Cleaning Data Report for House Price Prediction Project

Handling Outliers in the Dataset

Objective
In the context of house price prediction, outliers in key numerical features such as *OverallQual* and *GrLivArea* can skew the model's predictions and affect its performance. These outliers need to be treated effectively to ensure the dataset is well-prepared for model training. We developed a custom function to handle outliers flexibly and efficiently.

Methodology
To address outliers in the dataset, we utilized the **Interquartile Range (IQR)** method, which is effective for detecting values that fall significantly outside the typical range of a variable.

1. **IQR Calculation**:
   The IQR is the range between the 25th percentile (Q1) and the 75th percentile (Q3). Values that lie below Q1 by more than 1.5 times the IQR or above Q3 by more than 1.5 times the IQR are considered outliers. These are defined as:
   - **Lower Bound**: Q1−1.5×IQRQ1 - 1.5 \times IQRQ1−1.5×IQR
   - **Upper Bound**: Q3+1.5×IQRQ3 + 1.5 \times IQRQ3+1.5×IQR

Code Implementation
We developed a Python function, `handle_outlier`, to handle outliers using the IQR method. This function offers flexibility by allowing the user to either cap outliers or remove them based on the dataset's needs.

```
def handle_outlier(col,df):
  q1=df[col].quantile(0.25)
  q3=df[col].quantile(0.75)
  IQR=q3-q1
  lower_b=q1-1.5*IQR
  upper_b=q3+1.5*IQR
  for i in range(len(df)):
    if df.loc[i,col]>upper_b :  df.loc[i,col]=upper_b
    elif df.loc[i,col]<lower_b:  df.loc[i,col]=lower_b
handle_outlier('OverallQual',df)
handle_outlier('GrLivArea',df)
```

Conclusion

By using this outlier-handling approach, we effectively reduced the potential impact of extreme values in the dataset. This ensures the model can focus on learning meaningful patterns without being influenced by anomalies, leading to better prediction accuracy.

Handling Missing Values for Basement Features

Objective
To handle missing values in the basement-related categorical columns, we assigned a meaningful category to represent the absence of a basement in the dataset.

Methodology
The columns related to basement characteristics, including *BsmtQual*, *BsmtCond*, *BsmtExposure*, *BsmtFinType1*, and *BsmtFinType2*, had missing values. These missing values likely indicate the absence of a basement, so we replaced them with the category **'None'** to reflect this condition.

```
bsmt_str_cols = ['BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1',
'BsmtFinType2']
df[bsmt_str_cols] = df[bsmt_str_cols].fillna('None')
```

Conclusion
By filling missing values with 'None', we ensured that the data accurately reflects the absence of basement features without introducing incorrect assumptions.

Handling Missing Values for Garage Features

Objective
To address missing values in garage-related categorical columns, we replaced the missing entries with a category that reflects the absence of a garage.

Methodology
The columns related to garage characteristics, including *GarageType*, *GarageFinish*, *GarageQual*, and *GarageCond*, had missing values. These missing values likely indicate that there is no garage for certain houses. Therefore, we replaced these missing values with the category **'None'** to signify the absence of a garage.

```
gar_str_cols = ['GarageType', 'GarageFinish', 'GarageQual', 'GarageCond']
df[gar_str_cols] = df[gar_str_cols].fillna('None')
```

Conclusion
By imputing the missing values with 'None', we ensured the dataset accurately represents properties without garages, preventing any misinterpretation of missing data.

Handling Missing Values for Masonry Veneer Type

Objective
To handle missing values in the *MasVnrType* column, which indicates the type of masonry veneer used in a house, we assigned a value representing the absence of this feature.

Methodology
The *MasVnrType* column had missing values, likely indicating that no masonry veneer was used. To reflect this, the missing values were replaced with the category **'None'**.

```
df["MasVnrType"] = df["MasVnrType"].fillna("None")
```

Conclusion
By filling missing values with 'None', the data now correctly represents houses without masonry veneer, ensuring consistency in the dataset.

Handling Missing Values for Fireplace Quality

Objective
To address missing values in the *FireplaceQu* column, which represents the quality of a fireplace, we assigned a value that indicates the absence of a fireplace.

Methodology
The *FireplaceQu* column had missing values, likely indicating that a house does not have a fireplace. To capture this, the missing values were replaced with the category **'None'**.

```
df["FireplaceQu"] = df["FireplaceQu"].fillna("None")
```

Conclusion
By imputing the missing values with 'None', the dataset now properly reflects properties without fireplaces, maintaining consistency in the data.

Handling Missing Values for Electrical System

Objective
To ensure data completeness in the *Electrical* feature, which indicates the type of electrical system in a house, we removed rows with missing values for this feature.

Methodology
The *Electrical* column contained a small number of missing values. Since this feature is essential and categorical imputation might introduce incorrect assumptions, we opted to remove rows with missing values. This ensures that only rows with valid entries for the electrical system remain.

```
df = df.dropna(axis=0, subset=['Electrical'])
```

Conclusion
By dropping rows with missing *Electrical* values, we maintained the integrity of the dataset without introducing assumptions about unknown electrical systems.

Handling Missing Values for Garage Year Built

Objective
To handle missing values in the *GarageYrBlt* column, which represents the year a garage was built, we assigned a value that indicates the absence of a garage.

Methodology
The *GarageYrBlt* column had missing values, likely corresponding to houses without garages. Instead of imputing with a year, we replaced the missing values with **0** to clearly indicate that no garage exists for those properties.

```
df['GarageYrBlt'] = df['GarageYrBlt'].fillna(0)
```

Conclusion
By filling missing values with 0, we accurately represent properties without garages while ensuring the column remains numeric for further analysis.

Handling Missing Values for Lot Frontage

Objective
To address missing values in the *LotFrontage* column, which represents the linear feet of street-connected property, we imputed the missing values using the average *LotFrontage* for each neighborhood.

Methodology
The *LotFrontage* column contained missing values. Since lot sizes can vary by neighborhood, we used **group-based imputation** to fill missing values. Specifically, the mean *LotFrontage* for each neighborhood was calculated and used to impute the missing values, ensuring the imputation aligns with local property characteristics.

```
df['LotFrontage'] =
df.groupby('Neighborhood')['LotFrontage'].transform(lambda val:
val.fillna(val.mean()))
```

Conclusion
By using the neighborhood-specific mean to fill missing values, we ensured that the imputation reflects neighborhood characteristics, preserving the integrity of the *LotFrontage* feature for model training.

Handling Missing Values for Masonry Veneer Area

Objective
To handle missing values in the *MasVnrArea* column, which represents the area of masonry veneer in square feet, we assigned a value that indicates no masonry veneer.

Methodology
The *MasVnrArea* column had missing values, likely corresponding to houses without masonry veneer. To reflect this, the missing values were replaced with **0**, indicating that no masonry veneer is present.

```
df["MasVnrArea"] = df["MasVnrArea"].fillna(0)
```

Conclusion
By filling missing values with 0, we accurately captured properties without masonry veneer, ensuring consistency while keeping the column numeric for further analysis.

Handling Missing Values for Specific Entries in Masonry Veneer Area

Objective
To address missing values in the *MasVnrArea* column for specific entries, we filled them using the mean *MasVnrArea* value based on the corresponding *MasVnrType*.

Methodology
For the records with indices 688 and 1241, the *MasVnrArea* values were missing. Instead of filling these with a general value, we imputed the missing values with the mean *MasVnrArea* based on the respective *MasVnrType* (i.e., the type of masonry veneer). This ensures that the imputation aligns with the typical veneer area for each type.

```
df.loc[688, 'MasVnrArea'] = df_mean[df.loc[688, 'MasVnrType']]

df.loc[1241, 'MasVnrArea'] = df_mean[df.loc[1241, 'MasVnrType']]
```

Conclusion
By using the type-specific mean *MasVnrArea*, we provided a more accurate and contextually appropriate imputation for these specific missing values, preserving the integrity of the dataset.

Dropping Irrelevant or Sparse Columns

Objective
To streamline the dataset and remove columns with sparse or irrelevant information, we dropped several features that were not essential for predicting house prices.

Methodology
The following columns were dropped:

- **Id**: A unique identifier for each property, which does not contribute to the predictive power of the model.
- **PoolQC**, **MiscFeature**, **Alley**, **Fence**: These columns had a high percentage of missing values. Retaining these sparse features could introduce noise into the model without adding significant value.

```
df = df.drop(['Id', 'PoolQC', 'MiscFeature', 'Alley', 'Fence'], axis=1)
```

Conclusion
By dropping these columns, we reduced the complexity of the dataset and ensured that only relevant features are used for model training, enhancing model performance and interpretability.

Log Transformation of Property Sale Price

Objective
To normalize the distribution of the *Property_Sale_Price* column and reduce the effect of extreme values, we applied a natural logarithm transformation.

Methodology
The distribution of house prices (*Property_Sale_Price*) is typically right-skewed, with a few very high values. To address this skewness and make the data more suitable for modeling, we created a new column, **Property_Sale_Price_natural_log**, by applying the natural logarithm function. This transformation helps stabilize variance and makes the data more normally distributed, which can improve model performance.

```
df['Property_Sale_Price_natural_log'] = np.log(df['Property_Sale_Price'])
```

Conclusion
By applying a log transformation to the *Property_Sale_Price* column, we improved the data distribution, making it more suitable for regression-based predictive modeling.

Calculating Garage Age

Objective
To create a more meaningful feature for the model, we calculated the age of the garage by deriving it from the year it was built.

Methodology
Using the *GarageYrBlt* column (the year the garage was built), we computed a new feature called **GarageAge**. This feature represents the age of the garage in 2024 by subtracting the garage's construction year from the current year.

```
df['GarageAge'] = 2024 - df['GarageYrBlt']
```

Conclusion
By creating the *GarageAge* feature, we provided a more interpretable and relevant variable for modeling the influence of garage age on house prices, enhancing the dataset's predictive capabilities.

Dropping the Garage Year Built Column

Objective
After calculating the garage age, we removed the redundant *GarageYrBlt* column to simplify the dataset.

Methodology
The *GarageYrBlt* column was used to calculate the new feature *GarageAge*, which provides a more relevant and interpretable variable for modeling. Since *GarageYrBlt* is now redundant, it was dropped from the dataset to reduce complexity and avoid duplication.

```
df = df.drop(['GarageYrBlt'], axis=1)
```

Conclusion
By dropping *GarageYrBlt*, we streamlined the dataset and retained only the more meaningful *GarageAge* feature, which better contributes to the model.

One-Hot Encoding of Categorical Variables

Objective
To convert categorical features into a format suitable for machine learning algorithms, we applied one-hot encoding to the dataset.

Methodology
Categorical variables cannot be directly used in most machine learning models. To address

this, we applied **one-hot encoding**, which converts each category into a separate binary feature (0 or 1) for each unique category in the original column. This was achieved using the `pd.get_dummies()` function, ensuring that all categorical variables were transformed into numerical representations.

```
df = pd.get_dummies(df, dtype='int')
```

Conclusion
By applying one-hot encoding, we transformed all categorical features into numerical format, making the dataset ready for model training without losing any valuable categorical information.

Exporting the Cleaned Dataset

Objective
To preserve the cleaned and preprocessed dataset for future use, we exported it to a CSV file.

Methodology
After completing the data cleaning and preprocessing steps, the final dataset was saved to a CSV file named **cleaned_house_data.csv** using the `to_csv()` function. This ensures the processed data can be easily reloaded for analysis or model training.

```
df.to_csv('cleaned_house_data.csv')
```

Conclusion
By exporting the dataset, we ensured that the cleaned data is stored in a reusable format, ready for further analysis and modeling tasks.