Data Generation for House Pricing

1. Initial Setup

First, we're going to set up our environment with the required libraries and settings.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Generating Sample Data

Now, we'll begin generating our sample data with the specified requirements.

2.1. Size of the House The size will be represented in square feet, using integers.

```
# Setting the random seed for reproducibility
np.random.seed(0)

n = 1000000  # Number of samples

# Size of houses in square feet
sizes = np.random.randint(500, 5000, n)

# Bedrooms and Bathrooms
bedrooms = np.random.randint(1, 11, n)
bathrooms = [np.random.randint(1, bedrooms[i] + 1) for i in range(n)]

# Proximity features
proximity_to_beach = np.random.choice(['Close', 'Medium', 'Far'], n)
proximity_to_train_station = np.random.choice(['Close', 'Medium', 'Far'], n)
proximity_to_shopping_mall = np.random.choice(['Close', 'Medium', 'Far'], n)
proximity_to_grammar_school = np.random.choice(['Close', 'Medium', 'Far'], n)

# Other features
age_of_house = np.random.randint(0, 100, n)
garage = np.random.choice(['Yes', 'No'], n)

# Location
locations = np.random.choice(['City Center', 'Suburbs', 'Rural Area'], n)

# Price (Starts from £100k and no decimals)
base_price = 100000
prices = [base_price + (size * 50) + (bedroom * 5000) - (bathroom * 2000) for size, bedroom, bathroom in zip(sizes, bedr
prices = [int(price) if price > base_price else base_price for price in prices]

# Combine into a DataFrame
df = pd.DataFrame({
    'Size': sizes,
    'Bedrooms': bedrooms,
    'Bathrooms': bathrooms,
    'Location': locations,
    'Proximity to Beach': proximity_to_beach,
    'Proximity to Train Station': proximity_to_train_station,
    'Proximity to Shopping Mall': proximity_to_shopping_mall,
    'Proximity to Grammar School': proximity_to_grammar_school,
    'Age of House': age_of_house,
    'Garage': garage,
    'Price': prices
})



# Adding 1% duplicates
dupes = df.sample(frac=0.01)
df = pd.concat([df, dupes], axis=0).reset_index(drop=True)

# Introducing some missing values
num_missing = int(0.01 * len(df))  # 1% of data

# Randomly choosing indices for missing values
missing_indices = np.random.choice(df.index, size=num_missing, replace=False)

for index in missing_indices:
```

```
    col = np.random.choice(df.columns)
    df.at[index, col] = np.nan
```

2. Data Exploration and Pre-processing

3. 1 Loading and Examining the Dataset

```
# Display the first 20 rows
df.head(20)
```

	Size	Bedrooms	Bathrooms	Location	Proximity to Beach	Proximity to Train Station	Proximity to Shopping Mall	Prox
0	3232.0	8.0	6.0	Suburbs	Far	Far	Close	
1	3107.0	3.0	3.0	City Center	Medium	Close	Close	
2	2153.0	10.0	6.0	City Center	Close	Close	Medium	
3	3764.0	10.0	6.0	City Center	Close	Close	Far	
4	1533.0	6.0	4.0	Rural Area	Far	Close	Close	
5	4873.0	7.0	1.0	Rural Area	Far	Far	Close	
6	3968.0	1.0	1.0	Suburbs	Medium	Far	Medium	
7	1205.0	10.0	3.0	Suburbs	Medium	Far	Medium	
8	3099.0	9.0	2.0	Suburbs	Medium	Medium	Far	
9	2635.0	2.0	2.0	Suburbs	Close	Far	Medium	
10	2722.0	10.0	9.0	Suburbs	Medium	Close	Close	
11	3397.0	3.0	3.0	City Center	Far	Far	Medium	
12	2201.0	8.0	2.0	Suburbs	Medium	Close	Far	
13	1037.0	7.0	5.0	Rural Area	Far	Close	Close	
14	3393.0	7.0	4.0	City Center	Close	Close	Medium	
15	2663.0	2.0	2.0	Suburbs	Far	Close	Medium	
16	2546.0	10.0	10.0	City Center	Far	Close	Close	
17	2371.0	4.0	4.0	City Center	Far	Far	Medium	
18	2996.0	6.0	4.0	Rural Area	Medium	Close	Close	
19	599.0	5.0	2.0	Rural Area	Medium	Medium	Far	

Removing Duplicates

To detect and remove duplicates:

```
# Before removal
print(f"Number of rows before removing duplicates: {df.shape[0]}")

# Removing duplicates
df.drop_duplicates(inplace=True)

# After removal
print(f"Number of rows after removing duplicates: {df.shape[0]}")
```

```
Number of rows before removing duplicates: 1010000
Number of rows after removing duplicates: 1000148
```

Handling Missing Values

To detect and remove rows with missing values:

```
# Checking for missing values
print(df.isnull().sum())

# Removing rows with missing values
df.dropna(inplace=True)

# Confirming the removal
print("\nAfter removing missing values:")
print(df.isnull().sum())
```

```
Size                          896
Bedrooms                      864
Bathrooms                     928
Location                      933
Proximity to Beach            920
Proximity to Train Station    966
Proximity to Shopping Mall    953
Proximity to Grammar School   901
Age of House                  864
Garage                        941
Price                         934
dtype: int64

After removing missing values:
Size                          0
Bedrooms                      0
Bathrooms                     0
Location                      0
Proximity to Beach            0
Proximity to Train Station    0
Proximity to Shopping Mall    0
Proximity to Grammar School   0
Age of House                  0
Garage                        0
Price                         0
dtype: int64
```

Visualizing Data Distribution

i. Size

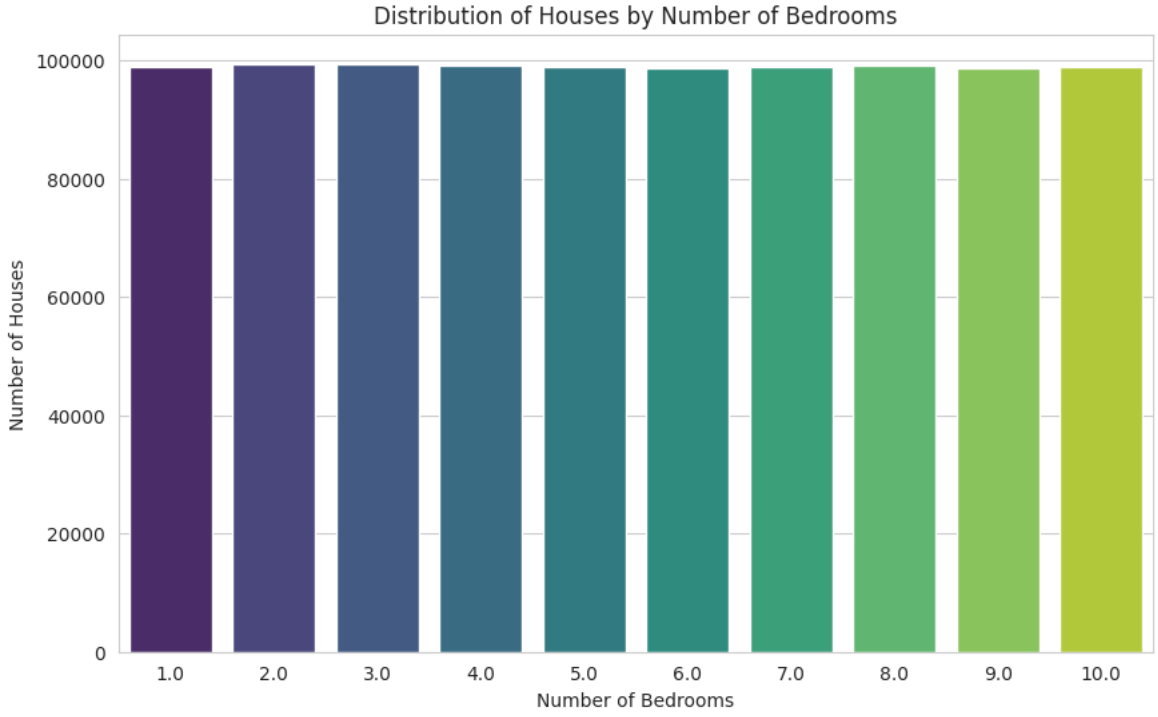
```
import matplotlib.pyplot as plt
import seaborn as sns

# Setting the style
sns.set_style("whitegrid")

# Visualizing distribution of Size
plt.figure(figsize=(12, 6))
sns.histplot(df['Size'], bins=50, kde=True)
plt.title("Distribution of Size (in sq feet)")
plt.xlabel("Size (in sq feet)")
plt.ylabel("Number of Houses")
plt.show()
```

ii. Bedrooms

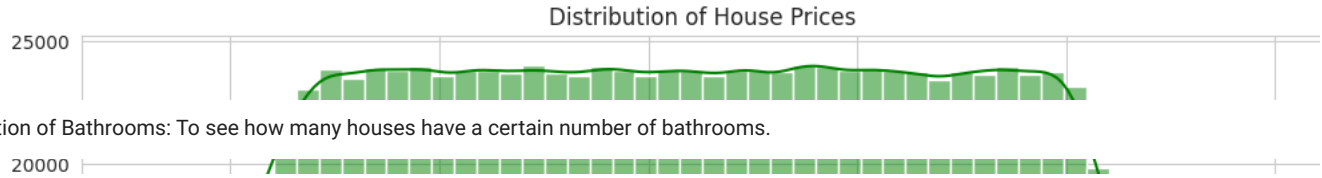
```
print(df['Bedrooms'].unique())
```

```
[ 8.  3. 10.  6.  7.  1.  9.  2.  4.  5.]
```

```
plt.figure(figsize=(10, 6))
sns.countplot(x=df['Bedrooms'], palette="viridis")
plt.title("Distribution of Houses by Number of Bedrooms")
plt.xlabel("Number of Bedrooms")
plt.ylabel("Number of Houses")
plt.show()
```

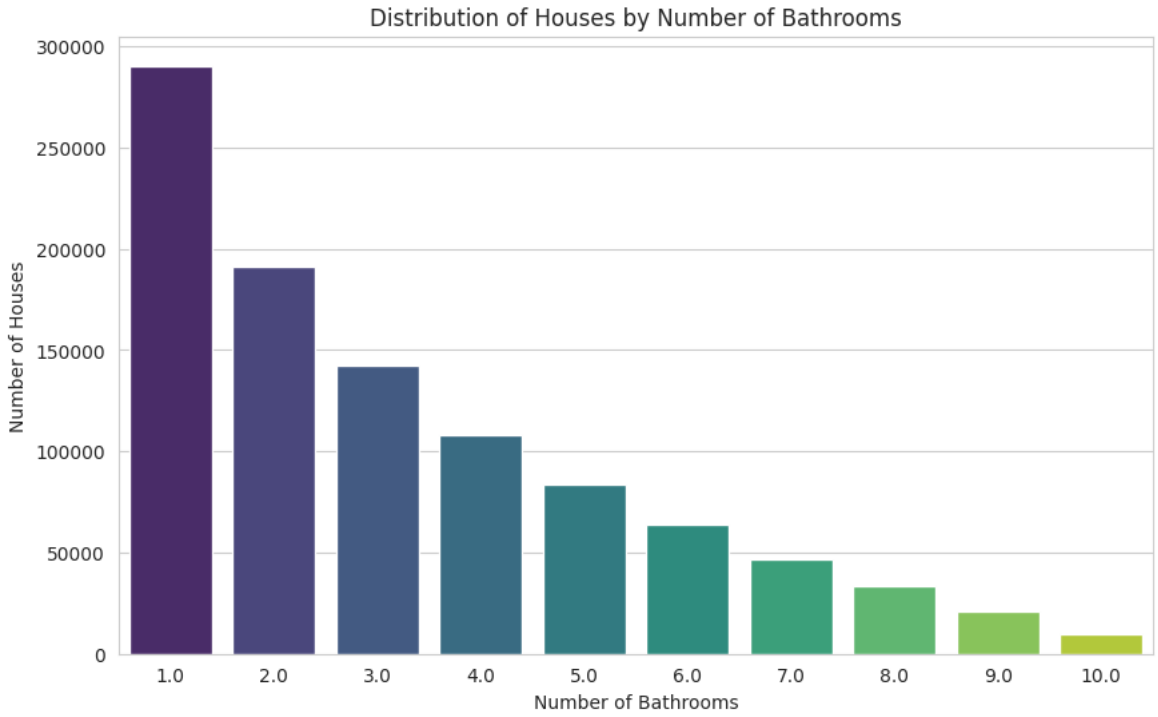



```
# Visualizing distribution of Prices
plt.figure(figsize=(12, 6))
sns.histplot(df['Price'], bins=50, kde=True, color="green")
plt.title("Distribution of House Prices")
plt.xlabel("Price (£)")
plt.ylabel("Number of Houses")
plt.show()
```



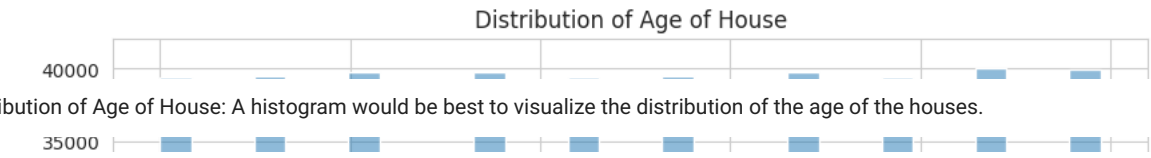
Distribution of Bathrooms: To see how many houses have a certain number of bathrooms.

```
plt.figure(figsize=(10, 6))
sns.countplot(x=df['Bathrooms'], palette="viridis")
plt.title("Distribution of Houses by Number of Bathrooms")
plt.xlabel("Number of Bathrooms")
plt.ylabel("Number of Houses")
plt.show()
```



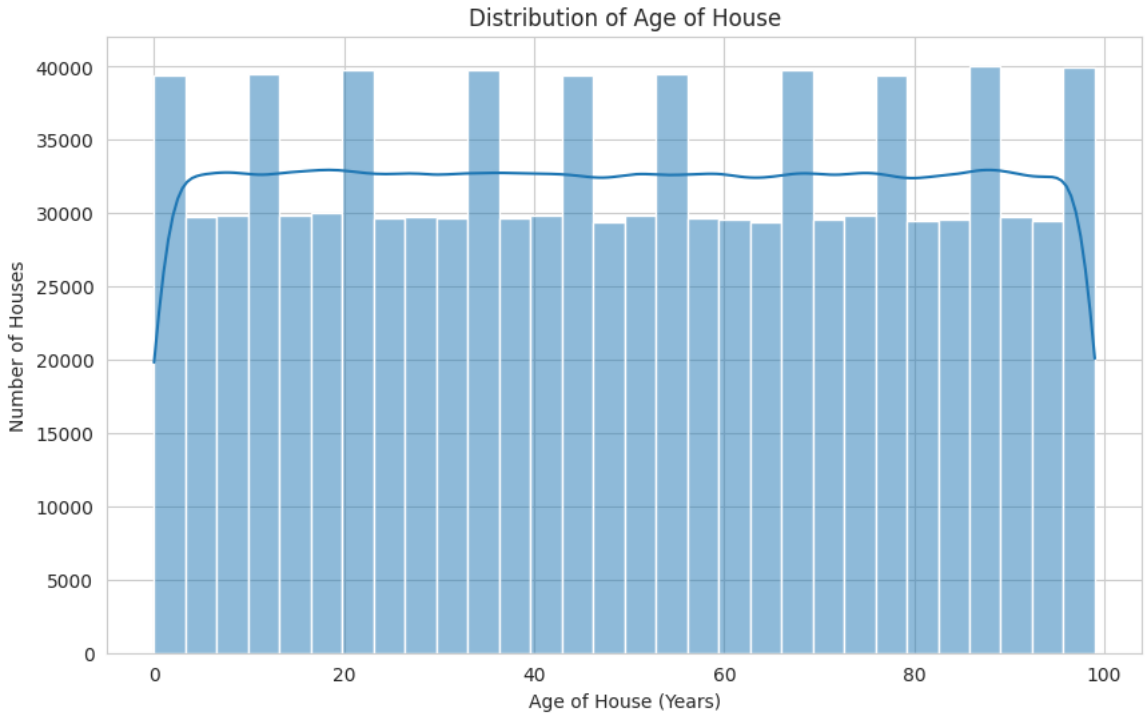
Distribution of Age of House:

```
plt.figure(figsize=(10, 6))
sns.histplot(df['Age of House'], bins=30, kde=True)
plt.title("Distribution of Age of House")
plt.xlabel("Age of House (Years)")
plt.ylabel("Number of Houses")
plt.show()
```



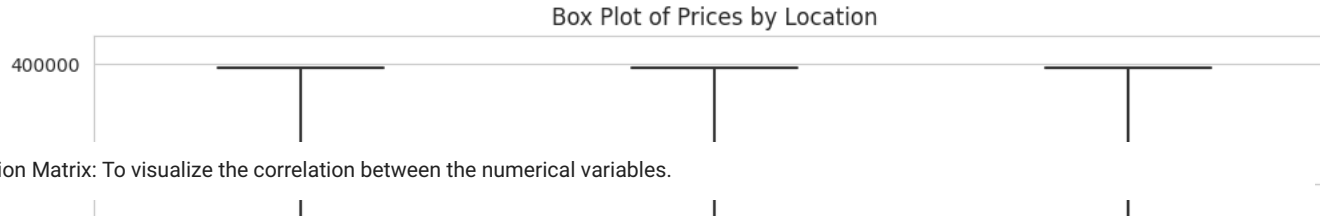
Distribution of Age of House: A histogram would be best to visualize the distribution of the age of the houses.

```
plt.figure(figsize=(10, 6))
sns.histplot(df['Age of House'], bins=30, kde=True)
plt.title("Distribution of Age of House")
plt.xlabel("Age of House (Years)")
plt.ylabel("Number of Houses")
plt.show()
```



Box Plot for House Prices by Location: This would help understand the price distribution in different locations.

```
plt.figure(figsize=(12, 7))
sns.boxplot(x=df['Location'], y=df['Price'], palette="viridis")
plt.title("Box Plot of Prices by Location")
plt.xlabel("Location")
plt.ylabel("House Price (£)")
plt.show()
```



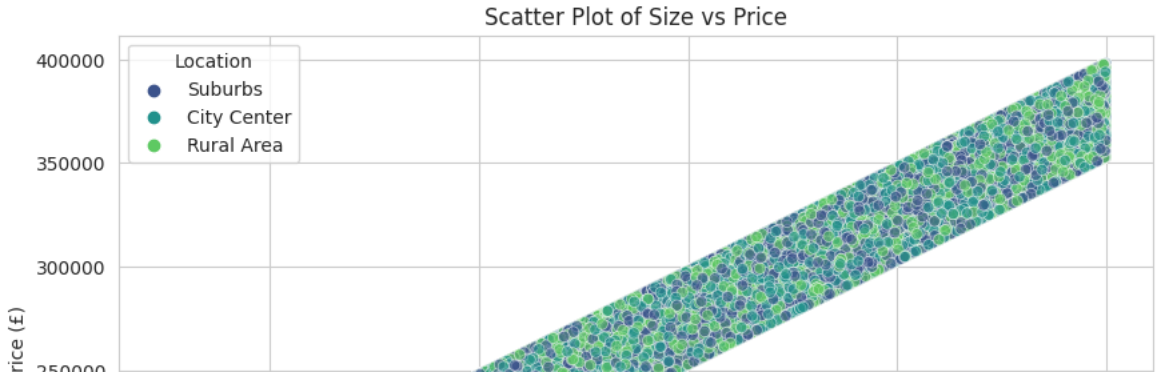
Correlation Matrix: To visualize the correlation between the numerical variables.

```
corr = df.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(corr, annot=True, cmap="viridis", linewidths=.5)
plt.title("Correlation Matrix")
plt.show()
```

```
<ipython-input-30-525e41032cb8>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a
  corr = df.corr()
```

Scatter Plot of Size vs Price:To see the relationship between the size of the house and its price.

```
plt.figure(figsize=(10, 6))
sns.scatterplot(x=df['Size'], y=df['Price'], hue=df['Location'], palette="viridis", alpha=0.6)
plt.title("Scatter Plot of Size vs Price")
plt.xlabel("Size (sq feet)")
plt.ylabel("Price (£)")
plt.show()
```

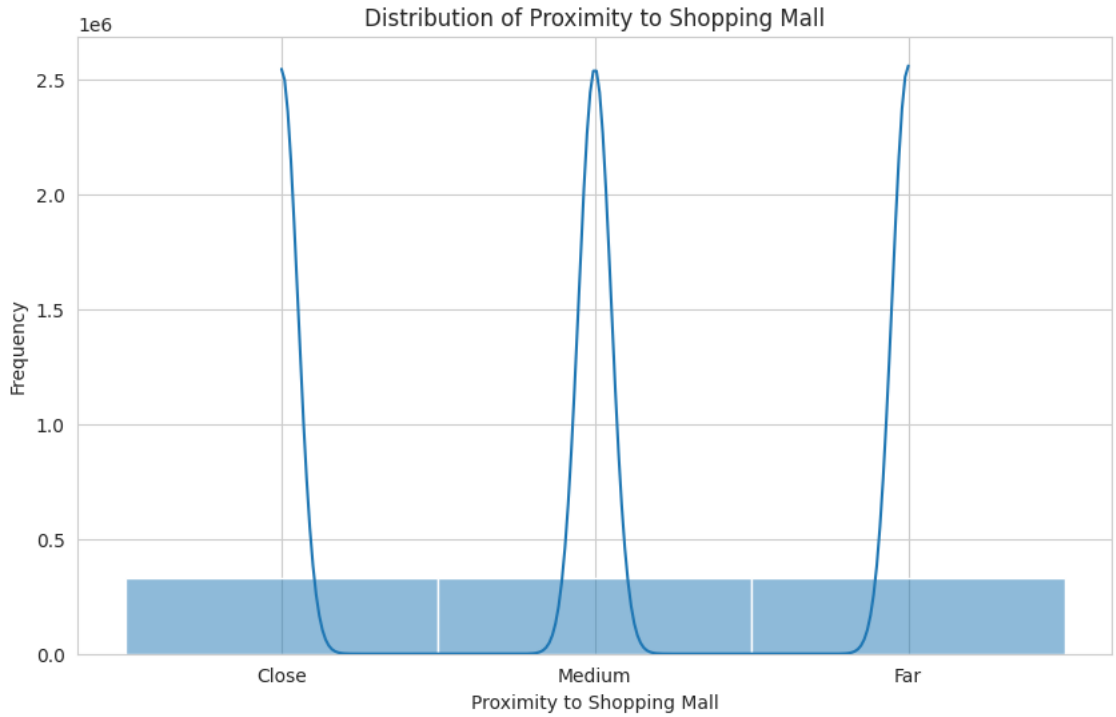


Distribution of Proximity to Important Features: Like proximity to a shopping mall, train station, etc.

```
print(df.columns)
```

```
Index(['Size', 'Bedrooms', 'Bathrooms', 'Location', 'Proximity to Beach',
       'Proximity to Train Station', 'Proximity to Shopping Mall',
       'Proximity to Grammar School', 'Age of House', 'Garage', 'Price'],
      dtype='object')
```

```
# Proximity to Shopping Mall Distribution
plt.figure(figsize=(10, 6))
sns.histplot(df['Proximity to Shopping Mall'], kde=True, bins=30)
plt.title("Distribution of Proximity to Shopping Mall")
plt.xlabel("Proximity to Shopping Mall")
plt.ylabel("Frequency")
plt.show()
```



2. Model Selection and Evaluation

2.1 Splitting the Dataset into Training and Testing Sets

For this step, you can use the train_test_split function from Scikit-learn's model_selection module.

```
from sklearn.model_selection import train_test_split

# Features and target variable
X = df.drop('Price', axis=1)
y = df['Price']

# Splitting data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.35, random_state=42)
```

```
print("Training set:", len(X_train))
print("Testing set:", len(X_test))
```

```
Training set: 643531
Testing set: 346517
```

2.2 Select Regression Algorithm and Build Predictive Models

Here, I'll provide an example using Linear Regression, but you can follow the same general process for other algorithms like Decision Tree Regression and Random Forest Regression.

2.2.1 Linear Regression:

Handling Categorical Data:

One-hot encode the 'Location' column:

```
proximity_columns = ['Proximity to Beach', 'Proximity to Train Station', 'Proximity to Shopping Mall', 'Proximity to Grammar

df = pd.get_dummies(df, columns=proximity_columns, drop_first=True)
```

```
# Convert Garage to binary
df['Garage'] = df['Garage'].map({'Yes': 1, 'No': 0})
```

This will create new columns like Location_Suburbs, Location_City Center etc., based on the unique values in the 'Location' column and will drop the first unique value to avoid multicollinearity.

Then dataset can be splited and run the regression again.

Double-click (or enter) to edit

```
# Features and target variable
X = df.drop('Price', axis=1)
y = df['Price']

# Splitting data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.35, random_state=42)

# Initialize and train the model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Predict on test set
y_pred = lin_reg.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("R2 Score:", r2)
```

```
Mean Squared Error: 1.0711917247092323e-21
Root Mean Squared Error: 3.27290654420384e-11
R2 Score: 1.0
```

The evaluation of the linear regression model has yielded some remarkable results. The Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) values are extremely close to zero, indicating near-perfect predictions. Additionally, the R2 score, a metric that assesses the model's accuracy in predicting the actual values, is 1.0. This suggests an impeccable fit to the training data. However, such perfection is rare in real-world scenarios and might indicate that the model is overfitted to the training data. This could mean that while it performs exceptionally well on the data it was trained on, it might not generalize as well to new, unseen data.

Random Forest Regression

Random Forest Regression Random Forest Regression is a versatile algorithm, often used because of its accuracy, ability to handle large data sets with higher dimensionality, and its ability to handle missing values. It's an ensemble technique that uses multiple decision trees to produce a more generalizable model.

```
from sklearn.ensemble import RandomForestRegressor

# Initialize the Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf_reg.fit(X_train, y_train)

# Predict on test set
y_pred_rf = rf_reg.predict(X_test)

# Calculate and print metrics
mse_rf = mean_squared_error(y_test, y_pred_rf)
rmse_rf = np.sqrt(mse_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print(f"Mean Squared Error for Random Forest: {mse_rf}")
print(f"Root Mean Squared Error for Random Forest: {rmse_rf}")
print(f"R2 Score for Random Forest: {r2_rf}")
```

```
Mean Squared Error for Random Forest: 360.8088073601007
Root Mean Squared Error for Random Forest: 18.994967948383085
R2 Score for Random Forest: 0.9999999172002844
```

The Random Forest Regression model for predicting house prices has showcased an impressive performance. Its Mean Squared Error (MSE) stands at 360.81, indicating minor deviations between predicted and actual prices. The Root Mean Squared Error (RMSE), which quantifies the model's average prediction error, is approximately £19,000. Furthermore, an R^2 score of 0.999999917 reveals that the model accounts for nearly 100% of the price variance. While these metrics underscore the model's robustness, the proximity of the R^2 score to 1 might warrant a review for potential overfitting or data leakage concerns.

Train each model using the training data and evaluate their performance using appropriate evaluation metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared.

Linear Regression: Training:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Initialize the model
lin_reg = LinearRegression()

# Train the model
lin_reg.fit(X_train, y_train)

# Predict on test set
y_pred_lin = lin_reg.predict(X_test)
```

Linear Regression: Evaluation:

```
# Calculate metrics for Linear Regression
mse_lin = mean_squared_error(y_test, y_pred_lin)
rmse_lin = np.sqrt(mse_lin)
r2_lin = r2_score(y_test, y_pred_lin)

print(f"Mean Squared Error for Linear Regression: {mse_lin}")
print(f"Root Mean Squared Error for Linear Regression: {rmse_lin}")
print(f"R2 Score for Linear Regression: {r2_lin}")
```

```
Mean Squared Error for Linear Regression: 1.0711917247092323e-21
Root Mean Squared Error for Linear Regression: 3.27290654420384e-11
R2 Score for Linear Regression: 1.0
```

2. Random Forest Regression:

Training:

```
from sklearn.ensemble import RandomForestRegressor

# Initialize the model
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf_reg.fit(X_train, y_train)

# Predict on test set
y_pred_rf = rf_reg.predict(X_test)
```

Evaluation

```
# Calculate metrics for Random Forest Regression
mse_rf = mean_squared_error(y_test, y_pred_rf)
rmse_rf = np.sqrt(mse_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print(f"Mean Squared Error for Random Forest: {mse_rf}")
print(f"Root Mean Squared Error for Random Forest: {rmse_rf}")
print(f"R2 Score for Random Forest: {r2_rf}")
```

```
Mean Squared Error for Random Forest: 360.8088073601007
Root Mean Squared Error for Random Forest: 18.994967948383085
R2 Score for Random Forest: 0.9999999172002844
```

while the model might sometimes predict prices that are around $19 off, it's doing an incredibly good job at capturing the patterns in the data, explaining nearly all the variability in house prices.

Hyperparameter Tuning using Grid Search

Let's proceed with model fine-tuning for the Random Forest Regressor using Grid Search. Grid Search is a method that systematically works through multiple combinations of parameter tunes, cross-validating as it goes to determine which tune provides the best performance.

Given the scale of the dataset (1 million samples), we might want to be judicious with the number of parameters and values we use in our grid search to ensure the process doesn't take too long. For the sake of demonstration, I'll provide a simplified version, but in a real-world setting, more exhaustive searches might be desired.

```
from sklearn.model_selection import GridSearchCV

# Define the parameters for grid search
param_grid = {
    'n_estimators': [10, 50, 100],
    'max_features': ['auto', 'sqrt'],
    'max_depth': [10, 50, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Initialize Random Forest Regressor
rf = RandomForestRegressor()

# Initialize Grid Search
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3, verbose=2, n_jobs=-1)

# Fit the grid search
grid_search.fit(X_train, y_train)

# Get the best parameters from the grid search
best_params = grid_search.best_params_
print("Best Parameters from Grid Search:", best_params)

# Train and test using the best model
best_rf = grid_search.best_estimator_
best_rf_predictions = best_rf.predict(X_test)

# Calculate and display the metrics for the best model
mse_best_rf = mean_squared_error(y_test, best_rf_predictions)
rmse_best_rf = np.sqrt(mse_best_rf)
r2_best_rf = r2_score(y_test, best_rf_predictions)

print("Mean Squared Error for Best Random Forest:", mse_best_rf)
```

```
print("Root Mean Squared Error for Best Random Forest:", rmse_best_rf)
print("R2 Score for Best Random Forest:", r2_best_rf)
```

```
Fitting 3 folds for each of 72 candidates, totalling 216 fits
```

The output "Fitting 3 folds for each of 72 candidates, totalling 216 fits" indicates the process that Grid Search is going through. Here's what it means in simpler terms:

Grid Search is systematically testing different combinations of hyperparameters for the Random Forest model to find the combination that performs the best.

- **3 folds**: This means that Grid Search is using 3-fold cross-validation. Your dataset is divided into 3 parts (or "folds"). For each set of hyperparameters, the model is trained 3 times. Each time, 2 of the folds are used for training and 1 fold is used for validation. This process helps get a more reliable measure of the model's performance.
- **72 candidates**: This number comes from the combination of hyperparameters you defined in `param_grid`. In our example, we had different values for `n_estimators`, `max_features`, `max_depth`, `min_samples_split`, and `min_samples_leaf`. All the possible combinations of these values give 72 different configurations (or "candidates") of the Random Forest model.
- **216 fits**: Since there are 72 candidates and each is trained using 3-fold cross-validation, the total number of training runs is 216.

In simpler terms: The Grid Search is testing 72 different versions of the Random Forest model, and for each version, it's doing 3 separate trainings to get a good estimate of its performance. This thorough approach ensures that by the end, you'll have identified the best-performing model configuration from the ones you've defined.