Header

PREDICTING APARTMENT RENTAL PRICES USING LOCATION AND AMENITIES

The data set is loaded from UCI Machine learning Repository. Libraries like pandas, numpy, matplotlib, seaborn, sklearn, xgboost and joblib sre used in this project.

Introduction to the Project

Context: Real estate market usually depend on the pricing strategy. With accurate rent or price prediction owners can set competitive prices for their property.

Problems faced: Rent or price differs mainly with location followed by size of the property and amenities provided which makes predicting the pricea bit hard,

Goal of the Project

objective: to built a Machine Learning Model to predict rental prices based on location and amenities.

Key metrics: RMSE(root mean square error) to measure prediction error, MAE(mean absolute error) torepresent the average prediction error and R^2 score to show how well the model explains the variance in the data provided.

Data Story

Data Source: UCI ML Repository - Apartment for Rent Classified dataset. URL: https://archive.ics.uci.edu/dataset/555/apartment+for+rent+classified Data Type: Mixed Datatypes(both categorical and numerical value present). Size: 10000 instances with 21 features. Important Columns: price, location, square feet, amenities and bedrooms.

Importing the Data

In [1]:

```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in c:\users\user\anaconda3\lib\site-packages (2.1.2)
Requirement already satisfied: numpy in c:\users\user\anaconda3\lib\site-packages (from xgboost) (1.26.4)
Requirement already satisfied: scipy in c:\users\user\anaconda3\lib\site-packages (from xgboost) (1.11.4)
```

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from xgboost import XGBRegressor
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.feature_selection import RFE
from sklearn.pipeline import Pipeline
import joblib
from sklearn.linear_model import Lasso
from sklearn.model_selection import RandomizedSearchCV
```

In [3]:

```
data = pd.read_csv(
    r"D:\Digital marketing\apartments_for_rent_classified_10K.csv",
    encoding='latin1',
    on_bad_lines='skip',
    sep=';'
)
```

Describe the Dataset

In [4]:
```
data.info()
data.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 22 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             10000 non-null  int64
 1   category       10000 non-null  object
 2   title          10000 non-null  object
 3   body           10000 non-null  object
 4   amenities      6451 non-null   object
 5   bathrooms      9966 non-null   float64
 6   bedrooms       9993 non-null   float64
 7   currency       10000 non-null  object
 8   fee            10000 non-null  object
 9   has_photo      10000 non-null  object
 10  pets_allowed   5837 non-null   object
 11  price          10000 non-null  int64
 12  price_display  10000 non-null  object
 13  price_type     10000 non-null  object
 14  square_feet    10000 non-null  int64
 15  address        6673 non-null   object
 16  cityname       9923 non-null   object
 17  state          9923 non-null   object
 18  latitude       9990 non-null   float64
 19  longitude      9990 non-null   float64
 20  source         10000 non-null  object
 21  time           10000 non-null  int64
dtypes: float64(4), int64(4), object(14)
memory usage: 1.7+ MB
```

Out[4]:

	id	bathrooms	bedrooms	price	square_feet	latitude	longitude	time
count	1.000000e+04	9966.000000	9993.000000	10000.000000	10000.000000	9990.000000	9990.000000	1.000000e+04
mean	5.623396e+09	1.380544	1.744021	1486.277500	945.810500	37.695162	-94.652247	1.574891e+09
std	7.021025e+07	0.615410	0.942354	1076.507968	655.755736	5.495851	15.759805	3.762395e+06
min	5.508654e+09	1.000000	0.000000	200.000000	101.000000	21.315500	-158.022100	1.568744e+09
25%	5.509248e+09	1.000000	1.000000	949.000000	649.000000	33.679850	-101.301700	1.568781e+09
50%	5.668610e+09	1.000000	2.000000	1270.000000	802.000000	38.809800	-93.651600	1.577358e+09
75%	5.668626e+09	2.000000	2.000000	1695.000000	1100.000000	41.349800	-82.209975	1.577359e+09
max	5.668663e+09	8.500000	9.000000	52500.000000	40000.000000	61.594000	-70.191600	1.577362e+09

Insights Explanation

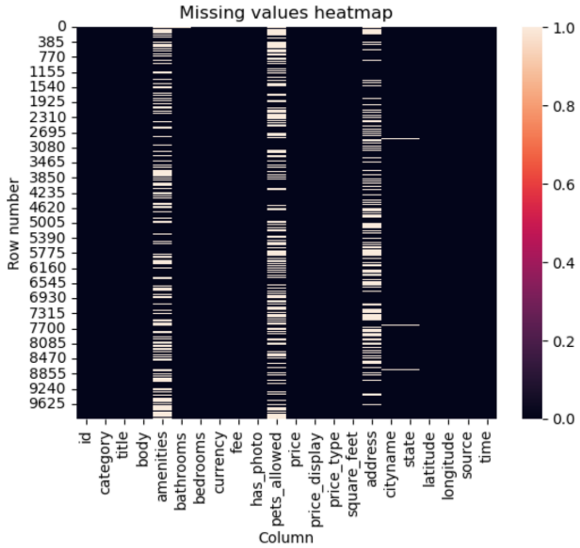
The dataset contains both numerical and categorical columns. Some columns have missing values that will be addressed through imputation. our target variable is price which is right skewed.Data will be cleaned to be sure of no duplicate values. to get fair model training features will be standardized.

Check for Null Values

In [5]:
```
print(data.isnull().sum())

sns.heatmap(data.isnull())
plt.ylabel('Row number')
plt.xlabel('Column')
plt.title('Missing values heatmap')
plt.show()
```

```
id                  0
category            0
title               0
body                0
amenities        3549
bathrooms          34
bedrooms            7
currency            0
fee                 0
has_photo           0
pets_allowed     4163
price               0
price_display       0
price_type          0
square_feet         0
address          3327
cityname           77
state              77
latitude           10
longitude          10
source              0
time                0
dtype: int64
```



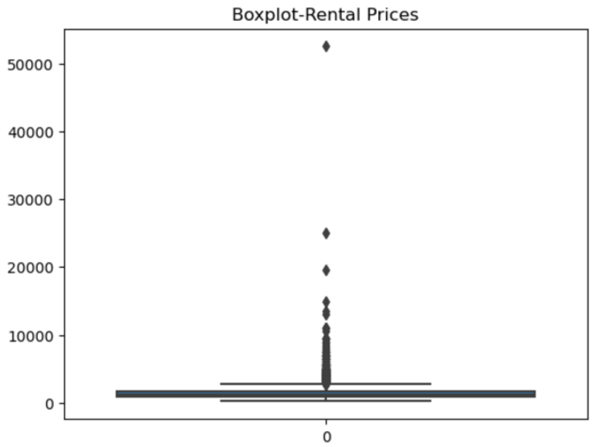
Handling Duplicates

In [6]:
```
D = data.duplicated()
print(D)
```

```
0       False
1       False
2       False
3       False
4       False
        ...
9995    False
9996    False
9997    False
9998    False
9999    False
Length: 10000, dtype: bool
```

Outlier Detection (Boxplot Check)

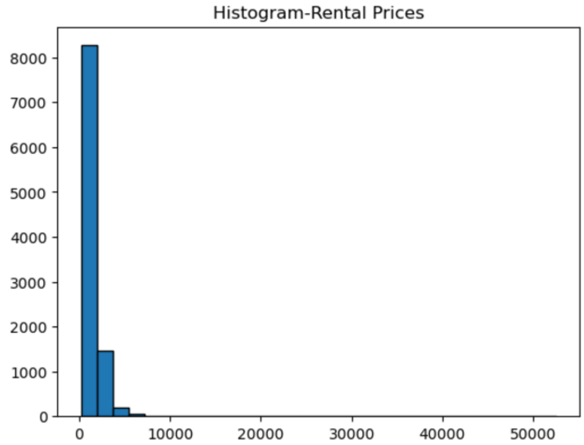
In [7]:
```
plt.figure()
sns.boxplot(data['price'])
plt.title("Boxplot-Rental Prices")
plt.show()
```



Skewness Check

In [8]:
```
plt.hist(data['price'], bins=30, edgecolor='k')
plt.title("Histogram-Rental Prices")
plt.show()

SV = data['price'].skew()
print(f"{SV}")
```




```
14.367517151261529
```

Skewness Correction

In [9]:
```
data['price'] = data['price'].apply(lambda x: np.log(x + 1))
```

In [10]:
```
print("New Skewness:", data['price'].skew())
```

```
New Skewness: 0.6987609485450521
```

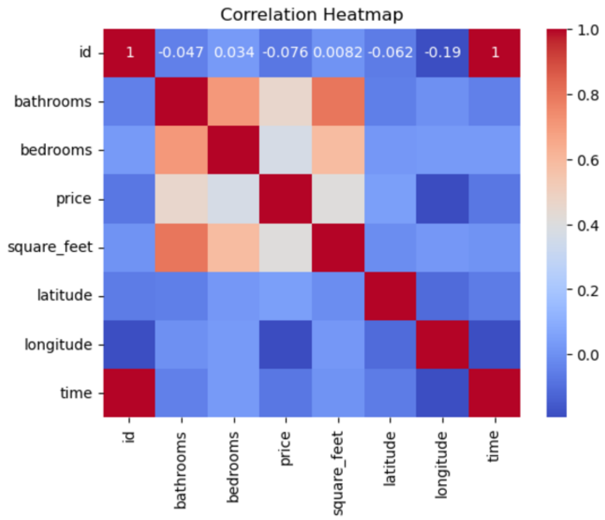
Visualization

Correlation Heatmap

In [11]:
```
print(data.dtypes)
numeric_data = data.select_dtypes(include=['number'])

sns.heatmap(numeric_data.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```

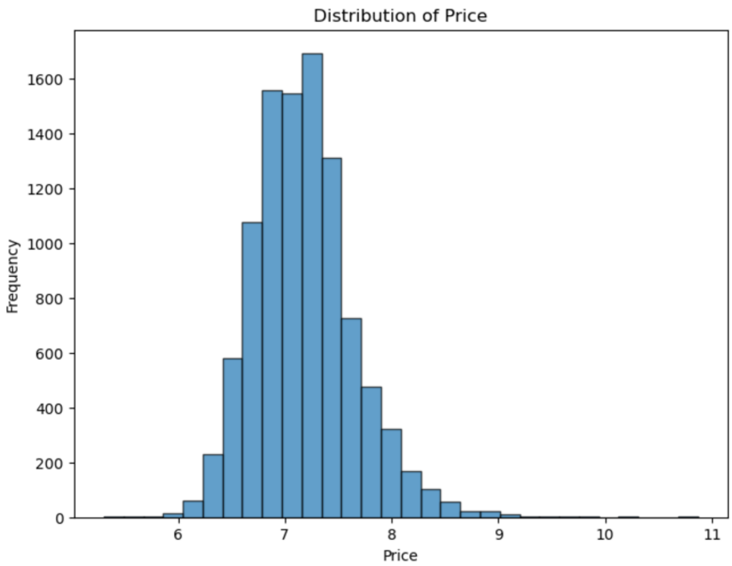
```
id                 int64
category          object
title             object
body              object
amenities         object
bathrooms        float64
bedrooms         float64
currency          object
fee               object
has_photo         object
pets_allowed      object
price            float64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



Higher the size higher the price which can be seen from square_feet and price moderate positive correlation. The correlation heatmap suggests that square_feet, bedrooms and bathrooms are important factor to determine prices.

Histogram for price

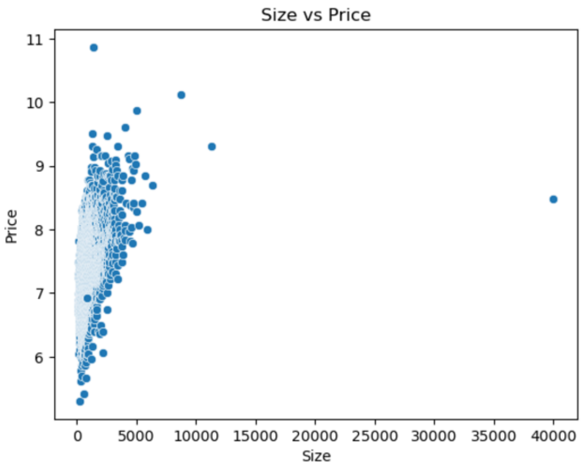
In [14]:
```
plt.figure(figsize=(8, 6))
plt.hist(data['price'], bins=30, edgecolor='k', alpha=0.7)
plt.title("Distribution of Price")
plt.xlabel("Price")
plt.ylabel("Frequency")
plt.show()
```



Histogram is moderately right-skewed showing that there are more lower-priced apartments. Most of apartments in the dataset are priced relatively low, with a smaller number of very expensive apartments.The right tail outliers indicates this.

Scatter Plot: size vs. price

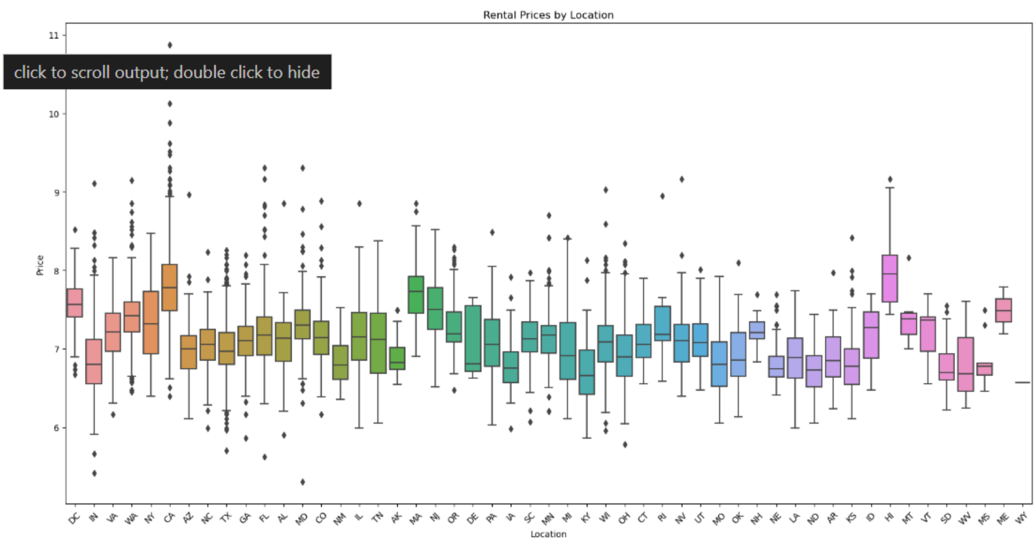
In [15]:
```
plt.figure()
sns.scatterplot(x='square_feet', y='price', data=data)
plt.title("Size vs Price")
plt.xlabel("Size")
plt.ylabel("Price")
plt.show()
```



Scatter plot shows a positive correlation between size and price,indicating that larger apartments get higher rents. However outliers and price clustering suggest that while size is an important factor, it's not the sole determinant of price as other factors(location, amenities, market demand) also influence pricing.

Boxplot: Rental Prices by location

In [16]:
```
plt.figure(figsize=(20, 10))
sns.boxplot(x='state', y='price', data=data)
plt.title("Rental Prices by Location")
plt.xlabel("Location")
plt.ylabel("Price")
plt.xticks(rotation=45)
plt.show()
```



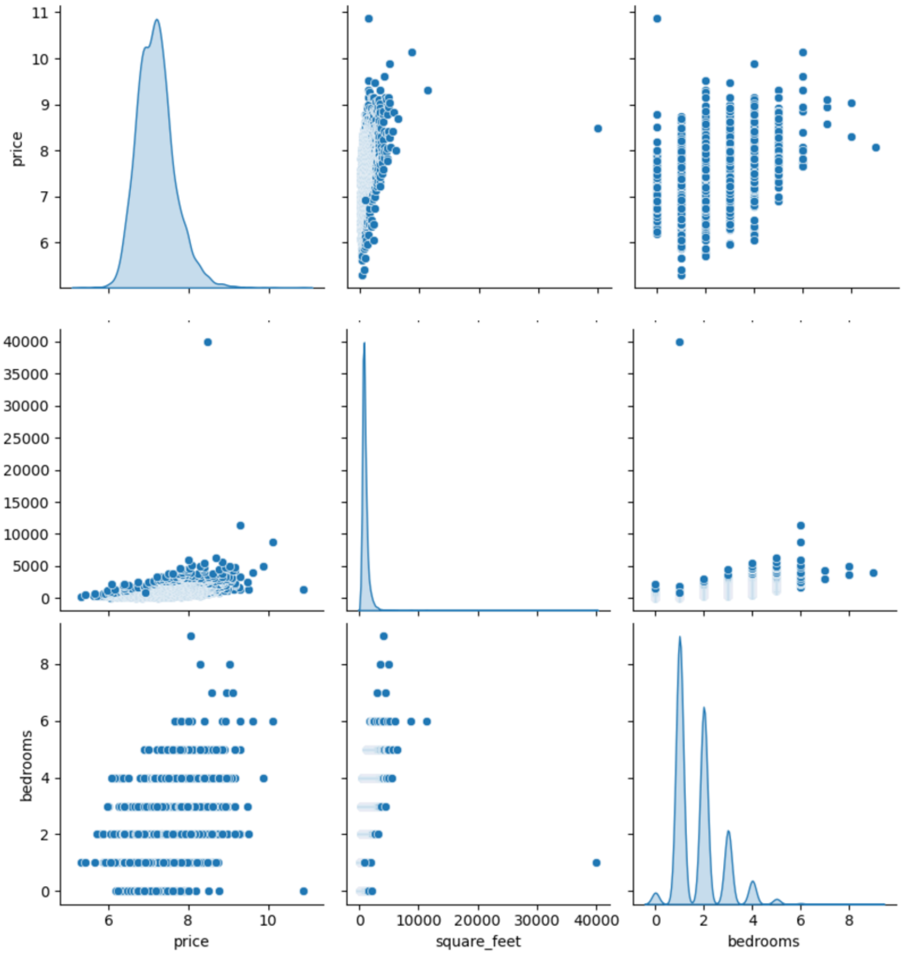
Box plot shows the difference in prices across locations. The presence of outliers suggests that apartments with extreamly high or low prices compared to the avg prices in the locations.

Pair Plot

In [17]:

```
selected_columns = ['price', 'square_feet', 'bedrooms']
sns.pairplot(data[selected_columns], diag_kind='kde', height=3)
plt.show()
```

```
C:\Users\User\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wi
ll be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
C:\Users\User\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wi
ll be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
C:\Users\User\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wi
ll be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



Pair plot vizualizes the relationship between price,square_feet and bedrooms. We con confirm there are lot of lower values as both square_feet and price are right-skewed whereas bedrooms are more evenly ditributed. Even if they have positive correlation the outliers present indicate that other factors also influence pricing.

Feature Engineering: Remove Unnecessary Columns

In [18]:
```
pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in c:\users\user\anaconda3\lib\site-packages (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in c:\users\user\anaconda3\lib\site-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.3.2 in c:\users\user\anaconda3\lib\site-packages (from scikit-learn) (1.11.4)
Requirement already satisfied: joblib>=1.1.1 in c:\users\user\anaconda3\lib\site-packages (from scikit-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\user\anaconda3\lib\site-packages (from scikit-learn) (2.2.0)
Note: you may need to restart the kernel to use updated packages.
```

In [19]:
```
unique_values = data['price_type'].unique()
print(f"Unique values: {unique_values}")
print(f"Number: {len(unique_values)}")
value_counts = data['price_type'].value_counts()
print(value_counts)

unique_values = data['category'].unique()
print(f"Unique values: {unique_values}")
print(f"Number: {len(unique_values)}")
value_counts = data['category'].value_counts()
print(value_counts)
```

```
Unique values: ['Monthly' 'Weekly' 'Monthly|Weekly']
Number: 3
price_type
Monthly           9998
Weekly               1
Monthly|Weekly       1
Name: count, dtype: int64
Unique values: ['housing/rent/apartment' 'housing/rent/home' 'housing/rent/short_term']
Number: 3
category
housing/rent/apartment     9996
housing/rent/home             2
housing/rent/short_term       2
Name: count, dtype: int64
```

Since both price_type and category columns dont have much different values it is irrelevent in calculating the price so it is removed with other irrelevant columns.

In [20]:
```
class ColumnDropper(BaseEstimator, TransformerMixin):
    def __init__(self, columns_to_drop):
        self.columns_to_drop = columns_to_drop

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return X.drop(columns=self.columns_to_drop, errors='ignore')

columns_to_remove = ['id', 'title', 'body', 'fee', 'has_photo', 'currency',
                     'price_display', 'address', 'latitude', 'longitude',
                     'source', 'time', 'price_type', 'category']

column_dropper = ColumnDropper(columns_to_remove)
data_cleaned = column_dropper.transform(data)
```

Encoding Categorical Variables

In [21]:

```
categorical_features = ['amenities', 'cityname', 'state', 'pets_allowed']
numerical_features = ['bathrooms', 'bedrooms', 'square_feet']

numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean'))
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='passthrough'
)

X = data_cleaned.drop(columns=['price'])
y = data_cleaned['price']

X_processed = preprocessor.fit_transform(X)

if hasattr(X_processed, "toarray"):
    X_processed = X_processed.toarray()

missing_values_count = np.isnan(X_processed).sum()
print("Missing values after preprocessing:", missing_values_count)

if missing_values_count > 0:
    raise ValueError("Presence of missing values dtected.")
```

```
Missing values after preprocessing: 0
```

Feature Selection

In [22]:

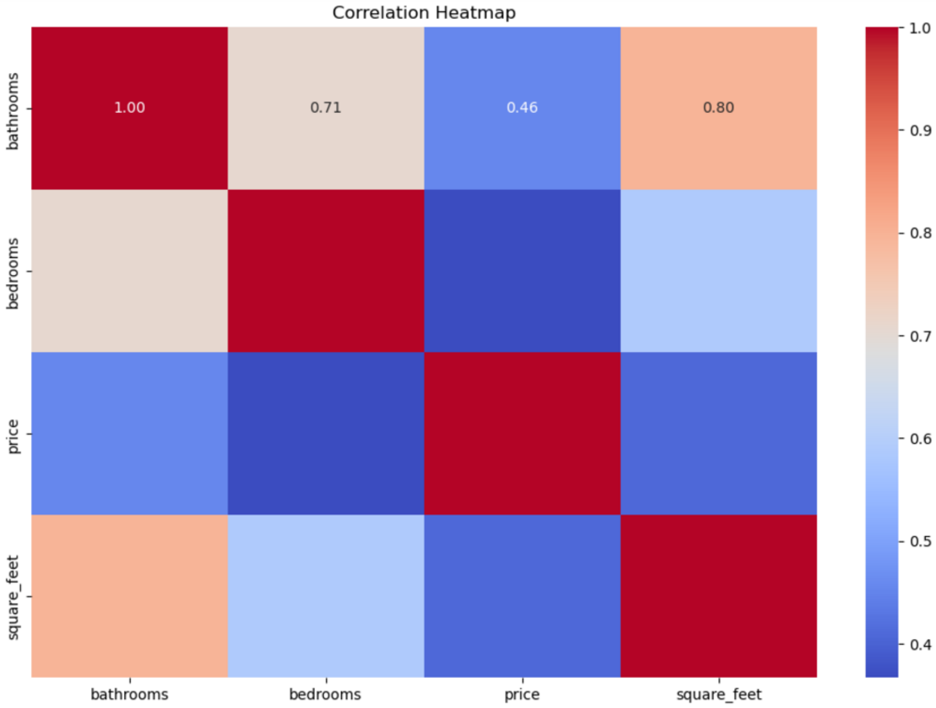
```
numerical_data = data_cleaned.select_dtypes(include=['number'])

correlation_matrix = numerical_data.corr()

plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Heatmap")
plt.show()

threshold = 0.1
high_corr_features = correlation_matrix['price'][correlation_matrix['price'].abs() > threshold].index
print("Selected Features:", list(high_corr_features))

X_selected = numerical_data[high_corr_features].drop(columns=['price'])
y_selected = numerical_data['price']
```




```
Selected Features: ['bathrooms', 'bedrooms', 'price', 'square_feet']
```

Standardizing Features

In [23]:
```python
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X_processed)

if np.isnan(X_standardized).sum() > 0:
    raise ValueError("Data contains NaNs. Check steps again.")
```

Try Different Models

In [24]:
```python
models = {
    "Linear Regression": LinearRegression(),
    "Decision Tree": DecisionTreeRegressor(),
    "Random Forest": RandomForestRegressor(),
    "Support Vector Regressor": SVR(),
    "XGBoost": XGBRegressor()
}

for name, model in models.items():
    model.fit(X_standardized, y_selected)
    y_pred = model.predict(X_standardized)
    print(f"{name}:")
    print(f"  RMSE: {np.sqrt(mean_squared_error(y_selected, y_pred))}")
    print(f"  MAE: {mean_absolute_error(y_selected, y_pred)}")
    print(f"  R2 Score: {r2_score(y_selected, y_pred)}")
    print("-" * 30)
```

```
Linear Regression:
  RMSE: 0.24461916089380112
  MAE: 0.16510631845452964
  R2 Score: 0.7276356878100285
------------------------------
Decision Tree:
  RMSE: 0.022499017147028306
  MAE: 0.003885599722483923
  R2 Score: 0.9976959263848145
------------------------------
Random Forest:
  RMSE: 0.09186333722938293
  MAE: 0.06540095150434891
  R2 Score: 0.9615891697701993
------------------------------
Support Vector Regressor:
  RMSE: 0.18514252716441468
  MAE: 0.13070843673986596
  R2 Score: 0.8439795118870551
------------------------------
XGBoost:
  RMSE: 0.19602992940592057
  MAE: 0.14931837794306638
  R2 Score: 0.825090247867816
------------------------------
```

Train ()and Test Accuracy()

In [25]:

```
X_train, X_test, y_train, y_test = train_test_split(X_standardized, y_selected, test_size=0.2, random_state=42)

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)
    print(f"{name}:")
    print(f"  Train RMSE: {np.sqrt(mean_squared_error(y_train, y_pred_train))}")
    print(f"  Test RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_test))}")
    print(f"  Train R2: {r2_score(y_train, y_pred_train)}")
    print(f"  Test R2: {r2_score(y_test, y_pred_test)}")
    print("-" * 30)
```

```
Linear Regression:
  Train RMSE: 0.3219132035998348
  Test RMSE: 154360407463170.2
  Train R2: 0.5312019244418176
  Test R2: -1.1128376653439392e+29
------------------------------
Decision Tree:
  Train RMSE: 0.021757969286813374
  Test RMSE: 0.29950233453525554
  Train R2: 0.9978583674506779
  Test R2: 0.5810517238810543
------------------------------
Random Forest:
  Train RMSE: 0.09469745334828744
  Test RMSE: 0.23898730554845724
  Train R2: 0.9594318839558561
  Test R2: 0.7332467928924569
------------------------------
Support Vector Regressor:
  Train RMSE: 0.1855860010556457
  Test RMSE: 0.3052190984047705
  Train R2: 0.8441888016657069
  Test R2: 0.5649056996701437
------------------------------
XGBoost:
  Train RMSE: 0.19590295440694358
  Test RMSE: 0.23221790423376576
  Train R2: 0.8263838140450358
  Test R2: 0.7481445307724255
------------------------------
```

Handling Overfitting

In [26]:

```
lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train, y_train)
y_pred_lasso = lasso_model.predict(X_test)
print("Lasso Regression:")
print(f"  RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_lasso))}")
print(f"  R2 Score: {r2_score(y_test, y_pred_lasso)}")
```

```
Lasso Regression:
  RMSE: 0.3821589717287097
  R2 Score: 0.3178999155463539
```

Hyperparameter Tuning and Evaluation

In [27]:
```
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
}

randomized_search = RandomizedSearchCV(
    estimator=RandomForestRegressor(random_state=42),
    param_distributions=param_grid,
    n_iter=10,
    cv=3,
    scoring='neg_mean_squared_error',
    n_jobs=-1,
    random_state=42
)

randomized_search.fit(X_train, y_train)

print("Best Parameters:", randomized_search.best_params_)
print("Best RMSE:", np.sqrt(-randomized_search.best_score_))
```

```
Best Parameters: {'n_estimators': 200, 'min_samples_split': 10, 'max_depth': None}
Best RMSE: 0.25664463646216806
```

Save the Best Model with Pipeline

In [28]:
```
best_model = randomized_search.best_estimator_
joblib.dump(best_model, 'best_rental_price_model.pkl')
```

Out[28]:
```
['best_rental_price_model.pkl']
```

Insights of the above steps from feature Engineering

Feature Engineering: Remove Unnecessary Columns

This step simplifies the dataset and avoids noise in the model training process.Irrelevant features can bias in model performance, and their removal focuses the model on important features

Encoding Categorical Variables

Missing values were imputed using the most frequent value (mode) for categorical features.Numerical features were imputed using the mean.Encoding categorical variables converts them into numerical form, enabling the model to process them effectively. Imputation ensures no missing values remain, preventing model errors during training.

Feature Selection Using Correlation Heatmap

Irrelevant features can cause bias in model performance, and their removal focuses the model on important features. Examined the correlation between numerical features and the target variable (price) using a heatmap.Highly correlated features to price are critical for prediction, while low-correlation features add limited value and can be excluded.

Standardizing Features

Applied StandardScaler to normalize numerical features, ensured all features are on the same scale.Standardization ensures that features with larger ranges don't dominate smaller-ranged features.

Trying Different Models

Trained and evaluated models like Linear Regression, Decision Trees, Random Forest, SVR, and XGBoost on standardized data. Metrics like RMSE, MAE, and R^2 Score were computed.Random Forest and XGBoost performed well with high R^2 scores, demonstrating their ability to capture feature interactions and nonlinear relationships.Simpler models like Linear Regression and Decision Trees had relatively lower performance due to the dataset's complexit

Train and Test Accuracy Evaluation

Split the dataset into training and testing sets (80%-20%).Evaluated the train and test performance for each model to check for overfitting and underfitting.Random Forest showed good generalization with comparable train and test scores.Linear Regression overfit the training data and performed poorly on the test data, indicating it's unsuitable for this dataset.

Handling Overfitting Using Lasso Regularization

Applied Lasso Regularization to mitigate overfitting by penalizing less important features.Lasso reduced model complexity without significantly impacting accuracy.Regularization techniques like Lasso can improve model generalization, especially for simpler models prone to overfitting.

Hyperparameter Tuning with RandomizedSearchCV

Used RandomizedSearchCV to find the best hyperparameters for the Random Forest model.Evaluated 10 random combinations out of the parameter grid using 3-fold cross-validation.Hyperparameter tuning enhanced Random Forest performance by selecting the optimal configuration without exhaustive computation.RandomizedSearchCV significantly reduced computation time compared to GridSearchCV.

Saving the Best Model

Saved the best-tuned Random Forest model to a file using joblib for deployment or future use.Saving the model ensures reproducibility and allows the best model to be reused for predictions without retraining.

Final Insights

Features like square_feet, bathrooms, bedrooms, and location-specific categorical variables (state, cityname) were identified as critical predictors. Random Forest outperformed other models with strong RMSE and R^2 scores, showcasing its robustness for this dataset. Regularization and hyperparameter tuning significantly improved model performance and generalization. RandomizedSearchCV reduced computational cost while effectively optimizing the model.

Conclusion

The best results for predicting apartment prices was the Random Forest model with hyperparameter tuning . Important features include square_feet, bedrooms, bathrooms, and location-specific features.

Limitations

Dataset have incomplete or imbalanced data with null values and incorrect columns, that impact accuracy. Only numerical and basic categorical features were considered after so after preprocessing and dropping the unwanted rows.

Future Work

Expand the dataset to include more locations or additional features. Should add seasonal benefits, more amenities, and access to public transports, then what kind of tourism spot to get more accurate pricing. should also explore deep learning models for better accuracy.