Midterm Exam Top-up Score Report

By 6738089121 Natnicha Charukitphaisarn

1 Introduction

The objective of this task was to construct a complete *machine-learning pipeline* to predict Airbnb listing prices.

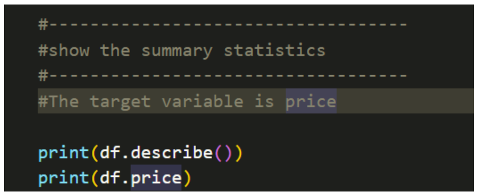
The instructions include

- **Exploratory Data Analysis (EDA)**
- **Data Preprocessing** (cleaning, encoding, scaling, splitting)
- **Model Building** using two classical algorithms
- **Model Evaluation** using MSE and R^2 metrics

2 Exploratory Data Analysis (EDA)

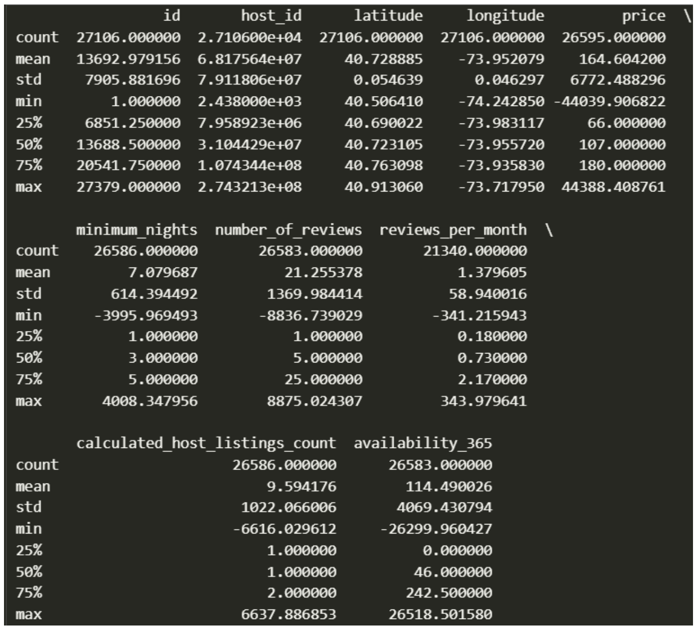
2.1 Summary Statistics

These lines are to show the data on the dataframe created from the csv file



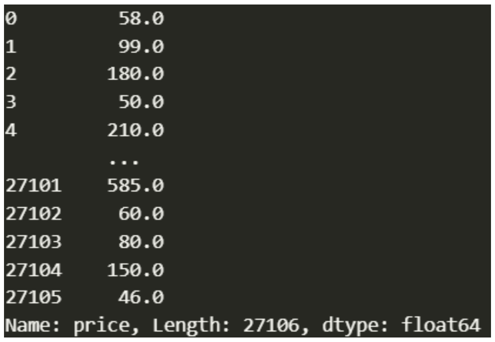
```
#----------------------------------
#show the summary statistics
#----------------------------------
#The target variable is price

print(df.describe())
print(df.price)
```

The result for df.describe() is



```
                 id       host_id      latitude     longitude         price  \
count  27106.000000  2.710600e+04  27106.000000  27106.000000  26595.000000
mean   13692.979156  6.817564e+07     40.728885    -73.952079    164.604200
std     7905.881696  7.911806e+07      0.054639      0.046297   6772.488296
min        1.000000  2.438000e+03     40.506410    -74.242850 -44039.906822
25%     6851.250000  7.958923e+06     40.690022    -73.983117     66.000000
50%    13688.500000  3.104429e+07     40.723105    -73.955720    107.000000
75%    20541.750000  1.074344e+08     40.763098    -73.935830    180.000000
max    27379.000000  2.743213e+08     40.913060    -73.717950  44388.408761

       minimum_nights  number_of_reviews  reviews_per_month  \
count    26586.000000       26583.000000       21340.000000
mean         7.079687          21.255378           1.379605
std        614.394492        1369.984414          58.940016
min      -3995.969493       -8836.739029        -341.215943
25%          1.000000           1.000000           0.180000
50%          3.000000           5.000000           0.730000
75%          5.000000          25.000000           2.170000
max       4008.347956        8875.024307         343.979641

       calculated_host_listings_count  availability_365
count                    26586.000000      26583.000000
mean                         9.594176        114.490026
std                       1022.066006       4069.430794
min                      -6616.029612     -26299.960427
25%                          1.000000          0.000000
50%                          1.000000         46.000000
75%                          2.000000        242.500000
max                       6637.886853      26518.501580
```

The result for df.price :



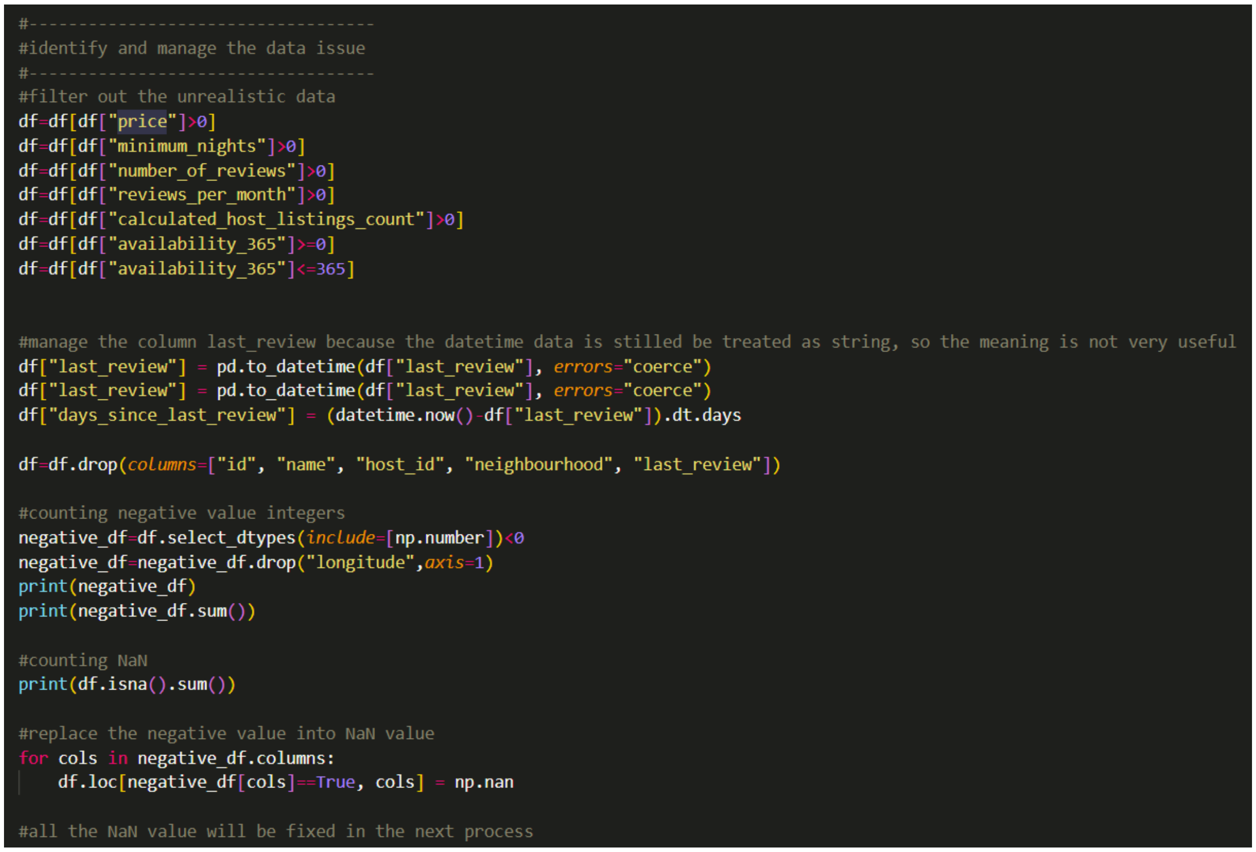
```
0         58.0
1         99.0
2        180.0
3         50.0
4        210.0
         ...
27101    585.0
27102     60.0
27103     80.0
27104    150.0
27105     46.0
Name: price, Length: 27106, dtype: float64
```

From this inspection, I noticed that many columns display unrealistic extremes, indicating input errors or placeholder values.

2.2 Data Issues Identified

From the csv file inspection, there appear to be some data issues that might affect the ML process such as,

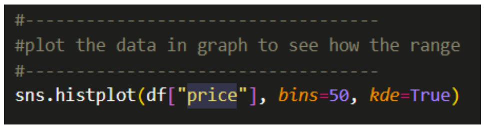
- Invalid ranges (negative value received from counting)
- Outliers
- Missing values
- Categorical columns (require encoding)



```
#-----------------------------------
#identify and manage the data issue
#-----------------------------------
#filter out the unrealistic data
df=df[df["price"]>0]
df=df[df["minimum_nights"]>0]
df=df[df["number_of_reviews"]>0]
df=df[df["reviews_per_month"]>0]
df=df[df["calculated_host_listings_count"]>0]
df=df[df["availability_365"]>=0]
df=df[df["availability_365"]<=365]


#manage the column last_review because the datetime data is stilled be treated as string, so the meaning is not very useful
df["last_review"] = pd.to_datetime(df["last_review"], errors="coerce")
df["last_review"] = pd.to_datetime(df["last_review"], errors="coerce")
df["days_since_last_review"] = (datetime.now()-df["last_review"]).dt.days

df=df.drop(columns=["id", "name", "host_id", "neighbourhood", "last_review"])

#counting negative value integers
negative_df=df.select_dtypes(include=[np.number])<0
negative_df=negative_df.drop("longitude",axis=1)
print(negative_df)
print(negative_df.sum())

#counting NaN
print(df.isna().sum())

#replace the negative value into NaN value
for cols in negative_df.columns:
    df.loc[negative_df[cols]==True, cols] = np.nan

#all the NaN value will be fixed in the next process
```

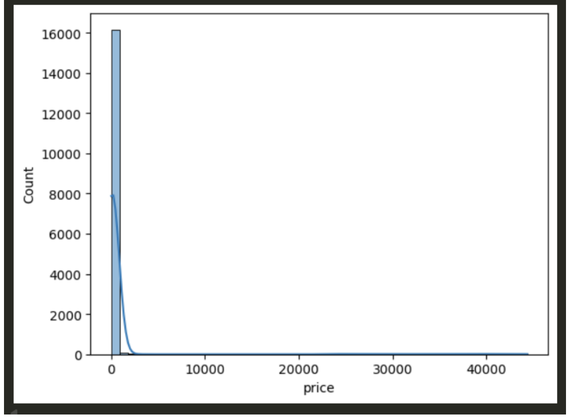
As the processes are described in the screen capture, in the end, I got a cleaned dataframe with unrealistic data removed, turn the datetime data to be more effient in an integer form (number of days), and turn the negative numbers(if still left) into NaN value to be handled in the next process.

Before moving on to data preprocessing, I inspect the price column once more as graphs

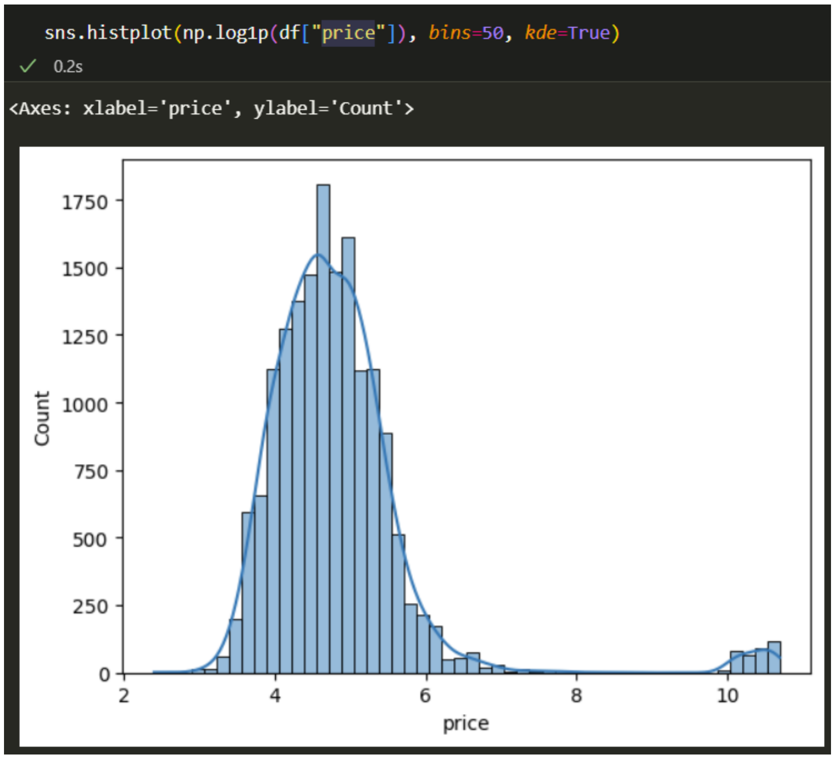


```
#-----------------------------------
#plot the data in graph to see how the range
#-----------------------------------
sns.histplot(df["price"], bins=50, kde=True)
```

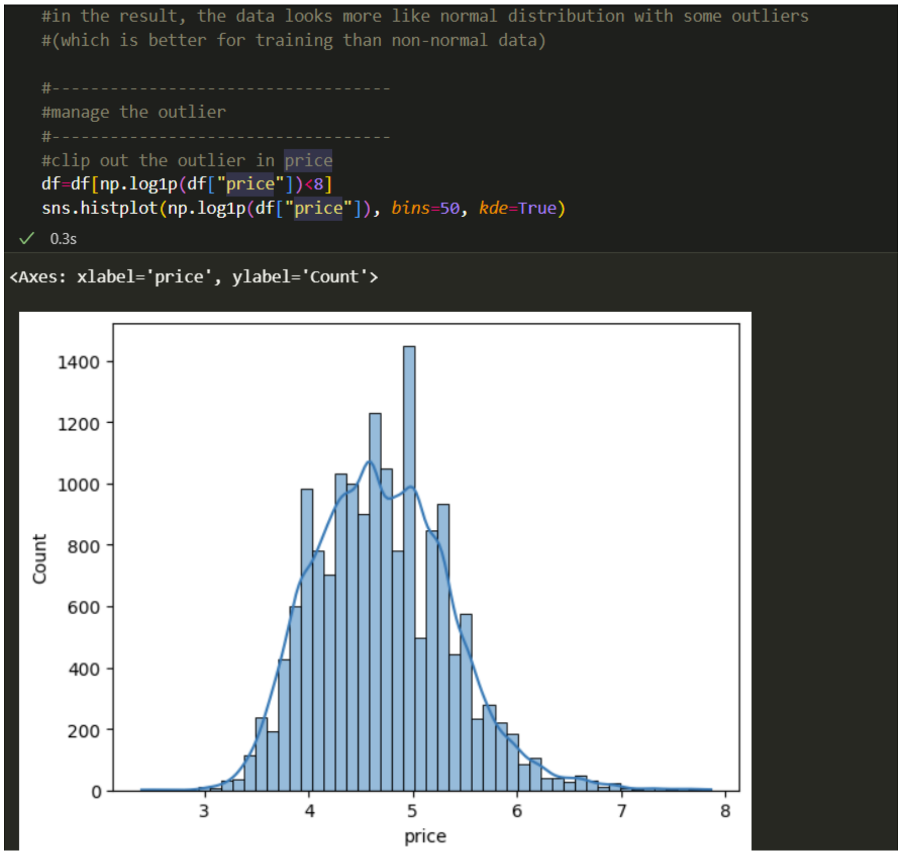
```
<Axes: xlabel='price', ylabel='Count'>
```



From the graph, I found out that the data is not normally distributed at all, so I addressed this issue by applying logarithm.



```
sns.histplot(np.log1p(df["price"]), bins=50, kde=True)
```
```
0.2s
```
```
<Axes: xlabel='price', ylabel='Count'>
```



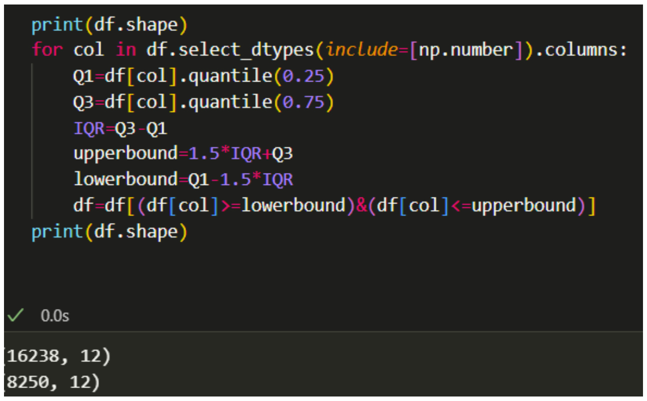
```
#in the result, the data looks more like normal distribution with some outliers
#(which is better for training than non-normal data)

#----------------------------------
#manage the outlier
#----------------------------------
#clip out the outlier in price
df=df[np.log1p(df["price"])<8]
sns.histplot(np.log1p(df["price"]), bins=50, kde=True)
```

✓ 0.3s

```
<Axes: xlabel='price', ylabel='Count'>
```

The result appeared to be much more clear, so it is ready to be used for training

Next, I continue to remove the outlier in the other columns by using IQR along with checking the dataframe's shape



```
print(df.shape)
for col in df.select_dtypes(include=[np.number]).columns:
    Q1=df[col].quantile(0.25)
    Q3=df[col].quantile(0.75)
    IQR=Q3-Q1
    upperbound=1.5*IQR+Q3
    lowerbound=Q1-1.5*IQR
    df=df[(df[col]>=lowerbound)&(df[col]<=upperbound)]
print(df.shape)
```

✓ 0.0s

```
(16238, 12)
(8250, 12)
```

Discussion for data analysis:

- for numeric data
    - data in numeric columns: price, minimum_nights, number_of_reviews, reviews_per_month, calculated_host_listings_count, availability_365
    - most data should not be a negative value because it is received by counting so negative value is unrealistic data within appropriate ranges are better for model training than scattered data, so filter out the outliers by using IQR

- for nominal data
    - the NaN value data should be next replaced with "most frequent" data to be usable

3 Data Preprocessing

3.1 performing proper preprocessing using scikit-learn

In this, 3 steps are performed

1. **Numeric pipeline:**

   SimpleImputer(strategy="mean") ⟶ StandardScaler()
2. **Categorical pipeline:**

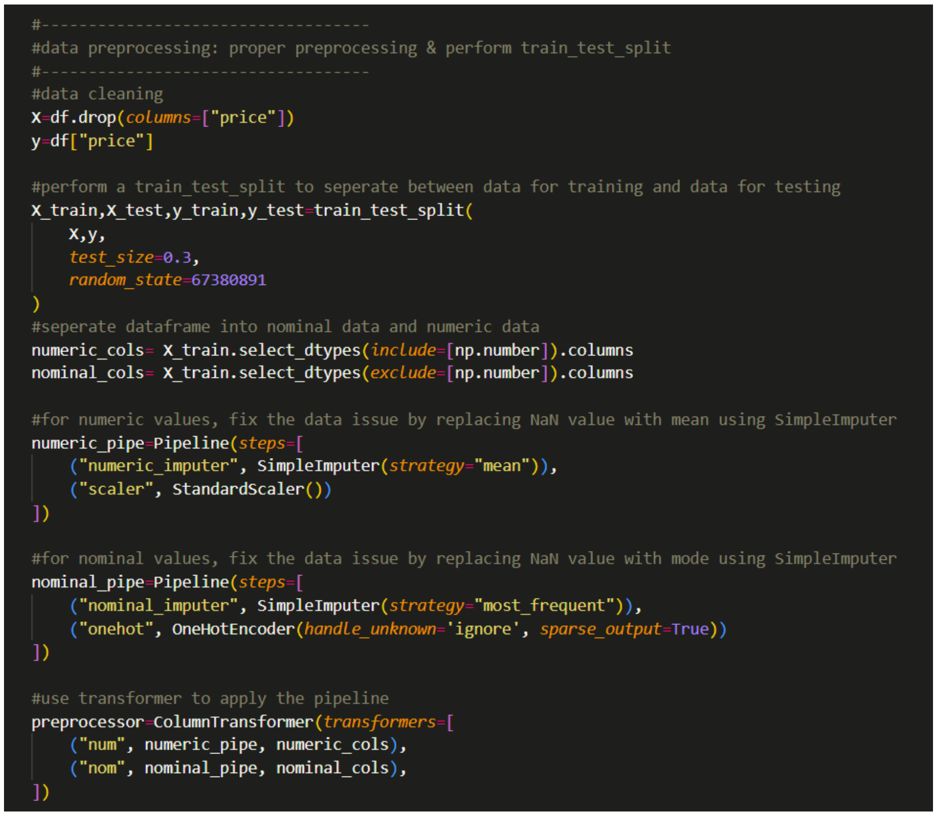
   SimpleImputer(strategy="most_frequent") ⟶ OneHotEncoder(handle_unknown='ignore')
3. Combined using a **ColumnTransformer** to process both numeric and categorical columns simultaneously.

3.2 perform a proper train-test-split

Performed a **70/30 split** with

random_state = 67380891 — the first 8 digits of the student ID.

The same preprocessor was applied to both training and test sets to prevent data leakage.



```python
#-----------------------------------
#data preprocessing: proper preprocessing & perform train_test_split
#-----------------------------------
#data cleaning
X=df.drop(columns=["price"])
y=df["price"]

#perform a train_test_split to seperate between data for training and data for testing
X_train,X_test,y_train,y_test=train_test_split(
    X,y,
    test_size=0.3,
    random_state=67380891
)
#seperate dataframe into nominal data and numeric data
numeric_cols= X_train.select_dtypes(include=[np.number]).columns
nominal_cols= X_train.select_dtypes(exclude=[np.number]).columns

#for numeric values, fix the data issue by replacing NaN value with mean using SimpleImputer
numeric_pipe=Pipeline(steps=[
    ("numeric_imputer", SimpleImputer(strategy="mean")),
    ("scaler", StandardScaler())
])

#for nominal values, fix the data issue by replacing NaN value with mode using SimpleImputer
nominal_pipe=Pipeline(steps=[
    ("nominal_imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown='ignore', sparse_output=True))
])

#use transformer to apply the pipeline
preprocessor=ColumnTransformer(transformers=[
    ("num", numeric_pipe, numeric_cols),
    ("nom", nominal_pipe, nominal_cols),
])
```

4 Model Building

Two models were trained using the same preprocessed features for efficiency comparison

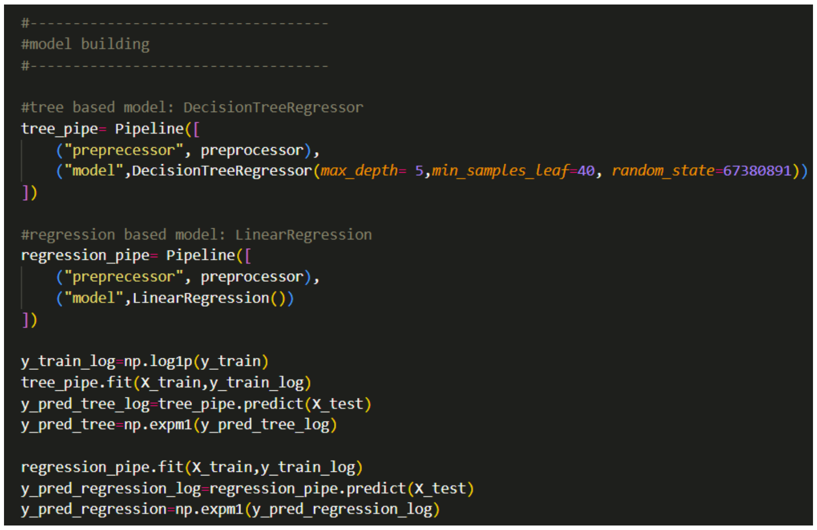
4.1 DecisionTreeRegressor (Tree-Based)

- **Model:** DecisionTreeRegressor(min_samples_leaf = 40, random_state = 67380891)
- **Target:** trained on log1p(price) to stabilize variance and improve prediction

4.2 LinearRegression (Regression-Based)

- **Model:** LinearRegression()
- **Target:** trained on log1p(price) to stabilise variance and improve prediction

Both models were integrated into **pipelines** for end-to-end reproducibility.



```
#-----------------------------------
#model building
#-----------------------------------

#tree based model: DecisionTreeRegressor
tree_pipe= Pipeline([
    ("preprecessor", preprocessor),
    ("model",DecisionTreeRegressor(max_depth= 5,min_samples_leaf=40, random_state=67380891))
])

#regression based model: LinearRegression
regression_pipe= Pipeline([
    ("preprecessor", preprocessor),
    ("model",LinearRegression())
])

y_train_log=np.log1p(y_train)
tree_pipe.fit(X_train,y_train_log)
y_pred_tree_log=tree_pipe.predict(X_test)
y_pred_tree=np.expm1(y_pred_tree_log)

regression_pipe.fit(X_train,y_train_log)
y_pred_regression_log=regression_pipe.predict(X_test)
y_pred_regression=np.expm1(y_pred_regression_log)
```

5 Evaluation and Results

5.1 Performance Metrics (on Test Set)

Model	Mean Squared Error (MSE)	R² Score
DecisionTreeRegressor	2261.09	0.488
LinearRegression	2837.459	0.357

5.2 Interpretation:

- MSE (Mean Squared Error) measures how far, on average, predictions deviate from actual prices. Therefore, lower MSE is better. (closer to the actual price)
- r2 (Coefficient of Determination) measures how much variance in price the model explains. Therefore, higher r2 is better. (more similar to the actual price)

5.3 Interesting findings:

- The results confirm that **non-linear tree models outperform simple linear regressors** on this data.
- Categorical and spatial effects make linear assumptions less suitable.
- Using the logarithmic target helped both models achieve stable learning.

5.4 Conclusion

It can be concluded that The DecisionTreeRegressor performs better (r2: 0.488>0.357) because Airbnb prices depend on nonlinear and categorical factors, which linear regression cannot keep up effectively. Still, both models show room for improvement — consider ensemble methods and refined preprocessing to push r2 beyond 0.6.