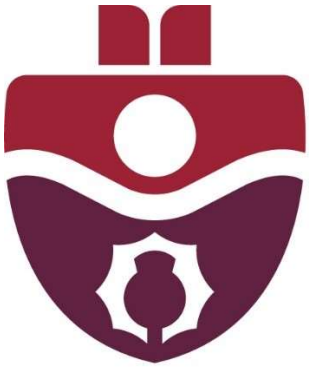

Saint Mary's University

Data Mining Competition

SmartRent: Revolutionizing Rental Market Analysis in Halifax

MCDA 5580: Data and Text Mining
MCDA 5560: Business Intelligence and Data Visualization

Team DNR

Aditya Chaudhari(A00477010)

Jeeten Jain(A00477424)

Sivleen Kaur(A00474480)

Rishabh Khevaria(A00476814)

Table of Contents

Executive Summary

Background

In an effort to revolutionize rental market analysis in Halifax, our team undertook a comprehensive data mining competition as part of MCDA 5580: Data and Text Mining. The project's goal was to employ advanced data collection and analysis techniques to accurately predict rental prices, providing stakeholders with actionable insights to navigate the complex rental landscape of Halifax.

Data Collection and Preprocessing

The team embarked on an extensive data collection exercise, scraping data from a variety of sources to ensure a representative sample of the Halifax rental market. The collected data spanned various critical variables such as price, location, property type, and amenities, among others. Following collection, the data underwent rigorous preprocessing, including cleaning, handling missing values, outlier removal, and standardization, to prepare it for analysis.

Methodology

Utilizing the prepared dataset, the team applied several machine learning models to predict rental prices. The models included Linear Regression, Random Forest, and XGBoost, each evaluated for its predictive accuracy using metrics such as R-squared, Adjusted R-squared, Mean Squared Error (MSE), Mean Absolute Error (MAE), and where applicable, Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE).

Results

The evaluation of the models revealed the XGBoost algorithm as the superior model, demonstrating the highest accuracy in predicting rental prices. This model outperformed others across all metrics, indicating its effectiveness in capturing the complex, non-linear relationships between the variables and rental prices. Key findings from the XGBoost model include:

High R-squared value (0.81033), indicating the model explains approximately 81.03% of the variance in rental prices.

The lowest MSE and MAE values among the models, suggesting accurate predictions with minimal error.

Significant feature importance attributed to variables such as property size, number of rooms, and location-based scores, highlighting their impact on rental prices.

Recommendations

Based on the analysis and results, Team DNR recommends stakeholders in the Halifax rental market to leverage the insights from the XGBoost model for decision-making. Property managers,

investors, and renters can use the model's predictions to understand the key drivers of rental prices, optimize listing prices, or make informed rental choices. Furthermore, the team suggests ongoing refinement of the model by incorporating additional data sources and exploring advanced feature engineering techniques to enhance predictive accuracy.

Data Collection

Overview

Our data collection strategy was meticulously designed to ensure a thorough and representative sampling of the Halifax rental market. To capture the multifaceted nature of the market, we employed a combination of data scraping techniques tailored to the unique characteristics of each target website. This comprehensive approach enabled us to collect a rich dataset that includes URL, title, price, location, description, attributes, property type, number of rooms, and other critical variables that affect rental prices.

Each technique was selected for its effectiveness in navigating the unique challenges posed by different online platforms.

- **API Scraping**: This process involves programmatically sending requests to the website's server to retrieve data. It's highly efficient for bulk extraction and yields structured data, suitable for immediate analysis.
- **Web Crawling**: This entails using bots to automatically traverse the web and collect information from various sites. Our web crawlers were sophisticated enough to handle complex site navigation, including the ability to interact with pagination and extract data embedded in the website's HTML or JavaScript.
- **Selenium Automation**: Selenium is a powerful tool that automates web browsers, allowing us to interact with web pages as a user would. This is particularly useful for sites that load data dynamically with user actions or those that implement anti-scraping measures.

Sites Scraped

The diversity of our data sources was a key element in providing a comprehensive understanding of the market:

- **Kijiji**: We utilized a web crawler specifically designed to handle the structure and depth of Kijiji's listings, capturing a wide array of data from one of the largest Canadian classified advertising websites. [7] [Script1] [Script2]
- **444Rent**: Another web crawler was implemented to comb through 444Rent's extensive database, ensuring every property listing was visited and relevant data was extracted. [10] [Script9] [Script10]
- **Templeton Apartments**: For Templeton Apartments, we first executed API scraping to collect individual listing URLs. A web crawler then meticulously extracted detailed data from each listing's URL. [6] [Script5] [Script6]
- **Capreit Apartments**: API scraping provided us with a comprehensive list of individual listing URLs from Capreit Apartments, which our web crawler followed for detailed data collection. [8] [Script3] [Script4]
- **Rent Seeker**: Similar to the above platforms, Rent Seeker's individual listing URLs were obtained via API scraping, and extensive data was then extracted using a web crawler. [13] [Script13] [Script14]
- **West22 Living**: We deployed a web crawler to navigate West22 Living's listings, a source offering high-end living options in Halifax. [12] [Script12]
- **Zumper**: For Zumper, we employed API scraping due to its effectiveness in efficiently gathering large volumes of structured data. [9] [Script7]

- **Apartments.com**: Apartments.com required a combination of Selenium for dynamic interaction and a web crawler to ensure comprehensive data collection. [11] [Script11]
- **Killiam Apartments**: Similarly, Killiam Apartments needed Selenium to navigate through the complex, dynamic content before our web crawler could extract the data. [5] [Script8]

Scraped Data Fields

Our data fields were selected through a strategic process aimed at capturing the most significant factors that influence rental prices. The rationale behind each field's selection is rooted in an understanding of the rental market's drivers, as well as the predictive modelling requirements.

- **URL**
  - **Purpose**: It acts as a unique identifier for each property listing.
  - **Reasoning**: A direct link allows us to reference the original listing for further details, follow-up, and data verification purposes. In the context of data integrity, it provides a method for tracing back and updating information should it change over time.
- **Title & Description**
  - **Purpose**: These contain descriptive text about the property.
  - **Reasoning**: Beyond basic information, the title and description often include keywords that signal a property's features and amenities that are not captured in other fields. They can indicate the level of luxury and the target demographic and often reflect marketing effectiveness, which can indirectly influence the rental price.
- ***Price***
  - ***Purpose****: Represents the rental cost of a property.*
  - ***Reasoning****: As the key* ***outcome variable*** *in our analysis, the price is the most direct reflection of a property's market value and the primary metric against which all other factors are assessed.*
  - ***Distinction****: Unlike the engineered features designed to inform and enhance our predictive model, the price is not a variable we manipulate or derive from other data. Instead, it stands as the target outcome we aim to predict. This distinction is crucial, as it underscores the objective of our analysis: to accurately forecast rental prices based on a comprehensive understanding of influencing factors. Understanding and predicting the price enables stakeholders to make informed decisions in the rental market, reflecting its central role in our analytical endeavors.*
- **Location**
  - **Purpose**: Indicates the geographical positioning of the property.
  - **Reasoning**: The desirability of a property's location, including its neighborhood, proximity to essential services, and overall geographic appeal is a top determinant of rental prices. Location analytics allow us to establish patterns in price distribution across regions.
- **Attributes & Property Type**
  - **Purpose**: Define the characteristics and structural category of a property.

    - **Reasoning**: These fields help differentiate properties in terms of features such as building type, available amenities, and age. This differentiation is crucial as it affects how a property is positioned in the market and its competitive pricing.
- **Rooms & Den Included**
    - **Purpose**: Quantify the living space in terms of rooms available.
    - **Reasoning**: The number and type of rooms directly correlate with the property's suitability for potential tenants, affecting its functional value and, consequently, its rental price. The inclusion of a den, often considered a luxury, can cater to needs for additional space or a home office, reflecting contemporary living trends.
- **Number of Bathrooms**
    - **Purpose**: Count the number of bathrooms within the property.
    - **Reasoning**: An indicator of both convenience and luxury, the number of bathrooms can significantly impact a property's desirability and can be a benchmark for comparison with similar properties, thereby affecting the rental price.
- **Latitude & Longitude**
    - **Purpose**: Provide precise geospatial coordinates.
    - **Reasoning**: These enable the integration of the listings with mapping and location analysis tools, allowing us to perform location-based pricing analytics and visualize the distribution of properties relative to various amenities and points of interest.
- **Parking Included & Utilities Included**
    - **Purpose**: Indicate the inclusion of additional conveniences.
    - **Reasoning**: In urban areas, parking availability is often a premium factor and can significantly influence the rental price. Similarly, the inclusion of utilities simplifies tenant expenses and can be a deciding factor for budget-conscious renters, thus impacting rental pricing strategies.
- **Pet Friendly**
    - **Purpose**: Shows whether a property is accommodating to tenants with pets.
    - **Reasoning**: A pet-friendly designation can expand the property's appeal to a broader demographic, potentially increasing demand and allowing for a premium in rental pricing in markets where pet-friendly options are limited.
- **Size (sqft)**
    - **Purpose**: Measures the property's square footage.
    - **Reasoning**: Size is a fundamental metric that has a proportional relationship with rental prices. It serves as a basis for price-per-square-foot calculations, which are standard in industry comparisons and valuations.
- **Furnished**
    - **Purpose**: Specify if the property comes furnished.
    - **Reasoning**: Furnishing a property can cater to short-term renters or those looking for a move-in-ready home, often justifying a higher rental price due to the added convenience and initial cost savings for tenants.
- **Building Name**
    - **Purpose**: States the name of the building or complex.

- **Reasoning**: The name can reflect the reputation, brand value, and perceived quality of the property. It can influence tenant perception and, by extension, the property's market position and rental price.

In summary, each data field was chosen for its evidence-based impact on rental prices and its utility in building a robust analytical model. The fields not only serve to directly inform about specific property characteristics but also offer insights into broader market trends, tenant preferences, and property positioning.

Engineered Fields

In our analysis of the Halifax rental market, "Engineered Fields" play a pivotal role, serving as a transformative step where raw data is meticulously crafted into meaningful variables. This process, known as feature engineering, allows us to incorporate domain expertise and uncover hidden patterns that directly influence rental prices. By refining and creating these variables, we enrich our dataset, enhancing the predictive power of our models. This section will highlight the techniques and rationale behind these engineered fields, illustrating their crucial contribution to our understanding of the factors that drive rental pricing dynamics in Halifax.

- **Walk Score**:
  - **Purpose**: Provides a measure of a location's walkability.
  - **Reasoning**: A high Walk Score often correlates with greater tenant demand, reflecting the desirability of a location based on the ability to perform errands without a car. It has become a key factor in tenant decision-making, particularly in urban areas.
  - **Implementation**: The Walk Score is derived by evaluating the walkability of an area based on the distance to amenities such as restaurants, parks, schools, and stores. This metric was integrated into our dataset by mapping property locations against a database of local amenities to calculate a score that quantifies each property's convenience and accessibility on foot, enriching our analysis with a spatial dimension that directly impacts rental appeal and pricing. For the data collection relevant to Halifax, Halifax's official website suggests using walkscore.com to get this parameter [1] [2]
- **Transit Score:**
  - **Purpose**: Offers an assessment of the accessibility and convenience of public transit from a location.
  - **Reasoning:** A high Transit Score indicates robust public transit options, which can significantly attract tenants who rely on public transportation for their daily commutes. This metric is particularly valuable in urban settings where traffic congestion and parking challenges make public transit a preferred option. The availability and efficiency of public transportation can influence a property's appeal, potentially leading to higher rental demand and prices in well-serviced areas.
  - **Implementation:** The Transit Score is calculated by analyzing various factors such as transit route density, frequency of service, and proximity to public transit stations. This feature was modeled by integrating transit data with geographical information, providing

a quantifiable measure that enhances our dataset's ability to predict rental prices based on transit accessibility. For the data collection relevant to Halifax, Halifax's official website suggests using walkscore.com to get this parameter. [1] [2]

- **Bike Score:**
  - **Purpose:** Measures the bike-friendliness of a location, based on bike lanes, hills, road connectivity, and access to destinations.
  - **Reasoning:** A high Bike Score suggests an environment that supports cycling, both for recreation and as a practical mode of transportation. This can attract a segment of tenants who prioritize sustainability and health, or who prefer biking over other forms of transit due to convenience or cost. In cities where traffic and parking are problematic, properties with a high Bike Score may see increased demand, reflecting the growing trend towards bike-friendly lifestyles and the appeal of locations that facilitate such choices.
  - **Implementation**: The Bike Score is calculated by assessing factors like the presence and quality of bike lanes, terrain flatness, and the proximity to essential services reachable by bike. This feature was modeled by combining geographic data analysis with city infrastructure information to quantify the ease of biking in and around a property's location. Incorporating the Bike Score into our dataset allows us to capture an additional dimension of a property's attractiveness, further refining our predictive model's ability to assess rental values in the Halifax market. For the data collection relevant to Halifax, Halifax's official website suggests using walkscore.com to get this parameter. [1] [2]
- **Time to Nearest Hospital:**
  - **Purpose:** Indicates the time required to reach the nearest hospital from a property, highlighting healthcare accessibility.
  - **Reasoning:** Proximity to healthcare facilities is a significant factor for many tenants, particularly those with medical needs or families with children. Properties that offer shorter travel times to hospitals can be more attractive, potentially influencing rental prices. This metric underscores the practical aspects of living in a particular location, emphasizing safety and accessibility as key components of property desirability.
  - **Implementation:** To calculate the Time to Nearest Hospital, we utilized the geographical coordinates (latitude and longitude) of each property and the locations of Halifax's hospitals. We then employed Google's Distance Matrix API to determine the travel time by the most efficient route to the nearest hospital. This approach allowed us to incorporate a precise and practical measure into our dataset, enhancing our model's capability to factor in healthcare accessibility when predicting rental prices. [3]
- **Time to Nearest Police Station:**
  - **Purpose:** Assesses the time required to reach the nearest police station from a property, reflecting on safety and emergency response accessibility.

    - **Reasoning:** The proximity to law enforcement facilities is a vital consideration for tenants prioritizing safety and quick emergency response times. Properties with shorter distances to police stations may be perceived as safer and more desirable, which can influence rental demand and pricing. This metric captures an essential aspect of location desirability that goes beyond conventional amenities, focusing on the sense of security it provides to potential tenants.
    - **Implementation:** To determine the Time to Nearest Police Station, we analyzed the geographical coordinates of each property alongside the locations of police stations within Halifax. Utilizing Google's Distance Matrix API, we calculated the travel times, offering a straightforward and reliable estimation of how quickly residents can access police services. This measure enriches our dataset by integrating a critical component of public safety, thereby offering a more comprehensive analysis of factors that affect rental values in the Halifax market. [3]
- **Time to Nearest Store**:
    - **Purpose**: Evaluates the time required to travel from a property to the nearest retail store, highlighting convenience and lifestyle compatibility.
    - **Reasoning**: Accessibility to retail stores is a crucial convenience factor for tenants, reflecting the ease with which daily necessities and services can be accessed. Shorter travel times to stores enhance a property's appeal, especially for those valuing proximity to shopping options. This can affect rental demand, as properties closer to retail amenities often command higher prices due to their added convenience.
    - **Implementation**: To assess the Time to Nearest Store, we utilized the geographical coordinates (latitude and longitude) of properties and identified the locations of the nearest retail stores in Halifax. By employing Google's Distance Matrix API, we were able to calculate the most efficient travel times to these retail outlets. This process provided us with a tangible measure of convenience, enriching our dataset with valuable information on how retail accessibility influences rental pricing in the market. [3]
- **Time to Nearest Store:**
    - **Purpose**: Measures the travel time required from a property to the nearest pharmacy, emphasizing health care convenience and accessibility.
    - **Reasoning**: Proximity to pharmacies is a key factor for tenants who prioritize easy access to medications and health care products. Properties that offer short travel times to pharmacies are likely to be more attractive, particularly for individuals with health conditions or those seeking the convenience of nearby health services. This proximity can influence rental prices, as it contributes to the overall desirability of a location by ensuring essential health care needs are readily accessible.

- **Implementation**: To calculate the Time to Nearest Pharmacy, we analyzed the latitude and longitude of each property in our dataset and identified the locations of pharmacies in Halifax. Utilizing Google's Distance Matrix API, we determined the quickest travel times to these pharmacies. This approach allowed us to accurately gauge the convenience offered by a property in terms of health care accessibility, providing a valuable dimension to our analysis that highlights how such amenities can impact rental market dynamics. [3]

Feature correlation

Correlation Matrix

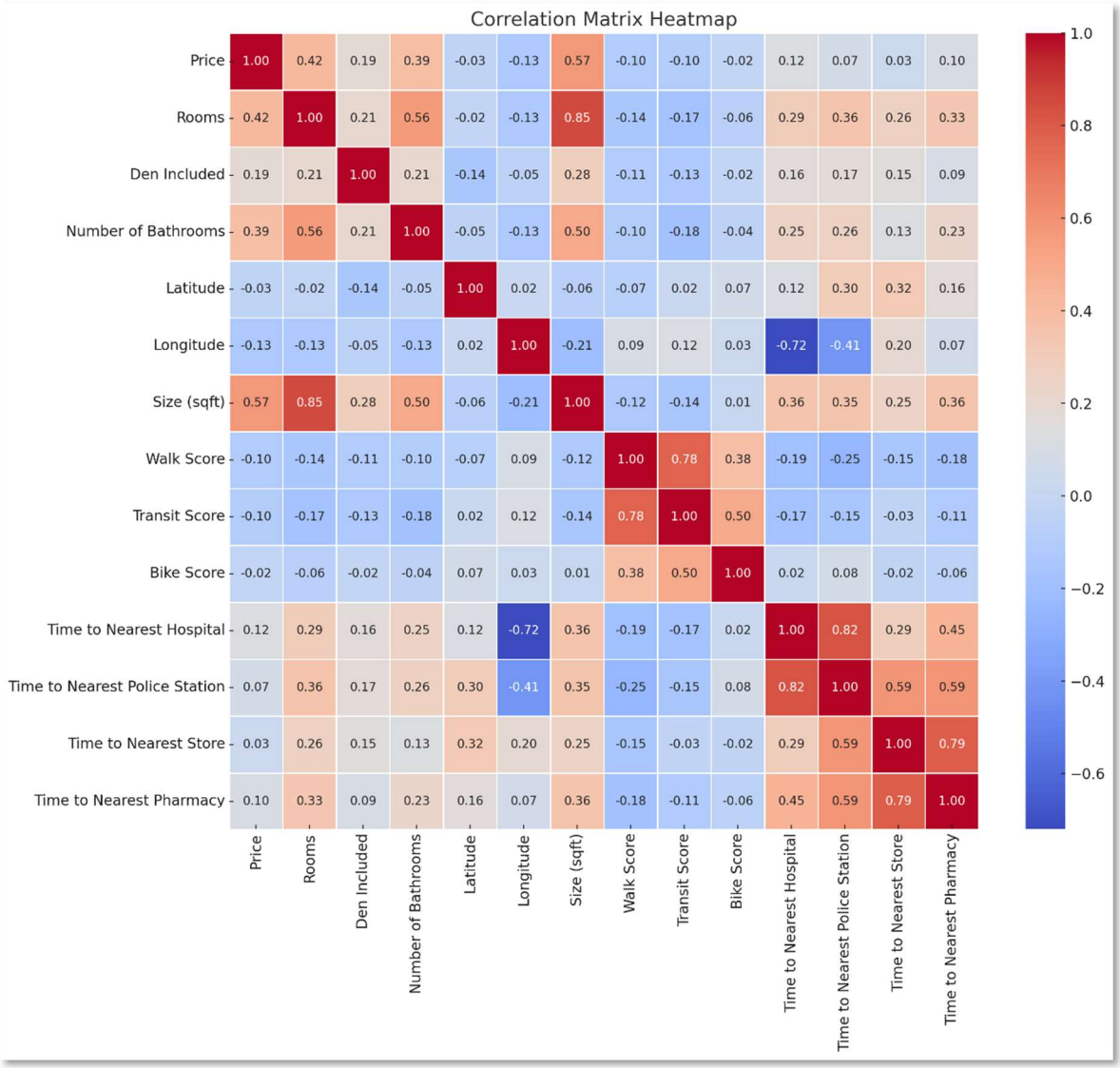

Fig 1. Correlation Matrix Heatmap for filtered data

The correlation matrix visualized in the heatmap provides a comprehensive overview of the linear relationships between various continuous variables in our dataset. Each cell in the matrix represents the correlation coefficient between two variables, ranging from -1 to 1. A positive value (towards 1) indicates a direct linear relationship, meaning as one variable increases, the other tends to increase as well. Conversely, a negative value (towards -1) signifies an inverse relationship, where an increase in one variable typically leads to a decrease in the other. Values close to 0 suggest a weak or no linear relationship between the pair of variables.

For example, a strong positive correlation between Price and Size (sqft) implies that larger properties tend to command higher rental prices. On the other hand, variables with little to no correlation suggest that changes in one do not reliably predict changes in the other. The correlation matrix is a foundational tool in statistical analysis and predictive modeling, helping to identify variables that are potentially useful predictors of the outcome (in this case, rental price) and to detect multicollinearity, where two or more predictors are highly correlated, which can impact the performance of some types of predictive models.

Chi Square Tests

The Chi-square test is a statistical method used to determine if there is a significant association between two categorical variables. It compares the observed frequencies in each category of a contingency table with the frequencies expected if there were no relationship between the categories. The formula for the Chi-square test statistic

$$\chi_c^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where *Oi* represents the observed frequency for category *i* and *Ei* represents the expected frequency for category *i*, assuming no association between the variables. The result of the Chi-square test is evaluated against a Chi-square distribution table, with the degrees of freedom determined by the dimensions of the table, to ascertain the significance of the association. A significant result suggests a likely association between the variables, whereas a non-significant result suggests no evidence of association.

The Chi-square test is fundamentally designed to assess the independence of two categorical variables or the goodness of fit between observed frequencies and expected frequencies in categorical distributions. It relies on counting occurrences within discrete categories and comparing these counts to what would be expected under the assumption of independence. Continuous variables, such as price, don't naturally fit into this framework because they can take on an infinite number of values within a range, lacking the discrete categories necessary for the Chi-square test's calculations. Applying the Chi-square test directly to a continuous variable like price would not make sense because it would involve comparing individual prices as if they were distinct categories, which fails to acknowledge the inherent continuity and the distribution of the

data. Thus, to use the Chi-square test with continuous data like price, one must first categorize or bin the data into discrete groups, transforming it into a form that aligns with the test's assumptions and applicability.

With 7 Buckets below are the Chi Square test results for the dataset:

Feature	**Chi-square Statistic**	**p-value**
Rooms	98.22	3.74e-23
Number of Bathrooms	15.90	6.67e-05
Latitude	37.42	1.46e-06
Longitude	20.59	0.0022
Size (sqft)	92.15	2.38e-18
Walk Score	9.75	0.045
Transit Score	6.86	0.144
Bike Score	1.65	0.80
Time to Nearest Hospital	18.50	0.0051
Time to Nearest Police Station	33.30	9.19e-06
Time to Nearest Store	21.63	0.0014
Time to Nearest Pharmacy	34.55	5.27e-06

By using 7 buckets for binning, we aimed to capture a finer granularity in the distribution of each continuous variable relative to Property Type. The results indicate varying degrees of statistical significance across the features, with most variables showing a statistically significant association with Property Type, as evidenced by p-values less than 0.05. Features like Rooms, Size (sqft), and Time to Nearest Police Station continue to show strong associations, whereas Bike Score exhibits a weak association with Property Type in this analysis. These findings offer insights into how different property characteristics and locational factors relate to the types of rental properties in the dataset, underscoring the complexity of the rental market's dynamics.

Extract, transform, and load (ETL)

The ETL (Extract, Transform, Load) process for this project was designed to gather, cleanse, and prepare data for training purposes in a machine learning context. Our ETL pipeline was composed of three main phases: data extraction via web scraping and API calls, data transformation using Python, and data loading into an MS SQL database. Below, we detail each step of this process.

Data Extraction

Data extraction was performed through a combination of web scraping and API calls. We utilized web scraping techniques to retrieve unstructured data from various online sources. For this purpose, we employed Selenium, a powerful tool that provides a high level of control over web browsers, allowing us to interact with JavaScript-heavy websites dynamically. Selenium enabled us to simulate human browsing behavior effectively, thereby accessing and extracting data that are not readily available through straightforward HTTP requests.

In addition to web scraping, we also used APIs to fetch structured data. APIs offer a more straightforward and reliable method of data extraction, providing direct access to structured data by querying the servers with specific parameters. The combination of web scraping and API calls ensured a comprehensive data collection process, capturing a wide array of information necessary for our analysis and model training.

Data Transformation

Once data was extracted, the next step involved cleaning and transforming this data into a usable format. This phase was crucial, as the raw data collected through scraping and APIs often contained inconsistencies, missing values, and errors. We used Python, renowned for its robust libraries and tools for data manipulation (such as Pandas and NumPy), to perform data cleaning tasks. These tasks included removing duplicates, handling missing values, standardizing data formats, and transforming data into suitable forms for analysis.

Python scripts were developed to automate the transformation process, ensuring that the data was normalized and consistent across various metrics. This step was vital for preparing the dataset for effective machine learning, as clean and well-prepared data significantly improves the accuracy and efficiency of the models.

Data Loading

The final step in the ETL process was loading the transformed data into an MS SQL database. This stage was designed to store the cleaned data in a structured and queryable format, facilitating easy access and retrieval for machine learning training and future analysis. The data loading was accomplished using a Python script, which established a connection to the MS SQL database and employed SQL commands to insert data into the database. [Script 21]

Exploratory Data Analysis:

Sample data:

Price	Property Type	Rooms	Den Included	Number of Bathrooms	Latitude	Longitude	Size (sqft)	Walk Score	Transit Score	Bike Score	Time to Nearest Hospital	Time to Nearest Police Station	Time to Nearest Store	Time to Nearest Pharmacy
1829	Apartment	1	0	1	44.6869	-63.58055	723	89	65	80	591	673	284	182
1612	Apartment	1	0	1	44.6364	-63.57163	634	89	65	80	246	164	92	102
1350	Apartment	1	0	1	44.661	-63.61568	634	89	62	50	472	552	167	236
2900	House	4	0	1.5	44.5953	-63.63582	2267	89	65	80	1046	989	383	370
3900	House	3	0	2	44.6317	-63.57793	2000	89	65	80	243	257	186	217
2800	House	2	0	1	44.6431	-63.59898	954	68	61	63	323	260	139	217
3100	House	4	0	3.5	44.6753	-63.65824	2267	68	50	40	1052	993	314	313
1275	House	2	0	1.5	44.645	-63.57466	1032	89	65	80	282	246	233	195
3000	House	3	0	1	44.6341	-63.57301	1800	99	73	72	251	185	113	178

Table 1: Sample of the Filtered Dataset

The sample data provides a snapshot of the features we're analyzing, including 'Price', 'Property Type', and various descriptors of the properties such as 'Rooms', 'Number of Bathrooms', and 'Size (sqft)'. We also consider location-based attributes like 'Latitude', 'Longitude', and 'Walk Score', alongside convenience metrics such as 'Time to Nearest Hospital' and 'Time to Nearest Store'. Through EDA, we aim to understand the distribution and relationships of these features, how they might influence the rental price, and any inherent trends or outliers present in the data. This process is crucial for guiding subsequent data preprocessing, feature selection, and the choice of modeling techniques to ensure our analysis is both rigorous and relevant to the dynamics of the rental market.

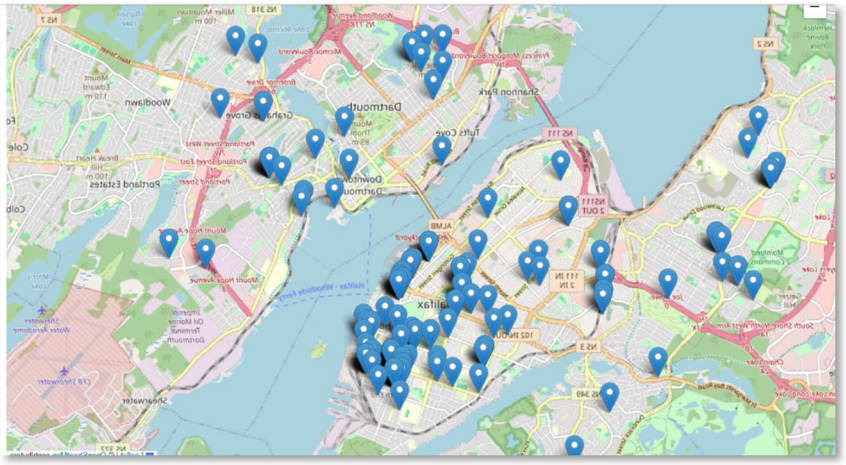

Fig 2: Geographical Distribution of few of the Rental Properties in Halifax

Scatter plot of prize vs Size:

The scatter plot reveals a pattern where, generally, as the size of rental properties increases, so does the rental price, highlighting a positive relationship between these two variables. The plot also shows a notable concentration of data in the mid-size range, suggesting this is where most rental properties fall. Outliers on the higher end of the price scale could represent premium properties or those in desirable locations. The variability in price for properties with similar sizes indicates other factors like location, amenities, or property features are also affecting rental prices. Overall, the plot underscores size as a key factor in rental pricing but suggests a more complex interplay of multiple factors influencing the final rental price.

Fig 3: Scatter plot of Price vs Size of property

Data Visualization in Tableau

Dashboard 1 – Rental Listing [15]

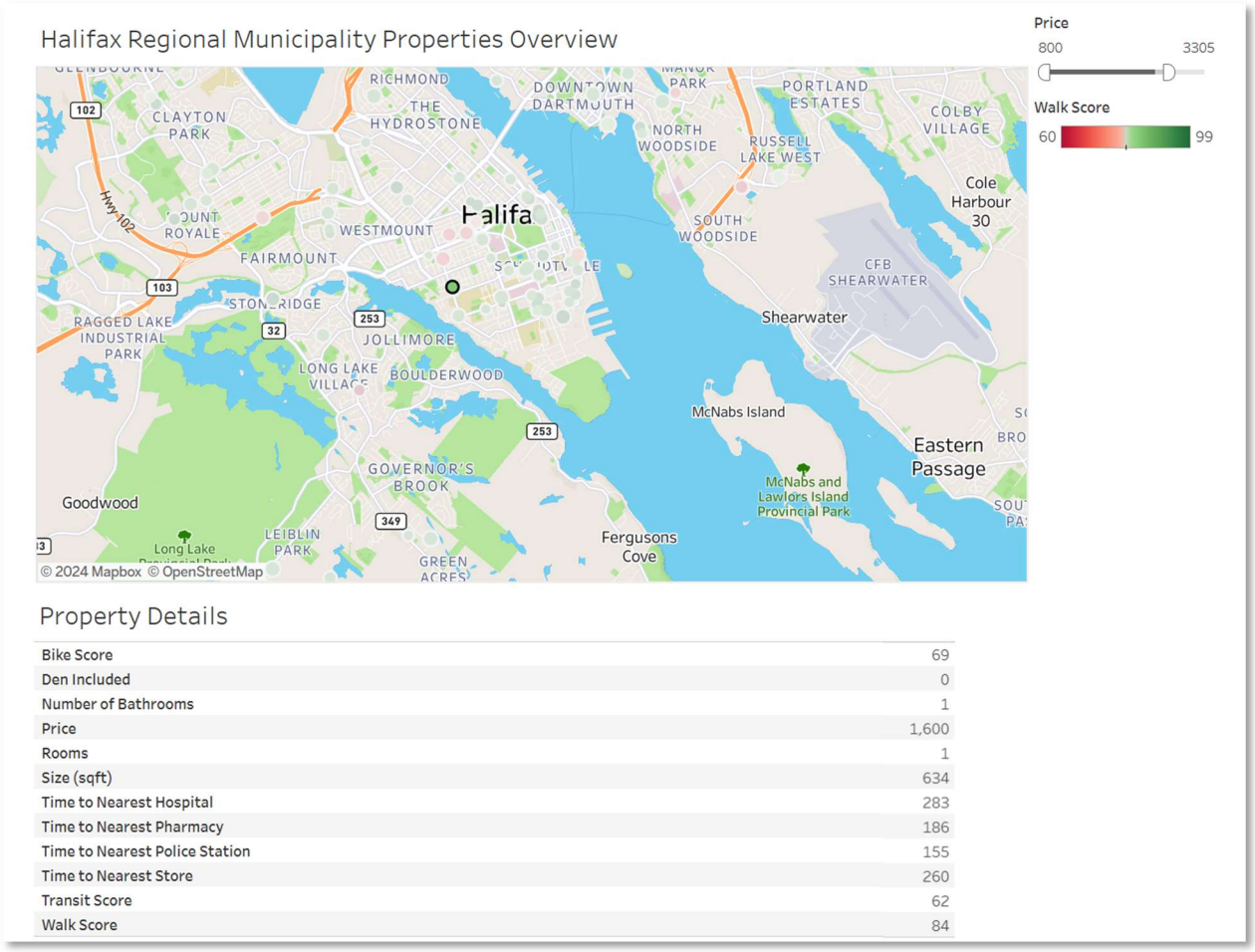

Fig 4: Tableau Dashboard for Rental Listings in Halifax

Key Features:

Map Overview: The primary feature of the dashboard is an interactive map that displays the geographical placement of rental properties. Users can interact with the map to zoom in and out for detailed or broadened geographical views.

Property Details upon Hover: Hovering over a property marker on the map reveals a summary of the property's details such as location coordinates, property type, and pricing information, providing an immediate snapshot without further interaction.

Selection Feature: Selecting a property on the map populates a detailed information table below, offering insights into the property's amenities, scores like Bike and Walk Scores, and proximity to essential services.

Price Filter Slider: A price filter slider is available to refine property listings according to budget constraints. The map view updates accordingly to reflect properties within the selected price range.

Walk Score Indicator: The dashboard features a Walk Score gradient, indicating the property's convenience for walking to nearby amenities, an important factor for many renters.

Aesthetic and Functionality: With its clear design and interactive components, the dashboard serves as a user-friendly platform for renters to identify and evaluate properties based on various preferences.

Dashboard 2 – Under Construction Properties [16]

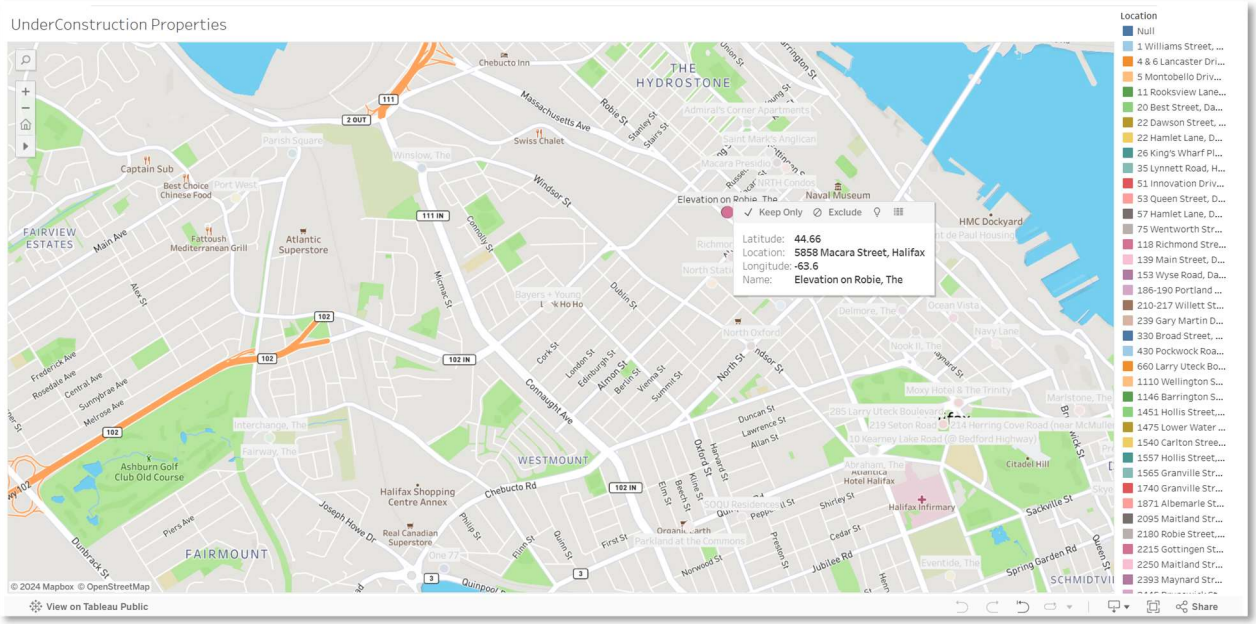

Fig 5: Tableau Dashboard for Under Construction Properties in Halifax

This dashboard presents a specialized view of under-construction properties, providing a valuable tool for prospective investors, realtors, or interested parties seeking information about new developments. The main feature is an interactive map highlighting the locations of properties currently under construction. Each property is marked on the map, and hovering over these marks reveals key details about the property, such as the latitude and longitude coordinates, along with the property's name and location address. For instance, when hovering over a particular property at "5858 Cunard Street, Halifax", the dashboard displays the name "Elevation on Robie", giving users an immediate sense of where the property is situated and what it's called. Additionally, the dashboard includes a sidebar listing all the addresses of the under-construction properties, allowing users to easily identify and select properties of interest directly from the list. The map's layout and functionality enable an intuitive navigation experience, catering to individuals looking for real-time updates on new property developments within a specific geographical area.

Dashboard 3 – Parking Rate [17]

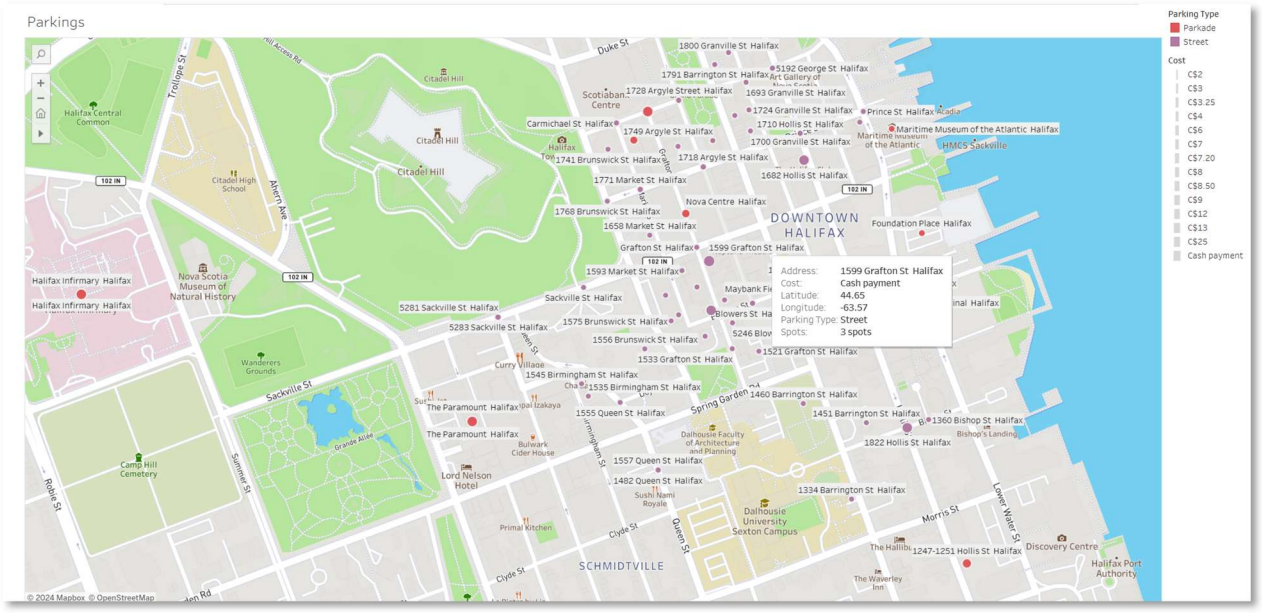

Fig 6: Tableau Dashboard for Parking rates in Halifax

This interactive Tableau dashboard provides an invaluable resource for individuals looking to find parking in the downtown Halifax area. It displays various parking locations, distinguishing between street parking and parking garages (parkades), which are clearly marked on the map. Each parking spot is represented by a pin, with the color of the pin corresponding to the cost of parking—ranging from lower-cost options, indicated in light

colors, to higher-cost options in darker colors, with a special designation for locations where only cash payment is accepted.

Hovering over a parking pin reveals detailed information about the specific location, such as the full address, cost, latitude, longitude, parking type, and the number of available spots. For example, when you hover over the parking location at "1599 Grafton St Halifax," the dashboard displays the parking as a cash payment option with a cost of 44.65, situated at a latitude of approximately 44.65 and longitude of -63.57, categorized as street parking with 3 spots available.

On the right side of the dashboard, there's a legend for Parking Type, color-coded to show the distinction between parkade and street parking, as well as a cost filter, which enables users to select the range of parking rates they are willing to pay. This feature helps users quickly identify the most cost-effective parking spots, tailor their parking choice to their budget, and find the most convenient location based on their destination in downtown Halifax.

Data Preprocessing [Script15] [Script16]

Data cleaning

In the process of refining our dataset for the purpose of predicting rental prices in Halifax, certain columns were selectively excluded from the final analysis. The reasoning behind the exclusion of these columns was primarily based on their perceived relevance and contribution to the accuracy and effectiveness of the predictive model. Here's a detailed rationale for dropping these columns:

URL: While providing a direct link to the listing, the URL is essentially a unique identifier for each property that does not carry intrinsic predictive value regarding the rental price. Its role is more pertinent to data referencing and verification rather than analysis. Due to the dynamic nature of data collection from various sources and the fluctuating availability of apartment listings, maintaining URLs as a consistent reference proved to be impractical.

Title & Description: These text-based fields contain qualitative information about the property, which, although rich in detail, require complex natural language processing (NLP) techniques to systematically extract features that could influence price prediction. Despite our efforts to leverage NLP to glean actionable insights, our models struggled to extract information that significantly enhanced price prediction accuracy.

Location: The precise location, often detailed as an address or neighborhood name, was replaced with more quantifiable geospatial metrics (Latitude and Longitude) to model the impact of location on rental prices systematically. This substitution allows for a more nuanced analysis of geographical factors without the need for text parsing or categorical handling of numerous unique locations.

Attributes: This broadly defined field likely encompasses a variety of property characteristics which, unless systematically categorized and quantified, pose a challenge for direct inclusion in a predictive model. The decision to exclude this field stems from the preference to focus on more directly quantifiable features that reliably contribute to model accuracy.

Parking Included & Utilities Included: While these features can influence rental prices, their impact may vary widely depending on the specific context of the Halifax rental market and the demographic priorities of renters. The exclusion of these variables was based on the lack of reliable data for all the datapoints.

Pet Friendly: Similar to parking and utilities, whether a property is pet-friendly can affect its desirability. However, the decision to exclude this variable was made under the assumption that its effect on rental prices could be indirectly captured by other variables or that it does not uniformly influence prices across all property types and locations.

Furnished: The inclusion of a property's furnished status was reconsidered due to the scarcity of data on this attribute within our dataset. The variability in the definition of "furnished" across listings further complicates its analysis, making it difficult to quantify its impact on rental prices consistently. Given these challenges and the limited availability of relevant data, the decision was made to exclude this variable to maintain the analytical model's focus and reliability.

Building Name: While the name of a building or complex might carry some brand value or reputation, incorporating this into the model would require a nuanced understanding of the local real estate market. The variability and specificity of building names, coupled with the potential for numerous unique entries, limit its straightforward applicability in a predictive model focused on broader market trends.

Distance from Hospital, Distance from Police Station, Distance to Nearest Store, Distance to Nearest Pharmacy: These distance measures were transformed into 'Time to Nearest' metrics to provide a more standardized and practical measure of accessibility and convenience from the property to key amenities and services. Time metrics are often more relevant to potential renters' decision-making processes than raw distance measures, offering a clearer indication of a location's convenience.

Handling Missing Values

In our data preprocessing efforts, we implemented a structured approach to address missing values across the dataset, ensuring its readiness for the analysis of rental prices. The strategies applied

were carefully chosen to maintain the dataset's completeness and enhance its reliability for predictive modeling. Below is a refined explanation of the methods used to manage missing values:

Handling Missing Values in "Den Included": We addressed missing entries in the Den Included column by assigning a value of 0. This decision was based on the presumption that the absence of explicit mention indicates the lack of a den feature in the property, thereby logically filling these gaps with a value that denotes absence.

Cleaning and Imputing Size:

Numeric Conversion and Initial Cleaning: The Size (sqft) data was first transformed into a numeric format, with non-numeric values set to NaN. This step ensured a uniform data type conducive to numerical analysis. Entries reflecting sizes less than 50 sqft were considered unrealistic and thus marked as missing by setting them to NaN.

Imputation with Mean Values: To fill the missing values in Size (sqft), we computed and applied the average size per room category across the dataset. This imputation strategy is underpinned by the reasonable assumption of a correlation between the size of a property and the number of rooms it contains, allowing for a contextually grounded approach to addressing missing size data.

Managing Missing "Number of Bathrooms" Data:

Standardization: NaN values within the Number of Bathrooms column were replaced with empty strings, standardizing the data for further processing.

Rule-Based Imputation: We introduced a heuristic for imputing missing bathroom data, where properties with one room were assumed to have one bathroom. For properties with more than one room, the number of bathrooms was estimated to be half the number of rooms. This method reflects common residential configurations, offering a pragmatic solution for estimating absent bathroom information.

Excluding Entries Without Essential Geospatial Data:

We removed records lacking valid Latitude and Longitude information, as these entries could not be accurately positioned for geographical analysis. Ensuring the presence of this geospatial data is critical for assessing the impact of location on rental prices, a key factor in our analysis.

Through these specific actions to handle missing values, our preprocessing script aimed to optimize the dataset for subsequent analysis phases, focusing on enhancing data accuracy and completeness. This foundational work supports the development of a robust model for predicting rental prices, minimizing the potential distortions caused by incomplete data.

Outlier Removal

This script processes a dataset to refine it further for analysis, specifically focusing on geographical data refinement, imputation of missing values for selected features using the K-Nearest Neighbors (KNN) method, and the removal of outliers.

Geographical Constraints

The dataset undergoes a filtering process based on predefined geographical bounds for latitude and longitude. This step ensures the analysis is concentrated on data relevant to a specific area, enhancing the focus and applicability of findings.

Missing Values Imputation

Selected features, including geographical coordinates and accessibility scores, are identified for imputation to address missing values. The K-Nearest Neighbors (KNN) algorithm is employed for this purpose, utilizing the proximity of data points in the feature space to estimate missing values. This method relies on the assumption that similar data points (in terms of location and accessibility) can provide a reasonable basis for filling in missing information, thereby preserving the dataset's overall integrity and consistency.

Outlier Detection and Removal

- A robust statistical method, the Modified Z-Score, is applied across numerical features to identify outliers. This technique calculates deviations from the median, normalized by the Median Absolute Deviation (MAD), offering resilience against outliers that could skew the analysis. By setting a threshold for outlier identification, the process selectively retains data points within a specified range of typical values, thereby minimizing the influence of extreme or anomalous observations on the dataset.
- The final step in the dataset refinement involves applying the outlier removal criteria to the entire dataset, ensuring that only data points considered to be within normal bounds are retained for analysis. This careful exclusion of outliers further contributes to the dataset's reliability, laying a solid foundation for accurate and meaningful insights.

This structured approach to data preparation—encompassing focused geographical selection, sophisticated imputation for missing values, and meticulous outlier removal—reflects a comprehensive strategy to optimize the dataset for detailed analysis and modeling. By addressing these key aspects of data quality, the methodology ensures that the resulting dataset is both accurate and representative, providing a strong basis for exploring the dynamics of rental prices or other phenomena of interest.

Standardization

Standardization is a data preprocessing technique that involves rescaling the features of a dataset so that they have a mean of 0 and a standard deviation of 1. This is achieved by subtracting the mean value of each feature from the data points and then dividing it by the standard deviation of each feature. Mathematically, it can be represented as:

$$z = \frac{x_i - \mu}{\sigma}$$

where x is the original value, μ is the mean of the feature, and σ is the standard deviation of the feature.

How Standardization Fits Our Dataset and Analysis:

1. **Algorithm Compatibility:** Many machine learning algorithms, especially those involving optimization algorithms, assume that all features are centered around zero and have similar variances. Standardization makes our dataset more compatible with these algorithms, potentially improving model performance.
2. **Feature Scale Unity:** In our dataset, if features are measured on different scales (e.g., square footage vs. number of bathrooms), standardization brings all variables to the same scale, mitigating the risk that features with larger magnitudes dominate those with smaller scales in the model.
3. **Outlier Robustness:** Although standardization does not reduce the impact of outliers, it transforms the dataset in a way that maintains outlier effects while ensuring they don't skew the entire dataset's scale. This property is particularly useful in datasets where outliers are significant but should not disproportionately influence the analysis.
4. **Improved Model Training Efficiency:** By ensuring that features have a mean of 0 and a standard deviation of 1, standardization can speed up the convergence of many machine learning algorithms, making the training process more efficient.

Why Standardization Over Normalization for Our Dataset:

Normalization rescales data into a fixed range, typically [0, 1], which is useful when we need to bound our values or when the algorithms we use are sensitive to the absolute sizes of the feature

values (e.g., neural networks). However, standardization is often more appropriate for our purposes due to several reasons:

Data Distribution: If the features of the dataset follow a normal distribution or when the machine learning algorithm assumes normally distributed data (e.g., Linear Regression, Logistic Regression), standardization aligns more closely with these assumptions than normalization.

Variance Preservation: Standardization preserves the variance of the original data (except for cases with outliers), which is crucial for algorithms that rely on variance to make decisions.

Given these considerations, standardization is chosen as a preprocessing step for our dataset to align with the analytical techniques employed and to optimize the dataset for the anticipated machine learning algorithms, thereby enhancing the robustness and interpretability of our predictive models.

Application of Standardization for our dataset:

Preparation and Splitting of Data

- The dataset is first loaded, and the features (X) are separated from the target variable (Price, y). This separation allows for the independent processing of input variables and the target for prediction.
- The dataset is then divided into training and testing sets to evaluate the model's performance on unseen data, ensuring a robust assessment of its predictive capabilities.

Standardization and One-Hot Encoding

- Within the preprocessing phase, specific continuous features are identified for standardization. These include variables such as Rooms, Number of Bathrooms, Size (sqft), and various scores and time-related features that could vary significantly in scale and distribution across the dataset.
- The “StandardScaler” from scikit-learn is applied to these selected features to rescale them to have a mean of 0 and a standard deviation of 1. This transformation mitigates the risk of certain features disproportionately influencing the model due to their scale, facilitating a more balanced and fair contribution from all features.
- Additionally, categorical features, in this case, Property Type, undergo one-hot encoding to convert them into a format suitable for modeling. This encoding expands the categorical variable into a series of binary features, each representing a category, to ensure that categorical data is appropriately handled in the analysis.

Methodology

The methodology section detailed the team's strategic approach to data collection, preprocessing, and model training. This included:

Data Collection: Employing diverse scraping techniques to extract data from various rental websites, ensuring a comprehensive understanding of Halifax's rental market.

Data Preprocessing: Rigorous cleaning, handling missing values, outlier removal, and standardization processes were applied to prepare the dataset for analysis. This step was critical for optimizing model performance.

Model Training and Evaluation: The team implemented and evaluated three machine learning models, using cross-validation scores to assess their predictive accuracy. The performance metrics considered were R-squared, Adjusted R-squared, Mean Squared Error (MSE), Mean Absolute Error (MAE), and, where applicable, Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE).

Model Training

Linear Regression [Script17]

This script embodies a structured approach to employing Linear Regression as a predictive model for rental prices, integrated within a cross-validation framework to assess model performance robustly. Key components of the script include data preprocessing, model pipeline construction, and performance evaluation through cross-validation.

Using Linear Regression within a cross-validation setup, our script processes and analyzes rental price data. Here's a simplified breakdown of its methodology and significance:

Preprocessing Steps

- The data is initially split into features (X) and the target variable (Price, y).
- Features are categorized as either numerical or categorical. Categorical features are transformed through one-hot encoding to convert them into a machine-readable format. Numerical features are standardized, ensuring they have a zero mean and unit variance, which helps in aligning scales across different features for the Linear Regression model.

Building the Model Pipeline

- A pipeline combining preprocessing steps with a Linear Regression model is set up. This ensures that all preprocessing, including scaling and encoding, is automatically applied when the model is trained or used for predictions.
- K-Fold cross-validation, specifically with 5 folds, is utilized to evaluate the model's performance across different segments of the data. This method systematically divides the data into training and testing sets, iterates through these partitions, and evaluates model performance, offering a comprehensive view of its predictive accuracy.

Performance Metrics and Evaluation

- For each fold in the cross-validation, the script calculates performance metrics: R-squared, Mean Squared Error (MSE), and Mean Absolute Error (MAE), providing insights into the accuracy and reliability of the model predictions.
- Adjusted R-squared is also computed to account for the number of predictors in the model, giving a more accurate measure of model performance by adjusting for the complexity of the model.
- Finally, the script averages these metrics across all folds to present an overall performance score. This aggregate view helps in understanding the model's generalizability and effectiveness in predicting rental prices.

Model Validation

Cross-validation scores:

Mean R2	**0.24279482857447193**
Mean Adjusted R2	**0.1543212347794339**
Mean MSE	**225555.82537425464**
Mean MAE	**314.0605892631509**

Table 2: Cross validation scores of linear regressions

Interpretation of Scores

Mean R2 (R-squared): 0.2428: This score indicates that, on average, the model explains about 24.28% of the variance in rental prices across the cross-validation folds. While this shows some level of predictive ability, a significant portion of the variance remains unexplained by the model, suggesting room for improvement.

Mean Adjusted R2: -0.1543: The negative adjusted R-squared value suggests that, when accounting for the number of predictors used in the model relative to the number of observations, the model performs worse than a simple, intercept-only model. This could be indicative of the model being overly complex given the data or not having the right combination of predictors to adequately capture the underlying relationships.

Mean MSE (Mean Squared Error): 225,555.83: The MSE quantifies the average squared difference between the actual and predicted rental prices. The relatively high value here points to substantial discrepancies in some predictions, which could be due to outliers, model misspecification, or other factors not captured by the model.

Mean MAE (Mean Absolute Error): 314.06: This score reflects the average absolute difference between predicted and actual rental prices. Similar to MSE, this value suggests that the model's predictions can deviate considerably from the true prices, though it offers a more intuitive sense of the average error magnitude.

Conclusion

The Linear Regression model, as it stands, offers a foundational look at predicting rental prices, capturing a modest portion of the variance in the dependent variable. However, the low R-squared and particularly the negative adjusted R-squared highlight significant limitations in its current form. These results suggest that the model, while a useful starting point, requires refinement and possibly the incorporation of more relevant features, interaction terms, or the exploration of non-linear models to better capture the complexities of rental price determination.

The relatively high MSE and MAE further reinforce the need for model improvement or the consideration of alternative, more complex modeling approaches that can account for the nuances and variability inherent in rental price data. The goal moving forward would be to identify and

incorporate additional predictors, consider the potential for non-linear relationships, and possibly leverage more sophisticated modeling techniques to enhance predictive performance and achieve a more accurate and reliable model for estimating rental prices.

Random Forest [Script18]

Random Forest is a method commonly used for classification and regression tasks. It operates by constructing multiple decision trees during the training phase and outputting the mode of the classes (classification) or mean/average prediction (regression) of the individual trees. Random Forest is known for its robustness and versatility, capable of handling both numerical and categorical data.

Relevance and Advantages Over Linear Regression

In the context of predicting rental prices, a Random Forest model may offer several advantages over Linear Regression:

- **Complex Relationship Modeling**: Rental prices can be influenced by a complex interplay of features, including location, property size, and nearby amenities. Random Forest's ability to model non-linear relationships and interactions between features makes it potentially more effective in capturing these dynamics.
- **Feature Interactions**: Unlike Linear Regression, which requires explicit creation of interaction terms to model relationships between features, Random Forest can automatically consider and utilize interactions between features through its tree-based structure.
- **Reduced Overfitting**: The ensemble nature of Random Forest, particularly its use of bagging (Bootstrap Aggregating) and feature randomness, helps in reducing overfitting, making the model more generalizable to unseen data.
- **Improved Predictive Performance**: Given its flexibility and the ensemble approach, Random Forest may achieve higher predictive accuracy than Linear Regression, especially in cases where the relationship between features and the target variable is complex and non-linear.
- **Insights into Feature Importance**: Random Forest provides a straightforward method for evaluating the importance of each feature in the prediction, offering valuable insights that can guide further model refinement and understanding of the factors influencing rental prices.

Data Preparation and Feature Selection

- The dataset is loaded, and a distinction is made between the feature matrix X (independent variables) and the target vector y (rental prices), setting the stage for supervised learning.
- A split is performed to divide the data into training and testing sets, ensuring a robust evaluation framework where the model's performance is assessed on unseen data.

Preprocessing Pipeline

Standardization and One-Hot Encoding: Key to our preprocessing strategy is the dual approach of standardizing numerical features and one-hot encoding categorical features. Standardization ensures that numerical columns have a mean of 0 and a standard deviation of 1, addressing scale discrepancies among features. One-hot encoding transforms categorical variables into a binary matrix, making them suitable for regression analysis. This preprocessing is encapsulated within a 'ColumnTransformer', allowing for seamless integration into the model pipeline.

Random Forest Regressor Pipeline

- **Model Selection**: The Random Forest Regressor is chosen for its ability to model complex, non-linear relationships and its robustness to overfitting, courtesy of its ensemble approach. Configured with 100 trees (n_estimators=100) and a consistent random state for reproducibility, it forms the core of our predictive model.
- **Pipeline Integration**: The preprocessor and the regressor are combined into a single pipeline. This integration not only streamlines the modeling process but also ensures that preprocessing steps are appropriately applied during both training and prediction phases.

Model Evaluation

- **Cross-Validation**: To assess the model's performance and generalizability, we employ cross-validation with five folds. This method splits the training data into five subsets, using each in turn for validation while training on the remaining four. The primary metric for evaluation is the negative Mean Squared Error (MSE), with scores averaged across all folds to provide a comprehensive measure of model accuracy.
- **Training and Prediction**: Following cross-validation, the model pipeline is fitted to the entire training dataset, and predictions are made on the test set to evaluate the model's performance in a real-world scenario.

Model Validation

Cross-validation scores:

Mean R2	**0.7952931972442445**
Mean Adjusted R2	**0.6997633559582253**
Mean MSE	**48336.01885871195**
Mean MAE	**152.7890186017686**
Root Mean Squared Error (RMSE)	**219.8545402276513**
Mean Absolute Percentage Error (MAPE)	**7.1089265469526 %**

Table 3: Cross validation scores of Random Forest

Interpretation of Scores

Mean Absolute Error (MAE): 152.79: On average, the model's predictions are approximately 152.79 currency units away from the actual rental prices. This value gives a direct, intuitive sense of the average error magnitude in the model's predictions, indicating relatively close predictions to the actual values for most of the dataset.

Mean Squared Error (MSE): 48,336.02: The MSE, which averages the squares of prediction errors, is significantly higher than the MAE due to its penalization of larger errors. An MSE of 48,336.02 suggests that there are instances where the model's predictions deviate considerably from the actual values, though this metric alone does not specify how widespread such instances are.

Root Mean Squared Error (RMSE): 219.85: The RMSE, being the square root of the MSE, reduces the scale of the error to be comparable to the original data, indicating an average prediction error of 219.85 currency units. This metric is particularly useful for understanding the magnitude of errors in the same units as the target variable.

R-squared (R2): 0.7953: The R-squared value indicates that approximately 79.53% of the variance in rental prices is explained by the model. This high value suggests that the model captures a substantial portion of the variability in rental prices, indicating strong predictive power.

Adjusted R-squared: 0.6998: Adjusted R-squared takes into account the number of predictors in the model, offering a more accurate measure of predictive performance when multiple features are involved. A value of 0.6998 suggests that, after adjusting for the number of features, the model still explains a considerable amount of variance in rental prices, although less than indicated by the unadjusted R-squared.

Mean Absolute Percentage Error (MAPE): 7.11%: The MAPE provides a relative measure of the average error magnitude, indicating that the model's predictions are, on average, 7.11% away from the actual rental prices. This percentage offers a sense of accuracy in terms that are easily understandable and directly comparable across different models or datasets.

Feature Importance

Feature importance in a Random Forest model is a measure used to interpret the contribution of each feature to the predictive power of the model. It is calculated during the training process of the forest.

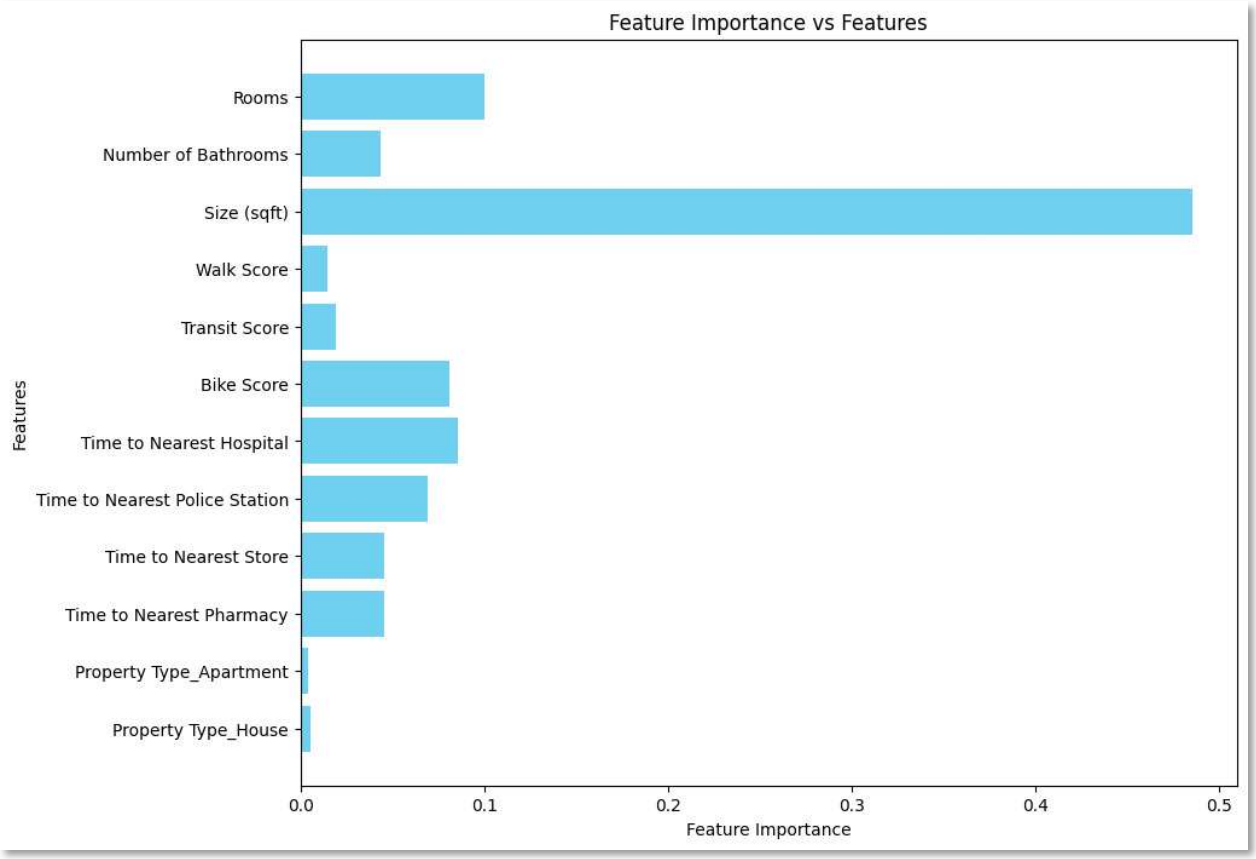

Fig 7: Feature Importance vs Feature plot of random forest

Interpretation of the Graph:

Size (sqft): This feature appears to be the most important predictor of rental prices, which is intuitive as the size of a property is typically a key determinant of its value.

Rooms & Number of Bathrooms: These features also show significant importance, suggesting that the number of habitable spaces contributes considerably to rental pricing decisions.

Location-Related Scores: Walk Score, Transit Score, and Bike Score, which indicate the property's accessibility and convenience, have moderate importance in predicting rental prices. This aligns with the understanding that accessibility to services and amenities is a valued aspect of real estate.

Proximity to Amenities: Time to the nearest hospital, police station, store, and pharmacy have varying degrees of importance, with time to the nearest store being more influential than the others. This might reflect the priority tenants place on everyday convenience.

Property Type: The feature importance indicates that the type of property, specifically whether it's an apartment or a house, holds relatively less importance compared to size and room count. However, it still provides some value in the model's predictions.

Conclusion

The model demonstrates strong predictive capability, as evidenced by the high R-squared value and relatively low error metrics (MAE, MSE, RMSE, and MAPE). The differences between R-squared and Adjusted R-squared highlight the importance of considering model complexity, but both values affirm the model's effectiveness.

The error metrics, particularly the MAE and RMSE, suggest that while the model's predictions are generally close to the actual values, there are instances of significant prediction errors that could be targeted for improvement in future iterations.

In conclusion, the Random Forest Regressor model shows promising results in predicting rental prices, with strong explanatory power and reasonable accuracy. The detailed performance evaluation suggests areas for further refinement, potentially through feature engineering, hyperparameter tuning, or exploring additional modeling techniques, to enhance prediction accuracy and reduce error margins.

XGBoost [Script19]

XGBoost, short for Extreme Gradient Boosting, is an advanced implementation of gradient boosting that is highly efficient, flexible, and portable. It is a powerful machine learning algorithm that uses decision tree ensembles, which means it builds the model in stages, and each stage corrects the errors of the previous stage, making the model strong and robust.

Key Features of XGBoost:

Gradient Boosting Framework: XGBoost applies the principle of boosting weak learners (decision trees) in a sequential manner to form a strong predictive model.

Speed and Performance: It is designed to be highly efficient, scalable, and fast, which makes it applicable to large datasets and complex problems.

Regularization: Includes built-in L1 (LASSO regression) and L2 (Ridge regression) regularization which prevents overfitting and improves model performance.

Handling Sparse Data: XGBoost can handle sparse data, such as missing values or zero entries, without extensive preprocessing.

Cross-Validation: Integrated cross-validation process that allows assessing the performance of each stage of training and improving the model accordingly.

How XGBoost Fits the Use Case

For predicting rental prices, XGBoost can capture the complex nonlinear relationships that exist in real estate data. It's particularly suitable when the dataset has numerous features influencing the price, such as property size, location scores, and proximity to amenities. XGBoost can also handle various data types, making it ideal for datasets with a mix of categorical and numerical features.

- **Complex Feature Interactions**: XGBoost can automatically learn complex feature interactions that are difficult to model with linear methods.
- **Robustness to Overfitting**: Regularization in XGBoost helps to reduce overfitting, which can be a risk in high-dimensional data.
- **Feature Importance**: It provides feature importance scores, which can be insightful for understanding the driving factors behind rental prices and guiding future data collection and feature engineering.

Model Overview:

Data Preprocessing: The script defines numeric and categorical features, standardizing the former and one-hot encoding the latter to prepare them for modeling.

Model Pipeline: A pipeline is created that includes the preprocessing steps and the XGBoost regressor, streamlining the process from data transformation to model fitting.

Cross-Validation: It employs KFold cross-validation to evaluate the model's performance in a robust manner, ensuring that the results are not dependent on a particular data split.

Performance Metrics: The script assesses the model's success using R-squared, MSE, and MAE, and calculates an adjusted R-squared to consider the number of predictors used.

Feature Importances: After training, the model's feature importances are extracted, offering insights into which features most strongly influence rental price predictions.

Cross-validation scores:

Mean R2	**0.4889666974540369**
Mean Adjusted R2	**0.2464807892723523**
Mean MSE	**135971.57309880885**
Mean MAE	**222.16674597537877**

Table 4: Cross Validation Score for XGBOOST

Interpretations:

Mean R^2 (R-squared): 0.489

The R-squared value indicates the proportion of the variance in the dependent variable that is predictable from the independent variable(s). Your mean R^2 of approximately 0.489 suggests that around 48.9% of the variance in your target variable can be explained by your model. This is a moderate level of explanatory power, indicating your model captures nearly half of the variability in the data, but there's still a significant portion of variability that it doesn't account for.

Mean Adjusted R^2: 0.246:

The Adjusted R-squared value adjusts the R-squared value based on the number of predictors in the model relative to the number of observations. This is crucial for models with multiple predictors, as it penalizes the addition of irrelevant predictors. Your mean Adjusted R^2 of approximately 0.246 is significantly lower than the mean R^2, which could indicate that your model may have too many predictors or some of the predictors are not significantly contributing to the model's explanatory power. This discrepancy suggests a need to possibly refine the model by selecting more relevant features.

Mean MSE (Mean Squared Error): 135971.573

The Mean Squared Error measures the average of the squares of the errors, essentially quantifying the difference between the predicted and actual values. A lower MSE indicates a better fit of the model to the data. Your MSE is context-dependent; its interpretation relies on the scale of your target variable and the context of the problem. In isolation, 135971.573 might seem high or low, depending on the actual values you are predicting. Comparatively, it’s useful for model selection, with a preference for models with lower MSE values.

Mean MAE (Mean Absolute Error): 222.167

The Mean Absolute Error represents the average of the absolute differences between the predicted and actual values, providing a linear score that summarizes the magnitude of errors in prediction. Like MSE, a lower MAE indicates a better fit. An MAE of 222.167 means that, on average, the predictions of your model are about 222.167 units away from the actual values. The interpretation of the MAE also depends on the context and scale of your target variable.

In summary, your XGBoost model demonstrates a moderate level of predictive accuracy, explaining around 48.9% of the variance in the dependent variable. However, the significant

difference between the R-squared and Adjusted R-squared values suggests that the model might benefit from feature selection or dimensionality reduction. Additionally, the MSE and MAE provide a baseline for the average error in predictions, which can guide further model refinement and comparison. Consider these metrics in the context of your specific application to determine the model's suitability and areas for improvement.

Model Selection

When comparing the cross-validation scores of Linear Regression, Random Forest, and XGBoost models, we consider several key metrics: Mean R^2, Mean Adjusted R^2, Mean MSE (Mean Squared Error), Mean MAE (Mean Absolute Error), and where provided, RMSE (Root Mean Squared Error) and MAPE (Mean Absolute Percentage Error). Each of these metrics offers insight into different aspects of model performance, such as the proportion of variance explained (R^2), model fit (MSE, RMSE), prediction error (MAE), and the model's performance relative to the scale of the data (MAPE).

Linear Regression:

Mean R^2: 0.243, Mean Adjusted R^2: 0.154, Mean MSE: 225556, Mean MAE: 314.061

This model shows the lowest R^2 values, indicating a weaker ability to explain the variance in the dependent variable. The higher MSE and MAE suggest that it also has a higher average prediction error compared to the other models.

Random Forest:

Mean R^2: 0.795, Mean Adjusted R^2: 0.700, Mean MSE: 48336, Mean MAE: 152.789, RMSE: 219.855, MAPE: 7.109%

XGBoost outperforms both Linear Regression and Random Forest across all metrics. It has the highest R^2 and Adjusted R^2 values, indicating it explains a substantial portion of the variance. It also has the lowest values for MSE, MAE, RMSE, and MAPE, suggesting it makes more accurate predictions with less error.

XGBoost (FineTuned):

Mean R^2: 0.81, Mean Adjusted R^2: 0.72, Mean MSE: 49115, Mean MAE: 146.14

Random Forest performs better than Linear Regression but is outperformed by XGBoost. It has moderate R^2 and Adjusted R^2 values and falls in the middle in terms of MSE and MAE.

Conclusion and Recommendation:

The XGBoost model is the best among the three for our case, based on the cross-validation scores provided. Its high R^2 and Adjusted R^2 values indicate a strong ability to explain the variance in the target variable. Additionally, its low MSE, MAE, RMSE, and MAPE values suggest it makes accurate predictions with lower error rates. This model balances explanatory power and prediction accuracy, making it the most suitable choice for scenarios where both aspects are critical. The superiority of the Random Forest model in handling complex nonlinear relationships and

interactions between variables, without the need for extensive feature engineering as might be the case with Linear Regression, likely contributes to its stronger performance.

Limitations of RF Implementation

The RF model has demonstrated robustness in capturing non-linear relationships and interactions between features. However, its ensemble nature that combines multiple decision trees can lead to increased computational complexity, especially with large datasets or a high number of trees. This complexity can result in longer training times and difficulty in real-time prediction scenarios. Additionally, RF models can be less interpretable compared to simpler models like linear regression because understanding the decision path across numerous trees can be challenging. The model's predictive power is contingent on the quality of the data and the relevance of the features selected. Overfitting, though mitigated by the ensemble approach, can still occur if the hyperparameters are not properly tuned to balance bias-variance trade-off

XG BOOST Parameter Tuning [Script20]

In the parameter tuning section for your XGBoost model, we've identified a set of hyperparameters that optimize model performance. Each of these hyperparameters plays a crucial role in controlling the model's behavior and capability to learn from the data efficiently. Here is a detailed explanation of the optimized hyperparameters:

n_estimators: 60

This parameter specifies the number of trees (or rounds) in the ensemble. A total of 60 trees are used in your model, striking a balance between learning complexity and computational efficiency. Too many trees can lead to overfitting, while too few might not capture the underlying patterns well enough.

max_depth: 7

The maximum depth of a tree controls the maximum number of levels in each decision tree. A depth of 7 allows the model to learn complex patterns by making more splits but is restrained enough to prevent the model from becoming overly complex and overfitting.

learning_rate: 0.12453744969333737

Also known as the "eta" value, this parameter scales the contribution of each tree to the final prediction. A learning rate of approximately 0.125 ensures that the model learns at a moderate pace, allowing for more robust generalization by preventing rapid overfitting to the training data.

min_child_weight: 1

This parameter sets the minimum sum of instance weight (hessian) needed in a child. With a setting of 1, it ensures that the model is sensitive to splitting at nodes that capture a minimal number of observations, which can be important for learning fine-grained patterns without causing too much fragmentation.

subsample: 0.7981473804836309

Subsample denotes the fraction of the training data sampled for growing trees. A subsample rate of roughly 0.8 means that each tree is built on 80% of the data, which helps in preventing overfitting by introducing more variability into the model training process.

colsample_bytree: 0.8664132457059034

This parameter sets the fraction of features (columns) to be randomly sampled for each tree. At approximately 0.866, it ensures that each tree in the ensemble considers about 86% of the available features when making splits. This diversity among trees helps enhance model robustness and prevents overfitting to specific parts of the data.

Performance Metrics:

RMSE (Root Mean Squared Error): 211.62267975341152 - This metric shows the standard deviation of the prediction errors, indicating that on average, the model's predictions are within 211.623 units of the actual values. The lower RMSE reflects the model's accuracy.

MAE (Mean Absolute Error): 146.14096408420139 - This represents the average absolute difference between the predicted values and the actual values, offering a clear measure of prediction accuracy without considering direction. An MAE of 146.141 suggests predictions are relatively close to actual values.

R^2: 0.8103356020062977 - This indicates that approximately 81% of the variance in the target variable is explained by the model, highlighting a strong predictive power.

Adjusted R^2: 0.7218255496092367 - Adjusted for the number of predictors in the model, this value suggests that after accounting for the number of features, the model still explains a significant portion of the variance, confirming its effectiveness.

In summary, these tuned hyperparameters collectively enhance the XGBoost model's ability to capture the underlying patterns in the data effectively, ensuring a balance between bias and variance, and resulting in high predictive performance.

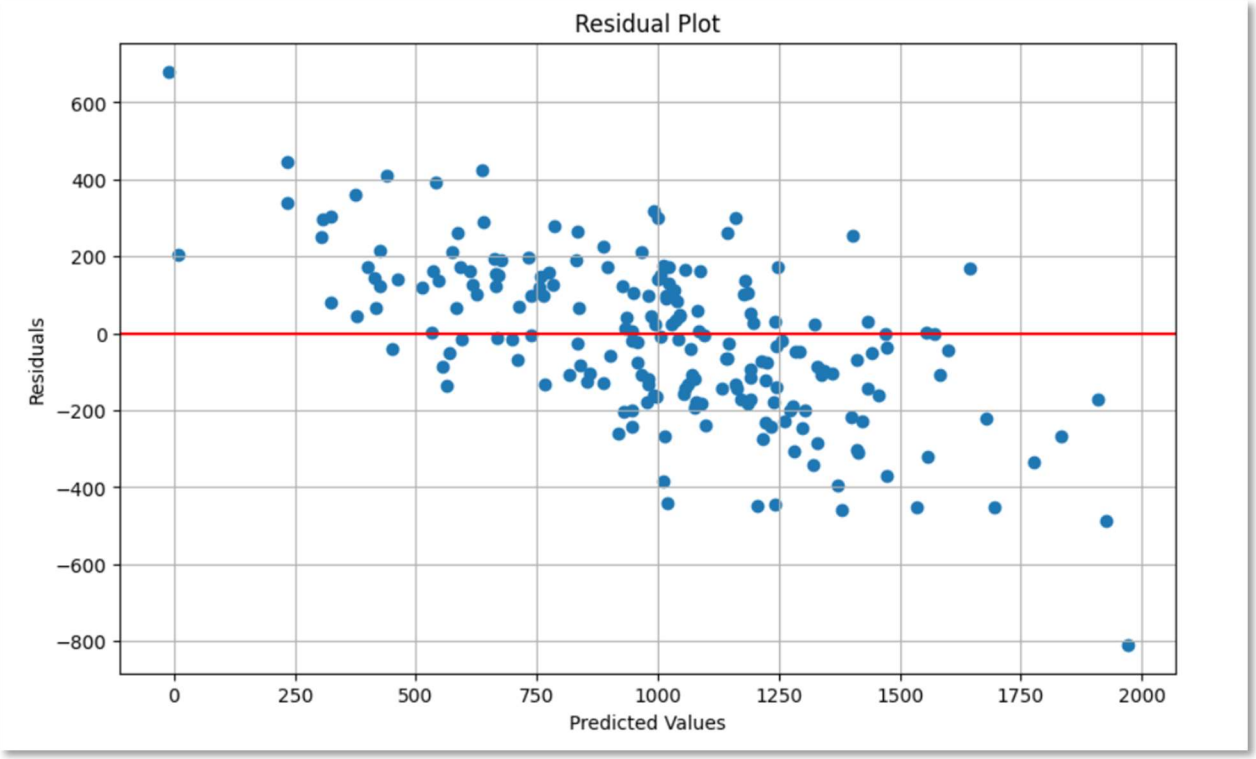

Fig 8 : Residual Plot for Optimized XGBoost

The residual plot for the XGBoost model provides several positive indicators of the model's performance:

Randomness Around Zero: A significant number of residuals are distributed around the zero line, indicating that the model predictions are often close to the actual values. This demonstrates that the model has predictive capabilities and is generally reliable.

No Clear Bias: The residuals do not show a clear bias towards underestimation or overestimation since they scatter fairly evenly above and below the zero line. This suggests that the model, on average, is correctly capturing the central tendency of the target variable.

Symmetry in Distribution: The residuals seem to be symmetrically distributed, which is a good sign in regression analysis. It indicates that the model does not persistently predict too high or too low across the range of predictions.

Consistent Variance at Lower Predicted Values: For lower predicted values, the variance of residuals appears consistent, implying that the model performs well for a significant portion of the data, especially in the lower range of predicted values.

No Systematic Patterns for Majority of Data: For the main body of the data (ignoring potential outliers), there are no strong systematic patterns that suggest the model is failing to capture nonlinear trends or interactions between features for most of the data points.

Future Works

Future work for enhancing the analytical capabilities of our rental market analysis system in Halifax could take several promising directions. Each of these avenues not only aims to refine the precision of the predictive models but also to enrich the insights that can be gleaned from the data:

- Dynamic Web Scraping Integration: Implementing dynamic web scraping capabilities through frameworks like LangChain can provide real-time updates to our dashboards. By automating the data extraction process, we ensure that our models and visualizations reflect the most current market trends. This approach will help in capturing the rapid changes in the rental landscape, adjusting to seasonal fluctuations, and responding to market disruptions in a timely manner.
- Incorporation of Macroeconomic Indicators: Including macroeconomic indicators such as inflation rates, employment statistics, and housing market trends could significantly improve the model's ability to predict rental prices. These indicators offer context that can influence rental prices beyond the property features, such as a change in purchasing power or economic downturns.
- Immigration and Student Population Inflows: The influx of students and immigrants has a direct impact on the demand for rental properties. By incorporating immigration statistics and university enrollment data, we can better forecast rental market demand, particularly in areas close to educational institutions or communities with high immigrant populations.
- Time Series Data Collection: Utilizing web archives to collect historical rental listings could allow for the creation of a robust time series dataset. This data would enable the use of sophisticated forecasting models that take into account historical trends and cyclical patterns in rental prices.
- Local Government and Policy Changes Analysis: Local government policies, zoning laws, and housing regulations can have significant impacts on rental prices. Future models could integrate data on policy changes, such as new housing subsidies or rent control measures, to anticipate their effects on the rental market.
- Enhanced Geographic Data Utilization: The development of a more granular geographic analysis, considering neighborhood-specific characteristics, public transport accessibility, and community services, could provide deeper insights into the desirability of rental properties. Geo-spatial analysis could be used to identify emerging hotspots and declining areas within the city.
- Consumer Sentiment Analysis: Harnessing the power of natural language processing to analyze consumer sentiment on social media and rental review platforms could provide an additional layer of predictive power. Sentiment analysis could capture the intangible factors that influence rental desirability and price, such as the perceived quality of life or community vibrancy.

- Advanced Feature Engineering Using AI: Implementing AI-driven feature engineering to uncover additional predictive variables and interactions that are not apparent through manual analysis. Machine learning algorithms could be trained to identify complex patterns and propose new features that could refine the predictive models further.

Benefits of the System

The envisioned dashboard system stands to offer substantial benefits to various stakeholders in the Halifax real estate market. As a centralized hub, it serves as a confluence point for developers, buyers, and the general populace, providing valuable insights and predictions on rent prices. Here's a detailed exposition of the system's advantages:

For Developers and Investors

- Market Pulse Monitoring: The system offers developers a pulse on the current market conditions, enabling them to make data-driven decisions on where and when to initiate new construction projects.
- Investment Strategy Formulation: By analyzing real-time data, investors can craft nuanced investment strategies that align with market trends, demographic shifts, and economic forecasts, maximizing their return on investment.
- Risk Mitigation: With access to up-to-date rental estimates and market analytics, developers and investors can identify potential risks and adjust their portfolios to mitigate exposure.

For Prospective Buyers and Tenants

- Informed Decision-Making: The dashboard empowers buyers and renters with information that helps them make informed decisions about the affordability and suitability of properties.
- Price Negotiation: By providing a benchmark of expected rent prices, the system equips tenants with data to negotiate fair lease terms.
- Neighborhood Selection: The integration of geographic data aids in identifying desirable neighborhoods based on amenities, community characteristics, and accessibility, aligning living arrangements with lifestyle preferences.

For Real Estate Professionals

- Client Service Enhancement: Real estate agents can utilize the dashboard to provide clients with enriched service offerings, including accurate price estimations and market analyses.

- Listing Optimization: The system enables agents to optimize property listings by adjusting prices in line with market predictions, enhancing the attractiveness of their properties.
- Trend Analysis: Agents can stay ahead of the curve by analyzing trends and patterns within the dashboard, offering them a competitive edge in the market.

For Municipal Planning and Policy-Making

- Urban Planning Support: Local government officials and urban planners can use the system to understand housing trends, aiding in the development of more effective housing policies and urban development plans.
- Impact Assessment of Regulations: The dashboard provides a means to assess the impact of housing regulations and policies, enabling continuous refinement and improvement of governance in the housing sector.

For the General Public

- Accessibility of Information: By democratizing access to rental market data, the system contributes to transparency, allowing the public to stay informed about market conditions without needing specialized knowledge.
- Community Engagement: The system could potentially incorporate community-sourced reviews and feedback, fostering a platform for communal insights and experiences.

Future Development and Scalability

- Estimation of Future Rent Prices: The application is positioned to evolve into a predictive tool for future rent prices, helping users anticipate market changes.
- Expansion Capabilities: While initially focused on Halifax, the system has the potential to scale and integrate data from additional regions, potentially offering a nationwide analysis of rental markets.

Conclusion

The comprehensive analysis and model evaluation led to the conclusion that the XGBOOST model was the most suitable for predicting rental prices in Halifax. Its success was attributed to its ability to capture the nuanced relationships between various predictors and rental prices, delivering accurate and reliable predictions. The project highlighted the potential of data mining and machine learning techniques in transforming real estate market analyses, offering stakeholders valuable insights to make informed decisions. The findings also underscore the importance of continuous model refinement and exploration of advanced algorithms to further enhance predictive accuracy.

References

.[1] Cycling & Walking | Downtown Halifax Business Commission

[2] Halifax Regional Municipality NS - Walk Score

[3] Distance Matrix API overview | Google for Developers

[4] **Southwest Apartments**: Apartments Halifax | Southwest Properties

[5] **Killiam Apartments:** Home | killam (killamreit.com)

[6] **Templeton Apartments:** Nova Scotia and Halifax Apartments for Rent | Templeton Properties

[7] **Kijiji:** Kijiji in Halifax. - Buy, Sell & Save with Canada's #1 Local Classifieds.

[8] **Capreit Apartments:** Rental Apartments in Canada - Canadian Apartment Properties REIT (capreit.ca)

[9] **Zumper:** Zumper - Houses, Condos, and Apartments for Rent

[10] **444Rent:** 444Rent.com | Halifax Apartments for Rent | Dartmouth Apartments for Rent

[11] **Apartments.com:** Apartments.com Canada - Find apartments and homes for rent in Canada.

[12] **West22 Living:** West22 | West22 (west22living.com)

[13] **RentSeeker:** Apartments for Rent in Canada - Your Trusted Apartment Finder - RentSeeker.ca

[14] **ChatGPT**: For improving quality of writing and grammar check.

[15] **Rental dashboard**: Property listings

[16] **Parking dashboard** : Parking listings

[17] **Under Construction properties dashboard** : Under Construction properties

Appendix

Data Scraping

- Kijiji
    - Script1: Scrape Listings

```
import requests
from bs4 import BeautifulSoup
import re
import csv

def get_total_listings(page_url):
with requests.Session() as session:
response = session.get(page_url)
if response.status_code == 200:
soup = BeautifulSoup(response.content, 'lxml')
results_text = soup.find("h2", {"data-testid": "srp-results"}).text
numbers = re.findall(r'\d+', results_text)
if numbers:
total_listings = int(numbers[-1])
return total_listings
else:
return 0
else:
return 0

def get_listings(page_url):
with requests.Session() as session:
response = session.get(page_url)
if response.status_code == 200:
soup = BeautifulSoup(response.content, 'lxml')
listings = soup.find_all("li", attrs={"data-testid": lambda value: value and value.startswith('listing-card-list-item-')})
urls = []
for listing in listings:
link = listing.find("a", {"data-testid": "listing-link"})
if link and 'href' in link.attrs:
full_url = "https://www.kijiji.ca" + link['href'] if not link['href'].startswith("http") else link['href']
urls.append(full_url)
return list(set(urls))
else:
return None

def process_base_urls(base_urls):
# Open a CSV file to write the URLs
with open('listings.csv', 'w', newline='', encoding='utf-8') as file:
```

```
writer = csv.writer(file)
writer.writerow(['Listing URLs']) # Write a header row, if desired

for base_url, search_params in base_urls:
first_page_url = f"{base_url}{search_params}"
total_listings = get_total_listings(first_page_url)
listings_per_page = 40
total_pages = (total_listings // listings_per_page) + (1 if total_listings % listings_per_page > 0 else 0)

for page in range(1, total_pages + 1):
page_url = f"{base_url}/page-{page}{search_params}" if page > 1 else first_page_url
print(f"Fetching {page_url}")

listing_urls = get_listings(page_url)
if listing_urls:
print(f"Found {len(listing_urls)} listings on page {page}.")
for url in listing_urls:
writer.writerow([url]) # Write each URL to the CSV file
else:
print(f"No more listings found at page {page}. Ending search.")
break

# List of tuples containing base URLs and their search parameters
base_urls = [
("https://www.kijiji.ca/b-for-rent/city-of-halifax/house",
"/k0c30349001117003 21?sort=dateDesc&radius=50.0&address=Halifax%2C+NS&ll=44.6475811%2C-63.5727683")
# Add more (base_url, search_params) tuples as needed
]

process_base_urls(base_urls)
```

- Script2: Get all apartments data

```
import requests
from bs4 import BeautifulSoup
import csv
import logging
import googlemaps
import re

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Initialize Google Maps client with your API key
gmaps = googlemaps.Client(key='AIzaSyDpYVAoSI-Cx4HANcaKWR-0eSqvzr8XDVk')
```

```
def parse_attributes(attr_element):
attribute_strings = []

two_lines_attrs = attr_element.find_all('li', class_="twoLinesAttribute-633292638")
for attr in two_lines_attrs:
dt = attr.find('dt')
dd = attr.find('dd')
if dt and dd:
attribute_strings.append(f"{dt.get_text(strip=True)}: {dd.get_text(strip=True)}")

parking_element = attr_element.find('dt', class_="twoLinesLabel-2332083105", text="Parking Included")
if parking_element:
parking_value = parking_element.find_next_sibling('dd', class_="twoLinesValue-2653438411").text
attribute_strings.append(f"Parking Included: {parking_value}")

group_attrs = attr_element.find_all('li', class_="attributeGroupContainer-1655609067")
for group_attr in group_attrs:
group_title = group_attr.find('h4').get_text(strip=True) if group_attr.find('h4') else None
included_items = []
items = group_attr.find_all('li', class_='withIcons-3012323231')
for item in items:
yes_icon = item.find('svg', class_='yesIcon-3148225030')
if yes_icon:
included_items.append(item.get_text(strip=True))
if included_items:
attribute_strings.append(f"{group_title}: {', '.join(included_items)}")

return '; '.join(attribute_strings)


def fetch_and_parse_attributes(url, url_number):
logging.info(f"Fetching URL number {url_number}: {url}")
response = requests.get(url)
details = {'URL': url}

if response.status_code == 200:
soup = BeautifulSoup(response.content, 'lxml')

title = soup.find('h1', class_="title-4206718449").text.strip() if soup.find('h1',
class_="title-4206718449") else 'N/A'
price = soup.find('div', class_="priceContainer-1877772231").text.strip() if soup.find('div',
class_="priceContainer-1877772231") else 'N/A'
location = re.sub(r'\(View Map\)', '',
soup.find('div', class_="locationContainer-1255831218").text).strip() if soup.find('div',
class_="locationContainer-1255831218") else 'N/A'

# Flexible description fetching logic
```

```
description_container = soup.find('div', class_="descriptionContainer-2067035870")
description = "N/A"
if description_container:
# Check for direct text within div
if description_container.find('p'):
paragraphs = description_container.find_all('p')
description = ' '.join([p.text.strip() for p in paragraphs if p.text.strip()])
else:
# Assume the text might be directly inside the div if no <p> tags are found
description = description_container.get_text(separator=' ', strip=True)

details['Title'] = title
details['Price'] = price
details['Location'] = location
details['Description'] = description
details['Attributes'] = ' | '.join(
filter(None, [parse_attributes(attr) for attr in soup.find_all('div', class_="attributeCard-923006758")]))

# Existing attribute handling
parking_included = soup.find('dt', class_="twoLinesLabel-2332083105", text="Parking Included")
details['Parking Included'] = parking_included.find_next_sibling('dd',
class_="twoLinesValue-2653438411").text if parking_included else 'N/A'

utilities_included_element = soup.find('h4', class_="attributeGroupTitle-889029213", text="Utilities
Included")
details[
'Utilities Included'] = '1' if utilities_included_element and not
utilities_included_element.find_next_sibling(
'ul', class_="list-2534755251").text.strip().lower() == "not included" else '0'

pet_friendly_element = soup.find('dt', class_="twoLinesLabel-2332083105", text="Pet Friendly")
details['Pet Friendly'] = '1' if pet_friendly_element and pet_friendly_element.find_next_sibling('dd',
class_="twoLinesValue-2653438411").text.strip().lower() == "yes" else '0'

size_element = soup.find('dt', class_="twoLinesLabel-2332083105", text="Size (sqft)")
details['Size (sqft)'] = size_element.find_next_sibling('dd',
class_="twoLinesValue-2653438411").text if size_element else 'N/A'

# New: Handle Furnished
furnished_element = soup.find('dt', class_="twoLinesLabel-2332083105", text="Furnished")
details['Furnished'] = '1' if furnished_element and furnished_element.find_next_sibling('dd',
class_="twoLinesValue-2653438411").text.strip().lower() == "yes" else '0'

unit_row = soup.find('div', class_="unitRow-2439405931")
# Correcting the way to extract Number of Bathrooms
if unit_row:
spans = unit_row.find_all('span', class_="noLabelValue-774086477")
details['Property Type'] = spans[0].text if spans else 'N/A'
```

```
property_config = spans[1].text.split(': ')[1] if len(spans) > 1 else 'N/A'
details['Den Available'] = '1' if 'den' in property_config.lower() else '0'
details['Property Config'] = re.sub(r'\s*\+\s*den', '', property_config, flags=re.I)
details['Number of Bathrooms'] = spans[2].text.split(': ')[1] if len(spans) > 2 else 'N/A'

geocode_result = gmaps.geocode(location)
if geocode_result:
lat = geocode_result[0]['geometry']['location']['lat']
lng = geocode_result[0]['geometry']['location']['lng']
details['Latitude'] = lat
details['Longitude'] = lng
else:
details['Latitude'] = 'N/A'
details['Longitude'] = 'N/A'

logging.info(f"Successfully parsed attributes for URL number {url_number}.")
return details
else:
logging.error(f"Failed to fetch URL: {response.status_code}")
return None


urls = []
with open('listings.csv', 'r', newline='', encoding='utf-8') as file:
csv_reader = csv.reader(file)
next(csv_reader) # Skip the header
urls = [row[0] for row in csv_reader]

listings_details = []
for index, url in enumerate(urls, start=1):
detail = fetch_and_parse_attributes(url, index)
if detail:
listings_details.append(detail)

fieldnames = ['URL', 'Title', 'Price', 'Location', 'Description', 'Attributes', 'Property Type', 'Property
Config',
'Den Available', 'Number of Bathrooms', 'Latitude', 'Longitude', 'Parking Included', 'Utilities Included',
'Pet Friendly', 'Size (sqft)', 'Furnished']

with open('data_v3.csv', 'w', newline='', encoding='utf-8') as file:
writer = csv.DictWriter(file, fieldnames=fieldnames)
writer.writeheader()
writer.writerows(listings_details)

logging.info("Completed writing listing details to CSV.")
```

- Capreit Apartments
    - Script3: Scrape Listings

```
import requests
import csv

def fetch_listings(cities):
url = 'https://www.capreit.ca/wp-admin/admin-ajax.php?action=property_json&language=en'
headers = {
'Accept': '*/*',
'Accept-Language': 'en-CA,en-US;q=0.9,en;q=0.8',
'Content-Length': '0',
'Origin': 'https://www.capreit.ca',
'Referer': 'https://www.capreit.ca/apartments-for-rent/',
'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/123.0.0.0 Safari/537.36',
'X-Requested-With': 'XMLHttpRequest',
}

response = requests.post(url, headers=headers)
if response.status_code == 200:
listings = response.json()
return [listing for listing in listings if listing.get('city') in cities]
else:
return f"Failed to fetch listings, status code: {response.status_code}"

def write_listings_to_csv(listings, filename='listings.csv'):
with open(filename, 'w', newline='', encoding='utf-8') as file:
writer = csv.writer(file)
# Write the header
writer.writerow([
'ID', 'Title', 'Address', 'City', 'Province', 'Postal Code', 'Min Rent', 'Price Range',
'Bedroom Range', 'Earliest Date', 'Vacancy Message', 'Latitude', 'Longitude', 'URL'
])

# Write the listing data
for listing in listings:
writer.writerow([
listing.get('id', ''),
listing.get('title', ''),
listing.get('address', ''),
listing.get('city', ''),
listing.get('province', ''),
listing.get('postal_code', ''),
listing.get('min_rent', ''),
listing.get('price_range', ''),
listing.get('bedroom_range', ''),
listing.get('earliest_date', ''),
```

```
listing.get('vacancy_message', ''),
listing.get('latitude', ''),
listing.get('longitude', ''),
listing.get('url', '')
])

# Specify the cities you are interested in
cities = ['Halifax', 'Bedford', 'Dartmouth']

# Fetch the listings
filtered_listings = fetch_listings(cities)

# Check if the listings were fetched successfully
if isinstance(filtered_listings, list):
# Write the filtered listings to a CSV file
write_listings_to_csv(filtered_listings)
print("Listings have been written to 'listings.csv'.")
else:
print(filtered_listings)
```

- Script4: Get all apartments data

```
import csv
import requests
from bs4 import BeautifulSoup

def fetch_details_from_url(url):
try:

response = requests.get(url)
response.raise_for_status()

# Parse the HTML content with BeautifulSoup
soup = BeautifulSoup(response.text, 'html.parser')

# Initialize containers even if 'nearby_container' is not found
schools, transportation, points_of_interest, shopping_dining = [], [], [], []

# Attempt to find the 'Nearby' container div
nearby_container = soup.find('div', class_='property-features-content-container-nearby')
if nearby_container:
# Find all nearby items by category
for nearby_item in nearby_container.find_all('div',
class_='property-features-content-container-nearby-item'):
category = nearby_item.find('h5').text.strip(':')
details = [li.get_text(strip=True) for li in nearby_item.find_all('li')]
```

```
# Assign details to the corresponding dictionary
if category == 'Schools':
schools = details
elif category == 'Transportation':
transportation = details
elif category == 'Points of Interest':
points_of_interest = details
elif category == 'Shopping & Dining':
shopping_dining = details
else:
print(f"The 'Nearby' container was not found in the HTML at {url}")

# Extract scores
scores = {score.get('data-label'): score.get('data-score')
for score in soup.select('.chart-score[data-label]')}

walk_score = scores.get('Walk Score', 'No walk score listed')
bike_score = scores.get('Bike Score', 'No bike score listed')

# Find all the sections that contain rent coverage details
coverage_sections = soup.select('.property-options-details-covered')

# Extract what rent covers and doesn't cover
rent_covers = [li.get_text(strip=True) for li in coverage_sections[0].select('ul li')]
rent_does_not_cover = [li.get_text(strip=True) for li in coverage_sections[1].select('ul li')] if len(
coverage_sections) > 1 else []

# Extract the features like 'Barrier Free Entrances', 'Cats', and 'Dogs (small/med)'
features = []
for feature in soup.select('.property-options-header-list li')[1:]:
feature_text = feature.get_text(strip=True)
if feature_text:
features.append(feature_text)

# Extract unit features
unit_features_header = soup.find('h2', string=lambda s: s and 'What Your New Home Has Built In' in s)
if unit_features_header:
unit_features_list = unit_features_header.find_next_sibling('div').find('ul')
unit_features = [li.get_text(strip=True) for li in
unit_features_list.find_all('li')] if unit_features_list else []
else:
unit_features = []

# Extract building features based on the header 'Building Features'
building_features_header = soup.find('h2', string=lambda text: 'Building Features' in text)
building_features = []
if building_features_header:
```

```
# The 'ul' might not be a direct sibling but within the same container 'div'
container_div = building_features_header.find_next('div')
if container_div:
building_features_ul = container_div.find('ul')
if building_features_ul:
building_features = [li.get_text(strip=True) for li in building_features_ul.find_all('li')]

# Extract apartment name and address
apartment_name_tag = soup.select_one('.property-hero-details-address-title[itemprop="name"]')
apartment_name = apartment_name_tag.get_text(strip=True) if apartment_name_tag else "Apartment
name not listed"

address_tag = soup.select_one('.property-hero-details-address-street[itemprop="address"]')
address = address_tag.get_text(strip=True) if address_tag else "Address not listed"

# Extract phone number
phone_number_tag = soup.select_one('.property-hero-details-contact a[href^="tel:"]')
phone_number = phone_number_tag.get_text(strip=True) if phone_number_tag else "Phone number not
listed"

# Create a list to hold all the scraped data
apartments = []

# Find all list items with class 'property-options-list-item'
for item in soup.select('.property-options-list-item'):
availability = item.select_one('.property-options-list-item-availability').get_text(strip=True)
price = item.select_one('.property-options-list-item-price b')
price = price.get_text(strip=True) if price else "Price not listed"
details = item.select('.property-options-list-item-details .property-options-item')

apartment_type = details[0].get_text(strip=True) if details else "Type not listed"
square_footage = details[1].get_text(strip=True) if len(details) > 1 else "Size not listed"

# Add the scraped data to the list
apartments.append({
'name': apartment_name,
'address': address,
'availability': availability,
'price': price,
'type': apartment_type,
'size': square_footage,
'phone_number': phone_number,
'features': ', '.join(features),
'rent_covers': ', '.join(rent_covers),
'rent_does_not_cover': ', '.join(rent_does_not_cover),
'walk_score': walk_score,
'bike_score': bike_score,
'unit_features': ', '.join(unit_features),
```

```
'building_features': ', '.join(building_features),
'nearby_schools': ', '.join(schools),
'nearby_transportation': ', '.join(transportation),
'nearby_points_of_interest': ', '.join(points_of_interest),
'nearby_shopping_dining': ', '.join(shopping_dining)
})

for apartment in apartments:
print(apartment)
return apartments

except requests.RequestException as e:
print(f"Error fetching details from {url}: {e}")
return None

def read_csv_and_fetch_details(filename,output_filename):
with open(filename, 'r', encoding='utf-8') as file:
reader = csv.DictReader(file)
all_details = []
for row in reader:
url = row['URL']
if url:
details = fetch_details_from_url(url)
if details:
all_details.append(details)

print(all_details)


if all_details:
with open(output_filename, 'a', newline='', encoding='utf-8') as csvfile: # Use 'a' to append
fieldnames = ['name', 'address', 'availability', 'price', 'type', 'size', 'phone_number', 'features',
'rent_covers', 'rent_does_not_cover', 'walk_score', 'bike_score', 'unit_features',
'building_features', 'nearby_schools', 'nearby_transportation', 'nearby_points_of_interest',
'nearby_shopping_dining']
writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
csvfile.seek(0, 2)
if csvfile.tell() == 0:
writer.writeheader()
for detail_list in all_details:
for detail in detail_list:
writer.writerow(detail)



read_csv_and_fetch_details('listings.csv', 'apartments_data.csv')
```

- Templeton Apartments
    - Script5: Scrape Listings

```
import requests
import csv

# Define the URL, headers, and parameters for the request
url = "https://api.theliftsystem.com/v2/search"
headers = {
"Accept": "application/json, text/javascript, */*; q=0.01",
"Accept-Language": "en-CA,en-US;q=0.9,en;q=0.8",
"Connection": "keep-alive",
"Origin": "https://www.templetonproperties.ca",
"Referer": "https://www.templetonproperties.ca/",
"Sec-Fetch-Dest": "empty",
"Sec-Fetch-Mode": "cors",
"Sec-Fetch-Site": "cross-site",
"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/123.0.0.0 Safari/537.36 Edg/123.0.0.0",
}
params = {
"locale": "en",
"only_available_suites": "",
"show_all_properties": "",
"client_id": "584",
"auth_token": "sswpREkUtyeYjeoahA2i",
"city_ids": "1170",
"geocode": "",
"min_bed": "-1",
"max_bed": "5",
"min_bath": "-1",
"max_bath": "10",
"min_rate": "0",
"max_rate": "10000",
"property_types": "apartments,houses",
"order": "min_rate ASC, max_rate ASC, min_bed ASC, max_bed ASC",
"limit": "9999",
"offset": "0",
"count": "false",
"show_custom_fields": "true",
"show_amenities": "true",
}

# Send the GET request
response = requests.get(url, headers=headers, params=params)

# Proceed if the request was successful
if response.status_code == 200:
```

```
listings = response.json()

# Open (or create) a CSV file to write the data
with open('listings.csv', mode='w', newline='') as file:
writer = csv.writer(file)

# Write the header row
writer.writerow(
['ID', 'Name', 'Type', 'Address', 'City', 'Province', 'Contact Name', 'Contact Phone', 'Min Rate',
'Max Rate', 'URL', 'Availability Status', 'Min Availability Date', 'Availability Status Label',
'Availability Count'])

# Iterate over the listings and write their details
for listing in listings:
writer.writerow([
listing['id'],
listing['name'],
listing['property_type'],
listing['address']['address'],
listing['address']['city'],
listing['address']['province'],
listing['contact']['name'],
listing['contact']['phone'],
listing['statistics']['suites']['rates']['min'],
listing['statistics']['suites']['rates']['max'],
listing['permalink'],
listing.get('availability_status', 'N/A'),
listing.get('min_availability_date', 'N/A'),
listing.get('availability_status_label', 'N/A'),
listing.get('availability_count', 'N/A'),
])
print("Data has been written to listings.csv")
else:
print("Failed to retrieve data")
```

- Script6: Get all apartments data

```
import csv
import requests
from bs4 import BeautifulSoup

def fetch_details_from_url(url):
try:
# Send a GET request to the URL
response = requests.get(url)
response.raise_for_status()
```

```
# Parse the HTML content with BeautifulSoup
soup = BeautifulSoup(response.text, 'html.parser')

# Extract the title, phone, and address based on the provided classes
title_tag = soup.find('h1', class_='title')
phone_tag = soup.find('span', class_='phone')
address_tag = soup.find('span', class_='address')

title = title_tag.get_text(strip=True) if title_tag else "Title not listed"
phone = phone_tag.get_text(strip=True) if phone_tag else "Phone number not listed"
address = address_tag.get_text(strip=True) if address_tag else "Address not listed"

def extract_amenities():
amenity_groups = soup.find_all('div', class_='amenity-group')
amenities = {
'Suite Amenities': [],
'Building Amenities': [],
'Utilities': [],
'Parking': [],
'Pet Policy': []
}

for group in amenity_groups:
# Extract the title of the amenity group
group_title = group.find('h2').get_text(strip=True)
# Normalize the title to match our amenities dictionary keys
if 'suite' in group_title.lower():
key = 'Suite Amenities'
elif 'building' in group_title.lower():
key = 'Building Amenities'
elif 'pet policy' in group_title.lower():
key = 'Pet Policy'
else:
continue # Skip any groups that don't match the expected titles

# Extract each amenity from the group
for amenity in group.find_all('div', class_='amenity-holder'):
amenities[key].append(amenity.get_text(strip=True))

# Join each list of amenities into a string
for key, value in amenities.items():
amenities[key] = ', '.join(value)

return amenities

# Now call the function and store its result
amenities = extract_amenities()
```

```
def extract_utilities():
utilities_list = []
utilities_section = soup.find('section', class_='widget utilities')
if utilities_section:
utility_holders = utilities_section.find_all('div', class_='utility-holder')
for utility in utility_holders:
# Extract the name and inclusion status (e.g., "Heat Included")
name = utility.find('span', class_='name').get_text(strip=True) if utility.find('span',
class_='name') else ''
included = utility.find('span', class_='included').get_text(strip=True) if utility.find('span',
class_='included') else ''
utilities_list.append(f"{name} {included}")
return ', '.join(utilities_list)

# Now call the function and store its result
amenities['Utilities'] = extract_utilities()

def extract_parking_info():
# Find the h2 tag with text 'Parking'
parking_header = soup.find('h2', string=lambda text: 'Parking' in text)

# Initialize an empty string for parking info
parking_info = ""

# If the parking header is found, find the next <p> tag which contains the parking info
if parking_header:
parking_tag = parking_header.find_next('p')
if parking_tag:
parking_info = parking_tag.get_text(strip=True)

return parking_info

# Now call the function and store its result
amenities['Parking'] = extract_parking_info()

# Create a list to hold all the scraped data
apartments = []

# Find the container that holds suite information
suites_custom_container = soup.find('div', class_='suites-custom')
if suites_custom_container:
suite_groups = suites_custom_container.find_all('div', class_='suite-group-suites')
for suite_group in suite_groups:
suites = suite_group.find_all('div', class_='suite')
for suite in suites:
suite_details = {
'Title': title,
```

```
'Phone': phone,
'Address': address,
'Suite Type': suite.find('div', class_='suite-type').get_text(strip=True),
'Unit': suite.find('div', class_='suite-unit').get_text(strip=True),
'Beds': suite.find('div', class_='suite-bed').get_text(strip=True),
'Baths': suite.find('div', class_='suite-bath').get_text(strip=True),
'Price': suite.find('div', class_='suite-rate').get_text(strip=True),
'Availability': suite.find('div', class_='suite-availability').get_text(strip=True)
}
# Include amenities
suite_details.update(amenities)

apartments.append(suite_details)

# Print out the results
for apartment in apartments:
print(apartment)

return apartments

except requests.RequestException as e:
print(f"Error fetching details from {url}: {e}")
return None

def read_csv_and_fetch_details(filename,output_filename):
with open(filename, 'r', encoding='utf-8') as file:
reader = csv.DictReader(file)
all_details = []
for row in reader:
url = row['URL']
if url:
details = fetch_details_from_url(url)
if details:
all_details.append(details)

print(all_details)


if all_details:
with open(output_filename, 'a', newline='', encoding='utf-8') as csvfile: # Use 'a' to append
fieldnames = ['Title', 'Phone', 'Address', 'Suite Amenities', 'Building Amenities', 'Utilities', 'Parking', 'Pet
Policy', 'Suite Type', 'Unit', 'Beds', 'Baths', 'Price', 'Availability']
writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
csvfile.seek(0, 2)
if csvfile.tell() == 0:
writer.writeheader()
for detail_list in all_details:
for detail in detail_list:
```

```
writer.writerow(detail)



read_csv_and_fetch_details('listings.csv', 'apartments_data.csv')
```

- Script7: Zumper

```
import requests
import pandas as pd

url = "https://www.zumper.com/api/svc/inventory/v1/listables/maplist/search"

headers = {
  "authority": "www.zumper.com",
   "accept": "*/*",
   "accept-language": "en-GB,en-US;q=0.9,en;q=0.8",
   "content-type": "application/json",
   "origin": "https://www.zumper.com",
   "referer": "https://www.zumper.com/apartments-for-rent/halifax-ns",
   "sec-ch-ua": '"Chromium";v="122", "Not(A:Brand";v="24", "Google Chrome";v="122"',
   "sec-ch-ua-mobile": "?0",
   "sec-ch-ua-platform": '"Windows"',
   "sec-fetch-dest": "empty",
   "sec-fetch-mode": "cors",
   "sec-fetch-site": "same-origin",
   "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/122.0.0.0 Safari/537.36",
   "x-csrftoken":
"9Gm1gZPWhuSLsRXPbY8tBSXcILgvWIQ4Aw05yZYlQfbwZGCnBACkI3oAYwZkfU7y6QqV9xOg
K38krdGQ",
   "x-zumper-xz-token": "axebrc4orkx.25aqt42wu",
}

data = {
   "url": "halifax-ns",
   "limit": 50,
   "propertyTypes": {"exclude": [16, 17]},
   "external": True,
   "offset": 0,
   "descriptionLength": 580
}
```

```
response = requests.post(url, headers=headers, json=data)

if response.status_code == 200:
    # Check for a successful response
    if response.status_code == 200:
        # Convert the response to JSON
        json_response = response.json()

        # The data you are interested in is under the 'listables' key
        listables = json_response['listables']

        # Define the desired columns
        columns = [
            "address", "city", "lat", "lng", "pets", "agent_name", "brokerage_name",
            "state", "building_name", "city_state", "amenity_tags", "building_amenity_tags",
            "date_available", "max_price", "min_price", "max_bedrooms", "min_bedrooms",
            "max_bathrooms", "min_bathrooms", "floorplan_count", "min_lease_days",
            "max_lease_days", "min_square_feet", "max_square_feet", "zipcode",
            "max_all_bathrooms", "min_all_bathrooms"
        ]

        # Extract the data and construct a list of dictionaries
        extracted_data = []
        for listing in listables:
            # Extract only the required fields for each listing
            row_data = {key: listing.get(key) for key in columns}
            extracted_data.append(row_data)

        # Convert the extracted data into a pandas DataFrame
        df = pd.DataFrame(extracted_data, columns=columns)

        # Write the DataFrame to a CSV file
        csv_file_path = 'apartments_data.csv'
        df.to_csv(csv_file_path, index=False)
        print(f"Data has been written to {csv_file_path}")
    else:
        print(f"Failed to fetch data: {response.status_code} - {response.text}")
```

- Script8: Killiam Apartments

```
import time
import pandas as pd
```

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from itertools import islice


chrome_options = Options()
# chrome_options.add_argument("--headless")
chrome_options.add_argument('--disable-gpu')
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument("--start-maximized")
chrome_options.add_argument("--window-size=1920x1080")


url = 'https://killamreit.com/'


driver.get(url)


data = []


listings = driver.find_elements(By.CLASS_NAME, 'listing-container')
for listing in listings:
listing_data = {
'Place': listing.find_element(By.CLASS_NAME, 'place-text').text,
'Listing_Name': listing.find_element(By.CLASS_NAME, 'listing-title').text,
'Listing_Address': listing.find_element(By.CLASS_NAME, 'address-field').text,
'Price': listing.find_element(By.CLASS_NAME, 'price-label').text,
'Bedrooms': listing.find_element(By.CLASS_NAME, 'bedroom-count').text,
'Bathrooms': listing.find_element(By.CLASS_NAME, 'bathroom-count').text,
'Size': listing.find_element(By.CLASS_NAME, 'property-size').text,
'Availability': listing.find_element(By.CLASS_NAME, 'availability-date').text,
'Common / Social Room': listing.find_element(By.CLASS_NAME, 'social-room-availability').text,
'Balcony': listing.find_element(By.CLASS_NAME, 'balcony-availability').text,
'Wheelchair Accessible': listing.find_element(By.CLASS_NAME, 'wheelchair-access-flag').text,
'Dog Friendly': listing.find_element(By.CLASS_NAME, 'dog-friendly-flag').text,
'Underground / Parking Garage': listing.find_element(By.CLASS_NAME, 'parking-garage-availability').text,
'EV Charging Stations': listing.find_element(By.CLASS_NAME, 'ev-charging-flag').text,
'Security': listing.find_element(By.CLASS_NAME, 'security-feature-flag').text,
'Air Conditioning': listing.find_element(By.CLASS_NAME, 'air-conditioning-availability').text,
'In-Suite Laundry': listing.find_element(By.CLASS_NAME, 'laundry-suite-availability').text,
'Guest Suite': listing.find_element(By.CLASS_NAME, 'guest-suite-availability').text,
'Terrace': listing.find_element(By.CLASS_NAME, 'terrace-availability').text,
'Laundry (In Building)': listing.find_element(By.CLASS_NAME, 'building-laundry-availability').text,
'Elevator': listing.find_element(By.CLASS_NAME, 'elevator-availability').text,
'Cat Friendly': listing.find_element(By.CLASS_NAME, 'cat-friendly-flag').text,
```

```
'Hot Water Included': listing.find_element(By.CLASS_NAME, 'hot-water-included-flag').text,
'Solar Panel': listing.find_element(By.CLASS_NAME, 'solar-panel-feature').text,
'Parking Lot': listing.find_element(By.CLASS_NAME, 'parking-lot-availability').text,
'Fitness Gym': listing.find_element(By.CLASS_NAME, 'gym-availability').text,
'Storage Space': listing.find_element(By.CLASS_NAME, 'storage-space-availability').text,
'Heat Included': listing.find_element(By.CLASS_NAME, 'heat-included-flag').text,
}
for key in listing_data:
if 'available' in listing_data[key]:
listing_data[key] = 1
elif 'unavailable' in listing_data[key]:
listing_data[key] = 0

data.append(listing_data)



driver.quit()

df = pd.DataFrame(data)

df.to_csv('scraped_data.csv', index=False)
```

- 444Rent
    - Script9: Scrape Listings

```
import csv
import requests
from bs4 import BeautifulSoup

# The base URL and the specific page you want to scrape
base_url = "https://www.444rent.com/"
target_url = "https://www.444rent.com/apartments.asp" # Example target URL

# Use requests to fetch the content from the URL
response = requests.get(target_url)
html_content = response.text

# Use BeautifulSoup to parse the HTML
soup = BeautifulSoup(html_content, 'html.parser')

# Find all <a> tags in the HTML
a_tags = soup.find_all('a')

# Extract href attributes that start with "details.asp?"
urls = [base_url + a['href'] for a in a_tags if 'href' in a.attrs and a['href'].startswith('details.asp?')]
```

```
# Write the URLs to a CSV file, including a header
csv_file_path = 'D:\\Hackathon\\444rent\\listings.csv'
with open(csv_file_path, 'w', newline='', encoding='utf-8') as file:
writer = csv.writer(file)
# Write the header
writer.writerow(['URL'])
# Write each URL
for url in urls:
writer.writerow([url])

print(f"Extracted {len(urls)} URLs and wrote them to listings.csv")
```

- Script10: Get all apartments data

```
import csv
import requests
from bs4 import BeautifulSoup
import pandas as pd

# Load the URLs from the listings.csv file
listings_csv_path = 'D:\\Hackathon\\444rent\\listings.csv'
urls_df = pd.read_csv(listings_csv_path)

# Prepare to write extracted data to a new CSV
extracted_info_path = 'D:\\Hackathon\\444rent\\extracted_info.csv'


def extract_info_from_row(soup, parameter_name):
# Find the row with the specific parameter
param_row = soup.find('b', class_='normal', string=lambda t: parameter_name in t)
if param_row:
# Navigate to the appropriate <td> that contains the value(s)
parent_td = param_row.find_parent('td').find_next_sibling('td')
if parent_td:
# Find all nested tables within the parent <td>
nested_tables = parent_td.find_all('table')
# Extract the text from the second <td> of each row
values = [table.find_all('td')[1].text.strip() for table in nested_tables if len(table.find_all('td')) > 1]
return '; '.join(values)
return 'N/A'


fieldnames = ['URL', 'Price', 'Availability', 'Address', 'Type', 'Area', 'Lease', 'Included', 'Parking', 'Storage',
'Deposit', 'Policy']
```

```
# Define a function to extract information based on provided parameters
def extract_info(url):
response = requests.get(url)
if response.status_code == 200:
soup = BeautifulSoup(response.content, 'html.parser')
suitemain_div = soup.find(id='suitemain')

data = {
'URL': url,
'Price': soup.find(id='pricemain').text if soup.find(id='pricemain') else 'N/A',
'Availability': soup.find(id='pricemain').find_next_sibling('div').text if soup.find(
id='pricemain') else 'N/A', # Adjust based on actual structure
'Address': soup.find(id='addressmain').text if soup.find(id='addressmain') else 'N/A',
'Type': soup.find(id='suitemain').text.strip() if suitemain_div else 'N/A',
'Area': soup.find(id='floorplanmain').text if soup.find(id='floorplanmain') else 'N/A',
'Lease': extract_info_from_row(soup, 'Lease:'),
'Included': extract_info_from_row(soup, 'Included:'),
'Parking': extract_info_from_row(soup, 'Parking:'),
'Storage': extract_info_from_row(soup, 'Storage:'),
'Deposit': extract_info_from_row(soup, 'Deposit:'),
'Policy': extract_info_from_row(soup, 'Policy:')
}
return data
else:
return None


# Open the new CSV file for writing
with open(extracted_info_path, 'w', newline='', encoding='utf-8') as file:
writer = csv.DictWriter(file, fieldnames=fieldnames)
writer.writeheader()

# Extract information for each URL and write to the CSV
for url in urls_df['URL']:
info = extract_info(url)
if info:
writer.writerow(info)

print(f"Extraction complete. Data written to {extracted_info_path}")
```

- Script11: Apartments.com

```
import time
import pandas as pd
from selenium import webdriver
from selenium.webdriver.common.by import By
```

```
from selenium.webdriver.chrome.options import Options

chrome_options = Options()
##chrome_options.add_argument("--headless")
chrome_options.add_argument('--disable-gpu')
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument("--start-maximized")
chrome_options.add_argument("--window-size=1920x1080")

driver = webdriver.Chrome(options=chrome_options)

url = 'https://www.apartments.com/halifax-ns/'
driver.get(url)
time.sleep(2)
total_elements = driver.find_elements(By.CLASS_NAME, "carouselInner")
total_length = len(total_elements)

columns = ["name", "address", "rent", "bedroom", "bathroom", "area", "utilities"]
df = pd.DataFrame(columns=columns)
for i in range(total_length):
elements_with_class = driver.find_elements(By.CLASS_NAME, "carouselInner")
elements_with_class[i].click();
time.sleep(2)

propertyName = driver.find_element(By.ID, "propertyName").text
propertyAddress = [ele.text for ele in driver.find_elements(By.CLASS_NAME, "delivery-address")]
rentInfo = [ele.text for ele in driver.find_elements(By.CLASS_NAME, "rentInfoDetail")]
amneties = [ele.text for ele in driver.find_elements(By.CLASS_NAME, "amenityCard")]
address = ", ".join(propertyAddress) if propertyAddress else "0"
rent = rentInfo[0] if rentInfo else "0"
bedroom = rentInfo[1].split(" ")[0] if len(rentInfo) > 1 else "0"
bathroom = rentInfo[2].split(" ")[0] if len(rentInfo) > 2 else "0"
area = rentInfo[3] if len(rentInfo) > 3 else "0"
utilities = ", ".join(amneties) if amneties else "0"

df = pd.concat(
[df, pd.DataFrame([[propertyName, address, rent, bedroom, bathroom, area, utilities]],
columns=columns)],
ignore_index=True)

driver.back()
time.sleep(1)

# Export the DataFrame to a CSV file
df.to_csv('apartments_data.csv', index=False)

driver.quit()
```

- Script12: West22 Living

```
import requests
from bs4 import BeautifulSoup
import pandas as pd

def extract_unit_details(unit_link):

unit_response = requests.get(unit_link)
unit_soup = BeautifulSoup(unit_response.content, 'html.parser')
table = unit_soup.find('table', class_='responsive add-border')

if table:
utilities = ['Heat', 'Hot Water', 'Power']
policies = ['Tenant Insurance Required', 'Smoke Free Building', 'Cats Allowed']

# Initialize dictionaries to hold the binary values for utilities and policies
utilities_data = {utility: 0 for utility in utilities}
policies_data = {policy: 0 for policy in policies}

# Extracting utilities
utilities_list = table.find('th', string='Utilities').find_next('td').find_all('li')
for utility in utilities_list:
utility_name = utility.get_text(strip=True)
if utility_name in utilities_data:
utilities_data[utility_name] = 1

# Extracting policies
policies_list = table.find('th', string='Policy').find_next('td').find_all('p')
for policy in policies_list:
policy_name = policy.get_text(strip=True)
if policy_name in policies_data:
policies_data[policy_name] = 1

# Lease and Deposit can be added as it is
lease = table.find('th', string='Lease').find_next('td').get_text(strip=True)
deposit = table.find('th', string='Deposit').find_next('td').get_text(strip=True)
parking = table.find('th', string='Parking').find_next('td').get_text(strip=True)

# Returning a combined dictionary of all details
return {
'LEASE': lease,
'PARKING': parking,
'DEPOSIT': deposit,
**utilities_data,
**policies_data
```

```
}
else:
return {}


url = "https://west22living.com/availability/"
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')

headers = ['Unit', 'Type', 'Area ft^2', 'Available', 'Price','Unit_Link']

rows = []
for row in soup.find('table', class_='responsive').find_all('tr', class_='availability-row'):
cells = row.find_all('td')
unit_link = row.find('td', class_='table-unit-type').a['href']
items = [cell.get_text(strip=True) for cell in cells if cell.get_text(strip=True) != '']
row_data = {headers[i]: items[i] for i in range(len(items))}
row_data['Unit_Link'] = unit_link

unit_details = extract_unit_details(unit_link)
row_data.update(unit_details)

rows.append(row_data)

df = pd.DataFrame(rows)

df['Listing_Address'] = '7037 Mumford Road'

csv_filename = 'listings.csv'
df.to_csv(csv_filename, index=False)

print(f"Data extracted and saved to {csv_filename}")
```

- RentSeeker
    - Script13: Scrape Listings

```
import requests
import json
import csv

# Headers from the curl command
headers = {
'Accept-Language': 'en-CA,en-US;q=0.9,en;q=0.8',
'Connection': 'keep-alive',
'Origin': 'https://www.rentseeker.ca',
'Referer': 'https://www.rentseeker.ca/',
'Sec-Fetch-Dest': 'empty',
```

```
'Sec-Fetch-Mode': 'cors',
'Sec-Fetch-Site': 'cross-site',
'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/123.0.0.0 Safari/537.36 Edg/123.0.0.0',
'accept': 'application/json',
'content-type': 'application/x-www-form-urlencoded',
'sec-ch-ua': '"Microsoft Edge";v="123", "Not:A-Brand";v="8", "Chromium";v="123"',
'sec-ch-ua-mobile': '?0',
'sec-ch-ua-platform': 'macOS'
}

# Data payload from the curl command
data = json.dumps({
"params":
"query=&hitsPerPage=1000&page=0&numericFilters=%5B%5B%22type%3D2%22%5D%5D&insideB
oundingBox=%5B%5B44.56773828133929%2C-63.716516013232415%2C44.72967569049512%2C-
63.43396138676757%5D%5D"
})

# Target URL
url = 'https://8hvk5i2wd9-dsn.algolia.net/1/indexes/rentseeker_prod_properties/query?x-algolia-
agent=Algolia%20for%20JavaScript%20(3.33.0)%3B%20Browser&x-algolia-application-
id=8HVK5I2WD9&x-algolia-api-key=68a749c1cd4aff1ca2c87a160617bd61'

# Make the request
response = requests.post(url, headers=headers, data=data)

# Check for successful response
if response.status_code == 200:
data = response.json()

# Extract listings URLs
listings = [hit['url'] for hit in data['hits']]

# Export URLs to a CSV file
with open('listings.csv', 'w', newline='') as csvfile:
writer = csv.writer(csvfile)
writer.writerow(['URL']) # Write the header row
for url in listings:
writer.writerow([url])

print('Listings URLs exported to listing.csv')

else:
print('Request failed with status code:', response.status_code)
```

- Script14: Get all apartments data

```
import requests
from bs4 import BeautifulSoup
import pandas as pd

# Function to extract data for a single property
def fetch_details(url):
try:

response = requests.get(url)
response.raise_for_status() # Raise an error if the request fails

# Parse the HTML content
soup = BeautifulSoup(response.content, 'html.parser')

def extract_data(soup):
# The data will be a list of dictionaries
data = []

# Find the accordion containing the apartment details
accordion = soup.find('div', class_='accordion')

# Iterate through each accordion-button to extract the apartment details
for button in accordion.find_all('button', class_='accordion-button'):
# Create a dictionary to store the details for each apartment
apartment_details = {}

# Extract the individual details
container = button.find('div', class_='container')
rows = container.find_all('div', class_='row')

# There should only be one row of details per button
for row in rows:
columns = row.find_all('div', recursive=False)
apartment_details['Bedroom Type'] = columns[0].get_text(strip=True)
apartment_details['Pricing'] = columns[1].get_text(strip=True)
apartment_details['Bathrooms'] = columns[2].get_text(strip=True)
apartment_details['Sq. Ft.'] = columns[3].get_text(strip=True)
apartment_details['Availability'] = columns[4].get_text(strip=True)

# Add the dictionary to our data list
data.append(apartment_details)

return data

def extract_address(soup):
overview_section = soup.find('section', id='overview')
```

```
address = overview_section.find('h1').text.strip()
return address

def extract_property_features(soup):
# Initialize an empty string for features
property_features = ''

# Find the features section
features_section = soup.find('section', id='features')

# Proceed only if the section was found
if features_section:
# Get all the 'feature' divs within the 'features' section
feature_divs = features_section.find_all('div', class_='feature')

# Collect all features
features = []
for feature_div in feature_divs:
# The text of each feature is within the 'text' class
text_div = feature_div.find('div', class_='text')
features.extend([p.get_text(strip=True) for p in text_div.find_all('p')])

property_features = ', '.join(features)

return property_features

def extract_included_in_rent(soup):
# Try finding the section by the class 'list', if it fails, then try 'list' as an id
included_in_rent_section = soup.find('section', class_='list') or soup.find('section', id='list')

# Check if the section was found, if not, return an empty string or appropriate message
if included_in_rent_section is None:
return 'Not Found'

# Find the ul element that contains the list items
included_ul = included_in_rent_section.find('ul')

# If the ul is not found, return an empty string or appropriate message
if included_ul is None:
return 'Not Found'

# Extract the text from each list item
included_items = [li.get_text(strip=True) for li in included_ul.find_all('li')]

return ', '.join(included_items)

def extract_title(soup):
# Find the title container
```

```
title_container = soup.find('div', class_='name')

# If the container is found, extract the title text
if title_container:
title = title_container.find('h2').get_text(strip=True)
else:
title = 'Title Not Found'

return title

# Extract the data
data = extract_data(soup)
address = extract_address(soup)
property_features = extract_property_features(soup)
included_in_rent = extract_included_in_rent(soup)
property_title = extract_title(soup)

# Create a dataframe for the floor plans
df = pd.DataFrame(data)

# Add the address and property features as new columns to the dataframe
df['Address'] = address
df['Property Features'] = property_features
# Check if included_in_rent is not 'Not Found' before adding to dataframe
if included_in_rent != 'Not Found':
# Add the 'Included in Rent' information to your dataframe
df['Included in Rent'] = included_in_rent

df['Property Title'] = property_title
return df

except requests.RequestException as e:
print(f"Error fetching details from {url}: {e}")
return None

# Main function to read the input CSV and write to an output CSV
def read_csv_and_fetch_details(input_csv, output_csv):
# Read the CSV file containing URLs
urls_df = pd.read_csv(input_csv)

# Prepare an empty DataFrame to collect all the details
all_details_df = pd.DataFrame()

# Iterate over each URL in the DataFrame
for index, row in urls_df.iterrows():
url = row['URL']
print(f"Fetching details for {url}...")
details = fetch_details(url)
```

```
all_details_df = pd.concat([all_details_df, details], ignore_index=True)

# Check if the DataFrame is not empty before saving it to CSV
if not all_details_df.empty:
all_details_df.to_csv(output_csv, index=False)
print(f"Details saved to {output_csv}")
else:
print("No details were extracted.")

# Specify the name of your input and output CSV files
input_csv_file = 'listings.csv'
output_csv_file = 'apartments_data.csv'

# Call the function to start the process
read_csv_and_fetch_details(input_csv_file, output_csv_file)
```

Data Cleaning

- Script15

```
import pandas as pd
import numpy as np

data = pd.read_csv('raw_dataV2.csv')

# Clean column names
data.columns = data.columns.str.strip()

# Clean string data in all columns
for col in data.select_dtypes(include=['object']).columns:
data[col] = data[col].astype(str).str.strip() # Convert to string and remove leading/trailing spaces
data[col] = data[col].apply(lambda x: " ".join(x.split())) # Replace multiple spaces with a single space only if x is a string

# Create a full context column so we can extract more information using NLP
data['context'] = data[['Title', 'Description', 'Attributes', 'Building Name']].astype(str).agg(', '.join, axis=1)

# Modify the 'Rooms' and 'Number of Bathrooms' columns with a regex that accounts for potential non-matches
data['Rooms'] = data['Rooms'].astype(str).str.extract(r'(\d+)').astype(float)
data['Number of Bathrooms'] = data['Number of Bathrooms'].astype(str).str.extract(r'(\d+(?:\.\d+)?)', expand=False).astype(float)

# Fill blanks in 'Den Included' with 0
data['Den Included'] = data['Den Included'].replace(np.nan, 0)

def clean_and_convert_size(size):
# Convert size to string to safely use string methods
size_str = str(size)
```

```
# Replace "Not Available" and blanks with NaN
if pd.isna(size) or size_str.strip() in ["", "Not Available", "nan"]:
return np.nan
# Remove commas and " sq ft", "Sq Ft", "sqft", "Ft2" for uniformity
size_cleaned = size_str.replace(",", "").lower()
for non_numeric in [" sq ft", "sqft", "ft2", "sq ft"]:
size_cleaned = size_cleaned.replace(non_numeric, "")
# Handle ranges by calculating the average
if '-' in size_cleaned:
size_range = size_cleaned.split('-')
try:
size = int(np.mean([int(s.strip()) for s in size_range]))
except ValueError:
return np.nan
else:
try:
size = int(size_cleaned.strip())
except ValueError:
return np.nan
return size


# Apply the cleaning function to the "Size (sqft)" column
data['Size (sqft)'] = data['Size (sqft)'].apply(clean_and_convert_size)

# Impute the data for Size column
data['Size (sqft)'] = pd.to_numeric(data['Size (sqft)'], errors='coerce') # Convert to numeric, invalid parsing to NaN
data.loc[data['Size (sqft)'] < 50, 'Size (sqft)'] = np.nan # Set values less than 50 to NaN
avg_size_per_room = data.groupby('Rooms')['Size (sqft)'].mean()
data['Size (sqft)'] = data.apply(
lambda row: avg_size_per_room[row['Rooms']] if pd.isna(row['Size (sqft)']) else row['Size (sqft)'],
axis=1
)
data['Size (sqft)'] = data['Size (sqft)'].astype(int)

# Handle missing values
def handle_bathrooms(row):
if row["Number of Bathrooms"] == "":
if row['Rooms'] == 1:
row["Number of Bathrooms"] = 1
else:
row["Number of Bathrooms"] = row['Rooms']//2
return row
data['Number of Bathrooms'] = data['Number of Bathrooms'].fillna("")
data = data.apply(lambda row: handle_bathrooms(row), axis=1)
```

```
data.drop(['URL', 'Title', 'Location', 'Description', 'Attributes', 'Parking Included', 'Utilities Included', 'Pet Friendly', 'Furnished', 'Building Name', 'Distance from Hospital', 'Distance from Police Station', 'Distance to Nearest Store', 'Distance to Nearest Pharmacy', 'context'], axis=1, inplace=True)
data = data.dropna(subset=['Latitude', 'Longitude'])

# Write file for easier viewing
data.to_csv('cleaned_data.csv', index=False)
data.head(5)
```

- Script16

```
import pandas as pd
import numpy as np
from sklearn.impute import KNNImputer

# Assuming 'data' is your DataFrame
data = pd.read_csv('cleaned_data.csv')

# Apply your filters
data = data[(data['Latitude'] >= 44.4) & (data['Latitude'] <= 45.1)]
data = data[(data['Longitude'] >= -64) & (data['Longitude'] <= -63.2)]

# Reset index after filtering
data.reset_index(drop=True, inplace=True)

# Prepare data for imputation
columns_for_imputation = ['Longitude', 'Latitude', 'Transit Score', 'Bike Score', 'Walk Score']
data_for_imputation = data[columns_for_imputation]

# Initialize and perform the imputation
imputer = KNNImputer(n_neighbors=3)
imputed_data = imputer.fit_transform(data_for_imputation)

# Convert imputed data to DataFrame and reset index
imputed_df = pd.DataFrame(imputed_data, columns=columns_for_imputation)
imputed_df.reset_index(drop=True, inplace=True)

# Round scores and convert to integer
imputed_df['Transit Score'] = np.round(imputed_df['Transit Score']).astype(int)
imputed_df['Bike Score'] = np.round(imputed_df['Bike Score']).astype(int)
imputed_df['Walk Score'] = np.round(imputed_df['Walk Score']).astype(int)

# Assign the imputed and rounded scores back
data[columns_for_imputation] = imputed_df

# Outlier Removal
def modified_z_score(column):
```

```
median = column.median()
mad = np.median(np.abs(column - median))
if mad == 0: # Prevent division by zero
return column * 0 # Return a zeroed series to preserve shape
return 0.6745 * (column - median) / mad

# Apply the Modified Z-Score Method
threshold = 3.5 # Adjust based on your tolerance for outliers
cleaned_indices = set(data.index) # Initialize with all indices

for col in data.select_dtypes(include=['float64', 'int64']).columns:
z_scores = modified_z_score(data[col])
# Update cleaned indices to keep rows within the threshold
cleaned_indices = cleaned_indices.intersection(set(data.index[(z_scores < threshold) & (z_scores > -threshold)]))

# Convert set to list before filtering
cleaned_indices_list = list(cleaned_indices)

# Filter the DataFrame to only include rows with indices in cleaned_indices_list
data = data.loc[cleaned_indices_list]

# Write file for easier viewing
data.to_csv('filtered_data.csv', index=False)
```

Linear Regression

- Script17

```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Function to calculate adjusted R2
def adjusted_r2_score(r2, n, p):
"""
Calculate the adjusted R-squared score.
:param r2: The R-squared score.
:param n: The number of samples.
:param p: The number of features.
:return: The adjusted R-squared score.
"""
```

```
return 1 - (1 - r2) * (n - 1) / (n - p - 1)

# Load the data
data = pd.read_csv('filtered_data.csv')

# Separate the target variable and features
X = data.drop('Price', axis=1)
y = data['Price']

# Define categorical and numerical features
categorical_features = X.select_dtypes(include=['object', 'bool']).columns.tolist()
numerical_features = X.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Define the preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Define the preprocessing for numerical data
numerical_transformer = Pipeline(steps=[
('scaler', StandardScaler())
])

# Combine preprocessors
preprocessor = ColumnTransformer(
transformers=[
('num', numerical_transformer, numerical_features),
('cat', categorical_transformer, categorical_features)
]
)

# Create the modeling pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
('regressor', LinearRegression())])

# Initialize KFold for cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for each fold
r2_scores = []
mse_scores = []
mae_scores = []
adjusted_r2_scores = []

for train_index, test_index in kf.split(X):
X_train, X_test = X.iloc[train_index], X.iloc[test_index]
y_train, y_test = y.iloc[train_index], y.iloc[test_index]
# Fit the model on the training data
```

```
model.fit(X_train, y_train)
# Predict on the test data
y_pred = model.predict(X_test)
# Calculate metrics
r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
# Calculate the number of features after preprocessing
X_test_transformed = model.named_steps['preprocessor'].transform(X_test)
p = X_test_transformed.shape[1]
n = len(y_test)
adjusted_r2 = adjusted_r2_score(r2, n, p)
# Append to lists
r2_scores.append(r2)
mse_scores.append(mse)
mae_scores.append(mae)
adjusted_r2_scores.append(adjusted_r2)

# Display mean metrics
print("Cross-validation scores:")
print(f"Mean R2: {np.mean(r2_scores)}")
print(f"Mean Adjusted R2: {np.mean(adjusted_r2_scores)}")
print(f"Mean MSE: {np.mean(mse_scores)}")
print(f"Mean MAE: {np.mean(mae_scores)}")
```

Random Forest

- Script18

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor
from scipy.stats import zscore

data = pd.read_csv('filtered_data.csv')

# Identify your feature matrix X and target vector y
X = data.drop('Price', axis=1)
y = data['Price']

# Split the 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)
```

```
# Specify columns for standardization and one-hot encoding
columns_to_standardize = ['Rooms', 'Number of Bathrooms', 'Size (sqft)', 'Walk Score', 'Transit Score', 'Bike Score',
'Time to Nearest Hospital', 'Time to Nearest Police Station', 'Time to Nearest Store',
'Time to Nearest Pharmacy']
columns_to_encode = ['Property Type']

# Define the preprocessing for numeric and categorical features
preprocessor = ColumnTransformer(
transformers=[
('num', StandardScaler(), columns_to_standardize),
('cat', OneHotEncoder(), columns_to_encode)
])

# Create a pipeline that combines the preprocessor with a model
model_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('regressor', RandomForestRegressor(n_estimators=100, random_state=42))])

# Evaluate the pipeline using cross-validation
scores = cross_val_score(model_pipeline, X_train, y_train, cv=5, scoring='neg_mean_squared_error')
print(f"Average MSE: {-scores.mean()}")

# Fit the pipeline on the training data
model_pipeline.fit(X_train, y_train)

# Make predictions on the test set
predictions = model_pipeline.predict(X_test)
```

XGBRegressor

- Script19

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from xgboost import XGBRegressor

# Adjusted R2 score function
def adjusted_r2_score(r2, n, p):
return 1 - (1 - r2) * (n - 1) / (n - p - 1)

# Load the data
data = pd.read_csv('filtered_data.csv') # Adjust the path accordingly
```

```
# Separate the dataset into features and target variable
X = data.drop('Price', axis=1)
y = data['Price']

# Define the preprocessing steps
numeric_features = ['Rooms', 'Number of Bathrooms', 'Size (sqft)', 'Walk Score', 'Transit Score', 'Bike Score',
'Time to Nearest Hospital', 'Time to Nearest Police Station', 'Time to Nearest Store',
'Time to Nearest Pharmacy']
categorical_features = ['Property Type']

preprocessor = ColumnTransformer(
transformers=[
('num', StandardScaler(), numeric_features),
('cat', OneHotEncoder(), categorical_features)
])

# Initialize the XGBoost regressor
xgb_regressor = XGBRegressor(objective='reg:squarederror', random_state=42)

# Create the pipeline
model_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('regressor', xgb_regressor)])

# Cross-validation setup
kf = KFold(n_splits=5, shuffle=True, random_state=42)
r2_scores = []
adjusted_r2_scores = []
mse_scores = []
mae_scores = []

for train_index, test_index in kf.split(X):
X_train, X_test = X.iloc[train_index], X.iloc[test_index]
y_train, y_test = y.iloc[train_index], y.iloc[test_index]
model_pipeline.fit(X_train, y_train)
y_pred = model_pipeline.predict(X_test)

r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
adjusted_r2 = adjusted_r2_score(r2, n=X_test.shape[0], p=X_test.shape[1])

r2_scores.append(r2)
adjusted_r2_scores.append(adjusted_r2)
mse_scores.append(mse)
mae_scores.append(mae)

# Display the mean of the evaluation metrics
```

```
print(f"Mean R2: {np.mean(r2_scores)}")
print(f"Mean Adjusted R2: {np.mean(adjusted_r2_scores)}")
print(f"Mean MSE: {np.mean(mse_scores)}")
print(f"Mean MAE: {np.mean(mae_scores)}")

# Extract feature importances
feature_importances = model_pipeline.named_steps['regressor'].feature_importances_
print("Feature Importances:", feature_importances)
```

XGBRegressor – Parameter tuning

- Script20

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from xgboost import XGBRegressor
import optuna

# Load the dataset
data = pd.read_csv('filtered_data.csv')

# Preprocessing
X = data.drop('Price', axis=1)
y = data['Price']
categorical_features = X.select_dtypes(include=['object']).columns

# Define a preprocessing pipeline
preprocessor = ColumnTransformer(
transformers=[
('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features),
],
remainder='passthrough'
)

# Splitting the data into training and test sets for final evaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

def objective(trial):
param = {
'n_estimators': trial.suggest_int('n_estimators', 50, 200),
'max_depth': trial.suggest_categorical('max_depth', [None, 3, 5, 7, 9]),
```

```
'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3),
'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
'subsample': trial.suggest_float('subsample', 0.5, 1.0),
'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0)
}

model = XGBRegressor(**param)
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('model', model)])
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, X_train, y_train, cv=kf, scoring='neg_mean_squared_error', n_jobs=-1)
rmse = np.mean([np.sqrt(-score) for score in scores])
return rmse

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50) # You can adjust the number of trials

# Best hyperparameters
print(f"Best hyperparameters: {study.best_trial.params}")

# Train the model with the best parameters
best_params = study.best_trial.params
best_model = XGBRegressor(**best_params)
best_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('model', best_model)])

best_pipeline.fit(X_train, y_train)

# Predict and evaluate on the test set
y_pred = best_pipeline.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
n = len(y_test)
p = X_test.shape[1]
adjusted_r2 = 1 - ((1 - r2) * (n - 1) / (n - p - 1))

# Performance Metrics
print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"R^2: {r2}")
print(f"Adjusted R^2: {adjusted_r2}")
```

Data Loading to MSSQL

Script 21 (Loading the extracted data on MSSQL server)

```
import pandas as pd
from sqlalchemy import create_engine, text

# Load the data from the CSV file
file_path = 'D:/Hackathon/filtered_data.csv'
df = pd.read_csv(file_path)

# Simplify column names: replace spaces and special characters with
underscores
df.columns = df.columns.str.replace(' ', '_').str.replace('(',
'').str.replace(')', '').str.replace('/', '_per_')

# Confirm new column names
print("New column names:", df.columns)

# Database connection string
username = 'r_khevaria'
password = 'islandENGLISH72'
host = 'dev.cs.smu.ca'
port = '3306'
database = 'r_khevaria'
engine =
create_engine(f'mysql+pymysql://{username}:{password}@{host}:{port}/{database
}')

# SQL command to create a new table with simplified column names
sql_command = """
CREATE TABLE IF NOT EXISTS rentListings (
    Price DECIMAL(10, 2),
    Property_Type VARCHAR(255),
    Rooms INT,
    Den_Included BOOLEAN,
    Number_of_Bathrooms INT,
    Latitude DECIMAL(9, 6),
    Longitude DECIMAL(9, 6),
    Size_sqft INT,
    Walk_Score INT,
    Transit_Score INT,
    Bike_Score INT,
    Time_to_Nearest_Hospital INT,
    Time_to_Nearest_Police_Station INT,
    Time_to_Nearest_Store INT,
    Time_to_Nearest_Pharmacy INT
);
"""

# Execute the SQL command to create the table if it doesn't exist
with engine.connect() as connection:
    connection.execute(text(sql_command))

# Load the data into the new database table
df.to_sql('rentListings', con=engine, if_exists='append', index=False)
```

```
print("Data loaded successfully into the 'rentListings' table.")
```