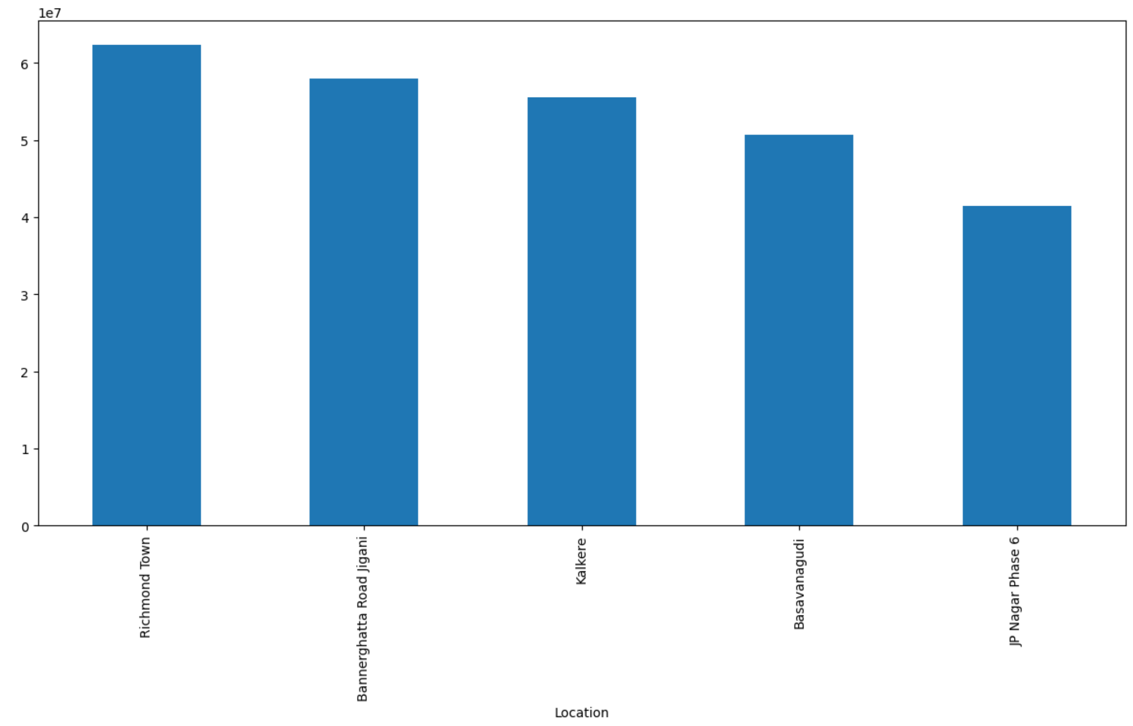


Figure 9: Bangalore's most expensive areas

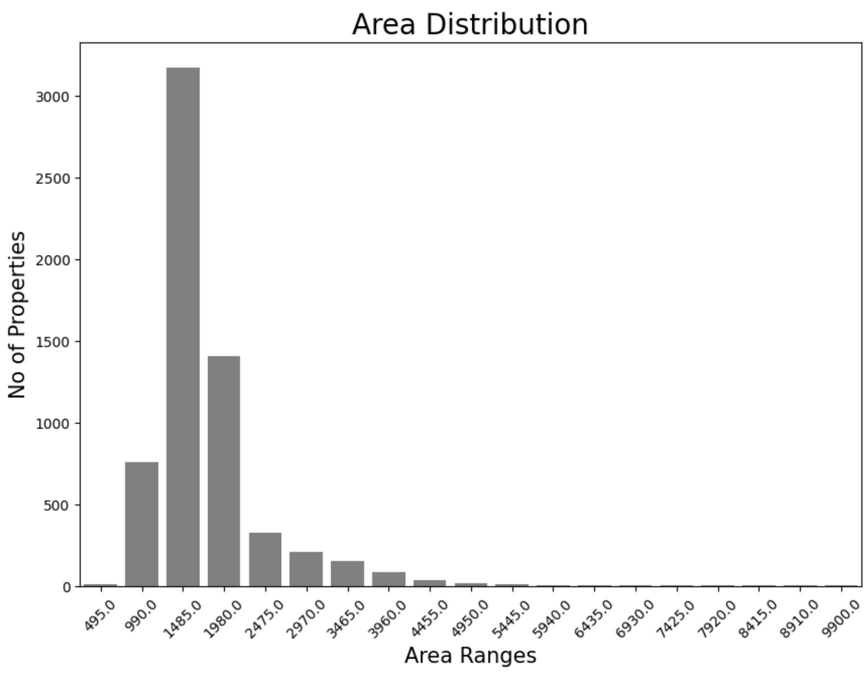


Figure 10: Area distribution

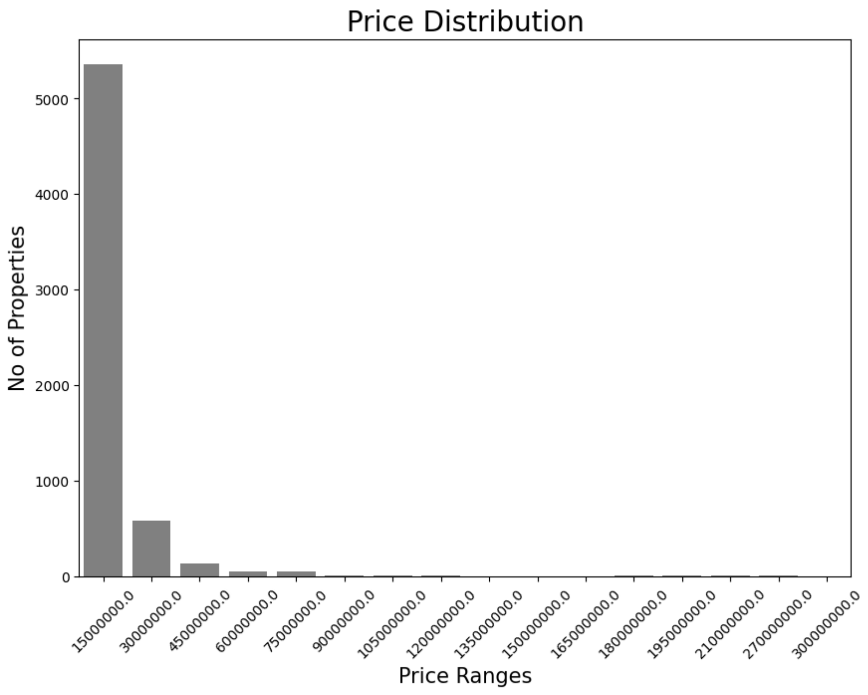


Figure 11: Price distribution

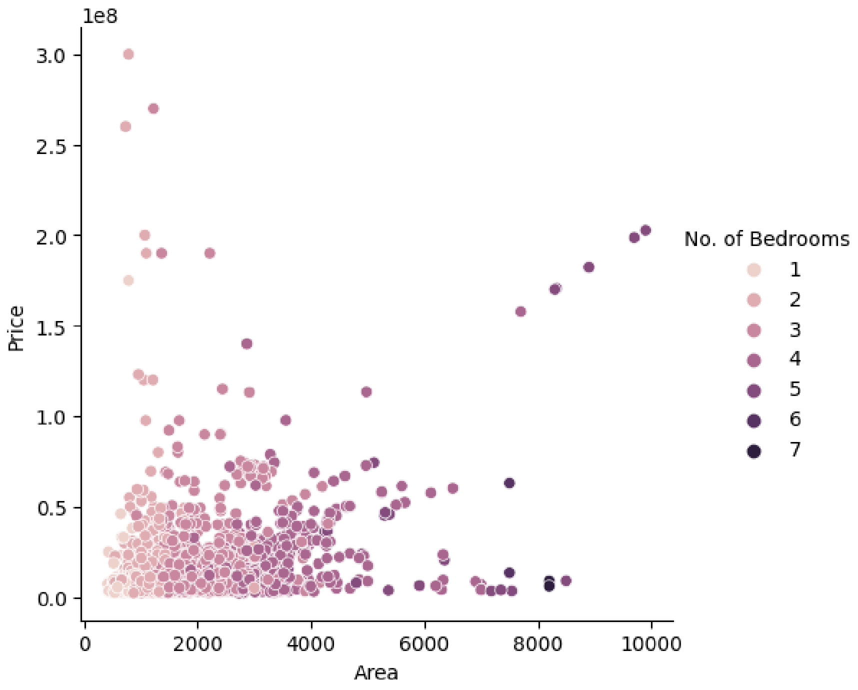


Figure 12: Number of bedrooms

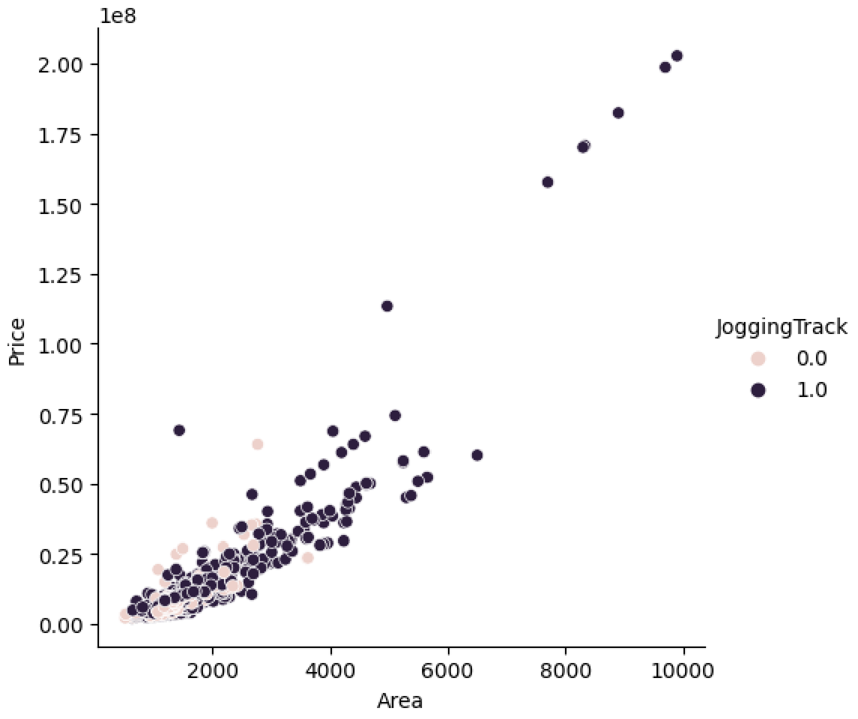


Figure 13: Jogging track access

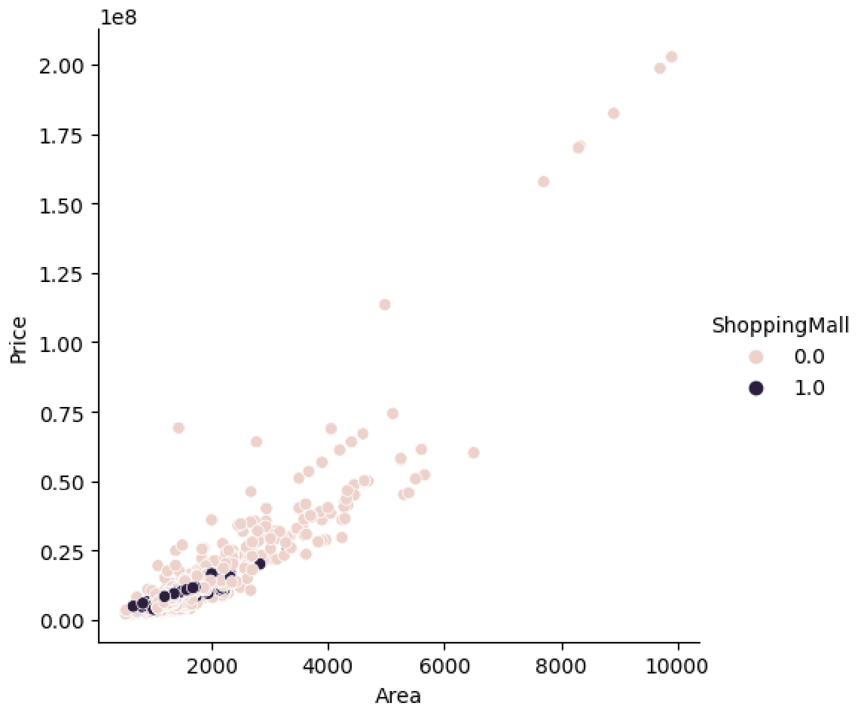


Figure 14: Shopping mall access

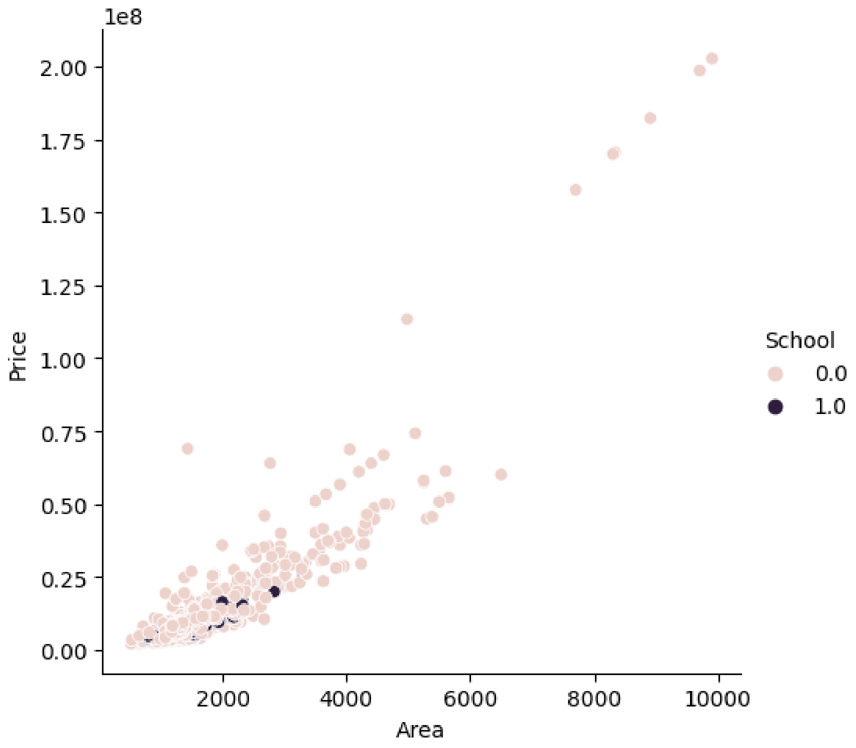


Figure 15: School access

2 First step modeling

Our fundamental interest of research are tree-based models, so we selected four for later exploration. The ones we wanted to examine are: Random Forest, XGBoost, LightGBM and Catboost. For each of them we created a considerable number of experimental models with and without optimized hyperparameters to choose the best five models. To compare these algorithms for regression of prices in India's city Bangalore we used several metrics of calculating the error of predictions: RMSE - Root Mean Square Error, MSE – Mean Square Error, MAE – Mean Absolute Error.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(x_i - y_i)^2}{N}} \tag{1}$$

$$MSE = (RMSE)^2 \tag{2}$$

$$MAE = \frac{\sum_{i=1}^{N}|x_i - y_i|}{N} \tag{3}$$

Where:

- N = sample size
- x_i = predicted value
- y_i = observed value

Our next move was to select the best three models of the remaining twenty (five from each algorithm). Here are the descriptions and results of the first step modeling.

2.1 Random Forest Models

2.1.1 Method description

The Random Forest model for regression task is an ensemble algorithm that combines multiple Regression Trees (RTs). Each RT is trained using a random subset of features and the output is the average of the individual RTs. The sum of squared errors for a tree T is calculated by:

$$S = \sum_{c \in leaves(T)} \sum_{i \in C}(y_i - m_c)^2 \tag{4}$$

Where:

- $m_c = \frac{1}{n_c}\sum_{i \in C} y_i$ is the prediction for leaf c.

The main aim of each split in RT is to minimize the S.

2.1.2 Hyperparameters to optimize

Typical Random Forest implementations have five hyperparameters to consider into in-depth optimization. These are:

- **number of trees in model (num.trees)** - the more rows in the data, the more trees are needed but there is a danger of overfitting the model.
- **maximal tree depth (max.depth)** - the deeper the tree, the more splits it has and it captures more information about the data.
- **minimal node size (min.node.size)** - setting this number larger causes smaller trees to be grown (and thus take less time).
- **number of variables to possibly split at in each node (mtry)** - the sum of the carnality of each variable. To illustrate: in case three variables A, B, C are selected at a node i, with A being a binary feature, B an integer in the interval [1,100] and C a continuous variable taking on 200 distinct values in the current sample, the number of possible splits is 2+100+200=302. The bigger the mtry, the bigger randomness of splits in RT.

- **splitting rule (splitrule)** – regression tasks require one of four possible splitting rules: estimation by variance ('variance'), by maximally selected rank statistics ('maxstat'), by random choice of a certain number of attributes ('extratrees') or by beta distributions of variables ('beta').

There are four classic approaches to tune hyperparameters: *manual* – based on our intuition, experience or guessing, *grid search* – creating a grid of parameters combinations and iterating through it to find the best combination, *random search* - randomly searching hyperparameters using a grid and *automated hyperparameter tuning* – based on gradient descent algorithm.

2.1.3 Libraries used and optimizing methods

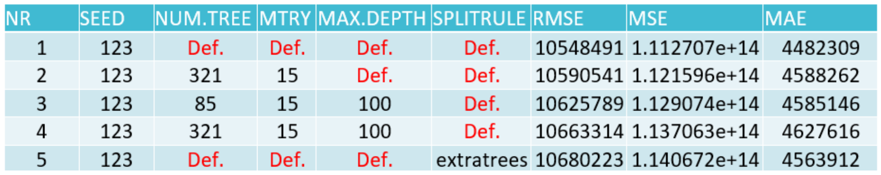
In R programming language the popular package for creating Random Forest models is called **'ranger'**. The default parameters of the RF implementation in this package are:

- num.trees = 500
- max.depth = 0 (means unlimited depth)
- min.node.size = 5 (for regression)
- mtry = (rounded down) square root of the number variables
- splitrule = 'variance'

At the early stage of modeling, we decided to compute model with all default parameters to see what the basic RMSE, MSE and MAE metrics are for the Bangalore dataset. Then we started changing *splitrule* parameter to see if any is better than the 'variance' rule. Nevertheless, the default *splitrule* proved to be the best one for the regression task. Our next move with exploring Random Forest in R language was tuning parameters with **'mlr3'** library, which is responsible for efficient, object-oriented programming on the building blocks of machine learning. To tune the RF algorithm, we had to specify:

- the search space: *num.trees* from 10 to floor of 10% of training observations at the first attempt and to 1000 at the second attempt, *mtry* from 2 to 15 at the first attempt and to 100% of training observations at the second attempt, *max.depth* from the set of 5,15 and 100.
- maximum of iterations in each tuning: we set it to 10.
- control object defining tuning type: we chose random search in this case.

2.1.4 Best models presentation



NR	SEED	NUM.TREE	MTRY	MAX.DEPTH	SPLITRULE	RMSE	MSE	MAE
1	123	Def.	Def.	Def.	Def.	10548491	1.112707e+14	4482309
2	123	321	15	Def.	Def.	10590541	1.121596e+14	4588262
3	123	85	15	100	Def.	10625789	1.129074e+14	4585146
4	123	321	15	100	Def.	10663314	1.137063e+14	4627616
5	123	Def.	Def.	Def.	extratrees	10680223	1.140672e+14	4563912

Figure 16: Scores of test sample

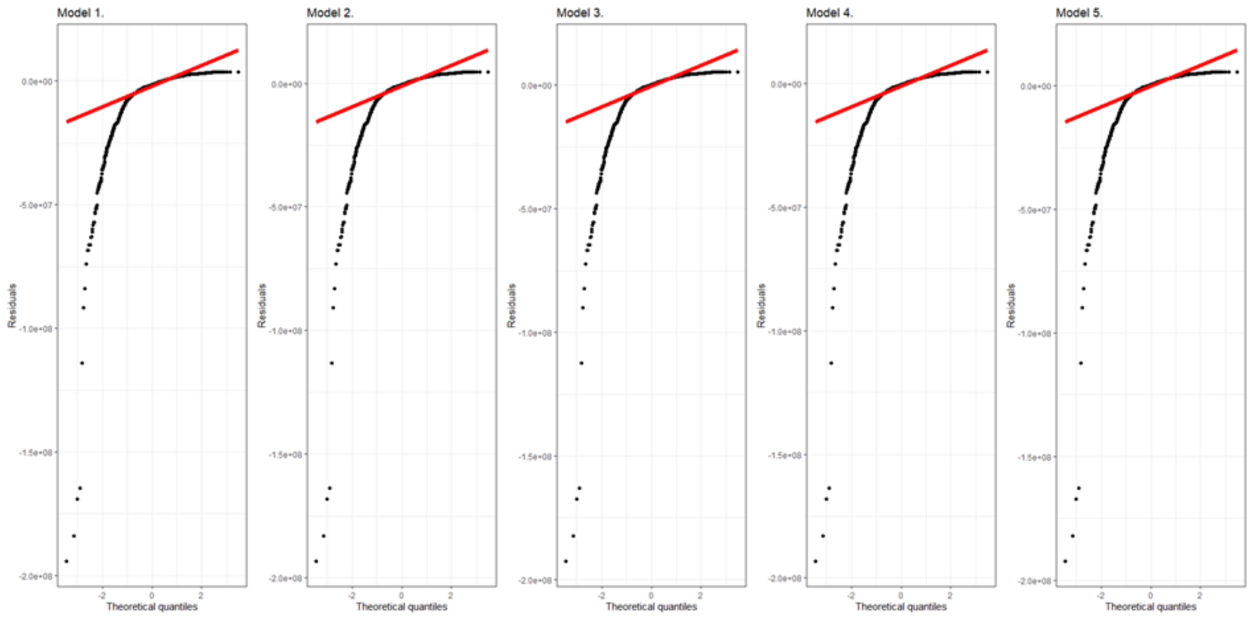


Figure 17: Residual plots

As we can see at the residual plots above RF algorithm with simple optimization of parameters gives unsatisfactory results. Nearly 50% of test scores are unfitted.

2.2 XGBoost Models

2.2.1 Method description

XGBoost is an optimized gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way.

2.2.2 Hyperparameters to optimize

The XGBoost hyperparametres we decided to consider into optimization are:

- **eta (learning rate)** - step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- **max.depth** - step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- **min.child.weight** - step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- **nrounds** – the number of decision tress in the final model

2.2.3 Libraries used and optimizing methods

The default values of the hyperparameters presented above are:

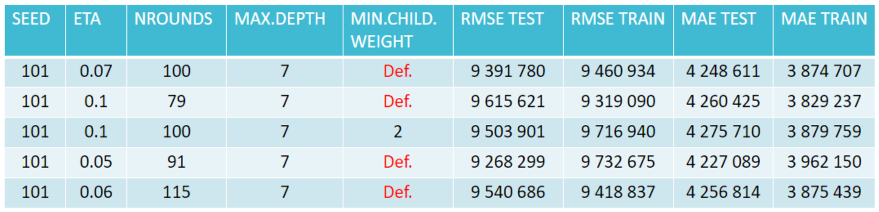
- eta = 0.3
- max.depth = 6
- min.child.weight = 1

- nrounds (needs to be specified by the user himself)

Unfortunately the XGBoost R package does not have any inbuilt feature for doing grid/random search. To overcome it, we used **’mlr’** library to perform the extensive parametric search and try to obtain optimal accuracy. Another approach was using **xgb.cv** function for calculating best nrounds value for the model and at the same time following the rules presented below:

- If RMSE for xgboost model created using default parameters is not satisfying, decrease *eta* value.
- Don’t introduce *gamma* until you can observe a significant difference in train and test error.
- Tune the regularization parameters *(alpha, lambda)* if required using the best parameters obtained so far.
- At last, carefully increase/decrease eta and follow the procedure.

2.2.4 Best models presentation



SEED	ETA	NROUNDS	MAX.DEPTH	MIN.CHILD. WEIGHT	RMSE TEST	RMSE TRAIN	MAE TEST	MAE TRAIN
101	0.07	100	7	Def.	9 391 780	9 460 934	4 248 611	3 874 707
101	0.1	79	7	Def.	9 615 621	9 319 090	4 260 425	3 829 237
101	0.1	100	7	2	9 503 901	9 716 940	4 275 710	3 879 759
101	0.05	91	7	Def.	9 268 299	9 732 675	4 227 089	3 962 150
101	0.06	115	7	Def.	9 540 686	9 418 837	4 256 814	3 875 439

Figure 18: Scores of test and train sample

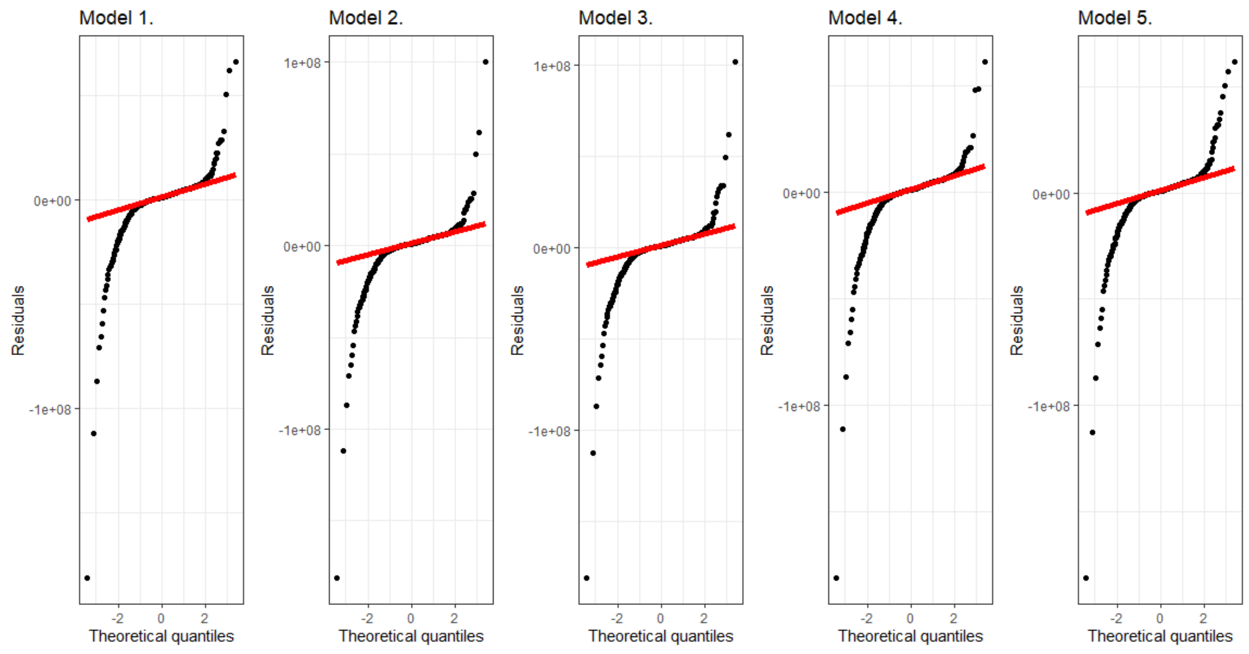


Figure 19: Residual plot

Again, the results from the residual plot above are unsatisfactory - about a half of test scores are unfitted.

2.3 Catboost models

2.3.1 Method description

Catboost is another algorithm that was used in this project. It's relatively new as the initial release took place in 2017 and stable release occurred at the beginning of April this year (2022). Nevertheless, as for year 2021 it was listed the top-7 most frequently used ML framework in the 2021 survey.
CatBoost is based on the theory of decision trees and gradient boosting. The core idea of boosting is to sequentially combine many weak models (models performing slightly better than random chance) in order to create a strong competitive predictive model. The sequential fitting of decision trees by gradient boosting, results in fitted trees learning from the mistakes of former trees and hence reduced errors. This process of adding a new function to existing ones is continued until the selected loss function is no longer minimized.

2.3.2 Hyperparameters to optimize

Due to our research, the key parameters to optimize in this case are:

- **iterations (num_trees)** - the maximum number of trees that can be built. To speed up the training, the number of iterations can be decreased. When the number of iterations decreases, the learning rate needs to be increased.
- **depth (max_depth)** – the maximal tree depth. The higher the value, the more complex a model will become, which indicates increased performance on the training set, but might cause overfitting. In most cases, the optimal depth ranges from 4 to 10.
- **learning_rate (eta)** - the parameter, which controls how much to change the model, while moving toward a minimum of a loss function. The lower the value, the better the accuracy but slower the model learns. On the contrary, higher value may result in unstable training process.

2.3.3 Libraries used and optimizing methods

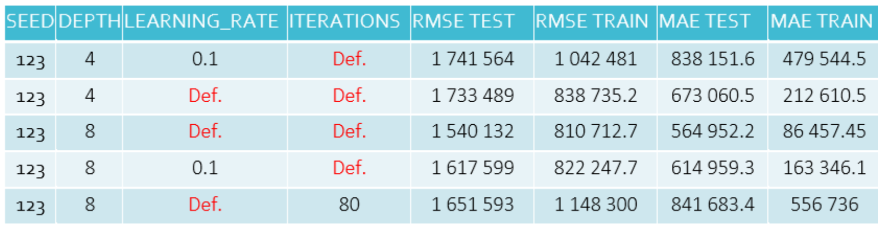
The default values of the hyperparameters presented above are:

- iterations = 1000
- depth = 6
- learning_rate = 0.3

In order to avoid long computing time we decided to optimize the hyperparameters using random search. The Caret library provides random search hyperparameters tunning for Catboost in R. Unfortunately it doesn't cooperate well with regression models yet, as they occur to the function as classification. Catboost is steadily increasing in popularity, thus not all methods are supported for it in R yet. Therefore, the tunning was conducted manually. The choice of the values for parameter grid was inspired by the research on hyperparameter tunning performed in assembled articles.
The first model was built using the default parameters so as to compare whether the tunning was successful. The metrics used for comparison were RMSE and MAE – for both of them the lower the score, the better. In total of the 20 models built, we managed to achieve better scores then the default model.

2.3.4 Best models presentation

Models were built on Bangalore dataset. The metrics scores are presented in Indian rupees. As for the day 10.05.2022, 1 000 000 INR was equivalent to approx. 12 300 EUR.



SEED	DEPTH	LEARNING_RATE	ITERATIONS	RMSE TEST	RMSE TRAIN	MAE TEST	MAE TRAIN
123	4	0.1	Def.	1 741 564	1 042 481	838 151.6	479 544.5
123	4	Def.	Def.	1 733 489	838 735.2	673 060.5	212 610.5
123	8	Def.	Def.	1 540 132	810 712.7	564 952.2	86 457.45
123	8	0.1	Def.	1 617 599	822 247.7	614 959.3	163 346.1
123	8	Def.	80	1 651 593	1 148 300	841 683.4	556 736

Figure 20: Scores calculated on test and training samples.

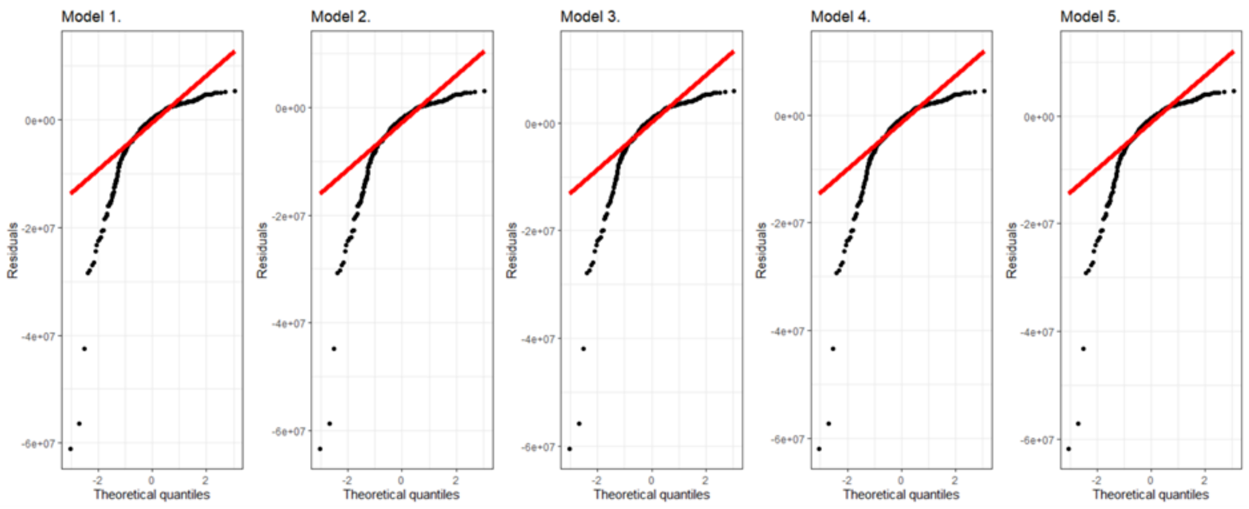


Figure 21: Residual plot

By the graphics presented above the performance of 5 chosen models (calculated on test sample) can be judged . The results leave the room for improvement.

2.4 LightGBM models

2.4.1 Method description

LightGBM is a gradient boosting framework that uses tree-based learning algorithms. It is designed to be distributed and efficient with the following advantages: Faster training speed and higher efficiency. Lower memory usage. Better accuracy. Support of parallel, distributed, and GPU learning. Capable of handling large-scale data. While other algorithms trees grow horizontally, the LightGBM algorithm grows vertically meaning it grows leaf-wise and other algorithms grow level-wise. LightGBM chooses the leaf with a large loss to grow. It can lower more loss than a level-wise algorithm when growing the same leaf.

2.4.2 Hyperparameters to optimize

Core parameters used during tuning of hyperparameters:

- **num_leaves** - LightGBM adds nodes to trees based on the gain from adding that node, regardless of depth. Because of this growth strategy, it isn't straightforward to use max_depth alone to limit the complexity of trees. The num_leaves parameter sets the maximum number of nodes per tree. Decrease num_leaves to reduce training time.
- **max_depth** - limit the max depth for the tree model. This is used to deal with over-fitting when a dataset is small. The tree still grows leaf-wise
- **learning_rate** - the size of the "step" that the algorithm takes. learning_rate will not have any impact on training time, but it will impact the training accuracy. As a general rule, if you reduce num_iterations, you should increase learning_rate.
- **num_iterations** - The num_iterations parameter controls the number of boosting rounds that will be performed. Since LightGBM uses decision trees as the learners, this can also be thought of as the "number of trees". If you try changing num_iterations, change the learning_rate as well.

LightGBM library has its function that consumes data set and gives wanted shape of data, that works the best with LightGBM models, so Bangalore cleared data we prepared with that function too. During modeling, besides generic LightGBM parameters we tried, F(y) - function was used on the Price column. Since values of the Price column are from a wide range, we tried to examine models with a logarithm of Price but models with that change did not reach a better score.

2.4.3 Libraries used and optimizing methods

The default values of the hyperparameters presented above are:

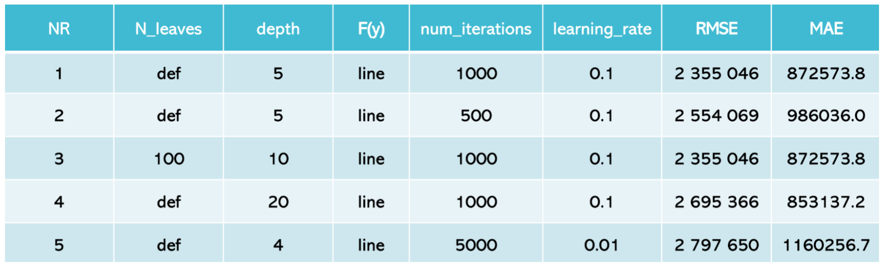
- num_leaves = 31
- max_depth = -1
- learning_rate = 0.1
- num_iterations = 100

Hyperparameters were tuned with the use self-written grid search where the covered grid was:

- max_depth = [2,4,5, 10, 15, 20],
- learning_rate = [0.01, 0.1, 1],
- num_iterations = [500, 1000, 1500]

as so models with self-chosen parameters were examined. Hyperparameters were chosen with respect to RMSE. What was unexpected is that the self-chosen parameters appeared to be the most relevant ones.

2.4.4 Best models presentation



NR	N_leaves	depth	F(y)	num_iterations	learning_rate	RMSE	MAE
1	def	5	line	1000	0.1	2 355 046	872573.8
2	def	5	line	500	0.1	2 554 069	986036.0
3	100	10	line	1000	0.1	2 355 046	872573.8
4	def	20	line	1000	0.1	2 695 366	853137.2
5	def	4	line	5000	0.01	2 797 650	1160256.7

Figure 22: Scores calculated on test samples.

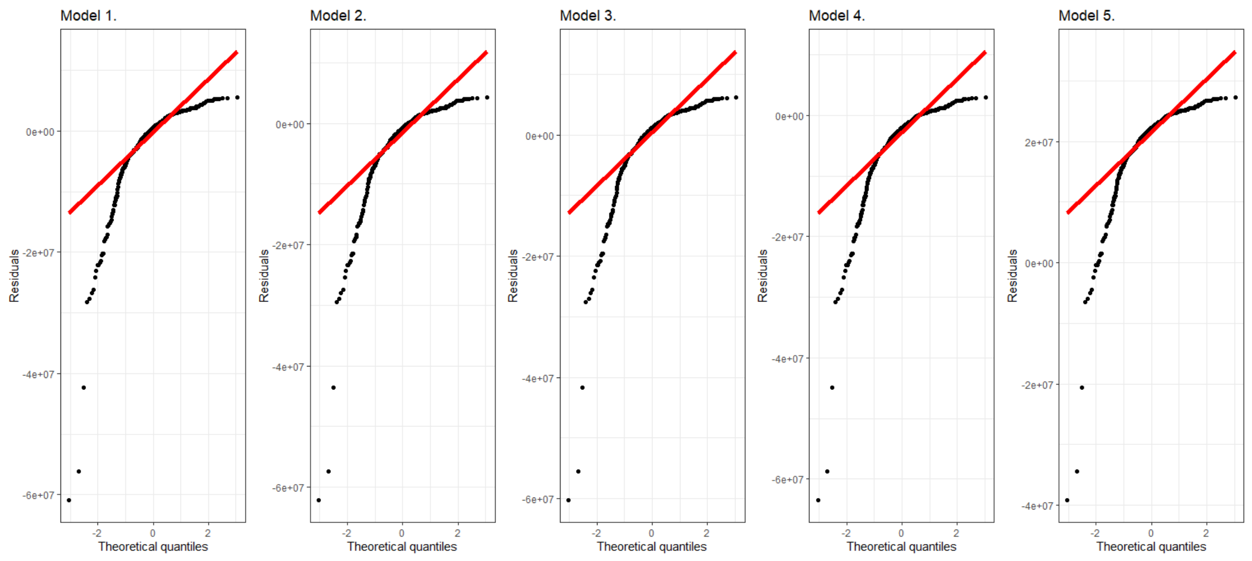


Figure 23: Residual plot

The results presented above are quite promising, even if at the beginning expectations according to models were higher. We can see on the plot that great mispredictions appear but after model examination, we decided not to consider these values as outliers since we want our models to be more relevant for a whole range of prices that appear on the real estate market. Overall and we consider them good and reliable models.

2.5 Conclusion

The obtained scores for models suggest that the best algorithm for inspected case is Catboost. Presented Catboost model number 3 had the lowest metric scores, which indicates that it has the best performance and will be further examined.