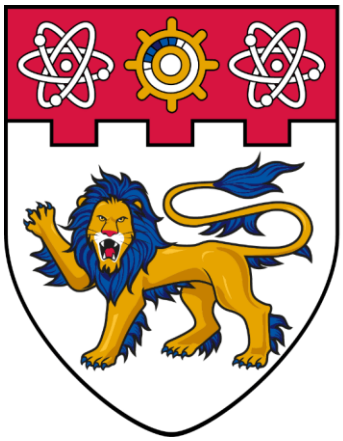


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Realty Price Prediction for Sberbank Russian Housing Market

CZ4041 – Machine Learning

Rank 532/3265 (Top 17%)

Supervisor: Sinno Jialin PAN

Submitted By: Group 50

Name	Matric No.
Liew Zhi Li (Sherna)	U1821610C
Huang NengQi	U1921454E
Mark Tan Rong Hui	U1923902B
Goh Wei Jie Benjamin	U1921615J

School of Computer Science & Engineering

Nanyang Technological University, Singapore

Table of Contents

Figures

Tables

1 Introduction

1.1 Problem Statement

Can we make accurate predictions on Russian realty prices?

Sberbank is the largest, state-owned bank in Russia and offers predictions on realty prices to help customers in making the decision to purchase property. There is a myriad of housing features such as apartment size, number of bedrooms, and location that makes realty price prediction a complex task. Although the housing market is relatively stable, the country's volatile economy makes predicting realty prices more difficult, thus, this requires a more advanced approach than simple regression models. The goal of this Kaggle competition is to utilize the wide array of features of Russian housing market dataset to develop models to predict realty prices [1].

1.2 Evaluation Score on Kaggle

This Kaggle competition uses the Root Mean Squared Logarithmic Error (RMSLE) between the actual data and predicted prices, where n is the total number of observations in the dataset, p_i is the prediction of target, and a_i is the actual target for i.

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(a_i + 1))^2}$$

Equation 1: Root Mean Squared Logarithmic Error (RMSLE)

RMSLE is a measure of the ratio of the predicted and actual values. It is the Root Mean Squared Error of the log-transformed predicted and the log-transformed actual values [2]. RMSLE adds 1 to both actual and predicted values before applying natural logarithm to avoid taking the natural log of possible zero values. Thus, this is useful if the actual or predicted value contains zero-value elements.

The following table details the task distribution of each member in this group.

Name	Contribution
Liew Zhi Li (Sherna)	• Exploratory Data Analysis • Data Pre-Processing • Feature Engineering • Decision Tree • Report • Presentation • Video
Huang NengQi	• Random Forest • Report • Presentation
Mark Tan Rong Hui	• Exploratory Data Analysis • Feature Engineering • LightGBM • Naïve XGBoost • Custom Model • Report • Presentation
Goh Wei Jie Benjamin	• Exploratory Data Analysis • Data Pre-Processing • Feature Engineering • XGBoost Regressor • AdaBoost • SGD • Random Forest • Report • Presentation

The YouTube presentation video link can be found at: https://youtu.be/AiheoHESNiw

The final .csv submission is the LightGBM Raw model, which is the file final_submission.csv.

2 Exploratory Data Analysis (EDA)

2.1 Overview of Datasets

Sberbank provided three datasets in this Kaggle competition. The first dataset is a file containing a set of 100 macroeconomic indicators in macro.csv, with each feature labelled with a transaction timestamp. The following figure and table below illustrate a small subset of the top 5 row's information in macro.csv. The full information can be found in the data_dictionary.txt.

	timestamp	oil_urals	gdp_quart	gdp_quart_growth	cpi
0	2010-01-01	76.10	NaN	NaN	NaN
1	2010-01-02	76.10	NaN	NaN	NaN
2	2010-01-03	76.10	NaN	NaN	NaN
3	2010-01-04	76.10	NaN	NaN	NaN
4	2010-01-05	76.10	NaN	NaN	NaN

Figure 1: Subset of Top 5 Rows in macro.csv

Column	Description
timestamp	Transaction timestamp
oil_urals	Crude Oil Urals ($/bbl)
gdp_quart	GDP
gdp_quart_growth	Real GDP growth
cpi	Inflation - Consumer Price Index Growth

Table 1: Subset of Data Dictionary

The training and testing instances are found in train.csv and test.csv and consists of 292 and 291 columns respectively. The target variable price_doc is found in train.csv. The other features represent the property's surrounding neighborhoods.

The train dataset consists of data collected from 20 August 2011 to 30 June 2015, a total period of 3 years, 10 months, and 11 days. The test dataset spans from 1 July 2015 to 30 May 2016, a total period of 10 months, 30 days. The following figure and table illustrate a subset of the train.csv and its data dictionary.

	id	timestamp	full_sq	life_sq	floor
0	1	2011-08-20	43	27.00	4.00
1	2	2011-08-23	34	19.00	3.00
2	3	2011-08-27	43	29.00	2.00
3	4	2011-09-01	89	50.00	9.00
4	5	2011-09-05	77	77.00	4.00

Figure 2: Subset of Top 5 Rows in train.csv

Column	Description
price_doc	Sale price
id	transaction id
timestamp	date of transaction
full_sq	total area in square meters
life_sq	living area in square meters
floor	for apartments, floor of the building

Table 2: Subset of Data Dictionary

The following table illustrates the number of distinct datatypes for all three datasets.

macro.csv (2484, 100)		train.csv (30471, 292)		test.csv (7662, 291)	
float64	94	float64	119	float64	116
int64	2	int64	157	int64	159
object	4	object	16	object	16

Table 3: Summary of Distinct Datatypes of all Datasets

2.2 Missing Values

The following figures illustrate the percentage of missing values for the features in train.csv. There is a total of 51 out of 292 variables that contains missing values. Features with missing values can negatively affect the machine learning model, thus imputation is required.

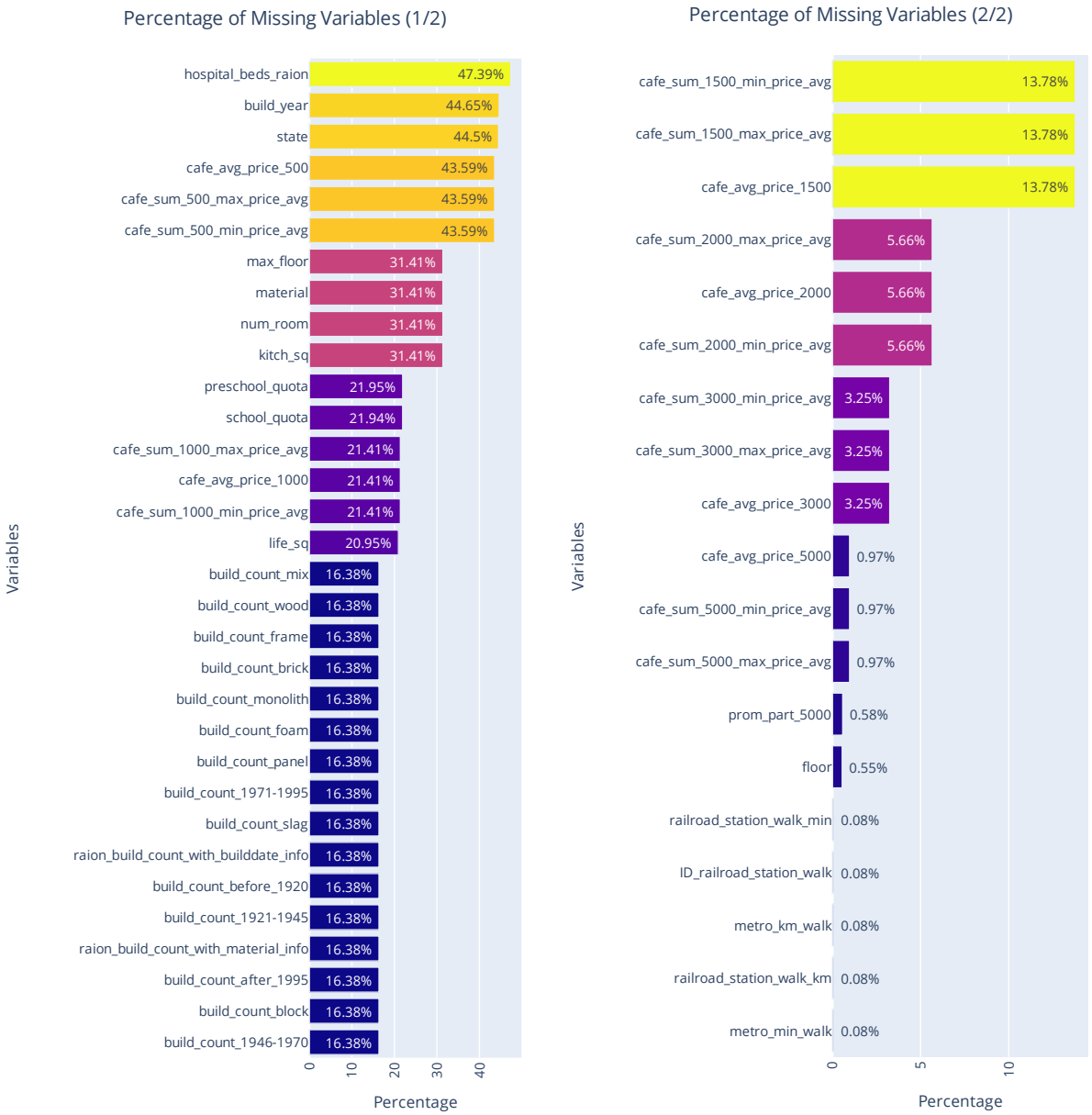


Figure 3: Percentage of Missing Variables

2.3 Univariate Analysis for Target Variable 'price_doc'

The following table and figure illustrate the descriptive statistics and the histogram for price_doc with 100 bins.

Descriptive Statistics for price_doc	
count	30,471.00
mean	7,123,035.28
std	4,780,111.33
min	100,000.00
25%	4,740,002.00
50%	6,274,411.00
75%	8,300,000.00
max	111,111,112.00

Table 4: Descriptive Statistics for price_doc

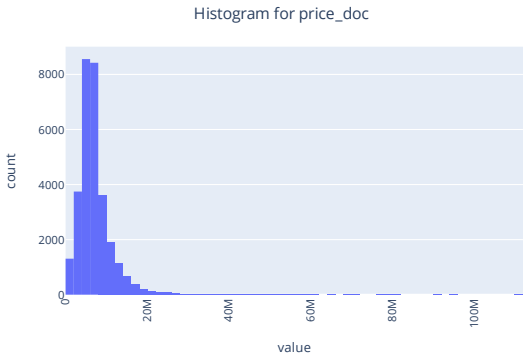


Figure 4: Histogram for price_doc

The average sale price of realty is around 7.1 million rubles.

In the histogram, the realty with the highest count of 8557 are priced in the 4 to 6 million Rubles range, followed by 8423 realties priced in the 6 to 8 million range, followed by 3746 realty priced in the 2 to 4 million range, followed by 3627 realty priced in the 8 to 10 million range.

The following figure illustrates the scatterplot for the price_doc variable.

Figure 5: Scatterplot for price_doc

It is observed that there are about 30,000 properties priced from 100,000 up to 20 million Rubles. From 20 million Rubles onwards, there is a sharp, rapid increase in the sale price and in smaller quantities. From this scatterplot, it is evident there are a few outliers on the far top right, where sale price is above 80 million. The highest yellow point represents the maximum property price at 111,111,112.00 Rubles, followed by the second orange point at 95 million, and 91 million.

2.3.1 Time Series of Median Price over Time

The following figure illustrates the time series graph of the median price_doc over time.

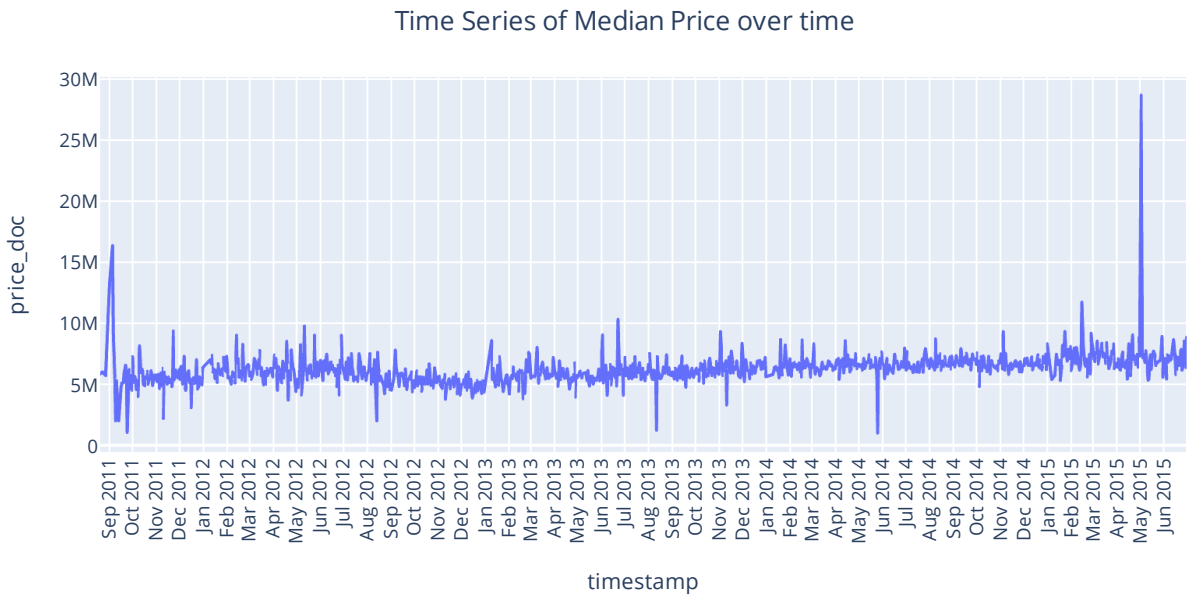


Figure 6: Time Series of Median Price over Time

For the entire period from 2011 to 2015, the median property price consistently fluctuates in the range of 5 to 7 million Rubles.

However, there are a few anomalies present. In 2011, the median price rose sharply in a few days from 5.7 million Rubles on August 27 to 16 million Rubles on September 5, before dropping to 2 million Rubles on September 9.

There are also a few repeated anomalies, where property price dropped below this expected range on dates in Q3-4 of the years, such as on September 24, 2011, at 1.05 million Rubles, November 10, 2011, at 2.2 million Rubles, December 16, 2011, at 3.07 million Rubles, and on August 13, 2012, at 2 million Rubles, August 11, 2013, at 1.23 million Rubles, November 10, 2013, at 3.3 million Rubles, May 25, 2014, at 1 million Rubles.

Interestingly, this repeated dip in median property price in Q3-4 of 2011, 2012, and 2013 did not happen in Q3-4 of 2014.

Lastly, there is an anomaly where the property price rose astronomically from 7.3 million Rubles on April 30, 2015, to 28.65 million rubles on May 2, 2015, which is the highest price in the entire history of the dataset.

All these anomalies could indicate some critical major events that had a huge influence property price. One plausible reason is that the property prices in Russia are directly related to petroleum prices, thus this could be one of the myriads of external factors that contributed to these property price trends and anomalies [3].

2.3.2 Top 5 Most Expensive & Inexpensive Properties

The following tables show the top 5 most expensive properties sorted by sub_area in descending order and the descriptive statistics for property prices in Teplyj Stan.

Top 5 Most Expensive Properties			
Rank	**price_doc**	**full_sq**	**sub_area**
1	111111112	55	Teplyj Stan
2	95122496	220	Presnenskoe
3	91066096	185	Ramenki
4	80777440	184	Ramenki
5	78802248	206	Pokrovskoe Streshnevo

Table 5: Top 5 Most Expensive Properties

Descriptive Statistics for Property Prices in Teplyj Stan	
count	165.00
mean	8,563,557.53
std	8,877,604.58
min	990,000.00
25%	6,200,000.00
50%	7,800,000.00
75%	10,000,000.00
max	111,111,112.00

Table 6: Descriptive Statistics for Property Prices in Teplyj Stan

The variable sub_area denotes the district the property is located in. The most expensive property of 111 million Rubles is found in Teplyj Stan. However, it is unusually small with a total area of 55 square meters, compared to the other top 4 most expensive houses where the sizes ranged from 184 to 220 square meters. There are 165 property prices for Teplyj Stan. The average property price is 8.5 million Rubles. At the 75th percentile, the property price in this sub area is priced at 10 million Rubles. Based on observation, the property with the highest price of 111 million Rubles is indeed an extreme outlier and this suggests it could be a luxury property. Thus, the training set that consists of property prices of higher than 80 million Rubles onwards is considered an outlier.

On the other end of the spectrum, the table on the left shows the top 5 most inexpensive properties sorted by descending order, and the table on the right shows the descriptive statistics for property prices in Gol'janovo. The most inexpensive property is priced at 100,000 Rubles and is found in Gol'janovo. Interestingly, although it is the most inexpensive, it is slightly bigger compared to the other top 4 properties that are slightly more expensive.

Top 5 Most Inexpensive Properties			
Rank	**price_doc**	**full_sq**	**sub_area**
1	100000	72	Gol'janovo
2	190000	38	Beskudnikovskoe
3	200000	47	Savelki
4	260000	40	Novo-Peredelkino
5	300000	31	Caricyno

Table 7: Top 5 Most Inexpensive Properties

Descriptive Statistics for Property Prices in Gol'janovo	
count	295.00
mean	5,792,339.82
std	2,687,402.44
min	100,000.00
25%	4,800,000.00
50%	5,900,000.00
75%	7,100,000.00
max	19,300,000.00

Table 8: Descriptive Statistics for Property Prices in Gol'janovo

There are 295 property prices for this sub area. The average property price is 5.7 million Rubles.

2.4 Bivariate Analysis of Different Features against Target Variable 'price_doc'

2.4.1 Median Price vs Year

The following figure illustrates the median price over the years.

Figure 7: Bar Chart for Median Price for each Year

There is a steady, gradual increasing trend in the median property sale prices through the years from 5.5 million Rubles in 2011 to 7.1 million Rubles in 2015.

2.4.2 Floor Number vs Price

The following figure illustrates distribution of floor numbers vs median price. The variable floor represents the number of floors of the property.

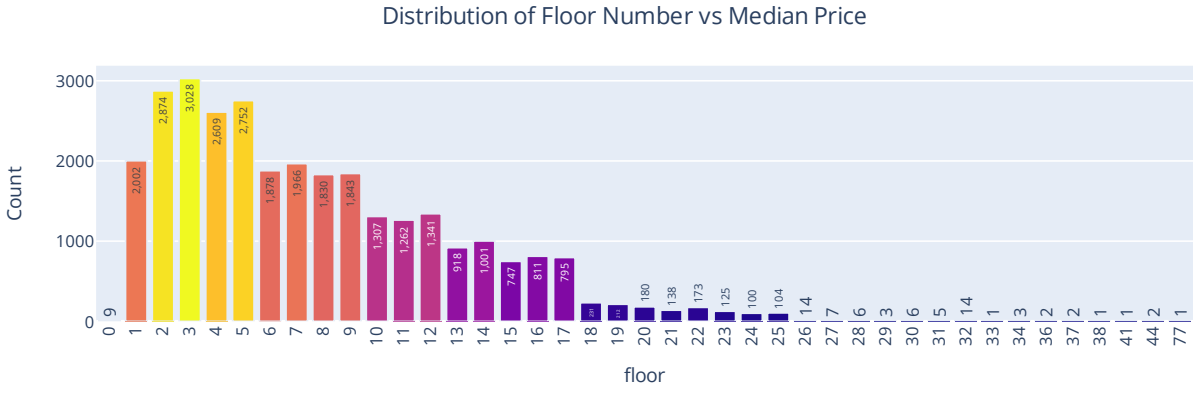


Figure 8: Bar Chart for Floor Number Distribution

Most houses in Russia typically consists of 1 to 17 floors. There are also significant drops in count between floors: 5 & 6, 9 & 10, 12 & 13, 17 & 18. Based on observation, the distribution of floor number is right skewed.

2.4.3 Median Price over Floor Number

The following figure illustrates the line graph of the median price over floor number.

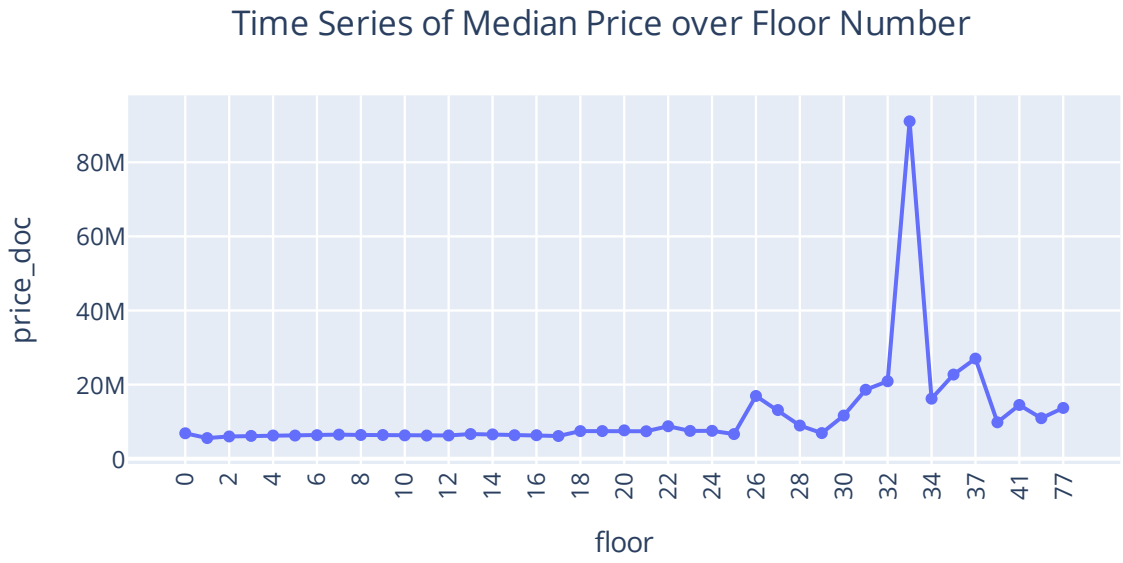


Figure 9: Line Graph of Median Price over Floor Number

Properties with floors 0 to 25 range from 5 – 7 million rubles. For properties with floor 18 to 25, there is a slight increase compared to lower floors. From properties with 25 floor onwards, there is an increase in price.

Based on observation, there is an astronomical increase in median price from floor 32 at 20 million rubles to floor 33 at 91 million. Most property with 30 – 35 floors are in the range from 11 – 16 million rubles. Thus, this suggests that specific property with 33 floors at 91 million is an anomaly.

A property with 0 floor is an individual, stand-alone house, which is slightly more expensive compared to apartment type of housing/housing with multiple floors.

2.4.4 Sub_area vs Mean Price

The following figure illustrates the bar chart for the sub_area against mean price.

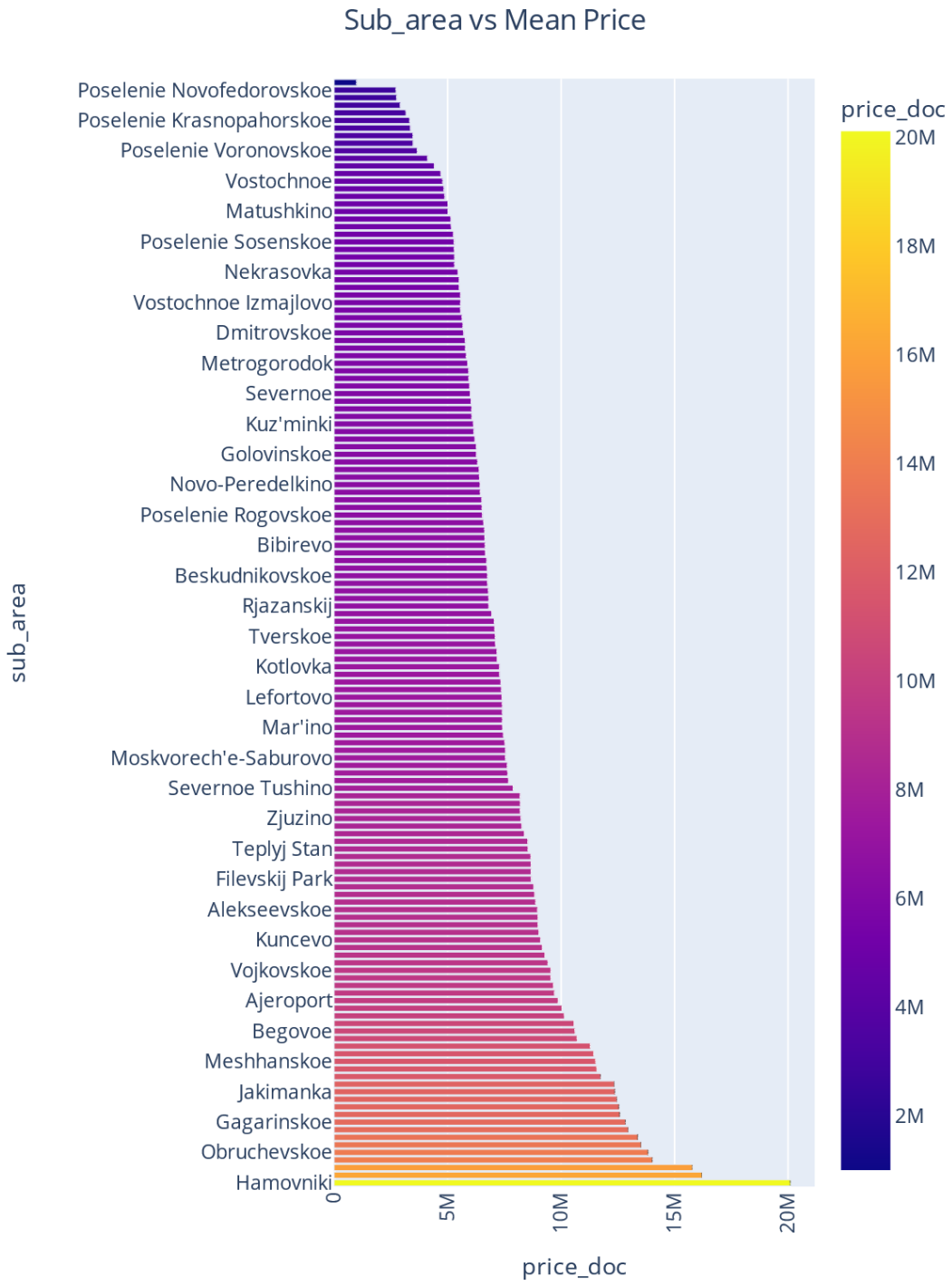


Figure 10: Bar Chart for sub_area vs mean price_doc

The sub area with the lowest average price is Poselenie Klenovskoe, whereas the sub area with the highest average price is Hamovniki.

2.4.5 Full_sq vs price_doc

The following figure illustrates the scatterplot for full_sq vs price_doc.

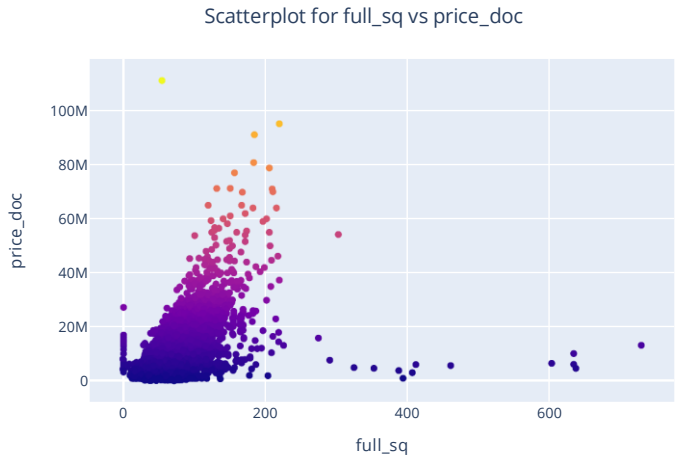


Figure 11: Scatterplot for full_sq vs price_doc

As the square area of the property increases, the sale prices increase linearly. There are a few anomalies, such as the yellow point that has property price of 111 million Rubles. There are also anomalies with abnormally small full_sq of either 0 or 1. This may be noisy or erroneous data.

2.4.6 Green_zone_part vs price_doc

The figure on the left shows the bar chart for green_zone_part vs median price_doc, and the figure on the right shows the scatterplot for green_zone_part vs price_doc. The variable green_zone_part represents the proportion of area of greenery in the total area.

Figure 12: Bar Chart for green_zone_part vs median price_doc

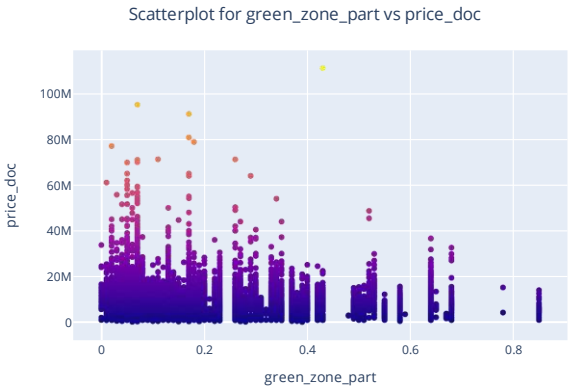


Figure 13: Scatterplot for green_zone_part vs price_doc

For the bar chart, for 0 to 0.9 for green zone, the property prices range from 5.4 million rubles to 7.5 million Rubles. However, for green_zone_part = 0.8, there is a sharp increase in median price at 9.7 million, which could be potential anomalies.

For the scatterplot, as the proportion of greenery increases, the price decreases.

2.4.7 Nine Facilities vs price_doc

The following figures illustrate the scatterplots for the first four facilities, school_km, public_healthcare_km, shopping_centers_km, swim_pool_km against price_doc individually. The variables indicate the distance in kilometers of the facilities to properties.

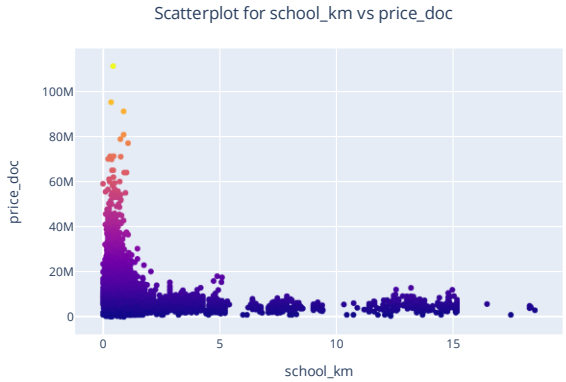


Figure 14: Scatterplot for school_km vs price_doc

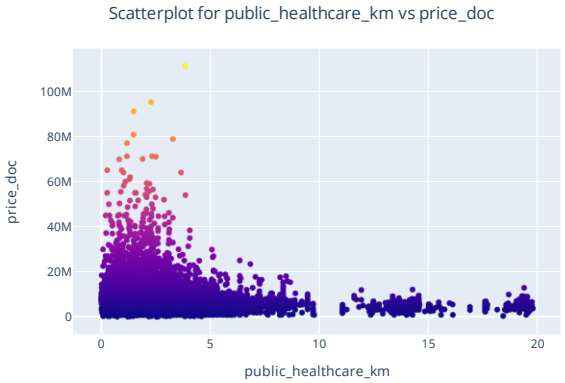


Figure 15: Scatterplot for public_healthcare_km vs price_doc

Figure 16: Scatterplot for shopping_centers_km vs price_doc

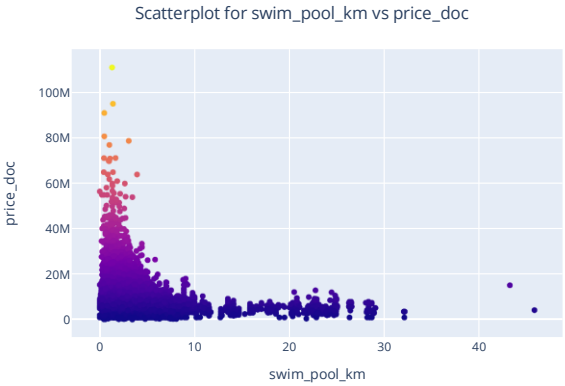


Figure 17: Scatterplot for swim_pool_km vs price_doc

Generally, the lower the distance to schools, public health care, shopping centers and swimming pools facilities, the pricier the property.

The following figures illustrate the scatterplots for the next five facilities, fitness_km, metro_km_walk, railroad_station_walk_km, raion_popul against price_doc against price_doc individually.

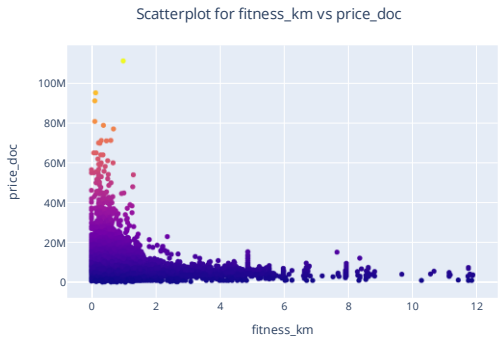


Figure 18: Scatterplot for fitness_km vs price_doc

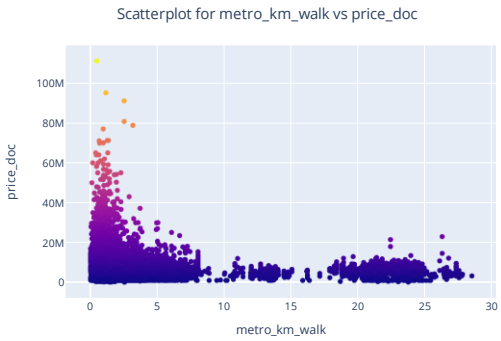


Figure 19: Scatterplot for metro_km_walk vs price_doc

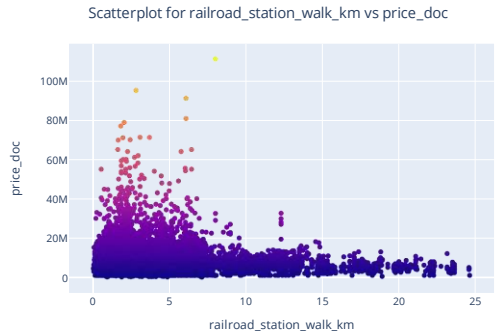


Figure 20: Scatterplot for railroad_station_walk_km vs price_doc

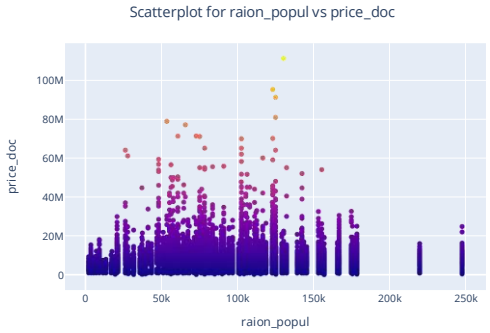


Figure 21: Scatterplot for raion_popul vs price_doc

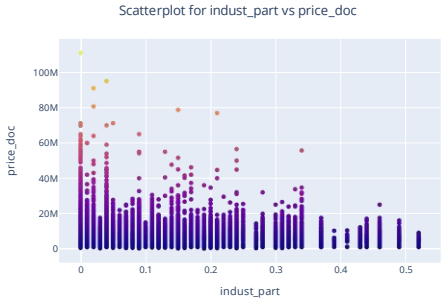


Figure 22: Scatterplot for indust_part vs price_doc

Generally, the lower the distance to fitness facilities, metro, railway stations, the pricier the property.

For raion_popul, when the population of a district is too dense or sparse, the prices are lower.

For indust_part, the smaller the proportion of industrial zones in the area, the higher the prices.

2.5 Correlation Analysis

2.5.1 Features of Corr >= 0.25 with Respect to Price

Since there are 292 columns in the training dataset, it is too large to visualize in one heat map. Thus, we have divided the data into portions based on the correlation (corr) value with respect to price to analyze part by part. The Pearson's Correlation is used.

The following figure illustrates the heat map of features with corr >= 0.25 with respect to price.

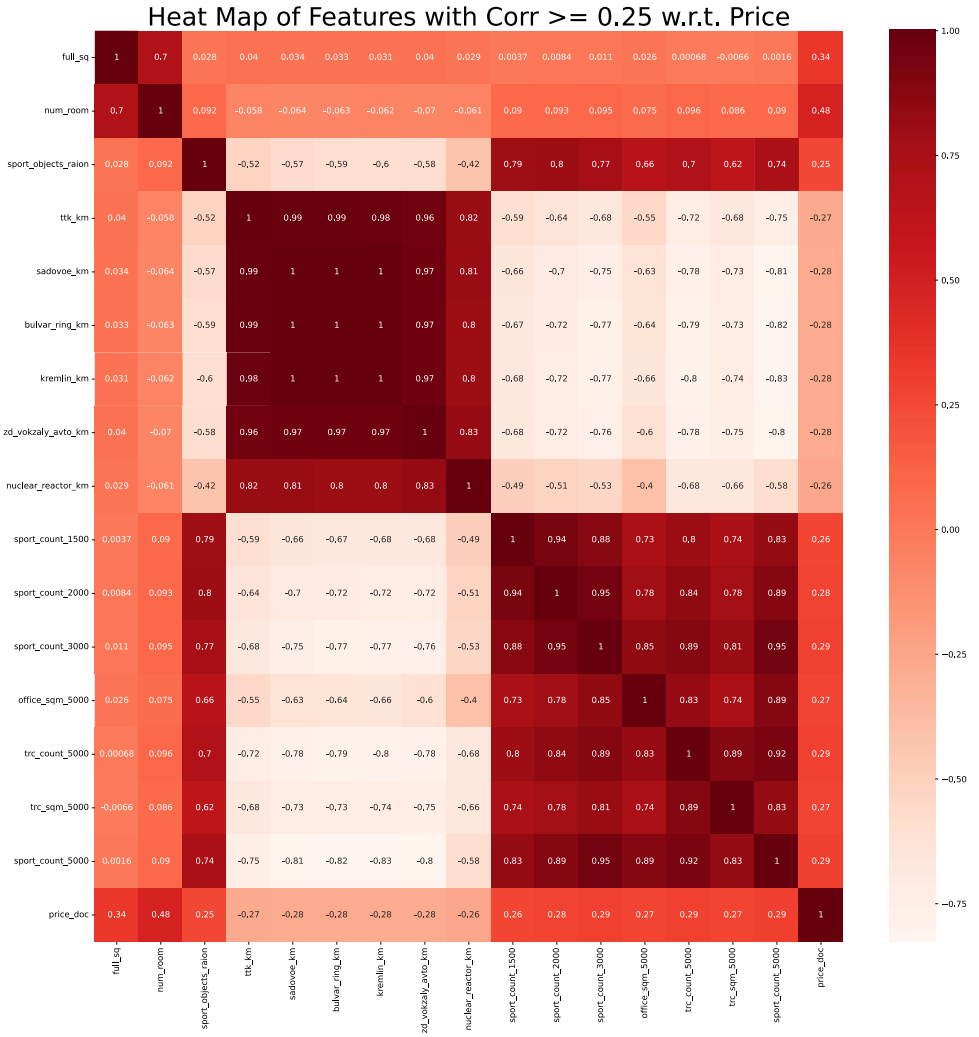


Figure 23: Heat Map of Features with Corr >= 0.25 w.r.t. Price

The first set of features that are highly correlated to each other are: ttk_km, sadovoe_km, bulvar_ring_km, kremlin_km, zd_vokzaly_avto_km, nuclear_reactor_km. (Top left maroon squares)

The second set of features that are highly correlated to each other are: sport_count_1500, sport_count_2000, sport_count_3000, office_sqm_5000,trc_count_5000, trc_sqm_5000, sport_count_5000. (Bottom right maroon squares)

The variables full_sq & num_room are highly correlated with price_doc with corr = 0.34 and 0.48 respectively. (Top leftmost corner)

2.5.2 Features of 0.20 <= Corr < 0.25 with Respect to Price

The following figure illustrates the heat map of features with the range of values: 0.20 <= corr < 0.25 with respect to price.

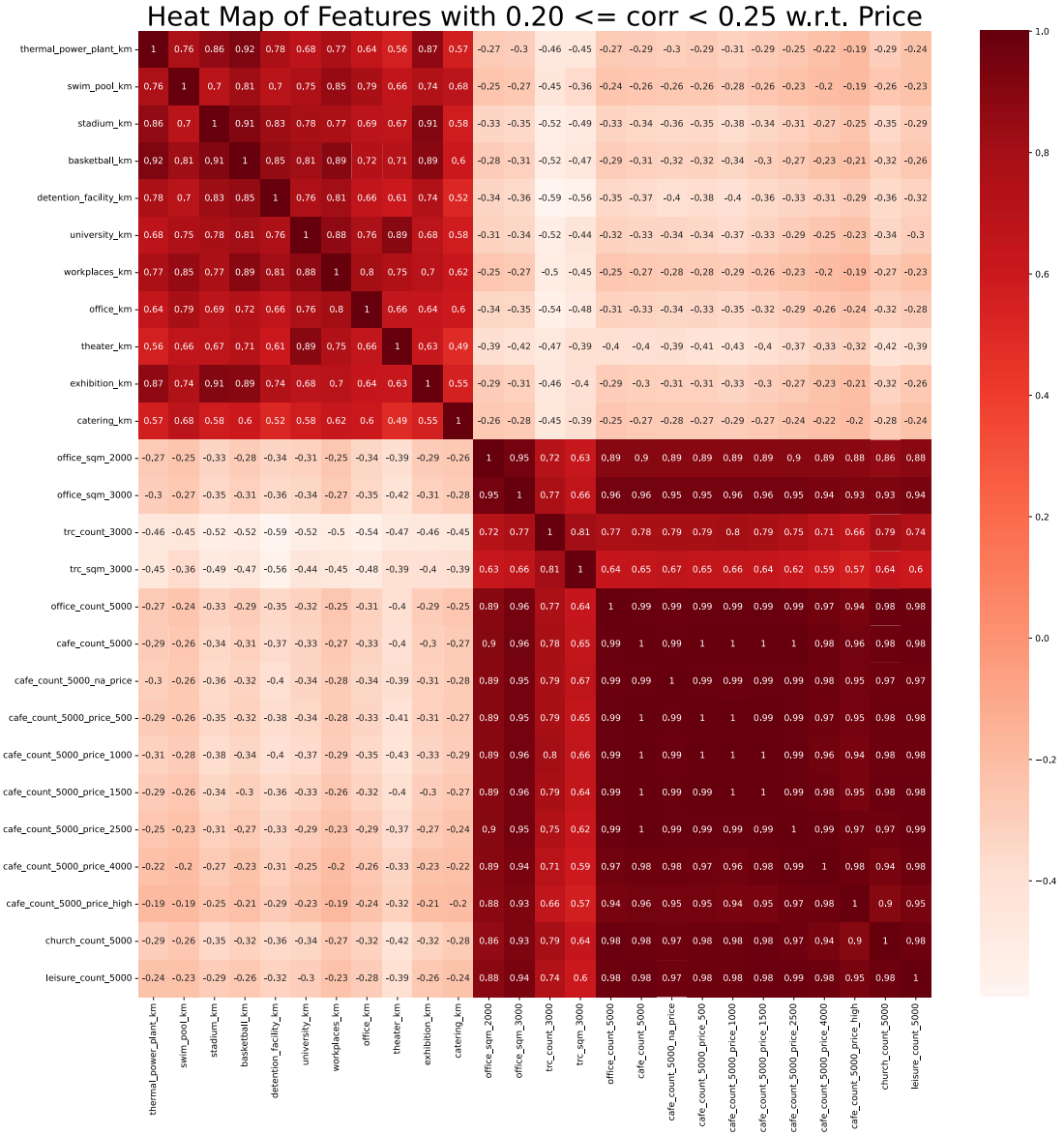


Figure 24: Heat Map of Features with 0.20 <= corr < 0.25 w.r.t. Price

The first set of features that are highly correlated to each other are: thermal_power_plant_km, swim_pool_km, stadium_km, basketball_km, detention_facility_km, university_km, workplaces_km, office_km, theater_km, exhibition_km, catering_km. (Top leftmost maroon squares)

The second set of features that are highly correlated to each other are: office_sqm_2000, office_sqm_3000, trc_count_3000, trc_sqm_3000, office_count_5000, cafe_count_5000, cafe_count_5000_na_price, cafe_count_5000_price_500, cafe_count_5000_price_1000, cafe_count_5000_price_1500, cafe_count_5000_price_2500, cafe_count_5000_price_4000, cafe_count_5000_price_high, church_count_5000, leisure_count_5000. (Bottom rightmost maroon squares)

2.5.3 Features of 0.15 <= Corr < 0.20 with Respect to Price

The following figure illustrates the heat map of features with the range of values: 0.15 <= corr < 0.20 with respect to price.

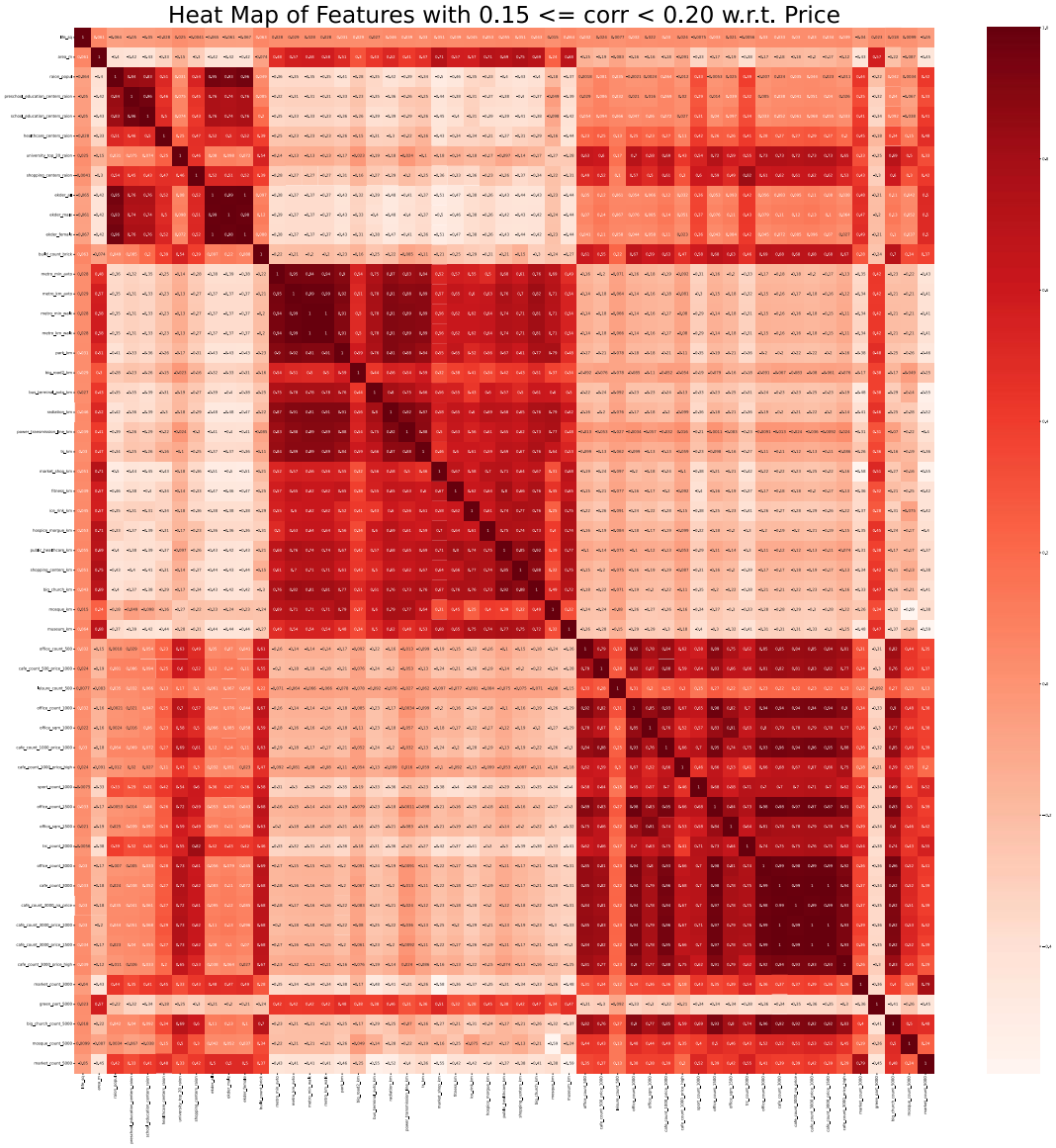


Figure 25: Heat Map of Features with 0.15 <= corr < 0.20 w.r.t. price

The first set of features that are highly correlated to each other are the features from raion_popul to ekder_female, with exceptions of those with low correlation levels. (Top leftmost maroon squares)

The second set of features that are highly correlated to each other are the features from metro_km_avto to museum_km, with exceptions of those with low correlation levels. (Middle maroon squares)

The third set of features that are highly correlated to each other are the features from office_count_500 to market_count_5000, with exceptions of those with low correlation levels. (Bottom rightmost maroon squares)

2.5.4 Features of 0.10 <= Corr < 0.15 with Respect to Price

The following figure illustrates the heat map of features with the range of values 0.10 <= corr < 0.15 with respect to price.

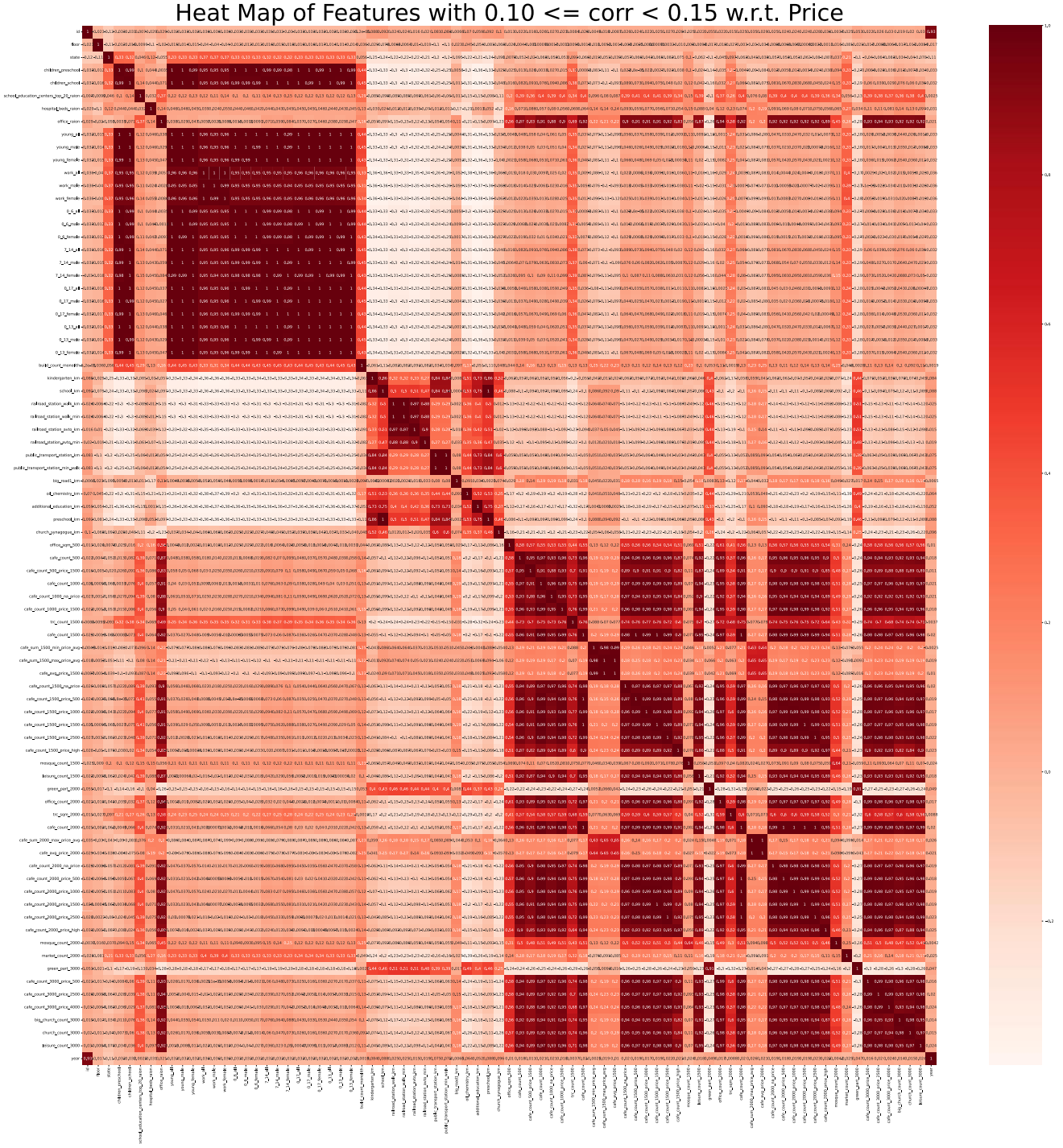


Figure 26: Heat Map of Features with 0.10 <= corr < 0.15 w.r.t. price

The first set of features that are highly correlated to each other are: children_preschool, children_school.

The second set of features that are highly correlated to each other are: young_all, young_male, young_female, work_all,work_male, work_female, 0_6_all, 0_6_male, 0_6_female, 7_14_all, 7_14_male, 7_14_female, 0_17_all, 0_17_male,0_17_female, 0_13_all, 0_13_male, 0_13_female.

The third set of features that are highly correlated to each other are: railroad_station_walk_km, railroad_station_walk_min,railroad_station_avto_km, railroad_station_avto_min.

The fourth set of features that are highly correlated to each other are: public_transport_station_km, public_transport_station_min_walk

The fifth set of features that are highly correlated to each other are: cafe_count_500, cafe_count_500_price_1500, cafe_count_1000, cafe_count_1000_na_price, cafe_count_1000_price_1500, trc_count_1500, cafe_count_1500.

The sixth set of features that are highly correlated to each other are: cafe_sum_1500_min_price_avg, cafe_sum_1500_max_price_avg, cafe_avg_price_1500

The seventh set of features that are highly correlated to each other are: cafe_count_1500_na_price, cafe_count_1500_price_500, cafe_count_1500_price_1000, cafe_count_1500_price_1500, cafe_count_1500_price_2500, cafe_count_1500_price_high.

The eighth set of features that are highly correlated to each other are: cafe_count_3000_price_500, cafe_count_3000_price_2500, cafe_count_3000_price_4000, big_church_count_3000, church_count_3000, leisure_count_3000.

We only look at features with corr > 0.1. This is because features with corr < 0.1 will not be of significance to the model we will be building. Thus, features with corr < 0.1 may be discarded.

2.6 Conclusion of EDA

The training data contains a lot of features with missing values, noise, erroneous data and potential anomalies present in the dataset. This could adversely affect the trained model. In addition, there is also a large chunk of features with high multicollinearity as the correlation values between features are very close to 1. With the presence of high collinearity between features in the dataset, this could negatively affect the trained model.

3 Challenges

From our exploratory data analysis, we have found three main challenges in our training dataset.

3.1 Features with Missing Values

The first challenge is the presence of features with lots of missing values. As mentioned previously, there are a total of 51 features out of 292 that contain missing values, and the first six variables have very high percentages of above 40% of missing values. When we have missing values, it can potentially cause a bias or reduce the accuracy of the models trained. As such, we must devise a way to handle missing values. For example, we can experiment with different techniques of missing values imputation using statistically sampled values such as mean, median or via other algorithms such as KNN Imputer.

3.2 Potential Anomalies

The second challenge is the presence of potential anomalies, noise, and erroneous data in the training dataset. As aforementioned in the Exploratory Data Analysis, there are some properties with extremely high prices. There are also properties with blatantly erroneous values, such as total size of a property being 0 or 1, or very small values that is not realistically possible. Thus, this is one aggravating factor that adversely affects the performance and accuracy of the trained models.

3.3 High Collinearity

The third challenge is the presence of multiple sets of features with high collinearity in the training dataset. When there are multiple sets of features with high correlation with each other, the trained model will capture these randomness or noise from collinear features. This can lead to the problem of overfitting, where the trained model will perform well on the training dataset but perform poorly on the unseen testing dataset. Thus, special consideration to account for this factor is required in the feature engineering stage.

4 Data Preparation

4.1 Data Pre-Processing: Cleaning of Data

The first step in our data preparation was to process and clean our datasets. We first list out distinct data types that were present amongst all the features. The main objective of this is to first determine if the data consists of only numerical types, namely integers and floats or if there were categorical types as well denoted by "numpy.dtype[object_]".

```
====== Dataset size - train: (30471, 292) ======
Number of distinct datatypes:
<class 'numpy.dtype[int64]'>      157
<class 'numpy.dtype[float64]'>    119
<class 'numpy.dtype[object_]'>     16
dtype: int64
```

Figure 27: Distinct Datatypes in Dataset

Next, for data cleaning we will be merging the train and test datasets first. We first extract a copy of the 'id' and 'price_doc' columns from the train dataset that will be used in train and test splitting later. The train dataset minus its 'price_doc' column is then merged with the test dataset to be cleaned together.

Since we are predicting house prices, we can drop features or columns that have low correlation to the price column. This is done using the function corr() and we drop all the features that have less than 5% correlation value. Following that, we also dropped columns that are IDs as they do not help in determining the price.

```
build_year                    0.0022
kitch_sq                      0.0287
school_quota                  0.0140
culture_objects_top_25_raion  0.0443
full_all                      0.0253
male_f                        0.0264
female_f                      0.0243
16_29_all                     0.0223
16_29_male                    0.0231
16_29_female                  0.0216
build_count_block             0.0315
build_count_wood              0.0425
build_count_frame             0.0303
build_count_panel             0.0201
build_count_foam              0.0107
build_count_slag              0.0240
build_count_mix               0.0330
build_count_1921-1945         0.0203
build_count_1971-1995         0.0097
build_count_after_1995        0.0259
cemetery_km                   0.0249
ID_railroad_station_walk      0.0218
water_km                      0.0266
mkad_km                       0.0206
big_market_km                 0.0483
prom_part_500                 0.0090
trc_sqm_500                   0.0004
cafe_sum_500_min_price_avg    0.0364
cafe_sum_500_max_price_avg    0.0379
cafe_avg_price_500            0.0374
cafe_count_500_na_price       0.0492
big_church_count_500          0.0262
church_count_500              0.0147
mosque_count_500              0.0185
market_count_500              0.0404
trc_sqm_1000                  0.0416
cafe_count_1000_price_4000    0.0363
prom_part_3000                0.0227
cafe_sum_3000_min_price_avg   0.0051
cafe_sum_3000_max_price_avg   0.0022
cafe_avg_price_3000           0.0033
cafe_sum_5000_min_price_avg   0.0322
cafe_sum_5000_max_price_avg   0.0333
cafe_avg_price_5000           0.0329
```

Figure 28: List of features with < 5% correlation to 'price_doc'

4.2 Feature Engineering

For feature engineering, we have extended the ‘timestamp’ feature into two new features: ‘year’ and ‘year_month’ which contain the year and month & year respectively.

From the created ‘year_month’ feature we get ‘sales_year_month’ using the groupby and size methods.

Using ‘life_sq’ and ‘full_sq’, we created three new features: ‘living_area_ratio’, ‘non_living_area’ and ‘non_living_area_ratio’ which data such as the ratio of living space and non-living space within the house.

From ‘life_sq’ and ‘num_room’ we can get ‘room_area_avg’ which describes the average area per room in the house.

Using ‘floor’ and ‘max_floor’, we can describe the floor compared to the number of floors in a building in relative terms with a new feature: ‘relative_floor’

We use mean aggregation of ‘max_floor’ by ‘sub_area’ to create ‘sub_area_building_height_avg’.

In the same way, we can create a feature describing the average distance to the Kremlin: ‘sub_area_kremlin_dist_avg’ using ‘sub_area’ and ‘kremlin_km’.

4.3 Missing Values Imputation

To begin, we first obtain a count of the number of columns with missing values in them. We have identified there are 36 columns that contains missing values.

```
hospital_beds_raion                    17859
room_area_avg                          14897
state                                  14253
relative_floor                         10355
max_floor                              10355
num_room                                9586
material                                9572
preschool_quota                         8284
cafe_sum_1000_min_price_avg             7746
cafe_avg_price_1000                     7746
cafe_sum_1000_max_price_avg             7746
non_living_area_ratio                   7562
non_living_area                         7562
living_area_ratio                       7562
life_sq                                 7559
raion_build_count_with_material_info    6209
build_count_brick                       6209
build_count_before_1920                 6209
raion_build_count_with_builddate_info   6209
build_count_monolith                    6209
build_count_1946-1970                   6209
cafe_avg_price_1500                     5020
cafe_sum_1500_min_price_avg             5020
cafe_sum_1500_max_price_avg             5020
cafe_sum_2000_max_price_avg             2149
cafe_sum_2000_min_price_avg             2149
cafe_avg_price_2000                     2149
prom_part_5000                           270
floor                                    167
metro_min_walk                            59
railroad_station_walk_km                  59
railroad_station_walk_min                 59
metro_km_walk                             59
product_type                              33
green_part_2000                           19
full_sq                                    3
dtype: int64

Missing Values Count: 36
```

Figure 29: List of Features Containing Missing Values

The main method used is median imputation for features that have less than 30% missing values. For numerical values the median values were used and as for any categorical data amongst these features the mode value was used. This is accounted for 33 out of the 36 columns.

For the remaining 3 columns with greater than 30% missing values, we used KNN Imputer.

```
hospital_beds_raion   0.4683
room_area_avg         0.3907
state                 0.3738
dtype: float64

Missing Values Count: 3
```

Figure 30: List of Remaining Features Containing Missing Values

The KNNImputer from sklearn.impute is used here for the remaining columns: 'hospital_beds_raion', 'room_area_avg' and 'state'.

4.4 Prepare Train and Test Datasets

At this stage, the data has been cleaned, and missing values resolved, and the feature engineering is completed. We then proceed to prepare and split the data into train and cross-validation sets. Since we have merged the train and test datasets at the start, we separated them. We can easily do this as we extracted a copy of the train dataset earlier containing the 'id' column.

Then from this extracted train dataset, we create yTrain which contains 'price_doc' and xTrain which we drop 'id', 'timestamp' and 'price_doc'.

Using sklearn's train_test_split we can obtain our training and cross-validation datasets.

```
# split into training and cross-validation
x_tr, x_cv, y_tr, y_cv = train_test_split(xTrain, yTrain, test_size=0.15,random_state=42)
```

Figure 31: Preparing Training and Cross-validation Data

For the final step we process the data as categorical or numerical using the label encoder from sklearn.preprocessing.

```
# categoricals
for cat in categoricals:
  le = preprocessing.LabelEncoder()
  le.fit(x_tr[cat])

  x_cv[cat] = x_cv[cat].map(lambda s: '<unknown>' if s not in le.classes_ else s)
  le.classes_ = np.append(le.classes_, '<unknown>')

  x_tr[cat] = le.transform(x_tr[cat])
  x_cv[cat] = le.transform(x_cv[cat])

# numericals
for num in numericals:
  min = x_tr[num].min()
  max = x_tr[num].max()
  x_tr[num] = (x_tr[num] - min)/(max-min)
  x_cv[num] = (x_cv[num] - min)/(max-min)
```

Figure 32: Processing Data using Label Encoder

After this has been completed, we are now ready to implement the various model approaches and methods.

5 Methodology

5.1 Approach 1 – Random Forest

5.1.1 Overview

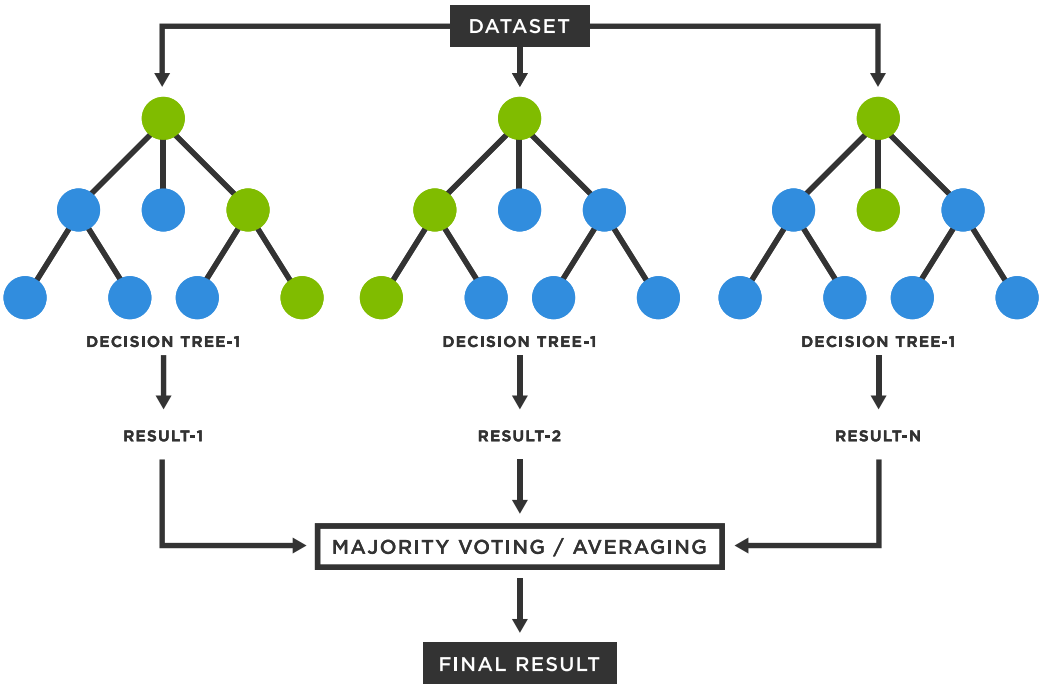


Figure 33: Random Forest [4]

Random Forest is a supervised learning algorithm based on ensemble learning, it is used for both classification and regression. It takes a subset of training data and a subset of features to build a decision tree. It builds multiple such decision trees and combines them together to get a more accurate and stable prediction. For regression problems, the prediction is the average prediction across the decision trees. For classification problems prediction is the majority vote class label across the decision trees.

5.1.2 Motivation

Random forest is considered a highly accurate and robust method because of the number of decision trees involved in the process. And it does not suffer from overfitting. The main reason being that it takes the average of all the predictions, which cancels out the biases. The random forest algorithm works well when the datasets have both categorical and numerical features.

5.1.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The SelectFromModel() function from sklearn is used to filter out the best features used for the model. The RandomForestRegressor() function from sklearn is used to construct the random forest regression model. The hyperparameter n_estimators and max_depth of the random forest must be tuned to obtain the best result. Hence RandomizedSearchCV() from sklearn is used. It runs through all the different parameters fed by the paramDist to produce the most optimal combination of parameters based on a scoring metric.

```
# define parameter distributions to use for tuning
paramDist = {
    'n_estimators': [50, 100, 150, 200, 250, 300],
    'max_depth': [10, 11, 12, 13, 14, 15]
}

# use RandomizedSearchCV to determine best values for parameters
randomSearchModel = RandomizedSearchCV(RandomForestRegressor(), param_distributions = paramDist, verbose
= 10, n_jobs = -1, cv = 2, n_iter = 15)
randomSearchModel.fit(trainFiltered, y_tr)
```

The most optimal hyperparameter for n_estimators is 300.

The most optimal hyperparameter for max_depth is 15.

Then the RandomForestRegressor() use these parameters and trains the final model.

5.1.4 Results

- Train MSE: 0.0878
- Test MSE: 0.225
- Train RMSE: 0.2962
- Test RMSE: 0.4743

5.2 Approach 2 – Decision Trees

5.2.1 Overview

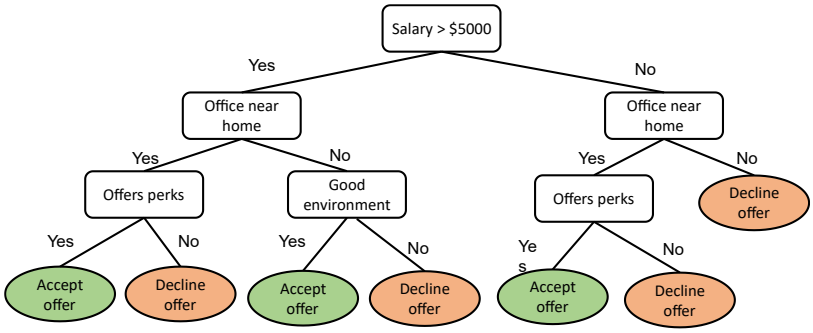


Figure 34: Decision Trees

Decision Trees are a supervised learning algorithm used for both classification and regression. The decision tree model learns simple decision rules inferred from data features to make prediction on the value of a target variable. In general, the deeper the tree, the more complex the decision rules and the better fitting the model.

5.2.2 Motivation

A Decision Tree is simple enough to understand and easy to interpret. It requires little data preparation. The cost of using the tree is logarithmic to the number of data points used to train the tree. Decision Trees can handle both numerical and categorical data. The decision tree model can be validated via statistical tests, thus accounting for the reliability of the model.

5.2.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The DecisionTreeRegressor() from sklearn is used to construct the decision tree regressor model. The hyperparameter max_depth represents the depth of the tree and must be selected by the user. To tune the hyperparameters for the model, sklearn's GridSearchCV is utilized. The technique runs through all the different parameters fed by the parameter grid to produce the most optimal combination of parameters based on a scoring metric. The following code shows the list of values for max_depth to be passed into the parameter grid for GridSearchCV.

```
dtModel = DecisionTreeRegressor()

# Define 'max_depth' params to use for tuning GridSearchCV
paramDepth ={'max_depth' : [5, 10, 11, 12, 13, 14, 15, 20, 25, 30, 35, 40, 45, 50]}

gridSearchModel = GridSearchCV(dtModel, param_grid = paramDepth, verbose = 10, cv = 3, n_jobs = -1)
gridSearchModel.fit(x_tr, y_tr)
```

The most optimal hyperparameter for max_depth is 5. The Decision Tree Regressor is tuned with the most optimal hyperparameter and fitted to the training data.

5.2.4 Results

- Train MSE: 0.2341
- Test MSE: 0.2449
- Train RMSE: 0.4838
- Test RMSE: 0.4949

5.3 Approach 3 – XGBoost

5.3.1 Overview

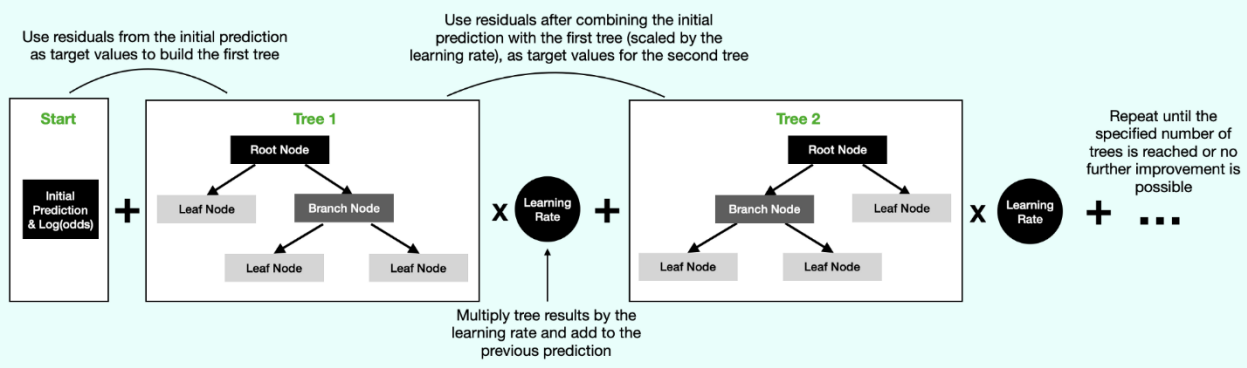


Figure 35: XGBoost [5]

Extreme Gradient Boosting (XGBoost) is a gradient boosting algorithm that is also based on ensemble learning, it is used for both regression and classification. Ensembles are constructed from decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. This is a type of ensemble machine learning model referred to as boosting. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm.

5.3.2 Motivation

Two main reasons we used XGBoost are its execution speed and model performance. XGBoost is fast when compared to most other implementations of gradient boosting, and it gives good prediction results.

5.3.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The XGBRegressor() function from sklearn is used to construct the XGBoost model. The hyperparameter colsample_bytree, learning_rate, max_depth, n_estimators and subsample of the XGBoost must be tuned to obtain the best results. Hence RandomizedSearchCV() from sklearn is used. It runs through all the different parameter distributions defined in paramDist to determine the most optimal combination of parameters based on the best score.

```
xgbModel = XGBRegressor(nthread = -1)

# define parameter distributions to use for tuning
paramDist = {
    "colsample_bytree": [0.1, 0.25, 0.3, 0.5, 0.75, 0.8, 1],
    "learning_rate": [0.01, 0.03, 0.05, 0.07, 0.1, 0.13, 0.15],
    "max_depth": [10, 11, 12, 13, 14, 15],
    "n_estimators": [50, 100, 150, 200],
    "subsample": [0.1, 0.3, 0.5, 0.7, 1]
}

# use RandomizedSearchCV to determine best values for hyperparameters
randomSearchModel = RandomizedSearchCV(xgbModel, param_distributions = paramDist, verbose = 10, n_jobs =
-1, cv = 2, n_iter = 15)
randomSearchModel.fit(x_tr, y_tr)
```

The most optimal value for ‘colsample_bytree’ is 0.25.

The most optimal value for ‘learning_rate’ is 0.1.

The most optimal value for ‘max_depth’ is 15.

The most optimal value for ‘n_estimators’ is 100.

The most optimal value for ‘subsample’ is 1.

These hyperparameter values will be used to train the final XGBRegressor model.

5.3.4 Results

- Train MSE: 0.0063
- Test MSE: 0.2355
- Train RMSE: 0.0794
- Test RMSE: 0.4853

5.4 Approach 4 – SGD Regressor

5.4.1 Overview

Stochastic Gradient Descent (SGD) is a learning model that can be used for both linear classifiers and regressors. It is efficient in fitting linear classifiers and regressors under convex loss functions, examples of which include linear SVMs and logistic regression.

5.4.2 Motivation

SGD is efficient and easy to implement with ease of parameter tuning. It has also been successful when applied to large-scale machine learning problems and can easily scale up to problems with large training and feature size.

5.4.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The SGDRegressor() function from sklearn is used to construct the SGD model. The hyperparameters 'alpha' and 'max_iter' must be tuned to obtain the best result. Hence RandomizedSearchCV() from sklearn is used. It fits a total of 3 folds of 15 iterations using the defined parameter distributions and returns the most optimal combination of parameters to use based on the best score.

```
sgdModel = SGDRegressor()

# define parameter distributions to use for tuning
paramDist = {
    "alpha": [1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4],
    "learning_rate": ['optimal'],
    "loss": ["squared_loss"],
    "max_iter": [500, 1000, 1500, 2000],
    "penalty": ["l2"]
}

# use RandomizedSearchCV to determine best values for hyperparameters
randomSearchModel = RandomizedSearchCV(sgdModel, param_distributions = paramDist, verbose = 10, n_jobs =
-1, cv=3, n_iter = 15)
randomSearchModel.fit(x_tr, y_tr)
```

The most optimal value for 'alpha' and 'max_iter' are 1e4 and 1000 respectively.

5.4.4 Results

- Train MSE: 91.1125
- Test MSE: 90.4085
- Train RMSE: 9.5453
- Test RMSE: 9.5083

From our findings, we discovered that SGD Regressor has the worst model performance due to its extremely high MSE and RMSE scores.

5.5 Approach 5 – AdaBoost

5.5.1 Overview

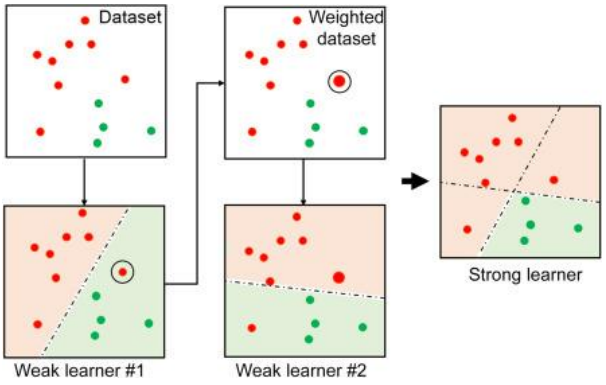


Figure 36: AdaBoost [6]

Adaptive Boosting (AdaBoost) is an ensemble learning method that makes use of an iterative algorithm that learns and avoids mistakes from weak classifiers to build a stronger version. Examples of these weak classifiers are decision stumps which are very much like the trees in Random Forest except it only has one node and two leaves, therefore making it weak. By creating a stump for each variable, AdaBoost iteratively classifies and adjusts the weights forming a larger stronger classifier.

5.5.2 Motivation

AdaBoost is very easily implemented and can be used without needing to tweak a lot of parameters unlike other algorithms. It is also not prone to overfitting.

5.5.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The AdaBoostRegressor() function from sklearn is used to construct the AdaBoost model. The hyperparameter 'n_estimators' will be tuned using RandomizedSearchCV() from sklearn. It fits a total of 3 folds of 6 iterations using the defined distribution of 'n_estimators' and returns the most optimal value to use to 'n_estimators' based on the best score.

```
adbModel = AdaBoostRegressor()

# define 'n_estimators' distributions to use for tuning
paramDist = { "n_estimators": [50, 100, 150, 200, 250, 300] }

# use RandomizedSearchCV to determine best value for 'n_estimators'
randomSearchModel = RandomizedSearchCV(adbModel, param_distributions = paramDist, verbose = 10, n_jobs =
-1, cv = 3)
randomSearchModel.fit(x_tr, y_tr)
```

The most optimal value for 'n_estimators' is determined to be 50.

5.5.4 Results

- Train MSE: 0.5598
- Test MSE: 0.5538
- Train RMSE: 0.7482
- Test RMSE: 0.7442

5.6 Approach 6 – LightGBM

5.6.1 Overview

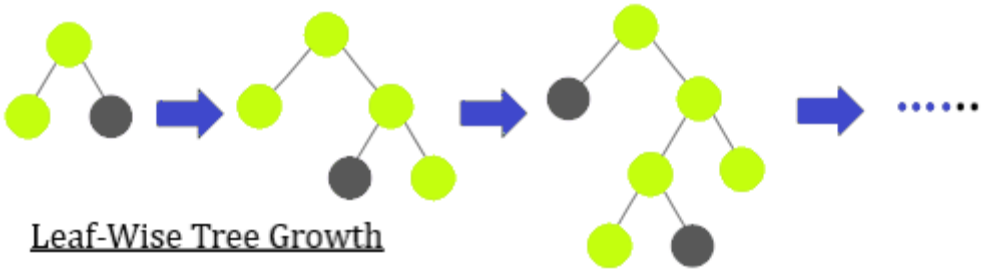


Figure 37: LightGBM [7]

Light Gradient Boosting Machine (LightGBM) is a gradient boosting model for machine learning that uses tree-based algorithms.

5.6.2 Motivation

LightGBM has exceptional training speed and efficiency on large-scale data and has very good accuracy. It is competitive with some of the better algorithms out there and has proven results in various machine learning problems.

5.6.3 Experiments

The data has been pre-processed and feature engineering has been performed and missing values are imputed as described in Section 4. We have opted to use 85% of the original training dataset for training. The model was imported from lightgbm. The hyperparameters were tested with some default values and the results were satisfactory, with consistently low MSE and RMSE scores, and decent Kaggle score.

5.6.4 Results

- Train MSE: 0.1414
- Test MSE: 0.2117
- Train RMSE: 0.3760
- Test RMSE: 0.4601

Kaggle Public Score: 0.32355

5.7 Approach 7 – Naive XGBoost with Raw Data

5.7.1 Overview

While our XGBoost and LightGBM results were better compared to the results of the other models, the scores were not good compared to the scores on the Kaggle leaderboard. Hence, we wanted to test if our data pre-processing was affecting the score. So, we tried predicting the raw train data without pre-processing on the Naïve XGBoost model.

5.7.2 Research

From our research in the Kaggle discussion boards, we found that the community discovered a set of so-called 'magic numbers', which will improve the prediction score [8]. This is due to the community finding out that it is highly possible the test set that the Kaggle competition is using has a lower average price than what most models will be able to predict with the provided train set, as they found the train set to have included a lot of possible fake prices which will affect the predictions. Additionally, it is difficult for models to predict these fake prices, as the community was only able to deduce these fake prices from external research and data scraping of Russian property prices. Hence, they found that reducing the price values before training will improve the prediction on the Kaggle test set.

```
xtrainnxgb = dfs["train"]
xtrainnxgb=xtrainnxgb[(xtrainnxgb.price_doc>1e6) & (xtrainnxgb.price_doc!=2e6) & (xtrainnxgb.price_doc!=3e6)]
xtrainnxgb.loc[(xtrainnxgb.product_type=='Investment') & (xtrainnxgb.build_year<2000),'price_doc']*=0.895
xtrainnxgb.loc[xtrainnxgb.product_type!='Investment','price_doc']*=0.96
```

5.7.3 Experiments

The data was taken from the train set, then processed to fit into RandomSearchCV for hyperparameter tuning. Six hyperparameters, "colsample_bytree", "learning_rate", "max_depth", "n_estimators", "subsample", "min_child_weight", were tuned with RandomSearchCV with 2 folds over 20 iterations. The best values for the hyperparameters were then used for the Naïve XGBoost model. We tested with just the raw data, and then with the magic numbers.

5.7.4 Results

Without magic numbers:

Public score: 0.31978

With magic numbers:

Public score: 0.31275

5.8 Approach 8 – LightGBM with Raw Data

5.8.1 Overview

Since the Naïve XGBoost model showed an improvement when predicting the raw train data, we decided to try it on our LightGBM model as well as it was the next best performer on the pre-processed data.

5.8.2 Experiments

Like our previous LightGBM model, we took the default hyperparameter values we tested with previously as they gave good results. The data was taken from the train set, then some minor processing to remove some outliers to prevent overfitting. We tested with just the processed data, and then with the magic numbers.

5.8.3 Results

Without magic numbers:

Public score: 0.32789

With magic numbers:

Public score: 0.31041

5.9 Conclusion of Results

Model	Train MSE	Test MSE	Train RMSE	Test RMSE
Random Forest	0.0878	0.2250	0.2962	0.4743
Decision Tree	0.2341	0.2449	0.4838	0.4949
XGBoost	0.0063	0.2355	0.0794	0.4853
SGD	91.1125	90.4085	9.5453	9.5083
AdaBoost	0.5598	0.5538	0.7482	0.7442
LightGBM	0.1414	0.2117	0.3760	0.4601

Table 9 Model Performance on Train and Cross-Validate data

XGBoost, Random Forest and LightGBM were the better performing models on our train and cross validation datasets with the lowest MSE and RMSE scores. SGD had a terrible performance with extremely high MSE and RMSE scores. However, when predicting on the test.csv data, we found that our models did not perform as well as we had expected.

Hence, we decided to try predicting on a very minimally processed dataset as we found out that our pre-processing could have negatively impacted the scores. We found that the minimally processed datasets gave much better results than our pre-processed datasets. Additionally, from our research on the Kaggle discussion board, we found some magic values that could improve the prediction of our models [8]. Hence, we tested those values on our Naïve XGBoost and LightGBM model with the minimally processed dataset and found the scores to have improved. Thus, we took the best score out of those two models, which is the LightGBM model with 0.31041 public score.

6 Solution Novelty

6.1 Overview

We wanted to try and come up with a model that could improve on our existing models. We approached this using Regression Decision Tree and sampling.

Firstly, we split our train dataset into an 80-20 split. Then we take the split train set and split it further into a 50-50 split, called A1 and A2. We use A1 to do sampling with replacement to create ‘k’ samples. Now that we have k samples, we create k models and train each of these models with the k samples using Regression Decision Tree.

With the k models, we pass the A2 set to each model to get k predictions for A2 set from each model. We use these k predictions to create a new dataset, and using A2’s target values, we train a new meta model.

For the evaluation, we use the 20% test set we split earlier, and pass it to each base model to get ‘k’ predictions. We create another new dataset with these k predictions, then pass it to the meta model we trained earlier, to get the final prediction.

We then calculate the performance of the meta model, by using the final prediction as well as the targets for the test set.

6.2 Result

Test MSE: 0.2323

Kaggle public score: 0.36742

While the Test MSE result is decent relative to our other models, the final Kaggle public score is still much worse compared to several of our models.

7 Leaderboard

The following table illustrates the methods used, the private and public score, screenshots of the submission results, and the public rank of our approaches.

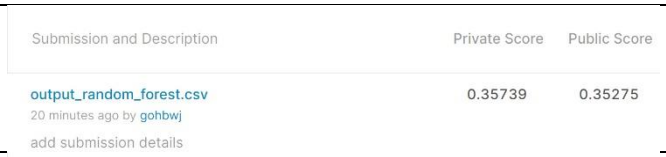
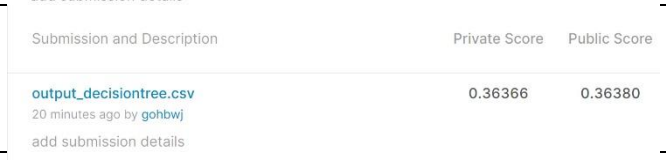
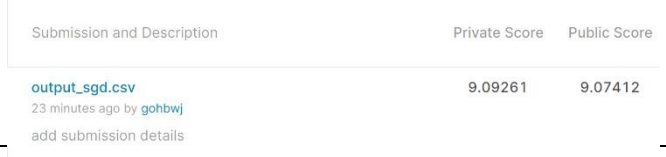
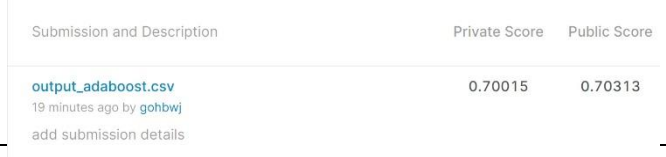
Model	Private Score	Public Score	Screenshot	Public Rank (Out of 3265)
Random Forest	0.35739	0.35275		2668
Decision Tree	0.36366	0.36380		2734
XGBoost	0.37180	0.36423		2735
SGD Regressor	9.09261	9.07412		3260
AdaBoost	0.70015	0.70313		3220
LightGBM	0.32451	0.32355		1831
Naïve XGBoost Raw	0.32162	0.31741		1384
LightGBM Raw	0.31443	0.31041		532
Custom Model	0.36969	0.36742		2751

Table 10: Leaderboard Scores and Ranking

8 Learning Points

8.1 Importance of Exploratory Data Analysis

The process of Exploratory Data Analysis is of paramount importance and cannot be neglected. We were able to identify all the challenges of working with this dataset, such as the presence of many features with missing values, noisy data, potential anomalies, and many features with high collinearity. To resolve these challenges, we were required to perform data cleaning, missing data imputation using various techniques, and feature engineering to overcome these challenges.

8.2 Influence of Data Pre-Processing

The method and steps taken to process the data can wildly affect the resulting scores of the various models. The methods that make sense in theory to carry out may in fact sometimes degrade the performance of the models as demonstrated in our various approaches above. We have discovered that our heavily pre-processed datasets might have been the reason why we could not improve our scores further.

One possible reason is that pre-processing the data based on our exploratory analysis may have caused our models to overfit on the Train set. This can be seen by comparing the Train and Test RMSE scores to the Kaggle public score of each model, which shows they do better on the provided Train and Test set compared to the Kaggle hidden Test set, which overall fares much worse.

We also suspect there is a possibility that there are fake data in the Train set, which was found by the Kaggle community discussion boards. Thus, this is another factor that caused model overfitting.

Hence, due to these problems, we decided to perform a simpler approach for the pre-processing. With the minimally pre-processed data, we used the Naïve XGBoost and LightGBM models and obtained better results. Thus, we have learnt not to rely solely on the pre-processed data, but to understand the inherent problems associated with the dataset and take a more open-minded approach by running the prediction models on minimally processed data first and compare the performances.

9 Conclusion

From this Kaggle competition, our team have gained valuable insights and a deeper understanding of the processes and workflow in a machine learning task.

Firstly, exploratory data analysis is an important step that allows for better understanding of the dataset before working with it. It provides context on how some features relate to others and can even point out potential key features that can help in predictions. We also gained a comprehensive overview of the dataset and identified issues that could pose as challenges in our model training process, such as the presence of missing values, noisy or erroneous data, potential anomalies, and many features with high collinearity. All these challenges required us to think deeply on how to they are required to be handled in the data pre-processing stage.

Secondly, data pre-processing is an imperative step that cannot be neglected. In this project, we have cleaned and prepared the dataset by handling missing values and dropping superfluous columns. We also performed feature engineering by creating new features from existing ones. Though in our case, we have discerned from our initial model predictions that the data pre-processing can greatly influence the performance of the predictions both for better or worse, depending on how complete the provided Train dataset is due to finding possible fake data in the Train data which may have caused overfitting, and how different the provided Test set is to the Kaggle hidden Test set.

In this project, our team has utilized a total of 6 models: Random Forest, Decision Tree, XGBoost, SGD, AdaBoost and LightGBM. The hyperparameters of the models were tuned using various techniques and the results of the models were compared and analyzed. We have also tested and compared the results of our models on datasets with varying levels of pre-processing. From our analysis and findings, we have found LightGBM to be the best performing model on both the pre-processed and minimally processed data. The result from the minimally processed dataset is better with a public score of 0.31041, ranking at 532 out of 3265 which puts us amongst the top 17%.

In conclusion, through this Kaggle competition, our team has gained a deeper understanding of the entire machine learning workflow, starting from the exploratory data analysis, data preparation such as data pre-processing, feature engineering, model hyperparameter tuning, model training, and model evaluation. We have also explored the effects of different levels of data pre-processing and understood there are inherent problems associated with the dataset that conventional machine learning approaches may not guarantee optimal results. All of these have provided us with better insights on how we can approach different machine learning problems in the future.

References

[1] Sberbank, ‘Sberbank Russian Housing Market’, 2017. https://www.kaggle.com/c/sberbank-russian-housing-market/ (accessed Feb. 03, 2022).

[2] Harun, ‘Metrics - MSE, R^2, RMSLE’, 2019. https://hrngok.github.io/posts/metrics/ (accessed Feb. 03, 2022).

[3] Y. Kozhevnikova, ‘Moscow’s real estate market goes from boom to bust’, *Inman*, 2015. https://www.inman.com/2015/10/23/moscows-real-estate-market-goes-from-boom-to-bust/ (accessed Feb. 26, 2022).

[4] TIBCO, ‘What is a Random Forest?’ https://www.tibco.com/reference-center/what-is-a-random-forest (accessed Apr. 08, 2022).

[5] S. Dobilas, ‘XGBoost: Extreme Gradient Boosting — How to Improve on Regular Gradient Boosting?’, *Towards Data Science*, 2021. https://towardsdatascience.com/xgboost-extreme-gradient-boosting-how-to-improve-on-regular-gradient-boosting-5c6acf66c70a (accessed Apr. 08, 2022).

[6] S. P. Abirami, G. Kousalya, Balakrishnan, and R. Karthick, ‘Chapter 14 - Varied Expression Analysis of Children With ASD Using Multimodal Deep Learning Technique’, in *Deep Learning and Parallel Computing Environment for Bioengineering Systems*, A. K. Sangaiah, Ed. Academic Press, 2019, pp. 225–243.

[7] GeeksforGeeks, ‘LightGBM (Light Gradient Boosting Machine)’, 2021. https://www.geeksforgeeks.org/lightgbm-light-gradient-boosting-machine/ (accessed Apr. 08, 2022).

[8] A. Chavakula, ‘What’s this “Magic numbers” business?’, *Kaggle*, 2017. https://www.kaggle.com/c/sberbank-russian-housing-market/discussion/35044 (accessed Apr. 23, 2022).