Predictive Classification Modeling of Real Estate Pricing Tiers

Laxmi Sulakshana Rapolu, Daniel Shifrin, and Outhai Xayavongsa

University of San Diego

AAI-501 Intro to Artificial Intelligence

Professor David Friesen

April 15, 2024

Predictive Classification Modeling of Real Estate Pricing Tiers

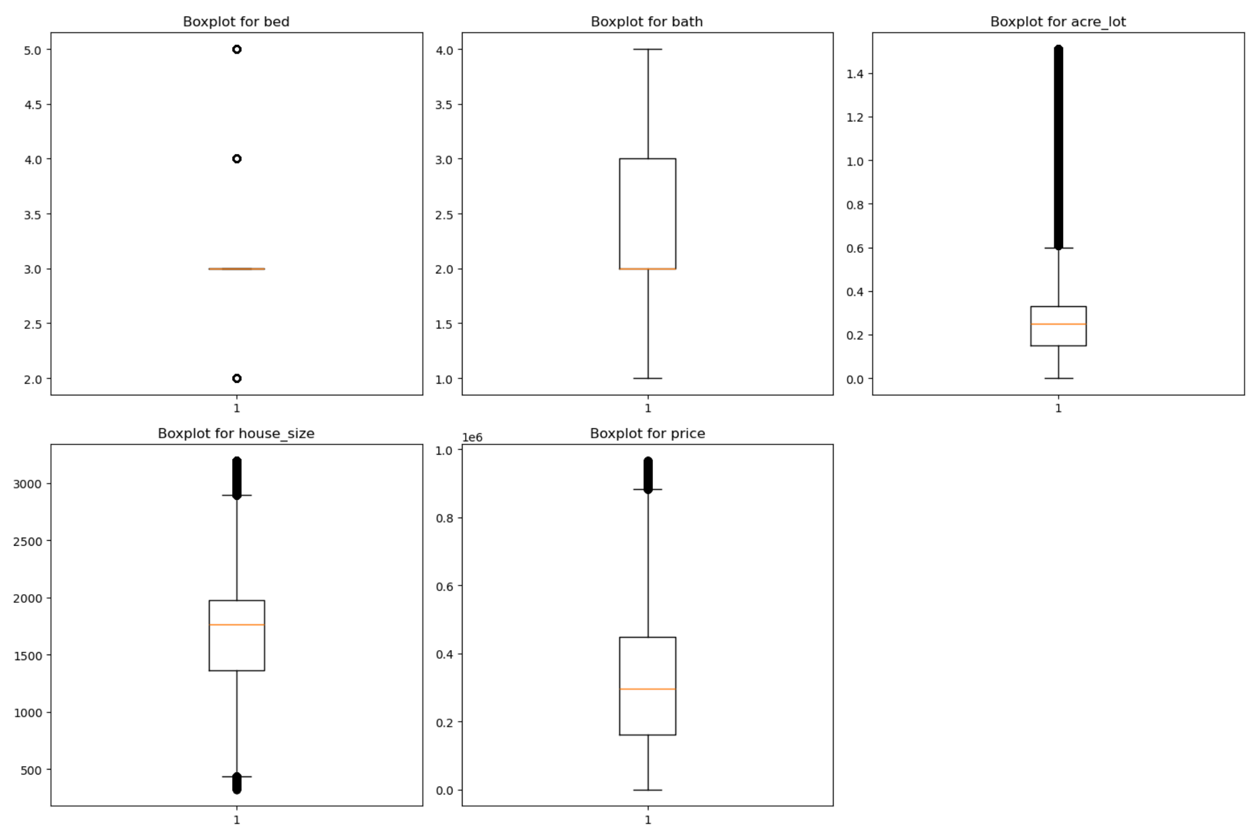
Over the past fifty years, the US housing market has experienced profound changes, caused by economic shifts, changes in policy, and evolving societal norms (Rothstein et al., 2024). The recent pandemic underscored the value of personal living spaces, prompting 22% of U.S. adults to seek more spacious accommodations, leading to a homeownership rate spike to 67.9% (Sweet, 2020). The market faces challenges like high home prices and elevated mortgage rates, complicating affordability (Rothstein et al., 2024).

Our project aims to demystify the real estate landscape by analyzing a vast dataset with over 2 million listings to classify properties into pricing tiers (Skakib, 2024). Employing advanced analytics, we segment properties from Starter Homes to Luxury Estates, offering insights to navigate the market's complexities. This research addresses current market challenges and aims to provide a clearer path for stakeholders in this dynamic real estate environment.

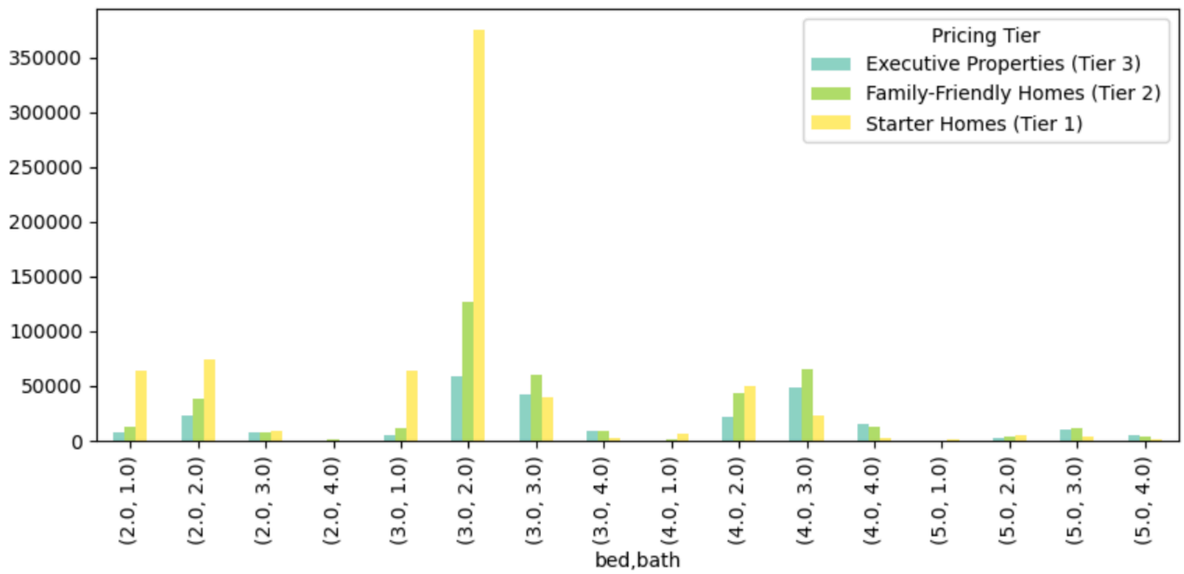
In our project, we meticulously cleaned the dataset to ensure its integrity for analysis. We removed irrelevant columns ('brokered_by', 'status', 'prev_sold_date', 'street', 'zip_code') and handled missing values by imputing medians for numerical ('bed', 'acre_lot', 'bath', 'house_size', 'price') and modes for categorical ('city', 'state') columns. Irrelevant states were filtered out, and duplicate rows were removed. Outliers in numerical columns were detected using boxplots and the interquartile range (IQR) method, and subsequently eliminated. These steps ensured a clean and reliable dataset, crucial for our real estate pricing tier classification project.

The dataset with outliers includes about 2 million properties, with an average price of around $537,725, and a wide range of prices. Most properties have 3 bedrooms and 2 bathrooms, and the most common city and state are Houston and Florida, respectively. The most common pricing tier is “Starter Homes (Tier 1)”. After removing outliers, the dataset includes about 1.38 million properties, with an average price of around $323,319. The sizes of the lot and the house are generally smaller, but Houston and Florida remain the most common city and state, and “Starter Homes (Tier 1)” remains the most common pricing tier.

The box plots illustrate the distribution of five variables: bed, bath, acre_lot, house_size, and price. They serve as valuable tools for pinpointing outliers and gaining insights into the data's distribution. Each plot shows the median, the interquartile range, and the range, providing information about the variability and skewness of the data. The bed and bath plots show that most properties have between 2 and 5 bedrooms and between 1 and 4 bathrooms, respectively. The acre_lot, house_size, and price plots are all heavily skewed to the right, indicating that most properties have a small lot size, are smaller in size, and are cheaper, respectively. However, there are a few properties that are significantly larger or more expensive.



The below graph shows the pricing tiers of different types of homes based on their bedroom and bathroom count. There seems to be an unusual spike in the price for Starter Homes with 3 bedrooms and 2 bathrooms. This could indicate a higher demand or value for such properties.



The Chi-Square test results indicate that there is a significant association between the Pricing Tier and each of the other variables (bed, bath, acre_lot, house_size, price, city, and state). In other words, the number of bedrooms, number of bathrooms, lot size, house size, price, city, and state of a property are not independent of its Pricing Tier. This suggests that the Pricing Tier of a property is likely to correlate with or be linked to these other attributes. For instance, luxury estates (associated with a higher Pricing Tier) are expected to typically feature more bedrooms, larger lot sizes, and higher prices compared to starter homes (linked with a lower Pricing Tier).

In the analysis after removing outliers, residential properties were categorized into three pricing tiers—Executive, Family-Friendly, and Starter Homes. The visual data revealed that Executive properties were the most spacious and expensive, while Starter Homes were the smallest and least expensive.

Figure 3

Pricing Tier Bar Plot without outliers

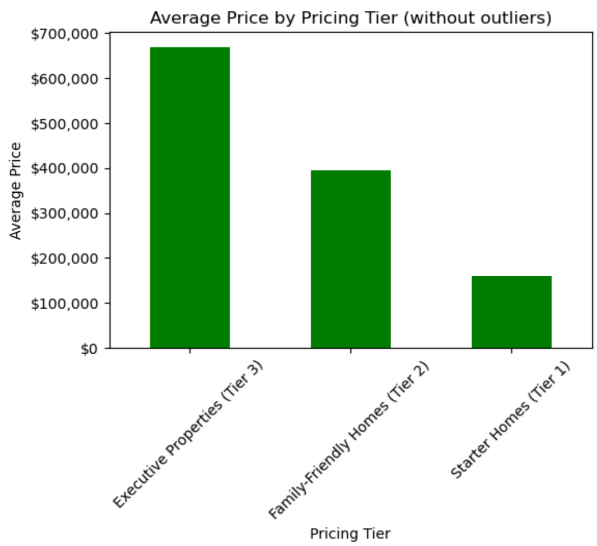
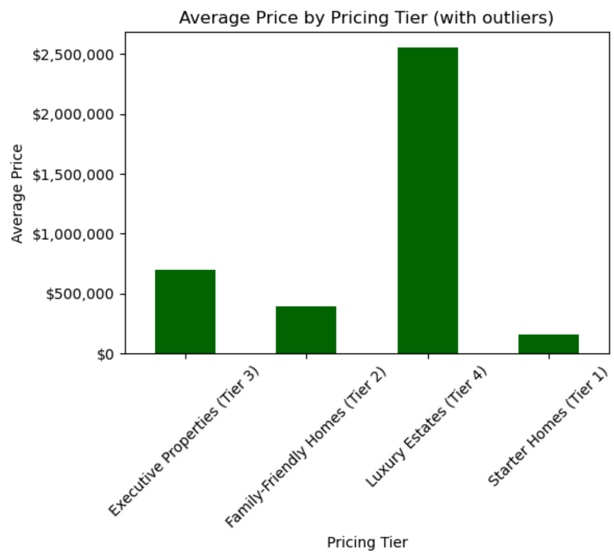


Figure 4

Pricing Tier Bar Plot with outliers

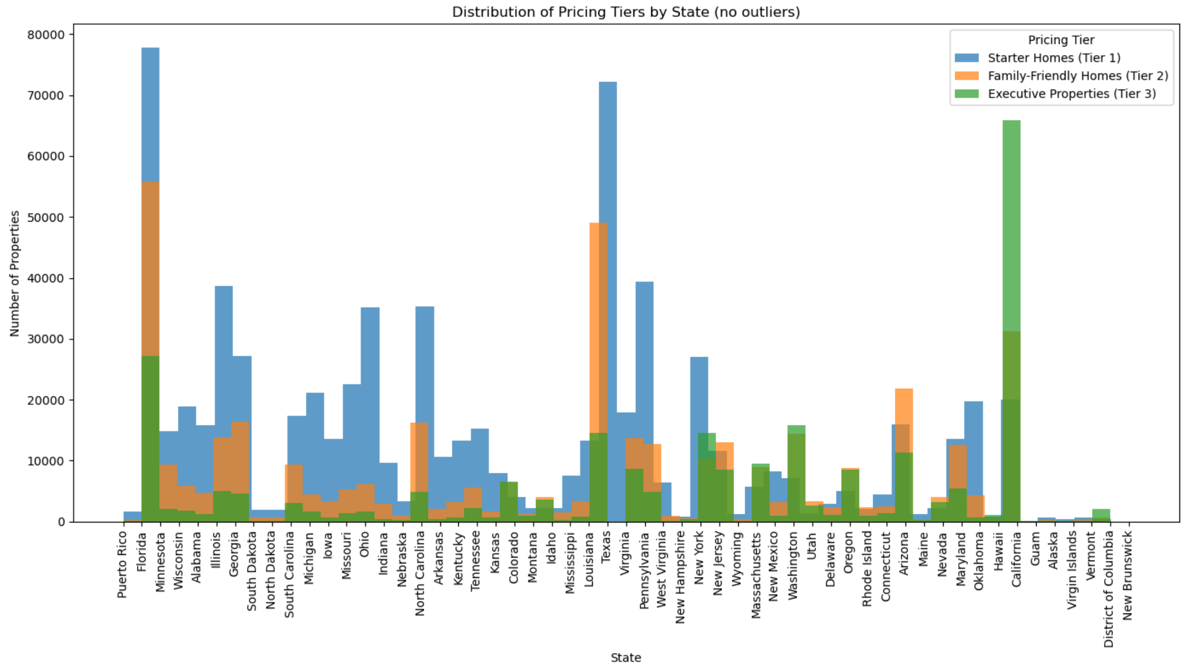


When reintegrating outliers, the Luxury Estates Tier was added. The bar charts indicated a significant jump in price, suggesting a market segment with properties that are exceptionally more expensive, thus drastically raising the average values. Comparing data across different pricing tiers reveals a pronounced stratification in the market and the disproportionate impact of high-end luxury properties on market averages.

Certain states exhibit prominent spikes, shown in Figure 3, which represent a higher concentration of properties in comparison to other states. These spikes, especially if they occur in states with larger populations or more robust housing markets, could be indicative of a thriving real estate sector. They also may point to a state's economic health or a higher demand for housing. The absence of Luxury Properties in these spikes, due to the removal of outliers from the data, suggests that the figures presented are more representative of the typical housing market, excluding the exceptionally high-end luxury segment.

Figure 5

Distribution of Property Pricing Tiers by State



Feature Selection: We conducted a correlation analysis on a preprocessed dataset, focusing solely on numeric columns after removing outliers. Utilizing Pearson correlation coefficients, a correlation matrix was computed to examine relationships between numeric variables and the target variable ('price'). Visualizing the correlation matrix through a heatmap provided a clear understanding of the strength and direction of correlations. From this analysis, we identified the top three features with the highest absolute correlation coefficients with the target variable, designated as Feature 1, Feature 2, and Feature 3. Further examination of these features revealed the extent of their relationship with the target variable.

Split Data: The dataset, featuring numerical and categorical features, includes a categorical target variable representing pricing tiers. Categorical variables underwent label encoding to convert them into numerical representations, enabling integration into machine learning algorithms. Each categorical column was encoded individually using sklearn's LabelEncoder module. Encoded categorical features were then combined with numerical features to form the final feature matrix denoted as 'X'. The dataset was split into training and testing sets using train_test_split from sklearn.model_selection, with an 80-20 ratio and a random_state of 42 for reproducibility. The training set comprised 1,107,708 samples, while the testing set contained 276,928 samples. This meticulous preparation and splitting of the dataset are crucial for constructing proficient machine-learning models, ensuring diverse data for model learning and generalization to new instances. These structured datasets serve as the foundation for subsequent model training and evaluation, facilitating the development of robust machine-learning solutions.

Random Forest Classifiers (RFC), which consist of many decision trees to create a more accurate output, outshine **Gradient Boosting Classifiers** (GBC) that build trees in order while correcting errors to minimize loss, especially in predicting real estate pricing tiers. The RFC, with its 100 decision trees, achieves a notable accuracy of about 76% on the test data, outperforming the GBC's 68%. This superiority is further evidenced in the precision, recall, and F1-score metrics across various pricing tiers—Starter Homes, Family-Friendly Homes, and Executive Properties—where the RFC consistently delivers higher values than the GBC. Additionally, the RFC's AUC-ROC value of approximately 0.894, compared to the GBC's 0.830, indicates its superior capability in discerning between different real estate pricing tiers. These performance metrics suggest that the RFC, with its ensemble approach for enhancing accuracy and stability, is more suited for real estate pricing tier prediction tasks, offering valuable insights for deciding on the real estate market.

We evaluated a **K-Nearest Neighbors (KNN)** classifier on a dataset featuring properties categorized into different pricing tiers. Using 5 neighbors, the classifier achieved an accuracy of approximately 70.24% on the testing set, indicating its capability to classify properties accurately. A detailed classification report revealed the classifier's performance across pricing tiers. For Executive Properties (Tier 3), the classifier exhibited a moderate ability with precision, recall, and F1-score of 0.59, 0.62, and 0.61 respectively. Similarly, for Family-Friendly Homes (Tier 2), performance metrics stood at 0.58, 0.57, and 0.57 for precision, recall, and F1-score. Notably, the classifier demonstrated the highest effectiveness in identifying Starter Homes (Tier 1), with precision, recall, and F1-score values of 0.81 each. Although the KNN classifier showed moderate performance, there's potential for improvement through parameter tuning, exploring alternative algorithms, or feature engineering. Nevertheless, its ability to categorize properties into pricing tiers lays a solid foundation for decision-making processes in real estate or related domains.

The **Logistic Regression** model was trained with a maximum of 1000 iterations to ensure convergence, achieving an accuracy of approximately 57.77% on the testing set. The classification report revealed insights into the model's performance across different pricing tiers. For Executive Properties (Tier 3), the model's precision, recall, and F1-score were 0.45, 0.18, and 0.26 respectively, with challenges in correctly identifying Tier 3 properties. Family-Friendly Homes (Tier 2) showed similar performance, with precision, recall, and F1-score of 0.44, 0.31, and 0.37 respectively, indicating room for improvement. However, the model excelled in classifying Starter Homes (Tier 1) with the highest precision, recall, and F1-score values of 0.63, 0.87, and 0.73 respectively, highlighting its effectiveness in identifying Tier 1 properties.

The **Naïve Bayes** model, with its foundation in Bayes' theorem, is a commendable choice for classification tasks due to its simplicity, efficiency, and surprising performance even with the strong independence assumptions it makes. In the context of real estate price tier classification, the model's ease of implementation and computational efficiency are particularly advantageous when dealing with high-dimensional datasets. The reported accuracy of 60% indicates a moderate level of predictive power, which, while not exceptionally high, does suggest that the model has learned to some extent from the data. However, the precision and recall figures for the individual classes indicate room for improvement, especially in terms of the model's ability to correctly identify the middle and high price tiers (Tier 2 and Tier 3). The AUC-ROC score of 0.7163 is a more encouraging sign, suggesting that the model can reasonably distinguish between the classes.

The implementation of a **Deep Learning Multilayer Perceptron (MLP)** model in the real estate price tier classification project is a strategic approach to tackle the complexity inherent in real estate data. The MLP's ability to discern intricate patterns and its adaptability in architecture design is well-suited for the high-dimensional nature of real estate datasets. The implementation involved preprocessing the data, defining the model, compiling it, and then training it. After training, the model was evaluated and showed decent performance with an accuracy of approximately 73.25%. However, there were areas for improvement, particularly in terms of precision, F1 score, and AUC-ROC score. While the accuracy is relatively high, the precision, recall, F1 score, and particularly the ROC-AUC score suggest that the model's performance could be improved. The ROC-AUC score, below 0.5, implies that the model may perform no better than random chance in certain classifications.

In summary, our project explored predictive classification modeling for real estate pricing tiers using various machine learning algorithms. Through rigorous data preprocessing and feature selection, we ensured dataset integrity and relevance. Among the models tested, the Random Forest Classifier emerged as the most effective, offering high accuracy and precision across pricing tiers. The K-Nearest Neighbors classifier showed moderate performance, while Logistic Regression exhibited challenges in identifying certain tiers. Additionally, Naïve Bayes and Deep Learning Multilayer Perceptron models were explored, showing promise but requiring further optimization. Overall, our project provides valuable insights for real estate decision-making, highlighting the potential of machine learning in navigating the complexities of property pricing. Continued research and refinement hold the key to further enhancing model performance and driving innovation in the real estate industry.

References

Marx, S. (2024, April 2). *Majority of older homeowners plan to stay in their home as they age: Redfin.* HousingWire. Retrieved April 6, 2024, from https://www.housingwire.com/articles/majority-of-older-homeowners-plan-to-stay-in-their-home-as-they-age-redfin/#:~:text=Furthermore%2C%20nearly%2080%25%20of%20boomers,bought%20a%20new%20home%20today

Rothstein, R., Basile, C., & Jennings, C. (2024, April 2). *Housing Market Predictions For 2024: When Will Home Prices Be Affordable Again?* Forbes Advisor. Retrieved April 3, 24, from https://www.forbes.com/advisor/mortgages/real-estate/housing-market-predictions/

Shakib, A. S. (2024). USA Real Estate Dataset. Kaggle. Retrieved March 31, 2024, from https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset/data?select=realtor-data.zip.csv

Sweet, J. (2020, October 23). *History of the housing market over the last 50 years*. Stacker. Retrieved April 3, 2024, from https://stacker.com/money/history-housing-market-over-last-50-years

"Typical U.S. Home Values by State." (2024). Datawrapper. Retrieved April 6, 2024, from https://www.datawrapper.de/_/UjLtA/

Yu, H., & Wu, J. (2016). Real Estate Price Prediction with Regression and Classification. In CS 229 Autumn 2016 Project, *CS 229 Autumn 2016 Project Final Report*. https://cs229.stanford.edu/proj2016/report/WuYu_HousingPrice_report.pdf

Banjara, B. (2023, November 5). *Guide to Build Real Estate Price Prediction Model using ML algorithms*. Medium. https://medium.com/@rubiks1200/guide-to-build-real-estate-price-prediction-model-using-ml-algorithms-254b441057bb

List of Contributions **https://github.com/oxayavongsa/aai-501-final-team-4**

Laxmi S Rapolu	***Daniel Shifrin***	***Outhai Xayavongsa***
Researched & Contributed to selecting a dataset - Car dataset	Contributed to selecting a dataset	Researched & Contributed Link for the Real Estate Data Set
Data collection from Kaggle - code	Contributed Problem Statements	Created Problem Statements
Contributed to proposal	Contributed to proposal	Team Leader & Organize Task
Contributed to selecting a problem statement	Feature Selection code, paper, and presentation	Contribution to Proposal & Submit
Organize code	Splitting Data code, paper, presentation	Define Price Tier
Remove duplicate rows - code	K-Nearest Neighbors (KNN) code, paper, & presentation	Clean & Prep Dataset - Code, Paper, Presentation
Outlier detection and removal - code	Logistic Regression code, paper, & presentation	‘Extras’ EDA Visuals pg. 37-44 of this document
Descriptive statistics and visualization - code, final paper, presentation	Conclusion for Paper & Presentation	Created Entire ReadMe File on GitHub
Inferential statistics - code, final paper, presentation	Contributed to references - Paper	Correlation Matrix Code only - Revised Feature Selection
Label encoding for target variable code	Pep 8 Review - Entire Code & Convert to PDF	Revised Data Split Code
Naive Bayes Model - code, final paper, presentation	Review written sections - APA 7 Review & Citation	Random Forest - Code, Paper, Presentation
Deep Learning (MLP) Model - code, final paper, presentation	Convert Completed PowerPoint to MP4	Gradient Boosting - Code, Paper, Presentation
Contributed to reference list - Final paper		‘Extras’ Model Summary Code pg. 55-56
Review the Entire final paper and citation in APA 7		Introduction- Paper & Presentation
License - GitHub repository		Reviewed Entire Presentation & Combined Slides
Model Training, Testing, & Metrics Comparison - presentation		Contributed to Reference List - Paper & Powerpoint
		Submit Final Deliverables & Link to YouTube

Real Estate Pricing Tier Classification

Names: Thai, Laxmi, and Daniel

Introduction

The real estate market is a dynamic and complex environment where property values fluctuate due to a myriad of factors. Our final project aims to demystify this volatility by developing a predictive model that can accurately classify properties into distinct price tiers. By analyzing a comprehensive dataset of real estate transactions, we intend to uncover patterns and indicators that influence pricing, thereby providing valuable insights for buyers, sellers, and investors alike. This endeavor not only seeks to enhance market transparency but also to empower stakeholders with a tool for informed decision-making in the real estate domain.

Dataset

The dataset comprises over 4 million entries of U.S. real estate listings, segmented by state and zip code, sourced from Realtor.com, a leading property listing platform. It encompasses a comprehensive range of data points including housing status, number of bedrooms and bathrooms, land size in acres, city, state, postal code, living space in square feet, previous sale date, and the listing or recently sold price. This rich dataset provides a granular view of the current real estate market, offering valuable insights for various analyses and applications.

Dataset URL: https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset/data

Packages

```
# Install necessary packages (if not already installed)
!pip install kaggle pandas numpy matplotlib seaborn xgboost scipy
scikit-learn

# Import required libraries
import kaggle
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import xgboost as xgb
import tensorflow as tf
from scipy.stats import chi2_contingency
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score,
mean_absolute_error, classification_report, accuracy_score,
confusion_matrix, roc_auc_score, roc_curve, auc
```

```python
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dropout

# Print a success message
print("All necessary packages are installed and imported
successfully.")
```

```
Requirement already satisfied: kaggle in
./anaconda3/lib/python3.11/site-packages (1.6.8)
Requirement already satisfied: pandas in
./anaconda3/lib/python3.11/site-packages (2.1.4)
Requirement already satisfied: numpy in
./anaconda3/lib/python3.11/site-packages (1.26.4)
Requirement already satisfied: matplotlib in
./anaconda3/lib/python3.11/site-packages (3.8.0)
Requirement already satisfied: seaborn in
./anaconda3/lib/python3.11/site-packages (0.12.2)
Requirement already satisfied: xgboost in
./anaconda3/lib/python3.11/site-packages (2.0.3)
Requirement already satisfied: scipy in
./anaconda3/lib/python3.11/site-packages (1.12.0)
Requirement already satisfied: scikit-learn in
./anaconda3/lib/python3.11/site-packages (1.2.2)
Requirement already satisfied: six>=1.10 in
./anaconda3/lib/python3.11/site-packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in
./anaconda3/lib/python3.11/site-packages (from kaggle) (2024.2.2)
Requirement already satisfied: python-dateutil in
./anaconda3/lib/python3.11/site-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in
./anaconda3/lib/python3.11/site-packages (from kaggle) (2.31.0)
Requirement already satisfied: tqdm in
./anaconda3/lib/python3.11/site-packages (from kaggle) (4.65.0)
Requirement already satisfied: python-slugify in
./anaconda3/lib/python3.11/site-packages (from kaggle) (5.0.2)
Requirement already satisfied: urllib3 in
```

```
./anaconda3/lib/python3.11/site-packages (from kaggle) (2.0.7)
Requirement already satisfied: bleach in
./anaconda3/lib/python3.11/site-packages (from kaggle) (4.1.0)
Requirement already satisfied: pytz>=2020.1 in
./anaconda3/lib/python3.11/site-packages (from pandas) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in
./anaconda3/lib/python3.11/site-packages (from pandas) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=6.2.0 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in
./anaconda3/lib/python3.11/site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: joblib>=1.1.1 in
./anaconda3/lib/python3.11/site-packages (from scikit-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
./anaconda3/lib/python3.11/site-packages (from scikit-learn) (2.2.0)
Requirement already satisfied: webencodings in
./anaconda3/lib/python3.11/site-packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in
./anaconda3/lib/python3.11/site-packages (from python-slugify->kaggle)
(1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
./anaconda3/lib/python3.11/site-packages (from requests->kaggle)
(2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
./anaconda3/lib/python3.11/site-packages (from requests->kaggle) (3.4)

2024-04-06 00:22:17.393052: I
tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow
binary is optimized to use available CPU instructions in performance-
critical operations.
To enable the following instructions: AVX2 FMA, in other operations,
rebuild TensorFlow with the appropriate compiler flags.

All necessary packages are installed and imported successfully.
```

Exploratory Data Analysis

Data collection

Pre-requisites

To download the dataset, please follow the below steps:

1. Create a Kaggle account
2. After logging in, go to your user settings page on Kaggle.
3. Click the "Create New Token" button.
4. This action will download a file named kaggle.json to your computer.
5. Copy this file to the directory 'C:\Users<your_username>.kaggle'

```
# After ensuring you've met the prerequisites, execute the following
command to download the dataset
!kaggle datasets download -d ahmedshahriarsakib/usa-real-estate-
dataset

with zipfile.ZipFile("usa-real-estate-dataset.zip","r") as zip_ref:

    # extracting content in the zipfile
    zip_ref.extractall()
```

```
usa-real-estate-dataset.zip: Skipping, found more recently modified
local copy (use --force to force download)
```

```
# Load the dataset
df = pd.read_csv('realtor-data.zip.csv')

# Inspect the data
print(df.head())
print(df.info())
```

```
   brokered_by    status     price  bed  bath  acre_lot     street  \
0     103378.0  for_sale  105000.0  3.0   2.0      0.12  1962661.0
1      52707.0  for_sale   80000.0  4.0   2.0      0.08  1902874.0
2     103379.0  for_sale   67000.0  2.0   1.0      0.15  1404990.0
3      31239.0  for_sale  145000.0  4.0   2.0      0.10  1947675.0
4      34632.0  for_sale   65000.0  6.0   2.0      0.05   331151.0

        city        state  zip_code  house_size prev_sold_date
0   Adjuntas  Puerto Rico     601.0       920.0            NaN
1   Adjuntas  Puerto Rico     601.0      1527.0            NaN
2 Juana Diaz  Puerto Rico     795.0       748.0            NaN
3      Ponce  Puerto Rico     731.0      1800.0            NaN
4   Mayaguez  Puerto Rico     680.0         NaN            NaN
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2226382 entries, 0 to 2226381
Data columns (total 12 columns):
```

```
 #   Column          Dtype
---  ------          -----
 0   brokered_by     float64
 1   status          object
 2   price           float64
 3   bed             float64
 4   bath            float64
 5   acre_lot        float64
 6   street          float64
 7   city            object
 8   state           object
 9   zip_code        float64
 10  house_size      float64
 11  prev_sold_date  object
dtypes: float64(8), object(4)
memory usage: 203.8+ MB
None
```

Data cleaning

```
# Print the list of columns in the original DataFrame before cleaning
print("Original columns:", df.columns)

# Remove columns that are irrelevant
df.drop(['brokered_by', 'status', 'prev_sold_date', 'street',
'zip_code'], axis=1, inplace=True)

# Handle missing values for numerical columns by imputing with the
median
numerical_columns = ['bed', 'bath', 'acre_lot', 'house_size', 'price']
df[numerical_columns] = df[numerical_columns].apply(lambda x:
x.fillna(x.median()), axis=0)

# Assuming 'city' and 'state' are the only categorical columns with
missing values
# Handle missing values for categorical columns by imputing with the
mode (most common value)
categorical_columns = ['city', 'state']
df[categorical_columns] = df[categorical_columns].apply(lambda x:
x.fillna(x.mode()[0]), axis=0)

# Display all unique values in the 'state' column
print("\nAll states (Unfiltered):")
print(df['state'].unique())

# Define the list of states to delete
delete_states = ['New Brunswick', 'Puerto Rico', 'Virgin Islands',
'Guam']

# Filter the DataFrame to keep only rows where the 'state' column does
```

```
not belong to the states to delete
df_cleaned = df[~df['state'].isin(delete_states)]

# Display USA states only in the 'state' column
print("\nUSA states only:")
print(df_cleaned['state'].unique())

# Recheck info
print("\nCleaned DataFrame:")
print(df_cleaned.info())
print("Cleaned columns:", df_cleaned.columns)

# Save the cleaned dataset
df_cleaned.to_csv('clean-realtor-data.zip.csv', index=False)

Original columns: Index(['brokered_by', 'status', 'price', 'bed',
'bath', 'acre_lot', 'street',
       'city', 'state', 'zip_code', 'house_size', 'prev_sold_date'],
      dtype='object')

All states (Unfiltered):
['Puerto Rico' 'Virgin Islands' 'Massachusetts' 'Connecticut'
 'New Hampshire' 'Vermont' 'New Jersey' 'New York' 'South Carolina'
 'Tennessee' 'Rhode Island' 'Virginia' 'Wyoming' 'Maine' 'Georgia'
 'Pennsylvania' 'West Virginia' 'Delaware' 'Louisiana' 'Ohio'
'California'
 'Colorado' 'Maryland' 'Missouri' 'District of Columbia' 'Wisconsin'
 'North Carolina' 'Kentucky' 'Michigan' 'Mississippi' 'Florida'
'Alabama'
 'New Brunswick' 'Texas' 'Arkansas' 'Idaho' 'Indiana' 'Illinois'
 'New Mexico' 'Iowa' 'Minnesota' 'South Dakota' 'Nebraska' 'North
Dakota'
 'Montana' 'Oklahoma' 'Kansas' 'Oregon' 'Utah' 'Nevada' 'Washington'
 'Arizona' 'Hawaii' 'Guam' 'Alaska']

USA states only:
['Massachusetts' 'Connecticut' 'New Hampshire' 'Vermont' 'New Jersey'
 'New York' 'South Carolina' 'Tennessee' 'Rhode Island' 'Virginia'
 'Wyoming' 'Maine' 'Georgia' 'Pennsylvania' 'West Virginia' 'Delaware'
 'Louisiana' 'Ohio' 'California' 'Colorado' 'Maryland' 'Missouri'
 'District of Columbia' 'Wisconsin' 'North Carolina' 'Kentucky'
'Michigan'
 'Mississippi' 'Florida' 'Alabama' 'Texas' 'Arkansas' 'Idaho'
'Indiana'
 'Illinois' 'New Mexico' 'Iowa' 'Minnesota' 'South Dakota' 'Nebraska'
 'North Dakota' 'Montana' 'Oklahoma' 'Kansas' 'Oregon' 'Utah' 'Nevada'
 'Washington' 'Arizona' 'Hawaii' 'Alaska']

Cleaned DataFrame:
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 2221871 entries, 3403 to 2226381
Data columns (total 7 columns):
 #   Column      Dtype  
---  ------      -----  
 0   price       float64
 1   bed         float64
 2   bath        float64
 3   acre_lot    float64
 4   city        object 
 5   state       object 
 6   house_size  float64
dtypes: float64(5), object(2)
memory usage: 135.6+ MB
None
Cleaned columns: Index(['price', 'bed', 'bath', 'acre_lot', 'city',
'state', 'house_size'], dtype='object')
```

Define Pricing Tier

```
# Define classification criteria for pricing tiers
def establish_pricing_tier(price):
    if price <= 300000:
        return 'Starter Homes (Tier 1)'
    elif price <= 500000:
        return 'Family-Friendly Homes (Tier 2)'
    elif price <= 1000000:
        return 'Executive Properties (Tier 3)'
    else:
        return 'Luxury Estates (Tier 4)'

# Apply classification criteria to create 'Pricing Tier' column
df_cleaned.loc[:, 'Pricing Tier'] =
df_cleaned['price'].apply(establish_pricing_tier)

# Define the order of pricing tiers
pricing_tier_order = [
    'Starter Homes (Tier 1)',
    'Family-Friendly Homes (Tier 2)',
    'Executive Properties (Tier 3)',
    'Luxury Estates (Tier 4)'
]

# Add 'Pricing Tier Order' column to DataFrame using .loc
df_cleaned.loc[:, 'Pricing Tier Order'] =
pd.Categorical(df_cleaned['Pricing Tier'],
categories=pricing_tier_order, ordered=True)

# Reorder the DataFrame based on the pricing tier order
df_cleaned = df_cleaned.sort_values(by='Pricing Tier Order')
```

```
# Summary of classification criteria
print("\nClassification Criteria for Residential Properties:")
print("Starter Homes (Tier 1): Affordable options below $300,000, ideal for first-time homebuyers or those on a budget.")
print("Family-Friendly Homes (Tier 2): Mid-range properties between $300,001 and $500,000, suitable for families.")
print("Executive Properties (Tier 3): Upscale living between $500,001 and $1,000,000, featuring modern amenities.")
print("Luxury Estates (Tier 4): High-end properties above $1,000,000, offering expansive living spaces and premium finishes.")
```

```
Classification Criteria for Residential Properties:
Starter Homes (Tier 1): Affordable options below $300,000, ideal for first-time homebuyers or those on a budget.
Family-Friendly Homes (Tier 2): Mid-range properties between $300,001 and $500,000, suitable for families.
Executive Properties (Tier 3): Upscale living between $500,001 and $1,000,000, featuring modern amenities.
Luxury Estates (Tier 4): High-end properties above $1,000,000, offering expansive living spaces and premium finishes.
```

Remove Duplicate Rows

```
# Remove duplicate rows
print('Number of duplicate rows: ', df_cleaned.duplicated().sum())
df_cleaned.drop_duplicates(inplace=True)
print('Number of duplicate rows after removal: ', df_cleaned.duplicated().sum())
```

```
Number of duplicate rows:  210635
Number of duplicate rows after removal:  0
```

Check for Missing Values

```
# Check for missing values (null)
print(df_cleaned.isnull().sum())
```

```
price                 0
bed                   0
bath                  0
acre_lot              0
city                  0
state                 0
house_size            0
Pricing Tier          0
Pricing Tier Order    0
dtype: int64
```

```
# Check for missing values (na)
df_cleaned.isna().sum()
```

```
price                  0
bed                    0
bath                   0
acre_lot               0
city                   0
state                  0
house_size             0
Pricing Tier           0
Pricing Tier Order     0
dtype: int64
```

Descriptive Statistics

```
# Summary statistics for numerical columns
df_cleaned.describe()
```

```
              price           bed          bath      acre_lot  
house_size
count  2.011236e+06  2.011236e+06  2.011236e+06  2.011236e+06  
2.011236e+06
mean   5.377258e+05  3.232182e+00  2.394645e+00  1.351008e+01  
2.535563e+03
std    2.235748e+06  1.429030e+00  1.511285e+00  7.179132e+02  
7.337463e+05
min    0.000000e+00  1.000000e+00  1.000000e+00  0.000000e+00  
4.000000e+00
25%    1.749000e+05  3.000000e+00  2.000000e+00  1.700000e-01  
1.449000e+03
50%    3.300000e+05  3.000000e+00  2.000000e+00  2.600000e-01  
1.760000e+03
75%    5.590000e+05  4.000000e+00  3.000000e+00  7.000000e-01  
2.171000e+03
max    2.147484e+09  4.730000e+02  8.300000e+02  1.000000e+05  
1.040400e+09
```

```
# Summary statistics for categorical columns
df_cleaned.describe(include = ['object', 'category'])
```

```
           city    state            Pricing Tier      Pricing Tier 
Order
count   2011236  2011236                 2011236                   
2011236
unique    19995       51                       4                   
4
top     Houston  Florida  Starter Homes (Tier 1)  Starter Homes (Tier 
1)
freq      22268   212247                  928144                   
928144
```

```
# Counts of unique values in descending order
df_cleaned.value_counts()
```

```
price          bed  bath  acre_lot  city            state
house_size  Pricing Tier                       Pricing Tier Order

0.000000e+00  2.0  2.0   0.26      Brentwood       California  1520.0
Starter Homes (Tier 1)          Starter Homes (Tier 1)            1
4.599000e+05  3.0  2.0   0.13      Riverview       Florida     1738.0
Family-Friendly Homes (Tier 2)  Family-Friendly Homes (Tier 2)    1
                                   Peoria          Arizona     1653.0
Family-Friendly Homes (Tier 2)  Family-Friendly Homes (Tier 2)    1
                                   Murrieta        California  1143.0
Family-Friendly Homes (Tier 2)  Family-Friendly Homes (Tier 2)    1
                                   Lockeford       California  1598.0
Family-Friendly Homes (Tier 2)  Family-Friendly Homes (Tier 2)    1

..
2.259900e+05  4.0  2.0   0.26      Xenia           Ohio        1760.0
Starter Homes (Tier 1)          Starter Homes (Tier 1)            1
                         0.23      Palm Bay        Florida     1449.0
Starter Homes (Tier 1)          Starter Homes (Tier 1)            1
                                   Citrus Springs  Florida     1650.0
Starter Homes (Tier 1)          Starter Homes (Tier 1)            1
                         0.15      Arizona City    Arizona     1209.0
Starter Homes (Tier 1)          Starter Homes (Tier 1)            1
2.147484e+09  2.0  2.0   0.12      International   California  885.0
Luxury Estates (Tier 4)         Luxury Estates (Tier 4)           1
Name: count, Length: 2011236, dtype: int64
```

```
# Counts of unique values in descending order for original dataset
df.value_counts()
```

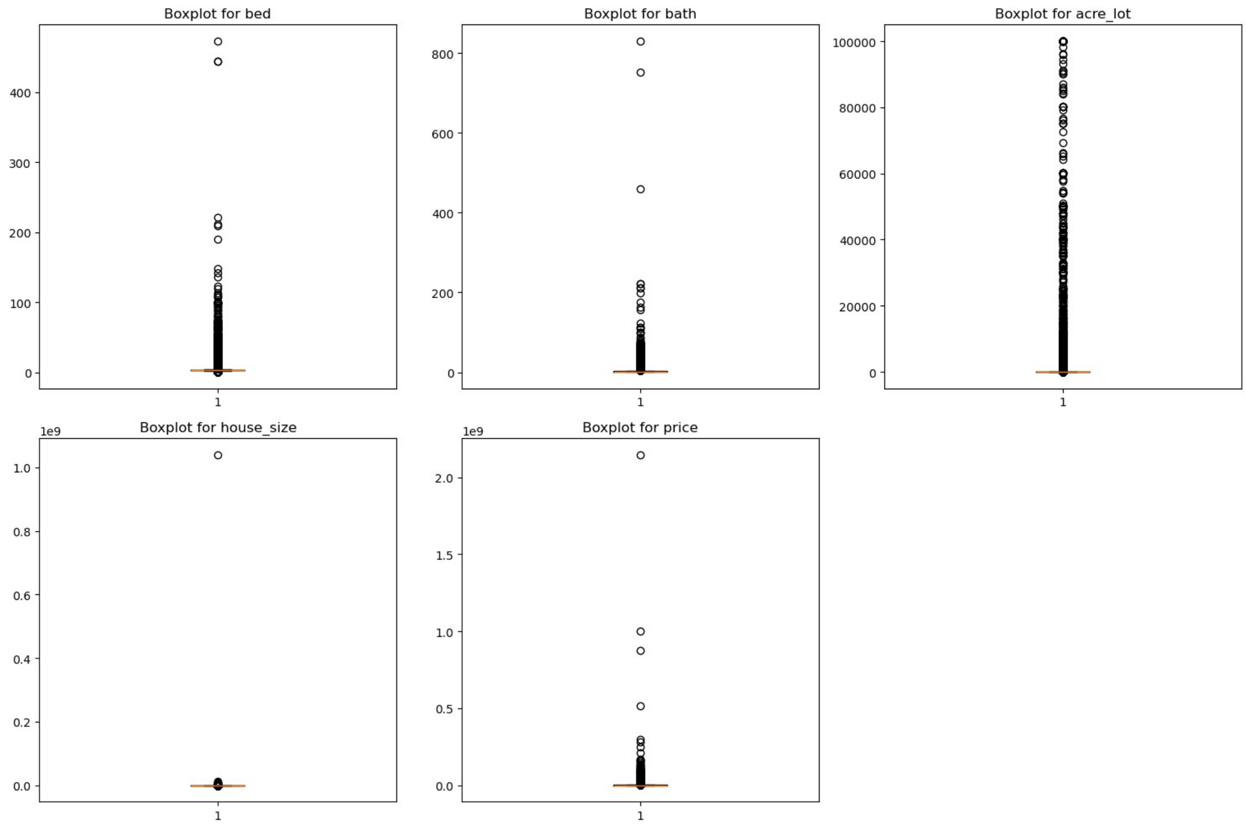
```
price          bed  bath  acre_lot  city            state
house_size
2.500000e+04  3.0  2.0   0.23      Port Charlotte  Florida     1760.0
169
2.000000e+04  3.0  2.0   0.23      Port Charlotte  Florida     1760.0
145
3.500000e+04  3.0  2.0   0.23      Palm Bay        Florida     1760.0
127
1.500000e+04  3.0  2.0   0.25      Lehigh Acres    Florida     1760.0
126
2.990000e+04  3.0  2.0   0.23      Port Charlotte  Florida     1760.0
118

...
2.390000e+05  2.0  2.0   0.04      Richmond        Virginia    1080.0
1
                                   Lancaster       Ohio        1449.0
```

```
1
                                   Lakeport        California  981.0
1
                                   Greenacres      Florida     1381.0
1
2.147484e+09  2.0  2.0   0.12      International   California  885.0
1
Name: count, Length: 2015556, dtype: int64
```

Data Visualization

Outlier Detection

```
# Check for outliers with boxplot for numerical columns

# Create a figure and a 2x3 grid of subplots
fig, axs = plt.subplots(2, 3, figsize=(15, 10))

# Plot data on the first 5 subplots
axs[0, 0].boxplot(df_cleaned[numerical_columns[0]])
axs[0, 0].set_title(f'Boxplot for {numerical_columns[0]}')

axs[0, 1].boxplot(df_cleaned[numerical_columns[1]])
axs[0, 1].set_title(f'Boxplot for {numerical_columns[1]}')

axs[0, 2].boxplot(df_cleaned[numerical_columns[2]])
axs[0, 2].set_title(f'Boxplot for {numerical_columns[2]}')

axs[1, 0].boxplot(df_cleaned[numerical_columns[3]])
axs[1, 0].set_title(f'Boxplot for {numerical_columns[3]}')

axs[1, 1].boxplot(df_cleaned[numerical_columns[4]])
axs[1, 1].set_title(f'Boxplot for {numerical_columns[4]}')

# Remove the last (empty) subplot
fig.delaxes(axs[1][2])

# Display the figure with subplots
plt.tight_layout()
plt.show()
```

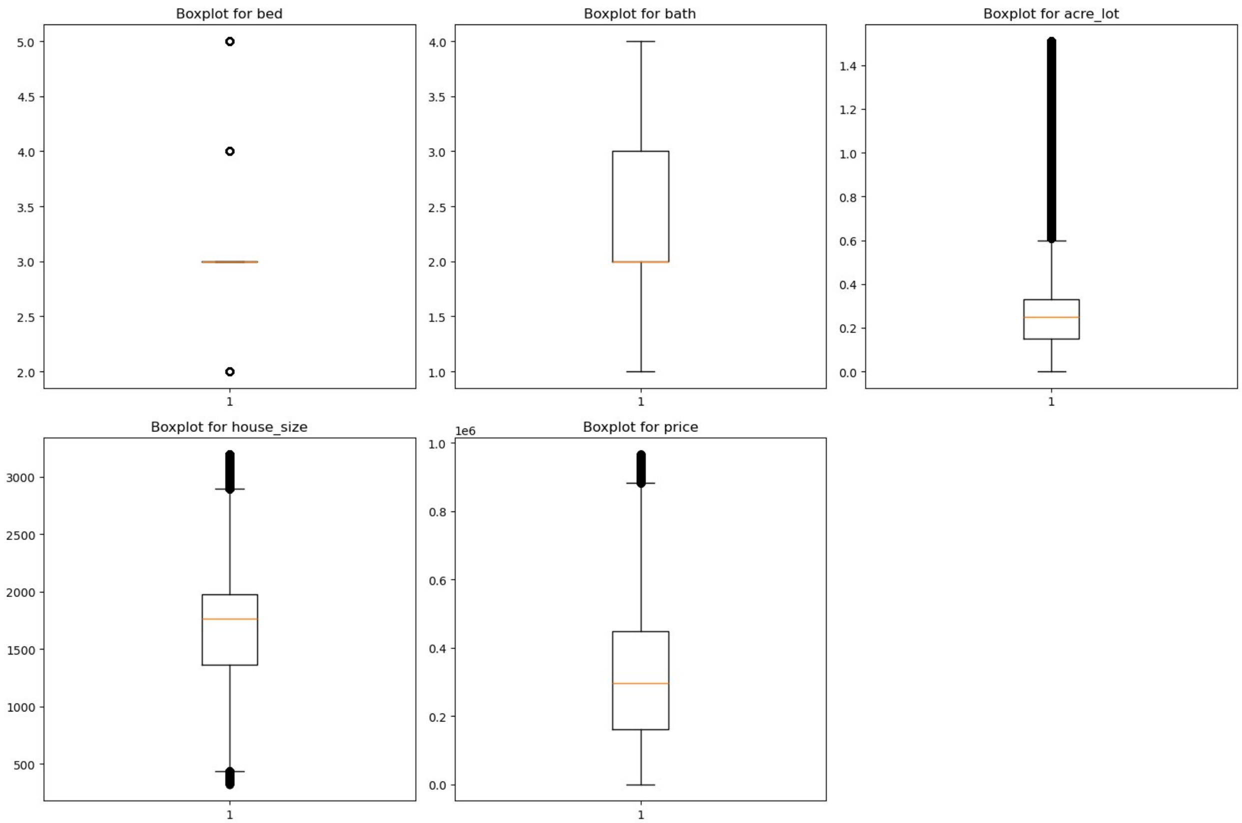



```
# Remove outliers for numerical columns

# Print total number of rows in filtered dataframe
print(f'Total rows with outliers : {df_cleaned.shape[0]}')

# Copy dataframe
df_no_outliers = df_cleaned.copy()

# Loop through numerical columns
for col in numerical_columns:
    # Calculate IQR
    Q1 = df_no_outliers[col].quantile(0.25)
    Q3 = df_no_outliers[col].quantile(0.75)
    IQR = Q3 - Q1

    # Define bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Remove outliers
    df_no_outliers = df_no_outliers[(df_no_outliers[col] >=
lower_bound) & (df_no_outliers[col] <= upper_bound)]

    # Print total number of rows after removal of outliers
```

```
    print(f'Total rows with out outliers for column {col} :
{df_no_outliers.shape[0]}')
```

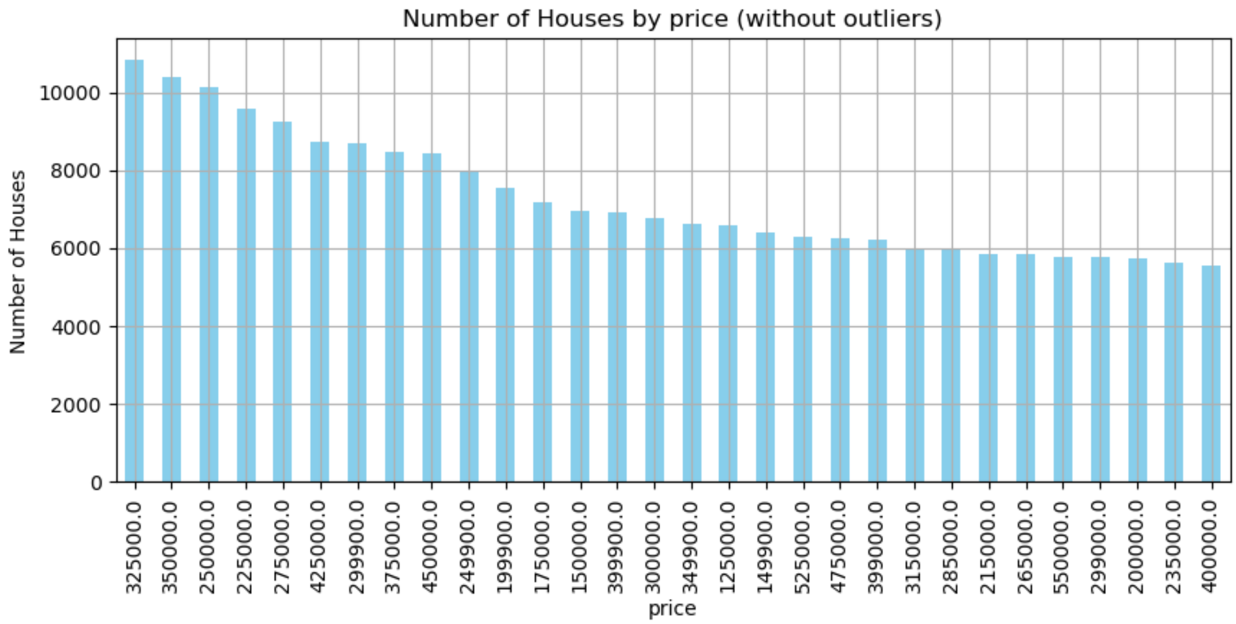
```
Total rows with outliers : 2011236
Total rows with out outliers for column bed : 1903347
Total rows with out outliers for column bath : 1852296
Total rows with out outliers for column acre_lot : 1545777
Total rows with out outliers for column house_size : 1465043
Total rows with out outliers for column price : 1382006
```

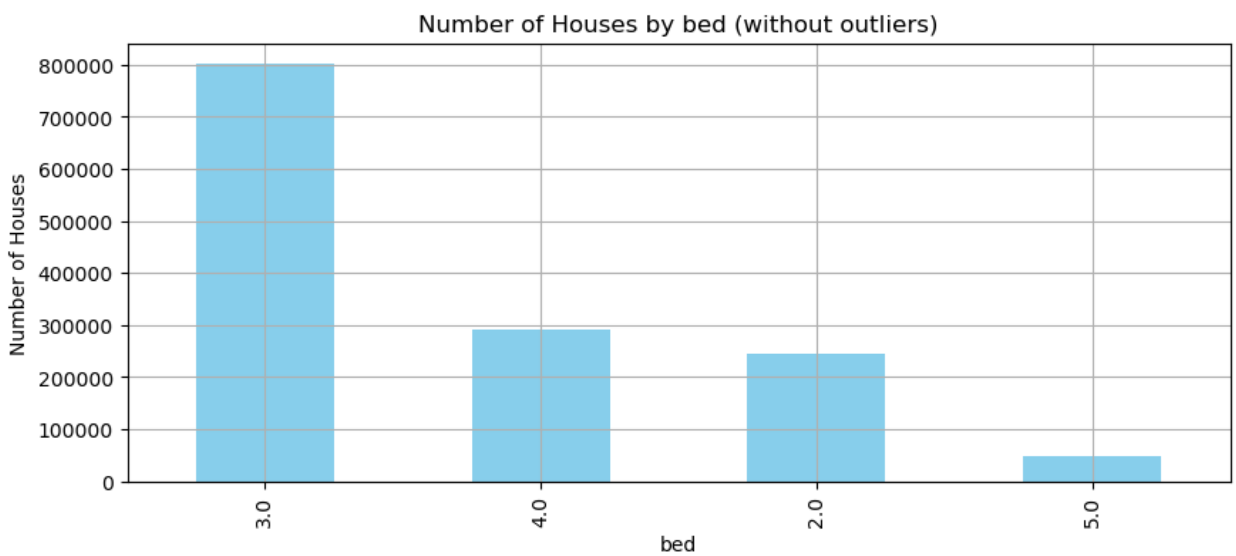
```
# Plot boxplots after removal of outliers from numerical columns

# Create a figure and a 2x3 grid of subplots
fig, axs = plt.subplots(2, 3, figsize=(15, 10))

# Plot data on the first 5 subplots
axs[0, 0].boxplot(df_no_outliers[numerical_columns[0]])
axs[0, 0].set_title(f'Boxplot for {numerical_columns[0]}')

axs[0, 1].boxplot(df_no_outliers[numerical_columns[1]])
axs[0, 1].set_title(f'Boxplot for {numerical_columns[1]}')

axs[0, 2].boxplot(df_no_outliers[numerical_columns[2]])
axs[0, 2].set_title(f'Boxplot for {numerical_columns[2]}')

axs[1, 0].boxplot(df_no_outliers[numerical_columns[3]])
axs[1, 0].set_title(f'Boxplot for {numerical_columns[3]}')

axs[1, 1].boxplot(df_no_outliers[numerical_columns[4]])
axs[1, 1].set_title(f'Boxplot for {numerical_columns[4]}')

# Remove the last (empty) subplot
fig.delaxes(axs[1][2])

# Display the figure with subplots
plt.tight_layout()
plt.show()
```

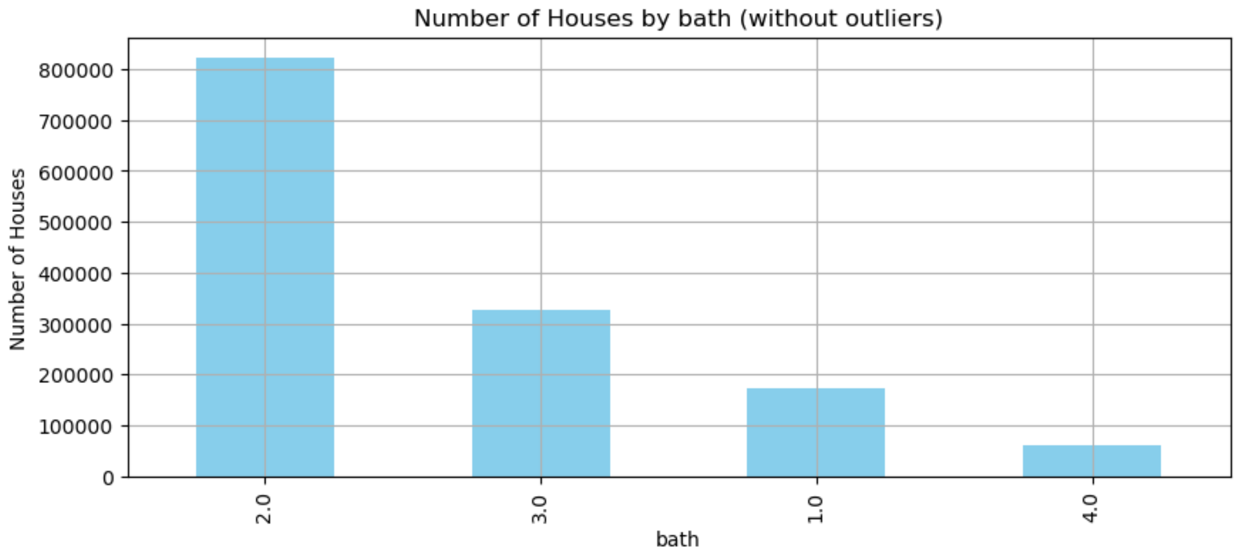


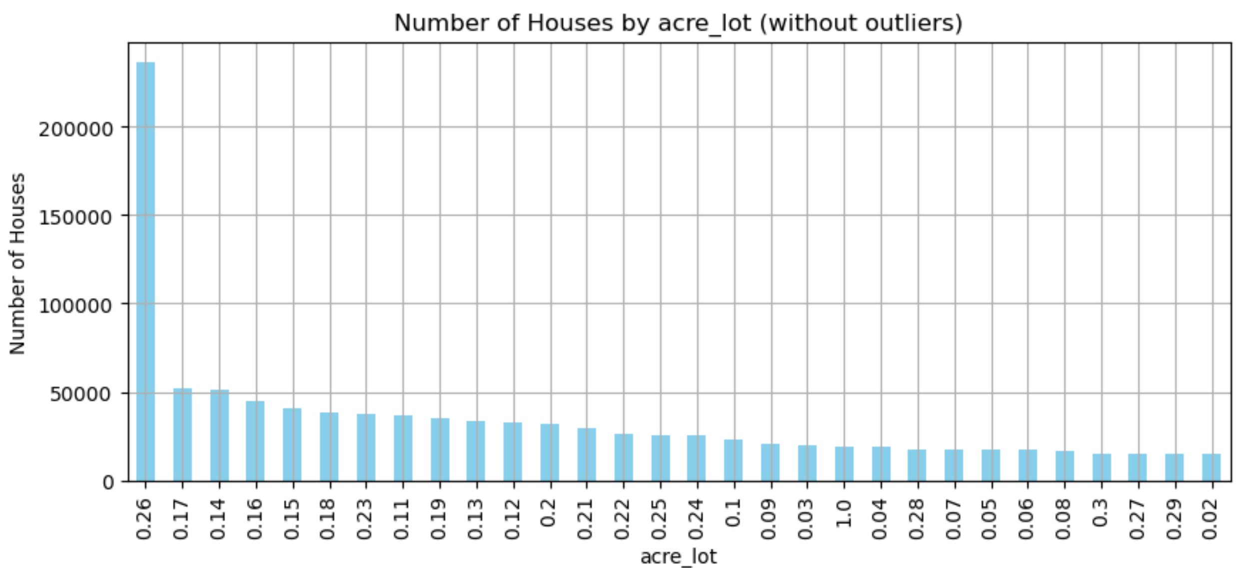
Visualization

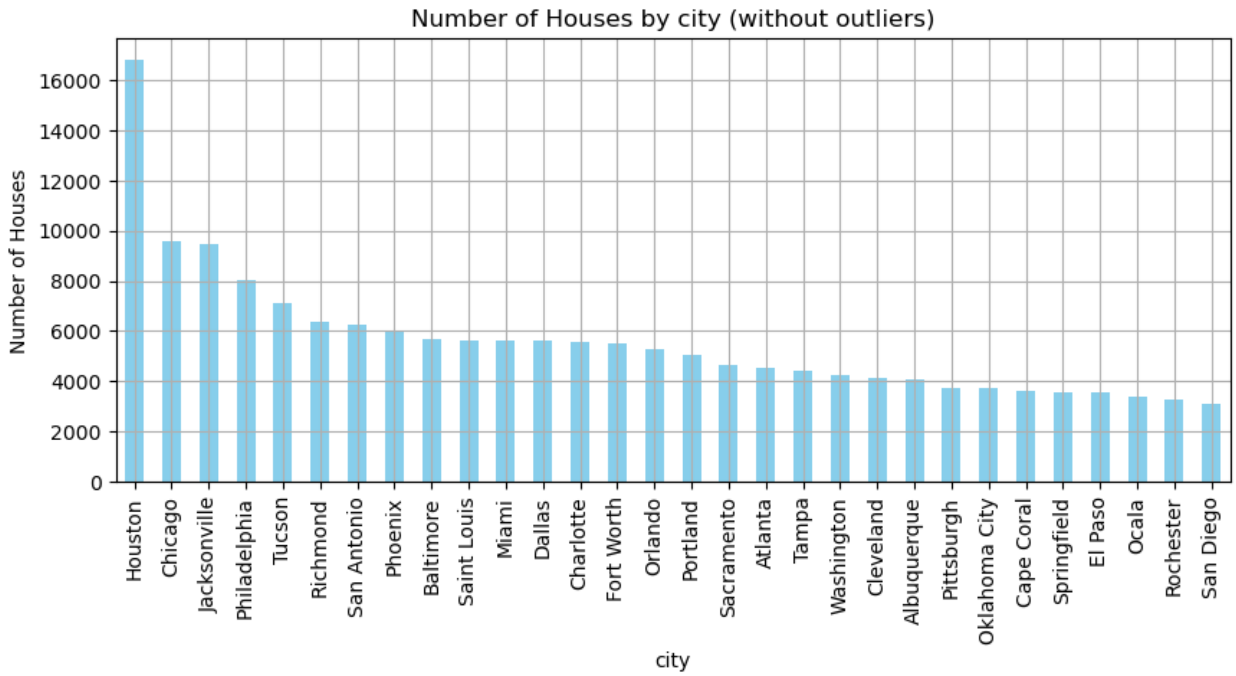
```
# Plot bar graph for each column (for data frame without outliers)
for col in df.columns:
    # Pick top 30 unique counts for bar chart
    df_no_outliers[col].value_counts().head(30).plot(kind='bar',
figsize=(10,4), color='skyblue')

    # Set title, labels and display
    plt.title(f'Number of Houses by {col} (without outliers)')
    plt.xlabel(col)
    plt.ylabel('Number of Houses')
    plt.grid(True)
    plt.show()
```

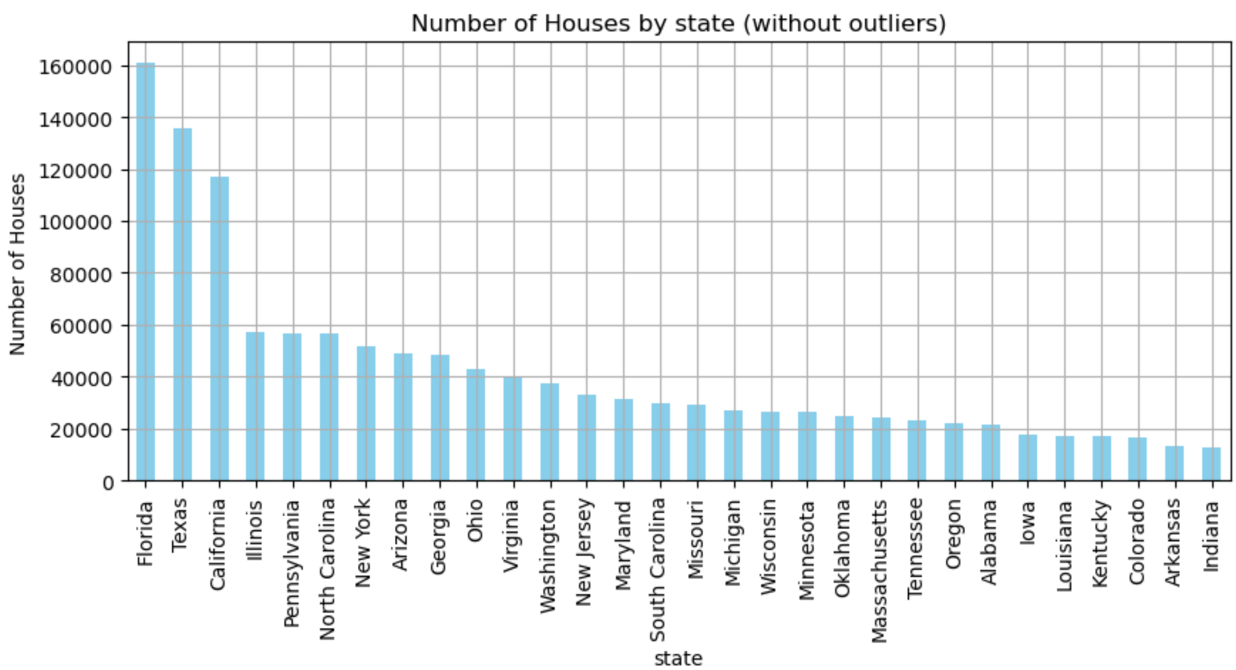

Number of Houses by price (without outliers)
Number of Houses
0
2000
4000
6000
8000
10000
325000.0
350000.0
250000.0
225000.0
275000.0
425000.0
299900.0
375000.0
450000.0
249900.0
199900.0
175000.0
150000.0
399900.0
300000.0
349900.0
125000.0
149900.0
525000.0
475000.0
399000.0
315000.0
285000.0
215000.0
265000.0
550000.0
299000.0
200000.0
235000.0
400000.0
price

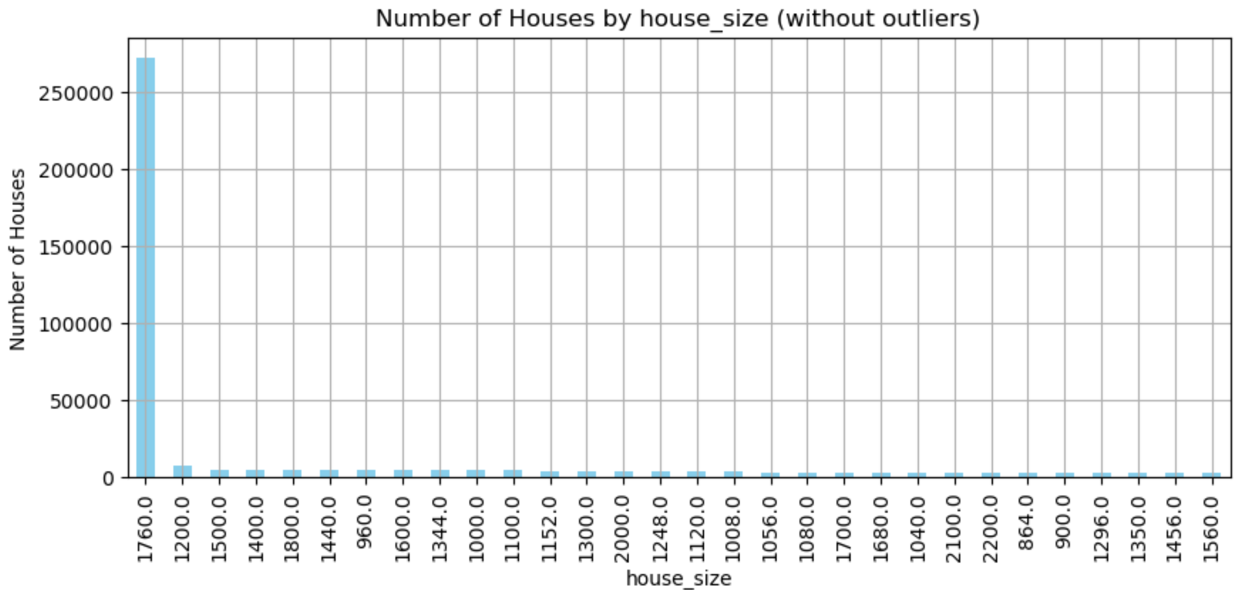

Number of Houses by bed (without outliers)
Number of Houses
0
100000
200000
300000
400000
500000
600000
700000
800000
3.0
4.0
2.0
5.0
bed


Number of Houses by bath (without outliers)
Number of Houses
800000
700000
600000
500000
400000
300000
200000
100000
0
2.0
3.0
1.0
4.0
bath

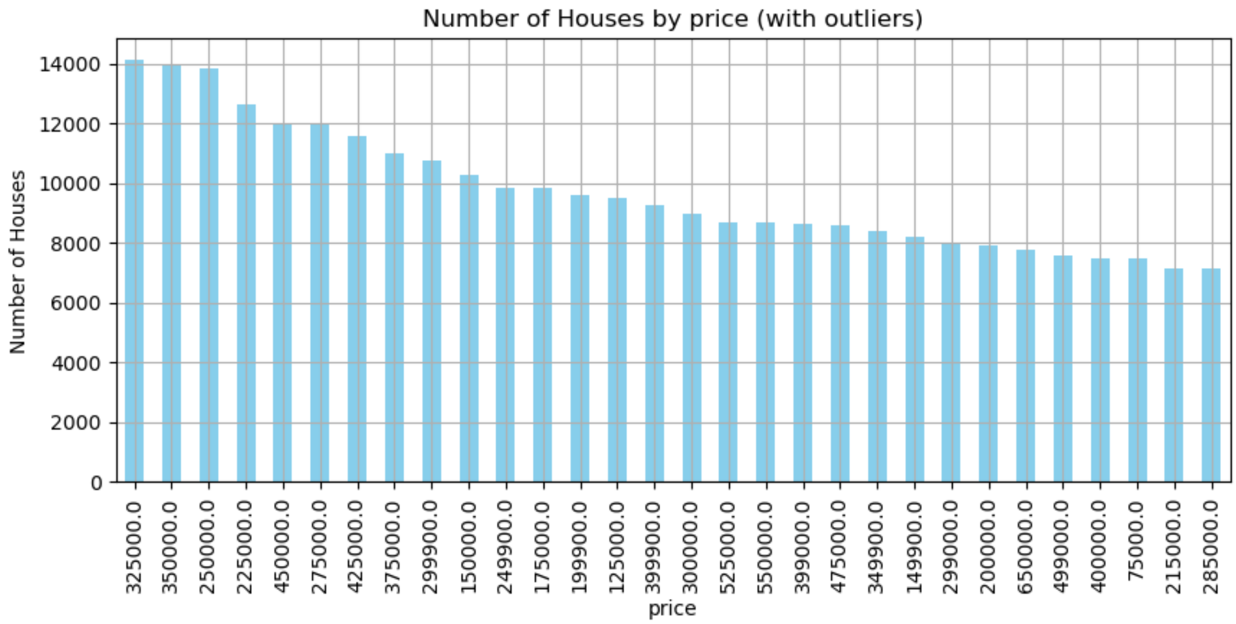

Number of Houses by acre_lot (without outliers)
Number of Houses
200000
150000
100000
50000
0
0.26
0.17
0.14
0.16
0.15
0.18
0.23
0.11
0.19
0.13
0.12
0.2
0.21
0.22
0.25
0.24
0.1
0.09
0.03
1.0
0.04
0.28
0.07
0.05
0.06
0.08
0.3
0.27
0.29
0.02
acre_lot

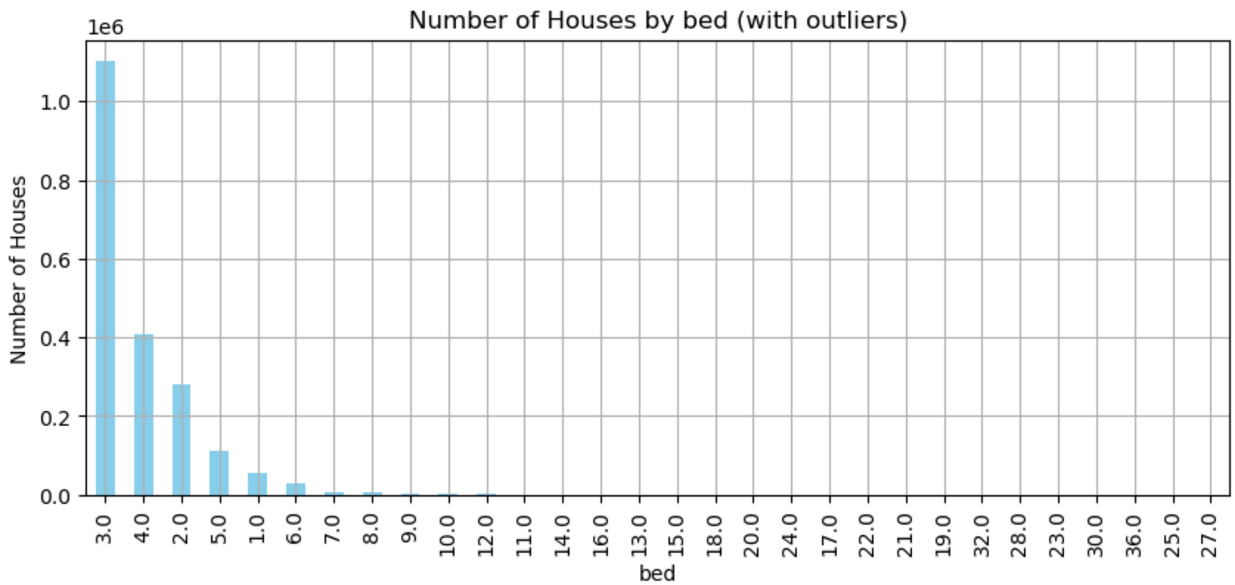

Number of Houses by city (without outliers)
Number of Houses
0
2000
4000
6000
8000
10000
12000
14000
16000
Houston
Chicago
Jacksonville
Philadelphia
Tucson
Richmond
San Antonio
Phoenix
Baltimore
Saint Louis
Miami
Dallas
Charlotte
Fort Worth
Orlando
Portland
Sacramento
Atlanta
Tampa
Washington
Cleveland
Albuquerque
Pittsburgh
Oklahoma City
Cape Coral
Springfield
El Paso
Ocala
Rochester
San Diego
city

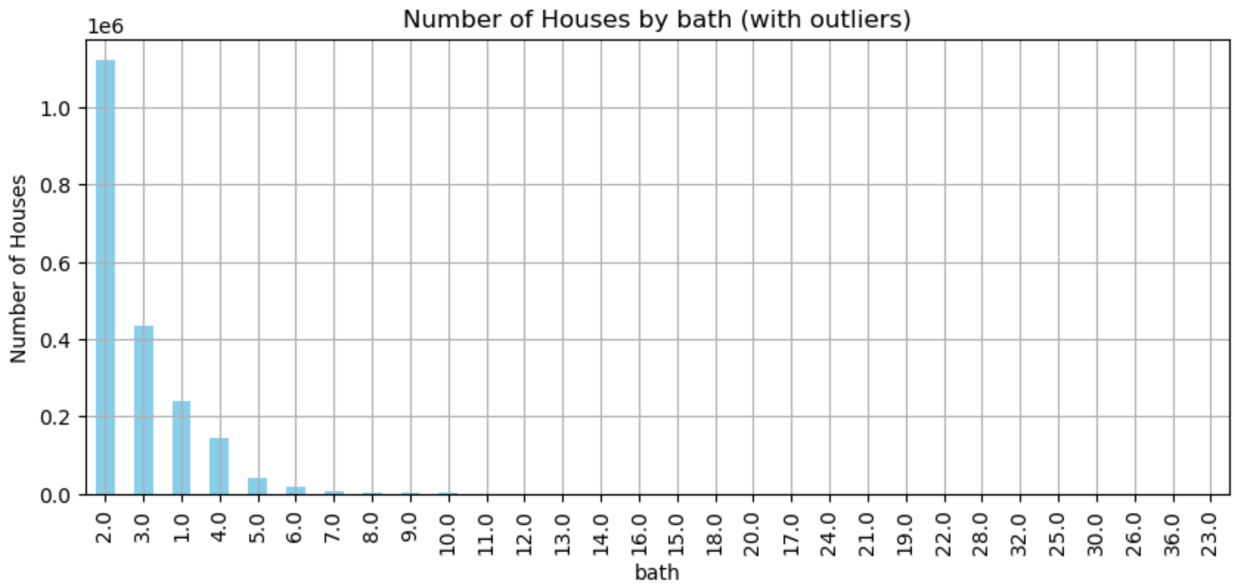

Number of Houses by state (without outliers)
Number of Houses
0
20000
40000
60000
80000
100000
120000
140000
160000
Florida
Texas
California
Illinois
Pennsylvania
North Carolina
New York
Arizona
Georgia
Ohio
Virginia
Washington
New Jersey
Maryland
South Carolina
Missouri
Michigan
Wisconsin
Minnesota
Oklahoma
Massachusetts
Tennessee
Oregon
Alabama
Iowa
Louisiana
Kentucky
Colorado
Arkansas
Indiana
state

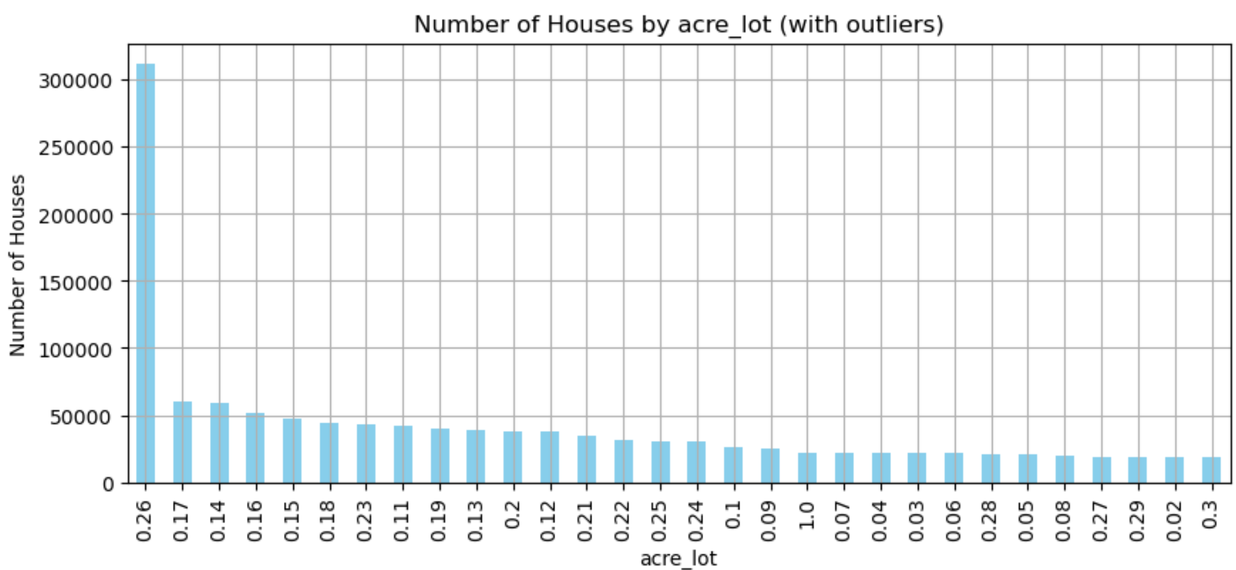


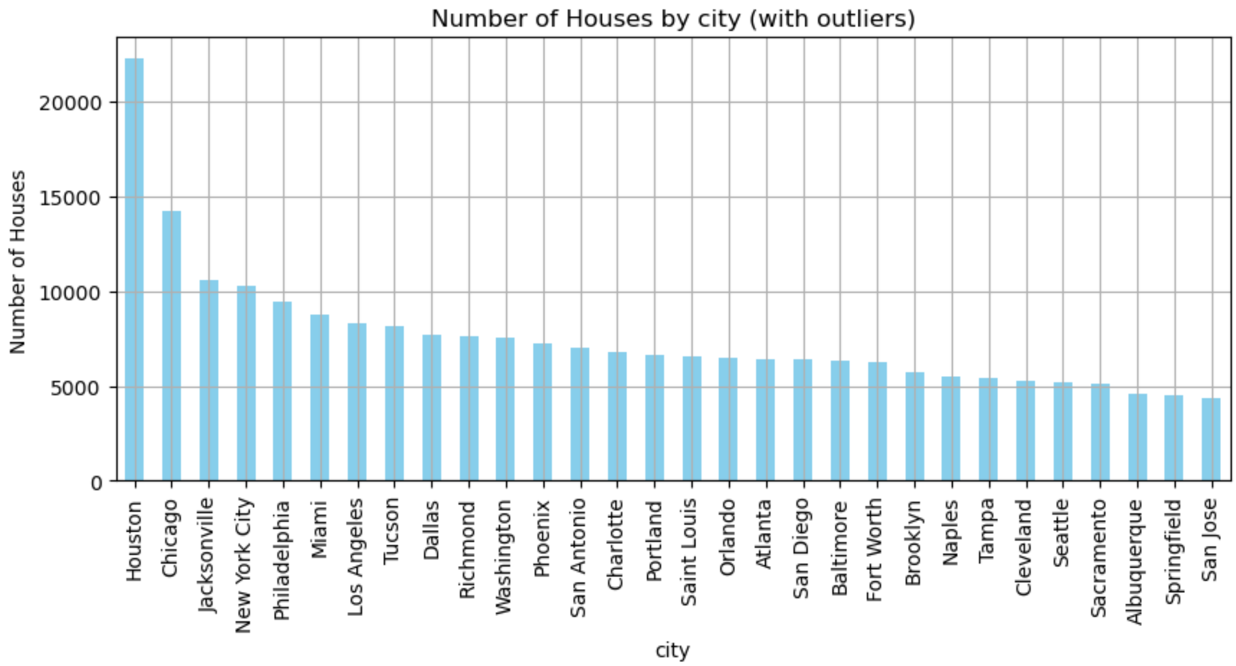

```
# Plot bar graph for each column (for data frame with outliers)
for col in df.columns:
    # Pick top 30 unique counts for bar chart
    df_cleaned[col].value_counts().head(30).plot(kind='bar',
figsize=(10,4), color='skyblue')

    # Set title, labels and display
    plt.title(f'Number of Houses by {col} (with outliers)')
    plt.xlabel(col)
    plt.ylabel('Number of Houses')
    plt.grid(True)
    plt.show()
```

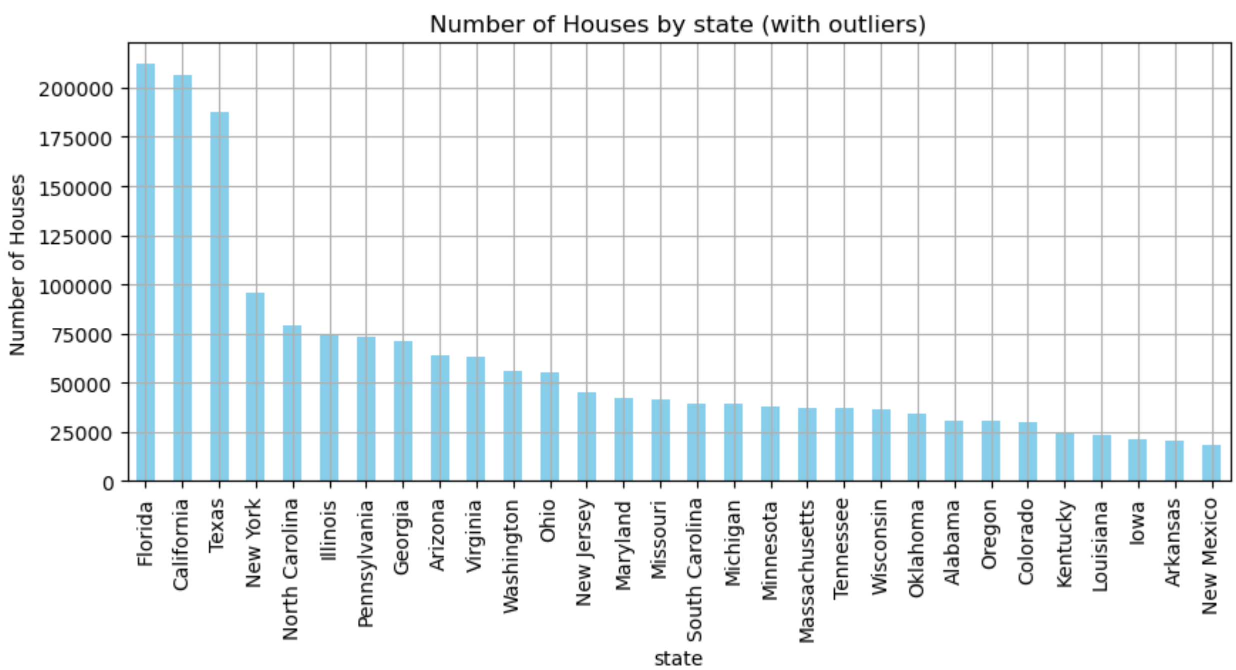

Number of Houses by price (with outliers)
Number of Houses
0
2000
4000
6000
8000
10000
12000
14000
325000.0
350000.0
250000.0
225000.0
450000.0
275000.0
425000.0
375000.0
299900.0
150000.0
249900.0
175000.0
199900.0
125000.0
399900.0
300000.0
525000.0
550000.0
399000.0
475000.0
349900.0
149900.0
299000.0
200000.0
650000.0
499000.0
400000.0
75000.0
215000.0
285000.0
price

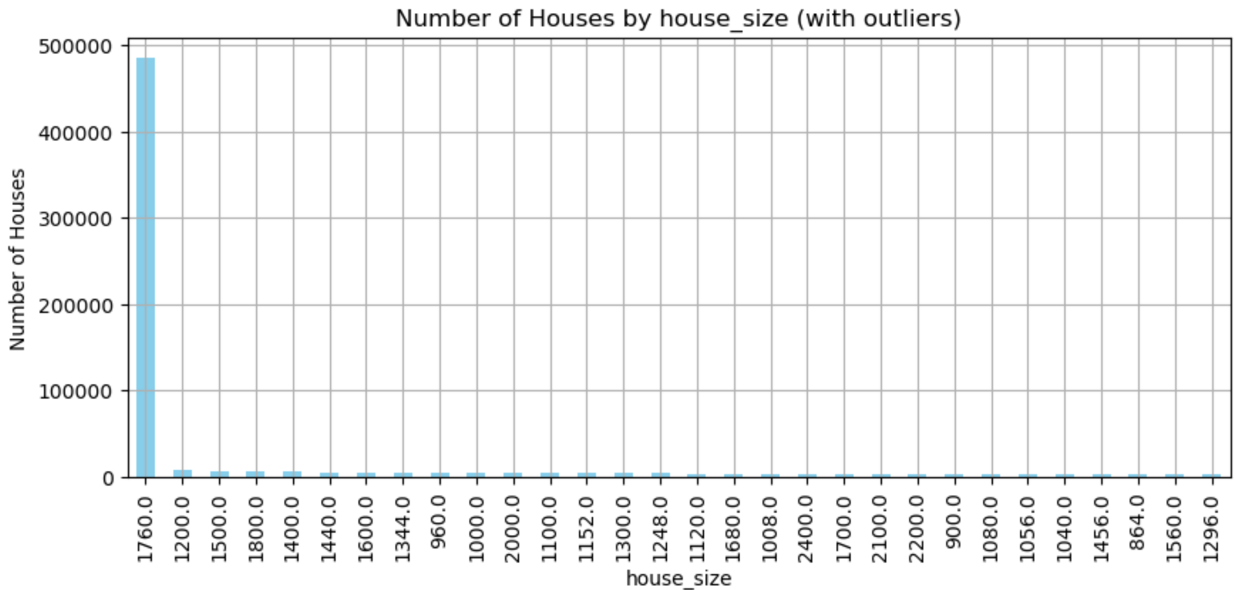

Number of Houses by bed (with outliers)
1e6
Number of Houses
0.0
0.2
0.4
0.6
0.8
1.0
3.0
4.0
2.0
5.0
1.0
6.0
7.0
8.0
9.0
10.0
12.0
11.0
14.0
16.0
13.0
15.0
18.0
20.0
24.0
17.0
22.0
21.0
19.0
32.0
28.0
23.0
30.0
36.0
25.0
27.0
bed


Number of Houses by bath (with outliers)
1e6
1.0
0.8
0.6
0.4
0.2
0.0
Number of Houses
2.0 3.0 1.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 16.0 15.0 18.0 20.0 17.0 24.0 21.0 19.0 22.0 28.0 32.0 25.0 30.0 26.0 36.0 23.0
bath


Number of Houses by acre_lot (with outliers)
300000
250000
200000
150000
100000
50000
0
Number of Houses
0.26 0.17 0.14 0.16 0.15 0.18 0.23 0.11 0.19 0.13 0.2 0.12 0.21 0.22 0.25 0.24 0.1 0.09 1.0 0.07 0.04 0.03 0.06 0.28 0.05 0.08 0.27 0.29 0.02 0.3
acre_lot


Number of Houses by city (with outliers)
Number of Houses
0
5000
10000
15000
20000
city
Houston
Chicago
Jacksonville
New York City
Philadelphia
Miami
Los Angeles
Tucson
Dallas
Richmond
Washington
Phoenix
San Antonio
Charlotte
Portland
Saint Louis
Orlando
Atlanta
San Diego
Baltimore
Fort Worth
Brooklyn
Naples
Tampa
Cleveland
Seattle
Sacramento
Albuquerque
Springfield
San Jose


Number of Houses by state (with outliers)
Number of Houses
0
25000
50000
75000
100000
125000
150000
175000
200000
state
Florida
California
Texas
New York
North Carolina
Illinois
Pennsylvania
Georgia
Arizona
Virginia
Washington
Ohio
New Jersey
Maryland
Missouri
South Carolina
Michigan
Minnesota
Massachusetts
Tennessee
Wisconsin
Oklahoma
Alabama
Oregon
Colorado
Kentucky
Louisiana
Iowa
Arkansas
New Mexico



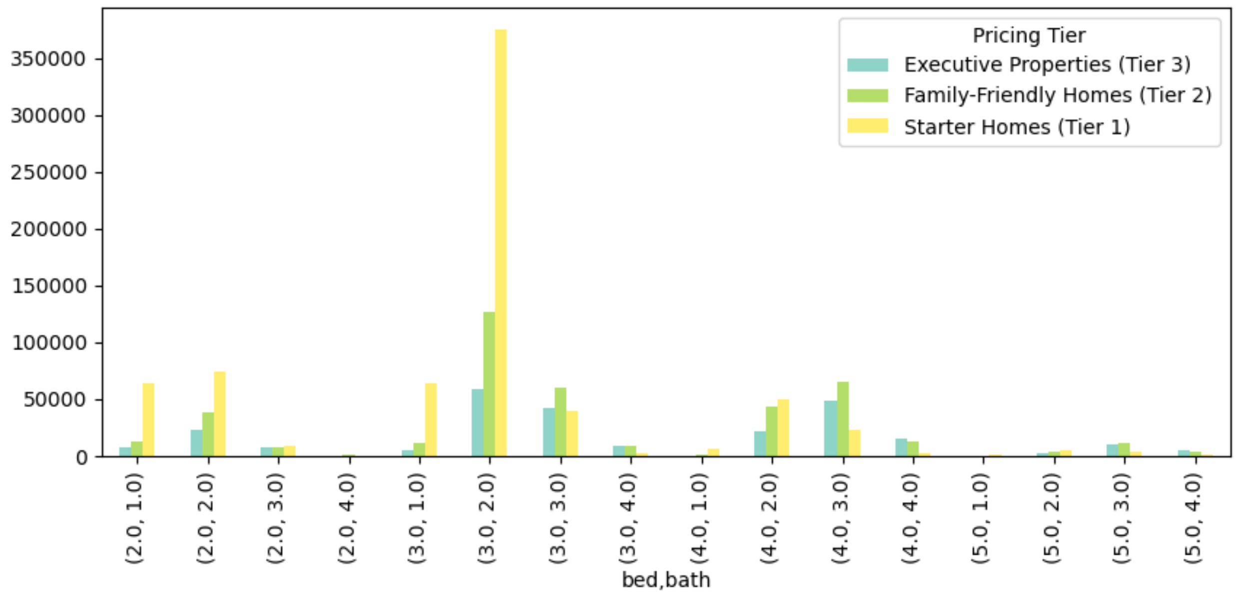

```
# Group by state, city, bed and bath
# Display mean values for acre_lot, house_size and price
print('Group by state, city, bed and bath for dataframe without
outliers')
df_no_outliers.groupby(['state', 'city', 'bed',
'bath']).agg({'acre_lot' : 'mean', 'house_size' : 'mean', 'price' :
'mean'})

Group by state, city, bed and bath for dataframe without outliers

                              acre_lot   house_size          price
state   city       bed bath
Alabama Abbeville  2.0 1.0    0.593333   887.666667  179333.333333
                       2.0    0.697500  1111.000000  111700.000000
                   3.0 1.0    0.255000   970.000000   33494.000000
                       2.0    0.569792  1738.416667  101290.625000
                   4.0 2.0    0.476667  1818.000000  364833.333333
...                                ...          ...            ...
Wyoming Wright     4.0 3.0    0.240000  2105.500000  227450.000000
                   5.0 3.0    0.100000  1986.000000  129900.000000
                       4.0    0.310000  2128.000000  274900.000000
        Yoder      2.0 1.0    0.260000  1456.000000   95000.000000
                   3.0 2.0    0.480000  2080.000000   99000.000000

[120878 rows x 3 columns]

# Group by state, city, bed and bath
# Display mean values for acre_lot, house_size and price
print('Group by state, city, bed and bath for dataframe with
outliers')
df_cleaned.groupby(['state', 'city', 'bed', 'bath']).agg({'acre_lot' :
'mean', 'house_size' : 'mean', 'price' : 'mean'})
```

```
Group by state, city, bed and bath for dataframe with outliers

                              acre_lot   house_size          price
state   city       bed bath
Alabama Abbeville  1.0 1.0    0.500000   700.000000   14500.000000
                       2.0    1.000000   659.000000  295000.000000
                   2.0 1.0    0.593333   887.666667  179333.333333
                       2.0    0.697500  1111.000000  111700.000000
                   3.0 1.0    1.540000  1136.666667  152296.000000
...                                ...          ...            ...
Wyoming Wyoming    3.0 2.0   39.700000  1760.000000   99900.000000
        Yoder      2.0 1.0    0.260000  1456.000000   95000.000000
                   3.0 2.0    0.480000  2080.000000   99000.000000
                   4.0 2.0   10.000000  2356.000000  325000.000000
                       3.0   35.000000  2494.000000  598000.000000

[209444 rows x 3 columns]

# Counts of unique values in descending order for price_tier (for
dataframe without outliers)
df_no_outliers['Pricing Tier'].value_counts()

Pricing Tier
Starter Homes (Tier 1)             720184
Family-Friendly Homes (Tier 2)    407111
Executive Properties (Tier 3)      254711
Name: count, dtype: int64

# Counts of unique values in descending order for price_tier (for
dataframe with outliers)
df_cleaned['Pricing Tier'].value_counts()

Pricing Tier
Starter Homes (Tier 1)             928144
Family-Friendly Homes (Tier 2)    503125
Executive Properties (Tier 3)      401646
Luxury Estates (Tier 4)            178321
Name: count, dtype: int64

# Plot distribution of price tier by number of beds and baths (for
dataframe without outliers)

# Group data by bed, bath and price tier
grouped = df_no_outliers.groupby(['bed', 'bath', 'Pricing Tier'],
observed=False).size().reset_index(name='count')

# Create and print pivot table
pivot_table = pd.pivot_table(grouped, values='count', index=['bed',
'bath'], columns=['Pricing Tier'])
print(pivot_table)
```

```python
# Plot bar chart for pivot table
pivot_table.plot(kind='bar', figsize=(10,4), colormap='Set3')
plt.show()
```

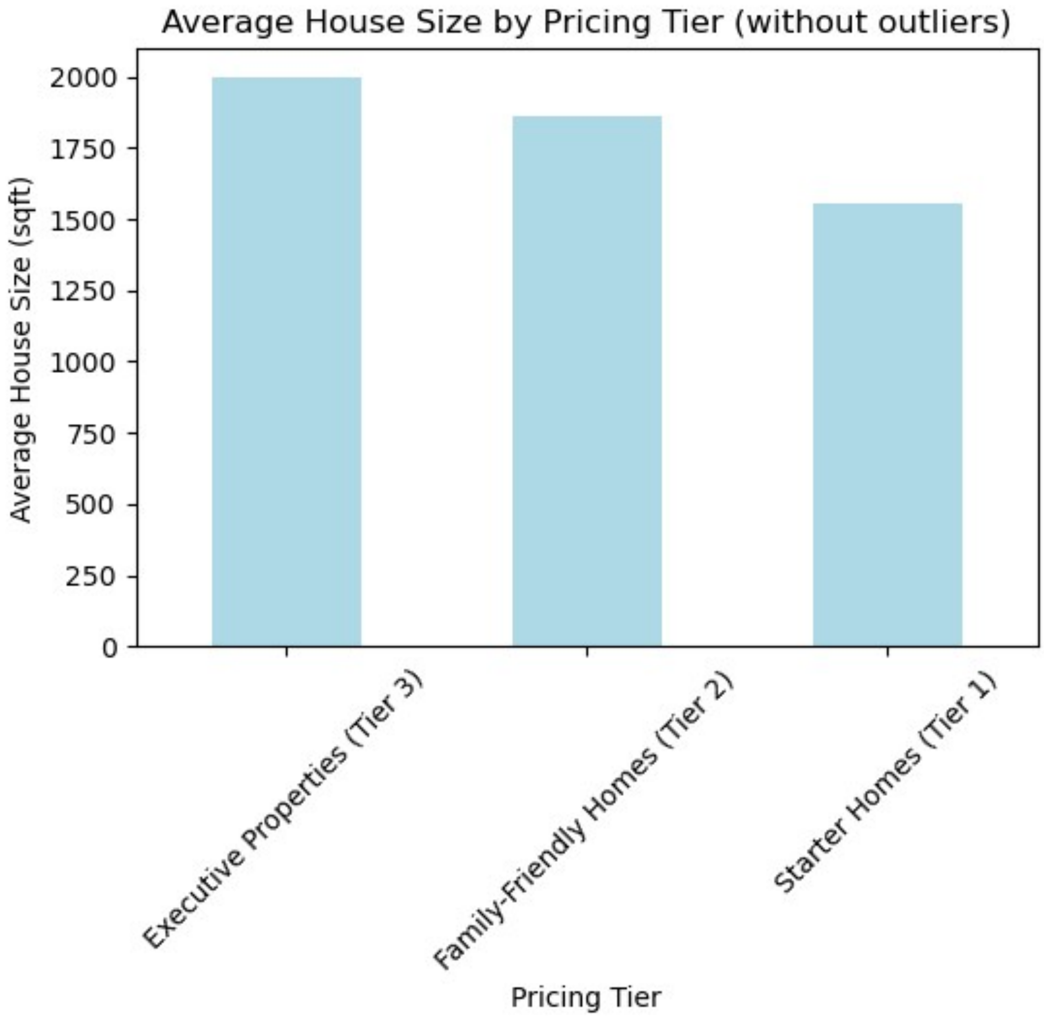
```
Pricing Tier  Executive Properties (Tier 3)  Family-Friendly Homes
(Tier 2)  \
bed bath

2.0 1.0                              6849.0
12866.0
    2.0                             23483.0
38408.0
    3.0                              6906.0
8053.0
    4.0                               433.0
513.0
3.0 1.0                              5505.0
11548.0
    2.0                             58513.0
125941.0
    3.0                             41742.0
59924.0
    4.0                              8529.0
8213.0
4.0 1.0                               435.0
1186.0
    2.0                             21439.0
43658.0
    3.0                             48127.0
65399.0
    4.0                             15540.0
12534.0
5.0 1.0                                28.0
67.0
    2.0                              2343.0
3445.0
    3.0                              9786.0
11195.0
    4.0                              5053.0
4161.0

Pricing Tier  Starter Homes (Tier 1)
bed bath
2.0 1.0                      63188.0
    2.0                      74291.0
    3.0                       8645.0
    4.0                        293.0
3.0 1.0                      64286.0
    2.0                     375599.0
    3.0                      39280.0
```

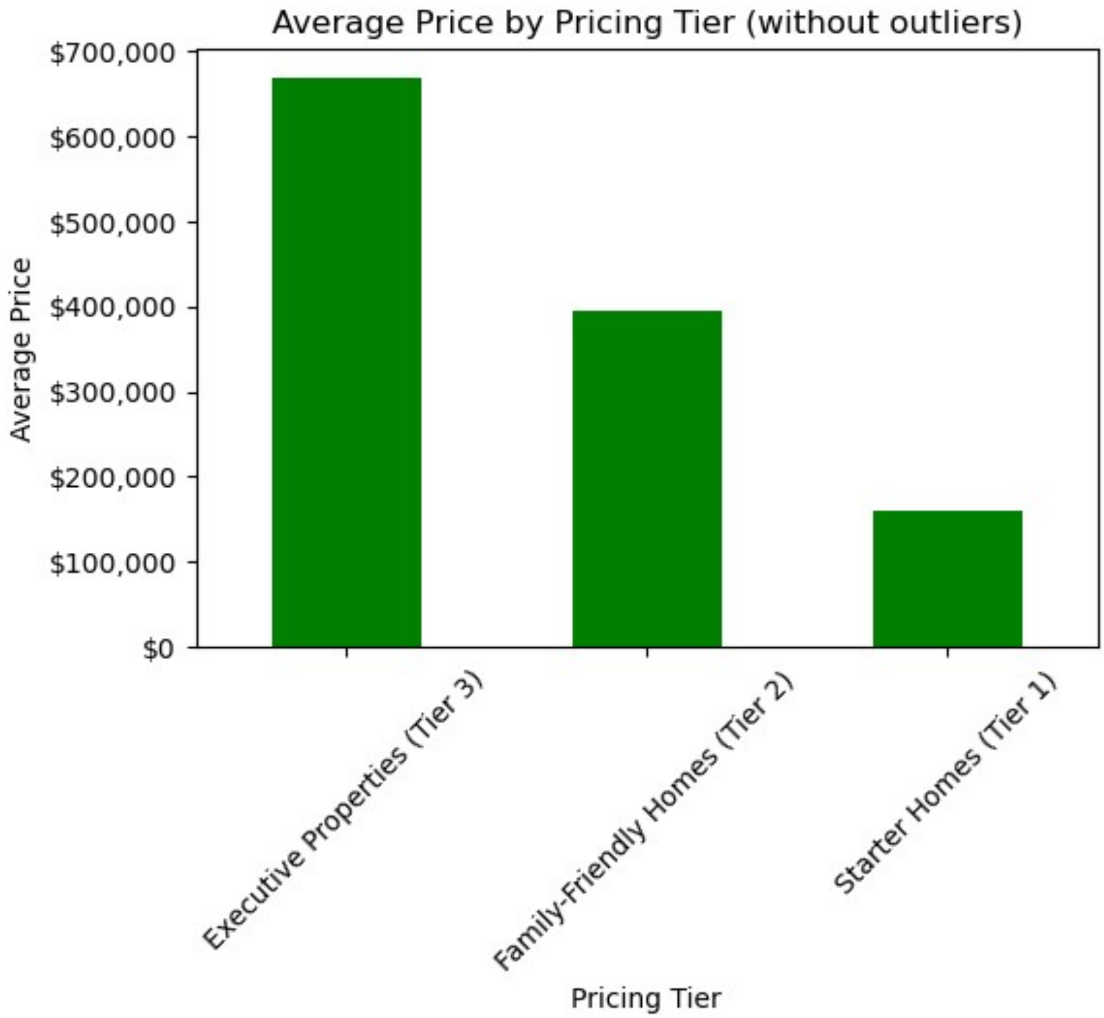
```
    4.0                          2227.0
4.0 1.0                          6785.0
    2.0                         49277.0
    3.0                         22898.0
    4.0                          2567.0
5.0 1.0                           642.0
    2.0                          5459.0
    3.0                          4024.0
    4.0                           723.0
```




```python
# Plot distribution of properties by property tier (for dataframe
without outliers)
plt.figure(figsize=(6, 4))
df_no_outliers.groupby('Pricing Tier', observed=True)
['house_size'].mean().sort_index().plot(kind='bar', color='lightblue')
plt.title('Average House Size by Pricing Tier (without outliers)')
plt.xlabel('Pricing Tier')
plt.ylabel('Average House Size (sqft)')
plt.xticks(rotation=45)
plt.show()

# Plot average price by property tier (for dataframe without outliers)
plt.figure(figsize=(6, 4))
ax = df_no_outliers.groupby('Pricing Tier', observed=True)
['price'].mean().sort_index().plot(kind='bar', color='green')
plt.title('Average Price by Pricing Tier (without outliers)')
plt.xlabel('Pricing Tier')
plt.ylabel('Average Price')
plt.xticks(rotation=45)
```

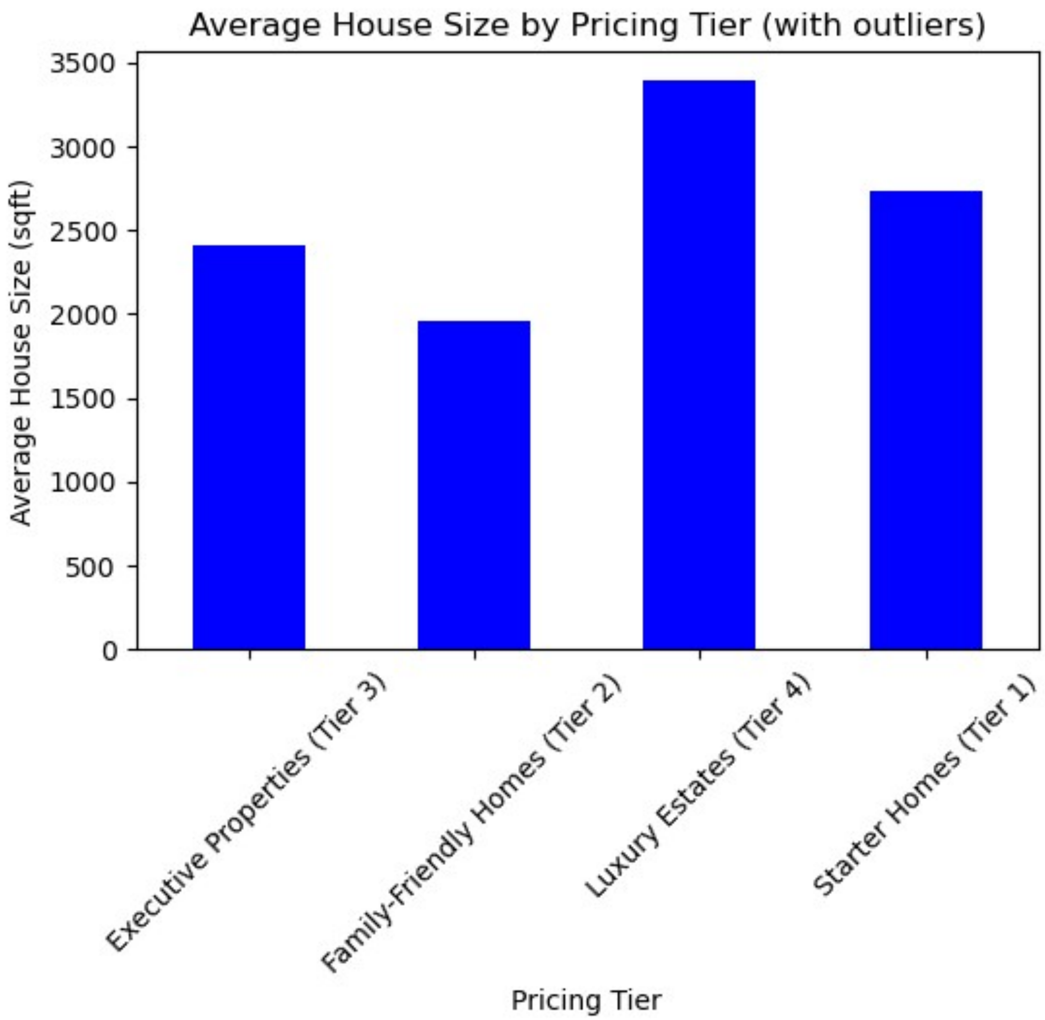
```python
# Format y-axis labels as dollar amounts
formatter = '${x:,.0f}'
ax.yaxis.set_major_formatter(formatter)
plt.show()
```

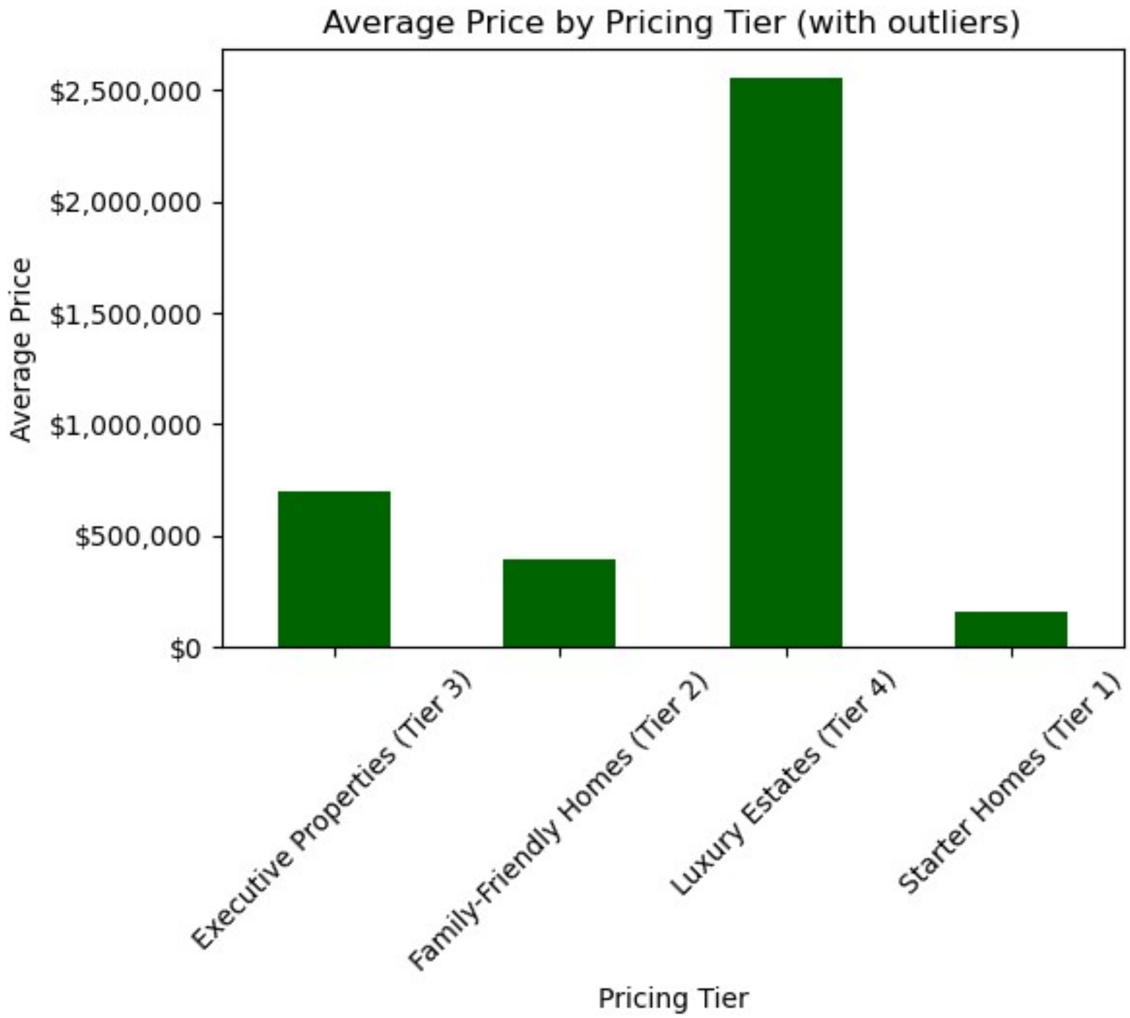





```
# Plot distribution of properties by property tier (for dataframe with
outliers)
plt.figure(figsize=(6, 4))
df_cleaned.groupby('Pricing Tier', observed=True)
['house_size'].mean().sort_index().plot(kind='bar', color='blue')
plt.title('Average House Size by Pricing Tier (with outliers)')
plt.xlabel('Pricing Tier')
plt.ylabel('Average House Size (sqft)')
plt.xticks(rotation=45)
plt.show()

# Plot average price by property tier (for dataframe with outliers)
plt.figure(figsize=(6, 4))
ax = df_cleaned.groupby('Pricing Tier', observed=True)
['price'].mean().sort_index().plot(kind='bar', color='darkgreen')
plt.title('Average Price by Pricing Tier (with outliers)')
plt.xlabel('Pricing Tier')
plt.ylabel('Average Price')
```

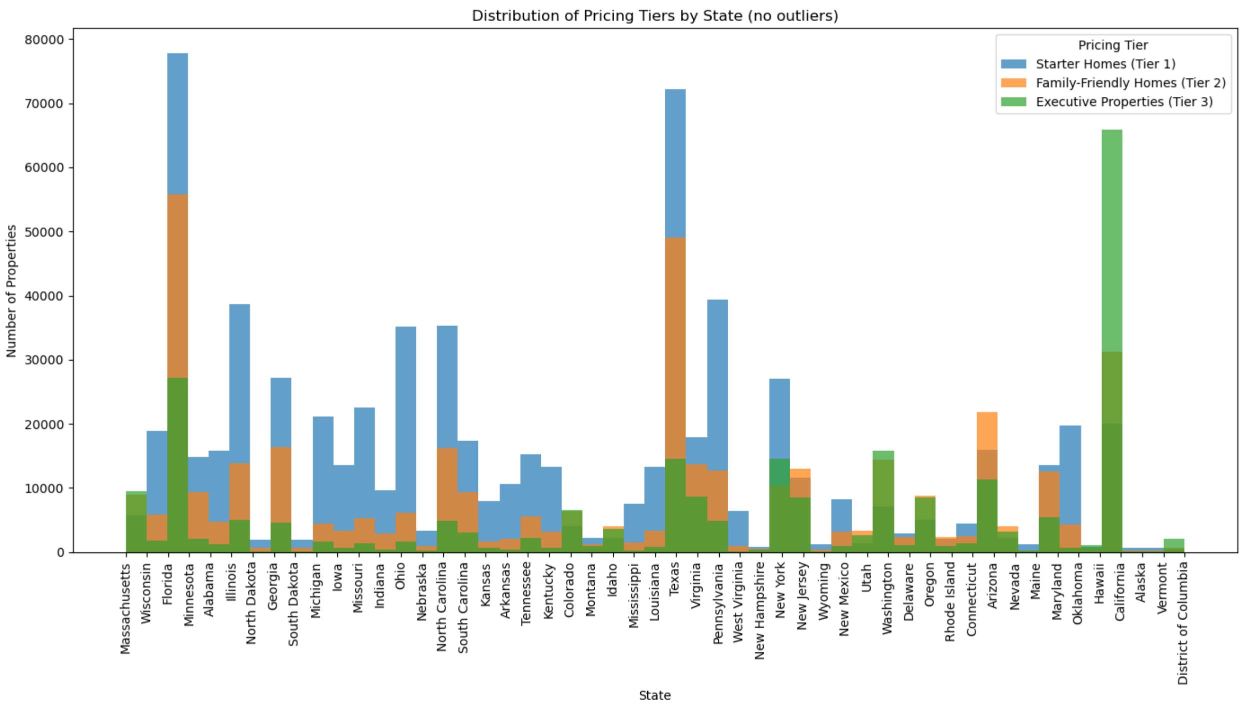
```
plt.xticks(rotation=45)

# Format y-axis labels as dollar amounts
formatter = '${x:,.0f}'
ax.yaxis.set_major_formatter(formatter)
plt.show()
```





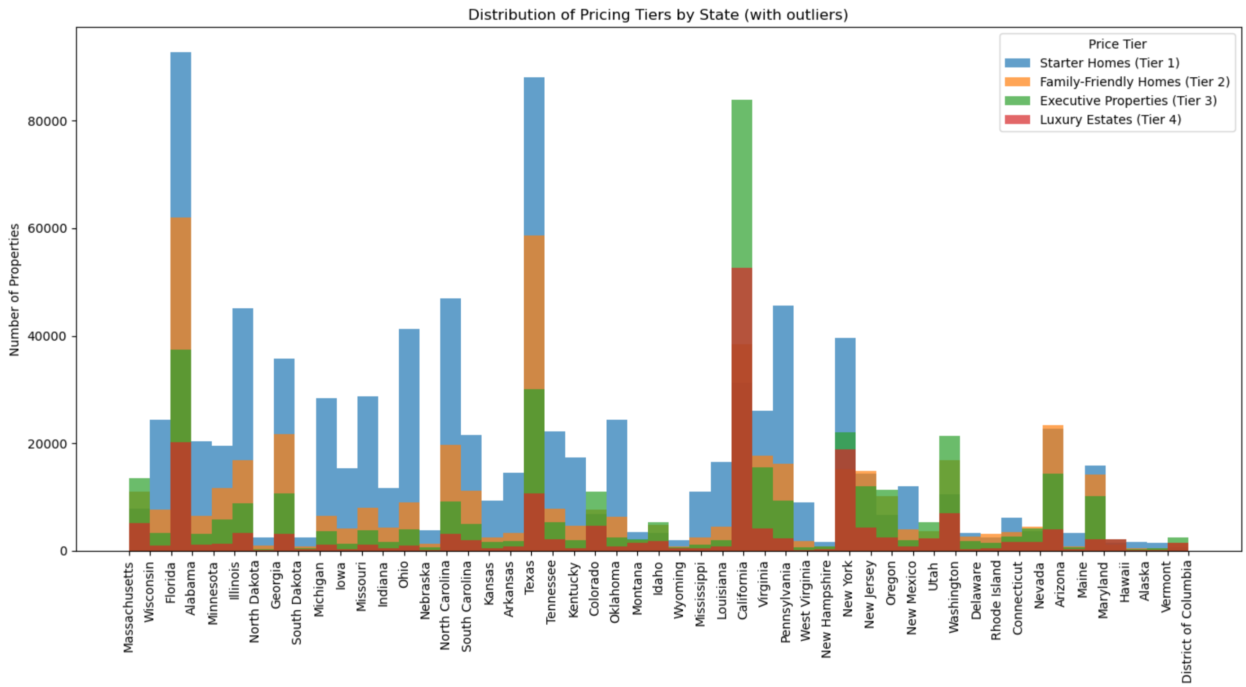

```
# Create a figure and axis objects with outliers removed
fig, ax = plt.subplots(figsize=(14, 8))

# Iterate over each pricing tier
for tier in df_no_outliers['Pricing Tier'].unique():
    # Filter data for the current pricing tier
    tier_data = df_no_outliers[df_no_outliers['Pricing Tier'] ==
tier].copy()

    # Convert 'state' column to string type
    tier_data['state'] = tier_data['state'].astype(str)

    # Plot histogram for the current pricing tier
    ax.hist(tier_data['state'],
bins=len(df_no_outliers['state'].unique()), alpha=0.7, label=tier,
stacked=True)

# Set labels and title
```

```
ax.set_xlabel('State')
ax.set_ylabel('Number of Properties')
ax.set_title('Distribution of Pricing Tiers by State (no outliers)')
ax.legend(title='Pricing Tier')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

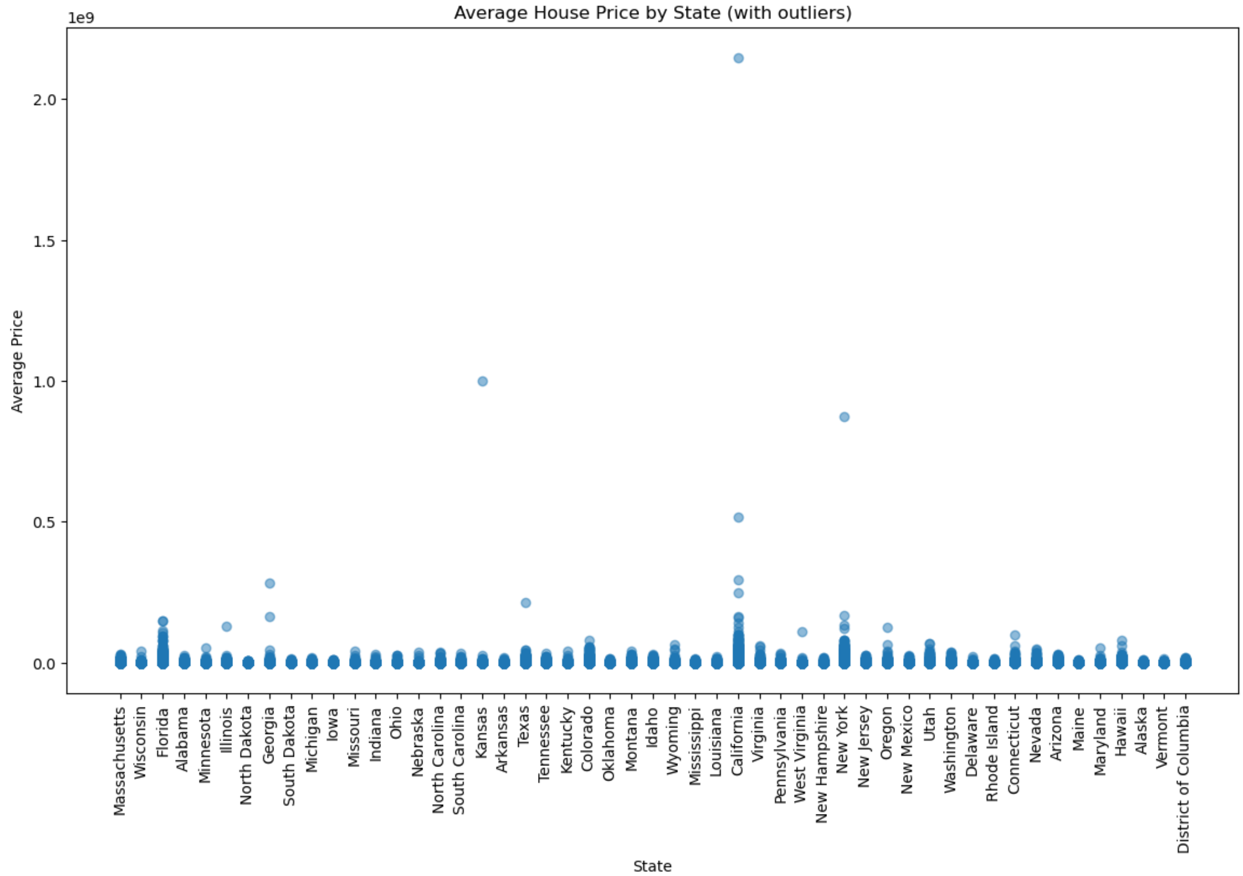



```
# Create a figure and axis objects with outliers
fig, ax = plt.subplots(figsize=(14, 8))

# Iterate over each pricing tier
for tier in df_cleaned['Pricing Tier'].unique():
    # Filter data for the current pricing tier
    tier_data = df_cleaned[df_cleaned['Pricing Tier'] == tier].copy()

    # Convert 'state' column to string type
    tier_data['state'] = tier_data['state'].astype(str)

    # Plot histogram for the current pricing tier
    ax.hist(tier_data['state'],
bins=len(df_cleaned['state'].unique()), alpha=0.7, label=tier,
stacked=True)

# Set labels and title
ax.set_xlabel('State')
ax.set_ylabel('Number of Properties')
ax.set_title('Distribution of Pricing Tiers by State (with outliers)')
```

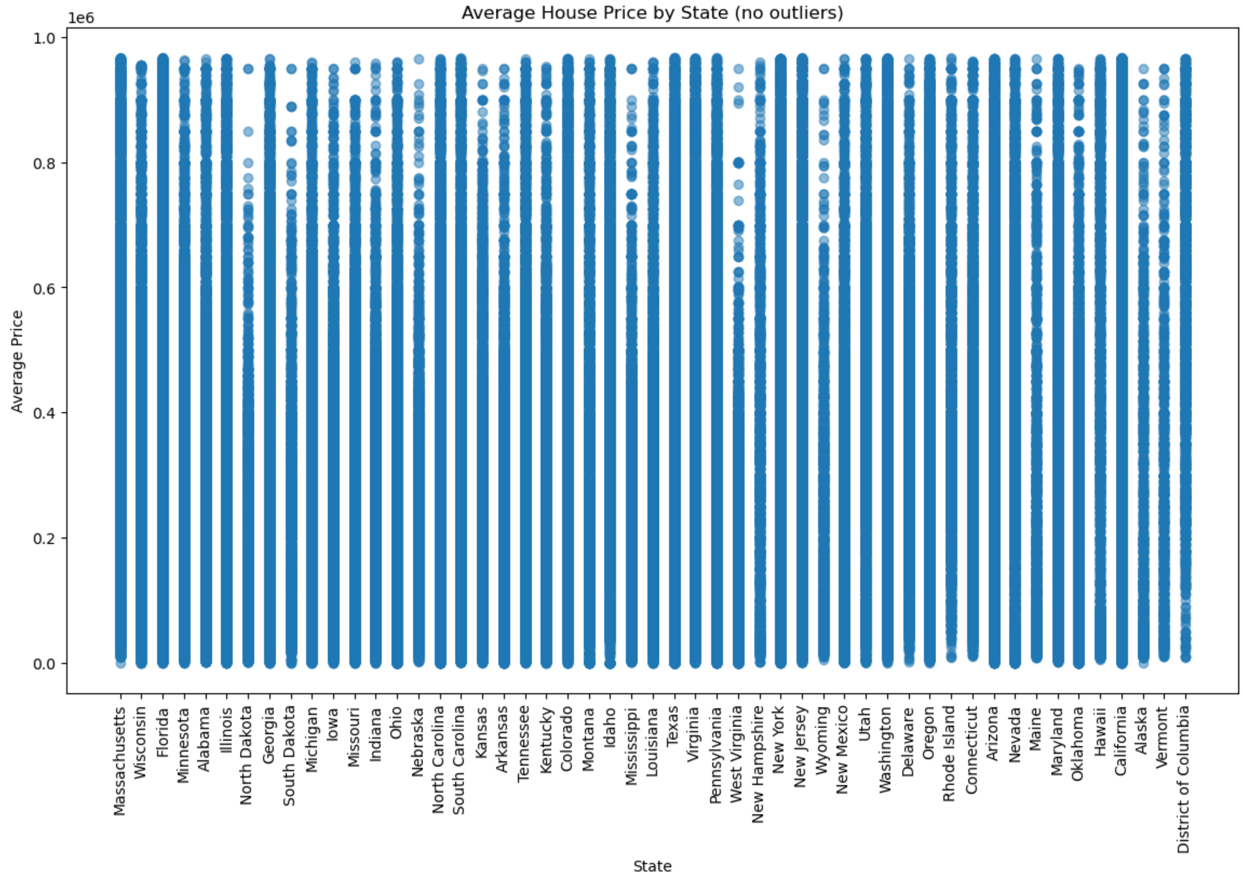
```python
ax.legend(title='Price Tier')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```




```python
# Group by state and calculate the average price
average = df_cleaned.groupby('state')['price'].mean().reset_index()

# Convert 'state' column to strings with outliers removed
df_cleaned['state'] = df_cleaned['state'].astype(str)

# Explore scatterplot
# Plot scatter plot of state locations
plt.figure(figsize=(14, 8))
plt.scatter(df_cleaned['state'], df_cleaned['price'], alpha=0.5)
plt.title('Average House Price by State (with outliers)')
plt.xlabel('State')
plt.ylabel('Average Price')
plt.xticks(rotation=90)
plt.show()
```




```
# Group by state and calculate the average price
average = df_no_outliers.groupby('state')
['price'].mean().reset_index()

# Convert 'state' column to strings with outliers removed
df_no_outliers['state'] = df_no_outliers['state'].astype(str)

# Explore scatterplot
# Plot scatter plot of state locations
plt.figure(figsize=(14, 8))
plt.scatter(df_no_outliers['state'], df_no_outliers['price'],
alpha=0.5)
plt.title('Average House Price by State (no outliers)')
plt.xlabel('State')
plt.ylabel('Average Price')
plt.xticks(rotation=90)
plt.show()
```



Inferential Statistics

```
# Columns for Chi-Square test
columns_for_chi2 = numerical_columns + categorical_columns

# Create a contingency table for each column
contingency_tables = {}
for col in columns_for_chi2:
    contingency_tables[col] = pd.crosstab(df_cleaned[col],
df_cleaned['Pricing Tier'])

# Perform Chi-Square tests
alpha = 0.05
significant_results = {}
for col, table in contingency_tables.items():
    chi2, p_value, _, _ = chi2_contingency(table)
    significant_results[col] = p_value < alpha

# Print results
for col, significant in significant_results.items():
    if significant:
        print(f"Chi-Square test for {col}: Significant association
(reject null hypothesis) - Pricing Tier and {col} are associated.")
```

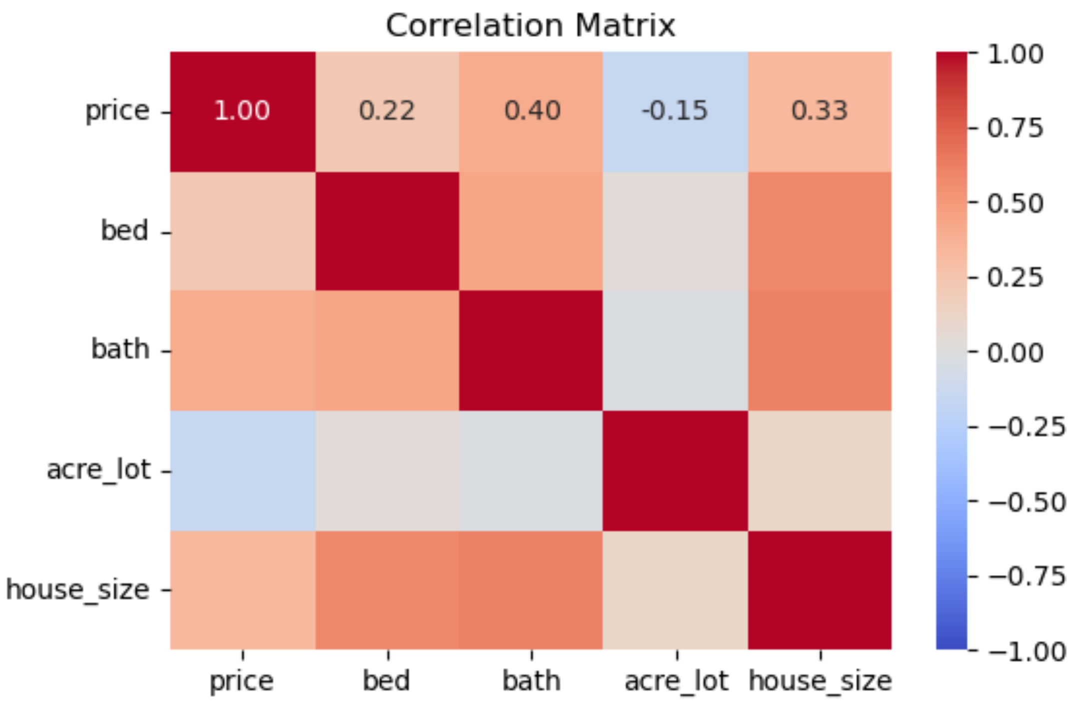
```
    else:
        print(f"Chi-Square test for {col}: No significant association (fail to reject null hypothesis) - Pricing Tier and {col} are independent.")
```

```
Chi-Square test for bed: Significant association (reject null hypothesis) - Pricing Tier and bed are associated.
Chi-Square test for bath: Significant association (reject null hypothesis) - Pricing Tier and bath are associated.
Chi-Square test for acre_lot: Significant association (reject null hypothesis) - Pricing Tier and acre_lot are associated.
Chi-Square test for house_size: Significant association (reject null hypothesis) - Pricing Tier and house_size are associated.
Chi-Square test for price: Significant association (reject null hypothesis) - Pricing Tier and price are associated.
Chi-Square test for city: Significant association (reject null hypothesis) - Pricing Tier and city are associated.
Chi-Square test for state: Significant association (reject null hypothesis) - Pricing Tier and state are associated.
```

Feature Selection

```
# Select only numeric columns for correlation analysis
numeric_columns = df_no_outliers.select_dtypes(include=np.number).columns
df_numeric = df_no_outliers[numeric_columns]

# Calculate the correlation matrix
correlation_matrix = df_numeric.corr()

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", vmin=-1, vmax=1)
plt.title('Correlation Matrix')
plt.show()

# Determine the features with the highest absolute correlation coefficients with the target variable
target = 'price'  # Assuming this is the target variable
highest_correlation_features = correlation_matrix[target].abs().nlargest(4).index[1:]  # Exclude the target itself

# Print out the features with the highest absolute correlation coefficients
print("\nTop 3 features with highest absolute correlation coefficients:")
for feature in highest_correlation_features:
```

```
    correlation = correlation_matrix.loc[feature, target]
    print(f"{feature}: {correlation:.2f}")
```




```
Top 3 features with highest absolute correlation coefficients:
bath: 0.40
house_size: 0.33
bed: 0.22
```

Data Split

```
# Separate features and target variable
X_numerical = df_no_outliers[['bath', 'bed', 'acre_lot',
'house_size']]
X_categorical = df_no_outliers[['city', 'state']]

# Apply label encoding to target variable
le = LabelEncoder()
y = le.fit_transform(df_no_outliers['Pricing Tier'])

# Encode categorical variables
label_encoders = {}
for column in X_categorical.columns:
    label_encoders[column] = LabelEncoder()
    X_categorical.loc[:, column] =
label_encoders[column].fit_transform(X_categorical[column])
```

```python
# Concatenate numerical and categorical features
X = pd.concat([X_numerical, X_categorical], axis=1)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Print the shapes of the resulting datasets
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (1105604, 6) (1105604,)
Testing set shape: (276402, 6) (276402,)
```

Model Selection and Analysis

Random Forest Classifier

```python
# Train Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Make predictions on the test data
rf_predictions = rf.predict(X_test)

# Calculate predicted probabilities for each class
rf_probs = rf.predict_proba(X_test)

# Evaluate Random Forest Classifier
accuracy_rf = accuracy_score(y_test, rf_predictions)
print("Random Forest Classifier Accuracy:", accuracy_rf)
```

```
Random Forest Classifier Accuracy: 0.7598497840102459
```

```python
# Evaluate and Analyze Precision, Recall, F1-Score, Support and
Accuracy for Random Forest Classifier
# Calculate precision, recall, F1-score, and support
print("Classification Report for Random Forest Classifier:")
print(classification_report(y_test, rf_predictions))

# Calculate AUC-ROC for multi-class classification (OvR strategy)
auc_roc = roc_auc_score(y_test, rf_probs, multi_class='ovr')
print("AUC-ROC for Random Forest Classifier:", auc_roc)
```

```
Classification Report for Random Forest Classifier:
              precision    recall  f1-score   support

           0       0.71      0.66      0.69     51095
           1       0.64      0.62      0.63     81491
           2       0.84      0.87      0.85    143816
```

```
    accuracy                           0.76    276402
   macro avg       0.73      0.72      0.72    276402
weighted avg       0.76      0.76      0.76    276402

AUC-ROC for Random Forest Classifier: 0.8939799303508766
```

Gradient Boosting Classifier

```python
# Train Gradient Boosting Classifier
gbc = GradientBoostingClassifier(n_estimators=100, random_state=42)
gbc.fit(X_train, y_train)

# Make predictions on the test data
gbc_predictions = gbc.predict(X_test)

# Evaluate Gradient Boosting Classifier
accuracy_gbc = accuracy_score(y_test, gbc_predictions)
print("Gradient Boosting Classifier Accuracy:", accuracy_gbc)
```

```
Gradient Boosting Classifier Accuracy: 0.682133269657962
```

```python
# Evaluate and Analyze Precision, Recall, F1-Score, Support and
Accuracy for Gradient Boosting Classifier
# Calculate precision, recall, F1-score, and support
print("Classification Report for Gradient Boosting Classifier:")
print(classification_report(y_test, gbc_predictions))
```

```
Classification Report for Gradient Boosting Classifier:
              precision    recall  f1-score   support

           0       0.63      0.45      0.53     51095
           1       0.54      0.49      0.51     81491
           2       0.75      0.87      0.81    143816

    accuracy                           0.68    276402
   macro avg       0.64      0.60      0.62    276402
weighted avg       0.67      0.68      0.67    276402
```

KNN Classifier

```python
# Train KNN Classifier
knn_classifier = KNeighborsClassifier(n_neighbors=5)  # You can adjust
the number of neighbors as needed
knn_classifier.fit(X_train, y_train)

# Evaluate KNN Classifier
y_pred_knn = knn_classifier.predict(X_test)
accuracy_knn = accuracy_score(y_test, y_pred_knn)
print("KNN Classifier Accuracy:", accuracy_knn)
```

```
KNN Classifier Accuracy: 0.7017423897077445
```

```python
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(y_test, y_pred_knn)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.59      0.62      0.61     51095
           1       0.58      0.57      0.57     81491
           2       0.81      0.81      0.81    143816

    accuracy                           0.70    276402
   macro avg       0.66      0.67      0.66    276402
weighted avg       0.70      0.70      0.70    276402
```

Logistic Regression

```python
# Train Logistic Regression Model
logistic_regression = LogisticRegression(max_iter=1000)
logistic_regression.fit(X_train, y_train)

# Evaluate Logistic Regression Model
y_pred_logistic = logistic_regression.predict(X_test)
accuracy_logistic = accuracy_score(y_test, y_pred_logistic)
print("Logistic Regression Accuracy:", accuracy_logistic)
```

```
Logistic Regression Accuracy: 0.5780565987221511
```

```python
# Generate classification report for logistic regression
report = classification_report(y_test, y_pred_logistic)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.46      0.18      0.26     51095
           1       0.43      0.31      0.36     81491
           2       0.63      0.87      0.73    143816

    accuracy                           0.58    276402
   macro avg       0.51      0.45      0.45    276402
weighted avg       0.54      0.58      0.54    276402
```

Naive Bayes

```python
# Define and train the Naive Bayes model
model_nb = GaussianNB()
```

```
model_nb.fit(X_train, y_train)

# Evaluate Naive Bayes model
y_pred_nb = model_nb.predict(X_test)
accuracy_nb = accuracy_score(y_test, y_pred_nb)
print("Naive Bayes Accuracy:", accuracy_nb)

# Generate classification report for Naive Bayes model
report = classification_report(y_test, y_pred_nb)
print(report)

# Calculate predicted probabilities for each class
model_nb_probs = model_nb.predict_proba(X_test)

# Calculate AUC-ROC for multi-class classification (OvR strategy)
model_nb_auc_roc = roc_auc_score(y_test, model_nb_probs,
multi_class='ovr')
print("AUC-ROC for Naive Bayes:", model_nb_auc_roc)
```

```
Naive Bayes Accuracy: 0.6000571631174882
              precision    recall  f1-score   support

           0       0.49      0.25      0.33     51095
           1       0.45      0.35      0.39     81491
           2       0.66      0.87      0.75    143816

    accuracy                           0.60    276402
   macro avg       0.54      0.49      0.49    276402
weighted avg       0.57      0.60      0.57    276402

AUC-ROC for Naive Bayes: 0.7162791330859447
```

Deep Learning (MLP Model)

```
# One-hot encode the target variable
y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the number of features
num_features = X_train_scaled.shape[1]

# Define the number of unique classes for classification
num_classes = len(df_no_outliers['Pricing Tier'].unique())

# Create a Sequential model
model_dl = Sequential([
```

```
    Input(shape=(num_features,)),  # Input layer
    Dense(units=64, activation='relu'),  # Hidden layer
    Dense(units=32, activation='relu'),  # Hidden layer
    Dropout(0.3),  # Dropout layer
    Dense(units=num_classes, activation='softmax')  # Output layer for
classification
])

# Compile the MLP
model_dl.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy', tf.metrics.Precision(),
tf.metrics.Recall()])

# Print a summary of the model
model_dl.summary()

# Define the early stopping criteria
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
# Fit the model to the training data
model_dl.fit(X_train_scaled, y_train_encoded, validation_split=0.2,
epochs=25, batch_size=32, callbacks=[early_stopping])

# Evaluate Deep Learning (MLP) model
loss, accuracy, precision, recall = model_dl.evaluate(X_test_scaled,
y_test_encoded, verbose=0)
f1_score = 2 * (precision * recall) / (precision + recall)

# Get the model's predictions
model_dl_pred = model_dl.predict(X_test)

# Calculate the AUC-ROC score for multi-class classification (OvR
strategy)
roc_auc = roc_auc_score(y_test_encoded, model_dl_pred,
multi_class='ovr')

print('Deep Learning (MLP) Metrics:')
print(f'loss: {loss},\naccuracy: {accuracy},\nprecision: {precision},\
nrecall: {recall},\nf1_score: {f1_score},\nroc_auc: {roc_auc}')

Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ dense_6 (Dense)                 │ (None, 64)             │
448 │
```

```
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dense_7 (Dense)                  │ (None, 32)              │
2,080 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dropout_1 (Dropout)              │ (None, 32)              │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dense_8 (Dense)                  │ (None, 3)               │
99 │
└──────────────────────────────────┴─────────────────────────┴────────
──────┘

 Total params: 2,627 (10.26 KB)

 Trainable params: 2,627 (10.26 KB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 36s 1ms/step - accuracy: 0.6142 -
loss: 0.8480 - precision_1: 0.7049 - recall_1: 0.4558 - val_accuracy:
0.6364 - val_loss: 0.7953 - val_precision_1: 0.7246 - val_recall_1:
0.4894
Epoch 2/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 35s 1ms/step - accuracy: 0.6378 -
loss: 0.8013 - precision_1: 0.7229 - recall_1: 0.4962 - val_accuracy:
0.6524 - val_loss: 0.7716 - val_precision_1: 0.7236 - val_recall_1:
0.5284
Epoch 3/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 39s 1ms/step - accuracy: 0.6476 -
loss: 0.7852 - precision_1: 0.7231 - recall_1: 0.5192 - val_accuracy:
0.6568 - val_loss: 0.7617 - val_precision_1: 0.7322 - val_recall_1:
0.5289
Epoch 4/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 35s 1ms/step - accuracy: 0.6512 -
loss: 0.7774 - precision_1: 0.7234 - recall_1: 0.5288 - val_accuracy:
0.6556 - val_loss: 0.7608 - val_precision_1: 0.7345 - val_recall_1:
0.5223
Epoch 5/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 35s 1ms/step - accuracy: 0.6514 -
loss: 0.7757 - precision_1: 0.7244 - recall_1: 0.5290 - val_accuracy:
0.6603 - val_loss: 0.7553 - val_precision_1: 0.7249 - val_recall_1:
0.5527
Epoch 6/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 41s 1ms/step - accuracy: 0.6536 -
loss: 0.7708 - precision_1: 0.7249 - recall_1: 0.5353 - val_accuracy:
0.6601 - val_loss: 0.7560 - val_precision_1: 0.7362 - val_recall_1:
```

```
0.5334
Epoch 7/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 36s 1ms/step - accuracy: 0.6539 -
loss: 0.7708 - precision_1: 0.7251 - recall_1: 0.5344 - val_accuracy:
0.6603 - val_loss: 0.7542 - val_precision_1: 0.7257 - val_recall_1:
0.5531
Epoch 8/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 39s 1ms/step - accuracy: 0.6560 -
loss: 0.7663 - precision_1: 0.7264 - recall_1: 0.5398 - val_accuracy:
0.6627 - val_loss: 0.7504 - val_precision_1: 0.7336 - val_recall_1:
0.5447
Epoch 9/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 32s 1ms/step - accuracy: 0.6560 -
loss: 0.7665 - precision_1: 0.7280 - recall_1: 0.5381 - val_accuracy:
0.6636 - val_loss: 0.7487 - val_precision_1: 0.7371 - val_recall_1:
0.5431
Epoch 10/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 30s 1ms/step - accuracy: 0.6569 -
loss: 0.7644 - precision_1: 0.7295 - recall_1: 0.5379 - val_accuracy:
0.6642 - val_loss: 0.7464 - val_precision_1: 0.7335 - val_recall_1:
0.5516
Epoch 11/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 25s 903us/step - accuracy: 0.6577 -
loss: 0.7640 - precision_1: 0.7302 - recall_1: 0.5387 - val_accuracy:
0.6632 - val_loss: 0.7438 - val_precision_1: 0.7371 - val_recall_1:
0.5466
Epoch 12/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 26s 931us/step - accuracy: 0.6582 -
loss: 0.7624 - precision_1: 0.7317 - recall_1: 0.5388 - val_accuracy:
0.6647 - val_loss: 0.7436 - val_precision_1: 0.7263 - val_recall_1:
0.5623
Epoch 13/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 38s 1ms/step - accuracy: 0.6586 -
loss: 0.7614 - precision_1: 0.7319 - recall_1: 0.5388 - val_accuracy:
0.6673 - val_loss: 0.7417 - val_precision_1: 0.7397 - val_recall_1:
0.5498
Epoch 14/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 34s 1ms/step - accuracy: 0.6585 -
loss: 0.7605 - precision_1: 0.7318 - recall_1: 0.5399 - val_accuracy:
0.6670 - val_loss: 0.7391 - val_precision_1: 0.7351 - val_recall_1:
0.5594
Epoch 15/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 32s 1ms/step - accuracy: 0.6591 -
loss: 0.7596 - precision_1: 0.7328 - recall_1: 0.5412 - val_accuracy:
0.6679 - val_loss: 0.7410 - val_precision_1: 0.7333 - val_recall_1:
0.5625
Epoch 16/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 36s 1ms/step - accuracy: 0.6600 -
loss: 0.7579 - precision_1: 0.7332 - recall_1: 0.5408 - val_accuracy:
```

```
0.6688 - val_loss: 0.7371 - val_precision_1: 0.7356 - val_recall_1:
0.5639
Epoch 17/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 39s 1ms/step - accuracy: 0.6603 -
loss: 0.7578 - precision_1: 0.7342 - recall_1: 0.5402 - val_accuracy:
0.6664 - val_loss: 0.7369 - val_precision_1: 0.7289 - val_recall_1:
0.5712
Epoch 18/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 37s 1ms/step - accuracy: 0.6603 -
loss: 0.7577 - precision_1: 0.7351 - recall_1: 0.5398 - val_accuracy:
0.6677 - val_loss: 0.7381 - val_precision_1: 0.7350 - val_recall_1:
0.5602
Epoch 19/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 37s 1ms/step - accuracy: 0.6612 -
loss: 0.7558 - precision_1: 0.7347 - recall_1: 0.5419 - val_accuracy:
0.6636 - val_loss: 0.7398 - val_precision_1: 0.7399 - val_recall_1:
0.5464
Epoch 20/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 36s 1ms/step - accuracy: 0.6612 -
loss: 0.7559 - precision_1: 0.7360 - recall_1: 0.5431 - val_accuracy:
0.6652 - val_loss: 0.7360 - val_precision_1: 0.7295 - val_recall_1:
0.5664
Epoch 21/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 28s 993us/step - accuracy: 0.6610 -
loss: 0.7551 - precision_1: 0.7356 - recall_1: 0.5406 - val_accuracy:
0.6706 - val_loss: 0.7322 - val_precision_1: 0.7428 - val_recall_1:
0.5563
Epoch 22/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 26s 953us/step - accuracy: 0.6623 -
loss: 0.7546 - precision_1: 0.7360 - recall_1: 0.5431 - val_accuracy:
0.6704 - val_loss: 0.7347 - val_precision_1: 0.7415 - val_recall_1:
0.5578
Epoch 23/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 26s 948us/step - accuracy: 0.6619 -
loss: 0.7534 - precision_1: 0.7348 - recall_1: 0.5436 - val_accuracy:
0.6700 - val_loss: 0.7337 - val_precision_1: 0.7470 - val_recall_1:
0.5480
Epoch 24/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 27s 959us/step - accuracy: 0.6638 -
loss: 0.7532 - precision_1: 0.7370 - recall_1: 0.5452 - val_accuracy:
0.6723 - val_loss: 0.7281 - val_precision_1: 0.7477 - val_recall_1:
0.5549
Epoch 25/25
27641/27641 ━━━━━━━━━━━━━━━━━━━━ 27s 962us/step - accuracy: 0.6640 -
loss: 0.7510 - precision_1: 0.7368 - recall_1: 0.5454 - val_accuracy:
0.6741 - val_loss: 0.7278 - val_precision_1: 0.7539 - val_recall_1:
0.5454
8638/8638 ━━━━━━━━━━━━━━━━━━━━ 5s 592us/step
Deep Learning (MLP) Metrics:
```

```
loss: 0.7291128635406494,
accuracy: 0.6737794876098633,
precision: 0.7538483142852783,
recall: 0.545712411403656,
f1_score: 0.633112979315551,
roc_auc: 0.5003207175046559
```

Model Summary

```
# Define model names
models = ['Random Forest', 'Gradient Boosting', 'KNN', 'Logistic
Regression', 'Naive Bayes', 'Deep Learning (MLP)']

# Precision, Recall, F1-score, Support, and AUC-ROC for each model
precision = [0.71, 0.63, 0.58, 0.43, 0.45, 0.75]  # Example precision
values
recall = [0.66, 0.49, 0.57, 0.31, 0.35, 0.87]  # Example recall values
f1_score = [0.69, 0.51, 0.57, 0.36, 0.39, 0.81]  # Example F1-score
values
auc_roc = [0.89, 0.72, 0.66, 0.45, 0.49, 0.50]  # Example AUC-ROC
values

# Plotting
fig, ax = plt.subplots(figsize=(12, 8))

# Metrics to plot
metrics = ['Precision', 'Recall', 'F1-Score', 'AUC-ROC']
metric_scores = [precision, recall, f1_score, auc_roc]

# Colors for bars
colors = ['navy', 'darkorange', 'darkgreen', 'darkred']

# Bar width
bar_width = 0.15

# X-axis positions for bars
x = np.arange(len(models))

for i, metric in enumerate(metrics):
    ax.bar(x - 1.5*bar_width + i*bar_width, metric_scores[i],
color=colors[i], width=bar_width, label=metric)
    for j, value in enumerate(metric_scores[i]):
        ax.text(x[j] - 1.5*bar_width + i*bar_width, value + 0.01,
f'{value:.0%}', ha='center', va='bottom', fontsize=8)

# Set labels and title
ax.set_xlabel('Models')
ax.set_ylabel('Scores')
ax.set_title('Model Metrics Comparison')
ax.set_xticks(x)
```

```
ax.set_xticklabels(models, rotation=45)
ax.yaxis.set_major_formatter(plt.FuncFormatter(lambda x, _:
f'{x:.0%}'))  # Format y-axis as percentage
ax.legend(loc='upper center', bbox_to_anchor=(0.65, 0.9), shadow=True,
ncol=2)

plt.tight_layout()
plt.show()
```