Data Science Challenge

In [65]:
```
# If you'd like to install packages that aren't installed by default, uncomment the last two lines of this cell and r
# This will ensure your notebook has all the dependencies and works everywhere

import sys
!{sys.executable} -m pip install --upgrade scikit-learn
```

```
Requirement already up-to-date: scikit-learn in /opt/conda/lib/python3.7/site-packages (1.0.2)
Requirement already satisfied, skipping upgrade: joblib>=0.11 in /opt/conda/lib/python3.7/site-packages (from scikit-l
earn) (0.17.0)
Requirement already satisfied, skipping upgrade: scipy>=1.1.0 in /opt/conda/lib/python3.7/site-packages (from scikit-l
earn) (1.3.3)
Requirement already satisfied, skipping upgrade: numpy>=1.14.6 in /opt/conda/lib/python3.7/site-packages (from scikit-
learn) (1.19.2)
Requirement already satisfied, skipping upgrade: threadpoolctl>=2.0.0 in /opt/conda/lib/python3.7/site-packages (from
scikit-learn) (3.1.0)
```

In [ ]:
```
```

In [66]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [1]:
```
#Libraries
import pandas as pd
pd.set_option("display.max_columns", 101)
```

Data Description

Column	Description
`id`	Unique id of every property
`area`	Area of the structure (sq. feet)
`bathrooms`	Number of bathrooms

Column	Description
`bedrooms`	Number of rooms
`condo_fee`	Condominium fee `(US $$)`
`parking_spots`	Number of parking spots
`attached_rooms`	Number of bedrooms with direct access to a bathroom.
`type`	Kind of real estate
`lat`	Latitude of the property
`lon`	Longitude of the property
`year_built`	Year the real property was built
`overall_condition`	Abstract evaluation of the owner
`has_elevator`	If the property has elevator or not
`leasures_available`	List of leasures in the property
`price`	Price of the real estate `(US $$)`

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

EDA

In [395…
```
# Dataset is already loaded below
data = pd.read_csv("train.csv")
```

In [396…
```
data.head()
```

Out[396]:

	id	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_built
0	794	380	6.0	4.0	0.0	3.0	3.0	house	-5.855204	-35.220087	1974.0
1	199	60	2.0	2.0	0.0	1.0	1.0	apartment	-5.808983	-35.227251	NaN
2	1849	430	NaN	NaN	0.0	8.0	4.0	house	-5.843724	NaN	1980.0
3	33	55	2.0	2.0	280.0	0.0	1.0	apartment	-5.882136	-35.172217	1991.0
4	179	50	2.0	2.0	420.0	1.0	2.0	apartment	-5.866124	-35.182770	2002.0

In [334…
```
#Explore columns
data.columns
```

Out[334]:
```
Index(['id', 'area', 'bathrooms', 'bedrooms', 'condo_fee', 'parking_spots',
       'attached_rooms', 'type', 'lat', 'lon', 'year_built',
       'overall_condition', 'has_elevator', 'leasures_available', 'price'],
      dtype='object')
```

In [335…
```
#Description
data.describe()
```

Out[335]:

	id	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	lat	l
count	4056.000000	4056.000000	4036.000000	4030.000000	4044.000000	4046.000000	4044.000000	4045.000000	4041.0000
mean	2377.640039	107.705621	2.268831	2.635732	354.116222	1.618389	1.369683	-5.835676	-35.2035
std	1386.164798	95.332348	1.240710	0.835317	366.596604	1.032332	1.018299	0.034061	0.0165
min	0.000000	20.000000	0.000000	0.000000	0.000000	-7.000000	0.000000	-5.899991	-35.2449
25%	1177.750000	57.000000	1.000000	2.000000	0.000000	1.000000	1.000000	-5.866124	-35.2138
50%	2362.500000	91.000000	2.000000	3.000000	320.000000	2.000000	1.000000	-5.828402	-35.2032
75%	3592.250000	128.250000	3.000000	3.000000	561.000000	2.000000	2.000000	-5.808077	-35.1967
max	4771.000000	4200.000000	22.000000	22.000000	2500.000000	24.000000	22.000000	-5.726315	-35.1702

In [336…

```
data['year_built'] = data['year_built'].astype(str)
data['year_built'] = data['year_built'].replace('nan', np.nan)
```

In [337…

```
#checking null counts in each column
null_counts = data.isnull().sum()

# Print the result
print("Null counts per column:")
print(null_counts)
```

```
Null counts per column:
id                       0
area                     0
bathrooms               20
bedrooms                26
condo_fee               12
parking_spots           10
attached_rooms          12
type                    10
lat                     11
lon                     15
year_built            1523
overall_condition        0
has_elevator             0
leasures_available    1429
price                    0
dtype: int64
```

In [338…
```
#nearly half of year_built and leasures_available values are NULL
```

In [ ]:
```
```

In [339…
```
obj = (data.dtypes == 'object')
object_cols = list(obj[obj].index)
print("Categorical variables:",len(object_cols))

int_ = (data.dtypes == 'int')
num_cols = list(int_[int_].index)
print("Integer variables:",len(num_cols))

fl = (data.dtypes == 'float')
fl_cols = list(fl[fl].index)
print("Float variables:",len(fl_cols))
```

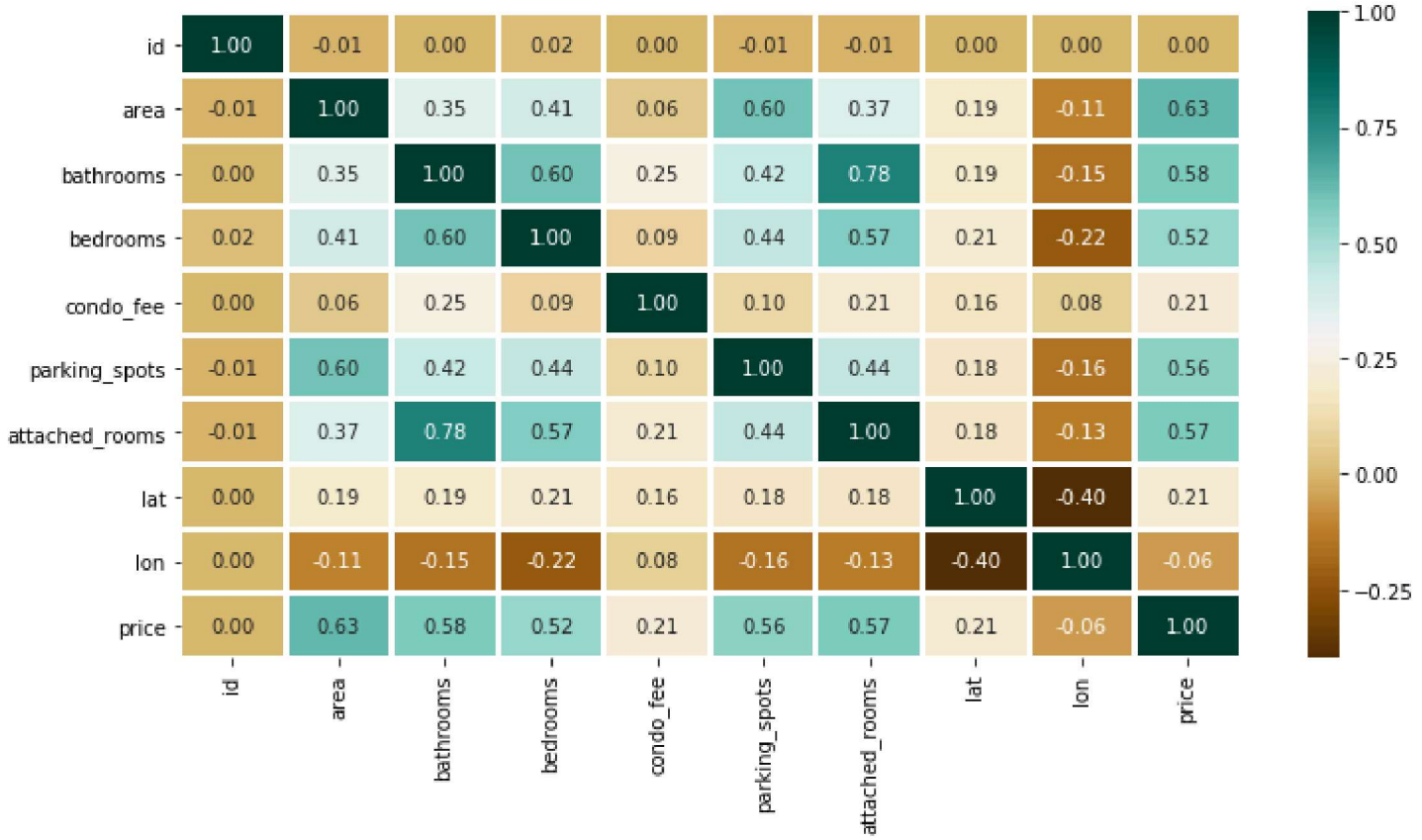
```
Categorical variables: 5
Integer variables: 3
Float variables: 7
```

In [ ]:
```
```

Plotting correlations between numeric type variables

In [340…
```
plt.figure(figsize=(12, 6))
sns.heatmap(data.corr(),
            cmap = 'BrBG',
            fmt = '.2f',
            linewidths = 2,
            annot = True)
```

Out[340]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7987934290>



In [ ]:
```
# lon and year_built has lowest correlations with our target variable (price).
# area and bathrooms has strongest correlation with price.
```

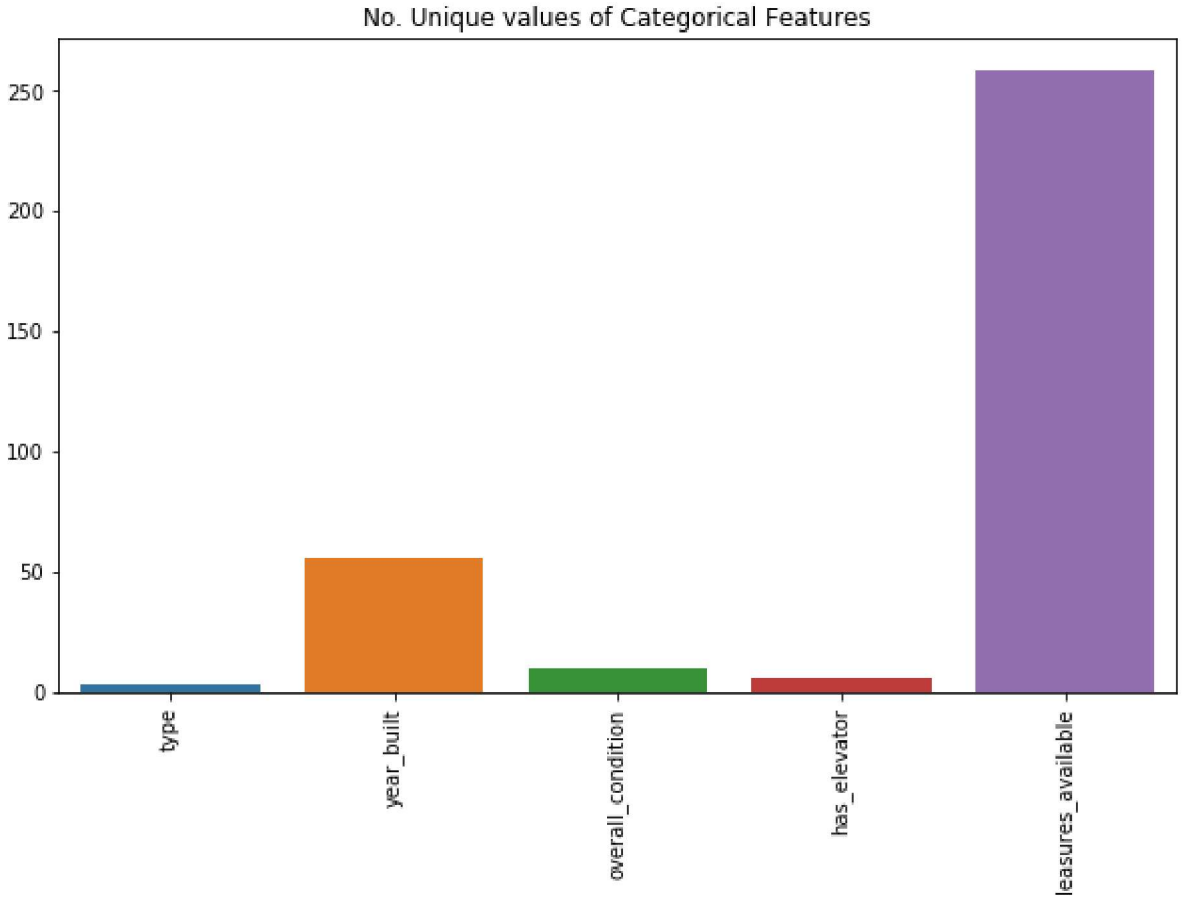
In [ ]:
```

```

Plotting unique values count and distribution for categorical columns

In [341…
```
unique_values = []
for col in object_cols:
  unique_values.append(data[col].unique().size)
plt.figure(figsize=(10,6))
plt.title('No. Unique values of Categorical Features')
plt.xticks(rotation=90)
sns.barplot(x=object_cols,y=unique_values)
```

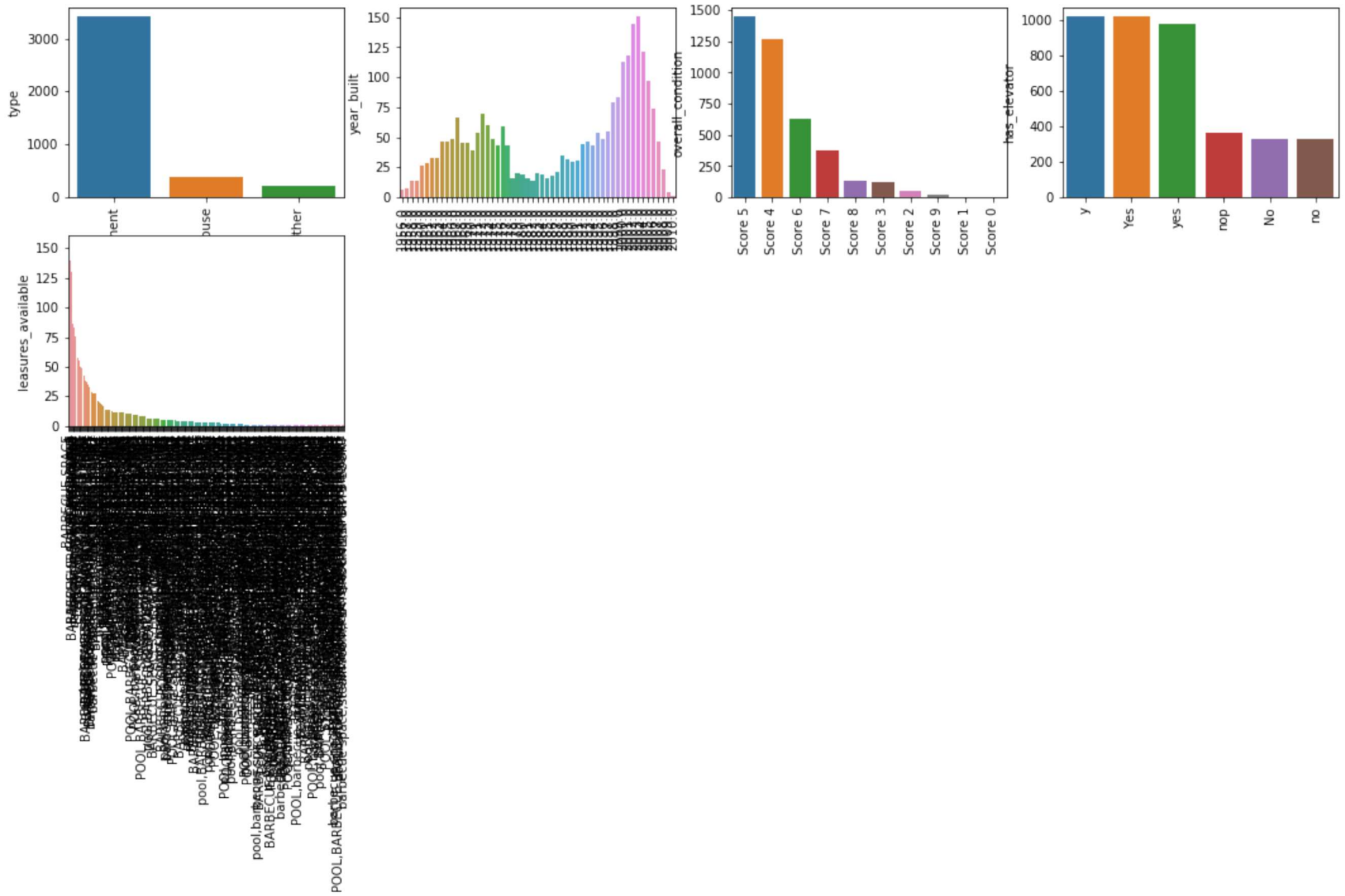
Out[341]: <matplotlib.axes._subplots.AxesSubplot at 0x7f798777bd50>



In [342…
```
#count of unique values for each categorical columns
plt.figure(figsize=(18, 36))
plt.title('Categorical Features: Distribution')
```

```
plt.xticks(rotation=90)
index = 1

for col in object_cols:
    y = data[col].value_counts()
    plt.subplot(11, 4, index)
    plt.xticks(rotation=90)
    sns.barplot(x=list(y.index), y=y)
    index += 1
```



In [ ]:

Plotting unique values count and distribution for categorical columns (after lowercasing leasure_available values)

In [343…

```
unique_count = data['leasures_available'].nunique()
print("Number of unique values in 'leasures_available':", unique_count)
```

```
Number of unique values in 'leasures_available': 258
```

In [344…

```
#lowercase all values in leasures_available
data['leasures_available'] = data['leasures_available'].str.lower()

unique_count = data['leasures_available'].nunique()
print("Number of unique values in 'leasures_available':", unique_count)
```

```
Number of unique values in 'leasures_available': 53
```

In [345…

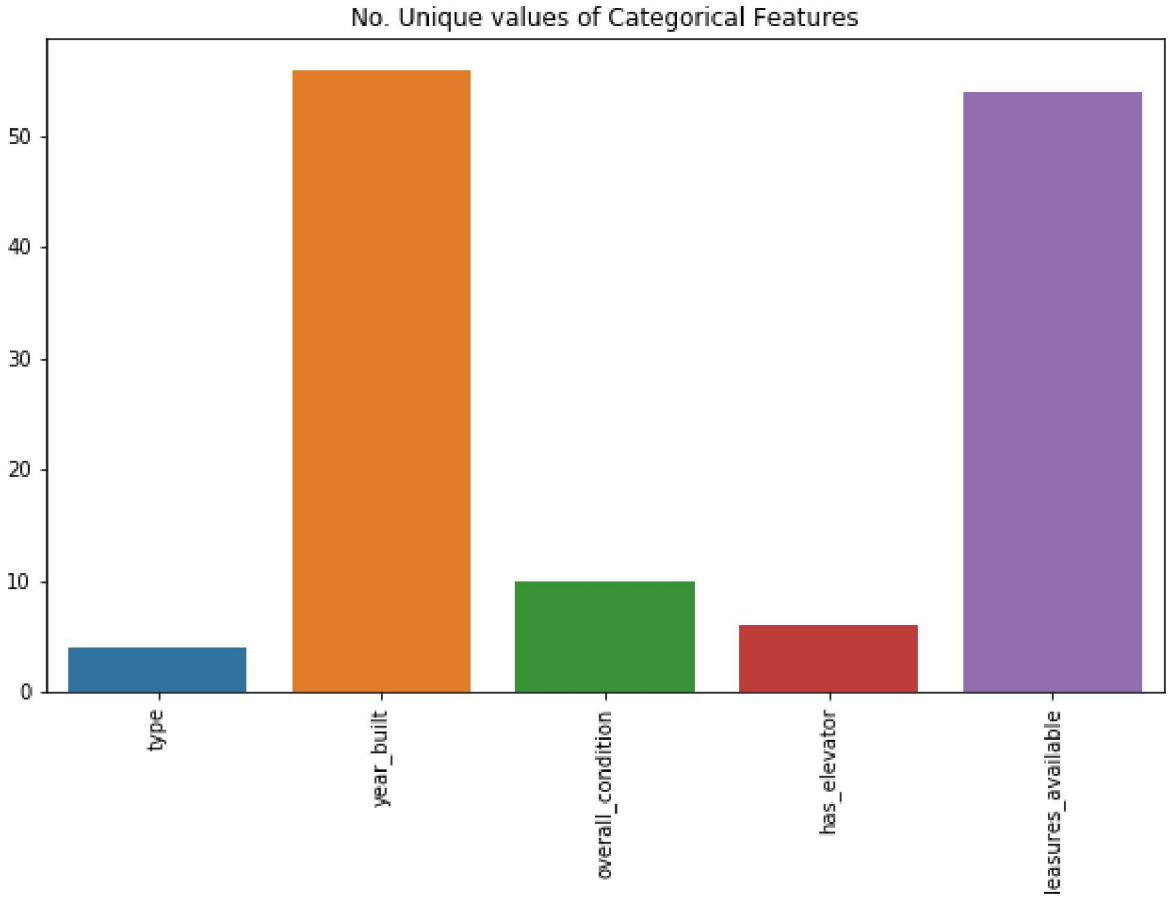
```
unique_values = data['leasures_available'].unique()
print("Unique values in 'leasures_available':")
print(unique_values)
```

```
Unique values in 'leasures_available':
[nan 'barbecue space' 'sports court' 'barbecue space,playground'
 'playground,sports court' 'pool' 'barbecue space,playground,sports court'
 'playground' 'pool,barbecue space,playground,sports court'
 'barbecue space,sports court'
 'pool,barbecue space,gym,playground,sports court' 'pool,playground'
 'pool,barbecue space,playground' 'pool,barbecue space'
 'pool,barbecue space,gym,playground' 'pool,sports court'
 'pool,playground,sports court' 'gym'
 'barbecue space,gym,playground,sports court'
 'gym,playground,sports court' 'gym,sports court'
 'barbecue space,gym,playground' 'barbecue space,gym'
 'pool,barbecue space,gym' 'pool,gym,playground'
 'pool,barbecue space,sports court'
 'barbecue space,steam room,gym,playground' 'gym,playground'
 'steam room,gym' 'barbecue space,gym,sports court'
 'pool,gym,playground,sports court' 'pool,barbecue space,gym,sports court'
 'steam room,playground' 'steam room' 'pool,steam room,playground'
 'pool,gym' 'barbecue space,steam room,gym,playground,sports court'
 'pool,steam room' 'barbecue space,steam room,playground'
 'steam room,playground,sports court'
 'barbecue space,steam room,sports court'
 'barbecue space,steam room,playground,sports court'
 'pool,barbecue space,steam room'
 'pool,steam room,gym,playground,sports court' 'pool,gym,sports court'
 'pool,barbecue space,steam room,gym,playground'
 'pool,steam room,gym,playground' 'steam room,gym,playground,sports court'
 'pool,barbecue space,steam room,playground,sports court'
 'pool,barbecue space,steam room,gym,playground,sports court'
 'steam room,sports court' 'pool,barbecue space,steam room,gym'
 'pool,barbecue space,steam room,playground' 'steam room,gym,playground']
```

In [346…

```
#plotting unique values for categorical columns
unique_values = []
for col in object_cols:
  unique_values.append(data[col].unique().size)
plt.figure(figsize=(10,6))
plt.title('No. Unique values of Categorical Features')
plt.xticks(rotation=90)
sns.barplot(x=object_cols,y=unique_values)
```

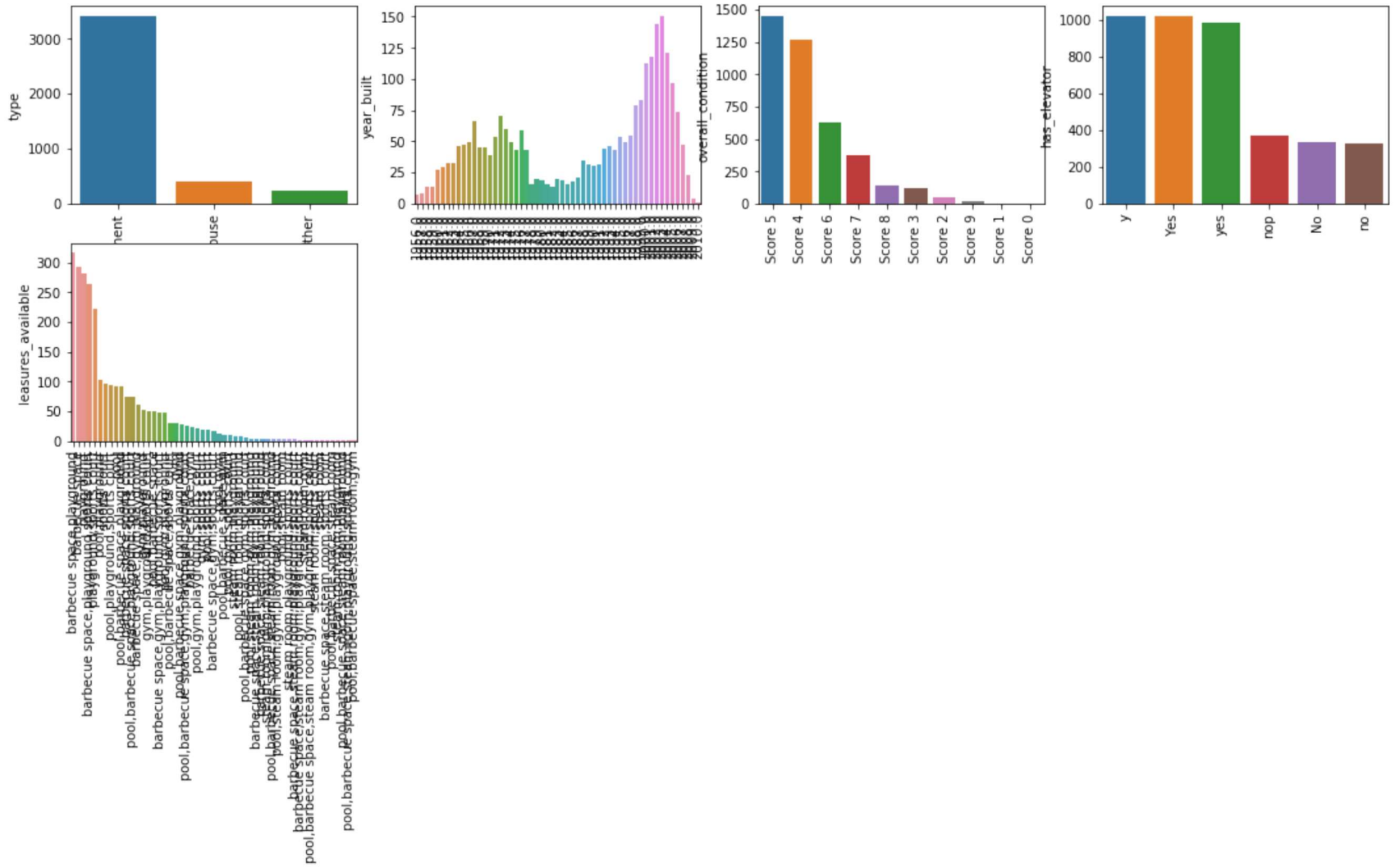
Out[346]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7987a7ced0>



In [347…

```
#count of unique values for each categorical columns
plt.figure(figsize=(18, 36))
plt.title('Categorical Features: Distribution')
plt.xticks(rotation=90)
index = 1

for col in object_cols:
    y = data[col].value_counts()
    plt.subplot(11, 4, index)
    plt.xticks(rotation=90)
    sns.barplot(x=list(y.index), y=y)
    index += 1
```


type
year_built
overall_condition
has_elevator
leasures_available
Score 5
Score 4
Score 6
Score 7
Score 8
Score 3
Score 2
Score 9
Score 1
Score 0
y
Yes
yes
nop
No
no

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Data Cleaning

In [397…

```
#standardize has_elevator values
data['has_elevator'] = data['has_elevator'].apply(lambda x: 'yes' if 'y' in x.lower() else 'no')
data
```

Out[397]:

	id	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_bui
0	794	380	6.0	4.0	0.0	3.0	3.0	house	-5.855204	-35.220087	1974
1	199	60	2.0	2.0	0.0	1.0	1.0	apartment	-5.808983	-35.227251	Na
2	1849	430	NaN	NaN	0.0	8.0	4.0	house	-5.843724	NaN	1980
3	33	55	2.0	2.0	280.0	0.0	1.0	apartment	-5.882136	-35.172217	1991
4	179	50	2.0	2.0	420.0	1.0	2.0	apartment	-5.866124	-35.182770	2002
...	...	...	...	...	...	...	...	...	...	...	
4051	1246	67	1.0	2.0	298.0	1.0	1.0	apartment	-5.863309	-35.232432	2002
4052	919	155	3.0	3.0	450.0	2.0	1.0	apartment	-5.803210	-35.200205	2001
4053	2595	180	5.0	3.0	1300.0	3.0	3.0	apartment	-5.786901	-35.190263	Na
4054	228	277	1.0	3.0	0.0	2.0	1.0	house	-5.857640	-35.201449	1980
4055	1063	57	2.0	2.0	0.0	1.0	1.0	apartment	-5.843197	-35.212447	1962

4056 rows × 15 columns

In [398…
```
#lowercase all values in leasures_available
data['leasures_available'] = data['leasures_available'].str.lower()
```

In [399…
```
#fill in none for na values in leasures available
data['leasures_available'].fillna('none', inplace=True)
data
```

Out[399]:

	id	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_bui
0	794	380	6.0	4.0	0.0	3.0	3.0	house	-5.855204	-35.220087	1974
1	199	60	2.0	2.0	0.0	1.0	1.0	apartment	-5.808983	-35.227251	Na
2	1849	430	NaN	NaN	0.0	8.0	4.0	house	-5.843724	NaN	1980
3	33	55	2.0	2.0	280.0	0.0	1.0	apartment	-5.882136	-35.172217	1991
4	179	50	2.0	2.0	420.0	1.0	2.0	apartment	-5.866124	-35.182770	2002
...	...	...	...	...	...	...	...	...	...	...	
4051	1246	67	1.0	2.0	298.0	1.0	1.0	apartment	-5.863309	-35.232432	2002
4052	919	155	3.0	3.0	450.0	2.0	1.0	apartment	-5.803210	-35.200205	2001
4053	2595	180	5.0	3.0	1300.0	3.0	3.0	apartment	-5.786901	-35.190263	Na
4054	228	277	1.0	3.0	0.0	2.0	1.0	house	-5.857640	-35.201449	1980
4055	1063	57	2.0	2.0	0.0	1.0	1.0	apartment	-5.843197	-35.212447	1962

4056 rows × 15 columns

In [400…

```
#dropping columns that are not of interest
data.drop(['id'],
            axis=1,
            inplace=True)
```

In [401…

```
#convert year_built to object
data['year_built'] = data['year_built'].astype(str)
data['year_built'] = data['year_built'].replace('nan', np.nan)
```

In [402…
```
numerical_cols = data.select_dtypes(exclude=['object']).columns
numerical_cols = numerical_cols.drop('year_built', errors='ignore')
numerical_cols
```

Out[402]:
```
Index(['area', 'bathrooms', 'bedrooms', 'condo_fee', 'parking_spots',
       'attached_rooms', 'lat', 'lon', 'price'],
      dtype='object')
```

In [403…
```
categorical_columns = data.select_dtypes(include=['object']).columns
categorical_columns
```

Out[403]:
```
Index(['type', 'year_built', 'overall_condition', 'has_elevator',
       'leasures_available'],
      dtype='object')
```

In [404…
```
#impute numerical_cols with median

for col in numerical_cols:
    mean = data[col].median()
    data[col].fillna(mean, inplace = True)
```

In [405…
```
#impute categorical cols with mode

for col in categorical_columns:
    mode = data[col].mode()[0]
    data[col].fillna(mode, inplace = True)
```

In [406…
```
#DROPPING NULL ROWS
#data.dropna(inplace=True)
```

In [ ]:
```

```

In [407…
```
#checking null counts in each column
null_counts = data.isnull().sum()

# Print the result
print("Null counts per column:")
print(null_counts)
```

```
Null counts per column:
area                  0
bathrooms             0
bedrooms              0
condo_fee             0
parking_spots         0
attached_rooms        0
type                  0
lat                   0
lon                   0
year_built            0
overall_condition     0
has_elevator          0
leasures_available    0
price                 0
dtype: int64
```

In [ ]:

In [408…
```
from sklearn.preprocessing import OneHotEncoder
```

In [409…
```
s = (data.dtypes == 'object')
object_cols = list(s[s].index)
print("Categorical variables:")
print(object_cols)
print('No. of. categorical features: ',
      len(object_cols))
```

```
Categorical variables:
['type', 'year_built', 'overall_condition', 'has_elevator', 'leasures_available']
No. of. categorical features:  5
```

In [410…
```
#one hot encoding
#OH_encoder = OneHotEncoder(sparse=False)
#OH_cols = pd.DataFrame(OH_encoder.fit_transform(data[object_cols]))
#OH_cols.index = data.index
#OH_cols.columns = OH_encoder.get_feature_names()
#df_final = data.drop(object_cols, axis=1)
#df_final = pd.concat([df_final, OH_cols], axis=1)
```

In [411…
```
df_final = data
```

In [412…
```
#ordinal encoding
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()

# Fit and transform the selected columns
df_final[object_cols] = ordinal_encoder.fit_transform(df_final[object_cols])
```

In [ ]:

In [413…
```
df_final
```

Out[413]:

	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_built	overall_c
0	380	6.0	4.0	0.0	3.0	3.0	1.0	-5.855204	-35.220087	18.0	
1	60	2.0	2.0	0.0	1.0	1.0	0.0	-5.808983	-35.227251	47.0	
2	430	2.0	3.0	0.0	8.0	4.0	1.0	-5.843724	-35.203220	24.0	
3	55	2.0	2.0	280.0	0.0	1.0	0.0	-5.882136	-35.172217	35.0	
4	50	2.0	2.0	420.0	1.0	2.0	0.0	-5.866124	-35.182770	46.0	
...	...	...	...	...	...	...	...	...	...	...	
4051	67	1.0	2.0	298.0	1.0	1.0	0.0	-5.863309	-35.232432	46.0	
4052	155	3.0	3.0	450.0	2.0	1.0	0.0	-5.803210	-35.200205	45.0	
4053	180	5.0	3.0	1300.0	3.0	3.0	0.0	-5.786901	-35.190263	47.0	
4054	277	1.0	3.0	0.0	2.0	1.0	1.0	-5.857640	-35.201449	24.0	
4055	57	2.0	2.0	0.0	1.0	1.0	0.0	-5.843197	-35.212447	6.0	

4056 rows × 14 columns

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Splitting the data for testing and evaluation

In [67]:
```
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
```

In [414…
```
X = df_final.drop(['price'], axis=1)
Y = df_final['price']

# Split the training set into
# training and validation set
X_train, X_valid, Y_train, Y_valid = train_test_split(
    X, Y, train_size=0.8, test_size=0.2, random_state=0)
```

In [ ]:

SVM Model

In [57]:
```
#trying with SVM (usually lowest MAPE value)
```

In [70]:
```
from sklearn import svm
from sklearn.svm import SVC
```

In [206…
```
import numpy as np

def mean_absolute_percentage_error(y_true, y_pred):
```

```
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true))
```

In [ ]:

In [365…
```
model_SVR = svm.SVR(kernel='linear')
model_SVR.fit(X_train,Y_train)
Y_pred = model_SVR.predict(X_valid)

print(mean_absolute_percentage_error(Y_valid, Y_pred))
```

```
0.3261841795090271
```

In [366…
```
feature_importance = model_SVR.coef_
feature_importance
```

Out[366]:
```
array([[2244.85885556, 1073.28440693,  464.68776722,   86.70954675,
         663.37553443,  986.37553443, -289.80264926,    5.92501702,
           3.13199922, -122.92683789,   11.53416678,  -24.15863235,
         764.05158039]])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Linear Regression

In [124…
```
from sklearn.linear_model import LinearRegression
```

In [367…
```
model_LR = LinearRegression()
model_LR.fit(X_train, Y_train)
Y_pred = model_LR.predict(X_valid)

print(mean_absolute_percentage_error(Y_valid, Y_pred))
```

```
0.29711752591022766
```

In [368…
```
feature_importance = model_LR.coef_
feature_importance
```

Out[368]:
```
array([ 1.79051491e+03,  4.37020854e+03, -2.04406871e+04,  5.20503733e+01,
        4.80221715e+04,  5.86382106e+04, -5.87126188e+04,  5.27588191e+05,
        1.72137150e+06, -1.59019900e+02,  1.08541751e+03, -3.54683536e+03,
        6.22517521e+02])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

XGBOOST

In [131…
```
pip install xgboost
```

```
Collecting xgboost
  Downloading xgboost-1.6.2-py3-none-manylinux2014_x86_64.whl (255.9 MB)
     |████████████████████████████████| 255.9 MB 28 kB/s s eta 0:00:01     |████████                        | 76.9 MB
46.9 MB/s eta 0:00:04     |████████████████████████        | 211.7 MB 47.6 MB/s eta 0:00:01
Requirement already satisfied: numpy in /opt/conda/lib/python3.7/site-packages (from xgboost) (1.19.2)
Requirement already satisfied: scipy in /opt/conda/lib/python3.7/site-packages (from xgboost) (1.3.3)
Installing collected packages: xgboost
Successfully installed xgboost-1.6.2
Note: you may need to restart the kernel to use updated packages.
```

In [132…
```
import xgboost as xgb
```

In [369…
```
model_xgb = xgb.XGBRegressor()
model_xgb.fit(X_train, Y_train)
```

```
Y_pred = model_xgb.predict(X_valid)
```

In [370…
```
print(mean_absolute_percentage_error(Y_valid, Y_pred))
```

```
0.2157209311561152
```

In [371…
```
feature_importance = model_xgb.feature_importances_
feature_importance
```

Out[371]:
```
array([0.20905288, 0.03635485, 0.0136193 , 0.01361444, 0.26713493,
       0.16515063, 0.19443482, 0.02410395, 0.03764498, 0.01303148,
       0.0070142 , 0.0093499 , 0.00949358], dtype=float32)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

XGBoost with GridSearch

In [276…
```
from sklearn.model_selection import GridSearchCV
```

In [372…
```
# Define the parameter grid
param_grid = {
    'learning_rate': [0.01, 0.1, 0.3],  # Learning rate
    'max_depth': [3, 5, 7],  # Maximum depth of the trees
    'n_estimators': [100, 200, 300]  # Number of trees
}

# Initialize XGBoost regressor
model_xgb = xgb.XGBRegressor()

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=model_xgb, param_grid=param_grid, cv=5, scoring='neg_mean_absolute_error')

# Perform grid search
```

```
grid_search.fit(X_train, Y_train)

# Get the best parameters
best_params = grid_search.best_params_
print("Best parameters:", best_params)
```

```
Best parameters: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}
```

In [ ]:

In [415…

```
# Define the best parameters
best_params = {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}

# Create an instance of XGBRegressor with the best parameters
best_model_xgb = xgb.XGBRegressor(**best_params)


# Train the model with the best parameters
best_model_xgb.fit(X_train, Y_train)

# Make predictions
Y_pred = best_model_xgb.predict(X_valid)
```

In [416…

```
print(mean_absolute_percentage_error(Y_valid, Y_pred))
```

```
0.2080985509744779
```

In [417…

```
feature_importance = best_model_xgb.feature_importances_
feature_importance
```

Out[417]:

```
array([0.19302306, 0.03427662, 0.01461706, 0.01452867, 0.28658062,
       0.16392262, 0.19606945, 0.02211263, 0.03851841, 0.00920714,
       0.00781178, 0.00838655, 0.01094541], dtype=float32)
```

In [ ]:

In [ ]:

In [ ]:

Feature importance for best Model (XGBoost + GridSearch)

1. parking_spots 0.28658062
2. type 0.19606945
3. area 0.19302306
4. attached_rooms 0.16392262
5. lon 0.03851841
6. bathrooms 0.01461706
7. lat 0.02211263
8. bedrooms 0.01461706
9. condo_fee 0.01452867
10. leasures_available 0.01094541
11. year_built 0.00920714
12. has_elevator 0.00838655
13. overall_condition 0.00781178

In [ ]:

In [ ]:

Training final best model (XGBoost + GridSearch) with all training dataset available

In [419…
```
X_train = df_final.drop(['price'], axis=1)
Y_train = df_final['price']
```

In [421…
```
best_params = {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}

best_model_xgb = xgb.XGBRegressor(**best_params)
```

```
# Train the model with the best parameters
best_model_xgb.fit(X_train, Y_train)
```

Out[421]:
```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, gamma=0, gpu_id=-1, grow_policy='depthwise',
             importance_type=None, interaction_constraints='',
             learning_rate=0.1, max_bin=256, max_cat_to_onehot=4,
             max_delta_step=0, max_depth=7, max_leaves=0, min_child_weight=1,
             missing=nan, monotone_constraints='()', n_estimators=200, n_jobs=0,
             num_parallel_tree=1, objective='reg:squarederror',
             predictor='auto', random_state=0, reg_alpha=0, ...)
```

In [ ]:

In [ ]:

Using the Model to predict test dataset

In [422…
```
# Test Dataset
data = pd.read_csv("test.csv")
```

In [423…
```
data
```

Out[423]:

	id	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_built
0	2590	54	2.0	2.0	0	1	1	apartment	-5.882136	-35.172217	2004.0
1	242	92	3.0	3.0	575	2	1	apartment	-5.867429	-35.196668	1966.0
2	2946	98	2.0	4.0	0	2	2	apartment	-5.857640	-35.201449	NaN
3	758	182	5.0	3.0	1200	3	3	apartment	-5.812787	-35.204990	1993.0
4	1160	69	2.0	3.0	250	1	0	apartment	-5.843724	-35.221448	2001.0
...	...	...	...	...	...	...	...	...	...	...	...
711	4426	99	4.0	3.0	490	2	2	apartment	-5.824351	-35.218133	NaN
712	466	56	2.0	2.0	230	1	1	apartment	-5.882136	-35.172217	2004.0
713	3092	55	1.0	2.0	300	1	1	apartment	-5.865419	-35.209774	2005.0
714	3772	109	4.0	4.0	750	2	3	apartment	-5.816526	-35.214127	NaN
715	860	69	2.0	2.0	340	1	0	apartment	-5.867429	-35.196668	NaN

716 rows × 14 columns

In [424…

```
#Same data cleaning applied to Test Data

#standardize has_elevator values
data['has_elevator'] = data['has_elevator'].apply(lambda x: 'yes' if 'y' in x.lower() else 'no')

#lowercase all values in leasures_available
data['leasures_available'] = data['leasures_available'].str.lower()
```

```
#fill in none for na values in leasures available
data['leasures_available'].fillna('none', inplace=True)

#dropping columns that are not of interest
data.drop(['id'],
            axis=1,
            inplace=True)

#convert year_built to object
data['year_built'] = data['year_built'].astype(str)
data['year_built'] = data['year_built'].replace('nan', np.nan)

numerical_cols = data.select_dtypes(exclude=['object']).columns
numerical_cols = numerical_cols.drop('year_built', errors='ignore')

categorical_columns = data.select_dtypes(include=['object']).columns




#impute numerical_cols with median

for col in numerical_cols:
    mean = data[col].median()
    data[col].fillna(mean, inplace = True)

#impute categorical cols with mode

for col in categorical_columns:
    mode = data[col].mode()[0]
    data[col].fillna(mode, inplace = True)

s = (data.dtypes == 'object')
object_cols = list(s[s].index)
print("Categorical variables:")
print(object_cols)
print('No. of. categorical features: ',
      len(object_cols))

df_final = data
```

```
#ordinal encoding
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()

# Fit and transform the selected columns
df_final[object_cols] = ordinal_encoder.fit_transform(df_final[object_cols])
```

```
Categorical variables:
['type', 'year_built', 'overall_condition', 'has_elevator', 'leasures_available']
No. of. categorical features:  5
```

In [425…

```
df_final
```

Out[425]:

	area	bathrooms	bedrooms	condo_fee	parking_spots	attached_rooms	type	lat	lon	year_built	overall_cc
0	54	2.0	2.0	0	1	1	0.0	-5.882136	-35.172217	48.0	
1	92	3.0	3.0	575	2	1	0.0	-5.867429	-35.196668	10.0	
2	98	2.0	4.0	0	2	2	0.0	-5.857640	-35.201449	43.0	
3	182	5.0	3.0	1200	3	3	0.0	-5.812787	-35.204990	37.0	
4	69	2.0	3.0	250	1	0	0.0	-5.843724	-35.221448	45.0	
…	…	…	…	…	…	…	…	…	…	…	
711	99	4.0	3.0	490	2	2	0.0	-5.824351	-35.218133	43.0	
712	56	2.0	2.0	230	1	1	0.0	-5.882136	-35.172217	48.0	
713	55	1.0	2.0	300	1	1	0.0	-5.865419	-35.209774	49.0	
714	109	4.0	4.0	750	2	3	0.0	-5.816526	-35.214127	43.0	
715	69	2.0	2.0	340	1	0	0.0	-5.867429	-35.196668	43.0	

716 rows × 13 columns

In [426…

```
X_test = df_final
Y_pred = best_model_xgb.predict(X_test)
```

In [428…
```
Y_pred_df = pd.DataFrame(Y_pred, columns=['Y_pred'])
```

In [430…
```
data = pd.read_csv("test.csv", usecols=['id'])
```

In [431…
```
merged_data = pd.concat([data, Y_pred_df], axis=1)
```

In [433…
```
merged_data = merged_data.rename(columns={'Y_pred': 'price'})
```

In [434…
```
merged_data
```

Out[434]:

	id	price
0	2590	257900.281250
1	242	432631.281250
2	2946	472557.750000
3	758	734058.125000
4	1160	226819.875000
...	...	...
711	4426	471246.343750
712	466	236959.484375
713	3092	215764.750000
714	3772	508365.031250
715	860	193887.265625

716 rows × 2 columns

In [435…
```
#Submission
merged_data.to_csv('submissions.csv',index=False)
```

In [ ]:

In [ ]:

Visualization, Modeling, Machine Learning

Can you build a model that can predict property prices and identify how different features influence their decision? Please explain your findings effectively to technical and non-technical audiences using comments and visualizations, if appropriate.

- **Build an optimized model that effectively solves the business problem.**
- **The model would be evaluated on the basis of mean absolute percent error.**
- **Read the Test.csv file and prepare features for testing.**

In [ ]:

High Level Summary

The optimized model utilizes the XGBoost regression algorithm, with the best parameters obtained from grid search, a hyperparameter tuning method. XGBoost is essentially an advanced implementation of gradient boosting algorithms designed to optimize performance and computational speed. Similar to gradient boosting, it utilizes an ensemble of decision trees, where each tree corrects the errors of its predecessor, gradually improving the model's predictive performance. The boosting process combines the predictions of multiple weak learners to create a strong learner that generalizes well to unseen data.

The model utilizes all provided features, and their importance is ranked as follows:

1. parking_spots
2. type
3. area
4. attached_rooms
5. lon
6. bathrooms
7. lat
8. bedrooms
9. condo_fee

10. leasures_available
11. year_built
12. has_elevator
13. overall_condition

For null values, numerical variables are imputed with their median, while categorical variables are imputed with their mode, with the exception of "leasures_available," in which null values are replaced with 'none'. Categorical variables undergo ordinal encoding as part of the X_train dataset.

The MAPE (Mean Absolute Percentage Error) of the model is estimated to be around 0.20.

Machine learning is an art of experimentation, so in an ideal world, experimenting with different combinations of features would be best to obtain a more optimized model.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

The management wants to know what are the most important features for your model. Can you tell them?

> ## Task:

- **Visualize the top 20 features and their feature importance.**

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: