Real Estate Price Prediction in Bangladesh

Project overview

In this project, I explore a real estate dataset, perform data cleaning, conduct exploratory data analysis (EDA), and build a linear regression model to predict property prices in Bangladesh. The dataset focuses on real estate listings from various cities across Bangladesh, including Dhaka, Chattogram, Cumilla, Narayanganj City, and Gazipur. It provides information on features such as bedrooms, bathrooms, floor area, and their corresponding prices in Bangladeshi Taka (৳). This data was sourced from a real estate website and is available on Kaggle.

1. Data reading

The first step is to import the necessary Python libraries to handle the dataset:

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

After downloading the dataset from Kaggle, I used the Pandas read_csv function to load it into a DataFrame:

```
df = pd.read_csv("house_price_bd.csv")
```

To get a quick understanding of the dataset, I used the head() function to see the first few rows:

```
df.head()
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
0	We Are Offering You A Very Spacious 1960 Sq Ft...	3.0	4.0	3	vacant	1960.0	dhaka	৳39,000,000	Gulshan 1, Gulshan
1	Valuable 1705 Square Feet Apartment Is Ready T...	3.0	3.0	1	vacant	1705.0	dhaka	৳16,900,000	Lake Circus Road, Kalabagan
2	1370 square feet apartment is ready to sale in...	3.0	3.0	6	vacant	1370.0	dhaka	৳12,500,000	Shukrabad, Dhanmondi
3	2125 Square Feet Apartment For Sale In Bashund...	3.0	3.0	4	vacant	2125.0	dhaka	৳20,000,000	Block L, Bashundhara R-A
4	Buy This 2687 Square Feet Flat In The Nice Are...	3.0	3.0	4	vacant	2687.0	dhaka	৳47,500,000	Road No 25, Banani

Next, I examined the structure and size of the dataset:

```
df.shape
```

output:

(3865, 9)

The dataset contains **3865 rows** and **9 columns**.

I used describe() to get some basic statistics about the dataset:

```
df.describe()
```

output:

	Bedrooms	Bathrooms	Floor_area
count	2864.000000	2864.000000	3766.000000
mean	3.133031	2.992668	1940.299522
std	2.215457	0.978434	6024.921935
min	1.000000	1.000000	84.000000
25%	3.000000	3.000000	1100.000000
50%	3.000000	3.000000	1380.000000
75%	3.000000	3.000000	1860.000000
max	50.000000	10.000000	195840.000000

Finally, to understand the type of data in each column, I used:

```
df.info()
```

output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3865 entries, 0 to 3864
Data columns (total 9 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Title             3865 non-null   object 
 1   Bedrooms          2864 non-null   float64
 2   Bathrooms         2864 non-null   float64
 3   Floor_no          3181 non-null   object 
 4   Occupancy_status  3766 non-null   object 
 5   Floor_area        3766 non-null   float64
 6   City              3865 non-null   object 
 7   Price_in_taka     3865 non-null   object 
 8   Location          3859 non-null   object 
dtypes: float64(3), object(6)
memory usage: 271.9+ KB
```

The results show that the **Title** column is of type object, which makes sense. Other columns like **Bedrooms** and **Bathrooms** are float64, but they could ideally be integers. Columns like **Price_in_taka** contain a currency symbol and will need to be cleaned. Additionally, the **Floor_no** column is of type object but should be numeric.

2. Data cleaning

Before making any modifications, I created a copy of the dataset to ensure the original data remains untouched:

```
df_clean = df.copy()
```

2.1 Duplicates

Checking for duplicate rows is essential to avoid bias in the model. I started by checking the number of duplicates:

```
df_clean.duplicated().sum()
```

output:

934

Since there were 934 duplicate rows, I removed them using:

```
df_clean.drop_duplicates(inplace = True)
```

3.1 Missing data

To identify missing values in the dataset, I used:

```
df_clean.isna().sum()
```

output:

```
Title                0
Bedrooms           831
Bathrooms          831
Floor_no           575
Occupancy_status    89
Floor_area          89
City                 0
Price_in_taka        0
Location             6
dtype: int64
```

For instance, the **Floor_no** column had missing values. I examined this column to understand the type of data it contains:

```
df_clean["Floor_no"].value_counts()
```

output:

```
Floor_no
1                        379
4                        306
5                        293
2                        274
3                        261
6                        249
7                        200
8                        184
9                        126
10                        23
11                        20
12                        16
13                         7
Merin City - Purbach       6
14                         2
1st                        1
8th                        1
A1,A2,A3,A4,A5,A6,A7       1
0+7                        1
1F                         1
4th to 8th Backside        1
18                         1
5th                        1
17                         1
G+7                        1
Name: count, dtype: int64
```

The column is set to object due to mixed formatting issues (e.g., strings like "1st" or "8th"). Therefore, cleaning and standardizing this column is necessary before modeling.

Dropping Unusable Rows

In some cases, such as for "Merin City - Purbach", there is no useful information for the number of **Bedrooms** and **Bathrooms**, making these rows unusable for further analysis:

```
df_clean[df_clean["Floor_no"] == "Merin City - Purbach"]
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
3313	Plot For Sale In A Suitable Place In Bproperty...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj
3388	For Sale, Residential Plot Is Situated In Bpro...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj
3392	Plot For Sale In A Suitable Place In Bproperty...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj
3393	Plot For Sale In A Suitable Place In Bproperty...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj
3432	In The Beautiful Location Of Bproperty Village...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj
3492	Plot For Sale In A Suitable Place In Narayanga...	NaN	NaN	Merin City - Purbach	vacant	2160.0	narayanganj-city	৳4,575,000	Rupganj, Narayanganj

Since we cannot recover this data, I dropped the 6 rows corresponding to this entry:

```
df_clean = df_clean[df_clean['Floor_no'] != 'Merin City - Purbach']
```

Handling "4th to 8th Backside"

For entries like "4th to 8th Backside", I decided to create multiple rows for each floor, as this property refers to an apartment that spans several floors. I replaced the row with four new rows representing each floor (from 4th to 8th):

```
df_clean[df_clean['Floor_no'] == '4th to 8th Backside']
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	4th to 8th Backside	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward

It’s an apartment as the title mention, so the solution I chose can be correct, to do so I used:

```
# Rows to replace (indices where 'Floor_no' contains '4th to 8th Backside')
rows_to_replace = df_clean[df_clean['Floor_no'] == '4th to 8th Backside']

# Create new rows for each floor from 4 to 8
new_rows = []
for i in range(4, 9):  # Floors from 4 to 8
    for index, row in rows_to_replace.iterrows():
        new_row = row.copy()
        new_row['Floor_no'] = i  # Set the floor number as a numeric value
        new_rows.append(new_row)

# Convert the list of new rows to a DataFrame
new_rows_df = pd.DataFrame(new_rows)

# Remove the original rows
```

```
df_clean = df_clean.drop(rows_to_replace.index)

# Use pd.concat() to append the new rows to the DataFrame
df_clean = pd.concat([df_clean, new_rows_df], ignore_index=True)
```

the rows:

```
new_rows_df
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	4	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	5	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	6	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	7	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward
2674	An Apartment Is Up For Sale In Dakshin Kattali...	3.0	3.0	8	vacant	1250.0	chattogram	৳3,800,000	Dakshin Kattali, 11 No. South Kattali Ward

Cleaning Entries with "th", "st", and "F"

For entries such as "8th", "5th", "1st", and "1F", I removed the non-numeric characters:

```
df_clean[(df_clean["Floor_no"] == "8th") | (df_clean["Floor_no"] == "5th") | (df_clean["Floor_no"] == "1st") |
(df_clean["Floor_no"] == "1F")]
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
34	An Apartment For Sale Is All Set For You To Se...	3.0	3.0	8th	vacant	1100.0	dhaka	৳4,700,000	Dhaka Uddan, Mohammadpur
741	Imagine A Spacious Flat That Comes With Your A...	4.0	5.0	1st	vacant	2380.0	dhaka	৳29,000,000	Block D, Bashundhara R-A
1551	At Bayazid, Flat For Sale Close To Bayazid Thana	3.0	3.0	1F	vacant	1313.0	chattogram	৳6,500,000	Chadra Nagar, Bayazid
2126	Take This Residential Flat Is For Sale At Baya...	3.0	2.0	5th	vacant	1162.0	chattogram	৳5,000,000	Chadra Nagar, Bayazid

I removed the characters using:

```
df_clean["Floor_no"] = df_clean["Floor_no"].str.replace(r"th|st|F", "", regex=True)
```

Handling "G+7" and "0+7"

These entries refer to the total number of floors, where "G+7" refers to "Ground + 7 floors". I converted these to numeric values representing 8 floors:

```
df_clean[(df_clean["Floor_no"] == "G+7") | (df_clean["Floor_no"] == "0+7")]
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
217	19200 SQ FT Full-Building is now for sale in M...	46.0	10.0	G+7	vacant	19200.0	dhaka	৳75,000,000	Section 1, Mirpur
1458	A Residential Building Which Is Up For Sale At...	21.0	10.0	0+7	vacant	13300.0	chattogram	৳105,000,000	Rose Valley Residential Area, 9 No. North Paha...

I removed the characters using:

```
df_clean["Floor_no"] = df_clean["Floor_no"].str.replace(r"G\+7|0\+7", "8", regex=True)
```

Cleaning "A1, A2, ..., A7"

This entry referred to a building with 7 apartments. I assumed the building had 4 floors, with 2 apartments per floor, and updated the Floor_no accordingly:

```
df_clean[df_clean["Floor_no"] == "A1,A2,A3,A4,A5,A6,A7"]
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
1104	A Full Building Is For Sale In Sugandha Reside...	18.0	10.0	A1,A2,A3,A4,A5,A6,A7	vacant	10890.0	chattogram	৳105,000,000	Sugandha Residential Area, Panchlaish

```
df_clean["Floor_no"] = df_clean["Floor_no"].str.replace("A1,A2,A3,A4,A5,A6,A7", "4")
```

Handling Missing Values in Floor_no

After cleaning, I checked the distribution of Floor_no values:

```
df_clean["Floor_no"].value_counts(dropna = False)
```

output:

```
Floor_no
NaN    580
1      381
4      307
5      294
2      274
3      261
6      249
7      200
8      187
9      126
10      23
11      20
12      16
13       7
14       2
18       1
17       1
Name: count, dtype: int64
```

We still had **580 missing values** in this column. I converted the column to float to handle the missing values:

```
df_clean["Floor_no"] = df_clean["Floor_no"].astype("float")
```

Clarifying Property Types Using the Title Column

The Floor_no feature can be misleading, as it could represent either the floor the apartment is on or the total number of floors in a building. To clarify, I added a new column, **Type of Property for Sale**, by extracting relevant information from the Title column. I defined four property types: **Building**, **Apartment**, **Commercial**, and **House** using regular expressions:

```
building_pattern = r"building|complex|condominium|residence|property"
apartment_pattern = r"apartment|flat|living space|apt."
commercial_pattern = r"office|shop|retail|business|commercial space|warehouse|factory|store|business
center|showroom|industrial space|market|plaza|outlet|commercial"
house_pattern = r"house|residential|cottage|bungalow|duplex|townhouse|family home|detached house|semi-detached
house|ranch|estate|home"

df_clean = df_clean.assign(
    Building = np.where(
        df_clean["Title"].str.lower().str.contains(building_pattern, regex=True),1, 0),
    Appartment = np.where(
        df_clean["Title"].str.lower().str.contains(apartment_pattern, regex=True), 1, 0),
    Commercial = np.where(
        df_clean["Title"].str.lower().str.contains(commercial_pattern, regex=True), 1, 0),
    House = np.where(
        df_clean["Title"].str.lower().str.contains(house_pattern, regex=True), 1, 0)
    )
```

Filling Missing Values Using Medians

Now that the Floor_no column was cleaned and clarified, I addressed the missing values in **Bedrooms**, **Bathrooms**, and **Floor_no**. I calculated the median for each property type (Building, Apartment, House, Commercial) and floor number, and used these medians to fill in the missing values:

1. **Calculate Medians** for each Floor_no and property type:

```
# Create dictionaries to hold median floor numbers
median_floor_no = {
    "Building": None,
    "Appartment": None,
    "House": None,
    "Commercial": None
}

# Calculate median floor_no for each property type
for property_type in ["Building", "Appartment", "House", "Commercial"]:
    sub_df = df_clean[df_clean[property_type] == 1]
    median_floor_no[property_type] = sub_df["Floor_no"].median()

def fill_missing_floor_no(row):
    property_type = "Building" if row["Building"] == 1 else (
        "Appartment" if row["Appartment"] == 1 else (
            "House" if row["House"] == 1 else (
                "Commercial" if row["Commercial"] == 1 else None
            )
        )
    )

    if property_type:
        if pd.isna(row["Floor_no"]):
            row["Floor_no"] = median_floor_no[property_type]

    return row

# Apply the function to fill missing Floor_no values
df_clean = df_clean.apply(fill_missing_floor_no, axis=1)
```

Median of bedroom and bathrooms based on the floor number

```
# Create dictionaries to hold median values
median_values = {
    "Building": {},
    "Appartment": {},
    "House": {},
    "Commercial": {}
}

# Calculate medians for each property type and floor_no
for property_type in ["Building", "Appartment", "House", "Commercial"]:
```

```
    for floor in range(1, 19):
        sub_df = df_clean[(df_clean["Floor_no"] == floor) & (df_clean[property_type] == 1)]
        median_values[property_type][floor] = {
            "Bedrooms": sub_df["Bedrooms"].median(),
            "Bathrooms": sub_df["Bathrooms"].median()
        }

def fill_missing_values(row):
    property_type = "Building" if row["Building"] == 1 else (
        "Appartment" if row["Appartment"] == 1 else (
            "House" if row["House"] == 1 else (
                "Commercial" if row["Commercial"] == 1 else None
            )
        )
    )

    if property_type:
        floor = row["Floor_no"]
        if floor in median_values[property_type]:
            if pd.isna(row["Bedrooms"]):
                row["Bedrooms"] = median_values[property_type][floor]["Bedrooms"]
            if pd.isna(row["Bathrooms"]):
                row["Bathrooms"] = median_values[property_type][floor]["Bathrooms"]

    return row

# Apply the function to fill missing values
df_clean = df_clean.apply(fill_missing_values, axis=1)
```

This method ensured that the missing values were filled in a meaningful way, based on the type of property and its corresponding floor number.

Checking for Remaining Missing Values

After performing the data cleaning steps, I checked for any remaining missing values in the dataset:

```
df_clean.isna().sum()
```

output:

```
Title                0
Bedrooms            94
Bathrooms           94
Floor_no            79
Occupancy_status    89
Floor_area          89
City                 0
Price_in_taka        0
Location             6
Building             0
Appartment           0
Commercial           0
House                0
Cumilla_city         0
Dhaka_city           0
Gazipur_city         0
Narayanganj_city     0
vacant               0
dtype: int64
```

At this point, some missing values remain, especially in **Bedrooms**, **Bathrooms**, **Floor number**, and **Floor area**. To handle this, I applied the following steps.

Dropping Rows with Insufficient Information

I removed rows that had missing values in critical features, specifically **Bedrooms**, **Bathrooms**, **Floor_no**, and **Floor_area**. These features are essential for our analysis, and rows without them cannot be used for modeling:

```
df_clean = df_clean.drop(df_clean[(df_clean["Floor_no"].isna()) &
                        (df_clean["Bedrooms"].isna()) &
                        (df_clean["Bathrooms"].isna()) &
                        (df_clean["Floor_area"].isna())].index)
```

After this, I checked the remaining missing values again:

```
df_clean.isna().sum()
```

output:

```
Title                0
Bedrooms            81
Bathrooms           81
Floor_no            66
Occupancy_status    76
Floor_area          76
City                 0
Price_in_taka        0
Location             1
Building             0
Appartment           0
Commercial           0
House                0
Cumilla_city         0
Dhaka_city           0
Gazipur_city         0
Narayanganj_city     0
vacant               0
dtype: int64
```

Filling Missing Location Data

Upon examining the missing values in the **Location** column, I found that there was only one row with a missing value. By looking at the **Title** of the property, I was able to infer the location:

```
df_clean[df_clean["Location"].isna()]
```

output:

	Title	Bedrooms	Bathrooms	Floor_no	Occupancy_status	Floor_area	City	Price_in_taka	Location
383	1254 Sq Ft Flat For Sale In Kalachandpur	3.0	3.0	6.0	vacant	1254.0	dhaka	8300000.0	NaN

From the title we can fill this row:

```
df_clean.loc[df_clean["Location"].isna(), "Location"] = "Kalachandpur"
```

Dropping Rows with Missing Key Features

I further cleaned the dataset by dropping rows where key features like **Bedrooms** and **Floor_area** were missing. These features are crucial for predicting property prices, so rows without them were removed:

```
df_clean = df_clean.dropna(subset=["Bedrooms", "Floor_area"])
```

Final Check for Missing Values

To ensure the dataset was clean and ready for further analysis, I performed one last check for missing values:

```
df_clean.isna().sum()
```

output:

```
Title               0
Bedrooms            0
Bathrooms           0
Floor_no            0
Occupancy_status    0
Floor_area          0
City                0
Price_in_taka       0
Location            0
Building            0
Appartment          0
Commercial          0
House               0
Cumilla_city        0
Dhaka_city          0
Gazipur_city        0
Narayanganj_city    0
vacant              0
dtype: int64
```

Final Shape of the Cleaned Dataset

After all the cleaning steps, the dataset was reduced to **2759 rows** and **18 columns**:

```
df_clean.shape
```

output:

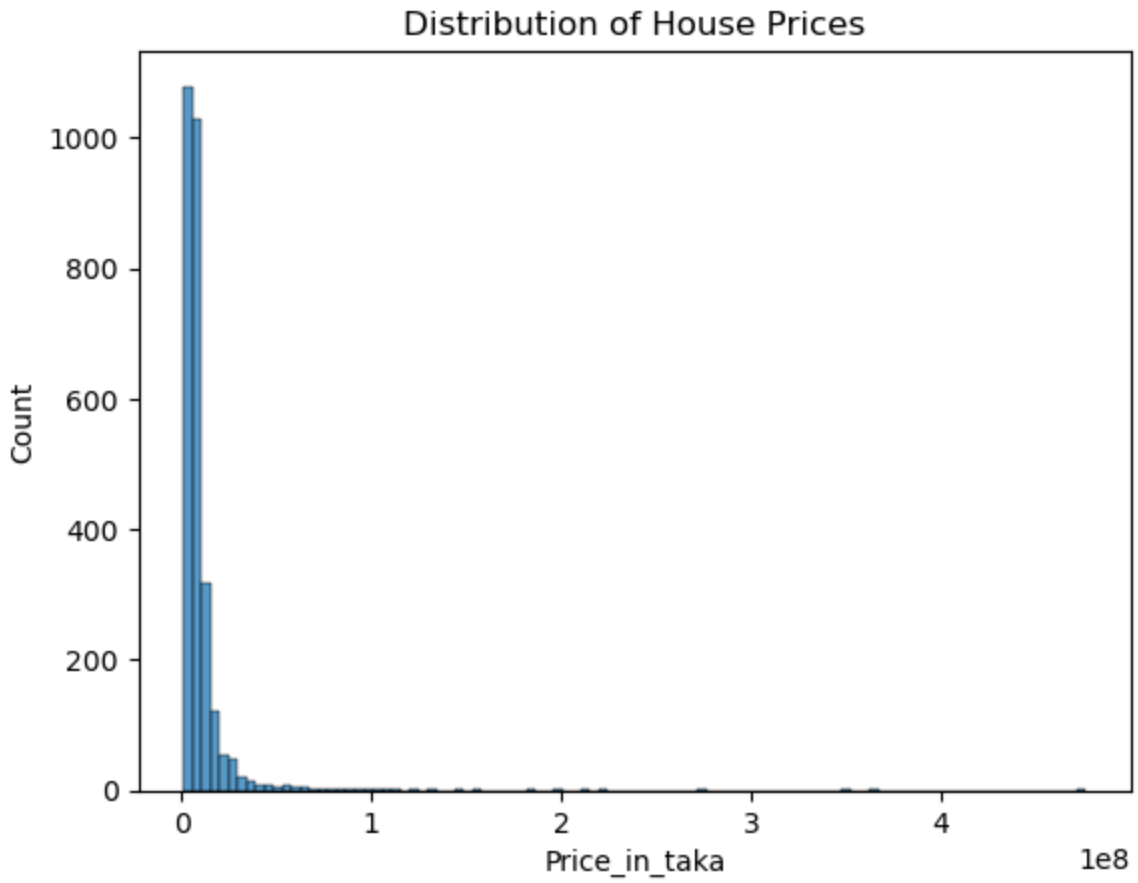
(2759, 18)

3. Exploratory data analysis (EDA)

Exploring the target

With the cleaned dataset in place, I moved on to exploratory data analysis (EDA). The first step was to explore the target variable, **Price_in_taka**, to understand its distribution. I plotted a histogram to visualize the distribution of house prices:

```
sns.histplot(df_clean['Price_in_taka'], bins = 100)
plt.title('Distribution of House Prices')
plt.show()
```

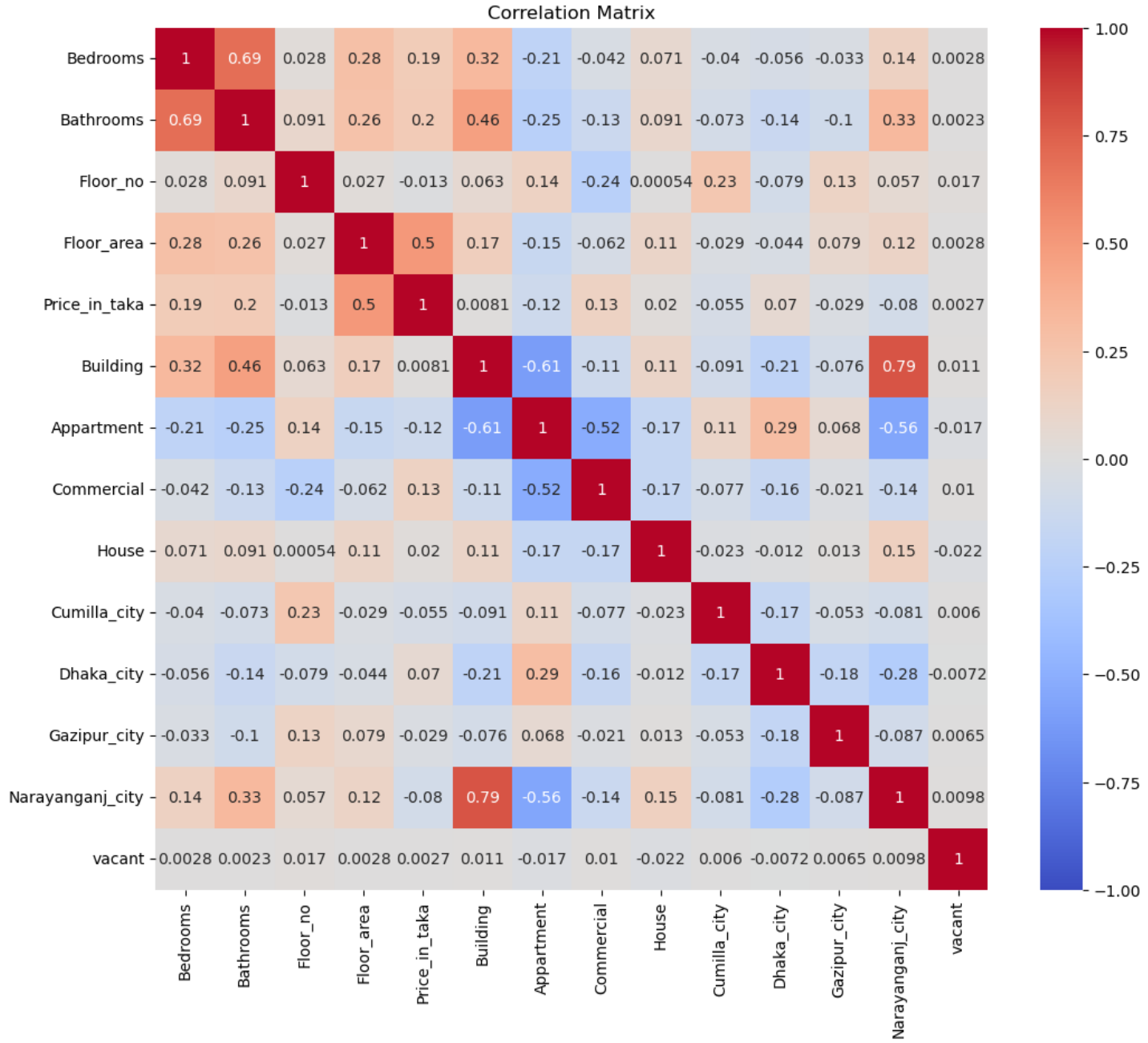


The histogram of **Price_in_taka** shows a left-skewed distribution, which is typical for price data. Most properties fall within a lower price range, while a few have very high prices. This skewness indicates the presence of some expensive properties that drive up the average. In such cases, log-transforming the target variable during modeling can help normalize the distribution and improve model performance.

Correlation Analysis

To explore the relationships between the numeric features, I created a **correlation matrix** using a heatmap. This allowed me to identify how strongly each feature correlates with the target variable:

```
plt.figure(figsize=(12, 10))
sns.heatmap(df_clean.corr(numeric_only = True), annot = True, vmin = -1, vmax = 1, cmap = "coolwarm")
plt.title('Correlation Matrix')
plt.show()
```



The heatmap illustrates the correlation between various numerical features and **Price_in_taka**. **Floor_area** stands out with a strong positive correlation, confirming its importance as a predictor. Other features, like **Bedrooms** and **Bathrooms**, also show moderate correlations. This matrix is useful for selecting which features to focus on during feature engineering and modeling, as strong correlations typically suggest a linear relationship.

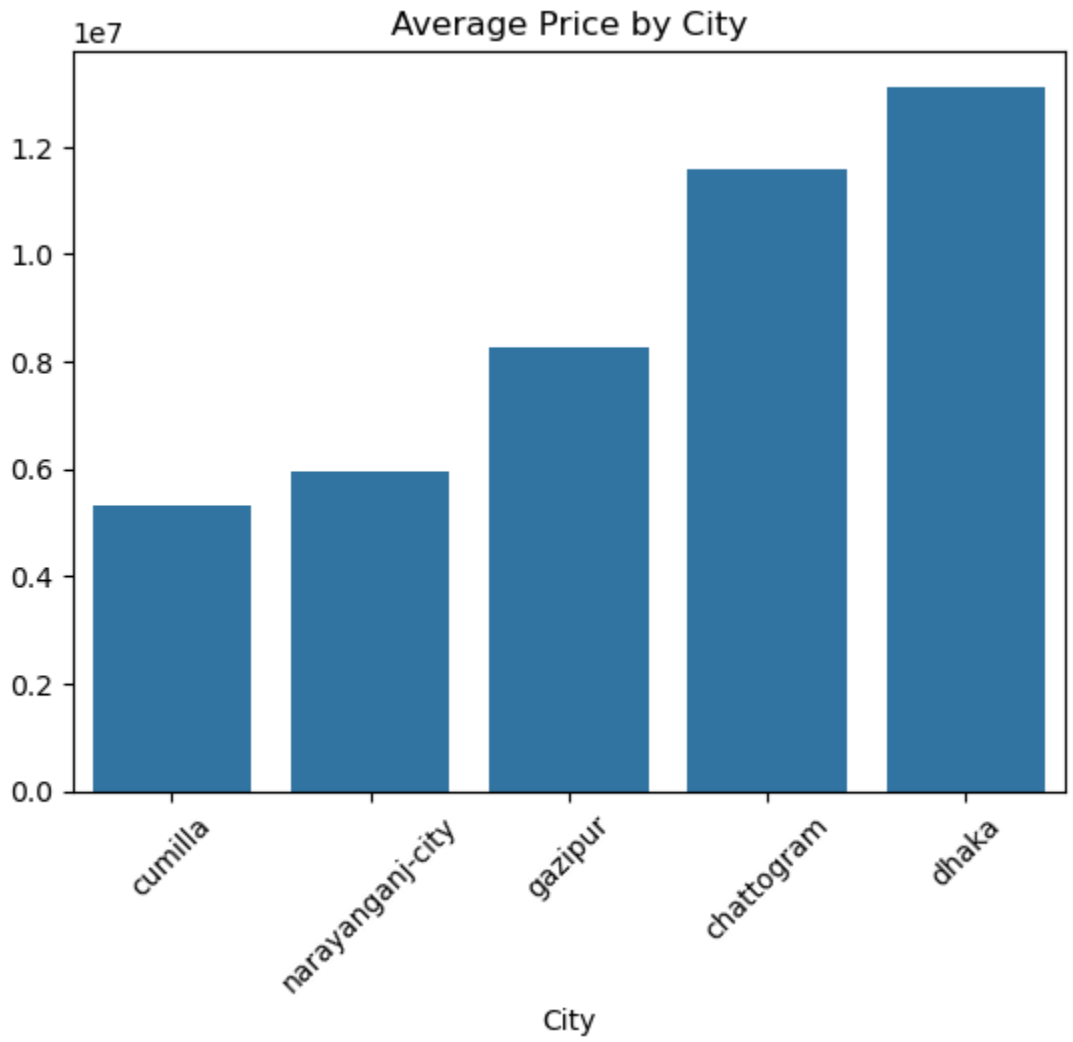
Exploring the Features

City-Wise Analysis

Next, I analyzed the average price of properties across different cities to see if there were any regional trends. I used a **barplot** to visualize the average price per city:

```
city_group = df_clean.groupby('City')['Price_in_taka'].mean().sort_values()
sns.barplot(x=city_group.index, y=city_group.values)
plt.title('Average Price by City')
```

```
plt.xticks(rotation=45)
plt.show()
```

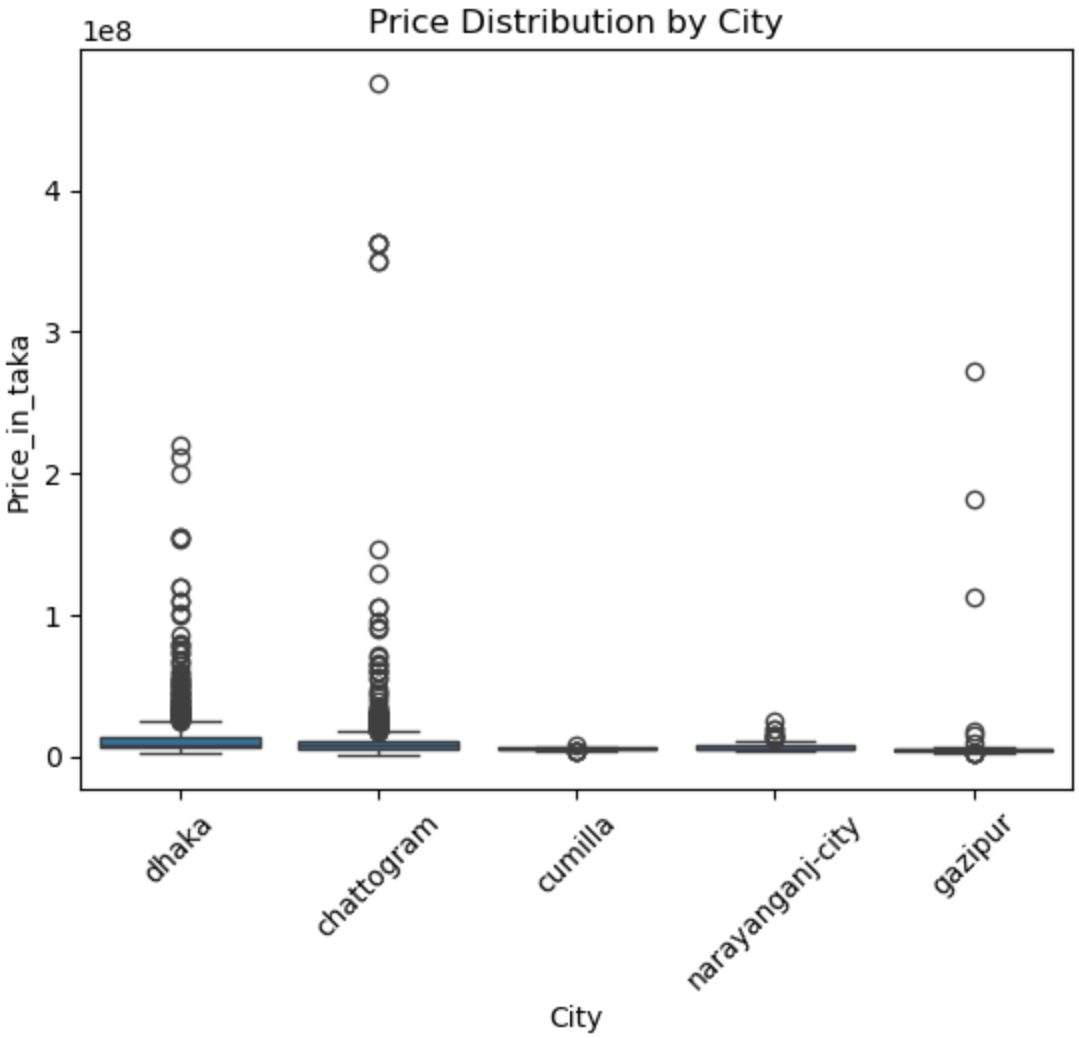


The barplot of average prices across cities reveals significant variation between regions. Dhaka stands out with the highest average property prices, likely due to its status as the capital city and economic hub. This regional analysis highlights that location is a critical factor in determining property prices, and it will be essential to include city-specific information in the model.

Price Distribution by City

I also visualized the distribution of prices for each city using a **boxplot** to see the range and variation of property prices within each city:

```
sns.boxplot(x='City', y='Price_in_taka', data=df_clean)
plt.title('Price Distribution by City')
plt.xticks(rotation=45)
plt.show()
```

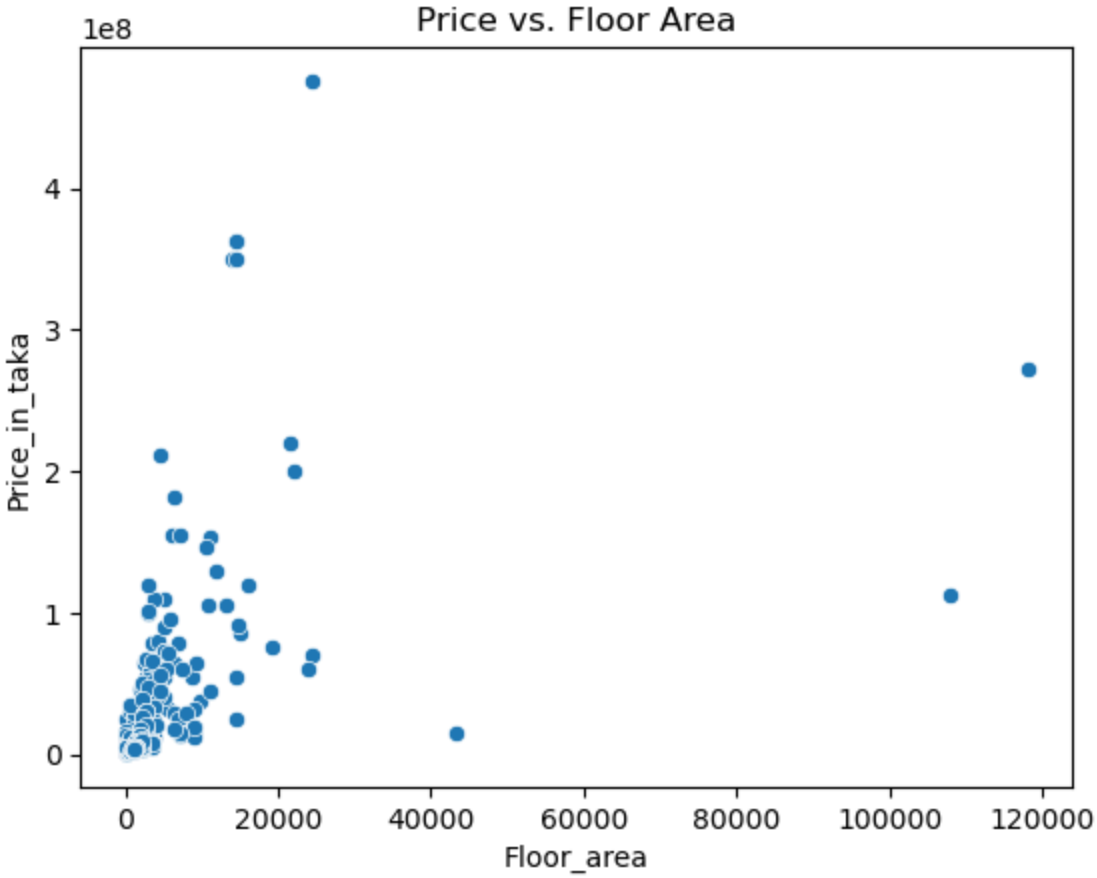


The boxplot provides a more granular view of the price distribution in each city. **Dhaka** shows a broad range of prices, with several outliers at the higher end. This suggests that while most properties fall within a moderate price range, some premium properties significantly inflate the average. This variation within cities suggests that using city as a categorical feature in the model could capture location-based price trends

Price vs. Floor Area

I examined the relationship between **Floor_area** and **Price_in_taka** with a scatter plot. This allowed me to assess whether larger floor areas were associated with higher property prices:

```
sns.scatterplot(x='Floor_area', y='Price_in_taka', data=df_clean)
plt.title('Price vs. Floor Area')
plt.show()
```

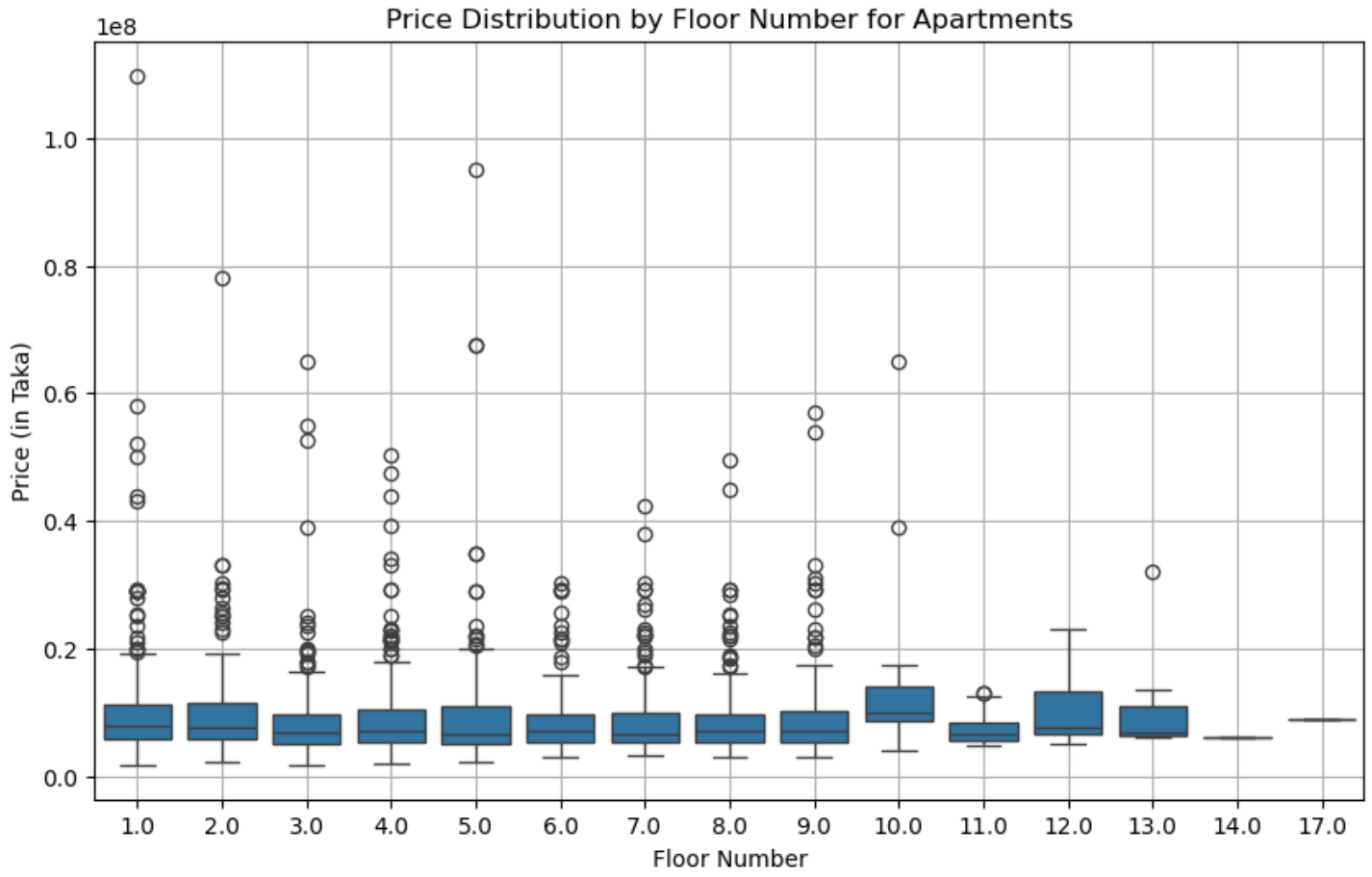


The scatterplot of **Price_in_taka** vs. **Floor_area** shows a positive relationship, confirming that larger properties tend to have higher prices. However, the spread of data points suggests that the relationship isn't strictly linear, with some properties commanding higher prices even with smaller floor areas. This may be due to additional factors like location, property type, or other amenities.

Price vs Floor Number for Apartments

I began by examining the relationship between **Price** and **Floor_no** for **apartments**. A boxplot was used to show the distribution of prices across different floors:

```
# Filter the data for apartments
df_apartments = df_clean[df_clean['Appartment'] == 1]

# Box plot for Apartments
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_apartments, x='Floor_no', y='Price_in_taka')
plt.title('Price Distribution by Floor Number for Apartments')
plt.xlabel('Floor Number')
plt.ylabel('Price (in Taka)')
plt.grid(True)
plt.show()
```



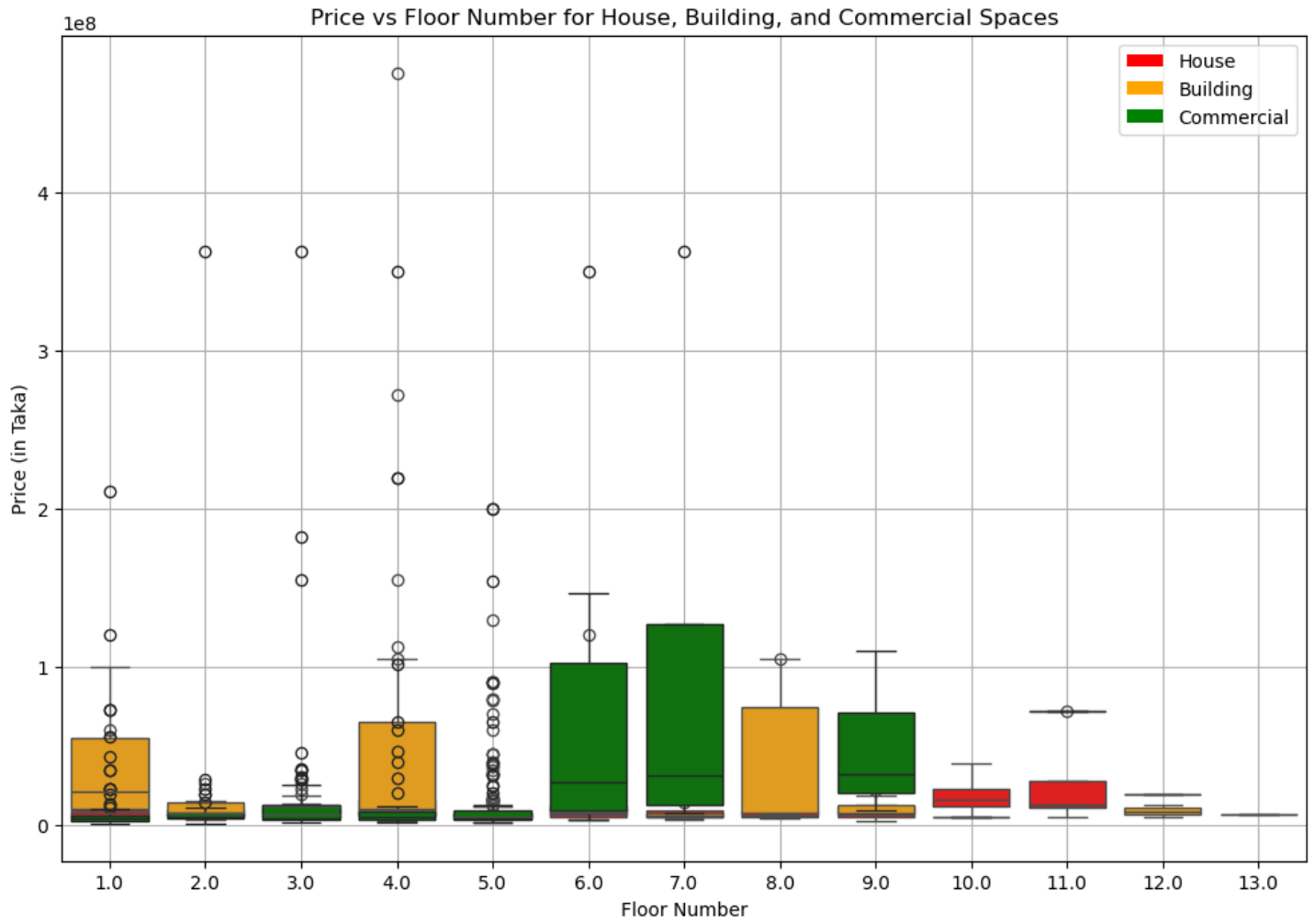
The boxplot of **Price_in_taka** vs. **Floor_no** for apartments shows the distribution of prices across different floors. There seems to be no clear pattern where higher floors consistently command higher prices. This suggests that floor number might not be a major determinant of price in apartments, or the effect could be masked by other factors like location and property type.

Price vs Floor Number for Houses, Buildings, and Commercial Spaces

Next, I looked at the relationship between **Price** and **Floor_no** for **houses**, **buildings**, and **commercial spaces**. I created separate boxplots for each property type:

```
# Filter the data for house, building, or commercial properties
df_others = df_clean[(df_clean['House'] == 1) | (df_clean['Building'] == 1) | (df_clean['Commercial'] == 1)]

plt.figure(figsize=(12, 8))

# Boxplot for house, building, and commercial, without using the 'label' argument
sns.boxplot(data=df_others[df_others['House'] == 1], x='Floor_no', y='Price_in_taka', color='red')
sns.boxplot(data=df_others[df_others['Building'] == 1], x='Floor_no', y='Price_in_taka', color='orange')
sns.boxplot(data=df_others[df_others['Commercial'] == 1], x='Floor_no', y='Price_in_taka', color='green')

# Titles and labels
plt.title('Price vs Floor Number for House, Building, and Commercial Spaces')
plt.xlabel('Floor Number')
plt.ylabel('Price (in Taka)')
plt.grid(True)
```

```
# Create a custom legend since label parameter is not available in sns.boxplot
from matplotlib.patches import Patch
legend_elements = [Patch(facecolor='red', label='House'),
                   Patch(facecolor='orange', label='Building'),
                   Patch(facecolor='green', label='Commercial')]

plt.legend(handles=legend_elements)

# Show the plot
plt.show()
```

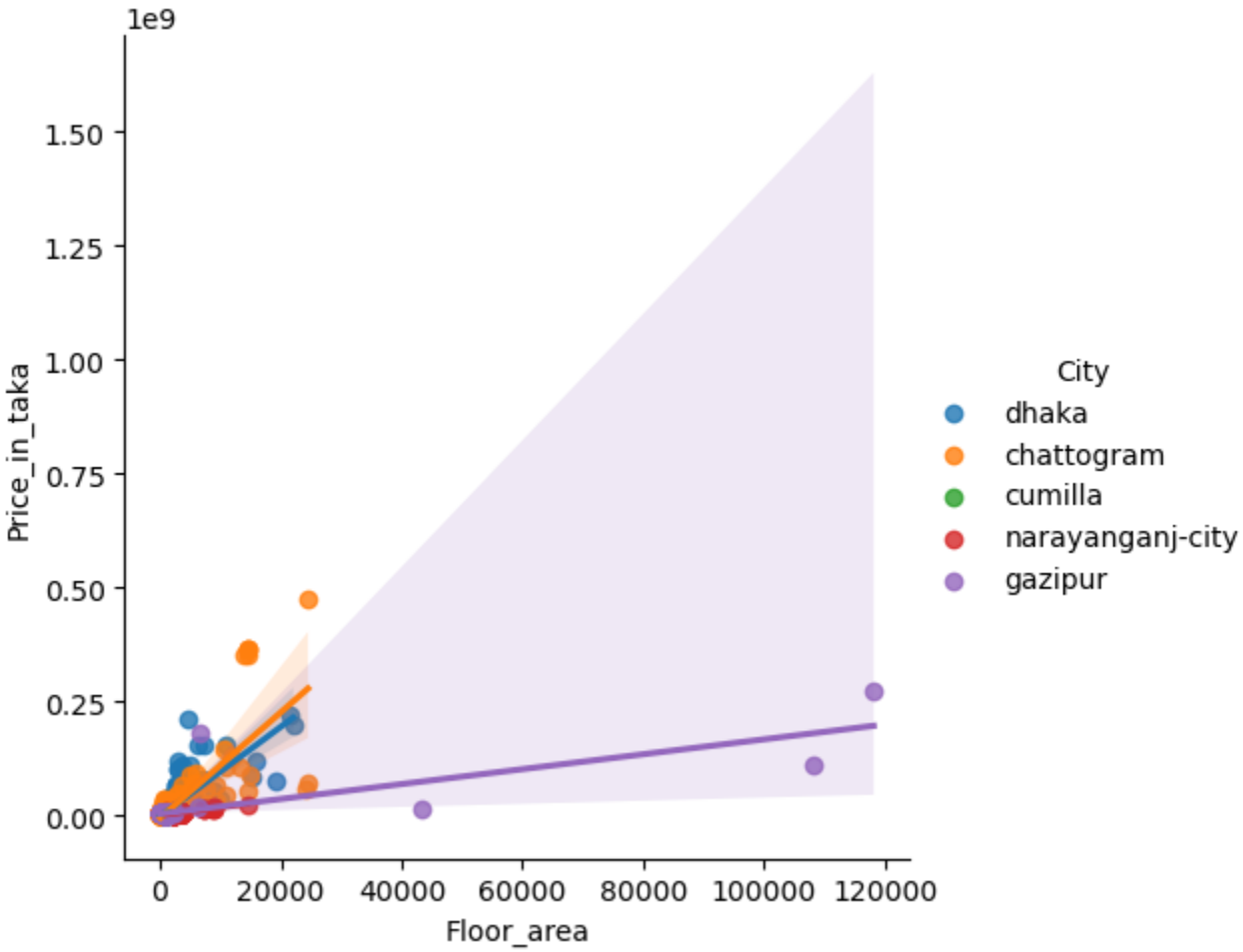


The combined boxplot for **houses**, **buildings**, and **commercial spaces** provides an interesting comparison. Each property type has distinct price patterns based on the floor number. For **commercial spaces**, higher floors appear to command lower prices, while for **houses** and **buildings**, the floor number seems to have less of an impact. This indicates that floor number might play different roles in determining prices depending on the type of property.

Price vs Floor Area with City as an Interaction Term

I explored the relationship between **Price** and **Floor_area** while factoring in the **City** as a color-coded interaction term using sns.lmplot():

```
sns.lmplot(df_clean, x = "Floor_area", y = "Price_in_taka", hue = "City");
```



The **lmplot** of **Price_in_taka** vs. **Floor_area**, with **City** as the interaction term, shows how the relationship between floor area and price varies across cities. The color coding by city provides a clear comparison, revealing that **Dhaka** has consistently higher prices for similar floor areas compared to other cities. This emphasizes the importance of location when predicting property prices.

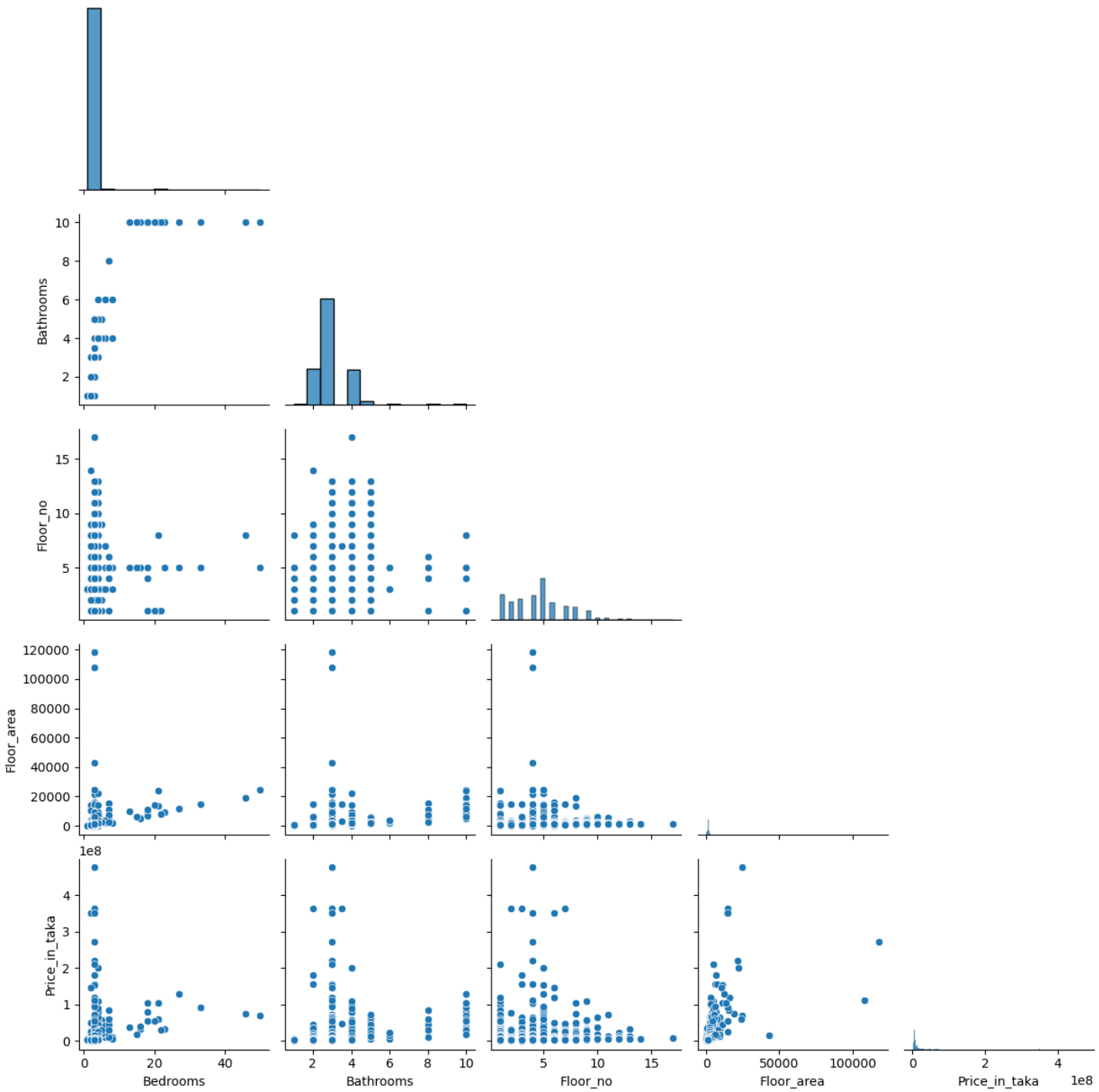
4. Preparing Data for Modeling

Before moving into the modeling phase, I created a new DataFrame containing relevant features for the regression model:

```
df_model = df_clean[["Bedrooms", "Bathrooms", "Floor_no", "Floor_area",
                     "Building", "Appartment", "Commercial", "House",
                     "Cumilla_city", "Dhaka_city", "Gazipur_city", "Narayanganj_city", "vacant", "Price_in_taka"]]
```

I then generated a **pairplot** to visualize relationships between key features and the target variable (Price_in_taka):

```
sns.pairplot(df_model[["Bedrooms", "Bathrooms", "Floor_no", "Floor_area", "Price_in_taka"]], corner = True)
```



5. Modeling

Simple Linear Regression

I started with a simple linear regression model, where **Floor_area** was the feature most correlated with the target variable **Price_in_taka**. I log-transformed both the feature and the target to stabilize variance and improve model accuracy:

```
import statsmodels.api as sm
from sklearn.metrics import mean_absolute_error as mae
from sklearn.metrics import r2_score as r2
from sklearn.model_selection import train_test_split
```

```
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
import scipy.stats as stats
from statsmodels.stats.outliers_influence import variance_inflation_factor as vif



# From correlation we see floor area is most correlated with price with 0.5

X = sm.add_constant(np.log(df_model[["Floor_area"]]))
y = np.log(df_model["Price_in_taka"])

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

kf = KFold(n_splits = 5, shuffle = True, random_state = 2024)

# Create a list to store validation scores for each fold
cv_lm_r2s = []
cv_lm_mae = []

# Loop in each fold in X and y
for train_ind, val_ind in kf.split(X, y):
    # Subset data based on CV folds
    X_train, y_train = X.iloc[train_ind], y.iloc[train_ind]
    X_val, y_val = X.iloc[val_ind], y.iloc[val_ind]
    # Fit the model on fold's training data
    model = sm.OLS(y_train, X_train).fit()
    # Append validation score to list
    cv_lm_r2s.append(r2(y_val, model.predict(X_val)))
    cv_lm_mae.append(np.exp(mae(y_val), np.exp(model.predict(X_val))))

print("All validation R2s: ", [round(x, 3) for x in cv_lm_r2s])
print(f"Cross validation R2s: {round(np.mean(cv_lm_r2s), 3)} +- {round(np.std(cv_lm_r2s), 3)}")

print("All validation MAEs: ", [round(x, 3) for x in cv_lm_mae])
print(f"Cross validation MAEs: {round(np.mean(cv_lm_mae), 3)} +- {round(np.std(cv_lm_mae), 3)}")
```

Output:

All validation R2s: [0.384, 0.281, 0.397, 0.322, 0.319]

Cross validation R2s: 0.341 +- 0.044

All validation MAEs: [5516230.491, 4111686.53, 6391301.513, 5603411.06, 6689574.22]

Cross validation MAEs: 5662440.763 +- 896207.763

Model Overview: This model uses **Floor_area** as the only predictor for **Price_in_taka**, both log-transformed to handle the skewed distribution of the target variable.

Performance

Cross Validation R^2: 0.341 ± 0.044

Cross Validation MAE: 5,662,440 ± 896,207

The **R^2** score is relatively low, indicating that the model explains only about 34% of the variance in house prices. This suggests that a single predictor, **Floor_area**, is insufficient to capture the complexity of the data.

While **Floor_area** is the most correlated feature, relying solely on it limits the model's predictive power. This simple model provides a baseline but isn't sufficient for robust predictions.

Multiple Linear Regression

Next, I engineered more features for a multiple linear regression model. Some of the new features included:

- Log transformations for the target and **Floor_area**
- Interactions between **Bedrooms**, **Bathrooms**, and **Floor_area**
- Polynomials for **Bedrooms** and **Bathrooms**

```
df_model = df_model.assign(
    price_log = np.log(df_model["Price_in_taka"]),
    Floor_area_log = np.log(df_model["Floor_area"]),
    Bedrooms_Bathrooms = df_model["Bedrooms"] + df_model["Bathrooms"],
    Buil_Commer_House = df_model["Building"] + df_model["Commercial"] + df_model["House"],
    Bedrooms_2 = df_model["Bedrooms"] ** 2,
    Bathrooms_2 = df_model["Bathrooms"] ** 2,
    Bedrooms_3 = df_model["Bedrooms"] ** 3,
    Bathrooms_3 = df_model["Bathrooms"] ** 3,
    Floor_no_2 = df_model["Floor_no"] ** 2,
    Bedrooms_log = np.log(df_model["Bedrooms"]),
    Bathrooms_log = np.log(df_model["Bathrooms"]),
    Bedrooms_Bathrooms_log = np.log(df_model["Bedrooms"] + df_model["Bathrooms"]),
    Bedrooms_Bathrooms_2 = (df_model["Bedrooms"] + df_model["Bathrooms"]) ** 2,
    Floor_area_number_ration = np.where(df_model["Appartment"] == 1, np.log(df_model["Floor_area"]),
                                        np.log(df_model["Floor_area"] / df_model["Floor_no"])),
    Bedrooms_Floor_area_log = df_model["Bedrooms"] * np.log(df_model["Floor_area"]),
    Bathrooms_Floor_area_log = df_model["Bathrooms"] * np.log(df_model["Floor_area"]),
    Bedrooms_Bathrooms_Floor_area_log = (df_model["Bedrooms"] + df_model["Bathrooms"]) *
np.log(df_model["Floor_area"]),
    Bedrooms_2_Floor_area_log = df_model["Bedrooms"] ** 2 * np.log(df_model["Floor_area"]),
    Bathrooms_2_Floor_area_log = df_model["Bathrooms"] ** 2 * np.log(df_model["Floor_area"]),
)
```

I also created bins for **Bedrooms**:

```
df_model["Bedroom_bins"] = pd.cut(
    df_model["Bedrooms"],
    bins=[0, 3, 6, 10, np.inf],
    labels=["1-3 Small", "4-6 Medium", "7-10 Large", "11+ Extra-large"])
```

Finally, I trained the multiple linear regression model:

```
X = sm.add_constant(df_model[["Bedrooms", "Bedrooms_2", "Bathrooms", "Bathrooms_2",
                              "Floor_area_log", "Appartment", "Floor_no",
```

```
                        "Cumilla_city", "Dhaka_city", "Gazipur_city", "Narayanganj_city"]])
y = df_model["price_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

kf = KFold(n_splits = 5, shuffle = True, random_state = 2024)

# Create a list to store validation scores for each fold
cv_lm_r2s = []
cv_lm_mae = []

# Loop in each fold in X and y
for train_ind, val_ind in kf.split(X, y):
    # Subset data based on CV folds
    X_train, y_train = X.iloc[train_ind], y.iloc[train_ind]
    X_val, y_val = X.iloc[val_ind], y.iloc[val_ind]
    # Fit the model on fold's training data
    model = sm.OLS(y_train, X_train).fit()
    # Append validation score to list
    cv_lm_r2s.append(r2(y_val, model.predict(X_val)))
    cv_lm_mae.append(mae(np.exp(y_val), np.exp(model.predict(X_val))))

print("All validation R2s: ", [round(x, 3) for x in cv_lm_r2s])
print(f"Cross validation R2s: {round(np.mean(cv_lm_r2s), 3)} +- {round(np.std(cv_lm_r2s), 3)}")

print("All validation MAEs: ", [round(x, 3) for x in cv_lm_mae])
print(f"Cross validation MAEs: {round(np.mean(cv_lm_mae), 3)} +- {round(np.std(cv_lm_mae), 3)}")
```

output:

All validation R2s: [0.719, 0.631, 0.726, 0.648, 0.705]

Cross validation R2s: 0.686 +- 0.039

All validation MAEs: [3878698.407, 2843080.774, 4637207.744, 3896625.063, 4810351.864]

Cross validation MAEs: 4013192.77 +- 696519.369

Model Overview: This model incorporates multiple features, including interactions between **Bedrooms**, **Bathrooms**, and **Floor_area**. It also includes polynomial terms for **Bedrooms** and **Bathrooms** to capture non-linear relationships.

Performance:

Cross Validation R^2: 0.686 ± 0.039

Cross Validation MAE: 4,013,192 ± 696,519

The **R^2** is significantly improved compared to the simple linear model, explaining about 68.6% of the variance in the target. The lower **MAE** also indicates that the model predictions are closer to the actual values.

The multiple linear regression model shows a significant improvement due to the inclusion of more features and polynomial terms. It captures more of the complexity in the data, making it a much better choice than the simple model.

After developing the multiple linear regression model, I explored several advanced regression techniques to enhance predictive performance:

1. Ridge Regression

Initially, I attempted to apply Ridge Regression using a selection of the best-performing features. However, I observed only a minimal improvement in performance. To fully assess the model's capabilities, I then included all engineered features in the Ridge Regression model, resulting in a notable performance increase of approximately 7%.

2. Lasso Regression

Following Ridge Regression, I implemented Lasso Regression. This model is particularly useful for feature selection, as it shrinks some coefficients to zero, effectively eliminating less significant features. By analyzing the coefficients from the Lasso model, I identified which features contributed the most to the predictions.

3. Elastic Net Regression

Next, I used Elastic Net, which combines the strengths of both Ridge and Lasso regressions. This hybrid approach allows for regularization while also performing variable selection. Elastic Net is particularly beneficial when dealing with datasets that have multicollinearity.

4. Model Performance Evaluation

To determine the best model for my dataset, I evaluated the performance of Ridge, Lasso, Elastic Net, and the multiple linear regression models. The evaluation criteria included:

- **R^2 Score**: Indicates the proportion of variance in the dependent variable that can be explained by the independent variables.
- **Mean Absolute Error (MAE)**: Represents the average absolute differences between the predicted and actual values, providing insights into the model's accuracy.

After thorough comparison and validation, the performance metrics for each model were carefully analyzed to select the most effective approach for predicting house prices.

Regularization Techniques

Ridge regression

```
X = sm.add_constant(df_model.drop(["Price_in_taka", "price_log", "Bedroom_bins",
                                   "vacant", "Floor_area", "Floor_area_number_ration",
                                   "Bedrooms_log", "Floor_no", "Bathrooms", "Floor_no",
                                   "Bedrooms_Bathrooms", "Buil_Commer_House",
                                   "Bedrooms_2", "Bedrooms_Bathrooms_2",
                                   "Bathrooms_2", "Bedrooms_Bathrooms_log",
                                   "Bedrooms_Floor_area_log", "Bathrooms_2_Floor_area_log",
```

```
                                    "Bedrooms_Bathrooms_Floor_area_log",
                                    "Bedrooms_2_Floor_area_log"], axis = 1))
y = df_model["price_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

std = StandardScaler()
X_tr = std.fit_transform(X.values)
X_te = std.transform(X_test.values)

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

ridge_model = RidgeCV(alphas = alphas, cv = 5)

ridge_model.fit(X_tr, y)
print(f"Alpha: {ridge_model.alpha_}")
print(f"Train R²: {ridge_model.score(X_tr, y)}")
print(f"Mean absolute error: {mae(np.exp(y), np.exp(ridge_model.predict(X_tr)))}")
```

output:

Alpha: 1.2750512407130128

Train R^2: 0.7631584775507354

Mean absolute error: 3468979.236352783

Model Overview: Ridge regression adds regularization to control for overfitting by penalizing large coefficients.

Performance:

Train R^2: 0.763

MAE: 3,468,979

Ridge regression improves the model's performance by 7%, which suggests that some degree of regularization helps improve generalization.

Ridge regression performs better than simple and multiple linear regression by controlling for overfitting. It slightly outperforms the multiple linear regression model in terms of error metrics and is a solid option if overfitting is a concern.

Lasso regression

```
from sklearn.linear_model import LassoCV

X = sm.add_constant(df_model.drop(["Price_in_taka", "price_log", "Bedroom_bins",
                                   "vacant", "Floor_area", "Floor_area_number_ration",
                                   "Bedrooms_log", "Floor_no", "Bathrooms", "Floor_no",
                                   "Bedrooms_Bathrooms", "Buil_Commer_House",
```

```
                                    "Bedrooms_2", "Bedrooms_Bathrooms_2",
                                    "Bathrooms_2", "Bedrooms_Bathrooms_log",
                                    "Bedrooms_Floor_area_log", "Bathrooms_2_Floor_area_log",
                                    "Bedrooms_Bathrooms_Floor_area_log",
                                    "Bedrooms_2_Floor_area_log"], axis = 1))
y = df_model["price_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

std = StandardScaler()
X_tr = std.fit_transform(X.values)
X_te = std.transform(X_test.values)

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

Lasso_model = LassoCV(alphas = alphas, cv = 5)

Lasso_model.fit(X_tr, y)
print(f"Alpha: {Lasso_model.alpha_}")
print(f"Train R²: {Lasso_model.score(X_tr, y)}")
print(f"Mean absolute error: {mae(np.exp(y), np.exp(Lasso_model.predict(X_tr)))}")
```

output:

Alpha: 0.001

Train R^2: 0.7622230595108627

Mean absolute error: 3466443.13665751

Model Overview: Lasso regression performs feature selection by shrinking some coefficients to zero, effectively choosing only the most important features.

Performance:

Train R^2: 0.762

MAE: 3,466,443

The performance of Lasso is similar to Ridge, also we notice that alpha is 0.001 so nearly no regularization.

Lasso regression is useful when you want a more interpretable model, as it automatically reduces less relevant features. It performs similarly to Ridge, so either could be a good choice depending on the context.

Elastic net regression

```
from sklearn.linear_model import ElasticNetCV

X = sm.add_constant(df_model.drop(["Price_in_taka", "price_log", "Bedroom_bins",
                                    "vacant", "Floor_area", "Floor_area_number_ration",
                                    "Bedrooms_log", "Floor_no", "Bathrooms", "Floor_no",
```

```
                                    "Bedrooms_Bathrooms", "Buil_Commer_House",
                                    "Bedrooms_2", "Bedrooms_Bathrooms_2",
                                    "Bathrooms_2", "Bedrooms_Bathrooms_log",
                                    "Bedrooms_Floor_area_log", "Bathrooms_2_Floor_area_log",
                                    "Bedrooms_Bathrooms_Floor_area_log",
                                    "Bedrooms_2_Floor_area_log"], axis = 1))
y = df_model["price_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

std = StandardScaler()
X_tr = std.fit_transform(X.values)
X_te = std.transform(X_test.values)

alphas = 10 ** np.linspace(-3, 3, 200)
l1_ratios = np.linspace(.1, 1, 10)

enet_model = ElasticNetCV(alphas = alphas, l1_ratio = l1_ratios, cv = 5)

enet_model.fit(X_tr, y)

print(f"Alpha: {enet_model.alpha_}")
print(f"Lambda: {enet_model.alpha_}")
print(f"Train R²: {enet_model.score(X_tr, y)}")
print(f"Mean absolute error: {mae(np.exp(y), np.exp(enet_model.predict(X_tr)))}")
```

output:

Alpha: 0.001

Lambda: 0.001

Train R^2: 0.7627697500503838

Mean absolute error: 3463015.244198938

Model Overview: Elastic Net is a hybrid of Ridge and Lasso, combining both L1 and L2 regularization.

Performance:

Train R^2: 0.763

MAE: 3,463,015

Elastic Net delivers similar performance to Ridge and Lasso, alpha 0.001 & lambda 0.001 so we are using only ridge regularization.

Elastic Net strikes a balance between Ridge and Lasso, making it a versatile model for situations where you want both regularization and automatic feature selection. Its performance is on par with the other regularization methods

And finaly linear regression giving the same features for the previous 3 models:

```
from sklearn.linear_model import ElasticNetCV

X = sm.add_constant(df_model.drop(["Price_in_taka", "price_log", "Bedroom_bins",
                                   "vacant", "Floor_area", "Floor_area_number_ration",
                                   "Bedrooms_log", "Floor_no", "Bathrooms", "Floor_no",
                                   "Bedrooms_Bathrooms", "Buil_Commer_House",
                                   "Bedrooms_2", "Bedrooms_Bathrooms_2",
                                   "Bathrooms_2", "Bedrooms_Bathrooms_log",
                                   "Bedrooms_Floor_area_log", "Bathrooms_2_Floor_area_log",
                                   "Bedrooms_Bathrooms_Floor_area_log",
                                   "Bedrooms_2_Floor_area_log"], axis = 1))

y = df_model["price_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

kf = KFold(n_splits = 5, shuffle = True, random_state = 2024)

# Create a list to store validation scores for each fold
cv_lm_r2s = []
cv_lm_mae = []

# Loop in each fold in X and y
for train_ind, val_ind in kf.split(X, y):
    # Subset data based on CV folds
    X_train, y_train = X.iloc[train_ind], y.iloc[train_ind]
    X_val, y_val = X.iloc[val_ind], y.iloc[val_ind]
    # Fit the model on fold's training data
    model = sm.OLS(y_train, X_train).fit()
    # Append validation score to list
    cv_lm_r2s.append(r2(y_val, model.predict(X_val)))
    cv_lm_mae.append(mae(np.exp(y_val), np.exp(model.predict(X_val))))

print("All validation R2s: ", [round(x, 3) for x in cv_lm_r2s])
print(f"Cross validation R2s: {round(np.mean(cv_lm_r2s), 3)} +- {round(np.std(cv_lm_r2s),
3)}")

print("All validation MAEs: ", [round(x, 3) for x in cv_lm_mae])
print(f"Cross validation MAEs: {round(np.mean(cv_lm_mae), 3)} +- {round(np.std(cv_lm_mae),
3)}")
```

output:

All validation R2s: [0.791, 0.747, 0.764, 0.702, 0.748]

Cross validation R2s: 0.75 +- 0.029

All validation MAEs: [3156743.592, 2295671.461, 6114911.578, 3679431.634, 4523251.419]

Cross validation MAEs: 3954001.937 +- 1300357.269

Final Model Selection

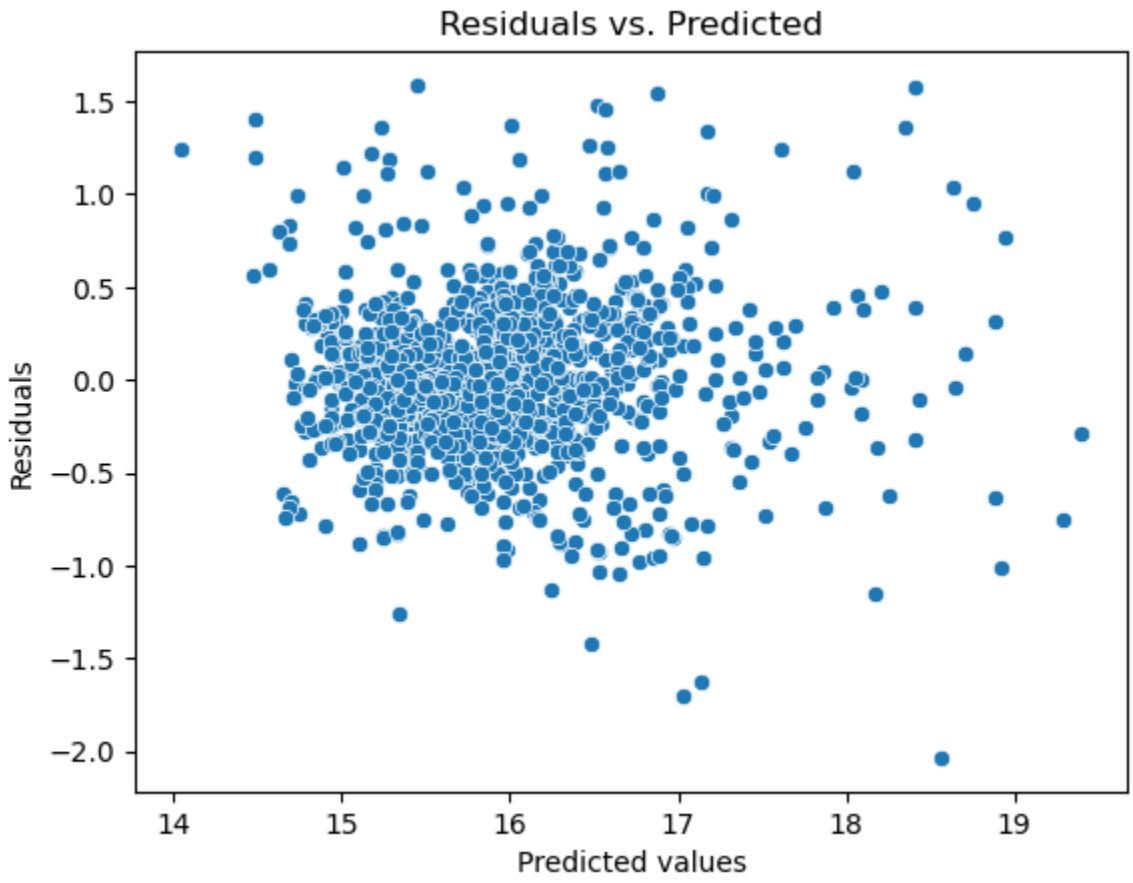
Based on the comparative analysis of Ridge Regression, Lasso Regression, Elastic Net, and Multiple Linear Regression, I selected the model that offered the best balance between predictive accuracy and interpretability.

Conclusion: The **Ridge Regression** model emerged as the most reliable choice, delivering improved performance while controlling for overfitting.

Checking linear regression assumptions

Equal variance of error:

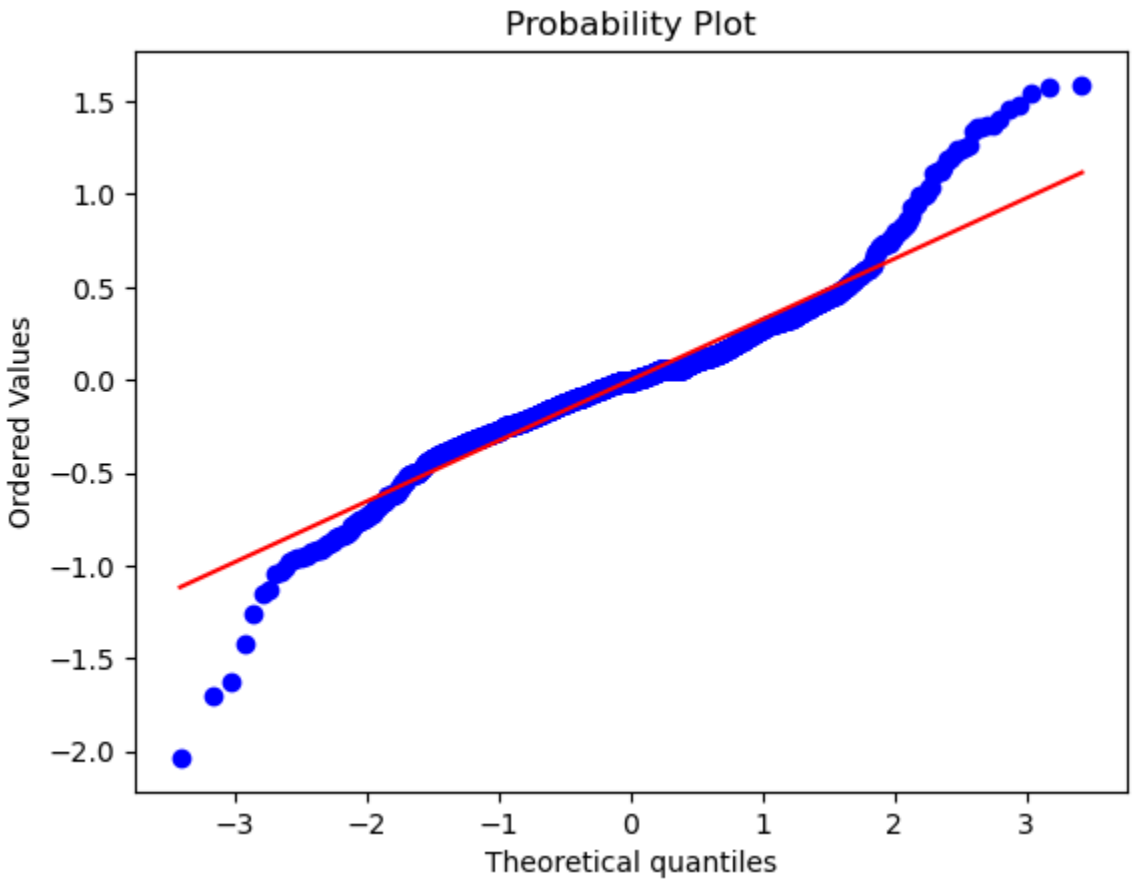
```
# Calculate the residuals
residuals = y - ridge_model.predict(X_tr)

# Plot predicted values vs. residuals
sns.scatterplot(x=ridge_model.predict(X_tr), y=residuals)
plt.title('Residuals vs. Predicted')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.show()
```



The errors are evenly spread along the x-axis, indicating that the assumption of homoscedasticity (equal variance of errors) is met.

Normality of errors:

```
stats.probplot(residuals, dist = "norm", plot = plt);
```



The points fall along the line within -2 to +2 standard deviations, indicating that the errors follow a normal distribution.

No perfect multicollinearity:

```
variables = X

# Calculate VIF for each feature
vif_data = pd.Series([vif(variables, i) for i in range(variables.shape[1])], index=X.columns)

# Display the VIF results
print(vif_data)
```

output:

All variance inflation factor (VIF) values are below 5, indicating that multicollinearity is not a major concern. Some features appear multiple times due to feature engineering (e.g., polynomial terms), but this is expected and does not suggest perfect multicollinearity.

Testing of the model

```
print(f"Test R²: {ridge_model.score(X_te, y_test)}")
print(f"Test Mean absolute error: {mae(np.exp(y_test), np.exp(ridge_model.predict(X_te)))}")
```

output:

Test R^2: 0.7759

Test Mean absolute error: 3,392,546 Regularization Techniques