

Analysis and prediction of rental house prices in Bucharest, September 2020

Serdal Aslantas

1. Introduction

This project is about analyzing and predicting rental prices of houses in Bucharest based on the data collected in September 2020 from the official website of the real estate agency [imobiliare.ro](https://www.imobiliare.ro). The size of the dataset is 1.25 MB. These predictions can be used as a reference by the investors and business owners, as well as housing project companies.

The project is completed in five steps:

1. Collecting the dataset
2. Exploratory Data Analysis
3. Cleaning the dataset
4. Training different models
5. Conclusion

The details of each step are explained under the respective parts.

Choosing the models, we kept in mind the fact that we need to solve an actual regression problem. The output variable (the price) is a real and continuous value while all the other variables need to be independent of each other but dependent only on the price.

2. The Dataset

2.1. Data Structure

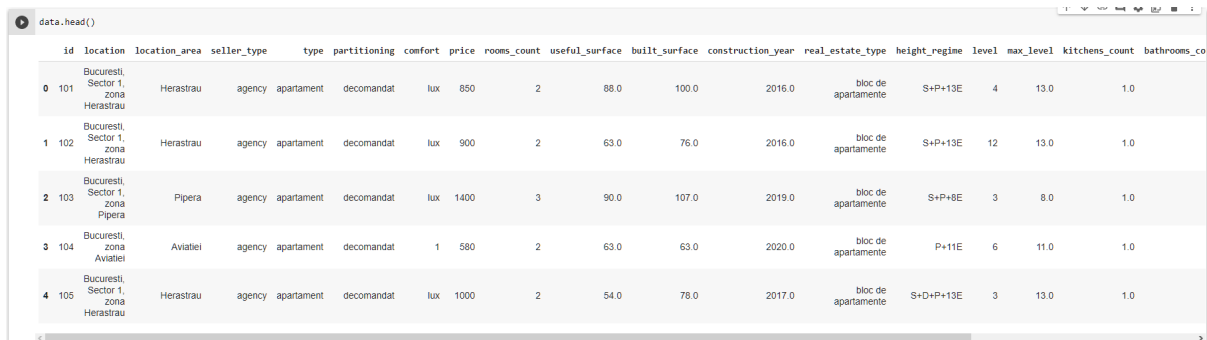
The original dataset employed in this study can be found on [kaggle](https://www.kaggle.com), and it contains data that has been collected from [imobiliare.ro](https://www.imobiliare.ro) by using a personal web scraper implemented in *Node.js*. The dataset presents different details extracted from renting house advertisements, like location area, house comfort, number of rooms, construction year etc. All of the information has been collected during September 2020, for Bucharest, Romania.

In this dataset we can find two similar tables which contain data about rental prices and full prices of the houses, however, in this study we will look at rental prices only. There are 9806 examples of houses for rent, each containing 21 features. All of these features can be found in **Appendix, Table A.1**, where you can observe their data type and meaning.

Bear in mind however, that these will not represent the features to be used in the model training. This is just the original data, which has many missing and messy values, therefore it would be mandatory to perform some data preprocessing.

2.2. Data Analysis

After reading the data we used the method `head()`. This method is good to see if the lines and the columns are in order and there are no errors reading it.



	id	location	location_area	seller_type	type	partitioning	comfort	price	rooms_count	useful_surface	built_surface	construction_year	real_estate_type	height_regime	level	max_level	kitchens_count	bathrooms_count
0	101	Bucuresti, Sector 1, zona Herastru	Herastru	agency	apartament	decomandat	lux	850	2	88.0	100.0	2016.0	bloc de apartamente	S+P+13E	4	13.0	1.0	
1	102	Bucuresti, Sector 1, zona Herastru	Herastru	agency	apartament	decomandat	lux	900	2	63.0	76.0	2016.0	bloc de apartamente	S+P+13E	12	13.0	1.0	
2	103	Bucuresti, Sector 1, zona Pipera	Pipera	agency	apartament	decomandat	lux	1400	3	90.0	107.0	2019.0	bloc de apartamente	S+P+8E	3	8.0	1.0	
3	104	Bucuresti, Sector 1, zona Aviatiei	Aviatiei	agency	apartament	decomandat	1	580	2	63.0	63.0	2020.0	bloc de apartamente	P+11E	6	11.0	1.0	
4	105	Bucuresti, Sector 1, zona Herastru	Herastru	agency	apartament	decomandat	lux	1000	2	54.0	78.0	2017.0	bloc de apartamente	S+D+P+13E	3	13.0	1.0	

Taking a look at the data shown above, we can easily see that there are some columns (`location`, `location_area`, `seller_type`, `type`, `partitioning`, `comfort`, `real_estate_type`) that are represented through text and it is clear for us that we will need to do some preprocessing on them in order to parse them into numerical data.

Keeping that in mind, we moved forward into finding out if the data has any unusable rows or rows with `NaN` values that will need further complex preprocessing. We decided to use the method `isnull()` by making the sum of all the `NaN` values found on each column as follows:

id	0
location	0
location_area	1
seller_type	0
type	0
partitioning	172
comfort	186
price	0

```
rooms_count          0
useful_surface       46
built_surface        994
construction_year    1875
real_estate_type     37
height_regime        239
level                0
max_level            251
kitchens_count       3087
bathrooms_count      411
garages_count        9084
parking_lots_count   7326
balconies_count      4073
```

By observing the output, we can see that most of the columns (i.e. *partitioning*, *comfort*, *useful_surface*, *build_surface*, *construction_year*, *real_estate_type*, *height_regime*, *max_level*, *kitchens_count*, *bathrooms_count*, *garages_count*, *parking_lots_count* and *balconies_count*) will need further preprocessing. The steps for the preprocessing of these values can be consulted on **2.3 Data Cleaning**. We decided not to delete the rows with missing values, but rather to complete them with either a mean or a mode, depending on the context of that variable.

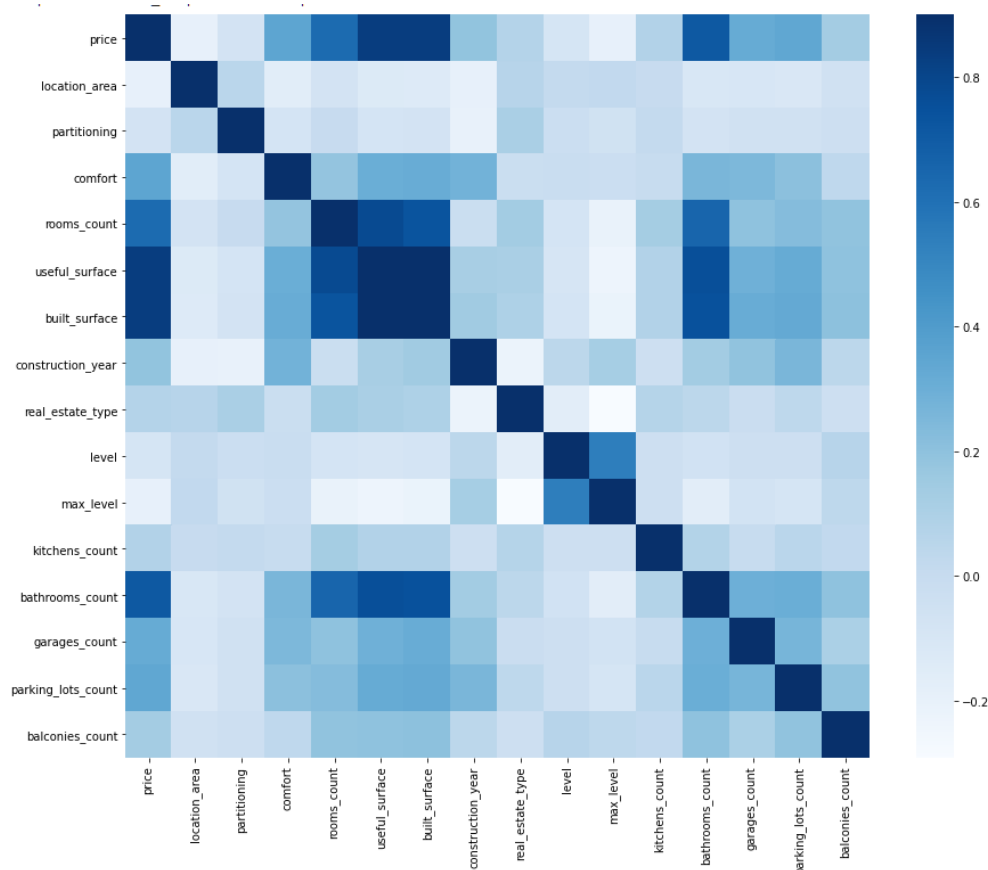
Further analysis was made with the help of the method *describe()*. We used that method in order to analyse the distribution of the data (how many rows are usable at first, the *min* and *max* of each column, and the first, second and third quartile)

	id	price	rooms_count	useful_surface	built_surface	construction_year	max_level	kitchens_count	bathrooms_count	garages_count	parking_lots_count	balconies_count
count	9806.000000	9806.000000	9806.000000	9760.000000	8812.000000	7931.000000	9555.000000	6719.000000	9395.000000	722.000000	2480.000000	5733.000000
mean	16756.294412	695.909443	2.312666	71.160246	82.549478	1995.563359	7.264260	1.005358	1.349548	1.081717	1.150806	1.231467
std	9593.031139	1808.575876	0.982632	38.682463	51.445611	21.709828	3.055412	0.179231	0.622316	0.284063	0.555002	0.625388
min	101.000000	24.000000	1.000000	6.000000	1.000000	1880.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
25%	8513.250000	360.000000	2.000000	50.000000	55.000000	1983.000000	5.000000	1.000000	1.000000	1.000000	1.000000	1.000000
50%	16803.500000	480.000000	2.000000	60.000000	68.000000	1997.000000	8.000000	1.000000	1.000000	1.000000	1.000000	1.000000
75%	25023.750000	700.000000	3.000000	80.000000	90.000000	2016.000000	10.000000	1.000000	2.000000	1.000000	1.000000	1.000000
max	33410.000000	139000.000000	22.000000	500.000000	600.000000	2020.000000	52.000000	11.000000	11.000000	3.000000	15.000000	22.000000

The purpose of this analysis is to determine and eliminate the outliers that might appear in the dataset. For further data cleaning, the columns *price*, *rooms_count*, *kitchens_count*, *bathrooms_count*, *parking_lots* and *balconies_count* came to our attention.

Another important issue that we needed to take into consideration when analysing the data was the data correlation. In order to do that, we plotted how the features are correlated to each other (in order to avoid multicollinearity). We need to observe how each feature is

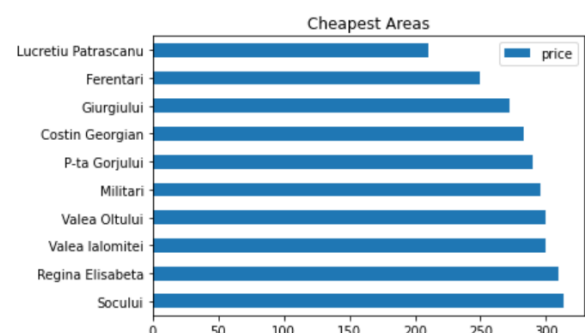
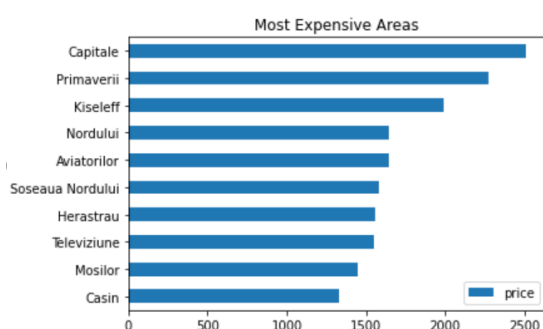
correlated to the price in order to find the most relevant columns. We used the following correlation matrix:



We can see that *price* is strongly correlated to *comfort*, *rooms_count*, *useful_surface*, *built_surface*, *bathrooms_count*, *garages_count* and *parking_lots_count*, judging on the darker hues depicted in this plot.

For an ideal model, all the independent variables are correlated to the dependent variable and are not correlated to each other. In this case we can observe the fact that *build_surface* and *useful_surface* are strongly correlated to each other, meaning that we should only keep one of them.

We also posed the question on what are the top 10 most expensive/cheapest areas in Bucharest in terms of rental prices. The results can be seen in the plots below.



2.3. Data Cleaning

As mentioned in the previous sections, in order to use the rental offers dataset for this task, we need to complete and clean the data. This step is made according to our general knowledge about the rental estate market.

The dataset contains 21 columns (described in **Appendix, Table A.1**), which present empty values and also useless information. The first step in order to clean the dataset is to eliminate those profitless columns: *id*, *location*, *seller_type*, *type*, *height_regime*; we have also deleted the *price* column from the features dataset, since it will be used for predictions.

The next step in cleaning the data is to complete the *null* values and this implies making assumptions about each situation. For the columns *garages_count*, *parking_lots_count*, *balconies_count*, if there is an empty value, we will complete it with the value **0**, assuming the fact that those apartments do not include that feature in the offer.

The column *level* may also contain strings as: **Parter**, **Ultimele doua etaje**, **Mansarda**, or **Demisol**. In order to replace them we have found a correlation between numbers and their value: **Demisol** \rightarrow **-1**, **Parter** \rightarrow **0**, **Ultimele doua etaje** \rightarrow (*last_floor*-1), **Mansarda** \rightarrow *last_floor*; where *last_floor* is the value extracted from *max_level*.

To complete the missing values from the column *max_level* we will calculate the mean height of the apartments in the same *location*. In case that there is no other offer in that specific area, we set the value of *max_level* to the value of *level*.

Location_area column does not contain null values; in this case we map each value of this column to an integer number (for example **Herastrau** \rightarrow **0**, **Floreasca** \rightarrow **1**, etc.)

There are some *null* values on the column *construction_year*; here we make the assumption that in the same area (*location_area*) all of the apartments have been built in the same period of time, so we calculate the mean of the *construction_year* for each area, and complete the missing values with the mean corresponding to their area.

For the missing values in the columns *build_surface* and *useful_surface* we use the following procedure: if only one value is missing, we complete it with the value of the other one; if both are missing, we calculate the average of *build_surface* and, respectively, the average of *useful_surface* along the entire dataset, and replace the empty values with the corresponding result. Since these two columns are dependent on each other, we will drop *build_surface* column.

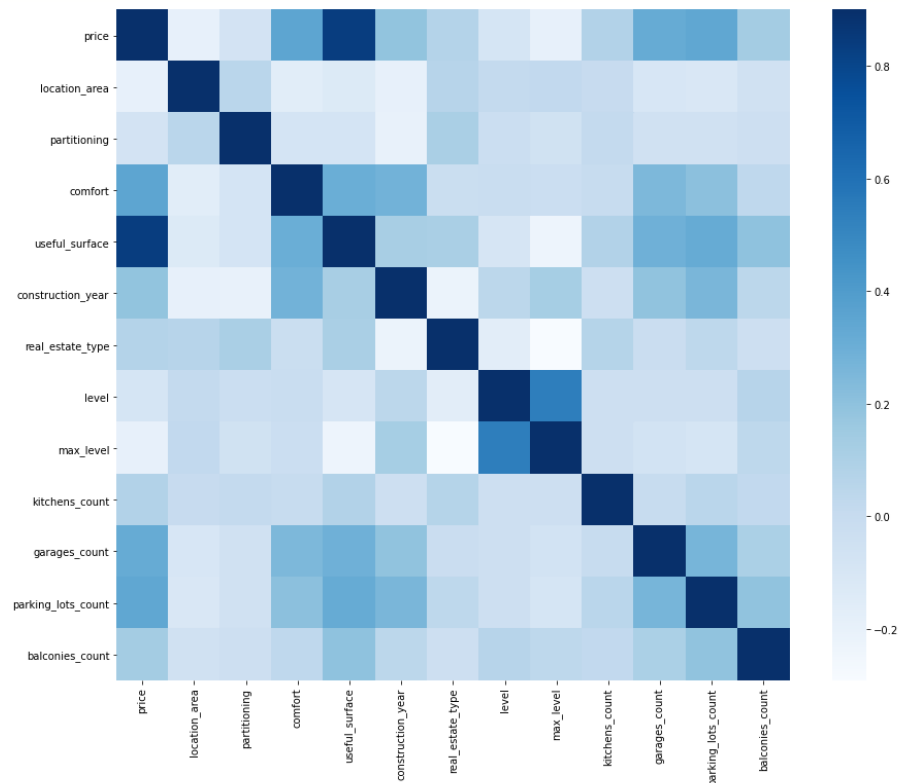
The *comfort* column will be filled in with values from **4** to **1**, where **4** is the best value and **1** the lowest, using the following mapping: **Lux** → **4**, **1** → **3**, **2** → **2**, **3** → **1**). If missing, the value will be set to **3**, which represents the most common type of comfort (**comfort 1**).

For the column *partitioning*, we will replace each value with a corresponding index (e.g. **Decomandat** → **1**). The *null* values will be filled in with **1** because **Decomandat** is the most common type of partitioning.

The column *real_estate_type* contains two values, which will be replaced with a number depending on the occurrence: **bloc de appartement** → **0** and **casa/vila** → **1**. Additionally, this column also presents *null* values that will be completed with the most common type, which is **bloc de appartement** (**0**).

Based on the assumption that every apartment having a useful surface less than **50 m²** has either one or two kitchens, we fill the empty values from *kitchens_count* column with **1**. In case there are more than two kitchens in an apartment having a useful surface smaller than **50 m²**, we will replace that value with **1**. The same procedure will be applied for the *bathrooms_count* column.

Regarding the *price* column, before deleting it from the features dataset, it will be used to delete those offers whose prices are too high or too small compared to the others.



Correlation matrix after cleaning the data

We have noticed that there are many columns that have discrete values (*location_area*, *partitioning*, *real_estate_type*); in these cases we can replace the respective column with a matrix of dimensions (*num_offers*, *num_unique*), where *num_offers* is the number of elements from the dataset and *num_unique* is the number of unique values in that column. For example, let us assume that the column *X* contains 5 unique values; for the *i*-th element having $X[i]=2$, the line that will replace the value 2 will be **[0, 0, 1, 0, 0]**; this procedure is called *one-hot encoding*.

In the following models, we will use both representations of the data: with and without one-hot encoding.

3. Dimensionality Reduction

Dimensionality reduction is a method by which we reduce the number of predictors used in training our models. This way, we can obtain a simpler model, whose performance can even be better than the one using more input variables.

It is the perfect case for our *one-hot encoding* approach, where, by creating more than 100 columns, the data complexity grows, and the *curse of the dimensionality* appears. In order to reduce the number of our variables, we can employ different algebraic methods, which project the data into smaller dimensions, while still preserving the main structure of it.

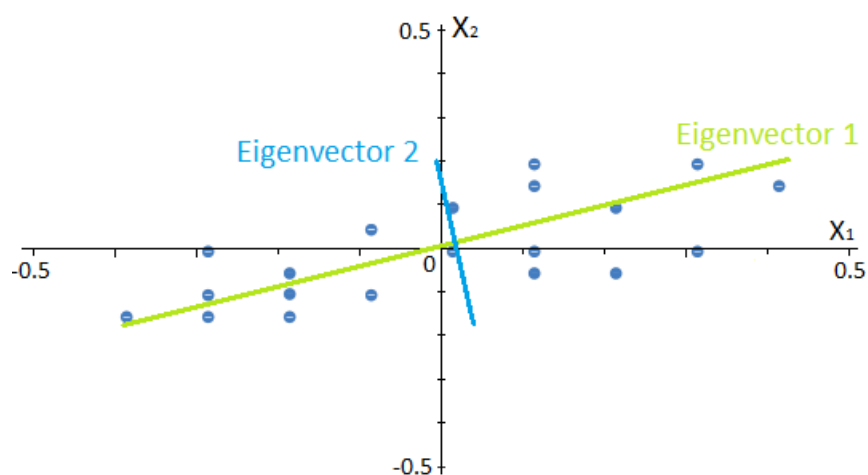
The methods that we have employed in this project are: **PCA**, **Kernel PCA**, and **t-SNE**, which are described below.

3.1. PCA

Principal Component Analysis (PCA) is a statistical technique that allows identifying underlying linear patterns in a dataset so it can be expressed in terms of another dataset of significantly lower dimension without much loss of information. The final dataset will therefore contain different variables, called *principal components*, able to explain most of the variance of the original dataset.

The method of reducing the dimensionality using this technique consists in more steps. First, the data needs to be centered, so we subtract the mean from each observation, this process being also named as *translating* the data. Then, we need to calculate the covariance matrix, in order to tell the linear dependencies between all the variables, which will be used in reducing the dataset's dimension.

The next step is to calculate the eigenvectors and the eigenvalues of the obtained matrix. Eigenvectors are defined as those vectors whose directions remain unchanged after any linear transformation has been applied. However, their length could be changed during the transformation, and so the result will be the vectors multiplied by some scalars, these being called *eigenvalues*. An example of eigenvectors can be checked in the figure below, and as you can see, there are only 2 eigenvectors for a 2 dimensional dataset. In general, a dataset containing N features, will give N eigenvectors. They represent the directions in which the data have higher variance, while their respective eigenvalues determine the amount of variance that the dataset has in that direction.

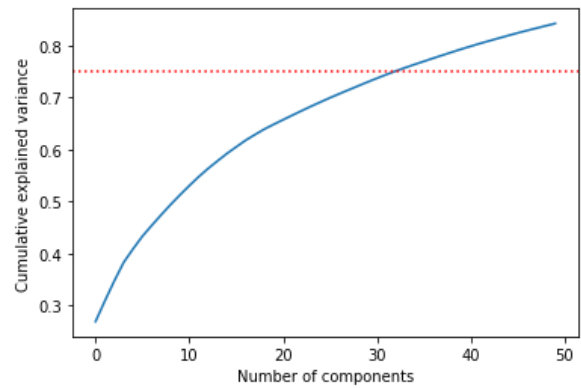


Corresponding eigenvectors for a 2D dataset. Bigger spreads will result in bigger eigenvalues

After calculating the eigenvectors and their eigenvalues, we have to select the principal components, which is selecting those eigenvectors onto which we project the data. In order to define a criterion, we must first define the relative variance of each component and the total variance of the dataset. The relative variance of an eigenvector measures how much information can be attributed to it, while the total variance of the dataset is the sum of all variables' variances. A common way to select the variables is to establish the amount of information that we want the final dataset to explain, referred to as the *explained variance ratio*. If this amount of information decreases, the number of principal components that we select will decrease as well.

In the figure below, you can see how the number of principal components (based on our one-hot encoded dataset) increases as the amount of gained information grows bigger. To make sure we keep a low number of variables while preserving a good amount of information, we have chosen a threshold of 75% explained variance ratio, which would give us around 32 components.

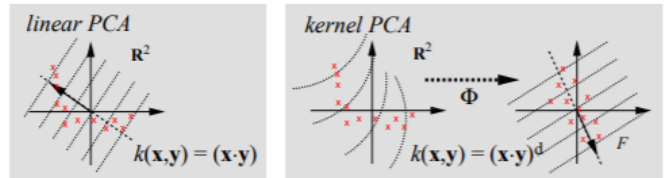
In the end, what is left to do is just performing the projection, using the selected components. We should keep in mind that this projection can explain most of the variance of the original dataset, but at the same time, we have lost the information about the variance along the unused components, this process being irreversible.



3.2. Kernel PCA

Kernel Principal Component Analysis was created as an extension of the above mentioned Principal Component Analysis, in the field of multivariate statistics, using techniques of

kernel methods. It is a method that deals with non-linear dimensionality reduction. As such, one can efficiently compute principal components in high dimensional feature spaces.



The main idea behind kernel PCA is that by using a nonlinear kernel function k instead of the standard dot product, we implicitly perform PCA in a possibly high dimensional space F which is nonlinearly related to input space. The dotted lines from the figure above are contour lines of constant feature value.

However, in practice, a large amount of data can lead to a large K , for which storage would be a problem. A solution to this problem would be clustering the dataset and populating the kernels with the means of the clusters.

3.3. t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) is another technique for dimensionality reduction, and is particularly well suited for the visualization of high-dimensional datasets. Contrary to PCA, it is not a mathematical technique but a probabilistic one. The original paper describes the working of t-SNE as:

“t-Distributed stochastic neighbor embedding (t-SNE) minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding”.

Essentially what this means is that it looks at the original data that is entered into the algorithm and looks at how to best represent this data using less dimensions by matching both distributions. The way it does this is computationally quite heavy and therefore there are some (serious) limitations to the use of this technique. For example one of the recommendations is that, in case of very high dimensional data, you may need to apply another dimensionality reduction technique before using t-SNE like PCA for dense data or TruncatedSVD for sparse data to reduce to a reasonable amount.

4. Models

4.1. Models Evaluation

In order to evaluate the trained models, we have chosen to split the dataset into 70% consisting of training data and 30% of testing data. The models will be tuned and validated based on the results obtained on the testing data.

We choose as evaluation metrics the *Mean Absolute Error* and the *Mean Squared Error*. Mean Absolute Error (MAE) measures the absolute average distance between the real data and the predicted data, but it fails to punish large errors in prediction that is why we wanted to also measure MSE. Mean Square Error (MSE) measures the squared average distance between the real data and the predicted data and the actual large errors can be detected better.

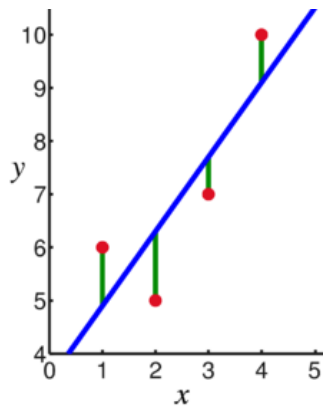
$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

$$MAE = \frac{1}{n} \sum_{i=1}^n |Y_i - \hat{Y}_i|$$

We also took into consideration the R^2 score of the models in order to detect the proportion of the variance for a dependent variable (in our case the price) that is explained by the independent variables from a regression.

$$R^2 = 1 - \frac{\sum_{i=1}^n (\hat{y}_i - y_i)^2}{\sum_{i=1}^n (y_i - \bar{y}_i)^2}$$

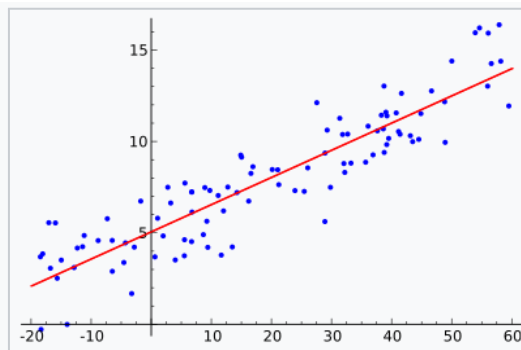
4.2. Linear Regression



Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable. For example, a modeler might want to relate the weights of individuals to their heights using a linear regression model.

A linear regression line has an equation of the form $Y = a + bX$, where X is the explanatory variable and Y is the dependent variable. The slope of the line is b , and a is the intercept (the value of y when $x = 0$).

4.2.1 Simple and multiple linear regression



Example of [simple linear regression](#), which has one independent variable

The very simplest case of a single [scalar](#) predictor variable x and a single scalar response variable y is known as [simple linear regression](#). The extension to multiple and/or [vector](#)-valued predictor variables (denoted with a capital X) is known as [multiple linear regression](#), also known as *multivariable linear regression*.

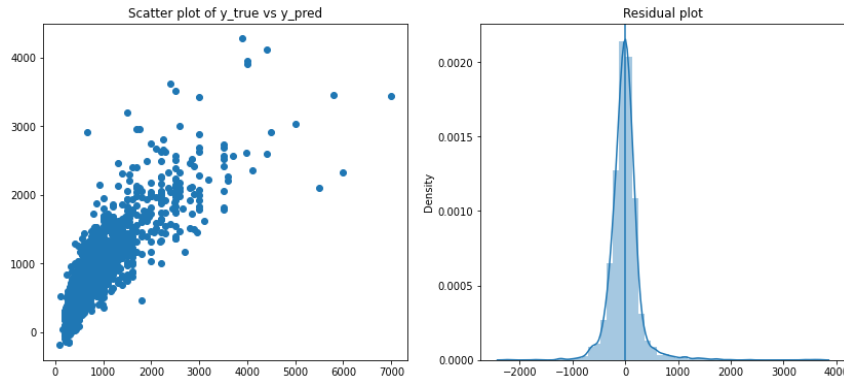
Multiple linear regression is a generalization of [simple linear regression](#) to the case of more than one independent variable, and a [special case](#) of general linear models, restricted to one dependent variable. The basic model for multiple linear regression is

$Y_i = b_0 + b_1X_{i1} + b_2X_{i2} + \dots + b_pX_{ip} + \epsilon_i$, for each observation $i = 1, \dots, n$.

In the formula above we consider n observations of one dependent variable and p independent variables. Thus, Y_i is the i^{th} observation of the dependent variable, X_{ij} is i^{th} observation of the j^{th} independent variable, $j = 1, 2, \dots, p$. The values β_j represent parameters to be estimated, and ϵ_i is the i^{th} independent identically distributed normal error.

The following scores have been obtained on the training dataset: **MSE = 97698.64**, **MAE = 184.57** and **R2 = 0.72**. In the **Table 4.1 (Conclusions)** we have mentioned the results on the test data and they are similar to the former values.

Below, we can see the correlation between the ground truth label of the test data and the predicted labels of the test data and residual plot of the test data.



4.3. Ridge Regression

Ridge Regression is an extension of linear regression that adds a regularization penalty to the loss function during training. It is especially useful when applied on data that suffers from multicollinearity, which commonly occurs in models with large numbers of features. When such a thing happens, least squares estimates are unbiased, but their variances are large, so they might be far from their true value. By adding a degree of bias to the estimates, ridge regression reduces the standard errors.

The L2 penalization is used by Ridge Regression in its cost function, which has the following form:

$$\|\beta\|_2 = \sqrt{\sum_{j=1}^n \beta_j^2}$$

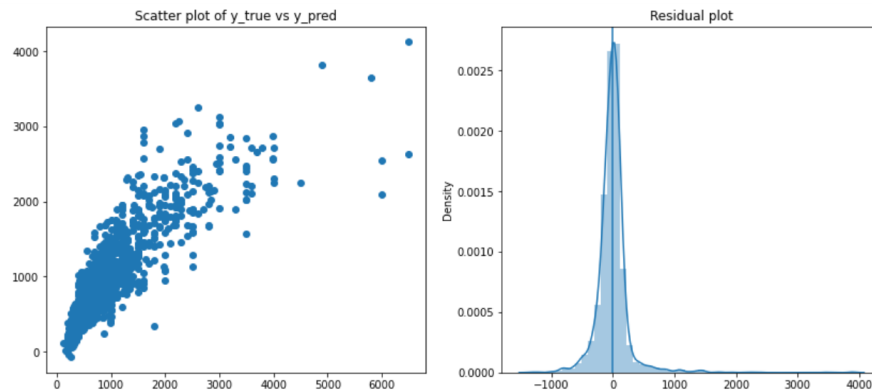
Ridge regression attempts to reduce the norm of the estimated vector and at the same time tries to keep the sum of squared errors small. In doing so, a penalty is added to the cost function, which then will look like this:

$$cost(y_{true_i}, y_{pred_i}) = \sum_{i=1}^{i=n} (y_{pred_i} - y_{true_i})^2 + \alpha ||W||_2,$$

where $\alpha \geq 0$ controls the amount of shrinkage: the larger the value of α , the stronger the regularization. When $\alpha = 0$, the cost function is identical to the cost function of linear regression.

Variables standardization is the initial procedure in ridge regression. Both the independent variable and dependent variable require standardization through subtraction of their averages and a division of the result with the standard deviations.

We applied Ridge Regression on our data, and by performing randomized search on α , we found that the best value is around **4**, which led to the results depicted in **Table 4.1 (Conclusions)**. Also, by fitting the model with the best *alpha*, we obtain the following plots.



4.4. Lasso Regression

Lasso Regression (*Least Absolute Shrinkage and Selection Operator*) is a regression method that performs both variable selection and regularization. The goal of Lasso Regression is to obtain the subset of predictors that minimizes prediction error for a quantitative response variable. It is very useful when dealing with large numbers of features, because it automatically does feature selection.

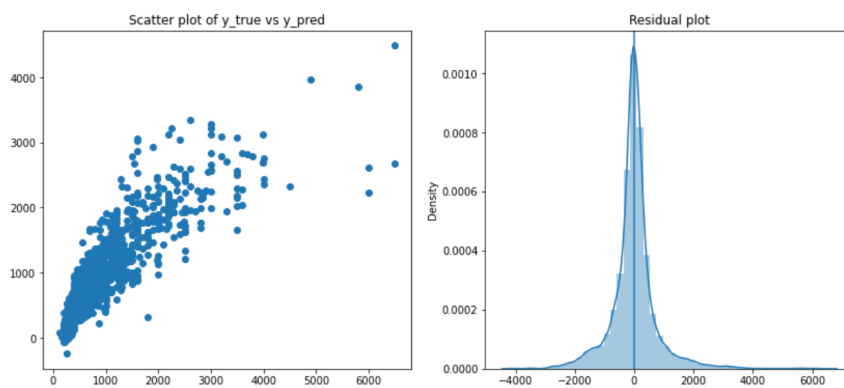
The only difference between Lasso and Ridge Regression is how they penalize the cost function. Lasso penalization(L1) has the following form:

$$||\beta||_1 = \sum |\beta_j|$$

Thus, Lasso is directly proportional to the absolute values of the coefficients, rather than Ridge, which is proportional to the squares of the coefficients. It has a similar effect to Ridge in terms of complexity tradeoff, as increasing α will raise the bias but lower the variance.

However, Lasso is more likely to perform feature selection, because for a fixed α Lasso is more likely to result in coefficients being set to zero.

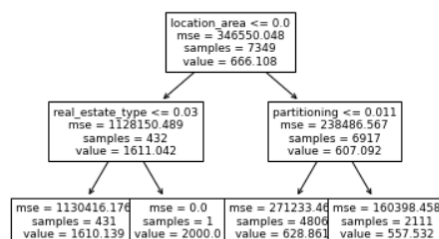
As before, by performing randomized search on α , we found that the optimal value would be **4.24**. Fitting this model returned the results depicted in **Table 4.1 (Conclusions)**, and the following graphs:



4.5. Decision Tree

Directed acyclic graphs used for classification and regression problems, **Decision Trees**, are supervised learning models where each node of the tree represents an attribute of the data, its branches represent the answer and the leaf represents one class.

Criterion: mse; Splitter: best; Maximum depth: 2; Maximum features: 1
MSE= 329748.4143879329 MAE= 332.48104103128327



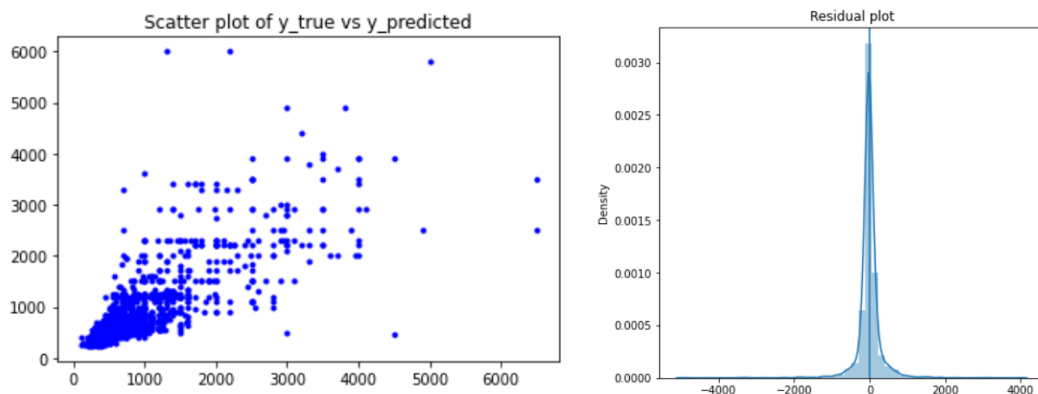
This procedure works as follows: the dataset is split based on its features (or columns) and, recursively, each feature will become a node until each element can be assigned to one class. It can also be interpreted as a set of if clauses: *if (value has attribute1) and (value has attribute2) and... and (value has attributeN) then is in the class N* (Wikipedia contributors).

The main advantage of the decision tree is that it is easily understandable, but it is not very efficient since its results are obtained from partitioning the space of the given training data and the outcome for the regression task will be the average of the values from the class that the testing element belongs to.

For this regression task we have used the **DecisionTreeRegressor** model from the library *sklearn.tree*. In order to use the model, we must set some parameters as presented on the official web-site (scikit-learn developers):

- **criterion** which represents the manner in which will be calculated the quality of a split (used for selecting the next attribute) and the possible values are: *mse*, *friedman_mse* and *mae*;
- **splitter** may have the values *best* (the best split) or *random* (the best random split) and it represents the manner in which each node will be split;
- **max_depth** represents the maximum depth of the tree; if set to *None* the tree will extend until all the the leaves are pure or when they contain a number of *min_samples_split* elements;
- **min_samples_split** this value will be used only if *max_depth* is *None*;
- **max_feature** represents the number of features that will be considered when looking for the best split although, all the features will be considered if can not be found one best split.

After testing with different values for our parameters, we have chosen the values for which the MAE and MSE remain as small as possible: **criterion=mse**, **splitter=best**, **max_depth=10**, **max_features=5**. It should also be mentioned the fact that the *one-hot encoding* representation does not improve the results shown in the **Table 4.1 (Conclusions)**.



4.6. Random Forest

Random Forests are an *ensemble* learning technique, which can be used for our regression task. Their scope is to reduce the high variance that a decision tree could present in its predictions, by employing a general procedure called *Bagging* (*a.k.a. Bootstrap Aggregation*). Thanks to this method, the variance of our model is reduced, so that each time we apply it to different data subsets we will get similar results.

The key proof of the random forests' performance relies on *The Central Limit Theorem*, which states that if we have a population with mean μ and standard deviation σ and take with replacement sufficiently large random samples from the population, then the distribution of the sample means will be approximately normally distributed.

In a more mathematical manner, the theorem can be expressed as follows:

Let X_1, X_2, \dots, X_n be a set of n independent random variables and each X_i has an arbitrary probability distribution with mean μ and variance σ^2 . Then the random variable $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$ is normally distributed with $\mu(\bar{X}) = \mu$, $\sigma(\bar{X}) = \frac{\sigma}{\sqrt{n}}$, $Var(\bar{X}) = \frac{\sigma^2}{n}$.

The conclusion of this theorem is that we would like to use different training datasets to train different models, and then “average” them in order to obtain a low variance model. Unfortunately, we own a single source of data, so we will apply a *Bootstrap* method, which consists in taking repeated samples from our only dataset.

This way, the random forest will consist of many regression decision trees, each one using its own bootstrapped training set, and will average the resulting predictions gotten from each tree.

During the parameters tuning phase, the following parameters have been adjusted:

- **n_estimators (NE)**. The number of trees in the random forest. The more the better, since it will improve the global belief in the right predictions. However, from a specific point the improvement starts to decrease in size and only more time is spent on the training.
- **min_samples_leaf (MSL)**. The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least *MSL* training samples in each of the left and right branches.
- **min_samples_split (MSS)**. The minimum number of samples required to split a node.

- **min_impurity_decrease (MID)**. A node will be split if this split induces a decrease of the impurity greater than or equal to *MID*. The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} * \left(Impurity - \frac{NR_t}{N_t} * Impurity_{right} - \frac{NL_t}{N_t} * Impurity_{left} \right), \text{ where}$$

N : the total number of samples

N_t : the number of samples at the current node

NR_t : the number of samples in the right child

NL_t : the number of samples in the left child

By experimenting randomized configurations for the random forest's hyperparameters, both with and without the *one-hot-encoding* approach on the training dataset, we have discovered that there is not a big difference between these two approaches.

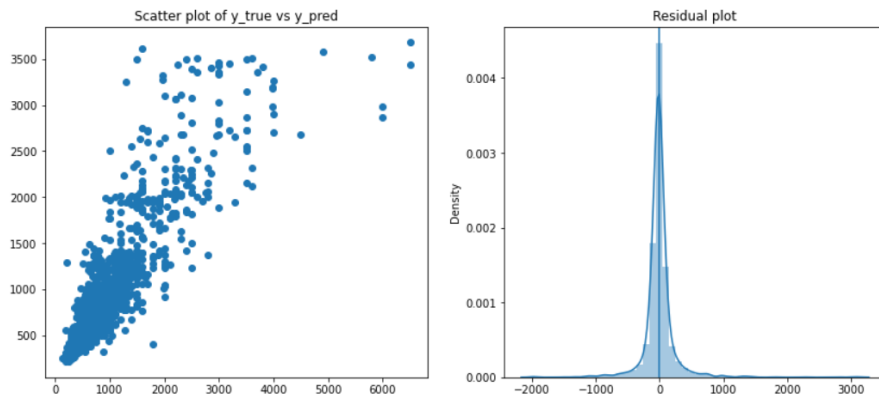
While using a *one-hot encoded* training dataset, setting the models' parameters to **NE=798, MSL=2, MSS=61, MID=0.08080** would give us the following errors:

- on training set: **MSE=227.980, MAE=10.650, R²=0.853**;
- on test set: **MSE=258.848, MAE=11.288, R²=0.803**.

However, not using the *one-hot encoding* approach would lead to similar results. In this case, when setting the random forest's parameters to **NE=857, MSL=8, MSS=33, MID=0.01212**, the model would present the following errors:

- on training set: **MSE=226.681, MAE=10.667, R²=0.853**;
- on test set: **MSE=259.576, MAE=11.347, R²=0.802**.

In the plots below we have depicted the predictions and residuals for the model using the *one-hot encoded* dataset.

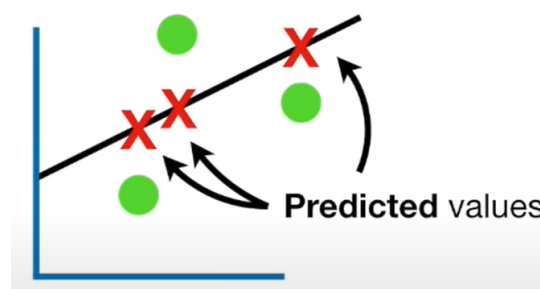


As it can be seen in the residual plot, the thinner body of the bell depicts that the variance is much smaller than other models, and this thanks to the *Bagging* technique.

4.7. Gradient Boosting Regressor

Gradient Boost model is based on decision trees and it starts with one single leaf, instead of a tree or a stump unlike *AdaBoost* for example. The starting point represents an initial guess of all the samples for the dependent variable, in our case the *price*. Because we are trying to predict a continuous value like *price*, the first guess that the algorithm takes would be the average value.

After the first step is fulfilled, *Gradient Boost* builds a tree. Compared to *AdaBoost*, it is similar because the tree built is based on the errors that were made by the previous tree, with the only difference that it is larger. *Gradient Boost* scales all the trees by the same amount



and then builds another tree based on the errors made by the previous tree. This step is repeated until additional trees fail to improve the fit.

Gradient Boost uses *Pseudo Residuals* (keeps track of all the errors made by the previous tree compared with the actual value). *Pseudo Residuals* are based on *Linear Regression* where the difference between the **Observed** values and the **Predicted** values result in **Residuals**.

After calculating the residuals, the model actually uses them for prediction, meaning that the model builds a tree to **predict the residuals**. By doing so, we obtain fewer leaves than *Residuals*, meaning that we can replace one leaf with its average.

In order to avoid performing on the training data way better than performing on test data, *Gradient Boost* uses a *Learning Rate* to scale the contribution every time a new tree is built. The *Learning Rate* is a value between **0** and **1**. In this case, I used a learning rate of **0.01** in order for the model not to jump over some essential steps (taking lots of small steps in the right direction results in better *Predictions* on a *Test Dataset* because it lowers the variance which is our actual $\frac{1}{2}(\text{Observed} - \text{Predicted})^2$ goal).

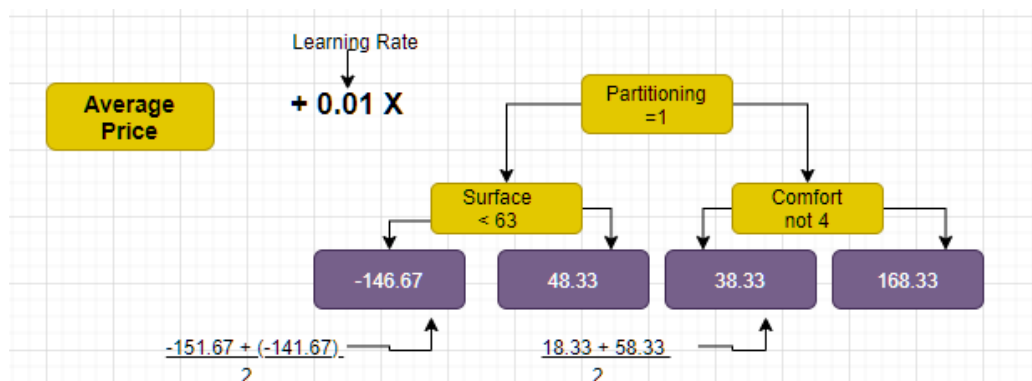
The loss function usually used on *Gradient Boost* and that generated the best results on our dataset as well is the huber loss function.

The final prediction, after building several decision trees based on the residuals is made by adding the residuals from the parents in order to obtain the final predicted *Price*.

Considering the simplified table applied on 6 rows of our dataset and 4 variables (3 independent variables and the price dependent variable), I will try and follow all the steps of the prediction. First, we start with the average price and the difference between the actual price and the average price can be found on the column named Residuals. The **average price** for this particular example is approx. **711.67**.

Partitioning	Comfort	Surface	Price	Residuals
0	4	63	880	168.33
1	3	63	760	48.33
1	4	54	560	-151.67
0	2	82	730	18.33
0	4	54	770	58.33
1	3	42	570	141.67

Based on the Residuals calculated on the previous step, a decision tree is made. The next step would be to calculate again the residuals based on this decision tree and so on until additional steps fail to improve the predicted price.

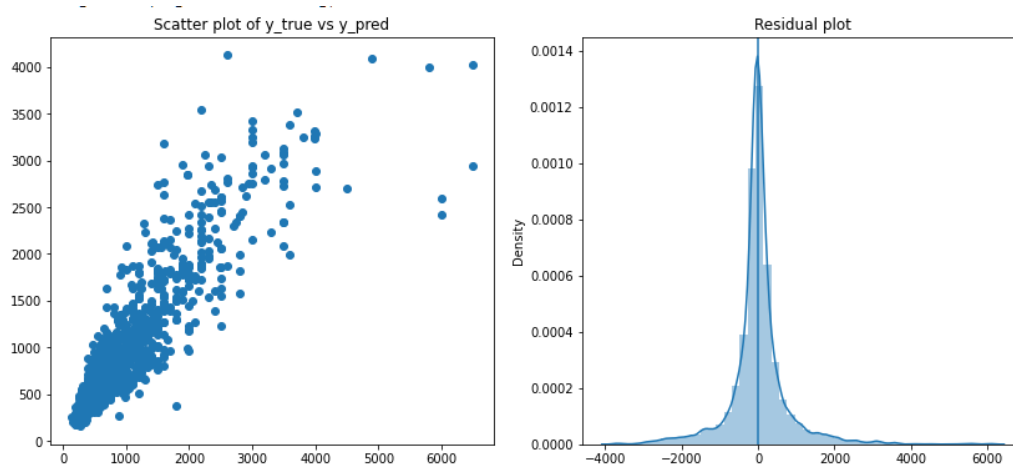


Note: the example above is a simplified sample which only allows up to four leaves in order to better understand the model. The model designed can have as many leaves as it should until it reaches a good fit.

Fitting the actual dataset into the model, we actually obtained the following results for the training data: $MSE=47601.67$, $MAE=105.64$, $R^2=0.864$. The differences between the training data and the test data are not very high, which is usually a good sign, since we obtained a MAE score equal to **115.95** and a MSE score for the test data of **56972.78**. This means that the squared average distance between the real data and the predicted data is **56972.78**. Also, for the test data we obtained a coefficient of multiple determination of **0.833**. Meaning that the **83%** of the variance of the dependent variable (in our case the price) is explained by the variance of all the other variables included in the regression model.

On the other hand, without using the one hot encoding approach for the categorical columns, we noticed that the results did not improve significantly. Obtaining for the training data: $MSE=39144.62$, $MAE=96.43$, $R^2=0.88$. The differences between the training data and the test data are still not very high, with $MSE=58201.56$, $MAE=116.64$, $R^2=0.82$.

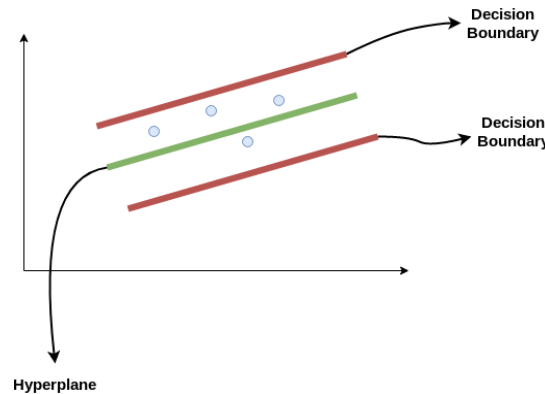
Plotting the residuals and the actual price vs the predicted price, we obtained the following:



As we can observe from the Residual plot, the errors follow a normal distribution which is a good thing for our model.

4.8. Linear SVR

Support Vector Regression (SVR) is a model which mimics the idea behind a Support Vector Machine (SVM), but adapted to work on continuous values. Here, instead of minimising the error rate (like other models are doing), we try to fit the error within a certain threshold, by finding the best fit line: the hyperplane that has the maximum number of points.



The idea behind SVR is to fit the hyperplane within the specified decision boundary

The objective in this case is basically to consider the points that are within the decision boundary lines. These lines can be at any distance ϵ from the hyperplane. Assuming that the equation of the hyperplane is $y = w * x + b$, then the equations of the decision boundary becomes $w * x + b = + \epsilon$ and $w * x + b = - \epsilon$. Therefore, any hyperplane that satisfies our SVR should satisfy:

$$- \epsilon < y - w * x + b < + \epsilon$$

Our main aim here is to decide a decision boundary at ϵ distance from the original hyperplane such that data points closest to the hyperplane or the support vectors are within that boundary line. Hence, we are going to take only those points that are within the decision boundary and have the least error rate, or are within the margin of tolerance. This will give us a better fitting model.

Just like the regular SVM, the SVR presents a few important parameters to be taken into account:

- **kernel**, which helps at finding a hyperplane in the higher dimensional space without increasing the computational cost. This increase in dimension is required when we are unable to find a separating hyperplane in a given dimension. In our project we employed the *linear kernel*, which works faster than other kernels, being described as $K(x, y) = x^T y$
- ϵ , a value which describes a margin of tolerance where no penalty is given to errors;

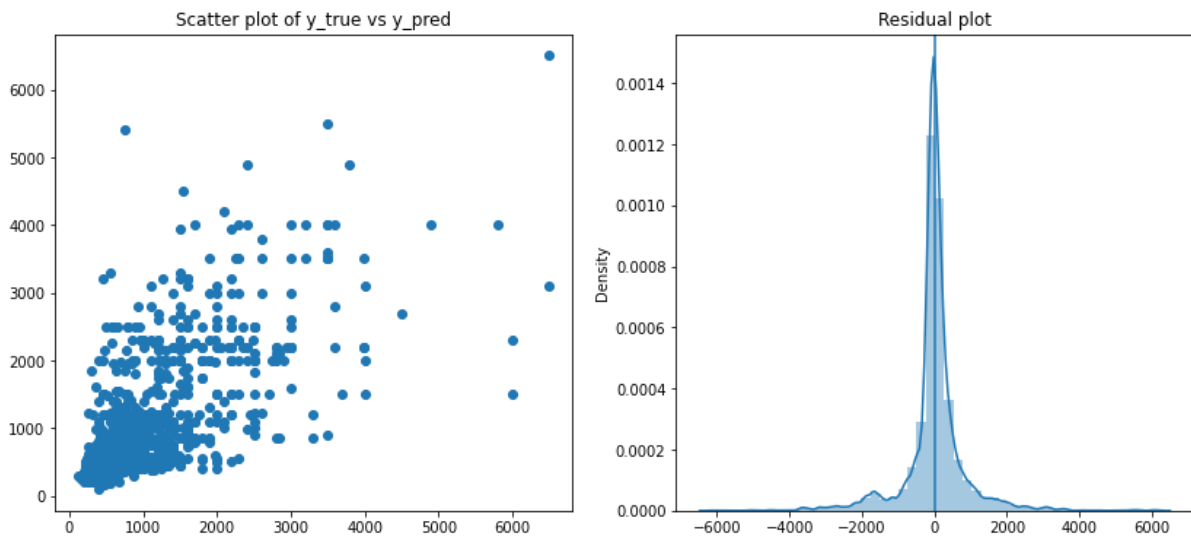
- C , a regularization parameter, which is inversely proportional to the strength of the regularization. This means that a lower C will give the model more power of generalization, making room for some errors, while a larger C will impose bigger penalties for misclassified data points;

Fitting a SVR on the *one-hot encoded* dataset, without any dimensionality reduction applied to it, we obtained the following results:

- on training set: **MSE=145338.706, MAE=174.827, $R^2=0.58$** ;
- on test set: **MSE=175676.505, MAE=203.678, $R^2=0.485$** .

We can tell that this model didn't perform very well, because of the big errors. One reason for this could be the fact that the data is not entirely linearly separable, and therefore the SVR couldn't find a boundary good enough to minimize the error as desired.

Just like we did in the other models' analysis, we have attached down below the plots for the residuals and the true vs predicted prices, which reflect the bigger errors compared to the other models.



4.9. K-Nearest Neighbours

The k-Nearest Neighbours (k-NN) algorithm is a non-parametric classification method, that could also be used for regression. When used in this scope, the output is the average of the values of k nearest neighbours.

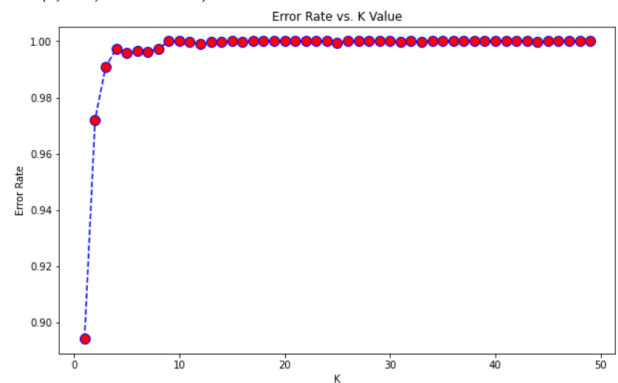
k-NN is a lazy algorithm, which means that all computation is deferred until evaluation. For the training step, the algorithm consists only of storing the feature vectors and class labels of the training samples. In the regression phase, the algorithm uses a weighted

average of the k nearest neighbors, weighted by the inverse of their distance. In most cases, the metric used is Euclidean distance.

The most important thing in ensuring the k-NN algorithm works properly is finding the best k . A good k can be selected by various heuristic techniques. The special case where the class is predicted to be the class of the closest training sample (i.e. when $k = 1$) is called the nearest neighbor algorithm.

k-NN is sensitive to the curse of dimensionality, which refers to phenomena that arise when working with high-dimensional data as opposed to low-dimensional data. In practice, when the number of dimensions is higher than 10, dimension reduction shall be performed before applying k-NN.

For our dataset, we first searched for the best value of k , by plotting the error rate versus k . From this plot, we can tell that the best value for k would be 1, after which the error rate quickly rises.



Fitting a 1-NN model on the *one-hot encoded* dataset, without any dimensionality reduction applied to it, we obtained the following results:

- on training set: **MSE=71.91, MAE=0.36, $R^2=1.00$** ;
- on test set: **MSE=142289.88, MAE=187.33, $R^2=0.58$** .

As it can be seen, the model performs very well on the training set, however it grossly overfits. As said, k-NN is not a suitable algorithm for when the data is in high dimensions, as the effects of the curse of dimensionality in this context mean that the Euclidean distance is unhelpful, because all queries are almost equidistant to the search query vector.

What might have worked better would have been manually reducing the number of features to at most then, and then applying k-NN.

5. Conclusions

5.1. First part

In terms of analysis of the models' scores, **Table 4.1** contains the best results obtained in our project's first part, ordered by their complexity. We can see that based on MSE and MAE metrics that we decided to follow, the best score was obtained by Gradient Boosting Regression, and also that the model that did not give the expected results was Linear Regression.

Looking at the residual plots obtained for every model, we can easily see that in most cases the errors have a normal distribution which is a good thing for our analysis. Further analysis shows the fact that for the Random Forest model, the Residual Plot is thinner than all the others meaning that the variance is smaller than other models.

Based on the experiments, we can also tell that one-hot encoding method applied to the dataset does not significantly improve the results of the presented methods.

To conclude, given the dataset that we decided to use, we faced in the beginning different challenges regarding the data analysis which helped in making decisions about what data cleaning we should implement. Other following challenges were the actual models' implementations and deciding the metrics that we wanted to use in order to score them. In the end, we looked at the results and compared the obtained values for a better understanding.

MODEL NAME	RESULTS		
	MAE	MSE	R2
Linear Regression	180.97	88679.86	0.734
Ridge Regression	153.09	82284.80	0.758
Lasso Regression	155.17	75580.85	0.778
Decision Trees	151.51	95329.32	0.741
Random Forests*	127.40	67592.75	0.801
Gradient Boosting Regressor*	114.26	54178.10	0.837
* This model used a <i>one-hot encoded</i> dataset.			

Table 4.1: First results for each model on the test data.

5.2. Second part

As it can be seen in the following table, we obtained the best results when we used the data as it was, without performing any dimensionality reduction method. This could be because we artificially increased the number of our features, by using the one hot encoding technique, in order to simulate a real-world big dataset.

As a result, we went from having 12 features to having 196. After that, we tried reducing the dimension, so after all of these transformations it is very likely that important details about our data were lost in translation. Therefore, it is important to note that dimensionality reduction is not always the best choice in training the perfect model.

		Random Forest	Gradient Boosting Regressor	Linear SVR	k-NN
No Dimension Reduced	MAE	127.40	122.09	174.827	187.33
	MSE	67592.75	60526.41	145338.706	142289.88
	R2	0.801	0.82	0.586	0.58
PCA	MAE	177.54	148.58	239.318	201.11
	MSE	118953.53	90906.92	249626.689	171794.27
	R2	0.65	0.73	0.268	0.50
Kernel PCA	MAE	178.99	173.28	242.57	195.58
	MSE	122583.65	119178.25	254208.632	157626.10
	R2	0.64	0.650	0.254	0.54
t-SNE	MAE	225.00	234.79	358.664	208.87
	MSE	174696.25	221148.82	423085.239	184003.10
	R2	0.49	0.35	-0.239	0.46

Table 4.2: Results from the second part of the project.

6. Bibliography

- [1] scikit-learn developers. <https://scikit-learn.org/>. Accessed at 11.01.2021.
- [2] <http://www.stat.yale.edu/Courses/1997-98/101/linreg.html>
- [3] https://people.eecs.berkeley.edu/~wainwrig/stat241b/scholkopf_kernel.pdf

Appendix

Feature Name	Data Type	Description
id	Integer	Unique identifier of the advertisement.
location	String	The full address location provided in the ad.
location_area	String	The specific area in town, extracted from the full address.
seller_type	String	The one who manages the ad: agency or prepay (owner).
type	String	The house type. Unique value: “apartament”.
partitioning	String	The house partitioning (“decomandat”, “circular” etc.)
comfort	String	The house comfort: “lux”, 1, 2, or 3.
price	Float	The rental price per month (Euro).
rooms_count	Float	The number of rooms that the house owns.
useful_surface	Float	The useful surface (w/o balconies, walls etc) in m ² .
built_surface	Float	The entire surface of the house in m ² .
construction_year	Integer	The construction year of the house.
real_estate_type	String	The type of the building: “bloc de apartamente” or “vila”.
height_regime	String	The height regime of the building.
level	String	The floor where the apartment is located.
max_level	Integer	The number of floors that the building owns.
kitchens_count	Integer	The number of kitchens that the house owns.
bathrooms_count	Integer	The number of bathrooms that the house owns.
garages_count	Integer	The number of garages that the house owns.
parking_lots_count	Integer	The number of parking lots that the house has to offer.
balconies_count	Integer	The number of balconies that the house owns.

Table A.1: Data types and descriptions for the dataset’s columns