COMP 1930 Report 1: Data Preprocessing and Model Selection

Data preprocessing is an essential stage in the machine learning process that significantly impacts model performance and prediction accuracy. Inconsistencies, missing values, and irrelevant features are typical in raw data, which might impair model performance. To optimize the effectiveness of machine learning algorithms, clean, well-structured, and correctly converted data is guaranteed by effective preprocessing.

The data preparation procedures used for the London Property Listings dataset are described in this paper. The report is organized as follows: First, the preparation methods used to clean and modify the dataset are covered. Then, the model selection for the regression problem is justified by considering both environmental consequences and technical performance. Lastly, challenges and limitations are highlighted, and potential solutions are provided at the end of the report.

Importing the libraries

In [282…
```python
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import json
import re
```

In [283…
```python
# Load the dataset
data = pd.read_csv("London Property Listings Dataset.csv")
```

Data exploration

In [283…
```python
# Preview the first few rows
data.head()
```

Out[283…

	Price	Property Type	Bedrooms	Bathrooms	Size	Postcode	Area	Price_Category	Area_Avg_Price
0	330000.0	Apartment	1.0	1.0	518.000000	E14	Eastern	Low	1.001684e+06
1	340000.0	Flat	1.0	1.0	887.498269	E14	Eastern	Low	1.001684e+06
2	340000.0	Apartment	1.0	1.0	934.569040	E14	Eastern	Low	1.001684e+06
3	340000.0	Flat	1.0	1.0	887.498269	E14	Eastern	Low	1.001684e+06
4	340000.0	Flat	1.0	1.0	388.000000	SW20	South Western	Low	1.516724e+06

In [283…
```python
# Preview the last few rows
data.tail()
```

Out[283…

	Price	Property Type	Bedrooms	Bathrooms	Size	Postcode	Area	Price_Category	Area_Avg_Price
29532	795000.0	Flat	3.0	2.0	840.000000	SW20	South Western	Medium	1.516724e+06
29533	795000.0	Flat	2.0	1.0	887.498269	E14	Eastern	Medium	1.001684e+06
29534	795000.0	Flat	2.0	2.0	753.000000	SE1	South Eastern	Medium	6.921048e+05
29535	795000.0	Flat	2.0	2.0	980.000000	SW11	South Western	Medium	1.516724e+06
29536	795000.0	Semi-Detached	3.0	1.0	2183.543103	N14	Northern	Medium	8.312952e+05

In [283…
```python
# Check column types and missing values
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29537 entries, 0 to 29536
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Price           29537 non-null  float64
 1   Property Type   29537 non-null  object
 2   Bedrooms        29537 non-null  float64
 3   Bathrooms       29537 non-null  float64
 4   Size            29537 non-null  float64
 5   Postcode        29537 non-null  object
 6   Area            29537 non-null  object
 7   Price_Category  29537 non-null  object
 8   Area_Avg_Price  29537 non-null  float64
dtypes: float64(5), object(4)
memory usage: 2.0+ MB
```

In [294…
```
# Counting the number of missing values in each column.
# None of the columns in the dataset had any missing values. Thus, there is no need to drop any missing values.
data.isnull().sum()
```

Out[294…
```
Price             0
Property_Type     0
Bedrooms          0
Bathrooms         0
Size              0
Area              0
Price_Category    0
Area_Avg_Price    0
dtype: int64
```

In [283…
```
# Display basic statistics
data.describe()
```

Out[283…

	Price	Bedrooms	Bathrooms	Size	Area_Avg_Price
count	2.953700e+04	29537.000000	29537.000000	2.953700e+04	2.953700e+04
mean	9.652355e+05	2.262620	1.621322	1.201678e+03	1.151853e+06
std	8.500518e+05	1.121841	1.120325	8.814953e+03	3.156087e+05
min	6.500000e+04	1.000000	1.000000	2.600000e+01	4.187500e+05
25%	5.000000e+05	1.000000	1.000000	8.200000e+02	1.001684e+06
50%	6.900000e+05	2.000000	1.000000	8.960000e+02	1.001684e+06
75%	1.075000e+06	3.000000	2.000000	1.184000e+03	1.516724e+06
max	5.950000e+06	14.000000	144.000000	1.500000e+06	1.706839e+06

In [283…
```
# Check the types for each column
data.dtypes
```

Out[283…
```
Price             float64
Property Type      object
Bedrooms          float64
Bathrooms         float64
Size              float64
Postcode           object
Area               object
Price_Category     object
Area_Avg_Price    float64
dtype: object
```

In [283…
```
#The column "Property Type" is divided with a space, and
#it has to be replaced with an underscore to be read by the computer
data.columns = data.columns.str.replace(' ','_')
```

In [284…
```
# Display column names
data.columns
```

Out[284…
```
Index(['Price', 'Property_Type', 'Bedrooms', 'Bathrooms', 'Size', 'Postcode',
       'Area', 'Price_Category', 'Area_Avg_Price'],
      dtype='object')
```

In [284…
```
# Counting the unique values in Property_Type
data.Property_Type.value_counts()
```

Out[284…
```
Property_Type
Flat             12896
Apartment        10837
Terraced          2810
Semi-Detached     1697
House             1297
Name: count, dtype: int64
```

In [284…
```
# Counting the unique values in Postcode
data.Postcode.value_counts()
```

Out[284…
```
Postcode
E14         10907
SW11          729
W2            617
SW6           518
SE1           493
            ...
IG11            1
ABILITY         1
SE39FZ          1
FINCHLEY        1
E35             1
Name: count, Length: 215, dtype: int64
```

In [284…
```
# Counting the unique values in Price_Category
data.Price_Category.value_counts()
```

Out[284…
```
Price_Category
Medium    15077
Low        7616
High       4247
Luxury     2597
Name: count, dtype: int64
```

In [284…
```python
# Counting the unique values in Area
data.Area.value_counts()
```

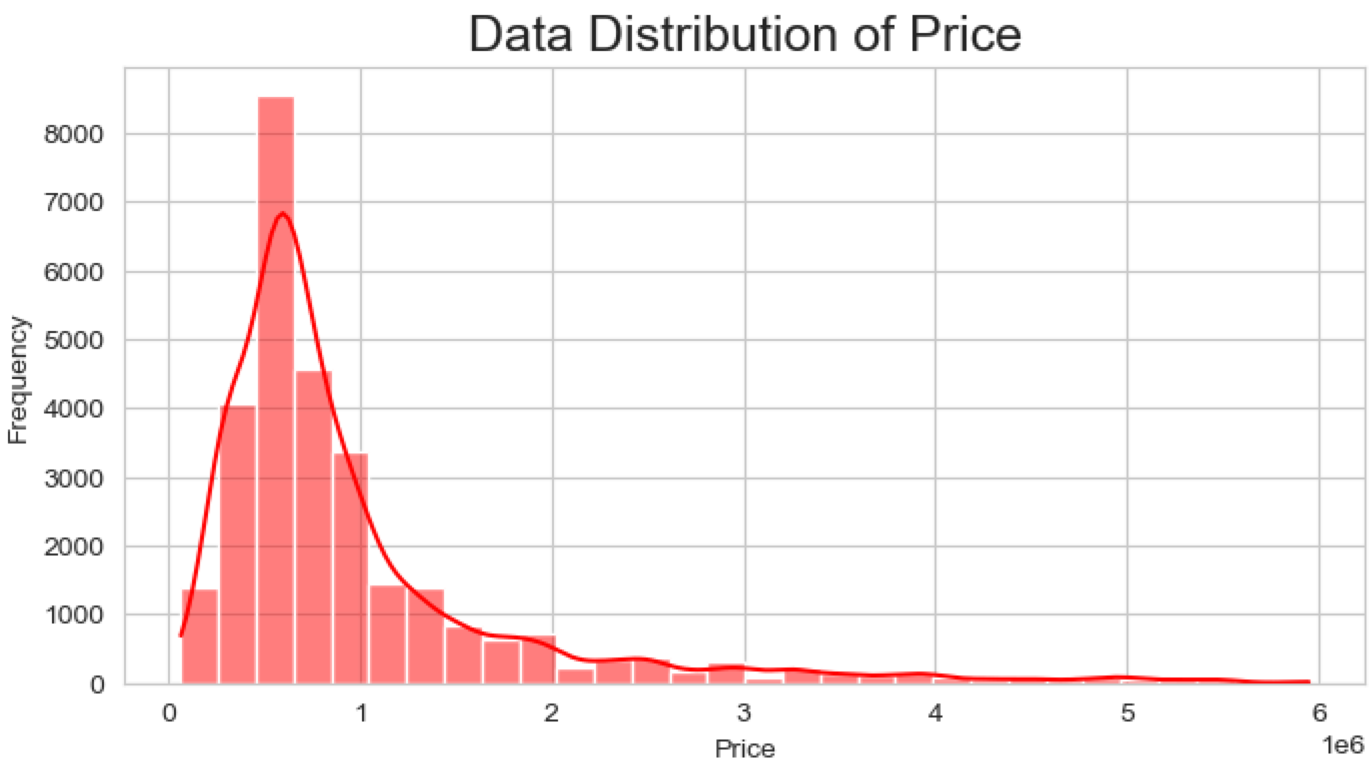
Out[284…
```
Area
Eastern                       13075
South Western                  4993
Western and Paddington         3265
South Eastern                  2813
North Western                  2376
Northern                       1686
Twickenham                      507
Eastern Central                 432
Western Central                 223
Enfield                         101
Harrow                           23
Croydon                          17
Kingston upon Thames             15
Bromley                           4
Ilford                            3
Sutton                            3
Kingston                          1
Name: count, dtype: int64
```

In [284…
```python
#Checking the unique values in the dataset
data.nunique()
```

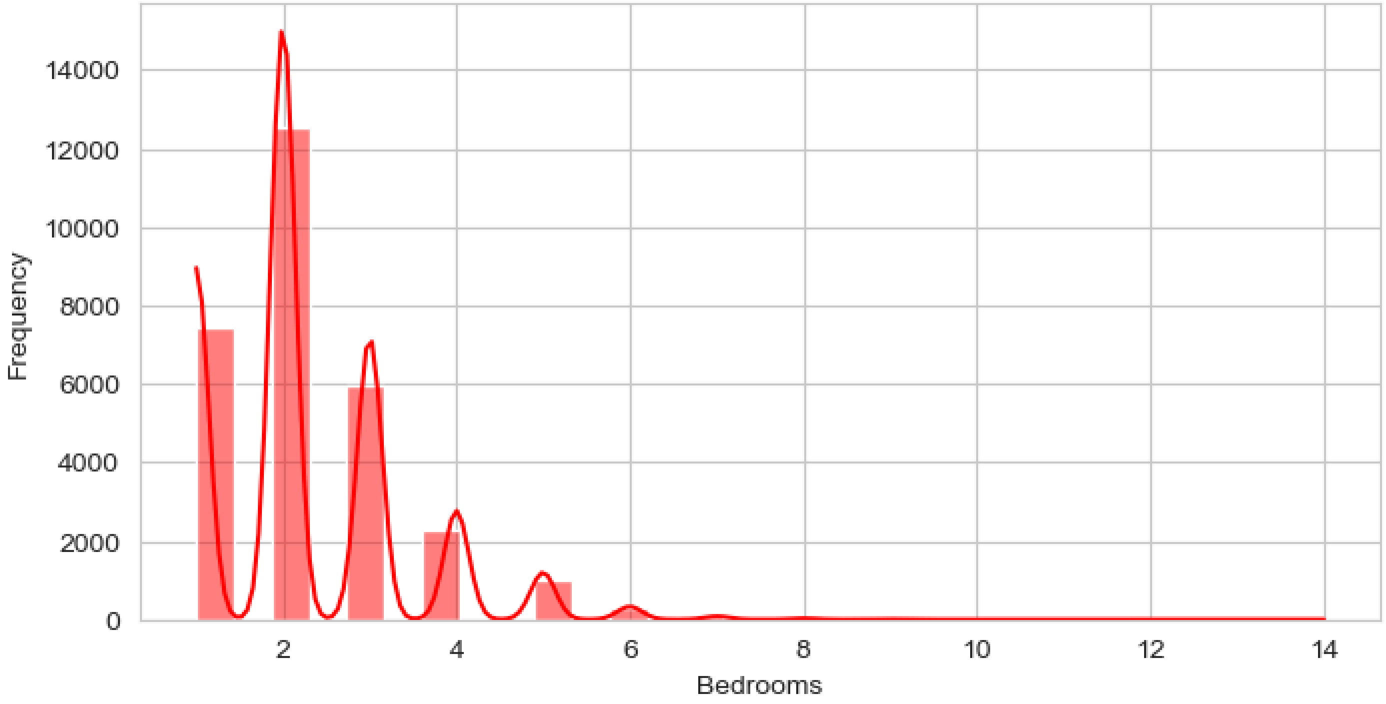
Out[284…
```
Price             1488
Property_Type        5
Bedrooms            13
Bathrooms           12
Size              2499
Postcode           215
Area                17
Price_Category       4
Area_Avg_Price      17
dtype: int64
```

Data Visualization

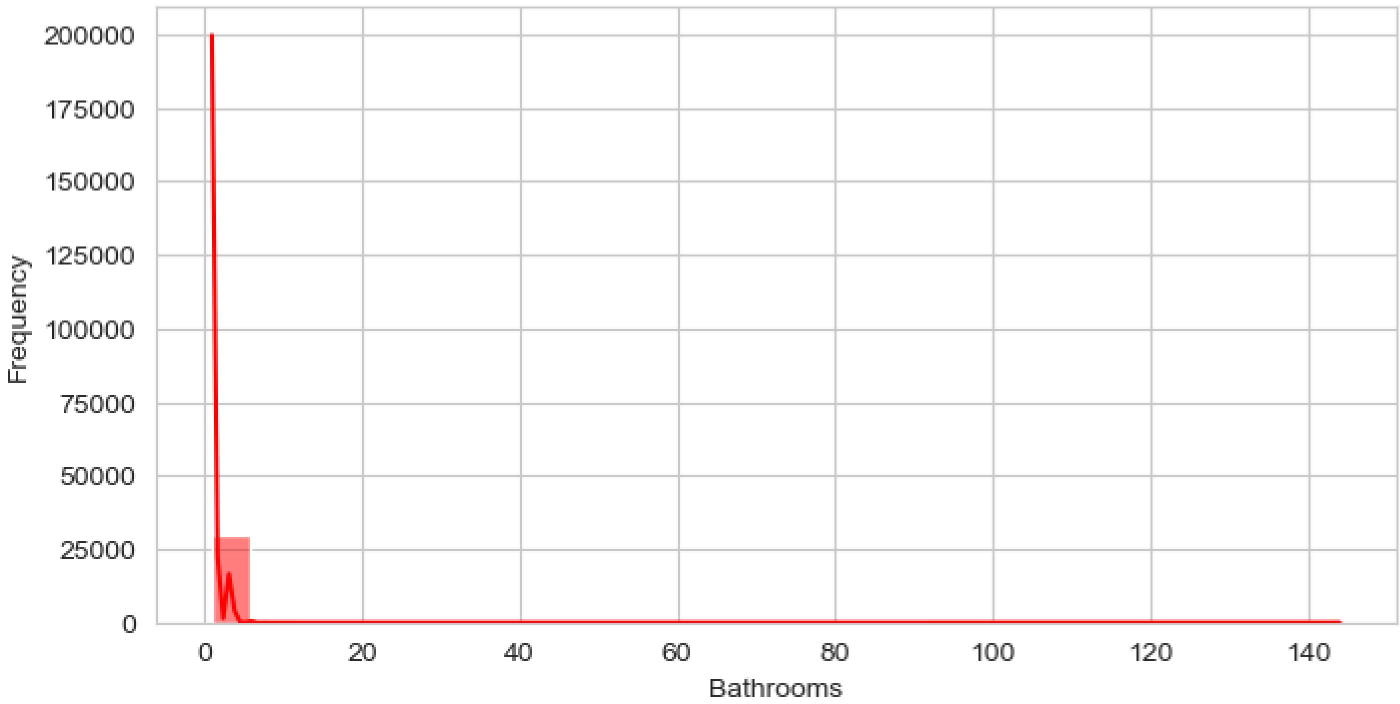
In [284…
```python
# Creating frequency distributions for all float data type columns
for col in data.select_dtypes(include=['float64']).columns:
    plt.figure(figsize=(8,4))
    sns.histplot(data[col], bins = 30, color = 'red', kde= True)
    plt.title(f"Data Distribution of {col}", fontsize = 18)
    plt.ylabel("Frequency")
    plt.xlabel(col)
    plt.show()
```



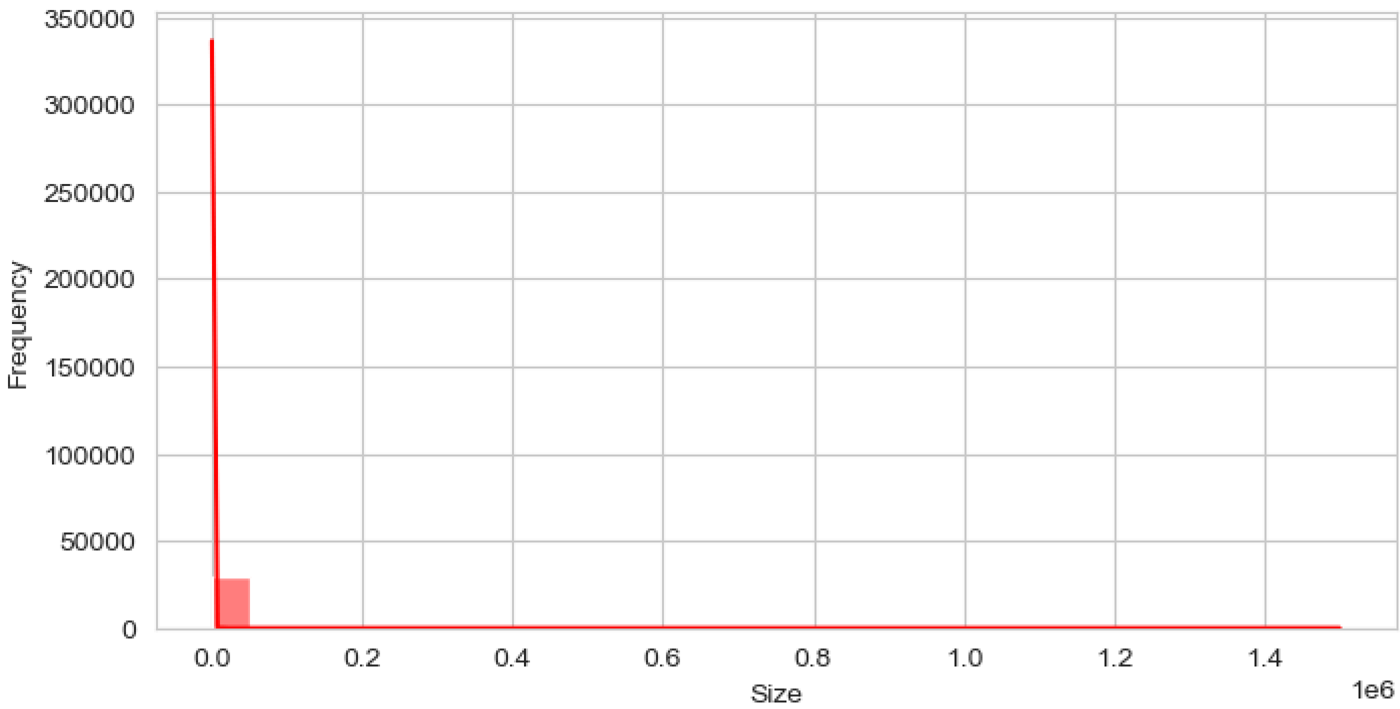
Data Distribution of Bedrooms



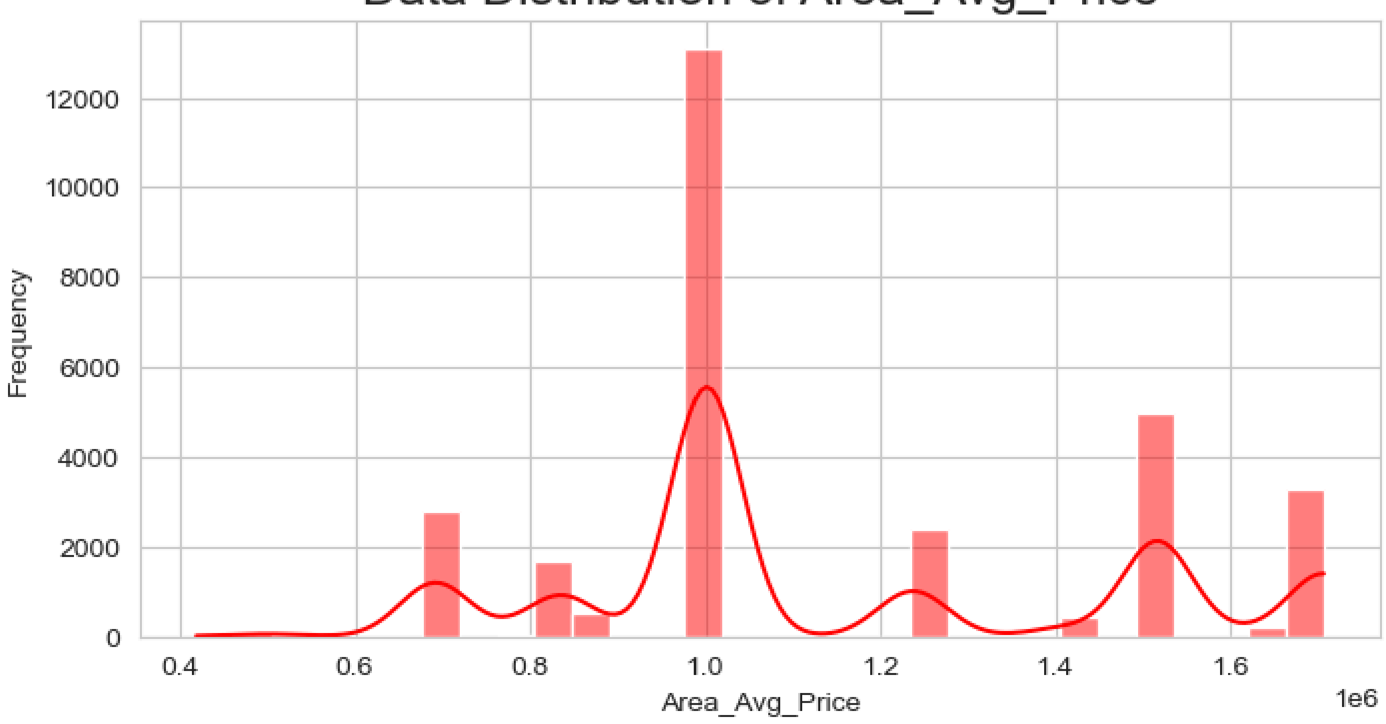
Data Distribution of Bathrooms



Data Distribution of Size



Data Distribution of Area_Avg_Price



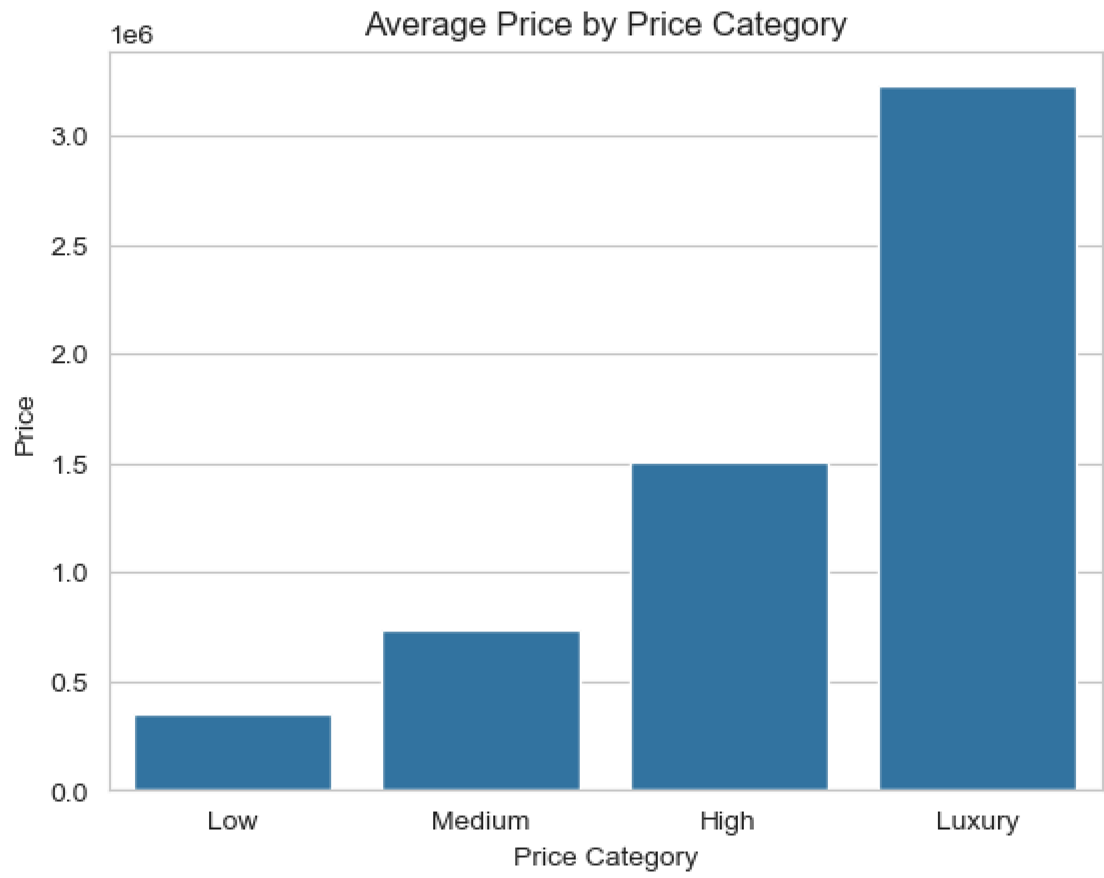
In [284…

```
# Observed an outlier and removed it
data.drop(data[data["Bathrooms"] == 144].index, inplace=True)
```

In [284…

```
# Plotted Price by Price Category
sns.barplot(x="Price_Category", y = 'Price',ci=None, data=data)
plt.title("Average Price by Price Category")
plt.xlabel("Price Category")
plt.ylabel("Price")
plt.show()
```

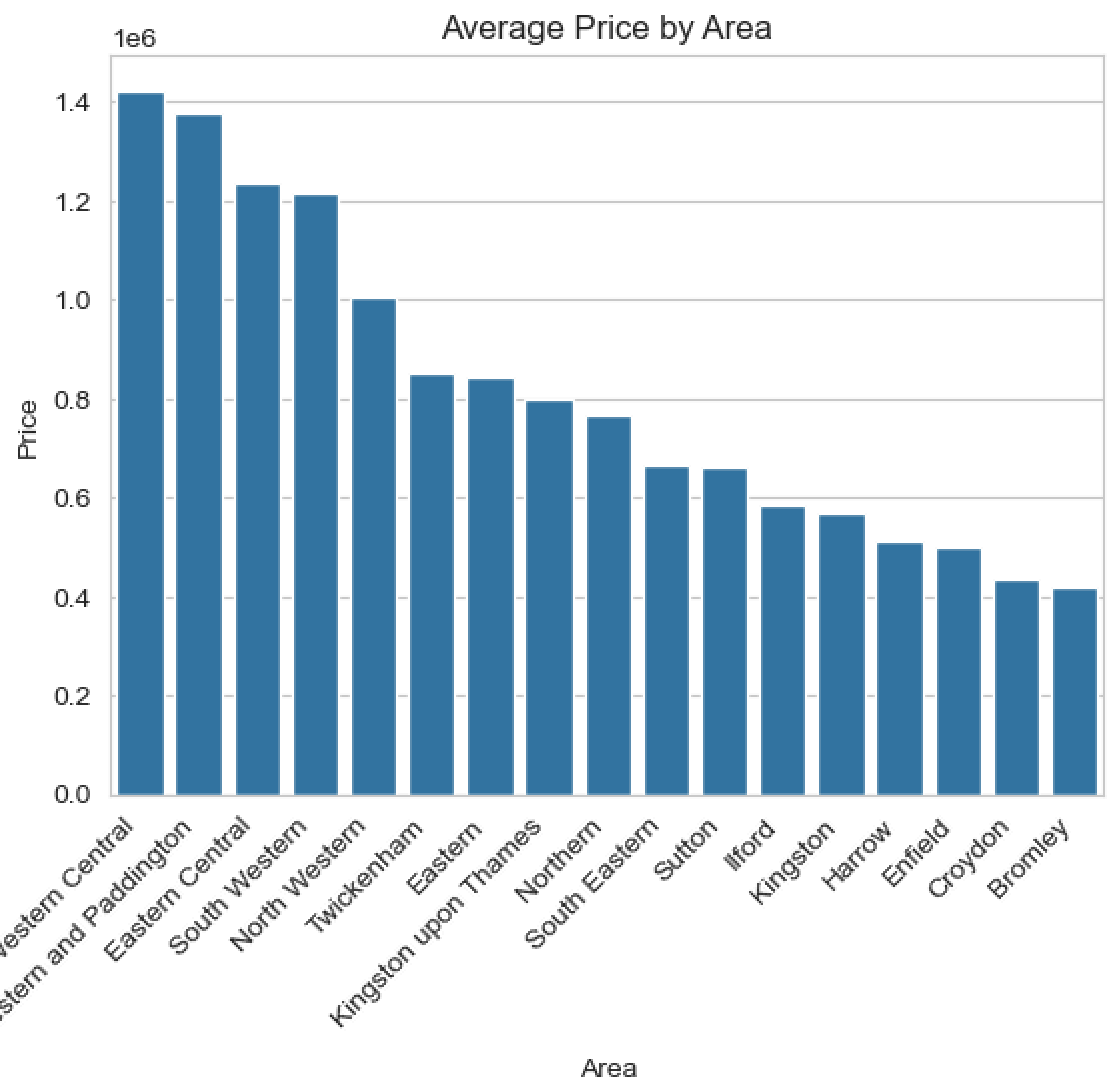
```
/var/folders/03/6nkjsrk54q3f89x4qy6tnl280000gn/T/ipykernel_68979/817929064.py:2: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  sns.barplot(x="Price_Category", y = 'Price',ci=None, data=data)
```



In [285…

```
# Plotted Average Price by Area
sorted_data = data.groupby("Area", as_index=False)["Price"].mean().sort_values(by="Price", ascending=False)
sns.barplot(x="Area", y= "Price", data=sorted_data,ci=None)
plt.title("Average Price by Area")
plt.xlabel("Area")
plt.ylabel("Price")
plt.xticks(rotation=45, ha='right')
plt.show()
```

```
/var/folders/03/6nkjsrk54q3f89x4qy6tnl280000gn/T/ipykernel_68979/3129991148.py:3: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  sns.barplot(x="Area", y= "Price", data=sorted_data,ci=None)
```

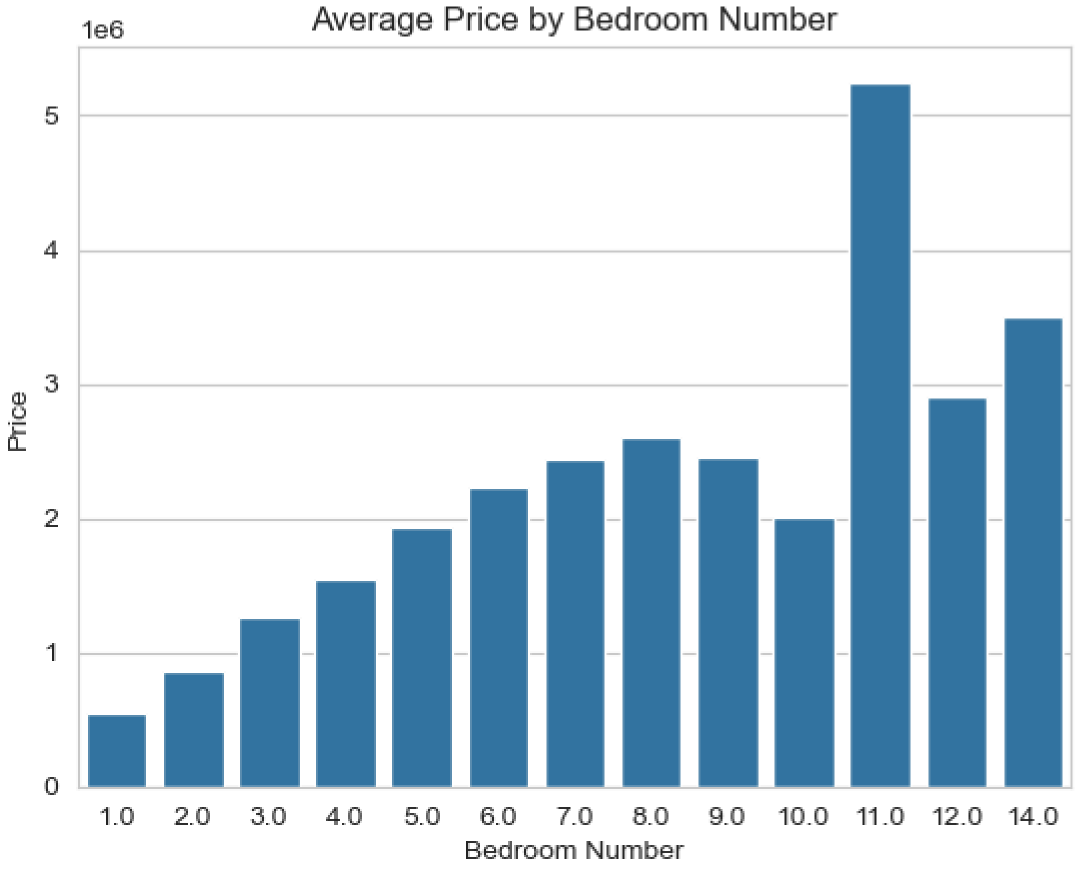


In [285…

```
# Plotted Average Price by Number of Bedrooms
sns.barplot(x="Bedrooms", y="Price", data=data, ci=None)
```

```
plt.title("Average Price by Bedroom Number")
plt.xlabel("Bedroom Number")
plt.ylabel("Price")
plt.show()
```

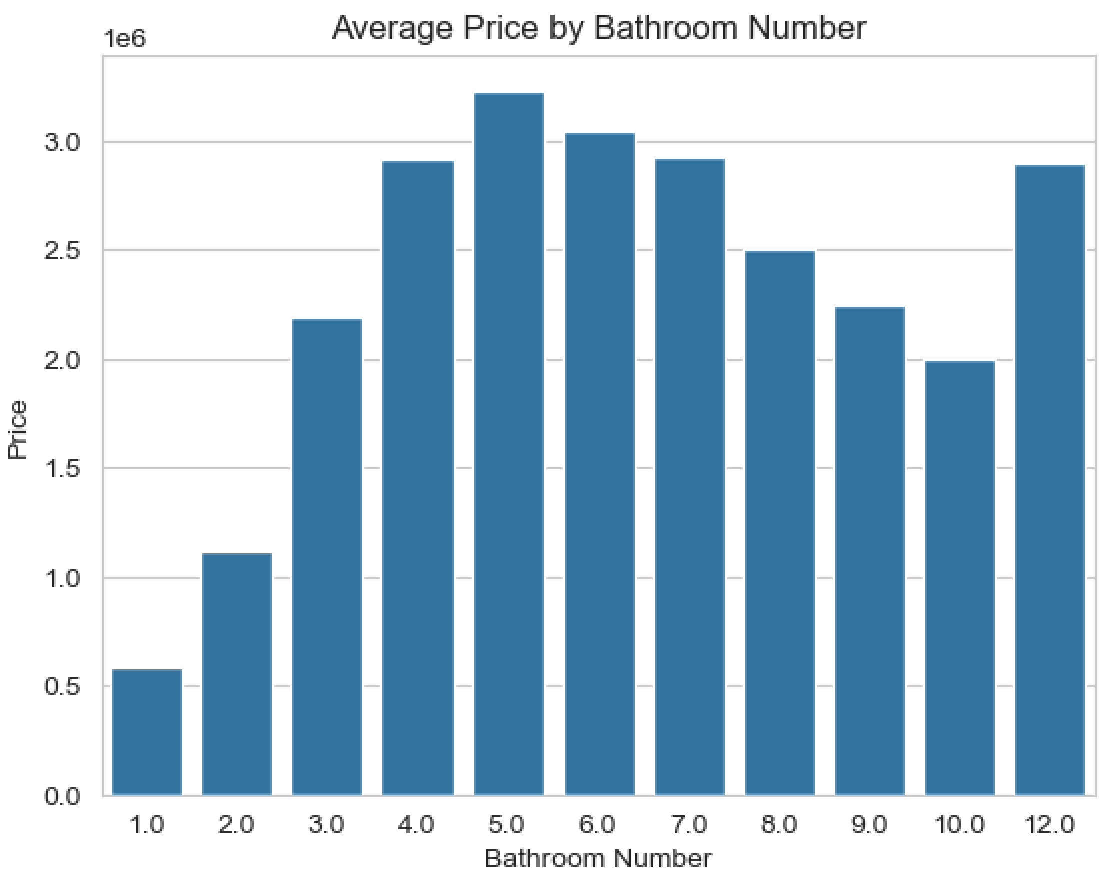
```
/var/folders/03/6nkjsrk54q3f89x4qy6tnl280000gn/T/ipykernel_68979/592497334.py:2: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  sns.barplot(x="Bedrooms", y="Price", data=data, ci=None)
```



In [285…

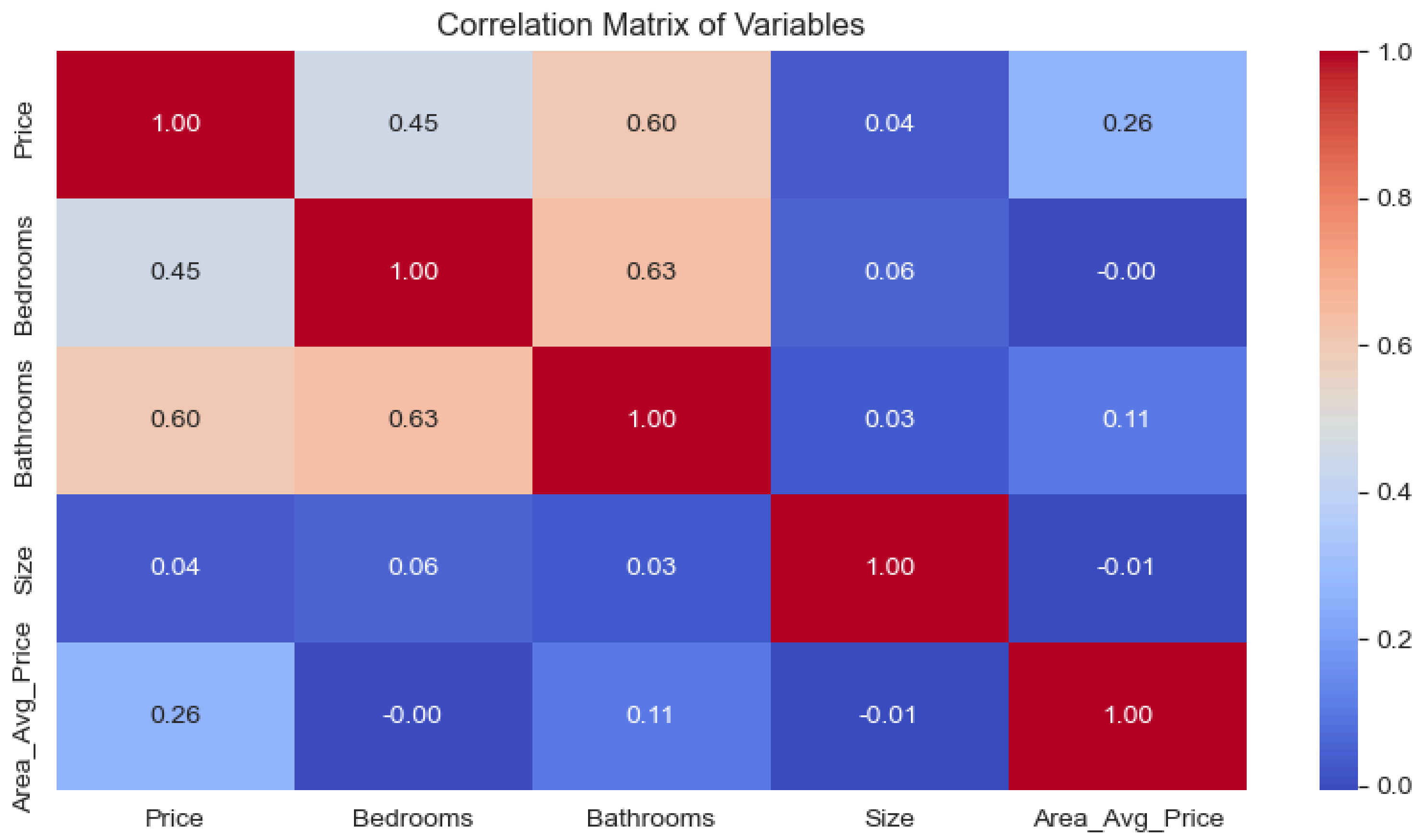
```
# Plotted Average Price by Number of Bathrooms
sns.barplot(x="Bathrooms", y="Price", data=data, ci=None)
plt.title("Average Price by Bathroom Number")
plt.xlabel("Bathroom Number")
plt.ylabel("Price")
plt.show()
```

```
/var/folders/03/6nkjsrk54q3f89x4qy6tnl280000gn/T/ipykernel_68979/1937866968.py:2: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  sns.barplot(x="Bathrooms", y="Price", data=data, ci=None)
```



In [285…

```
# Created a correlation matrix to assess the multicollinearity between variables
numericals = data.select_dtypes(include=['int64','float64'])
corr_matrix = numericals.corr()
plt.figure(figsize=(10,5))
sns.heatmap(corr_matrix,annot=True,cmap="coolwarm",fmt=".2f")
plt.title("Correlation Matrix of Variables")
plt.show()
```



Preprocessing Data

The dataset underwent several preprocessing procedures to guarantee data quality and applicability for machine learning models. These included feature scaling, encoding categorical variables, and removing collinear/irrelevant values.

Dropping columns

In [285…

```
columns = ["Postcode"]  # Define the list of columns to drop
data = data.drop(columns=columns, errors="ignore")  # Drop specified columns
print(data.columns)  # Display remaining columns
```

```
Index(['Price', 'Property_Type', 'Bedrooms', 'Bathrooms', 'Size', 'Area',
       'Price_Category', 'Area_Avg_Price'],
      dtype='object')
```

Feature Scaling

Since the Postcode column had no discernible impact on predicted performance, it was eliminated. Price, Area_Avg_Price, and Size were log-transformed to improve correlations' linearity and stabilize variance to address skewness in the data distribution. Better model convergence and resilience are made possible by this modification.

In [286…

```
# Apply log transformation to normalize skewed numerical features
data['Price'] = np.log1p(data["Price"])
data['Area_Avg_Price'] = np.log1p(data["Area_Avg_Price"])
data['Size'] = np.log1p(data["Size"])
data.head()
```

Out[286…

	Price	Property_Type	Bedrooms	Bathrooms	Size	Area	Price_Category	Area_Avg_Price
0	12.706851	Apartment	1.0	1.0	6.251904	Eastern	Low	13.817195
1	12.736704	Flat	1.0	1.0	6.789533	Eastern	Low	13.817195
2	12.736704	Apartment	1.0	1.0	6.841155	Eastern	Low	13.817195
3	12.736704	Flat	1.0	1.0	6.789533	Eastern	Low	13.817195
4	12.736704	Flat	1.0	1.0	5.963579	South Western	Low	14.232064

Encoding the Variables

For model compatibility, categorical variables like Property_Type, Area, and Price_Category were label-encoded to translate them into numerical values. In order to preserve efficiency and avoid needless dimensional expansion, which might result in sparsity problems, label encoding was chosen over one-hot encoding.

In [286…

```
# Encode categorical variables using LabelEncoder
categorical_variables = ["Property_Type","Area","Price_Category"]
data_encoded = data.copy()
le = {}
for col in categorical_variables:
    le[col]=LabelEncoder()
    data_encoded[col] = le[col].fit_transform(data_encoded[col])

data_encoded.head(20)
```

Out[286…

	Price	Property_Type	Bedrooms	Bathrooms	Size	Area	Price_Category	Area_Avg_Price
0	12.706851	0	1.0	1.0	6.251904	2	1	13.817195
1	12.736704	1	1.0	1.0	6.789533	2	1	13.817195
2	12.736704	0	1.0	1.0	6.841155	2	1	13.817195
3	12.736704	1	1.0	1.0	6.789533	2	1	13.817195
4	12.736704	1	1.0	1.0	5.963579	12	1	14.232064
5	12.736704	0	2.0	1.0	5.442418	15	1	14.301523
6	12.736704	1	1.0	1.0	6.144186	2	1	13.817195
7	12.736704	0	2.0	1.0	6.841155	11	1	13.447494
8	12.736704	1	2.0	2.0	6.501290	2	1	13.817195
9	12.736704	0	2.0	1.0	6.841155	2	1	13.817195
10	12.736704	1	2.0	1.0	6.789533	11	1	13.447494
11	12.736704	0	1.0	1.0	6.841155	2	1	13.817195
12	12.736704	0	1.0	1.0	6.841155	11	1	13.447494
13	12.736704	0	2.0	1.0	6.523562	2	1	13.817195
14	12.736704	1	2.0	1.0	6.789533	9	1	14.028430
15	12.736704	1	1.0	1.0	6.789533	2	1	13.817195
16	12.736704	1	1.0	1.0	6.386879	2	1	13.817195
17	12.736704	1	1.0	1.0	6.154858	2	1	13.817195
18	12.736704	0	1.0	1.0	6.361302	2	1	13.817195
19	12.736704	0	1.0	1.0	6.841155	11	1	13.447494

In [286…

```
data_encoded.info()
#Checking the datatypes after the encoding and standardization
```

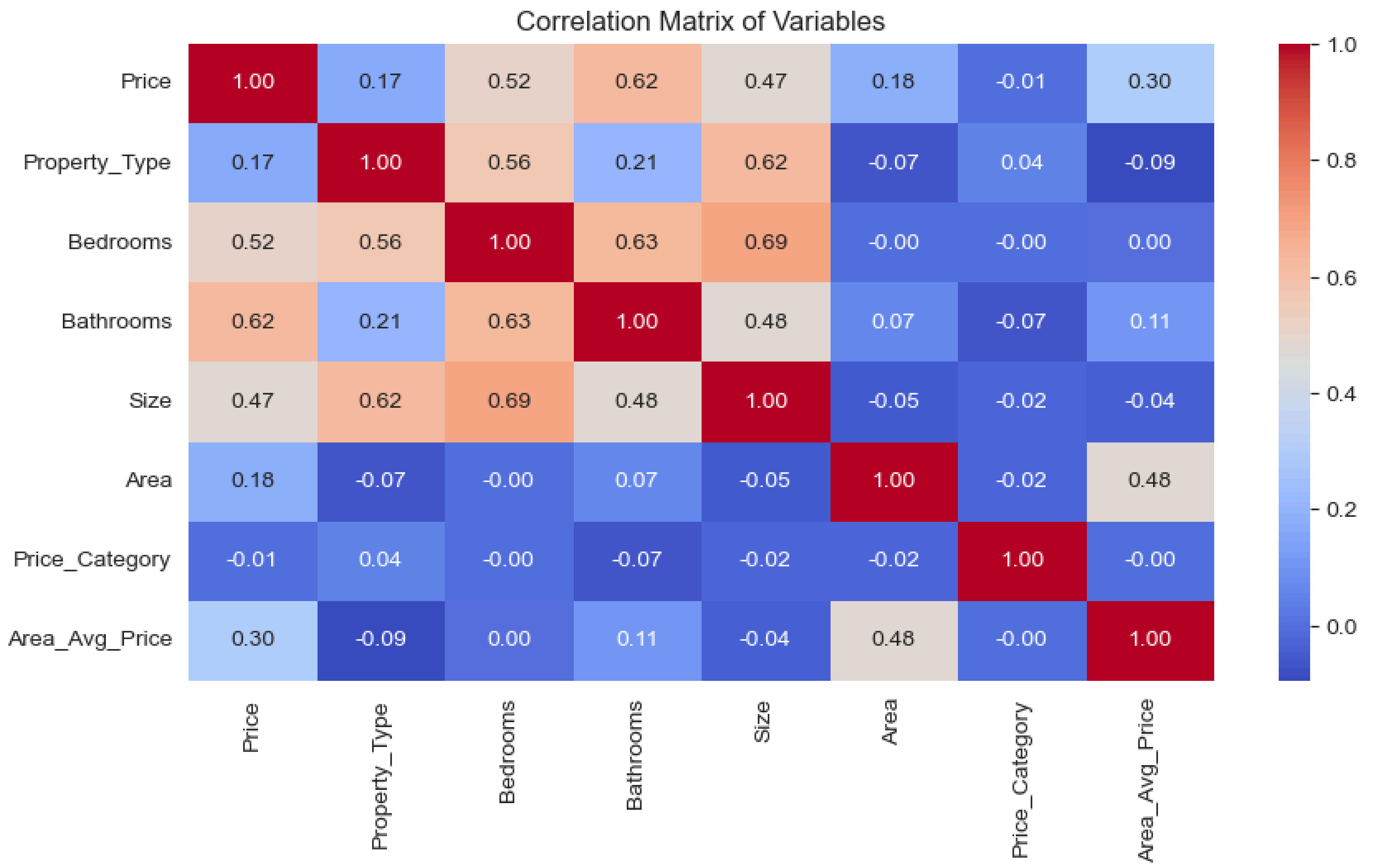
```
<class 'pandas.core.frame.DataFrame'>
Index: 29536 entries, 0 to 29536
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Price           29536 non-null  float64
 1   Property_Type   29536 non-null  int64
 2   Bedrooms        29536 non-null  float64
 3   Bathrooms       29536 non-null  float64
 4   Size            29536 non-null  float64
 5   Area            29536 non-null  int64
 6   Price_Category  29536 non-null  int64
 7   Area_Avg_Price  29536 non-null  float64
dtypes: float64(5), int64(3)
memory usage: 2.0 MB
```

In [286…

```
# Correct data types
data_encoded.dropna(subset=["Property_Type","Price_Category","Area"],inplace=True)
data_encoded["Property_Type"] = data_encoded["Property_Type"].astype(int)
data_encoded["Price_Category"] = data_encoded["Price_Category"].astype(int)
data_encoded["Area"] = data_encoded["Area"].astype(int)
```

In [286…

```
# Created a correlation matrix with all left variables to determine their suitability for machine learning models.
numericals = data_encoded.select_dtypes(include=['int64','float64'])
corr_matrix = numericals.corr()
plt.figure(figsize=(10,5))
sns.heatmap(corr_matrix,annot=True,cmap="coolwarm",fmt=".2f")
plt.title("Correlation Matrix of Variables")
plt.show()
```



Machine Learning Models

Highly correlated features such as Area and Area_Avg_Price were assessed, with Area removed due to redundancy. Similarly, the Bathrooms column was eliminated as it exhibited multicollinearity with Bedrooms, ensuring model interpretability and preventing overfitting.

This structured approach guarantees optimal data preparation, ensuring seamless integration with machine learning models.

In [286…

```
# Drop unsuitable columns
columns = [ "Bathrooms", "Area"]
data_encoded.drop(columns = columns, inplace = True)
data_encoded.head()
```

Out[286…

	Price	Property_Type	Bedrooms	Size	Price_Category	Area_Avg_Price
0	12.706851	0	1.0	6.251904	1	13.817195
1	12.736704	1	1.0	6.789533	1	13.817195
2	12.736704	0	1.0	6.841155	1	13.817195
3	12.736704	1	1.0	6.789533	1	13.817195
4	12.736704	1	1.0	5.963579	1	14.232064

In [287…

```
# Train-Test Split
train_set, test_set = train_test_split(data_encoded, test_size=0.2, random_state=42)

# Checking the split
train_set.shape, test_set.shape
```

Out[287…

```
((23628, 6), (5908, 6))
```

A rigorous evaluation of machine learning models was conducted to determine the optimal approach for price prediction. The models considered include:

1. Linear Regression: A fundamental baseline model used for benchmarking performance.
2. Polynomial Regression: Designed to capture nonlinear relationships, enhancing predictive accuracy.
3. Support Vector Regression (SVR): Handling complex feature interactions and outliers effectively.
4. Decision Tree Regression: Highly interpretable but prone to overfitting.
5. Random Forest Regression: An ensemble learning approach that enhances predictive power and reduces variance.

While polynomial regression enabled more intricate feature interactions, linear regression, the most basic model, served as a baseline. SVR was used to manage outliers efficiently. Random Forest is the better option because of its ensemble learning strategy, which reduces overfitting while preserving excellent prediction accuracy. Decision Tree Regression, despite its interpretability, frequently overfits the data.

Despite being computationally more costly than simpler models, Random Forest Regression achieves the best possible balance between complexity and performance. By combining several decision trees, the model can increase accuracy while lowering the chance of overfitting. It also gains from feature importance assessment, which improves feature selection techniques.

Careful consideration was given to the computational trade-off. Simpler models, such as linear regression, use less processing power but are not as accurate at making predictions in complicated datasets. On the other hand, high-complexity models such as Random Forest and SVR increase accuracy at the expense of higher processing demands.

In [287…
```
# 1. Linear Regression Model
# Define target (Y) and features (X)
target_col = "Price"
X_train = train_set.drop(columns=[target_col])
y_train = train_set[target_col]

X_test = test_set.drop(columns=[target_col])
y_test = test_set[target_col]

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

# Predictions
y_train_pred = lin_reg.predict(X_train)
y_test_pred = lin_reg.predict(X_test)

# Evaluation Metrics
mse_train = mean_squared_error(y_train, y_train_pred)
rmse_train = np.sqrt(mse_train)
mae_train = mean_absolute_error(y_train, y_train_pred)
r2_train = r2_score(y_train, y_train_pred)

mse_test = mean_squared_error(y_test, y_test_pred)
rmse_test = np.sqrt(mse_test)
mae_test = mean_absolute_error(y_test, y_test_pred)
r2_test = r2_score(y_test, y_test_pred)

# Display results
metrics_df = pd.DataFrame({
    "Metric": ["MSE", "RMSE", "MAE", "R²"],
    "Training Score": [mse_train, rmse_train, mae_train, r2_train],
    "Test Score": [mse_test, rmse_test, mae_test, r2_test]
})

# Linear Regression Metrics
metrics_df
```

Out[287…

	Metric	Training Score	Test Score
0	MSE	0.283987	0.277240
1	RMSE	0.532904	0.526536
2	MAE	0.408505	0.404138
3	R^2	0.418586	0.418907

In [287…
```
# Polynomial Regression Model
# Generate polynomial features
degree = 4
poly = PolynomialFeatures(degree=degree)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Initialize and train Polynomial Regression model (using Linear Regression on polynomial features)
poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)

# Predictions
y_train_pred_poly = poly_model.predict(X_train_poly)
y_test_pred_poly = poly_model.predict(X_test_poly)

# Evaluation Metrics
mse_train = mean_squared_error(y_train, y_train_pred_poly)
rmse_train = np.sqrt(mse_train)
mae_train = mean_absolute_error(y_train, y_train_pred_poly)
r2_train = r2_score(y_train, y_train_pred_poly)

mse_test = mean_squared_error(y_test, y_test_pred_poly)
rmse_test = np.sqrt(mse_test)
mae_test = mean_absolute_error(y_test, y_test_pred_poly)
r2_test = r2_score(y_test, y_test_pred_poly)

# Display results
metrics_df = pd.DataFrame({
    "Metric": ["MSE", "RMSE", "MAE", "R²"],
    "Training Score": [mse_train, rmse_train, mae_train, r2_train],
    "Test Score": [mse_test, rmse_test, mae_test, r2_test]
})

# Output the Polynomial Regression Metrics
metrics_df
```

Out [287...

	Metric	Training Score	Test Score
0	MSE	0.058702	0.090664
1	RMSE	0.242284	0.301104
2	MAE	0.187735	0.192683
3	R²	0.879819	0.809969

In [287...

```
# 3. SVR Moddel
# Initialize and train Support Vector Regression model (RBF kernel)
svr_model = SVR(kernel="rbf")
svr_model.fit(X_train, y_train)

# Predictions
y_train_pred_svr = svr_model.predict(X_train)
y_test_pred_svr = svr_model.predict(X_test)

# Evaluation Metrics
mse_train = mean_squared_error(y_train, y_train_pred_svr)
rmse_train = np.sqrt(mse_train)
mae_train = mean_absolute_error(y_train, y_train_pred_svr)
r2_train = r2_score(y_train, y_train_pred_svr)

mse_test = mean_squared_error(y_test, y_test_pred_svr)
rmse_test = np.sqrt(mse_test)
mae_test = mean_absolute_error(y_test, y_test_pred_svr)
r2_test = r2_score(y_test, y_test_pred_svr)

# Display results
metrics_df = pd.DataFrame({
    "Metric": ["MSE", "RMSE", "MAE", "R²"],
    "Training Score": [mse_train, rmse_train, mae_train, r2_train],
    "Test Score": [mse_test, rmse_test, mae_test, r2_test]
})

# Support Vector Regression Metrics
metrics_df
```

Out [287...

	Metric	Training Score	Test Score
0	MSE	0.254024	0.245232
1	RMSE	0.504008	0.495209
2	MAE	0.331125	0.327276
3	R²	0.479929	0.485996

In [287...

```
# 4. Decision Tree Regression Model
# Initialize and train Decision Tree Regression model
dt_model = DecisionTreeRegressor(random_state=42)
dt_model.fit(X_train, y_train)

# Predictions
y_train_pred_dt = dt_model.predict(X_train)
y_test_pred_dt = dt_model.predict(X_test)

# Evaluation Metrics
mse_train = mean_squared_error(y_train, y_train_pred_dt)
rmse_train = np.sqrt(mse_train)
mae_train = mean_absolute_error(y_train, y_train_pred_dt)
r2_train = r2_score(y_train, y_train_pred_dt)

mse_test = mean_squared_error(y_test, y_test_pred_dt)
rmse_test = np.sqrt(mse_test)
mae_test = mean_absolute_error(y_test, y_test_pred_dt)
r2_test = r2_score(y_test, y_test_pred_dt)

# Display results
metrics_df = pd.DataFrame({
    "Metric": ["MSE", "RMSE", "MAE", "R²"],
    "Training Score": [mse_train, rmse_train, mae_train, r2_train],
    "Test Score": [mse_test, rmse_test, mae_test, r2_test]
})

# Decision Tree Regression Metrics
metrics_df
```

Out [287...

	Metric	Training Score	Test Score
0	MSE	0.031588	0.074551
1	RMSE	0.177730	0.273040
2	MAE	0.101072	0.201641
3	R²	0.935329	0.843742

In [287...

```
# 5. Random Forest Regression Model
# Initialize and train Random Forest Regression model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```python
rf_model.fit(X_train, y_train)

# Predictions
y_train_pred_rf = rf_model.predict(X_train)
y_test_pred_rf = rf_model.predict(X_test)

# Evaluation Metrics
mse_train = mean_squared_error(y_train, y_train_pred_rf)
rmse_train = np.sqrt(mse_train)
mae_train = mean_absolute_error(y_train, y_train_pred_rf)
r2_train = r2_score(y_train, y_train_pred_rf)

mse_test = mean_squared_error(y_test, y_test_pred_rf)
rmse_test = np.sqrt(mse_test)
mae_test = mean_absolute_error(y_test, y_test_pred_rf)
r2_test = r2_score(y_test, y_test_pred_rf)

# Display results
metrics_df = pd.DataFrame({
    "Metric": ["MSE", "RMSE", "MAE", "R²"],
    "Training Score": [mse_train, rmse_train, mae_train, r2_train],
    "Test Score": [mse_test, rmse_test, mae_test, r2_test]
})

# Random Forest Regression Metrics
metrics_df
```

Out[287…

	Metric	Training Score	Test Score
0	MSE	0.034619	0.060607
1	RMSE	0.186062	0.246185
2	MAE	0.127774	0.185497
3	R^2	0.929123	0.872968

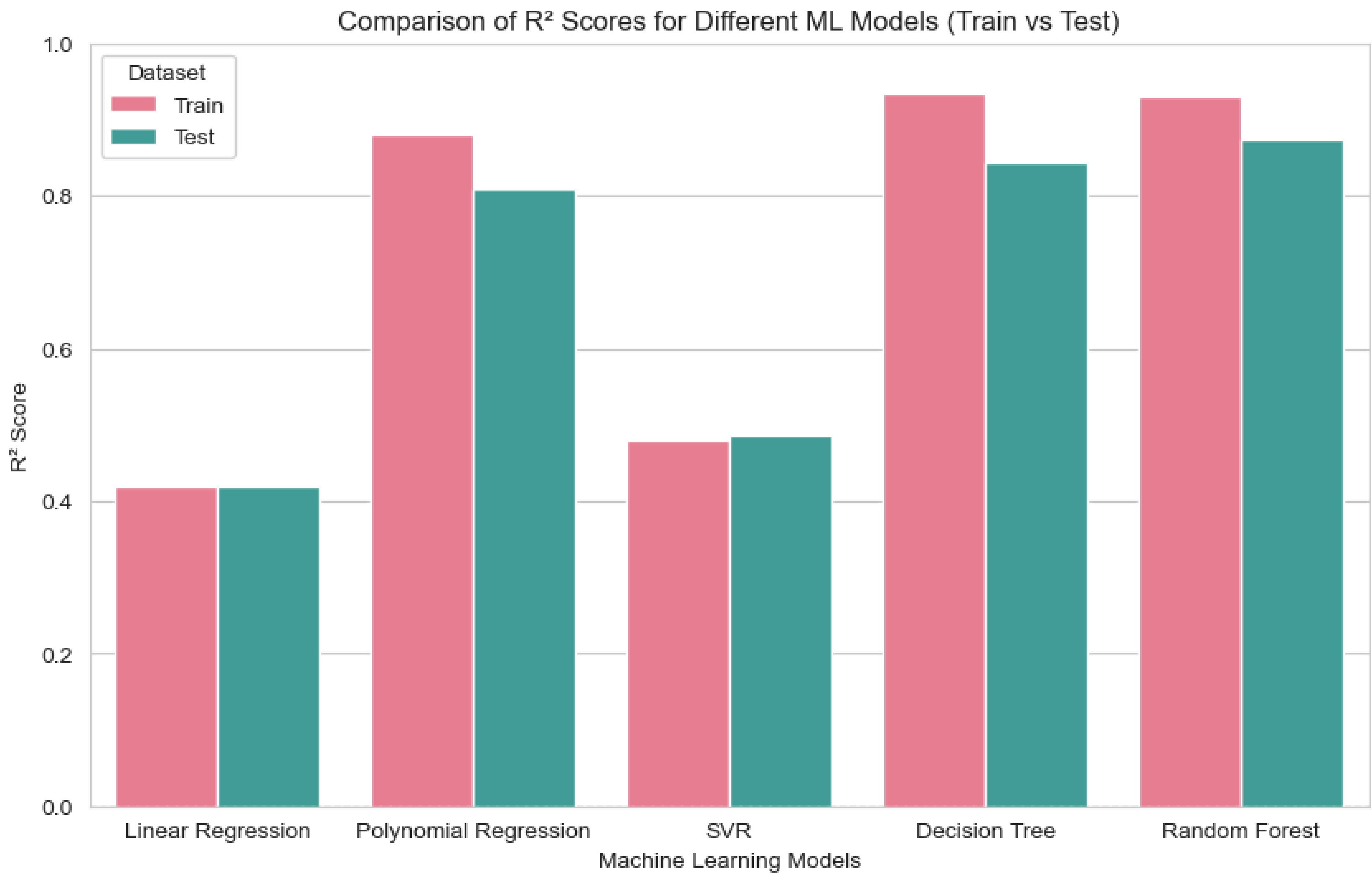
Results and Interpretation

Linear Regression, serving as a baseline, has an R^2 score of approximately 0.41, limiting its ability to capture complex relationships. Polynomial Regression significantly improves R^2 to around 0.87 for training and 0.80 for testing, showing enhanced pattern recognition but some overfitting. Support Vector Regression (SVR) marginally outperforms Linear Regression with an R^2 of about 0.48, effectively handling outliers but at a high computational cost. Decision Tree Regression achieves an R^2 of 0.94 in training but drops to 0.84 in testing, reflecting severe overfitting. Random Forest Regression provides the best balance with an R^2 of 0.92 for training and 0.87 for testing. Residual analysis confirms Random Forest's low prediction errors and minimal bias. Feature importance highlights size and price category as primary predictors, reinforcing the model's validity. Future improvements include hyperparameter tuning and external data integration.

In line with real estate market concepts that prioritize square footage and price segmentation, the results show that property size and price category are the most significant features. While the typical price of the area and the type of property are not very important, bedrooms are a secondary factor that contributes somewhat to price estimation. By confirming that the most pertinent predictors were kept for modeling, these results verify the feature selection procedure. The findings imply that physical qualities have a higher weight than categorical or regional influences in the hierarchical structure of feature importance used in property price estimation. This has important ramifications for real estate valuation models, suggesting that fundamental property qualities should take precedence over secondary market considerations in automated pricing systems. The results also imply that although nonlinear interactions do exist, they are not very strong, indicating that the best models for prediction are ensemble approaches like Random Forest or somewhat sophisticated models like Polynomial Regression.

In [287…

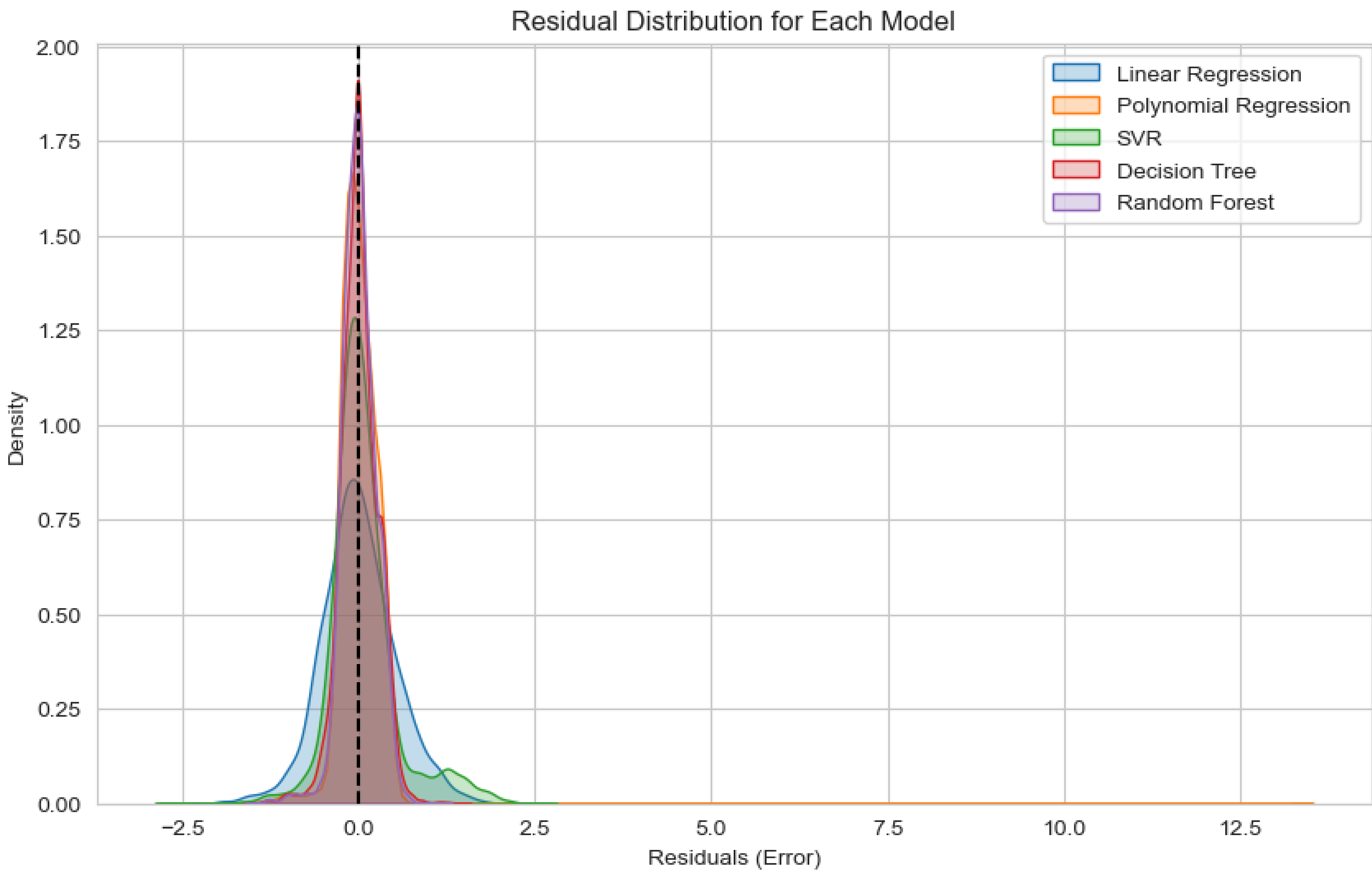
```python
# Replace with actual R² values from your models
models = ["Linear Regression", "Polynomial Regression", "SVR", "Decision Tree", "Random Forest"]
r2_train_scores = [0.418586, 0.879819, 0.479929, 0.935329, 0.929123]  # Training R² scores
r2_test_scores = [0.418907, 0.809969, 0.485996, 0.843742, 0.872968]  # Test R² scores

# Create DataFrame for Seaborn (reshaped correctly)
df_r2 = pd.DataFrame({
    "Model": models * 2,
    "R² Score": r2_train_scores + r2_test_scores,
    "Dataset": ["Train"] * len(models) + ["Test"] * len(models)
})

# Set Seaborn style and color palette
sns.set_style("whitegrid")
plt.figure(figsize=(10, 6))
sns.barplot(x="Model", y="R² Score", hue="Dataset", data=df_r2, palette="husl")

# Formatting
plt.axhline(y=0, color="black", linestyle="--", linewidth=0.8)  # Reference line at R² = 0
plt.ylim(0, 1)  # Adjust based on R² values
plt.xlabel("Machine Learning Models")
plt.ylabel("R² Score")
plt.title("Comparison of R² Scores for Different ML Models (Train vs Test)")
plt.xticks(rotation=0)
plt.legend(title="Dataset")

# Show plot
plt.show()
```



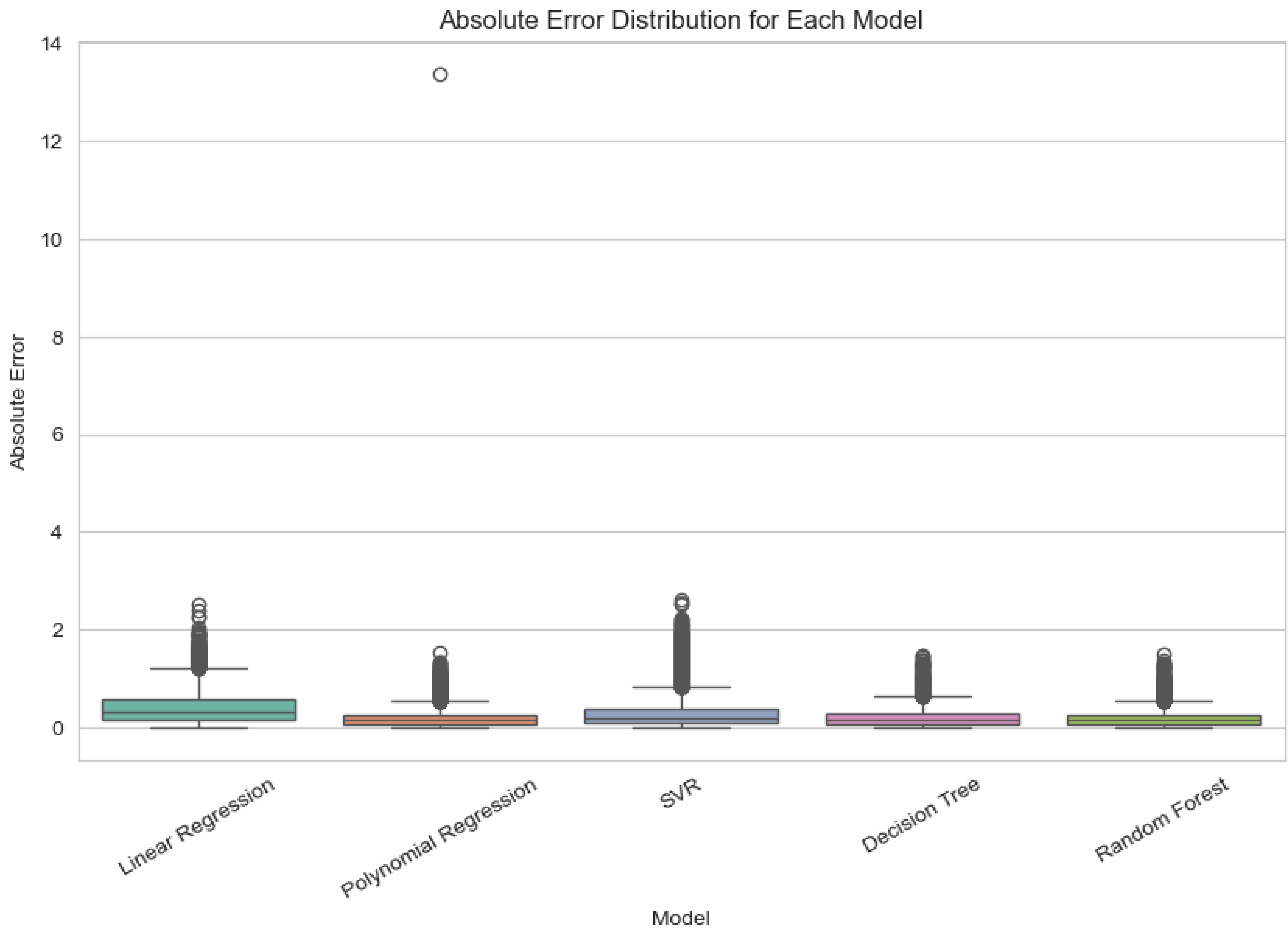
In [287…

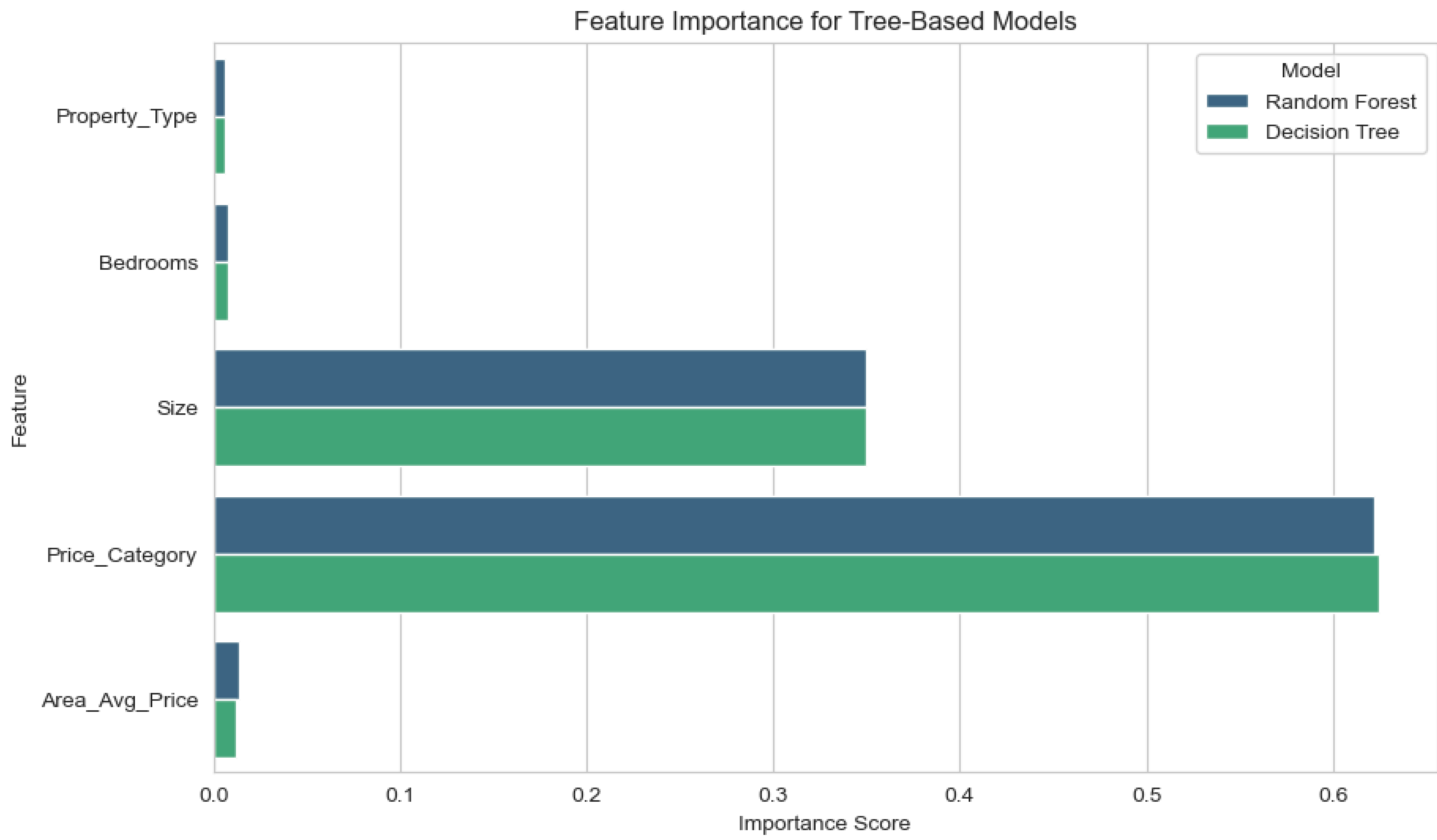
```python
# List of models
models = ["Linear Regression", "Polynomial Regression", "SVR", "Decision Tree", "Random Forest"]

# Residual Plot (Prediction Errors for Each Model)
plt.figure(figsize=(10, 6))
for model, residual in zip(models, [
    y_test - y_test_pred,
    y_test - y_test_pred_poly,
    y_test - y_test_pred_svr,
    y_test - y_test_pred_dt,
    y_test - y_test_pred_rf
]):
    sns.kdeplot(residual, fill=True, label=model)
plt.axvline(x=0, color="black", linestyle="--")
plt.xlabel("Residuals (Error)")
plt.ylabel("Density")
plt.title("Residual Distribution for Each Model")
plt.legend()
plt.show()

# Actual vs. Predicted Scatter Plot
plt.figure(figsize=(10, 6))
for model, y_pred in zip(models, [y_test_pred, y_test_pred_poly, y_test_pred_svr, y_test_pred_dt, y_test_pred_rf]):
    plt.scatter(y_test, y_pred, alpha=0.5, label=model)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], "--", color="black", label="Perfect Prediction")
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.title("Actual vs. Predicted Values for Different Models")
plt.legend()
plt.show()

# Box Plot of Absolute Errors Across Models
error_data = pd.DataFrame({
    "Model": np.repeat(models, len(y_test)),
    "Absolute Error": np.concatenate([
        abs(y_test - y_test_pred),
        abs(y_test - y_test_pred_poly),
        abs(y_test - y_test_pred_svr),
        abs(y_test - y_test_pred_dt),
        abs(y_test - y_test_pred_rf)
    ])
})
plt.figure(figsize=(10, 6))
sns.boxplot(x="Model", y="Absolute Error", data=error_data, palette="Set2")
plt.title("Absolute Error Distribution for Each Model")
plt.xticks(rotation=30)
plt.show()

# Feature Importance for Tree-Based Models (Random Forest & Decision Tree)
feature_importance_rf = rf_model.feature_importances_
feature_importance_dt = dt_model.feature_importances_
features = X_train.columns
df_importance = pd.DataFrame({
    "Feature": np.tile(features, 2),
    "Importance": np.concatenate([feature_importance_rf, feature_importance_dt]),
    "Model": ["Random Forest"] * len(features) + ["Decision Tree"] * len(features)
})
plt.figure(figsize=(10, 6))
sns.barplot(x="Importance", y="Feature", hue="Model", data=df_importance, palette="viridis")
plt.title("Feature Importance for Tree-Based Models")
```

```
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```




```
/var/folders/03/6nkjsrk54q3f89x4qy6tnl280000gn/T/ipykernel_68979/1594346662.py:44: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.boxplot(x="Model", y="Absolute Error", data=error_data, palette="Set2")
```





Limitations

Several issues arose despite thorough preprocessing and model selection. The class imbalance between price categories was present in the dataset, which could skew model predictions. Furthermore, feature selection presented trade-offs since, while removing collinear variables enhanced interpretability, it omitted essential interactions. Because complicated models required a large amount of processing time and resources, computational limits were also considered.

Additionally, although Random Forest Regression had good accuracy, it added complexity to the model, which reduced the effectiveness of real-time predictions. Hyperparameter adjustment was also necessary to avoid overfitting and preserve computational viability.

Future research could investigate feature engineering strategies like interaction terms and polynomial transformations to improve feature representation and overcome these difficulties. Sophistic hyperparameter tuning techniques like Bayesian Optimization could be used to optimize model parameters further. Alternative ensemble techniques like LightGBM and XGBoost might also be looked into to strike a compromise between prediction performance and efficiency. Performance may also be enhanced by incorporating deep learning models, like neural networks, especially for intricate feature interactions.

Future studies can use these tactics to improve the predictive modeling approach, guaranteeing improved precision, effectiveness, and practicality in estimating property prices.

In [294…

```
def count_words_in_markdown(notebook_path):
    """Counts the number of words in markdown cells of a Jupyter Notebook."""

    with open(notebook_path, 'r', encoding='utf-8') as f:
        notebook = json.load(f)

    word_count = 0

    # Loop through notebook cells
    for cell in notebook.get("cells", []):
        if cell.get("cell_type") == "markdown":  # Only consider markdown cells
            markdown_text = "".join(cell.get("source", []))
            words = re.findall(r'\b\w+\b', markdown_text)  # Extract words
            word_count += len(words)

    return word_count

notebook_path = "Final_final Assessment.ipynb"
word_count = count_words_in_markdown(notebook_path)

print(f"Total words in Markdown cells: {word_count}")
```

```
Total words in Markdown cells: 1052
```