AIRBNB RENT PREDICTION MODEL BASED ON LARGE DATA

A SENIOR PROJECT

SUBMITTED TO THE COLLEGE OF ENGINEERING

OF TENNESSEE STATE UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Eulice Winston IV

Nicholas Stepka

DECEMBER 2023

To the Department of Computer Science,

We are submitting a senior project by ”Eulice Winston IV” and ”Nicholas Stepka” entitled ”Airbnb Rent Prediction Model Based in Large Data”. We recommend that it be accepted in partial fulfillment of the requirements for the degree, Bachelor of Science in Computer Science.

Technical Advisor, Dr. Manar Samad

Course Instructor, Dr. Ali Sekmen

Department Chair, Dr. Tamara Rogers

Accepted for the College of Engineering:

Dr. Lin Li

Dean of the College of Engineering

ABSTRACT

This study explores the application of machine learning algorithms to establish fair rental prices in the dynamic real estate market, focusing on Airbnb listings in Nashville. Given the challenge of determining reasonable rental prices for both owners and renters, we employ a comprehensive approach combining data preprocessing, advanced regression modeling, and causal inference. Our methodology includes analyzing a large dataset using various machine learning models, with the HistGradientBoosting Regressor identified as the most effective for predicting rental prices. We emphasize feature importance and reduction to enhance computational efficiency while maintaining statistical accuracy.

The research further delves into causal analysis, utilizing state-of-the-art algorithms and domain expertise to discern the impact of specific property attributes on rental pricing, particularly examining the role of 'accommodates' in influencing price. The study employs rigorous empirical testing, including refutation methods, to validate the causal relationships and ensure the robustness of our findings.

The culmination of this project is a user-friendly web application developed using Python Flask, enabling users to input property features and receive predicted rental prices along with relevant metrics. This tool serves as a practical solution for property owners, renters, and investors in the Airbnb market, aiding in informed decision-making. Our study demonstrates the effectiveness of integrating machine learning, causal inference, and domain knowledge to address complex challenges in the real estate sector.

In terms of results, our HistGradientBoosting regression model achieved a commendable R2 score of 0.70, significantly enhancing predictive accuracy by reducing the feature set from over 200 to just 20. Additionally, for our causal question, we obtained a P-score of 0.98, affirming that the 'accommodates' feature significantly influences the pricing of Airbnb listings.

ACKNOWLEDGMENTS

We extend our deepest gratitude to all our professors, classmates, and family who have been responsible for our success. Their patience and trust in our capabilities have been a constant source of motivation. Your insightful direction was pivotal in steering this work to fruition, and we are profoundly thankful for his contributions to our success.

On a personal note, I, Nick Stepka, would like to express my sincere appreciation to Professor Samad for introducing me to the intricate world of causality. This exploration has not only broadened our analytical perspectives but also sparked a journey of intellectual discovery, revealing depths of thought and understanding that I had not anticipated. His mentorship has been instrumental in shaping our approach to this project and our overall academic growth, and has defined a path for my future success.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

CHAPTER 1

INTRODUCTION

This study aims to investigate the feasibility of using machine learning algorithms to establish fair rental prices in the constantly evolving real estate market. The challenge lies in the difficulty for both owners and renters to agree on reasonable rental prices, leading to missed opportunities and incorrect pricing decisions.

To address this issue, we plan to employ machine learning algorithms to analyze a large rental data set and develop a fair pricing model. The model will consider several key variables, including demographic information and property characteristics such as location, size, and amenities. By considering these variables, the model will be able to provide an accurate and scientifically validated market price for rental properties, allowing for confident rental decisions for both owners and renters.

In addition, the model will provide a quantitative importance score for each variable, allowing for further analysis and understanding of the factors that influence rental pricing. This tool will also benefit real estate investors, as they can use the model to determine the potential profitability of a property.

By utilizing state-of-the-art natural language processing techniques and machine learning algorithms, this project aims to provide a scientifically validated solution to the issue of determining fair rental prices in the constantly changing real estate market.

1.1 Need Analysis

The rental market is complex and constantly evolving, making it difficult for both renters and owners to agree on fair rental prices. There is a need for a scientifically validated pricing system that considers multiple variables such as demographic data, location, size, and amenities to provide an accurate and competitive market price for rental properties. Additionally, there is a need for a tool that can provide a quantitative importance score for each variable to help renters and owners make more informed rental decisions. The target audience for this tool includes renters, owners, and investors in the real estate market. Renters and owners will benefit from the tool as it will help them make informed rental decisions by providing them with accurate market prices and the importance score of each variable. Investors will be able to use the tool to determine the potential profitability of a property, helping them make better investment decisions.

1.2 Problem Statement

The rental market is constantly changing due to inflation. It is challenging for owners and renters to agree on fair rental prices. In the short-term, indecision leads to missed opportunities and overpaying for rent. While for owners pricing too high can lead to additional months where the property is not rented, a lower price may miss the opportunity for increased profits. This project aims to develop a fair pricing model by mining a large rental data set using machine learning (ML) algorithms. The model will consider many variables, including demographic data, to determine a competitive market price for rental properties. The model will analyze variables such as location, size, and amenities to provide accurate pricing, giving both owners and renters confidence in their rental decisions. This will additionally provide the quantitative importance score for individual variables in making the rental decision. In addition, investors in the real estate market will be able to use the tool to determine if a property on the market has the potential to have a profitable return on investment. The project will entail using state-of-the-art natural language processing (NLP) and ML techniques, providing a scientifically validated pricing system for the housing market.

1.3 Literature Review

The sharing economy has disrupted the traditional hospitality industry, and the rise of platforms such as Airbnb has created opportunities for property owners to rent their spaces and for travelers to experience unique and affordable accommodations. However, determining fair and competitive rental prices is a complex and challenging issue due to the lack of industry standards and the growing number of listings. To address this issue, machine learning (ML) and natural language processing (NLP) have emerged as powerful tools in the sharing economy to analyze data and improve pricing models.

1.3.1 Tabular Data and Airbnb Pricing

Tabular data, including tables, has been widely used in analyzing Airbnb pricing models. By leveraging tabular data, analysts can consider various factors such as location, amenities, and seasonality to develop pricing models for Airbnb rentals. Located on Kaggle [1], we will use a big data set with over 134,000 listings with over 100 variables. Through tabular data analysis, rental prices can be predicted accurately, helping property owners to set optimal prices and maximize profits.

1.3.2 Regression Model and Airbnb Pricing

Regression analysis is a statistical technique commonly used to analyze the relationship between variables in order to cause predictions [2]. By utilizing machine learning models such as linear regression, gradient boosting regression, and random Forest regression, analysts can predict rental prices accurately and identify the most significant factors that influence rental prices [5]. Furthermore, regularization and feature selection techniques can help avoid overfitting and select the most relevant features. This section will review the existing literature on the application of machine learning models for regression analysis in Airbnb pricing.

1.3.3 Different Machine Learning Models for Airbnb Pricing

According to an article by Garcia and peers, ”machine learning algorithms (ML) are increasingly being used for the mass appraisal of real estate and in automated valuation models successfully.” [8] Such models as random forest, gradient boosting, and neural networks, have been employed to predict the real estate market. In particular, a convolutional neural network has been utilized to successfully forecast rental prices based on images and text data. In an interview given to Venturebeat.com, Mike Curtis, the VP of Engineering at Airbnb, said they have implemented a machine-learned ranking model to offer predictive rankings of properties [9]. The available literature suggests that ML and natural language processing (NLP) techniques can be utilized to develop accurate and personalized pricing models for short-term rentals. Furthermore, regression analysis and tabular data can also be utilized to explore the factors that influence rental prices. Further research is required to explore the potential of different ML models and to develop more precise and personalized pricing models for the real-estate market.

1.3.4 Causality and Do Calculus in Airbnb Pricing

In addition to regression analysis, causality and do calculus, as developed by Judea Pearl, are powerful tools for analyzing Airbnb pricing models [7]. These tools have also been used to analyze hotel cancellations, as randomized controlled trials can be too costly or unethical [4] By applying causality and do calculus, analysts can determine the factors that have the most significant impact on Airbnb rental prices and develop personalized pricing models that benefit both property owners and travelers. Further research is needed to explore these tools' potential and develop more accurate and precise pricing models for the sharing economy.

1.4 Goal and Objectives

1.4.1 Research Goal

This project aims to develop a machine-learning model that accurately predicts Airbnb rental prices based on various factors, including property location, size, and amenities. By doing so, we aim to provide an effective tool for property owners to maximize their profits and for renters to make informed decisions on renting properties at fair prices.

1.4.2 Associated Objectives

The specific objectives are:

1. Collect Airbnb data from different cities across the United States, including information on property location, size, amenities, and rental prices.

2. Store collected data into structured Pandas DataFrame and separate each DataFrame for each area.

3. Get familiar with preprocessing techniques, machine/deep learning algorithms, libraries, and packages for natural language processing and regression analysis.

4. Preprocess and format the Airbnb data for feeding into the machine learning model.

5. Develop and test multiple regression models to identify the most effective model for predicting Airbnb rental prices.

6. Analyze the importance of different factors in predicting rental prices and provide a quantitative importance score for each variable.

7. Implement and fine-tune the selected regression model to achieve high accuracy and reliability in predicting rental prices.

8. Validate and evaluate the model's accuracy and effectiveness using internal and external testing datasets.

9. By achieving these objectives, we aim to provide a scientifically validated and reliable tool to determine fair and competitive rental prices for the rental market. Additionally, this tool can benefit real estate investors by providing insights into the potential profitability of properties in the market.

1.5 Project Organization

In the following chapters, we will provide a detailed overview of our project to predict Airbnb rental prices using machine learning algorithms.

- Chapter 2 will present the requirements analysis, consisting of both functional and non-functional requirements. We will outline the specific features and capabilities that our model must have, as well as any performance, usability, and security requirements.

- Chapter 3 will explain the system architecture and provide detailed designs for each model component. This will include the data collection and preprocessing steps, the feature engineering process, and the regression analysis model. We will also explain how each component interacts with one another to produce the final prediction.

- Based on the previous two chapters, Chapter 4 will provide information on the implementation of the model. We will explain the specific programming languages, libraries, and tools we will use to develop the model and describe the testing and debugging process.

- Chapter 5 will consist of the experimental results of our model, including its accuracy and reliability in predicting Airbnb rental prices. We will provide a detailed analysis of the results, including any limitations or challenges we encountered during the testing process.

- In Chapter 6, we will conclude the project and outline possible extensions or future directions for the research. We will also reflect on the project's overall success and identify any lessons learned or areas for improvement.

Overall, our project aims to provide a scientifically validated and reliable tool for the rental market to determine fair and competitive rental prices. We are confident that our structured approach will lead to a successful outcome.

CHAPTER 2

REQUIREMENTS ANALYSIS

This chapter will discuss the functional and non-functional requirements of our Airbnb rent prediction model. The functional requirements will cover the user-defined needs, the behavior of our application, and the necessary components to ensure proper functionality. The non-functional requirements explain the features we will implement to increase usability but are not required for the functionality of our application.

2.1 Functional Requirements

1. A dataset that contains more than 9000 Nashville listings of Airbnb properties to develop an understanding of how different variables affect the outcome of an Airbnb Price. It will include Airbnb-listed properties from the Nashville area.
    (a) Each row will contain a property that is listed on Airbnb.
    (b) Each column has attributes regarding property details such as price, number of beds, and customer feedback rating.
2. Structured tabular data from the database will be processed and curated.
    (a) The raw data needs to be prepared for predictive modeling.
    (b) The data must be imputed for missing values.
    (c) Not all columns will be factors to price, such as the host ID or host link, and need to be removed before training the data.
    (d) Columns presenting numeric and categorical variables will be kept and processed for predictive modeling.

3. A regression model to determine the relationships between variables.

   (a) The software shall develop and compare multiple advanced regression models to identify the most accurate model for predicting Airbnb rental prices.

   (b) The software shall learn the importance of different factors in property attributes in predicting rental prices and provide a quantitative relative importance score for each variable.

   (c) The software should provide a range of fair rental prices learning from a set of regression models.

   (d) The software shall provide a causal relationship between the property attributes and rents to provide insights into the cause and effect related to rental price rise and drop.

4. A graphical user app is to be created to allow user input.

   (a) The user should be able to access the software using a simple mouse click and see the application load.

   (b) The user will be able to enter property attributes to obtain a range of fair prices for a property.

   (c) When the user submits the information, the software should show what attribute played a major role in determining the rent.

5. Perform a comparison of various regression and causality tests to determine the most accurate method, presenting the results in a table.

   (a) The software shall evaluate different regression and causality tests, including their respective Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Squared Error (MSE), R-Squared, and Adjusted R-Squared values.

   (b) The comparison results will be presented in a table format, allowing for easy identification of the most accurate method for predicting rental prices.

(c) By identifying the most accurate method, the software ensures the highest quality model is utilized for rental price predictions.

2.2 Non-Functional Requirements

1. The software application will be developed and debugged on Jupyter notebook, but the final product will run using an equivalent python script.

   Rationale: The Python programming language has one of the largest communities because of the public libraries/packages available relating to Machine Learning algorithms.

2. Ensuring an R2 score above .70 (high)

   Rationale: Setting an R2 score above .70 will ensure accurate rental price predictions. This threshold significantly improves over the baseline and reduces pricing prediction errors.

3. The system should run on any desktop environment that runs python.

   Rationale: To make the software easily accessible, the application must be able to run in today's most common operating systems.

4. The end user needs to be able to enter property details to get a price result.

   Rationale: While it is good to prove the model is successful with a high R2 correlation, we need to make an application that the end user can easily enter in details to give back a prediction.

5. Acceptable confidence interval X +/ 95% sd.

   Rationale: The program's requirement to estimate rental prices within an acceptable confidence interval X +/- 95% sd ensures precise and reliable predictions, improving customer satisfaction and enabling property owners to set appropriate rental prices.

6. Regression and Causality Comparison Results

   Rationale: Comparing different regression and causality tests helps identify the most accurate method for predicting rental prices, ensuring the highest quality model is utilized.

CHAPTER 3

DESIGN

This chapter outlines our data-driven rental pricing model's system architecture and detailed design. The model will be developed using machine learning algorithms, focusing on data collection, data preprocessing, and model training and testing. In addition, we will be exploring causality by incorporating Microsoft Dowhy library to demonstrate causal discovery into our model.

3.1 System Architecture

Our rental price prediction model starts with data gathering and processing. First, we found a data set with around 134,000 Airbnb listings stored in a Pandas DataFrame containing property location, size, amenities, and rental prices.

The next subsystem handles data cleansing and preprocessing. We will remove irrelevant columns, address missing values, and convert categorical variables into numerical values. Feature engineering will create new features from raw data and select the most relevant ones for prediction. The cleaned data is fed into machine learning algorithms, such as linear regression and gradient-boosting tree regression. Hyperparameter tuning will optimize the model's performance.

The causal discovery analysis component will explore relationships between property attributes and rental prices, identifying causal factors influencing rental prices and providing valuable insights. While this method does use linear regression testing in a step, it goes beyond correlation to model the data to understand causal relations.

The final component is the user interface and visualization, offering an accessible platform for users to input property data and receive predictions and insights about rental prices. Visualization techniques like bar charts, scatter plots, and heatmaps will present the results intuitively and appealingly. In addition, as a matter of scientific exploration of knowledge and highlighting the difference between correlation and causation, the app will break down each prediction model to give the user the best context to give confidence to the results.

In summary, our rental price prediction model comprises interconnected subsystems providing an efficient and accurate solution for predicting Airbnb rental prices. The process includes data gathering and processing, preprocessing, feature engineering, machine learning algorithms, causal discovery analysis, user interface, and visualization components. In addition, using traditional correlation models and the new science of causality, we can demonstrate how data science attempts to move beyond correlation and understand the causality of results to make better decisions.

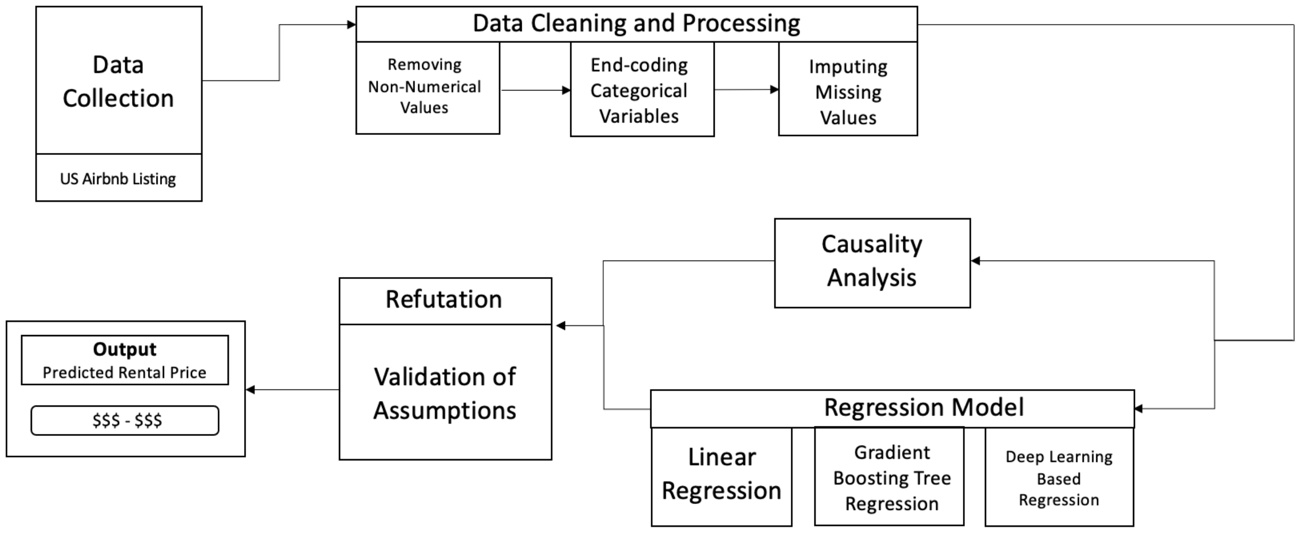

Data Collection
US Airbnb Listing
Data Cleaning and Processing
Removing Non-Numerical Values
End-coding Categorical Variables
Imputing Missing Values
Causality Analysis
Refutation
Validation of Assumptions
Output
Predicted Rental Price
$$$ - $$$
Regression Model
Linear Regression
Gradient Boosting Tree Regression
Deep Learning Based Regression

Figure 3.1: Data processing model.

3.2 Detailed Design

The system's design is based on a modular approach, ensuring that each component can be independently developed, tested, and optimized. This structure allows for easy modification and expansion of the system as new data sources and algorithms become available. The design encompasses the following subsections:

3.2.1 Data Collection

Data collection involves gathering Airbnb listings from various cities across the United States. The data set is a subset of all Airbnb listing in and is stored as raw data. The data set will include information on property location, size, amenities, and rental prices. We will extract this data from a publicly available website for local storage for analysis.

3.2.2 Data Cleansing

In this step, we will preprocess and clean the raw data. This involves imputing missing values, removing irrelevant columns, and converting categorical variables into numerical values. For example, removing non-numerical columns and finding important variables/features in the entire data set. This code will create binary columns for each unique amenity in your 'Amenities' column and merge them back into the original data frame. Each row will have a "1" if the amenity is present and a "0" if not. It will be vital to avoid the curse of dimensionality, which means it will be important to understand feature selection and pre-processing to build an enhanced model [10]. Data cleansing ensures that the machine learning models can effectively process the data and provide accurate predictions.

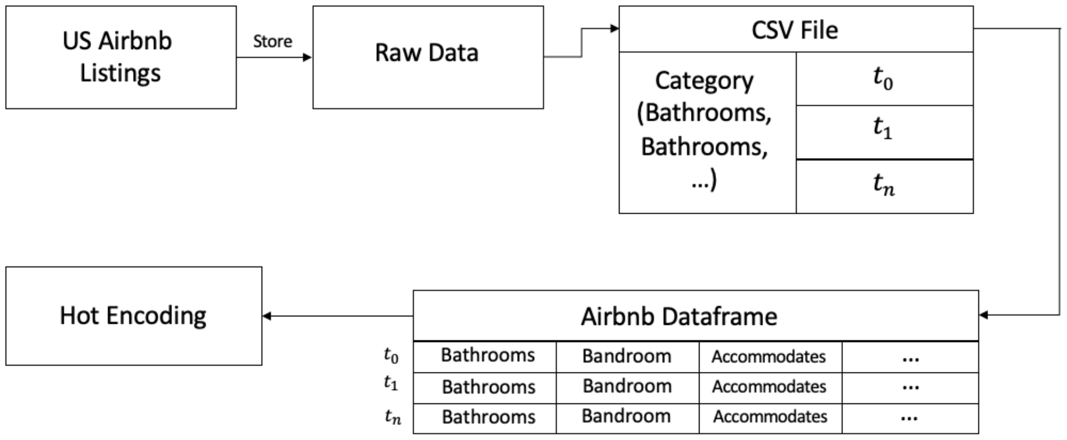


Figure 3.2: Data cleaning model.

3.2.3 Causal Discovery Analysis

Causal discovery analysis allows us to identify the factors that have the most significant impact on Airbnb rental prices by addressing confounding issues. Confounding issues arise when a variable influences the predictor and the outcome, leading to spurious correlations and potentially biased results. By employing the dowhy library, we will apply causality and do-calculus techniques to determine the causal relationships between property attributes and rental prices while accounting for potential confounders.

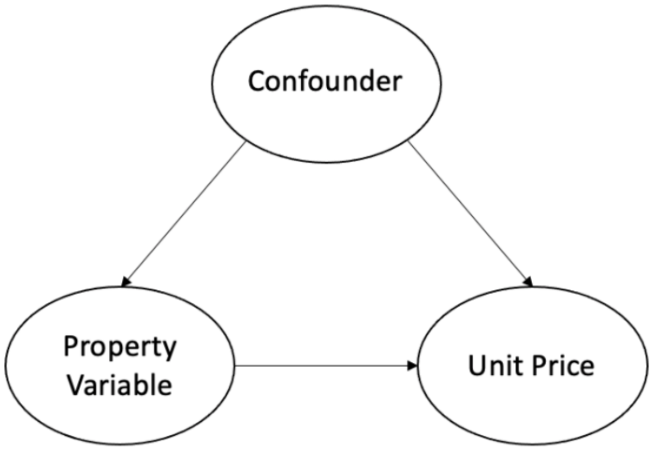


Figure 3.3: Directed acyclic airbnb causality model.

Addressing confounding issues is crucial for reliable and accurate causal insights; according to Judea Pearl in The Book of Why, he explains that counterfactual analysis will allow scientists to make more precise and definite statements than before [3]. We will identify potential confounding variables affecting property attributes and rental prices to achieve this. First, we can represent the variables' underlying causal structure using existing causal analysis graphical tools. Then, by adjusting the confounding factors, we can estimate the causal effects of the property attributes on rental prices.

3.2.4 Linear Regression Testing

Linear regression analysis is used to predict the value of a variable based on the value of another variable. We will develop a linear regression model to analyze the relationship between property attributes and rental prices. This model will be used as a baseline to compare the performance of other machine learning algorithms. We will evaluate the model's accuracy using metrics such as R2 score and mean squared error.

3.2.5 Decision Tree Regression Testing

Decision tree regression is a type of machine learning algorithm that recursively splits the data into subsets based on the most informative features, creating a tree-like structure. In this section, we will implement a decision tree regression model and compare its performance to the linear regression model. We will also fine-tune the model's hyperparameters to achieve the best possible accuracy.

3.2.6 Gradient Boosting Tree Regression Testing

Gradient boosting tree regression is a powerful machine learning algorithm that combines multiple weak learners to create a more accurate model. This section will implement a gradient-boosting tree regression model and compare its performance to the linear regression model. In addition, we will fine-tune the model's hyperparameters to achieve the best possible accuracy.

3.2.7 HistGradientBoostingRegressor Testing

HistGradientBoostingRegressor is a histogram-based gradient boosting algorithm available in the scikit-learn library. It is designed for large-scale data sets and is an efficient alternative to the traditional gradient boosting method. In this section, we will implement a HistGradientBoostingRegressor model and compare its performance to the linear regression, decision tree, gradient boosting tree, random forest, and XGBoost models. We will also fine-tune the model's hyperparameters to achieve the best possible accuracy.

3.2.8 Random Forest Regression Testing

Random forest regression is an ensemble learning method that combines multiple decision trees to improve prediction accuracy and reduce over fitting. In this section, we will implement a random forest regression model and compare its performance to the linear regression, decision tree, and gradient boosting tree models. We will also optimize the model's hyperparameters to obtain the highest accuracy.

3.2.9 XGBoost Regression Testing

XGBoost (eXtreme Gradient Boosting) is an optimized gradient boosting library designed for high-performance machine learning tasks. In this section, we will implement an XGBoost regression model and compare its performance to the linear regression, decision tree, gradient boosting tree, and random forest models. Furthermore, we will fine-tune the model's hyperparameters to achieve the best possible accuracy.

CHAPTER 4

IMPLEMENTATION

In this chapter, we delve into the design and implementation of our Airbnb pricing analysis. The process begins with the preprocessing and categorization of Airbnb listing data. This involves handling missing data, categorizing amenities, and converting certain categorical variables into numerical format. Subsequently, the chapter outlines the steps to download and prepare the Nashville Airbnb dataset. Two primary analysis techniques, traditional statistical methods and causal inference models, are employed on this dataset. For a comprehensive analysis, various amenities are grouped, and their potential influence on listing prices is investigated. Lastly, a causal model is used to understand the true impact of these amenities on price, taking into account confounding variables.

4.1 Software Environments

Visual Studio Code (VS Code) was the primary tool of choice throughout this project, with its seamless support for Jupyter notebooks and Python scripting. The following specifics were adhered to:

For the majority of the modeling, Python version 3.9.17 was employed. Due to library dependencies, Python version 3.7.16 was utilized for the causal analysis as it necessitates an older version of numpy. Visual Studio Code offers an integrated development environment, facilitating efficient coding, debugging, and testing capabilities. Its versatility in supporting multiple Python versions made it particularly suitable for this project. VS Code's integrated terminal and extensions further streamlined the data analysis and modeling process.

4.2 Front-end Development

To ensure that the insights and results from the Airbnb pricing analysis are accessible and interactive, a decision was made to develop a local web application. Flask, a lightweight web application framework written in Python, was chosen for this purpose due to its simplicity, flexibility, and compatibility with our Python-based analytical models.

Features of the Flask Web Application:

1. Interactive Dashboard: Users can select specific amenities or features to understand their influence on listing prices.

2. Predictive Tool: Leveraging the models developed, users can input specific listing features to receive a suggested price range.

3. Simplicity: Flask provides a straightforward way to develop web applications, making it perfect for projects that require a simple interface with powerful backend capabilities.

4. Integration with Python: Given that our analysis and models are developed in Python, Flask seamlessly integrates with our existing codebase.

5. Development Speed: Flask's minimalistic approach allows for rapid development and testing, ensuring that the web application can be iteratively improved.

6. Local Deployment: Flask supports easy local deployment, enabling stakeholders to run the web application on their machine without any intricate setup. By integrating Flask, the intention is to provide a user-friendly interface where stakeholders can easily understand the dynamics of Airbnb listing prices in Nashville and make informed decisions accordingly.

4.3 Libraries and Tools

In this section, we will go over the general libraries, how we handle the data, visualization tools, and the models that we use for machine learning.

4.3.1 General Utilities:

1. warnings: Used to handle and suppress warnings generated during code execution.
2. os: Essential for interacting with the operating system, especially for directory manipulations.
3. re: Provides support for regular expressions, useful for data cleaning and text processing.
4. Counter from collections: Helps in counting occurrences of elements, particularly useful for categorical data analysis.

4.3.2 Data Handling and Analysis

1. pandas: Essential for data manipulation and analysis.
2. numpy: Provides support for large arrays and matrices, along with a collection of mathematical functions.

4.3.3 Visualization

1. matplotlib: Provides a wide array of tools for generating plots, histograms, and other visual representations.
2. seaborn: Built on top of matplotlib, it offers a higher-level interface for statistical graphics.

4.3.4 Modeling and Machine Learning

1. xgboost: An optimized gradient boosting library known for its performance and efficiency.

2. scikit-learn: A comprehensive library containing tools for machine learning and statistical modeling, including:

3. LinearRegression: Regression algorithms.

4. DecisionTreeRegressor, RandomForestRegressor, GradientBoostingRegressor, HistGradient-BoostingRegressor: Tree-based regression models.

   (a) DecisionTreeRegressor, RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor: Tree-based regression models.

   (b) train_test_split: For splitting datasets into training and testing sets.

   (c) r2_score, mean_squared_error, mean_absolute_error: Metrics for evaluating regression models' performance.

   (d) permutation_importance: For assessing the importance of features in the model.

   (e) KFold: Used for cross-validation.

   (f) dowhy: Provides tools for estimating causal effects using machine learning methods, including:

4.3.5 Causal Analysis:

1. dowhy: Enables causal inference by providing a unified interface for model selection, estimation, and validation.

2. While we don't use the causal discovery code in the program, there are additional libarys that will be needed to run causal discovery.

4.3.6 Others:

1. logging: Useful for generating logs, aiding in debugging and keeping track of the application's flow.

2. IPython.display: Assists in displaying objects like images directly in Jupyter notebooks or IPython environments.

3. joblib: Provides utilities for saving and loading Python objects, enhancing the reusability of large datasets or trained models.

These libraries and tools are crucial in ensuring that the analysis is robust, the models are accurate, and the application is interactive and user-friendly.

4.4 Understanding the Data and Model Building

One of the most crucial stages in a data science project is understanding the nature of the data at hand. This involves diving deep into the dataset, analyzing distributions, identifying trends, and observing relationships among variables. With a dataset downloaded it needs to be imported into our python dataframe.

4.4.1 Data Exploration

The dataset for this project, ’nashvilleDF’, contains detailed listings of properties available for rent on Airbnb in Nashville. One of the key variables of interest is the ‘price‘, which represents the nightly rate for each property. Before any form of analysis could proceed, the ‘price‘ variable needed to be cleaned and transformed into a numeric format:

4.4.2 Analysis by Room Type

A fundamental question was understanding how prices vary by the type of room or property. This can provide insights into what guests are willing to pay more for and how different property types are positioned in the Nashville market.

```
# Calculate average price for each room type
avg_prices = nashvilleDF.groupby('room_type')['price'].mean()

# Plot
```

```
plt.figure(figsize=(10,6))
avg_prices.plot(kind='bar', color=['red', 'blue', 'green', 'yellow'])
plt.ylabel('Average Price')
plt.xlabel('Room Type')
plt.title('Average Price by Room Type')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Listing 4.1: A bar chart was generated to visualize the average price for each room type.

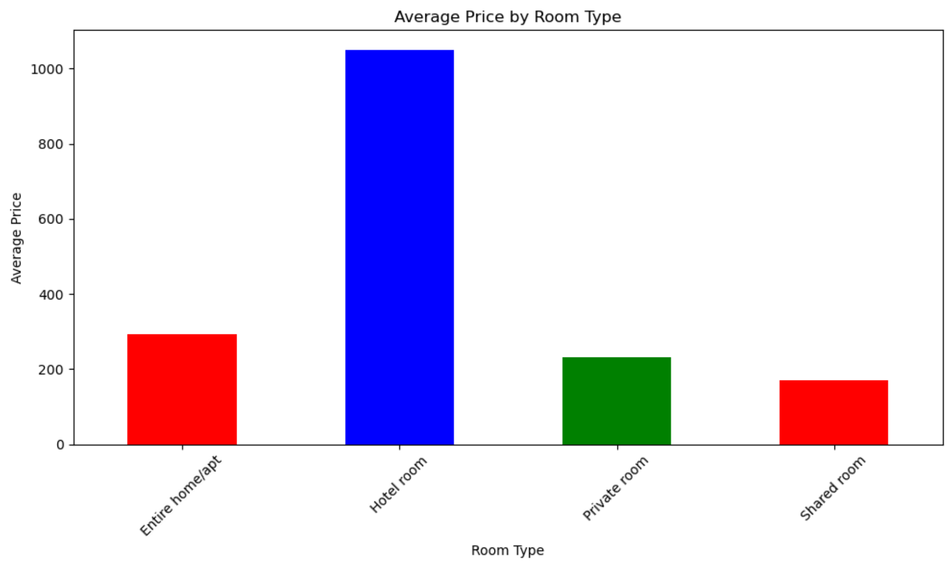


Figure 4.1: Visualizing the data to understand how different features affect price.

4.4.3 The visualization revealed four distinct categories of room types:

1. Entire home/apartment

2. Hotel room

3. Private room

4. Shared room

From the analysis, hotel rooms emerged as the most expensive on average. This observation could be attributed to the premium services and amenities often associated with hotels. The other room types also had distinct average prices, shedding light on the value perceptions of different accommodation types in the Nashville Airbnb market.

4.5 Building the Model

With a clearer understanding of the dataset's structure and the relationships between variables, the next step was to construct predictive models. The goal was to predict the price of a listing based on its features. The journey of model building, from feature selection to model evaluation, is detailed in the subsequent sections.

4.5.1 Property Type Transformation

The initial exploration of the 'property type' column, Table 4.1, revealed a myriad of categories, many of which were overlapping or closely related. The goal was to consolidate these into broader, more meaningful categories. The first observation was that numerous property types had the term "Private" in their descriptions.

```

print(nashvilleDF['property_type'].value_counts())

To streamline this, all types containing the term "Private" were grouped into a single
    "Private room" category:
private_mask = nashvilleDF['property_type'].str.contains('Private')
nashvilleDF.loc[private_mask, 'property_type'] = 'Private room'
```

Listing 4.2: Property type grouping.

4.5.2 Visualizing the Transformed Data

As shown in Figure 4.2, the 'property type' column was streamlined, and the next step was to visualize how the average price varies across these categories.

Table 4.1: Accommodation types and their counts.

Accommodation Type	Count
Entire home	3206
Entire rental unit	1642
Entire condo	1404
Entire townhouse	1010
Private room	723
Entire guest suite	304
Hotel	278
Entire guesthouse	205

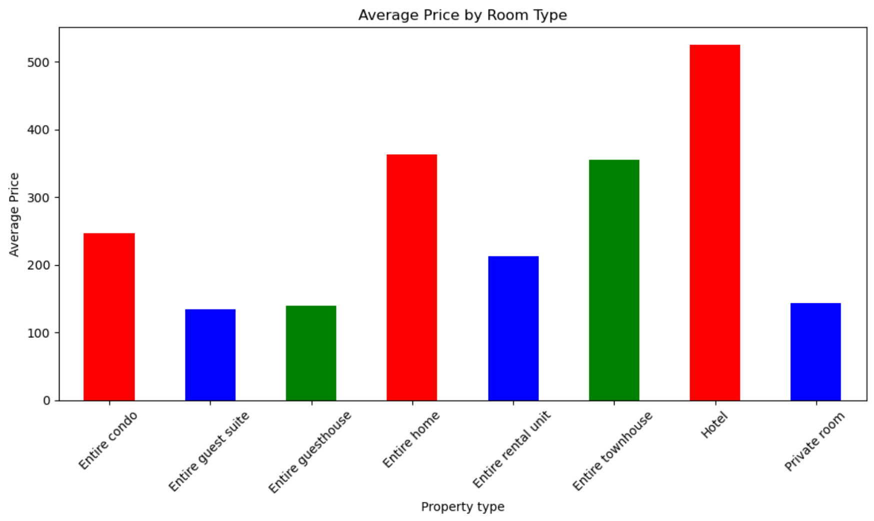


Figure 4.2: Property type vs cost.

4.5.3 Host Response Time

Shown in Table 4.2, one of the critical features in the dataset is the host's response time, which can influence a potential renter's decision.

Table 4.2: Response time by the host.

Response Time	Count
within an hour	7089
within a few hours	510
within a day	191
a few days or more	39

After validating a sizable amount of response time it is important to make this feature more model-friendly, as such the response times were encoded numerically, as shown in Table 4.3.

Table 4.3: Response time by the host.

Response Time cleaned	Count
within an hour	1
within a few hours	2
within a day	3
a few days or more	4

4.5.4 Boolean Columns Conversion

Several columns in the dataset had boolean values represented as 't' for true and 'f' for false. These columns include:

1. host is superhost
2. instant bookable
3. has availability
4. host has profile pic
5. host identity verified

```python
nashvilleDF['host_is_superhost'] = [
    1 if x == 't' else 0 for x in nashvilleDF.host_is_superhost
]
```

Listing 4.3: Example of normalizing true false values.

All these columns were converted to a binary representation where 't' was transformed to 1 and 'f' to 0, as shown in Listing 4.3. This transformation aids in the modeling process by providing a numerical representation of boolean data.

4.5.5 Handling Bathroom and Bedroom Data

The 'bathrooms text' column in the dataset contains textual information about the number of bathrooms in each property. To create a model-friendly feature, this textual information was converted to a numerical format, as shown in Listing 4.4. The numeric portion was extracted from the ’bathrooms text’ column, and the resultant values were stored in a new column called ’bathrooms’. The original ’bathrooms text’ column was then dropped to avoid redundancy.

```
nashvilleDF['bathrooms'] =
    nashvilleDF['bathrooms_text'].str.extract('(\d+\.?\d*)').astype(float)
# Drop the 'bathrooms_text' column
nashvilleDF = nashvilleDF.drop(columns=['bathrooms_text'])
```

Listing 4.4: Extract numeric portion from the ’bathrooms text’ column and assign to ’bathrooms’ column.

4.5.6 Initial Distribution of Bathrooms

The distribution of the ’bathrooms’ column in our dataset displayed a concentration of properties with between 1 and 4 bathrooms. However, certain properties reported an unusually high count, with some even reaching 18 bathrooms. This led to the suspicion of potential outliers in the data.

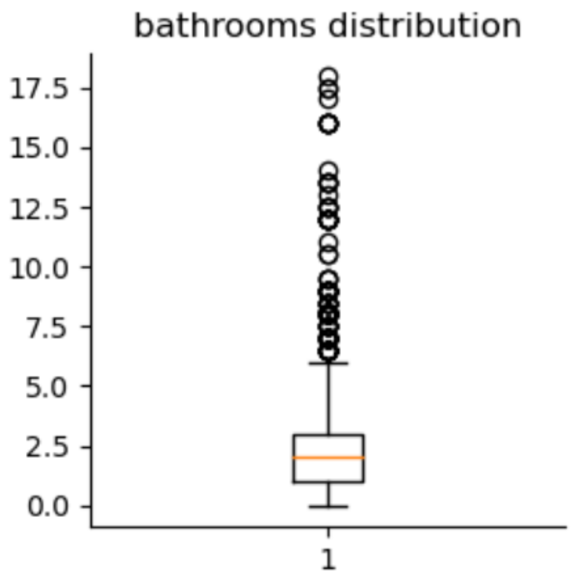


Figure 4.3: Distribution of bathrooms highlighting potential outliers.

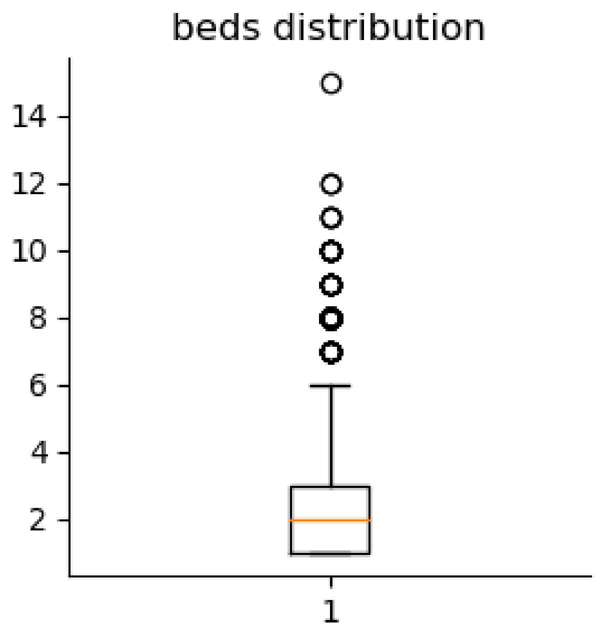


Figure 4.4: Distribution of bedrooms, for comparison.

4.5.7 Outlier Detection and Removal

To better visualize and confirm the presence of outliers in the 'bathrooms' distribution, we employed a boxplot. Further, a custom Python function, 'findOutliers', was developed to algorithmically identify and flag outliers. This function operates based on the interquartile range (IQR) method, marking data points outside the range defined by 'q75 + lims calar * iqr' as outliers. Here, 'lim scalar' is a user-defined scalar multiplier, and 'q75' represents the 75th percentile of the data.

```
def findOutliers(df, column, lim_scalar=4):
    q25, q50, q75 = df[column].quantile(q=[0.25, 0.5, 0.75])
    iqr = q75 - q25
    # Maximum value to be considered as an outlier
    max_ = q75 + lim_scalar * iqr
    # Identify the outliers
    outlier_mask = [True if x > max_ else False for x in df[column]]
    print('{} outliers found out of {} data points, {:.2f}% of the data. Max threshold is
    {}'.format(
        sum(outlier_mask), len(df[column]),
        100 * (sum(outlier_mask) / len(df[column])), max_))
    return outlier_mask
```

Listing 4.5: Function to identify outliers based on IQR method.

4.5.8 Data Pruning

Specific columns in the dataset were not relevant to the analysis or contained redundant information. These columns were thus dropped to streamline the dataset. The columns removed include URLs, names, descriptions, location-specific information, and other such columns not directly contributing to the modeling process.

4.5.9 Data Cleaning

The price column was crucial for the analysis, and its distribution was inspected. Any listings with prices outside the range of 200 to 1000 were removed to narrow down the focus to a more standard set of accommodations. Additionally, some columns contained percentage symbols or other non-numeric characters. These were cleaned, and missing values in the dataset were imputed using appropriate strategies, such as median or mean imputation.

```
nashvilleDF['price'] = nashvilleDF['price'].str.replace('$', '').str.replace(',',
    '').astype(float)
```

Listing 4.6: Example of cleaning up features.

4.5.10 Visualization of Price Distribution

After the cleaning and pruning, the distribution of the price column was visualized. The resulting histogram showed a bell-like curve, shown in Figure 4.5, indicating a standard distribution of prices in the dataset. This visualization can provide insights into the most common price range for Airbnb listings in the area.

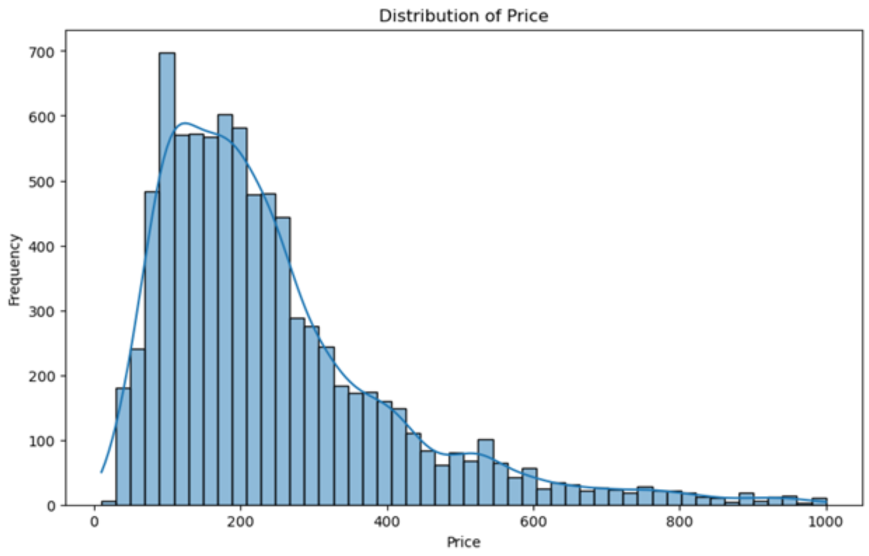


Figure 4.5: Distribution of price of all the properties.

4.5.11 Model Building and Feature Importance

The last step of our analysis involved building a model to predict Airbnb prices based on the cleaned and processed features. For this task, we chose the HistGradientBoostingRegressor, an ensemble method known for its ability to capture complex relationships and handle large datasets efficiently.

1. Cross-Validation: To ensure the robustness of our model, a 10-fold cross-validation was implemented. This approach divides the dataset into ten parts: in each iteration, nine parts are used for training, and one part is used for testing. This iterative process helps in understanding the model's average performance over different parts of the dataset and reduces the chances of overfitting to a particular subset.

2. Feature Importance: One of the major challenges in building machine learning models is understanding which features play a significant role in predictions. Using the permutation importance function, we can rank features based on their importance. This function works by randomly shuffling a feature's values and measuring how much the model's performance drops. If shuffling a feature's values causes a significant drop in performance, that feature is

deemed important.

3. Result:

   The model was trained and evaluated on each fold of the data. For every iteration:

   The model's accuracy on the test set was printed.

   The features were ranked by their importance.

   The average accuracy across all folds provided an aggregate measure of the model's performance.

   Additionally, the standard deviation of accuracy scores gave an indication of the model's consistency across different parts of the dataset.

   To provide further clarity on the significance of different features, an average rank of the top features across all folds was computed. This average rank highlighted the most consistently important features across all iterations.

4.5.12 Implications for User Interface

The insights from the feature importance analysis are invaluable for the development of the user interface. By understanding which features significantly impact price, the UI can be tailored to emphasize these key features. This ensures that users are prompted to provide the most relevant information when seeking price predictions. For instance, if the number of bathrooms consistently ranks as a top feature, it would be wise to prominently display this option in the predictive tool of the web application. On the other hand, less impactful features could be relegated to advanced settings or optional inputs. We chose to build a web application using HTML, CSS, and Python Flask. All three of these languages work together to create a functioning application.

HyperText Markup Language

HTML, or HypertText Markup Language, was used to build the structure and presentation of the web application, including text, images, links, forms, and multimedia. HTML uses tags and attributes to describe the elements and their relationships within a web page.

```html
<div class="topnav">
       <a class="active" href="{{ url_for('predictor') }}">Predictor</a>
       <a href="{{ url_for('about') }}">About</a>
</div>
```

Listing 4.7: Web pade tool bar code.

The tag ”form” allows users to input and submit data to a web server for processing. HTML forms serve as a means for the user to interact with the web application, provide information, and trigger actions. Listing 4.8 demonstrates what these forms tags look like in code.

```html
<form method="POST" action="{{ url_for('predictor') }}">
       {{ form.hidden_tag() }}
       <div class="row">
           <div class="column">
               {{ form.accommodates.label }}<br>{{
    form.accommodates(class="input-field") }}
               {{ form.bathrooms.label }}<br>{{ form.bathrooms(class="input-field") }}
               {{ form.nights_staying.label }}<br>{{
    form.nights_staying(class="input-field") }}
               {{ form.bedrooms.label }}<br>{{ form.bedrooms(class="input-field") }}
               {{ form.fireplace.label }}<br>{{ form.fireplace() }}
               {{ form.hot_tub.label }}<br>{{ form.hot_tub() }}
           </div>
           <div class="column">
               {{ form.cable.label }}<br>{{ form.cable() }}
               {{ form.wifi.label }}<br>{{ form.wifi() }}
               {{ form.review_scores_value.label }}<br>{{
    form.review_scores_value(class="input-field") }}
               {{ form.property_type.label }}<br>{{
    form.property_type(class="input-field") }}
               {{ form.room_type_num.label }}<br>{{
    form.room_type_num(class="input-field") }}
```

```
                {{ form.room_type_num.label }}<br>{{
    form.room_type_num(class="input-field") }}
            </div>
        </div>
        {{ form.submit(class="submit-button") }}
    </form>
```

Listing 4.8: HTML forms code.

Every page of the web page has its own unique HTML file. HTML is ofter combined with CSS to create interactive and visually appealing web pages.

Cascading Style Sheet

CSS, or Cascading Style Sheets, was used to control the visual presentation and layout of HTML elements on the web page. CSS allows the web application to adapt to various screen sizes and devices, customize the look and feel of the user interface, and create animations and interactions without relying on Javascript. As shown in Listing 4.9, CSS styles can be applied directly within a HTML document or linked externally to it, making it easier to maintain and update the appearance of a website.

```
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
    filename='style.css') }}">
```

Listing 4.9: Implementing the CSS style sheet to the HTML web page.

CSS allowed us to define how text, images, links, and elements should look on the website. CSS works by selecting HTML elements and applying style to them using selectors and properties. Selectors target specific elements, such as headings or paragraphs, and properties define the style, like the font size or background color.

```
body {
     height: 100%;
     background-color: #51504c;
     color: white;
     font-family:Cambria, Cochin, Georgia, Times, 'Times New Roman', serif;
}

input[type=text] {
     width: 50%;
     padding: 12px 20px;
     margin: 8px 0;
     box-sizing: border-box;
     border: 2px solid black;
     border-radius: 4px;
     background-color: rgba(128, 128, 128, 0.281);
}

button {
     background-color: #6e6e6e51;
     border: 2px solid black;
     color: rgb(250, 248, 248);
     padding: 15px 32px;
     text-align: center;
     text-decoration: none;
     display: inline-block;
     font-size: 12px;
}
```

Listing 4.10: CSS code.

Python Flask

Python Flask was used as the web framework for our web application. Flask is a lightweight and versatile web framework for building web applications in Python. Flask is often referred to as a micro-framework because it provides the essential components for building web applications without imposing a rigid structure or including a lot of built-in functionality. Flask allows you to define URL routes, which determine how different URLs or endpoints of the web application should be handled.

```
@app.route('/price')
def price():
   estimated_price = request.args.get('estimated_price', 'N/A')
   mae = request.args.get('mae', 'N/A')
```

```python
    return render_template('price.html', estimated_price=estimated_price, mae=mae)
```

Listing 4.11: Route example code.

Flask supports HTTP methods like GET, POST, PUT, DELETE, etc., which enables you to handle different types of requests from web clients.

```python
@app.route('/predictor', methods=['GET', 'POST'])
```

Listing 4.12: HTTP methods code

Flask includes a templating engine, called Jinja2, that allows you to create dynamic HTML templates. These templates can be populated with data from your Python code, enabling you to generate dynamic web content.

CHAPTER 5

RESULTS

The foundation of any robust data science project lies in the quality of data, the depth of its analysis, and the insights gleaned from it. This chapter dives deep into the results obtained from our rigorous exploration of a Nashville Airbnb dataset. By utilizing traditional statistical techniques and cutting-edge machine learning models, we are prepared to share inherent patterns, namely relationships between input features and predictive outcomes, that can provide actionable insights for stakeholders. Here, we will walk through the steps of our analytical journey, from identifying the model to creating a custom model for the problem, primarily using regression models to predict fair market pricing. The model-driven outcomes are further interpreted using state-of-the-art causal inferencing tools to rule out spurious correlation in data and provide more transparent and trustworthy predictors of model outcomes.

5.1 Regression analysis and results

To begin a machine learning project, one must first explore the data to decide what model to choose and build for the problem in hand. With a large variety of models available, the general practice of grooming the data is standard and allows us to test six different regression models for our project quickly. These regression models provide a function that describes the relationship between one or more independent variables and a dependent target variable. We tested various advanced regression models, including linear regressor, decision tree, Gradient Boost- ing tree, and HistGradientBoosting regressor. We chose the best regression model based on the testing results of the highest R2 score of 72.22. After testing, we decided that the HistGradientBoosting regressor was the best regression model for our project.

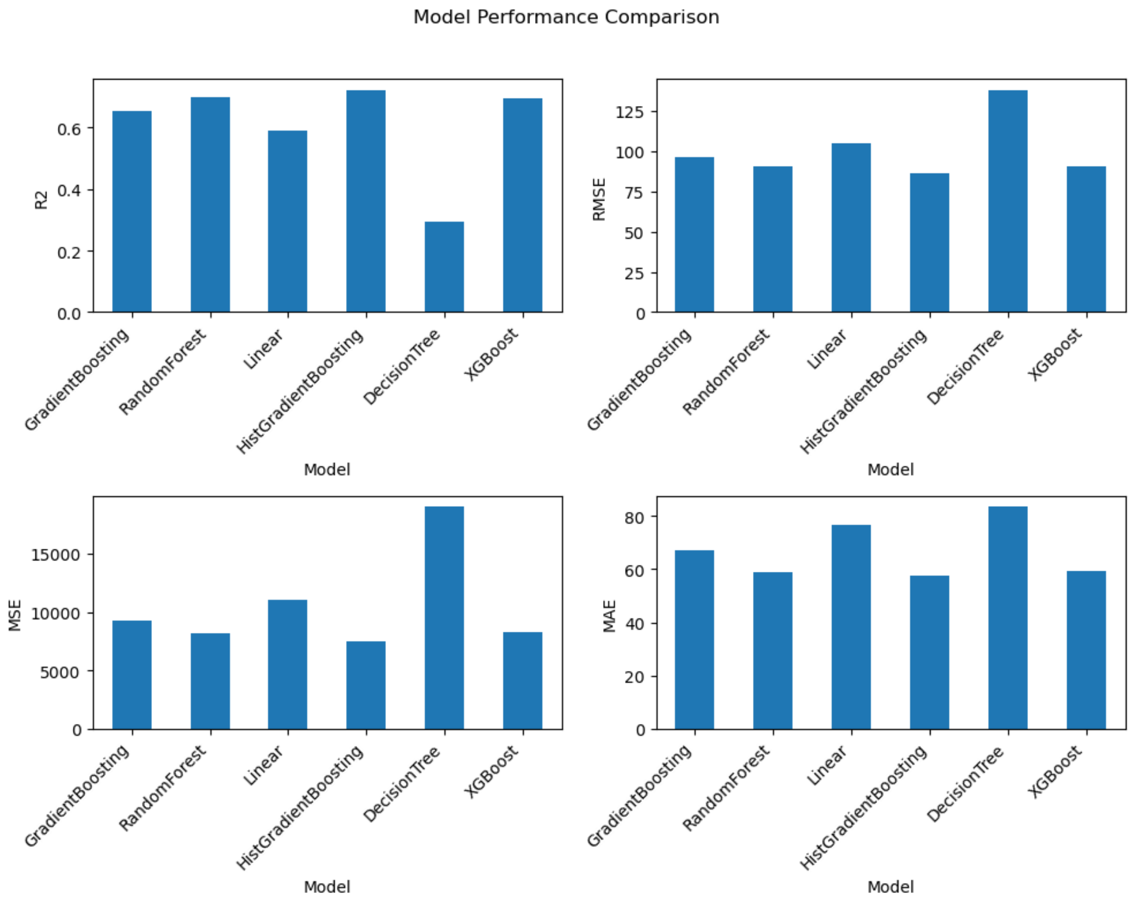


Figure 5.1: The results of the Regression Model testing.

5.2 Feature Selection Results

The data dimensionality or the number of features is critical for model performance. While more features can provide more information about the relationship, it can lead to 'the curse of dimensionality' effect, hurting the model performance. Concurrently, we also wanted to assess the computational efficiency of the model by observing the time taken to process various numbers of features. We knew that every additional feature would take additional computation time; with that in mind, we want to see when the results start to level out. Therefore, identifying the most important combination of features, namely feature selection, is an imperative step in machine learning modeling.

While our goal was to narrow down to approximately 20 distinct features, certain features, like 'property type', required one-hot encoding. This encoding process meant that to differentiate

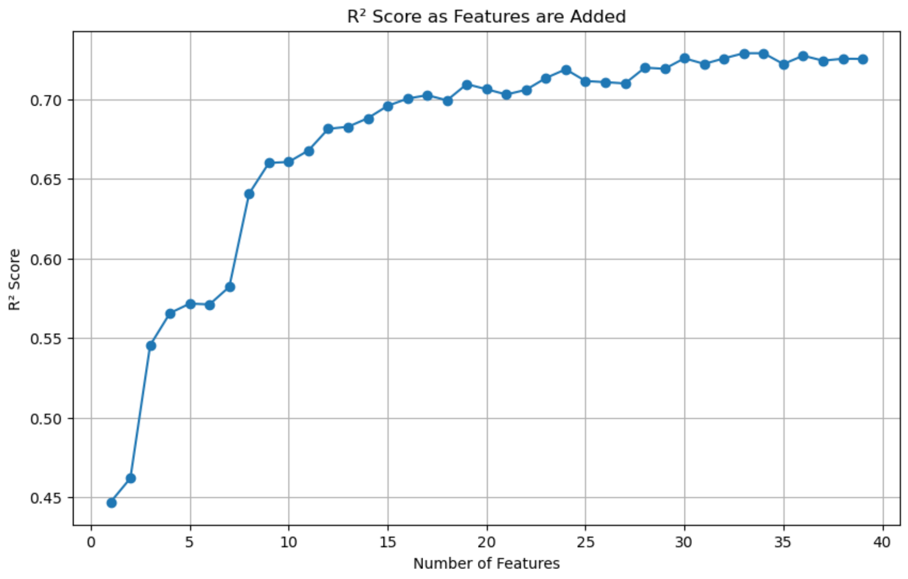


Figure 5.2: We can see the importance of the first 20 features and then see a more even increase in its feature importance but with greater time.

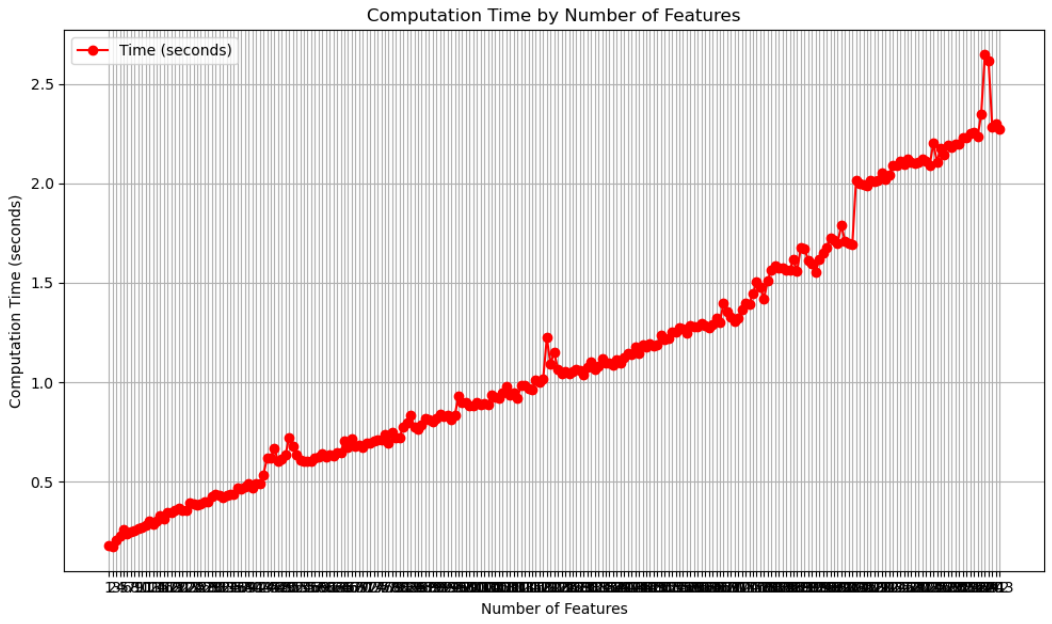


Figure 5.3: Knowing that each feature adds a linear time increase but adds no value to the r2 score allows us to reduce the features, resulting in faster computational time for the user using our software.

among categories such as a house, an apartment, or a motorhome, we had to introduce multiple feature columns. As a result, the actual number of feature columns utilized was 28, but they represent 20 unique features. When we re-ran the model, linear regression showed unexpectedly negative results. However, with the HistGradientBoosting model, we still achieved our target R-squared score of 70

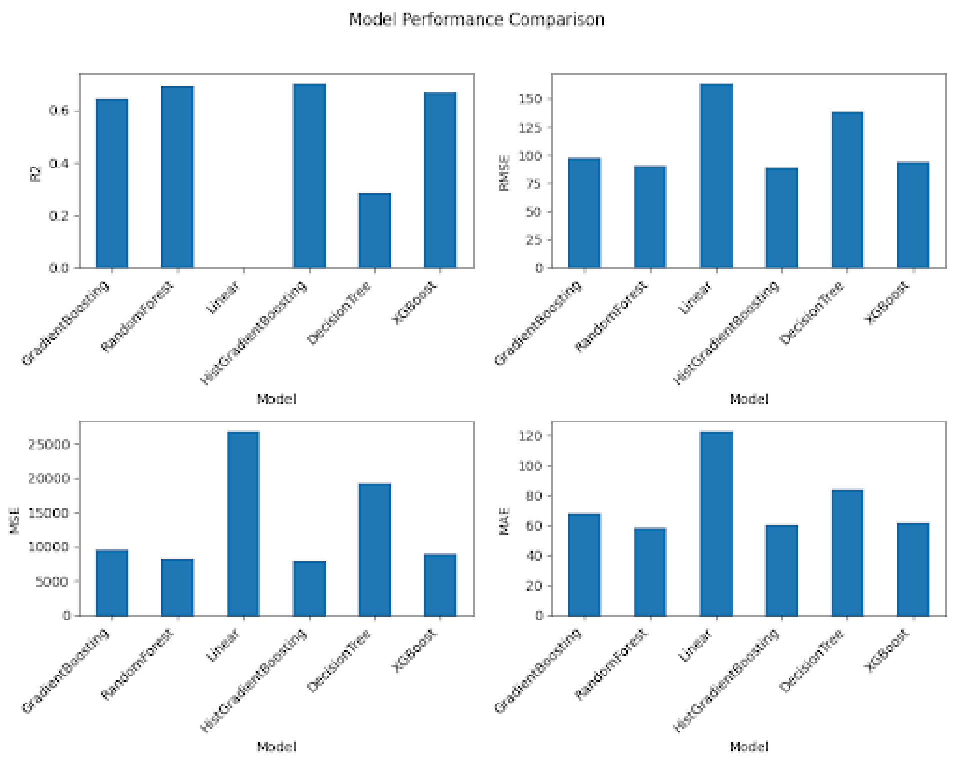


Figure 5.4: Updated results of the Regression Model testing.

Table 5.1: Final r2, MAE, MSE, and RMSE score.

Model	MAE	MSE	R2	RMSE
DecisionTree	83.962371	19214.576281	0.285639	138.616652
GradientBoosting	68.014278	9572.757100	0.644103	97.840468
HistGradientBoosting	60.517344	7977.994567	0.703393	89.319620
Linear	123.133481	26900.691076	-0.000116	164.014301
RandomForest	58.805755	8294.641718	0.691621	91.074924
XGBoost	61.770700	8923.464897	0.668243	94.464093

5.3 Causality

In the progression of our study, a central ambition was to integrate causal inference, allowing us to delve deeper into the underlying mechanisms of our data. By posing a causal question, we sought not just to establish correlations but to determine the causes and effects that drive the observed relationships. After formulating our causal query, we proceeded to generate an estimate that quantifies the magnitude and direction of this causal relationship. However, an estimate alone is not sufficiently robust. To ensure the veracity of our findings, we introduced a refutation test, producing a refutation score. A statistically significant refutation score acts as a litmus test for our model's accuracy. It indicates that our model's predictions are not merely the outcome of chance or spurious correlations but are likely grounded in genuine causal relationships. As we delved deeper into the importance of each feature, it paved the way for a clearer understanding of the potential causal questions that could be posed, thereby enhancing the depth and rigor of our analysis. With this foundational understanding, we then sought to pinpoint specific causal relationships that might have a tangible impact on property pricing values.

5.3.1 Causal Question

Every property has its own story to tell and price that should reflect it, but there are simple things a homeowner could do, like swapping a couch to a sofa bed to increase the number of beds and accommodate in a property. Our causal question was, ”If an owner increases the total accommodates, how much more can they rent out their property on average per day in the Nashville market?”

5.3.2 Causal Discovery

Domain knowledge is pivotal in constructing an accurate Direct Acyclic Graph(DAG), a foundational component of Judea Pearl's causal model framework. However, since the commencement of our project, several algorithms and models, including Linear Non-Gaussian Acyclic Models (LiNGAM), Greedy Equivalence Search (GES), and the Peter-Clark Algorithm, have emerged.

These methodologies leverage data to predict causal relationships. While these models are progressively becoming more precise, there remains a discernible room for improvement.

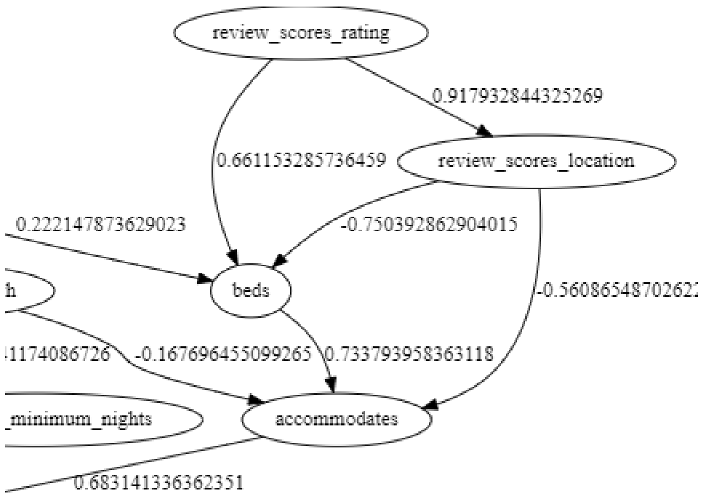


Figure 5.5: Subset of LiNGAM Model Visualization: The model identifies a pathway from 'beds' to 'accommodates,' validating one causal inquiry, but fails to connect 'price' to any feature resulting in an incomplete graph.

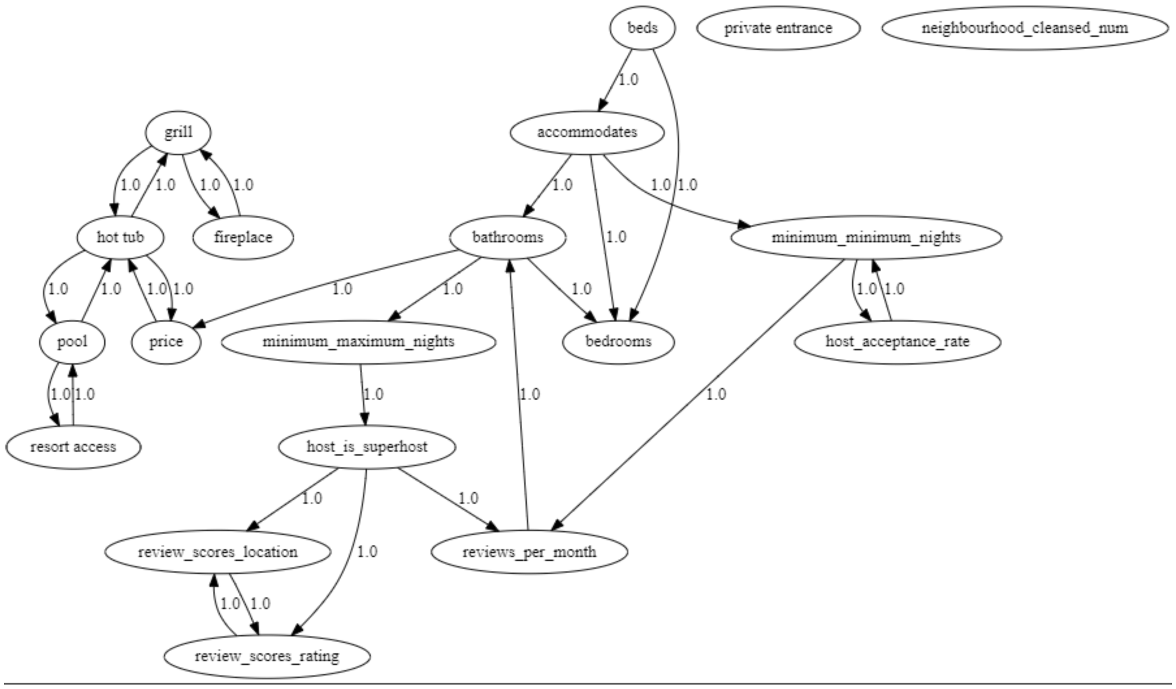


Figure 5.6: The PC method visualization adeptly clusters features such as 'resort access,' 'pool,' 'hot tub,' and 'grill,' along with 'bathrooms,' 'bedrooms,' 'beds,' and 'accommodation.' However it does not connect all features

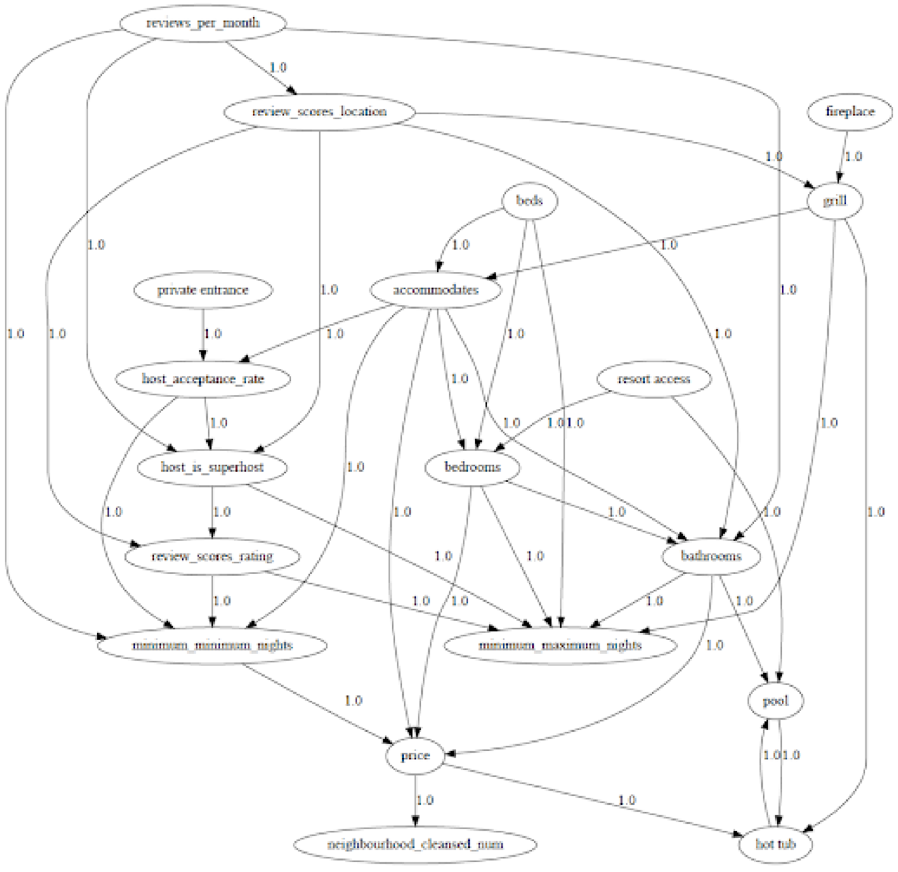


Figure 5.7: GES Graph: From our domain perspective, this appears to be the most accurate representation yet, though it's important to note that repeated executions yield varied graphs.

Summary of Causal Discovery Algorithms

LiNGAM Model

The LiNGAM algorithm successfully identifies expected causal directions, such as from 'beds' to 'accommodates', demonstrating its capability in uncovering linear causal relationships. Its failure to establish a connection from 'price' to other variables signals a need for model refinement, possibly requiring additional data or parameter adjustment. The numerical weights on the LiNGAM graph edges are indicative of the magnitude and direction of effects, offering valuable insights into how changes in one variable affect another.

PC Algorithm

The PC algorithm reveals potential causal structures in a binary manner, highlighting the presence of causal links without estimating their strengths. This necessitates further analysis, underpinned by domain knowledge, to unravel the underlying causal mechanisms and quantify the strengths of the relationships suggested by the algorithm.

GES Algorithm

The GES algorithm generates graphs that align with the best structural fit to the data, according to the chosen scoring function. Despite providing a structural overview, the variability across different executions and the absence of quantitative edge weights underscore the importance of domain expertise. Such knowledge is essential for validating the causal relationships and for enriching the algorithmic findings with contextual depth and insight.

In summary, the LiNGAM algorithm is adept at quantifying causal effects, while the PC and GES algorithms excel at outlining the structure of causal networks. Domain expertise plays a pivotal role across all methods for interpreting, validating, and refining the models. An integrated approach, combining multiple algorithmic outputs and informed by expert knowledge, is advocated to develop a robust and nuanced causal understanding.

Large Language Models Meets Causal DAG

The intersection of Large Language Models (LLMs) and Causal Discovery is an emerging area of interest among data scientists. A compelling illustration of this synergy can be found in an article penned by Aleksander Molak on Medium.com. Molak, also the author of the book ”Causal Inference and Discovery in Python,” showcased how integrating ChatGPT with machine learning causal discovery, specifically the PC method can foster a more robust model, courtesy of the LLM’s capabilities.[6] We emulated this approach, employing a subset of our features for experimentation. Here, we present our findings contrasting models with and without the influence of an LLM.

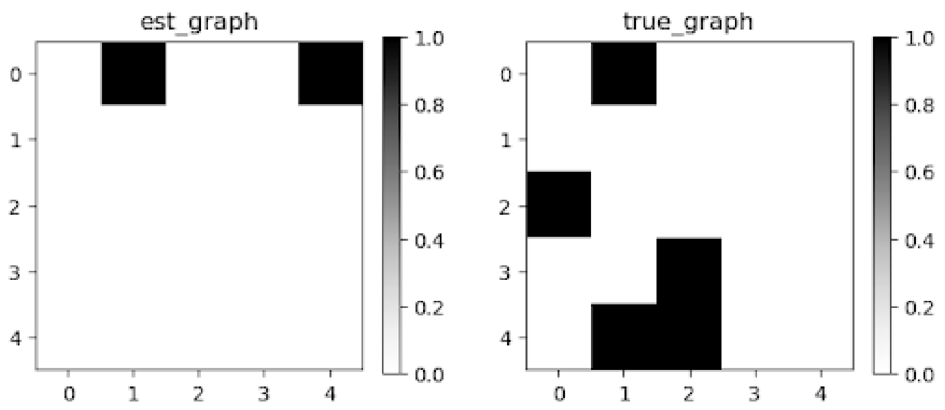


Figure 5.8: Metrics without LLM vs. True DAG: On the left, we have the DAG derived without the influence of an LLM, and on the right, the true DAG for reference.

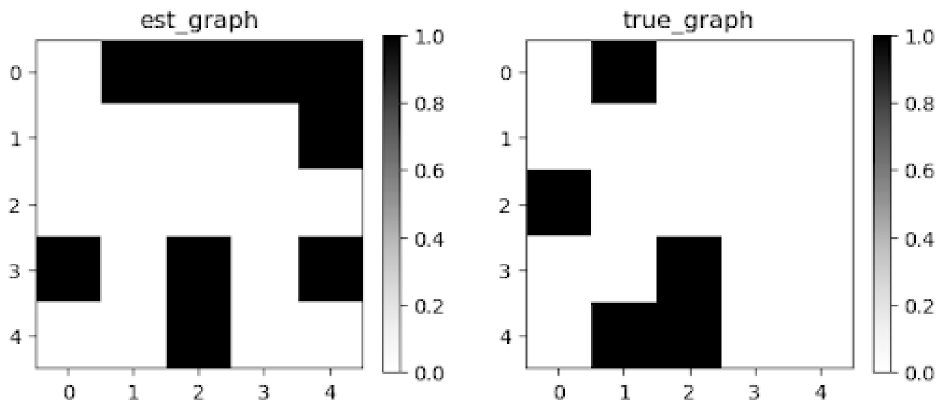


Figure 5.9: Metrics without LLM vs. True DAG: On the left, we have the DAG derived without the influence of an LLM, and on the right, the true DAG for reference.

Domain Knowledge DAG

Relying on our domain expertise, we crafted a DAG tailored to our dataset. Our next steps involve generating an estimate and a refutation score for our target variable, 'accommodates,' with 'price' as the outcome variable. A salient observation during our experimentation with neural network models was the mutable nature of our graphs upon each iteration. This reinforces our belief that while modern methods offer valuable insights into data relationships, the infusion of domain knowledge remains indispensable.

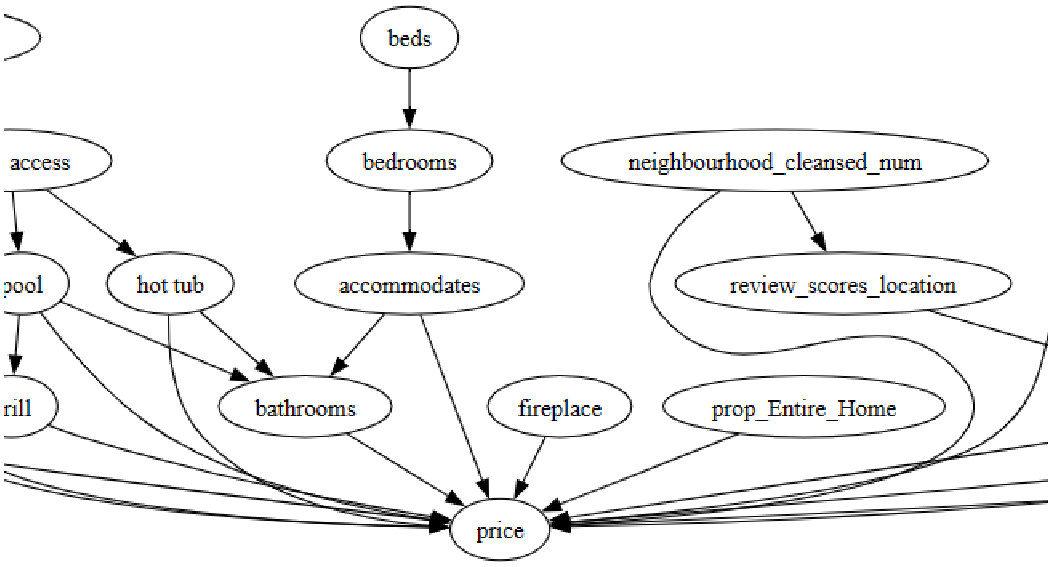


Figure 5.10: DAG Based on Domain Knowledge: Subset of our graph showing our interpretation and representation of the data, rooted in our expertise.

5.3.3 Causal Inference Estimator

After establishing the causal graph and understanding the potential relationships among the features, we moved to the next crucial step in our causality analysis: causal inference. This step enabled us to derive insights regarding the effect of our treatment variable, ’accommodates,’ on the outcome variable, ’price.’ We utilized a backdoor linear regression method, targeting the average treatment effect (ATE). Here is the Python code we employed: The causal model’s results showed a mean value of approximately 21.92 for the effect of ’accommodates’ on ’price.’ In essence, this suggests that for every unit increase in ’accommodates,’ the price could potentially increase by approximately 21.92 dollars after accounting for other variables. This effect was consistent across various conditions and feature combinations, as evidenced by the numerous conditional estimates provided. However, an essential aspect of causal inference is not just estimating the causal effect but validating its authenticity. Therefore, we introduced refutation methods to challenge our initial findings and assess their robustness.

5.3.4 Causal Refutation Test

In the realm of causal inference, the reliability of an estimated causal effect is paramount and the least understood step in causal inference. Refutation tests serve as a mechanism to gauge the robustness of these estimates by challenging them under controlled perturbations. For our analysis, we employed two such methods: the ’data subset refuter’ and the ’random common cause.’

Data Subset Refuter

The Data Subset Refuter method operates by randomly excising a subset of the data and reevaluating the causal effect. If the recalculated causal effect remains congruent with the original, even after the perturbation, it lends further credence to the robustness of our initial estimate. Upon application, the refutation test yielded a new effect of approximately 0.4924 post-data subset removal. This was remarkably consistent with our original estimate of 0.4929 derived from the standard scaled data. With a p-value of 0.96, the minimal difference between the two estimates was statistically negligible. After reverse-engineering the standardization, the real effect translated to an increase of approximately 21.94 for every additional accommodation unit, a slight rise from the original $21.92.

Random Common Cause

The Random Common Cause refutation method introduces a random common cause to the dataset and reevaluates the causal effect. The rationale is that if introducing a spurious variable doesn’t significantly alter the estimate, then the original effect is likely robust. In our scenario, the effect derived after accommodating the random common cause was The original beta value, represented as ”beta original,” is equal to 0.4928671844013451 with its .98 score, we know the results are valid.

5.3.5 Refutation Test Conclusion

The congruence between our original estimates and those obtained post-refutation affirms the reliability of our causal effect. The minute deviations observed, especially when contextualized with the real-world implications, underscore the robustness of our findings. Such consistency across different refutation techniques bolsters our confidence in the derived insights, emphasizing their potential applicability in real-world scenarios.

```
Estimated effect:0.4928598248111045
New effect:0.4928671844013451
p value:0.98
```

Listing 5.1: Results from Random Common Cause refutation test.

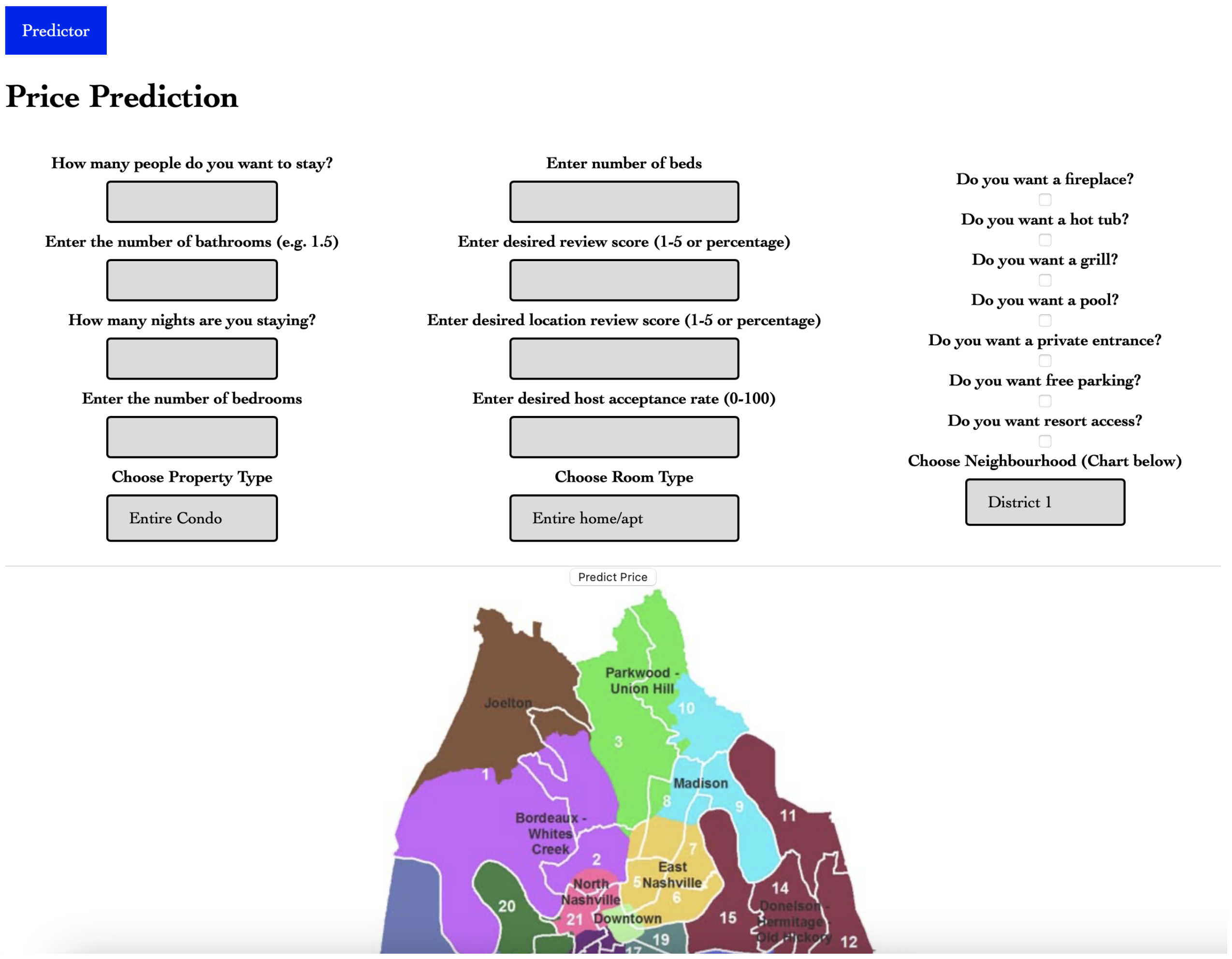
5.3.6 Causality Results

Our causal analysis journey, from formulating the right questions to employing modern techniques for causal discovery and validation, underscores the intricate balance between data-driven methods and domain expertise. While algorithms and models provide a structured pathway to unearth causal relationships, the significance of domain knowledge remains unparalleled, acting as a compass in navigating the complex terrains of causality.

Our findings shed light on the potential strategies homeowners in the Nashville market might employ to enhance their property's value. However, as with all models, it's essential to approach the results with a degree of caution and an understanding of the model's limitations. The ever-evolving domain of causal inference promises even more refined tools in the future, paving the way for more nuanced insights and strategies.

5.4 Web Application

The web application was designed to provide a user-friendly interface for the purpose of predicting feasible Airbnb rental pricing. The web app was built using the Python Flask framework, with the front-end built using HTML and CSS. The user will enter features of the property they would like to get a predicted price for using the opening page, shown in Figure 5.11.


Predictor

Price Prediction

How many people do you want to stay?

Enter the number of bathrooms (e.g. 1.5)

How many nights are you staying?

Enter the number of bedrooms

Choose Property Type: Entire Condo

Enter number of beds

Enter desired review score (1-5 or percentage)

Enter desired location review score (1-5 or percentage)

Enter desired host acceptance rate (0-100)

Choose Room Type: Entire home/apt

Do you want a fireplace?

Do you want a hot tub?

Do you want a grill?

Do you want a pool?

Do you want a private entrance?

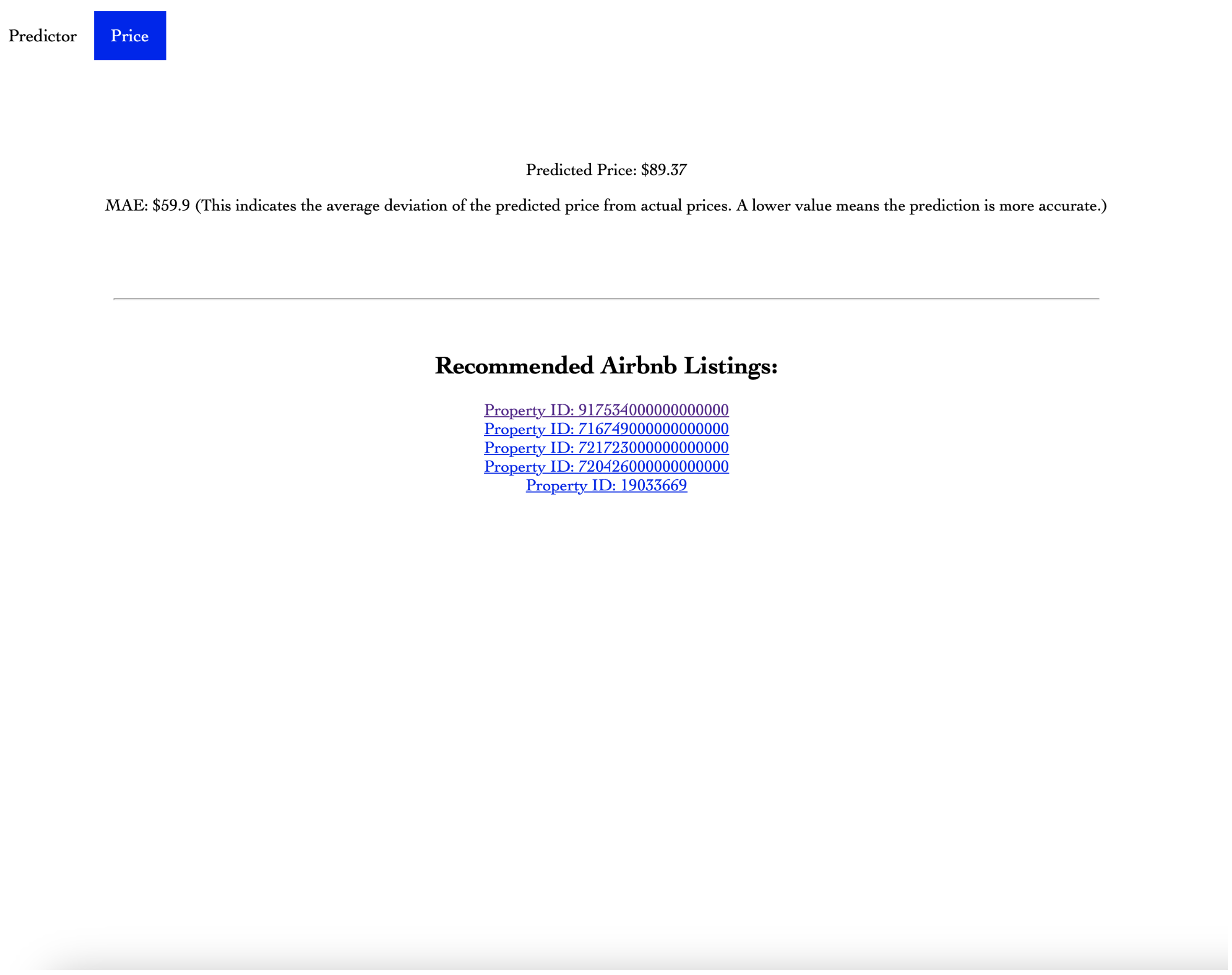
Do you want free parking?

Do you want resort access?

Choose Neighbourhood (Chart below): District 1

Figure 5.11: The screen where the user will enter property features to predict price.

After the user submits the features, their features are put through our prediction model. The model then returns three results, shown in Figure 5.12. It tells the user the predicted price, the mean absolute error, and five similar Airbnb properties. The mean absolute error is the amount the predicted price can vary.



Predicted Price: $89.37

MAE: $59.9 (This indicates the average deviation of the predicted price from actual prices. A lower value means the prediction is more accurate.)

Recommended Airbnb Listings:

Property ID: 917534000000000000
Property ID: 716749000000000000
Property ID: 721723000000000000
Property ID: 720426000000000000
Property ID: 19033669

Figure 5.12: This screen is presented when the user submits their property features.

CHAPTER 6

CONCLUSIONS

In this comprehensive analysis of the Nashville Airbnb market, we employed a series of methodological approaches, each underpinned by robust scientific principles. Beginning with the raw dataset, meticulous preprocessing techniques ensured the data's integrity and relevance, laying the groundwork for sophisticated modeling.

Our modeling phase was characterized by rigorous empirical testing across multiple regression models. The HistGradientBoosting Regressor emerged as the optimal model, demonstrating superior performance metrics in capturing the complexities of the dataset. From here we were able to do feature reduction to allow our Python flask app to run faster with minimal, and acceptable,l statistical loss.

Transitioning to causality, we ventured into the intricate domain of causal inference, systematically exploring the cause-effect relationships inherent in our data. Leveraging both state-of-the-art algorithms and established domain knowledge, our analysis identified 'accommodates' as a statistically significant determinant of property prices.

The confluence of advanced analytical techniques, rigorous empirical methodologies, and domain expertise culminated in a set of actionable insights. These insights, grounded in scientific rigor, provide a robust framework for stakeholders to make informed decisions in the Nashville Airbnb market. This study underscores the importance and potential of data-driven, scientifically rigorous approaches in understanding and optimizing complex market dynamics.

Appendices

APPENDIX 1

CREATING THE MODEL

A.1 Model Creation

```
# Import necessary libraries
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import xgboost as xgb
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor, HistGradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
import dowhy
from dowhy import CausalModel
import re
from collections import Counter



def load_data(filepath):
    df = pd.read_csv(filepath, header=None, skiprows=0)
    df.columns = df.iloc[0]
    df = df[1:]
    return df



# Load and preprocess data
file_path = r"C:\Users\nstep\TSU\SeniorProject\DataSet\listings.csv"
nashvilleDF = load_data(file_path)

#nashvilleDf to lower case

nashvilleDF.head()

pd.options.display.max_columns = 200
# Colors sourced from here: https://usbrandcolors.com/airbnb-colors/
bnb_red = '#FF5A5F'
bnb_blue = '#00A699'
bnb_orange = '#FC642D'
bnb_lgrey = '#767676'
bnb_dgrey = '#484848'
```

```python
bnb_maroon = '#92174D'
# Create diverging colormap for heatmaps
bnb_cmap = sns.diverging_palette(210,
                                 13,
                                 s=81,
                                 l=61,
                                 sep=3,
                                 n=16,
                                 as_cmap=True)

# Test colors
sns.palplot(sns.diverging_palette(210, 13, s=81, l=61, sep=3, n=16))



# Create color palette
bnb_palette = sns.color_palette(
    ["#FF5A5F", "#007989", "#8CE071", "#FC642D", "#92174D", "#01D1C1"])

# Test colors
sns.palplot(bnb_palette)


nashvilleDF.shape
#nashvilleDF.info()

#write nashvilleDF.info to csv
info = nashvilleDF.info()



# Visualize price table, changing them to floats and replacing the commas with a blank
prices = nashvilleDF['price'].apply(lambda s: float(s[1:].replace(',','')))

# Drop listings with a price of zero
prices = prices[prices!=0]

# Log prices
log_prices = np.log(prices)

print(log_prices.describe())

def plot_hist(n, titles, ranges):
    """
    Quick helper function to plot histograms
    """
    fig, ax = plt.subplots(n, figsize = (8, 7.5))
    for i in range(n):
        d, bins, patches = ax[i].hist(ranges[i], 50, density = 1, color='red', alpha = 0.85)
        ax[i].set_title(titles[i])
        ax[i].set_xlabel("Daily Listing Price in Dollars")
        ax[i].set_ylabel("Frequency")
    plt.tight_layout()
    plt.show()

# Plot histograms of price distribution
```

```
plot_hist(4, ['Distribution of Listing Prices: All Data', 'Distribution of Listing Prices: $0 - $1000',
               'Distribution of Listing Prices: $0 - $350','Log Transformed Distribution of Listing Prices: All Data'],
          [prices, prices[prices <= 1000], prices[prices < 350],log_prices])



# Convert relevant columns to datetime format
nashvilleDF.last_scraped = pd.to_datetime(
    nashvilleDF.last_scraped)

nashvilleDF.host_since = pd.to_datetime(
    nashvilleDF.host_since)

nashvilleDF.calendar_last_scraped = pd.to_datetime(
    nashvilleDF.calendar_last_scraped)

nashvilleDF.first_review = pd.to_datetime(
    nashvilleDF.first_review)

nashvilleDF.last_review = pd.to_datetime(
    nashvilleDF.last_review)



nashvilleDF.host_response_rate = nashvilleDF[
    'host_response_rate'].apply(lambda s: float(str(s).replace('%', '')))



nashvilleDF.property_type.value_counts()

sns.countplot(y=nashvilleDF['property_type'])
plt.ylabel('Property Types')
plt.xlabel('Number of Listings')
plt.title('Number of Listings by Propery Type')

# Step 1: Create a mask to identify rows with 'Private' in the 'property_type' column
private_mask = nashvilleDF['property_type'].str.contains('Private')

# Step 2: Update the values in the 'property_type' column based on the mask
nashvilleDF.loc[private_mask, 'property_type'] = 'Private room'

nashvilleDF.replace('Aparthotel','Hotel',inplace=True)
nashvilleDF.replace('Room in aparthotel','Hotel',inplace=True)
nashvilleDF.replace('Room in hotel','Hotel',inplace=True)
nashvilleDF.replace('Room in boutique hotel','Hotel',inplace=True)
nashvilleDF.replace('Entire serviced apartment','Entire condo',inplace=True)
nashvilleDF.replace('Entire loft','Entire condo',inplace=True)

private_mask = nashvilleDF['property_type'].str.contains('Shared')

nashvilleDF.loc[private_mask, 'property_type'] = 'Shared room'

# Display the updated value counts
updated_value_counts = nashvilleDF['property_type'].value_counts()
updated_value_counts
```

```

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['property_type'].value_counts()
print(updated_value_counts_after_drop)


# List of property types to drop
drop_list = [
    "Entire bungalow", "Entire cottage", "Tiny home", "Camper/RV", "Entire cabin",
    "Entire vacation home", "Entire place", "Shared room", "Entire villa",
    "Farm stay", "Entire home/apt", "Earthen home", "Barn", "Entire chalet",
    "Bus", "Shipping container", "Boat", "Tent"
]

# Drop rows based on the property_type values in drop_list
nashvilleDF = nashvilleDF[~nashvilleDF['property_type'].isin(drop_list)]

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['property_type'].value_counts()
print(updated_value_counts_after_drop)


updated_value_counts_after_drop = nashvilleDF['host_response_time'].value_counts()
print(updated_value_counts_after_drop)

# Host Response Time
# We will treat nan values as

nashvilleDF[
    'host_response_time'] = nashvilleDF.host_response_time.map({
        'within an hour':
        1,
        'within a few hours':
        2,
        'within a day':
        3,
        'a few days or more':
        4,
        np.nan:
        5
    })




updated_value_counts_after_drop = nashvilleDF['host_response_time'].value_counts()
print(updated_value_counts_after_drop)



# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_is_superhost'].value_counts()
print(updated_value_counts_after_drop)


nashvilleDF['host_is_superhost'] = [
    1 if x == 't' else 0 for x in nashvilleDF.host_is_superhost
]
# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_is_superhost'].value_counts()
print(updated_value_counts_after_drop)


# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['instant_bookable'].value_counts()
print(updated_value_counts_after_drop)

nashvilleDF['instant_bookable'] = [
    1 if x == 't' else 0 for x in nashvilleDF.instant_bookable
]

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['instant_bookable'].value_counts()
print(updated_value_counts_after_drop)


# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['has_availability'].value_counts()
print(updated_value_counts_after_drop)

nashvilleDF['has_availability'] = [
    1 if x == 't' else 0 for x in nashvilleDF.has_availability
]

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['has_availability'].value_counts()
print(updated_value_counts_after_drop)

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_has_profile_pic'].value_counts()
print(updated_value_counts_after_drop)

nashvilleDF['host_has_profile_pic'] = [
    1 if x == 't' else 0 for x in nashvilleDF.host_has_profile_pic
]

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_has_profile_pic'].value_counts()
print(updated_value_counts_after_drop)


# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_identity_verified'].value_counts()
print(updated_value_counts_after_drop)


nashvilleDF['host_identity_verified'] = [
    1 if x == 't' else 0 for x in nashvilleDF.host_identity_verified
]

# Display the updated value counts
updated_value_counts_after_drop = nashvilleDF['host_identity_verified'].value_counts()
```

```
print(updated_value_counts_after_drop)


# Property Type
# One hot encode
prop_type_dummies = pd.get_dummies(nashvilleDF.property_type,
                                   prefix='prop')

# Merge with df
nashvilleDF = nashvilleDF.merge(prop_type_dummies,
                                left_index=True,
                                right_index=True)


nashvilleDF.property_type.value_counts()


nashvilleDF.room_type.value_counts()


# Room Type
# Create numerical column for room type
nashvilleDF['room_type_num'] = nashvilleDF.room_type.map({
    'Entire home/apt':
    3,
    'Private room':
    2,
    'Hotel room':
    1,
})

# One hot encode
room_type_dummies = pd.get_dummies(nashvilleDF.room_type,
                                   prefix='room')

# Drop "hotel room" column as base case
del room_type_dummies['room_Hotel room']

# Merge with df
nashvilleDF = nashvilleDF.merge(room_type_dummies,
                                                    left_index=True,
                                                    right_index=True)


nashvilleDF.room_type_num.value_counts()


# Combine 'Private room' and 'Hotel room' into 'Private/Hotel room'
nashvilleDF['room_type'] = nashvilleDF['room_type'].replace(['Private room', 'Hotel room'], 'Private/Hotel room')

# Display the updated value counts for room_type column
updated_value_counts_room_type = nashvilleDF['room_type'].value_counts()
print(updated_value_counts_room_type)

# Map the updated room types to numeric values
nashvilleDF['room_type_num'] = nashvilleDF.room_type.map({
    'Entire home/apt': 2,
    'Private/Hotel room': 1
})

# Display the updated value counts for room_type_num column
updated_value_counts_room_type_num = nashvilleDF['room_type_num'].value_counts()
updated_value_counts_room_type_num


nashvilleDF.head()

nashvilleDF[nashvilleDF['bedrooms'].isnull()]


# Extract numeric portion from the 'bathrooms_text' column and assign to 'bathrooms' column
nashvilleDF['bathrooms'] = nashvilleDF['bathrooms_text'].str.extract('(\d+\.?\d*)').astype(float)

# Drop the 'bathrooms_text' column
nashvilleDF = nashvilleDF.drop(columns=['bathrooms_text'])


#show nashvilleDF['bedrooms'] that are empty
nashvilleDF.bathrooms.value_counts()


nashvilleDF.neighbourhood.value_counts()


nashvilleDF.neighbourhood.fillna('Empty',inplace=True)

nashvilleDF.neighbourhood.value_counts()


#drop neighbourhood from nashvilleDF
nashvilleDF.drop('neighbourhood',axis=1,inplace=True)
nashvilleDF.neighbourhood_cleansed.value_counts()

# Extract the district number from the 'neighbourhood_cleansed' column and assign to 'neighbourhood_cleansed_num'
nashvilleDF['neighbourhood_cleansed_num'] = nashvilleDF['neighbourhood_cleansed'].str.extract('(\d+)').astype(int)

# Display the updated value counts for the 'neighbourhood_cleansed_num' column
updated_neighbourhood_cleansed_num_value_counts = nashvilleDF['neighbourhood_cleansed_num'].value_counts()
updated_neighbourhood_cleansed_num_value_counts


import re
from collections import Counter

# Format amenities column for analysis
nashvilleDF.amenities = nashvilleDF.amenities.apply(
    lambda x: [i.strip() for i in re.sub('[^a-zA-Z,\/\s\d-]*', '', x.lower()).split(sep=',')] if isinstance(x, str) else
     x)

# Create a flat list of all amenities entries
amenities_list = [item for sublist in nashvilleDF.amenities for item in sublist if isinstance(sublist, list)]

```

```
# Count amenities occurrences
amenity_counts = Counter(amenities_list).most_common()

# Examine the top 20 amenities
top_amenity_counts = amenity_counts[0:20]

# Look at the 90 least common amenities
lowest_amenity_counts = amenity_counts[-90:]

# Total unique amenities
total_unique_amenities = len(set(amenities_list))

lowest_amenity_counts, total_unique_amenities



# Make a list of amenities of interest
amenities_of_interest = [x[0] for x in amenity_counts[0:70]]

#print amenity_counts 1 per row
for amenity in amenity_counts:
    print(amenity)

print(amenity_counts)


combine_keywords = {
    "coffee": "coffee",
    "shampoo": "shampoo",
    "toaster": "toaster",
    "crib": "crib",
    "hot tub": "hot tub",
    "refrigerator": "refrigerator",
    "gym": "gym",
    "resort access": "resort access",
    "microwave": "microwave",
    "kitchen": "kitchen",
    "camera": "camera",
    "cable": "cable",
    "grill": "grill",
    "stove": "stove",
    "backyard": "backyard",
    "bluetooth": "bluetooth",
    "wifi": "wifi",
    "oven": "oven",
    "sono": "sono",
    "disney": "disney",
    "high chair": "high chair",
    "tv": "tv",
    "hdtv": "tv", # Note: TV combines with HDTV
    "hbo": "hbo",
    "netflix": "netflix",
    "pool": "pool",
    "conditioner": "conditioner",
    "soap": "soap",
    "iron": "iron",
    "sony": "sony",
    "sound system": "sound system",
    "heating": " radiant heating",
    "stainless": "stainless",
    "washer": "washer",
    "dryer": "dryer",
    "free parking": "free parking",
    "baby monitor": "baby monitor",
    "baby bath": "baby bath",
    "body wash": "body wash",
    "changing table": "changing table",
    "books and toys": "books and toys",
    "clothing storage": "clothing storage",
    "exercise equipment": "exercise equipment",
    "carport": "carport",
    "free residential garage": "free residential garage",
    "game console": "game console",
    "fireplace": "fireplace",
    "paid parking garage": "paid parking garage",
    "paid parking lot off": "paid parking lot off",
    "paid parking lot on": "paid parking lot on",
    "paid parking on premises": "paid parking on premises",
    "paid valet parking": "paid valet parking",
    "hot water": "hot water",
    "free driveway": "free driveway",

}

# List to store new aggregated amenity counts
aggregated_amenity_counts = {}

# Loop through amenities and aggregate based on keywords
for amenity, count in amenity_counts:
    found = False
    for keyword, new_amenity in combine_keywords.items():
        if keyword in amenity.lower():
            # Special handling for "oven" to ensure it doesn't get overshadowed by "toaster"
            if keyword == "oven" and "toaster" in amenity.lower():
                continue
            # Special handling for "hot water" to ensure "hot water kettle" doesn't get combined
            if keyword == "hot water" and "kettle" in amenity.lower():
                continue
            if new_amenity not in aggregated_amenity_counts:
                aggregated_amenity_counts[new_amenity] = 0
            aggregated_amenity_counts[new_amenity] += count
            found = True
            break
    if not found:
        if amenity not in aggregated_amenity_counts:
            aggregated_amenity_counts[amenity] = 0
        aggregated_amenity_counts[amenity] += count

# Convert dictionary to list of tuples and sort
sorted_amenity_counts = sorted(aggregated_amenity_counts.items(), key=lambda x: x[1], reverse=True)

# Save sorted_amenity_counts to csv
```

```
df = pd.DataFrame(sorted_amenity_counts, columns=["Amenity", "Count"])
df.to_csv('new_amenity_counts.csv', index=False)

# Assuming nashvilleDF has a column 'amenities' that is either a string or list

# Ensure that the 'amenities' column is a list
nashvilleDF['amenities'] = nashvilleDF['amenities'].apply(lambda x: x if isinstance(x, list) else x.split(','))

# Define a function to aggregate amenities based on combine_keywords
def aggregate_amenities(amenities_list):
    aggregated_list = []
    for amenity in amenities_list:
        found = False
        for keyword, new_amenity in combine_keywords.items():
            if keyword in amenity.lower():
                aggregated_list.append(new_amenity)
                found = True
                break
        if not found:
            aggregated_list.append(amenity)
    return aggregated_list

# Apply the aggregate_amenities function to the amenities column
nashvilleDF['aggregated_amenities'] = nashvilleDF['amenities'].apply(aggregate_amenities)

# Create dummy variables
amenities_dummies = pd.get_dummies(nashvilleDF['aggregated_amenities'].apply(pd.Series).stack()).sum(level=0)

# Join the dummy variables to the original DataFrame and drop the amenities columns
nashvilleDF = nashvilleDF.join(amenities_dummies)
nashvilleDF = nashvilleDF.drop(["amenities", "aggregated_amenities"], axis=1)




# Convert columns to numeric and handle NaN values
columns_to_convert = [
    'review_scores_rating', 'review_scores_accuracy', 'review_scores_cleanliness',
    'review_scores_checkin', 'review_scores_communication', 'review_scores_location',
    'review_scores_value'
]

for col in columns_to_convert:
    nashvilleDF[col] = pd.to_numeric(nashvilleDF[col], errors='coerce')
    nashvilleDF[col].fillna(0, inplace=True)


# Define the size of the entire figure. (width, height)
fig, (ax1, ax2, ax3, ax4, ax5, ax6, ax7) = plt.subplots(7, 1, figsize=(10, 14))

ax1.hist(nashvilleDF.review_scores_rating, color='blue')
ax1.set_title('Rating')
ax2.hist(nashvilleDF.review_scores_accuracy, color='blue')
ax2.set_title('Accuracy')
ax3.hist(nashvilleDF.review_scores_cleanliness, color='blue')
ax3.set_title('Cleanliness')
```

```python
ax4.hist(nashvilleDF.review_scores_checkin, color='blue')
ax4.set_title('Check In')
ax5.hist(nashvilleDF.review_scores_communication, color='blue')
ax5.set_title('Communication')
ax6.hist(nashvilleDF.review_scores_location, color='blue')
ax6.set_title('Location')
ax7.hist(nashvilleDF.review_scores_value, color='blue')
ax7.set_title('Value')

# Adjust the spacing between subplots
plt.subplots_adjust(hspace=0.5)

plt.tight_layout()  # This ensures that the titles and plots don't overlap
plt.show()


def findOutliers(df, column, lim_scalar=4):
    """
    Returns outliers above the max limit for a column in a dataframe
    Adjust outlier cutoff to q75 + 4*iqr to include more data
    ---
    input: DataFrame, column(series),lim_scalar(float)
    output: DataFrame
    """
    q25, q50, q75 = df[column].quantile(q=[0.25, 0.5, 0.75])
    iqr = q75 - q25
    # max limits to be considered an outlier
    max_ = q75 + lim_scalar * iqr
    # identify the points
    outlier_mask = [True if x > max_ else False for x in df[column]]
    print('{} outliers found out of {} data points, {}% of the data. {} is the max'.format(
        sum(outlier_mask), len(df[column]),
        100 * (sum(outlier_mask) / len(df[column])),max_))
    return outlier_mask
#average number of bathrooms in nashvilleDF
# Bathrooms
# Fill na with 1
nashvilleDF.bathrooms.fillna(1,inplace=True)
nashvilleDF.bathrooms.mean()
# Look at the distribution of the bathrooms column
plt.figure(figsize=(3,3))
plt.boxplot(nashvilleDF.bathrooms)
sns.despine()
plt.title('bathrooms distribution');



# Remove bathroom outliers
nashvilleDF = nashvilleDF[np.logical_not(
    findOutliers(nashvilleDF, 'bathrooms'))]
# Look at the distribution of the bathrooms column
plt.figure(figsize=(3,3))
plt.boxplot(nashvilleDF.bathrooms)
sns.despine()
plt.title('bathrooms distribution')

```

```python


# Convert non-numeric values in the 'bedrooms' column to NaN
nashvilleDF['bedrooms'] = pd.to_numeric(nashvilleDF['bedrooms'], errors='coerce')

# Replace NaN values with 1
nashvilleDF['bedrooms'].fillna(1, inplace=True)

# Plot the boxplot for the cleaned 'bedrooms' column
plt.figure(figsize=(3,3))
plt.boxplot(nashvilleDF['bedrooms'])
sns.despine()
plt.title('beds distribution')
plt.show()




# Remove bedroom outliers
nashvilleDF = nashvilleDF[np.logical_not(
    findOutliers(nashvilleDF, 'bedrooms',lim_scalar=6))]


# Plot the boxplot for the cleaned 'bedrooms' column
plt.figure(figsize=(3,3))
plt.boxplot(nashvilleDF['bedrooms'])
sns.despine()
plt.title('beds distribution')
plt.show()


# Plot distribution of the number of bathrooms
plt.figure(figsize=(10, 6))
plt.hist(nashvilleDF['bathrooms'], bins=20, color='blue', edgecolor='black')
plt.title('Distribution of Number of Bathrooms')
plt.xlabel('Number of Bathrooms')
plt.ylabel('Number of Listings')
plt.grid(axis='y')
plt.show()

# Plot distribution of the number of bathrooms
plt.figure(figsize=(10, 6))
plt.hist(nashvilleDF['bedrooms'], bins=20, color='blue', edgecolor='black')
plt.title('Distribution of Number of Bathrooms')
plt.xlabel('Number of Bedrooms')
plt.ylabel('Number of Listings')
plt.grid(axis='y')
plt.show()



# Calculate estimated number of bookings
nashvilleDF['est_bookings'] = nashvilleDF.number_of_reviews * 2

# Replace entries where unit is brand new with est_bookings = 1
nashvilleDF['est_bookings'] = [
    1 if nashvilleDF.first_review.iloc[idx] == nashvilleDF.last_review.iloc[idx] else x
    for idx, x in enumerate(nashvilleDF.est_bookings)
]
# Convert 'minimum_nights' to numeric and handle NaN values (assuming a default value of 1)
nashvilleDF['minimum_nights'] = pd.to_numeric(nashvilleDF['minimum_nights'], errors='coerce')
nashvilleDF['minimum_nights'].fillna(1, inplace=True)

# Calculate estimated number of nights booked per year
# Use 3 days as the average length of a stay
# Unless the minimum number of days is greater than 3, then use that number
nashvilleDF['est_booked_nights_per_year'] = [
    3 if x < 3 else x
    for x in nashvilleDF.minimum_nights  # avg stay length
] * nashvilleDF.reviews_per_month * 2 * 12
# Calculate estimated number of nights booked
# Use 3 days as the average length of a stay
# Unless the minimum number of days is greater than 3, then use that number

nashvilleDF['est_booked_nights'] = (
    [
        3 if x < 3 else x
        for x in nashvilleDF.minimum_nights
    ] *  # avg stay length
    nashvilleDF['est_bookings'])
# Convert 'est_booked_nights' to numeric and handle NaN values
nashvilleDF['est_booked_nights'] = pd.to_numeric(nashvilleDF['est_booked_nights'], errors='coerce')
nashvilleDF['est_booked_nights'].fillna(0, inplace=True)

# Occupancy Rate = total_booked_nights / total_available_nights
nashvilleDF['occupancy_rate'] = nashvilleDF['est_booked_nights'] / (
    (nashvilleDF.last_review - nashvilleDF.first_review).dt.days + 1)

# The next line seems redundant and is the same as the previous one. Consider removing it.
# Occupancy Rate = total_booked_nights / total_available_nights
nashvilleDF['occupancy_rate'] = nashvilleDF['est_booked_nights'] / (
    (nashvilleDF.last_review - nashvilleDF.first_review).dt.days + 1)
# Convert 'availability_365' to numeric and handle NaN values
nashvilleDF['availability_365'] = pd.to_numeric(nashvilleDF['availability_365'], errors='coerce')
nashvilleDF['availability_365'].fillna(0, inplace=True)


# Convert 'est_booked_nights_per_year' to numeric and handle NaN values
nashvilleDF['est_booked_nights_per_year'] = pd.to_numeric(nashvilleDF['est_booked_nights_per_year'], errors='coerce')
nashvilleDF['est_booked_nights_per_year'].fillna(0, inplace=True)

# Calculate occupancy rate
#nashvilleDF['occupancy_rate2'] = nashvilleDF['est_booked_nights_per_year'] / (nashvilleDF['availability_365'] + 1)
Getting rid of data that isnt numerical or helpful.

columns_to_drop = [
    "listing_url",
    "scrape_id",
    "source",
    "name",
    "description",
    "neighborhood_overview",
    "picture_url",
    "host_id",
    "host_url",
    "host_name",
    "host_location",
    "host_about",
    "host_thumbnail_url",
    "host_picture_url",
    "host_neighbourhood",
    "host_verifications",
    "neighbourhood_cleansed",
    "neighbourhood_group_cleansed",
    "latitude",
    "longitude",
    'property_type',
    'room_type',
    'est_booked_nights_per_year',
    'est_booked_nights',
    'occupancy_rate',
    'host_listings_count',
    'est_bookings',
    'calculated_host_listings_count',
    'calculated_host_listings_count_private_rooms',
    'minimum_nights_avg_ntm',
    'maximum_nights_avg_ntm',
    'calculated_host_listings_count_entire_homes',
    'host_total_listings_count',
    'availability_90',
    'availability_60', 'minimum_nights',
    'maximum_minimum_nights', 'review_scores_communication',
    'smaller', 'ge', 'irish spring', 'lotion','suave', 'dove', 'dove anti-stress moisturizing cream bar','dr teals',
     'smaller fridge',
    'organic', 'olympic-sized','number_of_reviews_l30d', 'number_of_reviews_ltm',
    '5-10 years old', '2-5 years old',


]
nashvilleDF['price'] = nashvilleDF['price'].str.replace('$', '').str.replace(',', '').astype(float)

nashvilleDF = nashvilleDF.drop(columns=columns_to_drop, errors='ignore')


# Convert the 'price' column to numeric (assuming it contains strings representing numbers)
nashvilleDF['price'] = pd.to_numeric(nashvilleDF['price'], errors='coerce')

try:
    # Calculate the required statistics
    total_entries = len(nashvilleDF['price'])
    entries_500_or_less = len(nashvilleDF[nashvilleDF['price'] <= 500])
    entries_over_500 = len(nashvilleDF[nashvilleDF['price'] > 1000])

    results = total_entries, entries_500_or_less, entries_over_500
except Exception as e:
    error_message = str(e)
    results = None

```

```
results, error_message if results is None else None


#drop first_review  last_review last_scraped  host_since
nashvilleDF.drop(['first_review','last_review','last_scraped','host_since',
                  'calendar_updated','calendar_last_scraped','license'],axis=1,inplace=True)

#host_acceptance_rate has % on it, drop the % from it
nashvilleDF['host_acceptance_rate'] = nashvilleDF['host_acceptance_rate'].str.replace('%', '').astype(float)
missing_data = nashvilleDF.isnull().sum()
print(missing_data[missing_data > 0])



nashvilleDF = nashvilleDF.apply(pd.to_numeric, errors='coerce')

nashvilleDF["host_response_rate"] = nashvilleDF["host_response_rate"].fillna(nashvilleDF["host_response_rate"].median())
nashvilleDF["host_acceptance_rate"] =
     nashvilleDF["host_acceptance_rate"].fillna(nashvilleDF["host_acceptance_rate"].median())

# Impute missing values for beds with the median
nashvilleDF["beds"] = nashvilleDF["beds"].fillna(nashvilleDF["beds"].median())

# Convert reviews_per_month to an integer column
nashvilleDF["reviews_per_month"] = nashvilleDF["reviews_per_month"].astype("float64")

# Impute missing values for reviews_per_month and occupancy_rate with the mean
nashvilleDF["reviews_per_month"] = nashvilleDF["reviews_per_month"].fillna(nashvilleDF["reviews_per_month"].mean())
#limit price between 200-300
nashvilleDF = nashvilleDF[nashvilleDF['price'] >= 000]


nashvilleDF = nashvilleDF[nashvilleDF['price'] <= 1000]
missing_data = nashvilleDF.isnull().sum()
print(missing_data[missing_data > 0])
Series([], dtype: int64)
#write nashvilleDF to a csv file
nashvilleDF.to_csv('nashvilleDF.csv', index=False)
plt.figure(figsize=(10, 6))
sns.histplot(nashvilleDF['price'], bins=50, kde=True)
plt.title('Distribution of Price')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()


# Split data into features (X) and target (y)
X = nashvilleDF.drop('price', axis=1)
y = nashvilleDF['price']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize linear regression model
model = LinearRegression()

```

```python
# Train the model
model.fit(X_train, y_train)

# Predict on test set
y_pred = model.predict(X_test)
# Calculate and print R^2 score
r2 = r2_score(y_test, y_pred)
print(f"R^2 Score: {r2}")


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor, HistGradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
import xgboost as xgb

subsetdf = nashvilleDF.copy()
df = subsetdf.copy()

def prepare_data(df, target_col='price'):
    X = df.drop(target_col, axis=1)
    y = df[target_col]
    return train_test_split(X, y, test_size=0.2, random_state=42)

def create_models():
    return {
        'GradientBoosting': GradientBoostingRegressor(),
        'RandomForest': RandomForestRegressor(),
        'Linear': LinearRegression(),
        'HistGradientBoosting': HistGradientBoostingRegressor(),
        'DecisionTree': DecisionTreeRegressor(),
        'XGBoost': xgb.XGBRegressor(),
    }

def fit_models(X_train, y_train, models):
    for name, model in models.items():
        model.fit(X_train.astype(float), y_train.astype(float))
    return models

def evaluate_models(X_test, y_test, models):
    results = []
    for name, model in models.items():
        y_pred = model.predict(X_test.astype(float))

        r2 = r2_score(y_test.astype(float), y_pred)
        mse = mean_squared_error(y_test.astype(float), y_pred)
        rmse = np.sqrt(mse)
        mae = mean_absolute_error(y_test.astype(float), y_pred)

        results.append({
            'Model': name,
            'R2': r2,
            'RMSE': rmse,
            'MSE': mse,
            'MAE': mae
        })
    return pd.DataFrame(results)

def plot_model_performance(results_df):
    fig, axes = plt.subplots(2, 2, figsize=(10, 8))
    fig.suptitle('Model Performance Comparison')

    metrics = ['R2', 'RMSE', 'MSE', 'MAE']

    for idx, metric in enumerate(metrics):
        ax = axes[idx // 2, idx % 2]
        results_df.plot(x='Model', y=metric, kind='bar', ax=ax, legend=None)
        ax.set_ylabel(metric)
        ax.set_ylim(bottom=0)
        ax.set_xticklabels(results_df['Model'], rotation=45, ha='right')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

def main(df):
    X_train, X_test, y_train, y_test = prepare_data(df)
    models = create_models()
    models = fit_models(X_train, y_train, models)
    results_df = evaluate_models(X_test, y_test, models)
    plot_model_performance(results_df)
    table = pd.pivot_table(results_df, index='Model', values=['R2', 'RMSE', 'MSE', 'MAE'])
    print(table)

main(df)









from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance
from sklearn.model_selection import KFold

# Initialize the HistGradientBoostingRegressor model
model = HistGradientBoostingRegressor()

# Specify the number of folds for k-fold cross-validation
num_folds = 10

# Initialize the KFold object
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)


# Initialize an array to store the accuracy scores
scores = np.zeros(num_folds)
```

```

# Initialize an array to store feature importances for each fold
# The number of columns in X is assumed to be X.shape[1]
feature_importances = np.zeros((num_folds, X.shape[1]))

# Loop through each fold
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    # Split the data into training and test sets
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_test, y_test = X.iloc[test_index], y.iloc[test_index]

    # Train your model on the training set
    model.fit(X_train, y_train)

    # Evaluate your model on the test set
    scores[fold] = model.score(X_test, y_test)

    # Get feature importances using permutation importance
    perm_importance = permutation_importance(model, X_test, y_test, n_repeats=10, random_state=42)
    feature_importances[fold] = perm_importance.importances_mean

    # Print the results
    print(f'Fold {fold+1}: Accuracy = {scores[fold]}')

# Compute and print the average accuracy across all folds
print(f'Average Accuracy: {np.mean(scores)}')

# Compute the average feature importance across all folds
avg_feature_importances = feature_importances.mean(axis=0)

# Get the top 20 features based on average importance
top_features = np.argsort(avg_feature_importances)[-20:]
top_feature_names = X.columns[top_features]
top_feature_importances = avg_feature_importances[top_features]

top_feature_names, top_feature_importances






import pandas as pd
import numpy as np
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance
from collections import defaultdict

# Define function to get feature importances
def get_feature_importance(data, features):
    X = data[features]
    y = data['price']

    model = HistGradientBoostingRegressor()
    model.fit(X, y)

    # Using permutation importance to get feature importances
    result = permutation_importance(model, X, y, n_repeats=10, random_state=42)
    importances = result.importances_mean

    return dict(zip(features, importances))

# Iteratively get feature importances for random feature subsets
iterations = 100  # Number of iterations
num_features = 20

feature_columns = [col for col in nashvilleDF.columns if col != 'price']

collective_importances = defaultdict(float)

for _ in range(iterations):
    selected_features = np.random.choice(feature_columns, num_features, replace=False)

    importances = get_feature_importance(nashvilleDF, selected_features)

    for feature, importance in importances.items():
        collective_importances[feature] += importance

# Sort features by their collective importance
sorted_features = sorted(collective_importances.keys(), key=lambda x: collective_importances[x], reverse=True)

print(sorted_features)

from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance

# Split the data into features and target
X = nashvilleDF.drop('price', axis=1)
y = nashvilleDF['price']

# Initialize the model
model = HistGradientBoostingRegressor()

# Fit the model
model.fit(X, y)

# Get feature importances using permutation importance
perm_importance = permutation_importance(model, X, y, n_repeats=30, random_state=42)

# Get the top 20 features
top_20_features = sorted(zip(perm_importance.importances_mean, X.columns), reverse=True)[:20]
top_20_features



# Categorizing amenities into defined buckets
kitchen_amenities = [
    "coffee", "kitchen", "refrigerator", "dishes and silverware", "microwave",
    "oven", "stove", "cooking basics", "freezer", "wine glasses", "toaster",
    "hot water kettle", "grill", "blender", "barbecue utensils", "rice maker",
    "nespresso", "espresso machine", "french press", "bread maker", "dining table",
    "baking sheet", "gas", "we have a countertop burner"
]

cleaning_amenities = [
    "dryer", "washer", "shampoo", "iron", "hot water", "soap", "cleaning products",
    "shower gel", "conditioner", "cleaning available during stay", "body wash",
    "lotion", "suave", "irish spring", "la botegga", "dove anti-stress moisturizing cream bar"
]

safety_amenities = [
    "smoke alarm", "carbon monoxide alarm", "fire extinguisher", "first aid kit",
    "keypad", "smart lock", "lockbox", "camera", "lock on bedroom door", "safe",
    "baby safety gates", "outlet covers", "table corner guards", "window guards",
    "ev charger", "ev charger - level 2", "ev charger - level 1", "tesla only"
]

household_amenities = [
    "radiant heating", "essentials", "hangers", "self check-in", "bed linens",
    "free parking", "air conditioning", "private entrance", "extra pillows and blankets",
    "dedicated workspace", "free street parking", "long term stays allowed", "clothing storage",
    "ceiling fan", "central air conditioning", "private patio or balcony", "room-darkening shades",
    "luggage dropoff allowed", "single level home", "elevator", "fireplace",
    "books and reading material", "laundromat nearby", "board games", "backyard",
    "outdoor furniture", "patio or balcony", "outdoor dining area", "gym",
    "fire pit", "high chair", "crib", "pool", "hot tub", "breakfast", "resort access",
    "city skyline view", "exercise equipment", "portable fans", "childrenu2019s dinnerware",
    "building staff", "babysitter recommendations", "free driveway", "closet",
    "shared patio or balcony", "record player", "trash compactor", "sun loungers",
    "closet", "wardrobe", "portable heater", "bidet", "changing table", "bikes",
    "hammock", "2-5 years old", "5-10 years old", "and 10 years old", "and 5-10 years old",
    "rooftop", "saltwater", "infinity", "beach essentials", "mosquito net", "heated",
    "dvd player", "ev charger - level 1", "private sauna", "shared sauna", "window ac unit",
    "host greets you", "piano", "baby monitor", "ping pong table", "private living room",
    "portable air conditioning", "smoking allowed", "baby bath", "beach access", "ski-in/ski-out",
    "wood-burning", "outdoor shower", "ac - split type ductless system", "window ac unit",
    "all natural", "stainless", "portable", "sono", "but good size not standard large size",
    "smaller", "some type", "two racks", "organic", "olympic-sized", "smaller fridge"
]

bedroom_amenities = [
    "hangers", "bed linens", "extra pillows and blankets", "room-darkening shades",
    "crib", "single level home", "closet", "wardrobe", "and dresser", "changing table"
]

electronics_amenities = [
    "wifi", "tv", "ethernet connection", "cable", "roku", "hulu", "disney", "hbo",
    "netflix", "amazon prime video", "bluetooth", "game console", "xbox 360", "and xbox one",
    "sound system", "vizio soundbar", "chromecast"
]

extra_spaces_amenities = [
    "patio or balcony", "pool", "garden or backyard", "hot tub", "BBQ grill",
    "private living room", "gym", "fire pit", "outdoor dining area", "backyard",
    "outdoor furniture", "private patio or balcony", "gym", "fireplace", "rooftop",
    "beach access", "ski-in/ski-out", "outdoor shower", "resort access", "waterfront",
```

```python
    "resort view", "harbor view", "beach view", "sea view", "ocean view",
    "lake access", "mountain view", "valley view", "golf course view", "lake view",
    "marina view", "public or shared beach access", "canal view", "boat slip", "bay view",
    "park view"
]

review_amenities = [
    "number_of_reviews", "number_of_reviews_ltm", "number_of_reviews_l30d", "review_scores_rating",
    "review_scores_accuracy", "review_scores_cleanliness", "review_scores_checkin", "review_scores_communication",
    "review_scores_location", "review_scores_value"
]


# Return the categorized amenities
categorized_amenities = {
    "Kitchen Amenities": kitchen_amenities,
    "Cleaning Amenities": cleaning_amenities,
    "Safety Amenities": safety_amenities,
    "Household Amenities": household_amenities,
    "Bedroom Amenities": bedroom_amenities,
    "Electronics Amenities": electronics_amenities,
    "Extra Spaces Amenities": extra_spaces_amenities,
    "Review Amenities": review_amenities

}
import networkx as nx
import matplotlib.pyplot as plt

# Create a new directed graph
G = nx.DiGraph()

# Add nodes for each amenity group
amenity_groups_names = [
    "Kitchen Amenities", "Cleaning Amenities", "Safety Amenities",
    "Household Amenities", "Bedroom Amenities", "Electronics Amenities",
    "Extra Spaces Amenities", "Review Amenities"
]
for group in amenity_groups_names:
    G.add_node(group)

# Add edges from each amenity group to 'price'
for group in amenity_groups_names:
    G.add_edge(group, 'price')

# Plot the DAG
plt.figure(figsize=(10, 6))

# Use shell_layout instead of spring_layout
shell_nestings = [amenity_groups_names, ['price']]  # Define the concentric circles
pos = nx.shell_layout(G, nlist=shell_nestings)

nx.draw_networkx(G, pos, with_labels=True, node_size=4000, node_color='skyblue', font_size=10, font_weight='bold',
     width=2, edge_color='gray')
plt.title("Causal DAG representing the effect of amenity groups on price")
plt.show()

```

```python


from sklearn.model_selection import KFold
import numpy as np

# Load your data into a pandas DataFrame
data = nashvilleDF.copy()

# Split your data into features and target
X = data.drop('price', axis=1)
y = data['price']

# Define the number of folds
num_folds = 10

# Initialize the cross-validation method
kf = KFold(n_splits=num_folds)

# Initialize a linear regression model
model = HistGradientBoostingRegressor()

# Initialize an array to store the accuracy scores
scores = np.zeros(num_folds)

# Loop through each fold
for fold, (train_index, test_index) in enumerate(kf.split(X)):

    # Split the data into training and test sets
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_test, y_test = X.iloc[test_index], y.iloc[test_index]

    # Train your model on the training set
    model.fit(X_train, y_train)

    # Evaluate your model on the test set
    scores[fold] = model.score(X_test, y_test)

    # Print the results
    print(f'Fold {fold+1}: Accuracy = {scores[fold]}')

# Compute and print the average accuracy across all folds
print(f'Average Accuracy: {np.mean(scores)}')


model1_plot = nashvilleDF.groupby(['accommodates', 'bedrooms']).size().reset_index().pivot(columns='bedrooms',
                                                                                  index='accommodates', values=0)

model1_plot.plot(kind='bar', stacked=True, figsize=(12,5))
plt.title("Accomodates and Number of Bedrooms", size=20)
plt.xlabel("Accomodates", size=15)




# Filter rows where 'wifi' column equals 1
wifi_1_rows = nashvilleDF[nashvilleDF['wifi'] == 1]
```

```

# Calculate the percentage
percent_with_wifi_1 = (len(wifi_1_rows) / len(nashvilleDF)) * 100

print(f"Percentage of rows with 'wifi' = 1 in nashvilleDF: {percent_with_wifi_1:.2f}%")



# Filter rows where 'wifi' column equals 1
resort_1_rows = nashvilleDF[nashvilleDF['resort access'] == 1]

# Calculate the percentage
percent_with_resort_1 = (len(resort_1_rows) / len(nashvilleDF)) * 100

print(f"Percentage of rows with 'wifi' = 1 in nashvilleDF: {percent_with_resort_1:.2f}%")

from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance
from sklearn.model_selection import train_test_split, KFold
from joblib import Parallel, delayed
import numpy as np

def process_fold(fold, train_index, test_index, X, y):
    model = HistGradientBoostingRegressor()
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_test, y_test = X.iloc[test_index], y.iloc[test_index]
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    perm_importance = permutation_importance(model, X_test, y_test, n_repeats=10, random_state=42)
    sorted_idx = perm_importance.importances_mean.argsort()[::-1]
    sorted_features = [(X.columns[i], perm_importance.importances_mean[i]) for i in sorted_idx]
    return score, perm_importance.importances_mean, sorted_features

# Assuming nashvilleDF is defined
X = nashvilleDF.drop(['price', 'id'], axis=1)
y = nashvilleDF['price']
X.columns = X.columns.astype(str)

num_folds = 10
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)
scores = np.zeros(num_folds)
feature_importances = np.zeros((num_folds, X.shape[1]))
results = Parallel(n_jobs=-1)(delayed(process_fold)(fold, train_index, test_index, X, y)
                              for fold, (train_index, test_index) in enumerate(kf.split(X)))

accuracy_scores = []
for fold, (score, importance, sorted_features) in enumerate(results):
    scores[fold] = score
    feature_importances[fold] = importance

    # Append the score to accuracy_scores list
    accuracy_scores.append(score)

    print(f'Fold {fold+1}: Accuracy = {scores[fold]}')
    print("Features ranked by importance for fold:", fold+1)
    for name, imp in sorted_features:
        print(f"{name}: {imp}")
```

```

# Compute and print the average accuracy and standard deviation across all folds
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
print(f'Average Accuracy: {mean_accuracy}')
print(f'Standard Deviation of Accuracy: {std_accuracy}')

# Compute the average feature importance across all folds
avg_feature_importances = feature_importances.mean(axis=0)

# Get the top 20 features based on average importance
top_features = np.argsort(avg_feature_importances)[-20:]
top_feature_names = X.columns[top_features]
top_feature_importances = avg_feature_importances[top_features]

print("Top 20 Features and their Importances:")
print(list(zip(top_feature_names, top_feature_importances)))


all_top_features = []
for _, _, sorted_features in results:
    top_20 = [feature[0] for feature in sorted_features[:20]]
    all_top_features.extend(top_20)

# Calculate the average rank for each feature
feature_ranks = {feature: 0 for feature in set(all_top_features)}
for feature in all_top_features:
    feature_ranks[feature] += (all_top_features.count(feature) - all_top_features.index(feature) - 1)

average_ranks = {feature: rank / all_top_features.count(feature) for feature, rank in feature_ranks.items()}

# Sort by average rank
sorted_avg_ranks = sorted(average_ranks.items(), key=lambda x: x[1], reverse=True)

# Displaying the sorted average ranks
for feature, rank in sorted_avg_ranks:
    print(f"{feature}: {rank}")

# Plotting the graph
features = [item[0] for item in sorted_avg_ranks]
ranks = [item[1] for item in sorted_avg_ranks]

plt.figure(figsize=(12, 10))
plt.barh(features, ranks, color='skyblue')
plt.xlabel('Average Rank')
plt.ylabel('Feature Name')
plt.title('Average Rank of Top Features')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```

APPENDIX 2

CAUSAL DISCOVERY

B.1 Large Language Model Causal Discovery

```python
!pip install langchain.agents
import os
from itertools import combinations

import numpy as np
from scipy import linalg
from scipy import stats

import matplotlib.pyplot as plt

from langchain.agents import load_tools, initialize_agent
from langchain.agents import AgentType

from langchain.chat_models import ChatOpenAI

from castle.common import GraphDAG
from castle.metrics import MetricsDAG
from castle.algorithms import PC

from castle.common.priori_knowledge import PrioriKnowledge


COLORS = [
    '#00B0F0',
    '#FF0000',
    '#B0F000'
]



def check_if_dag(A):
    return np.trace(linalg.expm(A * A)) - A.shape[0] == 0




#Put in your OPENAI KEY HERE
os.environ['OPENAI_API_KEY'] = "sk-KEYASGFASGUASOGUHA"

all_vars = {
    'altitude': 0,
    'oxygen_density': 1,
    'temperature': 2,
    'risk_of_death': 3,
    'mehendretex': 4
}


SAMPLE_SIZE = 1000

altitude = stats.halfnorm.rvs(scale=2000, size=SAMPLE_SIZE)
temperature = 25 - altitude / 100 + stats.norm.rvs(
    loc=0,
    scale=2,
    size=SAMPLE_SIZE
)

mehendretex = stats.halfnorm.rvs(size=SAMPLE_SIZE)

oxygen_density = np.clip(
    1 - altitude / 8000
    - temperature / 50
    + stats.norm.rvs(size=SAMPLE_SIZE) / 20,
    0,
    1)

risk_of_death = np.clip(
    altitude / 20000
    + np.abs(temperature) / 100
    - oxygen_density / 5
    - mehendretex / 5
    + stats.norm.rvs(size=SAMPLE_SIZE) / 10,
    0,
    1
)



dataset = np.stack(
    [
        altitude,
        oxygen_density,
        temperature,
        risk_of_death,
        mehendretex
    ]
).T



true_dag = np.array(
    [
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0]
    ]
```

```
)


# PC discovery without LLM assist
pc = PC(variant='stable')
pc.learn(dataset)

print(f'Is learned matrix a DAG: {check_if_dag(pc.causal_matrix)}')

# Vizualize
GraphDAG(
    est_dag=pc.causal_matrix,
    true_dag=true_dag)

plt.show()

# Compute metrics
metrics = MetricsDAG(
    B_est=pc.causal_matrix,
    B_true=true_dag)

print(metrics.metrics)

# Instantiate and encode priori knowledge
priori_knowledge = PrioriKnowledge(n_nodes=len(all_vars))

llm = ChatOpenAI(
    temperature=0, # Temp == 0 => we want clear reasoning
    model='gpt-4')#'gpt-3.5-turbo')


# Load tools
tools = load_tools(
    [
        "wikipedia"
    ],
    llm=llm)


# Instantiate the agent
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    handle_parsing_errors=True,
    verbose=False)



def get_llm_info(llm, agent, var_1, var_2):

    out = agent(f"Does {var_1} cause {var_2} or the other way around?\
    We assume the following definition of causation:\
    if we change A, B will also change.\
    The relationship does not have to be linear or monotonic.\
    We are interested in all types of causal relationships, including\
    partial and indirect relationships, given that our definition holds.\
    ")

    print(out)

    pred = llm.predict(f'We assume the following definition of causation:\
    if we change A, B will also change.\
    Based on the following information: {out["output"]},\
    print (0,1) if {var_1} causes {var_2},\
    print (1, 0) if {var_2} causes {var_1}, print (0,0)\
    if there is no causal relationship between {var_1} and {var_2}.\
    Finally, print (-1, -1) if you don\'t know. Importantly, don\'t try to\
    make up an answer if you don\'t know.')

    print(pred)

    return pred

for var_1, var_2 in combinations(all_vars.keys(), r=2):
    print(var_1, var_2)
    out = get_llm_info(llm, agent, var_1, var_2)
    if out=='(0,1)':
        priori_knowledge.add_required_edges(
            [(all_vars[var_1], all_vars[var_2])]
        )

        priori_knowledge.add_forbidden_edges(
            [(all_vars[var_2], all_vars[var_1])]
        )

    elif out=='(1,0)':
        priori_knowledge.add_required_edges(
            [(all_vars[var_2], all_vars[var_1])]
        )
        priori_knowledge.add_forbidden_edges(
            [(all_vars[var_1], all_vars[var_2])]
        )

print('\nLLM knowledge vs true DAG')
priori_dag = np.clip(priori_knowledge.matrix, 0, 1)

print(f'\nChecking if priori graph is a DAG: {check_if_dag(priori_dag)}')

GraphDAG(
    est_dag=priori_dag,
    true_dag=true_dag)

plt.show()

print('\nRunning PC')

# Instantiate the model with expert knowledge
pc_priori = PC(
    priori_knowledge=priori_knowledge,
    variant='stable'
)
```

```python

# Learn
pc_priori.learn(dataset)

GraphDAG(
    est_dag=pc_priori.causal_matrix,
    true_dag=true_dag)

plt.show()

# Compute metrics
metrics = MetricsDAG(
    B_est=pc_priori.causal_matrix,
    B_true=true_dag)

print(metrics.metrics)

import numpy as np

# Define the order of variables
variables_order = ["accommodates", "price", "bedrooms", "beds", "bathrooms"]

# Initialize the adjacency matrix with zeros
n_vars = len(variables_order)
true_dag = np.zeros((n_vars, n_vars))

# Define the relationships as given in the digraph format
relationships = [
    ("accommodates", "price"),
    ("bedrooms", "accommodates"),
    ("beds", "bedrooms"),
    ("bathrooms", "bedrooms"),
    ("bathrooms", "price")
]

# Fill in the matrix based on the relationships
for source, target in relationships:
    i = variables_order.index(source)
    j = variables_order.index(target)
    true_dag[i][j] = 1

print(true_dag)



import os
from itertools import combinations
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from castle.common import GraphDAG
from castle.metrics import MetricsDAG
from castle.algorithms import PC

import os
from itertools import combinations
```

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from castle.common import GraphDAG
from castle.metrics import MetricsDAG
from castle.algorithms import PC
from castle.common.priori_knowledge import PrioriKnowledge

# Load the dataset
file_path = r"C:\Users\nstep\TSU\SeniorProject\df_selected1.csv"
nashvilleDF = pd.read_csv(file_path)
data_subset = nashvilleDF[['accommodates', 'price','bedrooms','beds','bathrooms']].dropna().values

# Define the order of variables and create the true_dag
variables_order = ["accommodates", "price", "bedrooms", "beds", "bathrooms"]
all_vars = {var: idx for idx, var in enumerate(variables_order)}
n_vars = len(variables_order)
true_dag = np.zeros((n_vars, n_vars))
relationships = [
    ("accommodates", "price"),
    ("bedrooms", "accommodates"),
    ("beds", "bedrooms"),
    ("bathrooms", "bedrooms"),
    ("bathrooms", "price")
]
for source, target in relationships:
    i = variables_order.index(source)
    j = variables_order.index(target)
    true_dag[i][j] = 1

# Run PC discovery without LLM assist
pc = PC(variant='stable')
pc.learn(data_subset)

# Visualize the result
GraphDAG(
    est_dag=pc.causal_matrix,
    true_dag=true_dag)

plt.show()

# Compute metrics
metrics = MetricsDAG(
    B_est=pc.causal_matrix,
    B_true=true_dag)
print(metrics.metrics)

# Instantiate and encode priori knowledge
priori_knowledge = PrioriKnowledge(n_nodes=len(all_vars))

# Instantiate the GPT agent (You'll need to provide the appropriate setup and imports for the LLM)
# This step is based on the code you provided
llm = ChatOpenAI(
    temperature=0, # Temp == 0 => we want clear reasoning
    model='gpt-4')#'gpt-3.5-turbo')

```

```python
tools = load_tools(
    ["wikipedia"],
    llm=llm)

agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    handle_parsing_errors=True,
    verbose=False)

def get_llm_info(llm, agent, var_1, var_2):

    out = agent(f"Does {var_1} cause {var_2} or the other way around?\
    We assume the following definition of causation:\
    if we change A, B will also change.\
    The relationship does not have to be linear or monotonic.\
    We are interested in all types of causal relationships, including\
    partial and indirect relationships, given that our definition holds.\
    ")

    print(out)

    pred = llm.predict(f'We assume the following definition of causation:\
    if we change A, B will also change.\
    Based on the following information: {out["output"]},\
    print (0,1) if {var_1} causes {var_2},\
    print (1, 0) if {var_2} causes {var_1}, print (0,0)\
    if there is no causal relationship between {var_1} and {var_2}.\
    Finally, print (-1, -1) if you don\'t know. Importantly, don\'t try to\
    make up an answer if you don\'t know.')

    print(pred)

    return pred

# Add priori knowledge from the LLM
for var_1, var_2 in combinations(all_vars.keys(), r=2):
    print(var_1, var_2)
    out = get_llm_info(llm, agent, var_1, var_2)
    if out=='(0,1)':
        priori_knowledge.add_required_edges([(all_vars[var_1], all_vars[var_2])])
        priori_knowledge.add_forbidden_edges([(all_vars[var_2], all_vars[var_1])])
    elif out=='(1,0)':
        priori_knowledge.add_required_edges([(all_vars[var_2], all_vars[var_1])])
        priori_knowledge.add_forbidden_edges([(all_vars[var_1], all_vars[var_2])])

# Check the priori knowledge
print('\nLLM knowledge vs true DAG')
priori_dag = np.clip(priori_knowledge.matrix, 0, 1)
print(f'\nChecking if priori graph is a DAG: {check_if_dag(priori_dag)}')

GraphDAG(
    est_dag=priori_dag,
    true_dag=true_dag)
plt.show()
```

```

# Now, run the PC algorithm with priori knowledge
print('\nRunning PC with Priori Knowledge')

# Instantiate the model with expert knowledge
pc_priori = PC(
    priori_knowledge=priori_knowledge,
    variant='stable'
)

# Learn using the dataset
pc_priori.learn(data_subset)

# Visualize
GraphDAG(
    est_dag=pc_priori.causal_matrix,
    true_dag=true_dag)
plt.show()

# Compute metrics
metrics_priori = MetricsDAG(
    B_est=pc_priori.causal_matrix,
    B_true=true_dag)
print(metrics_priori.metrics)
```

B.2 Causal Model

```
import dowhy
from dowhy import CausalModel

import numpy as np
import pandas as pd
import graphviz
import networkx as nx

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)

file_path = r"C:\Users\nstep\TSU\SeniorProject\df_selected1.csv"
nashvilleDF = pd.read_csv(file_path)

def make_graph(adjacency_matrix, labels=None):
    idx = np.abs(adjacency_matrix) > 0.01
    dirs = np.where(idx)
    d = graphviz.Digraph(engine='dot')
    names = labels if labels else [f'x{i}' for i in range(len(adjacency_matrix))]
    for name in names:
        d.node(name)
    for to, from_, coef in zip(dirs[0], dirs[1], adjacency_matrix[idx]):
        d.edge(names[from_], names[to], label=str(coef))
    return d

def str_to_dot(string):
    '''
    Converts input string from graphviz library to valid DOT graph format.
    '''
    graph = string.strip().replace('\n', ';').replace('\t','')
    graph = graph[:9] + graph[10:-2] + graph[-1] # Removing unnecessary characters from string
    return graph

print(nashvilleDF.shape)

nashvilleDF.head()


# Drop rows where any of the specified columns has a value of 1
columns_to_check = ['prop_Entire condo', 'prop_Entire guest suite', 'prop_Entire guesthouse',
                    'prop_Entire rental unit', 'prop_Entire townhouse', 'prop_Hotel',
                    'prop_Private room', 'free parking', 'room_Private room']

nashvilleDF = nashvilleDF[~nashvilleDF[columns_to_check].eq(1).any(axis=1)]

# Drop 'id' and the specified columns
columns_to_drop = ['id','prop_Entire home', 'room_Entire home/apt',] + columns_to_check
nashvilleDF = nashvilleDF.drop(columns=columns_to_drop)
nashvilleDF.head()

import graphviz

# Given DOT representation
causal_graph_updated = f"""
digraph {{
    grill -> price;
    pool -> price;
    "hot tub" -> price;
    "hot tub" -> bathrooms;
    pool -> bathrooms;
    "private entrance" -> price;
    "private entrance" -> "resort access";
    fireplace -> price;

    neighbourhood_cleansed_num -> price;
    neighbourhood_cleansed_num -> review_scores_location;
    review_scores_location -> review_scores_rating;
    review_scores_location -> reviews_per_month;
    review_scores_rating -> reviews_per_month;
    reviews_per_month -> price;
    host_is_superhost -> reviews_per_month;
    host_is_superhost -> price;
    host_is_superhost -> review_scores_rating;
    host_acceptance_rate -> reviews_per_month;
    host_acceptance_rate -> price;
    beds -> bedrooms;
    bedrooms -> accommodates;
    bedrooms -> bathrooms;
    accommodates -> price;
    "resort access" -> pool;
    "resort access" -> "hot tub";
    pool -> host_acceptance_rate;
    "hot tub" -> host_acceptance_rate;
    bathrooms -> price;
    minimum_minimum_nights -> price;
    minimum_maximum_nights -> price;

    "resort access" -> price;
}}
"""


model=CausalModel(
        data = nashvilleDF,
        treatment='accommodates',
        outcome='price',
        graph=causal_graph_updated)

model.view_model()
from IPython.display import Image, display
display(Image(filename="causal_model.png"))


# Identification
identified_estimand = model.identify_effect(proceed_when_unidentifiable=True)
print(identified_estimand)



from cdt.causality.graph import LiNGAM, PC, GES, SAM,CAM
import graphviz
from sklearn.preprocessing import StandardScaler


import os
os.environ['R_HOME'] = 'C:\\Program Files\\R\\R-4.3.1'
os.environ["PATH"] += os.pathsep + 'C:\\Program Files\\Graphviz\\bin'

scaler = StandardScaler()
nashvilleDF_scaled = pd.DataFrame(scaler.fit_transform(nashvilleDF), columns=nashvilleDF.columns)
from sklearn.decomposition import PCA

pca = PCA(n_components=min(nashvilleDF_scaled.shape) - 1)  # -1 to ensure the matrix is non-singular
nashvilleDF_pca = pca.fit_transform(nashvilleDF_scaled)




graphs = {}
labels = [f'{col}' for i, col in enumerate(nashvilleDF.columns)]
functions = {
    'LiNGAM' : LiNGAM,
    'PC': PC,
    'GES': GES,
    'SAM': SAM,
}


for method, lib in functions.items():
    obj = lib()
    output = obj.predict(nashvilleDF_scaled)
    adj_matrix = nx.to_numpy_array(output)
    adj_matrix = np.asarray(adj_matrix)
    graph_dot = make_graph(adj_matrix, labels)
    graphs[method] = graph_dot

# Visualize graphs
for method, graph in graphs.items():
    print("Method : %s" % (method))
    display(graph)



for method, graph in graphs.items():
        if method != "LiNGAM":
            continue
        print('\n*****************************************************************************\n')
        print("Causal Discovery Method : %s"%(method))

        # Obtain valid dot format
        graph_dot = str_to_dot(graph.source)

        # Define Causal Model
        model=CausalModel(
                data = nashvilleDF,
                treatment='accommodates',
                outcome='price',
                graph=graph_dot)

        # Identification
        identified_estimand = model.identify_effect(proceed_when_unidentifiable=True)
        print(identified_estimand)

        # Estimation
        estimate = model.estimate_effect(identified_estimand,
                                        method_name="backdoor.linear_regression",
                                        control_value=0,
                                        treatment_value=1,
                                        confidence_intervals=True,
                                        test_significance=True)
        print("Causal Estimate is " + str(estimate.value))





import graphviz

# Given DOT representation
causal_graph_updated = f"""
digraph {{
    grill -> price;
    private_entrance -> price;
    pool -> price;
    pool -> grill;
    "hot tub" -> price;
    "hot tub" -> bathrooms;
    pool -> bathrooms;
    "private entrance" -> price;
    "private entrance" -> "resort access";
    fireplace -> price;
    prop_Entire_Home -> price;

    neighbourhood_cleansed_num -> price;
    neighbourhood_cleansed_num -> review_scores_location;

    review_scores_location -> reviews_per_month;
    review_scores_rating -> reviews_per_month;
    reviews_per_month -> price;
    host_is_superhost -> reviews_per_month;
    host_is_superhost -> price;
    host_is_superhost -> review_scores_rating;
    host_acceptance_rate -> reviews_per_month;
    host_acceptance_rate -> price;
    beds -> bedrooms;
    bedrooms -> accommodates;
    accommodates -> bathrooms;
    accommodates -> price;
    "resort access" -> pool;
    "resort access" -> "hot tub";
    bathrooms -> price;

    minimum_minimum_nights -> price;
    minimum_maximum_nights -> price;

    "resort access" -> price;
}}
"""



#visualize the updated causal graph
graph = graphviz.Source(causal_graph_updated)
graph # display the graph






# Define Causal Model
model=CausalModel(
        data = nashvilleDF,
        treatment='accommodates',
        outcome='price',
        graph=causal_graph_updated)

# Identification
identified_estimand = model.identify_effect(proceed_when_unidentifiable=True)
print(identified_estimand)

```

```
# Estimation
estimate = model.estimate_effect(identified_estimand,
                                 method_name="backdoor.linear_regression",
                                 control_value=0,
                                 treatment_value=1,
                                 confidence_intervals=True,
                                 test_significance=True)
print("Causal Estimate is " + str(estimate.value))



# Refutation
# Add a random common cause variable
refutation = model.refute_estimate(identified_estimand, estimate,
                                   method_name="data_subset_refuter")
print(refutation)


# Refutation
# Add a random common cause variable
refutation = model.refute_estimate(identified_estimand, estimate,
                                   method_name="random_common_cause")
print(refutation)

from dowhy.causal_estimators import CausalEstimator
from dowhy.causal_refuters.data_subset_refuter import DataSubsetRefuter
from upload_data_page import upload_data
from feature_engineering import feature_engineering, display_data_preview, display_handle_missing_values,
     display_process_currency_percentage
from feature_engineering import display_drop_columns, display_data_transformation, display_encode_categorical,
     display_time_series_features, display_convert_to_datetime
from explore_data import explore_data, display_boxplot, display_binary_distribution, feature_comparison_graph_page
from regression import evaluate_model_page, display_model_performance_comparison, prepare_data, create_models,
     fit_models, evaluate_models, plot_model_performance
from regression import display_select_target_features_and_train, display_feature_importance,
     display_prediction_vs_actual, display_residuals_plot
from regression import display_correlation_heatmap, evaluate_model_page
from advance_data_analysis import advanced_data_analysis, perform_classification, perform_clustering,
     perform_dimensionality_reduction
from time_series_analysis import time_series_analysis, visualize_time_series_data,display_acf_pacf, fit_arima_model
import streamlit as st
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.ensemble import GradientBoostingRegressor, RandomForestClassifier, RandomForestRegressor,
     HistGradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
import xgboost as xgb
import seaborn as sns
import matplotlib.pyplot as plt
import streamlit as st
import pandas as pd
from sklearn.model_selection import train_test_split
```

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_classif
import plotly.express as px
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
import base64
from prophet import Prophet
import os
from dowhy import CausalModel
import re
import graphviz
from graphviz import Digraph
from io import BytesIO

def extract_relationships_from_dot(dot_representation):
    """Extract relationships from DOT format."""
    relationships = []

    # Split the string into lines and filter out non-edge lines
    lines = dot_representation.split("\n")
    edge_lines = [line.strip() for line in lines if "->" in line]

    for edge_line in edge_lines:
        # Extract cause and effect from the edge line
        cause, effect = edge_line.split("->")
        relationships.append((cause.strip(), effect.strip()))

    return relationships

def display_relationships_definition():
    st.subheader("Define Causal Relationships")

    # Ensure data is uploaded
    if 'data' not in st.session_state or st.session_state.data is None:
        st.warning("Please upload data first.")
        return

    columns = list(st.session_state.data.columns)

    # Dropdowns to select cause and effect columns
    cause_column = st.selectbox("Select Cause Column", columns)
    effect_column = st.selectbox("Select Effect Column", columns)

    # Add relationship button
    if st.button("Add Relationship"):
        relationship = (cause_column, effect_column)
        if "relationships" not in st.session_state:
            st.session_state.relationships = []
        if relationship not in st.session_state.relationships:
            st.session_state.relationships.append(relationship)
            st.success(f"Added relationship: {cause_column} -> {effect_column}")

    # Display existing relationships and allow removal
    if "relationships" in st.session_state and st.session_state.relationships:
        st.write("Defined Relationships:")

        st.write("Select Relationship to Remove:")
        # Create a dropdown with all relationships formatted as "cause -> effect"
        relationship_options = [f"{cause} -> {effect}" for cause, effect in st.session_state.relationships]
        selected_relationship_str = st.selectbox("Select Relationship:", relationship_options)

        # Extract cause and effect from the selected string
        selected_cause, selected_effect = selected_relationship_str.split(" -> ")

        if st.button(f"Remove {selected_cause} -> {selected_effect}"):
            st.session_state.relationships.remove((selected_cause, selected_effect))
            st.success(f"Removed relationship: {selected_cause} -> {selected_effect}")

    # Upload a .dot file and load the graph
    uploaded_file = st.file_uploader("Upload a .dot file to load a causal graph", type="dot")
    if uploaded_file:
        uploaded_graph = uploaded_file.read().decode()

        # Extract relationships from the uploaded graph
        uploaded_relationships = extract_relationships_from_dot(uploaded_graph)

        # Add the extracted relationships to st.session_state.relationships
        if "relationships" not in st.session_state:
            st.session_state.relationships = []

        for relationship in uploaded_relationships:
            if relationship not in st.session_state.relationships:
                st.session_state.relationships.append(relationship)

        st.success(f"Loaded {len(uploaded_relationships)} relationships from the provided .dot file.")

    # Generate causal graph button
    if st.button("Generate Causal Graph"):
        if "relationships" not in st.session_state or not st.session_state.relationships:
            st.warning("Please define at least one relationship before generating the graph.")
        else:
            # Generate graph using Graphviz
            dot = Digraph()
            for cause, effect in st.session_state.relationships:
                dot.edge(cause, effect)

            # Convert dot to string and save in session state
            st.session_state.dot_representation = dot.source
            st.session_state.generated_graph = True

            # Display the graph
            st.graphviz_chart(dot)

            # Provide a download link for the graph
            # Save the graph representation as a temporary .dot file
            temp_filename = "causal_graph.dot"
            with open(temp_filename, "w") as f:
                f.write(dot.source)

            st.markdown(generate_download_link(temp_filename, "Download causal graph (.dot)"), unsafe_allow_html=True)
```

```python


def display_causal_model_creation():
    """Sub-task for creating the causal model based on the defined graph."""

    columns = list(st.session_state.data.columns)

    # Check if the causal graph has been generated
    if not st.session_state.get("generated_graph", False):
        st.warning("Please generate the causal graph first.")
        return

    # Ensure that the dot_representation is not empty
    dot_representation = st.session_state.get("dot_representation", "")
    if not dot_representation:
        st.warning("Please generate or upload a causal graph first.")
        return

    # Causal model creation
    treatment = st.selectbox("Select Treatment (cause) Variable", columns)
    outcome = st.selectbox("Select Outcome (effect) Variable", columns)
    st.write("""
    The treatment variable is what you believe to be the cause in your causal relationship,
    and the outcome variable is the effect you are studying.
    """)

    if st.button("Create and Estimate Causal Model"):
        # Define Causal Model
        model = CausalModel(
            data=st.session_state.data,
            treatment=treatment,
            outcome=outcome,
            graph=st.session_state.get("dot_representation", "")
        )


        # Identification
        identified_estimand = model.identify_effect(proceed_when_unidentifiable=True)
        st.session_state.identified_estimand = identified_estimand
        st.write("Identified estimand:", identified_estimand)

        # Estimation
        estimate = model.estimate_effect(identified_estimand,
                                         method_name="backdoor.linear_regression",
                                         control_value=0,
                                         treatment_value=1,
                                         confidence_intervals=True,
                                         test_significance=True)
        st.write("Causal Estimate:", estimate.value)

        st.session_state.causal_model = model
        st.session_state.estimate = estimate
        st.success("Causal model created and estimated successfully!")



```

```python

def generate_download_link(filename, download_text):
    """Generate a download link for a given file and link text."""
    with open(filename, "rb") as f:
        file_data = f.read()
    b64 = base64.b64encode(file_data).decode()
    href = f'<a href="data:application/octet-stream;base64,{b64}" download="{filename}">{download_text}</a>'
    return href








def display_refutation_tests():
    """Sub-task for running refutation tests."""
    st.subheader("Refutation Tests")

    # Ensure that the causal model is created
    if "causal_model" not in st.session_state:
        st.warning("Please create a causal model first.")
        return

    # Refutation methods
    methods = ["Data Subset Refuter"]
    chosen_method = st.selectbox("Choose a Refutation Method", methods)

    # Customize parameters based on the chosen method
    if chosen_method == "Data Subset Refuter":
        subset_fraction = st.slider("Choose a fraction of data to keep", 0.1, 1.0, 0.5)

    # Run refutation
    if st.button("Run Refutation"):
        if "identified_estimand" not in st.session_state or "estimate" not in st.session_state:
            st.warning("Please create and estimate the causal model first.")
            return

        refuter = DataSubsetRefuter(
            data=st.session_state.data,
            causal_model=st.session_state.causal_model,
            identified_estimand=st.session_state.identified_estimand,
            estimate=st.session_state.estimate,
            subset_fraction=subset_fraction
        )
        results = refuter.refute_estimate()
        st.write("Refutation Results:", results)

        # Extract p_value from the results
        # Assuming it's in the results string, you might need to adjust the extraction method
        p_value_str = re.search(r'p value:(\d+\.\d+)', str(results))
        if p_value_str:
            p_value = float(p_value_str.group(1))
        else:
            p_value = None

        # Interpretation based on p-value and difference in effects
        original_effect = st.session_state.estimate.value
        new_effect = results.new_effect

        if p_value and p_value > 0.05 and abs(original_effect - new_effect) < 0.05 * abs(original_effect):  # Assuming a 5% relative difference threshold for "close"
            st.write("Interpretation: The original causal estimate is consistent and robust, even when using a subset of the data.")
        elif p_value and p_value <= 0.05:
            st.write("Interpretation: The original causal estimate may not be reliable, as it changes significantly with a subset of the data.")






def causality_page():
    st.header("Causality Analysis")

    # Ensure data is uploaded
    if 'data' not in st.session_state or st.session_state.data is None:
        st.warning("Please upload data first.")
        return

    # Moved task selection to sidebar
    task = st.sidebar.radio("Choose a Causality Sub-task", ["Define Relationships", "Create Causal Model", "Run Refutation Tests"])

    if task == "Define Relationships":
        display_relationships_definition()
    elif task == "Create Causal Model":
        display_causal_model_creation()
    elif task == "Run Refutation Tests":
        display_refutation_tests()
```

APPENDIX 3

WEB APPLICATION

C.1 Prediction Page

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <!-- Allows for the styling of the web page -->
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='Styles/style.css') }}">

    <title>Predictor</title>
</head>

<body>
    <!-- The tool bar on the web app -->
    <div class="topnav">
        <a class="active" href="{{ url_for('predictor') }}">Predictor</a>
    </div>

    <h1 >Price Prediction</h1>

    <form method="POST" action="{{ url_for('predictor') }}">
        {{ form.hidden_tag() }}
        <table>
            <th>
                {{ form.accommodates.label }}<br>{{ form.accommodates(class="input-field") }}<br>
                {% if form.accommodates.errors %}
                    <div class="error">
                    {% for error in form.accommodates.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.bathrooms.label }}<br>{{ form.bathrooms(class="input-field") }}<br>
                {% if form.bathrooms.errors %}
                    <div class="error">
                    {% for error in form.bathrooms.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.nights_staying.label }}<br>{{ form.nights_staying(class="input-field") }}<br>
                {% if form.nights_staying.errors %}
                    <div class="error">
                    {% for error in form.nights_staying.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.bedrooms.label }}<br>{{ form.bedrooms(class="input-field") }}<br>
                {% if form.bedrooms.errors %}
                    <div class="error">
                    {% for error in form.bedrooms.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.property_type.label }}<br>{{ form.property_type(class="input-field") }}<br>
            </th>

            <th>
                {{ form.beds.label }}<br>{{ form.beds(class="input-field") }}<br>
                {% if form.beds.errors %}
                    <div class="error">
                    {% for error in form.beds.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.review_scores_rating.label }}<br>{{ form.review_scores_rating(class="input-field") }}<br>
                {% if form.review_scores_rating.errors %}
                    <div class="error">
                    {% for error in form.review_scores_rating.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.review_scores_location.label }}<br>{{ form.review_scores_location(class="input-field") }}<br>
                {% if form.review_scores_location.errors %}
                    <div class="error">
                    {% for error in form.review_scores_location.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.host_acceptance_rate.label }}<br>{{ form.host_acceptance_rate(class="input-field") }}<br>
                {% if form.host_acceptance_rate.errors %}
                    <div class="error">
                    {% for error in form.host_acceptance_rate.errors %}
                        <p>Invalid character. Please enter number.</p>
                    {% endfor %}
                    </div>
                {% endif %}
                {{ form.room_type.label }}<br>{{ form.room_type(class="input-field") }}<br>
            </th>

            <th>
                {{ form.fireplace.label }}<br>{{ form.fireplace() }}<br>
                {{ form.hot_tub.label }}<br>{{ form.hot_tub() }}<br>
                {{ form.grill.label }}<br>{{ form.grill() }}<br>
                {{ form.pool.label }}<br>{{ form.pool() }}<br>
                {{ form.private_entrance.label }}<br>{{ form.private_entrance() }}<br>
                {{ form.free_parking.label }}<br>{{ form.free_parking() }}<br>
                {{ form.resort_access.label }}<br>{{ form.resort_access() }}<br>
                {{ form.neighbourhood_cleansed_num.label }}<br>{{ form.neighbourhood_cleansed_num(class="input-field")
     }}<br>
            </th>
        </table>
        {{ form.submit(class="submit-button") }}<br>

        <center>
            <img
     src="https://filetransfer.nashville.gov/portals/0/sitecontent/Planning/images/FrontPage/DistrictsAndCommunities600.jpg"
     alt="Nashville District Map">
        </center>
    </form>

    {% if estimated_price %}
        <div class="results-container">
            <div class="result">
                Predicted Price: ${{ estimated_price }}
            </div>
        </div>
    {% endif %}
</body>

</html>
```

C.2 Price Page

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <!-- Allows for the styling of the web page -->
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='Styles/style.css') }}">

    <title>Airbnb Price Result</title>
</head>
<body>
    <!-- The tool bar on the web app -->
    <div class="topnav">
        <a href="{{ url_for('predictor') }}">Predictor</a>
        <a class="active" href="{{ url_for('price') }}">Price</a>
    </div>

    <div class="price-container">
        <div class="price-content">
            <p>Predicted Price: ${{ estimated_price }}</p>
            <p>MAE: ${{ mae }} (This indicates the average deviation of the predicted price from actual prices. A lower
     value means the prediction is more accurate.)</p>
        </div>
    </div>

    <hr>
    <h2 style="text-align: center;">Recommended Airbnb Listings:</h2>
    <div style="text-align: center; margin-top: 20px;">
        {% for prop in recommended_properties.iterrows() %}
            <div>
                <a href="https://www.airbnb.com/rooms/{{ prop[1]['id']|int }}">Property ID: {{ prop[1]['id']|int }}</a>
            </div>
        {% endfor %}
    </div>
</body>
</html>
```

C.3 Style Sheet

```
body {
  height: 100%;
  background-color: #ffffff;
  color: rgb(0, 0, 0);
  font-family:Cambria, Cochin, Georgia, Times, 'Times New Roman', serif;
}

 /* Adds style to the input fields */
input[type=text] {
  width: 50%;
  padding: 12px 20px;
  margin: 8px 0;
  box-sizing: border-box;
  border: 2px solid black;
  border-radius: 4px;
  background-color: rgba(128, 128, 128, 0.281);
}

/* Adds style to the buttons */
button {
  background-color: #6e6e6e51;
  border: 2px solid black;
  color: rgb(250, 248, 248);
  padding: 15px 32px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 12px;
}

table {
  width: 100%;
  height: 400px;
  border-collapse: collapse;
}
```

```css
th, td {
  padding: 15px;
  text-align: left;
  border-bottom: 1px solid #ddd;
  text-align: center;
}

/* The error statement will show as Red */
.error {
  color: red;
}

/* Adds style to the drop down menus */
select {
  appearance: none;
  background-color: rgba(128, 128, 128, 0.281);
  border: 2px solid black;
  border-radius: 4px;
  outline: black;
  padding: 12px 20px;
  margin: 8px 0;
  width: 50%;
  font-family: inherit;
  font-size: inherit;
  color: rgb(0, 0, 0);
}

/* Add a black background color to the top navigation */
.topnav {
  margin: 0;
  background-color: transparent;
  overflow: hidden;
}

/* Style the links inside the navigation bar */
.topnav a {
  float: left;
  color: #000000;
  text-align: center;
  padding: 14px 16px;
  text-decoration: none;
  font-size: 17px;
}

/* Change the color of links on hover */
.topnav a:hover {
  background-color: #ddd;
  color: black;
}

/* Add a color to the active/current link */
.topnav a.active {
  background-color: #0d06f3;
  color: white;
}

```

```
.column {
  float: left;
  width: 50%;
}

/* Clear floats after the columns */
.row:after {
  content: "";
  display: table;
  clear: both;
}

form {
  text-align: center;
}

/* Center results on the page */
.results-container {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  height: calc(100vh - 50px); /* Adjust 50px based on your actual navbar height */
}

.result {
  font-size: 24px;
  margin-bottom: 20px;
}

.price-container {
  text-align: center; /* This will center the content horizontally */
  padding-top: 5rem; /* Adjust this value to increase/decrease space on top */
  padding-bottom: 1rem; /* Adjust this value to increase/decrease space at the bottom */
}

.price-content {
  padding-bottom: 3rem; /* Adjust this value to increase/decrease space below the MAE before the line */
}


hr {
  margin: 0 auto; /* This centers the horizontal line */
  width: 80%; /* Adjusts the width of the line */
}


h2 {
  margin-top: 3rem; /* Adjust this value to increase/decrease space above the title */
  margin-bottom: 1rem; /* Adjust this value to increase/decrease space below the title */
}
```

C.4 Back-end

```
import pandas as pd
from flask import Flask, render_template, request
from flask_wtf import FlaskForm
from wtforms import FloatField, SubmitField, BooleanField, SelectField
from wtforms.validators import DataRequired, InputRequired, NumberRange
from joblib import dump, load
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error
import warnings
from sklearn.model_selection import train_test_split
warnings.filterwarnings(action='ignore', category=UserWarning, module='sklearn')

app = Flask(__name__, static_folder='Static')
app.config['SECRET_KEY'] = 'wdazD1dRmBGVwVSi'

# Load or train the model
try:
    trained_model = load('trained_model.joblib')
    mean_features = nashvilleDF.drop(['price', 'id'], axis=1).mean().to_dict()
except:
    nashvilleDF = pd.read_csv('nashvilleDF.csv')
    mean_features = nashvilleDF.mean().to_dict()
    X = nashvilleDF.drop(['price', 'id'], axis=1)
    y = nashvilleDF['price']
    trained_model = HistGradientBoostingRegressor()
    trained_model.fit(X, y)
    dump(trained_model, 'trained_model.joblib')



X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

trained_model.fit(X_train, y_train)
y_pred = trained_model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)


# Prediction Page Forms
class PredictionForm(FlaskForm):
    accommodates = FloatField('How many people do you want to stay?', validators=[DataRequired(), NumberRange(min=1)])
    bathrooms = FloatField('Enter the number of bathrooms (e.g. 1.5)', validators=[DataRequired(), NumberRange(min=0)])
    nights_staying = FloatField('How many nights are you staying?', validators=[DataRequired(), NumberRange(min=1)])
    grill = BooleanField('Do you want a grill?')
    bedrooms = FloatField('Enter the number of bedrooms', validators=[DataRequired(), NumberRange(min=0)])
    fireplace = BooleanField('Do you want a fireplace?')
    hot_tub = BooleanField('Do you want a hot tub?')
    private_entrance = BooleanField('Do you want a private entrance?')
    free_parking = BooleanField('Do you want free parking?')
    review_scores_rating = FloatField('Enter desired review score (1-5 or percentage)', validators=[DataRequired(),
     NumberRange(min=0, max=100)])
    review_scores_location = FloatField('Enter desired location review score (1-5 or percentage)',
     validators=[DataRequired(), NumberRange(min=0, max=100)])
    resort_access = BooleanField('Do you want resort access?')
    beds = FloatField('Enter number of beds', validators=[DataRequired(), NumberRange(min=1)])
    pool = BooleanField('Do you want a pool?')
    host_acceptance_rate = FloatField('Enter desired host acceptance rate (0-100)', validators=[DataRequired(),
     NumberRange(min=0, max=100)])
    neighbourhood_cleansed_num = SelectField('Choose Neighbourhood (Chart below)', choices=[
        (1, 'District 1'), (2, 'District 2'), (3, 'District 3'), (4, 'District 4'), (5, 'District 5'), (6, 'District 6'), (7, 'District 7'), (8, 'District 8'),
        (9, 'District 9'), (10, 'District 10'), (11, 'District 11'), (12, 'District 12'), (13, 'District 13'), (14, 'District 14'), (15, 'District 15'), (16, 'District 16'),
        (17, 'District 17'), (18, 'District 18'), (19, 'District 19'), (20, 'District 20'), (21, 'District 21'), (22, 'District 22'), (23, 'District 23'), (24, 'District 24'),
        (25, 'District 25'), (26, 'District 26'), (27, 'District 27'), (28, 'District 28'), (29, 'District 29'), (30, 'District 30'), (31, 'District 31'), (32, 'District 32'),
        (33, 'District 33'), (34, 'District 34'), (35, 'District 35')
    ])
    property_type = SelectField('Choose Property Type', choices=[
        ('prop_Entire condo', 'Entire Condo'),
        ('prop_Entire guest suite', 'Entire Guest Suite'),
        ('prop_Entire guesthouse', 'Entire Guesthouse'),
        ('prop_Entire home', 'Entire Home'),
        ('prop_Entire rental unit', 'Entire Rental Unit'),
        ('prop_Entire townhouse', 'Entire Townhouse'),
        ('prop_Hotel', 'Hotel'),
        ('prop_Private room', 'Private Room')
    ])
    room_type = SelectField('Choose Room Type', choices=[
    ('room_Entire home/apt', 'Entire home/apt'),
    ('room_Private room', 'Private Room')
])


    submit = SubmitField('Predict Price')

# Route to the Prediction Model Page
@app.route('/')
@app.route('/predictor', methods=['GET', 'POST'])
def predictor():
    form = PredictionForm()
    estimated_price = None

    if form.validate_on_submit():
        # Creates the user_data dictionary (extends form.data)
        user_data = {key: value for key, value in form.data.items() if key != 'nights_staying' and key != 'csrf_token'}

        # Convert specific fields to appropriate values
        for key, value in user_data.items():
            if value == 'Yes':
                user_data[key] = 1
            elif value == 'No':
                user_data[key] = 0
            elif isinstance(value, str) and "%" in value:
                user_data[key] = float(value.strip('%')) / 100

        # Handle one-hot encoding for property_type
        property_types = [
            'prop_Entire condo', 'prop_Entire guest suite', 'prop_Entire guesthouse',
            'prop_Entire home', 'prop_Entire rental unit', 'prop_Entire townhouse',
            'prop_Hotel', 'prop_Private room'
        ]
        selected_property_type = user_data.pop('property_type')
        for prop_type in property_types:
            user_data[prop_type] = 1 if prop_type == selected_property_type else 0

        # Ensure the Amenities are correctly encoded
        if 'hot_tub' in user_data:
            user_data['hot tub'] = user_data.pop('hot_tub')
        if 'grill' in user_data:
            user_data['grill'] = user_data.pop('grill')
        if 'private_entrance' in user_data:
            user_data['private entrance'] = user_data.pop('private_entrance')
        if 'free_parking' in user_data:
            user_data['free parking'] = user_data.pop('free_parking')
        if 'resort_access' in user_data:
            user_data['resort access'] = user_data.pop('resort_access')
        if 'pool' in user_data:
            user_data['pool'] = user_data.pop('pool')

        # Populate missing features with their average values
        prepared_data = mean_features.copy()
        prepared_data.update(user_data)

        if 'id' in prepared_data:
            del prepared_data['id']
        if 'price' in prepared_data:
            del prepared_data['price']
        if 'submit' in prepared_data:
            del prepared_data['submit']

        # Now, let's filter the nashvilleDF dataframe based on the user's criteria:

        # Start with the entire dataset and filter it step by step
        filtered_properties = nashvilleDF.copy()

        # 1. Accommodates``
        filtered_properties = filtered_properties[filtered_properties['accommodates'] >= user_data['accommodates']]

        # 2. Bathrooms
        filtered_properties = filtered_properties[filtered_properties['bathrooms'] >= user_data['bathrooms']]

        # 3. Nights staying
        # Use nights_staying to filter the properties
        nights_staying_value = form.data['nights_staying']
        filtered_properties = filtered_properties[
            (filtered_properties['minimum_minimum_nights'] <= nights_staying_value) &
            (filtered_properties['minimum_maximum_nights'] >= nights_staying_value)
        ]

        # Now, drop the key from the dictionary since it's not needed anymore
        user_data.pop('nights_staying', None)


        # 4. Bedrooms
        filtered_properties = filtered_properties[filtered_properties['bedrooms'] >= user_data['bedrooms']]

        # Ensuring the amenities correctly encoded
```

```python


    # 5. Amenities
    amenities = ['fireplace', 'hot tub', 'grill', 'private entrance', 'free parking', 'resort access', 'pool']
    for amenity in amenities:
        if user_data[amenity]:
            filtered_properties = filtered_properties[filtered_properties[amenity] == 1]

    # 6. Review scores
    min_review_score = user_data['review_scores_rating'] - 1
    filtered_properties = filtered_properties[filtered_properties['review_scores_rating'] > min_review_score]

    # 7. Property type
    # The filtering for property type is inherently done through the one-hot encoding process

    # 8. Room type
    # Handle one-hot encoding for room_type
    room_types = {
        'room_Entire home/apt': 'Entire home/apt',
        'room_Private room': 'Private Room'
    }
    selected_room_type = user_data.pop('room_type')
    for encoded_name, original_name in room_types.items():
        user_data[encoded_name] = 1 if original_name == selected_room_type else 0

    # 9. Location Review Scores
    min_review_score_location = user_data['review_scores_location'] - 1
    filtered_properties = filtered_properties[filtered_properties['review_scores_location'] >
min_review_score_location]

    # 10. Beds
    filtered_properties = filtered_properties[filtered_properties['beds'] >= user_data['beds']]

    # Neighbourhood Cleansed
    neighbourhood = [
        'District 1', 'District 2', 'District 3', 'District 4', 'District 5', 'District 6', 'District 7', 'District
8',
        'District 9', 'District 10', 'District 11', 'District 12', 'District 13', 'District 14', 'District 15',
'District 16',
        'District 17', 'District 18', 'District 19', 'District 20', 'District 21', 'District 22', 'District 23',
'District 24',
        'District 25', 'District 26', 'District 27', 'District 28', 'District 29', 'District 30', 'District 31',
'District 32',
        'District 33', 'District 34', 'District 35'
    ]
    selected_neighbuorhood = user_data.pop('neighbourhood_cleansed_num')
    for neighbourhood_num in neighbourhood:
        user_data[neighbourhood_num] = 1 if neighbourhood_num == selected_neighbuorhood else 0

    # Host Acceptance Rate
    filtered_properties = filtered_properties[filtered_properties['host_acceptance_rate'] >=
user_data['host_acceptance_rate']]

    # Compare features between model and user data
    model_features = set(X.columns)
    prepared_features = set(prepared_data.keys())
```

```

        missing_features = model_features - prepared_features
        extra_features = prepared_features - model_features

        print("Missing features:", missing_features)
        print("Extra features:", extra_features)

        # Predict the price using the trained model
        try:
            ordered_data_values = [prepared_data[feature] for feature in X.columns]

            estimated_price = trained_model.predict([ordered_data_values])[0]
            estimated_price = round(estimated_price, 2)

            recommended_properties = filtered_properties.sort_values(by='review_scores_rating', ascending=False).head(5)

            return render_template('price.html', estimated_price=estimated_price, mae=round(mae, 2),
     recommended_properties=recommended_properties)

        except Exception as e:
            print(f"Error during prediction: {str(e)}")

    return render_template('prediction.html', title='Prediction Model', form=form, estimated_price=estimated_price)

@app.route('/price')
def price():
    estimated_price = request.args.get('estimated_price', 'N/A')
    mae = request.args.get('mae', 'N/A')
    return render_template('price.html', estimated_price=estimated_price, mae=mae)


if __name__ == '__main__':
    app.run(debug=True)
```

BIBLIOGRAPHY

[1] Inside Airbnb. “Airbnb - Listings”. In: (2021), https://public.opendatasoft.com/explore/dataset/airbnb–listings/information/?disjunctive.host_verificationsdisjunctive.amenitiesdisjunctive.features.

[2] Peter Bruce and Andrew Bruce. “Practical Statistics for Data Scientists”. In: *O’Reilly* (2017).

[3] D. Mackenzi J. Pearl. “The book of why: The new science of cause and effect”. In: *New York, NY: Basic Books* (2020).

[4] Emre Kiciman and Amit Sharama. “Dowhy: An end-to-end library for causal inference”. In: *arXiv.org* 2020.04216 (2020).

[5] Rik Kraan. “Demystifying decision trees, random forests gradient boosting”. In: *Towards Data Science* (2020).

[6] A. Molak. “Jane the Discoverer: Enhancing Causal Discovery with Large Language Models (Causal Python)”. In: (2023), https://towardsdatascience.com/jane–the–discoverer–enhancing–causal–discovery–with–large–language–models–causal–python–564a63425c93.

[7] J. Pearl and D. Mackenzie. “The book of why: The new science of cause and effect”. In: *New York: Basic Books* (2020).

[8] V. Raul Perez-Sanchez Raul-Tomas Mora-Garcia Maria-Francisace Cespedes-Lopez. “Housing Price Prediction Using Machine Learning Algorithms in COVID-19 Times”. In: *MDPI* (2022).

[9] VB Staff. “Airbnb VP talks about AI’s profound impact on results”. In: *Venture Beat* (2017).

[10] Naveen Venkat. *The Curse of Dimensionality: Inside Out*. `https://www.researchgate.net/publication/327498046_The_Curse_of_Dimensionality_Inside_Out`. Accessed April 1, 2019.