GUEST SATISFACTION PREDICTION

TEAM 17 - TEAM MEMBERS:

Jana Hani Mohamed - 2022170647
Toka Khaled Mostafa - 2022170106
Jana Essam Abdelfatah - 2022170112
Manar Mostafa Fathy - 2022170437
Rawan Mohamed Taha - 2022170160
Malak Fekry Mohamed - 2022170433

Table of Contents

Introduction

Guest Satisfaction Prediction is a project that aims to train a machine learning regression model how to predict the level of a guest's satisfaction with a certain lodging. This is accomplished by using data gathered from the popular booking website Airbnb.

Data Exploration

The dataset we're using here is the GuestSatisfactionPrediction.csv. It contains 8724 rows and 69 columns. Here is a quick look at all the columns and their descriptions:

Column	**Description**
Id	Unique identifier for the listing.
listing_url	URL to the Airbnb listing.
name	Title of the listing.
summary	Short summary description provided by the host.
space	Description of the space guests can access.
description	Full description of the listing.
neighborhood_overview	Overview of the neighborhood.
notes	Additional notes from the host.
transit	Information about nearby transit options.

access	Details on guest access to the property.
Interaction	Information on how hosts interact with guests.
house_rules	House rules set by the host.
thumbnail_url	Thumbnail image URL of the listing.
host_id	Unique identifier for the host.
host_url	URL to the host's profile.
host_name	Name of the host.
host_since	Date the host joined Airbnb.
host_location	Location of the host.
host_about	Bio or description provided by the host.
host_response_time	Average time the host takes to respond.
host_response_rate	Percentage of messages responded to.
host_acceptance_rate	Rate at which host accepts bookings.
host_is_superhost	Whether the host is a superhost.
host_neighbourhood	Host's self-described neighborhood.
host_listings_count	Number of listings the host has.
host_total_listings_count	Total listings count (including inactive).
host_has_profile_pic	Whether the host has a profile picture.
host_identity_verified	Whether the host's identity is verified.
Street	Street address of the listing.
Neighbourhood	Name of the local neighborhood.
neighbourhood_cleansed	Cleaned or standardized neighborhood name.
City	City of the listing.
State	State where the listing is located.
Zipcode	Zip code of the listing.
Market	General market area.

smart_location	Formatted location string.
country_code	Country code.
Country	Full country name.
Latitude	Latitude of the listing.
Longitude	Longitude of the listing.
is_location_exact	Whether the location is exact.
property_type	Type of property (e.g., apartment, house).
room_type	Type of room available.
Accommodates	Number of people the listing can accommodate.
Bathrooms	Number of bathrooms.
Bedrooms	Number of bedrooms.
Beds	Number of beds.
bed_type	Type of bed.
Amenities	List of included amenities.
square_feet	Size of the listing in square feet.
nightly_price	Price per night.
price_per_stay	Total price for a stay.
security_deposit	Security deposit amount.
cleaning_fee	Cleaning fee charged.
guests_included	Number of guests included in base price.
extra_people	Extra charge for additional guests.
minimum_nights	Minimum nights required per booking.
maximum_nights	Maximum nights allowed per booking.
number_of_reviews	Total number of reviews.
number_of_stays	Total number of stays.
first_review	Date of the first review.

last_review	Date of the most recent review.
review_scores_rating	Overall review score (target variable).
requires_license	Whether a license is required.
instant_bookable	Whether guests can book instantly.
is_business_travel_ready	Whether listing is ready for business travelers.
cancellation_policy	Policy on cancellations.
require_guest_profile_picture	Whether guest profile pictures are required.
require_guest_phone_verification	Whether guest phone verification is required.

From the look of this data we can conclude that we have 68 features that we can use to train our model(s). The 69th column , review_scores_rating, is the target column whose value our model needs to reliably predict.

Data Preprocessing

After going through our dataset and exploring all aspects of the data present in there, we then move onto preprocessing our data, the stage in which we clean up the data to make it more suitable for our machine learning model(s).

- ### Checking for Duplicates

We first start by checking our entire data for duplicates. This is to ensure that no redundant data is present to skew the predictive judgment of our model(s). We also specifically checked our 'id' column for duplicates. That is to ensure the integrity and uniqueness of each row. It's always better to have more varied data than not.

▪ Fixing Numerical Data

Our next step is to go through the numerical columns that may be misrepresented as object columns. This can occur due to special characters being present in the data (such as $, -, , etc...). Thus, it is up to us to remove any such characters from those columns using built-in python functions designed for that purpose.

Within that same vein, we must also change the datatype of those columns as not doing so may impact any work we may wish to perform on those columns later in the project. Thus, we must set those columns to a numerical datatype (i.e. float, int, etc..).

The columns in question were*: host_response_rate, nightly_price, price_per_stay, extra_people, security_deposit, cleaning_fee*, and *zipcode*. After removing the special characters from each one we assigned the datatype, float, to each one.

▪ Checking for Nulls

Our next step is to check the data for nulls. Null values are missing values that may impact our model(s)'s performance negatively as they imply a lack of data rather than a simple '0' value. Thus, it is our job to find what columns may be missing values and how many values are missing from each of the afflicted columns. This is done using an in-built python function designed to weed out any such values.

Now, there are many methods a developer can use to deal with nulls. Numerical columns, naturally, have different null handling methods than text columns. We employed several of those methods in this project.

In the numerical columns:

- Handling Nulls using Mode

Some columns are normal. In other words, they show no skewness when being visualized using histograms and other such visualization tools. The nulls in them were replaced with the ‘mode’ value in those columns.

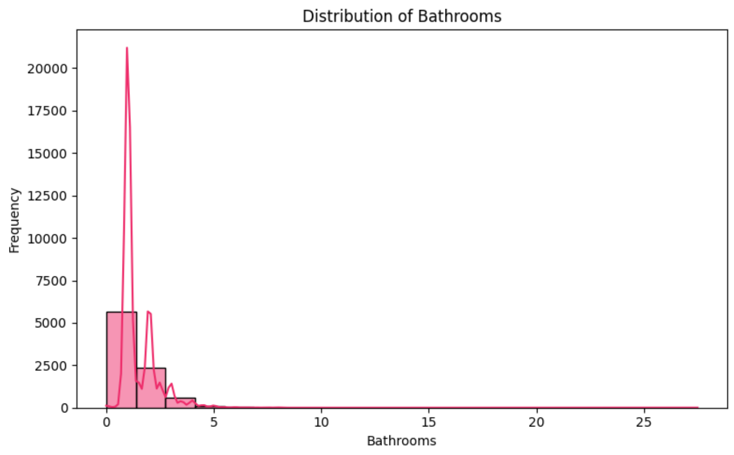
The columns in question were: *market, host_neighbourhood, state, neighbourhood, 'host_location',' host_response_time', 'zipcode', 'host_since', 'host_is_superhost', 'host_has_profile_pic', 'host_identity_verified', 'host_name'.*

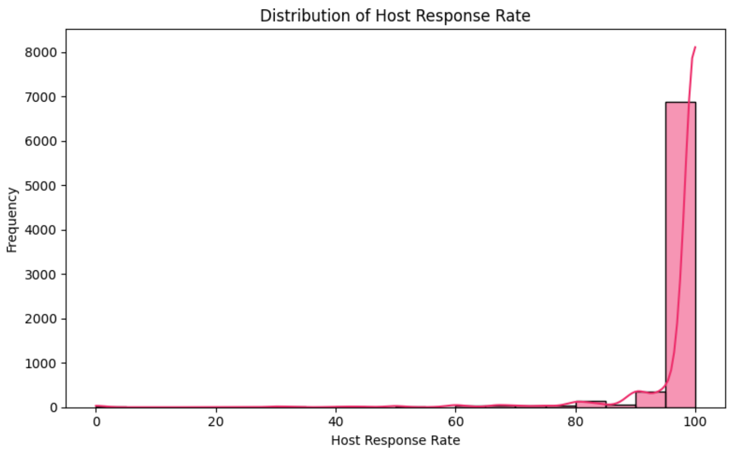
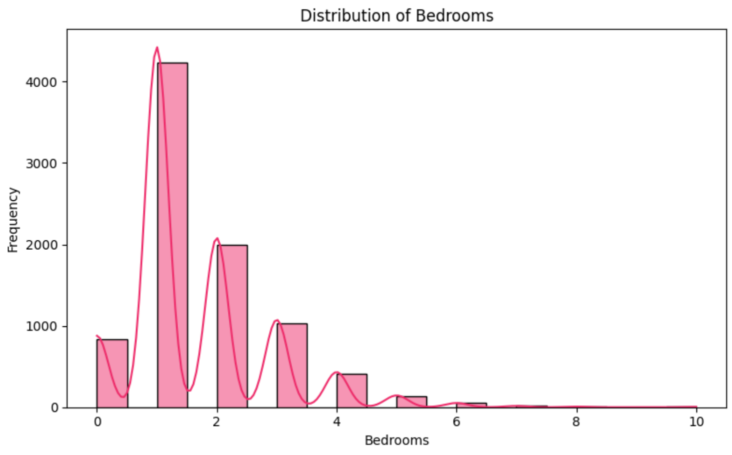
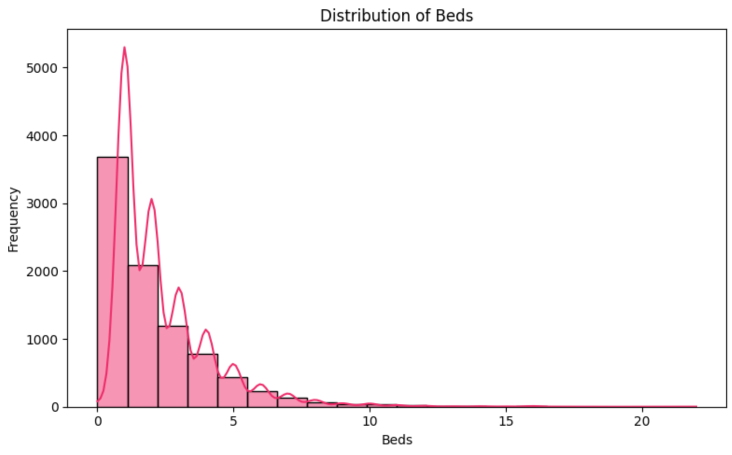
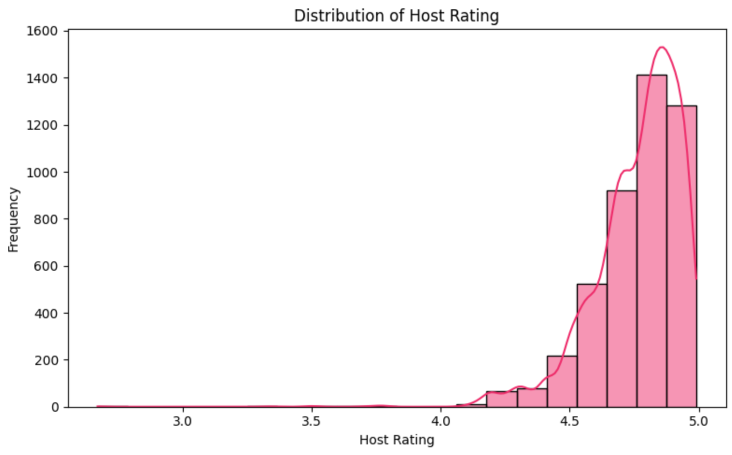
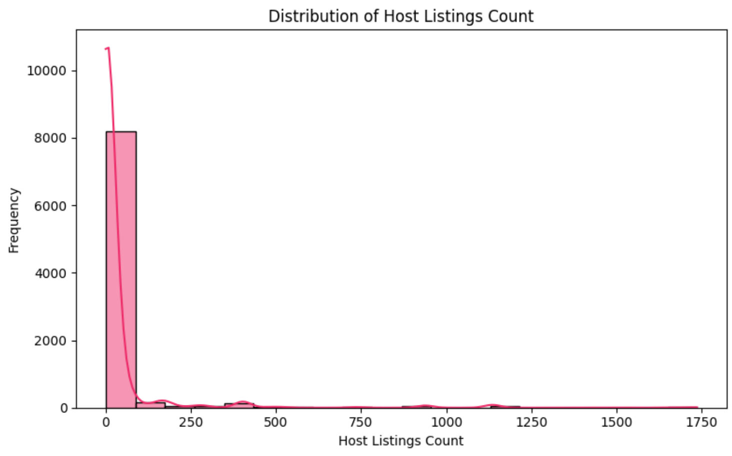
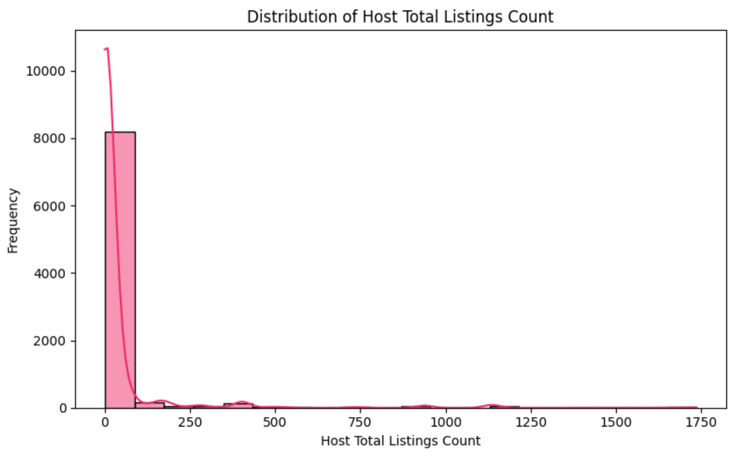
- Handling Nulls using Median

Some columns, on the other hand, are not normal. In other words, they show some degree of skewness when being visualized using histograms and other such visualization tools. The nulls in them were replaced with the ‘median’ value in those columns.

The columns in question were: *'bathrooms', 'bedrooms', 'beds', 'square_feet', 'host_rating', 'host_listings_count', 'host_total_listings_count', 'host_response_rate'.*

You can see the skewness of each of those columns clearly in the following histograms:




Distribution of Host Response Rate
Frequency
Host Response Rate
0
20
40
60
80
100
1000
2000
3000
4000
5000
6000
7000
8000

Distribution of Bedrooms
Frequency
Bedrooms
0
2
4
6
8
10
1000
2000
3000
4000

Distribution of Beds
Frequency
Beds
0
5
10
15
20
1000
2000
3000
4000
5000

Distribution of Host Rating
Frequency
Host Rating
3.0
3.5
4.0
4.5
5.0
200
400
600
800
1000
1200
1400
1600

Distribution of Host Listings Count
Frequency
Host Listings Count
0
250
500
750
1000
1250
1500
1750
2000
4000
6000
8000
10000

Distribution of Host Total Listings Count
Frequency
Host Total Listings Count
0
250
500
750
1000
1250
1500
1750
2000
4000
6000
8000
10000

- Handling Nulls by filling them with zero

In some columns, the only method of handling the nulls that would make sense from a logical standpoint was to fill them with zero. For example, for the *'cleaning fee'* and *'security_deposit'* columns a host may elect not to place a value on such matters. In such cases it wouldn't make sense for us as developers to place arbitrary values in place of the nulls. Thus, we chose to fill them with zeroes.

In the Text Columns

- Handling Nulls by filling them with default phrases

In some columns, such *as neighborhood_overview, notes, transit, access, interaction, house_rules,* and *host_about* we elected to fill the nulls with simple default phrases. For example, for the house_rules column we used "No house rules" and for the host_about we used "No host info" and so on so forth for the rest. This is because such columns do not require artificial data or any such artificial content that may be misleading at best. In such cases brevity is the best policy.

- Handling Nulls by filling them with AI Generated Text

There are columns, such as space*, description,* and *summary* we elected to use Cohere, a chat bot with an API Key that we inserted into the code to directly generate text data, based off of existing features in the data for maximum data integrity, into the null sections in the 3 columns. We elected to use this approach to preserve the structure of those columns as they contain quite the amount of descriptive information that we felt the need to preserve for later use in the project.

- Flooring Columns

After handling nulls, we move on to ensuring then proper handling of numerical columns, such as *bathrooms*, *bedrooms,* and *beds*, where having fractional values does not make sense. This is because, for the objects whose value those columns track, you can't have a fraction of a whole.

This is why for logical purposes we elected to take the floor value of the fractional values of those columns and change the datatype into 'int'.

▪ Text Processing

Processing text requires quite the number of special techniques, as opposed to handling numerical data, to make it as palatable as possible for the model we're going to be feeding it into. Those techniques include:

- ### Text Cleaning

  This is the process where we clean the text of all symbols. Special characters, punctuation, html tags, etc… Leaving symbols in the text makes it hard for our model(s) to process and articulate the data in an efficient manner which makes removing them a priority.

- ### Handling Contractions

  While contractions are readable to models, leaving them as they are is perhaps not the best choice. Models are not the best at differentiating between words and identifying whether or not they are actually all that different (for example, a model may think that can't and cannot are not the same word and may assign different meanings to each one) and thus it becomes important to us to handle any such variations in words. This can be achieved by making a 'dictionary' of sorts that contains each contraction and its meaning and then using said 'dictionary' in conjunction with a specialized python library that automatically handles contractions based on the definitions we've assigned them in the 'dictionary'.

- ### Tokenization

  This is the process where we split sentences into separate words. This is done using a specialized library which then returns a list. This is to help us with the next few steps.

- ### Removing Stopwords

  Stopwords are repetitive words in a sentence, such as articles (a, an, the). Removing them is important as the repetitiveness may skew the predictive judgement of the model(s).

- ### Lemmatization

  This step helps return words to their original forms. This is for the exact same reason why we remove contractions.

- ### Text Vectorization with TFIDF

  The TFIDF is a technique that turns our text into numbers that makes it easier for our model to process. It can be easy to come out on the other end of this with more features than is preferred thus we employed a few methods to ensure that our dimensionality does not get out of hand. First, we set a limit of 1000 in the TFIDF function itself. We then employed our second method.

- ### Dimensionality Reduction using SVD

  This method further reduces the dimensionality by setting a value, we chose 35 components, inside the method parameter ensuring that our features don't get out of hand.

▪ Sentiment Analysis

Some data can be analyzed for tone. Sentiment analysis helps us determine how a certain piece of text may sound to a reader. By analyzing certain columns for sentiment (positive or negative) we help determine how they may sound to guests. This technique was applied on the *interaction, host_about, and neighborhood_overview* columns through the library Text-Blob.

▪ Standardizing Short Text Columns

This technique helps us ensure the uniformity and clarity of short pieces of text by cleaning up messy data entries. This can be done by merging combo words (like super host into superhost), returning acronyms back to their original forms, applying proper capitalization on everything, etc…

This technique was applied *to host_name, host_location, host_neighbourhood, street, neighbourhood, neighbourhood_cleansed, city, state, market, smart_location, country_code, country, property_type,* and *room_type* columns.

For case insensitivity, we turn the newly uniform text to lower case.

▪ Encoding

Encoding is the process where we turn text data into numerical data by setting values to certain words. We employed numerous types:

- #### Label Encoder

  Columns such as *host_neighbourhood, street, neighbourhood, neighbourhood_cleansed, city, state, market, smart_location, country_code, country, property_type, host_name*, and *room_type* were encoded using the standard label encoder.

- #### Binary Mapping of Boolean Columns

  For Boolean columns, such as *host_has_profile_pic, host_identity_verified, is_location_exact, requires_license, instant_bookable, is_business_travel_ready, require_guest_phone_verification*, and *host_is_superhost* , we used binary encoding were t(true) -> 1 and f (false) -> 0.

  The only exception to the rule here is the *require_guest_profile_picture* column where we determined that it was better to not require that the guest has a profile picture. Thus, for this column alone, the true value was set to 0 while the false was set to 1.

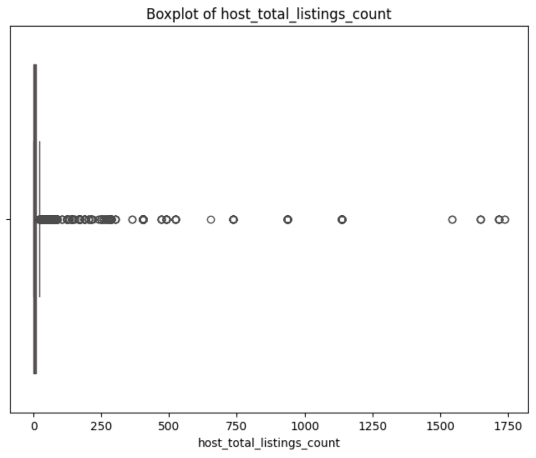
- #### Mapping Categorical Text Data to Numerical Values with Rank

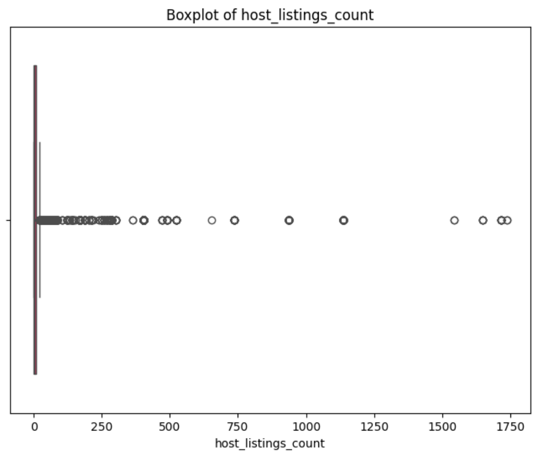
  Some columns have values that can be encoded and ranked based on desirability (with a greater value indicating better rank). Those columns are *host_response_time, bed_type,* and *cancellation_policy*.

▪ Handling Outliers Using IQR

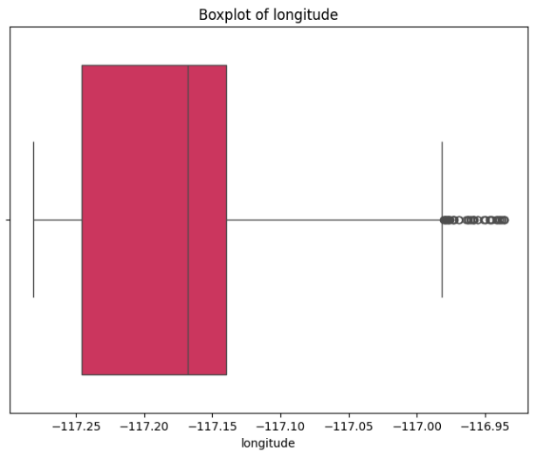
IQR is popular method for handling outliers in machine learning. We use the method to check for outliers. Once we've identified our outliers we then perform the equation needed to replace them with more suitable values for our dataset. The columns that we checked for outliers were: *host_total_listings_count, host_listings_count, latitude, longitude*, *accommodates, bathrooms, bedrooms, beds, guests_included,* *minimum_nights, maximum_nights, number_of_reviews, number_of_stays,* and *review_scores_rating*.

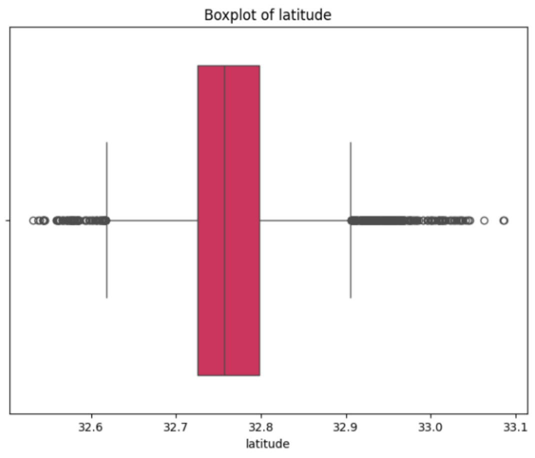
We can see just how many outliers are in each of the afflicted columns through the following visualizations:

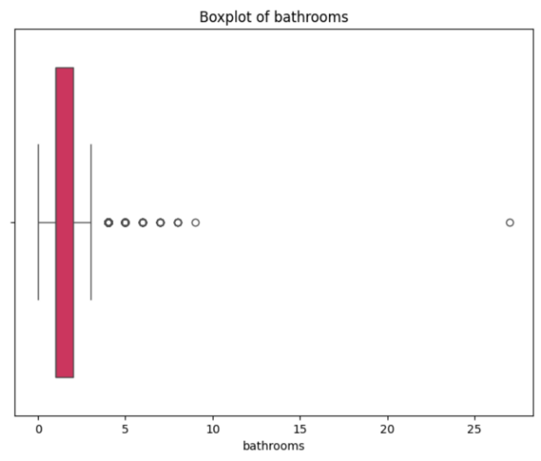
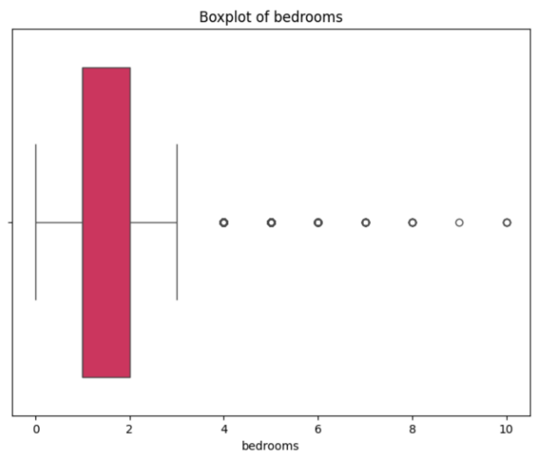
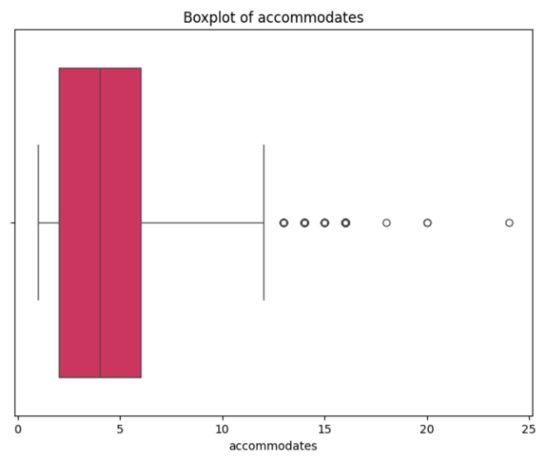
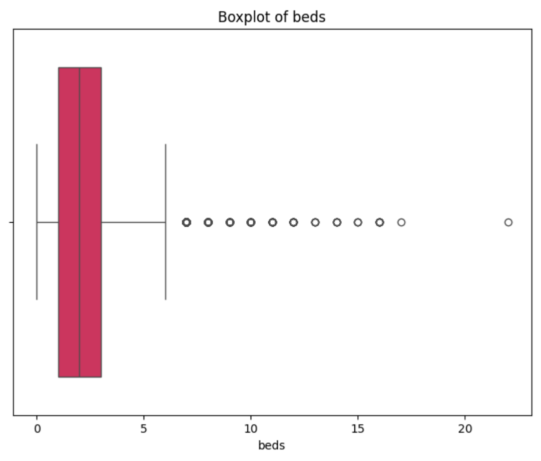
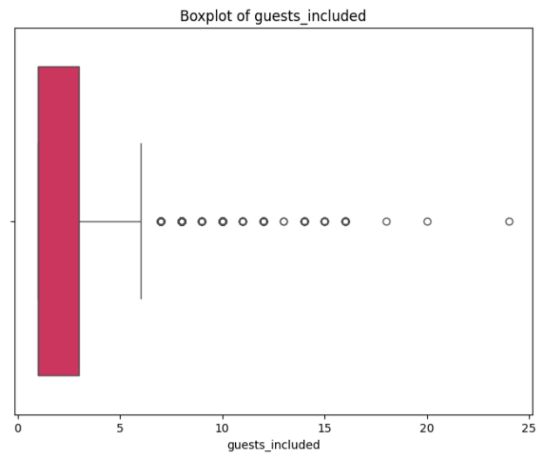
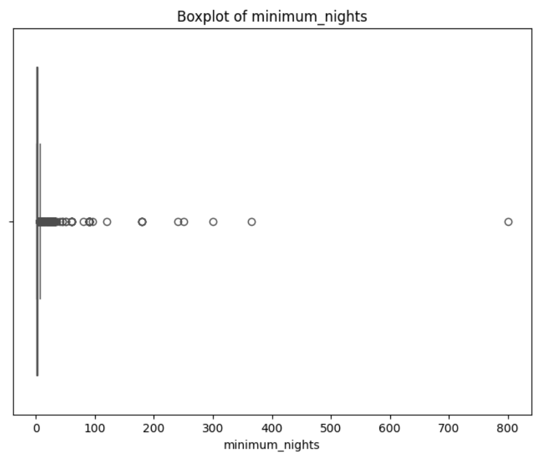


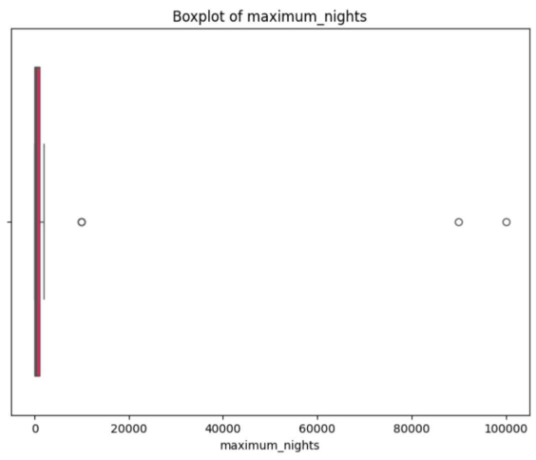


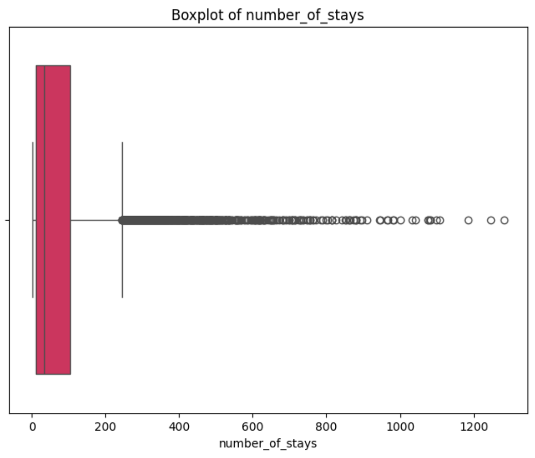
G

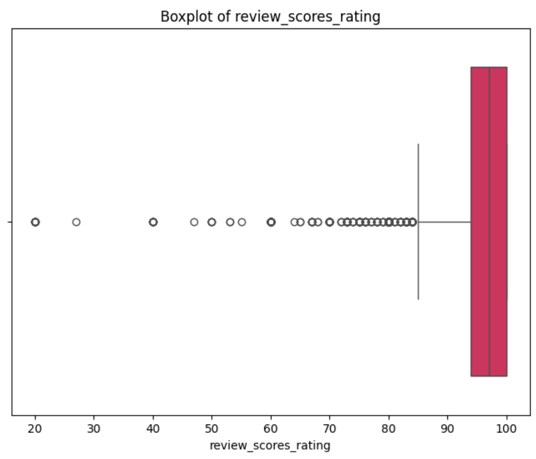


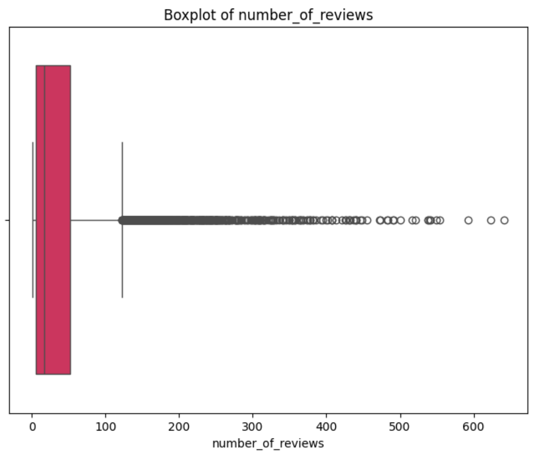



Boxplot of bathrooms
0
5
10
15
20
25
bathrooms

Boxplot of bedrooms
0
2
4
6
8
10
bedrooms

Boxplot of accommodates
0
5
10
15
20
25
accommodates

Boxplot of beds
0
5
10
15
20
beds

Boxplot of guests_included
0
5
10
15
20
25
guests_included

Boxplot of minimum_nights
0
100
200
300
400
500
600
700
800
minimum_nights









Feature Engineering

Feature engineering is the process by which we add columns, or features, to the dataset through a variety of techniques. For the purposes of our dataset and our project we have employed several of those techniques with the hopes of providing more varied data for our model(s) to train on.

There are three main techniques that we used here: Web Scraping, splitting the content of certain columns into new columns, and extracting features from date columns.

- ### Web Scraping

We were provided with URLs in our dataset (for both listings and hosts) that we used for the process. The URLs were, of course, from Airbnb and thus all our data was scraped from there. As some websites, Airbnb included, ban certain IP addresses to guard against suspicious activity (i.e. rapid viewing of multiple pages in a certain amount of time) the web scraping process was incredibly slow.

We scraped for two columns: The first column, called *host_ratings*, contained the host ratings. This was a feature we felt would be relevant to the guest's level of security when booking a lodging and thus would contribute to their overall satisfaction. We used the web scraping library Beautiful Soup for this and accomplished it using the column *host_url*.

The second column, called *guest_favorite*, was a Boolean column stating whether a certain listing was a guest favorite or not. Lodgings with the guest favorite stamp were marked as true and thus their value set to 1 while lodgings that were not guest favorites were marked as false and thus their value was set to 0. As Beautiful Soup could not detect this feature we had to switch to a different library, Selenium, to scrape for the URLs provided in the *listing_url* column. Selenium, however, is very slow and this process took us 8 hours to complete.

- ### Splitting Columns into More Features

Our dataset contained columns that contained entries that could be separated into different categories. The *amenities* column is one such column. It contained lists of amenities that a host provides within the lodging they put on the website (Ex: WiFi, working areas, pet(s) allowed, etc...).

Preprocessing such a column on its own without any modification is difficult so we cleaned up the data using standard methods then elected to sort each of those amenities into different categories (Ex: the Home Appliances category contains the home appliances, and the Luxury category contains items that may be considered luxuries). We then counted the amount of amenities provided in each category per listing and set that as the value for those new columns.

Another column this is performed on is the *host_location* column. Unlike with the amenities we did not create new categories to group the contents of this column in. Rather, we split it up into three new features: *host_country, host_state,* and *host_country.* After each feature was reliably split we performed standard label encoding to each of them.

▪ Extracting New Features from Date Columns

Our dataset contains several date columns. Each of those were used to extract different new features. For example, the column *host_since* was used to extract the column *years_active*. This was achieved by performing a certain calculation that estimated the years of host activity from the date in which they joined the platform. Guests may feel more positively inclined towards those they feel are more experienced hosts and thus we felt that including such a column that calculated how exactly long a host was on the website was a necessity.

Another example of this is the reviews_per_day column, which counted the frequency with which certain lodgings got reviews based on the *first_review, the last_review,* and *the number_of_reviews* column.

We also managed to extract a new feature, *total_cost*, from the columns *price_per_stay, cleaning_fee*, and *security_deposit*.

Feature Selection

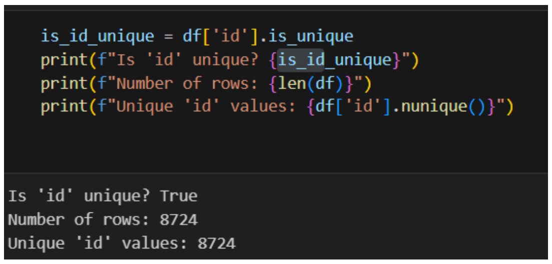
We are at the stage where we decide what features are going to go through to our model(s) for the training and testing.

- ### Dropping Unnecessary Columns

Before we can start correlating our features, we must first evaluate the relevance of each of them to the data.

Columns, like the listing_url, and the hosting_url serve a temporary purpose in the code. The URL columns helped us scrape for more relevant features but are otherwise irrelevant as far as model training goes. They provide no relevance to the target variable nor are they even tangentially related to the rest of the data.

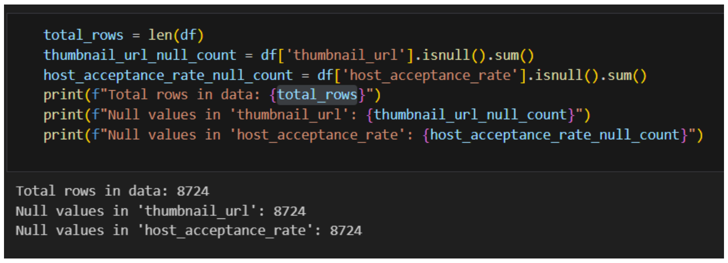
The id column is completely unique and thus provides no meaningful patterns for the model to learn on.



```
is_id_unique = df['id'].is_unique
print(f"Is 'id' unique? {is_id_unique}")
print(f"Number of rows: {len(df)}")
print(f"Unique 'id' values: {df['id'].nunique()}")
```

```
Is 'id' unique? True
Number of rows: 8724
Unique 'id' values: 8724
```

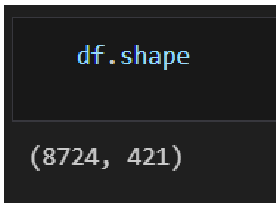
There are also columns that are completely null values. There in structure but not in value. Those columns, *thumbnail_url* and *host_acceptance_rate*, as such have been dropped along with the rest of the aforementioned columns.



```
total_rows = len(df)
thumbnail_url_null_count = df['thumbnail_url'].isnull().sum()
host_acceptance_rate_null_count = df['host_acceptance_rate'].isnull().sum()
print(f"Total rows in data: {total_rows}")
print(f"Null values in 'thumbnail_url': {thumbnail_url_null_count}")
print(f"Null values in 'host_acceptance_rate': {host_acceptance_rate_null_count}")
```

```
Total rows in data: 8724
Null values in 'thumbnail_url': 8724
Null values in 'host_acceptance_rate': 8724
```

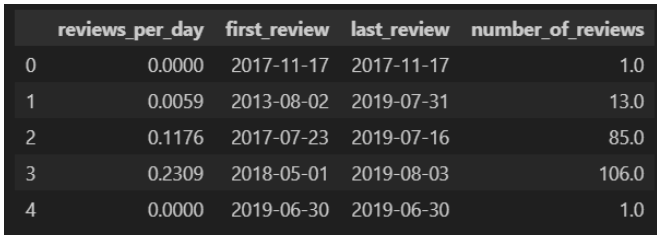
There are columns from which we extracted other features and have finally served their purpose, providing no more use to the model(s) training process beyond redundancy. Categorical columns like the *summary, space, transit, access, description, notes,* and *house_rules* are long text fields where the relevant parts have been extracted using the TFIDF and thus serve no more purpose.



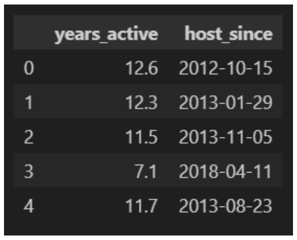
```
df.shape
```

```
(8724, 421)
```

There are Date columns like *the host_since, first_review*, and *last_review* where we extracted all that can be extracted from them and thus expended their use.

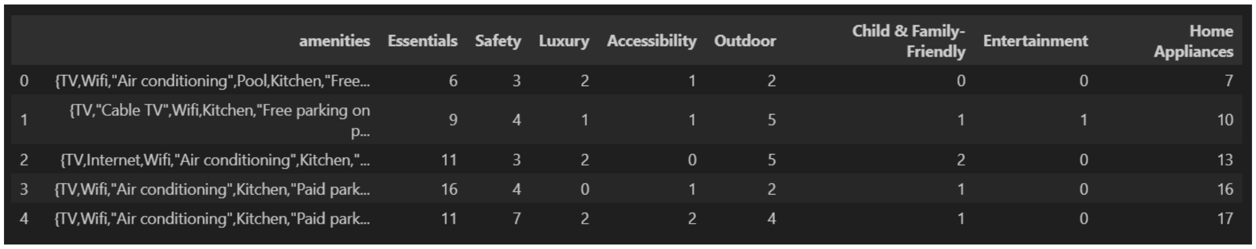


	reviews_per_day	first_review	last_review	number_of_reviews
0	0.0000	2017-11-17	2017-11-17	1.0
1	0.0059	2013-08-02	2019-07-31	13.0
2	0.1176	2017-07-23	2019-07-16	85.0
3	0.2309	2018-05-01	2019-08-03	106.0
4	0.0000	2019-06-30	2019-06-30	1.0

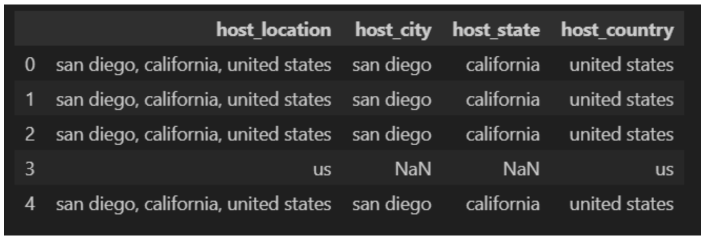


	years_active	host_since
0	12.6	2012-10-15
1	12.3	2013-01-29
2	11.5	2013-11-05
3	7.1	2018-04-11
4	11.7	2013-08-23

Columns like the *amenities* and the *host_location* have already been used to extract more features and as such keeping them around is not necessary as they provide no more use and have served their purpose.

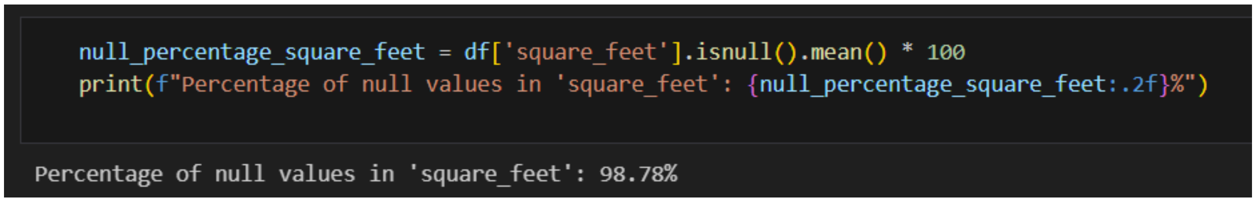


	amenities	Essentials	Safety	Luxury	Accessibility	Outdoor	Child & Family-Friendly	Entertainment	Home Appliances
0	{TV,Wifi,"Air conditioning",Pool,Kitchen,"Free...	6	3	2	1	2	0	0	7
1	{TV,"Cable TV",Wifi,Kitchen,"Free parking on p...	9	4	1	1	5	1	1	10
2	{TV,Internet,Wifi,"Air conditioning",Kitchen,"...	11	3	2	0	5	2	0	13
3	{TV,Wifi,"Air conditioning",Kitchen,"Paid park...	16	4	0	1	2	1	0	16
4	{TV,Wifi,"Air conditioning",Kitchen,"Paid park...	11	7	2	2	4	1	0	17



	host_location	host_city	host_state	host_country
0	san diego, california, united states	san diego	california	united states
1	san diego, california, united states	san diego	california	united states
2	san diego, california, united states	san diego	california	united states
3	us	NaN	NaN	us
4	san diego, california, united states	san diego	california	united states

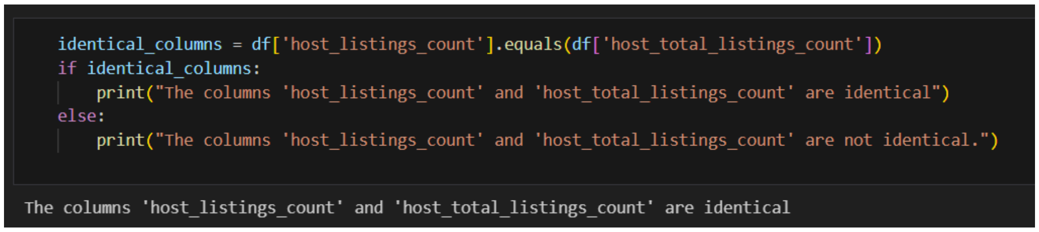
Columns like the *square_feet* column were not completely null but lacked so much data that they might as well have been. Such columns are not reliable and as such not a good enough fit for training.



```
null_percentage_square_feet = df['square_feet'].isnull().mean() * 100
print(f"Percentage of null values in 'square_feet': {null_percentage_square_feet:.2f}%")
```

```
Percentage of null values in 'square_feet': 98.78%
```

The *host_listings_count* is a redundant column as there is a better and more complete version of each one, in the form of *host_total_listings_count* that provide a much clearer look on the same aspects those two columns handle. Thus, they were dropped in favor of their better counterparts



```
identical_columns = df['host_listings_count'].equals(df['host_total_listings_count'])
if identical_columns:
    print("The columns 'host_listings_count' and 'host_total_listings_count' are identical")
else:
    print("The columns 'host_listings_count' and 'host_total_listings_count' are not identical.")
```

```
The columns 'host_listings_count' and 'host_total_listings_count' are identical
```

All of those columns are thus either redundant, irrelevant, or just provide no purpose to the goal of this project. As such all of them have been dropped in favor of more relevant data which will undergo a more thorough feature selection process to determine which ones are best suited to the purposes of our models.

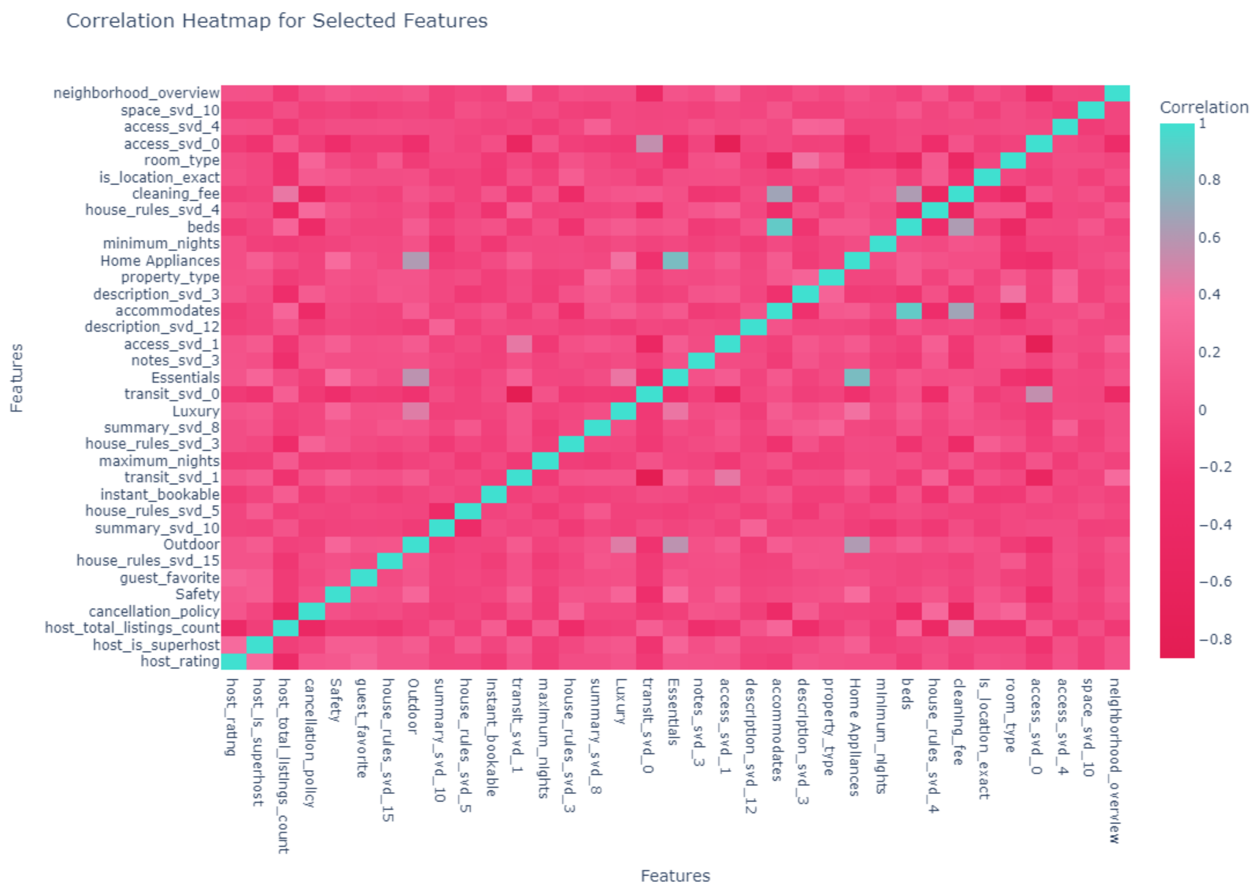
- ## Correlation

For the actual feature selection technique in this part of the project, we attempted different techniques. We found out rather quickly that the best technique for our dataset was the classic correlation.

We tried a variety of values in our correlation method and, eventually, came to the conclusion that passing the first 35 columns of the dataset was the best option with respect to the modeling accuracy.

The columns used are:

TF-IDF/SVD	**Sentiment**	**Normal**	**Scraping**	**Amenities**
house_rules_svd_15	neighborhood_overview	host_is_superhost	host_rating	Safety
summary_svd_10		host_total_listings_count	guest_favorite	Outdoor
house_rules_svd_5		cancellation_policy		Luxury
transit_svd_1		instant_bookable		Essentials
house_rules_svd_3		maximum_nights		Home Appliances
summary_svd_8		accommodates		
transit_svd_0		property_type		
notes_svd_3		minimum_nights		
access_svd_1		beds		
description_svd_12		cleaning_fee		
description_svd_3		is_location_exact		
house_rules_svd_4		room_type		
access_svd_0				
access_svd_4				
space_svd_10				



Model Training and Selection

Now that all our data has been preprocessed and carefully selected in order of relevance to the target variable we can start passing it to our model(s).

Our features (X): the 35 columns we got from the correlation.

Our target (Y): the *review_scores_rating* column.

- Train and Test Split
  - 80% train
  - 20% test
  - Random State = 0

Feature Scaling

Three feature scaling methods were tried: the Min-Max Scaler, Standard Scaler, and the Robust Scaler. After multiple rounds of trial and error it was determined that the one with the best effect on the models among the three was the Robust Scaler, thus our feature scaling is conducted using the Robust Scaler.

The Regression Metrics Used

- R^2 Score
- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- Mean Absolute Percentage Error (MAPE)

Linear Models

Note: Cross Validation has been applied on all the models that do not use the grid search.

1. Linear Regression

Cross Validation = 5

- **Cross-Validation R2 Scores:** [0.21456857 0.21958834 0.21833478 0.23624945 0.2542602]
- **Average Cross-Validation R2 Score:** 0.2286

- **Training R2 Score:** 0.2302
- **Test Set Metrics:**
  - **R2 Score:** 0.2589
  - **Mean Absolute Error (MAE):** 2.8767
  - **Mean Squared Error (MSE):** 14.3988
  - **Root Mean Squared Error (RMSE):** 3.7946
  - **Mean Absolute Percentage Error (MAPE):** 0.0307

2. Lasso Regression

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Best parameters (Lasso):** {'alpha': 0.001}
- **Best CV score (R^2) (Lasso):** 0.2213
- **Training R^2 Score:** 0.2302
- **Test Set Metrics (Lasso):**
  - **R2 Score:** 0.2586
  - **Mean Absolute Error (MAE):** 2.8771
  - **Mean Squared Error (MSE):** 14.4032
  - **Root Mean Squared Error (RMSE):** 3.7952
  - **Mean Absolute Percentage Error (MAPE):** 0.0307

3. Ridge Regression

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Best parameters (Ridge):** {'alpha': 46.41588833612773
- **Best CV score (R^2) (Ridge):** 0.2214
- **Training R^2 Score:** 0.2301
- **Test Set Metrics (Ridge):**
  - **R2 Score:** 0.2587
  - **Mean Absolute Error (MAE):** 2.8773
  - **Mean Squared Error (MSE):** 14.4016
  - **Root Mean Squared Error (RMSE):** 3.7949
  - **Mean Absolute Percentage Error (MAPE):** 0.0307

4. Elastic Net

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Best parameters (ElasticNet**): {'alpha': 0.021544346900318832, 'l1_ratio': 0.3333333333333333}
- **Best CV score (R^2) (ElasticNet):** 0.2215
- **Training R^2 Score:** 0.2294
- **Test Set Metrics (ElasticNet):**
  - **R2 Score:** 0.2575
  - **Mean Absolute Error (MAE):** 2.8805
  - **Mean Squared Error (MSE):** 14.4258
  - **Root Mean Squared Error (RMSE):** 3.7981
  - **Mean Absolute Percentage Error (MAPE):** 0.0307

Support Vector Regression (SVR)

1. Elastic Net

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Best parameters (SVR):** {'C': 3.1622776601683795, 'epsilon': 1.0, 'kernel': 'rbf'}
- **Best CV score (R^2) (SVR):** 0.1861
- **Training R^2 Score:** 0.2324
- **Test Set Metrics (SVR):**
  - **R2 Score:** 0.2353
  - **Mean Absolute Error (MAE):** 2.7171
  - **Mean Squared Error (MSE):** 14.8560
  - **Root Mean Squared Error (RMSE):** 3.8544
  - **Mean Absolute Percentage Error (MAPE):** 0.0293

Ensemble Models

1. Bagging Models

Random Forest

Grid search is applied here for hyperparameter tuning. Cross Validation = 3.

- **Best parameters (RandomForest):** {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 100}
- **Best CV score (R^2) (RandomForest):** 0.2258
- **Training R^2 Score:** 0.5070
- **Test Set Metrics (RandomForest):**
  - **R2 Score:** 0.2733
  - **Mean Absolute Error (MAE):** 2.8008
  - **Mean Squared Error (MSE):** 14.1187
  - **Root Mean Squared Error (RMSE):** 3.7575
  - **Mean Absolute Percentage Error (MAPE):** 0.0299

Decision Tree (Worst R^2 Score)

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Best parameters (DecisionTree):** {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5}
- **Best CV score (R^2) (DecisionTree):** 0.0680
- **Training R^2 Score:** 0.4434
- **Test Set Metrics (DecisionTree):**
  - **R2 Score:** 0.1178
  - **Mean Absolute Error (MAE):** 3.0066
  - **Mean Squared Error (MSE):** 17.1391
  - **Root Mean Squared Error (RMSE):** 4.1399
  - **Mean Absolute Percentage Error (MAPE):** 0.0321

2. Boosting Models

A new method of hyperparameter tuning, Optuna, is used for the Boosting Models as Grid Search proved to be inefficient. Optuna provided good results, however the results changed with every run thus, after acquiring the best R^2 score from between the runs the Optuna method has been commented in the project code.

XGBoost

- **Cross-Validation R^2 Scores:** [0.24719736 0.22765174 0.25276748 0.25258664 0.27623625]
- **Average Cross-Validation R^2 Score:** 0.2513
- **Training R^2 Score:** 0.3676
- **Test Set Metrics (XGBoost):**
    - **R2 Score:** 0.2835
    - **Mean Absolute Error (MAE):** 2.7730
    - **Mean Squared Error (MSE):** 13.9200
    - **Root Mean Squared Error (RMSE):** 3.7310
    - **Mean Absolute Percentage Error (MAPE):** 0.0296

CatBoost

Cross Validation = 5

- **Cross-Validation R^2 Scores:** [0.26211435 0.24339096 0.26098843 0.26117811 0.28801344]
- **Average Cross-Validation R^2 Score:** 0.2631
- **Training R^2 Score:** 0.3505
- **Test Set Metrics (CatBoost):**
    - **R2 Score:** 0.2999
    - **Mean Absolute Error (MAE):** 2.7538
    - **Mean Squared Error (MSE):** 13.6022
    - **Root Mean Squared Error (RMSE):** 3.6881
    - **Mean Absolute Percentage Error (MAPE):** 0.0294

- *LightGBM*
    - **Cross-Validation R^2 Scores:** [0.24765788 0.23894274 0.25249208 0.25552156 0.28740055]
    - **Average Cross-Validation R^2 Score:** 0.2564
    - **<u>Training R^2 Score:</u>** 0.3727
    - **<u>Test Set Metrics (LightGBM):</u>**
        - **R2 Score:** 0.2914
        - **Mean Absolute Error (MAE):** 2.7668
        - **Mean Squared Error (MSE):** 13.7674
        - **Root Mean Squared Error (RMSE):** 3.7104
        - **Mean Absolute Percentage Error (MAPE):** 0.0295

3. Stacking Model

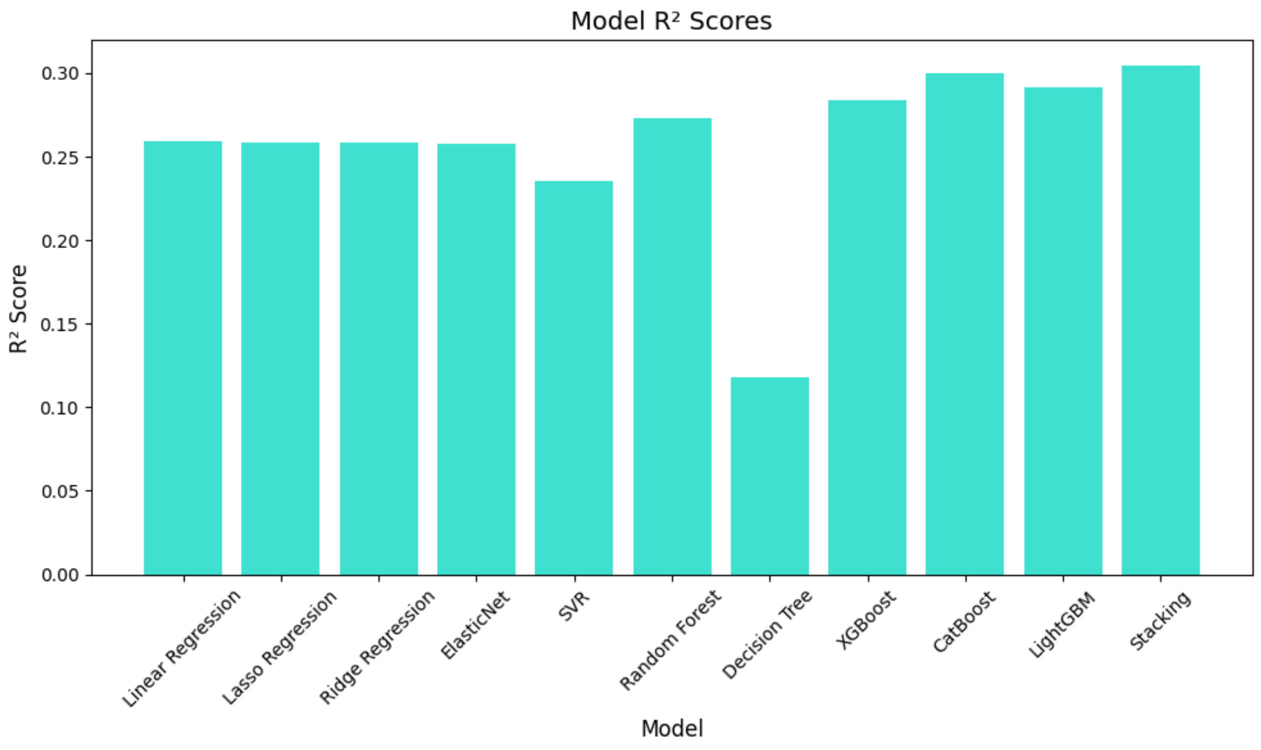
A stacking model is a model that uses multiple machine learning models in conjunction with one another to maximize its performance. For the purposes of our project we elected to use both the CatBoost and LightGBM as the base models along with the XGBoost model as the meta model.

- *Stacking Regressor Model with CatBoost, LightGBM, and XGBoost (Best R^2 Score)*
    - **<u>Training R^2 Score:</u>** 0.3543
    - **<u>Test Set Metrics (Stacking):</u>**
        - **R2 Score: 0.3044**
        - **Mean Absolute Error (MAE):** 2.7441
        - **Mean Squared Error (MSE):** 13.5134
        - **Root Mean Squared Error (RMSE):** 3.6761
        - **Mean Absolute Percentage Error (MAPE):** 0.0293

Model R^2 Comparison

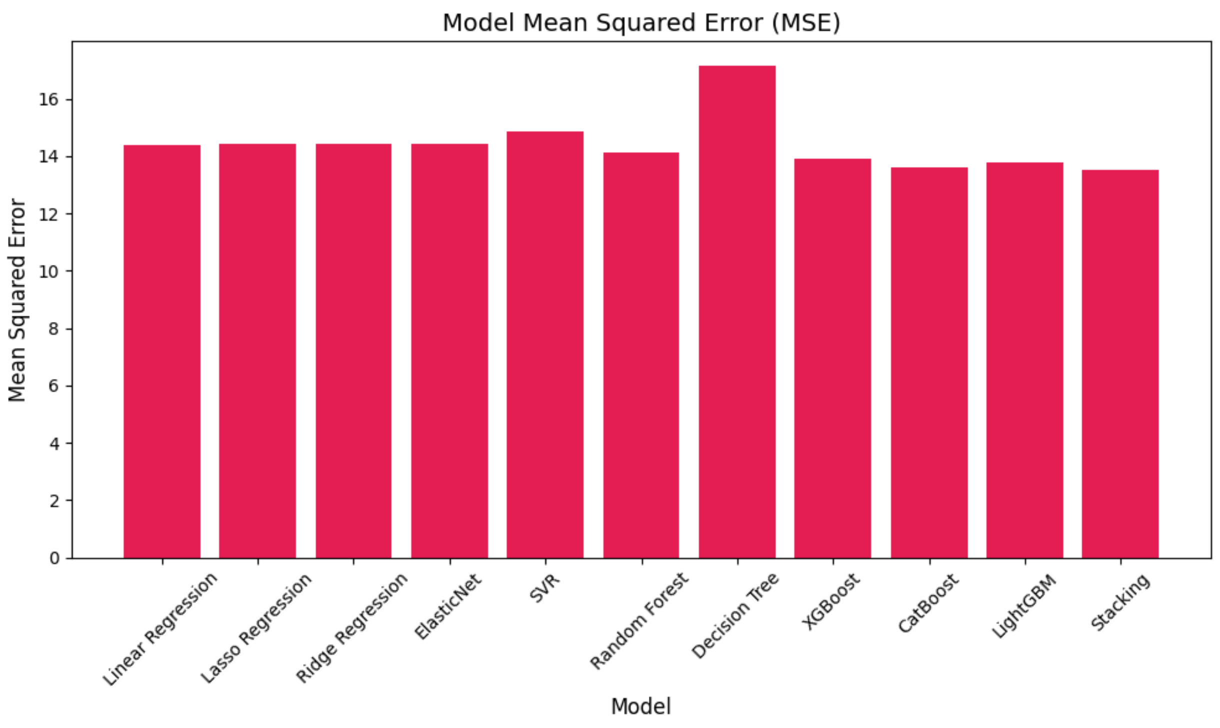
R^2 Comparison Table

Index	Model	R^2 Score
0	Linear Regression	0.2589
1	Lasso Regression	0.2586
2	Ridge Regression	0.2587
3	ElasticNet	0.2575
4	SVR	0.2353
5	Random Forest	0.2733
6	Decision Tree	0.1178
7	XGBoost	0.2835
8	CatBoost	0.2999
9	LightGBM	0.2914
10	Stacking	0.3044



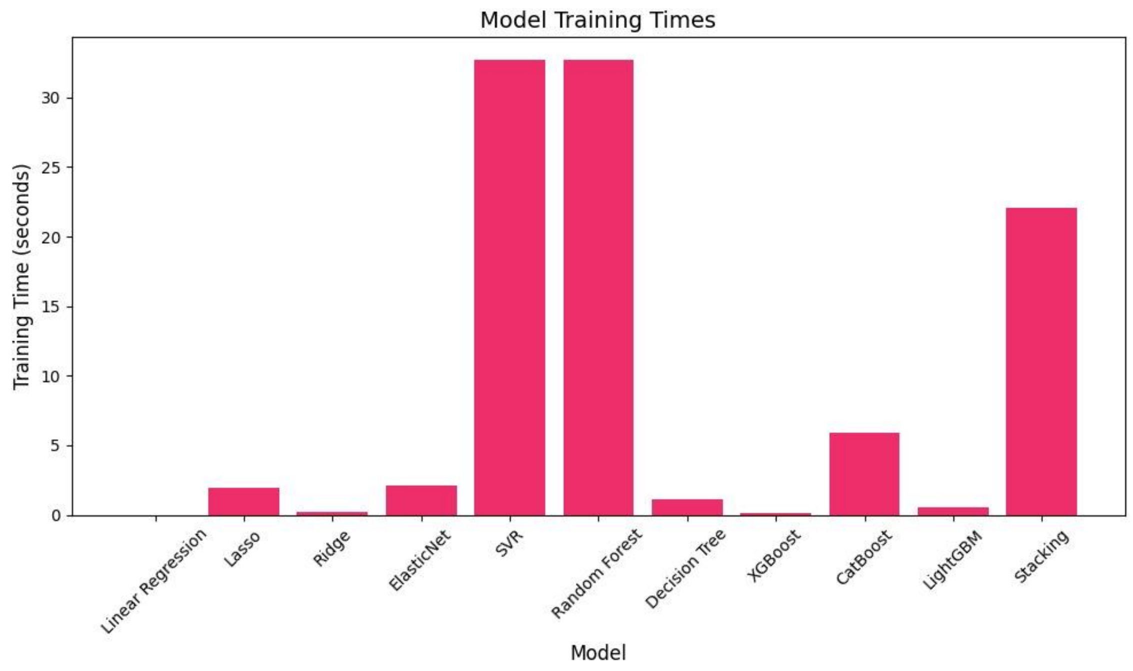
MSE Comparison Table

s Index	Model	Mean Squared Error (MSE)
0	Linear Regression	14.3988
1	Lasso Regression	14.4032
2	Ridge Regression	14.1016
3	ElasticNet	14.4258
4	SVR	14.8560
5	Random Forest	14.1187
6	Decision Tree	17.1391
7	XGBoost	13.9200
8	CatBoost	13.6022
9	LightGBM	13.7674
10	Stacking	13.5134

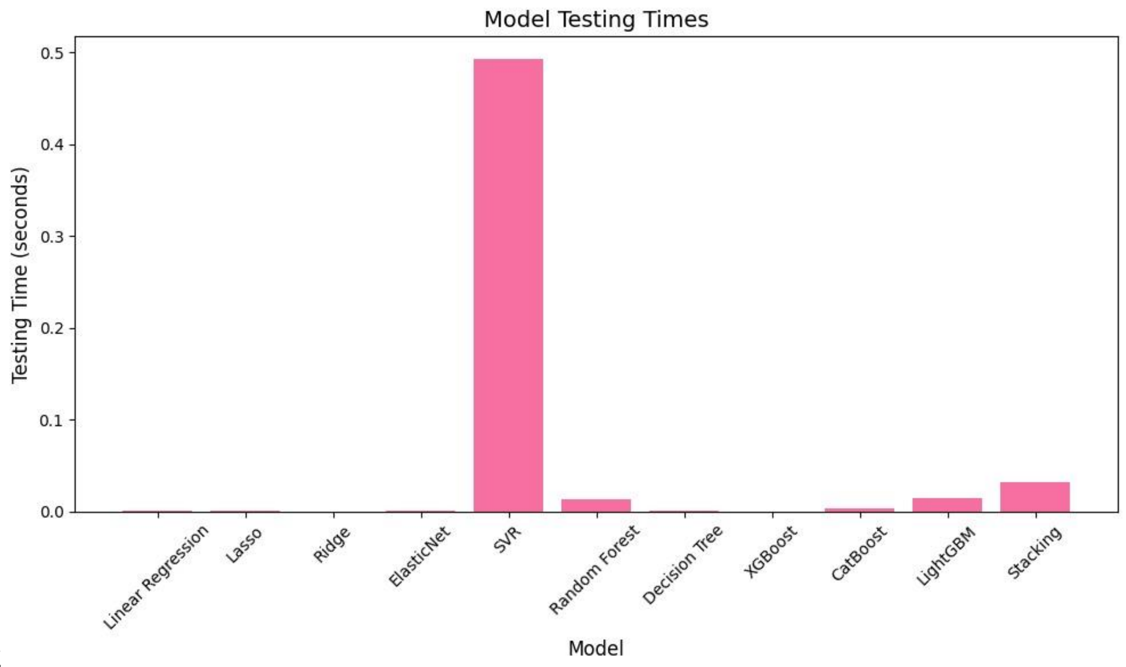


Model Time Comparison

Train Time Comparison Graph



Test Time Comparison Graph



Conclusion

The Stacking Model performed the best out of all the other models with an R^2 score of **0.3044**. This means that it explained the most variance in the target variable among all the models.

Following close behind were the Booster Models, with CatBoost and LightGBM performing strongly with R^2 scores of **0.2999** and **0.2914** respectively. XGBoost performed along the same vein though its R^2 score was **0.2835.**

Traditional Linear Models did not perform as well though their scores were not bad (~**0.26**) still, it showed that for this type of data using linear models may not be the best choice.

Random Forest slightly outperformed the Linear Models though it is not by much (**0.2733**). This shows that while still not ideal, Random Forest is a better choice for this type of dataset than the Linear Models.

The two worst R^2 scores are held by both the SVR Model and the Decision Tree Model. The SVR model performed worse than the Linear Models with an R^2 score of **0.2553** showing that using SVM-based approaches for this dataset is not the best idea, and though its R^2 score is far from desirable, it has nothing on the score produced by the Decision Tree, a disappointing **0.1178**.

Such a score suggests a severe underperformance which could be a result of the Decision Tree failing to capture all the underlying patterns in the data as effectively as the other models are.

Thus, we can finally conclude that Ensemble (Boosting) and Stacking models significantly outperform traditional regression methods and simpler methods in this dataset.

Classification

As for the preprocessing and Feature Engineering steps, they were the same as the regression preprocessing steps except we encoded target column **Guest Satisfaction** by rank with Very High being encoded as 2, High encoded as 1 and average encoded as 0 and we also changed number of SVD components from 35 to **50**

- ### Feature Selection

Discrete Features

We Tried two Feature Selection approaches **Chi Squared** and **Mutual Information.**

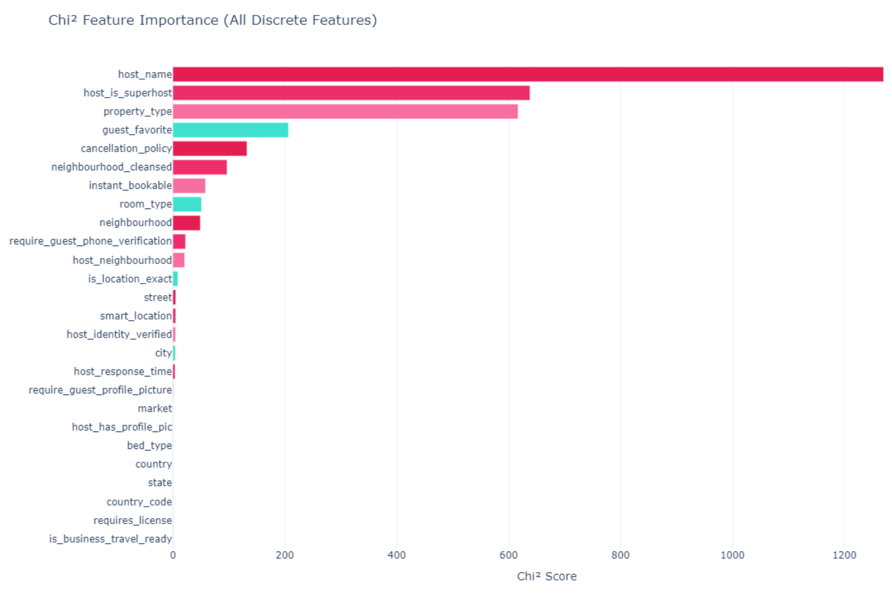
After comparing between Both discrete approaches of feature selection we came to conclude that **Chi Squared** performs better than Mutual Information

After multiple trials using Chi Squared we concluded that the best threshold is P_Value < 0.5

Top chi squared features

[host_name,host_is_superhost,property_type,guest_favorite,cancellation_policy,neighbourhood_cleansed, instant_bookable,room_type,neighbourhood,require_guest_phone_verification,host_neighbourhood, is_location_exact]

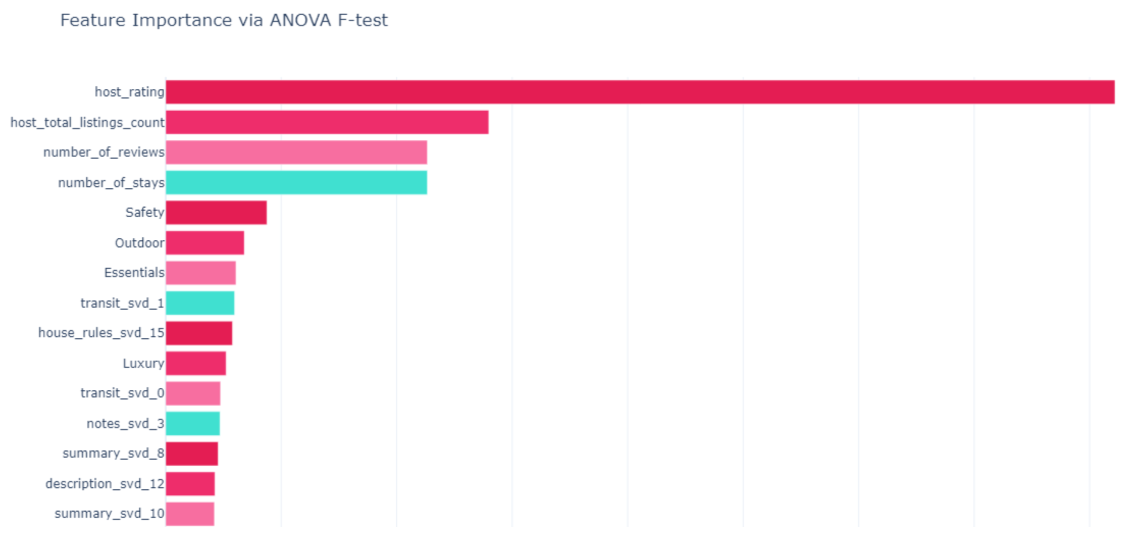
Chi² Feature Importance Graph



Continuous Features

For Continuous Features we used Anova with Threshold = 50

Top Features using Anova:
[host_rating,host_total_listings_count,number_of_reviews,number_of_stays,Safety,Outdoor,Essentials, transit_svd_1,house_rules_svd_15,Luxury]



The columns used are:

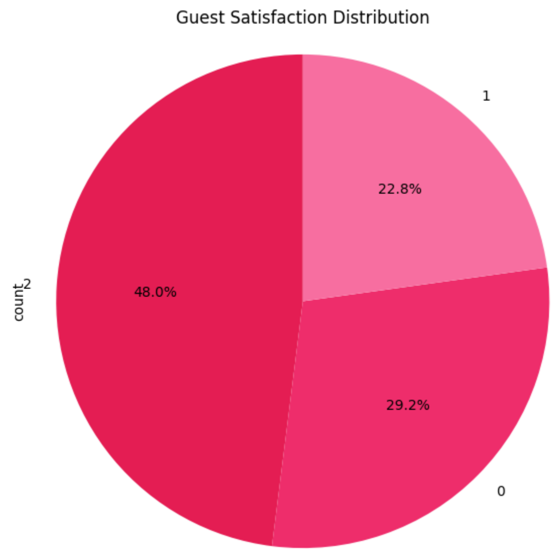
TF-IDF/SVD	Sentiment	Normal	Scraping	Amenities
house_rules_svd_15	neighborhood_overview	property_type	host_rating	Luxury
transit_svd_1		number_of_stays	guest_favorite	Outdoor
		host_is_superhost		Essentials
		is_location_exact		
		number_of_reviews		
		room_type		
		cancellation_policy		
		neighbourhood_cleansed		
		neighbourhood		
		host_name		
		is_location_exact		
		host_neighbourhood		
		host_total_listings_count		
		instant_bookable		
		require_guest_phone_verification		

Feature Scaling

Three feature scaling methods were tried: the Min-Max Scaler, Standard Scaler, and the Robust Scaler. After multiple rounds of trial and error it was determined that the one with the best effect on the models among the three was the Robust Scaler, thus our feature scaling is conducted using the Robust Scaler.

Imbalanced Classes

After scaling features using Robust Scaler, we found out that Guest Satisfaction is Imbalanced as illustrated below:



So we handled the imbalance using **SMOTE** in order to maximize accuracy.

▪ Model Training and selection

Now that all our data has been preprocessed and carefully selected in order of relevance to the target variable we can start passing it to our model(s).

Our features (X): the 22 columns we got from feature selection.

Our target (Y): the *guest satisifaction* column.

- ## Train and Test Split
  - 80% train
  - 20% test
  - Random State = 0
- ## The Classification Metrics Used
  - Accuracy
- ## Linear Models

Note: Cross Validation has been applied on all the models that do not use the grid search.

1. Logistic Regression

Cross Validation = 5

- **Cross-Validation Accuracy Scores:** [0.52148997 0.52722063 0.51805158 0.51747851 0.51834862]
- **Average Cross-Validation Accuracy Score:** 0.5205
- **Training Time:** 0.1335 seconds
- **Testing Time:** 0.0000 seconds
- **Test Accuracy Score:** 0.5696

Classification Report (Logistic Regression):

	precision	recall	f1-score	support
0	0.55	0.70	0.61	510
1	0.39	0.40	0.40	386
2	0.69	0.57	0.62	849
accuracy			0.57	1745
macro avg	0.54	0.56	0.54	1745
weighted avg	0.58	0.57	0.57	1745

2. Support Vector Classifier (SVC)

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Hyperparameters Grid:** 'C': [1, 10, 100], 'kernel': ['rbf', 'poly', 'sigmoid']
- **Best Hyper parameters (SVC):** {'C': 100,'kernel': 'rbf'}
- **Best CV score (Accuracy) (SVC):** 0.6097
- **Training Accuracy Score (SVC):** 0.6606
- **Total Training Time (SVC):** 23.0768 seconds
- **Total Testing Time (SVC):** 0.7658 seconds
- **Test Accuracy Score (SVC):** 0.5828080229226361

Classification Report (SVC):

	precision	recall	f1-score	support
0	0.58	0.62	0.60	510
1	0.38	0.42	0.40	386
2	0.69	0.63	0.66	849
accuracy			0.58	1745
macro avg	0.55	0.56	0.56	1745
weighted avg	0.59	0.58	0.59	1745

Ensemble Models

1- Bagging models

1- Random Forest

Grid search is applied here for hyperparameter tuning. Cross Validation = 3.

- **Hyperparameters Grid:** n_estimators: [50, 100,200], max_depth: [None,10], min_samples_split: [2, 5,10], min_samples_leaf: [1, 2]
- **Best Hyperparameters (RandomForest):** {max_depth: None, min_samples_leaf: 1, min_samples_split: 2, n_estimators: 200}
- **Best CV score (Accuracy) (RandomForest):** 0.7181
- **Training Accuracy Score:** 0.9978
- **Total Training Time:** 21.2400 seconds
- **Total Testing Time:** 0.0562 seconds
- **Test Accuracy Score:** 0.6114613180515759

Classification Report (RandomForest):

	precision	recall	f1-score	support
0	0.59	0.60	0.59	510
1	0.43	0.36	0.39	386
2	0.69	0.74	0.71	849
accuracy			0.61	1745
macro avg	0.57	0.56	0.56	1745
weighted avg	0.60	0.61	0.61	1745

- *2- Decision Tree (Worst Accuracy Score)*

Grid search is applied here for hyperparameter tuning. Cross Validation = 5.

- **Hyperparameters Grid**: max_depth: [10, 20, 30, None], min_samples_split: [2, 5, 10], min_samples_leaf: [1, 2, 4], max_features: [None, 'sqrt', 'log2']
- **Best Hyperparameters (DecisionTree):** {max_depth: 20, max_features: None, min_samples_leaf: 1, min_samples_split: 5}
- **Best CV score (Accuracy) (DecisionTree):** 0.6279
- **Training Accuracy Score:** 0.9367
- **Total Training Time:** 3.4292 seconds
- **Total Testing Time:** 0.0000 seconds
- **Test Accuracy Score (DecisionTree):** 0.5432664756446991
- **Test Classification Report (DecisionTree):**

	precision	recall	f1-score	support
0	0.52	0.58	0.55	510
1	0.36	0.40	0.38	386
2	0.66	0.59	0.62	849
accuracy			0.54	1745
macro avg	0.52	0.52	0.52	1745
weighted avg	0.55	0.54	0.55	1745

2- Boosting models

A new method of hyperparameter tuning, Optuna, is used for the Boosting Models as Grid Search proved to be inefficient. Optuna provided good results, however the results changed with every run thus, after acquiring the best Accuracy score from between the runs the Optuna method has been commented in the project code.

1- XGBoost

- **Hyperparameters Grid:**
  n_estimators: range(50,200), max_depth': range(3,11)
  learning_rate: [0.01, 0.02, 0.05, 0.1, 0.15, 0.2],
  subsample: [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0],
  colsample_bytree: [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]
- **Best Hyperparameters:** n_estimators: 75, max_depth: 6,
  learning_rate: 0.10063569413953903,
  subsample: 0.7773356344475959,
  colsample_bytree:0.9268996994266611
- **Cross-Validation Accuracy Scores:** [0.6252149 0.62406877 0.64469914 0.64011461 0.64965596]
- **Average Cross-Validation Accuracy Score:** 0.6368
- **Training Accuracy Score:** 0.7998
- **Total Training Time (XGBoost):** 0.3306 seconds
- **Total Testing time:** 0.0024 seconds
- **Test Accuracy Score (XGBoost):** 0.6257879656160459
- **Test Set Classification Report (XGBoost):**

	precision	recall	f1-score	support
0	0.59	0.65	0.62	510
1	0.45	0.38	0.42	386
2	0.71	0.72	0.72	849
accuracy			0.63	1745
macro avg	0.59	0.58	0.58	1745
weighted avg	0.62	0.63	0.62	1745

2- CatBoost

Cross Validation = 5

- **Hyperparameters Grid:**
  iterations: range(50,200), depth': range(3,11) 'learning_rate': [0.01, 0.02, 0.05, 0.1, 0.15, 0.2],
  l2_leaf_reg: [1, 2, 3, 5, 7, 10], random_strength: [0.5, 1.0, 1.5, 2.0],
  bagging_temp: [0.0, 0.5, 1.0],
  Border_count: [32, 64, 128, 255]
- **Best Hyperparameters:** iterations: 156, depth: 5,
  learning_rate: 0.1594827161532743,l2_leaf_reg: 6.544844148450393,
  Random_strength: 1.7448624019847634,
  Bagging_temp: 0.5394298973201219,Border_count: 208
- **Cross-Validation R^2 Scores:** [0.6286533 0.6252149 0.63896848 0.63381089 0.6559633]
- **Average Cross-Validation Accuracy Score:** 0.6365
- **Training Accuracy Score:** 0.7413
- **Total Training Time:** 0.6713 seconds
- **Total Testing Time:** 0.0056 seconds
- **Test Accuracy Score:** 0.6332378223495702
- **Classification Report (CatBoost):**

	precision	recall	f1-score	support
0	0.61	0.66	0.63	510
1	0.46	0.39	0.42	386
2	0.72	0.73	0.72	849
accuracy			0.63	1745
macro avg	0.59	0.59	0.59	1745
weighted avg	0.63	0.63	0.63	1745

3- LightGBM

- **Hyperparameters Grid**: n_estimators : range(50,200), Max_depth: range(3,10) ,learning rate: [0.01, 0.02, 0.05, 0.1, 0.15, 0.2], Num_leaves: range(20,150),min_child_samples : range(10,50), subsample: [0.6, 0.7, 0.8, 0.9, 1.0], colsample_bytree: [0.6, 0.7, 0.8, 0.9, 1.0]
- **Best Hyperparameters**: n_estimators : 172,Max_depth: 5 ,
- learning rate: 0.04485202298566927 Num_leaves: 103,min_child_samples : 43, subsample: 0.6353937353165494, colsample_bytree: 0.8161981454506166
-
- **Cross-Validation R^2 Scores:** [0.63667622 0.63094556 0.64756447 0.64183381 0.65080275]
- **Average Cross-Validation R^2 Score:** 0.6416
- **<u>Training Accuracy Score:</u>** 0.7579
- **<u>Total Training Time:</u>** 0.4669 seconds
- **<u>Total Testing Time:</u>** 0.0088 seconds
- **<u>Test Accuracy Score:</u>** 0.6309455587392551
- Classification Report (LightGBM):

	precision	**recall**	**f1-score**	**support**
0	0.59	0.65	0.62	510
1	0.46	0.37	0.41	386
2	0.72	0.73	0.73	849
accuracy			0.63	1745
macro avg	0.59	0.59	0.59	1745
weighted avg	0.62	0.63	0.63	1745

2 Stacking Model

A stacking model is a model that uses multiple machine learning models in conjunction with one another to maximize its performance. For the purposes of our project we elected to use both the CatBoost and LightGBM as the base models along with the XGBoost model as the meta model.

As for hyperparameters for each model we simply use the best hyperparameters except for meta model we use simpler hyperparameters to avoid over-fitting

Stacking Regressor Model with CatBoost, LightGBM, and XGBoost (Best Accuracy Scores)

- **Training Accuracy Score:** 0.7253
- **Total Training Time:** 24.5682 seconds
- **Total Testing Time:** 0.0583 seconds
- **Test Accuracy Score:** 0.646999140114613
- **Classification Report (Stacking):**

```
              precision   recall  f1-score   support

         0      0.62       0.63     0.62       510
         1      0.51       0.34     0.41       386
         2      0.70       0.80     0.75       849
    accuracy                        0.65      1745

   macro avg   0.61       0.59      0.59      1745

weighted avg   0.63      0.65      0.63      1745
```

3 Voting Model

Voting Classifier Model with CatBoost, LightGBM, and XGBoost we use best hyperparameters of each model

- **Training Accuracy Score:** 0.7695
- **Total Training Time:** 24.5682 seconds
- **Total Testing Time:** 0.0207 seconds
- **Test Accuracy Score:** 0.633810888252149
- **Classification Report (Voting):**

```
              precision   recall  f1-score   support

           0     0.61       0.66     0.63       510
           1     0.47       0.39     0.42       386
           2     0.71       0.73     0.72       849

    accuracy                         0.63      1745

   macro avg     0.60       0.59     0.59      1745

weighted avg     0.63       0.63     0.63      1745
```

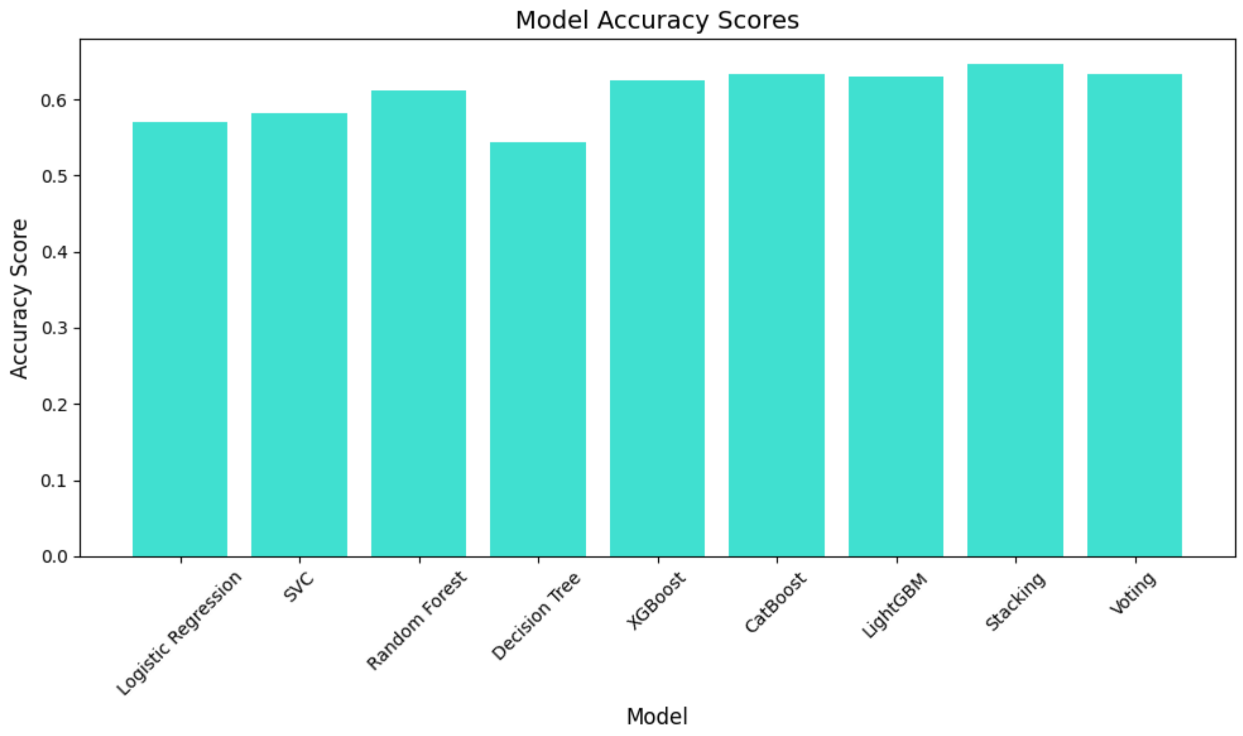
‘

▪ Model Accuracy Comparison

Accuracy Comparison Table

Index	Model	Accuracy Score
0	Logistic Regression	0.5696
1	SVC	0.5828
2	Random Forest	0.6115
3	Decision Tree	0.5433
4	XGBoost	0.6258
5	CatBoost	0.6332
6	LightGBM	0.6309
7	Stacking	0.6470
8	Voting	0.6338

Accuracy Comparison Graph

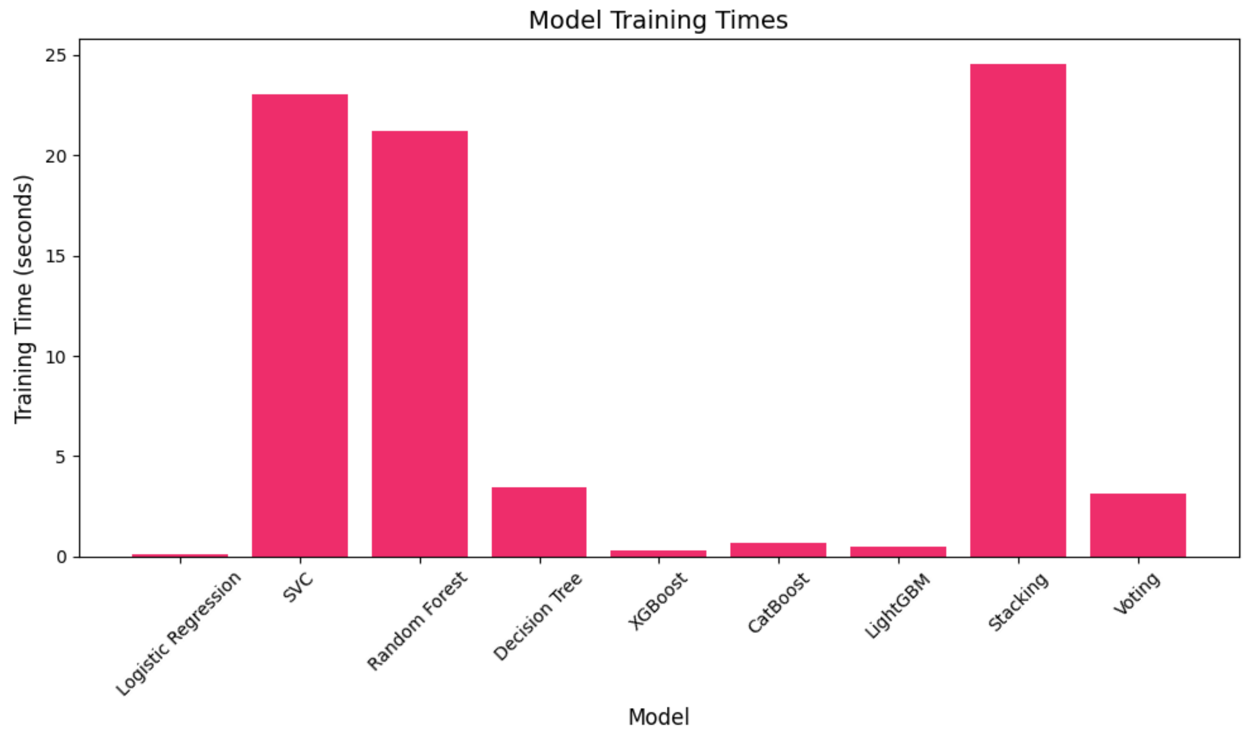


- ## Model Time Comparison

Train Time Comparison Table

Index	Model	Train Time (seconds)
0	Logistic Regression	0.133522
1	SVC	23.076761
2	Random Forest	21.240036
3	Decision Tree	3.429231
4	XGBoost	0.330573
5	CatBoost	0.671253
6	LightGBM	0.466891
7	Stacking	24.568236
8	Voting	3.156214

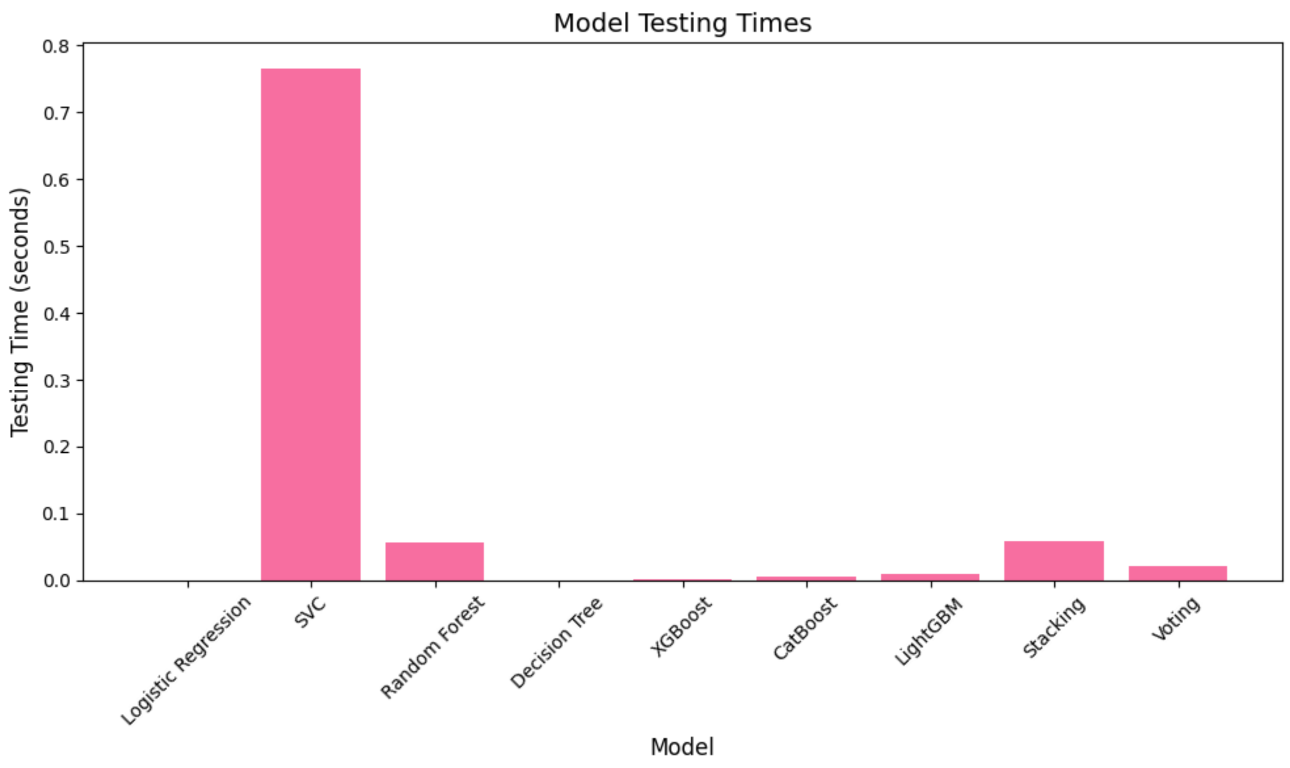
Train Time Comparison Graph



Test Time Comparison Table

Index	Model	Train Time (seconds)
0	Logistic regression	0.000000
1	SVC	0.765759
2	Random Forest	0.056248
3	Decision Tree	0.000000
4	XGBoost	0.002397
5	CatBoost	0.005621
6	LightGBM	0.008806
7	Stacking	0.058350
8	Voting	0.020729

Test Time Comparison Graph



- ## Deployment

Our Project has been deployed using **Streamlit**, a popular open-source framework for building and sharing data applications. The app is published on **Streamlit Cloud**, which provides free hosting for Streamlit applications.

Application Components

1. Navigation Menu

The app features a sidebar navigation menu with four main sections:

- **Home**: Project introduction and dataset information
- **Prediction With CSV File**: Upload CSV data for batch predictions
- **Prediction Form**: Interactive form for single prediction
- **Metrics**: Model performance evaluation

2. Home Page

- Project overview and objectives
- Dataset dictionaries for both regression and classification tasks
- Sample data previews
- Documentation link

3. Prediction Functionality

A. CSV File Prediction

- Supports both **Regression** (review scores rating) and **Classification** (guest satisfaction levels)
- Processes uploaded CSV files with comprehensive feature engineering:
    - Text preprocessing (TF-IDF + SVD dimensionality reduction)
    - Amenities categorization
    - Sentiment analysis
    - Feature scaling

B. Interactive Prediction Form

- Supports both **Regression** (review scores rating) and **Classification** (guest satisfaction levels)
- User-friendly form with:
  - Property details (type, room configuration, pricing)
  - Host information (rating, superhost status)
  - Amenities selection (categorized)
  - Text descriptions (transit, house rules, etc.)
- Real-time predictions with interpretation guidelines

4. Metrics Page

- Performance comparison of all trained models
- Interactive visualizations:
  - Bar charts comparing model scores
  - Gauge indicators for best model performance
- Metrics displayed:
  - **Regression**: R^2 scores
  - **Classification**: Accuracy scores

Access the App

You can access the live application through this link:

https://guest-satisfaction-project.streamlit.app/

Alternatively, scan this QR code with your mobile device to open the app:

Classification Conclusion

The Models were saved using **Joblib** and then we deployed using **streamlit,** The Stacking Model performed the best out of all the other models with an Accuracy score of **0.6470**. making it the most effective model for this classification task.

CatBoost, Voting Ensemble, LightGBM and XGBoost performing strongly with **accuracy** scores of **0.6332, 0.6338, 0.6309 and 0.6258** respectively followed closely, reaffirming that **gradient boosting** and **ensemble methods** consistently deliver top-tier performance.

Random Forest achieved a solid accuracy **0.6115**, outperforming base models but still trailing behind boosting and stacked models

SVC 0.5828 and **Logistic Regression 0.5696** offered modest performance, better than Decision Tree but not competitive overall.

Decision Tree had the lowest accuracy 0.5433, indicating limited generalization ability for this dataset.

Voting and boosting models **(XGBoost, CatBoost, LightGBM)** provided a strong tradeoff between accuracy and computational efficiency, making them viable for real-time or resource-limited scenarios.

Stacking, while most accurate, typically involves higher training and inference time, which suits batch processing or offline predictions.

SVC tends to have long training time and lower comparative accuracy, making it less suitable for this task.

Versions Used:

- Python Version: 3.11.0
- NumPy Version: 1.26.4
- Pandas Version: 2.2.0
- Scikit-Learn Version: 1.5.2