Ahmet Emre Belge, 3149309, Kaggle: Ahmet Emre Belge (ahmetemrebelge)

Introduction

This was a challenging machine learning project. The main problems were the ones usually encountered in similar projects in housing price regression: massive amount of missing or wrongly input values, outliers, and the numerous variables that affect the performance of the regression, where searching the best combination of them require a time-intensive trial-failure process. My approach to the problem: Observe the data and features, visualize the data, and clean it accordingly, deal with missing values, engineer relevant features, evaluate the model.

Observing the data, and initial encodings

Right off the bat, we can see that there are four columns: construction_year, expenses, energy_efficiency, and total_floors, which have missing values in more than 30 percent of the rows. I interpreted NaN values in the garden, elevator, and balcony columns as the house having no balcony and filled them with False, and then encoded them with numerical binary values. Along with them, I also encoded the conditions column with numbers from 0 to 3, with 0 being the worst conditions and 3 the best. I encoded categorical data throughout the model as the models that I intended the use work best with numerical inputs.

Calculating the impact of the columns on the model's output and dealing with NaNs

To calculate the real impact of the columns, I dropped the rows with NaN values in any of the columns to evaluate a baseline model and obtained the SHAP (SHapley Additive exPlanations) values for the features. To create a train/val split, I did not use the vanilla train_val split of sklearn. Instead, I used a custom function that distributes the data in a balanced way between the train and validation sets. I will dive deeper into the rationale behind this later, but the main reason is the high number of possible outliers in our dataset which create the need to having similarly distributed train and validation sets.

After obtaining the average impact on model output magnitude of each feature, I proceeded to drop the columns that have very low importance but also very high volumes of NaNs, which are total_floors, energy_efficiency, garden, and proximity_to_center. In particular, the proximity_to_center column did not seem very useful, and I already had the idea of creating my own version of it.

After dropping those columns, I proceeded to drop any rows with NaN values in any column apart from the expenses and construction_year columns, as the columns apart from those have all approximately under 5 percent of missing values, and they have a high number of rows where they are missing jointly. Therefore, dropping these will save us from the risk of possibly introducing bias with imputation without causing a significant loss of data. We will still try to impute expenses and construction_year as they seem to have a significant impact on the model and dropping the rows with them missing would cause a considerable data loss. After these initial modifications to the data, I moved on to exploratory data analysis, where I also dived deeper into clearning the data.

Exploratory data analysis

First, I plotted a correlation heatmap to also have an idea of how the variables are correlated between each other, before moving on to plotting each individual variable.

Univariate: Plotting the histograms of discrete columns (apart from construction_year), we can see that the data is very imbalanced, with the presence of high leverage points. The violin plots of continuous variables show us that there are some serious outliers in the data. I tried to clean the data as best as I could, but throughout different trials, I observed that cleaning the data according to common sense generally led to overprocessing the data and probably deleting data that is valuable for the model's performance. Therefore, instead of cleaning all outliers above or below a definite percentile, I only those that made absolutely no sense trying to remove the least amount of data possible in the process.

Price: Plotting the price above the 90th percentile, we can see the presence of these severe outliers. Observing points with prices below 10,000, I dropped rows with data that did not make sense, likely due to incorrect input, as it was a small number of rows. I did the same thing for large values, specifically above 10,000,000.

Expenses: I started by observing the percentiles, and after observing a large number of values below 10 in houses that probably have much higher expenses, I opted to make them NaN to then impute them, we already have a large number of expenses to impute, and I believe that it is better to try and impute 1000 more rows than to drop them. I instead dropped rows with expenses larger than 1200, as is was only a small amount of rows that would not lead to a significant loss of data.

Construction_year: There was a house labeled as constructed in the year 2500; I looked it up on Google Maps and it looked like a house from the year 2000. I then set any house with a construction_year under 1200 to 1200.

Surface: I dropped any rows with surface equal to 0, as there were a small number of them.

Bivariate: Looking at the violin plots of the discrete variables with price, we see a clear and positive relationship between prices and n_bathrooms and n_rooms, even though n_rooms had a significantly lower SHAP score.

To clean some noise, I set floors to be equal to 6 for rows that had floor values with occurrences less than 9 and set n_bathrooms to 8 for any row with n_bathrooms equal to 9.

After plotting the continuous variables against each other, we see that in the joint distributions with the dependent variable, there does not seem to be any clear predictors of the extreme values, which is a negative since outliers will quickly inflate the MSE if we cannot predict them well.

Imputation using random forest for construction_year and expenses

After trying to use k-NN and MICE to impute values in our dataset, I saw that these methods introduced a high amount of bias to the data and were not accurate enough. I got the best imputation performance out of a random forest model, which was trained on rows that did not have NaN values. Moreover, the data used to predict expenses did not have the construction_year column and vice versa. I set expenses and construction_year as the dependent variable, and after tuning the hyperparameters with Weights & Biases, I predicted them using the random forest model. In the end, I added the imputed construction_year column to the dataframe with only expenses imputed.

For the imputation of the test set, as we had NaN values in columns other than construction_year and expenses that we cannot drop, I first went on with a MICE imputation on the other columns and then proceeded with the same approach as the training set.

Feature Engineering

Throughout the project, I tried to engineer different features and tested their contribution to the model's performance. The features that I initially engineered were: distance to city center, minimum distance to a monument, minimum distance to a metro/vaporetto station, room/bathroom ratio, surface/room ratio, age, a dummy variable which noted if the house was within 3.5 kilometers from the center (as from the plots, after 3.5 km, I didn't see a clear relationship between price and distance), another dummy for being under 500 meters from a metro station, and of course, a dummy for the city.

In the end, the best performance was given when I had the features of distance to center, distance to monument, and city.

City: After seeing that the houses were either around 45/9, 45/12, or 41/11 coordinates, it was evident that they were all in the provinces of Milan, Venice, and Rome. So, after defining a dictionary with the coordinates of these cities, I assigned the cities according to the Euclidean distance of the houses from the cities' coordinates.

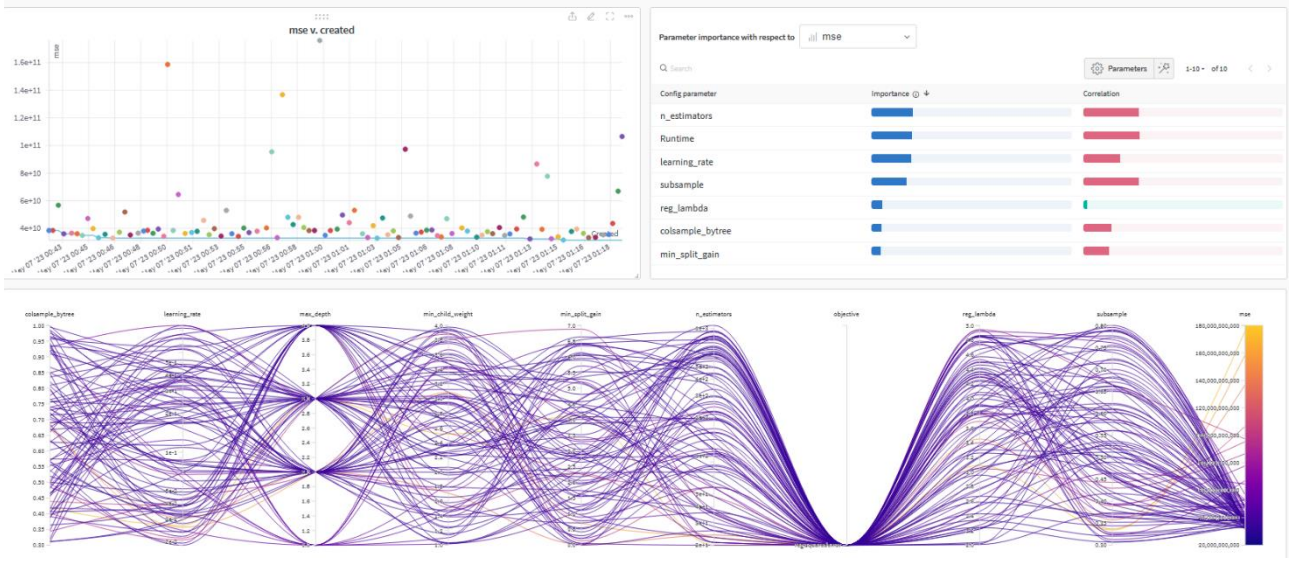
Distance_to_center: After encoding the cities, I used the geodesic distance function from the geopy library to calculate the distance of the house from the city center, instead of using the pre-calculated Euclidean distances. This was mainly because I wanted to have the distances in kilometers (to understand the values better myself), which I didn't have with the Euclidean distance, and converting the Euclidean distances would have taken a similar amount of time and computation.

Distance_to_monument: I created a dictionary with the coordinates of the top 10 landmarks in Milan, Venice, and Rome (In Rome, we have EUR as a +1 as it is a high-value neighborhood relatively outside of the city center). believe

this to be an important feature, even if it is highly correlated to distance_to_center, as it might help with locating higher-value neighborhoods within cities, as they are usually around the important landmarks.

Models

XGBoost: When talking about tabular data, like in our case, the algorithm that comes first to mind is extreme gradient boosting. As hyperparameters play a very important role in XGBoost's performance, to evaluate hyperparameters in the most efficient way, I opted for a random search using the Weights & Biases (wandb) library to beautifully visualize the different combinations of hyperparameters and choose the ones that provided good performance and would be able to prevent overfitting to some degree.



With these sweeps, I was able to visualize the MSE for different hyperparameter combinations and gain insights about each hyperparameter's correlation with the MSE. In the end, after short-listing the best sweeps, I decided on the most conservative hyperparameters, as the increase in MSE between the more conservative ones and the more overfitting-prone ones was not remarkable when compared to the losses that we would be getting in the test set. I gave special attention to the max_depth, colsample_bytree, n_estimators, and subsample hyperparameters, which, in higher values, increase the risk of overfitting, and parameters such as reg_lambda that, in higher values, counter overfitting without, in my case, significantly worsening the model performance.

Random Forest: As we have lots of outliers in the data that might lead to problems when we use MSE, I also wanted to try a model that is more robust to outliers, and Random Forest seemed like the obvious choice in this case. After tuning the hyperparameters using Weights & Biases, this model gave a slightly worse result compared to XGBoost, both in the validation set and the test set.

Note on train/validation split: Concerns about the sensitivity of MSE to outliers and potential differences in the topologies of cross-validation splits and the final test set led me to adopt a binary train/validation split instead of 5-fold cross-validation for hyperparameter tuning. By assuming that the empirical distribution in the train set closely follows the true underlying data generation process, I implemented a quantile-based train-test split with 80% of the data in each quantile assigned to the train set and 20% to the test set. This approach yielded better estimations of test error and a stronger correspondence between lower validation errors in hyperparameter tuning and lower errors on the platform.

Final predictions

Before training the model, I dropped the latitude and longitude, as I believed that the city column was sufficient and to prevent the risk of overfitting to the latitude and longitude values. After further tuning the XGBoost model’s hyperparameters, I calculated one last time the SHAP and tried dropping the 5 columns with the lowest scores which were n_rooms, floor, elevator, balcony and construction_year. Even though this version of the dataframe led to a small increase in the validation loss, it obtained a slightly better score on the platform, which in the end was my best score.

Conclusions and considerations

This project was very demanding because of the large number of factors affecting the model's performance and the incomplete and inaccurate data. Deciding whether to remove columns, clean the data more carefully, adjust the hyperparameters, or change the model entirely was a time-consuming and mentally demanding process. This highlights the challenges that come with building effective machine learning models.

In the end, the predictors that created the best-performing model were the most obvious ones: the surface area of the house, its distance to the city's center and points of interest, number of bathrooms, conditions, and as average prices vary between different regions and cities, the city. I do not think that it would be correct to say that high expenses lead to high prices, but these two variables are highly correlated, as a house with a high price tag has probably a bigger surface area, and perhaps in a more luxurious condominium, which lead to higher expenses. A surprise was the n_rooms column, which consistently ranked last in feature importance. This might come from the fact that the data on the number of rooms was not accurate, for instance, there were houses with a higher number of bathrooms than rooms, and in the presence of these errors, the number of bathrooms column, which I believe to be more correctly input, yielded more information about the price.