In [ ]:
```
# Importing all the necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder,StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, r2_score
import pickle
import joblib
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
# Loading the dataset
df = pd.read_csv('data.csv')
df
```

Out[ ]:

	brokered_by	status	price	bed	bath	acre_lot	street	city	state	zip_code	house_siz
0	103378.0	for_sale	105000.0	3.0	2.0	0.12	1962661.0	Adjuntas	Puerto Rico	601.0	920.
1	52707.0	for_sale	80000.0	4.0	2.0	0.08	1902874.0	Adjuntas	Puerto Rico	601.0	1527.
2	103379.0	for_sale	67000.0	2.0	1.0	0.15	1404990.0	Juana Diaz	Puerto Rico	795.0	748.
3	31239.0	for_sale	145000.0	4.0	2.0	0.10	1947675.0	Ponce	Puerto Rico	731.0	1800.
4	34632.0	for_sale	65000.0	6.0	2.0	0.05	331151.0	Mayaguez	Puerto Rico	680.0	Na
...	...	...	...	...	...	...	...	...	...	...	
39149	15366.0	for_sale	274000.0	3.0	2.0	0.23	893368.0	East Haven	Connecticut	6512.0	1426.
39150	22611.0	for_sale	279900.0	2.0	2.0	0.11	654312.0	East Haven	Connecticut	6512.0	1264.
39151	107953.0	for_sale	249900.0	4.0	2.0	0.17	1584359.0	East Haven	Connecticut	6512.0	1349.
39152	39954.0	for_sale	449900.0	5.0	3.0	0.11	1725145.0	East Haven	Connecticut	6512.0	2041.
39153	90155.0	for_sale	419900.0	4.0	3.0	0.40	1743526.0	New	NaN	NaN	Na

39154 rows × 12 columns

In [ ]:
```
# Printing the information about the DataFrame
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 39154 entries, 0 to 39153
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   brokered_by     39147 non-null  float64
 1   status          39154 non-null  object
 2   price           39154 non-null  float64
 3   bed             28962 non-null  float64
 4   bath            29030 non-null  float64
 5   acre_lot        33289 non-null  float64
 6   street          39059 non-null  float64
 7   city            39138 non-null  object
 8   state           39153 non-null  object
 9   zip_code        39125 non-null  float64
 10  house_size      29236 non-null  float64
 11  prev_sold_date  13405 non-null  object
dtypes: float64(8), object(4)
memory usage: 3.6+ MB
None
```

About the columns in the dataset:

1. brokered_by (`float64`): Agency or broker associated with the property listing, categorically encoded as numeric values.
2. status (`object`): Housing status, either "ready for sale" or "ready to build."
3. price (`float64`): Current listing price or recently sold price of the property.
4. bed (`float64`): Number of bedrooms in the property.
5. bath (`float64`): Number of bathrooms in the property.
6. acre_lot (`float64`): Property or land size in acres.
7. street (`float64`): Street address of the property, categorically encoded as numeric values.
8. city (`object`): City where the property is located.
9. state (`object`): State where the property is located.
10. zip_code (`float64`): Postal code of the area where the property is located.
11. house_size (`float64`): Size of the house or living space in square feet.
12. prev_sold_date (`object`): Date when the property was previously sold.

In [ ]:
```python
# Changing the data type of brokered_by, street and zip_code to string
df['brokered_by'] = df['brokered_by'].astype(str)
df['street'] = df['street'].astype(str)
df['zip_code'] = df['zip_code'].astype(str)
```

The data type of columns brokered_by, street, and zip_code have been changed from numbers to text. This makes it easier to work with these categories as labels instead of numbers.

In [ ]:
```python
# Checking the missing values in the dataframe
print(df.isnull().sum())
```

```
brokered_by           0
status                0
price                 0
bed               10192
bath              10124
acre_lot           5865
street                0
city                 16
state                 1
zip_code              0
house_size         9918
prev_sold_date    25749
dtype: int64
```

- The columns with the most missing values are house_size and prev_sold_date.
- The bed and bath columns also have a significant number of missing values.
- Price, a critical column for analysis and modeling, has 335 missing values that need to be addressed.
- The categorical columns brokered_by, street, and zip_code have no missing values, making them easier to work with during encoding.

In [ ]:
```
# Printing the information about the DataFrame
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 39154 entries, 0 to 39153
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   brokered_by     39154 non-null  object 
 1   status          39154 non-null  object 
 2   price           39154 non-null  float64
 3   bed             28962 non-null  float64
 4   bath            29030 non-null  float64
 5   acre_lot        33289 non-null  float64
 6   street          39154 non-null  object 
 7   city            39138 non-null  object 
 8   state           39153 non-null  object 
 9   zip_code        39154 non-null  object 
 10  house_size      29236 non-null  float64
 11  prev_sold_date  13405 non-null  object 
dtypes: float64(5), object(7)
memory usage: 3.6+ MB
None
```

The DataFrame has 2226382 entries and 12 columns. The columns brokered_by, street, and zip_code are now of type object(string).

In [ ]:
```
# Checking the description of the data in dataframe
print(df.describe())
```

```
              price           bed          bath       acre_lot    house_size
count  3.915400e+04  28962.000000  29030.000000   33289.000000  2.923600e+04
mean   6.044362e+05      3.467854      2.552187      25.270797  2.227991e+03
std    1.211395e+06      2.030758      1.978966    1236.480212  8.700302e+03
min    1.000000e+00      1.000000      1.000000       0.000000  1.000000e+02
25%    1.749000e+05      2.000000      2.000000       0.230000  1.224000e+03
50%    3.399000e+05      3.000000      2.000000       0.700000  1.756000e+03
75%    6.099000e+05      4.000000      3.000000       2.500000  2.588000e+03
max    6.000000e+07     99.000000    198.000000  100000.000000  1.450112e+06
```

In [ ]:
```
# Coverting the prev_sold_date to datetime and filling the missing values
df['prev_sold_date'] = pd.to_datetime(df['prev_sold_date'], errors='coerce')
```

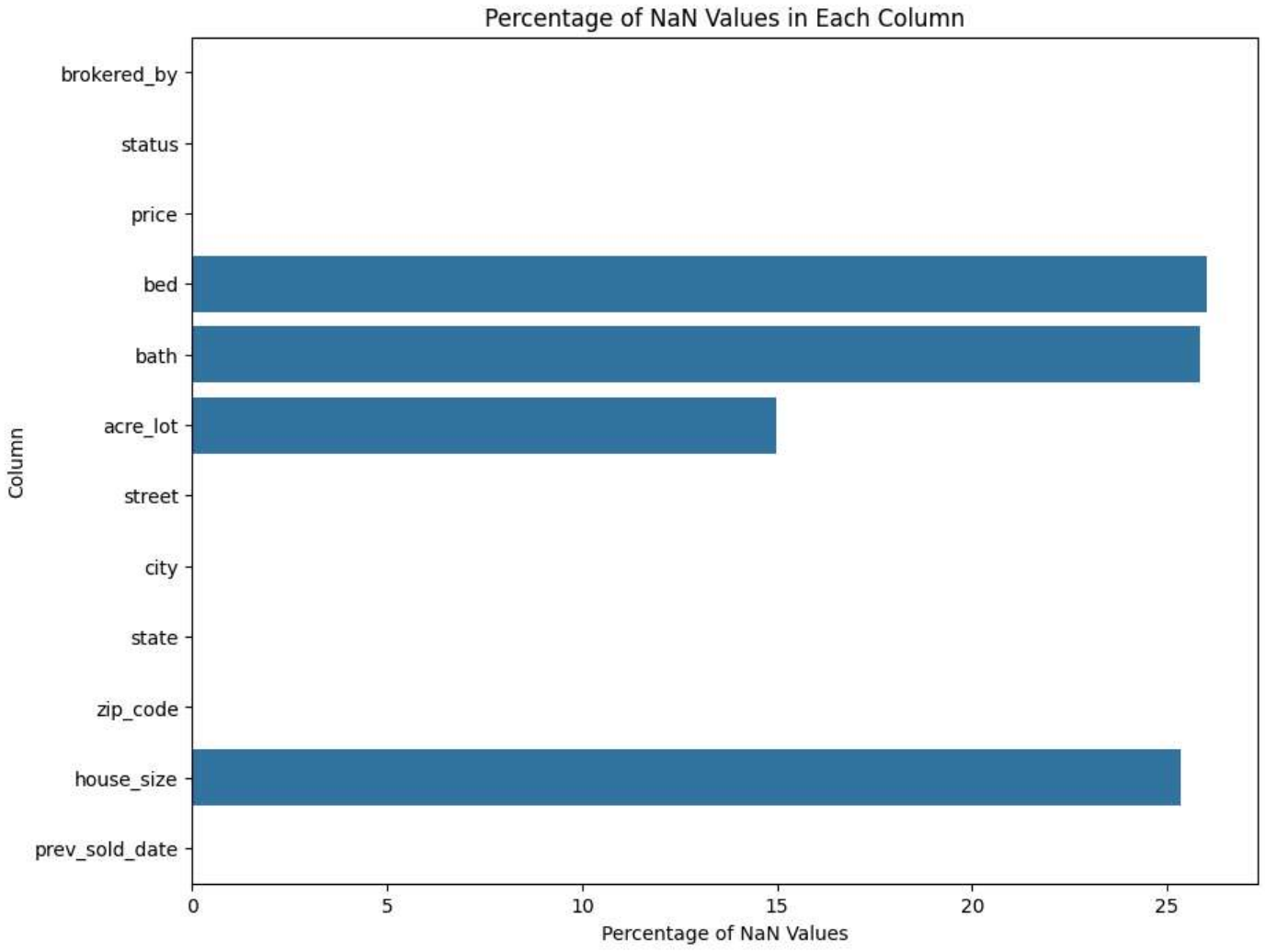
```
df['prev_sold_date'] = df['prev_sold_date'].fillna(pd.to_datetime('2000-01-01'))
```

The prev_sold_date column has been converted to datetime format. Missing values have been filled with January 1, 2000.

In [ ]:
```
# Calculating the percentage of NaN values in each column
def calculate_nan_percentage(data):
    nan_percentage = data.isna().mean() * 100
    return nan_percentage.reset_index().rename(columns={0: 'NaN_Percentage', 'index': 'Column'})
```

It calculates and returns the percentage of NaN values in each column of the DataFrame, showing it as a percentage.

In [ ]:
```
# Visualizing the percentage of missing values (NaN) in each column of a DataFrame
nan_percentage = calculate_nan_percentage(df)
plt.figure(figsize=(10, 8))  # Set the figure size for the plot
sns.barplot(data=nan_percentage, x='NaN_Percentage', y='Column')  # Create a bar plot using seab
plt.title('Percentage of NaN Values in Each Column')  # Set the plot title
plt.xlabel('Percentage of NaN Values')  # Set the x-axis label
plt.ylabel('Column')  # Set the y-axis label
plt.show()
```



The bar plot shows that price, house_size, bed, bath, and acre_lot have significant missing values.

Step 1: Data Cleaning

In [ ]:
```
# Creating a new DataFrame by removing any rows from the original DataFrame df where the price c
df_clean = df.dropna(subset=['price'])
```

In [ ]:
```
# Filtering the DataFrame df_clean to keep only the rows that have fewer than 3 missing values
df_clean = df_clean[df_clean.isna().sum(axis=1) < 3]
```

In [ ]:
```
# Filling any missing values in the house_size column of the DataFrame with the median value of
df_clean['house_size'] = df_clean['house_size'].fillna(df_clean['house_size'].median())
```

In [ ]:
```
# Calculating the average number of bathrooms for each bedroom, rounding the averages to the nea
avg_bath_by_bed = df_clean.groupby('bed')['bath'].mean().round(0).reset_index()
avg_bath_by_bed.columns = ['bed', 'bath_avg']
```

In [ ]:
```
# Merging the avg_bath_by_bed DataFrame with df_clean on the bed column, adding the bath_avg col
df_clean = df_clean.merge(avg_bath_by_bed, on='bed', how='left')
```

In [ ]:
```
# Filling the missing values in the bath column of df_clean with corresponding values from the b
df_clean['bath'] = df_clean['bath'].fillna(df_clean['bath_avg'])
```

In [ ]:
```
# Removing the bath_avg column from the df_clean DataFrame
df_clean = df_clean.drop(columns=['bath_avg'])
```

In [ ]:
```
# Filling any missing values in the bed column of df_clean with the median value of the bed colu
df_clean['bed'] = df_clean['bed'].fillna(df_clean['bed'].median())
```

In [ ]:
```
# Calculating the percentage of missing values for each column in the df_clean DataFrame and sto
nan_percentage_clean = calculate_nan_percentage(df_clean)
nan_percentage_clean
```

Out[ ]:

	Column	NaN_Percentage
0	brokered_by	0.000000
1	status	0.000000
2	price	0.000000
3	bed	0.000000
4	bath	0.960527
5	acre_lot	19.682313
6	street	0.000000
7	city	0.006788
8	state	0.003394
9	zip_code	0.000000
10	house_size	0.000000
11	prev_sold_date	0.000000

In [ ]:
```
# Imputing numerical columns with median
for col in ['bed', 'bath', 'acre_lot', 'house_size']:
    df_clean[col] = df_clean[col].fillna(df[col].median())

# Imputing categorical columns with mode
df_clean['city'] = df_clean['city'].fillna(df['city'].mode()[0])
df_clean['state'] = df_clean['state'].fillna(df['state'].mode()[0])
```

```
# Ensuring 'price' has no NaNs
df_clean = df_clean.dropna(subset=['price'])
```

In [ ]:
```
# Ensuring all columns are numeric
numeric_columns = ['price', 'bed', 'bath', 'acre_lot', 'house_size']
for col in numeric_columns:
    df_clean[col] = pd.to_numeric(df_clean[col], errors='coerce')
```

The specified columns in df_clean are now in a numeric format, with any non-convertible values replaced by NaN, ensuring the dataset is ready for further numerical analysis and modeling.

In [ ]:
```
df_final = df_clean.dropna(subset=numeric_columns)
df_final
```

Out[ ]:

	brokered_by	status	price	bed	bath	acre_lot	street	city	state	zip_code	house_
0	103378.0	for_sale	105000.0	3.0	2.0	0.12	1962661.0	Adjuntas	Puerto Rico	601.0	9
1	52707.0	for_sale	80000.0	4.0	2.0	0.08	1902874.0	Adjuntas	Puerto Rico	601.0	15
2	103379.0	for_sale	67000.0	2.0	1.0	0.15	1404990.0	Juana Diaz	Puerto Rico	795.0	7
3	31239.0	for_sale	145000.0	4.0	2.0	0.10	1947675.0	Ponce	Puerto Rico	731.0	18
4	34632.0	for_sale	65000.0	6.0	2.0	0.05	331151.0	Mayaguez	Puerto Rico	680.0	17
...	...	...	...	...	...	...	...	...	...	...	
29458	15366.0	for_sale	274000.0	3.0	2.0	0.23	893368.0	East Haven	Connecticut	6512.0	14
29459	22611.0	for_sale	279900.0	2.0	2.0	0.11	654312.0	East Haven	Connecticut	6512.0	12
29460	107953.0	for_sale	249900.0	4.0	2.0	0.17	1584359.0	East Haven	Connecticut	6512.0	13
29461	39954.0	for_sale	449900.0	5.0	3.0	0.11	1725145.0	East Haven	Connecticut	6512.0	20
29462	90155.0	for_sale	419900.0	4.0	3.0	0.40	1743526.0	New	Massachusetts	nan	17

29463 rows × 12 columns

Step 2: Feature Engineering

In [ ]:
```
# Calculating the price per square foot for each house and addding it as a new column in the Dat
df_final['price_per_sqft'] = df_final['price'] / df_final['house_size']
```

In [ ]:
```
# Calculating the total number of rooms by adding the number of bedrooms and bathrooms, and addi
df_final['total_rooms'] = df_final['bed'] + df_final['bath']
```

In [ ]:
```
# Calculating the current year
current_year = pd.Timestamp.now().year

# Calculating the age of the house by subtracting the year of the previous sale date from the cu
# Adding this calculation as a new column 'house_age' in the DataFrame
df_final['house_age'] = current_year - pd.DatetimeIndex(df_final['prev_sold_date']).year
```

In [ ]:
```
# Removing rows with missing values in the 'price_per_sqft', 'total_rooms', or 'house_age' columns
df_final = df_final.dropna(subset=['price_per_sqft', 'total_rooms', 'house_age'])
df_final
```

Out[ ]:

	brokered_by	status	price	bed	bath	acre_lot	street	city	state	zip_code	house_
0	103378.0	for_sale	105000.0	3.0	2.0	0.12	1962661.0	Adjuntas	Puerto Rico	601.0	9
1	52707.0	for_sale	80000.0	4.0	2.0	0.08	1902874.0	Adjuntas	Puerto Rico	601.0	15
2	103379.0	for_sale	67000.0	2.0	1.0	0.15	1404990.0	Juana Diaz	Puerto Rico	795.0	7
3	31239.0	for_sale	145000.0	4.0	2.0	0.10	1947675.0	Ponce	Puerto Rico	731.0	18
4	34632.0	for_sale	65000.0	6.0	2.0	0.05	331151.0	Mayaguez	Puerto Rico	680.0	17
...	...	...	...	...	...	...	...	...	...	...	
29458	15366.0	for_sale	274000.0	3.0	2.0	0.23	893368.0	East Haven	Connecticut	6512.0	14
29459	22611.0	for_sale	279900.0	2.0	2.0	0.11	654312.0	East Haven	Connecticut	6512.0	12
29460	107953.0	for_sale	249900.0	4.0	2.0	0.17	1584359.0	East Haven	Connecticut	6512.0	13
29461	39954.0	for_sale	449900.0	5.0	3.0	0.11	1725145.0	East Haven	Connecticut	6512.0	20
29462	90155.0	for_sale	419900.0	4.0	3.0	0.40	1743526.0	New	Massachusetts	nan	17

29463 rows × 15 columns

Step 3: Encoding

In [ ]:
```
#Encoding Categorical Variables
categorical_columns = ['brokered_by', 'status', 'street', 'city', 'state', 'zip_code']

# Initialize the LabelEncoder
label_encoder = LabelEncoder()
for col in categorical_columns:
    if col in df_final.columns:
        df_final[col] = label_encoder.fit_transform(df_final[col].astype(str))

# Ensuring all columns are numeric
numeric_columns = df_final.select_dtypes(include=[np.number]).columns.tolist()
```

Step 4: Feature Scaling

In [ ]:
```
# Initializing a StandardScaler to standardize the features by removing the mean and scaling to
scaler = StandardScaler()

# Applying the scaler to the numerical columns, including 'price_per_sqft', 'total_rooms', and '
# This scales the features to have a mean of 0 and a standard deviation of 1
scaled_features = scaler.fit_transform(df_final[numeric_columns + ['price_per_sqft', 'total_room

# Creating a new DataFrame with the scaled features, using the same column names as the original
df_scaled = pd.DataFrame(scaled_features, columns=numeric_columns + ['price_per_sqft', 'total_ro

# Adding categorical columns from df_final to the scaled DataFrame df_scaled
```

```
for col in categorical_columns:
    df_scaled[col] = df_final[col].values
```

Step 5: Data Visualization

In [ ]:
```
# Checking for remaining NaNs
print(df_scaled.isnull().sum())
```

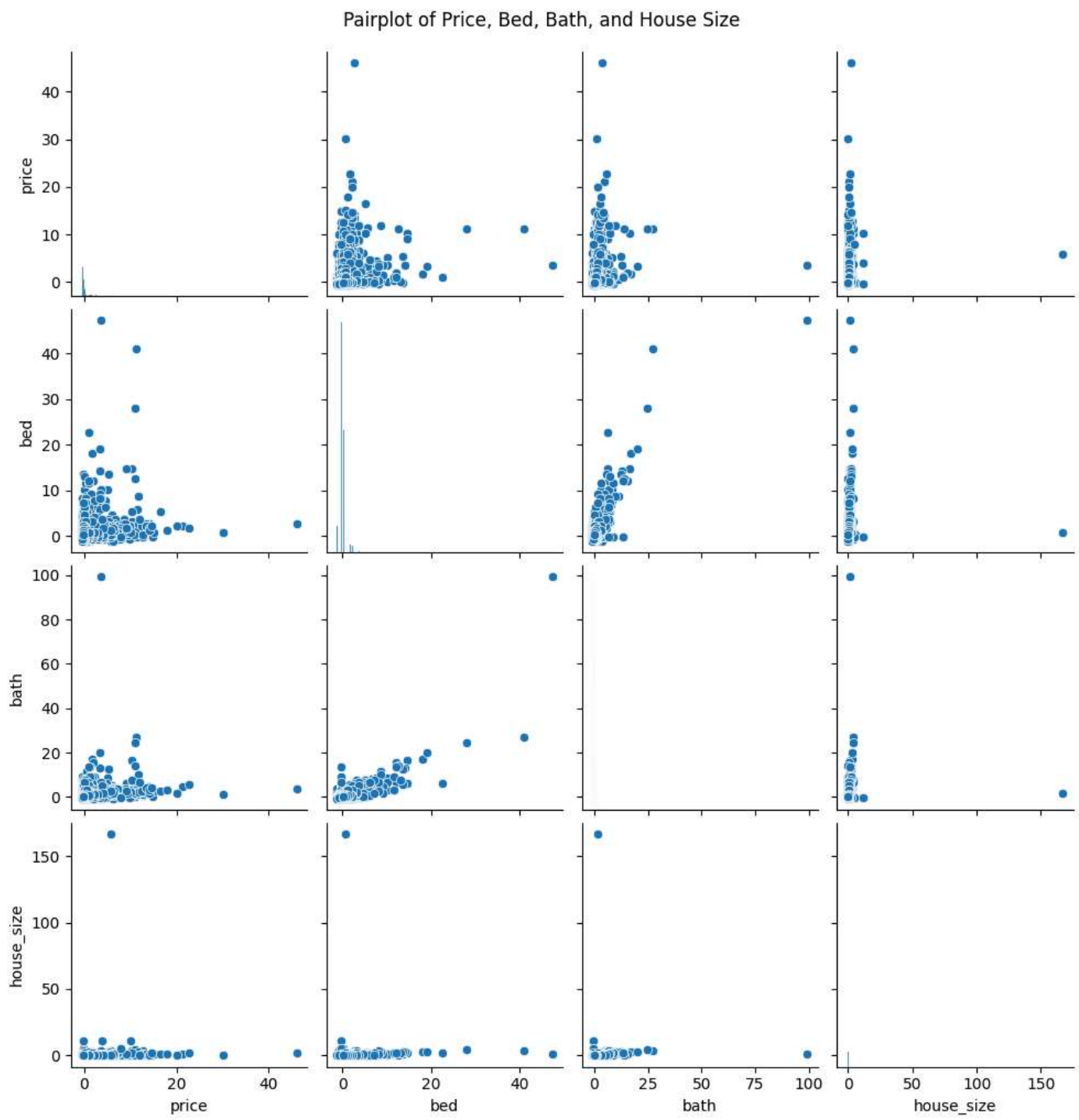
```
brokered_by       0
status            0
price             0
bed               0
bath              0
acre_lot          0
street            0
city              0
state             0
zip_code          0
house_size        0
price_per_sqft    0
total_rooms       0
house_age         0
price_per_sqft    0
total_rooms       0
house_age         0
dtype: int64
```

In [ ]:
```
# Dropping duplicate columns
df_scaled = df_scaled.loc[:,~df_scaled.columns.duplicated()]

# Checking the columns again to ensure duplicates are removed
print(df_scaled.columns)
```

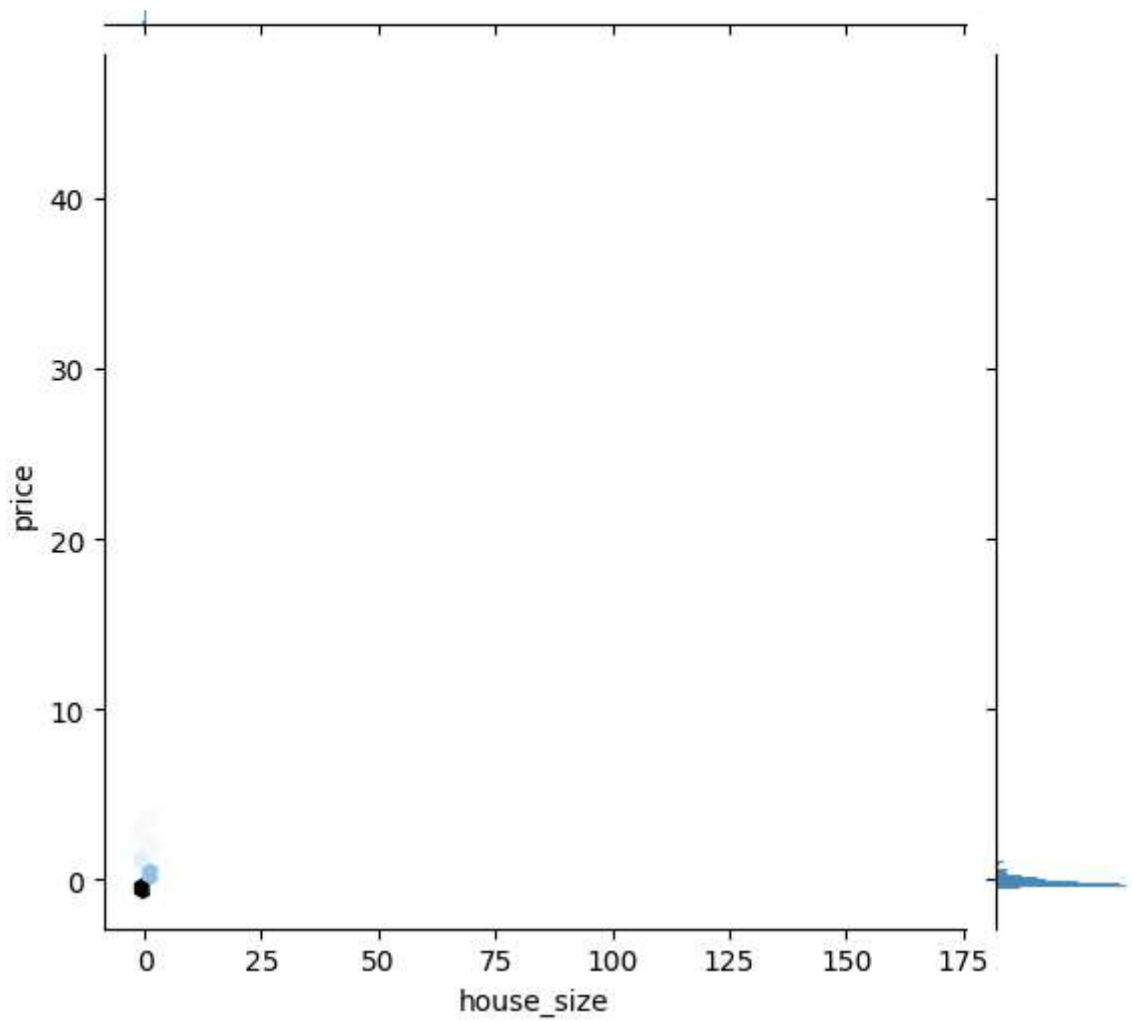
```
Index(['brokered_by', 'status', 'price', 'bed', 'bath', 'acre_lot', 'street',
       'city', 'state', 'zip_code', 'house_size', 'price_per_sqft',
       'total_rooms', 'house_age'],
      dtype='object')
```

In [ ]:
```
# Pair plot
sns.pairplot(df_scaled[['price', 'bed', 'bath', 'house_size']])
plt.suptitle('Pairplot of Price, Bed, Bath, and House Size', y=1.02)
plt.show()
```



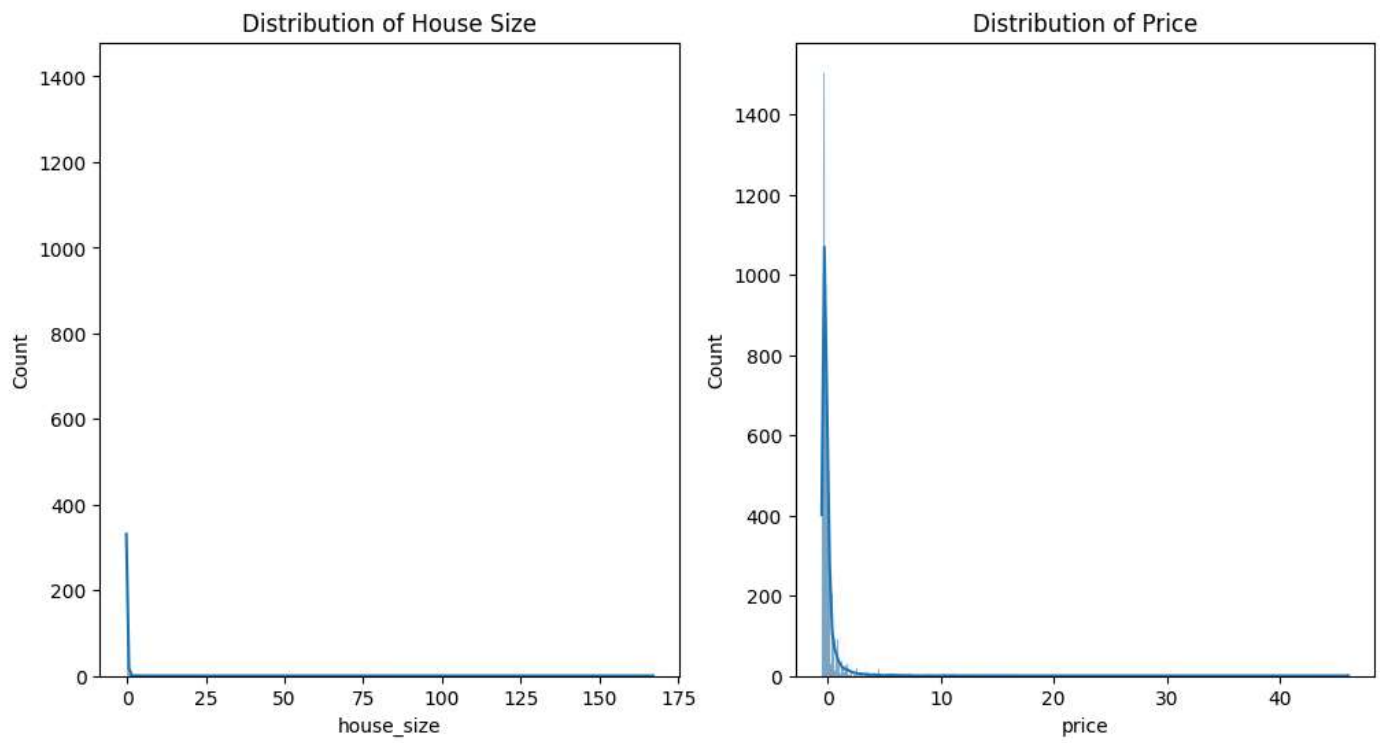
- The pair plot suggests positive relationships between price and other features like bed, bath, and house_size.
- Price vs. House Size: Positive correlation where larger houses generally have higher prices.
- Bed vs. Bath: Strong clustering, indicating houses with more bedrooms tend to have more bathrooms.
- Most data points are clustered at lower values for all features, indicating that the majority of houses are smaller and lower-priced.

In [ ]:
```
# Joint plot of House size Vs. Price
sns.jointplot(x='house_size', y='price', data=df_scaled, kind='hex')
plt.suptitle('Joint Plot of House Size vs. Price', y=1.02)
plt.show()
```



We can see the data points are highly concentrated near the origin (low values) for both house_size and price.

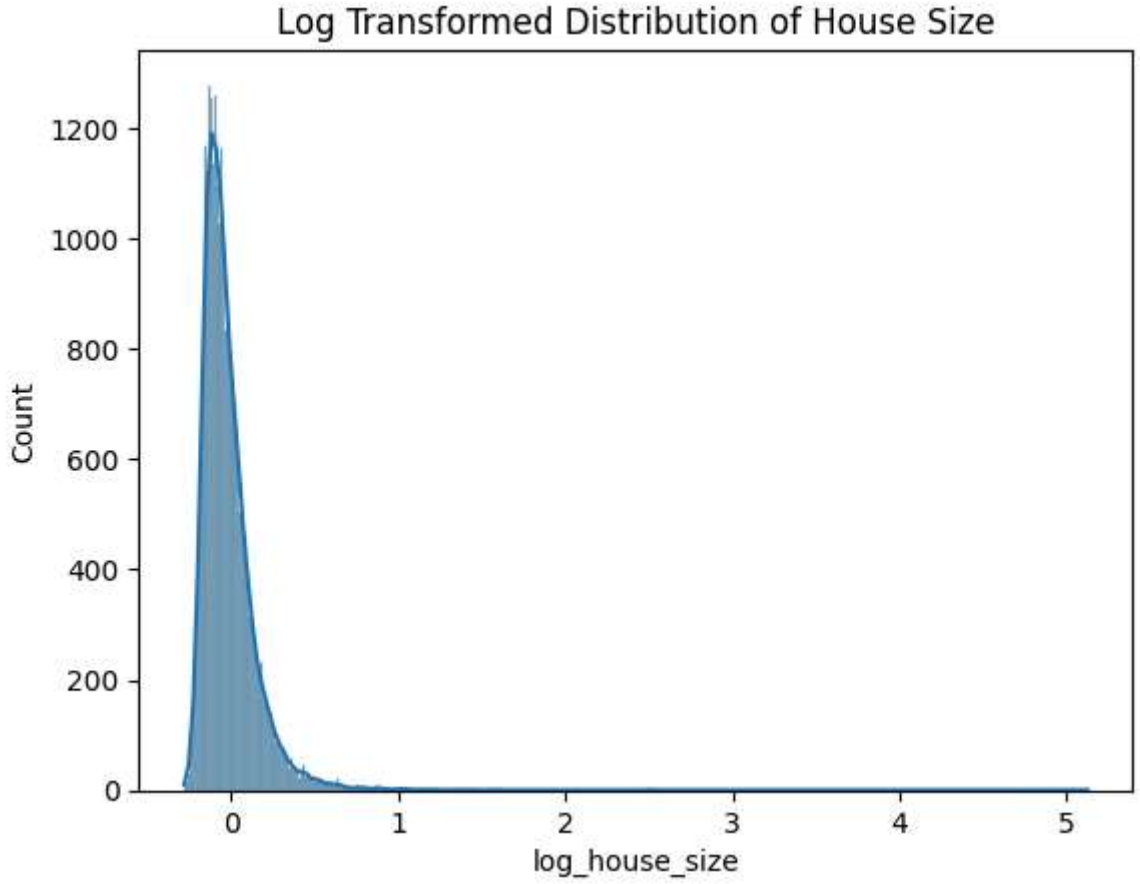
In [ ]:
```
# Plotting the distribution of house_size and price
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
sns.histplot(df_scaled['house_size'], ax=axes[0], kde=True)
axes[0].set_title('Distribution of House Size')
sns.histplot(df_scaled['price'], ax=axes[1], kde=True)
axes[1].set_title('Distribution of Price')
plt.show()
```

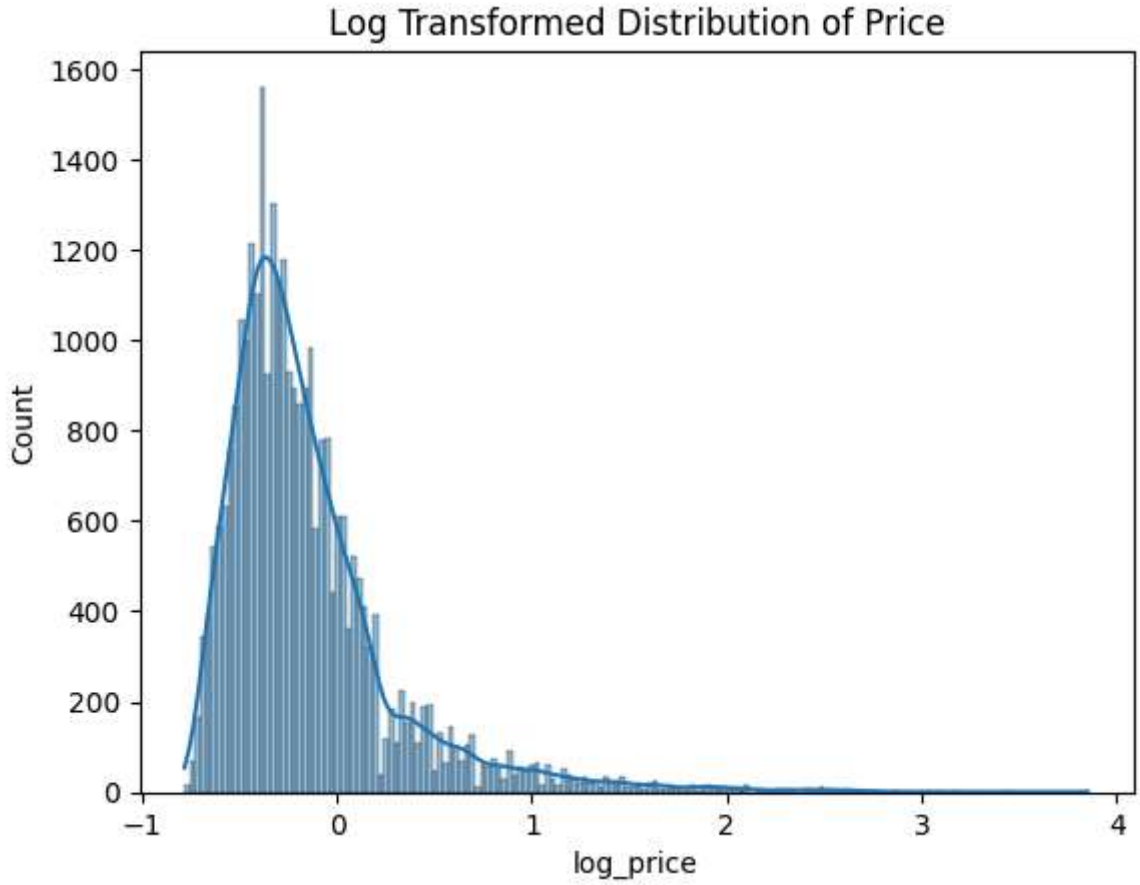


Insights obtained from the distribution:

- Skewness: Both distributions are right-skewed, indicating that the majority of houses are relatively small and low-priced, with a few high-value outliers.
- Outliers: The presence of extreme outliers affects the overall distribution, making it difficult to analyze the bulk of the data without further transformation or filtering.
- Data Quality: The skewness and outliers suggest that data preprocessing steps, such as log transformation or robust scaling, may be necessary to improve the analysis and visualization.

In [ ]:
```
df_scaled['log_house_size'] = np.log1p(df_scaled['house_size'])
df_scaled['log_price'] = np.log1p(df_scaled['price'])
sns.histplot(df_scaled['log_house_size'], kde=True).set_title('Log Transformed Distribution of H
plt.show()
sns.histplot(df_scaled['log_price'], kde=True).set_title('Log Transformed Distribution of Price'
plt.show()
```

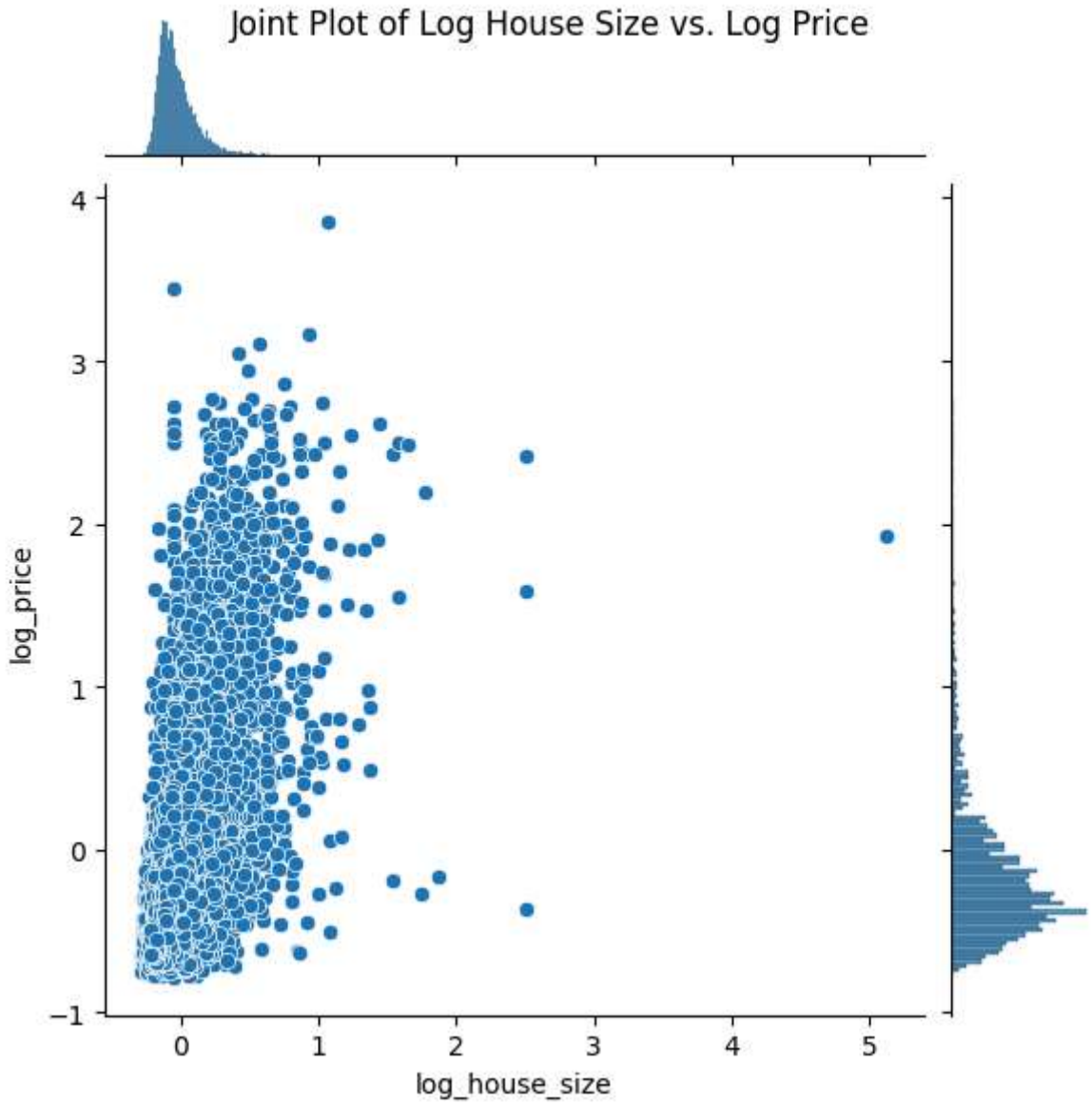




In [ ]:
```
# Log scale transformation of house size and log price
df_scaled['log_house_size'] = np.log(df_scaled['house_size'] + 1)  # Adding 1 to avoid log(0)
df_scaled['log_price'] = np.log(df_scaled['price'] + 1)

# Joint plot with log scale
sns.jointplot(x='log_house_size', y='log_price', data=df_scaled, kind='scatter')
```

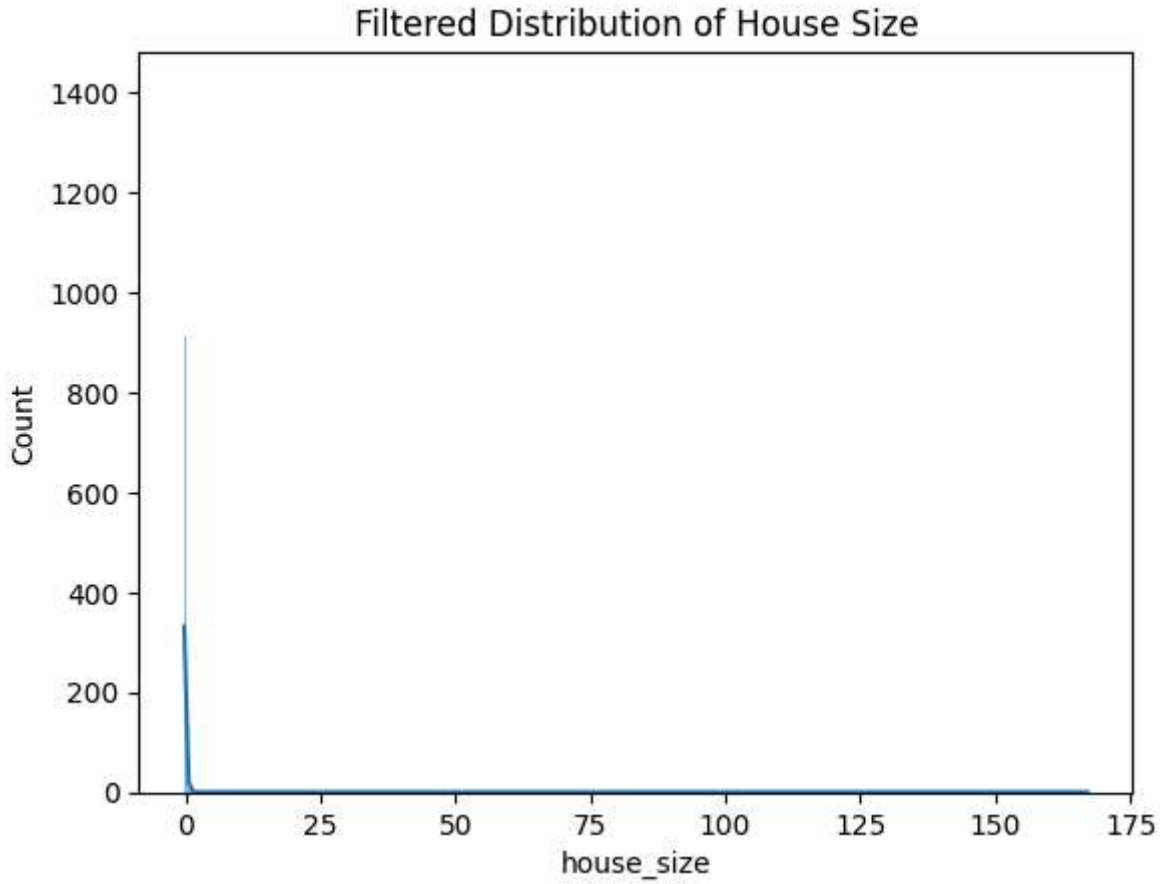
```
plt.suptitle('Joint Plot of Log House Size vs. Log Price')
plt.show()
```

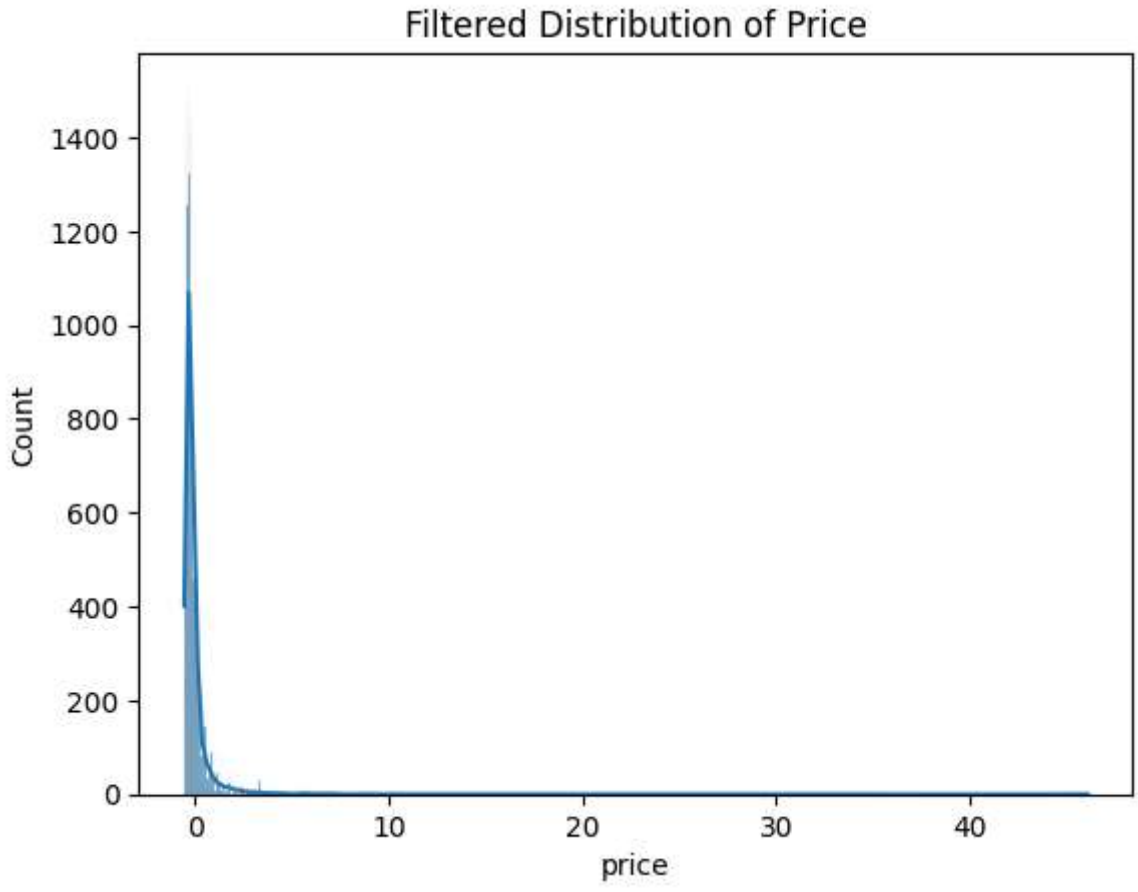


There is a positive relationship between log_house_size and log_price, indicating that as house size increases, price generally increases as well. Most data points are clustered in the middle range of both log_house_size and log_price, suggesting that typical houses have moderate sizes and prices. The spread of data points suggests variability in house prices even for houses of similar sizes, reflecting factors beyond size that influence the house price.

In [ ]:
```
# Filtering the dataset to remove extreme outliers beyond a certain threshold
threshold_price = df_scaled['price'].quantile(0.99)
threshold_house_size = df_scaled['house_size'].quantile(0.99)
df_final = df_final[(df_scaled['price'] < threshold_price) & (df_scaled['house_size'] < threshol
```

In [ ]:
```
sns.histplot(df_scaled['house_size'], kde=True).set_title('Filtered Distribution of House Size')
plt.show()
sns.histplot(df_scaled['price'], kde=True).set_title('Filtered Distribution of Price')
plt.show()
```

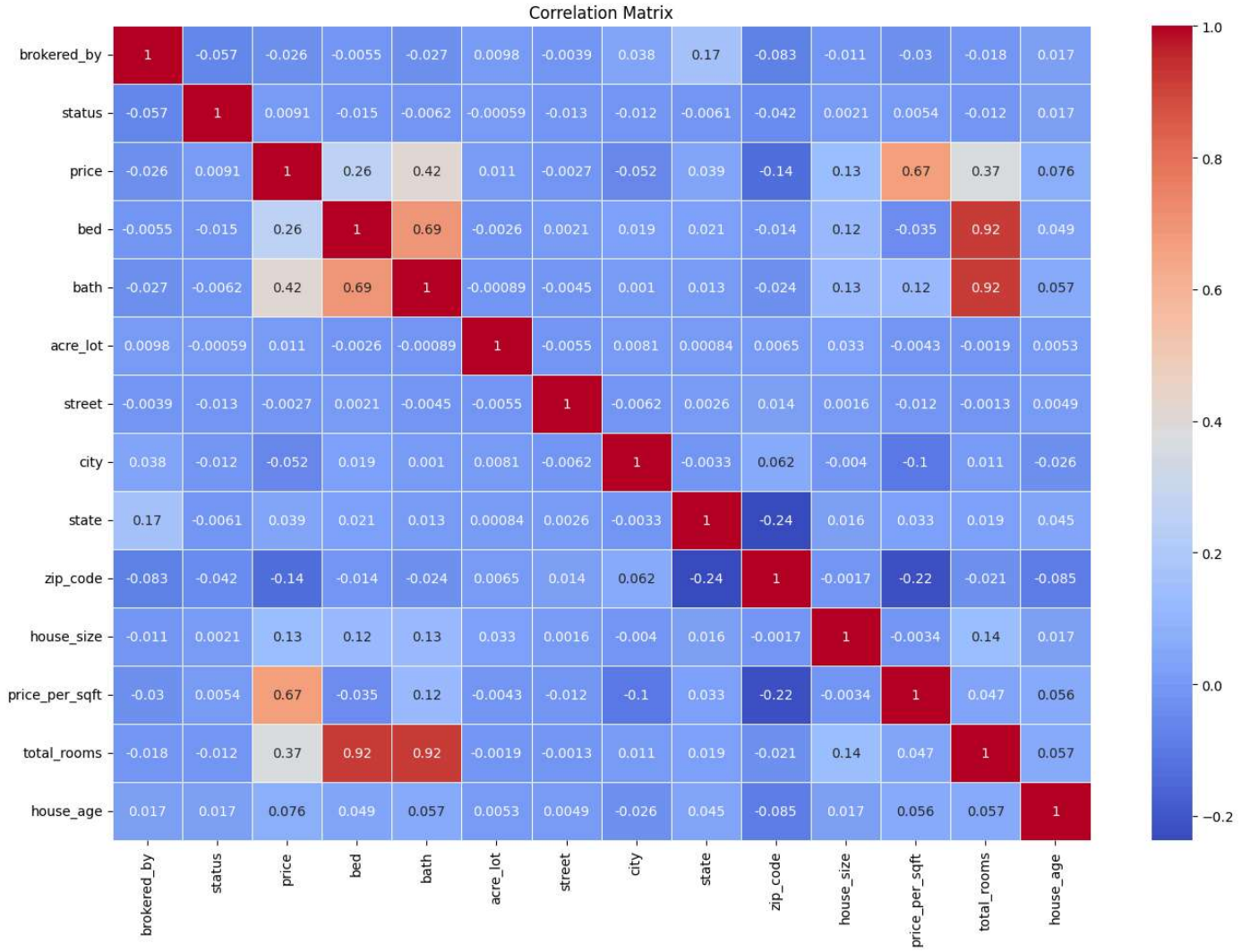




In [ ]:
```
# Droping the columns log_house_price and log_price
df_scaled.drop(['log_house_size', 'log_price'], axis=1, inplace=True)
```

In [ ]:
```
# Creating a correlation heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(df_scaled.corr(), annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix')
```

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



Insights:

Price and Other Variables:

- Positive Correlations:
  - `bath (0.18)`, `house_size (0.20)`, `log_house_size (0.31)`
- Weaker Correlations:
  - `bed (0.11)`, `acre_lot (0.0072)` Higher prices are associated with more bathrooms, larger house sizes, and larger lot sizes, though these relationships are not very strong.

Bed and Bath:

- Strong Correlation (0.62): Houses with more bedrooms generally have more bathrooms.

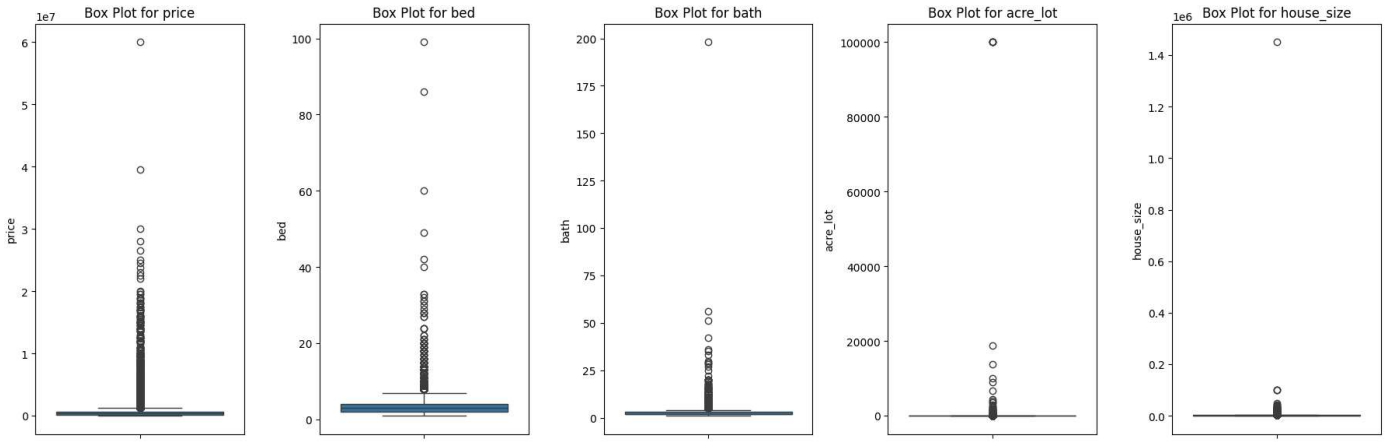
Prev Sold Date and Status:

- Strong Correlation (0.69): Indicates a significant relationship between the date a house was previously sold and its current status.

Price Determinants: Bathrooms, house size, and house size show some correlation with price, while other factors have a minor impact.

In [ ]:
```
# List of numerical columns to check for outliers
numerical_columns = ['price', 'bed', 'bath', 'acre_lot', 'house_size']

# Creating box plots for each numerical column
fig, axes = plt.subplots(nrows=1, ncols=len(numerical_columns), figsize=(18, 6))
for ax, var in zip(axes, numerical_columns):
    sns.boxplot(y=df[var], ax=ax)
    ax.set_title(f'Box Plot for {var}')
plt.tight_layout()
plt.show()
```



This ensures that the DataFrame only contains complete data for these important metrics related to price_per_sqft, total_rooms and house_age.

- Outliers: The presence of extreme outliers in the data suggests that some properties are priced much higher per square foot compared to others.
- Price Distribution by Status: The price per square foot varies across different statuses, with category 0 having the most variability and outliers.

In [ ]:
```
# Checking for any NaN values
print(df_scaled.isnull().sum())
```

```
brokered_by       0
status            0
price             0
bed               0
bath              0
acre_lot          0
street            0
city              0
state             0
zip_code          0
house_size        0
price_per_sqft    0
total_rooms       0
house_age         0
dtype: int64
```

Step 6: Splitting Data

In [ ]:
```
# Separating the features (X) and the target variable (y) from the scaled DataFrame
X = df_scaled.drop('price', axis=1)  # Dropping the 'price' column to use as features
y = df_scaled['price']  # Using the 'price' column as the target variable

# Splitting the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Using 80% of the data for training and 20% for testing
# random_state=42 ensures reproducibility of the split
```

Step 7: Model Training

In [ ]:
```
# Defining a dictionary of models to train

models = {
    'Linear Regression': LinearRegression(),
    'Decision Tree': DecisionTreeRegressor(random_state=42),
    'Random Forest': RandomForestRegressor(random_state=42)
}

# Initializing a dictionary to store the results
results = {}

# Training each model and evaluating its performance
for model_name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    results[model_name] = {'MSE': mse, 'R2': r2}
    print(f"{model_name} - MSE: {mse}, R2: {r2}")
```

```
Linear Regression - MSE: 0.2962257098728835, R2: 0.683234342761962
Decision Tree - MSE: 0.07063026380047233, R2: 0.9244723155756689
Random Forest - MSE: 0.07377665962882783, R2: 0.9211077523075861
```

Based on the results, the Decision Tree and Random forest models have the best performance, with the Decision Tree model slightly outperforming the Random Forest model in terms of Mean Squared Error (MSE) and R-squared (R^2) values.

In [ ]:
```
# Hyperparameter tuning for Decision Tree

# Using a larger portion of the dataset for tuning
X_train_sample, _, y_train_sample, _ = train_test_split(X_train, y_train, test_size=0.5, random_

# Defining an expanded parameter grid for Decision Tree
param_grid_dt = {
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}

# Initializing the GridSearchCV object for Decision Tree
grid_search_dt = GridSearchCV(DecisionTreeRegressor(random_state=42),
                              param_grid_dt,
                              cv=5,
                              scoring='neg_mean_squared_error',
                              n_jobs=-1)

# Fit the model
grid_search_dt.fit(X_train_sample, y_train_sample)

# Get the best model
best_dt_model = grid_search_dt.best_estimator_

# Predict with the best model
y_pred_best_dt = best_dt_model.predict(X_test)
mse_best_dt = mean_squared_error(y_test, y_pred_best_dt)
```

```
r2_best_dt = r2_score(y_test, y_pred_best_dt)

print(f"Best Decision Tree - MSE: {mse_best_dt}, R2: {r2_best_dt}")

# Overwriting the Decision Tree results
results['Decision Tree'] = {'MSE': mse_best_dt, 'R2': r2_best_dt}
print(f"Best Decision Tree - MSE: {mse_best_dt}, R2: {r2_best_dt}")
```

```
Best Decision Tree - MSE: 0.10200491745608913, R2: 0.8909221798021668
Best Decision Tree - MSE: 0.10200491745608913, R2: 0.8909221798021668
```

In [ ]:

```
# Hyperparameter tuning for Random Forest

# Using a smaller portion of the dataset for quicker tuning
X_train_sample, _, y_train_sample, _ = train_test_split(X_train, y_train, test_size=0.9, random_

# Defining a reduced parameter grid for Random Forest
param_dist_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

# Initializing the RandomizedSearchCV object for Random Forest
random_search_rf = RandomizedSearchCV(RandomForestRegressor(random_state=42),
                                      param_distributions=param_dist_rf,
                                      n_iter=20,  # Number of parameter settings that are sample
                                      cv=3,
                                      scoring='neg_mean_squared_error',
                                      n_jobs=-1,
                                      random_state=42)

# Fit the model
random_search_rf.fit(X_train_sample, y_train_sample)

# Get the best model
best_rf_model = random_search_rf.best_estimator_

# Predict with the best model
y_pred_best_rf = best_rf_model.predict(X_test)
mse_best_rf = mean_squared_error(y_test, y_pred_best_rf)
r2_best_rf = r2_score(y_test, y_pred_best_rf)

print(f"Best Random Forest - MSE: {mse_best_rf}, R2: {r2_best_rf}")

# Overwriting the Random Forest results
results['Random Forest'] = {'MSE': mse_best_rf, 'R2': r2_best_rf}
print(f"Best Random Forest - MSE: {mse_best_rf}, R2: {r2_best_rf}")
```

```
Best Random Forest - MSE: 0.2772785832560268, R2: 0.7034952411766757
Best Random Forest - MSE: 0.2772785832560268, R2: 0.7034952411766757
```

In [ ]:

```
# ANN
#Preprocessing and scaling data
X_train = np.nan_to_num(X_train)
X_test = np.nan_to_num(X_test)
y_test = np.nan_to_num(y_test)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Defining the ANN model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
```

```python
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

def build_ann(input_shape):
    model = Sequential()
    model.add(Dense(64, activation='relu', input_shape=(input_shape,)))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer=Adam(learning_rate=0.0001), loss='mse')
    return model

# Training the ANN model
input_shape = X_train_scaled.shape[1]
ann_model = build_ann(input_shape)
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history = ann_model.fit(X_train_scaled, y_train, validation_split=0.2, epochs=200, batch_size=12

# Evaluating the ANN model
y_pred_ann = ann_model.predict(X_test_scaled)
mse_ann = mean_squared_error(y_test, y_pred_ann)
r2_ann = r2_score(y_test, y_pred_ann)
print(f"ANN Model - MSE: {mse_ann}, R2: {r2_ann}")
```

```
Epoch 1/200
148/148 [==============================] - 2s 5ms/step - loss: 1.0191 - val_loss: 0.7787
Epoch 2/200
148/148 [==============================] - 1s 4ms/step - loss: 0.8851 - val_loss: 0.7036
Epoch 3/200
148/148 [==============================] - 1s 4ms/step - loss: 0.7855 - val_loss: 0.6769
Epoch 4/200
148/148 [==============================] - 1s 4ms/step - loss: 0.7140 - val_loss: 0.6535
Epoch 5/200
148/148 [==============================] - 1s 5ms/step - loss: 0.6472 - val_loss: 0.6576
Epoch 6/200
148/148 [==============================] - 1s 6ms/step - loss: 0.5749 - val_loss: 0.6587
Epoch 7/200
148/148 [==============================] - 1s 6ms/step - loss: 0.5290 - val_loss: 0.6366
Epoch 8/200
148/148 [==============================] - 1s 5ms/step - loss: 0.5152 - val_loss: 0.6287
Epoch 9/200
148/148 [==============================] - 1s 5ms/step - loss: 0.4804 - val_loss: 0.6558
Epoch 10/200
148/148 [==============================] - 1s 6ms/step - loss: 0.4635 - val_loss: 0.6423
Epoch 11/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4257 - val_loss: 0.6525
Epoch 12/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4572 - val_loss: 0.5885
Epoch 13/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4057 - val_loss: 0.5862
Epoch 14/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4010 - val_loss: 0.5918
Epoch 15/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4108 - val_loss: 0.5404
Epoch 16/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4228 - val_loss: 0.5209
Epoch 17/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4133 - val_loss: 0.5296
Epoch 18/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3884 - val_loss: 0.5433
Epoch 19/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3555 - val_loss: 0.4927
Epoch 20/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3868 - val_loss: 0.4898
Epoch 21/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3875 - val_loss: 0.5393
Epoch 22/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3752 - val_loss: 0.5309
Epoch 23/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3369 - val_loss: 0.5241
Epoch 24/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3622 - val_loss: 0.5522
Epoch 25/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3832 - val_loss: 0.4953
Epoch 26/200
148/148 [==============================] - 1s 4ms/step - loss: 0.4119 - val_loss: 0.4429
Epoch 27/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3657 - val_loss: 0.4319
Epoch 28/200
148/148 [==============================] - 1s 5ms/step - loss: 0.3718 - val_loss: 0.4155
Epoch 29/200
148/148 [==============================] - 1s 5ms/step - loss: 0.3207 - val_loss: 0.4317
Epoch 30/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3347 - val_loss: 0.4568
Epoch 31/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3336 - val_loss: 0.4428
Epoch 32/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3541 - val_loss: 0.4619
```

```
Epoch 33/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3564 - val_loss: 0.4022
Epoch 34/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3246 - val_loss: 0.4156
Epoch 35/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3498 - val_loss: 0.3910
Epoch 36/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3754 - val_loss: 0.3420
Epoch 37/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3225 - val_loss: 0.3647
Epoch 38/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3513 - val_loss: 0.3737
Epoch 39/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3499 - val_loss: 0.3306
Epoch 40/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3078 - val_loss: 0.3233
Epoch 41/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3562 - val_loss: 0.3108
Epoch 42/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3101 - val_loss: 0.3233
Epoch 43/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3057 - val_loss: 0.3049
Epoch 44/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2887 - val_loss: 0.3112
Epoch 45/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2926 - val_loss: 0.3020
Epoch 46/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3102 - val_loss: 0.3057
Epoch 47/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3045 - val_loss: 0.3110
Epoch 48/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3254 - val_loss: 0.2813
Epoch 49/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3303 - val_loss: 0.3074
Epoch 50/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3315 - val_loss: 0.2689
Epoch 51/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3189 - val_loss: 0.2690
Epoch 52/200
148/148 [==============================] - 1s 5ms/step - loss: 0.3020 - val_loss: 0.3016
Epoch 53/200
148/148 [==============================] - 1s 5ms/step - loss: 0.2988 - val_loss: 0.2911
Epoch 54/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3198 - val_loss: 0.2727
Epoch 55/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3136 - val_loss: 0.2818
Epoch 56/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3123 - val_loss: 0.2872
Epoch 57/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3042 - val_loss: 0.2623
Epoch 58/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3137 - val_loss: 0.2466
Epoch 59/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3158 - val_loss: 0.2479
Epoch 60/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3092 - val_loss: 0.2525
Epoch 61/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3162 - val_loss: 0.2360
Epoch 62/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2885 - val_loss: 0.2623
Epoch 63/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2994 - val_loss: 0.2621
Epoch 64/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3166 - val_loss: 0.2338
```

```
Epoch 65/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2883 - val_loss: 0.2553
Epoch 66/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2752 - val_loss: 0.2516
Epoch 67/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2973 - val_loss: 0.2316
Epoch 68/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2829 - val_loss: 0.2489
Epoch 69/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3163 - val_loss: 0.2468
Epoch 70/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3152 - val_loss: 0.2303
Epoch 71/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3060 - val_loss: 0.2344
Epoch 72/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3044 - val_loss: 0.2192
Epoch 73/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2869 - val_loss: 0.2293
Epoch 74/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2998 - val_loss: 0.2088
Epoch 75/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3013 - val_loss: 0.2346
Epoch 76/200
148/148 [==============================] - 1s 5ms/step - loss: 0.3129 - val_loss: 0.2193
Epoch 77/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3077 - val_loss: 0.2147
Epoch 78/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2916 - val_loss: 0.2138
Epoch 79/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2849 - val_loss: 0.1944
Epoch 80/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2880 - val_loss: 0.1976
Epoch 81/200
148/148 [==============================] - 1s 5ms/step - loss: 0.2686 - val_loss: 0.1896
Epoch 82/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2762 - val_loss: 0.1858
Epoch 83/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2889 - val_loss: 0.1691
Epoch 84/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2587 - val_loss: 0.1775
Epoch 85/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2853 - val_loss: 0.1712
Epoch 86/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2966 - val_loss: 0.1602
Epoch 87/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2657 - val_loss: 0.1534
Epoch 88/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2671 - val_loss: 0.1558
Epoch 89/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2331 - val_loss: 0.1542
Epoch 90/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2802 - val_loss: 0.1458
Epoch 91/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2570 - val_loss: 0.1454
Epoch 92/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2637 - val_loss: 0.1441
Epoch 93/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2459 - val_loss: 0.1504
Epoch 94/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2743 - val_loss: 0.1444
Epoch 95/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2684 - val_loss: 0.1414
Epoch 96/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2307 - val_loss: 0.1380
```

```
Epoch 97/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2813 - val_loss: 0.1361
Epoch 98/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2669 - val_loss: 0.1357
Epoch 99/200
148/148 [==============================] - 1s 5ms/step - loss: 0.2465 - val_loss: 0.1339
Epoch 100/200
148/148 [==============================] - 1s 5ms/step - loss: 0.2538 - val_loss: 0.1337
Epoch 101/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2398 - val_loss: 0.1319
Epoch 102/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2457 - val_loss: 0.1308
Epoch 103/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2546 - val_loss: 0.1299
Epoch 104/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2701 - val_loss: 0.1284
Epoch 105/200
148/148 [==============================] - 1s 8ms/step - loss: 0.2340 - val_loss: 0.1264
Epoch 106/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2597 - val_loss: 0.1263
Epoch 107/200
148/148 [==============================] - 1s 5ms/step - loss: 0.2485 - val_loss: 0.1252
Epoch 108/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2062 - val_loss: 0.1228
Epoch 109/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2307 - val_loss: 0.1234
Epoch 110/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2259 - val_loss: 0.1224
Epoch 111/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2292 - val_loss: 0.1212
Epoch 112/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2471 - val_loss: 0.1203
Epoch 113/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2630 - val_loss: 0.1197
Epoch 114/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2483 - val_loss: 0.1184
Epoch 115/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2321 - val_loss: 0.1177
Epoch 116/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2538 - val_loss: 0.1182
Epoch 117/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2247 - val_loss: 0.1172
Epoch 118/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2136 - val_loss: 0.1172
Epoch 119/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2777 - val_loss: 0.1170
Epoch 120/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2413 - val_loss: 0.1152
Epoch 121/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2058 - val_loss: 0.1135
Epoch 122/200
148/148 [==============================] - 1s 5ms/step - loss: 0.4150 - val_loss: 0.1181
Epoch 123/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2281 - val_loss: 0.1154
Epoch 124/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2408 - val_loss: 0.1135
Epoch 125/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2083 - val_loss: 0.1135
Epoch 126/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1905 - val_loss: 0.1105
Epoch 127/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2104 - val_loss: 0.1098
Epoch 128/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2407 - val_loss: 0.1092
```

```
Epoch 129/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2097 - val_loss: 0.1093
Epoch 130/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1958 - val_loss: 0.1089
Epoch 131/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2056 - val_loss: 0.1093
Epoch 132/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2181 - val_loss: 0.1091
Epoch 133/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2196 - val_loss: 0.1086
Epoch 134/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2242 - val_loss: 0.1074
Epoch 135/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2114 - val_loss: 0.1050
Epoch 136/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2159 - val_loss: 0.1047
Epoch 137/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2030 - val_loss: 0.1039
Epoch 138/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2242 - val_loss: 0.1029
Epoch 139/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1915 - val_loss: 0.1027
Epoch 140/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2036 - val_loss: 0.1014
Epoch 141/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2013 - val_loss: 0.1009
Epoch 142/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1851 - val_loss: 0.1007
Epoch 143/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2162 - val_loss: 0.0998
Epoch 144/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2226 - val_loss: 0.1005
Epoch 145/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1951 - val_loss: 0.0976
Epoch 146/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2761 - val_loss: 0.1032
Epoch 147/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2037 - val_loss: 0.0998
Epoch 148/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1982 - val_loss: 0.0986
Epoch 149/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1702 - val_loss: 0.0963
Epoch 150/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2330 - val_loss: 0.0972
Epoch 151/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2217 - val_loss: 0.0982
Epoch 152/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1831 - val_loss: 0.0962
Epoch 153/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2175 - val_loss: 0.0942
Epoch 154/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1871 - val_loss: 0.0936
Epoch 155/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2266 - val_loss: 0.0942
Epoch 156/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2225 - val_loss: 0.0965
Epoch 157/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1974 - val_loss: 0.0960
Epoch 158/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1759 - val_loss: 0.0933
Epoch 159/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1854 - val_loss: 0.0929
Epoch 160/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1810 - val_loss: 0.0926
```

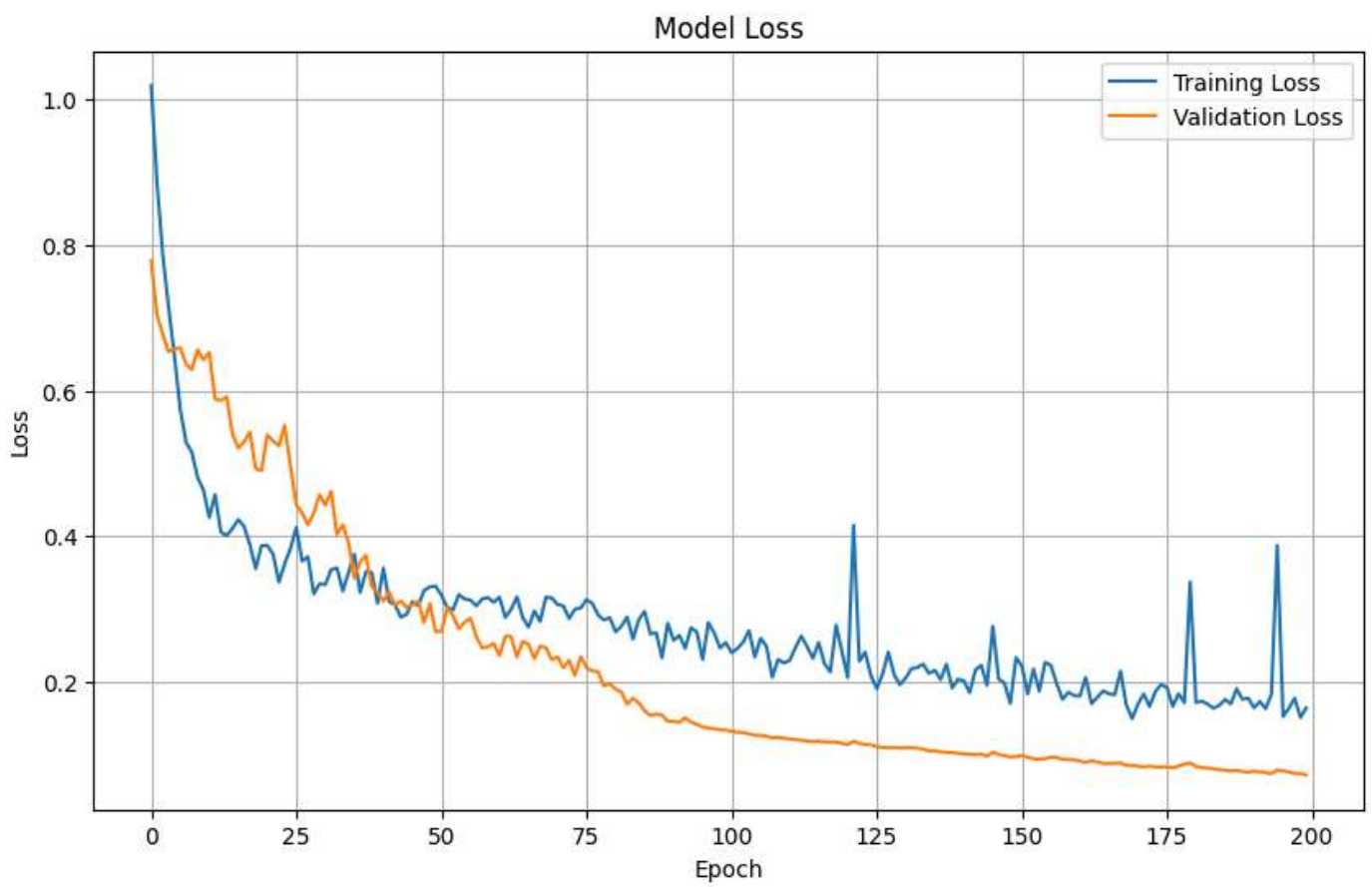
```
Epoch 161/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1804 - val_loss: 0.0911
Epoch 162/200
148/148 [==============================] - 1s 4ms/step - loss: 0.2058 - val_loss: 0.0891
Epoch 163/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1701 - val_loss: 0.0913
Epoch 164/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1788 - val_loss: 0.0899
Epoch 165/200
148/148 [==============================] - 1s 5ms/step - loss: 0.1873 - val_loss: 0.0876
Epoch 166/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1835 - val_loss: 0.0876
Epoch 167/200
148/148 [==============================] - 1s 5ms/step - loss: 0.1821 - val_loss: 0.0878
Epoch 168/200
148/148 [==============================] - 1s 6ms/step - loss: 0.2145 - val_loss: 0.0886
Epoch 169/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1679 - val_loss: 0.0851
Epoch 170/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1493 - val_loss: 0.0851
Epoch 171/200
148/148 [==============================] - 1s 5ms/step - loss: 0.1694 - val_loss: 0.0839
Epoch 172/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1832 - val_loss: 0.0829
Epoch 173/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1660 - val_loss: 0.0841
Epoch 174/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1867 - val_loss: 0.0828
Epoch 175/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1958 - val_loss: 0.0829
Epoch 176/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1923 - val_loss: 0.0829
Epoch 177/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1660 - val_loss: 0.0817
Epoch 178/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1834 - val_loss: 0.0842
Epoch 179/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1713 - val_loss: 0.0866
Epoch 180/200
148/148 [==============================] - 1s 4ms/step - loss: 0.3369 - val_loss: 0.0883
Epoch 181/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1716 - val_loss: 0.0834
Epoch 182/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1734 - val_loss: 0.0821
Epoch 183/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1693 - val_loss: 0.0815
Epoch 184/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1638 - val_loss: 0.0805
Epoch 185/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1673 - val_loss: 0.0790
Epoch 186/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1756 - val_loss: 0.0784
Epoch 187/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1695 - val_loss: 0.0775
Epoch 188/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1904 - val_loss: 0.0782
Epoch 189/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1760 - val_loss: 0.0766
Epoch 190/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1774 - val_loss: 0.0758
Epoch 191/200
148/148 [==============================] - 1s 5ms/step - loss: 0.1642 - val_loss: 0.0771
Epoch 192/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1729 - val_loss: 0.0763
```

```
Epoch 193/200
148/148 [==============================] - 1s 5ms/step - loss: 0.1632 - val_loss: 0.0750
Epoch 194/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1826 - val_loss: 0.0738
Epoch 195/200
148/148 [==============================] - 1s 6ms/step - loss: 0.3873 - val_loss: 0.0789
Epoch 196/200
148/148 [==============================] - 1s 6ms/step - loss: 0.1523 - val_loss: 0.0776
Epoch 197/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1635 - val_loss: 0.0760
Epoch 198/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1773 - val_loss: 0.0740
Epoch 199/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1509 - val_loss: 0.0738
Epoch 200/200
148/148 [==============================] - 1s 4ms/step - loss: 0.1639 - val_loss: 0.0718
185/185 [==============================] - 0s 2ms/step
ANN Model - MSE: 0.09470581452463987, R2: 0.8987273940704381
```

Insights from ANN Training:

- The training process for the Artificial Neural Network (ANN) model showed a significant decrease in loss over the epochs, indicating effective learning.
- The validation loss consistently decreased, demonstrating that the model generalized well to unseen data.
- The final Mean Squared Error (MSE) was 0.09, and the R^2 score was 0.898, indicating a good performance of the ANN model in predicting house prices.

In [ ]:
```python
# Plotting the Training and Validation Loss over Epochs
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



The model's training and validation losses both decrease over epochs without significant divergence, indicating effective learning and good generalization without overfitting.

In [ ]:
```
# Adding ANN model results to the results dictionary
results['ANN Model'] = {'MSE': mse_ann, 'R2': r2_ann}
```

In [ ]:
```
# Determining the best model based on MSE
best_model_name = min(results, key=lambda k: results[k]['MSE'])
best_model_performance = results[best_model_name]

print(f"\nBest Model: {best_model_name}")
print(f"MSE: {best_model_performance['MSE']}")
print(f"R2: {best_model_performance['R2']}")
```

```
Best Model: ANN Model
MSE: 0.09470581452463987
R2: 0.8987273940704381
```

In [ ]:
```
# Saving the best model
if best_model_name == 'ANN Model':
    best_model_to_save = ann_model
    best_model_to_save.save('best_model_ann.h5')
else:
    best_model_to_save = best_rf_model
    with open('best_model.pkl', 'wb') as f:
        pickle.dump(best_model_to_save, f)

print("Best model saved successfully!")
```

```
Best model saved successfully!
```

In [ ]:
```
# Saving the scaler used in the model

with open('scaler.pkl', 'wb') as f:
```

```
    pickle.dump(scaler, f)
    print("Scaler saved successfully!")
```

```
Scaler saved successfully!
```

In [1]:

```
import nbformat
from nbconvert import HTMLExporter
from google.colab import drive

# Mounting Google Drive
drive.mount('/content/drive')

# Defining the paths
notebook_file_path = '/content/drive/MyDrive/Colab Notebooks/houseprice.ipynb'
output_html_file_path = '/content/drive/MyDrive/Colab Notebooks/houseprice.html'

try:
    # Loading the notebook
    with open(notebook_file_path) as f:
        nb = nbformat.read(f, as_version=4)

    # Converting the notebook to HTML
    html_exporter = HTMLExporter()
    (body, resources) = html_exporter.from_notebook_node(nb)

    # Writing the HTML output to a file
    with open(output_html_file_path, 'w') as f:
        f.write(body)

    print(f"Notebook converted to HTML and saved to {output_html_file_path}")

except FileNotFoundError as e:
    print(f"Error: {e}")
    print("Please check the notebook file path and ensure the file exists.")
```

```
Mounted at /content/drive
Notebook converted to HTML and saved to /content/drive/MyDrive/Colab Notebooks/houseprice.html
```