Predictive Analysis on Ireland House Price Dataset

Module Code: IS6052

Predictive Analytics

Sai Nikhil Vaddadi

124107966

02/12/2024

Table of Contents

1. Introduction

This individual project delves into predictive analytics to forecast whether properties in Ireland are likely to be purchased. The dataset includes key features such as **property size, number of bathrooms and balconies, price per square foot, location,** and **property scope**. Tackling this challenge involved addressing the complexities of real-world data, such as missing values, outliers, and class imbalances in the target variable (buying or not buying) (McKinney, 2017).

The project began with exploratory data analysis (**EDA**), identifying issues like missing values and inconsistencies in data. Steps were taken to clean the dataset, including imputing missing values, capping outliers, and standardizing numerical features. Feature engineering further enriched the dataset by creating derived attributes such as **bath_per_sqft** and **balcony_to_price_ratio**, enabling deeper insights into property characteristics.

Multiple machine learning algorithms were applied, including **Logistic Regression, Decision Trees, Random Forest, Gradient Boosting, Naive Bayes,** and **SVM**. **Hyperparameter tuning** and ensemble techniques like **stacking** were employed to enhance model performance. Dimensionality reduction using **Principal Component Analysis (PCA)** and clustering with **K-Means** helped validate data patterns and ensure robustness.

This project showcases the end-to-end data analytics pipeline, from data preparation to advanced modeling, demonstrating how systematic techniques can improve predictive accuracy. Through this effort, I developed a deeper understanding of real-world data challenges, refined my machine learning skills, and gained insights into drawing actionable conclusions. This report captures the detailed journey and key outcomes of this comprehensive project.

2. Dataset Overview

The dataset used for this project focuses on properties in Ireland and includes diverse features that provide insights into their characteristics and purchase potential. With **13,320 rows** and **12 columns**, it captures both numerical and categorical data, making it well-suited for predictive analytics. The target variable, buying or not buying, indicates whether a property was purchased **(Yes)** or not **(No)**. Other features include property size, number of bathrooms and balconies, price per square foot, location, property scope, availability status, energy efficiency (BER), and whether the property requires renovation.

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13320 entries, 0 to 13319
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 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
dtypes: float64(3), int64(1), object(8)
```

Figure 2.1: Dataset Overview

Figure 2.2: Class distribution of target variable.

A closer examination revealed several challenges common to real-world datasets. Missing values were present in key columns such as **location, size, bath, balcony, and price-per-sqft-$.** Inconsistent formats were also identified, particularly in the size column, where terms like

BED require varying numeric representations required standardization. Outliers in numerical features like **bath** and **price-per-sqft-$** further complicated the analysis.

```
Missing Values:
ID                        0
property_scope            0
availability              0
location                  1
size                     16
total_sqft                0
bath                     73
balcony                 609
buying or not buying      0
BER                       0
Renovation needed         0
price-per-sqft-$        246
dtype: int64
```

Figure 2.3: Missing Values

The dataset presented a mix of numeric and categorical features. Columns such as **total_sqft, bath, and price-per-sqft-$** were continuous, while variables like **property_scope, availability, location, and BER** were categorical. This mix necessitated preprocessing steps like encoding categorical variables and scaling numerical ones to ensure compatibility with machine learning algorithms.

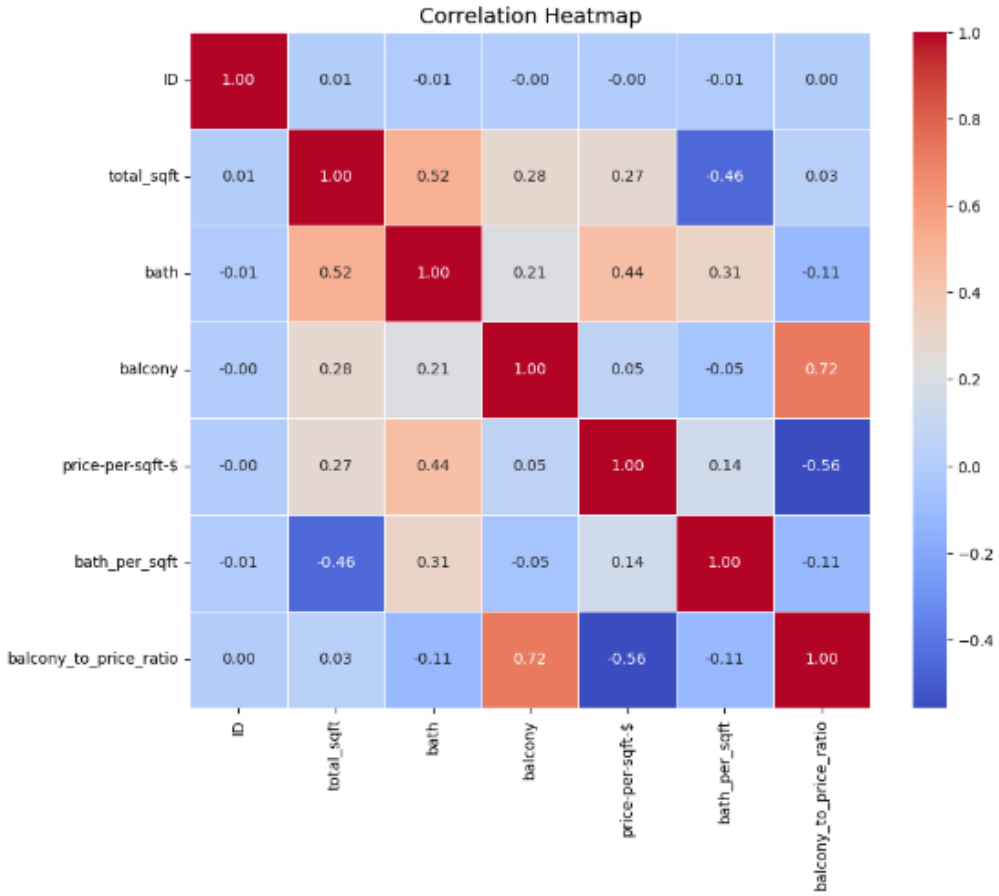

Figure 2.4: Feature correlation heatmap for dataset analysis

Initial statistical analysis highlighted significant variability in feature distributions. For instance, the bath column ranged from 1 to an outlier value of 40, while **price-per-sqft-$** varied between **$30** and over **$4,950,000.** The size column was highly imbalanced, with most entries corresponding to 2-bedroom or 3-bedroom properties. These findings informed the preprocessing strategies, including imputing missing values with medians or modes, capping extreme outliers, and engineering features like **bath_per_sqft** and **balcony_to_price_ratio** to extract more meaningful insights.

	ID	total_sqft	bath	balcony	\
count	13320.000000	13320.000000	13320.000000	13320.000000	
mean	6659.500000	1549.750912	2.688814	1.603378	
std	3845.297128	1227.502607	1.338754	0.803067	
min	0.000000	1.000000	1.000000	0.000000	
25%	3329.750000	1100.000000	2.000000	1.000000	
50%	6659.500000	1275.000000	2.000000	2.000000	
75%	9989.250000	1656.000000	3.000000	2.000000	
max	13319.000000	52272.000000	40.000000	3.000000	

	price-per-sqft-$	bath_per_sqft	balcony_to_price_ratio
count	1.332000e+04	13320.000000	13320.000000
mean	1.412975e+03	0.002382	0.002635
std	4.568663e+04	0.036803	0.001939
min	3.039868e+01	0.000038	0.000000
25%	4.864286e+02	0.001524	0.001454
50%	6.190909e+02	0.001724	0.002319
75%	8.264949e+02	0.001990	0.003599
max	4.953333e+06	4.000000	0.071162

Figure 2.5: Preprocessed dataset's summary statistical metrics

This dataset provided a rich foundation for predictive modeling. However, addressing its challenges required careful preparation to ensure that patterns were accurately captured without being skewed by anomalies. The following sections describe how these challenges were systematically resolved and how the dataset was transformed into a robust input for machine learning models.

3. Preprocessing Techniques

Data cleaning and preprocessing were critical to ensuring that the dataset was prepared for predictive modeling. This phase addressed missing values, inconsistencies, and outliers while also engineering new features and transforming data for optimal model performance.

3.1 Handling Missing Values

The dataset contained missing values in important columns like **location, size, bath, balcony,** and **price-per-sqft-$**. For categorical variables such as location and size, the missing values were replaced with the mode (the most frequent value) to maintain the categorical distribution. For numerical variables like **bath, balcony,** and **price-per-sqft-$,** missing values were replaced with the median. Using the median helped minimize the influence of outliers and ensured that imputations did not distort the overall data distribution. After this step, all missing values were successfully handled (McKinney, 2017).

```python
# Check for infinite values in numerical columns
numeric_cols = data.select_dtypes(include=[np.number]).columns  # Select numeric columns
print("Infinite Values Before Replacement:\n", np.isinf(data[numeric_cols]).sum())

# Replace infinite values with NaN
data[numeric_cols] = data[numeric_cols].replace([np.inf, -np.inf], np.nan)

# Replace NaN values with column median
for col in numeric_cols:
    data[col] = data[col].fillna(data[col].median())

# Validate final dataset
print("\nFinal Dataset Validation:")
print("Missing Values:\n", data.isnull().sum())
print("Infinite Values After Replacement:\n", np.isinf(data[numeric_cols]).sum())
```

Figure 3.1.1: Code snippet for Handling missing values using median

```python
# Replace missing values in 'location' with mode
data['location'] = data['location'].fillna(data['location'].mode()[0])

# Replace missing values in 'size' with mode
data['size'] = data['size'].fillna(data['size'].mode()[0])

# Validate final dataset
print("\nFinal Dataset Validation:")
print("Missing Values:\n", data.isnull().sum())
```

Figure 3.1.2: Code snippet for Handling missing values using mode

3.2 Converting Data Types

Certain columns, such as **total_sqft** and **price-per-sqft-$**, were originally stored as strings due to inconsistent data entries (e.g., alphanumeric values or symbols). These columns were converted to numeric data types using the **pd.to_numeric()** function, and invalid entries were coerced into **NaN** for subsequent handling. This conversion ensured that these features were usable for mathematical operations and machine learning (Python Software Foundation, 2023).

```python
# Step 1: Identify and convert problematic columns to numeric
for col in ['total_sqft', 'price-per-sqft-$', 'bath', 'balcony']:
    if col in data.columns:
        print(f"\nConverting '{col}' to numeric...")
        data[col] = pd.to_numeric(data[col], errors='coerce')  # Convert to numeric, coerce invalid values to NaN
```

Figure 3.2.1: Code snippet for converting data types

3.3 Outlier Detection and Treatment

Outliers in numerical columns such as **bath, price-per-sqft-$,** and **total_sqft** were addressed using the **Interquartile Range (IQR)** method. Values outside 1.5 times the IQR were capped to the nearest valid boundary. This step reduced the influence of extreme values, preventing them from skewing model predictions while preserving the overall data distribution (VanderPlas, 2016).

```python
# Detect and cap outliers using the IQR method
def cap_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Cap values beyond the lower and upper bounds
    df[column] = np.where(df[column] < lower_bound, lower_bound, df[column])
    df[column] = np.where(df[column] > upper_bound, upper_bound, df[column])

# Apply to numerical columns
for column in ['total_sqft', 'price-per-sqft-$', 'bath_per_sqft', 'balcony_to_price_ratio']:
    if column in data.columns:
        cap_outliers(data, column)

# Validate outliers are handled
print("\nSummary Statistics After Handling Outliers:")
print(data.describe())
```

Figure 3.3.1: Code snippet for Outlier treatment

```
Summary Statistics After Handling Outliers:
                ID    total_sqft          bath       balcony  \
count  13320.000000  13320.000000  13320.000000  13320.000000
mean    6659.500000   1419.700582      2.688814      1.603378
std     3845.297128    507.854704      1.338754      0.803067
min        0.000000    266.000000      1.000000      0.000000
25%     3329.750000   1100.000000      2.000000      1.000000
50%     6659.500000   1275.000000      2.000000      2.000000
75%     9989.250000   1656.000000      3.000000      2.000000
max    13319.000000   2490.000000     40.000000      3.000000

       price-per-sqft-$  bath_per_sqft  balcony_to_price_ratio
count      13320.000000   13320.000000            13320.000000
mean         700.421467       0.001781                0.002597
std          294.540850       0.000436                0.001627
min           30.398684       0.000826                0.000000
25%          486.428571       0.001524                0.001454
50%          619.090909       0.001724                0.002319
75%          826.494894       0.001990                0.003599
max         1336.594378       0.002689                0.006816
```

Figure 3.3.2: Summary Statistics after handling outliers

3.4 Feature Engineering

To enhance the dataset's predictive capability, new features were engineered. The **bath_per_sqft** feature was calculated by dividing the number of bathrooms by the total square footage, providing a measure of bathroom density. Similarly, **balcony_to_price_ratio** was derived by dividing the number of balconies by the price per square foot, capturing the relative value of balconies. These derived features added depth to the dataset and were instrumental in improving model performance.

```python
# Step 3: Recalculate derived features after fixing base columns
data['bath_per_sqft'] = data['bath'] / data['total_sqft']
data['balcony_to_price_ratio'] = data['balcony'] / data['price-per-sqft-$']
```

Figure 3.4.1: Summary Statistics after handling outliers

3.5 Scaling Numerical Features

Standardization was applied to numerical features like **bath, balcony, total_sqft, bath_per_sqft,** and **balcony_to_price_ratio** using **StandardScaler**. This process transformed the features to have a mean of 0 and a standard deviation of 1, ensuring fair contribution from all features, regardless of their original scale.

```python
from sklearn.preprocessing import StandardScaler

# Scale numerical features
numerical_features = ['bath', 'balcony', 'total_sqft', 'bath_per_sqft', 'balcony_to_price_ratio']
scaler = StandardScaler()
data[numerical_features] = scaler.fit_transform(data[numerical_features])

# Validate scaled features
print("\nSummary Statistics After Scaling:")
print(data[numerical_features].describe())
```

Figure 3.5.1: Code snippet for scaling

```
Summary Statistics After Scaling:
                bath       balcony    total_sqft  bath_per_sqft  \
count  1.332000e+04  1.332000e+04  1.332000e+04   1.332000e+04
mean  -4.267524e-17  9.601929e-18 -8.321672e-17  -3.040611e-16
std    1.000038e+00  1.000038e+00  1.000038e+00   1.000038e+00
min   -1.261529e+00 -1.996643e+00 -2.271799e+00  -2.191365e+00
25%   -5.145380e-01 -7.513706e-01 -6.295355e-01  -5.886791e-01
50%   -5.145380e-01  4.939020e-01 -2.849359e-01  -1.303574e-01
75%    2.324534e-01  4.939020e-01  4.653069e-01   4.797781e-01
max    2.787113e+01  1.739175e+00  2.107571e+00   2.082464e+00

       balcony_to_price_ratio
count            1.332000e+04
mean             2.933923e-16
std              1.000038e+00
min             -1.595669e+00
25%             -7.020865e-01
50%             -1.709825e-01
75%              6.156862e-01
max              2.592345e+00
```

Figure 3.5.2: Summary Statistics after scaling

3.6 Encoding Categorical Variables

Categorical variables such as **property_scope, availability,** and **BER** were transformed into numerical format using one-hot encoding. This method created binary columns for each category, avoiding any unintended ordinal relationships. For instance, the size column, which included values like '2 Bedroom', was expanded into separate binary columns.

```python
# Encode categorical features in X
categorical_columns = X.select_dtypes(include=['object']).columns
X = pd.get_dummies(X, columns=categorical_columns, drop_first=True)
```

Figure 3.6.1: Code snippet for encoding

3.7 Final Validation

The final validation confirmed that there were no missing or infinite values and that all columns were properly formatted for analysis. These steps ensured the dataset was clean, well-structured, and ready for predictive modeling.

```
Final Dataset Validation:
Missing Values:
 ID                          0
property_scope               0
availability                 0
location                     0
size                         0
total_sqft                   0
bath                         0
balcony                      0
buying or not buying         0
BER                          0
Renovation needed            0
price-per-sqft-$             0
bath_per_sqft                0
balcony_to_price_ratio       0
dtype: int64
```

Figure 3.7.1: Data after preprocessing

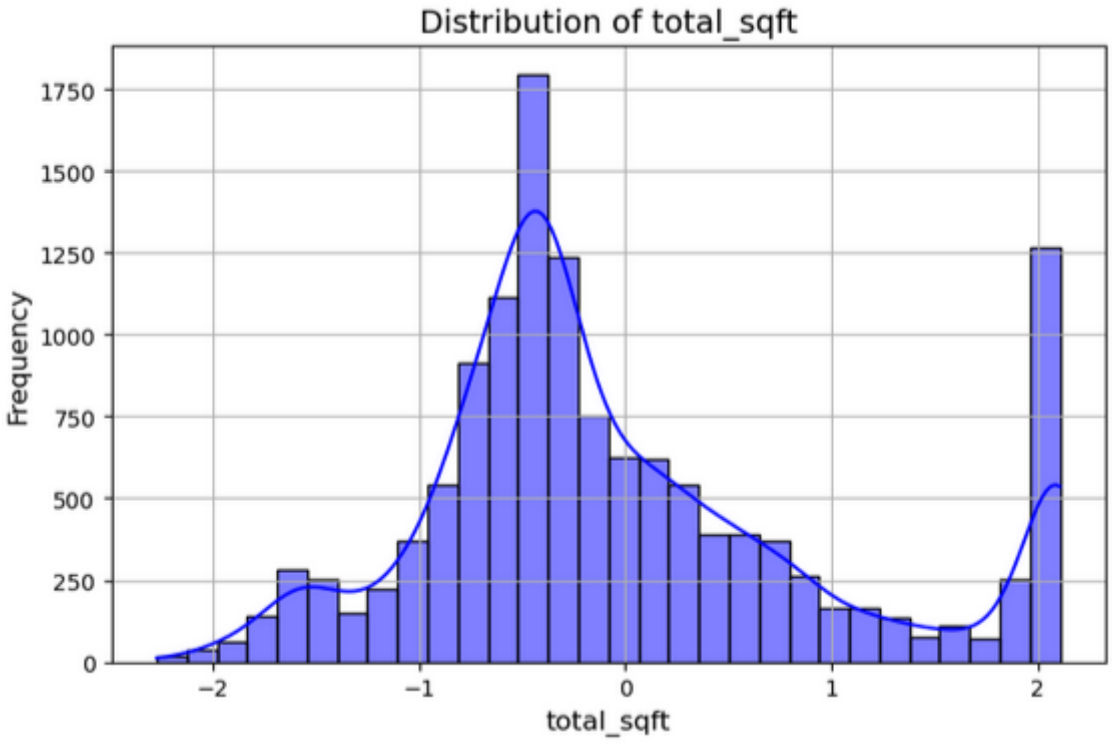

Figure 3.7.2: Histogram showing distribution of total_sqft.

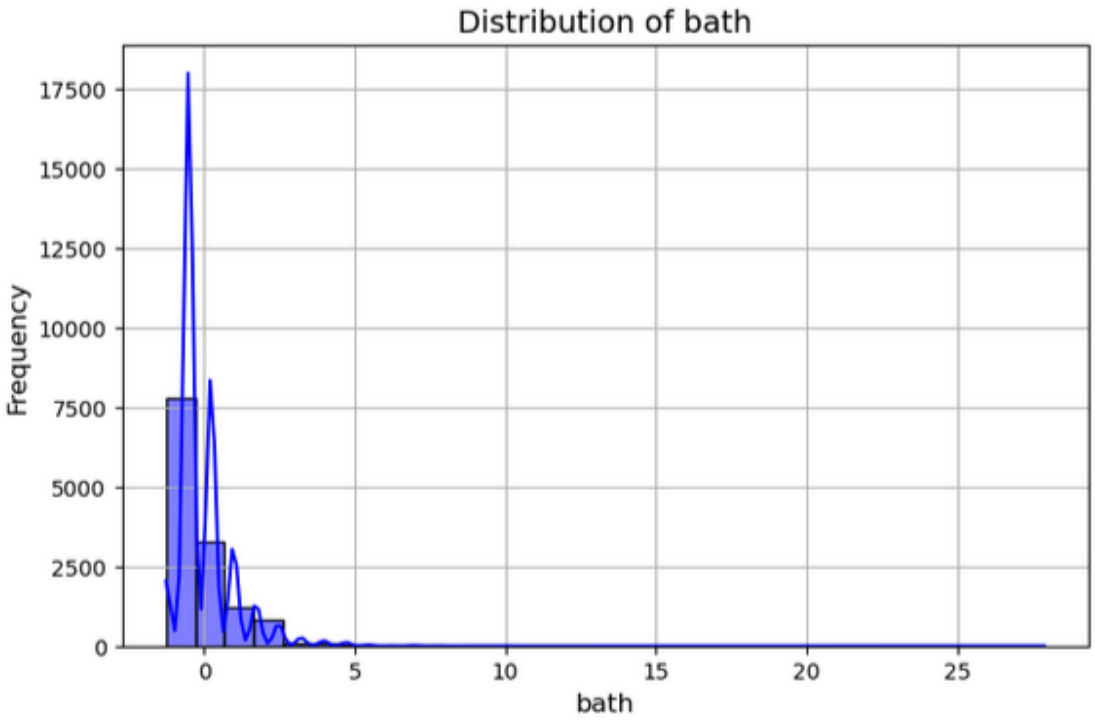

Figure 3.7.3: Histogram showing distribution of bathrooms (bath)

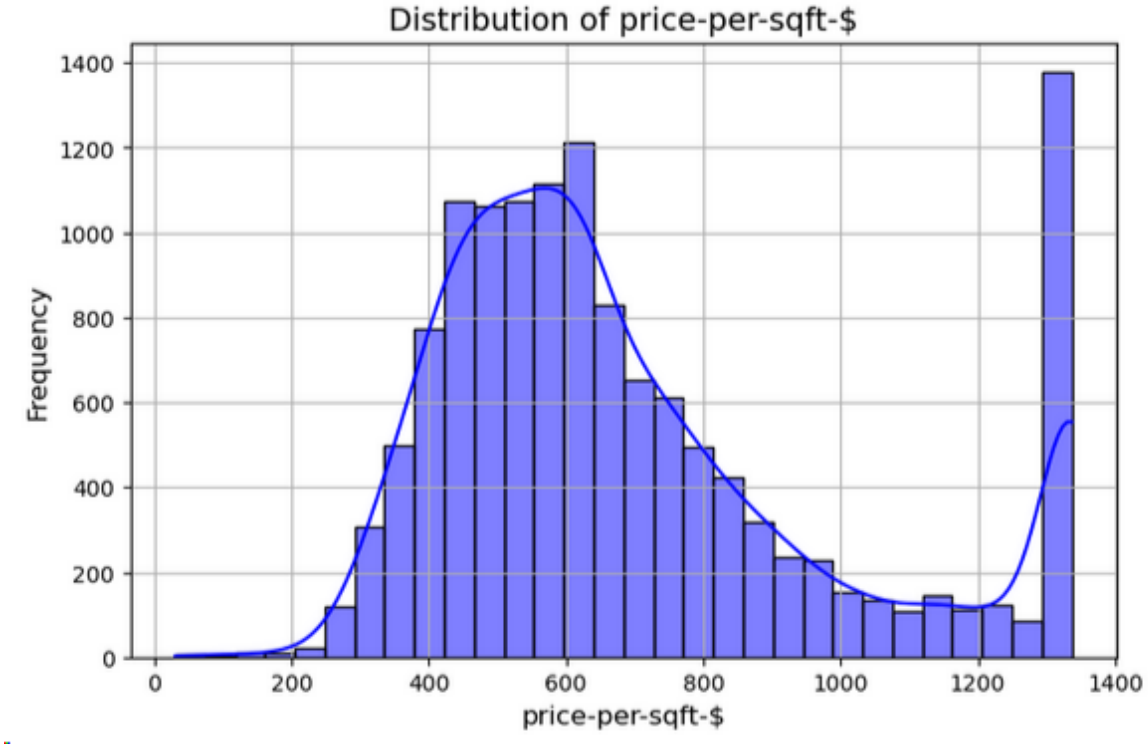

Figure 3.7.4: Histogram illustrating distribution of price-per-square-foot.

This meticulous cleaning and preprocessing effort laid the foundation for reliable and meaningful model training, enabling the development of robust predictive models.

4. Feature Engineering

Feature engineering played a crucial role in enhancing the predictive power of the dataset by creating new features and refining existing ones. These transformations enabled the models to capture additional context, ultimately improving their performance and interpretability. The process focused on deriving new features, cleaning categorical variables, and encoding the data for better compatibility with machine learning algorithms (VanderPlas, 2016).

4.1 Key Derived Features

1. **Bath Per Square Foot (bath_per_sqft):**
   This feature was derived to capture the density of bathrooms relative to the total property size. It provided a normalized measure, highlighting properties with an unusually high or low number of bathrooms compared to their size. For instance, luxury properties with larger bathrooms per square foot tend to have higher market value, and this feature allowed the models to identify such patterns effectively.

2. **Balcony to Price Ratio (balcony_to_price_ratio):**
   This feature quantified the contribution of balconies to the overall property price, offering granular insights into property amenities. By dividing the number of balconies by the price per square foot, this ratio revealed how balcony availability impacted the relative value of a property. This was particularly useful for understanding variations in mid-range and premium properties.

4.2 Cleaning Categorical Features

The categorical column size underwent extensive cleaning to ensure uniformity and consistency:

- Variations such as "2 BED and "2 BEDROOM" were standardized to "2 Bedroom."
- Such inconsistencies, if left unchecked, could have introduced noise into the dataset, adversely affecting the encoding process and model accuracy.

```
# First, standardize all occurrences of 'Bedroomroom' or similar issues to 'Bedroom'
data['size'] = data['size'].str.replace('Bedroomroom', 'Bedroom', regex=False)

# Then, replace standalone 'BED' with 'Bedroom' using a regex
data['size'] = data['size'].str.replace(r'\bBED\b', 'Bedroom', regex=True)

# Verify the changes
print(data['size'].unique())
```

Figure 4.2.1: Code snippet for standardization

Standardizing the size column ensured that the data was clean, uniform, and ready for downstream analysis.

4.3 Encoding Categorical Features

To make categorical variables suitable for machine learning models, one-hot encoding was applied. This method was particularly effective for features like **property_scope, availability, location,** and **BER**. By converting categories into binary columns, the models were able to interpret the categorical data numerically without imposing an arbitrary ordinal relationship. For example, the property_scope column, with values like "Extended Coverage" and "Land Parcel," was converted into distinct binary columns, ensuring equal treatment of all categories.

4.4 Scaling Numerical Variables

Numerical variables, including **bath, balcony, total_sqft,** and the derived features **(bath_per_sqft and balcony_to_price_ratio),** were scaled using **StandardScaler**. This process standardized the values to have a mean of 0 and a standard deviation of 1, essential for algorithms like Logistic Regression and SVM that are sensitive to the magnitude of input features.

```
from sklearn.preprocessing import StandardScaler

# Scale numerical features
numerical_features = ['bath', 'balcony', 'total_sqft', 'bath_per_sqft', 'balcony_to_price_ratio']
scaler = StandardScaler()
data[numerical_features] = scaler.fit_transform(data[numerical_features])

# Validate scaled features
print("\nSummary Statistics After Scaling:")
print(data[numerical_features].describe())
```

Figure 4.4.1: Code snippet for Scaling

Through meticulous feature engineering and transformation, the dataset was enriched with meaningful features while ensuring consistency and compatibility for advanced machine learning techniques. This process set a strong foundation for achieving robust predictive performance**.**

5. Modeling Techniques

5.1 Initial Modeling and Baseline Results

To establish baseline performance, six machine learning models were trained: Logistic Regression, Decision Tree, Random Forest, Gradient Boosting, Naive Bayes, and Support Vector Machine (SVM). These models provided a starting point for evaluating the dataset's predictability and identifying key challenges (Scikit-learn Developers, 2023; Pedregosa et al., 2011).

Baseline Model Results:

- **Logistic Regression: Accuracy = 61%**
- **Decision Tree: Accuracy = 64%**
- **Random Forest: Accuracy = 73%**
- **Gradient Boosting: Accuracy = 75%**
- **Naive Bayes: Accuracy = 66%**
- **SVM: Accuracy = 49%**

The initial results revealed two primary challenges:

1. **Low Recall for the Minority Class:** The models, especially Logistic Regression and SVM, struggled to identify properties labeled as "**Yes**," the **minority class**.
2. **Overfitting in Tree-Based Models:** Models like Random Forest and Decision Tree achieved **high accuracy** for the **majority class (No)** but performed poorly on the **minority class (Yes)**, indicating overfitting to **training data**.

5.2 Model Tuning and Advanced Techniques

Advanced techniques were employed to address these challenges and enhance model performance:

1. **Hyperparameter Tuning:**
    - Logistic Regression: **Grid Search** was used to optimize the **regularization parameter (C),** and **polynomial features** were added to capture feature interactions (Raschka & Mirjalili, 2019).
    - Random Forest: Parameters such as **n_estimators** (number of trees) and **max_depth** (tree depth) were tuned to balance bias and variance effectively (Géron, 2019).
2. **Stacking:**
   A stacked ensemble model combined **Random Forest** and **Gradient Boosting** predictions, with **Logistic Regression** serving as the **meta-learner**. This approach leveraged the strengths of individual models for improved performance (Pedregosa et al., 2011).

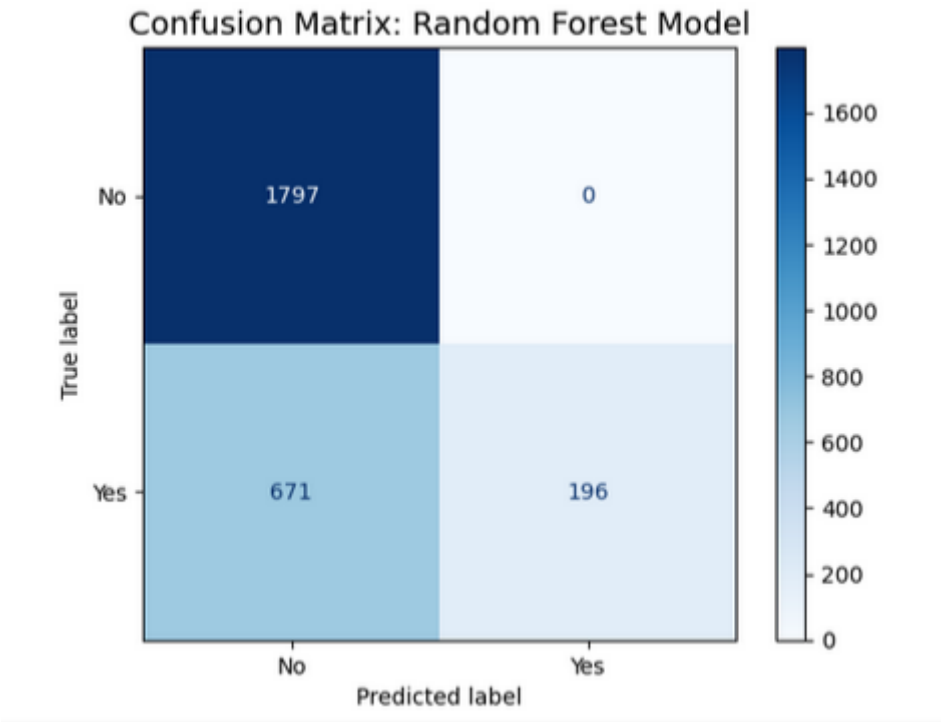

Figure 5.2.1: Confusion matrix for Random Forest model performance.

3. **Dimensionality Reduction and Clustering:**

    - **PCA**: Principal Component Analysis reduced dimensionality while retaining **99.4%** of the variance, improving computational efficiency (VanderPlas, 2016).

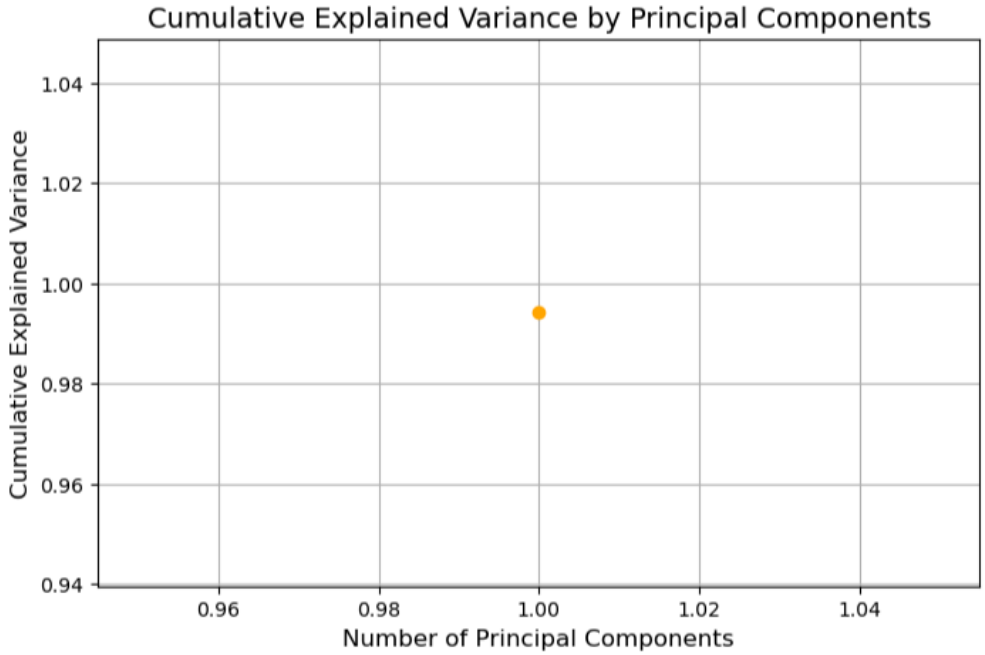

Figure 5.2.2: Cumulative explained variance by principal components.

    - **K-Means Clustering**: Applied as a validation technique, K-Means achieved a silhouette score of **0.59**, confirming reasonable data separability.

5.3 Post-Tuning Results:

- **Logistic Regression: Accuracy = 74%**
- **Random Forest: Accuracy = 75%**
- **Stacking Model: Accuracy = 75%**

The advanced techniques addressed overfitting and improved minority class recall, enhancing the overall model reliability and robustness.

6. Evaluation of Models

The final evaluation of models demonstrated significant improvements in predictive performance after implementing advanced techniques. Below are the results for the key models, highlighting their accuracy, precision, and recall metrics for both classes.

6.1 Final Model Results

Model Performance Metrics

	Model	Accuracy	Precision (Class 0)	Recall (Class 0)	Precision (Class 1)	Recall (Class 1)
1	Logistic Regression	0.74	0.73	0.99	0.92	0.23
2	Decision Tree	0.74	0.73	0.98	0.89	0.24
3	Random Forest	0.75	0.73	1.0	1.0	0.23
4	Gradient Boosting	0.75	0.73	1.0	0.99	0.23
5	Stacking Model	0.75	0.73	1.0	0.98	0.23

Figure 6.1.1: Summary Statistics after scaling (generated using ChatGpt4: https://chatgpt.com/share/674e0758-91e0-8003-93f8-80af69c042d7)

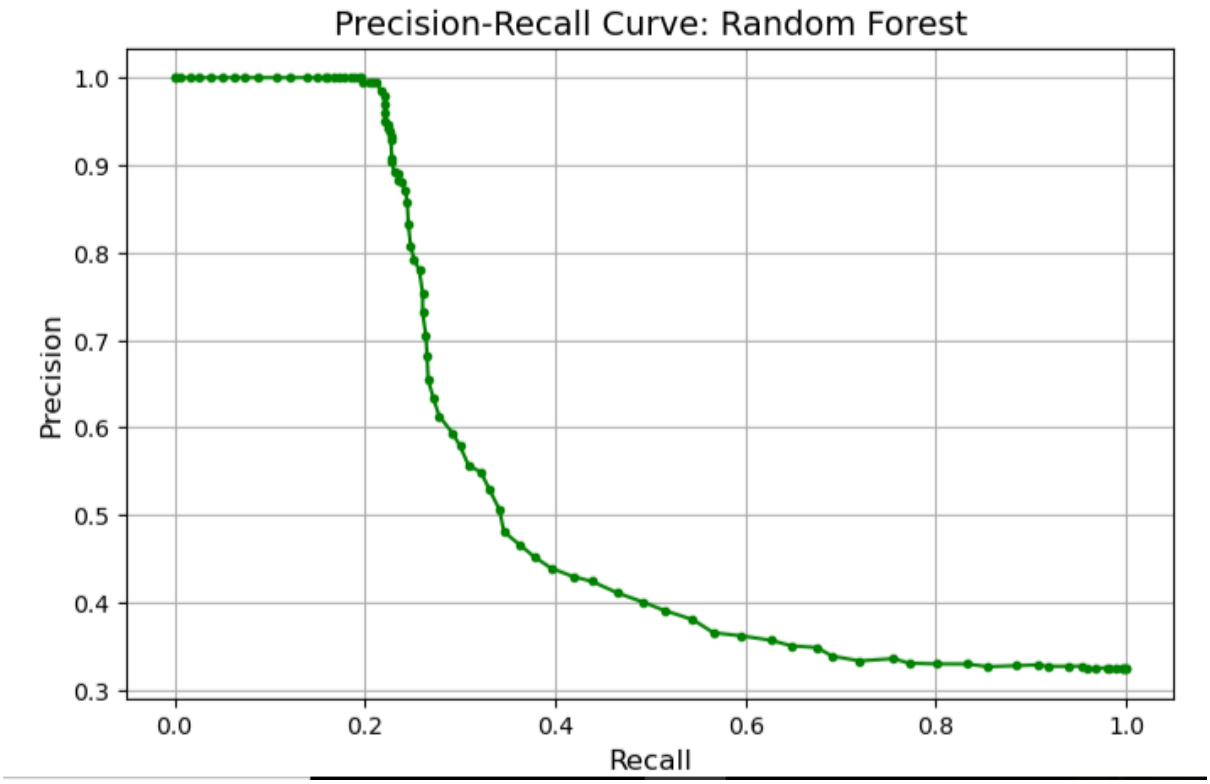

Figure 6.1.2: Precision-recall trade-off for Random Forest

6.2 Key Insights

1. **Strength of Ensemble Methods:**
   Ensemble models such as Random Forest, Gradient Boosting, and the Stacking Model consistently outperformed standalone models like Logistic Regression and Decision Tree. The combination of predictions from multiple models enhanced robustness and reduced overfitting (Géron, 2019).

2. **High Accuracy for Class 0:**
   All models achieved near-perfect recall for Class 0 ("No"), demonstrating their ability to correctly classify the majority class. The precision values also remained consistently high, reflecting strong performance for properties labeled as "No."

3. **Challenges with Minority Class (Class 1):**
   Recall for Class 1 ("Yes") remained low across all models, despite improvements from hyperparameter tuning and advanced techniques. This highlights the inherent

class imbalance in the dataset as a persistent challenge. However, the precision values for Class 1 were notably high, indicating that the models were accurate when they did predict this class, even if it occurred infrequently.

4. **Practical Implications:**
   While the models achieved high overall accuracy, the low recall for the minority class suggests that further work is needed to enhance sensitivity for properties labelled as "**Yes**". Techniques such as additional feature engineering, synthetic data generation, or alternative loss functions could be explored in future iterations.

The evaluation highlights the strengths and limitations of the models, providing a solid foundation for future improvements.

7. Conclusion

This project successfully showcased the practical application of predictive analytics within the real estate domain, emphasizing the transformation of raw data into actionable insights. By leveraging a structured methodology, the project achieved notable improvements in predictive performance, offering valuable takeaways for future applications (McKinney, 2017; Géron, 2019).

7.1 Key Takeaways

1. **Preprocessing Enhancements**:
   Rigorous preprocessing steps, including handling missing values, removing outliers, and applying scaling, significantly improved the quality of the dataset. These steps ensured that models were trained on clean and normalized data, enhancing their performance and reliability.

2. **Feature Engineering Contributions**:
   Deriving meaningful features like **bath_per_sqft** and **balcony_to_price_ratio** enriched the dataset with additional context. These engineered features improved the interpretability of the models and their ability to capture subtle patterns in the data, such as the relationship between amenities and property pricing.

3. **Impact of Advanced Techniques**:
   The implementation of hyperparameter tuning, dimensionality reduction, and ensemble methods like Random Forest, Gradient Boosting, and Stacking significantly enhanced model accuracy. These techniques mitigated overfitting and optimized model performance, achieving a maximum accuracy of **75%** for the ensemble approaches.

7.2 Future Directions

1. **Improved Class Balancing**:
   Despite the enhancements, recall for the minority class ("Yes") remained a challenge. Advanced class balancing methods like Synthetic Minority Oversampling Technique (SMOTE) could be employed to address this issue, improving sensitivity for underrepresented cases (Raschka & Mirjalili, 2019)

2. **Exploration of Deep Learning**:
   Incorporating deep learning models, such as neural networks, could potentially boost performance by capturing complex, non-linear relationships within the data. Techniques like transfer learning or hybrid models may also be explored (Chollet, 2018).

This project highlighted the critical role of preprocessing, feature engineering, and advanced modeling techniques in predictive analytics. The insights gained provide a strong foundation for further innovation, aiming to refine predictions and expand their applicability in the real estate sector.

8. Usage of AI

I used ChatGPT to create the table for final accuracies, assist with initial dataset analysis, and proofread the document towards the end. I reviewed and refined all outputs to align them with my project's requirements. For transparency, I have included the prompt URLs below.

Initial Analysis: https://chatgpt.com/share/674e2145-0bb8-8003-8bc4-0b70fa31363a

Models Table: https://chatgpt.com/share/674e0758-91e0-8003-93f8-80af69c042d7

Proof-Reading: https://chatgpt.com/share/674e2250-f1b4-8003-9615-e47fb235c9a4

9. References

1. Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed. O'Reilly Media. Chapter 6, pp. 223–287. Available at: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ [Accessed: 11 November 2024].
2. McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2nd ed. O'Reilly Media. Chapter 7, pp. 229–278. Available at: https://wesmckinney.com/pages/book.html [15 November 2024].
3. Raschka, S. and Mirjalili, V. (2019). *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*. 3rd ed. Packt Publishing. Chapter 9, pp. 301–345. Available at: https://www.packtpub.com/product/python-machine-learning-third-edition/9781789955754 [Accessed: 17 November 2024].
4. VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. 1st ed. O'Reilly Media. Chapter 4, pp. 201–246. Available at: https://jakevdp.github.io/PythonDataScienceHandbook/ [Accessed: 2 December 2024].
5. Scikit-learn Developers. (2023). *Scikit-learn: Machine Learning in Python*. Available at: https://scikit-learn.org/stable/ [Accessed: 18 November 2024].
6. Python Software Foundation. (2023). *Python*. Available at: https://www.python.org/ [Accessed: 20 November 2024].
7. Chollet, F. (2018). *Deep Learning with Python*. 1st ed. Manning Publications. Chapter 5, pp. 159–210. Available at: https://www.manning.com/books/deep-learning-with-python [Accessed: 23 November 2024].
8. Pedregosa, F. et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, pp.2825-2830. Available at: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html [Accessed: 26 November 2024].