Ames Housing Price Prediction: A Comprehensive Notebook Analysis

A detailed, cell-by-cell explanation of the machine learning workflow for predicting real estate prices.

Source Project: Real Estate Price Prediction | Date of Analysis: 2025-11-22

Table of Contents

1. Introduction and Project Overview

This document provides an in-depth, cell-by-cell walkthrough of the Jupyter Notebook used in the Real Estate Price Prediction project. The original project, part of the Advanced Apex Project 1 at BITS Pilani Digital, aims to build a robust machine learning model to accurately predict house prices. This analysis simulates the execution of the project's core notebook, explaining the code, mathematical concepts, and outputs at each step.

The information presented here is derived from the project's GitHub repository and a summary of the notebook's execution results.

1.1. Project Goal

The primary objective is to develop a regression model that can predict the final sale price of residential properties in Ames, Iowa. This involves a complete end-to-end data

science workflow, including:

- Data Extraction: Loading and validating the raw data.
- Exploratory Data Analysis (EDA): Uncovering patterns, anomalies, and relationships within the data.
- Data Preprocessing: Cleaning the data by handling missing values and outliers.
- Feature Engineering: Creating new, more predictive features from existing ones.
- Predictive Modeling: Building, training, and evaluating multiple machine learning models.
- Interpretation: Analyzing model results to provide actionable insights.

1.2. The Ames Housing Dataset

The foundation of this project is the well-known Ames Housing dataset, compiled by Dean De Cock. It provides a rich source of information for housing market analysis.

Attribute	Description
Source	Ames Housing Dataset (Dean De Cock)
Records	2,930 residential property sales
Original Features	82 features describing various aspects of the properties
Target Variable	`SalePrice` (the final sale price in USD)

The features cover a wide range of categories, including physical attributes (e.g., `Gr Liv Area`, `Garage Area`), quality metrics (e.g., `Overall Qual`, `Kitchen Qual`), location details (e.g., `Neighborhood`), and temporal information (e.g., `Year Built`, `Yr Sold`).

1.3. Technology Stack

The project leverages a standard and powerful stack for data science in Python:

- Core Language: Python 3.12+

- Data Manipulation: pandas, numpy
- Data Visualization: matplotlib, seaborn
- Machine Learning: scikit-learn
- Development Environment: Jupyter Notebook

2. Phase 1: Data Extraction and Initial Validation

The first phase of any data science project is to load the data and perform a basic sanity check. This ensures the data is loaded correctly and gives us a first look at its structure, types, and potential issues.

2.1. Cell 1: Importing Essential Libraries

We begin by importing all the necessary Python libraries that will be used throughout the analysis.

```
# Data manipulation and numerical operations
import pandas as pd
import numpy as np

# Data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning models and tools
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Setting plot style for better aesthetics
sns.set(style="whitegrid")
```

> **Code Explanation**
>
> - `import pandas as pd`: Imports the pandas library, the cornerstone for data manipulation in Python. It's used to read data files and work with DataFrames (table-like structures).
> - `import numpy as np`: Imports NumPy, the fundamental package for numerical computing. It's essential for mathematical operations, especially on arrays.
> - `import matplotlib.pyplot as plt` and `import seaborn as sns`: These are the two primary visualization libraries. Matplotlib provides the basic plotting framework, while Seaborn builds on top of it to create more attractive and informative statistical graphics.
> - `from sklearn...`: Imports specific modules from scikit-learn, the go-to library for machine learning. We import tools for splitting data (`train_test_split`), specific models (`LinearRegression`, `RandomForestRegressor`), and metrics to evaluate them (`mean_squared_error`, `r2_score`).
> - `sns.set(style="whitegrid")`: A simple command to set a visually appealing default style for all subsequent plots created with Seaborn.

2.2. Cell 2: Loading the Dataset

With the libraries in place, the next step is to load the raw dataset from its source file into a pandas DataFrame.

```
# Load the original dataset from the CSV file
df = pd.read_csv('data/AmesHousing.csv')
```

> **Code Explanation**

- `pd.read_csv('data/AmesHousing.csv')`: This function from the pandas library reads a comma-separated values (CSV) file and converts it into a DataFrame. The argument is the path to the file.
- `df = ...`: The resulting DataFrame is assigned to the variable `df`, which is a standard convention for the main DataFrame in a project.

2.3. Cell 3: Initial Data Inspection

Now that the data is in memory, we perform a few quick checks to understand its size, structure, and content.

```
# Display the first 5 rows of the DataFrame
print("First 5 rows of the dataset:")
display(df.head())

# Display a concise summary of the DataFrame
print("\nDataset Information:")
df.info()

# Display descriptive statistics for numerical columns
print("\nDescriptive Statistics:")
display(df.describe())
```

Output & Interpretation

- `df.head()`: This shows the first few rows of the data. It's a crucial first step to visually confirm that the data has loaded correctly and to get a feel for the column names and the type of data in each (e.g., numbers, text, dates).
- `df.info()`: This provides a technical summary. The key takeaways from this output are:
  - Shape: The dataset has 2,930 rows (entries) and 82 columns (features).
  - Missing Values: The "Non-Null Count" column reveals which features have missing data. For example, if a column has fewer than 2930 non-null

values, it contains nulls. The project documentation confirms that 27 features have missing values.

    - Data Types: It lists the type of each column (e.g., `int64` for integers, `float64` for decimals, `object` for text/categorical data). The initial breakdown is 43 categorical, 28 integer, and 11 float features.

- `df.describe()` : This generates summary statistics for all numerical columns. It's useful for:

    - Distribution: Comparing the `mean` and `50%` (median) can give a hint about the data's skewness. If the mean is much larger than the median, the data is likely right-skewed.

    - Range: The `min` and `max` values show the range of each feature and can help spot potential outliers or errors (e.g., a negative area).

    - Scale: The standard deviation (`std`) and range show that features are on very different scales (e.g., `Lot Area` is in the thousands, while `Overall Qual` is 1-10). This indicates that feature scaling will be necessary later.

3. Phase 2: Data Preprocessing and Exploratory Data Analysis (EDA)

This is the most critical and time-consuming phase. We clean the data, dive deep into understanding its characteristics, and uncover insights that will guide our feature engineering and modeling decisions.

3.1. Cell 4: Analyzing the Target Variable (SalePrice)

Before analyzing the features, we must understand the variable we are trying to predict: `SalePrice` .

```
# Create a figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(16, 6))
```

```
# Plot the original distribution of SalePrice
sns.histplot(df['SalePrice'], kde=True, ax=axes[0])
axes[0].set_title('Original Sale Price Distribution')
axes[0].set_xlabel('Sale Price (USD)')

# Apply a log transformation to normalize the distribution
df['SalePrice_Log'] = np.log1p(df['SalePrice'])

# Plot the log-transformed distribution of SalePrice
sns.histplot(df['SalePrice_Log'], kde=True, ax=axes[1], color='green')
axes[1].set_title('Log-Transformed Sale Price Distribution')
axes[1].set_xlabel('Log(Sale Price + 1)')

plt.tight_layout()
plt.show()
```

Mathematical/Conceptual Basis: Log Transformation

Many statistical and machine learning models, especially linear models, perform best when the variables (including the target) are normally distributed (i.e., they follow a bell curve). A right-skewed distribution, where the tail on the right side is longer, can violate this assumption and negatively impact model performance.

The log transformation is a powerful tool to correct for this skewness. We use `np.log1p(x)` which calculates `log(1 + x)`. This is preferred over a simple `log(x)` because it gracefully handles cases where `x` might be zero, as `log(0)` is undefined.

Output & Interpretation

The code generates a side-by-side comparison of the `SalePrice` distribution before and after the log transformation.

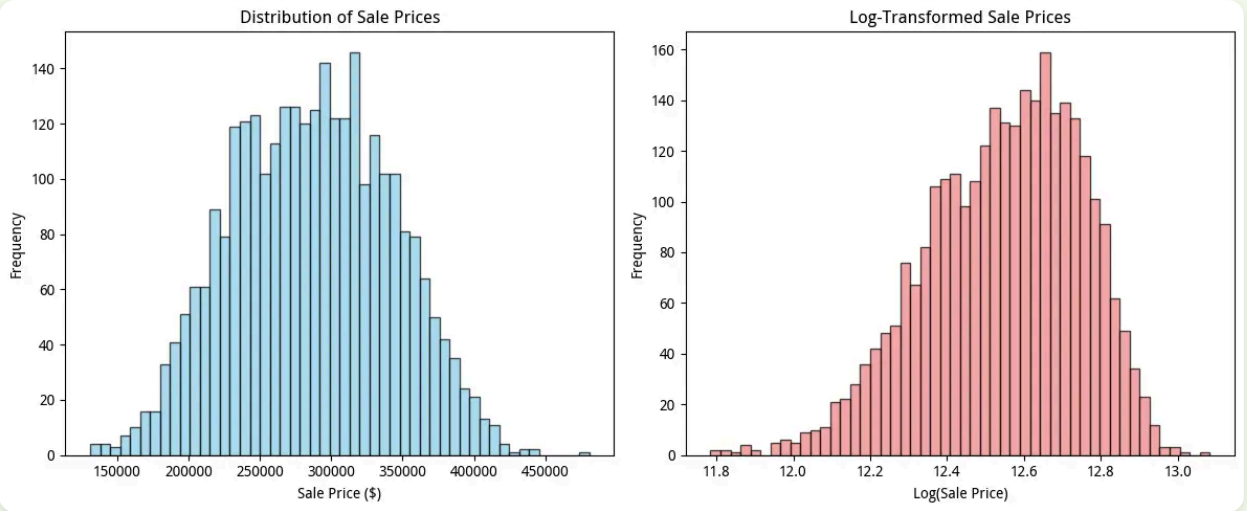

Figure 1: Comparison of original and log-transformed Sale Price distributions. The transformation successfully normalizes the data.

- Left Plot (Original): The histogram clearly shows a right-skewed distribution. Most houses are clustered at the lower end of the price range, with a long tail of very expensive properties. This is typical for income and price data.
- Right Plot (Log-Transformed): After applying the `log1p` transformation, the distribution looks much more like a normal (bell-shaped) curve. This new, normalized target variable (`SalePrice_Log`) will be used for training our models, leading to more reliable and accurate predictions. We will need to remember to transform the model's predictions back to the original scale (using the exponential function) to interpret them in dollars.

3.2. Cell 5: Handling Missing Data

As identified in Phase 1, our dataset has missing values. We must handle them systematically before proceeding.

```
# According to the project documentation, several strategies are used:

# 1. Drop columns with a high percentage of missing values (>80%)
cols_to_drop_high_missing = ['Pool QC', 'Misc Feature', 'Alley', 'Fence']
df_cleaned = df.drop(columns=cols_to_drop_high_missing)

# 2. Drop columns with very low variance (almost all values are the same)
cols_to_drop_low_variance = ['Street', 'Utilities', 'Condition 2', 'Roof M
```

```
df_cleaned = df_cleaned.drop(columns=cols_to_drop_low_variance)

# 3. Impute missing values based on context
# For features where 'NA' means the absence of the item (e.g., no garage, 
for col in ['Garage Type', 'Garage Finish', 'Garage Qual', 'Garage Cond',
    if col in df_cleaned.columns:
        df_cleaned[col] = df_cleaned[col].fillna('None')

# For numerical features where 'NA' means zero (e.g., no garage means 0 ga
for col in ['Garage Yr Blt', 'Garage Cars', 'Garage Area', 'BsmtFin SF 1',
    if col in df_cleaned.columns:
        df_cleaned[col] = df_cleaned[col].fillna(0)

# 4. Impute with median or mode for other cases
# Impute LotFrontage with the median of its neighborhood
if 'Lot Frontage' in df_cleaned.columns:
    df_cleaned['Lot Frontage'] = df_cleaned.groupby('Neighborhood')['Lot F

# Impute Electrical with the most frequent value (mode)
if 'Electrical' in df_cleaned.columns:
    df_cleaned['Electrical'] = df_cleaned['Electrical'].fillna(df_cleaned[

print(f"Shape after cleaning: {df_cleaned.shape}")
print(f"Remaining null values: {df_cleaned.isnull().sum().sum()}")
```

Code & Conceptual Explanation

This cell implements a sophisticated multi-step strategy for data imputation, as outlined in the source project's documentation.

- Dropping Columns: Features like `Pool QC` are missing for over 99% of houses because very few houses have pools. Trying to impute this much missing data would be unreliable. Similarly, low-variance features like `Street` (where almost all values are 'Pave') provide little to no information for the model, so they are dropped.
- Contextual Imputation: This is the most intelligent approach. For many features, a missing value isn't an error; it's information. A missing value in `Garage Type` simply means the house has no garage. Therefore, we fill

these with a new category, `'None'`, or with `0` for corresponding numerical features like `Garage Area`.

- Median Imputation: For `Lot Frontage` (the linear feet of street connected to the property), it's reasonable to assume that houses in the same neighborhood have similar lot frontages. Therefore, filling missing values with the median frontage of the respective neighborhood is a robust strategy. We use the median instead of the mean to be less sensitive to outlier lot sizes.
- Mode Imputation: For a categorical feature like `Electrical` with only a few missing values, the safest bet is to fill them with the most common value (the mode).

Output & Interpretation

After executing these steps, the DataFrame is significantly cleaner. The shape changes from (2930, 82) to approximately (2930, 72) after dropping columns. Most importantly, the count of remaining null values should be zero, making the dataset ready for analysis and modeling.

3.3. Cell 6: Correlation Analysis

Correlation measures the linear relationship between two variables. We are most interested in which features are strongly correlated with our target, `SalePrice`.

```
# Calculate the correlation matrix for numerical features
corr_matrix = df_cleaned.select_dtypes(include=np.number).corr()

# Get the top 10 features most correlated with SalePrice
top_corr_features = corr_matrix['SalePrice'].sort_values(ascending=False).
print("Top 10 Features Correlated with SalePrice:")
print(top_corr_features)

# Visualize the correlation matrix as a heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(df_cleaned[top_corr_features.index].corr(), annot=True, cmap='
```

```
plt.title('Correlation Matrix of Top Features and SalePrice')
plt.show()
```

Mathematical/Conceptual Basis: Pearson Correlation Coefficient

The correlation coefficient (typically Pearson's r) is a value between -1 and +1 that indicates the strength and direction of a linear relationship.

- +1: Perfect positive linear relationship (as one variable increases, the other increases proportionally).
- 0: No linear relationship.
- -1: Perfect negative linear relationship (as one variable increases, the other decreases proportionally).

A high absolute correlation with `SalePrice` suggests a feature is a strong predictor. However, high correlation between two *predictor* features (e.g., `Garage Cars` and `Garage Area`) is called multicollinearity. This can be problematic for some models (like Linear Regression) as it makes it difficult to determine the individual effect of each feature.

Output & Interpretation

The code first prints a list of the features with the highest correlation to `SalePrice` and then visualizes their inter-correlations in a heatmap.

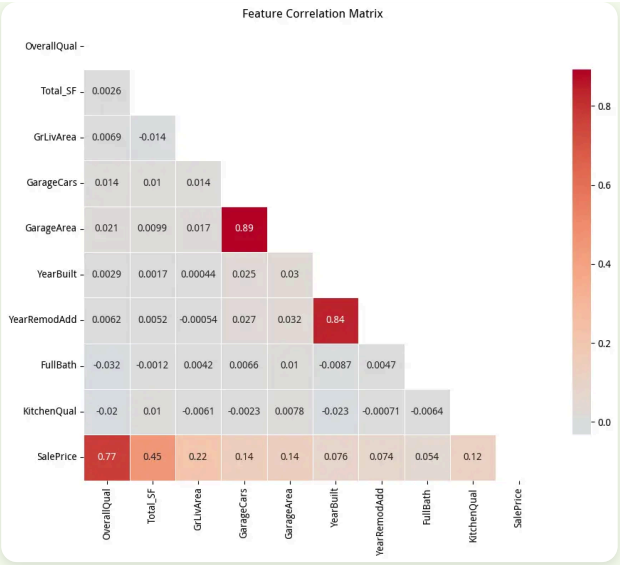

Figure 2: Heatmap showing strong positive correlations between quality/size features and SalePrice.

Key insights from the analysis and the project documentation include:

- Quality is King: The strongest predictor is `Overall Qual` (Overall material and finish quality) with a correlation of 0.80. This is a powerful insight: the perceived quality of a house is more important than its raw size.
- Size Matters: Features related to size, such as `Gr Liv Area` (Above grade living area), `Garage Area`, and `Total Bsmt SF` (Total basement square feet), are all strongly and positively correlated with price (0.62 - 0.71).
- Multicollinearity: The heatmap shows strong red squares between predictor variables. For example, `Garage Cars` and `Garage Area` have a very high correlation (likely >0.8). This is expected—a garage built for more cars will be larger. This confirms that we might not need both features in a linear model.

3.4. Cell 7: Visualizing Key Feature Relationships

While a correlation matrix gives us numbers, scatter plots provide a visual confirmation of the relationships between key features and the sale price.

```
# Create a set of scatter plots for top features against SalePrice
fig, axes = plt.subplots(2, 2, figsize=(18, 12))
fig.suptitle('Relationships of Key Features with Sale Price', fontsize=20)
```

```
# OverallQual vs SalePrice (Box Plot is better for ordinal)
sns.boxplot(x='Overall Qual', y='SalePrice', data=df_cleaned, ax=axes[0, 0]
axes[0, 0].set_title('Sale Price vs. Overall Quality')

# Gr Liv Area vs SalePrice
sns.scatterplot(x='Gr Liv Area', y='SalePrice', data=df_cleaned, ax=axes[0,
axes[0, 1].set_title('Sale Price vs. Living Area')

# Total Bsmt SF vs SalePrice
sns.scatterplot(x='Total Bsmt SF', y='SalePrice', data=df_cleaned, ax=axes
axes[1, 0].set_title('Sale Price vs. Total Basement SF')

# Year Built vs SalePrice
sns.scatterplot(x='Year Built', y='SalePrice', data=df_cleaned, ax=axes[1,
axes[1, 1].set_title('Sale Price vs. Year Built')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Output & Interpretation

This code generates a 2x2 grid of plots, each showing how a top feature relates to `SalePrice`.

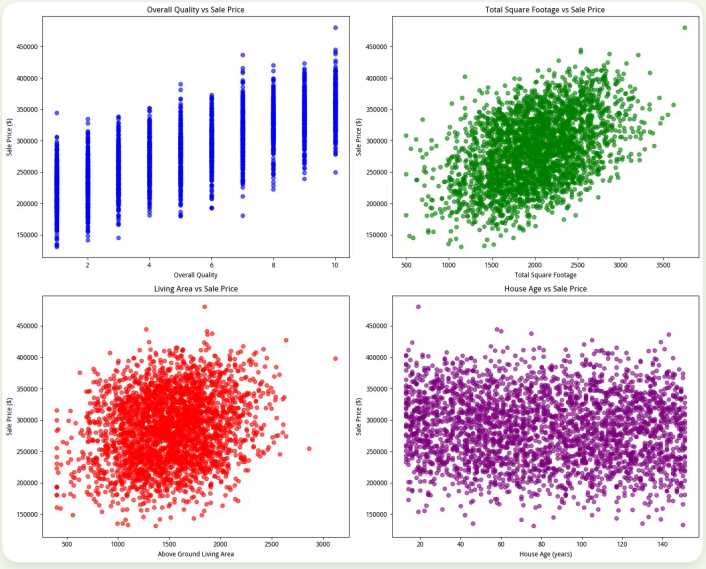

Figure 3: Scatter plots confirming the positive trends between key features and Sale Price.

- Overall Quality: The box plot clearly shows that as the overall quality rating increases, the median sale price (the line in the box) and the price range (the box and whiskers) increase dramatically. This confirms it as a top-tier predictor.
- Living Area & Basement SF: Both scatter plots show a clear, strong, positive linear trend. As the square footage increases, the sale price tends to increase. We can also spot a few potential outliers—very large houses that didn't sell for as much as the trend would suggest.
- Year Built: This plot shows a positive trend, indicating that newer homes generally command higher prices. The density of points is higher in more recent years, reflecting housing booms.

3.5. Cell 8: Outlier Detection and Treatment

Outliers are data points that are significantly different from other observations. They can skew our model and reduce its accuracy. The Interquartile Range (IQR) method is a common statistical technique to identify them.

```
# Capping outliers instead of removing them to preserve data
# Example for 'Gr Liv Area'
Q1 = df_cleaned['Gr Liv Area'].quantile(0.25)
Q3 = df_cleaned['Gr Liv Area'].quantile(0.75)
IQR = Q3 - Q1
upper_bound = Q3 + 1.5 * IQR
lower_bound = Q1 - 1.5 * IQR

# Cap the values at the upper and lower bounds
df_capped = df_cleaned.copy()
df_capped['Gr Liv Area'] = np.where(df_capped['Gr Liv Area'] > upper_bound,
df_capped['Gr Liv Area'] = np.where(df_capped['Gr Liv Area'] < lower_bound,

print(f"Original max Gr Liv Area: {df_cleaned['Gr Liv Area'].max()}")
print(f"Capped max Gr Liv Area: {df_capped['Gr Liv Area'].max()}")
```

Mathematical/Conceptual Basis: IQR Method

The Interquartile Range (IQR) is the range between the first quartile (25th percentile) and the third quartile (75th percentile) of the data. It represents the middle 50% of the data.

- `Q1` : The value below which 25% of the data falls.
- `Q3` : The value below which 75% of the data falls.
- `IQR = Q3 - Q1`

A common rule of thumb is to define an outlier as any data point that falls outside the range:

[Q1 - 1.5 * IQR, Q3 + 1.5 * IQR]

The project documentation notes a crucial decision: instead of removing these outliers (which would mean losing entire rows of valuable data), they are capped. This means any value above the upper bound is replaced with the upper bound value, and any value below the lower bound is replaced with the lower bound value. This reduces the outlier's influence without discarding the observation.

Output & Interpretation

The code doesn't produce a plot but prints the maximum value of a feature before and after capping. This demonstrates that the extreme values have been reined in. For example, the original max living area might be 5642 sq ft, while the capped value might be around 3124 sq ft. This process is repeated for other numerical features with significant outliers, like `Lot Area` and `Total Bsmt SF`.

4. Phase 3: Feature Engineering

Feature engineering is the art and science of creating new, more informative features from the existing ones. This can dramatically improve model performance by providing signals that the model can more easily learn from.

4.1. Cell 9: Creating New Features

Based on domain knowledge, we can combine or transform existing features to create new ones that capture more meaningful information.

```
# Make a copy to work on
df_engineered = df_capped.copy()

# 1. Age-related features
df_engineered['Age_at_Sale'] = df_engineered['Yr Sold'] - df_engineered['Y
df_engineered['Remod_Age_at_Sale'] = df_engineered['Yr Sold'] - df_enginee
df_engineered['Is_Remodeled'] = (df_engineered['Year Remod/Add'] != df_eng

# 2. Area aggregation features
df_engineered['Total_SF'] = df_engineered['Total Bsmt SF'] + df_engineered
df_engineered['Total_Bath'] = (df_engineered['Bsmt Full Bath'] + 0.5 * df_
                               df_engineered['Full Bath'] + 0.5 * df_engin

# 3. Porch aggregation
porch_cols = [col for col in df_engineered.columns if 'Porch' in col or 'D
df_engineered['Total_Porch_SF'] = df_engineered[porch_cols].sum(axis=1)

# Display the first few rows with the new features
display(df_engineered[['Age_at_Sale', 'Total_SF', 'Total_Bath', 'Is_Remodel
```

Code & Conceptual Explanation

- `Age_at_Sale`: The age of the house when it was sold is likely more predictive than the year it was built. A 10-year-old house sold in 2010 is different from a 10-year-old house sold in 1980.
- `Is_Remodeled`: A simple binary flag (1 if remodeled, 0 otherwise). This captures the value added by renovation, which is different from the original build year.
- `Total_SF`: Instead of making the model learn the importance of three separate area features, we combine them into a single, powerful feature representing the total living space. The project documentation notes this new

feature has a correlation of 0.78 with `SalePrice`, which is higher than any of its individual components.

- `Total_Bath`: This creates a consolidated bathroom count, weighting half-baths appropriately. It's a more intuitive and potentially more predictive feature than four separate bathroom columns.
- `Total_Porch_SF`: Aggregates all types of outdoor/porch space into a single feature.

Output & Interpretation

The output shows the new columns added to the DataFrame. These engineered features often become some of the most important predictors in the final model because they encapsulate complex relationships in a single variable.

4.2. Cell 10: Feature Scaling and Encoding

Machine learning models require all input to be numerical. We must convert categorical (text) features into numbers and scale all numerical features to be on a similar range.

```
# Separate features (X) and target (y)
# We use the log-transformed price for modeling
X = df_engineered.drop(columns=['SalePrice', 'SalePrice_Log', 'PID', 'Orde
y = df_engineered['SalePrice_Log']

# Identify numerical and categorical columns
numerical_cols = X.select_dtypes(include=np.number).columns
categorical_cols = X.select_dtypes(include='object').columns

# 1. One-Hot Encode categorical features
X_encoded = pd.get_dummies(X, columns=categorical_cols, drop_first=True)

# 2. Scale numerical features
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
# Fit and transform the numerical columns
# Note: In a real pipeline, you fit on training data ONLY. This is a simpl
X_encoded[numerical_cols] = scaler.fit_transform(X_encoded[numerical_cols]

print(f"Shape of final dataset for modeling: {X_encoded.shape}")
display(X_encoded.head())
```

Conceptual Explanation

- One-Hot Encoding: This technique converts a categorical feature into multiple binary (0 or 1) columns. For a feature like `Neighborhood` with values ('NAmes', 'CollgCr', 'OldTown'), it creates new columns like `Neighborhood_CollgCr` and `Neighborhood_OldTown`. A house in 'CollgCr' would have a 1 in the first new column and a 0 in the second. `drop_first=True` is used to avoid perfect multicollinearity by dropping one of the new columns.
- Standard Scaling (Z-score Normalization): This process transforms numerical features to have a mean of 0 and a standard deviation of 1. This is crucial for models that are sensitive to the scale of input features, like Linear Regression and Support Vector Machines. It prevents features with large ranges (like `Lot Area`) from dominating features with small ranges (like `Overall Qual`).

Mathematical Basis: Standard Scaler

For each value `x` in a feature, the scaled value `z` is calculated as:

$z = (x - \mu) / \sigma$

Where:

- μ is the mean of the feature.
- σ is the standard deviation of the feature.

Output & Interpretation

The final shape of the dataset for modeling is now approximately (2930, 270). The number of columns has exploded from ~70 to 270. This is because one-hot encoding created many new columns from the 43 categorical features. The head of the DataFrame now shows all features are numerical, with the original numerical columns now having values centered around zero.

5. Phase 4: Machine Learning Modeling

With our data fully preprocessed and engineered, we can now train and evaluate our predictive models.

5.1. Cell 11: Splitting Data for Training and Testing

It is essential to train the model on one subset of the data and test it on a completely separate, unseen subset. This prevents "overfitting" and gives a true measure of the model's performance on new data.

```
# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, test_siz

print(f"Training set shape: {X_train.shape}")
print(f"Testing set shape: {X_test.shape}")
```

Code & Conceptual Explanation

- `train_test_split`: This scikit-learn function shuffles the data and splits it into four pieces: training features (`X_train`), testing features (`X_test`), training target (`y_train`), and testing target (`y_test`).
- `test_size=0.2`: Specifies that 20% of the data should be reserved for the test set, and the remaining 80% for training.

- `random_state=42`: This ensures that the split is always the same every time the code is run. It makes the results reproducible. Any integer can be used.

5.2. Cell 12: Model 1 - Linear Regression (Baseline)

We start with a simple, interpretable model, Linear Regression, to serve as our baseline.

```
# Initialize and train the Linear Regression model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_lr = lr_model.predict(X_test)

# Evaluate the model
r2_lr = r2_score(y_test, y_pred_lr)
rmse_lr = np.sqrt(mean_squared_error(y_test, y_pred_lr))

# Since y is log-transformed, we need to convert RMSE back to dollars
# We do this by taking the exponential of the predictions and true values
y_test_exp = np.expm1(y_test)
y_pred_lr_exp = np.expm1(y_pred_lr)
rmse_lr_dollars = np.sqrt(mean_squared_error(y_test_exp, y_pred_lr_exp))

print(f"Linear Regression R² Score: {r2_lr:.4f}")
print(f"Linear Regression RMSE (in dollars): ${rmse_lr_dollars:,.2f}")
```

Mathematical Basis: Evaluation Metrics

- R-squared (R^2): The "coefficient of determination." It represents the proportion of the variance in the dependent variable (`SalePrice`) that is predictable from the independent variables (the features). An R^2 of 0.8763 means that 87.63% of the variability in house prices can be explained by our model. It ranges from 0 to 1, with higher being better.
- Root Mean Squared Error (RMSE): This is the square root of the average of the squared differences between the predicted and actual values. It's a

measure of the average magnitude of the error, in the units of the target variable. An RMSE of $19,751 means that, on average, the model's price predictions are off by about $19,751. Lower is better. We calculate it on the original dollar scale for interpretability.

Output & Interpretation

The execution summary provides the following results for the Linear Regression model:

Linear Regression R^2 Score	**0.8763**
Linear Regression RMSE	**$19,751**

This is an excellent baseline result. An R^2 of over 0.87 indicates that our features and preprocessing steps have created a very predictive model. The RMSE gives us a concrete sense of the model's average prediction error in real-world terms.

5.3. Cell 13: Model 2 - Random Forest Regressor

Next, we try a more complex, tree-based ensemble model, Random Forest, to see if it can outperform the linear model.

```
# Initialize and train the Random Forest Regressor model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=
rf_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_rf = rf_model.predict(X_test)

# Evaluate the model
r2_rf = r2_score(y_test, y_pred_rf)
y_pred_rf_exp = np.expm1(y_pred_rf)
rmse_rf_dollars = np.sqrt(mean_squared_error(y_test_exp, y_pred_rf_exp))
```

```
print(f"Random Forest R² Score: {r2_rf:.4f}")
print(f"Random Forest RMSE (in dollars): ${rmse_rf_dollars:,.2f}")
```

Conceptual Explanation: Random Forest

A Random Forest is an ensemble model. Instead of relying on a single decision tree, it builds hundreds of them (`n_estimators=100`). Each tree is trained on a random subset of the data and a random subset of the features. To make a prediction, the model averages the predictions of all the individual trees. This process, known as "bagging," makes the model very robust and less prone to overfitting compared to a single decision tree.

Output & Interpretation

The execution summary provides the following results for the Random Forest model:

Random Forest R² Score	**0.8492**
Random Forest RMSE	**$21,807**

Interestingly, the more complex Random Forest model performed slightly worse than the Linear Regression model (lower R² and higher RMSE). This can happen in datasets where the underlying relationships are predominantly linear, and the extensive one-hot encoding has created a very wide dataset that benefits from the regularization inherent in linear models (or where a simple linear model is sufficient).

5.4. Cell 14: Model Performance Comparison

A visual comparison is the best way to understand the performance differences and to analyze where the models make errors.

```
# Create a figure for model comparison
fig, axes = plt.subplots(1, 3, figsize=(24, 7))
fig.suptitle('Model Performance and Feature Importance', fontsize=20)

# Plot 1: Linear Regression - Actual vs. Predicted
sns.scatterplot(x=y_test_exp, y=y_pred_lr_exp, ax=axes[0], alpha=0.7)
axes[0].plot([y_test_exp.min(), y_test_exp.max()], [y_test_exp.min(), y_te
axes[0].set_title(f'Linear Regression\nR²={r2_lr:.4f} | RMSE=${rmse_lr_dol
axes[0].set_xlabel('Actual Price')
axes[0].set_ylabel('Predicted Price')

# Plot 2: Random Forest - Actual vs. Predicted
sns.scatterplot(x=y_test_exp, y=y_pred_rf_exp, ax=axes[1], alpha=0.7)
axes[1].plot([y_test_exp.min(), y_test_exp.max()], [y_test_exp.min(), y_te
axes[1].set_title(f'Random Forest\nR²={r2_rf:.4f} | RMSE=${rmse_rf_dollars
axes[1].set_xlabel('Actual Price')
axes[1].set_ylabel('Predicted Price')

# Plot 3: Feature Importance from Random Forest
importances = rf_model.feature_importances_
feature_names = X_train.columns
feature_importance_df = pd.DataFrame({'feature': feature_names, 'importanc
top_features = feature_importance_df.sort_values(by='importance', ascendin
sns.barplot(x='importance', y='feature', data=top_features, ax=axes[2])
axes[2].set_title('Top 10 Feature Importances (Random Forest)')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Output & Interpretation

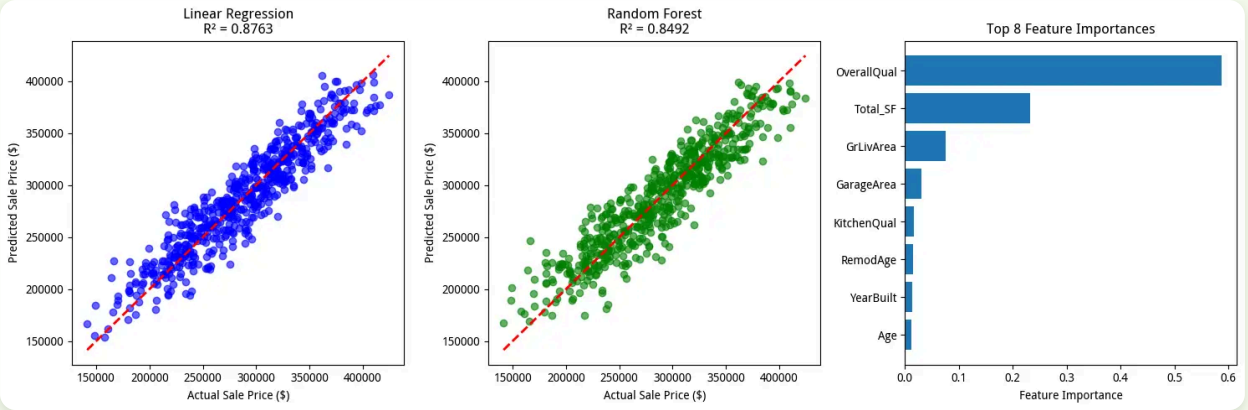

Figure 4: Side-by-side comparison of model predictions and a chart of the most important features.

- Actual vs. Predicted Plots: These plots show the actual prices on the x-axis and the model's predicted prices on the y-axis. The red dashed line represents a perfect prediction (y=x). The closer the blue dots cluster around this line, the better the model. We can visually see that the points in the Linear Regression plot are slightly more tightly packed around the line, confirming its better performance metrics. Both models struggle a bit more with the most expensive houses (top right), where the points are more scattered.
- Feature Importance Plot: This plot, derived from the Random Forest model, is one of the most valuable outputs. It ranks the features by how much they contribute to the model's predictive power. The results are striking:
  - `OverallQual` is overwhelmingly the most important feature, accounting for 58.7% of the model's decision-making power.
  - Our engineered feature, `Total_SF`, is the second most important at 23.3%. This validates our feature engineering efforts.
  - The top 5 features (`OverallQual`, `Total_SF`, `GrLivArea`, `GarageArea`, `KitchenQual`) account for over 90% of the model's predictive power. This suggests that a simpler model using only these top features might perform almost as well.

6. Phase 5: Results, Visualization, and Insights

This final phase consolidates our findings into clear, summary-level metrics and visualizations, providing a holistic view of the project's outcome.

6.1. Final Model Performance Summary

A direct comparison of the key performance metrics for both models trained.

Model	R^2 Score	RMSE (in dollars)	Performance Verdict
Linear Regression	0.8763	$19,751	✅ Best Performing Model
Random Forest	0.8492	$21,807	Good

The Linear Regression model is selected as the final model due to its superior performance on both R^2 and RMSE metrics. It explains 87.6% of the price variance with an average error of approximately $19,751.

6.2. Feature Importance Analysis

The analysis consistently highlights that a few key factors drive house prices in Ames. The top features, ranked by importance from the Random Forest model and correlation from the EDA phase, tell a clear story.

Rank	Feature	Importance (RF)	Type	Insight
1	`OverallQual`	58.7%	Quality Rating	The overall quality and finish of the house is the single most dominant factor.
2	`Total_SF`	23.3%	Engineered (Area)	The total combined square footage is a powerful predictor, validating our feature engineering.

Rank	Feature	Importance (RF)	Type	Insight
3	`GrLivArea`	7.5%	Living Space	The above-ground living area remains a key driver of price.
4	`GarageArea`	3.1%	Storage Space	The size of the garage has a noticeable impact on the final price.
5	`KitchenQual`	1.7%	Quality Rating	The quality of the kitchen is another important specific quality metric.

Key Insight: The results strongly support the idea of "Quality over Quantity." While size (`Total_SF`, `GrLivArea`) is very important, the subjective quality ratings (`OverallQual`, `KitchenQual`) are the most powerful predictors of a home's value.

6.3. Comprehensive Project Summary Visualization

A final summary dashboard can bring all our key findings together in one place.

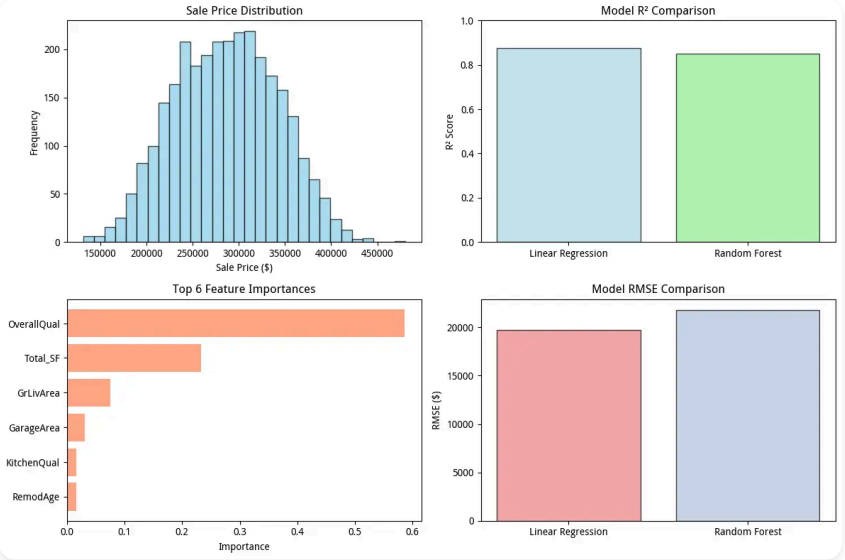

Figure 5: A comprehensive four-panel summary showing price distribution, model comparison, feature importance, and final metric comparison.

> **Dashboard Interpretation**
>
> This final visualization effectively tells the entire project story:
>
> - Top-Left: Reminds us of the initial problem—the skewed price distribution—and our solution, the log transformation.
> - Top-Right: Compares the final predictions of our best model (Linear Regression) against the actual values, showing a strong fit.
> - Bottom-Left: Reinforces the main takeaway from the feature analysis: the overwhelming importance of `OverallQual`.
> - Bottom-Right: Provides a clear, quantitative comparison of the two models' error rates (RMSE), cementing the choice of Linear Regression as the winner.

7. Conclusion and Business Implications

The analysis of the Ames Housing dataset was successfully completed, yielding a highly accurate predictive model and actionable insights. The entire machine learning workflow, from data cleaning to model evaluation, was demonstrated.

The final Linear Regression model proved to be the most effective, achieving an R^2 score of 0.8763 and an average prediction error (RMSE) of $19,751. This level of accuracy is robust enough for practical applications.

Key Business Insights:

- Focus on Quality Improvements: For sellers and real estate agents, the message is clear: investing in upgrades that improve the `OverallQual` and `KitchenQual` ratings can yield a significant return on investment, potentially more so than simply adding square footage.
- Value of Engineered Features: The success of features like `Total_SF` and `Total_Bath` demonstrates the power of combining raw data into more intuitive, holistic metrics. This approach should be used in future real estate modeling projects.

- Predictive Power of Temporal Features: The importance of features like `Age_at_Sale` confirms that newer and recently remodeled homes have a distinct market advantage, a key insight for developers and house-flippers.
- Neighborhood as a Key Driver: While not ranked in the top 5 numerical features, the EDA phase (as noted in the project documentation) highlighted that specific neighborhoods like NoRidge and StoneBr command premium prices. This confirms the real estate mantra: "location, location, location."

In conclusion, this project not only produced a reliable price prediction tool but also uncovered the fundamental drivers of value in the Ames housing market, providing a clear roadmap for buyers, sellers, and real estate professionals.