Overview

The Report covers the Kaggle Competition on House Prices in Ames, Iowa using Advanced Regression Techniques. The goal was to create a machine learning model that would accurately predict housing prices. To do this, I used Python to explore the dataset, followed by diving into different process features, and finally experimenting with different modeling strategies to find the best fit for this task.

Data preparation

The Kaggle competition gave access to a train and test dataset (in a .csv format). This was loaded into 2 distinct dataframes using pandas. Before I went into decomposing the data and applying different models, it was critical to understand the data structure itself. Initially, I used a Seaborne visual for this. However, it was not as incisive as I wanted. Thus, for this task, from previous experience, I found the Sweetviz model to be a very good EDA tool (Figure 1: Sweetviz EDA Example). This allowed me to look at the data integrity and structure. Upon initial inspection, I found several columns to contain >70% of missing/null/NaN values. My first instinct was to drop them. However, upon closer inspection I found that these values could signal towards a negative affirmation. For example, for the 'Alley' column, it had 6% values (defined between gravel and pavement) and 94% missing values. These missing values could very well mean no alley access. Thus, I decided not to drop these columns.

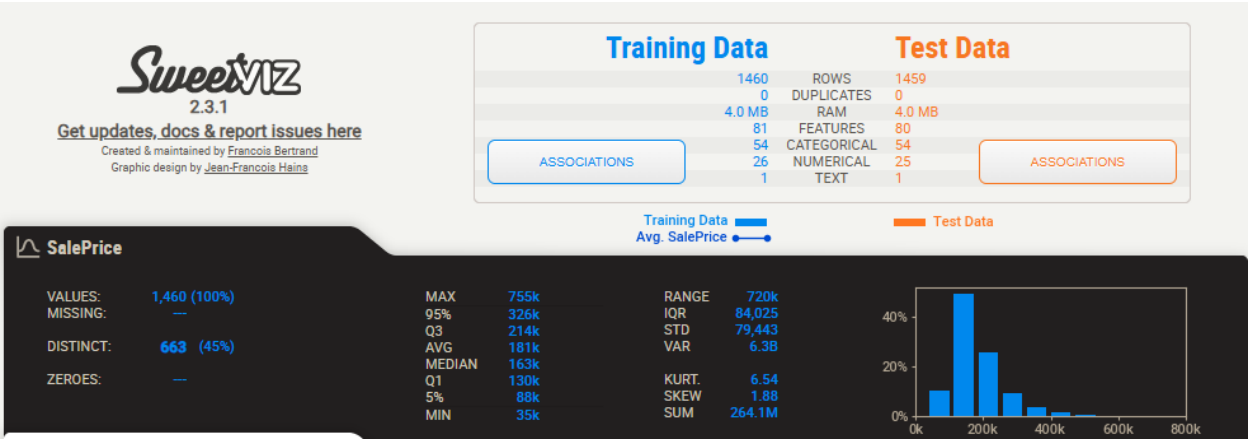

Figure 1: Sweetviz EDA Example

The next step was to see the data types in both the test and train dataframes. This again provides a strong understand of the sort of data at hand and later on (as was proven) was incredibly useful in manipulation and transformation.

To set up the splitting of data into a 80-20 split, I created two dataframes, 'X_full' and 'Y_full'. For the former, I drop 'Id' and 'SalePrice' to remove index and the dependent variable ('SalePrice'). For the latter, I transformed the column using log to reduce data skewness for extreme variables (as also mentioned on Kaggle). Futher data preparation was made (and this is where knowing the data types was helpful) was when accounting for null/missing/NaN, I divided the data into mode and mean. This was done so as some columns (e.g. 'GarageCars') cannot have a float value and will always have a integer; thus, using mode was more useful. On the contrary, some columns (e.g. 'LotFrontage') can very likely be a float as well; thus, a mean transformation was more appropriate. Further columns were created into numerical ('num_cols') and categorical ('cat_cols'). This was useful when using a SimpleImputer and OneHotEncoder to transform the data. Initially, I ran into an error when I concatenated the test and train set and dealt with null/missing/NaN values together. This created a bias between the test-train set by having statistical influence where there should have been none.

Multi-model selection

Coming to the main step of hyperparameter tuning, I divided it into 3 models to check for robustness and best fit. These were: Linear Regression, Random Forest Regressor, and XGBoost. Here are why the parameters values where chosen as is:

1. Linear Regression: naturally no parameters were set as none are required.
2. Random Forest Regressor:
    a. Model n estimators of between [100, 300] was chosen as 100 is the baseline in Scikit to reduce usage of computing power, and 300 is used as a benchmark to see efficiency differential boost vs computing power tradeoff.
    b. Model Max Depth of between [10, None] was chosen to have a balance between overfitted trees (which are averaged to cancel out errors) and a limit of 10 to act as an early "control".
3. XGBoost:
    a. Model n estimators of between [100, 1000] was chosen as 100 gives me a fast baseline and 1000 trees coupled with a low learning rate allows me to capture small patterns.
    b. Model max depth of between [2, 5] was chosen as 2 would highly control for outliers (e.g. niche neighborhoods or building materials), whereas 5 would all to cater to a more incisive and distinct analysis (e.g. "If the house is old, but has high overall quality, and is in a good neighborhood, then price goes up").

c. Learning rate of [0.05, 0.1] was chosen so that the model takes small steps (e.g. 0.05) to correct its errors, but with 1000 trees to work with, it won't run out of time before an optimal price predictions is found.

Final Model Selection

The automated GridSearch systematically evaluated all models and used algorithm with the lowest negative Log-root-mean-squared error (Log-RMSE) on the validation folds as the *Best Model*. To visualize the Best Model efficiency, the actual validation prices were plotted from lowest to highest to get a smooth actual-price curve. The model's predicted prices were then superimposed.

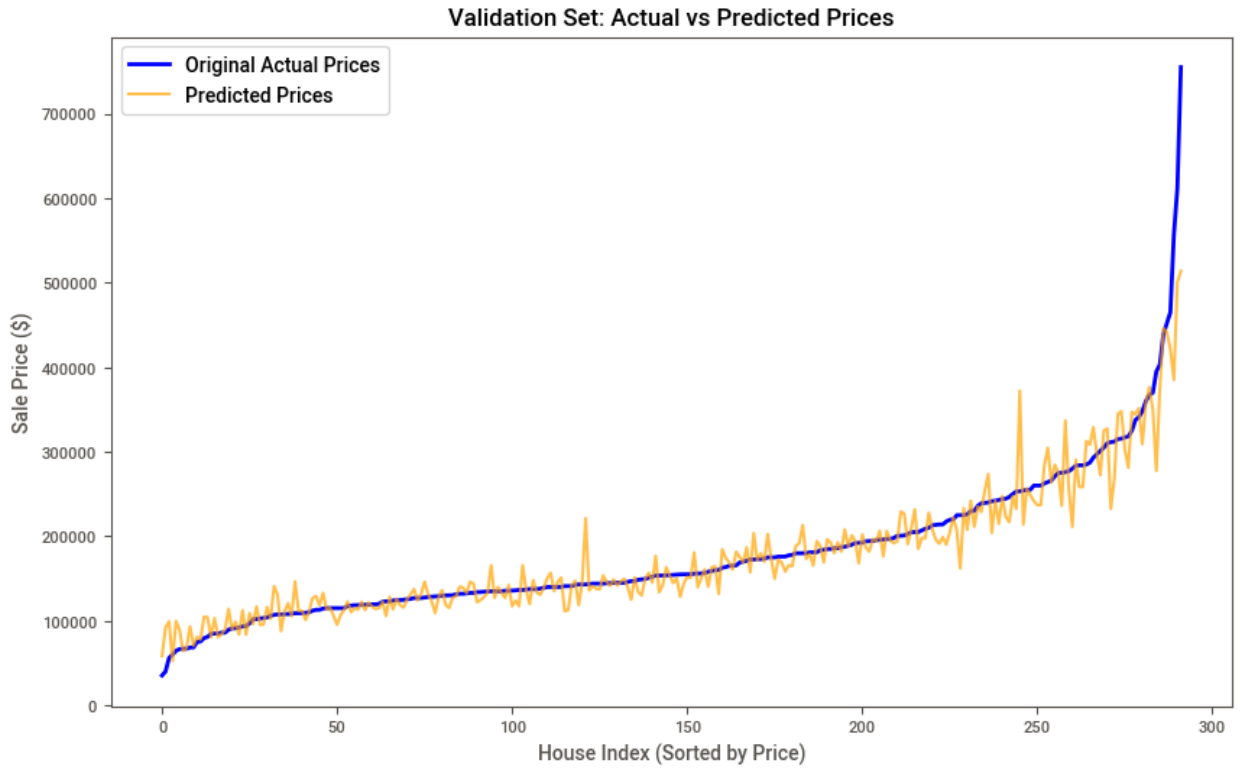

Figure 2: Actual vs Predicted Prices

The visual analysis seen from this graph revealed a distinct performance trend:

Strong Mid-Range Accuracy: Between the US$100,000 and US$250,000, the predicted values were very close to the actual price curve. This shows that the model demonstrated high accuracy in predicting the value of what can be termed middle-class homes, capturing the subtle variations in square footage and overall quality.

Luxury Variance: At the extreme right tail of the distribution (e.g. homes exceeding US$500,000), the model under-predicted actual prices. This can be attributed to various factors such as luxury homes possessing highly unique features that may not be captured in the standard categorical columns, or their low frequency in the training data limiting the model's ability to build on their pricing.

Conclusion

By prioritizing data isolation, feature engineering, and robust multi-model hyperparameter tuning, the pipeline successfully transforms the raw and messy data into highly accurate predictions. One of the final steps of reversing the initial log-transformation (via numpy.expm1) allowed a clean export into output_file.csv. The methodology I used shows that retaining useful information (such as treating missing categorical data as distinct features) and controlling tree depth yields a generalizable model capable of achieving a highly competitive score on Kaggle (Figure 3: Kaggle Submission Score).

Given more time and resources, one could further tweak the parameters to find the best fit. However, to optimize for both of these, coupled with sufficient enough value addition the parameters mentioned were utilized. I would also use other models and try mapping/dropping the columns to have a justified reason for why and how null values where dealt with. A similar logic can be applied to dropping outliers as well. Finally, I could have a stacked model. This would help leverage strengths from multiple models into one super one.

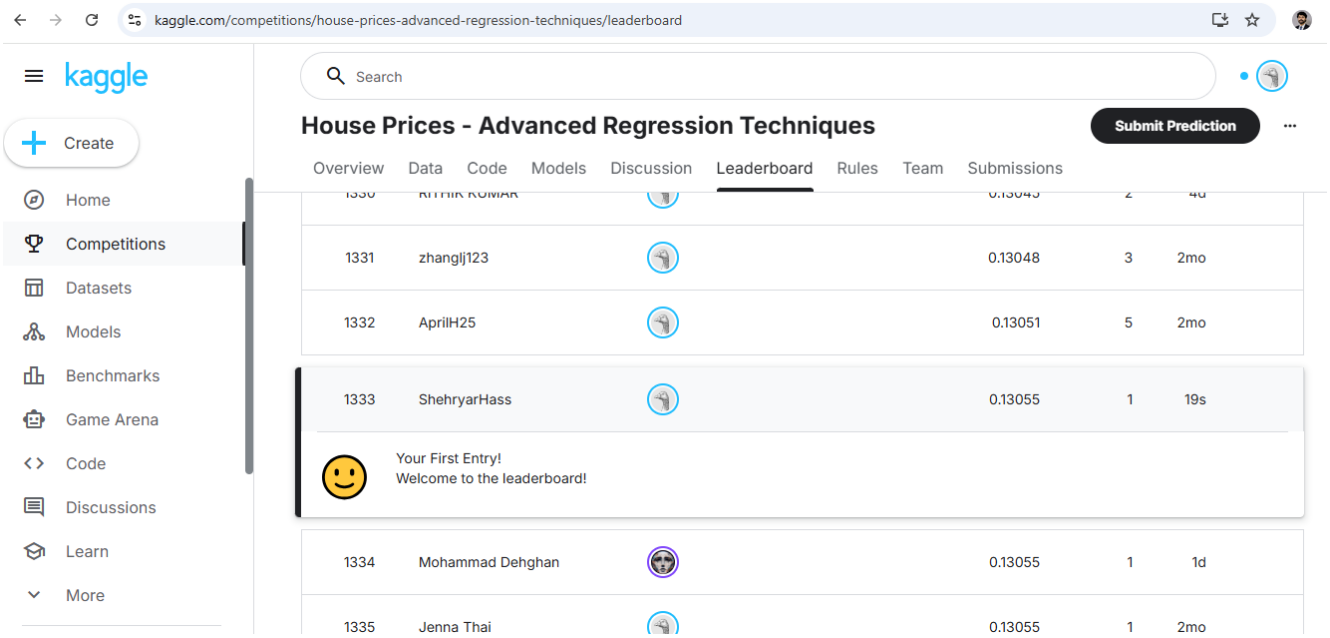

Figure 3: Kaggle Submission Score