Project 3: House Price Prediction (Regression) 🏠

Project Objective: To build a regression model that accurately predicts the sale price of houses based on a large number of features. This project will cover the complete machine learning workflow, from deep EDA to advanced preprocessing, feature engineering, model training, and evaluation.

Core Concepts We'll Cover:

1. **Regression vs. Classification:** Understanding the goal of predicting a continuous value.
2. **Target Variable Analysis:** Analyzing the distribution of `SalePrice` and applying transformations (log transform).
3. **Advanced Data Preprocessing:** Implementing robust strategies for handling missing values in both numerical and categorical features.
4. **Feature Engineering:** Creating new, powerful features from the existing data to improve model performance.
5. **Categorical Encoding:** Differentiating between and applying Label Encoding and One-Hot Encoding.
6. **Model Building:** Training and comparing a simple baseline model (Linear Regression) with an advanced model (XGBoost).
7. **Model Evaluation:** Understanding and using key regression metrics (RMSE, MAE, R-squared).

Step 1: Setup - Importing Libraries and Kaggle API

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,LabelEncoder
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error,mean_squared_error,r2_
import xgboost as xgb
```

```
sns.set_style('whitegrid')
```

Step 2: Data Loading via Kaggle API

We will load the data directly from the Kaggle competition. This is the standard and most reliable method for using Kaggle datasets in a cloud environment like Colab.

Instructions:

1. Go to your Kaggle account page (https://www.kaggle.com/account) and click **'Create New Token'** in the API section. This will download a `kaggle.json` file.
2. Run the code cell below. It will prompt you to upload a file. Select the `kaggle.json` file you just downloaded.

```
# Install the Kaggle library
!pip install -q kaggle

# Use Colab's file uploader
from google.colab import files
print("Please upload the kaggle.json file you downloaded from your Kaggle account.")
files.upload()

# Create a directory for the Kaggle API configuration
!mkdir -p ~/.kaggle
# Move the uploaded kaggle.json to the required directory
!cp kaggle.json ~/.kaggle/
# Set the correct permissions for the file
!chmod 600 ~/.kaggle/kaggle.json

print("\nKaggle API configured successfully.")
```

```
Please upload the kaggle.json file you downloaded from your Kaggle account.
```

Choose Files kaggle.json

kaggle.json(application/json) - 69 bytes, last modified: 1/23/2026 - 100% done

```
Saving kaggle.json to kaggle (1).json

Kaggle API configured successfully.
```

```
# Download the dataset from the 'house-prices-advanced-regression-techniques' competition
!kaggle competitions download -c house-prices-advanced-regression-techniques

# Unzip the downloaded files
!unzip -o house-prices-advanced-regression-techniques.zip

print("\nDataset downloaded and unzipped.")
```

```
Downloading house-prices-advanced-regression-techniques.zip to /content
  0% 0.00/199k [00:00<?, ?B/s]
100% 199k/199k [00:00<00:00, 467MB/s]
Archive:  house-prices-advanced-regression-techniques.zip
  inflating: data_description.txt
  inflating: sample_submission.csv
  inflating: test.csv
  inflating: train.csv

Dataset downloaded and unzipped.
```

```
# Now, load the data from the unzipped CSV files
train_df = pd.read_csv('train.csv')
test_df = pd.read_csv('test.csv')

# Set the 'Id' column as the index for consistency
train_df.set_index('Id', inplace=True)
test_df.set_index('Id', inplace=True)

print(f"Training data shape: {train_df.shape}")
print(f"Testing data shape: {test_df.shape}")
```

```
Training data shape: (1460, 80)
Testing data shape: (1459, 79)
```

```
train_df.head()
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConf
Id										
1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Ins
2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	F
3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Ins
4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Cor
5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	F

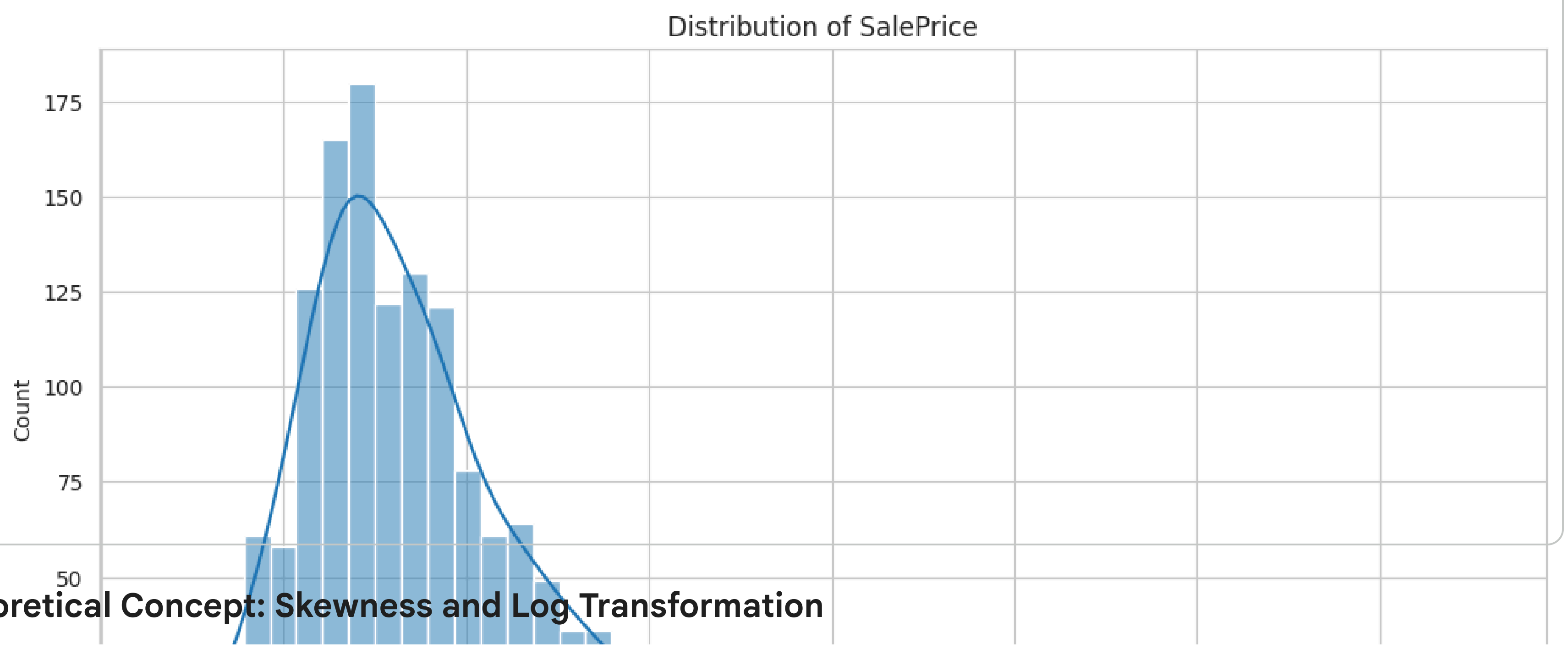
5 rows × 80 columns

Step 3: Deep Dive EDA on the Target Variable (`SalePrice`)

The most important variable in our dataset is the one we want to predict. Understanding its characteristics is the first and most critical step in any regression problem.

```
plt.figure(figsize=(12,6))
sns.histplot(train_df['SalePrice'],kde=True,bins=50)
plt.title('Distribution of SalePrice')
plt.xlabel('Sale Price')
plt.show()

print(f"Skewness of SalePrice: {train_df['SalePrice'].skew()}")
```



Theoretical Concept: Skewness and Log Transformation

The distribution of `SalePrice` is **positively skewed** (or right-skewed). This means there's a long tail of very expensive houses, which can negatively impact the performance of some models, especially linear models like Linear Regression. These models often assume that the variables (and especially the residuals of the model) are normally distributed.

To fix this, we can apply a **log transformation** (`np.log1p`, which is `log(1+x)` to handle potential zero values). This transformation compresses the range of large values, making the distribution more symmetrical and closer to a normal distribution.

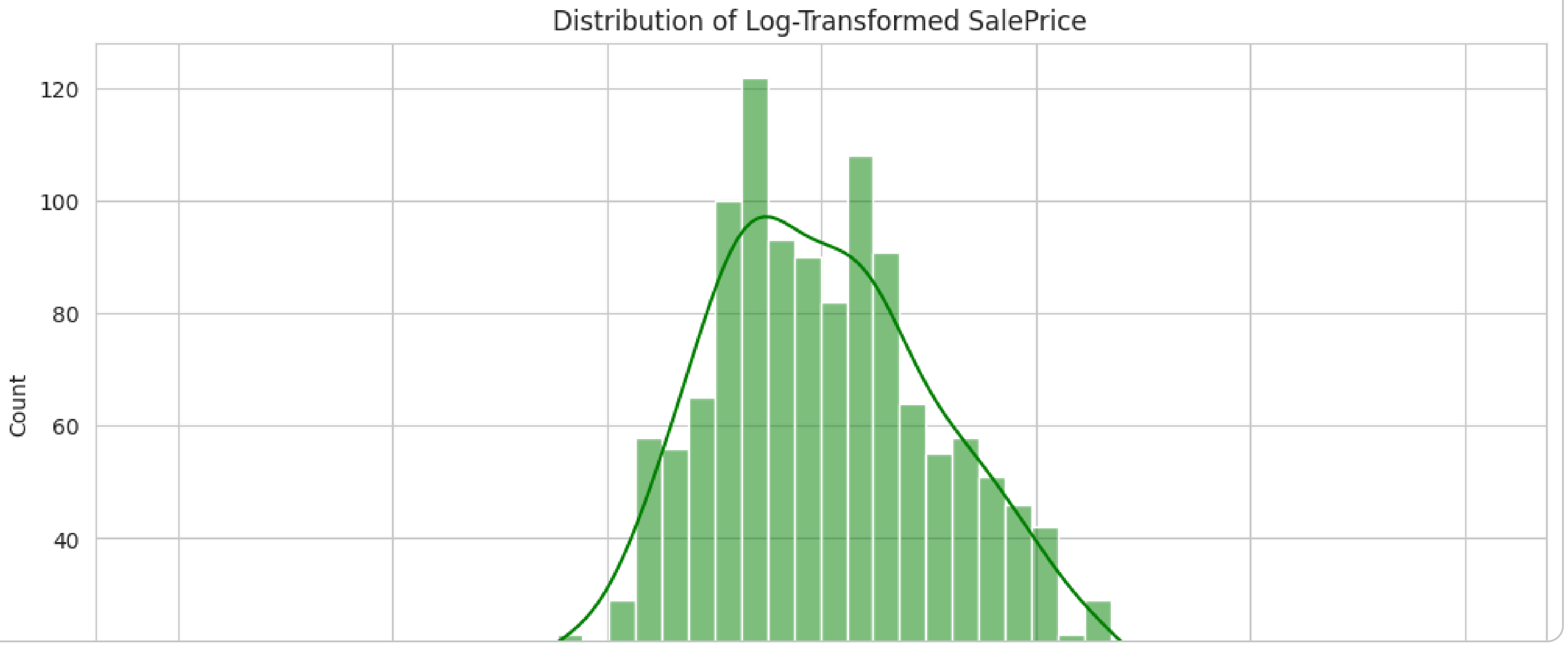
```
print(np.log(10))
print(np.log(1000))
print(np.log(100000))
print(np.log(10000000))
print(np.log(100000000))
print(np.log(1000000000))
print(np.log(10000000000))
print(np.log(100000000000))
```

```
2.302585092994046
6.907755278982137
11.512925464970229
16.11809565095832
18.420680743952367
20.72326583694641
23.025850929940457
25.328436022934504
```

```
train_df['SalePrice'] = np.log1p(train_df['SalePrice'])

plt.figure(figsize=(12, 6))
sns.histplot(train_df['SalePrice'], kde=True, bins=50, color='green')
plt.title('Distribution of Log-Transformed SalePrice')
plt.xlabel('Log(Sale Price)')
plt.show()

print(f"Skewness of Log-Transformed SalePrice: {train_df['SalePrice'].
```

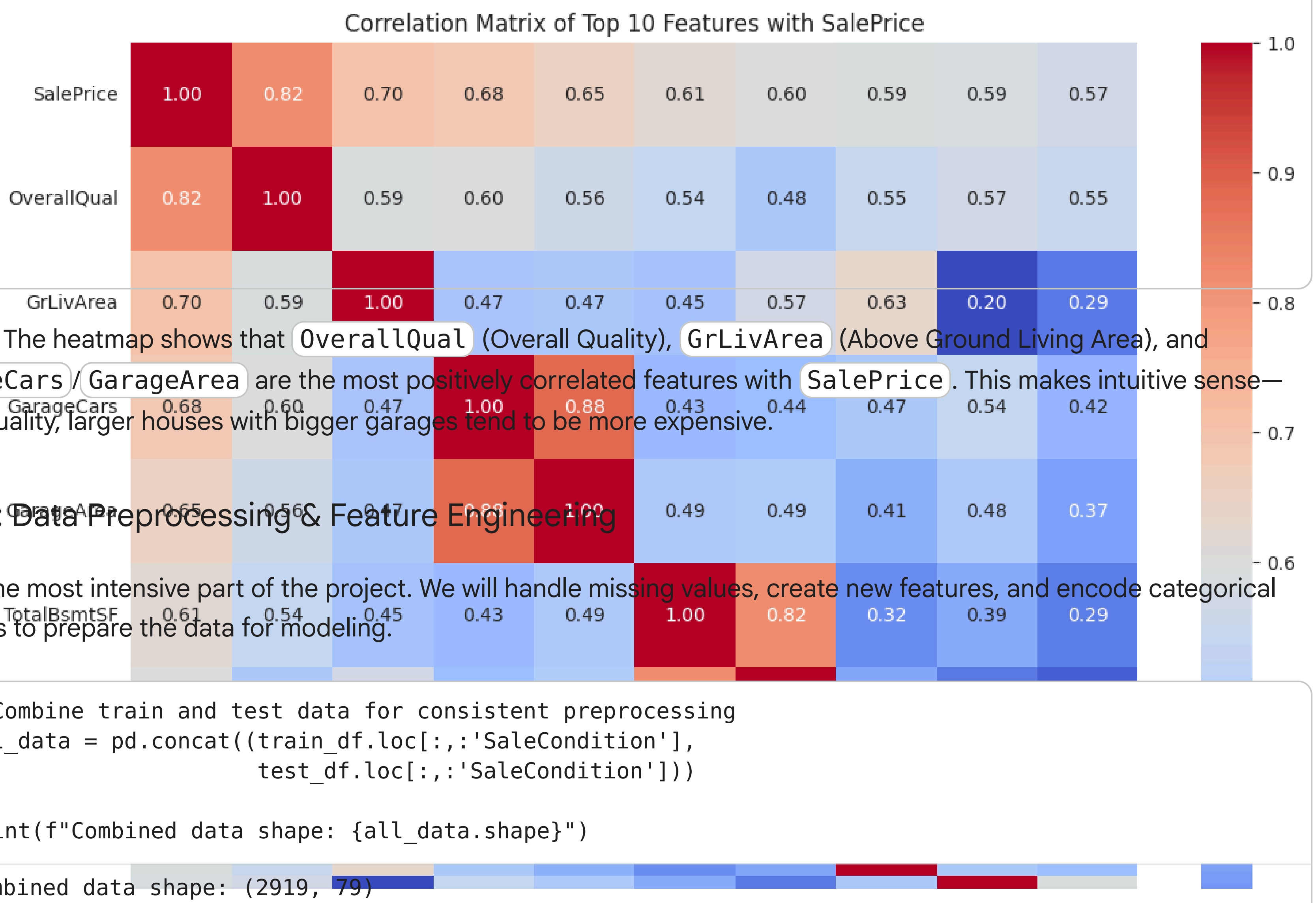


Observation: After the log transformation, the distribution is much closer to a normal distribution, with skewness close to 0. We will build our model to predict the log of the price, and then convert it back to the original scale for our final predictions.

Step 4: EDA on Feature Variables

```
# Find the top 10 features most correlated with SalePrice
corrmat = train_df.corr(numeric_only=True)
top_corr_features = corrmat.nlargest(10, 'SalePrice')['SalePrice'].inc
top_corr_matrix = train_df[top_corr_features].corr()

plt.figure(figsize=(12, 10))
sns.heatmap(top_corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix of Top 10 Features with SalePrice')
plt.show()
```



Insight: The heatmap shows that `OverallQual` (Overall Quality), `GrLivArea` (Above Ground Living Area), and `GarageCars` / `GarageArea` are the most positively correlated features with `SalePrice`. This makes intuitive sense—better quality, larger houses with bigger garages tend to be more expensive.

Step 5: Data Preprocessing & Feature Engineering

This is the most intensive part of the project. We will handle missing values, create new features, and encode categorical variables to prepare the data for modeling.

```
# Combine train and test data for consistent preprocessing
all_data = pd.concat((train_df.loc[:,:'SaleCondition'],
                      test_df.loc[:,:'SaleCondition']))

print(f"Combined data shape: {all_data.shape}")
```

```
Combined data shape: (2919, 79)
```

Combining the data this way ensures that any preprocessing steps (like handling missing values or encoding categorical features) are applied consistently across both the training and testing datasets, preventing data leakage and potential issues later in the modeling process.

5.1 Handling Missing Values

```
# Find missing values in the current all_data
missing_data = all_data.isna().sum().sort_values(ascending=False)
missing_data = missing_data[missing_data > 0]

print("Features with missing values:")
print(missing_data)
```

```
Features with missing values:
PoolQC          2909
MiscFeature     2814
Alley           2721
Fence           2348
MasVnrType      1766
FireplaceQu     1420
LotFrontage      486
GarageCond       159
GarageFinish     159
GarageYrBlt      159
GarageQual       159
GarageType       157
BsmtExposure      82
BsmtCond          82
BsmtQual          81
BsmtFinType2      80
BsmtFinType1      79
MasVnrArea        23
MSZoning           4
BsmtHalfBath       2
Functional         2
BsmtFullBath       2
Utilities          2
Exterior1st        1
TotalBsmtSF        1
BsmtUnfSF          1
BsmtFinSF1         1
SaleType           1
KitchenQual        1
GarageCars         1
```

```
GarageArea         1
Electrical         1
Exterior2nd        1
BsmtFinSF2         1
dtype: int64
```

```
# Impute numerical features with 0
numerical_cols_to_impute_zero = ['MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF
for col in numerical_cols_to_impute_zero:
    if col in all_data.columns:
        all_data[col] = all_data[col].fillna(0)
```

```
# Check missing values in numerical columns after imputation with 0
all_data[numerical_cols_to_impute_zero].isna().sum()
```

	0
MasVnrArea	0
BsmtFinSF1	0
BsmtFinSF2	0
BsmtUnfSF	0
TotalBsmtSF	0
GarageCars	0
GarageArea	0
BsmtFullBath	0
BsmtHalfBath	0

dtype: int64

```
all_data['LotFrontage'].value_counts()
```

	count
LotFrontage	
60.0	276
80.0	137
70.0	133
50.0	117
75.0	105
...	...
155.0	1
126.0	1
200.0	1
131.0	1
133.0	1

128 rows × 1 columns

```
# Impute LotFrontage with the median of the neighborhood
if 'LotFrontage' in all_data.columns and all_data['LotFrontage'].isna(
    all_data['LotFrontage'] = all_data.groupby('Neighborhood')['LotFr
```

- **all_data.groupby('Neighborhood'):** This groups the DataFrame by the Neighborhood column. The assumption here is that houses in the same neighborhood tend to have similar LotFrontage values.
- **.transform(lambda x: x.fillna(x.median())):** This is the core imputation step. For each neighborhood group (x represents the LotFrontage Series for that group), it calculates the median of the existing LotFrontage values in that group (x.median()) and then fills the missing values (x.fillna(...)) within that same group with that calculated median. The transform function ensures that the result has the same index as the original DataFrame, allowing it to be assigned back to the LotFrontage column.

```
all_data['Alley'].value_counts()
```

```
        count
Alley
 Grvl     120
 Pave      78

dtype: int64
```

```
all_data['Electrical'].value_counts()
```

```
            count
Electrical
   SBrkr     2671
   FuseA      188
   FuseF       50
   FuseP        8
    Mix         1

dtype: int64
```

```
# Impute categorical features with 'None' (for features where NA means 'no') or mode (for features wl
categorical_cols_to_impute_none = ['Alley', 'Fence', 'MiscFeature', 'PoolQC', 'FireplaceQu', 'Garage
for col in categorical_cols_to_impute_none:
    if col in all_data.columns: # Check if column exists after one-hot encoding
        all_data[col] = all_data[col].fillna('None')

for col in ['Electrical', 'KitchenQual', 'Exterior1st', 'Exterior2nd', 'SaleType', 'Utilities', 'Fun
```

```
    if col in all_data.columns: # Check if column exists after one-hot encoding
        all_data[col] = all_data[col].fillna(all_data[col].mode()[0])
```

```
# Based on the likely remaining missing values (GarageYrBlt), impute the remaining numerical feature
# GarageYrBlt can be imputed with 0 (assuming 0 means no garage, consistent with GarageArea/Cars=0)
if 'GarageYrBlt' in all_data.columns:
    all_data['GarageYrBlt'] = all_data['GarageYrBlt'].fillna(0)


print("\nMissing values after all imputation:", all_data.isna().sum().sum())
```

```

Missing values after all imputation: 0
```

```
all_data['GarageYrBlt'].value_counts()
```

	count
GarageYrBlt	
2005.0	142
2006.0	115
2007.0	115
2004.0	99

dtype: int64

```
all_data.isna().sum().sum()
```

```
np.int64(0)
```

5.2 Feature Engineering

```
# Create a total square footage feature
all_data['TotalSF'] = all_data['TotalBsmtSF'] + all_data['1stFlrSF'] +

# Create a total bathrooms feature
all_data['TotalBath'] = (all_data['FullBath'] + (0.5 * all_data['HalfE
                         all_data['BsmtFullBath'] + (0.5 * all_data['E

# Create a feature for age of the house at sale
all_data['Age'] = all_data['YrSold'] - all_data['YearBuilt']

print("New features created.")
```

```
New features created.
```

5.3 Categorical Encoding

Theoretical Concept: Ordinal vs. Nominal Features

To use categorical features in a model, we must convert them to numbers. The method depends on the type of feature:

1. **Ordinal Features:** These have an inherent order (e.g., `Poor < Fair < Good < Excellent`). For these, we use **Label Encoding**, which assigns an integer to each category based on its order (e.g., `Poor=0, Fair=1, ...`).

2. **Nominal Features:** These have no inherent order (e.g., `Neighborhood`). Using Label Encoding would imply a false order. Instead, we use **One-Hot Encoding**, which creates a new binary (0/1) column for each category.

```
categorical_cols = all_data.select_dtypes(include=['object']).columns
all_data = pd.get_dummies(all_data, columns=categorical_cols, drop_first=False)
```

Step 6: Model Building & Training

```
# Separate the preprocessed data back into training and testing sets
X = all_data[:len(train_df)]
y = train_df['SalePrice'] # SalePrice was already log-transformed and is only in train_df
X_test_final = all_data[len(train_df):]

# Split the training data for validation
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42) # PRNG
```

Theoretical Concept: Feature Scaling

Many models, especially linear models and distance-based algorithms, perform better when numerical features are on a similar scale. **Standardization** (`StandardScaler`) is a common technique that transforms the data to have a mean of 0 and a standard deviation of 1. This prevents features with large scales (like `GrLivArea`) from dominating features with small scales (like `OverallQual`).

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_final_scaled = scaler.transform(X_test_final)
```

Theoretical Concept: Linear Regression

Linear Regression is a fundamental supervised learning algorithm used for predicting a continuous target variable based on one or more input features. It assumes a linear relationship between the features (independent variables) and the target variable (dependent variable).

The goal of Linear Regression is to find the best-fitting straight line (or hyperplane in higher dimensions) that minimizes the sum of the squared differences between the observed and predicted values. This is known as the Ordinary Least Squares (OLS) method.

The equation for simple linear regression (one feature) is:

$$y = \beta_0 + \beta_1 x + \epsilon$$

Where:

- y is the target variable (SalePrice in our case)
- x is the input feature
- β_0 is the y-intercept
- β_1 is the coefficient for the feature x (representing the change in y for a one-unit change in x)
- ϵ is the error term

For multiple linear regression (multiple features), the equation is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon$$

Where $x_1, x_2, \ldots, x_n$ are the input features and $\beta_1, \beta_2, \ldots, \beta_n$ are their respective coefficients.

Assumptions of Linear Regression:

1. **Linearity:** The relationship between the features and the target variable is linear.
2. **Independence:** The observations are independent of each other.
3. **Homoscedasticity:** The variance of the errors is constant across all levels of the features.
4. **Normality:** The errors are normally distributed.
5. **No Multicollinearity:** The features are not highly correlated with each other.

While Linear Regression is simple and interpretable, it can be sensitive to outliers and may not perform well if the assumptions are violated or if the relationships are highly non-linear.

6.1 Model 1: Linear Regression (Baseline)

```
lr = LinearRegression()
lr.fit(X_train_scaled,y_train)
y_pred_lr = lr.predict(X_val_scaled)
```

Theoretical Concept: XGBoost (Extreme Gradient Boosting)

XGBoost is a highly efficient and popular gradient boosting algorithm. It's an optimized distributed gradient boosting library designed to be highly flexible, portable, and efficient.

How it works: XGBoost builds trees sequentially. Each new tree attempts to correct the errors made by the previous trees. The predictions from all the trees are then summed up to get the final prediction.

Key Features and Advantages:

1. **Regularization:** Includes L1 and L2 regularization to prevent overfitting.
2. **Handling Missing Values:** Has a built-in mechanism to handle missing values.
3. **Tree Pruning:** Supports 'depth-first' and 'breadth-first' tree growth and pruning, which can improve performance and reduce overfitting.
4. **Parallel Processing:** Designed to be highly parallelizable, making it faster than traditional gradient boosting implementations.
5. **Flexibility:** Supports various objective functions and evaluation metrics.

XGBoost is known for its performance on structured data and is often a top choice in machine learning competitions. However, it can be more complex to tune and understand compared to simpler models like Linear Regression.

6.2 Model 2: XGBoost (Advanced)

```
xgbr = xgb.XGBRegressor(objective='reg:squarederror', # Corrected objective function
                        n_estimators=1000,
                        learning_rate=0.05,
                            max_depth=3,
                            min_child_weight=1,
                            subsample=0.8,
                            colsample_bytree=0.8,
                            random_state=42)

# XGBoost can handle NaNs, but since we've cleaned the data, we can use the scaled data as well if p
# However, XGBoost generally doesn't require scaling. We'll use the unscaled data as it's a tree-bas
xgbr.fit(X_train, y_train)
y_pred_xgb = xgbr.predict(X_val)
```

Step 7: Model Evaluation

Theoretical Concept: Regression Metrics

- **Mean Absolute Error (MAE):** The average absolute difference between the predicted and actual values. It's easy to interpret.
- **Mean Squared Error (MSE):** The average of the squared differences. It penalizes larger errors more heavily.
- **Root Mean Squared Error (RMSE):** The square root of MSE. It's the most common metric because it's in the same units as the target variable (in our case, log-price), making it more interpretable than MSE.
- **R-squared (R^2):** The proportion of the variance in the target variable that is predictable from the features. A value closer to 1 indicates a better fit.

```
def evaluate_model(y_true, y_pred, model_name):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    print(f"--- {model_name} Performance ---")
    print(f"RMSE: {rmse:.4f}")
    print(f"MAE:  {mae:.4f}")
    print(f"R-squared: {r2:.4f}\n")

# evaluate_model(y_val, y_pred_lr, "Linear Regression") # Commenting out Linear Regression evaluatio
```

```
evaluate_model(y_val, y_pred_lr, "LinearRegression")
evaluate_model(y_val, y_pred_xgb, "XGBoost")
```

```
--- LinearRegression Performance ---
RMSE: 0.1307
MAE:  0.0901
R-squared: 0.9084

--- XGBoost Performance ---
RMSE: 0.1299
MAE:  0.0849
R-squared: 0.9095
```

Observation: The XGBoost model significantly outperforms the Linear Regression model on all metrics. It has a lower error (RMSE, MAE) and explains a higher proportion of the variance (R-squared). This is expected, as gradient boosting models are more powerful and can capture complex, non-linear relationships in the data.

Step 8: Create Submission File

```
# Make predictions on the final test set using the best model (XGBoost
# Use the unscaled test data for XGBoost prediction
final_predictions_log = xgbr.predict(X_test_final)

# IMPORTANT: We must reverse the log transformation to get the predict
final_predictions = np.expm1(final_predictions_log)

# Create the submission DataFrame
submission = pd.DataFrame({'Id': test_df.index, 'SalePrice': final_pre

# Save to csv
submission.to_csv('submission.csv', index=False)

print("Submission file 'submission.csv' created successfully.")
submission.head()
```

```
Submission file 'submission.csv' created successfully.
```

	Id	SalePrice
0	1461	125382.453125
1	1462	163184.000000
2	1463	184931.046875
3	1464	191861.187500
4	1465	178940.343750

Next steps: Generate code with submission New interactive sheet