



GUEST SATISFACTION PREDICTION

Presented By:

Nada Mohamed

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

From the look of this data we can conclude that we have 69 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



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



the 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



the 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'

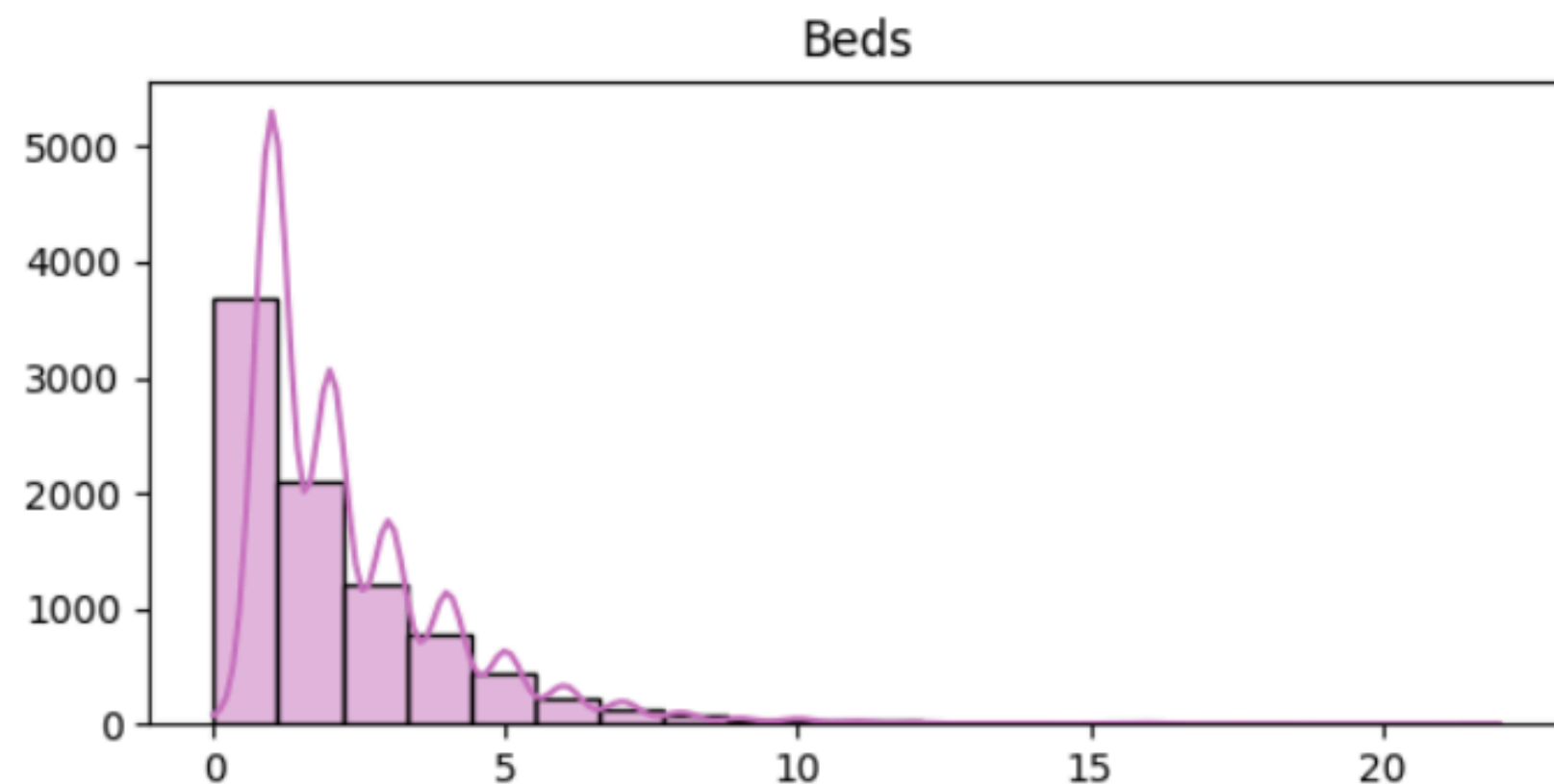
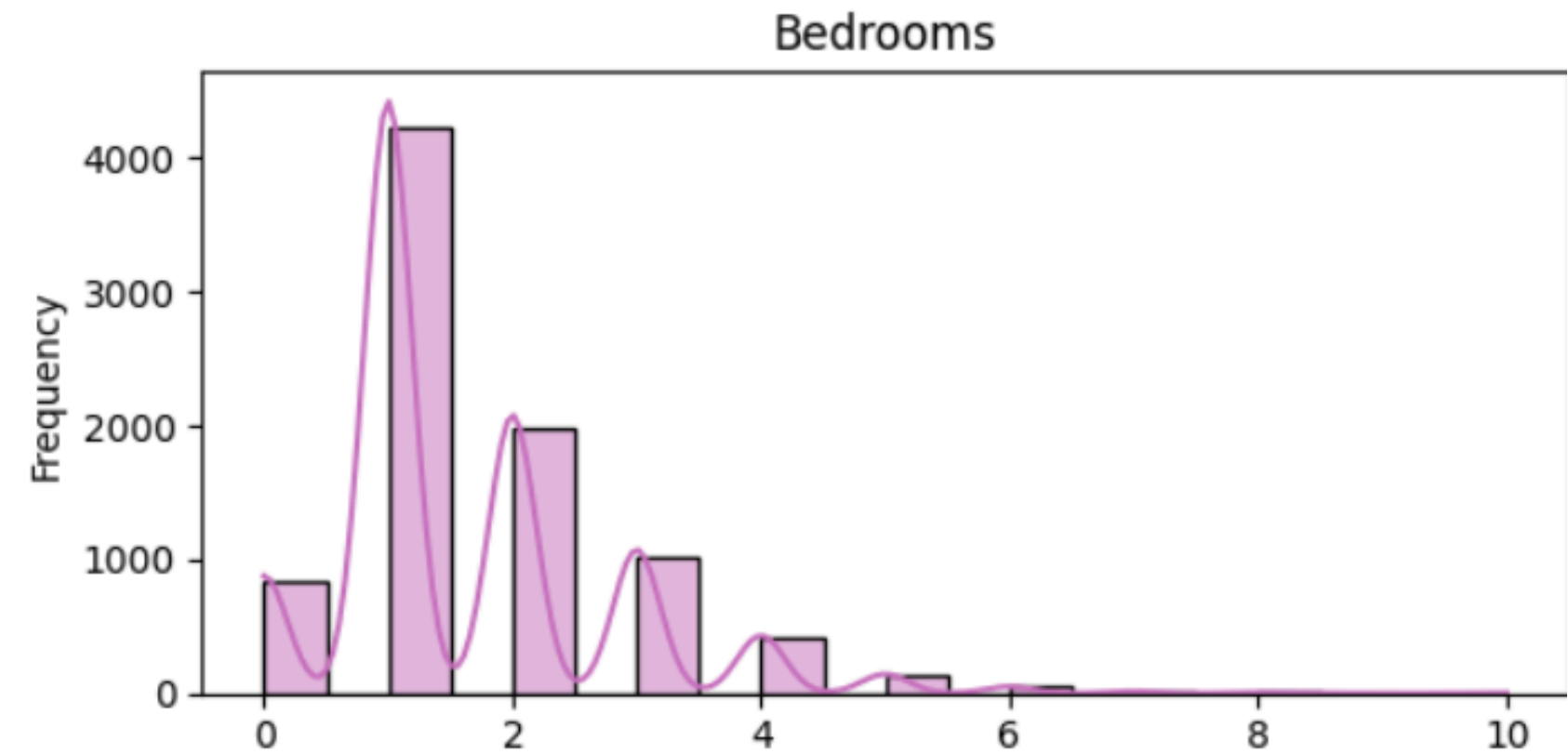
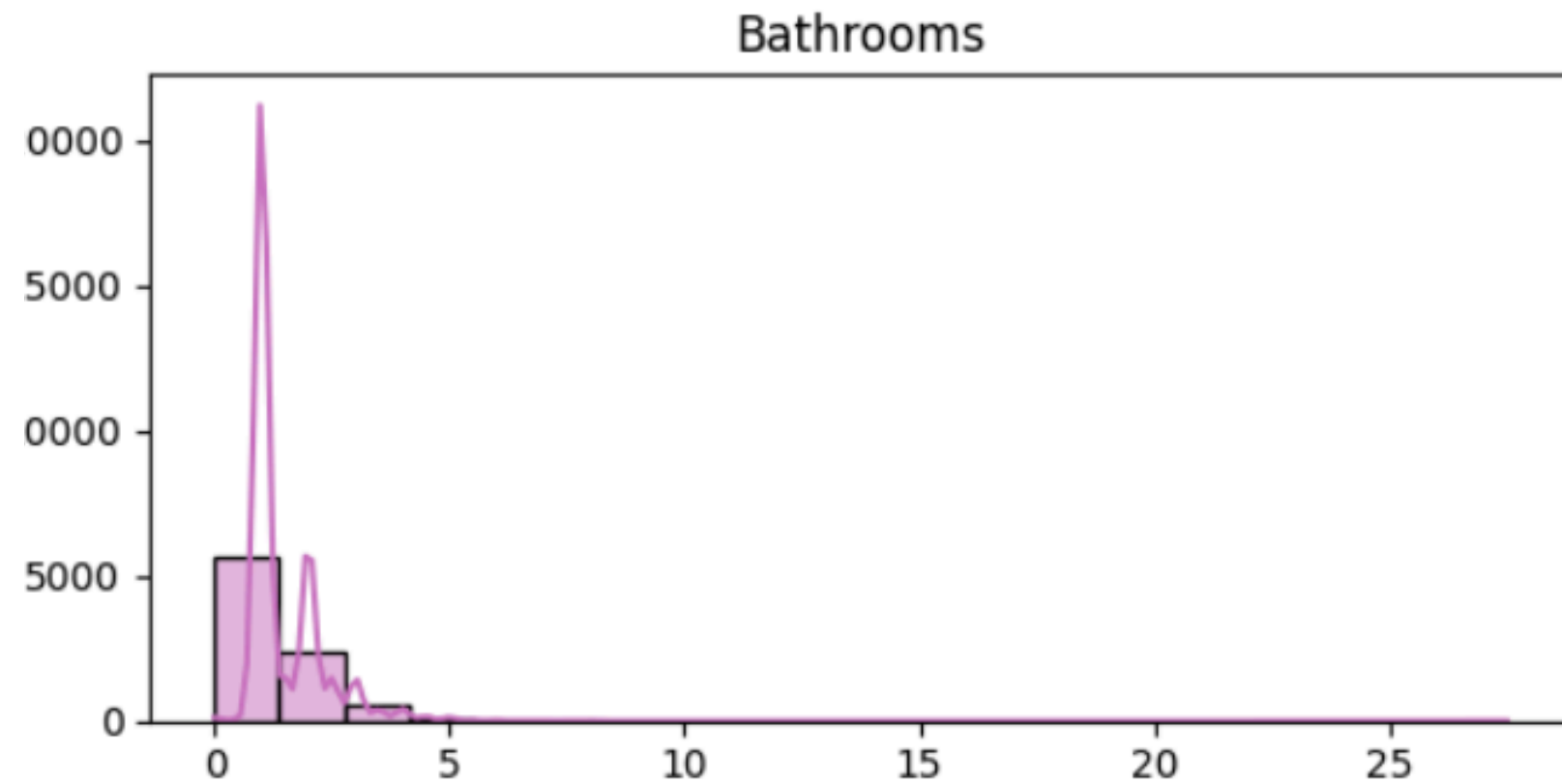


Handling Nulls using Median

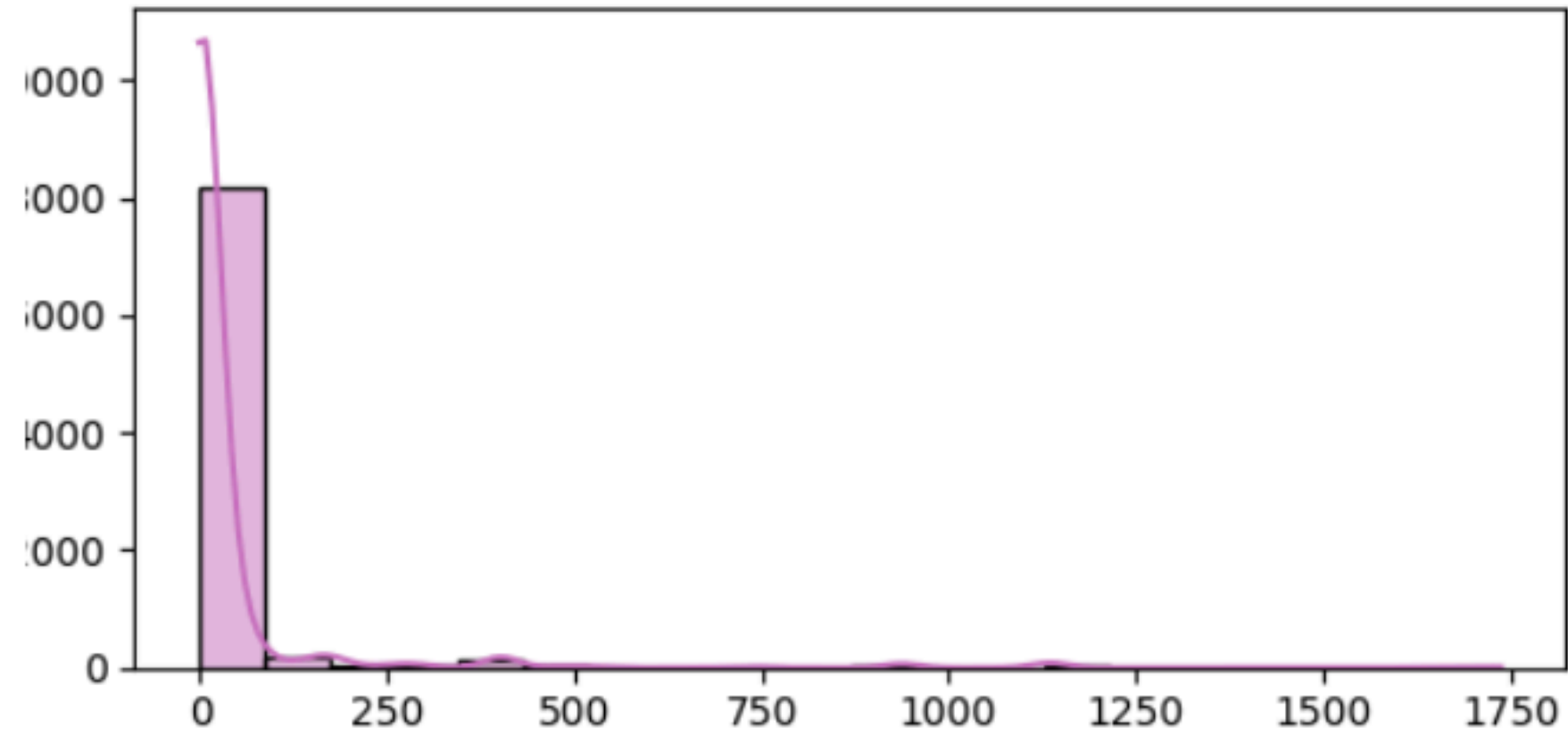
Some columns, on the other hand, are not normal. In other words, they show some degree of skewness when being visualized using histograms and other such visualization tools. The nulls in them were replaced with the 'median' value in those columns.

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

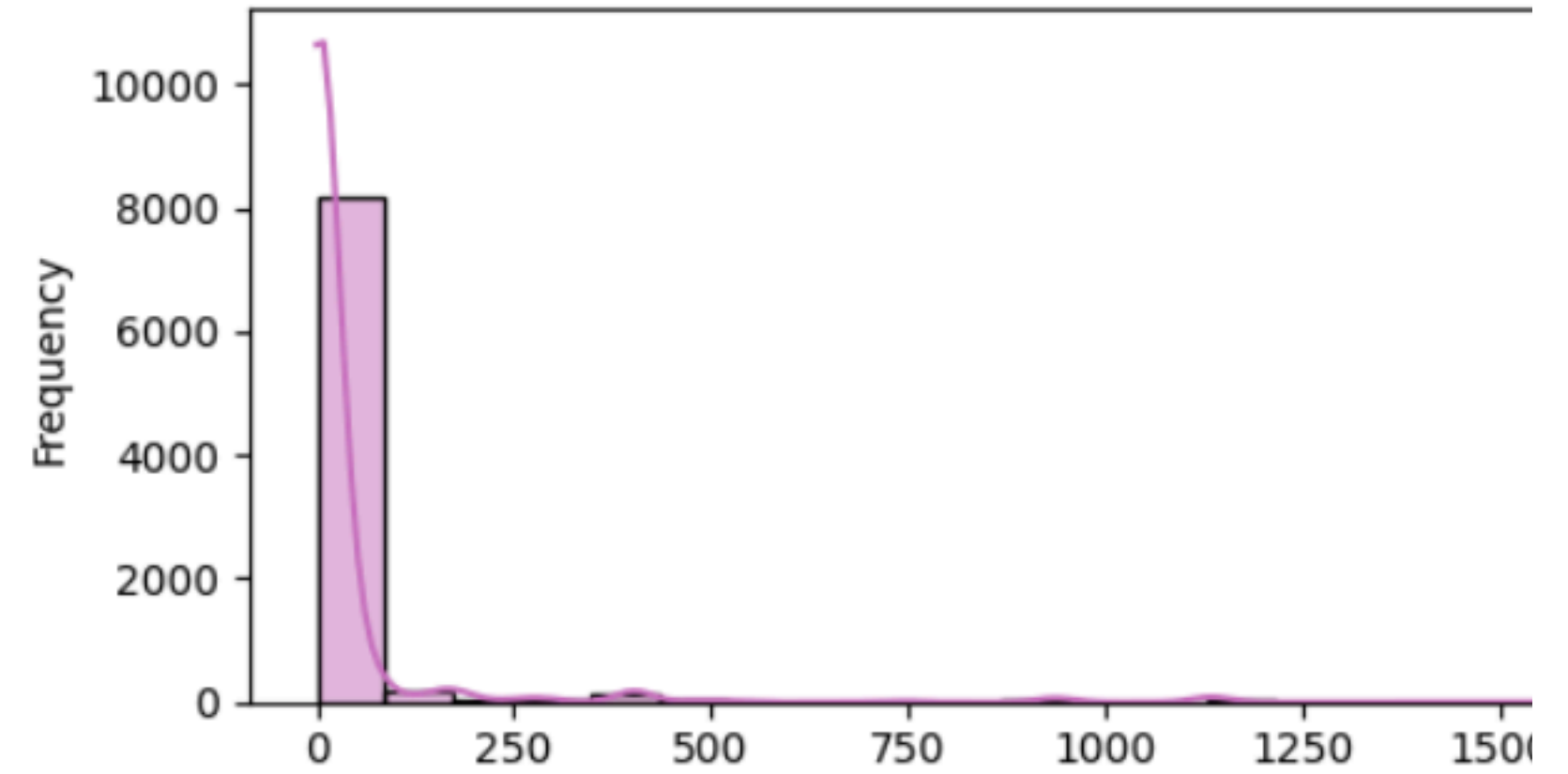
You can see the skewness of each of those columns clearly in the following histogram:



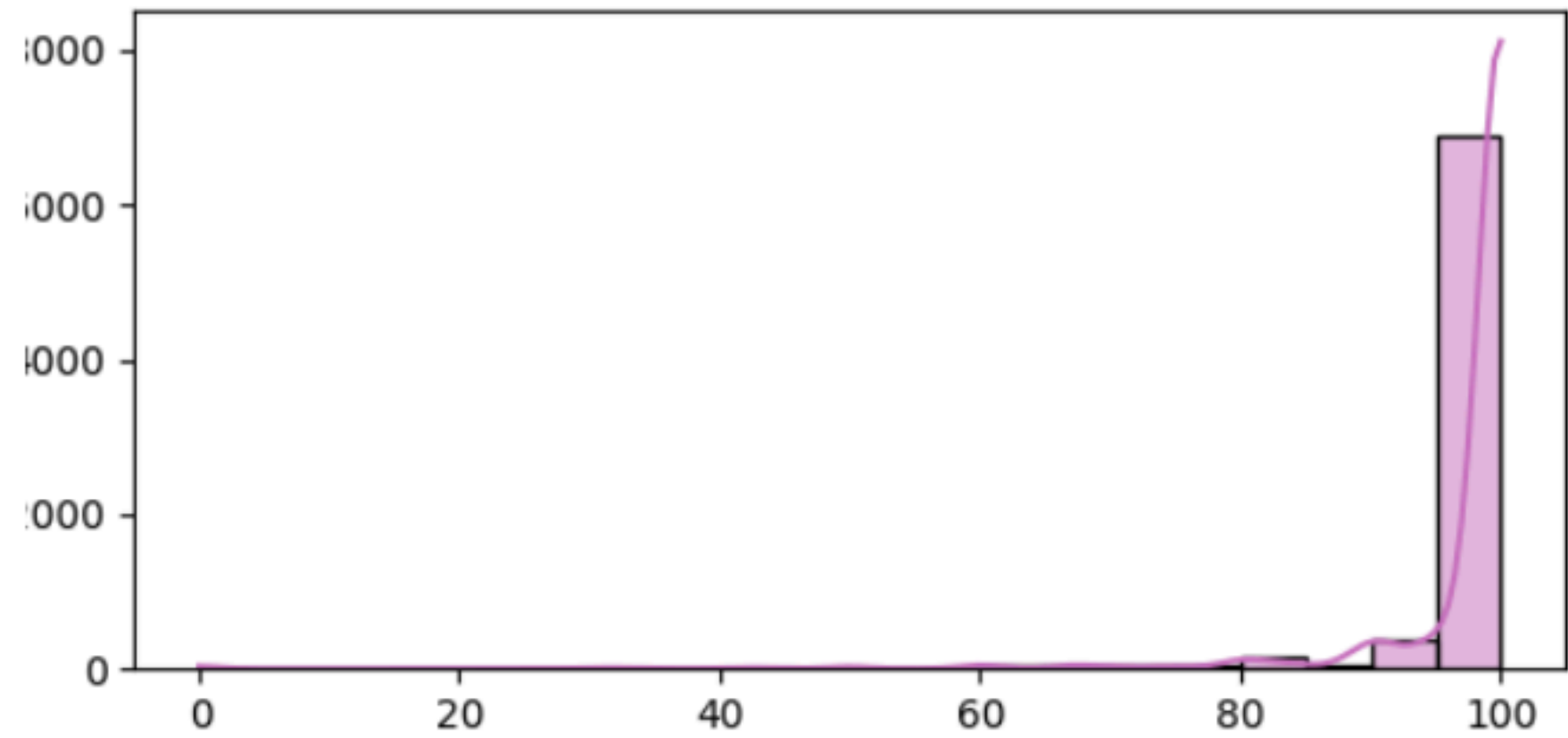
Host Listings Count



Host Total Listings Count



Host Response Rate





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 TextBlob



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 where `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.**

Splitting Columns into More Features



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.



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

Correlation Heatmap for Selected Features



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

- o 80% train

- o 20% test

- o 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

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



Linear Regression



Cross Validation = 5

o Cross-Validation R2 Scores: [0.21456857 0.21958834 0.21833478 0.23624945 0.2542602]

o Average Cross-Validation R2 Score: 0.2286

o **Training R2 Score: 0.2302**

o **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**



Lasso Regression

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

o Best parameters (Lasso): {'alpha': 0.001}

o Best CV score (R^2) (Lasso): 0.2213

o Training R^2 Score: 0.2302

o Test Set Metrics (Lasso):

❖ R^2 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





Ridge Regression

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

o Best parameters (Ridge): {'alpha': 46.41588833612773}

o Best CV score (R^2) (Ridge): 0.2214

o Training R^2 Score: 0.2301

o 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

Support Vector Regression (SVR)



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

- o Best parameters (SVR): {'C': 3.1622776601683795, 'epsilon': 1.0, 'kernel': 'rbf'}**

- o Best CV score (R^2) (SVR): 0.1861**

- o Training R^2 Score: 0.2324**

- o 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**



Random Forest

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

- o Best parameters (RandomForest): {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 100}

- o Best CV score (R^2) (RandomForest): 0.2258

- o Training R^2 Score: 0.5070

- o Test Set Metrics (RandomForest):

- ❖ R^2 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 R2 Score)



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

o Best parameters (DecisionTree): {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5}

o Best CV score (R^2) (DecisionTree): 0.0680

o Training R^2 Score: 0.4434

o Test Set Metrics (DecisionTree):

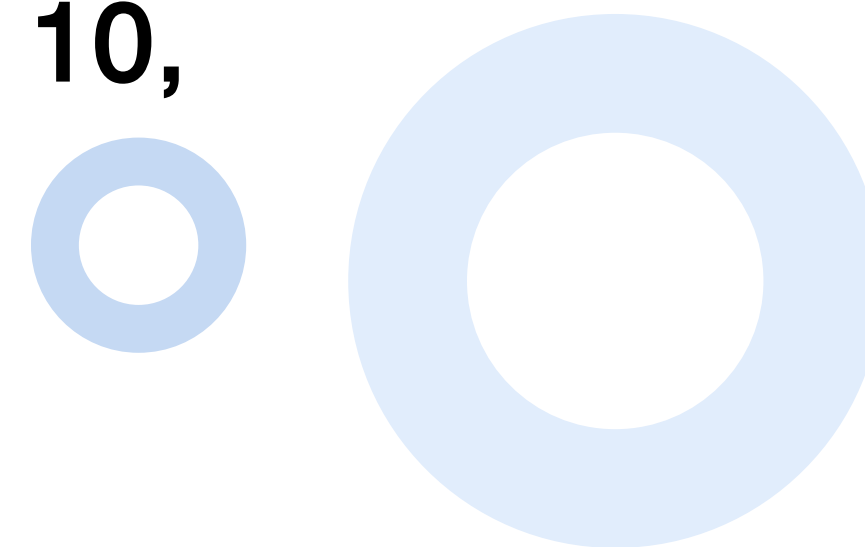
❖ R^2 Score: 0.1178

❖ Mean Absolute Error (MAE): 3.0066

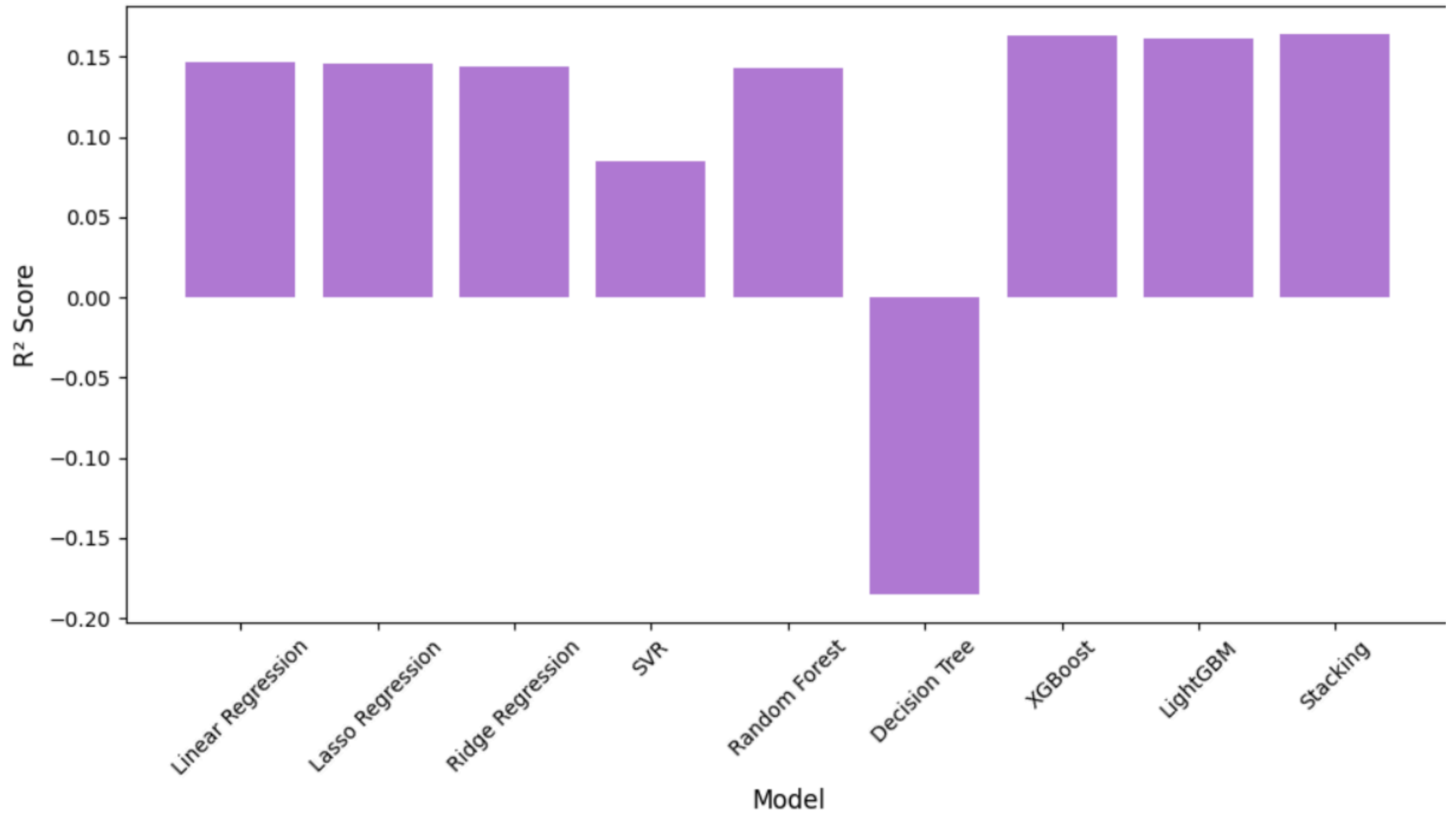
❖ Mean Squared Error (MSE): 17.1391

❖ Root Mean Squared Error (RMSE): 4.1399

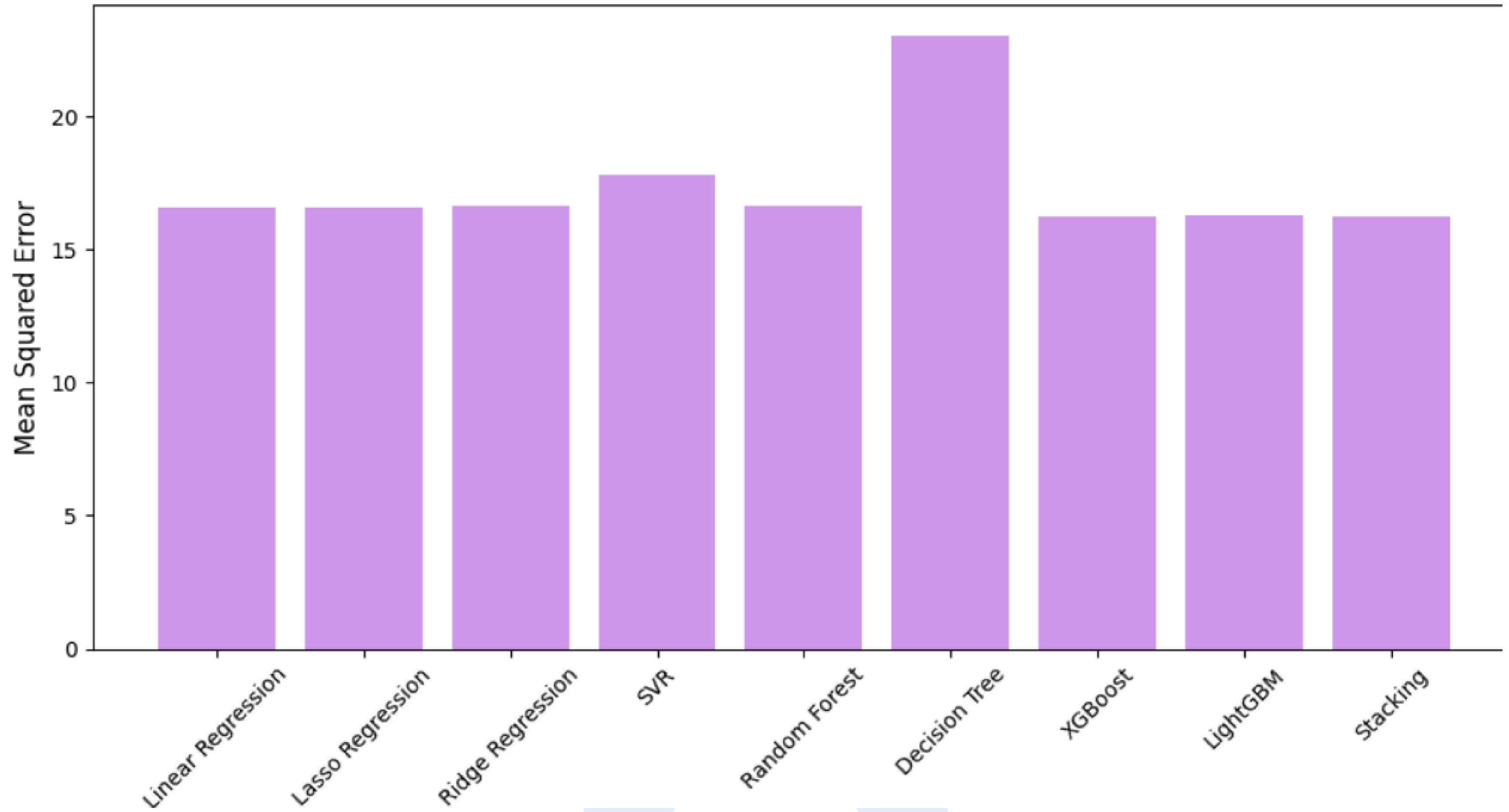
❖ Mean Absolute Percentage Error (MAPE): 0.0321



Model R² Scores



Model Mean Squared Error (MSE)




Conclusion

The Stacking Model performed the best out of all the other models with an R2 score of **0.3044**. This means that it explained the most variance in the target variable among all the models.

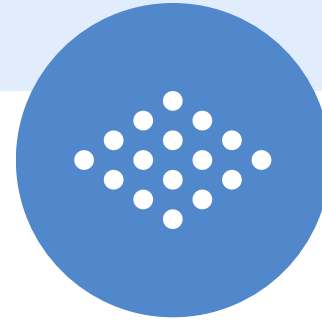
Following close behind were the Booster Models, with CatBoost and LightGBM performing strongly with R2 scores of **0.2999** and **0.2914** respectively. XGBoost performed along the same vein though its R2 score was **0.2835**.

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 R² scores are held by both the SVR Model and the Decision Tree Model. The SVR model performed worse than the Linear Models with an R² score of **0.2553** showing that using SVM-based approaches for this dataset is not the best idea, and though its R² 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.



**THANK
YOU!**



