

IS6052 – PREDICTIVE ANALYTICS INDIVIDUAL REPORT

Submitted by: Mrithul Madhu Kumar - 124104690

Submitted to: Prof Hadi Karimikia

Submitted on: 09/12/2024

TABLE OF CONTENTS

Introduction	3
Dataset Description	3
Issues noticed	5
Data cleaning and preparation.....	6
Handling missing values	9
Feature Engineering	10
Outlier detection and Handling	10
Predictive Analysis.....	21
Linear Regression.....	23
Random Forest Regressor	25
Hyper Parameter Tuning using GridSearchCV	25
Classifications	26
Logistic Regression.....	26
Handling Imbalance in the data.....	27
Decision Tree	29
Random Forest Classifier	30
Hyper Parameter Tuning for the Random Forest models	30
Results, Evaluation and Discussion	33
Logistic Regression.....	33
Decision Tree	35
Random Forest	36
Linear Regression.....	37
Random Forest Regressor	40
Conclusion.....	41
Use of AI tools	41
References	41

INTRODUCTION

Predictive analytics project was to analyse factors that impact a potential home buyer in Dublin while purchasing a house. The dataset was provided and predictive models were to be created to conduct analysis to reach meaningful conclusions.

DATASET DESCRIPTION

Dataset has 13320 rows and 12 columns where; ID, Bath, balcony and price-per-sqft-\$ contains numerical values, and, property score, availability, location, size, total_sqft, buying or not buying, BER, renovation needed are categorical values stored as object datatype. price-per-sqft-\$, bath and balcony are Ratio data, ID and BER are Ordinal data and, the remaining are Nominal data currently.

0	ID	13320	non-null	int64
1	property_scope	13320	non-null	object
2	availability	13320	non-null	object
3	location	13319	non-null	object
4	size	13304	non-null	object
5	total_sqft	13320	non-null	object
6	bath	13247	non-null	float64
7	balcony	12711	non-null	float64
8	buying or not buying	13320	non-null	object
9	BER	13320	non-null	object
10	Renovation needed	13320	non-null	object
11	price-per-sqft-\$	13074	non-null	float64

Fig 1: Dataset info

Column “property_scope” has 4 unique values, “location” has 5 unique values, “balcony” has 4 unique values, “buying or not buying” has 2 unique values, “BER” has 7 unique values, and “renovation needed” has 3 unique values.

df.nunique()
✓ [5] 22ms

	123 <unnamed>
ID	13320
property_scope	4
availability	81
location	5
size	30
total_sqft	2117
bath	19
balcony	4
buying or not buying	2
BER	7
Renovation needed	3
price-per-sqft-\$	7536

Fig 2: no of unique values in columns

Unique values in column : property_scope Extended Coverage 8790 Constructed Space 2418 Land Parcel 2025 Usable Interior 87 Name: count, dtype: int64 -----	----- Unique values in column : balcony 2.0 5113 1.0 4897 3.0 1672 0.0 1029 Name: count, dtype: int64 -----
Unique values in column : location Fingal 4875 DCC 3030 South Dublin 2610 Dun Laoghaire 2324 Other 480 Name: count, dtype: int64 -----	Unique values in column : buying or not buying No 9057 Yes 4263 Name: count, dtype: int64 -----

Fig 3a, 3b : unique values for columns

```

-----
Unique values in column : BER
C      1982
B      1955
E      1933
A      1883
F      1864
G      1859
D      1844
Name: count, dtype: int64
-----

Unique values in column : Renovation needed
Yes      7500
Maybe   3937
No       1883
Name: count, dtype: int64
-----

```

Fig 3c : unique value in column BER

ISSUES NOTICED

From the preliminary analysis we can identify that out of the 12 columns, only 7 columns do not contain any missing values. Location contains only 1 null value, size contains 16 null values, bath contains 73 null values, balcony contains 609 null values, and price-per-sqft-\$ contains 246 null values. Column such as total_sqft is stored as object data type but contains numerical data which requires transformation in the cleaning phase. Features like size include descriptive text (e.g., "2 BED", "4 Bedroom") that may need standardization.

It is also noticed that for the column total_sqft, some values are not in square feet but are in acres, yards, grounds etc. some values are in continuous form in a range for example "3090 - 5002". For the column availability, most values are in the form "ready to move" and rest are dates.

DATA CLEANING AND PREPARATION

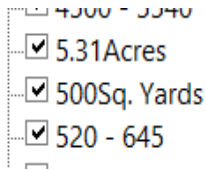


Fig 4: units in total_sqft column

I meticulously found out the different values in column “total_sqft”. These included values are different units such as acres, yards, meters, perch, guntha, grounds etc. I created a function in python to convert all these values to square feet and applied it to “total_sqft” column converting the values to standardised form and making the datatype of the column to float.

The next step I utilised was to standardise the “size” column. The numeric values of the column were separated and used to create a new column “size_by_no_of_bedrooms” and converted to float datatype. The older size column was discarded using the “drop(columns)” function since it was no longer needed. All the numeric columns the dataframe were then rounded off to 2 decimal points.

```
def convert_to_sqft(value):
    if pd.isna(value):
        return value
    value = str(value).strip()
    # Check for range values and take the average
    if '-' in value:
        values = value.split('-')
        values = [convert_to_sqft(v.strip()) for v in values]
        return sum(values) / len(values)
    # Conversion factors
    conversion_factors = {
        'sq. meter': 10.7639,
        'Sq. Meter' : 10.7639,
        'sq. meters': 10.7639,
        'sq. yard': 9,
        'sq. yards': 9,
        'Sq. Yards' : 9,
        'acre': 43560,
        'acres': 43560,
        'Acres': 43560,
        'perch': 272.25,
        'Perch': 272.25,
        'perches': 272.25,
        'meter': 10.7639,
        'meters': 10.7639,
        'yard': 9,
        'yards': 9,
        'cents' : 435.6,
        'Cents': 435.6,
        'Grounds' : 2400.350017649,
        'grounds' : 2400.350017649,
        'Guntha' : 1089,
        'guntha' : 1089
    }
```

Fig 5a: code to standardise units in the total_sqft column

```

def convert_to_sqft(value):
    # Check for units and convert
    for unit, factor in conversion_factors.items():
        if unit in value:
            num = float(re.findall(r'\d+\.?d*', value)[0])
            return num * factor

    # If no unit is found, assume it's already in sqft
    try:
        return float(re.findall(r'\d+\.?d*', value)[0])
    except (ValueError, IndexError):
        return None

# Apply the conversion function to the total_sqft column
df2['total_sqft'] = df2['total_sqft'].apply(convert_to_sqft)
✓ [57] 89ms
    
```

```
df2.info()
```

```
✓ [58] 15ms
```

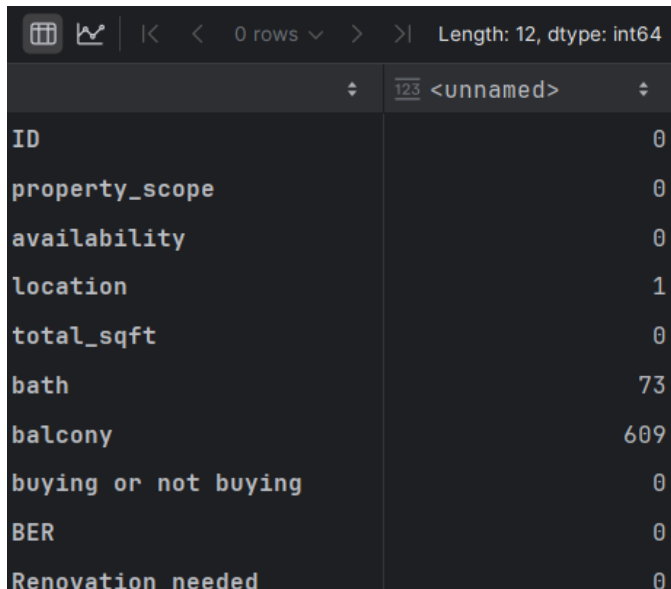
```

1  property_scope      13320 non-null object
2  availability        13320 non-null object
3  location            13319 non-null object
4  size                13304 non-null object
5  total_sqft          13320 non-null float64
6  bath                13247 non-null float64
7  balcony             12711 non-null float64
8  buying or not buying 13320 non-null object
9  BER                 13320 non-null object
10 Renovation needed    13320 non-null object
11 price-per-sqft-$     13074 non-null float64
dtypes: float64(4), int64(1), object(7)
    
```

Fig 5b: code to standardise units in the total_sqft column

HANDLING MISSING VALUES

The initial checking of dataframe showed multiple missing values in it.



	Count
ID	0
property_scope	0
availability	0
location	1
total_sqft	0
bath	73
balcony	609
buying or not buying	0
BER	0
Renovation needed	0

Fig 6: missing value counts

Missing values in columns like “size_by_no_of_bedrooms”, “price-per-sqft-€”, and “balcony” were replaced with respective median values since the median value is not affected by outliers. For categorical values like column “bath”, the missing values were replaced using mode of the column since it is the most frequent value in the column. “Location” column had one missing value, but since replacing the missing value didn’t seem logical, the row containing the missing value was dropped.

```

#filling size column with median values for missing data
df2['size_by_no_of_bedrooms'] = df2['size_by_no_of_bedrooms'].fillna(df2['size_by_no_of_bedrooms'].median())
df2['price-per-sqft-€'] = df2['price-per-sqft-€'].fillna(df2['price-per-sqft-€'].median()).round(2)
df2['balcony'] = df2['balcony'].fillna(df2['balcony'].median())#since median and mode is the same
✓ [93] < 10 ms

#Filling missing value with mode for categorical data
imputer = SimpleImputer(strategy='most_frequent')
df2['bath'] = imputer.fit_transform(df2[['bath']])
✓ [97] 12ms

#dropping row where location is unknown
df2 = df2.dropna(subset=['location'])
✓ [99] 20ms
  
```

Fig 7: code to handle missing values

After confirming that no missing values remain in the dataframe, I moved to next phase of handling the outliers.

FEATURE ENGINEERING

Throughout the project multiple feature engineering was created like making dummies for categorical columns such as Location, property_scope, BER and renovation_needed_or_not. A separate column total-price was created by multiplying values in total_sqft and Price_per_sqft which played a huge role in regression analysis. Similarly, standardization was done to check for improved performance in linear regression. Also, SMOTE method was utilised to oversample minority classes to increase accuracy of models.

OUTLIER DETECTION AND HANDLING

Box plots plotted in the EDA phase gave a clear understanding that there were significant outliers in the dataset. The columns with numerical data were plotted using the box plots to find outliers in them.

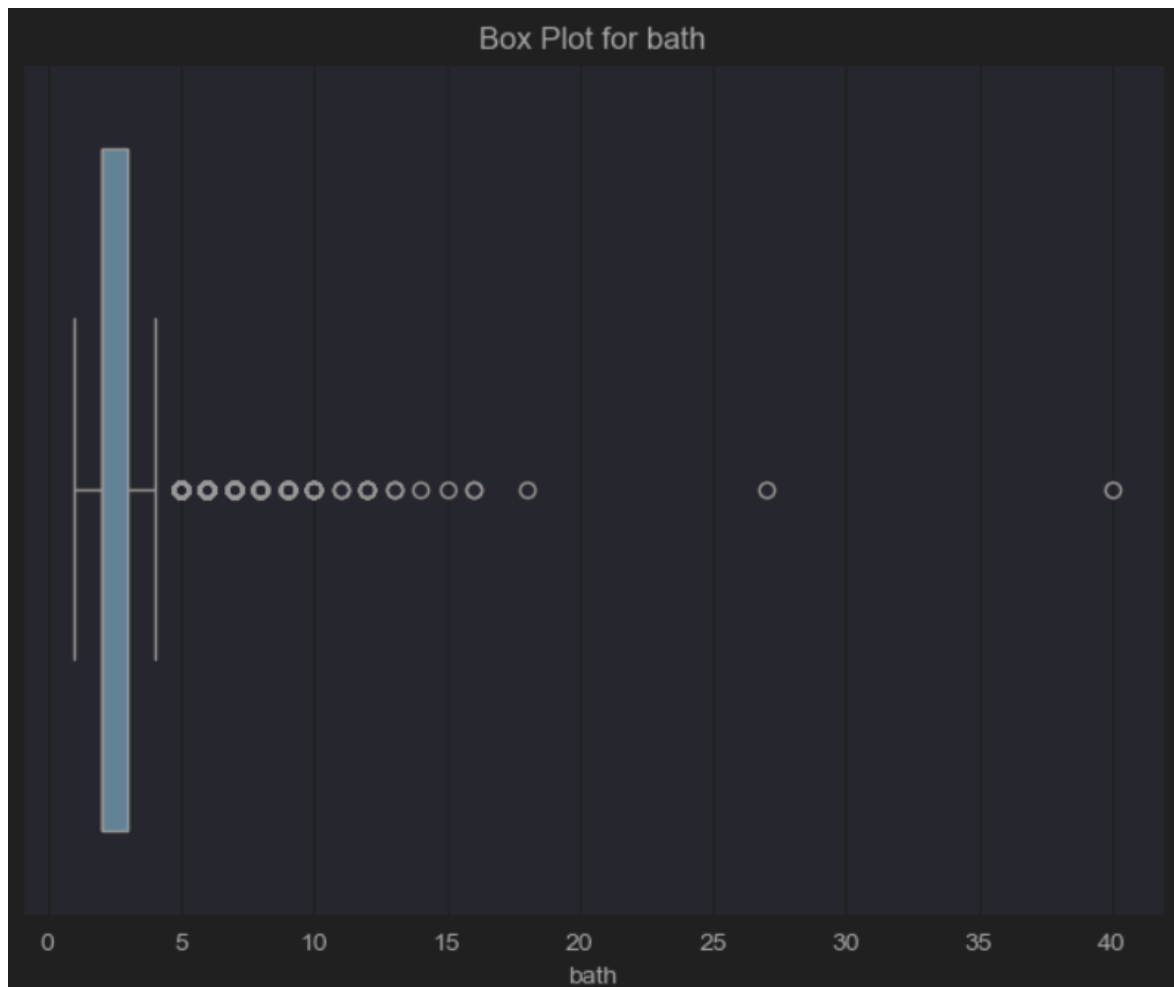
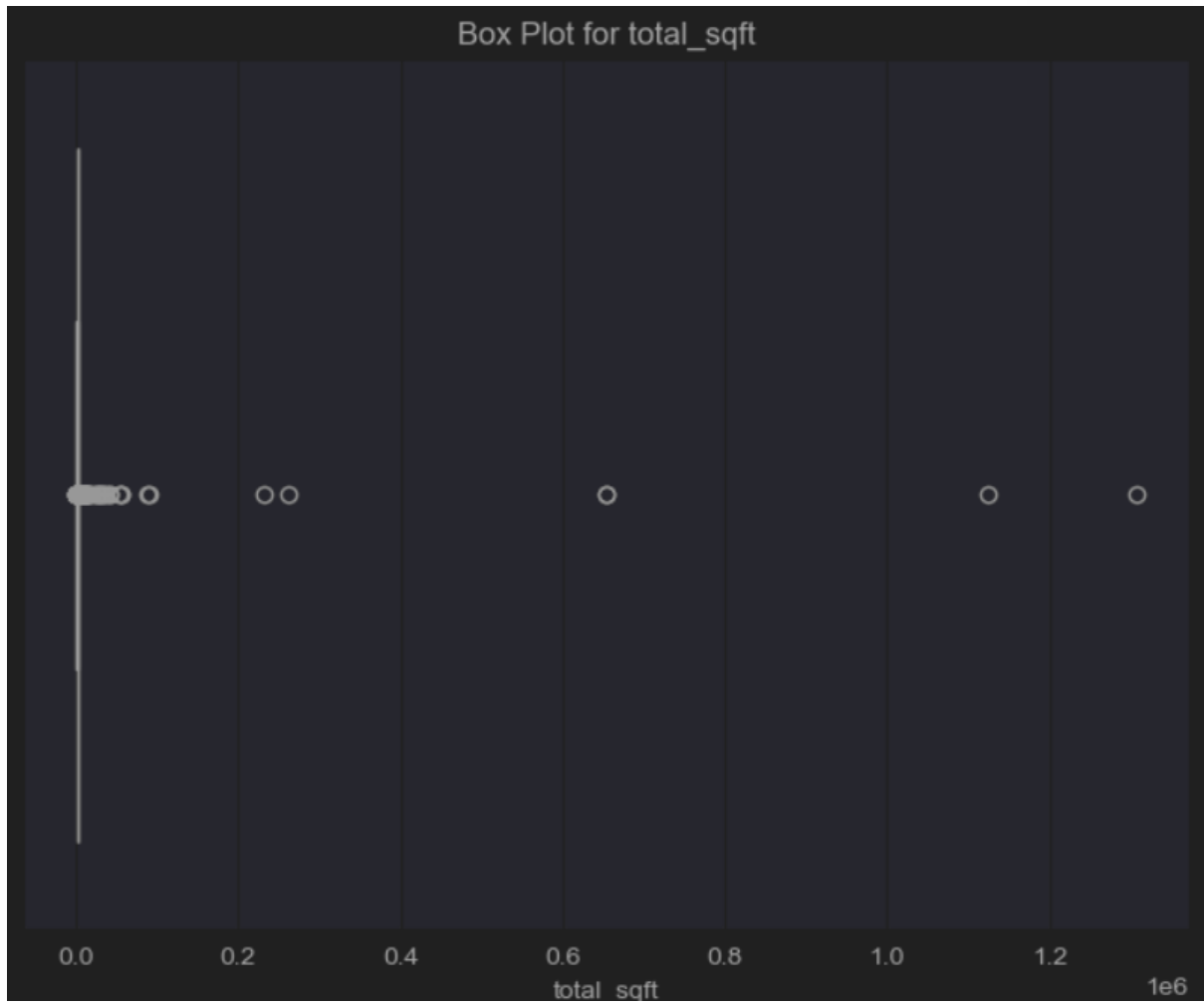
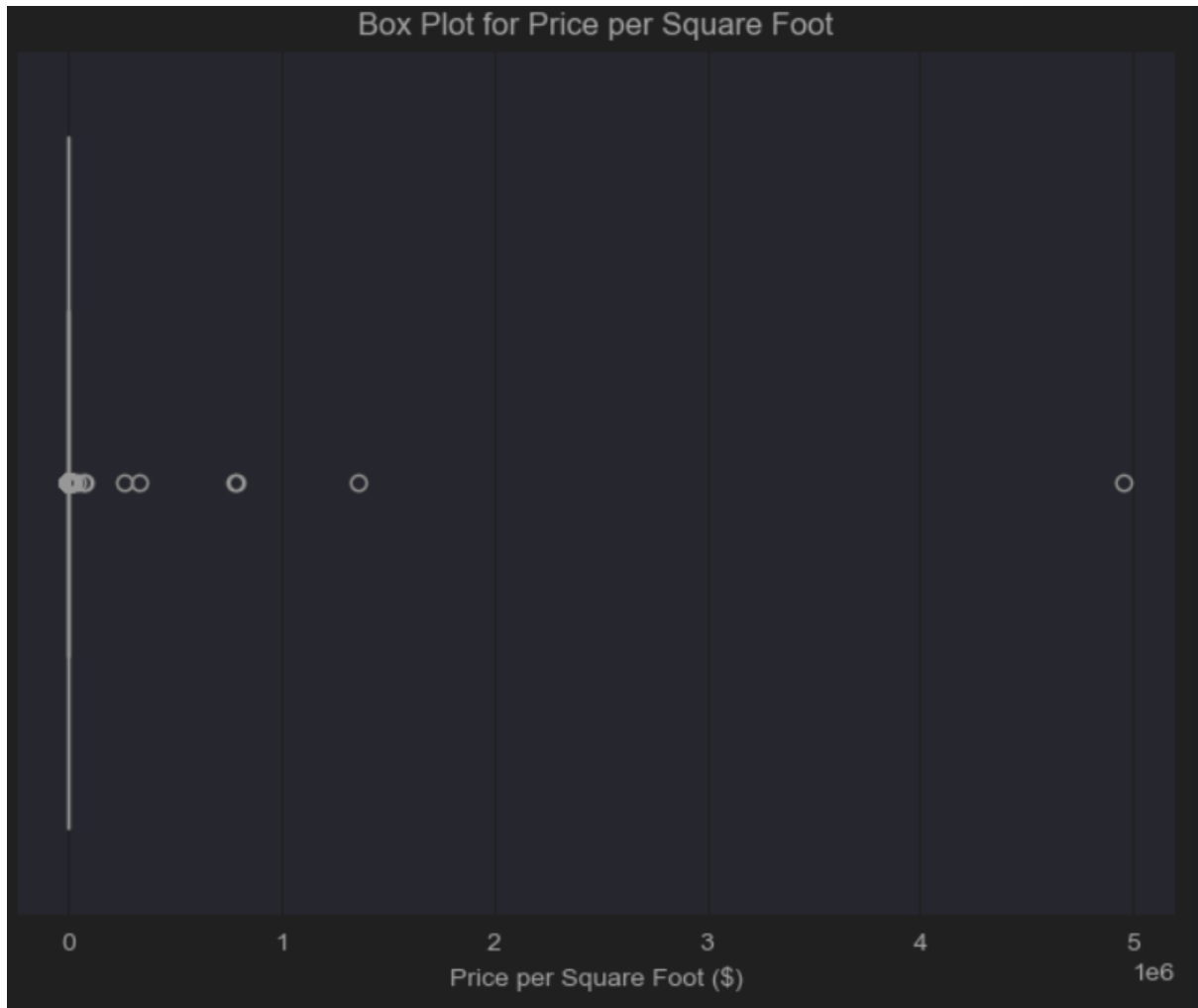
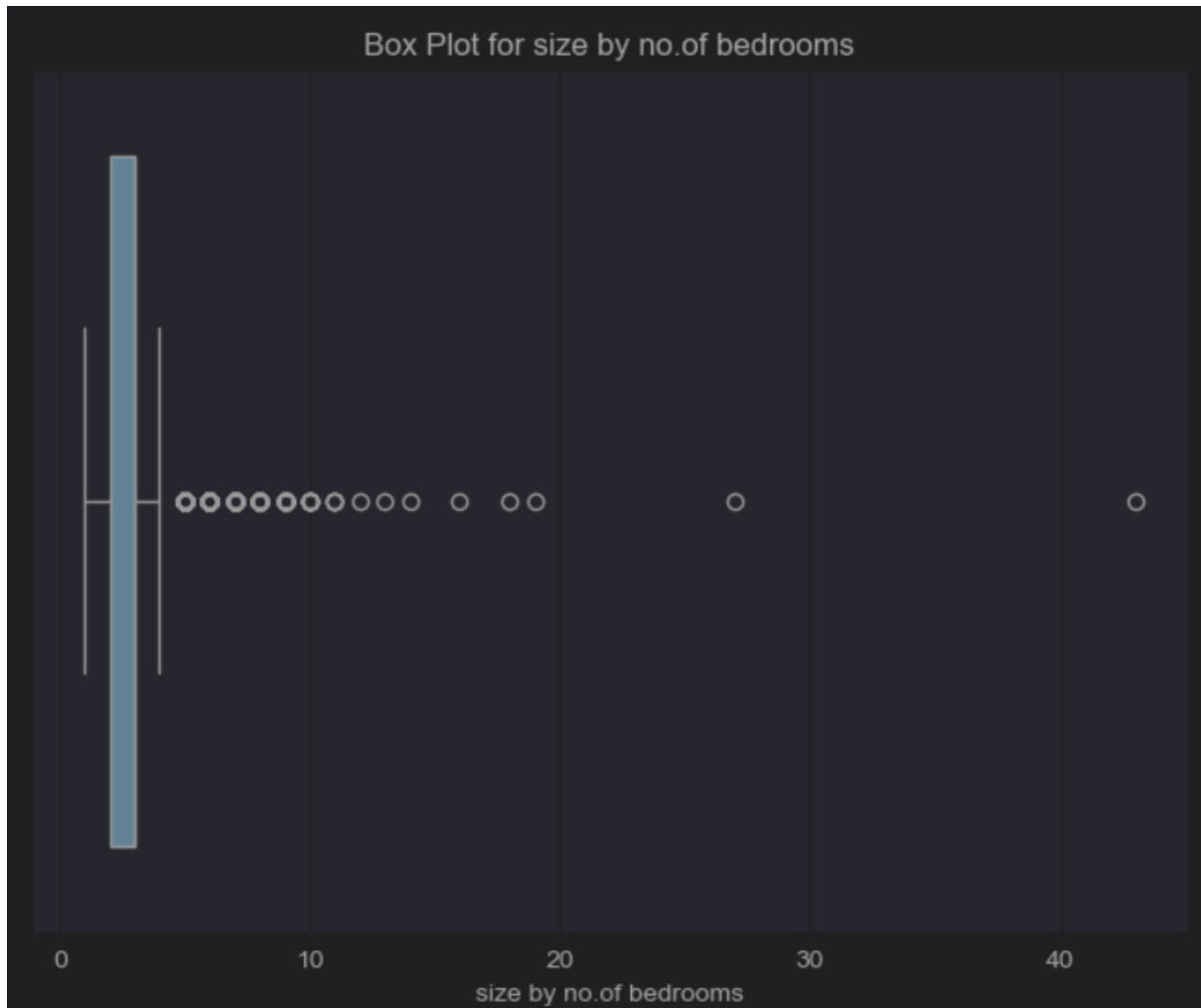
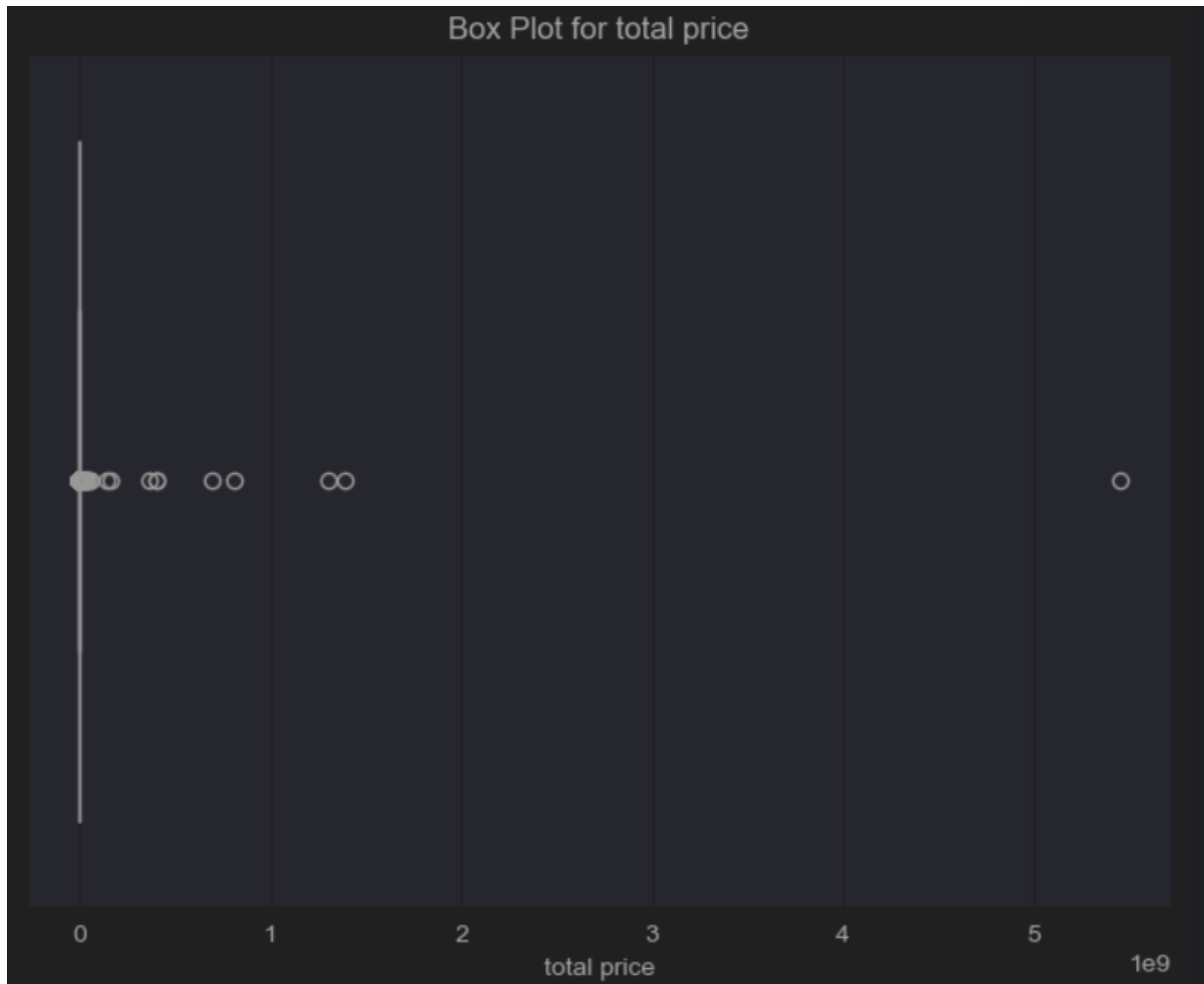


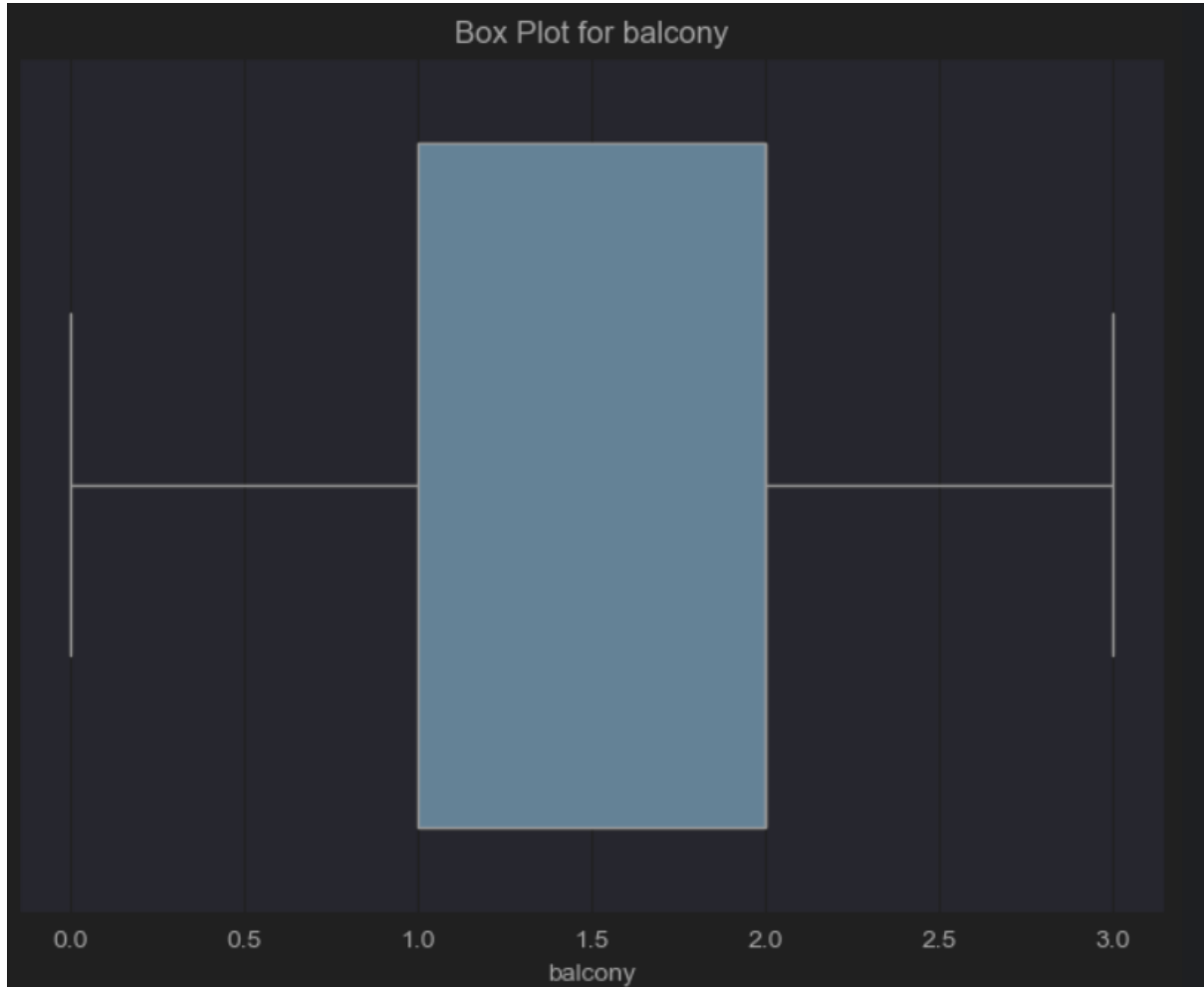
Fig 8: box plot for bath











The box plots helped in identifying that, other than the column “balcony”, all the rest of the numerical columns had significant number of outliers.

I decided to use Inter Quartile Range method to find and replace outliers. To implement this, values at the 25th percentile and 75th percentile were found for each column and stored them in variables namely Q1 and Q3. Inter Quartile Range is calculated by subtracting Q1 from Q3 and the difference is stored in the variable IQR. After calculating IQR, Lower bound is computed by subtracting 1.5 times the IQR from Q1. Similarly, Upper Bound is calculated by adding Q3 with 1.5 times the respective IQR. Lower bound and upper bound values are calculated for the respective columns to since any value lesser than lower bound and bigger than upper bound has to be treated as an outlier.

```
#checking for outliers
Q1_total_sqft = df2['total_sqft'].quantile(0.25)
Q3_total_sqft = df2['total_sqft'].quantile(0.75)
IQR_total_sqft = Q3_total_sqft - Q1_total_sqft
lower_bound_total_sqft = Q1_total_sqft - 1.5 * IQR_total_sqft
upper_bound_total_sqft = Q3_total_sqft + 1.5 * IQR_total_sqft
print('lower_bound_total_sqft :', lower_bound_total_sqft)
print('upper_bound_total_sqft :', upper_bound_total_sqft)

Q1_bath = df2['bath'].quantile(0.25)
Q3_bath = df2['bath'].quantile(0.75)
IQR_bath = Q3_bath - Q1_bath
lower_bound_bath = Q1_bath - 1.5 * IQR_bath
upper_bound_bath = Q3_bath + 1.5 * IQR_bath
print('lower_bound_bath :', lower_bound_bath)
print('upper_bound_bath :', upper_bound_bath)

Q1_size = df2['size_by_no_of_bedrooms'].quantile(0.25)
Q3_size = df2['size_by_no_of_bedrooms'].quantile(0.75)
IQR_size = Q3_size - Q1_size
lower_bound_size = Q1_size - 1.5 * IQR_size
upper_bound_size = Q3_size + 1.5 * IQR_size
print('lower_bound_size :', lower_bound_size)
print('upper_bound_size :', upper_bound_size)

Q1_price = df2['price-per-sqft-$'].quantile(0.25)
Q3_price = df2['price-per-sqft-$'].quantile(0.75)
IQR_price = Q3_price - Q1_price
lower_bound_price = Q1_price - 1.5 * IQR_price
upper_bound_price = Q3_price + 1.5 * IQR_price
print('lower_bound_price_per_sqft :', lower_bound_price)
print('upper_bound_price_per_sqft :', upper_bound_price)
```



```

Q1_price = df2['total_price'].quantile(0.25)
Q3_price = df2['total_price'].quantile(0.75)
IQR_price = Q3_price - Q1_price
lower_bound_total_price = Q1_price - 1.5 * IQR_price
upper_bound_total_price = Q3_price + 1.5 * IQR_price
print('lower_bound_total_price :', lower_bound_total_price)
print('upper_bound_total_price :', upper_bound_total_price)

✓ [366] 29ms

lower_bound_total_sqft : 230.0
upper_bound_total_sqft : 2550.0
lower_bound_bath : 0.5
upper_bound_bath : 4.5
lower_bound_size : 0.5
upper_bound_size : 4.5
lower_bound_price_per_sqft : -23.727500000000134
upper_bound_price_per_sqft : 1336.6925
lower_bound_total_price : -624254.0249999999
upper_bound_total_price : 2553754.815

df2[(df2.total_sqft < lower_bound_total_sqft) | (df2.total_sqft > upper_bound_total_sqft)]

✓ [277] 47ms
    
```

ID	property_scope	availability	location	total_sqft
1	Land Parcel	Ready To Move	South Dublin	2600.0

After finding the lower bound and upper bound for each column, they were used to find out the number of outliers in respective columns. 1193 outliers were found in the column “total_sqft”, 1039 in “bath”, 846 in “size_by_no_of_bedrooms”, 1288 in “price-per-sqft-€”, and 1294 in the column “total_price”.

```
df2[(df2.bath < lower_bound_bath) | (df2.bath > upper_bound_bath)]
✓ [278] 27ms
```

ID	property_scope	availability	location
1	Land Parcel	Ready To Move	South Dub

```
df2[(df2.size_by_no_of_bedrooms < lower_bound_size) | (df2.size_by
✓ [279] 35ms
```

ID	property_scope	availability	location
9	Land Parcel	Ready To Move	Fingal

```
df2[(df2['price-per-sqft-$'] < lower_bound_price) | (df2['price-pe
✓ [280] 22ms
```

property_scope	availability	location	ID
7 Extended Coverage	Ready To Move	Fingal	

```
df2[(df2.total_price < lower_bound_total_price) | (df2.total_price
✓ [281] 21ms
```

ID	property_scope	availability	location
7	Extended Coverage	Ready To Move	Fingal

My next step was to address these outliers. Since the number of outliers were high, dropping the rows containing the outliers was not a good choice since it would result in data loss. To avoid this, capping was used to remove the outliers. Capping is the process where values lesser than the Lower bound and values bigger than the Upper bound value of the respective columns will be replaced with respective lower and upper bound value of the same column. This way the outliers are effectively removed by updating them and there is no data loss which may affect the analysis.

```

#capping outliers to upper or lower limit values
new_df = df2.copy()
new_df.loc[(new_df['total_sqft']>upper_bound_total_sqft),'total_sqft'] = upper_bound_total_sqft
new_df.loc[(new_df['total_sqft']<lower_bound_total_sqft),'total_sqft'] = lower_bound_total_sqft
new_df.loc[(new_df['bath']>upper_bound_bath),'bath'] = upper_bound_bath
new_df.loc[(new_df['bath']<lower_bound_bath),'bath'] = lower_bound_bath
new_df.loc[(new_df['size_by_no_of_bedrooms']>upper_bound_size),'size_by_no_of_bedrooms'] =
    upper_bound_size
new_df.loc[(new_df['size_by_no_of_bedrooms']<lower_bound_size),'size_by_no_of_bedrooms'] =
    lower_bound_size
new_df.loc[(new_df['price-per-sqft-$']>upper_bound_price),'price-per-sqft-$'] = upper_bound_price
new_df.loc[(new_df['price-per-sqft-$']<lower_bound_price),'price-per-sqft-$'] = lower_bound_price
new_df.loc[(new_df['total_price'] < lower_bound_total_price),'total_price'] = lower_bound_total_price
new_df.loc[(new_df['total_price'] > upper_bound_total_price),'total_price'] = upper_bound_total_price

new_df[['total_sqft','bath','price-per-sqft-$','size_by_no_of_bedrooms','total_price']].describe().round(2)

```

✓ [284] 23ms

✓ [367] 58ms

	total_sqft	bath	price-per-sqft-\$	size_by_no_of_bedrooms	total_price
count	13319.00	13319.00	13319.00	13319.00	13319.00
mean	1431.79	2.57	700.44	2.69	1061747.6
std	528.90	0.92	294.57	0.87	677512.3
min	230.00	1.00	30.40	1.00	90800.4
25%	1100.00	2.00	486.43	2.00	567499.2
50%	1277.00	2.00	619.09	3.00	817200.0
75%	1680.00	3.00	826.54	3.00	1362001.5
max	2550.00	4.50	1336.69	4.50	2553754.8

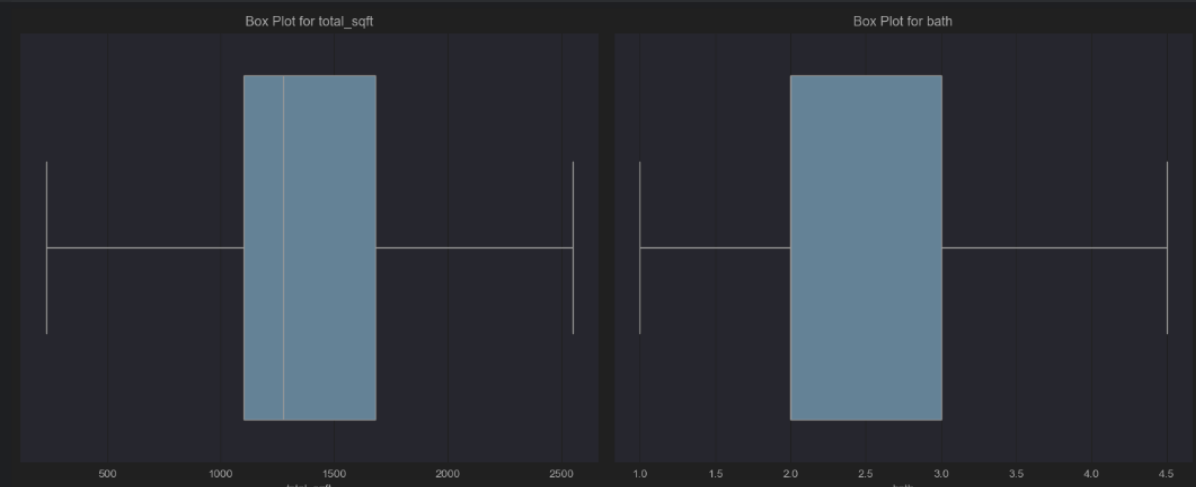
After updating the dataframe with updated values, the numerical columns were again plotted using boxplots to confirm that there were no outliers remaining.

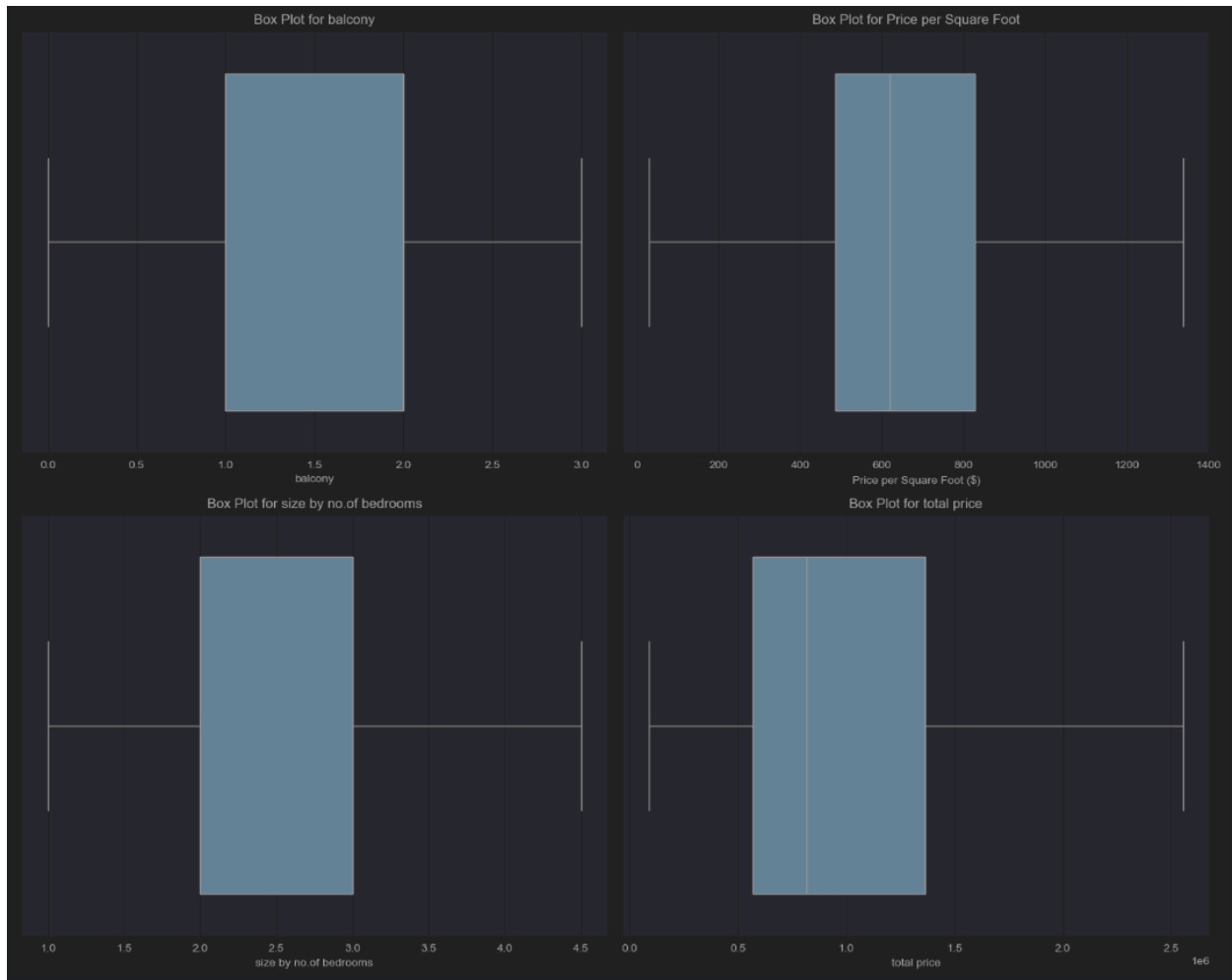
```

# Box plot for 'size by no of bedrooms'
sns.boxplot(x=new_df['size_by_no_of_bedrooms'], ax=axes[2, 0])
axes[2, 0].set_title('Box Plot for size by no.of bedrooms')
axes[2, 0].set_xlabel('size by no.of bedrooms')

# Box plot for 'total_price'
sns.boxplot(x=new_df['total_price'], ax=axes[2, 1])
axes[2, 1].set_title('Box Plot for total price')
axes[2, 1].set_xlabel('total price')

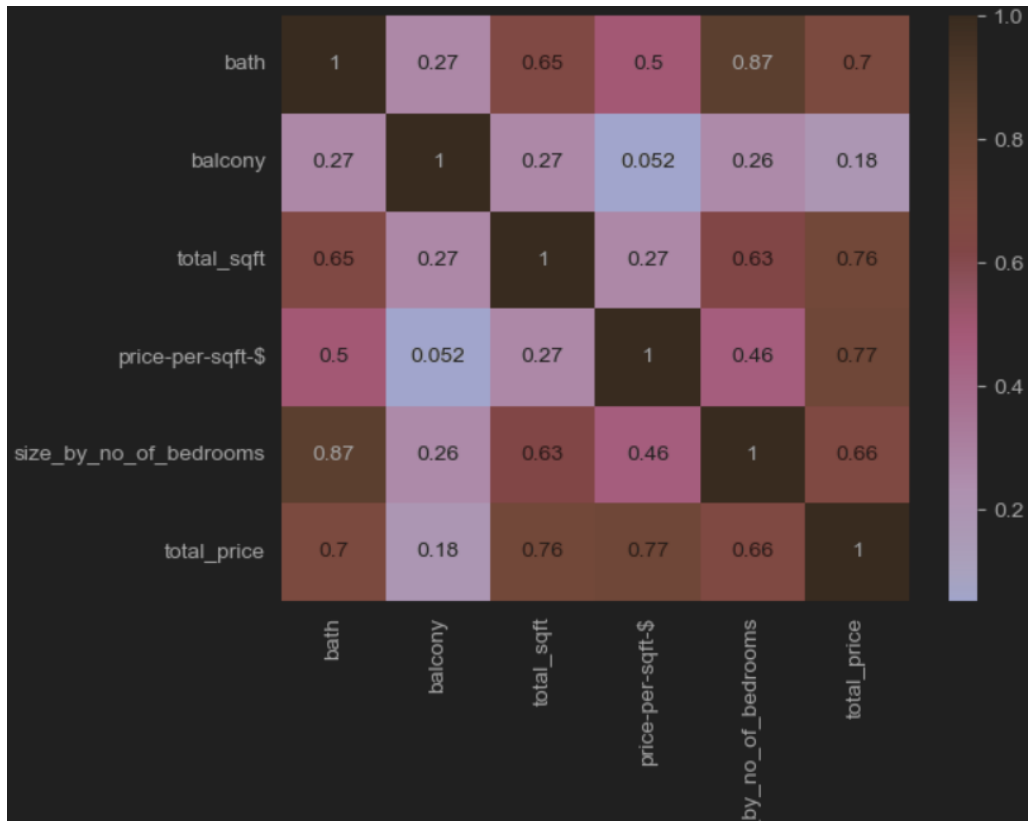
# Adjust layout
plt.tight_layout()
plt.show()
✓ [369] 1s 794ms
    
```



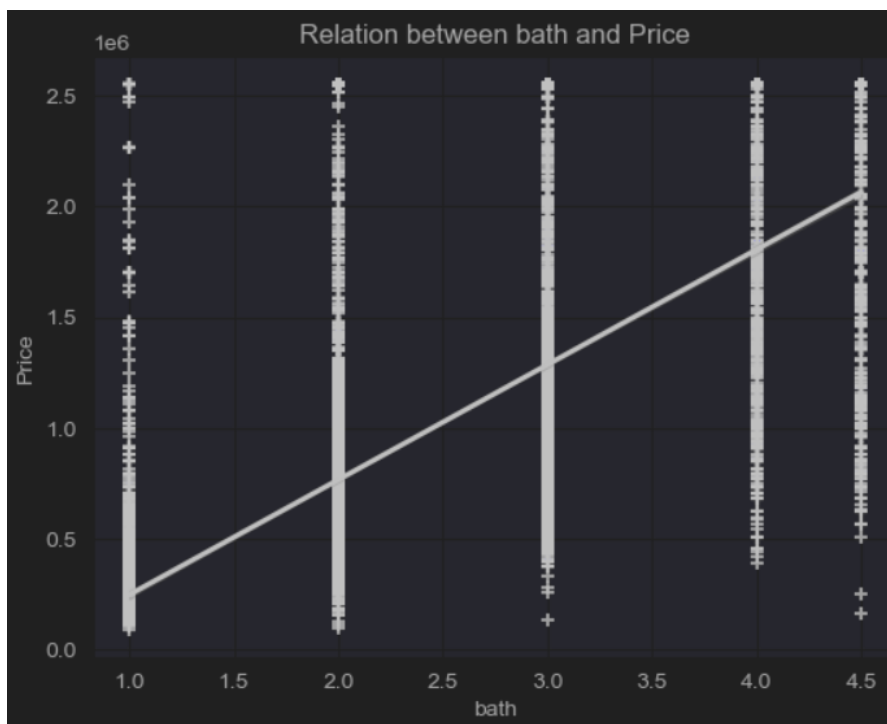


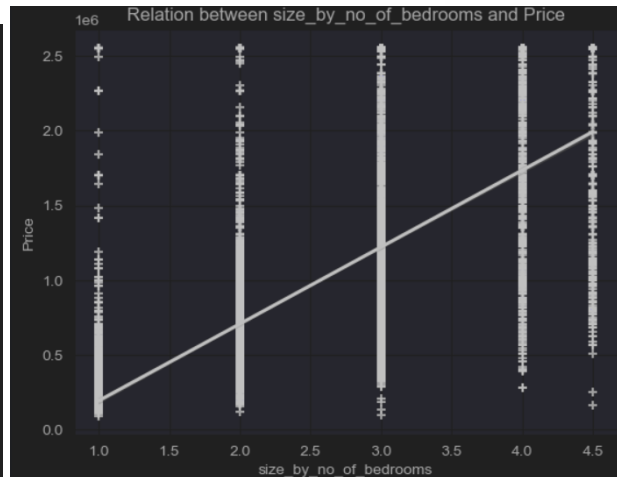
PREDICTIVE ANALYSIS

Once the dataset was properly cleaned by addressing missing values, outliers and standardisation, I started the predictive analysis phase. A correlation heatmap was created using the numerical values to understand the correlation between variables. The heatmap portrayed that the variables are positively correlated with each other. My main object was to check the correlation for the variables with the “total_price” variable. The variable “balcony” was the least correlated with price.



I then proceeded to do regression plots for the highly correlated variables with “total_price”.





LINEAR REGRESSION

I proceeded to create a linear regression model to predict the target variable “total_price” using features such as 'size_by_no_of_bedrooms', 'total_sqft' and 'bath'.

```

#linear regression
X = new_df[['size_by_no_of_bedrooms', 'total_sqft', 'bath']]
y = new_df['total_price']
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
print(f'R²: {r2}')
print(f'Mean Squared Error: {mse}')
print(f'Root Mean Squared Error: {rmse}')
print('Coefficients: ', model.coef_)
print('Intercept: ', model.intercept_)
✓ [422] 31ms

R²: 0.6584181800770788
Mean Squared Error: 155188076725.59674
Root Mean Squared Error: 393939.1789675111
Coefficients: [ 62566.81863308  665.53585117 218402.52370119]
Intercept: -618228.780933033
  
```

To further improve the results, I decided to create dummy variables of the categorical variables in the columns of the dataset. Dummies were created for 'location', 'BER', 'Renovation needed',

and 'property_scope'. I used all these features to train and predict my target variable “price-per-sqft-\$”.

```
X = df3[['size_by_no_of_bedrooms', 'total_sqft', 'bath', 'balcony', 'location_DCC', 'location_Dun
Laoghaire', 'location_Fingal', 'location_Other', 'location_South Dublin', 'BER_A', 'BER_B', 'BER_C',
'BER_D', 'BER_E', 'BER_F', 'BER_G', 'Renovation needed_Maybe', 'Renovation needed_No', 'Renovation
needed_Yes', 'property_scope_Constructed Space', 'property_scope_Extended Coverage', 'property_scope_Land
Parcel', 'property_scope_Usable Interior']]
y = df3['price-per-sqft-$']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
Training Mean Squared Error: 137358367908.62
Training Root Mean Squared Error: 370618.9
Testing Mean Squared Error: 136474230742.42
Testing Root Mean Squared Error: 369424.19
R²: 0.6996089062176831
formatted Coefficients: ['-12553.35', '784.27', '170841.57', '-12753.77', '-12368.29', '-44017.93', '16588.31',
'33644.05', '6153.86', '5357.44', '-4545.48', '-1596.54', '3739.39', '-4020.21', '-6023.24', '7088.63',
'-6142.01', '5357.44', '784.58', '-131945.05', '-147618.85', '308037.13', '-28473.23']
Normal Coefficients: [ -12553.35    784.27  170841.57  -12753.77  -12368.29  -44017.93
  16588.31   33644.05   6153.86   5357.44   -4545.48   -1596.54
  3739.39   -4020.21   -6023.24   7088.63   -6142.01   5357.44
    784.58 -131945.05 -147618.85  308037.13  -28473.23]
Intercept: -366727.28
```

The testing Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) along with training MSE and RMSE were calculated to compare Bias and Variance. Having more features enabled the Linear regression model to improve its predictions.

Standardisation was applied to the features and trained the model again to check for improvements in the model.

```
1 #applying standardization to check if there is any difference
2 scaler = StandardScaler()
3 X_train_scaled = scaler.fit_transform(X_train)
4 X_test_scaled = scaler.transform(X_test)
```

```
R²: 0.6994749410925141
Training Mean Squared Error: 137374302304.63
Training Root Mean Squared Error: 370640.39
Testing Mean Squared Error: 136535094022.93
Testing Root Mean Squared Error: 369506.55
Coefficients: [-7.52023000e+03  4.14979960e+05  1.56241880e+05 -1.11745800e+04
 4.13117846e+16  3.73780033e+16  4.75250368e+16  1.82726224e+16
 3.89756938e+16  3.48714400e+17 -2.71753642e+16 -2.72039697e+16
 1.89693703e+17  1.93115507e+17  1.90014606e+17  1.89747259e+17
-1.43857122e+17 -4.85195477e+17 -4.66133574e+17 -3.52597258e+18
-4.32112267e+18 -3.27072177e+18 -7.20238668e+17]
Intercept: 1064097.8554564943
```


RANDOM FOREST REGRESSOR

A Random Forest Regressor (RFR) was created for regression since Linear regression models were unable to capture the non-linear relationships between variables. For creating this regressor all the Boolean values from the dummy variables were converted to 0 and 1 and was incorporated in a new dataframe.

```
# Initialize the Random Forest Regressor
rf_regressor = RandomForestRegressor(random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
rmse = np.sqrt(mse)
print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")
print(f"RMSE: {rmse}")
✓ [463] 5s 236ms

Mean Squared Error: 123694347221.4321
R^2 Score: 0.7277384891315847
RMSE: 351702.07167634385
```

The RFR model gave better results compared to the linear regression models.

HYPER PARAMETER TUNING USING GRIDSEARCHCV

To improve the results of the model, hyperparameter tuning was applied using GridsearchCV which improved the performance of the model further by increasing the R square value to 0.75 and reducing the RMSE.

```
Mean Squared Error: 111945945750.22166
R^2 Score: 0.7535976945576371
RMSE: 334583.2418849182
```

CLASSIFICATIONS

LOGISTIC REGRESSION

Logistic regression is a statistical predictive model often used for classifications (IBM, 2024).

I decided to use logistic regression model to train and predict values for the target variable “buying or not buying”. I used all the remaining columns as features other than ID, availability and total_price.

```
X = df3[['size_by_no_of_bedrooms', 'total_sqft', 'bath', 'balcony', 'price-per-sqft-$', 'location_DCC',
'location_Dun Laoghaire', 'location_Fingal', 'location_Other', 'location_South Dublin', 'BER_A', 'BER_B',
'BER_C', 'BER_D', 'BER_E', 'BER_F', 'BER_G', 'Renovation needed_Maybe', 'Renovation needed_No',
'Renovation needed_Yes', 'property_scope_Constructed Space', 'property_scope_Extended Coverage',
'property_scope_Land Parcel', 'property_scope_Usable Interior']]
y = df3['buying or not buying']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
model = LogisticRegression(max_iter=2000)#class_weight='balanced'
model.fit(X_train, y_train)
# Make predictions on the testing data
y_pred = model.predict(X_test)
# Cross-validation
cv_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')
print(f'Cross-Validation Accuracy Scores: {cv_scores}')
print(f'Mean Cross-Validation Accuracy: {cv_scores.mean()}')
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, zero_division=0)
print('Coefficients:', model.coef_)
print('Intercept:', model.intercept_)
print(f'Accuracy: {accuracy}')
print('Confusion Matrix:')
print(conf_matrix)
print('Classification Report:')
print(class_report)
✓ [342] 8s 368ms

Cross-Validation Accuracy Scores: [0.75692163 0.74659784 0.74425153 0.75551384 0.74847489]
Mean Cross-Validation Accuracy: 0.7503519474425151
Coefficients: [[ 8.69417123e-02 -1.57287994e-04  2.59903218e-02 -4.63388014e-02
-2.23763124e-05 -3.10976260e-01  5.90574985e-01 -3.65332038e-01
-3.36888431e-01  6.48810948e-02  3.06277182e-01  2.09276395e-01
-4.46034752e-01 -9.34396846e-02 -1.49270407e-01 -1.30607731e-01
-5.39416526e-02 -2.36758358e-01  3.06277182e-01 -4.27259475e-01
 3.80083630e-06  6.07290471e-02 -4.65490778e-02 -3.71924420e-01]]
Intercept: [-0.39116115]
Accuracy: 0.753003003003003
Confusion Matrix:
[[1799  16]
 [ 642 207]]
```

Cross validation was also implemented to check accuracy through multiple iterations.

```

Confusion Matrix:
[[1799  16]
 [ 642 207]]
Classification Report:

```

	precision	recall	f1-score	support
No	0.74	0.99	0.85	1815
Yes	0.93	0.24	0.39	849
accuracy			0.75	2664
macro avg	0.83	0.62	0.62	2664
weighted avg	0.80	0.75	0.70	2664

HANDLING IMBALANCE IN THE DATA

The results provided by the model through confusion matrix and classification matrix highlighted that, the model was struggling to predict values for the class “Yes” for the target variable. This was pointing to imbalance in the data.

```

X_train.shape, y_train.shape
✓ [334] < 10 ms

((10655, 24), (10655,))

y_train.value_counts()
✓ [335] 16ms

```

buying or not buying	count
No	7241
Yes	3414

By checking the value counts for the classes in the training data, it was certain that there is a huge imbalance in the data available for classes “Yes” and “No” and this was negatively affecting the effectiveness of the model. To overcome this challenge, I decided to use Synthetic Minority Over-sampling Technique (SMOTE). SMOTE is an oversampling technique where new samples are synthesised for the minority class to balance it out with the majority class (imblearn, 2021). SMOTE is better than normal oversampling since the data for the minority class is not duplicated to balance the classes. Oversampling is considered instead of under sampling to prevent data loss.

Oversampling to balance the classes using SMOTE

```
# Apply SMOTE to the training data. This is a form of oversampling to balance the classes
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
# Split the resampled data into training and testing sets
X_train_resampled, X_test_resampled, y_train_resampled, y_test_resampled = train_test_split(
    X_train_resampled, y_train_resampled, test_size=0.3, random_state=42, stratify=y_train_resampled)
✓ [343] 172ms
```

```
y_train_resampled.value_counts()
```

```
✓ [336] 22ms
```

Length: 2, dtype: int64	
buying or not buying	count
No	7241
Yes	7241

After using SMOTE to balance the classes, logistic regression model was again created using the resampled data.

```
Cross-Validation Accuracy Scores: [0.80522682 0.79585799 0.80661075 0.7849038 0.7947706 ]
Mean Cross-Validation Accuracy: 0.7974739926183895
Coefficients: [[ 9.18090414e-02 -1.48004019e-04 3.86611275e-02 -3.24188094e-02
 6.98051631e-05 3.03399085e+00 3.91561241e+00 2.95094512e+00
 2.89880542e+00 3.36028119e+00 2.77980219e+00 2.64414068e+00
 2.08578059e+00 3.19083432e+00 3.05241021e+00 3.11366072e+00
 3.17572012e+00 2.18387415e+00 2.77980219e+00 1.21961521e+00
 2.49136768e+00 2.62359023e+00 2.52941556e+00 1.34381564e+00]]
Intercept: [-11.23861292]
Accuracy: 0.7972382048331416
```

Confusion Matrix:

```
[[2065 108]
```

```
[ 773 1399]]
```

Classification Report:

	precision	recall	f1-score	support
No	0.73	0.95	0.82	2173
Yes	0.93	0.64	0.76	2172
accuracy			0.80	4345
macro avg	0.83	0.80	0.79	4345
weighted avg	0.83	0.80	0.79	4345

This time, the model performed better for predicting “Yes” and “No” for the target variable. This can be observed from the confusion matrix and classification report. The model accuracy increased to 80 and the recall and f1-score for the classes No and Yes were 0.95,0.64 and 0.82,0.76 respectively. This outlined significant improvement in the logistic regression model’s performance for predicting Yes or No for the target variable “buying or not buying”.

DECISION TREE

I decided to create multiple models for the classification so that I can compare them and choose the best one. For this purpose, the next model I implemented was a decision tree.

```
# Create and train the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_resampled, y_train_resampled)

# Make predictions on the testing data
y_pred_dt = dt_model.predict(X_test_resampled)

# Evaluate the Decision Tree model
accuracy_dt = accuracy_score(y_test_resampled, y_pred_dt)
conf_matrix_dt = confusion_matrix(y_test_resampled, y_pred_dt)
class_report_dt = classification_report(y_test_resampled, y_pred_dt)

print(f'Decision Tree Accuracy: {accuracy_dt}')
print('Decision Tree Confusion Matrix:')
print(conf_matrix_dt)
print('Decision Tree Classification Report:')
print(class_report_dt)
```

✓ [348] 327ms

```
Decision Tree Accuracy: 0.72819332566168
Decision Tree Confusion Matrix:
[[1552  621]
 [ 560 1612]]
Decision Tree Classification Report:
```

	precision	recall	f1-score	support
No	0.73	0.71	0.72	2173
Yes	0.72	0.74	0.73	2172
accuracy			0.73	4345
macro avg	0.73	0.73	0.73	4345
weighted avg	0.73	0.73	0.73	4345

RANDOM FOREST CLASSIFIER

Similarly, I decide to create another model, the random forest to evaluate its performance for the same task.

```

1 # Create and train the Random Forest model
2 rf_model = RandomForestClassifier(random_state=42, class_weight='balanced')
3 rf_model.fit(X_train_resampled, y_train_resampled)
4 # Make predictions on the testing data
5 y_pred_rf = rf_model.predict(X_test_resampled)
6 # Evaluate the Random Forest model
7 accuracy_rf = accuracy_score(y_test_resampled, y_pred_rf)
8 conf_matrix_rf = confusion_matrix(y_test_resampled, y_pred_rf)
9 class_report_rf = classification_report(y_test_resampled, y_pred_rf)
10 print(f'Random Forest Accuracy: {accuracy_rf}')
11 print('Random Forest Confusion Matrix:')
12 print(conf_matrix_rf)
13 print('Random Forest Classification Report:')
14 print(class_report_rf)

```

✓ [351] 2s 438ms

Random Forest Accuracy: 0.7758342922899885

Random Forest Confusion Matrix:

```
[[1900  273]
 [ 701 1471]]
```

Random Forest Classification Report:

	precision	recall	f1-score	support
No	0.73	0.87	0.80	2173
Yes	0.84	0.68	0.75	2172
accuracy			0.78	4345
macro avg	0.79	0.78	0.77	4345
weighted avg	0.79	0.78	0.77	4345

HYPER PARAMETER TUNING FOR THE RANDOM FOREST MODELS

Hyper Parameter tuning is utilised to improve the performance of the model (Koehrsen, 2018).

RandomizedSearchCV

```

#using randomized search
random_param_grid = {
    'n_estimators': [500, 1000, 1500],
    'criterion': ['gini', 'entropy'],
    'max_depth': [10, 20, 30],
    'min_samples_split': [5,10,15],
    'min_samples_leaf': [1, 2, 4]#,
    #'class_weight': ['balanced', 'balanced_subsample']
}
random_grid_search_rf = RandomizedSearchCV(rf, random_param_grid, cv=5, scoring='accuracy',n_jobs=-1,
    random_state=42)
random_grid_search_rf.fit(X_train_resampled, y_train_resampled)

print('Best Score: ',random_grid_search_rf.best_score_)
print('Best Parameters: ',random_grid_search_rf.best_params_)
✓ [363] 1m 13s

Best Score: 0.8128632786767203
Best Parameters: {'n_estimators': 1500, 'min_samples_split': 10, 'min_samples_leaf': 4, 'max_depth': 30, 'criterion': 'entropy'}
    
```

```

Best Random Forest Accuracy: 0.8126582278481013
Best Random Forest Confusion Matrix:
[[2158  15]
 [ 799 1373]]
Best Random Forest Classification Report:

```

	precision	recall	f1-score	support
No	0.73	0.99	0.84	2173
Yes	0.99	0.63	0.77	2172
accuracy			0.81	4345
macro avg	0.86	0.81	0.81	4345
weighted avg	0.86	0.81	0.81	4345

GridSearchCV

Hyper Parameter Tuning for Random Forest

```

1 #Hyper Parameter Tuning for Random Forest to check for improvements in the result
2 rf = RandomForestClassifier(random_state=42, class_weight='balanced')
3 param_grid = {
4     'n_estimators': [500, 1000, 1500],
5     'criterion': ['gini', 'entropy'],
6     'max_depth': [10, 20, 30],
7     'min_samples_split': [5, 10, 15],
8     'min_samples_leaf': [1, 2, 4],
9     #'class_weight': ['balanced', 'balanced_subsample']
10 }
11 grid_search_rf = GridSearchCV(rf, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
12 grid_search_rf.fit(X_train_resampled, y_train_resampled)
13
14 print('Best Score: ', grid_search_rf.best_score_)
15 print('Best Parameters: ', grid_search_rf.best_params_)
✓ [359] 8m 47s

Best Score: 0.8148361517930034
Best Parameters: {'criterion': 'entropy', 'max_depth': 30, 'min_samples_leaf': 2,
                  'min_samples_split': 15, 'n_estimators': 1500}

```

```

best_rf_model = grid_search_rf.best_estimator_
# Make predictions with the best model
y_pred_best_rf = best_rf_model.predict(X_test_resampled)
# Evaluate the best Random Forest model
accuracy_best_rf = accuracy_score(y_test_resampled, y_pred_best_rf)
conf_matrix_best_rf = confusion_matrix(y_test_resampled, y_pred_best_rf)
class_report_best_rf = classification_report(y_test_resampled, y_pred_best_rf)

print(f'Best Random Forest Accuracy: {accuracy_best_rf}')
print('Best Random Forest Confusion Matrix:')
print(conf_matrix_best_rf)
print('Best Random Forest Classification Report:')
print(class_report_best_rf)
✓ [361] 1s 33ms

```

```

Best Random Forest Accuracy: 0.8126582278481013
Best Random Forest Confusion Matrix:
[[2158  15]
 [ 799 1373]]
Best Random Forest Classification Report:

```

	precision	recall	f1-score	support
No	0.73	0.99	0.84	2173
Yes	0.99	0.63	0.77	2172
accuracy			0.81	4345
macro avg	0.86	0.81	0.81	4345
weighted avg	0.86	0.81	0.81	4345

RESULTS, EVALUATION AND DISCUSSION

I used multiple predictive analytics models on the Ireland Housing dataset. For regression I used Linear regression model and random forest regressor. Similarly, for classification, logistic regression, decision tree and random forest classifier models were used. Each of these models were utilised due to their varying ability to capture underlying patterns and relationships within the data.

LOGISTIC REGRESSION

The logistic regression model achieved an accuracy of 79.7%, making it a reliable and interpretable method. It excels in precision, particularly for the "Yes" class (buyers), with a value of 0.93, indicating its ability to minimize false positives. However, its recall for class "Yes" (0.64) is lower, suggesting that the model misses a significant portion of "Yes" cases. Overall, the model has a good balance between precision and recall. Its linear nature makes it suitable for problems where relationships between features and the target variable are relatively simple. Additionally, the model's coefficients and cross-validation results indicate a stable performance across different subsets of the dataset.

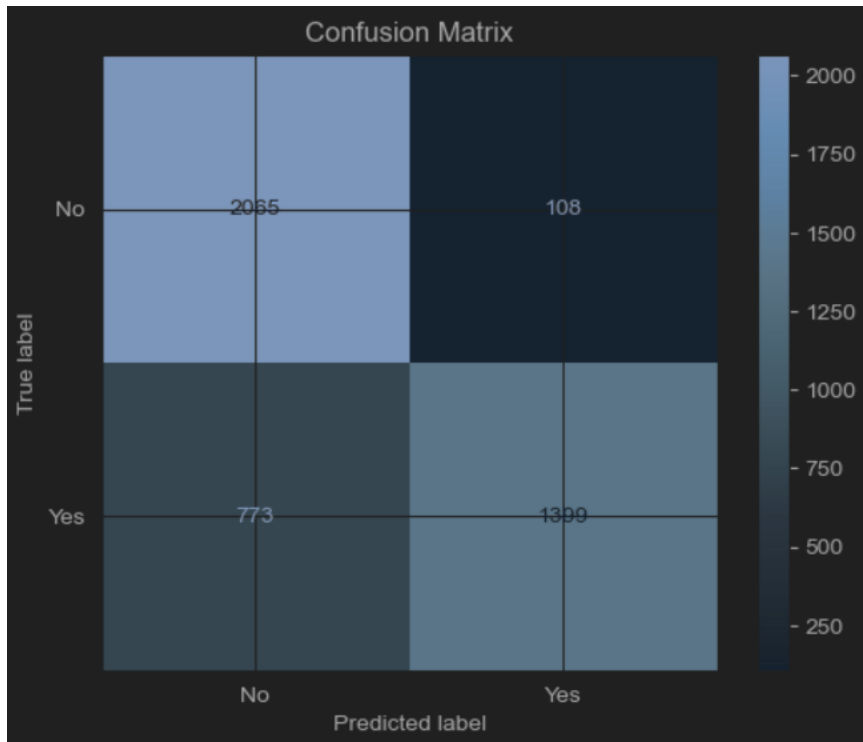
```
Cross-Validation Accuracy Scores: [0.80522682 0.79585799 0.80661075 0.7849038 0.7947706 ]
Mean Cross-Validation Accuracy: 0.7974739926183895
Coefficients: [[ 9.18090414e-02 -1.48004019e-04 3.86611275e-02 -3.24188094e-02
 6.98051631e-05 3.03399085e+00 3.91561241e+00 2.95094512e+00
 2.89880542e+00 3.36028119e+00 2.77980219e+00 2.64414068e+00
 2.08578059e+00 3.19083432e+00 3.05241021e+00 3.11366072e+00
 3.17572012e+00 2.18387415e+00 2.77980219e+00 1.21961521e+00
 2.49136768e+00 2.62359023e+00 2.52941556e+00 1.34381564e+00]]
Intercept: [-11.23861292]
Accuracy: 0.7972382048331416
```

```
Confusion Matrix:
[[2065 108]
 [ 773 1399]]
Classification Report:
              precision    recall  f1-score   support

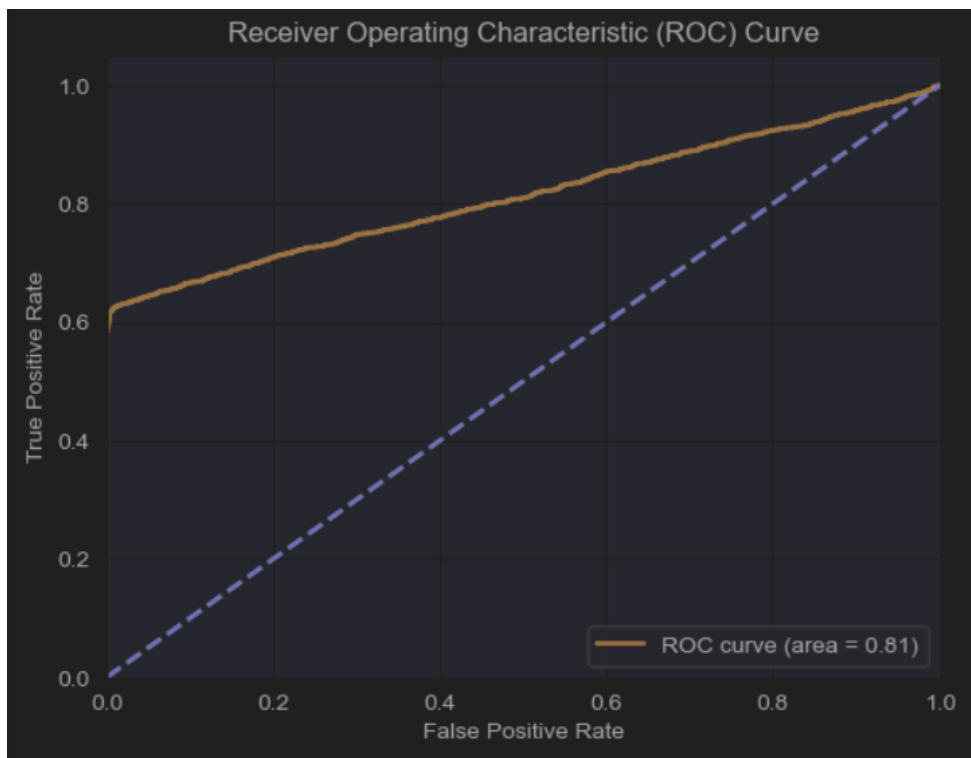
     No         0.73         0.95         0.82         2173
     Yes         0.93         0.64         0.76         2172

 accuracy              0.80         4345
 macro avg           0.83         0.80         0.79         4345
 weighted avg           0.83         0.80         0.79         4345
```

A confusion matrix was also plotted signifying the performance.



A **Receiver Operating Characteristic (ROC) Curve** was plotted to visualise the performance of the model.



The model has an Area Under the Curve (AUC) value of 0.81, which signifies the model has good discriminatory power and can correctly distinguish between the two classes for 81% of the cases.

DECISION TREE

The decision tree model, on the other hand, achieved a slightly lower accuracy of 72.8%. However, it outperformed logistic regression in recall for the "Yes" class (0.74). This model's strength lies in its ability to capture non-linear relationships in the data and provide interpretable decision rules. Despite its advantages, the decision tree's predictive power is relatively moderate, and its performance metrics suggest that it may be overfitting the training data to some extent. Nevertheless, its simplicity and interpretability make it a useful tool in scenarios where the goal is to extract clear decision rules.

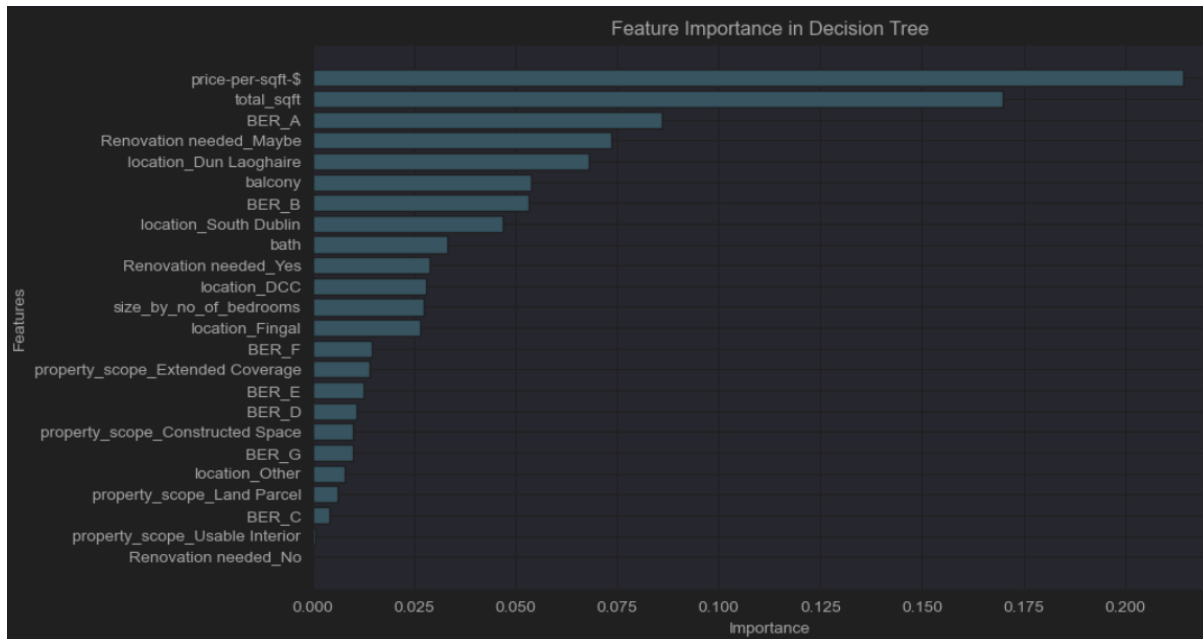
```

Decision Tree Accuracy: 0.72819332566168
Decision Tree Confusion Matrix:
[[1552  621]
 [ 560 1612]]
Decision Tree Classification Report:

```

	precision	recall	f1-score	support
No	0.73	0.71	0.72	2173
Yes	0.72	0.74	0.73	2172
accuracy			0.73	4345
macro avg	0.73	0.73	0.73	4345
weighted avg	0.73	0.73	0.73	4345

A feature importance graph was also plotted for the decision tree to identify the importance given to the features by the model. For the feature like “Renovation needed_no”, the model didn’t imply much importance suggesting that the underlying relationships were not fully captured and improvements were needed.



RANDOM FOREST

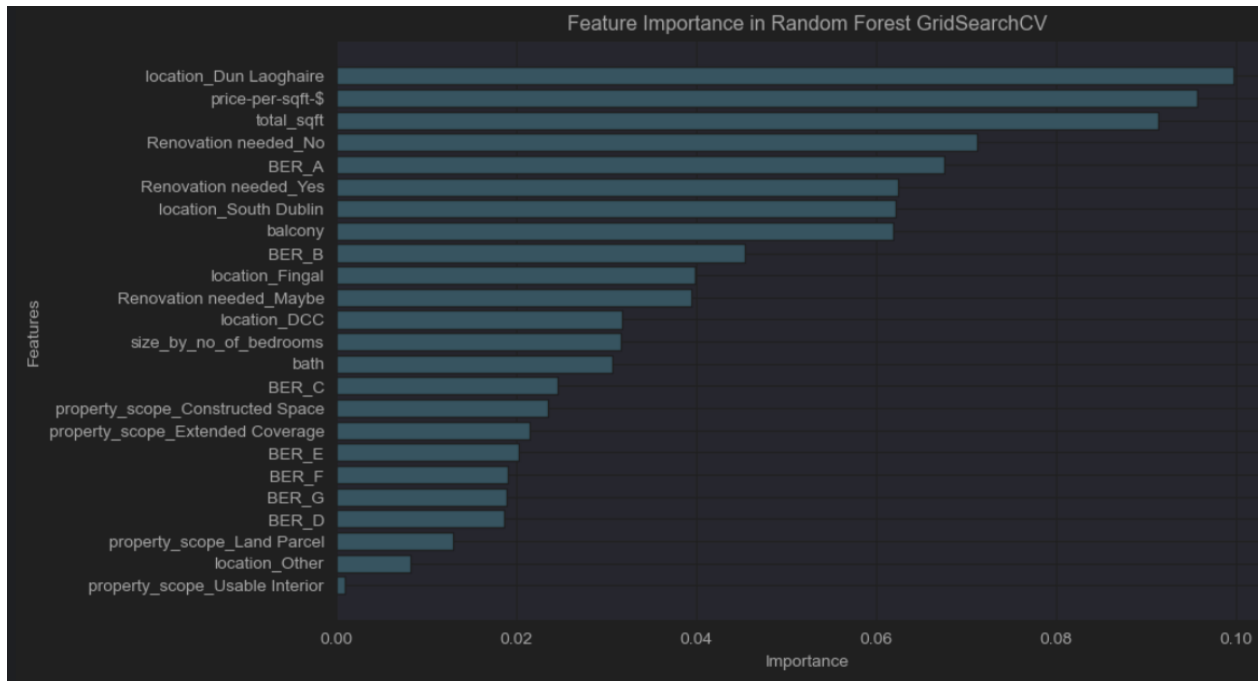
The random forest model after hyper parameter tuning using Grid Search demonstrated the highest accuracy, with a score of 81.3%, and the best overall performance among the three methods. It achieved near-perfect precision for the "Yes" class (0.99), meaning it is highly effective at minimizing false positives. Additionally, its recall for the "No" class was the highest among the models at 0.99, making it an excellent choice for scenarios prioritizing the accurate identification of the class. However, its recall for the "Yes" class (0.63) was slightly lower than that of the decision tree. The random forest's ability to capture complex, non-linear relationships in the data and its robustness against overfitting make it the most effective predictive tool for this dataset. However, it requires significant computational resources and is less interpretable than logistic regression.

```

Best Random Forest Accuracy: 0.8126582278481013
Best Random Forest Confusion Matrix:
[[2158  15]
 [ 799 1373]]
Best Random Forest Classification Report:

```

	precision	recall	f1-score	support
No	0.73	0.99	0.84	2173
Yes	0.99	0.63	0.77	2172
accuracy			0.81	4345
macro avg	0.86	0.81	0.81	4345
weighted avg	0.86	0.81	0.81	4345



Additionally, the feature importance graph created by the Random Forest model appears to be the most logical for the context signifying that it was able to capture the nonlinear characteristics within the data. The graph highlights that features such as “location_Dun Laoghaire”, “price_per_sqft_\$” had more importance while predicting than features “property_scope_Usable Interior”.

LINEAR REGRESSION

I used Linear regression model initially to predict the value for the target variable “total_price” with features like size_by_no_of_bedrooms, total_sqft, and bath as per the correlation analysis. The model received an R squared value of 0.658 signifying that approximately 65.8% of the variance in the dependent variable is explained by the independent variables.

```

R²: 0.6584181800770788
Mean Squared Error: 155188076725.59674
Root Mean Squared Error: 393939.1789675111
Coefficients: [ 62566.81863308    665.53585117 218402.52370119]
Intercept: -618228.780933033
  
```

While this suggests a moderate fit, it also highlights that 34.2% of the variance remains unexplained. This unexplained variance indicates that other factors influencing property prices may not have been included in the model. The coefficients of the model provide important

insights into how each independent variable influences the target variable. For every additional bedroom, the model predicts that the total price will increase by approximately 62,566.82. Similarly, for each additional square foot of total area, the price increases by 665.54. The most significant factor is the number of bathrooms, where each additional bathroom contributes 218,402.52 to the price. The Root Mean Squared Error (RMSE), calculated as 393,939.18 implies that, on average, the model's predictions deviate from actual prices by approximately 393,939.18. This suggests for improvements in model prediction.

For the second linear regression model I added more features by creating dummies of categorical columns. I created model with both standardised features and non-standardised features. Even though a better result was got with improvements in the R squared increasing to 0.699, and reduction in RMSE, still the model seem lacking.

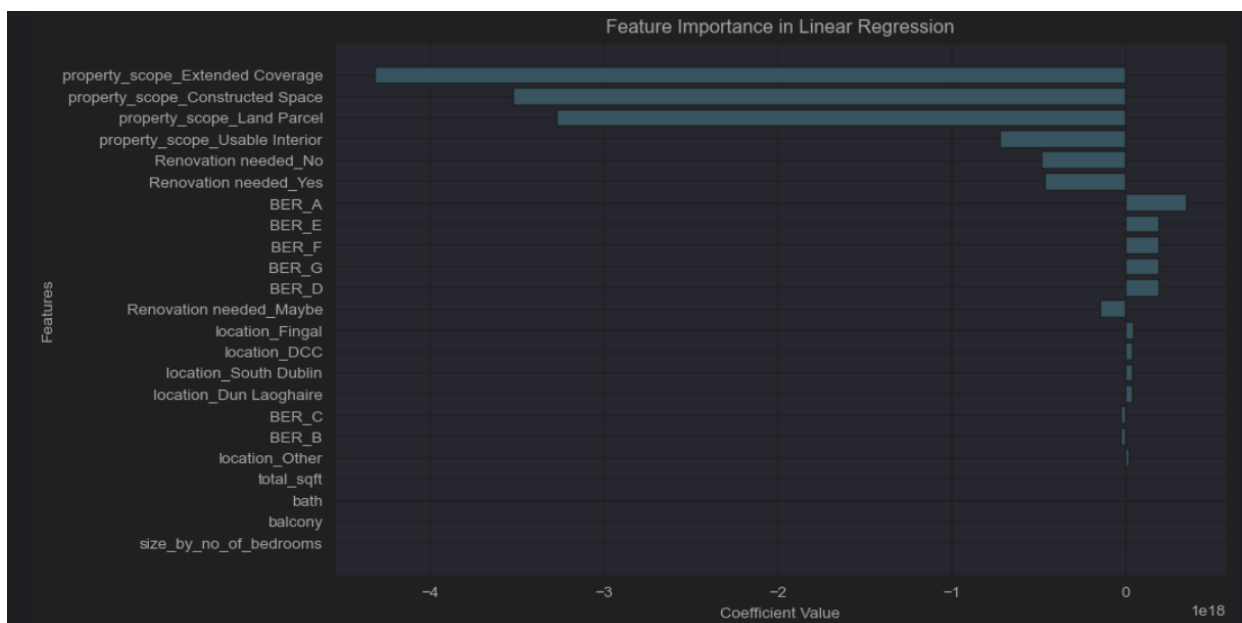
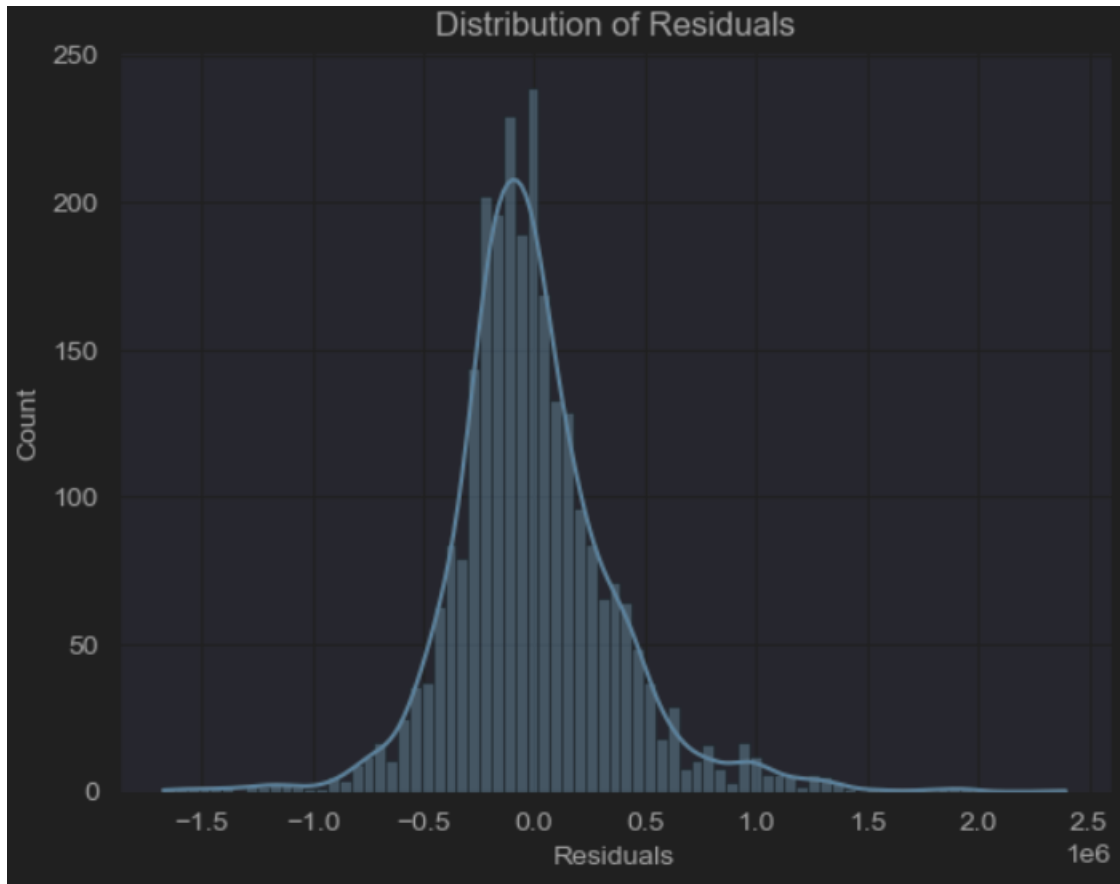
```

X = df3[['size_by_no_of_bedrooms', 'total_sqft', 'bath', 'balcony', 'location_DCC', 'location_Dun Laoghaire',
'location_Fingal', 'location_Other', 'location_South Dublin', 'BER_A', 'BER_B', 'BER_C', 'BER_D', 'BER_E', 'BER_F',
'BER_G', 'Renovation needed_Maybe', 'Renovation needed_No', 'Renovation needed_Yes', 'property_scope_Constructed
Space', 'property_scope_Extended Coverage', 'property_scope_Land Parcel', 'property_scope_Usable Interior']]
y = df3['total_price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
  
```

```

Training Mean Squared Error: 137358367908.62
Training Root Mean Squared Error: 370618.9
Testing Mean Squared Error: 136474230742.42
Testing Root Mean Squared Error: 369424.19
R²: 0.6996089062176831
formatted Coefficients: ['-12553.35', '784.27', '170841.57', '-12753.77', '-12368.29', '-44017.93', '16588.31',
'33644.05', '6153.86', '5357.44', '-4545.48', '-1596.54', '3739.39', '-4020.21', '-6023.24', '7088.63',
'-6142.01', '5357.44', '784.58', '-131945.05', '-147618.85', '308037.13', '-28473.23']
Normal Coefficients: [ -12553.35      784.27 170841.57 -12753.77 -12368.29 -44017.93
 16588.31  33644.05   6153.86  5357.44  -4545.48  -1596.54
 3739.39  -4020.21  -6023.24  7088.63  -6142.01   5357.44
 784.58 -131945.05 -147618.85 308037.13 -28473.23]
Intercept: -366727.28
  
```

The plot portraying the distribution of residuals was created to understand the efficiency of the regression model.



The feature importance graph portrays that Linear regression model lacks in identifying the non-linear relationship between variables thereby I decided to use another model, the Random Forest regressor to do regression rather than hyper parameter tuning with Lasso or Ridge.

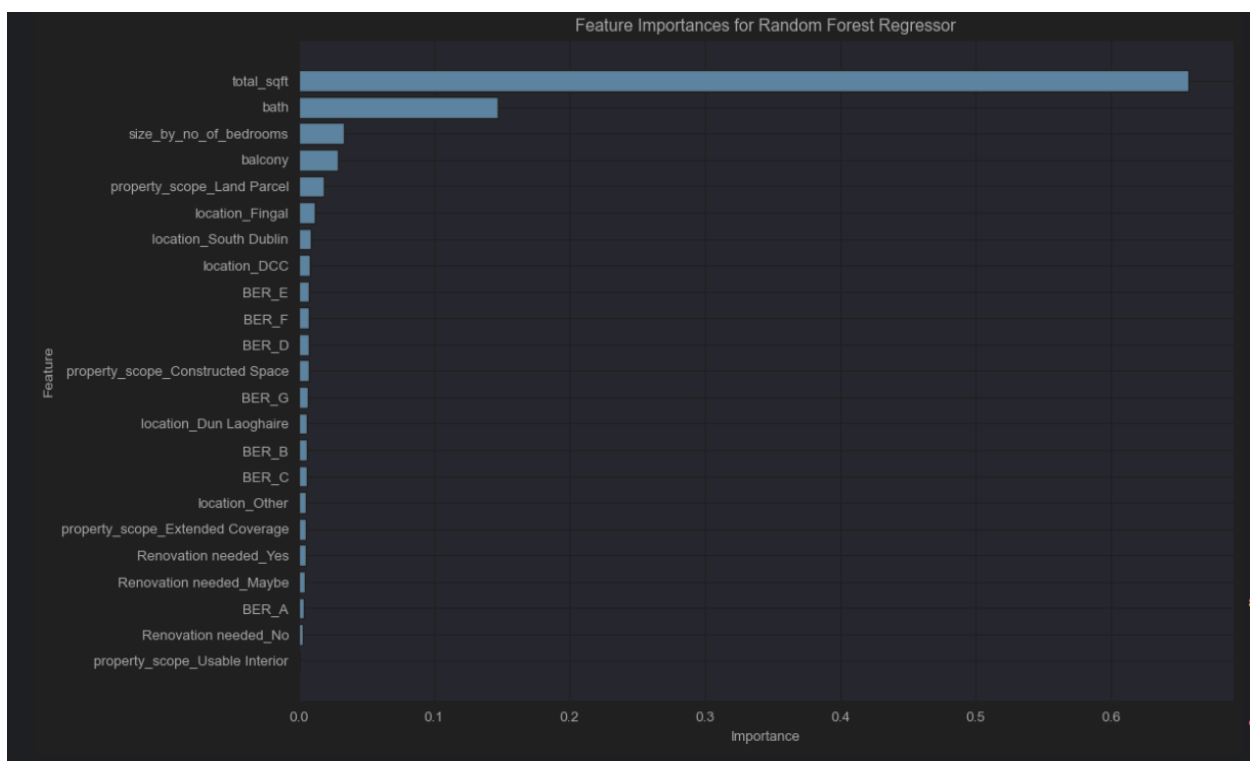
RANDOM FOREST REGRESSOR

The random forest regressor along with hyperparameter tuning using **GridSearchCV** gave the best result for the regression analysis.

```

Mean Squared Error: 111945945750.22166
R^2 Score: 0.7535976945576371
RMSE: 334583.2418849182
    
```

The R Squared value increased to 0.75 which implies that 75% of the variance for the dependent value can be predicted by the model. The RMSE also improved.



The feature importance graph for the random forest regressor also looked more apt to the context, indicating that it was able to identify non linear relationships in the dataset better than linear regression.

The inclusion of features such as location, BER rating, and renovation status played a crucial role in improving predictive power. In logistic regression, "price-per-sqft-\$" emerged as a significant factor, as reflected in its high coefficient values. Similarly, random forest feature importance analysis highlights the influence of these variables on the predictions. Proper handling of outliers, such as normalizing "total_sqft," ensured that extreme values did not bias the models. Creating "total_price" was very influential for regression. Moreover, the

application of SMOTE to balance the dataset improved the performance of the decision tree and random forest, particularly in addressing class imbalances.

CONCLUSION

In conclusion Random Forest models after hyper parameter tuning gave the best results for both classifications and regression. The most significant factors influencing a homebuyer according to the models are total_square_feet , no_of_bedrooms, Bathrooms, and location DCC along with BER rating.

USE OF AI TOOLS

AI tools such as Microsoft Copilot was utilised during this assignment especially for error handling and providing suggestions. They also helped in interpreting the results of models which sometimes was difficult to comprehend.

Here is a link to a session:

https://docs.google.com/document/d/1eIFHDNbLNKxdJZL8E_wnHCymqlqf7oc9ngcjZOPlLlk/edit?usp=sharing

Link to Jupyter notebook:

https://drive.google.com/drive/folders/1k_e_ccrcPm88PuMrGiR8rv9jmtDQsEBF?usp=drive_link

REFERENCES

Reference list

codebasics (2020). *Handling imbalanced dataset in machine learning | Deep Learning Tutorial 21 (Tensorflow2.0 & Python)*. YouTube. Available at: <https://www.youtube.com/watch?v=JnlM4yLFNuo> [Accessed 25 Nov. 2024].

Galarnyk, M. (2022). *Understanding Boxplots: How to Read and Interpret a Boxplot | Built in*. [online] builtin.com. Available at: <https://builtin.com/data-science/boxplot> [Accessed 15 Nov. 2024].

IBM (2024). *What Is Logistic Regression?* [online] IBM. Available at: <https://www.ibm.com/topics/logistic-regression> [Accessed 20 Nov. 2024].

imblearn (2021). *SMOTE — Version 0.9.0*. [online] imbalanced-learn.org. Available at: [https://imbalanced-](https://imbalanced-learn.org/)

learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html [Accessed 28 Nov. 2024].

Koehrsen, W. (2018). *Hyperparameter Tuning the Random Forest in Python*. [online] Medium. Available at: <https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74> [Accessed 29 Nov. 2024].

Krish Naik (2019a). *Tutorial 28- Ridge and Lasso Regression using Python and Sklearn*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=0yI0-r3Ly40> [Accessed 27 Nov. 2024].

Krish Naik (2019b). *Tutorial 43-Random Forest Classifier and Regressor*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=nxFG5xdpDto> [Accessed 22 Nov. 2024].

Krish Naik (2021). *Difference Between fit(), transform(), fit_transform() and predict() methods in Scikit-Learn*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=BotYLBQfd5M> [Accessed 25 Nov. 2024].

Learn with Ankith (2023). *Data Cleaning/Data Preprocessing Before Building a Model - A Comprehensive Guide*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=GP-2634exqA> [Accessed 15 Nov. 2024].

Naik, K. (2019c). *Tutorial 46-Handling imbalanced Dataset using python- Part 2*. YouTube. Available at: <https://www.youtube.com/watch?v=OJedgzdipC0> [Accessed 27 Nov. 2024].

Naik, K. (2020). *Machine Learning-Bias And Variance In Depth Intuition| Overfitting Underfitting*. YouTube. Available at: <https://www.youtube.com/watch?v=BqzgUnrNhFM> [Accessed 28 Nov. 2024].

Ryan & Matt Data Science (2023). *Hands-On Hyperparameter Tuning with Scikit-Learn: Tips and Tricks*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=LrCylIe0RJM> [Accessed 29 Nov. 2024].

Saturncloud.io. (2023). *How to Count NaN and Null Values in a Pandas DataFrame | Saturn Cloud Blog*. [online] Available at: <https://saturncloud.io/blog/how-to-count-nan-and-null-values-in-a-pandas-dataframe/> [Accessed 18 Nov. 2024].

Scikit-learn (2019). *5.3. Preprocessing data — scikit-learn 0.21.3 documentation*. [online] Scikit-learn.org. Available at: <https://scikit-learn.org/stable/modules/preprocessing.html> [Accessed 25 Nov. 2024].

Sustainable Energy Authority of Ireland. (2019). *Understand a BER*. [online] Available at: <https://www.seai.ie/ber/understand-a-ber-rating> [Accessed 18 Nov. 2024].