UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

BUDT 758T
Final Project Report
Spring 2023

Section 1: Team member names and contributions

Alexandra Dakhniuk - Business Cases and Data Visualization
Vinya Dengre - Model Development and Evaluation
Nishant Rajendrakumar Patel - Data Cleaning and Preparation
Chetana Shinde - Data Cleaning and Data Visualization
Maher Thakkar - Model Development and Evaluation

Section 2: Business Understanding

The Top Rating Score Airbnb model is a predictive model that uses machine learning algorithms to analyze Airbnb data and predict which properties are most likely to receive a high rating score from guests. The model takes into account a variety of factors such as property location (zip code, market city), sentiment analysis of listing details (description), amenities (amenities), monthly availability and yearly availability, host response rate, number of bedrooms per accommodate, connectivity to and from the list by considering various transportation options, type of room and property listed and many more to generate a prediction of the perfect rating score.

Airbnb hosts strive to earn and maintain perfect rating scores to increase their booking and in turn, maximize their revenue. The perfect rating score is a prime indicator of the property's quality, features, and user experience. In a majority of cases this might be one of the deciding factors for consumers in booking an Airbnb property.

Maintaining a perfect rating score is crucial for Airbnb hosts as it directly impacts their bookings and revenue. Guests often rely on the rating score as a deciding factor when choosing an Airbnb property. However, achieving and sustaining a perfect rating score is a challenging task that requires offering a high-quality property with competitive amenities and delivering outstanding hospitality services at an affordable cost.

Our predictive model helps hosts identify the factors that contribute to a perfect rating score. This is extremely valuable to individual hosts, Airbnb property management businesses and to Airbnb as a platform to improve or maintain a perfect rating for their listing.

The Airbnb perfect rating score model could benefit individual hosts who want to increase their bookings and revenue by improving their ratings. By analyzing the factors that contribute to a perfect rating score, hosts could identify areas of improvement and take additional actions to enhance their guests' experience. There are multiple factors that a guest might look at while booking an Airbnb however a perfect rating score is a major factor.

When a guest decides to plan their vacation stay through Airbnb platform, the first things they have to select are the location, the dates of bookings and the number of travelers. At this very point, hundreds or even thousands of listings are filtered out, so it directly affects your rankings as a host. Then, the user gets to use an interactive map, where they can choose a more specific location. As most users will intend to visit popular places, the precise location of your property will play a role here as well. At the same time the guests would also look at the proximity of the listing to other landmarks and attractions. Therefore it is important to consider what transportation options like bus, train, uber/lyft, bike shares are available.

Amenities are the next point on the list of things to look at for a guest. To improve the rating score it is vital for the host to include all the basic amenities and more. The more amenities you have, the more search visibility your listings get. For example bedroom comforts (which includes bed sheets, pillows, and hangers), bathroom comforts, family friendly amenities, security, wifi and entertainment and many more. Listing out all amenities and features can be a key to popular listing. There are also three booking features that the guest may select the same way: instant booking, self-check-in and free cancellation.

In addition, reliability is one of the key factors that guests consider when deciding whether to book an Airbnb property. A reliable host will communicate promptly(host_response_time) and clearly (host_response_rate) with guests, ensure that the property is clean and well-maintained, and be available to address any issues that may arise during the guest's stay. As well as host verification can play an important role in rating score as guests trust on listings increases when the host is verified.

For Airbnb as a platform, reliable hosts are also important because they help to build trust and confidence in the Airbnb brand. SuperHost listings are given higher visibility and are often displayed at the top of search results, making them more likely to be seen and considered by potential guests. SuperHost listings also receive corresponding marks and badges on their listing reviews and ratings. These marks, such as the SuperHost badge, indicate to guests that the host has achieved a high level of excellence and reliability. Furthermore, being a SuperHost positively impacts the rating score of an Airbnb listing. The rating score is a crucial aspect for both hosts and guests as it reflects the overall quality and satisfaction associated with a listing. By consistently meeting the SuperHost criteria, hosts can maintain a high rating score, which in turn helps attract more guests and build a positive reputation on the platform.
Similarly, the Airbnb description plays a crucial role in both attracting bookings and influencing the rating of a listing. The choice of words and the way the description is crafted can have a significant impact on how users perceive the property. It is essential to understand which words and phrases evoke positive or negative effects on potential guests. A well-written and compelling description that highlights the unique features, amenities, and benefits of the property can capture the attention of users and increase the likelihood of bookings. On the other hand, a poorly written or misleading description can lead to disappointment and negative reviews, affecting the rating of the listing. Hosts should strive to accurately portray their property, showcase its strengths, and provide transparent and detailed information to set proper guest expectations and create a positive experience from the moment users read the description.

There are many other factors that may highly influence the rating score of an Airbnb. Overall, the perfect rating score model could be a valuable tool for individual hosts and Airbnb property management businesses looking to improve their ratings and increase their revenue. By identifying the factors that contribute to a perfect rating score, hosts and property managers can take specific actions to enhance the guest experience, potentially leading to higher bookings, revenue, and customer satisfaction.

▪ Section 3: Data Understanding and Data Preparation

ID	FEATURE NAME	BRIEF DESCRIPTION	R CODE LINE NUMBERS
1	Cleaning_fee	This feature shows, how much, if any, is the cleaning fee the host charges. The R code checks if the cleaning_fee column in the full_data dataframe contains missing values (NA). If it does, it replaces those missing values with 0. If it doesn't, it leaves the values as they are.	81
2	Beds	This code fills in missing values for the "beds" feature in the "full_data" dataset with the mean value of the feature (excluding missing values). The "beds" feature represents the number of beds in the listing.	82
3	Bedrooms	This code is replacing missing values in the "bedrooms" feature with the median value of non-missing values. The "bedrooms" feature represents the number of bedrooms in the Airbnb listing.	83
4	Bathrooms	This code fills in missing values for the "bathrooms" feature with the median value of the non-missing data. It uses the ifelse() function to check if each value in the "bathrooms" column is missing (is.na()), and if so, replaces it with the median value calculated from the non-missing values (median(..., na.rm=TRUE)). The "bathrooms" feature in the data dictionary refers to the number of bathrooms in the listing.	84
5	Cancellation_policy	Cancellation_policy is a categorical variable in the full_data dataset that represents the cancellation policy of Airbnb listings. It has been processed in the previous code snippet to combine several similar policies into one category and convert it into a factor variable. The resulting variable has levels "strict" and other policies. This code is updating the values in the cancellation_policy column of the full_data data frame. If the value in that column is one of "strict", "super_strict_30", or "super_strict_60", then the value is changed to "strict". Otherwise, the value remains the same. This code appears to be enforcing a stricter cancellation policy across all listings in the dataset.	86-87
6	Bed_catrgory	This code creates a new column bed_category in full_data dataframe based on the values of the existing bed_type column. If the value in bed_type is 'Real Bed', then the corresponding value in bed_category will be 'bed', otherwise it will be 'other'. Finally, bed_category column is converted to a factor variable.	89-90

7	Has_cleaning_fee	The first two lines are parsing the cleaning_fee and price columns as numeric values. The third line is filling the missing values in cleaning_fee column with 0. Finally, the fourth line creates a new column called has_cleaning_fee which is set to "YES" if the cleaning fee is greater than 0, and "NO" otherwise. The has_cleaning_fee column is then converted to a factor variable.	91-100
8	Charges_for_extra	extra_people is a feature in the full_data dataset that represents the additional cost charged per guest after a certain number of guests specified in the listing. This code converts the extra_people column from a character string to a numeric data type by removing the $ sign and then creates a new column charges_for_extra to indicate whether the property charges extra fees for additional people. If the extra_people value is missing or zero, the charges_for_extra value is set to "NO", otherwise, it is set to "YES". Both charges_for_extra and extra_people columns are then converted to a factor data type.	102-104
9	Latitude	latitude is a numerical feature in the full_data dataset representing the latitude coordinates of the Airbnb listing location. It is measured in degrees and ranges from -90 to 90. Some missing values in the feature have been imputed using the na.approx function.	116-118
10	Market	The market feature represents the geographic location of the Airbnb listing. This variable categorizes the city/market of the listing. The code is re-categorizing the market variable so that any market with less than 300 listings is grouped into an "OTHER" category. This is done to reduce the number of categories and balance the distribution of the feature. Additionally, any missing values in the market variable are replaced with "OTHER". Finally, the market feature is converted to a factor variable for further analysis.	120-124
11	Price	Price is a numerical feature in the dataset representing the nightly rental price of the Airbnb listing in US dollars. The code replaces the dollar sign character in the price column of full_data with an empty string using gsub() function, then converts the resulting string into a numeric value using as.numeric() function.	127-129

		The purpose of this is to convert the price column from a character/string type to a numerical type.	
12	Monthly_price	This code updates the monthly_price column in the full_data dataframe. If the monthly_price is missing, it calculates the monthly price based on the price column and multiplies it by 30. Then, it removes any $ characters in the monthly_price column and replaces any, characters with an empty string. Finally, it converts the monthly_price column to numeric format.	130-134
13	Property_type	The code is recoding the feature "property_type" to group some of the categories into broader categories like apartment, hotel, house, condo, and other. The original categories are Apartment, House, Bed & Breakfast, Boutique hotel, Hostel, Serviced apartment, Bungalow, Condominium, Townhouse, and Loft.	136-140
14	License	The code first checks if the "license" column contains the word "pending" in any case (lower or upper), and if so, it replaces the value with "pending". Then, if the column contains a non-empty and non-NA value, it is replaced with "Has license". If the value is empty or NA, it is replaced with "Unknown". Finally, the code replaces any remaining NA values with "unknown" and converts the column to a factor. This creates a new factor variable with four levels: "Has license", "Unknown", "pending" and "unknown".	142-149
15	host_total_listings_count_mean.	This code calculates the mean value of the host_total_listings_count column in the full_data data frame and assigns it to the variable host_total_listings_count_mean. host_total_listings_count feature represents the number of listings the host has on Airbnb and has_total_listings_count appears to be a made-up feature name based on whether or not the host_total_listings_count is zero or greater than zero. Then, it replaces any missing values (NA) in the host_total_listings_count column with the calculated mean value. This is known as imputing missing data, and it is a common approach to handle missing values in a dataset. By replacing the missing values with the mean value of the column, the overall distribution and central tendency of the data are preserved.	167-168

16	Amenities	The code removes the characters \"{} from the amenities column in the full_data data frame using str_replace_all() function from the stringr package. This is done to remove unwanted characters from the amenities text. Then, it splits the resulting amenities column into a list of individual amenities using strsplit() function. Each element in the amenities column will be a list of amenities associated with the corresponding entry in the dataset.	107-108
17	Table_market	The code first creates a frequency table (table_market) for the market column in the full_data data frame. The code categorizes infrequent or missing values in the market column of the full_data data frame as "OTHER" and converts the column to a factor variable.	121-124
18	Instant_bookable	The code converts the values in the instant_bookable column of the full_data data frame. If the value is "TRUE", it is replaced with 1, indicating instant bookability. Otherwise, if the value is "FALSE" or any other value, it is replaced with 0, indicating non-instant bookability.	170
19	Is_Location_Exact	The code converts the values in the is_location_exact column of the full_data data frame. If the value is "TRUE", it is replaced with 1, indicating that the listing reports the exact location. Otherwise, if the value is "FALSE" or any other value, it is replaced with 0, indicating that the listing does not report the exact location.	171
20	Is_Business_Travel_Ready	The code converts the values in the is_business_travel_ready column of the full_data data frame. If the value is "TRUE", it is replaced with 1, indicating that the listing is available for business travel. Otherwise, if the value is "FALSE" or any other value, it is replaced with 0, indicating that the listing is not available for business travel.	172-174
21	Host_Response	The code assigns values to the host_response column of the full_data data frame based on the host_response_rate column. If the host_response_rate is missing, the corresponding host_response value is set to "MISSING". If the host_response_rate is "100%", the host_response value is set to "ALL". Otherwise, if the host_response_rate has any other value, the host_response value is set to "SOME". Finally, the host_response column is converted to a factor.	178-179

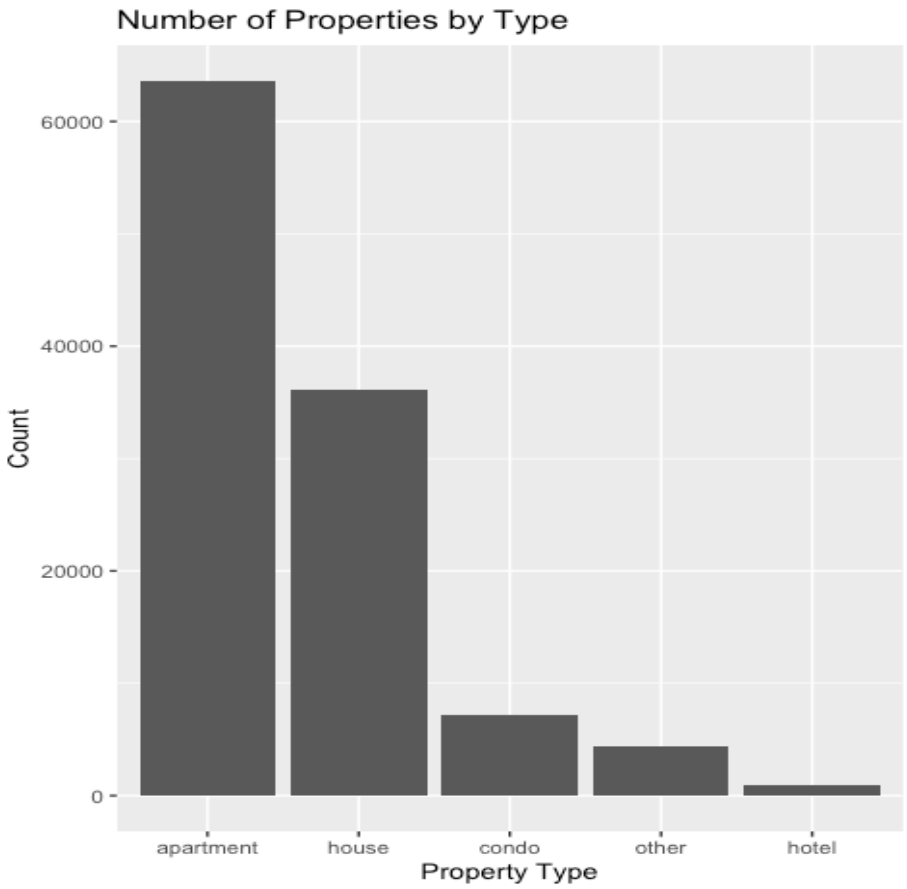
22	Host_Acceptance	The code assigns values to the host_acceptance column of the full_data data frame based on the host_acceptance_rate column. If the host_acceptance_rate is missing, the corresponding host_acceptance value is set to "MISSING". If the host_acceptance_rate is "100%", the host_acceptance value is set to "ALL". Otherwise, if the host_acceptance_rate has any other value, the host_acceptance value is set to "SOME". Finally, the host_acceptance column is converted to a factor.	181-182
23	Host_Is_Superhost	The code replaces missing values in the host_is_superhost column of the full_data data frame with FALSE. Then, it converts the values in the host_is_superhost column to 1 if they are "TRUE" and 0 otherwise.	185-186
24	Host_Identity_Verified	The code replaces missing values in the host_identity_verified column of the full_data data frame with FALSE. Then, it converts the values in the host_identity_verified column to 1 if they are "TRUE" and 0 otherwise.	189-190
25	Security_Deposit	The code removes dollar signs and commas from the security_deposit column of the full_data data frame using regular expressions. Then, it converts the values in the security_deposit column to numeric format.	213-214
26	Deposit_Mean	The code calculates the mean of the security_deposit column in the full_data data frame, excluding any missing values (NA). The result is stored in the variable deposit_mean. The code replaces missing values (NA) in the security_deposit column of the full_data data frame with the calculated mean value stored in the deposit_mean variable.	217-219
27	Guests_Included	The guests_included feature represents the number of guests included in the price of the rental. The code converts the guests_included column in the full_data data frame from its current data type to numeric.	223
28	Require_guest_phone_verification	After converting the require_guest_phone_verification feature to numeric values, the values in the require_guest_phone_verification column will be either 1 or 0. The value 1 indicates that guest phone verification is required, while the value 0 indicates that it is not required.	243
29	*Price_Per_Person*	*The code calculates the price_per_person by dividing the price column by the accommodates column in the full_data dataframe. It creates a new column called price_per_person to store the calculated values.*	246-249
30	*Ppp_ind*	*The code creates a new column called ppp_ind in the full_data dataframe. It assigns a value of 1 to the ppp_ind column if the corresponding price_per_person value is greater than the median*	*253-254*

		of price_per_person (ignoring any missing values), and assigns a value of 0 otherwise. Finally, the ppp_ind column is converted to a factor.	
31	*Has_minimum_nights*	*The code creates a new variable called has_min_nights in the full_data dataframe. It assigns the value "YES" to has_min_nights if the corresponding value in the minimum_nights column is greater than 1, and "NO" otherwise. The resulting has_min_nights variable is converted to a factor data type.*	*268*
32	Avg_Availability_30	The code calculates the average availability for each listing in the full_data dataframe based on the columns availability_30, availability_60, and availability_90. It computes the row-wise mean of these three columns and assigns the resulting values to a new variable called avg_availability_30.	*271*
33	Avg_Availability_365	The code calculates the average availability for each listing in the full_data dataframe based on the columns availability_365, availability_60, and availability_90. It computes the row-wise mean of these three columns and assigns the resulting values to a new variable called avg_availability_365.	*272*
34	Region	The code assigns a region to each listing in the full_data dataframe based on the state column. It uses nested if else statements to check the state value and assigns a corresponding region. The regions are categorized as follows: New England, Mid-Atlantic, Midwest ,Great Plains, Southeast, South Central, Southwest, Mountain, West, Other The assigned region is stored in a new variable called region.	*276-294*
35	Has_Bus	The code creates a new variable has_bus in the full_data dataframe. It checks the transit column for each listing and uses the grepl function with the pattern "bus" to check if the word "bus" is present in the transit description, ignoring case sensitivity. If a listing has the word "bus" in its transit description, the has_bus variable is assigned the value "bus". Otherwise, it is assigned an empty string.	*297*
36	Has_Train	The code creates a new variable has_train in the full_data dataframe. It checks the transit column for each listing and uses the grepl function with the pattern "train|subway|metro" to check if any of the words "train", "subway", or "metro" are present in the transit description, ignoring case sensitivity. If a listing has any of these words in its transit description, the has_train variable is assigned the value "train". Otherwise, it is assigned an empty string.	*298*

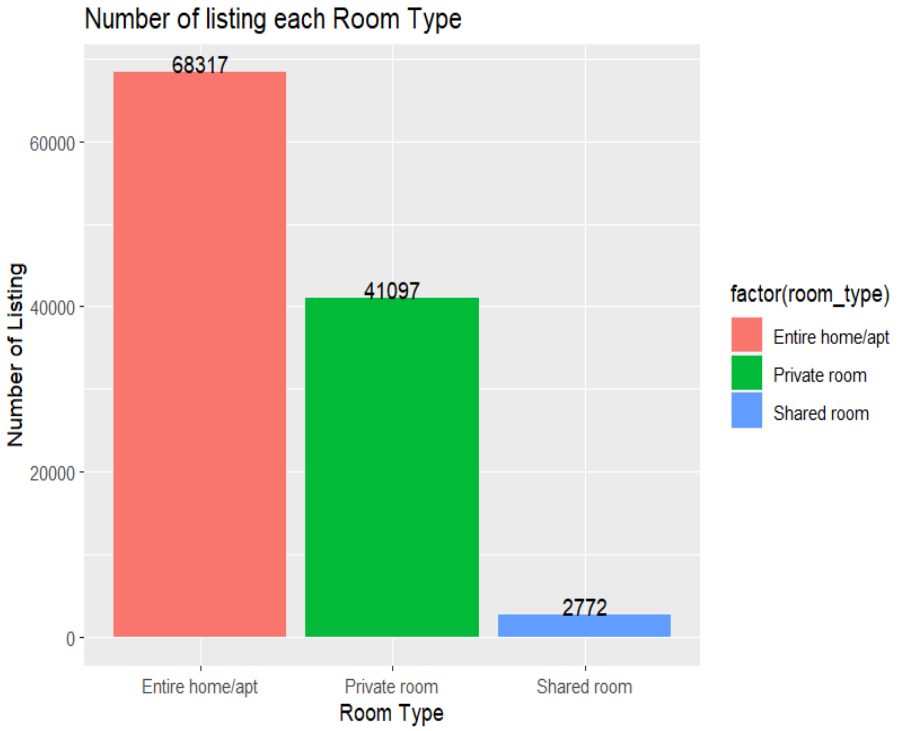
37	Has_Uber_Lyft	The code creates a new variable has_uber_lyft in the full_data dataframe. It checks the transit column for each listing and uses the grepl function with the pattern "uber|lyft" to check if either "uber" or "lyft" is present in the transit description, ignoring case sensitivity. If a listing has either "uber" or "lyft" mentioned in its transit description, the has_uber_lyft variable is assigned the value "uber". Otherwise, it is assigned an empty string.	299
38	Has_Bike_Share	The code creates a new variable has_bike_share in the full_data dataframe. It checks the transit column for each listing and uses the grepl function with the pattern "bike share" to check if "bike share" is present in the transit description, ignoring case sensitivity. If a listing mentions "bike share" in its transit description, the has_bike_share variable is assigned the value "bike". Otherwise, it is assigned an empty string.	300
39	Has_Streetcar	The code creates a new variable has_streetcar in the full_data dataframe. It checks the transit column for each listing and uses the grepl function with the pattern "streetcar" to check if "streetcar" is present in the transit description, ignoring case sensitivity. If a listing mentions "streetcar" in its transit description, the has_streetcar variable is assigned the value "streetcar". Otherwise, it is assigned an empty string.	301
40	House_Rules_Pet	The code creates a new variable house_rules_pet in the full_data dataframe. It checks the house_rules column for each listing and uses the grepl function with the pattern "pets" to check if "pets" is mentioned in the house rules, ignoring case sensitivity. If a listing includes any mention of pets in its house rules, the house_rules_pet variable is assigned the value TRUE, otherwise FALSE.	314
41	House_Rules_Smoking	The code creates a new variable house_rules_smoking in the full_data dataframe. It checks the house_rules column for each listing and uses the grepl function with the pattern "smoking" to check if "smoking" is mentioned in the house rules, ignoring case sensitivity. If a listing includes any mention of smoking in its house rules, the house_rules_smoking variable is assigned the value TRUE, otherwise FALSE.	315
42	House_Rules_Parties	The code creates a new variable house_rules_parties in the full_data dataframe. It checks the house_rules column for each listing and uses the grepl function with the pattern "parties" to check if "parties" is mentioned in the house rules, ignoring case sensitivity. If a listing includes any mention of parties in its house rules, the house_rules_parties variable is assigned the value TRUE, otherwise FALSE.	316

43	Host_Name_Frequency	The code creates a new variable host_name_frequency in the full_data dataframe. It calculates the frequency of each unique host_name by grouping the rows based on host_name and applying the length function. The ave function is used to apply the length function to each group and assign the corresponding frequency to each row in the host_name_frequency variable.	*317*
44	Host_Has_Profile_Pic	The code assigns the value "Unknown" to the host_has_profile_pic variable for rows where it is NA (missing). It is a way to handle missing values and replace them with a specific value.	*327*
45	Zipcode	The code modifies the zipcode column in the full_data dataset. It handles missing values by replacing them with the mean value from another column. It then applies formatting rules to ensure consistent and standard zip code formats. The modified zip code data is printed.	195-210

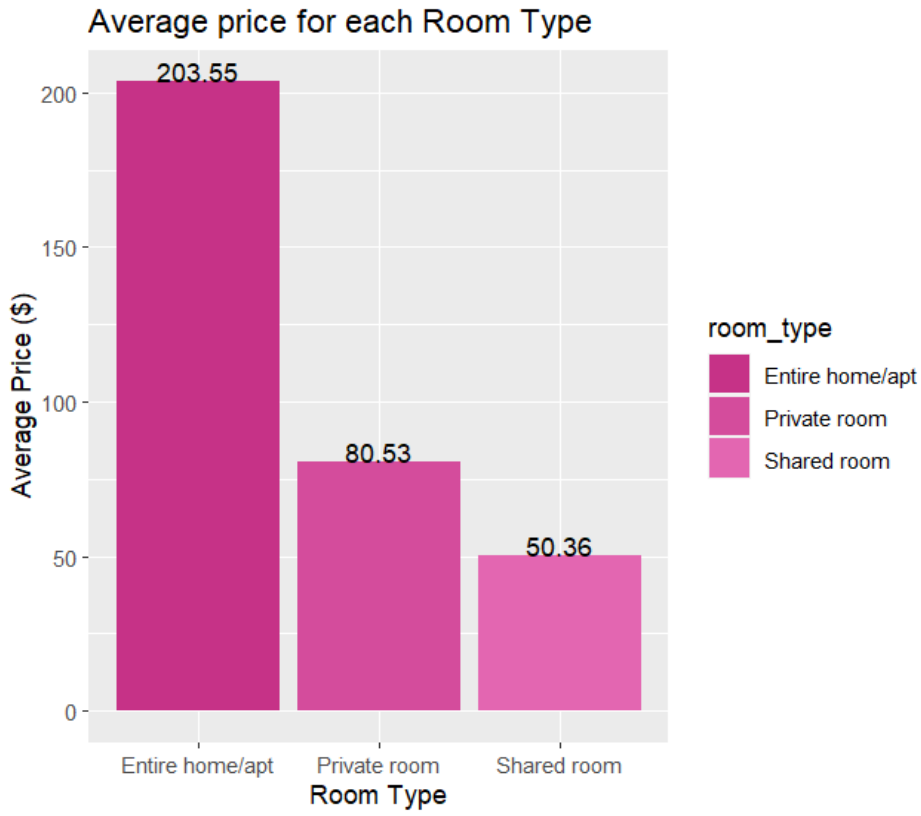
1) Graphs or tables demonstrating useful or interesting insights regarding features in the dataset.



From the bar chart above we can conclude that “apartment” and “house” are the most popular property types on the Airbnb platform, among other types are condo, hotel etc.

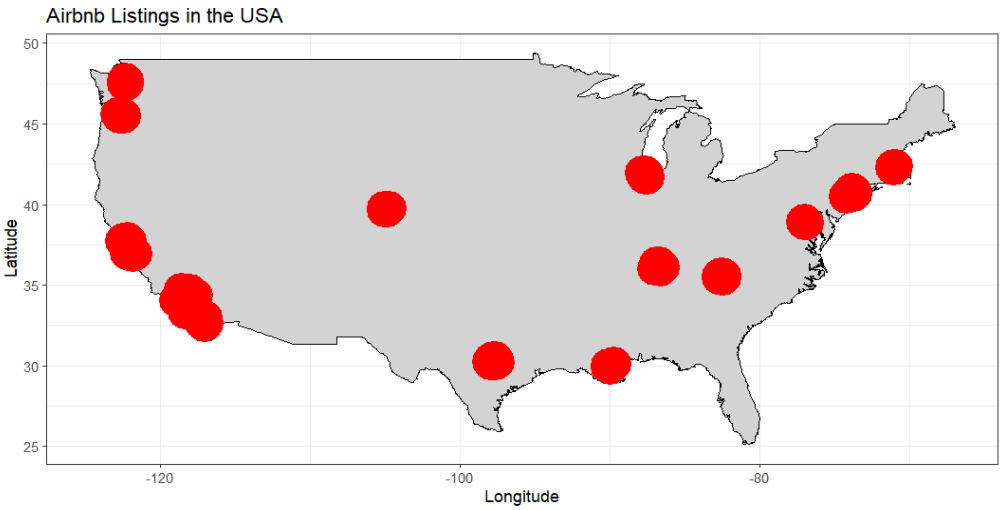


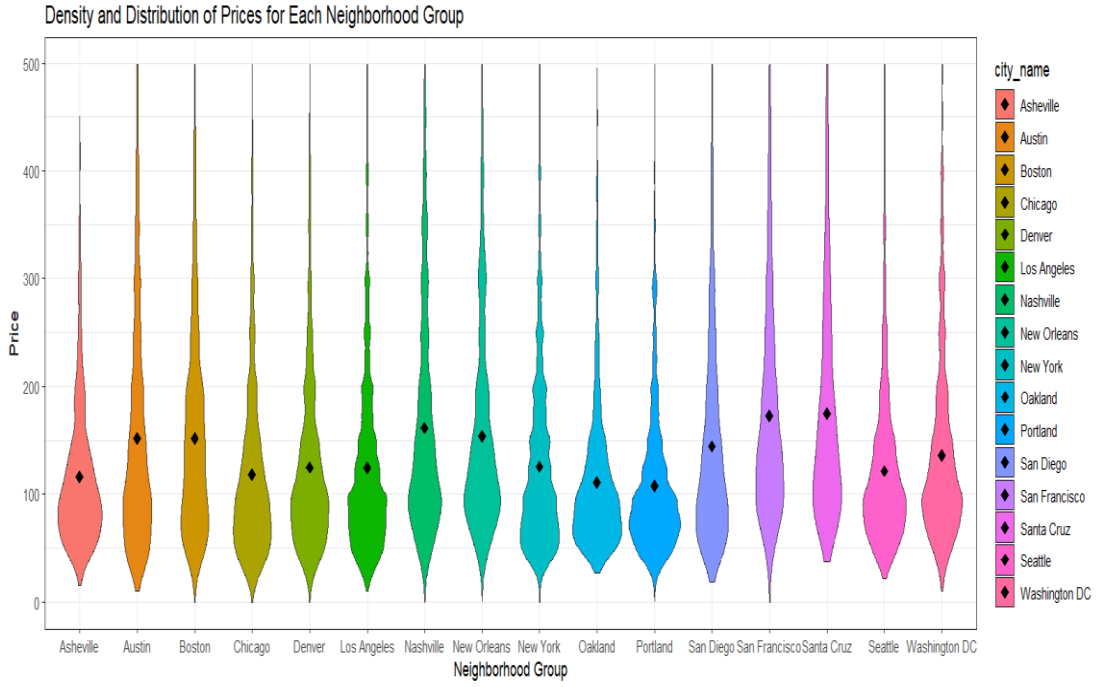
The most number of listings under Airbnb is for an entire house or apartment. From the chart, it is evident that there is a small percentage of shared rooms. Since the demographic that would prefer to book a Shared room Airbnb is very less, it is fair that it has the lowest number of listings.



The bar chart above shows the average price listed on Airbnb based on room type. The rental of an entire home or apartment charges the highest price out of the three. Shared rooms on the other hand are the cheapest at $50.36. At such a low cost, Airbnb could market their rooms as affordable especially for solo travelers. Solo travelers looking for a place to crash at night can consider shared rooms listed on Airbnb for cheap stay as well as an opportunity to make local friends.

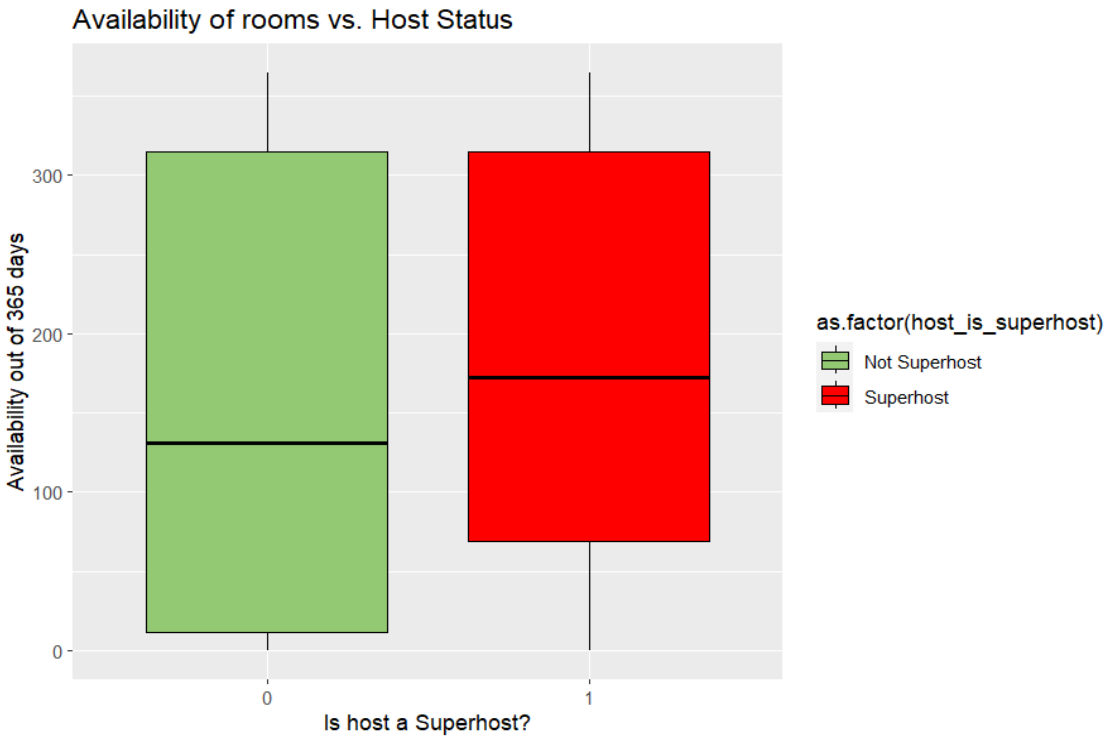
From the map, it becomes evident that a notable concentration of Airbnbs exists along the Eastern and Western coasts of the USA. Particularly, California stands out with a significantly higher number of Airbnb listings. This pattern may be attributed to various factors, such as the abundance of tourist attractions or the presence of numerous offices and businesses in this state.

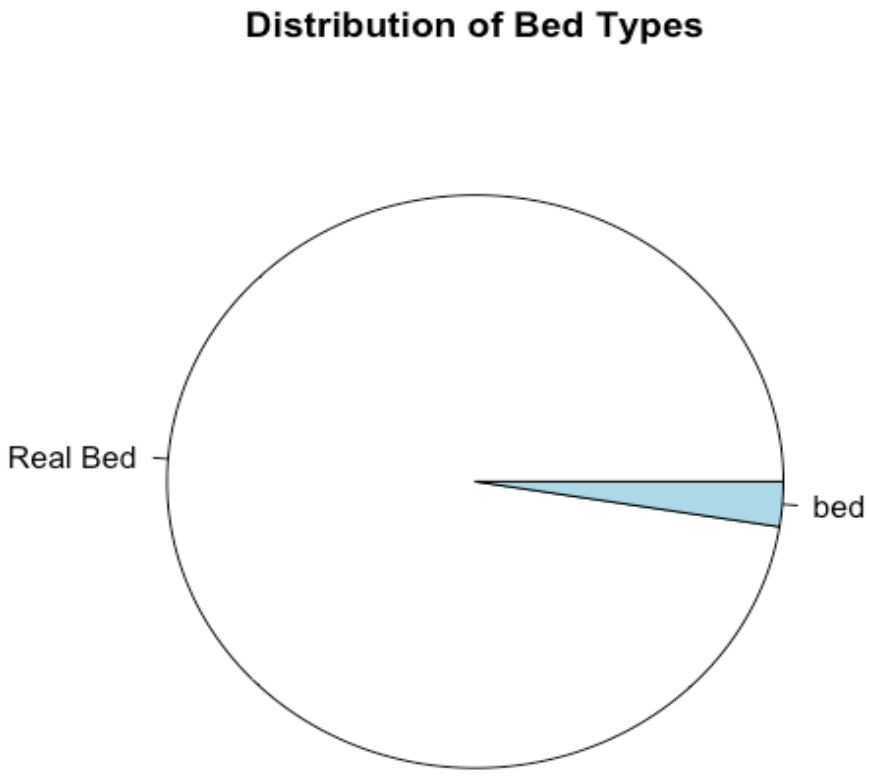




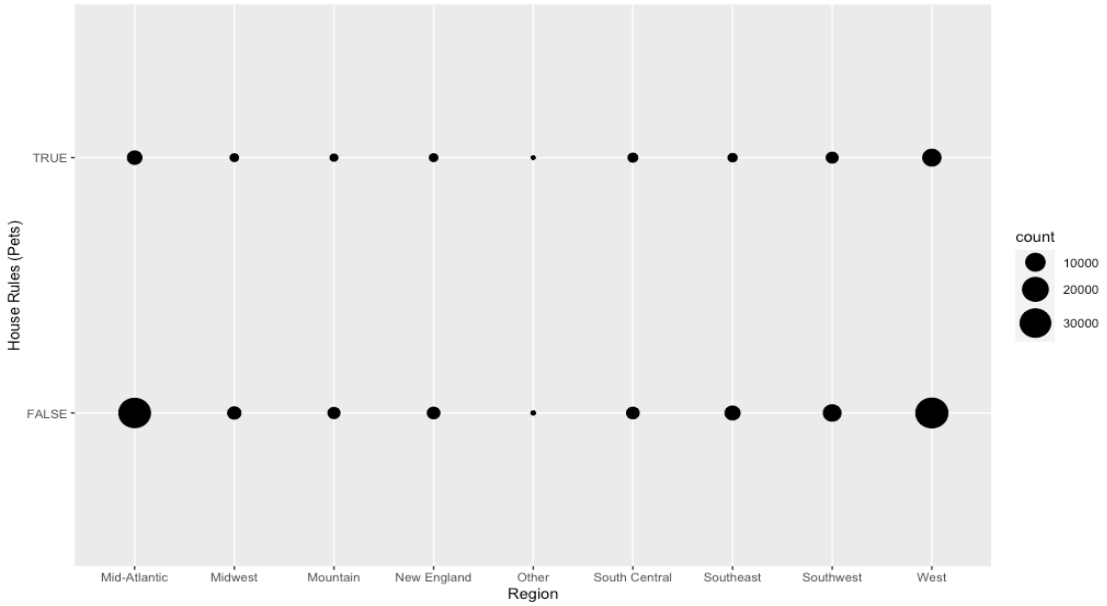
From the violin plot we can definitely observe a couple of things about distribution of prices for Airbnb by cities. First, we can state that San Francisco has the highest range of prices for the listings with $175 price as an average observation, followed by Boston with $150 per night. Oakland and Portland are the cheapest of them all with $110 per night. This distribution and density of prices were completely expected; for example, as it is no secret that San Francisco is one of the most expensive places in the world to live in, where Oakland on other hand appears to have lower standards of living.

The box plot of hosts with and without superhost status against the total number of days the listing is available for during the year. Out of the 365 days, you can see that the median difference between host status is about 50 days, where superhost are likely to have less days booked. From the box plot, both superhost and non-superhost have very different lower quartiles but similar upper quartiles. Airbnb could possibly look into ways for hosts that are superhost to increase availability of bookings.



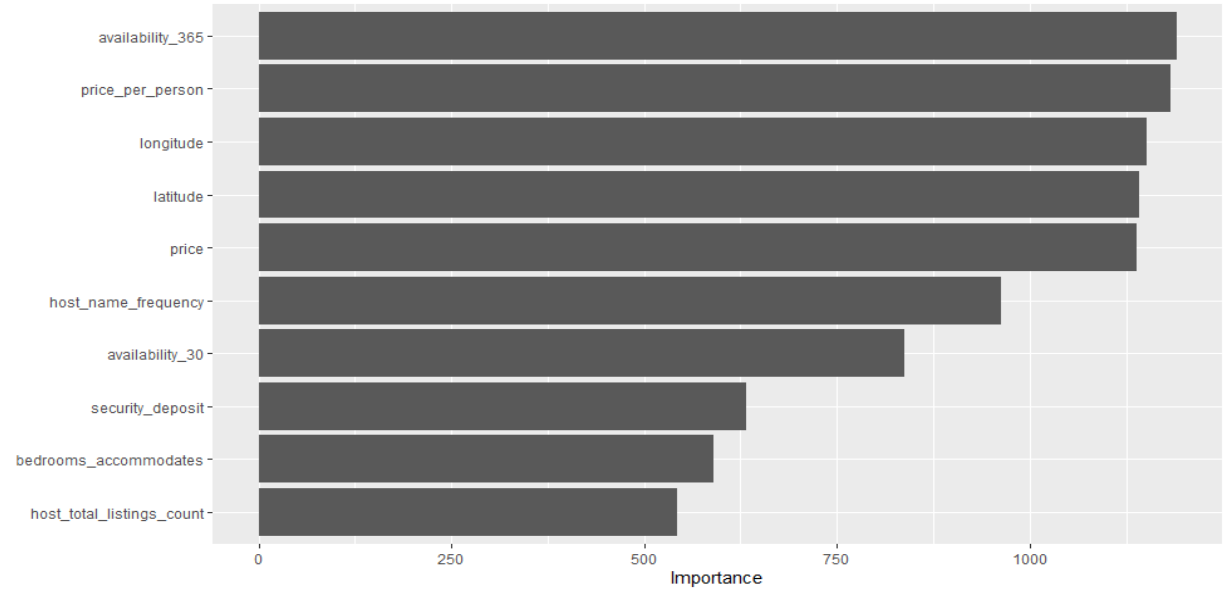


Majority of airbnb listings include “real_bed” in its description, while the word “bed” is only mentioned in a very small number of property descriptions.



From the scatterplot above we can conclude that from the small number of the airbnb properties that allow pets are mostly located in the West or Mid-Atlantic.

VIP for Random Forest



From the graph above we can see the importance of various features included.

2) **(optional) Any additional comments about the data or the steps you undertook for data preparation.**

Document Term Matrix: several important preprocessing steps for text data are performed before creating a document-term matrix **(DTM)** for machine learning. First, a custom function called **cleaning_tokenizer** is defined. This function takes a vector of text data and performs various cleaning operations on it. It **removes**

numbers, punctuation, and stems the words using the Porter stemming algorithm. Additionally, it **tokenizes** the text into individual words. The **itoken function** from the text2vec package is then used to create an iterator over the text data. Three iterators **(it_train, it_train_1, and it_train_2**) are created for different columns of the full_data dataframe. Next, **vocabulary** is created using the create_vocabulary function for each iterator. The vocabularies are then pruned using the **prune_vocabulary** function to remove infrequent terms. This helps reduce the dimensionality of the DTM. Afterwards, a vocab_vectorizer is created using the pruned vocabularies. This vectorizer will be used to convert the text data into a DTM. The **create_dtm function** is used to convert the text data from each iterator into a DTM based on the corresponding vectorizer. To integrate the DTM with other features, the **sparse matrices are converted to data frames (dtm_train_df, dtm_train_df_1, dtm_train_df_2)**. Then, these data frames are combined with the data_dummies dataframe using **cbind**. This combines the extracted text features with other features encoded as dummy variables. Finally, the resulting combined dataframe **data_dummies_combined_final is split back into the original training and testing sets (trainxy_new and testxy_new, respectively)**. Overall, this process prepares the text data by cleaning and tokenizing it, creating a vocabulary, and transforming it into a DTM. The resulting DTM can then be used for machine learning tasks such as classification or regression.

Sentiment Analysis: We conducted sentiment analysis on the descriptions in the full_data dataset to determine whether they had a positive or negative sentiment. We preprocessed the descriptions by **removing numbers, punctuation, and stopwords,** and then **tokenized** them into individual words. From the tokenized descriptions, we **created a vocabulary of unique words,** limiting it to 500 terms for simplicity. Using the vocabulary, we converted the tokenized descriptions into a **binary Document-Term Matrix (DTM)** where presence was represented as 1 and absence as 0. We utilized the **AFINN sentiment lexicon**, which contains words with associated sentiment scores, to perform the sentiment analysis. By comparing the words in the **DTM** with the sentiment lists from the **AFINN lexicon**, we counted the occurrences of positive and negative sentiment words in each document, allowing us to determine the sentiment polarity. The sentiment counts were transformed into a data frame and the sentiment scores were converted into factors **('P' for positive and 'N' for negative)** based on the comparison between positive and negative counts. This analysis provided insights into the **sentiment polarity of each document** in the full_data dataset, and the resulting sentiment scores can be used as additional features to potentially enhance accuracy and performance in our models.

External Data source Usage:

1. We extracted data from https://www.census.gov/ which had columns named as zip code, race, population, sex, maxAge and minAge. Further performed data cleaning using **unite to merge columns** race, sex, minAge, maxAge and further used pivot wider to clean the dataset on zip code level.
2. Post that we **merged the columns using join** on zip code columns.
3. Further we did a **sum of all age groups** to get the male and female population. Further **normalized** it to get the proper values. External dataset helped us get more data such as population and gender related along with race.
4. It helped **increase our performance** by 0.15 which was almost similar to our original predictions.

▪ Section 4: Evaluation and Modeling

1) Include a short (one-paragraph) description of the "winning" model, the variables included in the model, your estimated training and generalization performance, and how you decided that it was the winning model. Also list the line numbers in your R code where you generated the final predictions that you submitted for the contest.

1. The **"winning" mode**l in this context is an **XGBoost model** for regression. The model is trained to **predict the "perfect_rating_score" variable**. XGBoost uses gradient boosting and regularization to create accurate predictive models. It builds an ensemble of weak models, typically decision trees, to correct errors and improve performance. XGBoost is known for its high accuracy, scalability, and efficiency, making it popular in various domains.
2. The variables included in the model are all predictor variables except for the "perfect_rating_score" itself which are "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind","property_type", "bed_category", "bathrooms", "charges_for_extra" , "host_acceptance", "host_response", "market", "host_is_superhost", "availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights","host_total_listings_count".

3. We **converted these variables into dummies**. We did **create many new features** from different existing columns and text fields and also used **interaction terms**. We also **created DTM** for columns such as amenities, transit and did **sentiment analysis** for the description column and merged all the features and DTM to input all the features. Total we had **914 resulting variables after converting variables to dummy and merging them with dtms created.**

4. The estimated training and generalization performance of the model can be evaluated using cross-validation. In the provided code, the model is trained using the full dataset after performing **k-fold cross-validation**. The **root mean squared error (RMSE)** is used as the **evaluation metric during cross-validation**. The final model's performance is assessed based on the best iteration found through cross-validation. The winning model is determined based on its performance in terms of the evaluation metric, which is RMSE in this case. The cross-validation process helps to assess the model's performance on unseen data and **select the best-performing model based on the evaluation metric. Training Performance:** The training performance has **accuracy of 73.5%** (much higher than the baseline model) **, TPR as 31.9% and FPR as 8.92%.**

5. **We also did hypermater tuning on these variables: nrounds** = 1000, **nfold** = 8, **early_stopping_rounds** = 10 are a few hyperparameters we tuned. These values gave us the best results.

6. The **final predictions** for the contest are generated using the trained XGBoost model. The **test dataset (testxy_new) is prepared**, and the model is used to predict the "perfect_rating_scoreYES" variable for this dataset**. The predictions are then converted into binary class labels using a threshold of 0.4665**. The resulting predictions are stored in the variable y_pred_binary_final.

7. **The line numbers in the provided code where the final predictions are generated are:**

```
library(caret)
testx_d <- testxy_new[, -which(names(testxy_new) == "perfect_rating_scoreYES")]
x_test_final <- as.matrix(testx_d)
dtest_final <- xgb.DMatrix(data = x_test_final)
y_pred_final <- predict(model, newdata = dtest_final, type = "response")
y_pred_binary_final <- ifelse(y_pred_final >= 0.4665, "YES","NO")
# Print the predicted binary class labels
print(y_pred_binary_final)
```

y_pred_binary_final <- ifelse(is.na(y_pred_binary_final),"NO", y_pred_binary_final)

summary(y_pred_binary_final)

accuracy_perfect_xg <- sum(y_pred_binary_final== data_selected$perfect_rating_score) / nrow(data_selected)

#this code creates sample outputs in the correct format

write.table(y_pred_binary_final,"perfect_rating_score_group19_submission.csv",row.names = FALSE)

The predicted binary class labels (y_pred_binary_final) are printed, and the final predictions are written to a CSV file named "perfect_rating_score_group19_submission.csv" using the write.table() function.

2) For each type of model that you try, please list:

a) The type of model (i.e. model family – for instance: logistic regression, decision trees, etc.).

b) The R function and/or library used.

c) Estimated training and generalization performance for this model.

d) How you estimated the generalization performance for this model. This should include the methodology used (i.e. simple train/validation split, cross validation, nested holdout) as well as the specific parameters of the validation setup (i.e. how much data, how many folds, etc.).

e) The best-performing set of features that you used in the model (you may also optionally note other features that you tried but didn't include in the final feature set).

f) The line numbers in your R code where you trained the model and estimated its generalization performance.

g) (Required to receive at least 80 points): Any hyperparameters that you tuned and a list of the values that you tried.

h) (Required to receive at least 80 points): Any fitting curves that you created for this model.

1. **<u>Logistic regression:</u>**

a) Description: Logistic regression is a popular statistical model utilized for solving binary classification problems. It explores the association between one or more independent variables and a binary outcome variable by estimating the probabilities of the outcome variable belonging to a specific class. By fitting a logistic function to the data, the model predicts the likelihood of an event occurring or not based on the input features.

b) R Function/Library: The R function used for logistic regression is glm() from the base R library.

c) Training Performance: The training performance has accuracy of 72.3% , TPR as 28.5% and FPR as 9.25% keeping the cut off as 0.47.

Generalization Performance: The generalization performance has accuracy of 71% while testing on the testdata.

d) Training Performance: The model is trained using the glm() function with the specified formula and data.

Generalization Performance: The model's predictions are compared against the baselines classifier. The accuracy on the test dataset is compared with the baseline classifier accuracy which is 70.5%.

Methodology for Generalization Performance: The model uses a simple train/validation split approach. The dataset is split into training and validation subsets. The logistic regression model

is trained on the training subset, and its predictions are evaluated on the validation subset to estimate the generalization performance on the test dataset.

e) The features that we used are : "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind", "property_type", "bed_category", "bathrooms", "charges_for_extra", "host_acceptance", "host_response", "market", "host_is_superhost","availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights","host_total_listings_count". We did create many new features from different existing columns and text fields and also used interaction terms. We also created DTM for columns such as amenities, transit and did sentiment analysis for the description column and merged all the features and DTM to input all the features. Total we had 914 resulting variables after converting variables to dummy and merging them with dtms created.

f) <u>Lines of Code:</u> (R code is from 571-621)

- `logreg <- glm(train_xy_train$perfect_rating_scoreYES ~ ., data = train_xy_train, family = binomial)`
  Description: This line trains a logistic regression model. It uses the `glm()` function to fit the model, where the dependent variable is `perfect_rating_scoreYES` from the `train_xy_train` dataset. The formula `train_xy_train$perfect_rating_scoreYES ~ .` indicates that all other variables in the `train_xy_train` dataset are used as independent variables. The family parameter is set to "binomial" since logistic regression is used for binary classification.

- `valid_preds <- predict(logreg, newdata = train_xy_val)`
  Description: This line generates predictions for the validation dataset `train_xy_val` using the trained logistic regression model. The `predict()` function is used with the `newdata` parameter set to `train_xy_val`.

- `logit_preds <- ifelse(predict(logreg, newdata = train_xy_val, type = "response") >= 0.4, 1, 0)`
  Description: This line applies a threshold of 0.44 to the predicted probabilities from the logistic regression model. If the predicted probability is greater than or equal to 0.44, it is classified as 1; otherwise, it is classified as 0. The `predict()` function is used with the `newdata` parameter set to `train_xy_val` and the `type` parameter set to "response" to obtain the predicted probabilities.

- `actual_pred_lm = as.factor(train_xy_val$perfect_rating_score)`
  Description: This line converts the actual values of the dependent variable `perfect_rating_score` in the validation dataset (`train_xy_val`) into a factor variable. This is done to compare the predictions with the actual values.

- `class_valid_lg = factor(logit_preds, level = unique(actual_pred_lm))`
  Description: This line converts the predicted classifications (`logit_preds`) into a factor variable. The levels of the factor variable are set to the unique values of the actual predictions (`actual_pred_lm`). This is done to ensure consistency in the levels of the predicted and actual variables for calculating the confusion matrix.

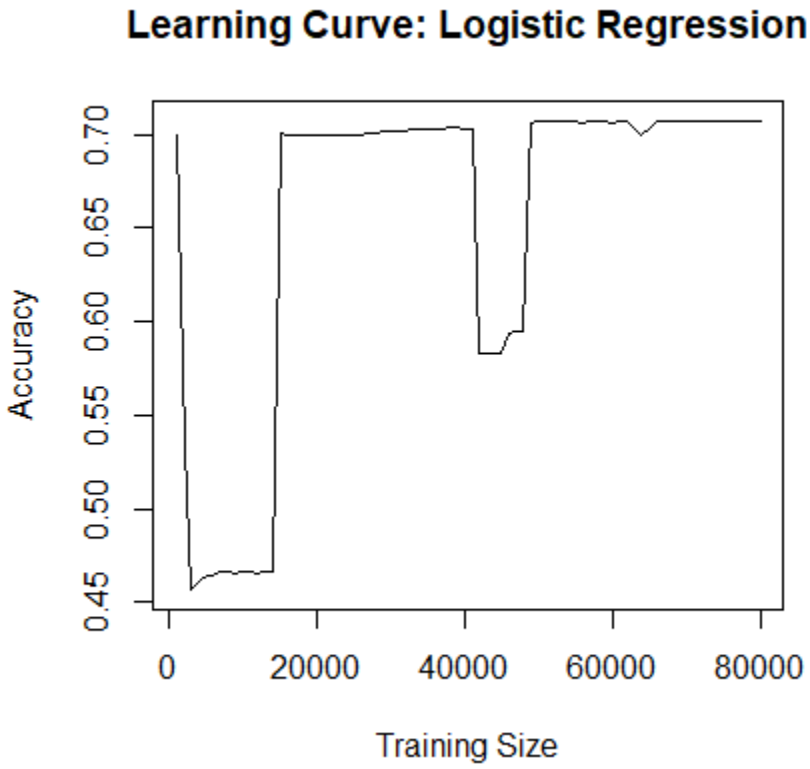
- `logit_cm <- confusionMatrix(data = class_valid_lg, reference = actual_pred_lm)`
  Description: This line calculates the confusion matrix using the predicted classifications (`class_valid_lg`) and the actual values (`actual_pred_lm`). The `confusionMatrix()` function from the `caret` library is used for this purpose.

- `accuracy <- logit_cm$overall['Accuracy']`
  Description: This line extracts the accuracy metric from the confusion matrix. It retrieves the overall accuracy value from the `logit_cm` object.

- `class_valid_lg <- factor(logit_preds, level = unique(actual_pred_lm))`
  Description: This line converts the predicted classifications (`logit_preds`) into a factor variable.

g) Hypertuning: No hyperparameter tuning was required.

h) Learning Curve



This graph also shows that accuracy increases as the training size increases. It also gives less accuracy than XGBOOST for us so helped us in comparing with the same train and validation split.

2. **<u>XGBoost:</u>**

a) Description: XGBoost uses gradient boosting and regularization to create accurate predictive models. It builds an ensemble of weak models, typically decision trees, to correct errors and improve

performance. XGBoost is known for its high accuracy, scalability, and efficiency, making it popular in various domains.

b) R function/ library: The R function used for XGBoost regression is xgboost() R library.

c) Training Performance: The training performance has **accuracy of 73.5%** , **TPR as 31.9%** and **FPR as 8.92%**.

Generalization Performance: The generalization performance has accuracy of 73.3% while testing on the test data and this is better than the baseline classifier as well. This has been the best generalization performance for us.

d) Methodology for Training and Generalization Performance:

Training and validation performance metrics: The training and generalization performance of the XGBoost model is evaluated using cross-validation. The mean squared error (RMSE) is used as the evaluation metric. The code calculates the cross-validated RMSE for different parameter settings and selects the best iteration based on early stopping.

Methodology for generalization performance estimation: K-fold cross-validation with 8 folds is used to estimate the generalization performance of the model. The dataset is split into 8 subsets, and the model is trained and evaluated 8 times, with each subset serving as the validation set once while the rest are used for training. The average performance across the folds is reported. The model's predictions are compared against the baselines classifier. The accuracy on the test dataset is compared with the baseline classifier accuracy which is 70.5%.

e) The features that we used are : "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind", "property_type", "bed_category", "bathrooms", "charges_for_extra", "host_acceptance", "host_response", "market", "host_is_superhost","availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights","host_total_listings_count". We did create many new features from different existing columns and text fields and also used interaction terms. We also created DTM for columns such as amenities, transit and did sentiment analysis for the description column and merged all the features and DTM to input all the features. Total we had 914 resulting variables after converting variables to dummy and merging them with dtms created.

f) Line of Code: (R code is from 624-719)

- cv_result <- xgb.cv(...)
  This line performs k-fold cross-validation using the xgb.cv() function. It takes the specified parameters (params), the training data (dtrain), and other arguments such as the number of rounds (nrounds) and number of folds (nfold). It trains and evaluates the XGBoost model on the training data using cross-validation.

- cv_error <- tail(cv_result$evaluation_log[, 1], 1)
  This line extracts the final cross-validation error from the evaluation log of the cross-validation result. The evaluation log contains information about the performance of the model at each iteration. Here, cv_result$evaluation_log[, 1] retrieves the values of the first column of the evaluation log, which represents the error metric (RMSE in this case). tail(..., 1) selects the last value from the column, which corresponds to the final error after all rounds of cross-validation.

- cv_stddev <- tail(cv_result$evaluation_log[, 2], 1)
  This line extracts the final cross-validation standard deviation from the evaluation log. Similar to the previous line, cv_result$evaluation_log[, 2] retrieves the values of the second column of the evaluation log, which represents the standard deviation of the error metric. tail(..., 1) selects the last value from the column, which corresponds to the final standard deviation after cross-validation.

- model <- xgb.train(...)
  This line trains the final XGBoost model using the full dataset. It uses the xgb.train() function and takes the specified parameters (params), the training data (dtrain), and the best iteration determined by cross-validation (cv_result$best_iteration). The best iteration is the number of rounds at which the model had the lowest error during cross-validation.

- y_pred <- predict(model, newdata = dtest)
  This line makes predictions on new data using the trained XGBoost model. It uses the predict() function with the trained model (model) and the new data (dtest). The predictions (y_pred) will be numeric values representing the target variable.

- y_pred_binary_final <- ifelse(y_pred_final >= 0.475, "YES", "NO")
  This line converts the predicted probabilities (y_pred_final) into binary predictions ("YES" or "NO") using a threshold of 0.459. If the predicted probability is equal to or greater than 0.459, it is classified as "YES"; otherwise, it is classified as "NO". The ifelse() function is used for this thresholding operation.

- accuracy_perfect_xg <- sum(y_pred_binary_final == data_selected$perfect_rating_score) / nrow(data_selected)
  This line calculates the accuracy of the XGBoost model by comparing the predicted values (y_pred_binary_final) with the actual values (data_selected$perfect_rating_score). The sum(... == ...) counts the number of matching predictions and nrow(data_selected) gives the total number of samples. The ratio of matching predictions to the total number of samples gives the accuracy of the model.

i) Hyperparameter:

nrounds = 1000, **nfold** = 8, **early_stopping_rounds** = 10 are a few hyperparameters we tuned. These values gave us the best results.

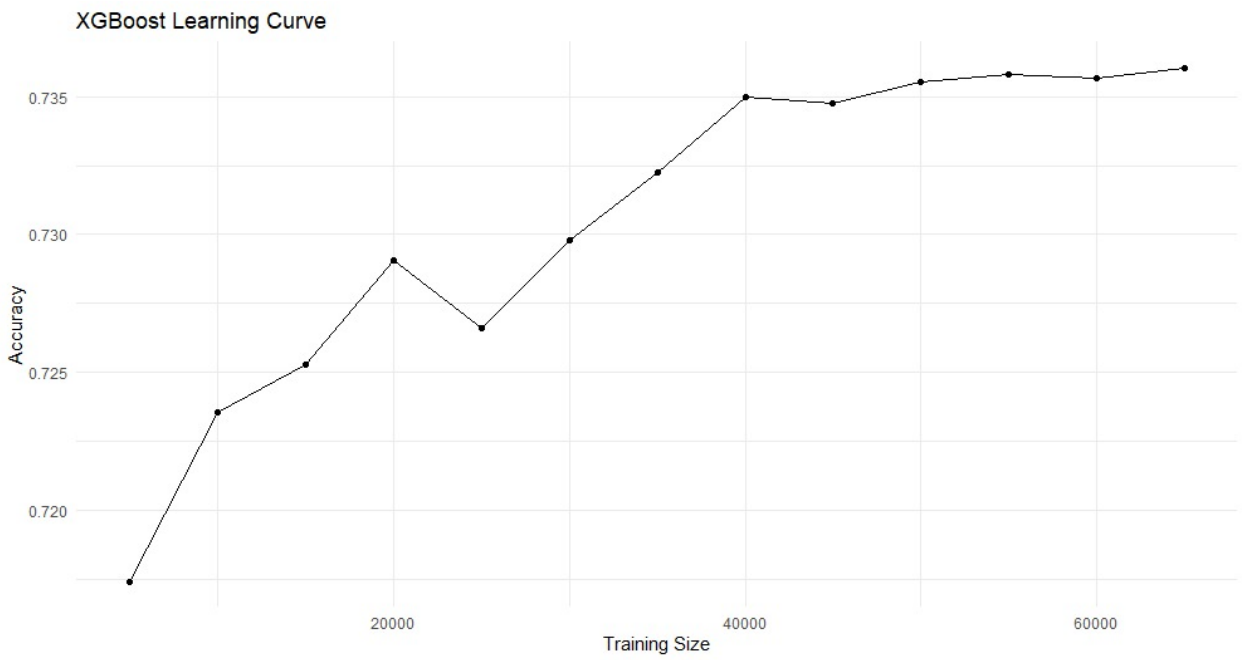
We tried nfolds ranging from 6 to 10 and early stopping rounds as 7-12 as well as nrounds as 600-1200. Explaining more in detail as below:

nrounds = 1000: This parameter specifies the maximum number of boosting rounds or iterations to perform during the training process. In each round, a weak learner (typically a decision tree) is added to the ensemble.

nfold = 8: This parameter specifies the number of folds to use in cross-validation. Cross-validation is a technique for estimating model performance by splitting the data into multiple subsets (folds) and training/evaluating the model on different combinations of these subsets.

early stopping rounds = 10: This parameter is used to implement early stopping during the training process. Early stopping allows the training to stop early if the performance on the validation set does not improve for a certain number of rounds

j) Learning Curve



This is the learning curve that we made for our winning model, it shows that the accuracy increased as the training size increased. This helped us know that we can split 70% training and 30% validation to get best results. Increasing more size was decreasing the accuracy.

3. **GLMNET:**

a) Description: Glmnet is an R package that implements regularized generalized linear models using the elastic net regularization method. It combines the advantages of both ridge and lasso regression by applying a penalty term to the model's coefficients. This penalty encourages sparsity in the coefficient estimates, effectively performing feature selection.

b) R function/ library: The R function used for GLMNET regression is glmnet and caret R library and cv.glmnet() to train model

c) Training Performance: The training performance has accuracy of 72.25% , TPR as 29.2% and FPR as 9.62%.

Generalization Performance: The generalization performance has accuracy of 72.1% while testing on the test data and this is better than the baseline classifier as well. Though, this is not our winning model. But we tried to compare different models for best accuracy and FPR TPR.

d) Methodology of Training and Generalization Performance:

Methodology of Training Performance: The generalization performance of the model is estimated using a holdout validation approach. The dataset is split into training and validation sets using the train_xy_train and train_xy_val data frames. The model is trained on the training set (x_train and y_train), and the resulting model is used to make predictions on the validation set (x_val). The performance metrics are then calculated based on the predicted values (y_pred_binary) and the actual values (actual_pred_lm).

Methodology for generalization performance estimation: The generalization performance is evaluated using the accuracy metric. The accuracy metric represents the overall correctness of the model's predictions. And it has been further compared with the baseline classifier to know the model's performance.

e) The features that we used are : "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind", "property_type", "bed_category", "bathrooms", "charges_for_extra", "host_acceptance", "host_response", "market", "host_is_superhost","availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights", "host_total_listings_count". We did create many new features from different existing columns and text fields and also used interaction terms. We also created DTM for columns such as amenities, transit and did sentiment analysis for the description column and merged all the features and DTM to input all the features. Total we had 914 resulting variables after converting variables to dummy and merging them with dtms created.

f) Line of Code: (R code is from 789-844)

- cv_fit <- cv.glmnet(x_train, y_train, family = "binomial", type.measure = "auc", nfolds = 10)
  Description: This line performs cross-validation using the cv.glmnet() function from the glmnet package. It trains a logistic regression model (family = "binomial") on the training data (x_train and y_train) using 10-fold cross-validation (nfolds = 10). The evaluation metric used is the area under the ROC curve (type.measure = "auc").

- best_lambda <- cv_fit$lambda.min
  Description: This line extracts the lambda value that corresponds to the minimum cross-validated error from the cv_fit object obtained from cross-validation. It represents the optimal regularization parameter for the logistic regression model.

- final_fit <- glmnet(x_train, y_train, family = "binomial", lambda = best_lambda)
  Description: This line trains the final logistic regression model using the glmnet() function. It uses the training data (x_train and y_train), sets the family to "binomial", and specifies the optimal lambda value (best_lambda) obtained from cross-validation.

- probabilities <- predict(final_fit, newx = x_val, type = "response")

Description: This line calculates the predicted probabilities for the validation data (x_val) using the final logistic regression model (final_fit). The type = "response" argument ensures that the probabilities are computed for the positive class.

- y_pred_binary <- ifelse(probabilities >= 0.436, 1, 0)
  Description: This line converts the predicted probabilities (probabilities) into binary predictions (y_pred_binary). If the probability is greater than or equal to 0.436, it assigns a value of 1; otherwise, it assigns a value of 0. The threshold of 0.436 is used to classify the instances.

- cm <- confusionMatrix(data = class_valid_lg, reference = actual_pred_lm)
  Description: This line computes the confusion matrix using the confusionMatrix() function from the caret package. It takes the predicted binary values (class_valid_lg) and the actual values (actual_pred_lm) as inputs.

- tp <- cm$table[2, 2], fp <- cm$table[2, 1], tn <- cm$table[1, 1], fn <- cm$table[1, 2]
  Description: These lines extract the values of true positives (tp), false positives (fp), true negatives (tn), and false negatives (fn) from the confusion matrix table.

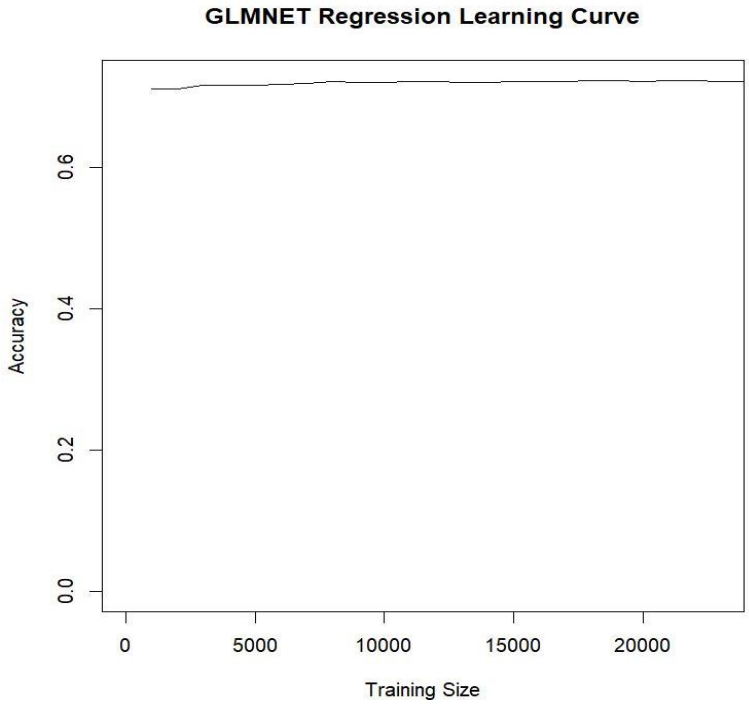
- tpr <- tp / (tp + fn), fpr <- fp / (fp + tn)
  Description: These lines calculate the true positive rate (TPR) and false positive rate (FPR) by dividing the respective values by the sum of true positives and false negatives (TP + FN) and the sum of false positives and true negatives (FP + TN), respectively.

- accuracy <- (tn + tp) / sum(tp + fp + tn + fn)
  Description: This line calculates the accuracy by summing the true negatives, true positives, false negatives, and false positives and dividing it by the total number of instances.

g) Hyperparameter Tuning

we tried tuning nfolds here in glmnet model with multiple values. The nfolds parameter determines the number of folds to be used in the cross-validation process.

h) Learning Curves



This learning curve for GLMNET also showed that the training size increases even then the accuracy remains constant, and even lesser than XGboost so It helped us validating that it is less good than our winning model.

4. **Random Forest:**

a) Description: Random Forest is an ensemble learning method that combines multiple decision trees to make accurate predictions by aggregating their results. It reduces overfitting, handles complex relationships, and is widely used for classification and regression tasks.

b) R function/ library: The R function used is ranger() function and is used to train the Random Forest model. The training data (train_xy_train) is provided as input, along with the target variable (perfect_rating_scoreYES). The mtry parameter determines the number of variables randomly selected at each split, and num.trees specifies the number of trees in the ensemble.

c) Training and Generalization Performance: The training performance has accuracy of 72.04% , TPR as 29.25% and FPR as 9.92%.

Generalization Performance: The generalization performance has accuracy of 71.8% while testing on the test data and this is better than the baseline classifier as well. Though, this is not our winning model. But we tried to compare different models for best accuracy and FPR TPR.

d) Methodology Training Generalization Performance: The generalization performance of the Random Forest model is estimated using a holdout validation approach. The dataset is split into training and validation sets. The model is trained on the training set and evaluated on the validation set, with the features (x_rf and x_val_rf) and target variables (y_rf and y_val_rf) appropriately defined. The model is then used to make predictions on the validation set using the predict() function. The predicted probabilities (rf_preds) are thresholded at 0.5 to obtain the binary classifications (rf_classifications).

Methodology for generalization performance estimation: The generalization performance is evaluated using the accuracy metric. The accuracy metric represents the overall correctness of the model's predictions. And it has been further compared with the baseline classifier to know the model's performance.

e) The features that we used are : "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind", "property_type", "bed_category", "bathrooms", "charges_for_extra", "host_acceptance", "host_response", "market", "host_is_superhost","availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights"," host_total_listings_count". We did create many new features from different existing columns and text fields and also used interaction terms. We also created DTM for columns such as amenities, transit and did sentiment analysis for the description column and merged all the features and DTM to input all the features. Total we had 914 resulting variables after converting variables to dummy and merging them with DTMs created.

f) Line of code: (R code is from 722-786)

- `rf.mod <- ranger(x = train_xy_train, y = train_xy_train$perfect_rating_scoreYES, mtry=22, num.trees=500, importance="impurity", probability = TRUE)`
  Description: This line trains a Random Forest model (`rf.mod`) using the `ranger()` function. The model is trained on the features (`train_xy_train`) and the target variable (`perfect_rating_scoreYES`) with 500 trees and a maximum of 22 randomly selected features at each split. The "impurity" criterion is used to measure node purity, and probabilities are calculated for predictions.

- `rf_preds <- predict(rf.mod, data=train_xy_val)$predictions[,2]`
  Description: This line generates predictions (`rf_preds`) for the validation set (`train_xy_val`) using the trained Random Forest model (`rf.mod`). The `predict()` function is applied to the model with the validation data, and the predicted probabilities for the positive class are extracted.

- `rf_classifications <- ifelse(rf_preds>0.5, 0, 1)`
  Description: This line converts the predicted probabilities (`rf_preds`) into binary classifications (`rf_classifications`) by assigning 0 to instances with probabilities less than or equal to 0.5 and 1 to instances with probabilities greater than 0.5. This threshold of 0.5 is commonly used to make binary predictions based on probabilities.

- `rf_acc <- mean(ifelse(rf_classifications == y_val_rf, 1, 0))`
  Description: This line calculates the accuracy (`rf_acc`) of the Random Forest model by comparing the predicted classifications (`rf_classifications`) with the true labels (`y_val_rf`) of the validation set. It computes the proportion of instances where the predicted classification matches the true label.

- `cm <- confusionMatrix(rf_classifications, y_val_rf)`
  Description: This line computes the confusion matrix (`cm`) using the `confusionMatrix()` function from the `caret` package. It takes the predicted classifications (`rf_classifications`) and the true labels (`y_val_rf`) as inputs and provides information on true positives, false positives, true negatives, and false negatives.

- `tpr <- tp / (tp + fn)`, `fpr <- fp / (fp + tn)`, `accuracy <- (tn + tp) / sum(tp+fp+tn+fn)`
  Description: These lines calculate the true positive rate (TPR), false positive rate (FPR), and accuracy using the extracted values from the confusion matrix. TPR is computed as the ratio of true positives to the sum of true positives and false negatives. FPR is calculated as the ratio of false positives to the sum of false positives and true negatives. Accuracy is determined by dividing the sum of true positives and true negatives by the total number of instances.

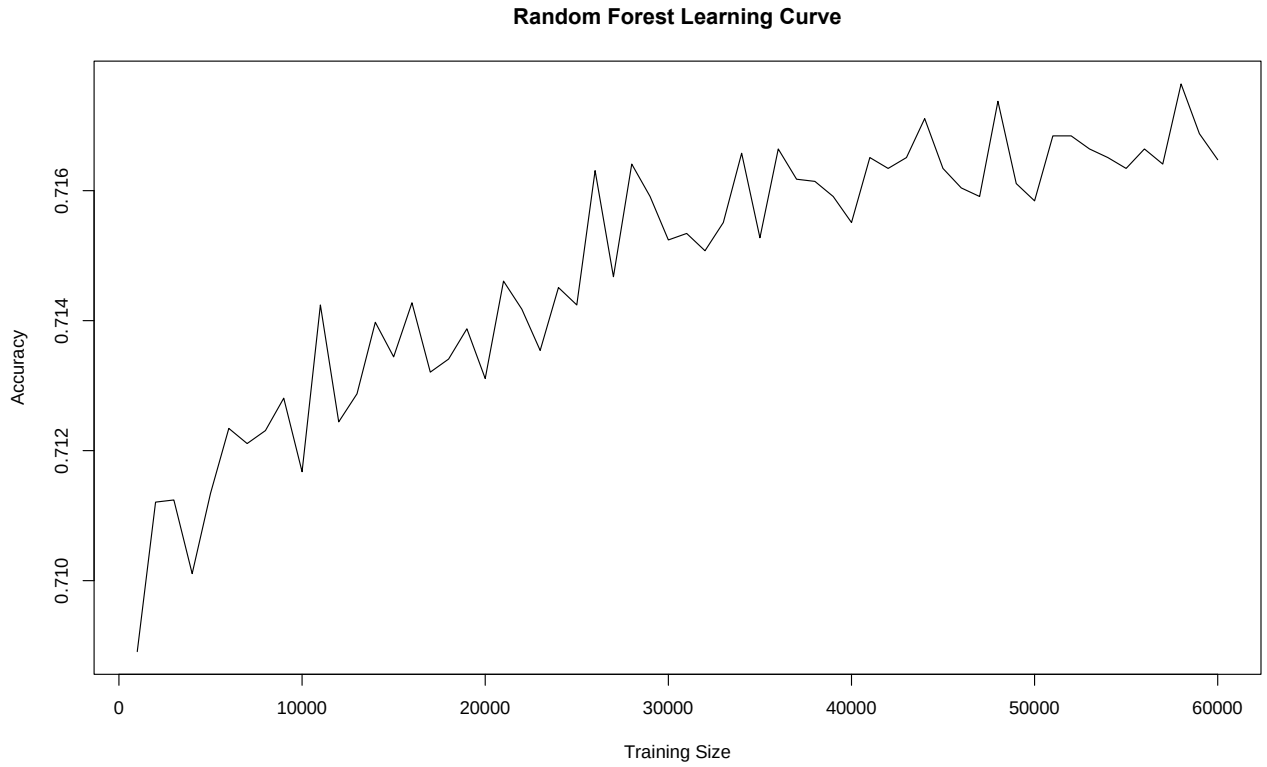
g) Hyperparameter Tuning:

```
rf.mod <- ranger(x = x_rf, y = y_rf,
                 mtry=22, num.trees=20,
                 importance="impurity",
                 probability = TRUE)
```

1. **mtry:** This parameter specifies the number of variables randomly sampled as candidates at each split. It controls the number of features considered at each split and influences the diversity and complexity of the trees. We tried a range of 18-25.

2. **num.trees:** This parameter determines the number of trees to grow in the Random Forest ensemble. Increasing the number of trees may lead to improved performance but comes with a higher computational cost. We tried a range from 80-200.

h)Learning Curve



This graph also shows that the accuracy increases as the training size increases. Though it gives good accuracy but still less than XGBOOST for us so helped us in comparing with the same train and validation split and on deciding our winning model.

5. Naive Bayes

a) Description: Naive Bayes is the type of model used in the code. Naive Bayes is a probabilistic machine learning model based on Bayes' theorem that assumes independence between features.Naive Bayes calculates the probability of each class based on the feature values and then predicts the class with the highest probability. It is widely used for classification tasks and is particularly effective with high-dimensional data. Naive Bayes is computationally efficient and works well even with limited training data.

b) R function/ library: The R function used in the code is naiveBayes(), which is part of the e1071 package. The naiveBayes() function allows for training and fitting Naive Bayes models in R. The e1071 package, a popular library for machine learning in R, provides various functions and tools for data analysis, including the Naive Bayes algorithm.

c) Training and Generalization Performance: The training performance has accuracy of 56.6% , TPR as 65.9% and FPR as 47.38%.

Generalization Performance: The generalization performance has accuracy of 44.5% while testing on the test data and this is just not even close to baseline classifier but we tried to compare different models for best accuracy and FPR TPR.

d) Methodology for Training Performance: To estimate the generalization performance of the Naive Bayes model we used a held-out data methodology, a simple train/validation split methodology is used. The dataset is divided into a training set (train_xy_train_subset) and a validation set (train_xy_val_subset). The Naive Bayes model is trained on the training set using the naiveBayes() function. Predictions are then made on the validation set using the trained model. The performance is evaluated by comparing the predicted values (nb_preds_factor) with the actual values (train_xy_val$perfect_rating_scoreYES).

Methodology for generalization performance estimation: The generalization performance is evaluated using the accuracy metric. The accuracy metric represents the overall correctness of the model's predictions. And it has been further compared with the baseline classifier to know the model's performance.

e)The features that we used are : "beds", "security_deposit", "has_cleaning_fee", "host_total_listings_count", "price_per_person", "ppp_ind", "property_type", "bed_category", "bathrooms", "charges_for_extra", "host_acceptance", "host_response", "market", "host_is_superhost","availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable", "is_location_exact", "bedrooms_accommodates", "region", "host_has_profile_pic", "host_name_frequency", "house_rules_pet", "house_rules_smoking", "house_rules_parties", "bedrooms_bathrooms_ratio", "beds_per_bedrooms", "beds_per_accomodates", "bathrooms_per_accomodates", "cancellation policy", "monthly_price", "city", "has_min_nights", "host_total_listings_count". We did create many new features from different existing columns and text fields and also used interaction terms. We also created DTM for columns such as amenities, transit and did sentiment analysis for the description column and merged all the features and DTM to input all the features. Total we had 914 resulting variables after converting variables to dummy and merging them with DTMs created.

f) Line of Code: (R code is from 938-986)

- `nb_model<-naiveBayes(train_xy_train_subset, train_xy_train$perfect_rating_scoreYES)`
  Description: This line trains the Naive Bayes model using the `naiveBayes()` function from the `e1071` package. It takes the subset of the training data (`train_xy_train_subset`) and the corresponding target variable (`train_xy_train$perfect_rating_scoreYES`) as inputs. The `naiveBayes()` function learns the probability distributions of the features given the class labels to build the Naive Bayes model.

- `nb_probs <- predict(nb_model, newdata = train_xy_val[, -1], type = "raw")[, 2]`
  Description: This line makes predictions on the validation set (`train_xy_val`) using the trained Naive Bayes model (`nb_model`). The `predict()` function is used to obtain the predicted probabilities for each class using the validation set as the new data. The `type = "raw"` argument ensures that the probabilities are returned. The `[ , 2]` indexing is used to extract the probabilities for the positive class (class "1").

- `nb_preds_factor <- factor(ifelse(nb_probs > 0.55, "1", "0"), levels = c("0", "1"))`

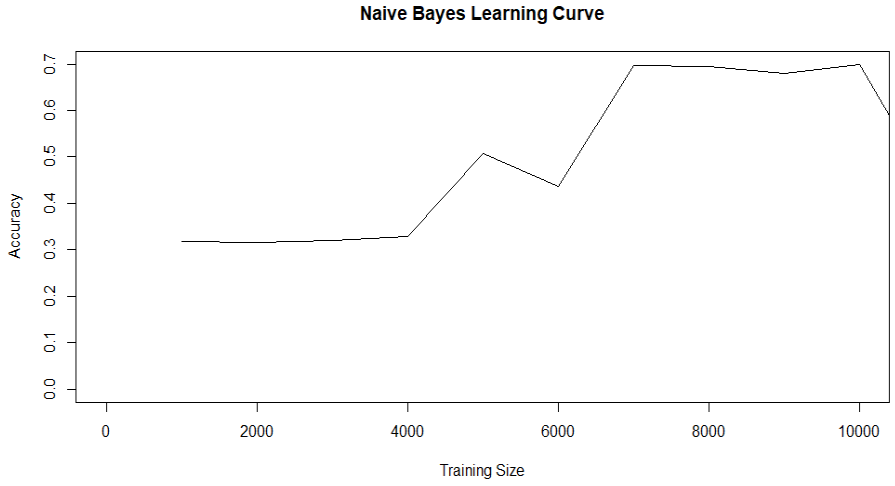
Description: This line converts the predicted probabilities (`nb_probs`) to binary predictions (`nb_preds_factor`) based on a threshold of 0.55. Probabilities greater than 0.55 are classified as positive (class "1"), while the rest are classified as negative (class "0"). The `factor()` function is used to convert the predictions into a factor variable with levels "0" and "1".

- `nb_cm <- confusionMatrix(data = nb_preds_factor, reference = train_xy_val$perfect_rating_scoreYES)`
  Description: This line calculates the confusion matrix (`nb_cm`) to evaluate the performance of the Naive Bayes model. The `confusionMatrix()` function from the `caret` package is used, taking the predicted values (`nb_preds_factor`) and the actual values (`train_xy_val$perfect_rating_scoreYES`) as inputs. The confusion matrix provides information on true positives, false positives, true negatives, and false negatives.

- `tpr <- tp / (tp + fn)`, `fpr <- fp / (fp + tn)`
  Description: These lines calculate the true positive rate (TPR) and false positive rate (FPR) based on the extracted values from the confusion matrix. TPR is computed as the ratio of true positives to the sum of true positives and false negatives, while FPR is computed as the ratio of false positives to the sum of false positives and true negatives.

g) Hyperparameter: No hyperparameter tuning was required.

h): Learning Curve:



This graph also shows that the accuracy increases as the training size increases. It also gives less accuracy than XGBOOST for us so helped us in comparing with the same train and validation split. So it was clear Naïve Bayes performs least good for us.

6. GBM(Gradient Boosting Machine):

a) Description: Gradient Boosting Machine (GBM) is a powerful machine learning model that combines multiple weak predictive models, typically decision trees, to create a strong predictive model. It is an ensemble method that iteratively builds the model by minimizing the errors of the previous models. GBM handles different data types, and the ability to capture complex relationships in the data.

b) R function/ library: The code uses the gbm() function from the gbm library to train the GBM model.

c) Training and Generalization Performance: The training performance has accuracy of 72.05% , TPR as 28.7% and FPR as 9.7%.

Generalization Performance: The generalization performance has accuracy of 71.6% while testing on the test data and though this is better than the baseline classifier but this is not our winning model. Though, this is not our winning model. But we tried to compare different models for best accuracy and FPR TPR.

d) Methodology Training Generalization Performance: The methodology for estimating the generalization performance involves a simple train/validation split. The dataset is divided into a training set and a validation set using the `createDataPartition()` function. The training set is used to train the GBM model with the specified parameters, while the validation set is used to evaluate the model's performance. The predictions made by the trained model on the validation set are compared to the actual target values to calculate accuracy. The confusion matrix is then computed to provide a detailed breakdown of the model's predictions. This approach allows for a preliminary assessment of the model's performance before applying it to unseen data, helping to gauge its effectiveness in generalizing to new observations.

Methodology for generalization performance estimation: The generalization performance is evaluated using the accuracy metric. The accuracy metric represents the overall correctness of the model's predictions. And it has been further compared with the baseline classifier to know the model's performance.

e) The features that we used are :
"beds", "security_deposit", "has_cleaning_fee.YES", "price_per_person", "ppp_ind.1",
"property_type.condo", "property_type.hotel", "property_type.house", "property_type.other", "bed_category.other",
"bathrooms", "charges_for_extra.YES", "host_acceptance.MISSING", "host_acceptance.SOME", "host_response.MISSING",
"host_response.SOME", "market.Boston", "market.Chicago", "market.D.C.", "market.Denver", "market.East.Bay..CA",
"market.Los.Angeles", "market.Monterey.Region", "market.Nashville", "market.New.Orleans",
"market.New.York", "market.North.Carolina.Mountains", "market.OTHER", "market.Other..Domestic.",
"market.Portland", "market.San.Diego", "market.San.Francisco", "market.Seattle", "host_is_superhost",
"availability_30", "price", "availability_365", "host_identity_verified", "latitude", "longitude", "instant_bookable",
"is_location_exact", "bedrooms_accommodates", "regionMidwest", "regionMountain", "regionNew.England", "regionOther",
"regionSouth.Central", "regionSoutheast", "regionSouthwest", "regionWest", "host_name_frequency",
"house_rules_petTRUE", "house_rules_smokingTRUE", "house_rules_partiesTRUE"

f) Line of code: (R code is from 846-935)

- `gbm_mod <- gbm(formula_gbm, data = train_xy_train, n.trees = 1000, interaction.depth = 6, shrinkage = 0.01, bag.fraction = 0.5, train.fraction = 0.7, cv.folds = 5, distribution = "gaussian")`
  Description: This line trains a Gradient Boosting Machine (GBM) model using the `gbm()` function from the `gbm` package in R. The model is trained on the `train_xy_train` dataset using the specified formula (`formula_gbm`), with parameters such as the number of trees (`n.trees`), interaction depth (`interaction.depth`), shrinkage (`shrinkage`), bag fraction (`bag.fraction`), train

fraction (`train.fraction`), and cross-validation folds (`cv.folds`).

- `gbm_preds <- predict(gbm_mod, newdata = train_xy_val, type = "response")`
  Description: This line makes predictions on the validation set (`train_xy_val`) using the trained GBM model (`gbm_mod`). The `predict()` function is used with the `type = "response"` argument to obtain predicted values.

- `gbm_classifications <- ifelse(gbm_preds > 0.44, 1, 0)`
  Description: This line converts the predicted probabilities (`gbm_preds`) into binary classifications based on a threshold of 0.44. Probabilities greater than 0.44 are classified as 1, indicating the positive class, while the rest are classified as 0, indicating the negative class.

- `gbm_acc<-mean(ifelse(gbm_classifications ==train_xy_val$perfect_rating_scoreYES, 1, 0))`
  Description: This line calculates the accuracy of the GBM model by comparing the predicted classifications (`gbm_classifications`) with the actual values (`train_xy_val$perfect_rating_scoreYES`) from the validation set. It calculates the mean of a logical vector where 1 represents a correct prediction and 0 represents an incorrect prediction.

- `gbm_cm <- confusionMatrix(factor(gbm_classifications, levels = c(0, 1)), factor(train_xy_val$perfect_rating_scoreYES, levels = c(0, 1)))`
  Description: This line creates a confusion matrix (`gbm_cm`) using the `confusionMatrix()` function from the `caret` package. It takes the predicted classifications (`gbm_classifications`) and actual values (`train_xy_val$perfect_rating_scoreYES`) as factors, specifying the levels of the classes as 0 and 1. The confusion matrix provides information about the true positives, false positives, true negatives, and false negatives.

- `tp <- gbm_cm$table[2, 2]`, `fp <- gbm_cm$table[2, 1]`, `tn <- gbm_cm$table[1, 1]`, `fn <- gbm_cm$table[1, 2]`
  Description: These lines extract the true positives (`tp`), false positives (`fp`), true negatives (`tn`), and false negatives (`fn`) from the confusion matrix (`gbm_cm`).

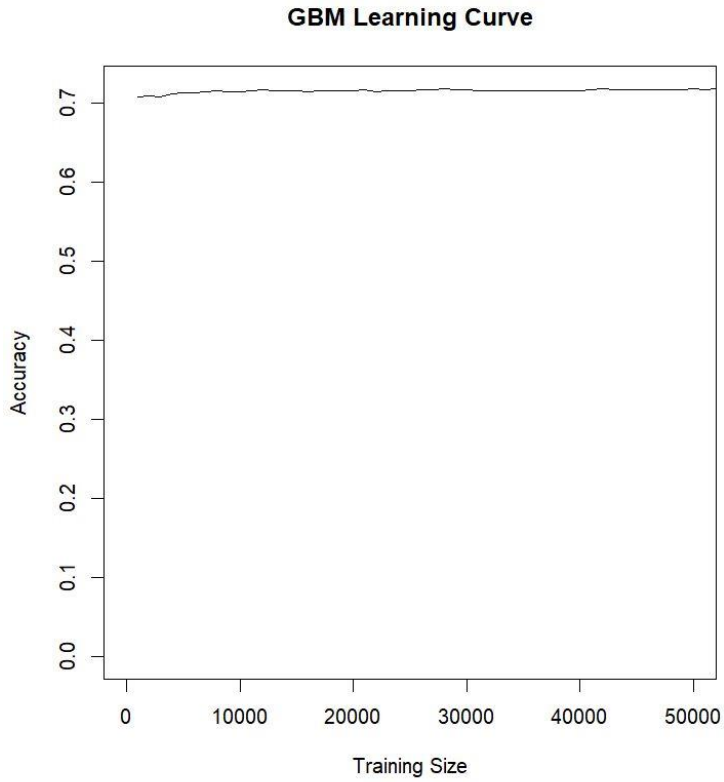
- `tpr <- tp / (tp + fn)`, `fpr <- fp / (fp + tn)`
  Description: These lines calculate the true positive rate (TPR) or sensitivity and false positive rate (FPR) or 1-specificity, respectively, using the extracted values from the confusion matrix.

g) Hyperparameters:

We used n.trees: Number of trees (boosting iterations) - Tried range from 100 to 1000
interaction.depth: Maximum depth of individual tree - tried range of 0 to 10
cv. folds: Number of folds used in cross-validation - tried range of 0 to 5
Got best results with the code below:

```
gbm_mod <- gbm(formula_gbm,
        data = train_xy_train,
        n.trees = 1000,
        interaction.depth = 6,
        cv.folds = 5,
        distribution = "gaussian")
```

h) Learning Curve



This learning curve for GBM showed that the training size increases even then the accuracy remains constant, and even lesser than XGboost so It helped us validating that it is less than the winning model for us.

Ensemble Method - Merging Models:

Additionally we tried merging the models we created above to see if that gives us better prediction . I **merged logistic regression with the XGboost model** to see if that helps with better accuracy and TPR & FPR. Basically we combined predictions on validation sets.

Below is the code:

```
# Combine predictions from validation set
ensemble_preds <- ifelse(valid_preds + y_pred >=0.38, 1, 0)
```
This is merging the predictions for logistic regression with XGboost model

```
# Evaluate the ensemble
actual_pred <- train_xy_val$perfect_rating_scoreYES
ensemble_cm <- table(ensemble_preds, actual_pred)
accuracy <- sum(diag(ensemble_cm)) / sum(ensemble_cm)
# Extract true positive, false positive, true negative, and false negative values from the confusion matrix
tp <- ensemble_cm[2, 2]
fp <- ensemble_cm[1, 2]
tn <- ensemble_cm[1, 1]
fn <- ensemble_cm[2, 1]
```

```
# Calculate true positive rate (TPR) and false positive rate (FPR)
tpr <- tp / (tp + fn)
fpr <- fp / (fp + tn)
```

This gave us a **TPR of 38.44% at the FPR of 9.8% which was better than the TPR & FPR of these models individually.**

Further we tried predicting it on our test set using the code below.

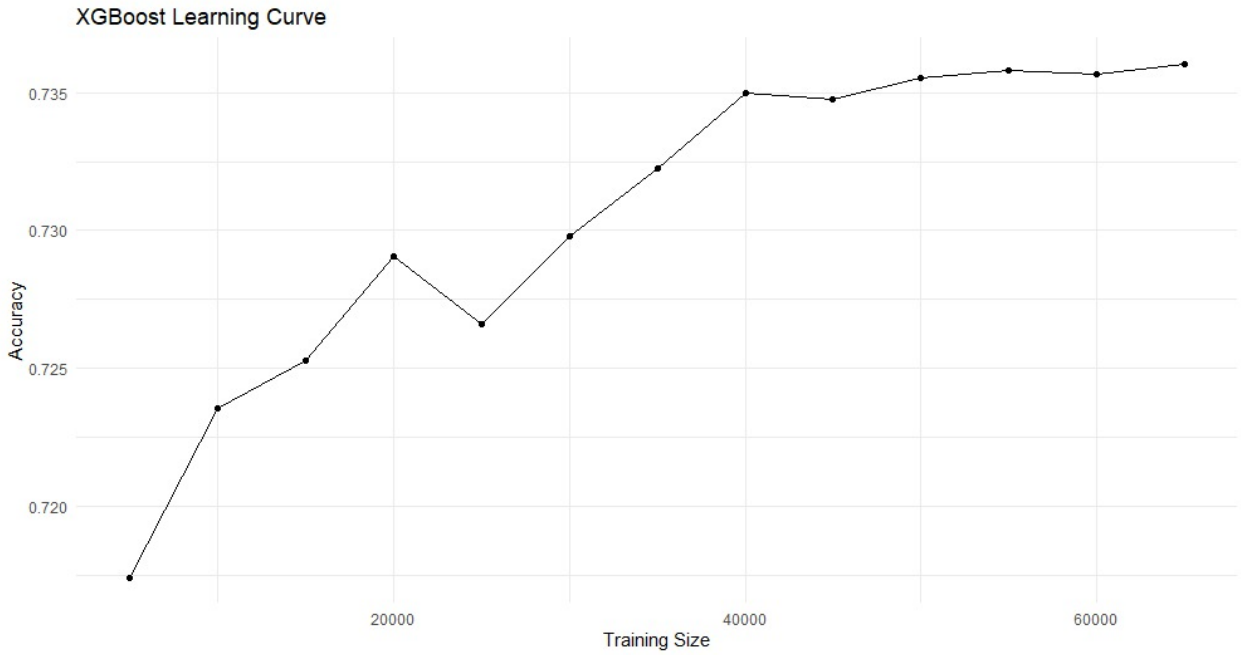
```
library(caret)
classifications_perfect_e <- ifelse(probs_perfect + y_pred_final> .38, "YES", "NO")
```

But, it didn't generalize well in our test dataset. This was a challenge which we faced but that's our next target to learn.

k) (Required to receive at least 85 points) Also include any learning curves that your team created as well as any insights generated by these curves.

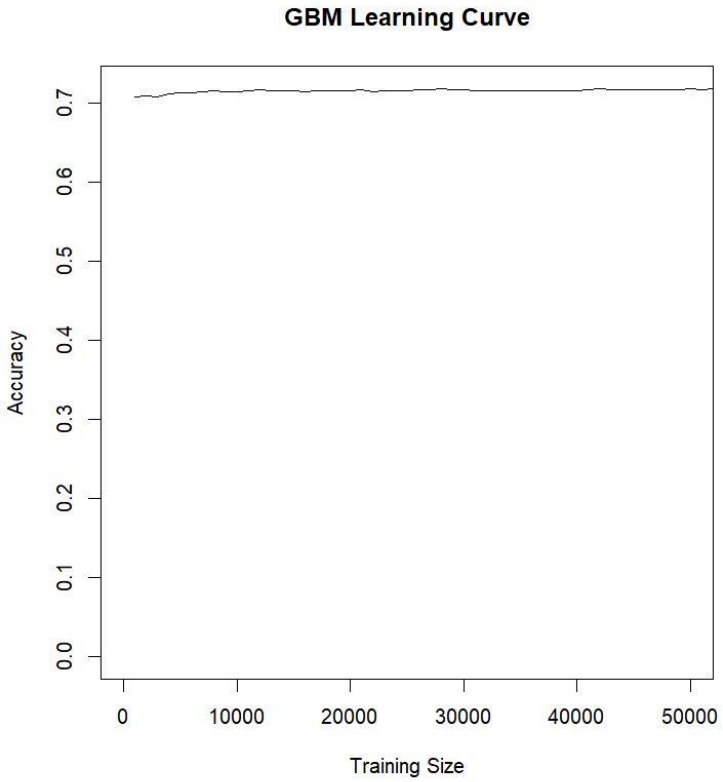
Learning Curves

i. Xgboost



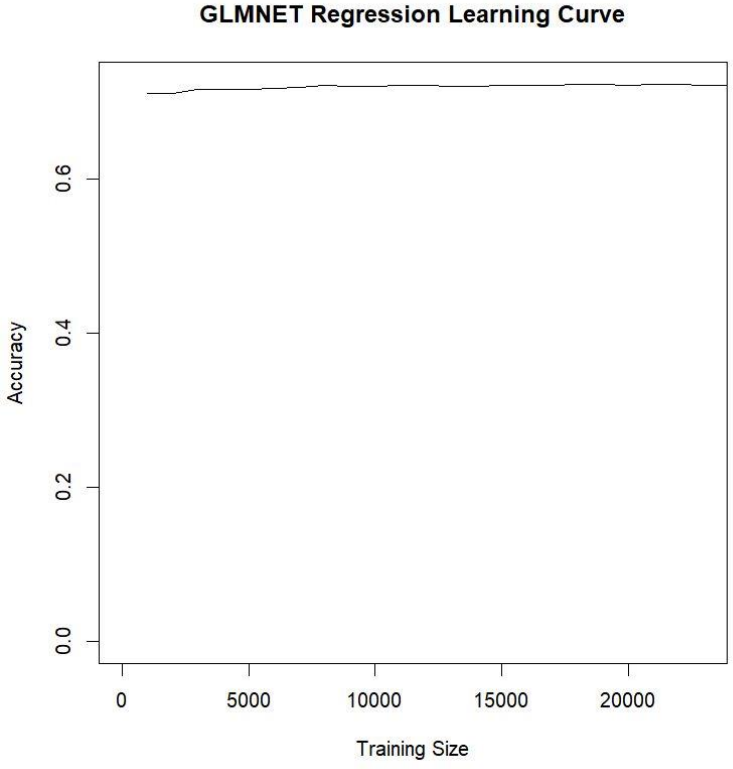
This is the learning curve that we made for our winning model, it shows that the accuracy increased as the training size increased. This helped us know that we can split 70% training and 30% validation to get best results. Increasing more size was decreasing the accuracy.

ii. Gbm



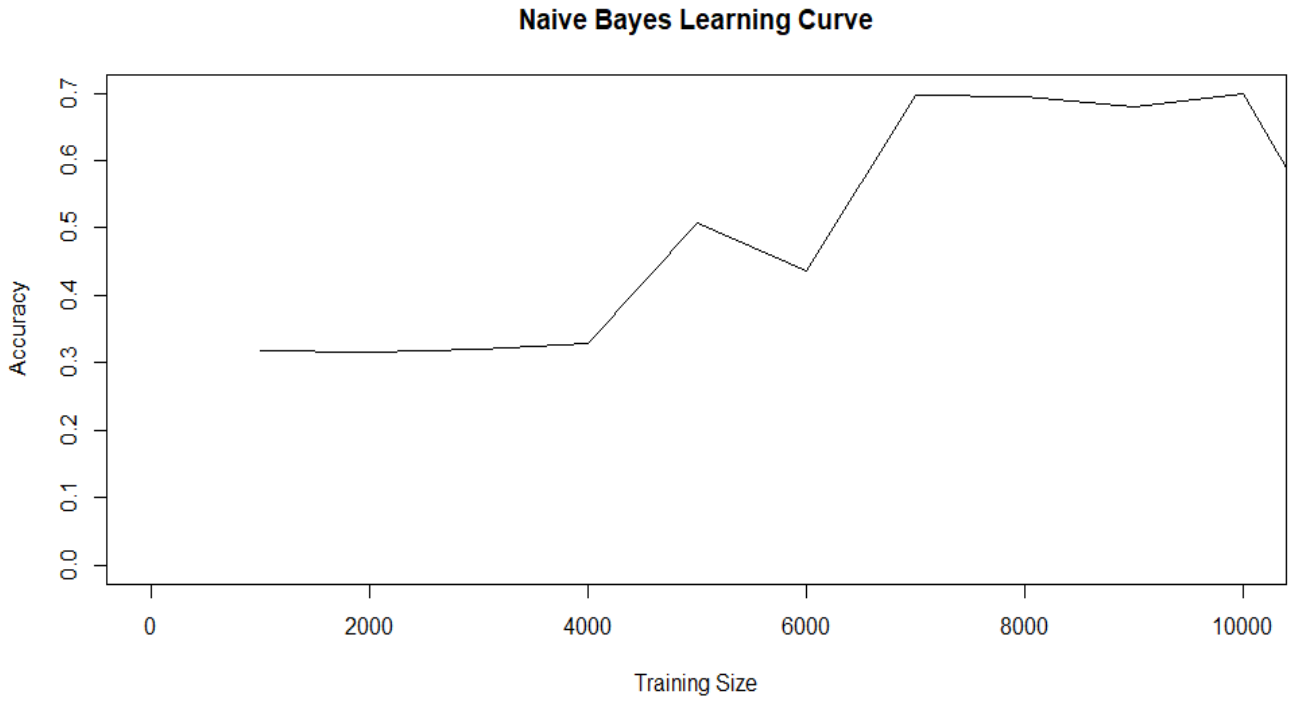
This learning curve for GBM showed that the training size increases even then the accuracy remains constant, and even lesser than XGboost so It helped us validating that it is less than the winning model for us.

iii. Glmnet



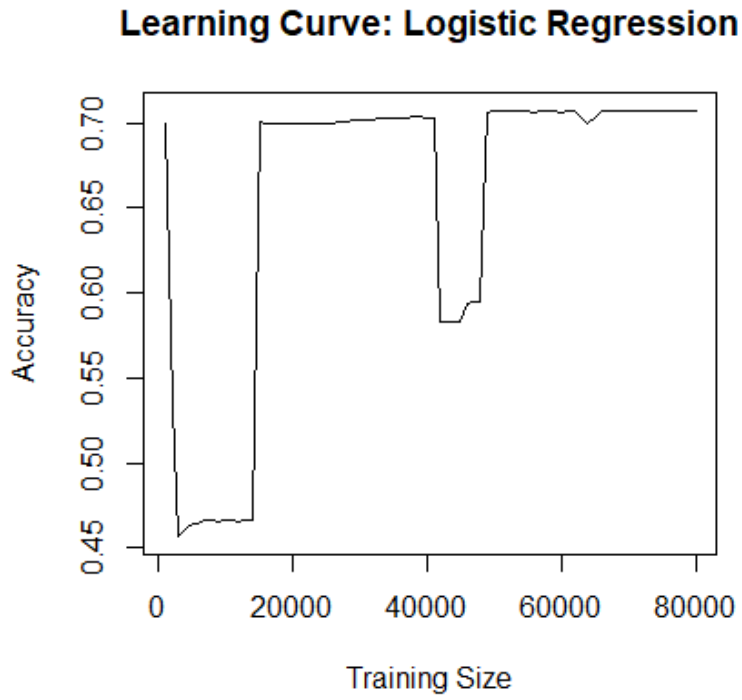
This learning curve for GLMNET also showed that the training size increases even then the accuracy remains constant, and even lesser than XGboost so It helped us validating that it is less good than our winning model.

iv. Naive Bayes



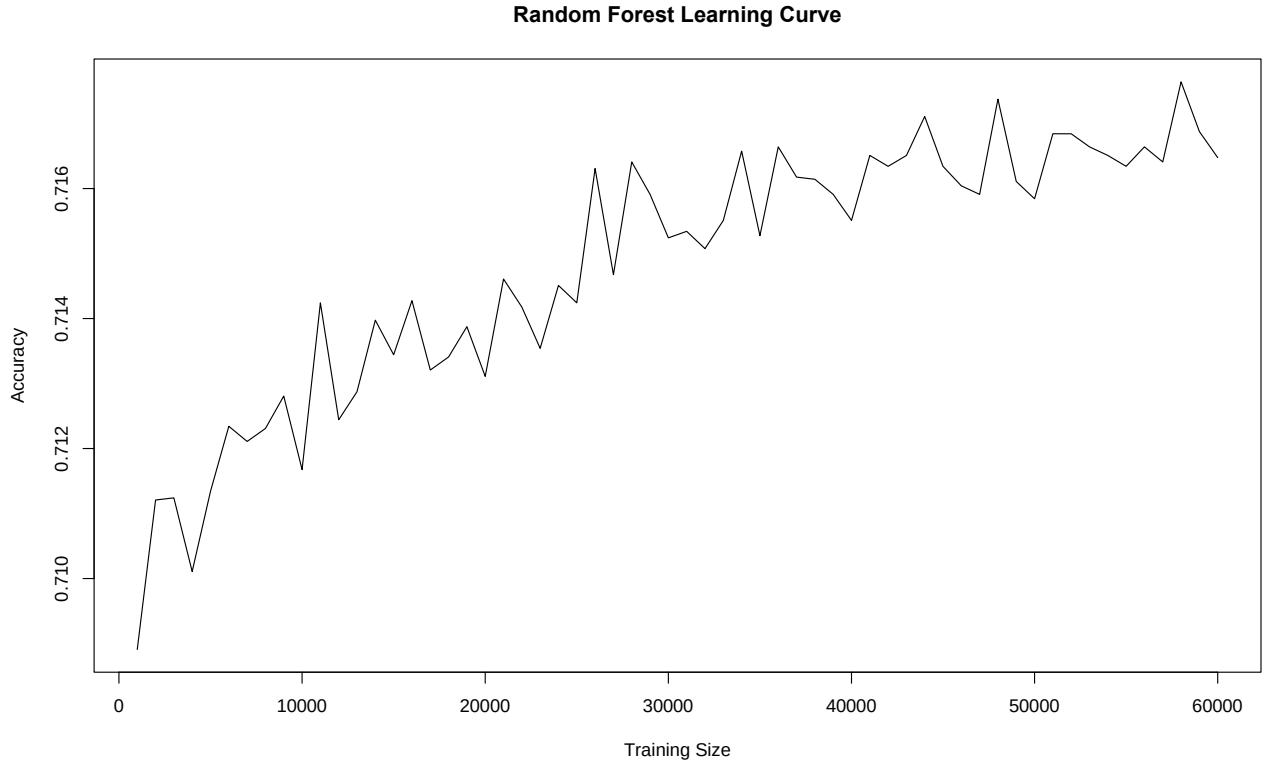
This graph also shows that the accuracy increases as the training size increases. It also gives less accuracy than XGBOOST for us so helped us in comparing with the same train and validation split. So it was clear Naïve Bayes performs least good for us.

v. Logistic Regression



This graph also shows that the accuracy increases as the training size increases. It also gives less accuracy than XGBOOST for us so helped us in comparing with the same train and validation split.

vi. Random Forest



This graph also shows that the accuracy increases as the training size increases. Though it gives good accuracy but still less than XGBOOST for us so helped us in comparing with the same train and validation split and on deciding our winning model.

Section 5: Reflection/takeaways

1) What did your group do well?

We were able to clean the data, add the document term matrix and logistic regression smoothly.
These are a few examples on how we handled **cleaning of the data**:

a) Handling missing values:

bathrooms: Replace missing values with the median of the bathrooms column.

b) Modifying categorical variables:

bed_category: Create a new variable based on the bed_type column. If it's "Real Bed", assign "bed", otherwise assign "other".

c) Parsing numeric values:

price: Convert the price column to numeric using the parse_number function.

d) Creating additional variables:

charges_for_extra: Create a new variable indicating whether there are charges for extra people or not.

e) Modifying binary variables:

is_business_travel_ready: Convert "TRUE" to 1, "FALSE" to 0, and replace missing values with "MISSING".

f) Numeric variable transformations:

security_deposit and price: Convert these columns to numeric by removing dollar signs and commas.

Then for the **DTM Part** we were easily able to **defining the cleaning_tokenizer function, tokenize text data, create the vocabulary, pruning the vocabulary, create the vectorizers, creating document-term matrices (DTMs) from it** , eventually **convert the sparse matrix to data frame and combine data frames** using the cbind function to combine the dummy variable data frame (data_dummies) with the three DTMs (dtm_train_df, dtm_train_df_1, dtm_train_df_2). The resulting data frames are stored as data_dummies_combined, data_dummies_combined_2, and data_dummies_combined_final, respectively.

Logistic regression was one of the more comfortable models which we already had a hang on. It demonstrated the utilization of logistic regression for binary classification. The model is trained using the glm() function, and a summary of the model's results is generated using summary(). Predictions are made on a validation dataset, and a threshold of 0.46 is applied to classify instances into positive or negative classes. The accuracy of the model is calculated using a confusion matrix, and the true positive rate (TPR) and false positive rate (FPR) are computed and printed. This approach provides a concise and informative analysis of the logistic regression model's performance.

2) What were the main challenges

The **main challenge** was to increase the true positive rate (TPR) and decrease the false positive rate (FPR) despite extensive feature engineering efforts.
Despite of the following efforts we faced the above challenge:

- **Various columns and new feature creations** were explored, including integrating external data by preprocessing and cleaning the "external" dataset.
- **The dataset was transformed by combining** gender, minimum age, and maximum age into a single column and converting it to a wide format for population counts.
- **Missing values were replaced** with zeros and specific columns related to gender and age groups were selected from the "full_data" dataset.
- **New variables were created** based on the "host_since" column, including day of the week, month, and season of registration.
- **Integrating different models** such as random forest and support vector machines (SVM) added complexity to the analysis process.

Despite these efforts, a decrease in the TPR was observed, and achieving the objective of increasing the FPR while keeping the TPR below 10% proved to be challenging. Nevertheless, a persevering approach was maintained to derive meaningful insights from the analysis.

3) What would your group have done differently if you could start the project over again?

- Firstly, we would have tried more thorough exploratory data analysis (EDA) and invested more time in understanding the data, identifying patterns, and exploring relationships between variables and to do better feature engineering.
- We would have tried our code without creating dummy variables and would have compared what changes are reflected with and without creation of dummy variables.
- Better validation strategy - we would pay more attention to the choice of validation strategy, considering techniques like cross-validation etc, which would provide a more robust assessment of model performance and reduce the risk of overfitting.
- Addressing class imbalance - if the project involves a classification task with imbalanced classes, we would pay special attention to addressing this issue

4) What would you do if you had another few months to work on the project?

- Advanced feature engineering: We would explore more sophisticated feature engineering techniques, such as interaction terms, polynomial features, dimensionality reduction methods or feature embeddings for categorical variables.
- Fine-grained model evaluation: We would perform more extensive model evaluation, including a deeper analysis of different evaluation metrics, robustness testing, and model sensitivity analysis.
- Model architecture exploration: We would delve into more advanced model architectures, such as deep learning models (e.g., neural networks, recurrent neural networks, or transformers), ensemble models, or hybrid models that combine different algorithms. This would allow us to leverage the power of these models and potentially uncover hidden patterns or improve prediction accuracy.

5) What advice do you have for a group starting this project next year?

As a group starting this project next year, I have several pieces of advice to offer:

- Perform thorough data exploration and EDA.
- Apply effective feature engineering techniques and try creating many new features and keep experimenting with it.
- Start working with external data sources early to boost the TPR.
- Experiment with various models and evaluate their performance.
- Address class imbalance using appropriate techniques.
- Regularize models and tune hyperparameters after cleaning and feature engineering.
- Continuously monitor and validate model performance.