COMP 4447

Final Project

Rodney Cordova

**Predicting Airbnb Listing Prices**

**Comparing Random Forests and XGBoost**

Decision tree-based algorithms are among the most common types of machine learning algorithms in use today. Decision trees function by following a tree like structure starting at a root node and splitting off many branches leading to a leaf node. Each branch represents a feature from the dataset and each internal node is a Boolean statement about the feature. The leaf nodes represent values for the the target variable. This framework can be used for both classification and regression models, making decision trees extremely flexible in application. Two of the most popular applications for decision tree algorithms are random forests and XGBoost.

In the following paper and the corresponding jupyter notebook I will be implementing both a random forest and XGBoost models on a common real-world dataset. The objective is to compare the performance of the two models to evaluate what method is better suited to making predictions in our specific scenario. To this end, this analysis will include data cleaning, feature engineering, exploratory data analysis, and model building and tuning.

**Dataset – Airbnb Listings:**

Since its founding in August 2008, the popularity of Airbnb has exploded. Airbnb provides a platform for hosts from around the world to rent extra bedrooms, condos, single family homes, and even mansions to guest for terms ranging from one day up to six months. With such diverse offerings, it makes sense that the price of short-term rentals can vary drastically. I will be using regression random forests and XGBoost models to attempt to predict the listing price of rentals in Denver, Colorado.

In order to get data on Airbnb’s listings my first thought was to use the Airbnb API. However, currently Airbnb is not accepting new access requests for their API. Therefore, the dataset I am using is from Inside Airbnb. Inside Airbnb sources publicly available information from the Airbnb website to aggregate data on listings, calendars, and reviews for many major cities all over the world. To build my models, I am using the June 28,2020 listings publication for Denver Colorado. The listings csv contains detailed information about all listings in Denver including price, neighborhood, host details, and much more totaling 106 features. The dataset contains numerous datatypes including strings, integers, and floats. I am also using a geojson file Inside Airbnb provides for the Denver neighborhoods for visualization purposes.

**Literature Review:**

Before implementing the models on the Airbnb dataset, it is important to fully understand how these methods work as well as their strengths and weaknesses. Doing so will allow us to fully interpret the results of our models. This literature review will examine “The Ultimate Guide to AdaBoost, random forests and XGBoost” by Julia Nikulski in order to establish the theoretical background of random forests and XGBoost (Nikulski, 2020).

Random forests, XGBoost and Adaboost fall into a category called tree-based ensemble algorithms. Tree-based ensemble algorithms are classified as weak learners. A weak learner is simply a learning algorithm that makes predictions slightly better than randomly. These ensemble algorithms have many advantages over other types of algorithms including being non-parametric, they can handle mixed data types, and are robust against overfitting and outliers. As a result, they can be used in a wide variety of applications that other algorithms cannot.

Central to the functionality of tree-based ensemble models is boosting and bagging. Boosting is the combination of many weak learners (trees) into a very accurate prediction. Boosting operates in a sequential process of making predictions for multiple rounds on a training dataset and iteratively improving the performance over previous rounds based on the errors of the previous rounds. On the other hand, bagging uses non-sequential learning. For each round of bagging a random subset of the training set is drawn, with replacement. The samples are used to build multiple decision trees. The average prediction of the multiple trees is then used as the overall prediction.

Random forests were developed in 2001 and are based off bagging. Random forests function on the idea that the overall prediction of many loosely correlated models is more accurate than a single model. To this end a collection of trees are built using subsamples of the training data with a random subset of features. The main advantage of random forests is that they are not heavily affected by noisy data and generalize well. One of the main disadvantages of random forests is training complexity. Hyperparameters that must be considered in any random forest include the number of features, number of trees, maximum depth of trees, whether to bootstrap samples, the minimum number of samples left in a node before a split, the minimum number of samples left in the final leaf node and more. Tuning of this many hyperparameters requires a lot of time and may result in overfitting.

Compared to random forests, XGBoost is a much newer algorithm. It was first introduced in 2016. XGBoost utilizes gradient tree boosting. Gradient tree boosting follows the boosting process listed above utilizing regression trees. Julia Nikulski describes the process as;

*“For each iteration i which grows a tree t, scores w are calculated which predict a certain outcome y. The learning process aims to*[*minimize the overall score*](https://medium.com/greyatom/a-quick-guide-to-boosting-in-ml-acf7c1585cb5)*which is composed of the loss function at i-1 and the new tree structure of t. This allows the algorithm to sequentially grow the trees and learn from previous iterations. Gradient descent is then used to compute the optimal values for each leaf and the overall score of tree t. “*

Overall XGboost builds off the best parts of boosting and random forests and adds sequential tree growth, gradient descent, parallel processing and regularization. The main advantage of XGBoost is speed and the reduction of variance due to its regularization parameter. Like random forests, XGBoost has a wide array of hyperparameters that must be tuned. Another disadvantage is that extreme values will lead to underfitting of the model making it not as resilient as random forests.

**Data Cleaning:**

Prior to building our models, preprocessing of the data needs to be conducted in order to ensure the dataset is in a form that can be used by the models. The first step is retrieving the data. An http request is used to download the dataset csv from the Inside Airbnb website. Then the csv is read into a pandas data frame for processing. The initial dataset consists of 106 features and 4084 observations. Features are split between integer, float, and object datatypes. The objects are mostly strings with some missing values.

**I – Feature Reduction:**

With 106 features, the dataset is quite large. Later we will be using one-hot encoding to convert categorical features into a usable format. This will only increase the number of features in the dataset. Therefore, we will start by removing features that are redundant or may cause problems with our models. Both of our models only function with integer, float, and categorical features.

There are eight URL features in the dataset for the listing information, host information, and images. These can not be used in the model, so they have been dropped. Additionally, there are 16 text features. Each of these features contain strings of various lengths with information about the listing or host. For example, the ‘transit’ feature describes the types of public transportation in the vicinity of the listing. It is possible to use natural language processing to extract useful information from the text features. However, for the purpose of this analysis we will instead drop these features from the dataset.

In some cases, multiple features contain the same information. When this happens one of the duplicate features should be removed. An interesting example of this is the ‘street’ feature. This feature shows the city and state of the listing. City and state each have their own features in the dataset so street is the concatenation of those features and should be dropped. Seven duplicate features were dropped for similar reasons.

The trees at the heart of random forests and XGBoost, use Boolean statements based on the features to split the internal nodes. If a feature only has a single value, then it will not be useful in the construction of trees. Therefore, the seven features with single unique values have been dropped.

**II – Feature Engineering:**

Often there are features that can be used within the models but are not in formats or datatypes that the models are expecting. Thus, feature engineering is used to transform the data into a useable format.

‘host\_response\_rate’ and ‘host\_acceptance\_rate’ are percentages representing the frequency a host responds to inquiries and provide acceptable solutions. Both features are strings (ex ‘100%’, ‘89%’, ‘75%’). They have been converted to floats by stripping the ‘%’, converting the remaining string to a float and dividing by 100 resulting in the decimal form of the percentage.

The dataset contains many dates. On their own the dates are not very useful, but we can convert them into a form that is. ‘host\_since’ is the date a host joined Airbnb. By converting ‘host\_since’ and ‘last\_scraped’ to pandas datetime objects we can calculate the difference between the two dates then dividing by 365 will produce the number of years a host has been on Airbnb as of the date the data was pulled. The result was saved as a new feature called ‘host\_for\_years’. The same process is applied to ‘first\_ review’ and ‘last\_review’ to get the years since the respective review.

The scrape data is information related to how the data was obtained by Inside AirBnb and does not have any baring on the listing price. Now that we are finished using it for calculating the updates to ‘host\_since’, ‘first\_review’, and ‘last\_review’ all the dates and scrape features can be dropped.

At his point we are left with sixteen categorical features. The categorical features are comprised of strings. Differences in the capitalization of the strings could result in extra levels being created. For example, ‘bed’ and ‘Bed’ would be different levels. Therefore, all the strings in the categorical features were converted to lower case. The ‘zipcode’ feature required some extra processing. Some observations had an extra ‘co’ in some strings, for example ‘co 80220’, so that was stripped along with any blank space. Missing values in ‘zipcode’ were strings of ‘nan’ those were replace with np.nan so that we could get an accurate assessment of missing values in the next step.

**III – Missing Values:**

To determine how best to handle missing values, I started by plotting the missing values (see Appendix A). For each feature in the plot, any observation that is white is a missing value. In total there are 22,371 missing values. Notice the target variable (price) is not missing any values but several other features are missing data. As a rule of thumb, features missing more that 60% of the values should be dropped, ‘square\_feet’, ‘weekly\_price’, and ‘monthly\_price’ are all missing over 90% of their values so they were dropped.

XGBoost has built-in methods for handling missing values. This is done by passing a sparse matrix to the boosting algorithm. The missing values are blanks within the matrix and the boosting algorithm can process them without issue. Random forests do not have a built-in method for handling missing values. Therefore, missing values are typically imputed. For categorical features, the missing values are imputed with the mode of the feature and for continuous feature the missing values are imputed with the mean of the feature. Since the objective of this analysis is to compare the performance of the two models on the same dataset, I decided to use the random forest method for imputing missing values in both cases.

**Exploratory Data Analysis:**

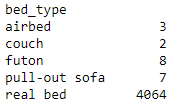
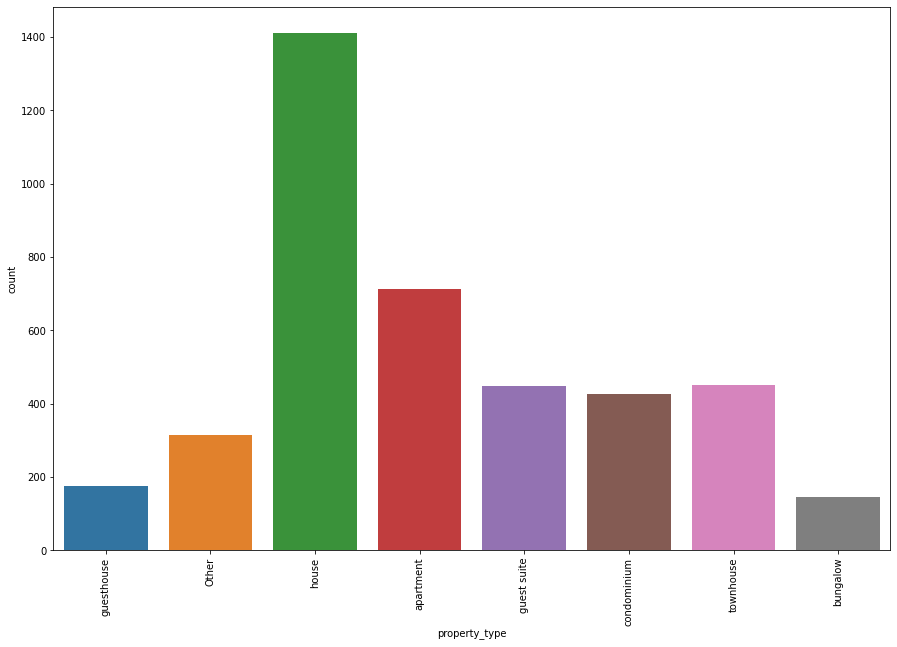
To begin EDA I decided to take a look at the distribution and box plot for the target variable of ‘price’ (see Appendix B). In the first version the data is extremely compressed. This is due to several outliers. There were ten observations in excess of three standard deviations of the mean with the max value at 10,000. Due to the large influence of these outliers, they were dropped, and the process was repeated.

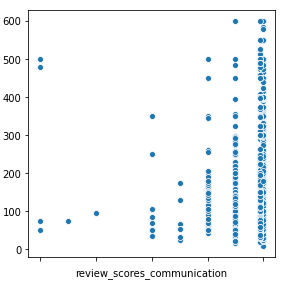
This time around the histogram appeared normal but the box plot was still compressed. When checking for outliers, the number of outliers increased to 106. These outliers were still clearly having an impact on the data, so they were also dropped.

In the third round, there is still a large presence of outliers. However, the impact of those outliers on the distribution of the data is visually much smaller. We know that tree-based ensemble algorithms are robust against outliers and noise. Thus, at this level I feel comfortable preceding with the outliers present. In fact, the outliers can in part be explained by pricing variations within the neighborhoods in Denver. To observe the difference in price by neighborhood I produced a choropleth of the mean price by neighborhood (see Appendix C). The neighborhoods closest to downtown Denver tend to have a higher price (dark red) well the neighborhoods furthest from downtown tend to have a lower price (light red). On the price histogram we also can see that the data is heavily skewed to the right. Typically, this would indicate that we may want to conduct normalization but since both our models are non-parametric, we do not need to do any transforming of data.

Next, we can look at the relationship between price and our other features with a correlation heat map (see Appendix D). Overall, the correlations between price and the other features (positive or negative) are small. With that said, there are strong correlations amongst many of the other features. Tree-based ensemble algorithms are typically good at handling multi-collinearity and features do not need to be removed or engineered to decrease correlations. Given the large feature set size with the current dataset I believe removing some of the highest correlated features may improve the model performance by having less features overall.

To determine the features to remove I extracted the features with absolute correlations greater than 0.9. Nine total features were returned. Of note our calculated field ‘host\_for\_years’ is among the highly correlated feature as it is highly correlated with ‘host\_id’. Host id’s are given to hosts as the enroll in Airbnb and are incremented sequentially with each host added. Thus, the lower host’s id the longer they have been a host. The highly correlated features were dropped from the dataset.

To get an idea of the distribution of levels within the categorical features I decided to use count plots (see Appendix E for an example or see jupyter notebook for all plots). Upon review I noticed several of the features have extremely imbalanced distribution of levels. To the left are the count of each level for the ‘bed\_type’ feature. When plotted, all levels other than real bed are indistinguishable from 0 because 99.5% of the observations fall into one level. We could drop the observations that are in the other levels, but this would mean losing the information those observations contribute to other features. Instead, I decided to group the rare levels together into an “Other” level. This was done by first determining the frequency of each level. Then a threshold is set as the minimum level frequency that is greater than 0.1. The mask and map functions are used to group all levels that have frequencies less than the threshold into the “Other” category (Gilad, 2020). This transformation was applied to ‘city’, ‘bed\_type’, ’property\_type’ and ‘host\_neighbourhood’. Overall, this transformation was successful at reducing the number of rare levels in the categorical features. Below is the updated count plot for property type. In total 11 levels were combined into ‘Other’.

For the final part of EDA I decided to plot scatter plots for each continues feature against price (see Appendix F for an example or see jupyter notebook for all plots). As suspected based on the correlation plot, there seems to be very little evidence of a relationship between price and any of the continues features. One thing that did stand out is several of the features related to reviews seem to group around 1,2,3,4 and 5 rather than a continues scale. These review survey features fall into a category of data called ordinal Liker scale data. There is much debate on how to handle this type of data as either continuous or ordinal. In many parametric models, treating Liker Scale data as continuous could be problematic. Tree-based ensemble models are non-parametric thus do not require data to follow any distribution. Therefore, in this analysis I will be treating the Liker scale data as continuous.

**Model Building**

Before constructing the models there is some last-minute data clean up that needs to be done. As previously mentioned, random forests and XGBoost require float, integer, or categorical data. We will be using sklearn to construct the models and it does not directly support categorical data. Therefore, we use one-hot encoding to convert out categorical features into numerical feature that sklearn can handle. The dataset is first split into two datasets. One containing the dependent target variable (y) and another containing the remaining independent variables (x). Then using np.get\_dummies all the object features are encoded. After the encoding we check the data types for both datasets and they now are all in a numerical form.

The next thing to do is randomly split the data into training and testing datasets. The training set is used to build, train, and tune the models and the testing set is used to assess the performance of the model. I decided to go with a 70% training 30% testing split. Meaning the training set has 2858 observations and the testing set has 1226 observations.

In terms of accessing the performance of the models I will be using mean absolute error (MAE). How does MAE work? For each prediction in the testing set a prediction error is determined by subtracting the predicted Value from the actual value. The absolute error is then determined by taking the absolute value of the prediction error. Then calculating the mean of all the absolute errors produces MAE, MAE represents the mean absolute value of the difference between the predicted value and actual value. Therefore, the smaller the MAE the more accurate the model is at predicting the actual value (M, 2018).

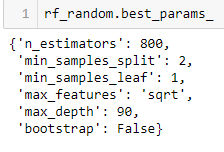
**I - Baseline Model:**

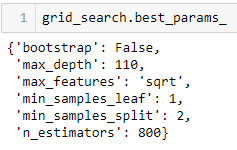
For comparison purposes I built a baseline model. The baseline model is simply the mean of the training price data. When the baseline model is applied to the testing data the resulting MAE is 62.0476.

**II – Random Forest:**

To begin a base random forest was trained using 1,000 trees and the default sklearn random forest regressor parameters. When this base random forest is used to predict the testing set the resulting MAE is 33.8743. Already we see a significant improvement in MAE compared to the baseline model.

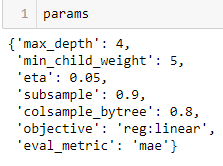
The next step is hyperparameter tuning. As previously mentioned, random forests have lots of parameters that can be tweaked to improve the performance of a model. Individually tuning each parameter would take an exorbitant amount of time so I will be using two methods to speed up the process, random search cross validation and grid search with cross validation. The cross validation in both these methods refers to K-Fold cross validation. In K-Fold CV the training set is split into K subsets (folds) then iteratively fit the model K times. With each fit training the data on K-1 of the folds. The K-Fold process is repeated many times using different model parameters. The best model is then selected and trained on the complete training set (Koehrsen, 2018).

In the random search cv, we start with a wide grid of hyperparameter ranges and randomly select from the grid to preform K-Fold CV. I am using 3 fold cv on 100 random candidates meaning 300 total fits are evaluated to find the best preforming set of parameters. The best parameters selected by the random search are to the right. When a random forest with the new parameters is used to predict the test set, we get a MAE of 32.8760. Thus, the parameters select did improve our model.

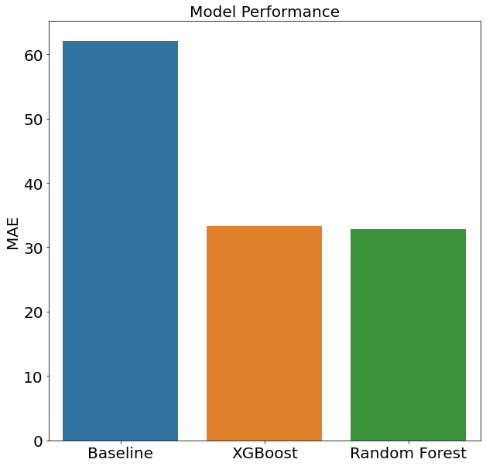
The parameters from the random search are then used as a basis for the grid search. The grid search operates in a similar manner to the random search but instead of testing random parameters it tests each combination within the grid. For example, the ‘max\_features’ parameter has a grid of 1,3,5, and 7. Each will be used to fit a model along with each other combination of features. As a result, the total number of candidates we have increases to 2800 and with 3-fold cv the total number of fits increases to 8400. The best parameters selected by the grid search are to the right. With these final parameters the random forests MAE decreased to 32.9951 when predicting the test set. The overall improvement from the baseline model it 29.0525 and hyperparameter tuning resulted in a 0.8792 improvement.

**III – XGBoost:**

For the XGBoost model we will start by fitting a model with the default parameters. We set the number of boosting rounds to 999 but the actual number of rounds preformed should be less than that. This is because early stopping rounds is set to 10. After each round the MAE is checked and if 10 rounds pass without the MAE improving the model will stop. With early stopping the best MAE occurred in round 23 and was 33.5633. Like the random forest, we will be using grid search cross validation to do hyperparameter tuning on the XGBoost model. One difference being we will only use the training MAE to judge the parameter tuning until the end when the fully tuned model is applied to the test set (Cambridge Spark, 2019). Using cv with the current parameters the best MAE returned is 36.7595. Again, I am using a grid search for tuning, but we will do the search with only a couple parameters at a time rather than all of them as in the random forest.

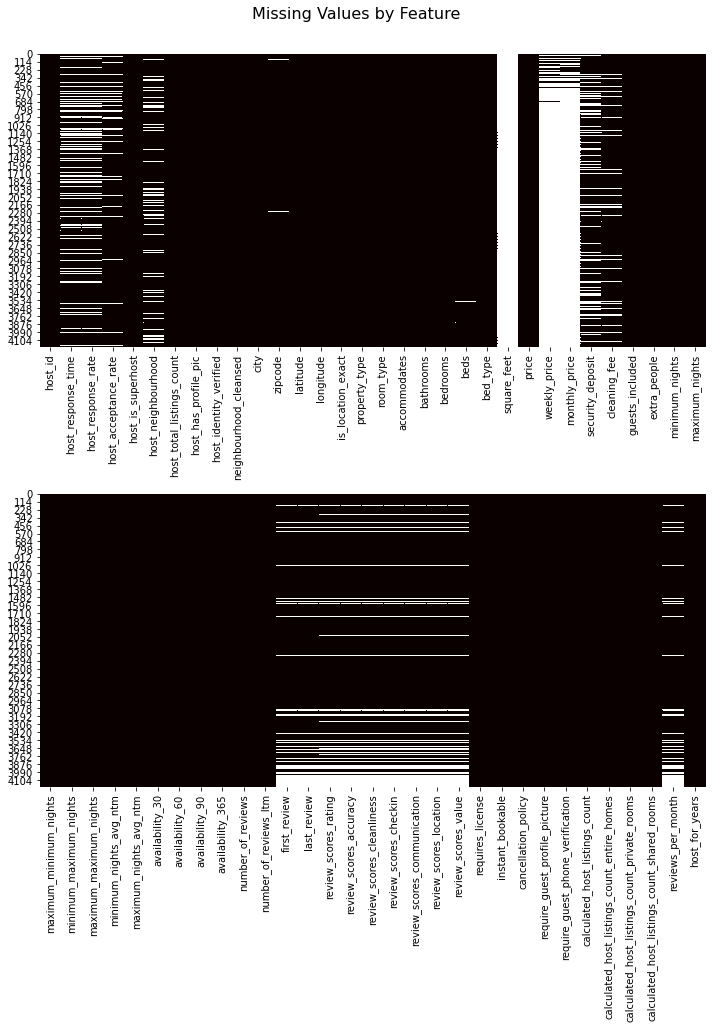
 I started the search by tuning the ‘max\_depth’ and ‘min\_child\_weight’ for values in the range 1 to 30. The best MAE produced by the search was 37.7403 in 10 rounds with ‘max\_depth’ of 4 and ‘min\_child\_weight’ of 5. The parameters are updated to reflect the best ‘max\_depth’ and ‘min\_child\_weight’ and we move on to check the ‘subsample’ and ‘colsample’ for values in the range 1 to 10. From this grid search the best of MAE was 46.8042 with ‘subsample’ of 0.9 and ‘colsample’ of 0.8. Finally, we will tune the learning rate, ‘eta’, for values of [.3, .2, .1, .05, .01, .005]. The best MAE result was 35.1127 in 293 rounds and the best ‘eta’ was 0.05. Putting it all together, the best set of parameters produced by the grid search are to the right. When the final tuned model is applied to the test set it results in a MAE of 33.2781 for an improvement over the baseline of 28.7694 and a hyperparameter tuning improvement of 0.2852.

**Conclusion:**

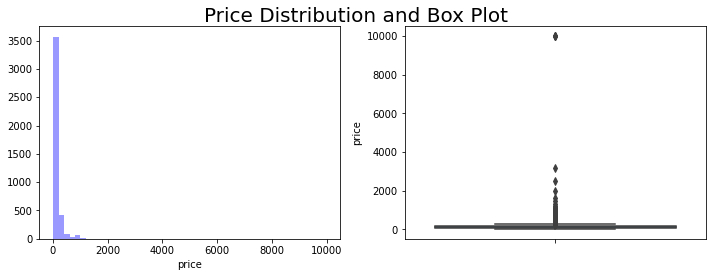
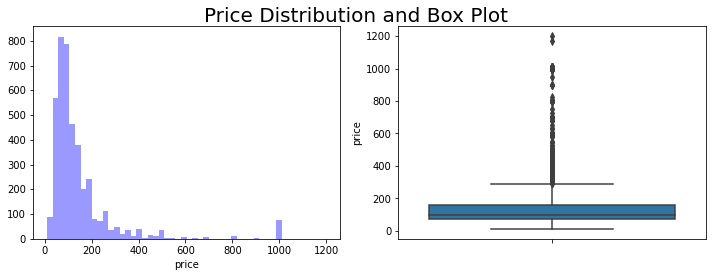
Overall, the random forest and XGBoost models preformed very similarly when it comes to predicting listing price. The MAE difference between the two models, 0.4282, is so small it is almost not visible when plotted but the random forest came out slightly better and both models preformed much better than the baseline prediction method using the training set mean. One interesting way that the models differ is on feature importance. Feature importance is a scoring method used by both random forest and XGBoost to denote how useful each feature is in predicting the target variable within the model. If we look at the top 10 features in each model (see appendix G and H) we see that the random forest ranks number of bathrooms as the most important feature well XGBoost ranks the number bedrooms as most important. Furthermore, the remainder of the top 10 features differ drastically by model. It is interesting that the models preformed similarly with such drastically different important features. Random forests are typically very robust against noise in the dataset. While noise in the dataset can lead to underfitting with XGBoost. Despite my best efforts to clean the dataset, it remains very noisy. I believe the difference in performance in our models is largely due to the noise in this particular dataset.

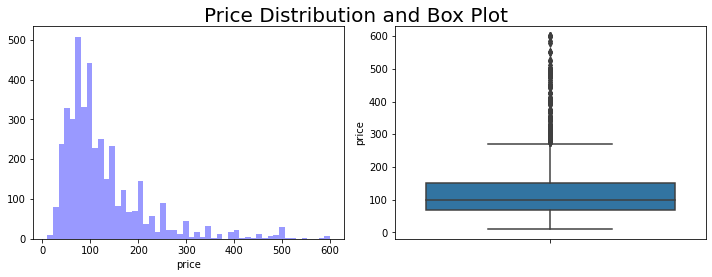
The objective of this analysis was to compare the performance of XGBoost and random forests on predicting AirBnB listing price. The models both performed relatively well at this objective. With that said, given more time these models could be improved. Additional feature selection could be done to optimize the features included in the model and reduce noise. Furthermore, natural language processing on the text fields that were originally dropped from the dataset could yield some useful features. I believe this is a good start, but future revisions could go a long way in improving the overall model fit.

Appendix A

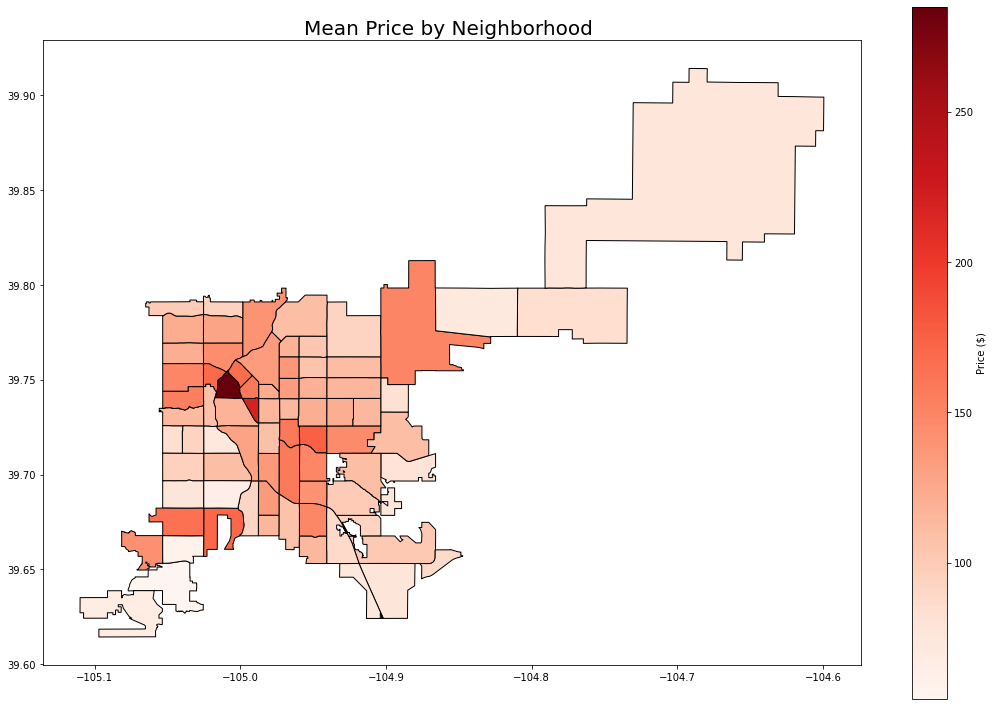


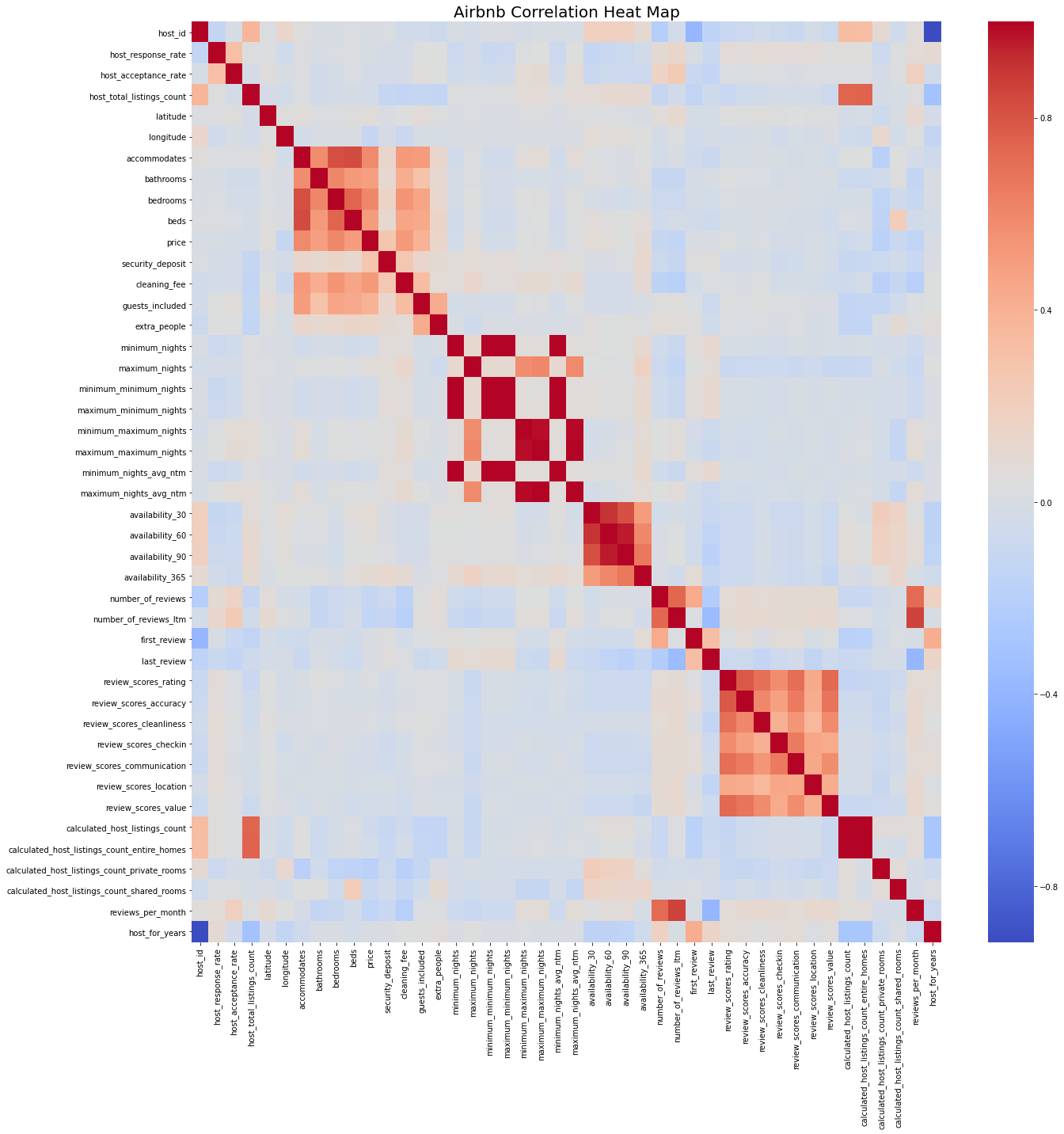
Appendix B

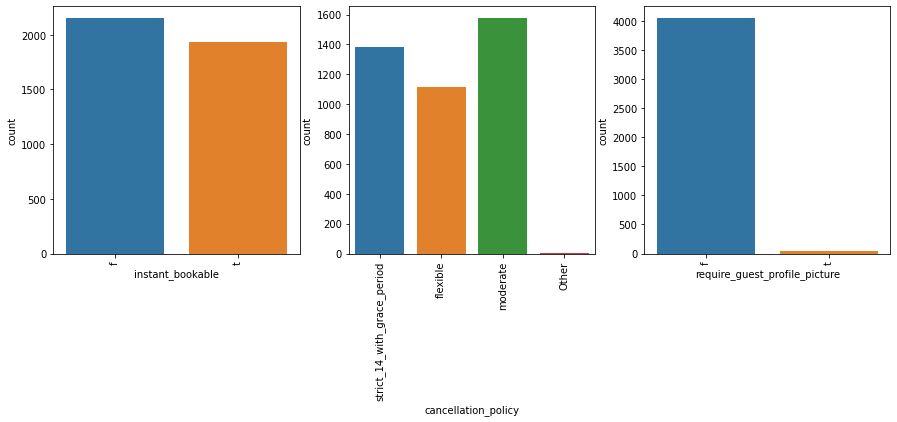


Appendix C

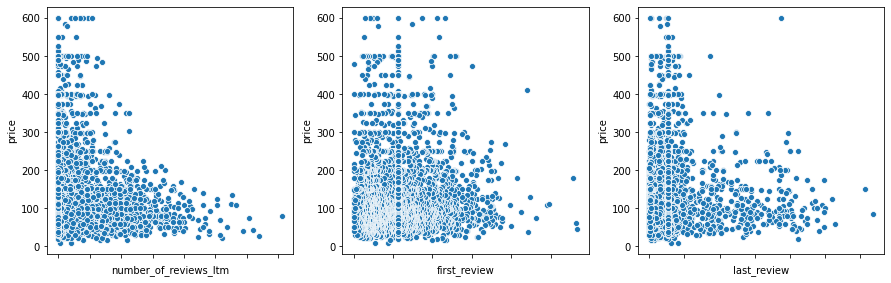


Appendix D

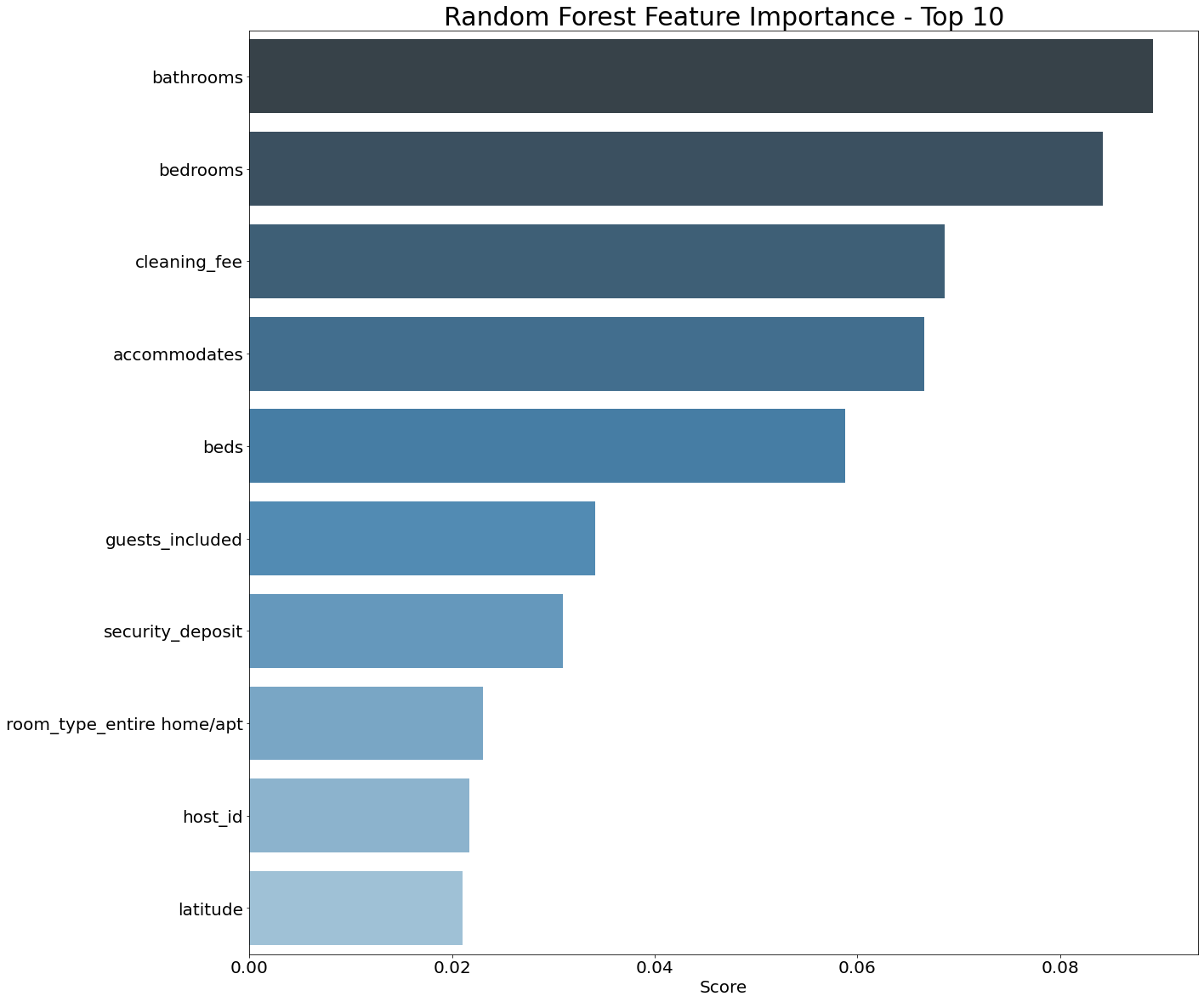
Appendix E



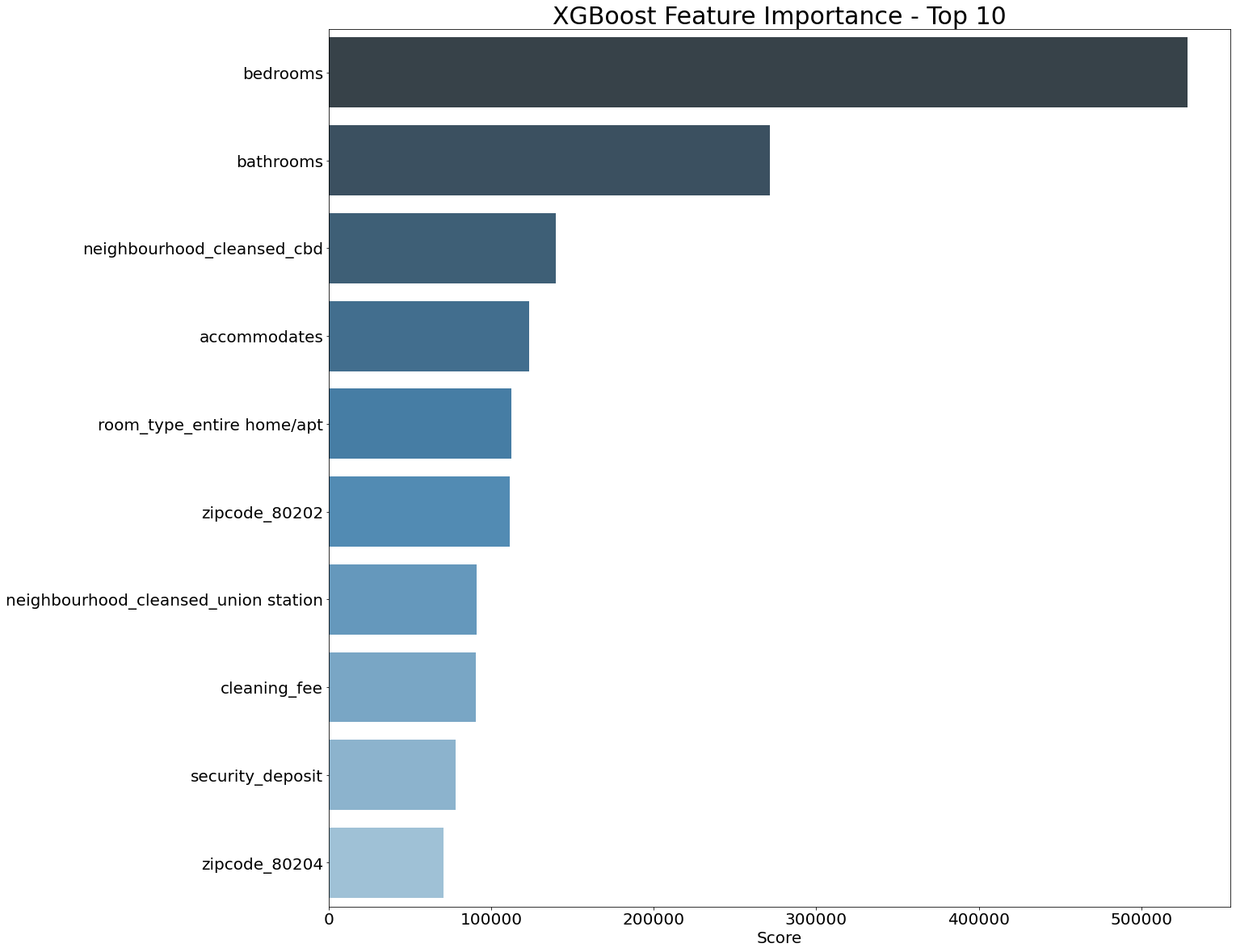
Appendix F



Appendix G



Appendix H



References

Cambridge Spark. (2019, December 23). Hyperparameter tuning in XGBoost. Retrieved August 24, 2020, from https://blog.cambridgespark.com/hyperparameter-tuning-in-xgboost-4ff9100a3b2f

Gilad, M. (2020, February 06). Handling rare categorical values in Pandas 🐼. Retrieved August 24, 2020, from https://medium.com/gett-engineering/handling-rare-categorical-values-in-pandas-d1e3f17475f0

Inside Airbnb. Adding data to the debate. (n.d.). Retrieved August 24, 2020, from http://insideairbnb.com/

Koehrsen, W. (2018, January 10). Hyperparameter Tuning the Random Forest in Python. Retrieved August 24, 2020, from https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

M, E. (2018, February 22). Mean Absolute Error ~ MAE [Machine Learning(ML)]. Retrieved August 24, 2020, from https://medium.com/@ewuramaminka/mean-absolute-error-mae-machine-learning-ml-b9b4afc63077

Nikulski, J. (2020, March 19). The Ultimate Guide to AdaBoost, random forests and XGBoost. Retrieved August 24, 2020, from https://towardsdatascience.com/the-ultimate-guide-to-adaboost-random-forests-and-xgboost-7f9327061c4f