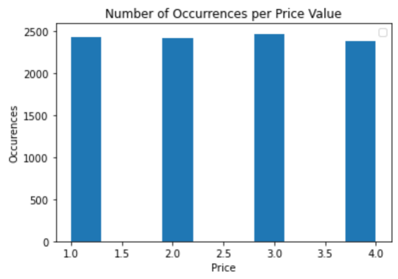
671 Kaggle Competition Writeup

Joey Li

March 5, 2021

1 Exploratory Analysis

First, I went through and simply checked what the unique values were for each given feature. From this, I noticed the 'is_business_travel_ready' feature had only one unique value, and thus discarded it. Likewise, I decided to discard the ID feature since it seemed unlikely to be related to the price in any meaningful way. Using a simple histogram plot, I checked that the data was balanced – there were roughly the same number of samples for each price value from 1 to 4.



Next, to get an idea of which features would be most useful toward predicting the price, I made violin plots plotting each variable against the price, as displayed in Figure 1. For variables which had too many unique values, I plotted the price along the x-axis and the feature along the y-axis, but in general, I plotted the price along the y-axis and the feature along the x-axis. Unsurprisingly, beds and room_type were important, but other features such as cancellation policy also appeared to be significant.

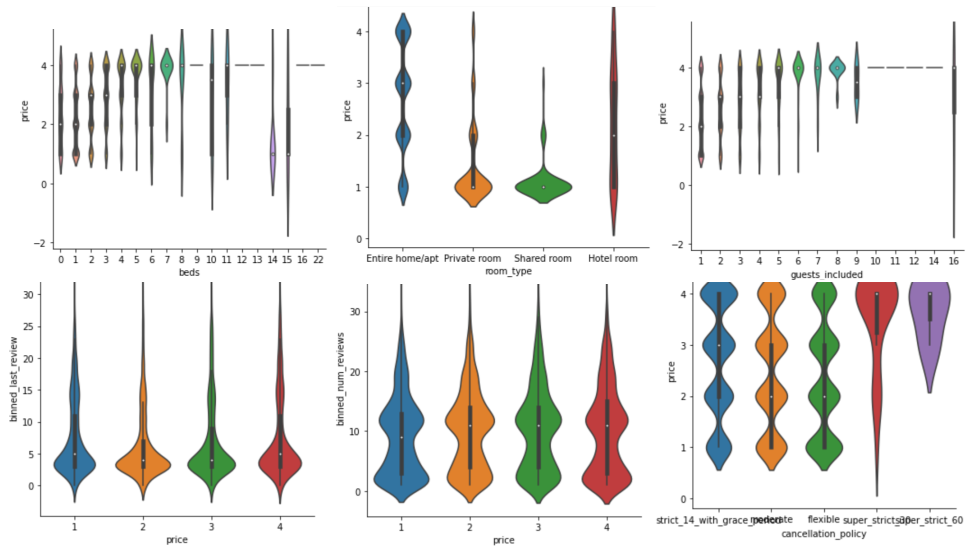


Figure 1: Violin Plots of Various Features

Most machine learning algorithms, including one model I used, the sklearn implementation of random forest, operate on numerical inputs. Thus, I used a few techniques to encode the categorical features into numerical features. For features such as ‘host_since’ and ‘last_review’ which recorded dates, I transformed these into new features ‘num_months_since_host’ and ‘num_months_since_last_review’ where the “current time” used in computing these features was taken to be the latest date among the values for the respective feature. For the remaining categorical features, I used the OrdinalEncoder from sklearn to encode them into numerical data, trying two different encoders. The first was a default encoder which randomly assigned numbers to the unique values of each categorical variable, ignoring whether there might be a natural ordering of values within a feature. For the second, I manually created the mapping between categorical values and numerical values, to preserve certain natural orderings of features in the encoding, such as the ordering from ‘flexible’ to ‘super_strict_60’ in the ‘cancellation_policy’ feature.

After converting my data to numerical features, I was able to make some more plots to visualize each feature. I borrowed some code from [1] to create a scatter plot of each feature versus each price point, with noise added to each data point to get a rough indication of how numerous each combination of feature value and price point was. This turned out to be only somewhat helpful in visualizing the impact of various features. I also computed a 2D PCA embedding of the encoded numerical data to see if there were discernible clusters corresponding to different price points. This was inconclusive – at least in two dimensions, the data was not separable.

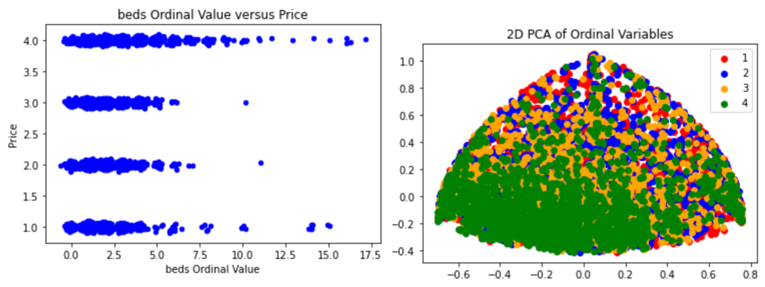
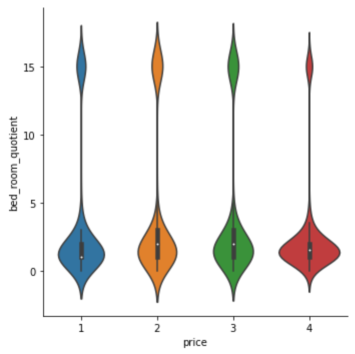


Figure 2: Two Representations of the Encoded Numerical Data

Examining certain features such as ‘extra_people’, it became clear that they had some extreme outliers. For the first model I used, random forest, this would not have mattered, but for the second model, k-nearest neighbors, it had a significant impact on the calculation of distances between data points. To adjust for this, I manually binned some values based on intuition about which values ought to be “equivalent.” In particular, the features ‘minimum_nights’, ‘number_of_reviews’, ‘months_since_last_review’, ‘calculated_host_listings’, ‘availability_365’, ‘number_of_months_since_host’, ‘extra_people’, and ‘maximum_nights’ were binned. Following binning, the data was also encoded using an ordinal encoder in order to be usable by random forest. Although as mentioned above, random forest is not affected by outliers, binned input was still useful for training the random forest in that the data was regularized in order to help with overfitting concerns.

Finally, I tried engineering two new features to help with classification. One was the bed_room_quotient, which is the quotient between the number of beds and bedrooms. Intuitively, one might expect that a lower quotient would correspond to a higher price point and vice versa, since more beds per bedroom would intuitively correspond to cheaper lodging and vice versa. Likewise, I defined the bed_bath_quotient between bedrooms and bathrooms. Making corresponding violin plots for these new features, they seemed potentially helpful, but training a random forest with these additional features did not measurably improve performance, and thus these features were not used in the final model.



2 Models

I employed two different models: a random forest and a k-nearest neighbors model.

Random forest was my first choice, since it is naturally adaptable to multiclass classification and can deal with the categorical data upon performing ordinal encoding. Additionally, random forest is generally a very powerful method able to learn nonlinearities well. Finally, I had used the sklearn package for random forest in the first homework assignment and knew it was straightforward enough to set up as a first working model. It is not particularly computationally efficient, but this was not an important constraint – I focused on getting a working model running before worrying about optimizing. The specific method I used for implementing random forest was sklearn.ensemble.RandomForestClassifier, which does not natively deal with categorical data.

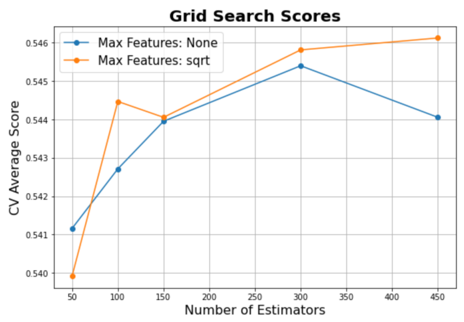
The k-nearest neighbors model was an attempt to take a very different approach using a method we had not covered in class, which I believed could be made to deal with the mixed data on hand more naturally. Through some Internet searches, I learned about the Gower distance, which is a metric on mixed data, that is, data with both categorical and numerical features. In particular, the Gower distance uses Hamming distance on categorical features and scaled Manhattan distance on numerical features and is a linear combination of these individual distances between feature values. This seemed particularly promising because it deals with certain categorical features such as 'neighborhood' more effectively. An ordinal encoder maps each neighborhood to a number, and correspondingly introduces some notion of order among neighborhood, which is generally undesirable since there is no natural order on neighborhoods. Using the Gower distance, however, one instead computes the Hamming distance on the 'neighborhood' feature, which more directly corresponds to a notion of alikeness between data points. Aside from these considerations, the k-nearest neighbors approach is also relatively friendly in that there are good, easy-to-use libraries online, such as sklearn.neighbors, which I used, and the model is easy to train and use. One main drawback of this approach was computational inefficiency. I used the gower package on pypi, developed by Michael Yang, to compute the distance matrix corresponding to my training points using the gower distance, and this took approximately 5-6 minutes. However, fortunately, this computation only had to be carried out once, so it was not prohibitive.

3 Training

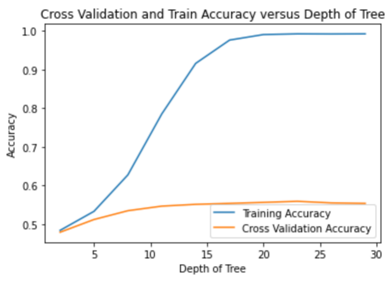
For random forest, the sklearn.ensemble module implements a standard version of the random forest algorithm, with one significant difference in that when computing the final prediction for a given data point, the sklearn implementation averages the probabilistic predictions of individual trees, rather than having each tree vote for one class and averaging those results. Otherwise, it follows the same protocol as any random forest model. It uses bagging, where for each step the model randomly chooses some subset of the data and some number of features as specified by max_features, and then computes a decision tree based on that subset of data and those features, using either the gini or entropy criterion to determine node splits, as specified in the construction of the classifier. Thus, fitting the model is done by simply choosing random subsets of features and fitting decision trees to these subsets. The decision tree fitting itself is done by DecisionTreeClassifier, using the CART algorithm, and at each split, the features for consideration are randomly permuted before choosing the feature which allows for the greatest decrease in entropy or gini index. The process ends when a maximum number of splits or a minimum number of nodes which we want to split is reached. I measured 3.14 seconds for fit time and 3.37 seconds for fit and predict in wall time for my final version of the random forest with parameters chosen to be n_estimators = 150, criterion = ‘entropy’, max_features = ‘sqrt’, min_samples_split = 5, max_depth=11.

For my version of k-nearest neighbors, I use a precomputed distance matrix, and thus the fitting and predicting should be fast – given the distance matrix, computing the k nearest neighbors to a point x is equivalent to finding the k smallest values in the row of the distance matrix corresponding to x, which can be done in $O(nk)$ or $O(n \log n)$ per [2]. The KNeighborsClassifier documentation is not particularly clear on how it is optimized for the case of a precomputed distance matrix, but we may assume it performs at a roughly comparable runtime. Thus, the main step which must be optimized for computational efficiency is the actual computation of the Gower matrix, which is done by the gower package published by Michael Yan. This is implemented in the obvious way, where a given matrix is split into categorical features and numerical features, the numerical features are rescaled to take values in a range of 0-1, and the distances are computed using the Hamming distance for categorical features and the Manhattan distance for numerical features. The main way this computation is optimized is simply by doing vectorized operations, as provided natively by NumPy. In this way, we are able to compute the distance matrix in 250 seconds of wall time for our input training data set of 9681 samples, and the fitting and transforming for our final classifier, which uses 21 neighbors and uniform weights, take approximately 3.03 seconds of wall time.

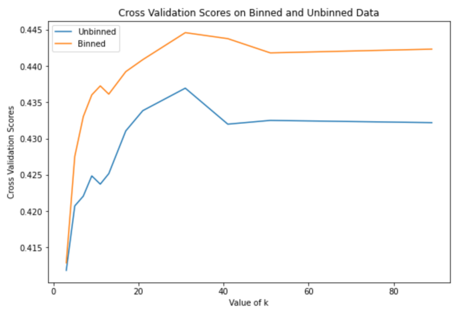
4 Hyperparameter Selection



For random forest, the main hyperparameters to tune were max_features and n_estimators, per the sklearn documentation. I took inspiration from [3], beginning with a randomized grid search, considering both gini and entropy criterion for splitting, ten values of n_estimators in the range from 50 to 500, values of None and 'sqrt' for max_features, nine values for max_depth in the range of 10 to 90, values of [2,5,10] for min_samples_split, and values of [1,2,4] for min_samples_leaf. The randomized search ran for 50 iterations, choosing random values for each of these features and creating a random forest based on these values. Printing out the top 10 classifiers by cross-validation accuracy, I found that using 'sqrt' as the argument for max_features generally performed better, the default parameters for min_samples_split and min_samples_leaf performed well enough although min_samples_split = 5 performed a little better, larger values of max_depth performed better, the entropy criterion performed somewhat better than the gini criterion, and generally models with more estimators performed better.



However, the plot to the right shows that values of max_depth larger than around 11 began to indicate overfitting. Thus, I fixed max_depth to be 11. Following, I performed a more systematized grid search on simply the values for max_features and n_estimators. Viewing the top classifiers from this grid search, it again became clear that the choice of max_features = 'sqrt' performed better, and larger values for n_estimators performed better, though there were diminishing returns after about 150 estimators, as seen in the figure on the previous page. The small dip in the orange line at 150 estimators seemed inconsequential, and so I chose the final parameters n_estimators = 150, max_depth = 11, criterion = 'entropy', min_samples_split =5, and max_features = 'sqrt'. During this grid search, it was also determined that the mean fit time for this model was approximately 3.47 seconds, with the mean score time around 0.008 seconds. For a model with 300 estimators, the mean fit time was around 6.76, with a score time approximately 0.25. Since the score difference was negligible, I stuck with the smaller model. To check once again that there were no issues of overfitting, this final model was specifically reevaluated and the training accuracy was around 0.78 with a cross-validation accuracy of 0.55, which I deemed acceptable.



For the k-nearest neighbors approach, one can see in the figure to the right that binning the data gave a noticeable improvement in cross-validation accuracy. The main hyperparameter to tune was the value of k. I evaluated the 5-fold cross-validation accuracy of the model for twelve different values of k, ranging from 3 to 89, not all equally spaced. The largest value was chosen to be 89 since [4] suggested that an optimal value for k-nearest neighbors is, in practice, sometimes around $\sqrt{n}$ where n is the number of data points. Given that we had 9681 data points and approximately 4/5 were used for training in each validation split, the value $k = 89$ was chosen to be an odd number around $\sqrt{4/5 \times 9681}$. I could also have chosen a number around $\sqrt{9681}$, but this likely would have produced largely similar results – from the figure above, one can check that the validation accuracies level off after about $k = 21$. One other hyperparameter which had to be chosen was the weighting system, which could either be unweighted, counting the contribution of each of the k-nearest neighbors equally, or weighted, which would upweight the contribution of nearer neighbors

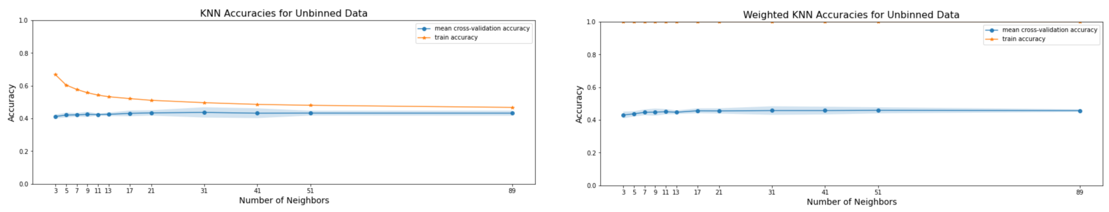
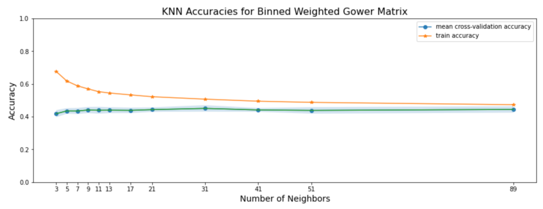


Figure 3: Uniform vs Weighted KNN Accuracies

among the k-nearest neighbors. From Figure 3, one can see that the weighted version generally had training accuracies close to 100% with validation accuracies approximately equal to that of the unweighted version. Thus, in my final model, I opted for the unweighted version. The other hyperparameters for the KNeighborsClassifier from sklearn.neighbors are related to optimizing the computation time for fitting the data, and I did not tweak them since I had no issues with training time. The final parameters chosen were $k = 21$ and weights = 'uniform', which gave a mean cross validation accuracy of 0.43 and a training accuracy of 0.51, which most certainly was not overfitted.



One final parameter for the k-nearest neighbors classifier was the actual computation of the Gower distance – we could have computed it differently, giving weights to the relative contributions of each feature toward the Gower distance. Based on intuition from the violin plots, I formulated some relative importances of features and used this in computing a new Gower matrix and ran k-nearest neighbors on this matrix, but unfortunately, the results were largely the same, as shown above.

5 Data Splits

To split the training data for cross validation, I used the KFold method from sklearn.model_selection to define a KFold object with 5 folds, and then consistently used this same KFold object for cross-validation each time so that direct comparisons could be made between the scores of different models. To avoid issues of overfitting, I used a plot_cross_validation method adapted from the first homework assignment which plotted both the cross validation accuracy and the training accuracy across different values of the hyperparameter. If the training accuracy became too high, I readjusted the hyperparameters. For example, running weighted k-nearest neighbors always achieved near 100% accuracy, and thus I chose to use unweighted k-nearest neighbors instead.

6 Errors and Mistakes

One other model I considered before trying the k-nearest neighbors approach was boosted stumps, which achieved 5-fold cross validation scores of approximately 0.5. This was lower than random forest, my first approach, but turned out to be higher than the k-nearest neighbors model I developed later. Perhaps with more hyperparameter tuning and some more feature engineering, boosted stumps could have performed better.

I also began developing my own version of the Gower distance matrix, to allow weighting different features differently when computing Gower distances. Unfortunately, my algorithm was extremely computationally inefficient. Furthermore, I found out very late on that the gower package already had a built in functionality for incorporating weights – it was not mentioned in the documentation, and I only discovered this after reading through the base code itself. Thus, I wasted some time trying to rewrite what had already been done.

The hardest part of the competition was simply making improvements upon the standard models for this classification problem. I had many ideas for how to proceed and could certainly continue to work on the same problem for a good amount of time without running out of ideas, but, unsurprisingly, most of the things I tried were ineffective.

7 Predictive Accuracy

My username is Joey Li. The random forest model consistently achieved a mean 5-fold cross validation score of approximately 0.55, while my k-nearest neighbors approach achieved a 5-fold cross validation score of approximately 0.43. Several graphs illustrating the performance of these models have been displayed throughout this report, but are included here again for convenience, with the specific model chosen displayed by a point:

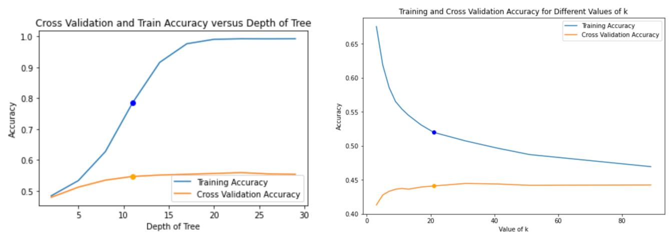


Figure 4: Validation and Train Accuracies of Final Models

References

[1] How to graph grid scores from gridsearchcv. https://stackoverflow.com/questions/37161563/how-to-graph-grid-scores-from-gridsearchcv. Accessed: 2020-11-18. 2

[2] k largest (or smallest) elements in an array. https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/. Accessed: 2020-11-18. 4

[3] Hyperparameter tuning the random forest in python. https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd7 Accessed: 2020-11-19. 4

[4] How can we find the optimum k in k-nearest neighbor? https://www.researchgate.net/post/How_can_we_find_the_optimum_K_in_K-Nearest_Neighbor. Accessed: 2020-11-19. 5