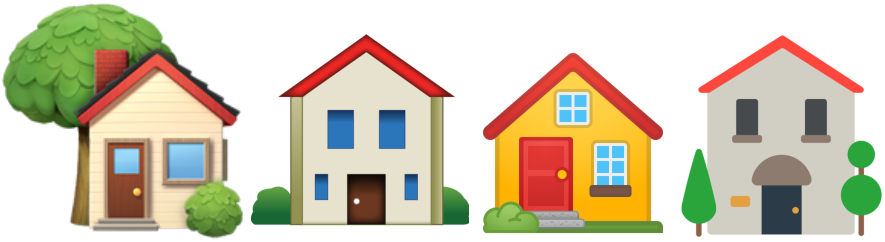
Prescient Price

Using Machine Learning to Predict the Price of Airbnb Listings

Claire Miles

April 2019



Introduction

Today's world is more connected than ever. With increasingly long nonstop flights, spontaneous translation apps, and social media friendships spawning worldwide, it's easier than ever to be a global citizen. One company catalyzing connections is Airbnb, which provides a service that allows property owners to rent their houses, apartments, or rooms in their hometowns, offering tourists a unique way to lodge in a new city - one that is arguably more authentic than lounging at a luxury resort.

By creating these home-away-from-homes, Airbnb has skyrocketed in popularity for those looking to live like locals and be more culturally connected to their destination. It's in Airbnb's best interest for its listings to be the best they can be, not only to crete an optimal user experience, but also to maximize the prices of their listings. Therefore, it's very useful to know what goes into a successful listing, and to be able to predict the price of a listing from its attributes. This will help Airbnb increase its revenue and help hosts optimize the value of their listings by predicting their success.

The inquiry: **How can we predict an Airbnb listing price based on other information about that listing?**

I used data from Inside Airbnb, an independent, non-commercial project that collects public data from the travel and accomodations company Airbnb. Data is collected monthly from major cities and metropolitan regions around the world, and includes information about that city's listings, reviews, and calendar data. I focused on data about Airbnb listings and the reviews for those listings. The listings data contains almost 100 columns of attributes for each listing, including number of bedrooms and bathroom, square footage, amenities provided, host information, etc. The reviews data includes the text of the reviews left for each listing. The information between the two datasets can be linked using provided ID numbers.

For this project, I used numpy, pandas, matplotlib/seaborn, scikit-learn, and vaderSentiment to craft a machine learning-ready dataset from several messy, human-collected csv files, crafted features from non-numeric data, and explored and evaluated several tree-based ML regression models including random forests, gradient boosting, and XGBoost. This report serves as a narrative version of the projects in all stages: 1) data collection, 2) data cleaning, 3) exploratory analysis, and 4) feature engineering and machine learning modeling. The accompanying code can be found on the author's Github.

Data Collection

Since I used data sourced from many files found on the same webpage, I decided to write a script that scrapes that page for the relevant URLs to my project. To complete this task, I used the requests and BeautifulSoup packages. First, I used requests to capture the response from the url where all of the csv files are hosted: http://insideairbnb.com/get-the-data.html. After catching that response in a variable, I read the text into another variable. I turned the text variable into a BeautifulSoup object using the BeautifulSoup function. After creating the BeautifulSoup object, I extracted the text from the HTML anchor tags, where the csv links are found in the "href" attribute of the tag. I stored the names of the csv urls in a list. Since the links led to gzipped (.csv.gz) files, I called the list `zipped_links`.

Next, I wrote the information from the online csv files to local files. Using a context manager, I looped through `zipped_links` to create custom filenames for each list item and write to that file. The filename includes the city name, the date the data was collected, and the information category (listings or reviews). Some links to files on the website were broken, so they had to be removed from the dataset.

After the raw files were scraped, created, and stored, the data was consolidated from monthly files into one single file for each listings and reviews. Because there are hundreds of thousands of lines of data in each file, it helped to create functions that did the heavy lifting:

- `consolidate_data` checks if the consolidated csv file for either listings or reviews data has been created for the designated city. If the file has not been created, it runs the `combine_files` function for that city, and then creates the csv file for that city.
- `combine_files` goes through files in the directory and checks for the designated city files of the specified kind. Then it appends the names of the files of that city to a list, and passes the list and the directory name to the `concat_files` function.
- `concat_files` creates a pandas dataframe for each file name in the list of files, then appends the dataframe to a list of dataframes. After all files in the list have been converted to pandas dataframes, it concatenates the dataframes together, drops duplicate rows, and resets the dataframe index.
- `export_csv` checks if the desired csv file does not exist in the current working directory, then converts the data frame to a csv file and moves the the desired folder in the destination directory.

Even though I started by analyzing just Los Angeles, having functions will also allow me to scale my analysis to multiple cities in the future.

After I created the functions, I defined the directory (the pathway where the raw data is stored) and destination folder (where I wanted to store the concatenated data on the computer) and ran the functions. The outcome was a single listings file and single reviews file for Los Angeles.

Data Cleaning

With the data collected and combined into two files `los-angeles_listings.csv` and `los-angeles_reviews.csv`, it still had to be cleaned. Major cleaning considerations included the size of the data, data types, redundancy in features, data entry errors (e.g capitalization and spelling), null values, and combining the listings and reviews data into a single dataframe.

First, I started out by getting a sense of the size of both dataframes. The listings dataframe `listing` had 750 thousand rows and 106 features while the reviews dataframe `review` had over 2 million rows but only 6 features. All the data is either of integer, float, or object data type. `listing` was much larger than `review` despite having many fewer rows.

Dropping Columns: Looking through the first row of `listing`, it looked like many rows repeated information or were not relevant to this project's analysis. For example, all rows related to location (neighbourhood, city, state) could be removed in favor of just the zip code. Also for this project, I intentionally did not use any non-categorical text data apart from that in `review`. Therefore, I put the names of all columns I did not want to keep into a list called `drop` and used the pandas drop function to remove those columns in place.

Inconsistencies in Categorical Data: It was important to correct inconsistencies in categorical data to prevent more categories being created than necessary. By converting all text in `listing` to lowercase, I immediately got rid of all capitalization-related data entry inconsistencies and could focus on spelling and formatting errors. I used the `value_counts()` pandas function to find such inconsistencies with the `bed_type` column and found that some entries are actually people's names! These entry errors are a vast minority in the data, and I decided to remove those rows from `listing`. After running `value_counts()` on other categorical variables, I found that removing the rows from the `bed_type` analysis got rid of most other inconsistencies in other columns as well. One minor flaw that I found in the `accommodates` column included a string with a '%', which prevented the column from being converted to a numerical data type. I removed that row from the `listing`. Finally, I fixed minor entry inconsistencies in the `property_type` column in which property types of the same name had slightly different entry formats.

The Amenities Feature: One interesting feature in `listing` is the `amenities` row, which is a string containing a list of the amenities offered in a listing. Although the format wasn't machine leaning-compatible, I saw this as a great opportunity to do some feature creation. From the row of `listing` above, you can see that `amenities` consists of strings wrapped in curly braces containing lists of amenities, and some amenity names are contained in quotes. To create a string without these extraneous characters, I removed the characters. After removing the extraneous characters, I created a list of all unique amenities represented in the dataset. I iterated through this list of unique amenities to create new feature-specific boolean columns in `listing` using my function `has_feature()`. The output of `has_feature` were individual boolean columns for each unique feature in the dataset that indicated whether a listing provided that feature. After creating these columns I was able to drop the original `amenities` column from `listing`.

Changing Data Types: Since machine learning algorithms in scikit-learn only take numeric data, I needed to make sure that all columns in `listing` were either integers, floats, booleans, or categorical. Going manually through the columns, I separated them into lists based on which data type they should be, and used these lists in the `change_datatypes()` function. Additionally, there were special columns that required specific processing because they included a special symbol: prices ($) and percentages (\%). The columns were put into their own lists as well for `change_datatypes()`. `change_datatypes()` used these lists to run data type conversion functions on `listing`.

Target Variable Null Values: For the final analysis, it wouldn’t be useful to have rows of data that do not have values for the target variable `price_USD`. Therefore, I dropped all columns where `price_USD` was null.

One Hot Encoding Categorical Variables: Before one hot encoding categorical variables, I saved and exported a small file containing columns of interest for plotting later in the analysis. Then, I created a list with the names of the columns that I wished to be one hot encoded, and used the pandas `get_dummies()` function to add those encoded columns to the list.

Incorporating Review Data: The `review` dataframe contains a row for each review written for an airbnb listing. I aggregated all the review text for each individual listing and added it to `listing` as an additional

column. I dropped all columns from the data that were not `listing_id` and `comments`, the two rows that would help me append the review text to the final dataframe. Using the reduced `review` dataframe, I created an `aggregate_reviews()` function that grouped together all review text for each listing. Then I added that grouped data as another feature to the dataframe and renamed the dataframe `df`.

Missing Values: Because machine learning algorithms can't handle missing values, those missing values needed to be filled in way that makes sense. First I created a list of column names that had null values and weren't booleans and another list of column names that had null values are were booleans. I filled the nan values differently for different columns:

- For `zipcode` I filled missing values with the mode.
- For other non-boolean columns, I filled missing values with the median.
- For boolean columns, I filled missing values with `False`

Cleaning Result: I ended up with a single cleaned dataframe with 1.1 million entries and 353 features of numeric type (except for the review text data, which will be turned into a numeric feature in the machine learning analysis). I saved the dataframe as the csv `df_clean.csv` for later use.

Exploratory Data Analysis

After cleaning the data, I performed some exploratory data analysis and plotting to get a better sense for trends in the data.

Price: It was important to look into the trends and behavior of the target variable `price_USD`. A histogram of price is right-skewed, meaning that the mean is larger than the median (Figure 1). This makes sense, since most Airbnb listings will target the majority of consumers that are looking for affordable options, and most listings are smaller apartments or individual rooms. On the other hand, larger and more luxurious Airbnbs exist for nice vacations and getaways. The violin plot and boxplot are additional representations of `price_USD` (Figure 2). There are many outliers on the more expensive end, with more listings being priced between $0 and $200.

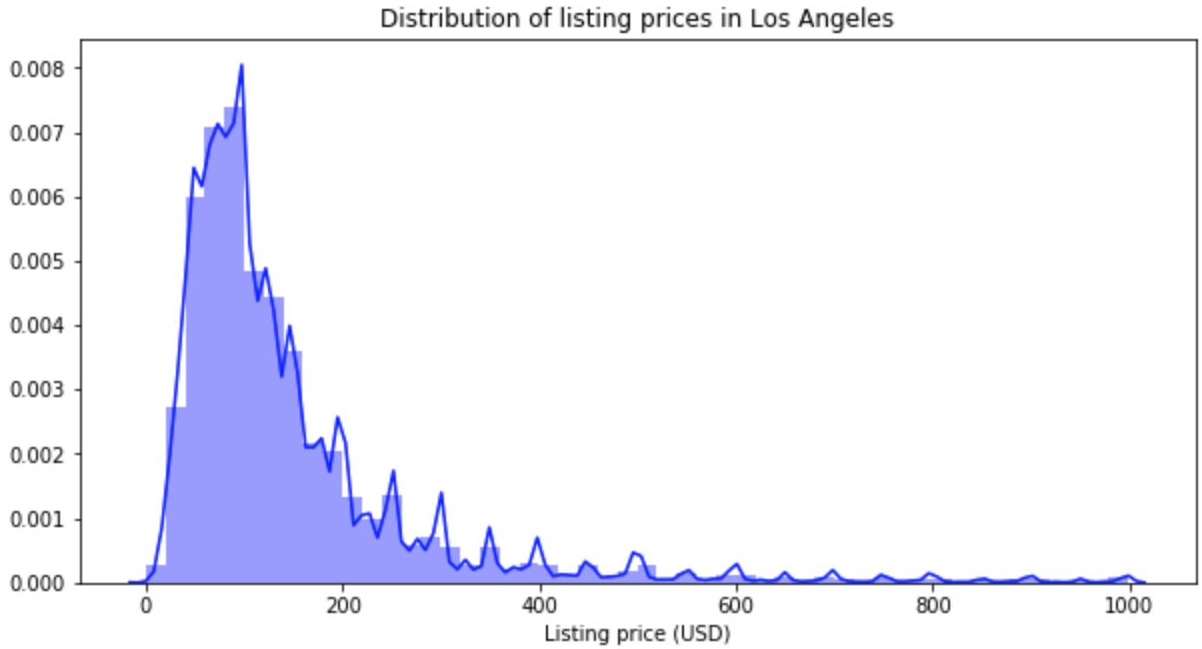


Figure 1. Histogram of price.

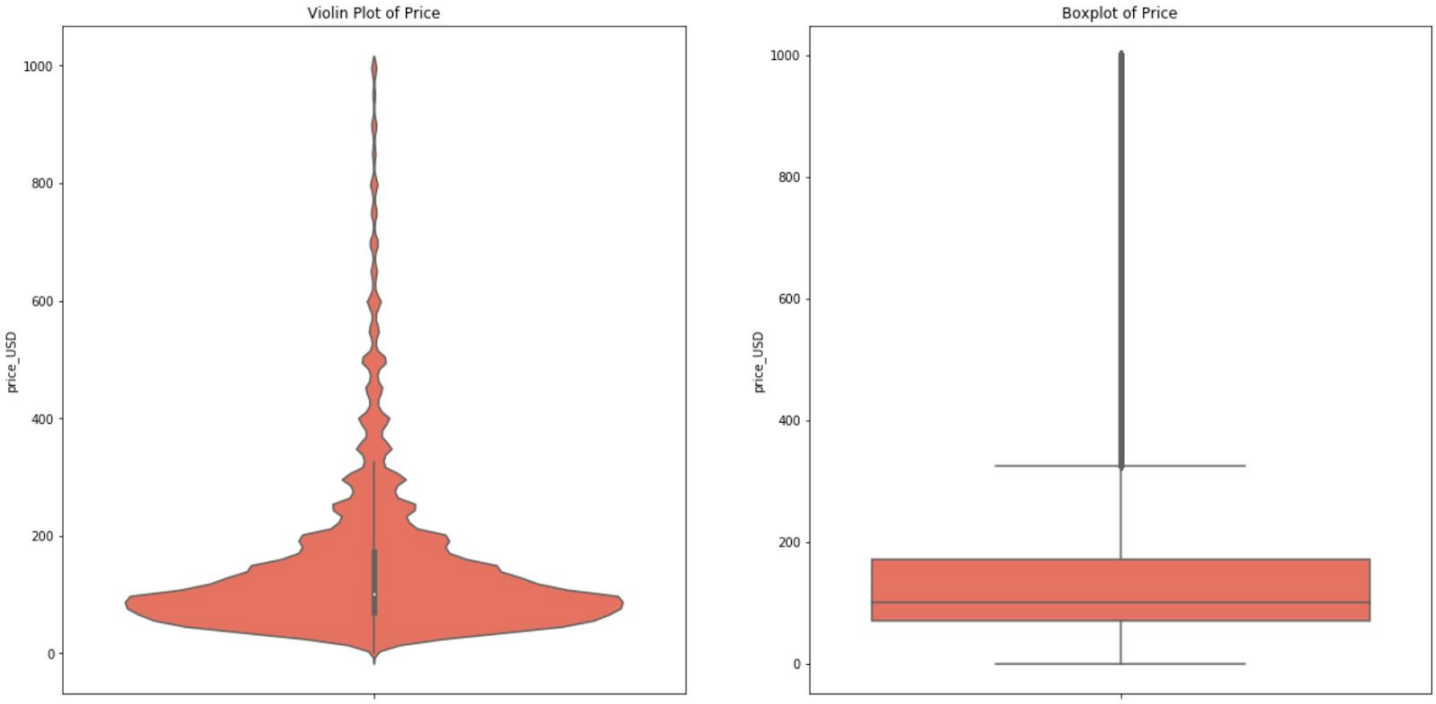


Figure 2. Violin plot (left) and boxplot (right) of price.

I also looked at price geographically by graphing the mean listing price by zip code. I created a GeoDataFrame using a shapefile of Los Angeles zip codes and the Airbnb information. A random selection of the 1.1 million points are plotted to make it possible for human eyes to discriminate the distribution of features on the map.

The map of mean price shows that the most expensive listings include (Figure 3):

- 90291, Venice
- 90046, Hollywood Hills
- 90026, Silver Lake

These neighborhoods are all wealthy, well located areas near the Los Angeles city center or on the beach. Most of the more expensive listings in general follow this pattern, with those closer to urban areas being more expensive.

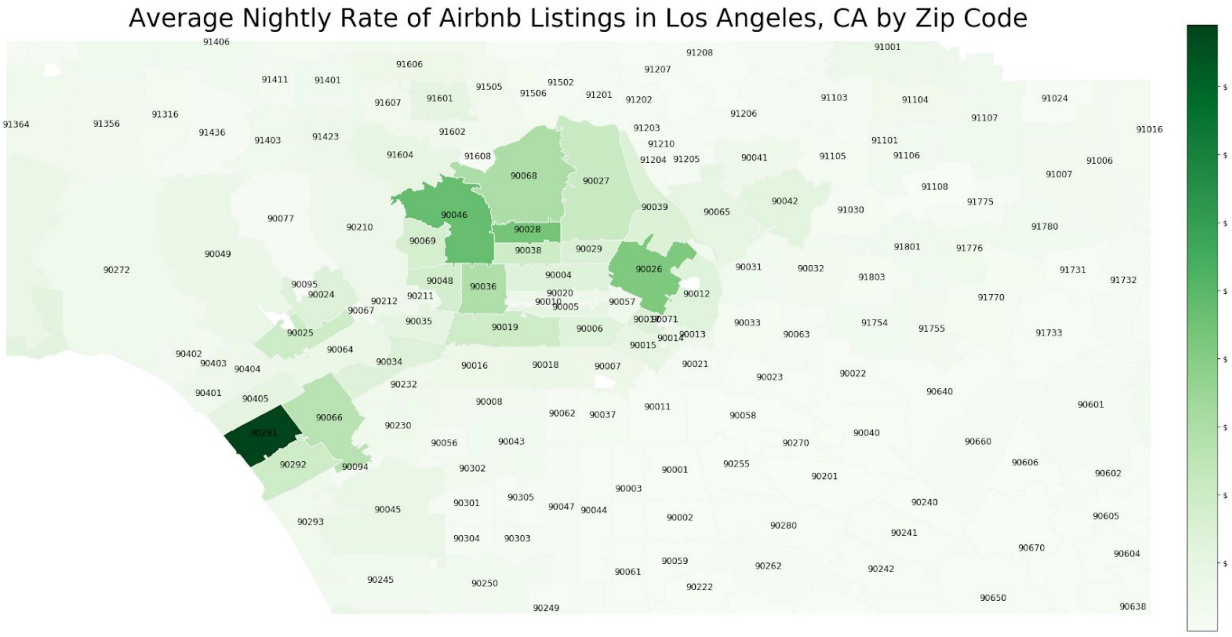


Figure 3. Map of average price per zip code in Los Angeles.

Accommodates: Another interesting variable is `accommodates`, which is the number of people that the Airbnb can hold. It looks like most Airbnbs are for 1 person, with most others being for 2-4 people (Figure 4). This makes sense because most listings are on the inexpensive side, and small listings for 1 person would be some of the least expensive listings on the platform.

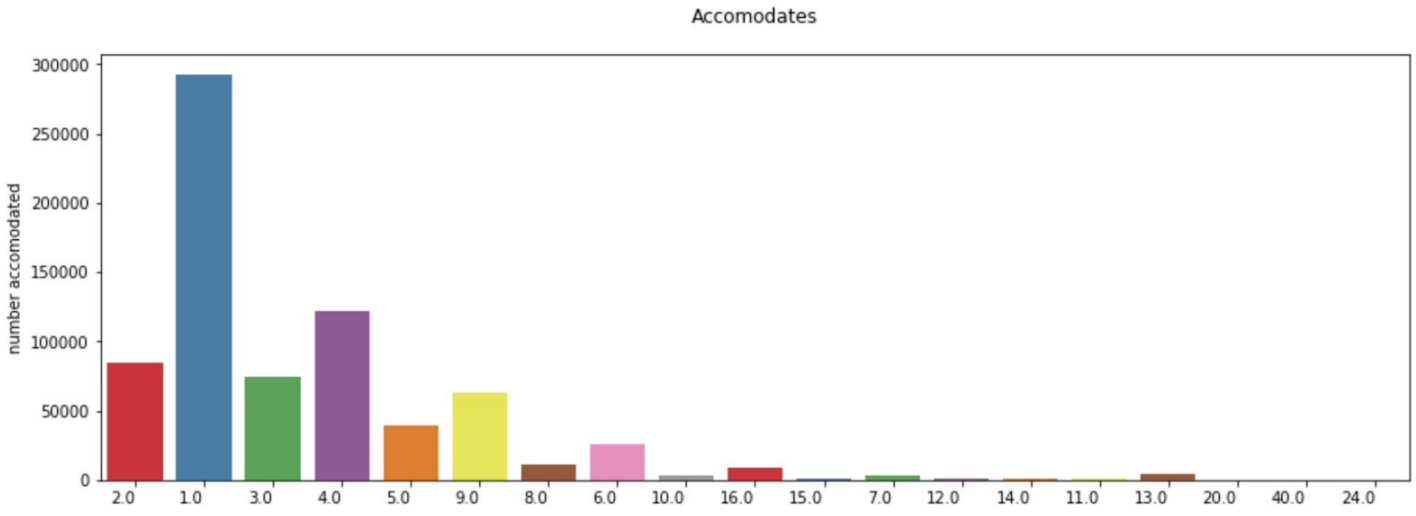


Figure 4. Bar plot of the number of people accommodated in a given listing.

Square Footage: Square footage is another variable that can be associated with the price of a listing. There is a similar trend in `square_feet` as for `price_USD` where the mean is greater than the median (Figure 5). Since

most listings are inexpensive, only accommodate one person, and are in urban areas, it makes sense that most are also on the smaller side.

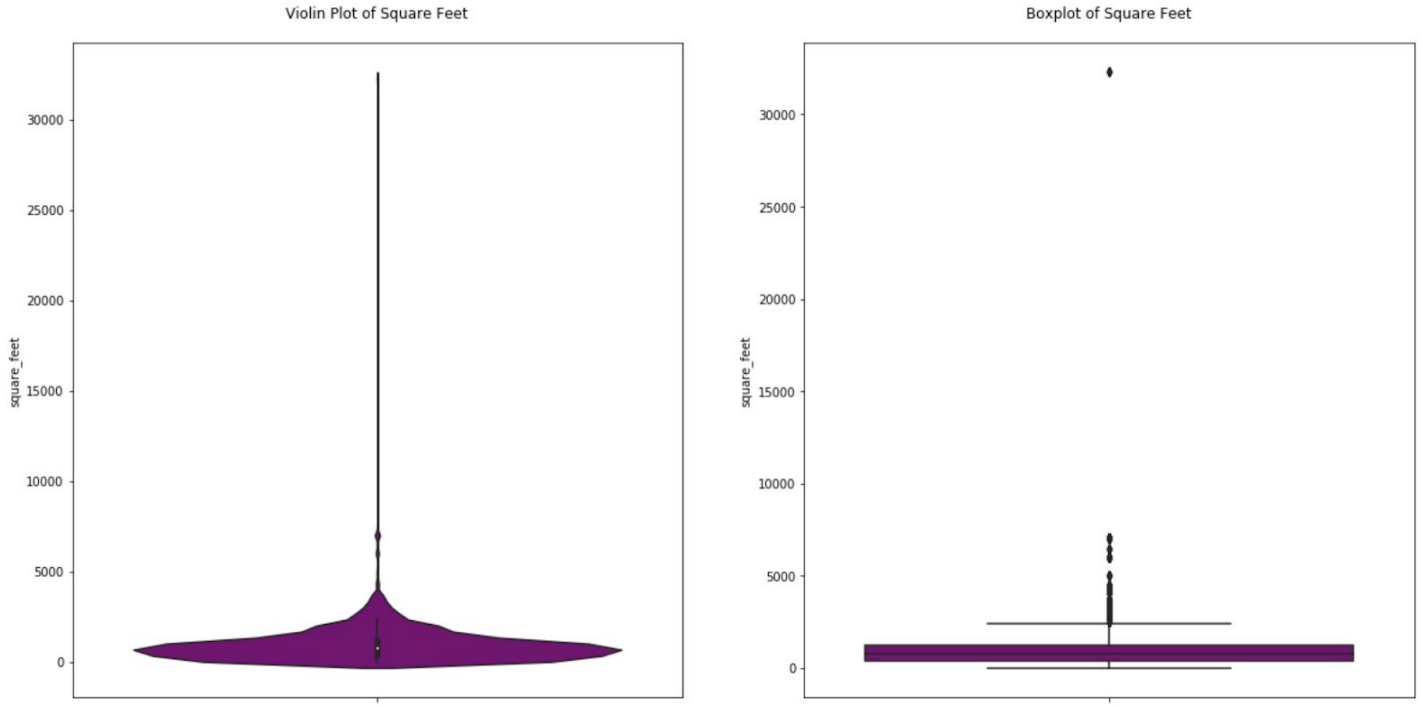


Figure 5. Violin plot (left) and boxplot (right) of square feet.

Property Type: Looking at the many different categories in the `property_type` feature, it's clear that the vast majority of listings are of type "house" or "apartment" (Figure 6). On the map, the distribution of these two types are plotted along with all other categories grouped together as "other". The map shows how apartments and houses greatly outnumber the listings in the "other" category, but all categories appear to be distributed rather equally across the city (Figure 7).

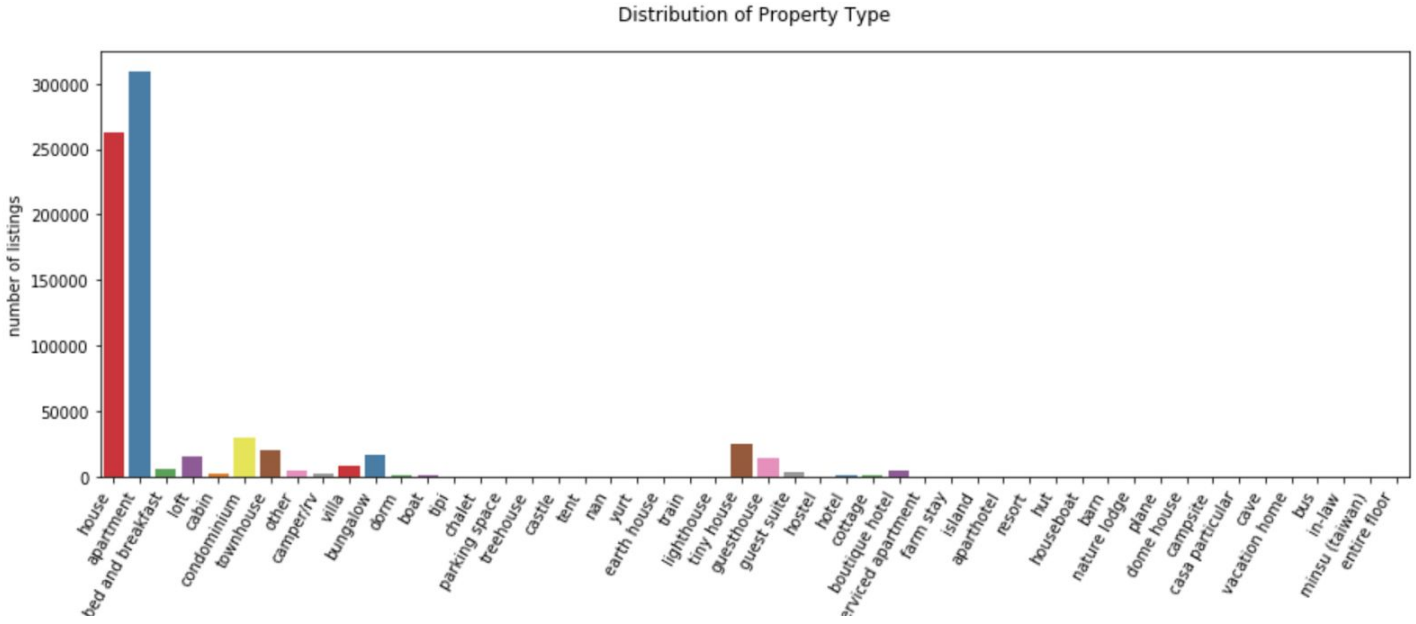


Figure 6. Bar plot of property type.

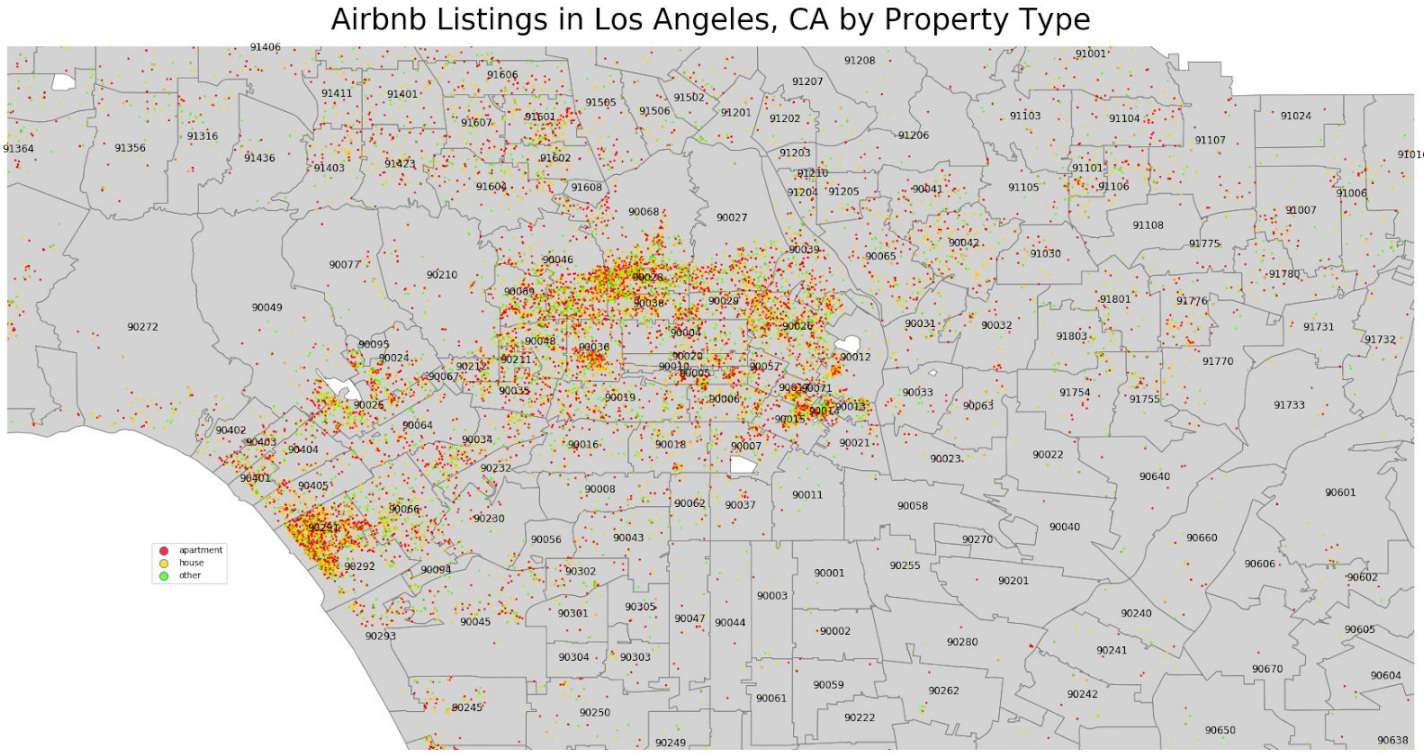


Figure 7. Map of listings by property type in Los Angeles.

Feature Importances: I briefly prepared a subset of the data for the random forests training by selecting only the numeric data types in the dataframe. To get rid of any null values (none should have been left after the data cleaning process, but I did this for good measure), I used the pandas interpolate function to fill in the values linearly. Then, I trained a RandomForestRegressor classifier on the data and plotted the features by their importances.

The random forest found the most important features to be number of people accommodated, location (longitude, latitude, zip code), the cost of extra people, room type, and the number of reviews (Figure 8). It's important to note, however, that this was a very simplistic first-pass at the data, and that the feature importance algorithm for random forests shows some bias towards continuous features or high-cardinality categorical variables. I re-used this method in my further analysis to pare down the data for use in the final model.

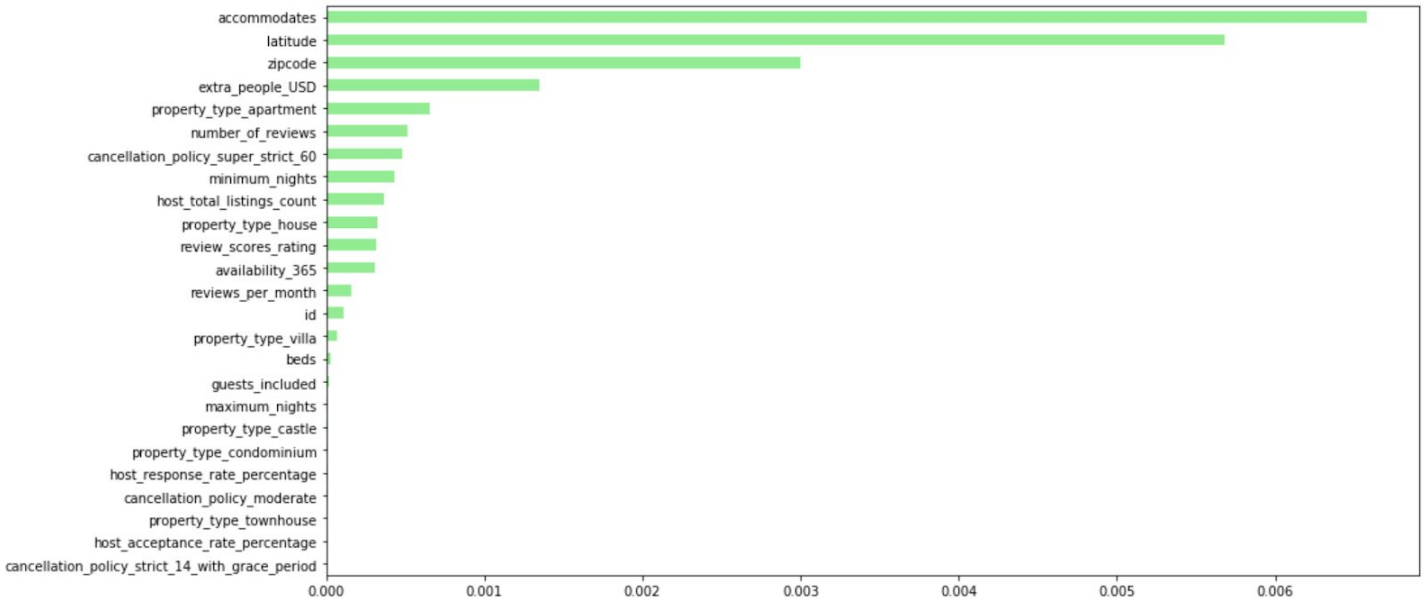


Figure 8. Bar plot of feature importances, with most important features towards the top of the plot.

Machine Learning Modeling

Feature Engineering: Most of the data being used for modeling is numeric, but the only string feature, the review text, can be turned into an additional numeric feature using natural language processing techniques. I chose to use VADER sentiment analysis to engineer this feature. According to it's Github page, VADER (Valence Aware Dictionary and sEntiment Reasoner) "is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media, and works well on texts from other domains". After running a string through the Sentiment Intensity Analyzer from the vaderSentiment package, the analyzer returns numeric scores for the positivity, negativity, and neutrality of a sentence.

Before using VADER, I preprocessed the reviews text to remove some extraneous characters such as "\\", "\[", and "\]". For VADER, stopwords and punctuation should be kept in since the classifier incorporates those elements into the analysis.

To use the Sentiment Intensity Analyzer, I wrote my own function called `get_review_sentiment()` that wraps the analyzer in a try-except loop and only returns the positive aspect of the score. Once all sentiment scores were retrieved from review data, I added them to the dataframe as a new column called `pos_score`. Therefore, the review text gets turned into a numeric feature about the positivity of the reviews that a certain listing receives.

For the initial exploration of the creation of `pos_score` and ease of use, I only used the first 1000 rows of the dataset. Figures 9 and 10 show examples of the most positive and least positive reviews. While the most positive reviews are very complimentary and use exclamation points and emojis, the least positive reviews are either in a non-English language (non-English languages are not yet supported by vaderSentiment) or don’t reflect that same positive emotion.

```
Most positive reviews:
----------------------

Great

Perfect 👌🏻

Marcy was great very sweet!

Amazing spectacular experience , highly recommend

Awesome host. Great communication and accommodation skills. Made sure I was taking care of.
```

Figure 9. The most positive reviews from the dataset used for the initial feature engineering exploration.

```
Least positive reviews:
-----------------------

普段通りの生活ができ、とてもすごしやすかった

房间和照片中的一样，房东想的很周到，并且带我们去了附近的亚洲超市，非常感谢！下次来洛杉矶还会选择住在这里。

The host canceled this reservation 4 days before arrival. This is an automated posting.

Hillary很热情,给予我们很大的帮助。地理位置非常好,靠近好莱坞影视城,房间非常干净、舒适、温馨,房间内的生活用品非常齐全。这是我们来美国住过民宿中的最好的一次。, Hillary很热情，给予我们很大的帮助。地理位置非常好，靠近好莱坞影视城，房间非常干净、舒适、温馨，房间内的生活用品非常齐全。这是我们来美国住过民宿中的最好的一次。, Hillary非常好，房子非常干净，周围很安全，去环球影视城很近。谢谢你

Ya es la segunda vez que elijo esta casa en West Hollywood, la elijo porque es muy linda, tiene todas las comodidades, todo funciona muy bien, tiene muchos cuartos y está en un barrio muy bonito y estratégico para moverse con comodidad en Los Angeles.Nina y Kevin son muy serviciales, están disponibles en todo momento y a cualquier hora para solucionar lo que se necesite y dar consejos para que la estadía sea perfecta.Son tan adorables y confiables que lo hacen sentir a uno como si fueran grandes amigos en esa ciudad. Gracias por todo!, The host canceled this reservation the day before arrival. This is an automated posting., The host canceled this reservation 42 days before arrival. This is an automated posting.
```

Figure 10. The least positive reviews from the dataset used for the initial feature engineering exploration.

Feature Reduction: The negative aspect to using the vaderSentiment package is the very slow speed. For example, 1000 loops of `get_review_sentiment()` takes over 20 minutes, so all 1.1 million data points would take over 10 days to process. Using a random forest regressor and the small dataset for the initial VADER analysis on 1000 rows of data, I analyzed the feature importances to see is `pos_score` would show up. Since `pos_score` ended up being in the top 25 features, I concluded that it was important to incorporate into the final model of this project. Since the dataset is already large and vaderSentiment analyses are very slow given my hardware constraints, I decided to only work with 50,000 rows of the dataset for the final modeling process.

Even though there will only be one-twentieth of the data in the final modeling process, I still used larger subsets of the data to narrow down the dataset to its most important features. With all features present and with all 1.1 million rows, the dataset was larger than my computer's RAM would allow. Using 500,000 rows of the original cleaned dataset, I used another random forest regressor to narrow down the 50 most important features that I would then keep for the final modeling process. I kept a list of these column names in a variable called `important_cols`.

Modeling: Like previously mentioned, I only used one-twentieth of the dataset and the columns that I previously identified as the most important for creating my model. While not ideal, it allows me to experiment with different methods within the time constraints of this project and the hardware constraints of my computer.

I ran the VADER feature engineering process to create the `pos_score` column for this particular dataset, and then saved the dataframe as a csv file on my computer called `df_ml.csv`. Then I dropped the `reviews` and `price_USD` columns to create my inputs to the model, and identified `price_USD` as the target of my model. After splitting the data into training and testing sets, I trained three different tree-based machine learning models, random forest, stochastic gradient boosting, and XGBoost, and used the root mean squared error to determine which model performed the best.

Tree-based ensemble models are ideal for this type of analysis, which uses a complex dataset with dozens of features and complex interactions between all of them. Trees allow for the flexibility to fit a model to the specific non-linear trends in the dataset without setting the degree of nonlinearity ahead of time. Ensemble models allow for the combination of several weak tree models to create a model that reduces the variance of the model without compromising too much on bias. Random Forest models are the most simplistic tree-based ensemble model that are built from many aggregated decision trees. The stochastic gradient boosting takes this one step further by improving and building upon iterative models based on their errors, while adding variation from choosing a random subsample of the data at each iteration. The use of subsets of data also makes it harder for the model to overfit to the training data. XGBoost takes gradient boosting even further by implementing second-order gradients of the loss function, which provides more information about minimizing the loss function of the model. It also uses advanced regularization which helps the model to generalize better in many cases. For this dataset, stochastic gradient boosting worked best, with the lowest RMSE score of 43.9.

Model Tuning: I used grid search cross validation to optimize the best performing model, which was the stochastic gradient boosting model. The grid search validation chooses the best model based on the negative mean squared error metric, which is a measure of how much the predicted value differs from the actual value.

RMSE Boxplots for Training and Validation Datasets: Boxplots showed that the stochastic gradient boosting model resulted in lower error during the cross-validation process than the XGBoost model, although the difference was not very far off, and the test set hasn't yet been evaluated (Figure 11).

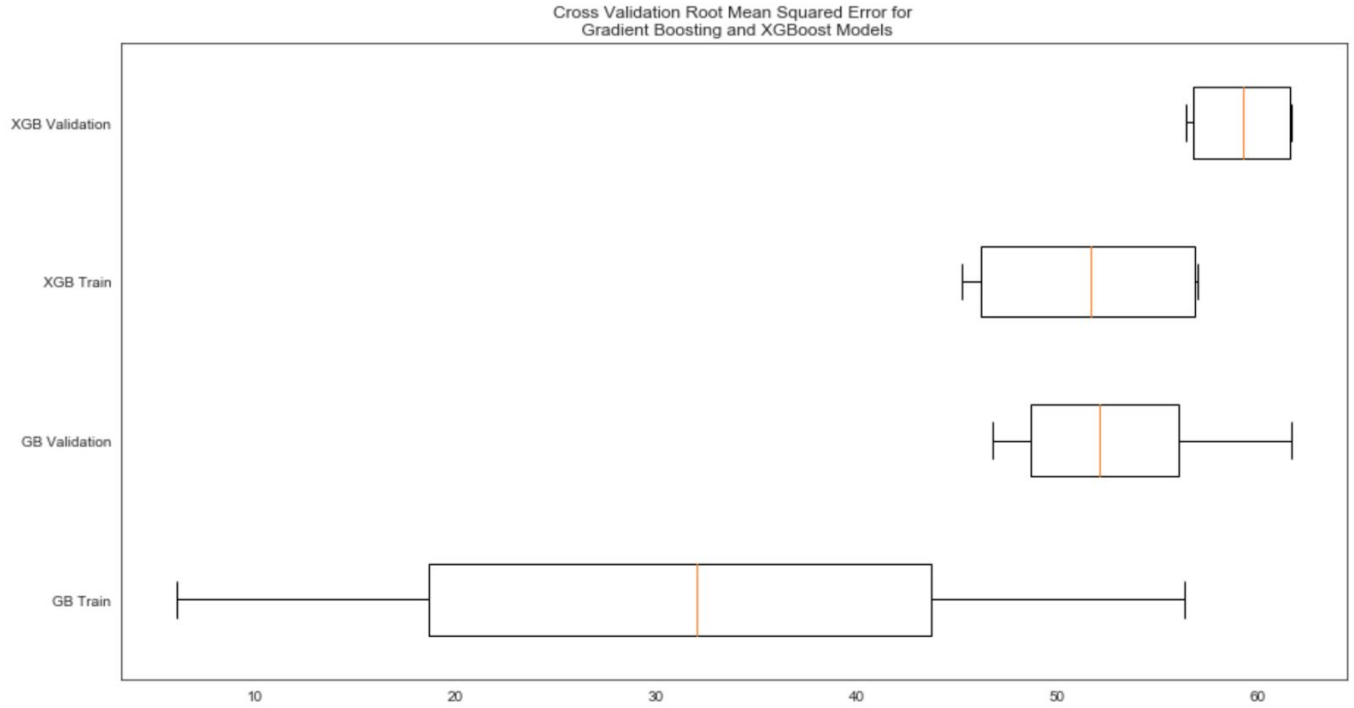


Figure 11. Box plots for the root mean square error for the XGBoost (top two) and stochastic gradient boosting models (bottom two).

Running Models on the Test Dataset: The initial modeling process used cross-validation to evaluate the ability of the models to generalize to data they haven't seen before, but it uses data that the models actually have seen in the process. A better test, however, is to use the test dataset, which is data that the model has not seen at any stage in the analysis.

The plots for deviance for the stochastic gradient boosting and xgboost models show the differences in error between the entire training set and the held-out test set, while the plots for feature importance show which features of the dataset were most important to the model (Figure 12).

As expected, the results for the training set for both models has less error overall than the results for the test set, with stochastic gradient boostings having less overall error. Additionally, there is significant overlap in the two model's most important features, which is evidence that the inputs to the models are likely to be reproducible between different types of models with varying parameters.

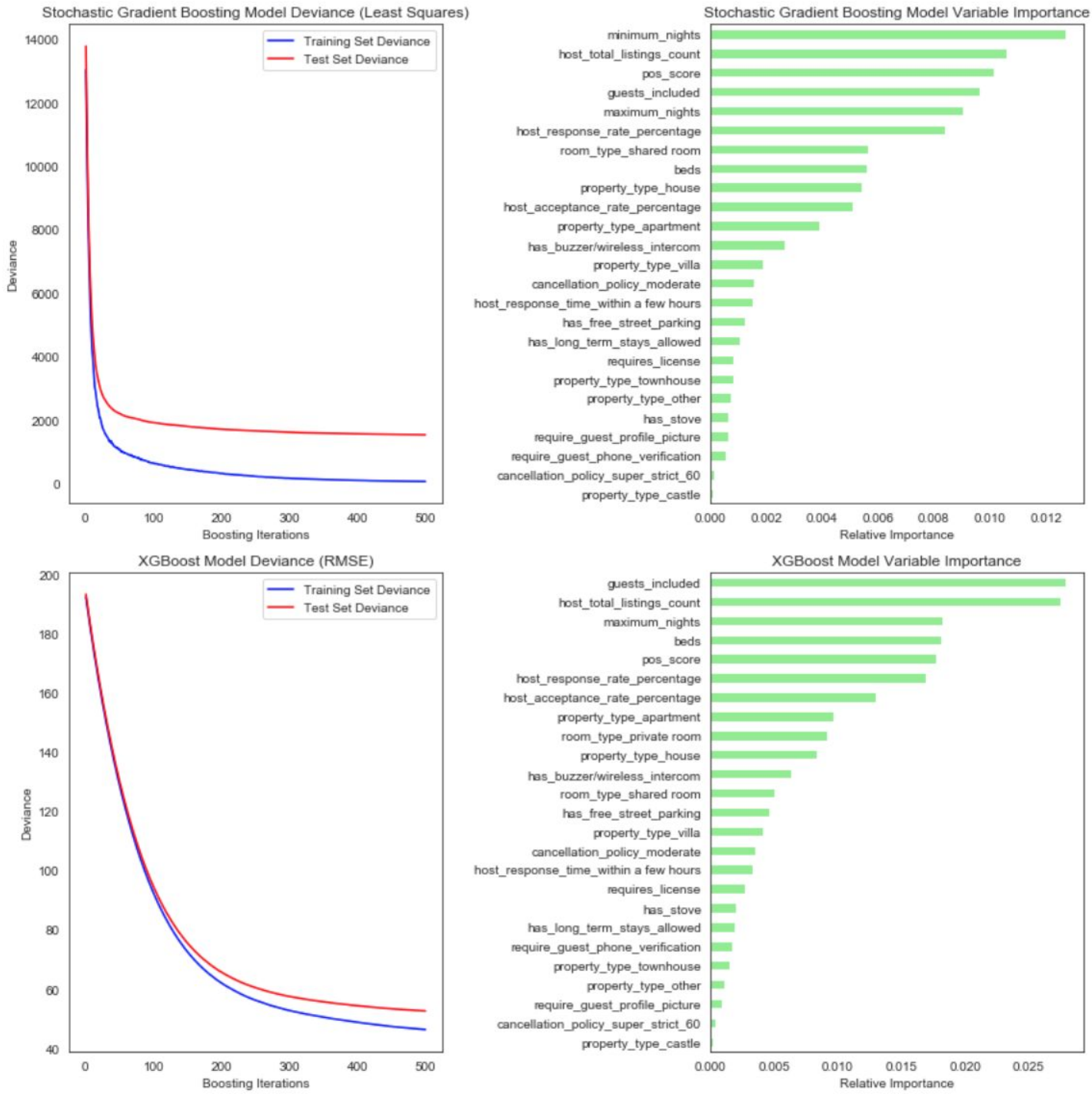


Figure 12. Model deviance plots for training and test sets of both models (left) and feature importance plots for both models (right).

Predicted Price vs. Actual Price: Comparing the predicted prices to their actual values communicates the strength of our model in a more understandable way. The scatter plots show this relationship for every point in the test set (Figure 13). For both models, there is a visible positive trend, which is evidence that the models are often predicting prices close to their actual values.

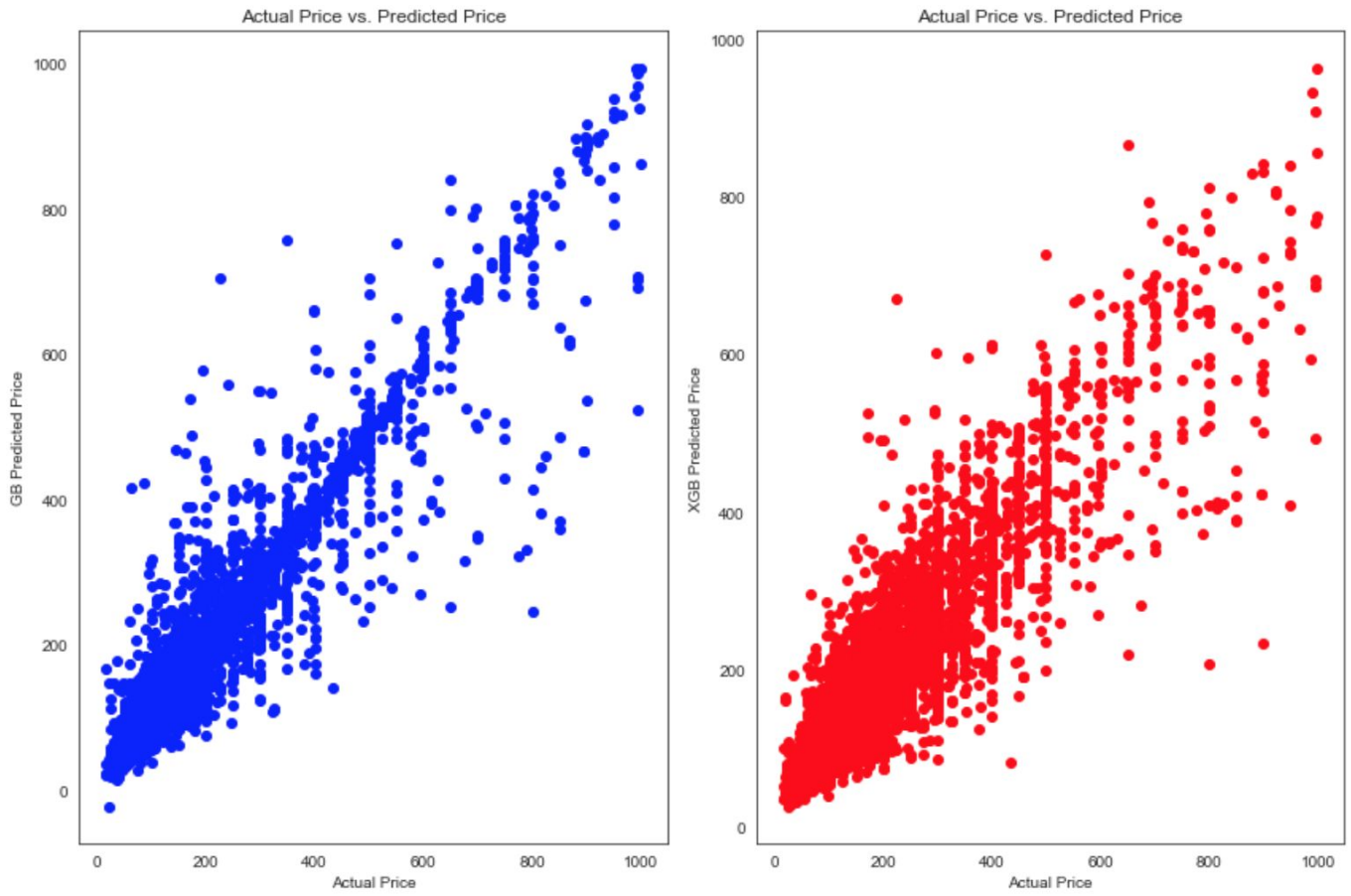


Figure 13. Scatter plots of actual prices in the test set vs. their predicted values by the stochastic gradient boosting model (left) and the XGBoost model (right).

Price Percent Change: To bring this analysis further, I calculated the percent change between the actual prices and their predicted values. For the stochastic gradient boosting model, 61.4% of price predictions are within 10% of their actual prices, while that number drops to about 32.5% for the xgboost model. In this sense, we see that the stochastic gradient boosting is much better for getting predictions that closer to their actuals than the xgboost model.

Conclusions & Next Steps

Conclusions:

The stochastic gradient boosting model for price prediction offers the opportunity for a company like Airbnb to increase their number of listings and be more transparent with its users. There are several uses for a model such as this, including:

1. For those looking to add listings, Airbnb can provide an estimate for how much a host can charge for their listing based on the most important features in the predictive model. For example, the potential host can input how many people they can accommodate, the amenities available, and the room type to get a range of appropriate price points, which could be calculated using the prediction and a confidence interval. This can help with encouraging people to follow-through with posting listings, or for forecasting the financial success of a listing.

2. The model can be implemented in marketing tactics to approach potential hosts with targeted advertising along the lines of "Have an extra private bedroom in Manhattan? You can make X dollars per month by putting your room on Airbnb!". In that way, Airbnb will amass more users and listings.

3. For existing listings, Airbnb can use the model to suggest improvements for hosts to easily increase the value of their listing(s). For example, one insight might be "Hosts that respond to messages within one hour increase the value of their listing by 10%" or "Add a buzzer or wireless intercom to your listing and charge $10 more per night". Hosts will likely appreciate suggestions for how to easy improve their listings and make more money. And as a result, Airbnb will increase its revenue too.

It's important to note that this model was only trained on a small sample of fifty thousand data points, but there are millions of other rows of data on which to train the model. In fact, adding more training data is probably one of the strongest ways to increase the accuracy of the model. A model with stronger predictive power will only increase the benefits that this model can bring.

Future Improvements: In order for the model to work best for the client, the most obvious next step would be to make it production ready. If given more time, I would implement a pipeline in which the data could be read directly into the program, automatically cleaned and processed for feature creation, and run through the machine learning model. This would allow for the client to use my model without having to go through all of the laborious steps themselves.

Looking more inside the model for the VADER sentiment analysis, I would do more preprocessing of the review data to remove non-English reviews to have a cleaner analysis that doesn't numerically de-value non-English entries. I would also like to include more string features into the dataset using VADER, like the listing description or rules data. Additionally, it would also be interesting to see what other NLP methods could be applied to the text data using a package like word2vec or doc2vec.

To make this analysis stronger, I would train these models on the entire Los Angeles dataset, and then to other cities represented in the entire Inside Airbnb dataset. The infrastructure for scaling this model has already been incorporated through the use of functions, but further generalization using object-oriented programming principles could make for an even stronger model. I would also spend more time tuning the model using Randomized Search or the more thorough Grid Search method.