

Pattern Recognition Project

Apartment Rent Prediction

Presented by:

(Team ID: CS_9)

Name	ID	Section
Mina Edwar Dawood Elias	2021170565	7
Nada AbdElmoneem Ahmed	2021170583	7
Nardine Guirguis Edward Amin	2021170576	7
Youssef Emad El-din Ibrahim	2021170640	8
Youssef Hany Ezzat Aly	2021170653	8
Dalia AbdElazim Mohamed	2021170179	3

Table of Contents

1.Introduction	4
1.1 Overview	4
2.Data Preprocessing	5
2.1 Data Loading	5
2.2 Data Exploration	5
2.3 Data Cleaning	6
2.3.1 Handling Missing Values	6
2.3.2 Handling Incorrect Formats	8
2.3.3 Handling Outliers	8
2.4 Processing Numerical Features	9
2.4.1 Different Techniques	9
2.5 Handling Categorical Features	10
2.5.1 Different Techniques	10
2.5.2 Encoding	11
2.6 Dropping Columns	13
2.7 Feature Engineering	14
2.8 Feature Selection	15
2.9 Train Test Split	17
3.Models	17
3.1 Linear Regression	17
3.1.1 Introduction	17
3.1.2 Error/Accuracy	17
3.1.3 Visualization	18
3.1.4 Improvements	18
3.2 Ridge Regression	19
3.2.1 Introduction	19
3.2.2 Error/Accuracy	19
3.2.3 Visualization	19
3.3 Polynomial Regression	20
3.3.1 Introduction	20

3.3.2 Error/Accuracy	20
3.3.3 Visualization	20
3.4 Decision Tree Regressor	21
3.4.1 Introduction	21
3.4.2 Error/Accuracy	21
3.4.3 Visualization	21
3.5 Gradient Boosting Regressor	22
3.5.1 Introduction	22
3.5.2 Error/Accuracy	22
3.5.3 Visualization	22
3.6 Random Forest	23
3.6.1 Introduction	23
3.6.2 Error/Accuracy	23
3.6.3 Visualization	23
3.7 XGBoost	23
3.7.1 Introduction	23
3.7.2 Error/Accuracy	24
3.7.3 Visualization	24
4.Improvements	24
5.Conclusion	25
6.Python Documentation	26

1.Introduction

1.1 Overview

In an era we are living in right now that is characterized by the unlimited access to data and computational power, The help of machine learning has become very important in various fields. This model presents an overview of Apartment Rent Prediction using different machine learning algorithms. We applied those algorithms on a real-world task using Apartment Rent Dataset.

In this task we aim to predict the rent price of apartment depending on some features inside the dataset used. We have faced a lot of challenges dealing with this data as it has some missing values, unclear formats ...etc. We applied different machine learning algorithms dealing with each one to provide the best accuracy it can provide. Finally we chose the best 2 models with the best accuracy provided to start using it to get the best predictions.

2.Data Preprocessing

2.1 Data Loading

We started our project by loading the dataset which is ApartmentRentPrediction.csv a comma separated values file into our environment in a pandas data frame (df) using the method read_csv().

2.2 Data Exploration

We started printing our data frame now to see number of columns and rows of the data and some samples of them. We found that our data contains 9000 rows and 22 columns.

Diving into the dataset we started to get the number of unique values in each column which can be useful later. After that we started to check the data type of each column and we found that some of the columns are categorical columns which we cannot deal with them instantly. We then started checking for missing values, duplicates and performed some statistical analysis on the numerical columns. We found that we had some missing values, and no duplicates in some columns as shown in the following figures.

```
####Nulls
df.isna().sum()
```

id	0
category	0
title	0
body	0
amenities	3185
bathrooms	30
bedrooms	7
currency	0
fee	0
has_photo	0
pets_allowed	3751
price	0
price_display	0
price_type	0
square_feet	0
address	2971
cityname	66
state	66
latitude	7
longitude	7
source	0
time	0

```
####Duplicates
duplicates = df[df.duplicated()]

if duplicates.empty:
    print("No duplicate rows found.")
else:
    print("Duplicate rows found:")
    print(duplicates)
```

No duplicate rows found.

```
####Checking the type of each column
df.dtypes
```

id	int64
category	object
title	object
body	object
amenities	object
bathrooms	float64
bedrooms	float64
currency	object
fee	object
has_photo	object
pets_allowed	object
price	int64
price_display	object
price_type	object
square_feet	int64
address	object
cityname	object
state	object
latitude	float64
longitude	float64
source	object
time	int64

```
###performing statistical analysis
df.describe()
```

	id	bathrooms	bedrooms	price	square_feet	latitude	longitude	time
count	9.000000e+03	8970.000000	8993.000000	9000.000000	9000.000000	8993.000000	8993.000000	9.000000e+03
mean	5.623668e+09	1.380769	1.744023	1487.286222	947.138667	37.67689	-94.778612	1.574906e+09
std	7.007402e+07	0.616171	0.942446	1088.561190	668.806214	5.51527	15.769232	3.755142e+06
min	5.508654e+09	1.000000	0.000000	200.000000	106.000000	21.31550	-158.022100	1.568744e+09
25%	5.509250e+09	1.000000	1.000000	950.000000	650.000000	33.66200	-101.858700	1.568781e+09
50%	5.668610e+09	1.000000	2.000000	1275.000000	802.000000	38.75550	-93.707700	1.577358e+09
75%	5.668626e+09	2.000000	2.000000	1695.000000	1100.000000	41.34980	-82.446800	1.577359e+09
max	5.668663e+09	8.500000	9.000000	52500.000000	40000.000000	61.59400	-70.191600	1.577362e+09

2.3 Data Cleaning

2.3.1 Handling Missing Values

As we saw before we have columns with lots of missing values, It's time to deal with them. Note we didn't drop any row as the dataset with 9k rows will be affected if we removed 3k rows, so we replaced them with different techniques in each column.

1.Column (amenities):

It contains categorical values so as a beginning we just filled them with 0 or we can replace them with None. We will further do some more preprocessing on this column which we will show later in this report.

```
df['amenities'] = df['amenities'].fillna(0)
```

2.Column (bathrooms) & Column (bedrooms):

They contain numerical values and there aren't a lot of missing values so we just replaced the missing values of each column with the mode(most frequent value) of this column.

```
#bathrooms and bedrooms since there are low number of missing values mean or mode replacement or we can drop them
df['bedrooms']=df['bedrooms'].fillna(df['bedrooms'].mode().iloc[0])
df['bathrooms']=df['bathrooms'].fillna(df['bathrooms'].mode().iloc[0])
```

3.Column (pets_allowed):

A categorical column and there are a lot of missing values. Here we just filled the missing values with 'No' and the other different values we mapped them to 'Yes' and we will further preprocess on this column.

```
df['pets_allowed'] = df['pets_allowed'].fillna('No').apply(lambda x: 'Yes' if x != 'No' else x)
df['pets_allowed']
#Converting any pets to Yes and Missing Values to No
```

4.Column (address):

A categorical column and there are a lot of missing values. Here we just filled the missing values with 'Unknown' and we will do some more preprocessing on it later.

```
###address
df['address'] = df['address'].fillna("Unknown")
```

5.Column (cityname) & Column (state):

categorical columns and there aren't a lot of missing values. Here we just filled the missing values of each column with the mode of this column and we will do some more preprocessing on it later.

```
df['cityname'] = df['cityname'].fillna(df['cityname'].mode().iloc[0])
df['state'] = df['state'].fillna(df['state'].mode().iloc[0])
```

6.Column (longitude) & Column (latitude):

numerical columns and there aren't a lot of missing values. Here we just filled the missing values of each column with the mean(average) of this column.

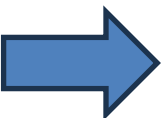
```
####Long and Lat columns we can fill NaN with the mean.
df['longitude'] = df['longitude'].fillna(df['longitude'].mean())
df['latitude'] = df['latitude'].fillna(df['longitude'].mean())
```

Now we successfully replaced all missing values and have no more missing values as shown in the following figure:

```
df.isna().sum()
id          0
category    0
title       0
body        0
amenities   0
bathrooms   0
bedrooms    0
currency    0
fee         0
has_photo   0
pets_allowed 0
price       0
price_display 0
price_type   0
square_feet  0
address     0
cityname     0
state       0
latitude    0
longitude   0
source      0
time        0
```

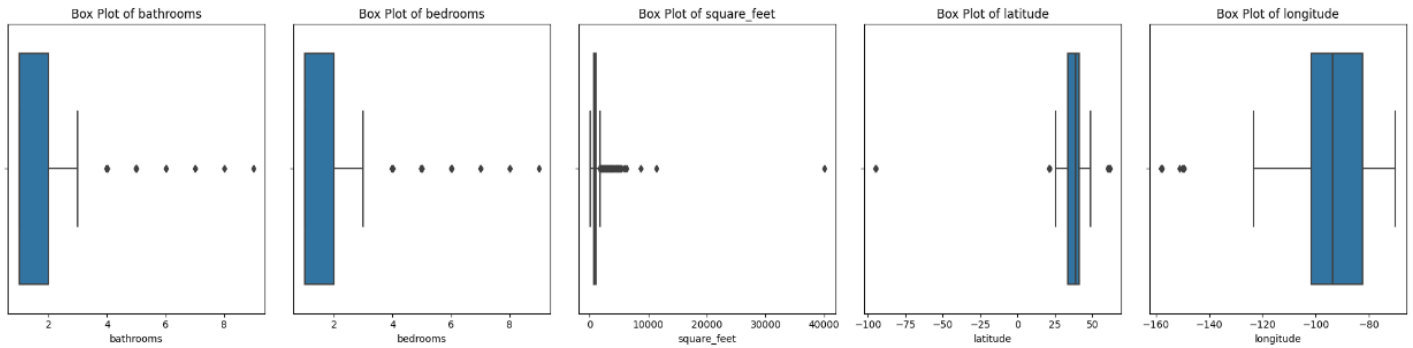
2.3.2 Handling Incorrect Formats

In exploring the price_display column which is our target column we found that the format is not clear as it must be numeric values containing no strings or symbols we found that it has values like \$1,194 which must be 1194 clear. We used regular expression to just replace the '\$' or ',' with an empty string also we made sure that we converted it's type to int after we did this as showed in the next figure.

<pre>df['price_display'] #Format is not clear</pre>		<pre>df['price_display'] = df['price_display'].str.replace(r'^0-9.', '', regex=True).astype(np.int64) df['price_display']</pre>
<pre>0 \$1,194 1 \$1,370 2 \$1,009 3 \$695 4 \$3,695 ... 8995 \$1,158 8996 \$2,035 8997 \$424 8998 \$1,417 8999 \$2,398 Name: price_display, Length: 9000, dtype: object</pre>		<pre>0 1194 1 1370 2 1009 3 695 4 3695 ... 8995 1158 8996 2035 8997 424 8998 1417 8999 2398 Name: price_display, Length: 9000, dtype: int64</pre>

2.3.3 Handling Outliers

- We first visualized the outliers of our numerical columns for outlier detection as shown in the figure:



- We made sure that our numerical columns outliers are replaced with the nearest non-outlier value (Winsorization).

2.4 Processing Numerical Features

2.4.1 Different Techniques

Column (id): In this column we found that each row has a unique id but it's number is very high so we just started giving each row a unique id also but numbered from 0 to 8999 instead of the high numbers like 5668617853 so it looks better for eyes as it looks like an index for the row.

```
##id column can be mapped to index 0 till 8999 (useless it is integer anyways and unique and we won't use it)
##we just do it to make it look easier
unique_ids = df['id'].unique()
id_mapping = {original_id: new_id for new_id, original_id in enumerate(unique_ids)}
df['id'] = df['id'].map(id_mapping)
df['id']
```

```
0      0
1      1
2      2
3      3
4      4
...
8995   8995
8996   8996
8997   8997
8998   8998
8999   8999
Name: id, Length: 9000, dtype: int64
```

Column (time): In this column we have found that time of the reported row has a special format but we don't care about the time so we just scaled it to make it looks easier. We divided the time rows with minimum value of the time column.

```
#### time column
max_time = df['time'].max()
min_time = df['time'].min()
print(f"Maximum time in : {max_time}")
print(f"Minimum time in : {min_time}")
```

```
Maximum time in : 1577362186
Minimum time in : 1568743976
```

```
df['time']=df['time']/df['time'].min()
```



```
df['time']
```

```
0    1.000008
1    1.005273
2    1.005492
3    1.005492
4    1.005492
```

```
...
```

```
8995    1.000024
8996    1.005491
8997    1.005492
8998    1.000018
8999    1.005492
```

```
Name: time, Length: 9000, dtype: float64
```

Column(bedrooms): In this column we noticed that there are 171 values = 0 which is nonsense so we replaced all 0's with 1's.

```
df['bedrooms'].value_counts()[0]
```

```
171
```

so we did this:

```
df['bedrooms']=df['bedrooms'].replace(0,1)
```

2.5 Handling Categorical Features

2.5.1 Different Techniques

- Exploring the csv file we found that In the **“body” column**, we started thinking for tokenizing it but we found that most of the sentences already extracted in other columns like the amenities .etc. most of the apartments have either a fixed price listed or a price range, it always starts with “\$(followed by numbers)We”, or in case of the price range “\$(followed by numbers) - \$(followed by numbers)We”. The next step is relatively simple, we used library re(regular expression) to identify the pattern we found then we just extract the price/minimum price in case of price range and put it in the “extracted_price” column, and then we use it as an indicator to the prices as shown in the function extract_price in the following fig **Note:** if no pattern noticed we just return 0

```
#####Extracting Price Info from body column#####
def extract_price(text):
    prices = re.findall(r'\$(\d+(?:\.\d+)?)', text)
    if prices:
        return prices[0]
    else:
        return 0
```

```
df['extracted_price'] = df['body'].apply(extract_price)
```



```
df['extracted_price']=df['extracted_price'].astype(np.float64)
df['extracted_price']
```

```
0    0.0
1   1370.0
2   1009.0
3    695.0
4   3695.0
```

```
...
```

```
8995    0.0
8996   2035.0
8997    424.0
8998    0.0
8999   2398.0
```

```
Name: extracted_price, Length: 9000, dtype: float64
```

- Moving to **the state and cityname columns** we processed using the same idea in both, We take the mean of prices of each city, then we represent the city with its mean of prices, not its original name, in a new column called “city_price_mean“, we then update the original column of “cityname” with the values of the “city_price_mean” column then we drop the “city_price_mean” column and then rename “cityname” column to

“CityavgRentprice”, doing the same to state column We take the mean of prices of each state, then we represent the state with its mean of prices, not its original name, in a new column called “state_price_mean”, we then update the original column of “state” with the values of the “state_price_mean” column then we drop the “state_price_mean” column and then rename “state” column to “StateavgRentprice” as shown in the fig:

```
#grouping the state according to its names and replacing with the avg price of this state name
state_encoding = df.groupby('state')['price_display'].mean().reset_index()
state_encoding.columns = ['state', 'state_price_mean']
df = pd.merge(df, state_encoding, on='state', how='left')
df['state']=df['state_price_mean']
df.drop(['state_price_mean'], axis=1, inplace=True)

#grouping the city according to its names and replacing with the avg price of this city name
city_encoding = df.groupby('cityname')['price_display'].mean().reset_index()
city_encoding.columns = ['cityname', 'city_price_mean']
df = pd.merge(df, city_encoding, on='cityname', how='left')
df['cityname']=df['city_price_mean']
df.drop(['city_price_mean'], axis=1, inplace=True)

#####Renaming them#####
df = df.rename(columns={'cityname': 'CityavgRentprice', 'state': 'StateavgRentprice'})
```

2.5.2 Encoding

As we have a regression problem which depends on numerical continuous variables we can’t continue to our models with non-numerical features so Encoding just transfer them from the categorical to numerical in different ways depending on its type.

2.5.2.1 Label Encoder

Before we start encoding we found that some columns has different unique values but some of this unique values only happens once so we just replace with the most frequent one that appears the most in the data as shown in the next figures.

df['source'].value_counts()	df['price_type'].value_counts() ##### We convert them to Monthly	df['category'].value_counts() ##### We convert them to housing/rent/apartment
source	price_type	category
RentLingo 6232	Monthly 8998	housing/rent/apartment 8997
RentDigs.com 2474	Weekly 1	housing/rent/short_term 2
ListedBuy 169	Monthly Weekly 1	housing/rent/home 1
RealRentals 58	Name: count, dtype: int64	Name: count, dtype: int64
GoSection8 27		
Listanza 21		
RENTOCULAR 15		
RENTCafé 1		
tenantcloud 1		
Real Estate Agent 1		
rentbits 1		
Name: count, dtype: int64		

Now jumping to the **label encoder** converts categorical data into a numerical representation by giving each category a special numerical label. We encoded the categorical columns into numerical ones by using object from the built-in LabelEncoder(). **We encoded columns (source, price_type, category, address, body, title, pets_allowed, has_photo, fee, currency)** as shown in the figure below:

```

label_encoder = LabelEncoder()
df['source'] = label_encoder.fit_transform(df['source'])
df['price_type'] = label_encoder.fit_transform(df['price_type'])
df['category'] = label_encoder.fit_transform(df['category'])
df['address'] = label_encoder.fit_transform(df['address'])
df['body'] = label_encoder.fit_transform(df['body'])
df['title'] = label_encoder.fit_transform(df['title'])
df['pets_allowed'] = label_encoder.fit_transform(df['pets_allowed'])
df['has_photo'] = label_encoder.fit_transform(df['has_photo'])
df['fee'] = label_encoder.fit_transform(df['fee'])
df['currency'] = label_encoder.fit_transform(df['currency'])

```

2.5.2.2 Different Technique

- Exploring the csv file we can notice that the amenities are showed in the amenities column with splitting between each one of them with a comma (,) so we did a function that split the contents of each cell in the amenities column after each comma and then count the number and update the original column cells.

```

####amenities column we fill NaN with 0 and we count each amenity.
## Ex: Gym , Parking , Pool = 3
def encode_amenities(value):
    items = value.split(',') if isinstance(value, str) else []
    count=len(items)
    return count

if df['amenities'].dtype == object:
    df['amenities'] = df['amenities'].apply(encode_amenities)

df['amenities']

0      6
1      9
2      7
3      0
4      0
..
8995   4
8996   0
8997   2
8998  11
8999   0
Name: amenities, Length: 9000, dtype: int64

```

We then replace the 0 values (Previously the NaN) with the mean or mode, which is 3 in our case because it is nonsense to have an apartment without any amenities.

```

####Replacing nan with 0 then with the mean or mode which is equal 3.
df.loc[df['amenities'] == 0, 'amenities'] = 3
df['amenities']

0      6
1      9
2      7
3      3
4      3
..
8995   4
8996   3
8997   2
8998  11
8999   3
Name: amenities, Length: 9000, dtype: int64

```

2.6 Dropping Columns

We started seeing what columns that doesn't have a real effect on our target variable or not meaningful to use it to predict the apartment rent price, so we dropped like 11 columns below is a summary of each column dropped with the reason of dropping it:

Column name	Reason of dropping
id	It is not more than an index (unique identifier) just for our row.
category	It has only one unique value which is 0 after encoding.
title	Non meaningful to take the title of the report to predict the price. Especially after encoding it to numbers which represent the number of characters in each row (doesn't make sense the more in title the more the price).
body	After exploring the body we found that all information is extracted in other columns so we don't need it anymore we encode it and drop it.
currency	It has only one unique value which is 0 after encoding.
fee	It has only one unique value which is 0 after encoding.
source	Non meaningful: We can't just use the source in which the report was reported to predict the price of an apartment.
time	After scaling the time we decided to drop it as it has no effect we don't predict the price according to the time it was reported.
price	After preprocessing the price_display column our target column so this price column is just a duplicate of our target column.
price_type	It has only one unique value which is 0 after encoding.
address	We decided to drop it as we already have the city, state, longitude and latitude so we can just figure out the area from them not with the address column which has maybe the no of apartment or no of the block.

Snapshots of the code where we dropped the columns and our dataframe now:

```
###Columns to drop : category, fee , currency , price_type (as they have only one value)
##Then drop them
###We can drop title, body as they dont affect the data
###column price might be dropped as we already preprocessed price_display so it is a duplicated column
columns_to_drop = ['id', 'category', 'title', 'body', 'currency', 'fee', 'source', 'time', 'price', 'price_type', 'address']
df = df.drop(columns=columns_to_drop)
```

```
df.head(5)
```

amenities	bathrooms	bedrooms	has_photo	pets_allowed	price_display	square_feet	CityavgRentprice	StateavgRentprice	latitude	longitude	extracted_price
6	3	2	1	1	1194	800	1152.095238	1190.845000	35.7585	-78.7783	0.0
9	1	1	1	1	1370	795	1602.509259	1298.712121	43.0724	-89.4003	1370.0
7	1	1	1	1	1009	560	1287.750000	1532.148148	29.6533	-82.3656	1009.0
3	1	1	1	1	695	600	952.650794	939.966292	41.2562	-96.0404	695.0
3	3	3	1	0	3695	1600	3084.543046	2811.288528	34.0372	-118.2972	3695.0

2.7 Feature Engineering

In this section we will introduce you to new 5 features we have made it after exploring some of the data:

1. AvgAreaPrice: This feature we decided to use longitude and latitude values in a feature so we divided the longitude and latitude each one to 10 equal ranges(bins). After that we started grouping this area by the avg price of this area.

```
#####New Feature ALERTTTTT##### AvgAreaPrice
df['longitude_range'] = pd.cut(df['longitude'], bins=10)
df['latitude_range'] = pd.cut(df['latitude'], bins=10)

range_means = df.groupby(['longitude_range', 'latitude_range'])['price_display'].mean()

df['AvgAreaPrice'] = df.groupby(['longitude_range', 'latitude_range'])['price_display'].transform('mean')

df.drop(['longitude_range', 'latitude_range'], axis=1, inplace=True)
```

2.NoOfRooms: This feature we simply calculated the total number of rooms in an apartment by adding the no of bedrooms to bathrooms in each row.

```
#####NEW FEATURE ALERT!!!!!!!!!!!!!!!!#####
df['NoOfRooms'] = df['bathrooms'] + df['bedrooms']
```

3.CityStateAvg: This feature we simply got the two columns we already had from the cityname and state which are CityavgRentprice and StateavgRentprice we added them together then divided them by 2.

```
#####NEW FEATURE ALERT!!!!!!!!!!!!!!!!#####
df['CityStateAvg'] = (df['CityavgRentprice'] + df['StateavgRentprice']) / 2
```

4.avgBathroomsPrice: This feature we grouped the unique values of no of bathrooms and decided to get for each different group (number of bathrooms) the avg price of this group for example the apartments which has 1 bathroom an avg price of 1266 and so on to all groups, price based on no of bathrooms.

```
#####NEW FEATURE ALERT!!!!!!!!!!!!!!!!#####
bath_encoding = df.groupby('bathrooms')['price_display'].mean().reset_index()
bath_encoding.columns = ['bathrooms', 'avgBathroomsPrice']
df = pd.merge(df, bath_encoding, on='bathrooms', how='left')
```

5.Trial: Finally, our last feature and we have achieved it after so many trials so that's why we called it Trial. Exploring the data with the new features we introduced and trying to get a pattern which we can get from it closer to the target we have discovered a pattern which if we added the CityavgRentprice + square_feet/2 + avgBathroomsPrice/3 + bedrooms*20 it gets us closer to the target. We basically gave the bedrooms cost (20), divided the avgBathroomsPrice by 3 which is the mode of the bathrooms, also divided the square_feet by 2 then added all those to the CityavgRentprice forming our new feature Trial.

```
#####NEW FEATURE ALERT!!!!!!!!!!!!#####
df['Trial'] = df['CityavgRentprice'] + df['square_feet']/2 + df['avgBathroomsPrice']/3 +df['bedrooms']*20
```

2.8 Feature Selection

- After Introducing new feature and handling their outliers, We are ready to select the features to be ready to enter the machine learning models to predict the apartment rent prices. We started by displaying the correlations values between the features and the target variable as shown in the next figure:

```
correlation_values = df.corr()['price_display'].sort_values(ascending=False)
print(correlation_values)
```

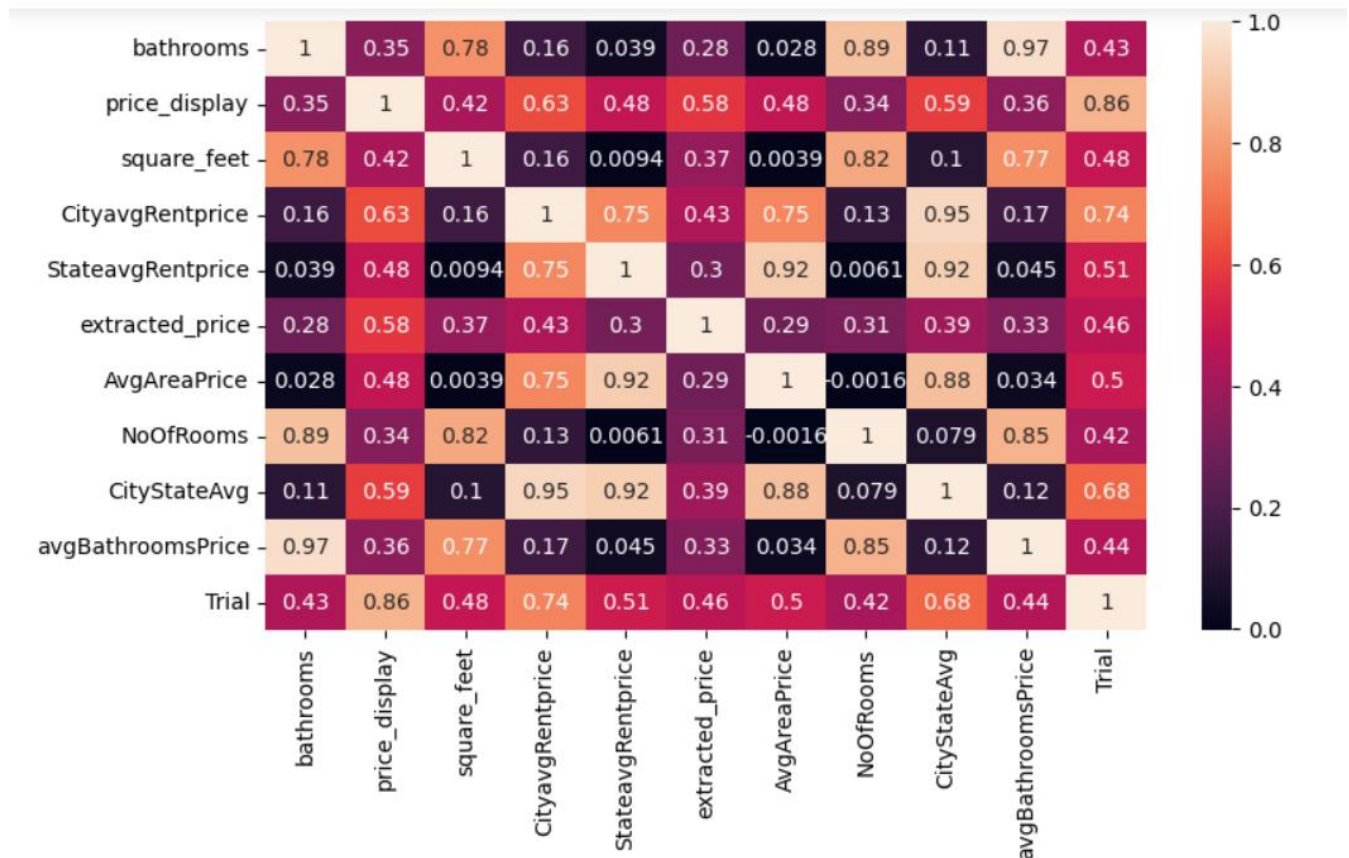
```
price_display      1.000000
Trial              0.859114
CityavgRentprice   0.625608
CityStateAvg       0.594949
extracted_price    0.578674
AvgAreaPrice       0.478770
StateavgRentprice  0.478157
square_feet        0.416907
avgBathroomsPrice  0.357807
bathrooms          0.346481
NoOfRooms          0.335961
bedrooms           0.285571
latitude           0.023148
has_photo          0.000501
pets_allowed       -0.043165
amenities          -0.075508
longitude          -0.188354
Name: price_display, dtype: float64
```

- As we can see there are highly correlated values with the target which will help us obtain better accuracies in our models, we decided to take values above 0.3 to obtain a good accuracy on both training and testing sets.

```
#####getting top features that affect the target to start using them in the model
corr=df.corr()
top_feature = corr.index[corr['price_display'] > 0.3]
plt.subplots(figsize=(12, 8))
top_corr = df[top_feature].corr()
sns.heatmap(top_corr, annot=True)
plt.show()

y = pd.DataFrame(df['price_display'])
x = df[top_feature].drop(['price_display','CityStateAvg','StateavgRentprice','NoOfRooms','avgBathroomsPrice'],axis=1)
x ##### Features used in the models
```


- The correlation matrix:



- After many trials to perform better in the models we decided to take features into our X dataframe which doing greater than 0.3 but with dropping some of them: CityStateAvg, StateavgRentPrice, NoOfRooms, avgBathroomsPrice as we found that they are similar to values of other features.
- So X has now the 6 features that is ready to predict the target variable which is in Y dataframe as in the following figure:

	bathrooms	square_feet	CityavgRentprice	extracted_price	AvgAreaPrice	Trial
0	3	800	1152.095238	0.0	1204.267368	2512.306290
1	1	795	1602.509259	1370.0	1456.708920	2442.027072
2	1	560	1287.750000	1009.0	1251.942308	2009.767813
3	1	600	952.650794	695.0	936.020921	1694.668606
4	3	1600	3084.543046	3695.0	2816.368957	4864.754099
...
8995	1	875	996.950000	0.0	2013.531191	1896.467813
8996	1	824	2102.277778	2035.0	1774.752119	2956.295590
8997	1	844	693.777778	424.0	825.902439	1577.795590
8998	1	489	1484.166667	0.0	2816.368957	2190.684479
8999	2	1066	2114.250000	2398.0	2816.368957	3266.408285

X: 9000 rows × 6 columns

Y
#####Target

	price_display
0	1194
1	1370
2	1009
3	695
4	3695
...	...
8995	1158
8996	2035
8997	424
8998	1417
8999	2398

Y: 9000 rows × 1 columns

2.9 Train Test Split

- Before splitting the data into training dataset and testing dataset we should scale our features we selected to be on the same scale so we used object from the StandardScaler() object and scaled X.

```
#####scaling the features#####  
sc_X = StandardScaler()  
X = sc_X.fit_transform(X)
```

- Then we started to split the data to 70% for training and 30% for testing by using the train_test_split and giving it test_size=0.3 why 30%? because we had 9k rows.

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=42)
```

Finally, we are ready to use the data in our machine learning models.

3.Models

In this section in each model we quickly introduce the model, We then used also different performance metrics to try and evaluate our model train and test prediction abilities, the most common ones are MSE and R2-score(Coefficient of determination) which indicates how much of our target variable is represented by the independent variables, it represents how good our regression model fits the data. Finally, We visualize the model with different plots to see how well it's fitting the data.

3.1 Linear Regression

3.1.1 Introduction

We used our training data to fit the model and then used the model to predict the test data using the fitted line , this model gave an accuracy (using r2-score) of roughly 83% on the training data and 64.6% on the test data, which points out that this was not the best suitable model for our data, and that our data is not best represented by a line.

3.1.2 Error/Accuracy

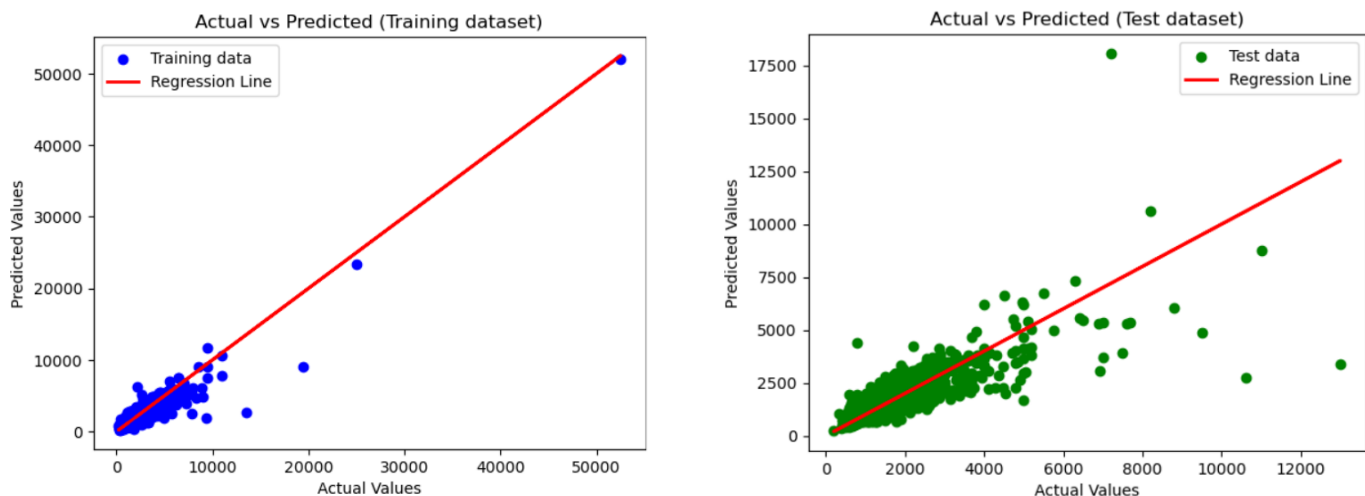
the MSE(mean square error) is not relatively that high due to ranges of the target column, train and test errors are close compared to each other which is a good indicator that the model

generalizes the data well, but it was relatively high compared to other models, for simplicity we can compare using the MAE which is the highest yet in our models.

- linear regression without cross validation

```
MAE: 296.4586667417895
MSE: 296387.03088831186
RMSE: 544.4143926167932
The accuracy of train model is 83.247%
The accuracy of test model is 64.579%
Train Mean Square Error 223488.24204759503
Test Mean Square Error 296387.03088831186
```

3.1.3 Visualization



Here we see the difference between predicted and actual values of the training data which shows that the model did a not so bad job predicting training data.

3.1.4 Improvements

- After trying to enhance the Linear Regression using **cross validation** and trying it with different number of folds till getting the best out of it, we choose the number of folds to be 5 which gave us test accuracy of 74% which is a lot better than the linear regression model fitted using the train test split data.

```
Cross-Validated Metrics:
MAE: 289.8887171676211
MSE: 290426.35631699
RMSE: 538.9121972241768
R2 Score: 0.7548800889213549
Cross-Validated R2 Scores: [0.77662797 0.83462861 0.6667373 0.71319957 0.72077763]
Mean R2 Score: 0.7423942157509918
Standard Deviation of R2 Scores: 0.057832010069375196
Training R2 Score: 0.7944100241826565
```

3.2 Ridge Regression

3.2.1 Introduction

For better predictions for the linear regression model we used ridge regression, having penalizing the weights of the features to reduce overfitting and trying different penalty cost(the alpha hyperparameter) ridge regression was slightly better than linear regression in predicting the test data with test accuracy 67.9%, but the accuracy of the training data was slightly lower which was roughly 80.88%, which assured us that our data can't be fitted well through a line and it's not the best suitable way to predict it.

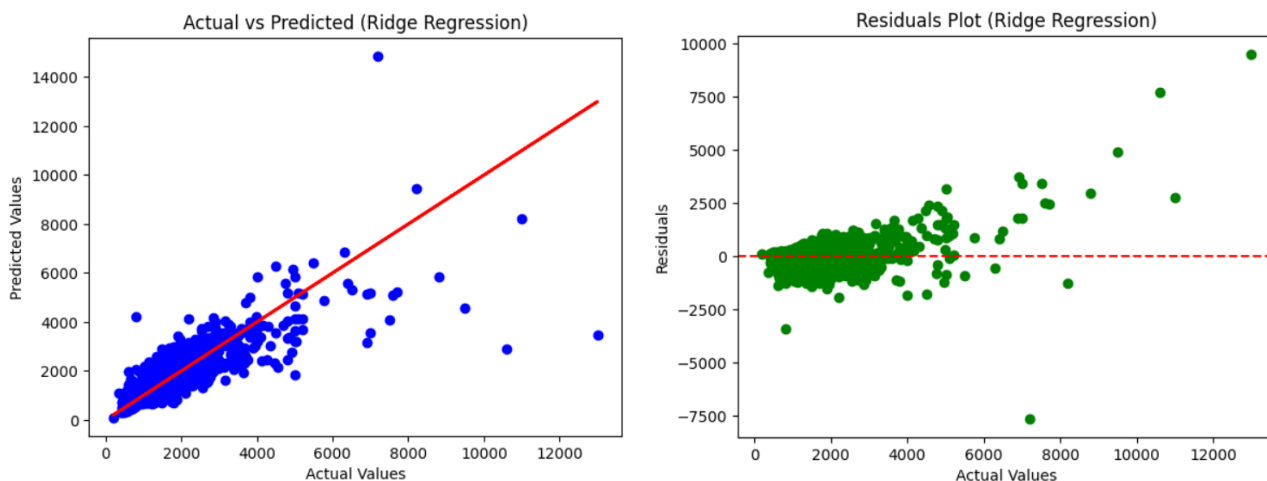
3.2.2 Error/Accuracy

The errors didn't differ a lot, but test accuracy improved.

```
MAE: 289.5327510590954
MSE: 267951.73598740174
RMSE: 517.6405470859115
The accuracy of train model is 80.887%
The accuracy of test model is 67.977%
Train Mean Square Error 254959.12100098346
Test Mean Square Error 267951.73598740174
```

3.2.3 Visualization

We visualized the regression plot on the test data only, and then we visualized it using Residuals Plot which if the points are scattered around the 0 that means that our model fits well. As you can see it is not fitting that good as there are some values scattered away from the 0.



3.3 Polynomial Regression

3.3.1 Introduction

Trying to increase the degree of the features to 2 gave us better accuracy for the model on both the test and the train data, after trying different degrees we decided to stick with 2 which gave us the best accuracy and to avoid overfitting due to the increase of features. Accuracy on the train data is 89.7% and accuracy on the test data is 81.3% which was better than linear regression in predicting on both the train and the test data.

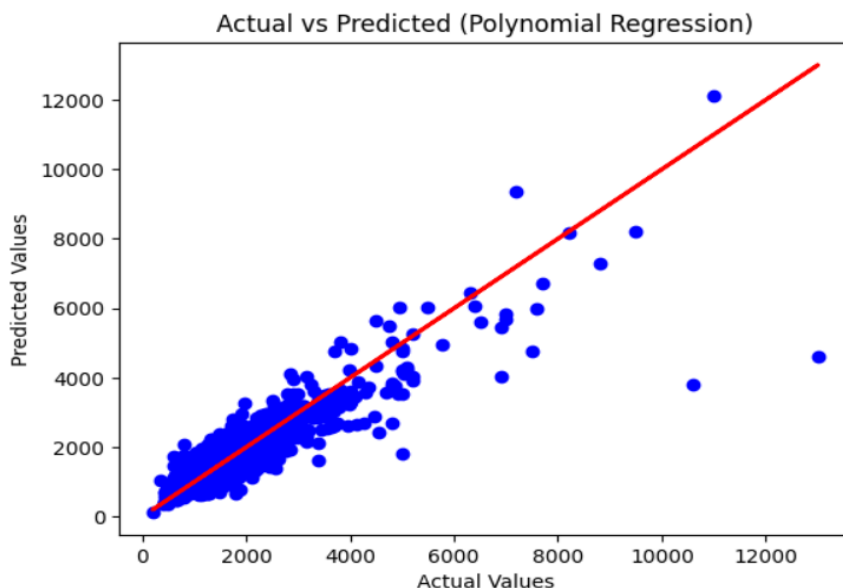
3.3.2 Error/Accuracy

We can notice that the error is decreased and the accuracy is improved.

```
MAE: 232.55821628440245
MSE: 156768.05796835865
RMSE: 395.93946250450796
The accuracy of train model is 89.666%
The accuracy of test model is 81.265%
Train Mean Square Error 137857.1999577
Test Mean Square Error 156768.05796835865
```

3.3.3 Visualization

We visualized the regression plot on the test data only.



3.4 Decision Tree Regressor

3.4.1 Introduction

Having different hyperparameters that majorly affects the overall performance of the model, like 'max depth' and so on, we starting **tuning the hyperparameters** to get the best hyperparameters to use it in the model and get the best values for our model. Our model's best accuracy possible was 97.9% for the train data and 73.9% for the test data which is good but not the best yet given that the polynomial model gave higher accuracy, but this gave us an intuition to try different ensemble techniques using multiple decision trees for better performance.

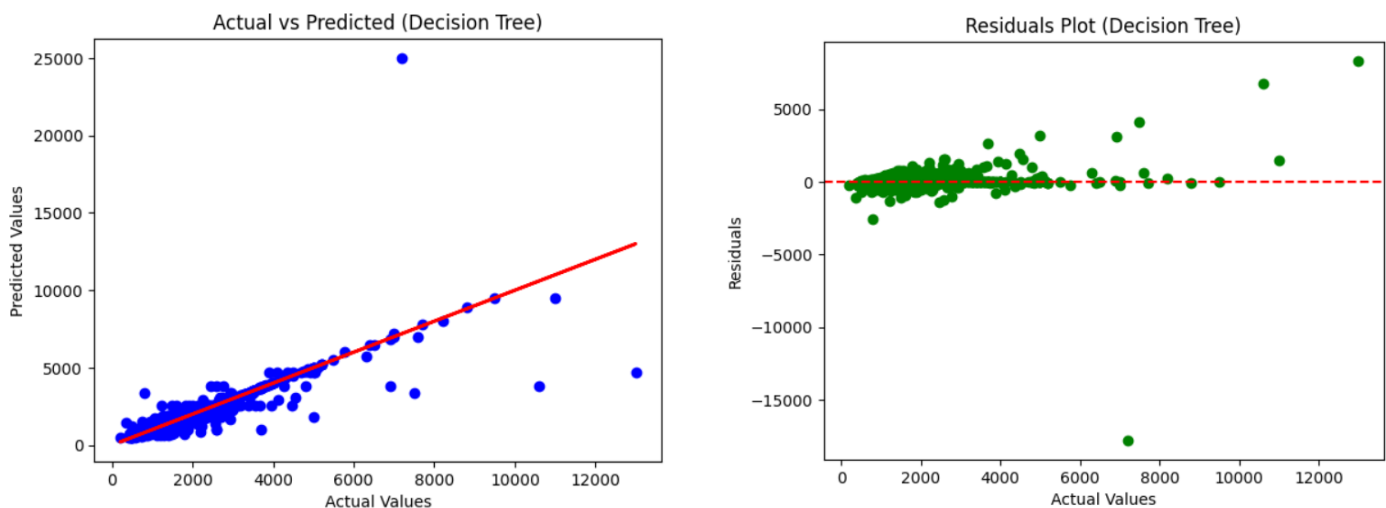
3.4.2 Error/Accuracy

MSE decreased on Train data but on Test data is very high so the model didn't generalize good.

```
MAE: 112.31493413498198
MSE: 217879.00961582194
RMSE: 466.77511674876365
The accuracy of train model is 97.895%
The accuracy of test model is 73.962%
Train Mean Square Error 28081.48799964535
Test Mean Square Error 217879.00961582194
```

3.4.3 Visualization

We visualized the regression plot on the test data and We visualized the Residuals plot.



3.5 Gradient Boosting Regressor

3.5.1 Introduction

Trying to improve the prediction performance of our model we used an ensemble technique which is gradient boosting which uses multiple decision trees to improve its weights sequentially to minimize the error based on the previous iteration, this worked well with our data with train accuracy of 96.3% and test accuracy of 82.9%, which assured our intuition for the decision tree model.

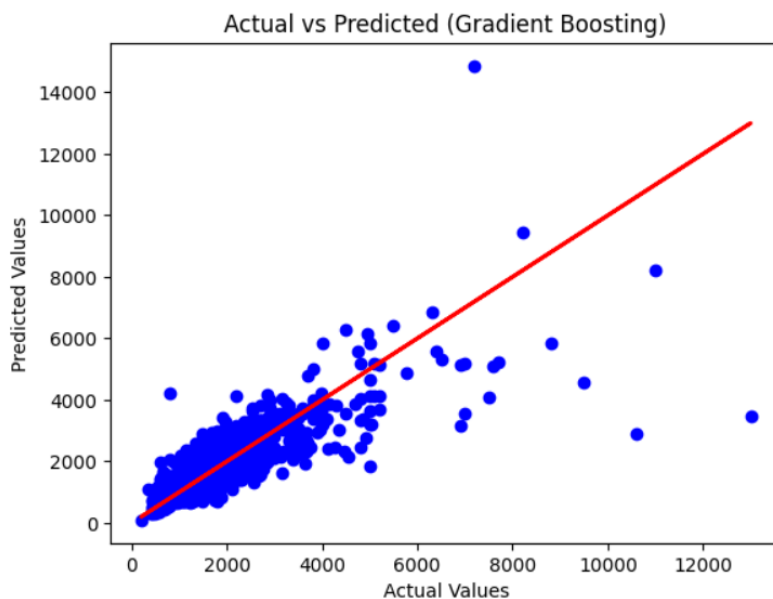
3.5.2 Error/Accuracy

We can see that the MSE decreased compared to that of the decision tree algorithm on test data which succeeded in generalizing better than decision tree.

```
MAE: 132.76761569025686
MSE: 142899.0129278887
RMSE: 378.0198578486171
The accuracy of train model is 96.342%
The accuracy of test model is 82.922%
Train Mean Square Error 48793.82397093042
Test Mean Square Error 142899.0129278887
```

3.5.3 Visualization

We visualized the regression plot on the test data only.



3.6 Random Forest

3.6.1 Introduction

Yet another ensemble technique but this time a bagging method with the majority voting of the decision trees, this gave us best accuracy yet which was 94.7% for the train data and 89.8 % for the test data, which points out that this was the most suitable model for our data, this was the best model in predicting values of the test data.

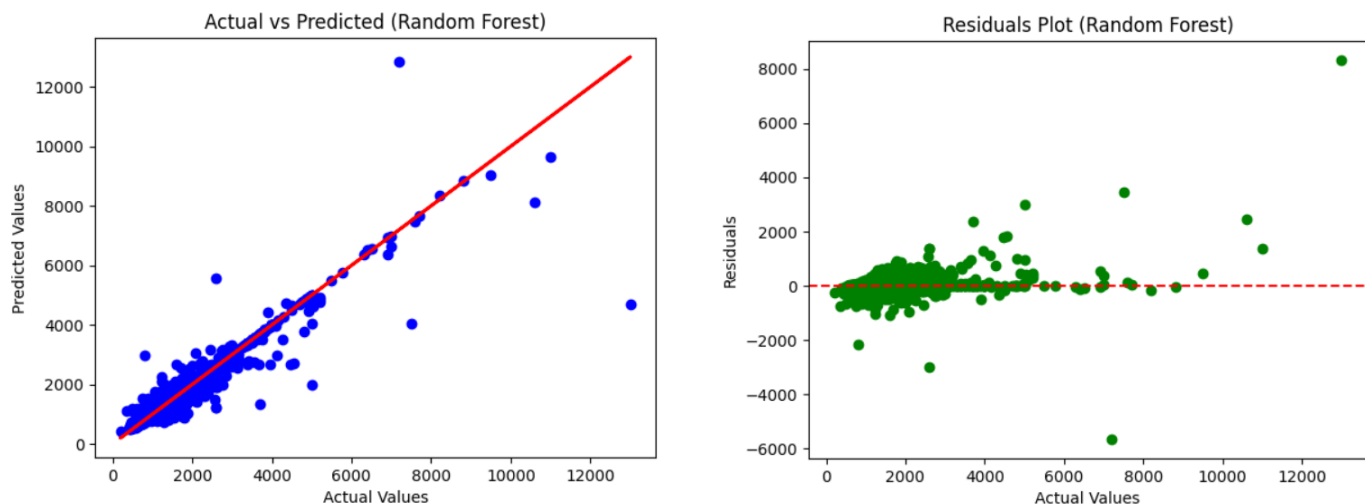
3.6.2 Error/Accuracy

We can notice that this gave us the best generalized model (with the best test accuracy among them all)

```
MAE: 91.17562585610386
MSE: 85719.02301747478
RMSE: 292.77811225819937
The accuracy of train model is 94.682%
The accuracy of test model is 89.756%
Train Mean Square Error 70939.03477350977
Test Mean Square Error 85719.02301747478
```

3.6.3 Visualization

Here we visualized the regression plot on the test data and Residuals Plot as well.



3.7 XGBoost

3.7.1 Introduction

Using another ensemble technique, extreme gradient boosting which is similar to gradient boosting but with some improved aspects like regularization (adding a penalty term like L1 or L2 regression) and parallelization that speeds up the training process, since the GBM

performed well we tried this model which gave a better accuracy of 98.2% for the train data and 86.7% for the test data.

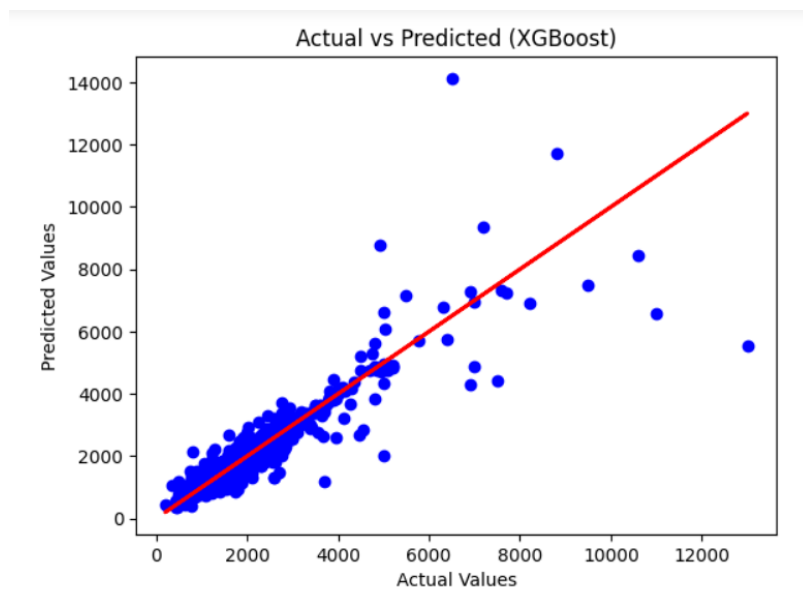
3.7.2 Error/Accuracy

We noticed that it improved the accuracy of both training model and testing model compared to the Gradient Boosting.

```
MAE (XGBoost): 111.71510357892072
MSE (XGBoost): 111461.65338733941
RMSE (XGBoost): 333.85873268096407
The accuracy of train model is 98.153%
The accuracy of test model is 86.679%
Train Mean Square Error 24639.263177273922
Test Mean Square Error 111461.65338733941
```

3.7.3 Visualization

We visualized the regression plot on the test data only.



We have finished our models, Now let's proceed to have a look on the improvements we did to get those accuracies.

4.Improvements

- New Features used or ignored: after introducing new features we started to see their correlations values and we started to drop or take them based on those values.
- Cross-Validation: as mentioned before we performed cross-validation to get better results on the linear regression model and we achieved higher results.

- Hyper-parameter tuning: we did hyperparameter tuning in almost every model to know the best parameters that can help us achieve more accuracy % and instead of doing tuning at every run we just got the best parameters from tries and fixed them in our models.
- Trying multiple models: wanting to achieve higher accuracies we tried 7 regression models.

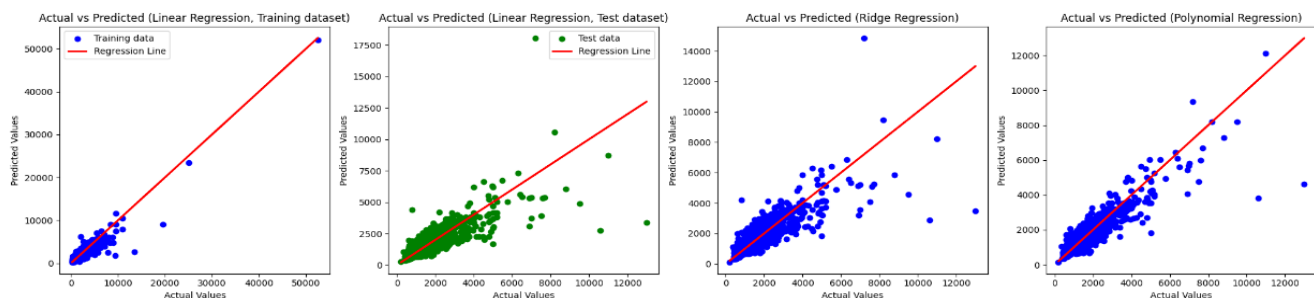
5.Conclusion

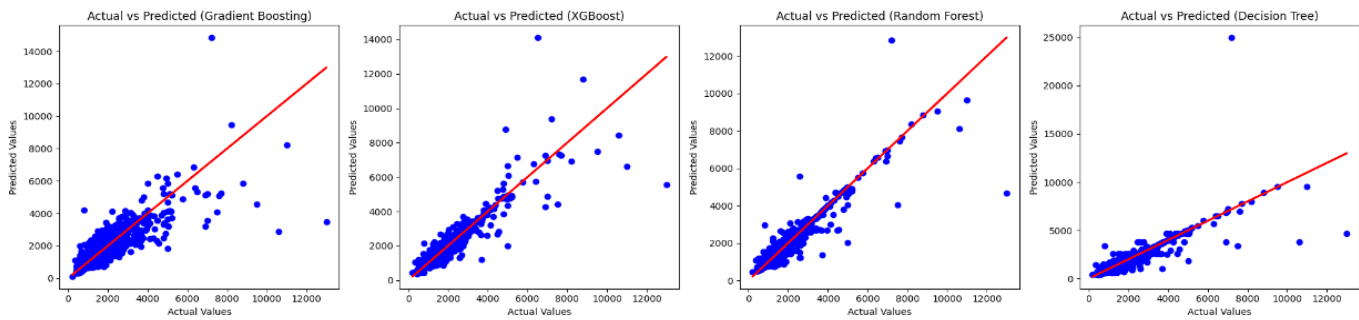
- After comparing the seven models, We noticed that the best generalized one was Random Forest Model, The best one on the training data only was XGBoost Model and the best one on the testing data was also the Random Forest, as shown in the following figure:

Model	MAE	MSE	RMSE	R-squared (Training)	R-squared (Testing)
Linear Regression	296.458667	296387.030888	544.414393	0.832465	0.645792
EnhancedLRwithCV	289.888717	290426.356317	538.912197	0.794410	0.742394
Ridge Regression	289.532751	267951.735987	517.640547	0.808874	0.679774
Polynomial Regression	232.558216	156768.057968	395.939463	0.896657	0.812649
Gradient Boosting	132.767616	142899.012928	378.019858	0.963422	0.829223
Random Forest	91.175626	85719.023017	292.778112	0.946822	0.897558
Decision Tree	112.314934	217879.009616	466.775117	0.978949	0.739616
XGBoost	111.715104	111461.653387	333.858733	0.981530	0.866794

Note the values of the errors can be very small like 0.4 , 0.2 ...etc. if we scaled the Y(target) data as well.

- We also visualized all the models beside each other so we can see the differences in the regression plot as shown in the following figures:





- First look into the data we had some intuitions that later was either proved or disproved:
 1. We thought that the more amenities an apartment have the more price it costs but this was later disproved.
 2. We also thought that the Linear Regression model isn't good for this problem which was later proved.

6. Python Documentation

Python version used:

Python 3.11.2

All Libraries used with the versions:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import itertools
import math
import re
import sklearn
import xgboost as xgb
from sklearn.preprocessing import LabelEncoder
from sklearn import linear_model
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
from scipy.stats import zscore
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, r2_score
```

```
pandas version: 2.0.3
numpy version: 1.25.1
seaborn version: 0.12.2
matplotlib version: 3.7.2
scikit-learn version: 1.3.0
mpl_toolkits.mplot3d version: N/A
itertools version: Built-in
math version: Built-in
LabelEncoder version: Built-in
linear_model version: 1.3.0
metrics version: 1.3.0
train_test_split version: 1.3.0
ExtraTreesClassifier version: 1.3.0
PolynomialFeatures version: 1.3.0
Ridge version: 1.3.0
DecisionTreeRegressor version: 1.3.0
GradientBoostingRegressor version: 1.3.0
RandomForestRegressor version: 1.3.0
r2_score version: 1.3.0
StandardScaler version: 1.3.0
XGBoost version: 2.0.3
```