Machine Learning Regression

Predicting the value of Apartment’s sale price in Daegu, South Korea

Satrio Bagus Prabowo

Context

- The city of Daegu is experiencing a population decline but an increase in the number of households, indicating that housing demand remains strong.
- Migration of young people to larger cities is putting pressure on housing demand, but government investment in infrastructure has the potential to raise property values.
- Apartment prices fell by about 3.87% in early 2025, reflecting regional market fluctuations.

Problem Statement, Stakeholders, and Problem Solving

Problem:

The apartment market in Daegu exhibits price fluctuations, making it challenging for buyers and investors to accurately assess property values.

Stakeholders:

- Potential buyers
- Investors or Seller

Problem Solving:

Develop a machine learning model to predict apartment sale prices based on key influencing features. The goal is to support homebuyers and investors in making informed and data-driven decisions regarding property purchases and investments.

Workflow

- **Data Understanding**
- **Data Cleansing**
- **Exploratory Data Analysis (EDA)**
- **Data Pre-Processing**
  - Outliers
  - Defining X and y
  - Encoding
  - Train, Test Split
  - Scaling
- **Modelling**
  - Benchmarking Model
  - Hyperparameter Tuning
  - Model Evaluation
  - Model Intepretation
- **Conclusion and Recommendation**

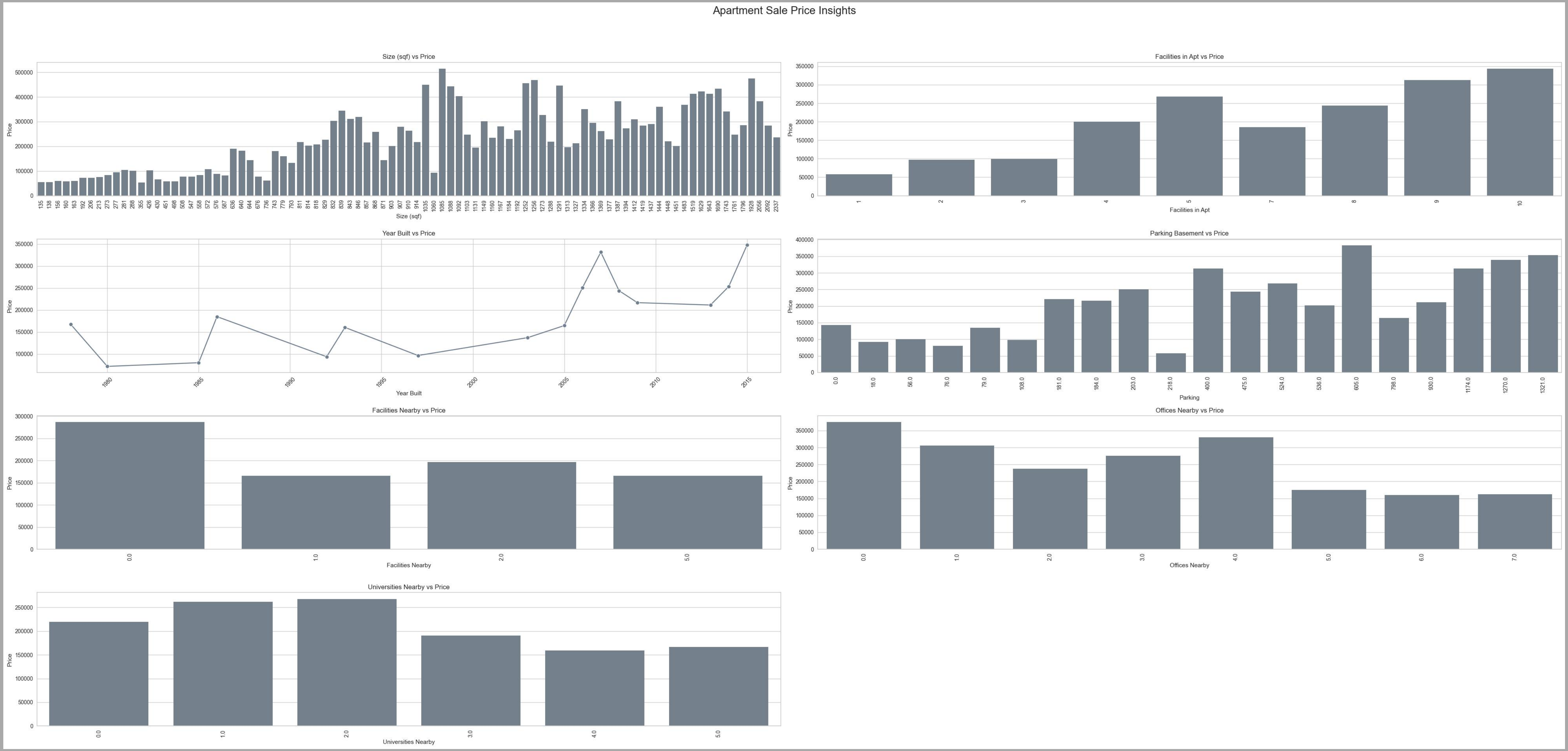
Data Understanding

Feature	Description
Hallway Type	Type of hallway layout in the apartment (e.g., apartment type)
TimeToSubway	Time (in minutes) to the nearest subway
SubwayStation	Name of the nearest subway
N_FacilitiesNearBy(ETC)	Number of nearby miscellaneous facilities (e.g., shops, convenience stores)
N_FacilitiesNearBy(PublicOffice)	Number of nearby miscellaneous facilities (e.g., shops, convenience stores)
N_SchoolNearBy(University)	Number of nearby universities
N_Parkinglot(Basement)	Number of basement parking lots available
YearBuilt	Year the apartment was built
N_FacilitiesInApt	Number of facilities inside the apartment complex (e.g., gym, playground)
Size(sqft)	Size of the apartment unit in square feet
SalePrice	Sale price of the apartment in Korean Won (₩)

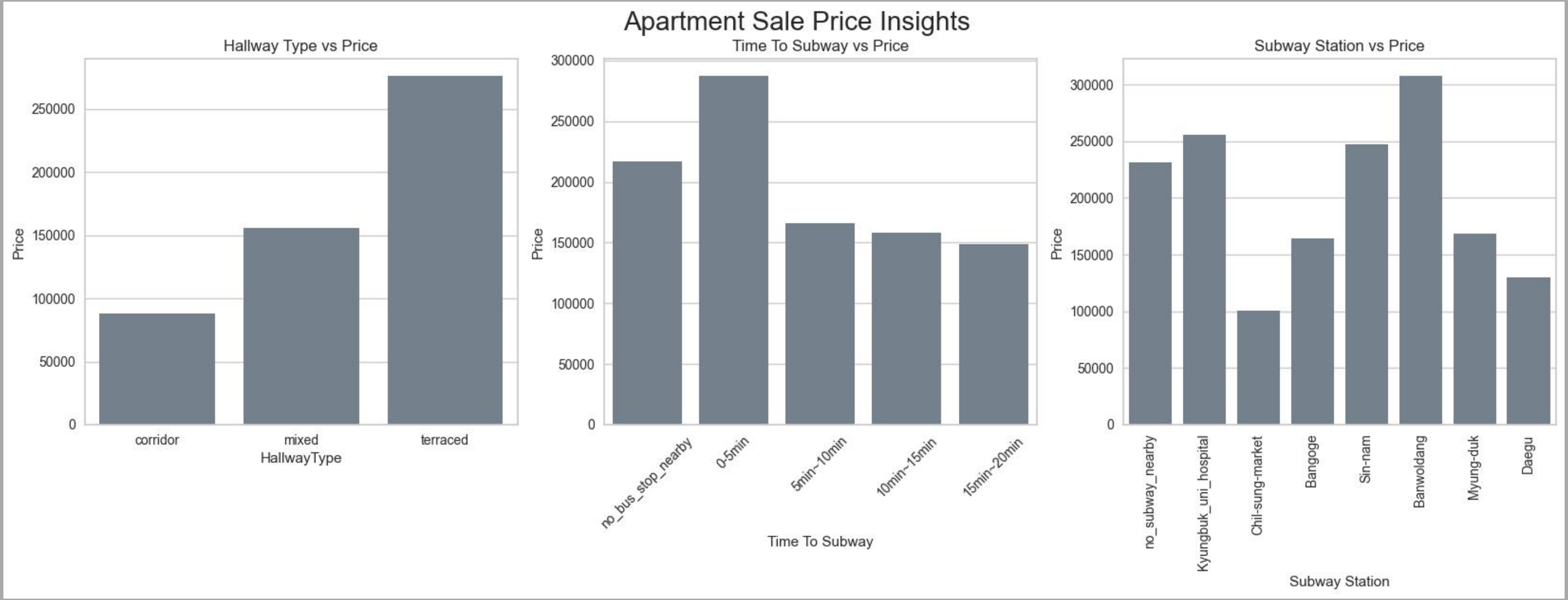
Data Cleansing

- There are no columns with missing values in the dataset.
- There are 1,422 rows in the dataset that appear to be duplicates. However, these cannot be dropped immediately because there is no unique ID or code to distinguish each apartment. It is possible that these rows represent different apartment units that happen to share the same features.
- There are no data type errors or anomalies in any of the columns.

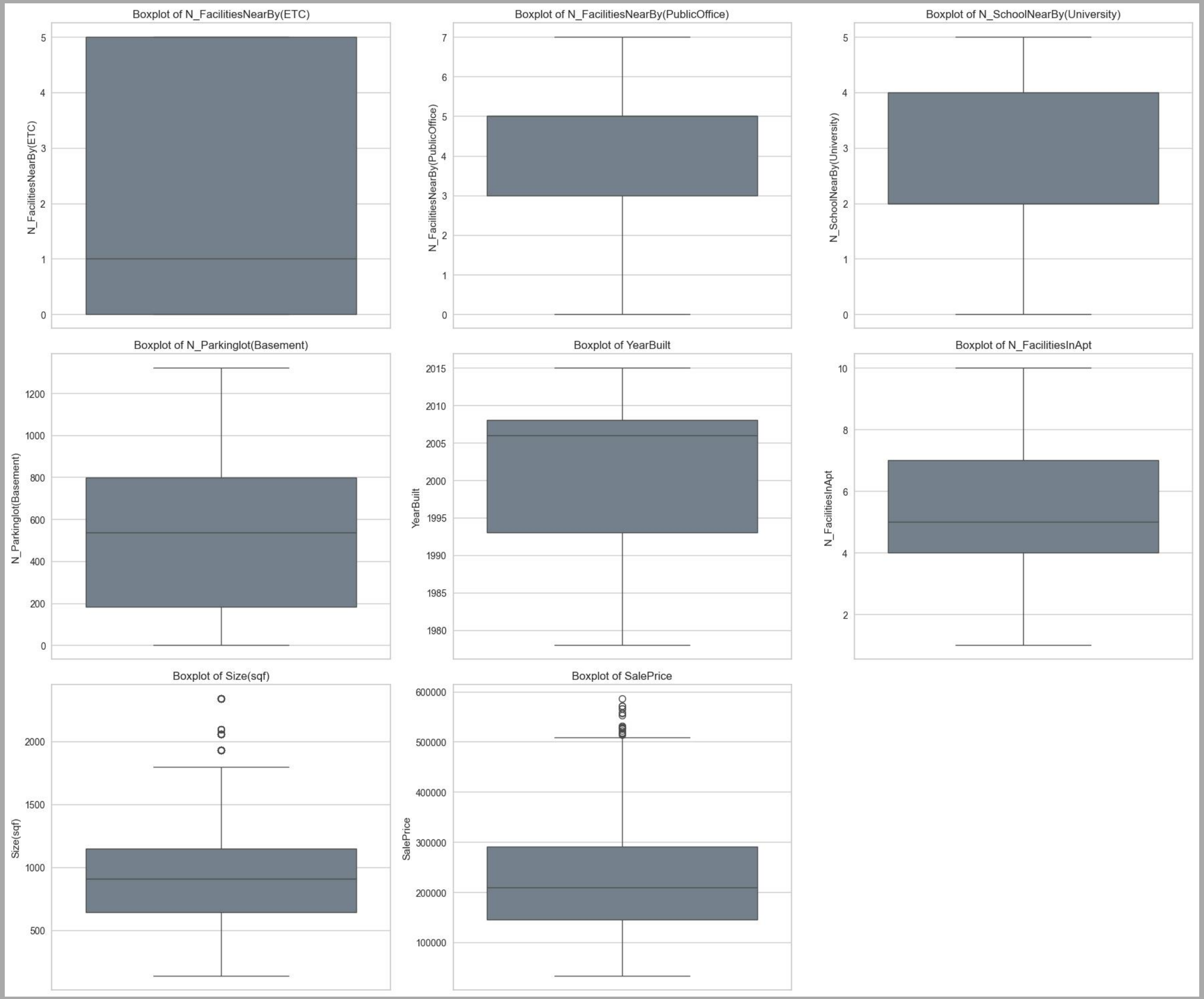
EDA : Numerical Columns vs Sale Price



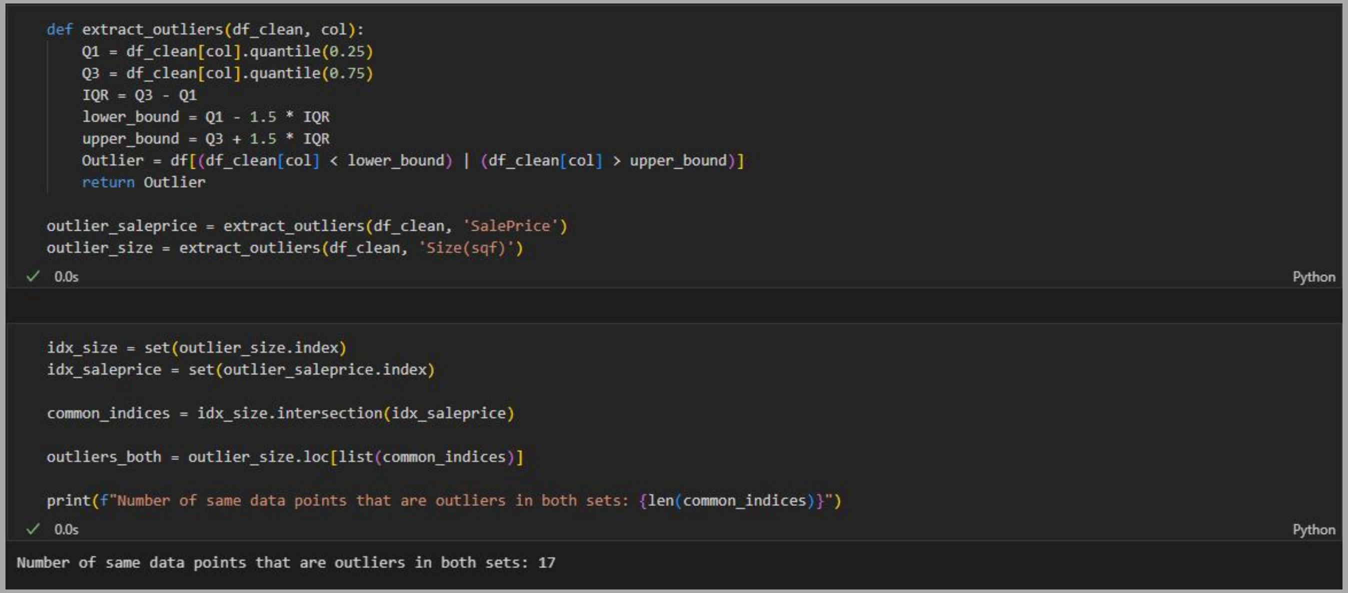
EDA : Categorical Columns vs Sale Price



Data Preparation: Outliers



There are outlier values in the `Size(sqf)` and `SalePrice` columns. Further investigation is needed to check whether there are data points that are outliers in both columns simultaneously. If such data exist, these outliers should not be dropped because, fundamentally, the `Size(sqf)` variable influences the `SalePrice`.



```python
def extract_outliers(df_clean, col):
    Q1 = df_clean[col].quantile(0.25)
    Q3 = df_clean[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    Outlier = df[(df_clean[col] < lower_bound) | (df_clean[col] > upper_bound)]
    return Outlier

outlier_saleprice = extract_outliers(df_clean, 'SalePrice')
outlier_size = extract_outliers(df_clean, 'Size(sqf)')
```

✓ 0.0s Python

```python
idx_size = set(outlier_size.index)
idx_saleprice = set(outlier_saleprice.index)

common_indices = idx_size.intersection(idx_saleprice)

outliers_both = outlier_size.loc[list(common_indices)]

print(f"Number of same data points that are outliers in both sets: {len(common_indices)}")
```

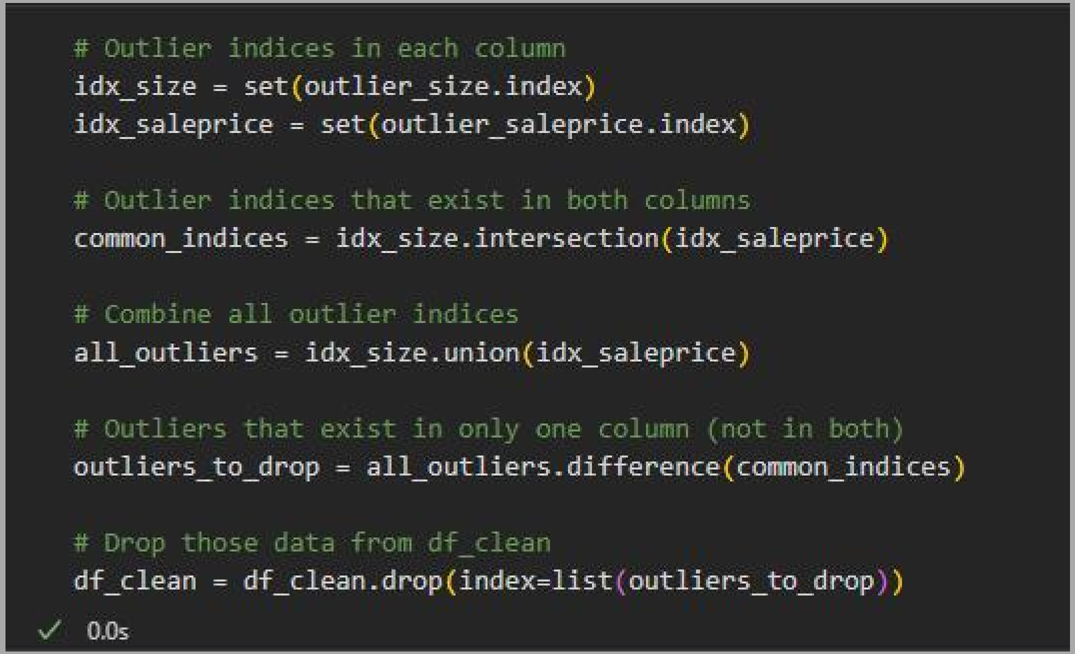
✓ 0.0s Python

```
Number of same data points that are outliers in both sets: 17
```

There are 17 data points that appear as outliers in both Size(sqf) and SalePrice. This indicates that the size strongly influences the sale price. Therefore, these outliers cannot be considered anomalies.

Data that are not included in these 17 common data points can be dropped during the "Data Preparation" phase because they are considered outside the observable data range suitable for building a machine learning model to predict prices

Drop Outliers



```
# Outlier indices in each column
idx_size = set(outlier_size.index)
idx_saleprice = set(outlier_saleprice.index)

# Outlier indices that exist in both columns
common_indices = idx_size.intersection(idx_saleprice)

# Combine all outlier indices
all_outliers = idx_size.union(idx_saleprice)

# Outliers that exist in only one column (not in both)
outliers_to_drop = all_outliers.difference(common_indices)

# Drop those data from df_clean
df_clean = df_clean.drop(index=list(outliers_to_drop))
```

✓ 0.0s

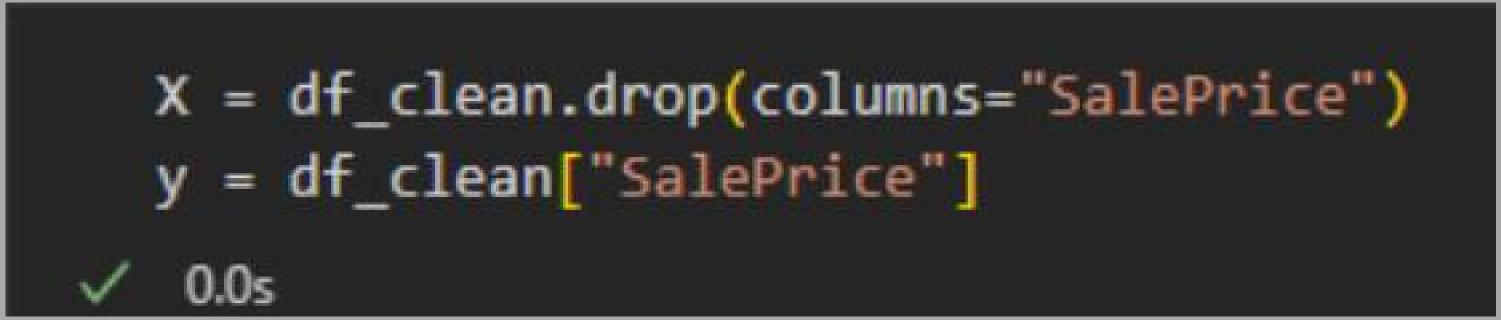
The process involves dropping outlier data except for 17 data points that are outliers in both Size (sqft) and Sale Price.

Data Preparation: Define X and y

Define **X** and **y**, where X contains the feature columns, and y is the target, which is the SalePrice column.

X = 'HallwayType', 'TimeToSubway', 'SubwayStation',
'N_FacilitiesNearBy(ETC)', 'N_FacilitiesNearBy(PublicOffice)',
'N_SchoolNearBy(University)', 'N_Parkinglot(Basement)', 'YearBuilt',
'N_FacilitiesInApt', 'Size(sqf)'

y = 'SalePrice'



```
X = df_clean.drop(columns="SalePrice")
y = df_clean["SalePrice"]
```

✓ 0.0s

Data Preparation: Encoding

The encoding process will be adjusted according to the types of categories present in the columns.

- **One Hot Encoding** for **`HallwayType`** dan **`SubwayStation`**

The variables HallwayType and SubwayStation are categorical features representing types or names without any inherent order or ranking. Therefore, One Hot Encoding is suitable for these variables, as it converts each category into a binary vector without implying any ordinal relationship. This allows the model to treat each category as a separate and distinct feature.

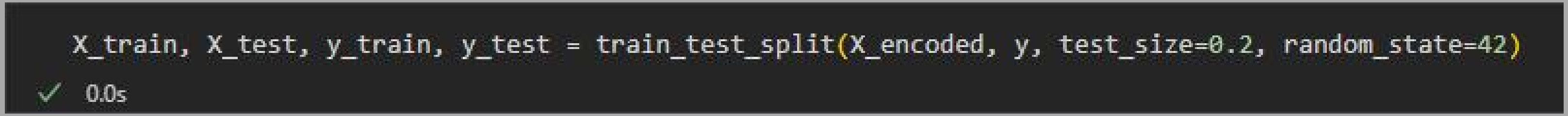
- **Ordinal Encoding** for **`TimeToSubway`**

The `TimeToSubway` variable is ordinal because the categories represent meaningful ordered intervals of travel time (e.g., 0-5 minutes is closer than 5-10 minutes). Therefore, Ordinal Encoding is the appropriate method to convert these categories into numerical values that preserve their natural order. This helps the model understand the relative proximity to the subway station.

Data Preparation: Splitting Data

Splitting is done to divide the data into training and testing sets. This is necessary to prevent information leakage during model training and evaluation, ensuring that the model's performance is assessed on unseen data.

Splitting using a proportion of 80% for Training data and 20% for Test data, with a random seed value of 42.



```
X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, test_size=0.2, random_state=42)
✓ 0.0s
```

Data Preparation: Scaling

Scaling is the process of changing the scale or range of feature values (input variables) to be within a certain range or to have a specific distribution. The goal is to ensure that the features in the dataset have comparable scales so that the machine learning model can learn more effectively and quickly.

From the available data, the columns that need scaling are those with continuous numerical values or columns that have a very large numerical range. Scaling is necessary so that the machine learning model can work optimally and the features have comparable scales.

Robust Scaling is chosen because it is particularly effective when the data contains outliers or is not normally distributed. Unlike other scalers that rely on the mean and standard deviation, RobustScaler uses the median and the interquartile range (IQR) to scale the data. This makes it more resistant to the influence of outliers and better suited for features with skewed distributions. Since the histplot analysis shows that most of the columns are not normally distributed—even though they do not contain extreme outliers—RobustScaler is a safe and reliable choice to normalize the scale of features without distorting their internal relationships.

Columns to be scaled:

1. `TimeToSubway` (result of ordinal encoding, discrete numerical values, usually scaled to make the model more sensitive to differences in levels)

2. `N_FacilitiesNearBy(ETC)` (number of facilities, numerical)

3. `N_FacilitiesNearBy(PublicOffice)` (number of facilities, numerical)

4. `N_SchoolNearBy(University)` (number of facilities, numerical)

5. `N_Parkinglot(Basement)` (number of facilities, numerical)

6. `YearBuilt` (year, large range, needs scaling to prevent dominance)

7. `N_FacilitiesInApt` (number of facilities, numerical)

8. `Size(sqf)` (area, numerical with large range)

Benchmarking Model

- Ridge

Ridge regression is a linear regression model with L2 regularization. It is especially useful when there is multicollinearity among features, as Ridge can stabilize the regression coefficients and prevent overfitting. Although it is a linear model, Ridge remains relevant as a baseline model to understand the basic performance of a stabilized linear approach.

- Decision Tree Regressor

This model is a non-linear algorithm and is very suitable for handling data that do not have a normal distribution or linear relationships among features. Decision Trees are naturally resistant to multicollinearity and do not require scaling. They are well-suited to capture complex relationships in property data such as location, size, and facilities.

- KNN Regressor

KNN works based on the proximity (similarity) between data points, making it very sensitive to scale and outliers. However, with proper preprocessing (such as RobustScaler), KNN can become an effective model, especially for small to medium-sized datasets. It does not make any assumptions about the data distribution shape, making it suitable for data that are not normally distributed.

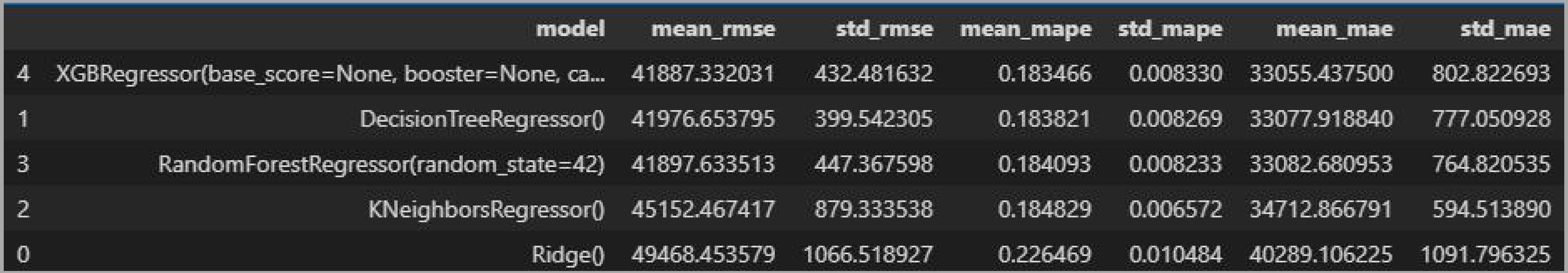
- Random Forest

Random Forest is a non-linear ensemble model that combines many decision trees. It is very robust in handling non-normal data, multicollinearity, and noise. Random Forest also provides feature importance, which can aid interpretation. It is one of the models well-suited for property datasets with many complex and interrelated variables.

- XG Boost

XGBoost is a more advanced boosting model than Random Forest. It builds trees sequentially by correcting errors from previous trees. XGBoost excels at handling outliers, multicollinearity, and non-normally distributed data. Additionally, XGBoost often delivers the best performance in numerical predictions and has strong regularization controls to prevent overfitting

Benchmarking Model



	model	mean_rmse	std_rmse	mean_mape	std_mape	mean_mae	std_mae
4	XGBRegressor(base_score=None, booster=None, ca...	41887.332031	432.481632	0.183466	0.008330	33055.437500	802.822693
1	DecisionTreeRegressor()	41976.653795	399.542305	0.183821	0.008269	33077.918840	777.050928
3	RandomForestRegressor(random_state=42)	41897.633513	447.367598	0.184093	0.008233	33082.680953	764.820535
2	KNeighborsRegressor()	45152.467417	879.333538	0.184829	0.006572	34712.866791	594.513890
0	Ridge()	49468.453579	1066.518927	0.226469	0.010484	40289.106225	1091.796325

Based on the benchmarking results, the model with the lowest RMSE, MAPE, and MAE values is XGBoost. XGBoost is chosen because it has the lowest mean MAPE, supported by also low RMSE and MAE values, indicating high accuracy. Additionally, the relatively small standard deviation values for these metrics demonstrate that the model's performance is stable and consistent across different cross-validation folds. This stability further confirms that XGBoost is the most reliable and robust model for this prediction task.

Hyperparameter Tuning

- model__colsample_bytree

The fraction of features (columns) randomly sampled for each tree. A value between 0 and 1; for example, 0.8 means 80% of features are used per tree to help reduce overfitting.

- model__gamma

The minimum loss reduction required to make a further partition on a leaf node. Higher values make the algorithm more conservative by preventing splits that do not improve the model significantly.

- model__learning_rate

Also called eta, it controls the step size shrinkage used in update to prevent overfitting. Smaller values slow down learning but can improve model robustness.

- model__max_depth

Maximum depth of each decision tree. Increasing this value makes the model more complex and prone to overfitting.

- model__min_child_weight

Minimum sum of instance weight (hessian) needed in a child node for a split to occur. Larger values make the algorithm more conservative.

- model__n_estimators

The number of boosting rounds or trees to build.

- model__reg_alpha

L1 regularization term on weights (Lasso regression). Increasing this value can make the model more sparse and reduce overfitting.

- model__reg_lambda

L2 regularization term on weights (Ridge regression). It helps control model complexity and prevent overfitting.

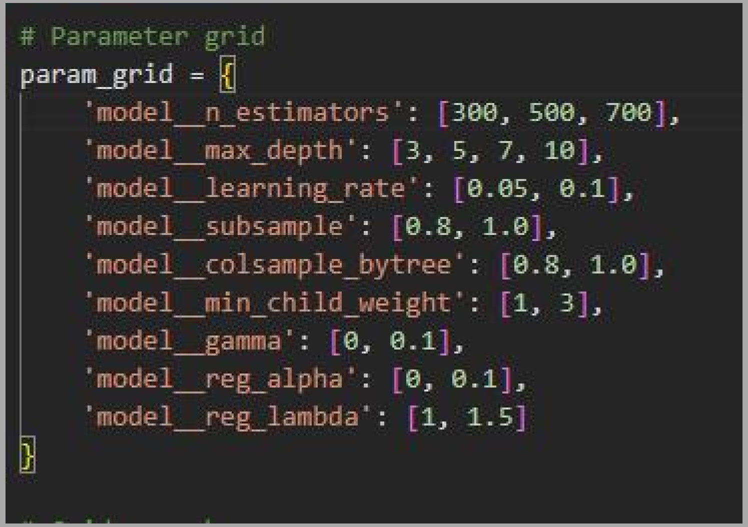
- model__subsample

The fraction of training instances randomly sampled for each tree. Values between 0 and 1; lower values prevent overfitting by introducing randomness.

First Hyperparameter Tuning

The MAPE score of the best XGBoost model based on the above grid search results is 0.18235780810956823 with the following parameters:

1. `model__colsample_bytree`: 0.8
2. `model__gamma`: 0
3. `model__learning_rate`: 0.1
4. `model__max_depth`: 3
5. `model__min_child_weight`: 1
6. `model__n_estimators`: 300
7. `model__reg_alpha`: 0.1
8. `model__reg_lambda`: 1
9. `model__subsample`: 0.8



```python
# Parameter grid
param_grid = {
    'model__n_estimators': [300, 500, 700],
    'model__max_depth': [3, 5, 7, 10],
    'model__learning_rate': [0.05, 0.1],
    'model__subsample': [0.8, 1.0],
    'model__colsample_bytree': [0.8, 1.0],
    'model__min_child_weight': [1, 3],
    'model__gamma': [0, 0.1],
    'model__reg_alpha': [0, 0.1],
    'model__reg_lambda': [1, 1.5]
}
```

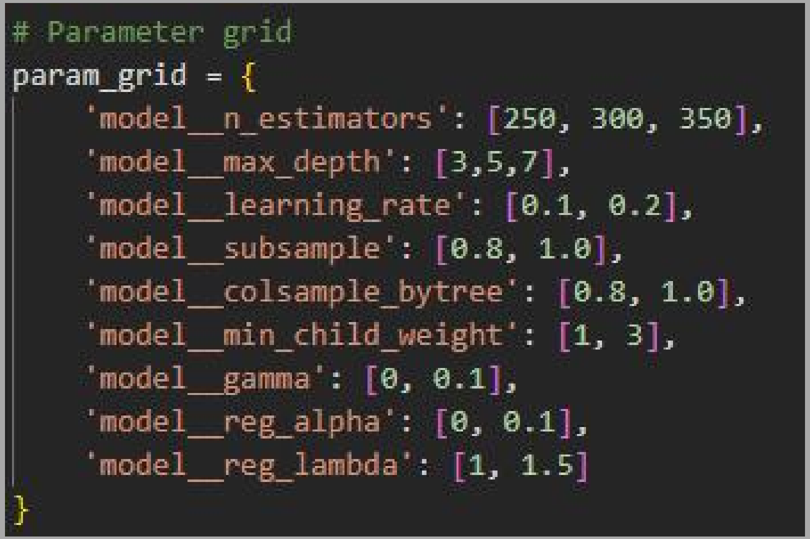
However, to ensure finding a model that can further reduce the MAPE score, hyperparameter tuning can be performed again based on these previous results.

Second Hyperparameter Tuning

After performing two rounds of hyperparameter tuning, the XGBoost model achieved a MAPE score of 0.1819722486577716 with the following parameters:

1. `model__colsample_bytree`: 0.8
2. `model__gamma`: 0
3. `model__learning_rate`: 0.2
4. `model__max_depth`: 3
5. `model__min_child_weight`: 1
6. `model__n_estimators`: 250
7. `model__reg_alpha`: 0.1
8. `model__reg_lambda`: 1
9. `model__subsample`: 0.8

←

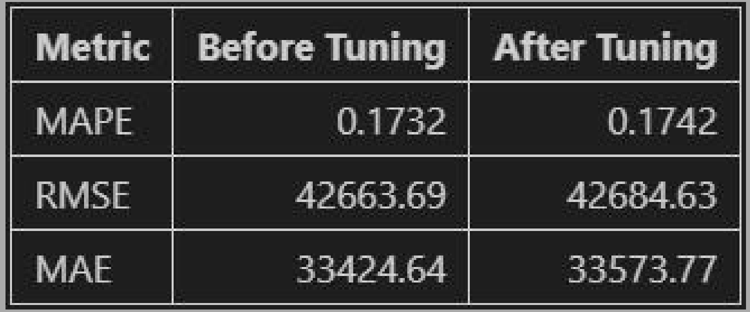


```
# Parameter grid
param_grid = {
    'model__n_estimators': [250, 300, 350],
    'model__max_depth': [3,5,7],
    'model__learning_rate': [0.1, 0.2],
    'model__subsample': [0.8, 1.0],
    'model__colsample_bytree': [0.8, 1.0],
    'model__min_child_weight': [1, 3],
    'model__gamma': [0, 0.1],
    'model__reg_alpha': [0, 0.1],
    'model__reg_lambda': [1, 1.5]
}
```

Therefore, this result can be considered the best XGBoost model obtained so far.

This means that by iteratively tuning the hyperparameters based on previous results, the model’s predictive accuracy improved slightly, reflected in the lower MAPE score. The selected parameters indicate a relatively shallow tree depth with moderate learning rate and subsampling, which balance bias and variance well for this dataset.

Comparison Between Before and After Tuning on Data Test



Metric	Before Tuning	After Tuning
MAPE	0.1732	0.1742
RMSE	42663.69	42684.63
MAE	33424.64	33573.77

- Very Small Differences in Metrics

The MAPE, RMSE, and MAE values after tuning are slightly higher compared to before tuning, but the differences are very small (around 0.001 for MAPE and less than 30 for RMSE and MAE). This indicates that the model’s performance is practically similar before and after tuning.

- Possible Data Variation and Randomness

These small differences can be caused by data variation, the way the train-test split was done, or randomness in the training and prediction processes. This is common in machine learning model evaluation.

- Tuning Aims for Generalization, Not Just Performance on One Test Set

Hyperparameter tuning usually optimizes the model to be more stable and perform well across various subsets of data, not just to improve performance on a single test set. Therefore, a slight increase in error on one test set does not necessarily mean tuning failed.

Model Evaluation: Residual Plot

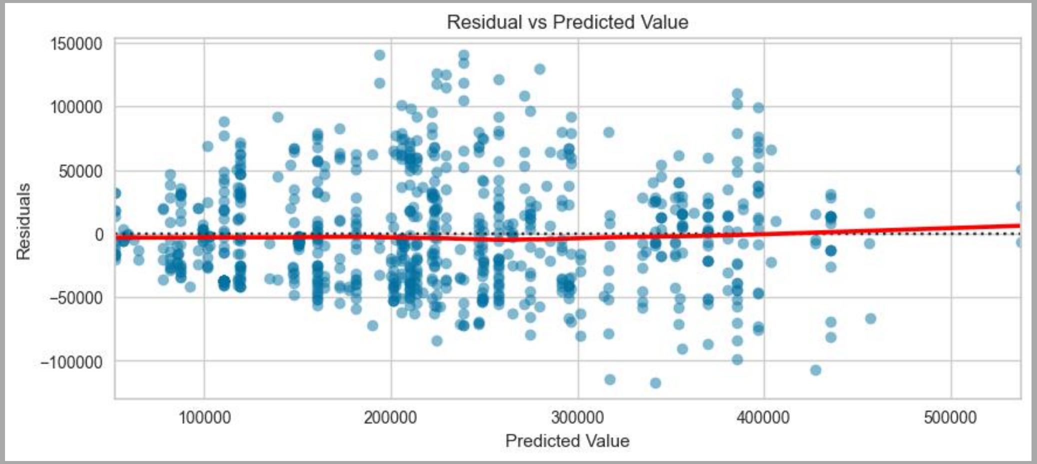
A residual plot is a scatter plot that displays the residuals on the vertical axis and the independent (explanatory) variable on the horizontal axis. A residual is the difference between the actual observed value and the predicted value from a regression model (residual = actual y − predicted y).

The residual plot helps to visually assess how well a regression model fits the data. Ideally, the residuals should be randomly scattered around the horizontal line at zero, showing no clear pattern. This randomness indicates that the model's assumptions are valid and that the model fits the data well.

In summary, a residual plot is used to:

- Evaluate the goodness of fit of a regression model.
- Detect non-linearity, unequal error variances, or outliers.
- Diagnose problems with the model assumptions.

The horizontal line at zero in the plot serves as a reference, where residuals above the line indicate under-prediction and residuals below indicate over-prediction by the model.



- Red trend line is almost flat:

The red (fit) line is nearly parallel to the X-axis, indicating there is no systematic bias across the entire range of predicted values.

- Residuals are fairly evenly spread:

The residuals are scattered around zero throughout the range of predicted values, with no funnel-shaped pattern (no widening or narrowing), which means the error variance is stable across the range.

- No severe heteroscedasticity:

The residual points do not form a fan-shaped pattern, indicating the model errors are fairly consistent across all predicted values.

- Presence of some outliers:

There are points far from the zero line, indicating some data are difficult to predict accurately.

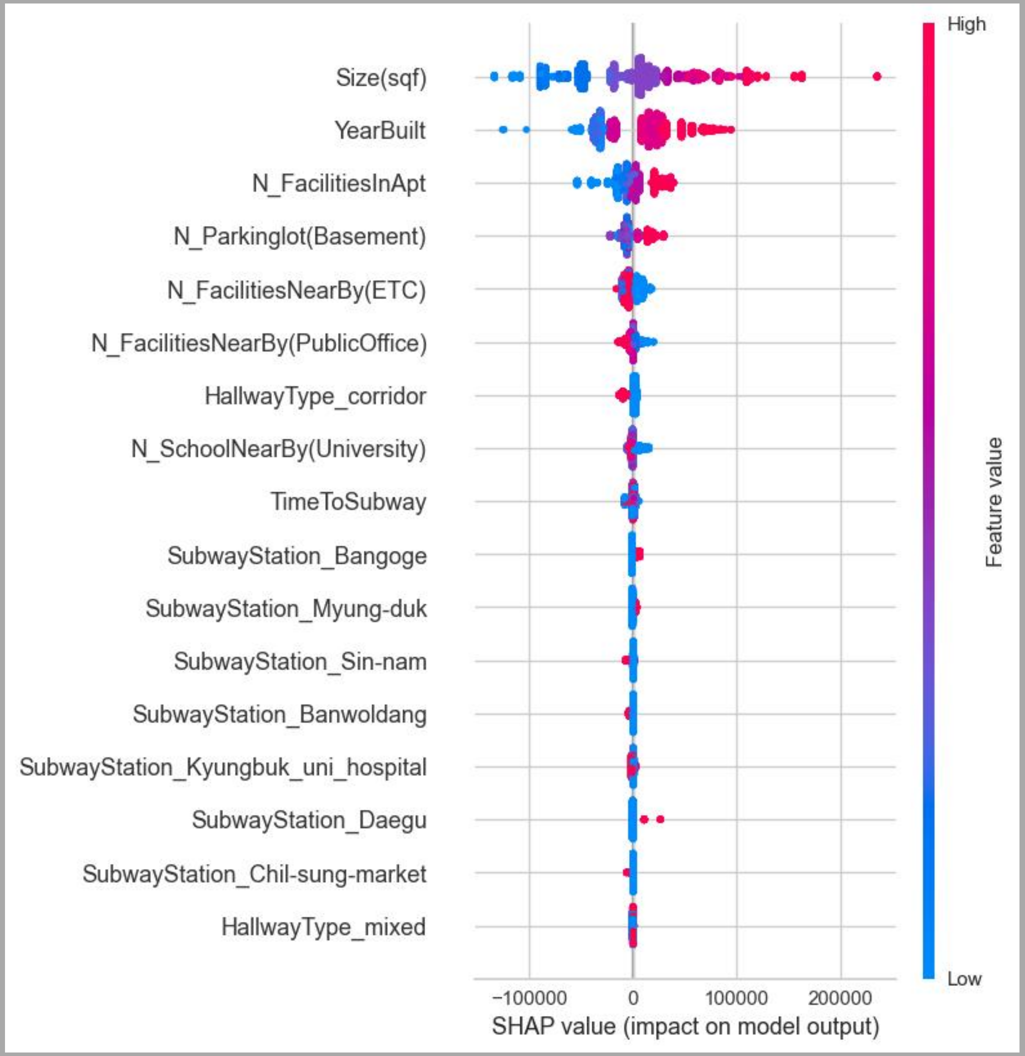
Model Intepretation: Shap

SHAP (Shapley Additive exPlanations) is a method for interpreting machine learning models based on cooperative game theory, specifically the Shapley values concept. It quantifies the contribution of each feature to a model's prediction by fairly distributing the prediction difference from a baseline across all features. SHAP calculates these contributions by considering all possible combinations of features, providing an additive explanation that sums up to the difference between the actual prediction and the average prediction. This approach enables both local explanations (for individual predictions) and global insights (overall feature importance), making complex or black-box models more transparent and understandable. SHAP is widely used to increase trust in machine learning models by clearly showing how each feature influences the outcome.

Basic Concept of SHAP:

- SHAP views the features in the data as "players" in a coalition game, and the model's prediction as the "payout" that must be fairly distributed among these features.
- The Shapley value calculates the average marginal contribution of each feature by considering all possible combinations of the other features in the model.
- The result is a SHAP value for each feature that indicates how much that feature "pushes" the prediction from the average value (baseline) towards the final predicted value.

Model Intepretation: Shap



- **Size (sqf)**

The larger the apartment size, the higher the selling price.

- **YearBuilt**

The newer the construction year of the apartment, the higher the price.

- **N_FacilitiesInApt**

The more facilities available in the apartment, the higher the selling price.

- **N_Parkinglot (Basement)**

The more parking spots available, the higher the price can be.

- **N_SchoolNearBy (University)**

The more public facilities (such as universities) near the apartment, the selling price can decrease.

Conclussion

Bussiness Conclussion:

The best model tested achieved a **MAPE value of 0.1742, or 17.4%**, indicating that it can predict approximately 82.6% of apartment prices accurately based on the given data. This model can serve as a useful reference for homebuyers when evaluating the price of an apartment they intend to purchase, with the understanding that they should prepare for a potential **additional cost of up to 17.4%** above the predicted price, as the model may underestimate the actual price by this margin. Similarly, investors or apartment sellers can use this predictive valuation model as a guideline for setting selling prices, while keeping in mind that the predicted **price could deviate by 17.4%**, potentially resulting in valuations lower than the actual market price. It is also important to recognize that pricing is influenced by factors beyond the dataset, such as government policies, changes in local facilities, or other variables affecting property value. Therefore, to maintain its relevance and improve accuracy, this predictive model should be regularly updated to reflect current market conditions.

Model Conclussion:

The machine learning model used is an **XGBoost Regressor** with the following parameter specifications:
1. `model__colsample_bytree`: 0.8
2. `model__gamma`: 0
3. `model__learning_rate`: 0.2
4. `model__max_depth`: 3
5. `model__min_child_weight`: 1
6. `model__n_estimators`: 250
7. `model__reg_alpha`: 0.1
8. `model__reg_lambda`: 1
9. `model__subsample`: 0.

This model achieves a Mean Absolute Percentage Error (MAPE) of 17.4%, meaning it accurately predicts approximately 82.6% of the data.

This result indicates the inherent limitation of the model, as it cannot perfectly predict 100% of a company’s selling price.

Recommendation

The machine learning model developed possesses inherent limitations in its predictive capabilities. To mitigate these limitations, it is advisable to perform regular updates by retraining the model with new and relevant data. Such periodic updates are particularly beneficial when additional features become available, which can enhance the model’s accuracy. For instance, incorporating data related to location, proximity to the city center, or other pertinent variables may significantly improve predictive performance.

Furthermore, users intending to employ this predictive model are encouraged to consider external factors that may influence valuation outcomes. These include governmental policies, market dynamics, and fluctuations driven by supply and demand. Accounting for such contextual elements will contribute to more precise and reliable predictions.