Final Project: Airbnb Prices in New York City

Data Science II (STAT 301-2)

Noah Holubow

15 March 2021

Contents

Overview & purpose

The goal of this project is to predict the expected nightly price of an Airbnb rental in New York City. Airbnb is a vacation rental marketplace based in San Francisco, California. Customers can browse rentals, see pictures, read reviews, and book all from the web platform or via the mobile app. As of September 30, 2020, Airbnb has 5.6 million active listings worldwide in over 100,000 cities and in over 220 countries and regions. According to its website, "the toal price of a reservation on Airbnb is based on the nightly rate set by the host, plus the addition of other fees or costs determined by either the host or Airbnb." These "other fees" include service, cleaning, extra guest, security, and tax fees. However, it is seemingly arbitrary how the base nightly rate is set by the host. Or perhaps there is not a whole lot of rhyme or reason to the price setting schemes used.

In an attempt to investigate this, I have designed this project around that very topic. I am primarily investigating the relationship between nightly price and a host of other factors, enumerated below. The dataset is confined to New York City and its five boroughs: Manhattan, Queens, Staten Island, Brooklyn, and The Bronx.

Data sets

My data source comes from Kaggle, which is a data science company and online community of data scientists. Kaggle is a subsidiary of Google. As specified above, this dataset covers Airbnb bookings in New York City in 2019. There are a total of 48,895 transactions recorded. The dataset provides a good mix of categorical and numerical data. Columns of interest include NYC borough, room type, minimum number of nights, number of reviews, reviews per month, host listings, yearly availability. These are the regressors. The outcome variable is then, of course, the nightly rate.

The dataset can be found here.

Loading packages & setting the seed

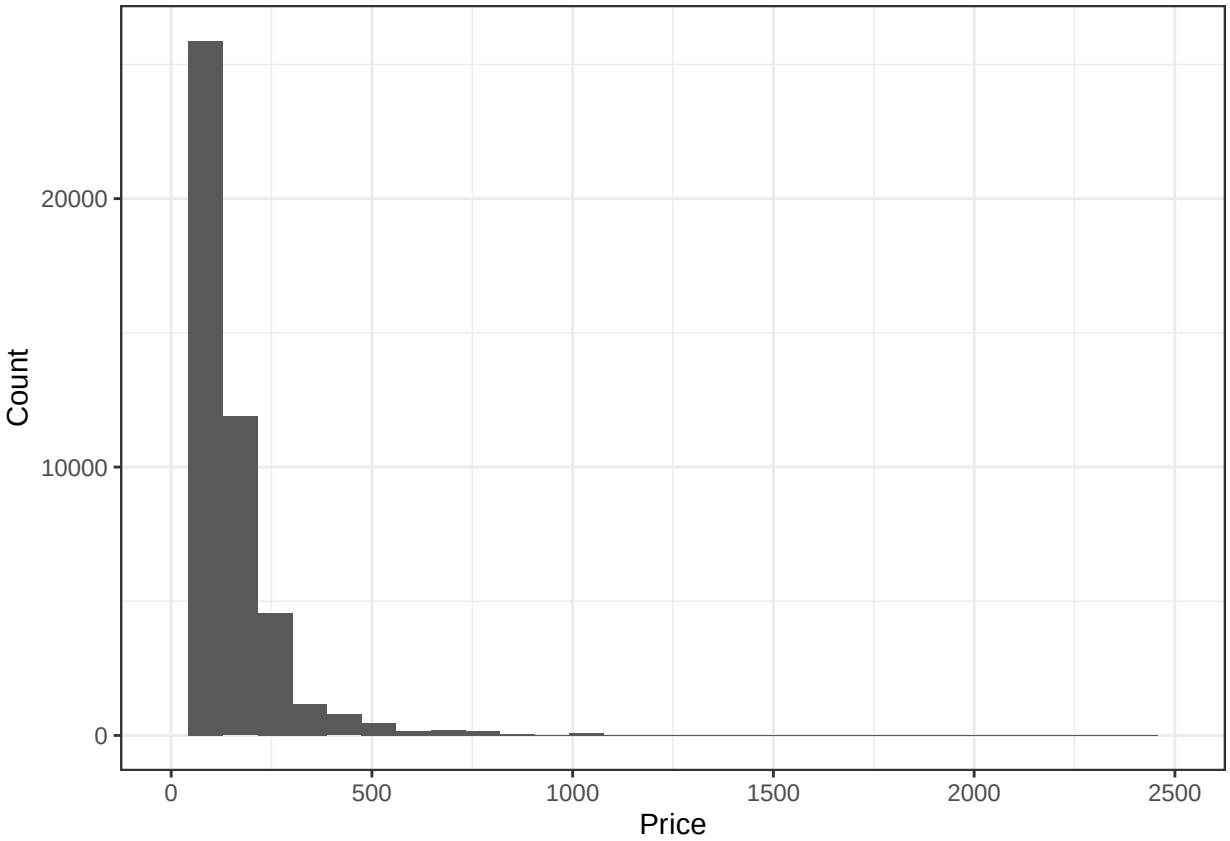
In order for the code to properly function, some packages must be loaded to facilitate the execution properly. Additionally, I am using a pseudorandom splitting process, whereby I will set a seed to reproduce the same results each time.

Importing & cleaning data

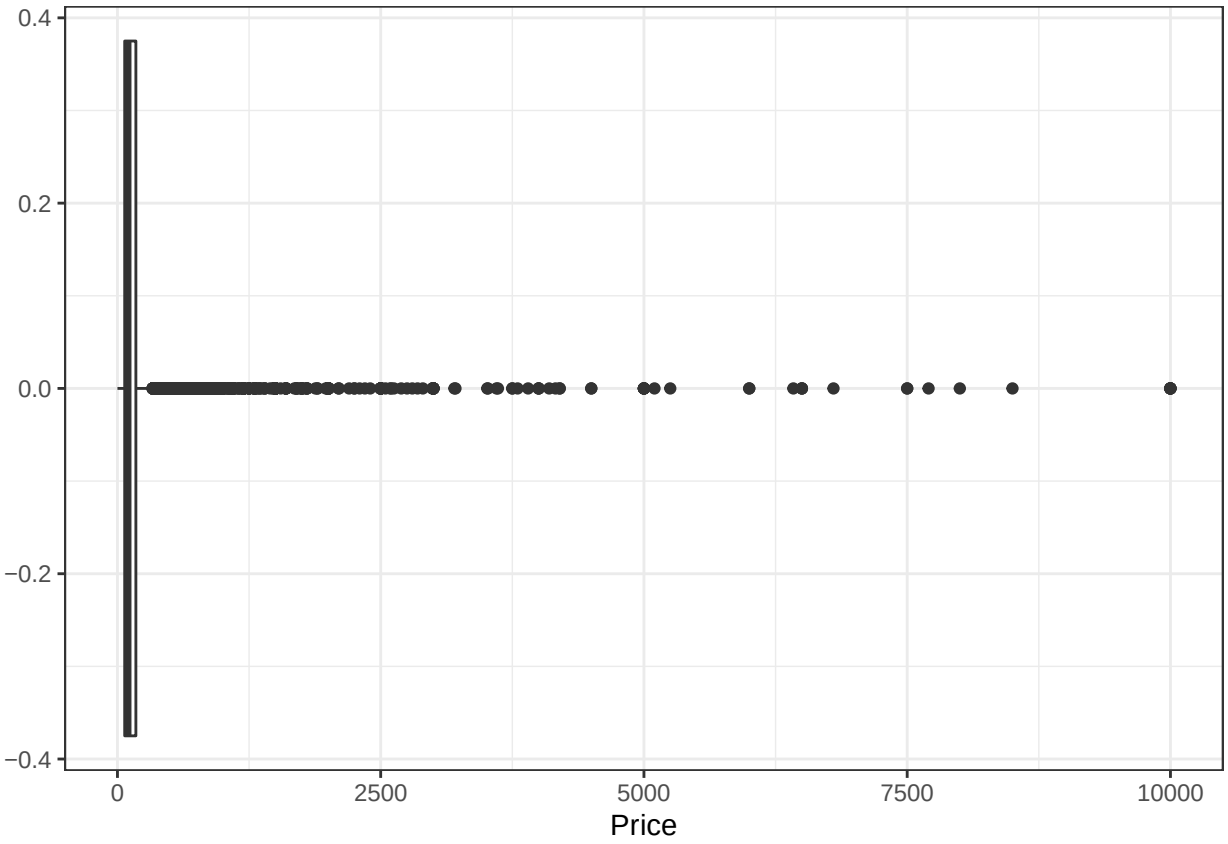
The following section imports and cleans the datasets to be used for analysis later on but is hidden for purposes of keeping the report tidy. Overall, the data looked very good in terms of missingness. I had 100% completion rate for all but one variable, `reviews_per_month`. Fortunately, since this is a continuous variable that has a fairly wide range, I imputed the missing values in my feature engineering in order to still include these observations. Aside from this, there was not a ton of data cleansing that needed to be done.

Exploratory data analysis

Prior to beginning the process of feature engineering, I took a brief look at the data to get a sense of what I am working with in this dataset. The first step was to look at the outcome variable, `price`. A quick histogram showing the distribution of prices reveals the following:

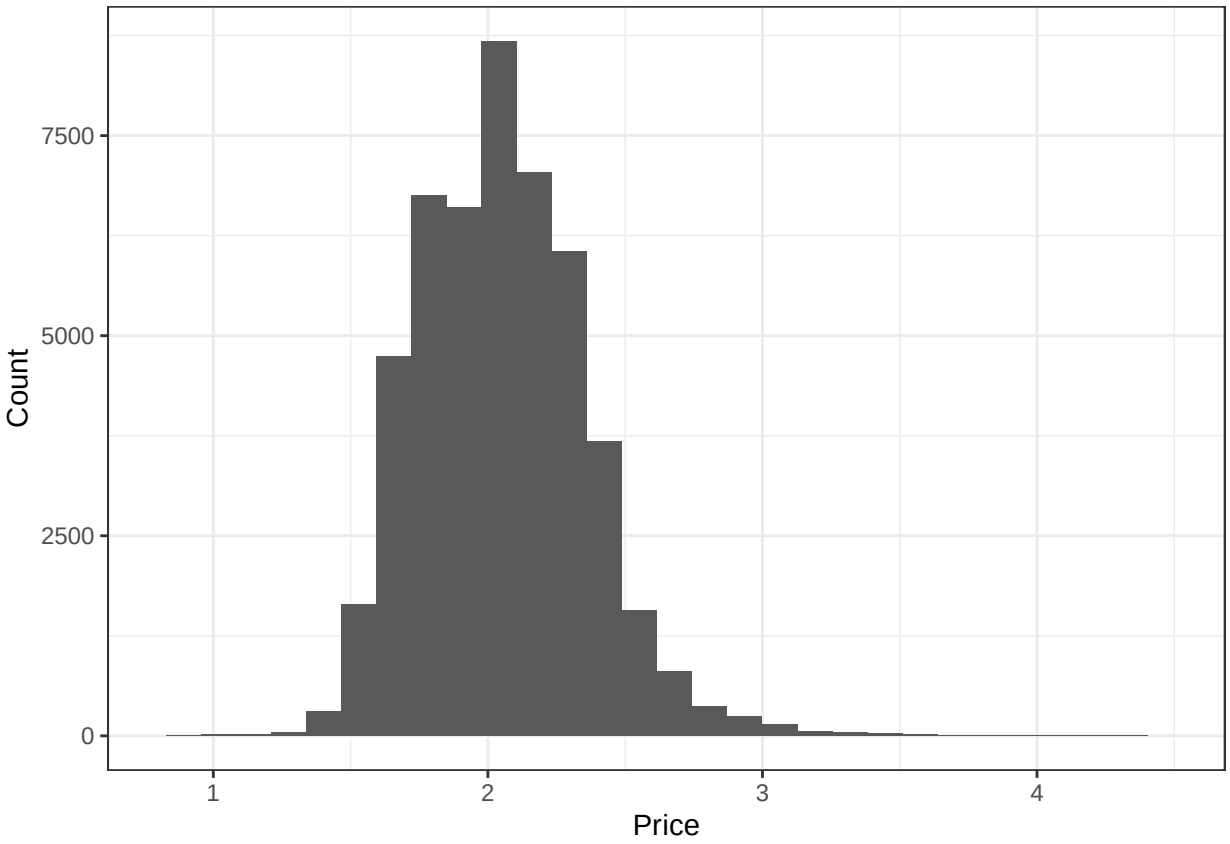


The prices are very significantly right skewed. This is most likely due to the fact that a decent Airbnb rental is going to fall into a typical bin (about $0-$250 or possibly a bit less). Outrageously expensive prices are not common but are also not totally out of the picture. There are a few in the $1000+ range here and there, causing the right skew. Looking at a boxplot reveals the same as well:



Because there is so much skewness, it makes sense to log-transform the price using a log of base 10. There are some observations with a price of 0, and since $\lim_{x \to 0} \log_{10} x = -\infty$, I removed these from the dataset entirely so as not to have values of `NA`.

Following a simple log-transformation and removal of prices with a value of 0, the following histogram now shows the distribution of prices:



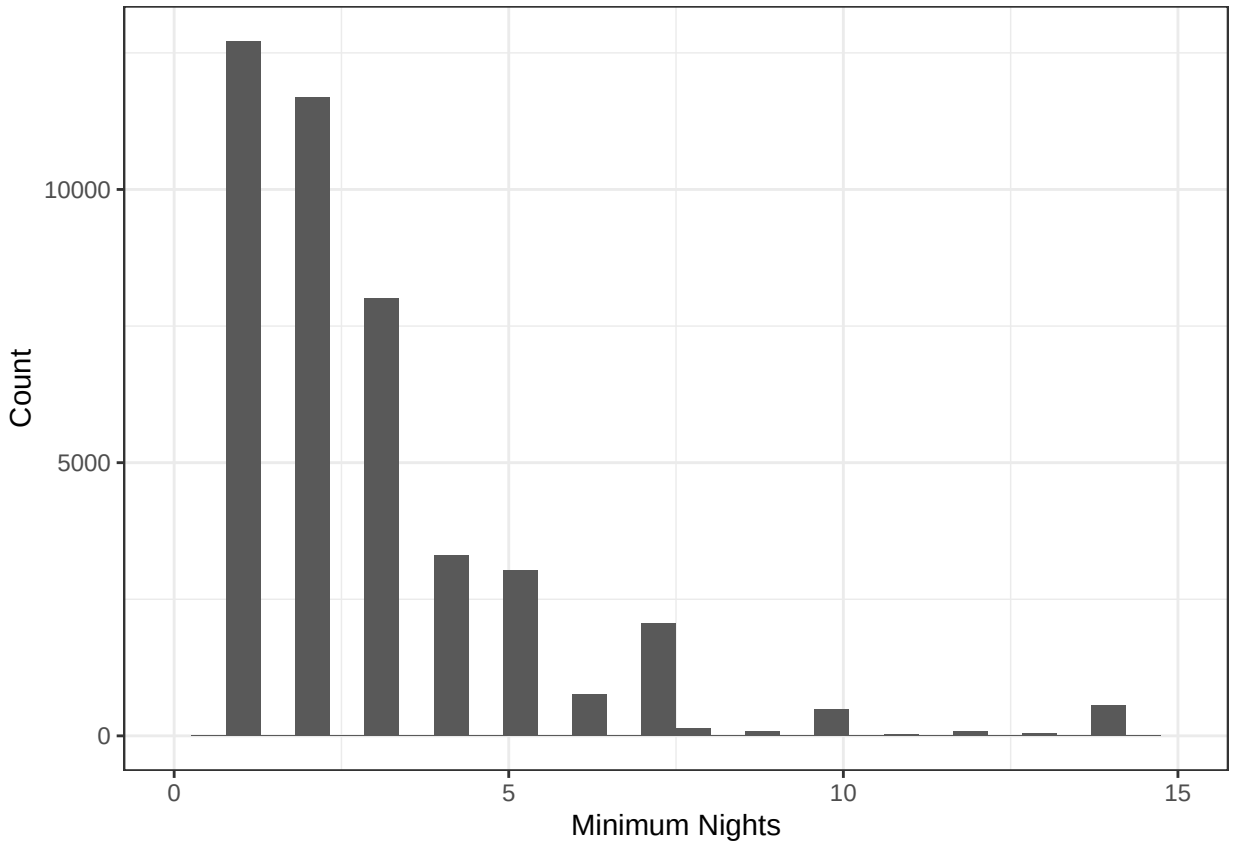
Now we can see that the distribution is much more normal with little skewness. Its average is well-centered, so this will work much better for the model. I have chosen to restrict the bounds on this plot from 0.8 to 4.5 of $\log(price)$, but the data is well captured within these bounds, if not all of it.

It is also worthwhile to inspect some of the other variables, too. One obvious variable that is categorically appealing is `neighbourhood_group`. This really just defines the borough in which the rental is located. We can see the breakdown of rentals by boroguh in a simple table:

Borough	Rental Count
Bronx	1090
Brooklyn	20095
Manhattan	21660
Queens	5666
Staten Island	373

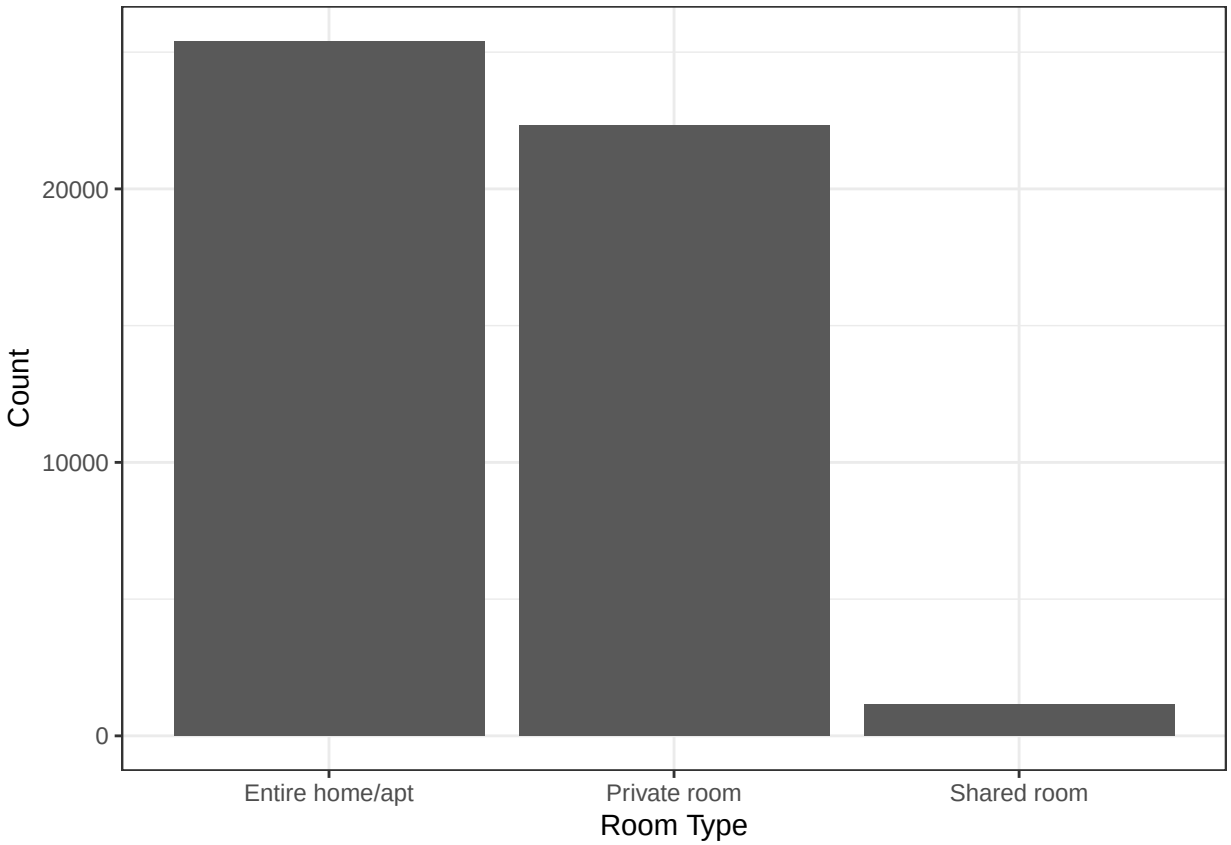
It is worth noting that much of the data come from the boroughs of Brooklyn and Manhattan, with a still decent number from Queens and relatively few from The Bronx and Staten Island. This does pose the question of whether or not the data should be stratified by price or neighborhood. In the end, I decided, as is described below, to stratify by price since there is some skewness, even after the log transformation.

It is also interesting to briefly inspect some of the other variables that I used in the model. Here we can see the distribution of `minimum_nights`, which takes into account the minimum number of nights required for the rental:



I have chosen to subset this graph to any listings equal to or below 15 minimum nights. While there are certainly some rentals even in the 1000+ minimum nights required range, we can see that most require between 1 and 5 nights.

Furthermore, it's interesting to take a look at the distribution of room type, too:



We can see that there is a decent split between homes/apartments and a standalone private room. Shared rooms are significantly in the minority. Overall, this will make for a good statistical comparison as there is not a huge difference between the two large categories.

Overall, the data seem ready to proceed with the feature engineering. The log-transformation did a very nice job of correcting for the majority of the skew and will likely lead to much better predictions.

Splitting data & resampling

I chose to split the log-transformed data in a standard way by creating a training and testing set. The proportion I used was 0.75 since we have around 50,000 observations, which is more than enough to create an adequately sized testing sit even at a much higher proportion. This led to a training set of 36,664 observations and a testing set of 12,220.

As briefly mentioned above, I chose to create strata around the price to further correct for the skewness and get an accurate sampling of the data across price bins. Afterwards I verified that the splitting was successful and that the dimensions were indeed correct.

Next I proceeded with conducting resampling on the training set. To do this, I used repeated V-fold cross-validation. This is a variation of simple cross-validation that creates r repeats of V-fold cross-validation and has $v \times r$ statistics produce the final resampling estimate. The summary statistics from each of the models approximate a normal distribution due to the Central Limit Theorem.

Cross-validation more generally partitions the data into v sets of nearly equal size called folds. To save computing power but still use a very common number of folds, I chose to go with $v = 5$. Generally 5 or 10 folds are sufficient. And since this is repeated V-fold cross validation, using the same reasoning, I used $r = 3$, where r is the number of repeats. Generally 3, 5, or 10 repeats are preferred.

Recipe

This section is arguably one of the most important parts of this entire project: the feature engineering. Here I constructed the recipes to define the model and explicate all required transformation steps. I created two recipes: one for the linear model and one for the machine learning models. The only difference is really in the creation of the dummy variables. For convenience, I have included a copy of the machine learning model here:

```
airbnb_recipe <- recipe(
  price ~  neighbourhood_group + room_type + minimum_nights + number_of_reviews +
    reviews_per_month + calculated_host_listings_count + availability_365, # vars
  data = airbnb_train) %>%
  step_impute_linear(reviews_per_month) %>%  # imputing the (few) missing data
  step_dummy(all_nominal(), one_hot = TRUE) %>%  #  dummy vars  with one-hot encoding
  step_normalize(all_predictors()) # center and scale predictors
```

The recipe first defines the regression model. I have chosen the following regressors for my model: `neighbourhood_group`, `room_type`, `minimum_nights`, `number_of_reviews`, `reviews_per_month`, `calculated_host_listings_count`, and `availability_365`. The definitions for each of these can be found in the codebook. The outcome variable is `price`. Out of all the variables in the dataset, these ones were sensical and ones that I felt could potentially have an impact on the price.

After defining the model, the first step was a linear imputation on `reviews_per_month`. Fortunately this was the only variable used in the model that had any degree of missingness whatsoever, albeit very minimal. Therefore, this was built into the feature engineering in order to avoid throwing out data.

The next step was creating dummy variables from the categorical variables, `neighbourhood_group` and `room_type`. On the linear model, simple dummy variable creation was used with one of the levels being withheld as the reference level. In the machine learning models, displayed above, one-hot encoding was used to create an integer representation for each of the levels.

Finally the normalization step was applied in order to center and uniformly scale all the data. At this point, the recipe was complete and ready to be prepped and baked in the models. The recipes, along with the split dataset and parameters for folding, were saved into an rda file to be used by each model in an R sourced job. I chose not to include a step in my recipe for interaction terms because I do not see any obvious interactions that would occur between the included variables.

Models

Four separate models were defined: linear, random forest, boosted trees, and k-nearest neighbors. Each of these is briefly described below, but I followed a similar process for each. The saved dataset at the end of the last step was loaded. A model was defined with the appropriate mode (regression) and respective engine for the given model. The parameters, if applicable, were defined, and I chose to tune them (e.g., in the boosted trees model, I tuned `mtry`, `min_n`, and `learn_rate`).

A grid was defined for the nonlinear models with the ranges of the parameters in order to try out many different possible combinations. I used 5 levels in the grid to try out. Then a workflow was created to add the model and the recipe. And finally, the tuning took place with the resamples on the grid of parameters. The results were written to a separate rda data file to avoid redundant computational processing.

I would like to briefly describe each of the four models used along with a rationale for their respective parameters:

Linear: this linear model is the simplest model used. This uses ordinary least squares (OLS) to predict the values of a variable using one or more explanatory variables. This attempts to minimize the sum of the squared residuals. There are no parameters on this model.

Random forest: this is an ensemble (multiple algorithms) learning method that constructs many decision trees and returns the average prediction of the individual trees, which are the building blocks of the model. The random forest corrects for overfitting the training set of the decision trees. In my model, I tuned the

parameters `mtry`, which is the number of randomly sampled predictors at each split, and `min_n`, which is the number of data points in a node that is required for the node to be split further. Since I have 8 predictors (granted, more variables due to the dummy one-hot encoding), I chose to use a range of mtry $= [2, 5]$, with 5 as the upper bound so as not to run out of random variables.

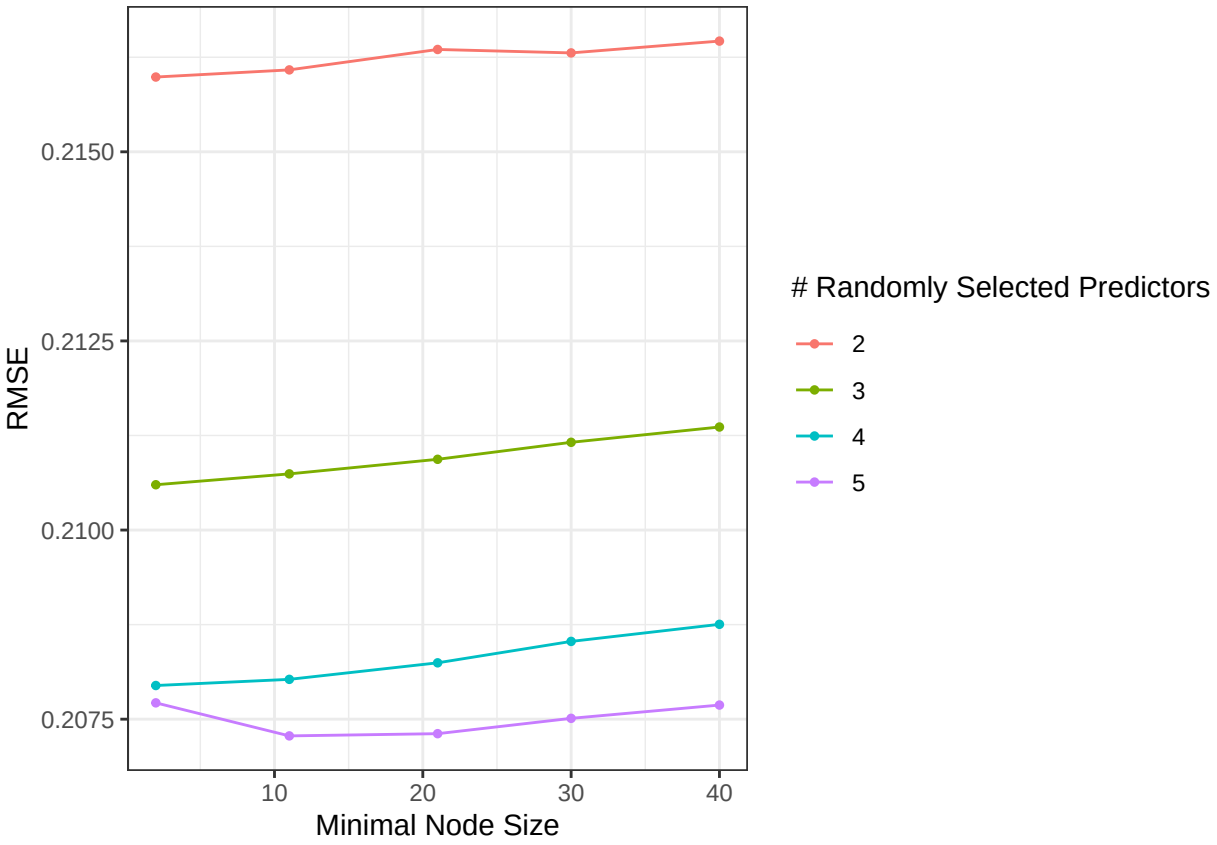
Boosted trees: this model is gradient boosting, which produces a model as an ensemble of weak decision trees. The decision tree becomes the weak learner, and each tree is built one at a time as opposed to the random forest, which independently builds trees irrespective of the others. Thus, this is a forward approach, and the trees learn from their predecessors (results are combined along the way). Here the same range on `mtry` was used, and `mtry` and `min_n` were both tuned. However, gradient boosting can accept a "learning rate," which defines how quickly an error in prediction would be corrected from tree to tree. Because $0 <$ learn rate ≤ 1, I chose a range of $[-5, 0]$ (e.g., $10^0 = 1$). This parameter was also tuned.

K-nearest neighbors (KNN): this is a non-parametric (no assumptions) model where the k closest training examples in the dataset are fed in and the smallest distance values are chosen as the basis for the assignment into a class. The only defined value is the number of neighbors used. I tuned this in my model.

Inspecting tuned info

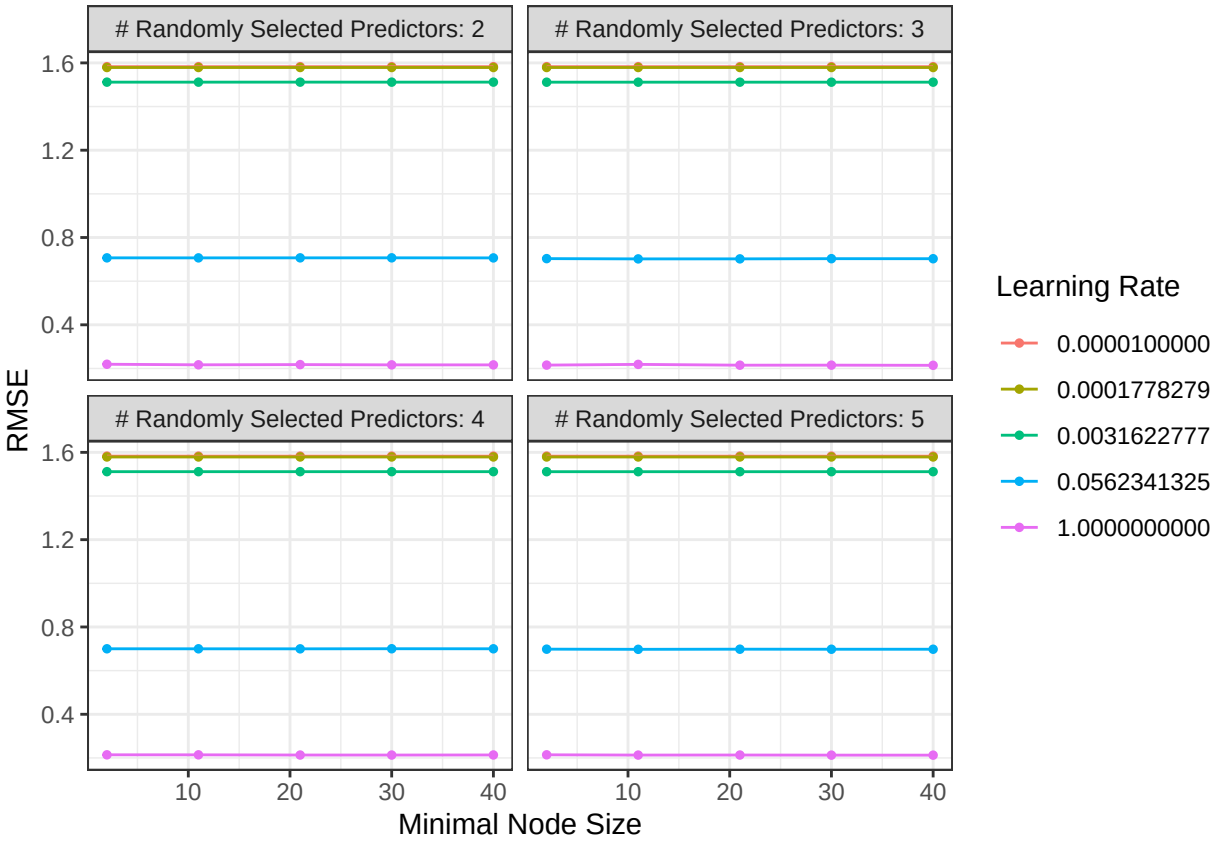
Once the models were complete, I loaded the outputted datasets to extract the tuned data. The first thing I did was inspect some basic tuning plots of the processed data in order to get a sense of how the various parameters performed. In each case, I chose to look to look at the root mean squared error (RMSE). This is the square root of the mean of the square of all of the error, and it is considered a good general-purpose metric to evaluate predictions. RMSE is calculated as: RMSE $= \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$, where $\hat{y}_i$ is the predicted value, y_i is the observed value, and n is the number of observations.

First is the random forest model's tuning plot:



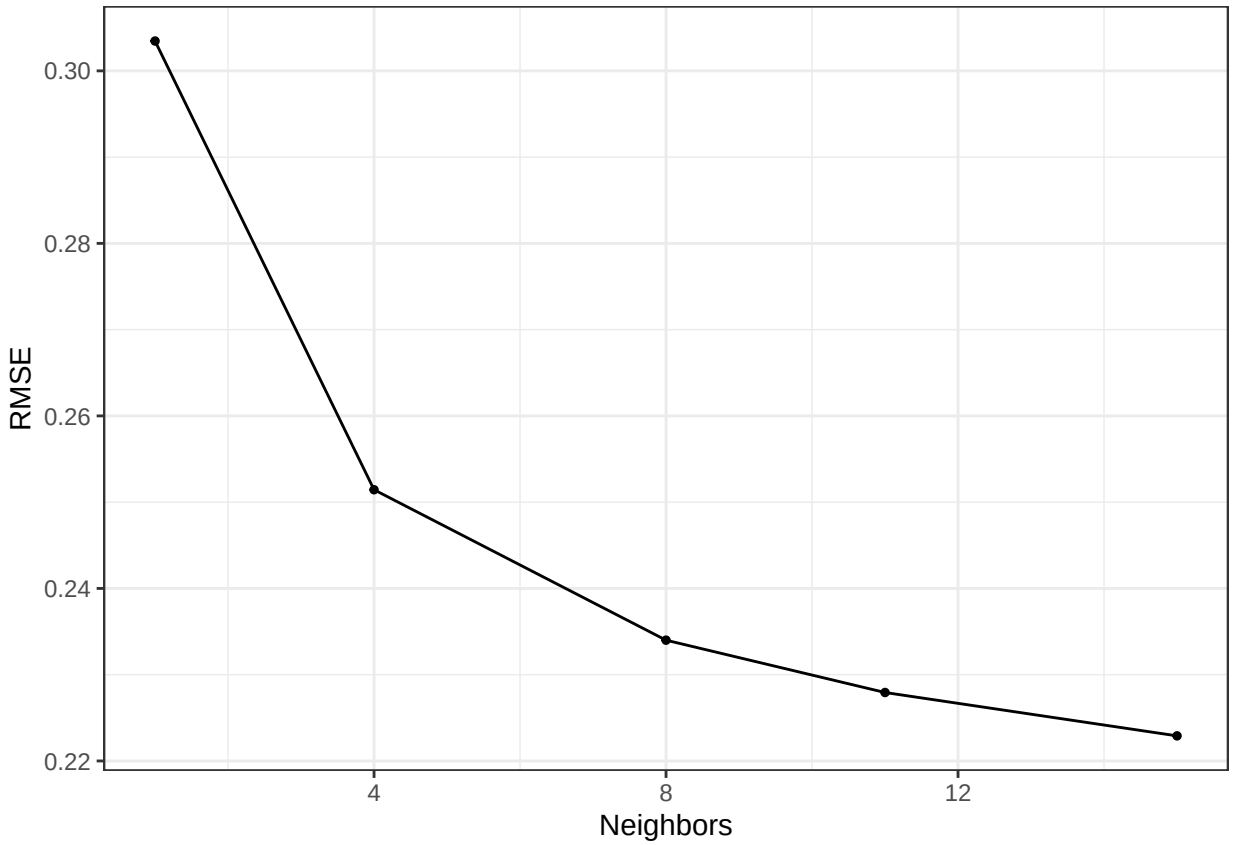
This output is somewhat surprising. Normally we find that at higher levels of random variables, the RMSE decreases with larger values of the minimum number of data points in a node required for splitting. We somewhat see that when there are 5 random variables and the minimal node size is about 12 or so. However, that line does curve back upwards. We would have expected something of the opposite nature, but this is not an indication that the results are completely useless. In fact, as we will see later, this does become important when evaluating the models collectively.

Next we turn to the boosted trees model, which used gradient boosting:



Again, this output is surprising given that it is not the typical relationship one might ordinarily expect to find with a boosted trees model. We see that a higher learning rate consistently performs the best. This is the upper bound (1). The RMSE does not vary by number of random predictors or minimal node size either. The only parameter that matters is the learning rate, suggesting that a larger learning rate is better for training the data. This is a bit counterintuitive since smaller learning rates do not sacrifice accuracy as much as larger ones.

Finally, we look at the KNN model:



The KNN model appears more similar to a typical KNN tuning diagram in that it slopes downward as the number of neighbors increases. It is interesting to note that there is a significant decrease in the RMSE up to 4 neighbors and then the slope becomes flatter in successive increments of 4.

Comparing models

Now it is time to compare the models numerically using the root mean squared error as the preferred basis for comparison. In order to do this, I created a brief table arranged in ascending order of RMSE value. Clearly the one at the top is the best model in terms of RMSE:

model_type	mtry	min_n	.estimator	mean	std_err	neighbors	learn_rate
rf	5	11	standard	0.2072788	0.0010548	NA	NA
rf	5	21	standard	0.2073080	0.0010415	NA	NA
rf	5	30	standard	0.2075107	0.0010352	NA	NA
rf	5	40	standard	0.2076862	0.0010312	NA	NA
rf	5	2	standard	0.2077156	0.0010377	NA	NA
rf	4	2	standard	0.2079448	0.0010413	NA	NA

As we can see, the random forest (`rf`) model with an `mtry` of 5 and a `min_n` of 11 performs the best, with a slightly smaller RMSE than the next random forest model with a `min_n` of 21. The `neighbors` and `learn_rate` columns have a value of `NA` because these values are only applicable to the KNN models. The lower the value of RMSE, the better.

While r-squared is typically not the best value to use when comparing models in this way, it is still worth examining across the various models:

model_type	mtry	min_n	.estimator	mean	std_err	neighbors	learn_rate
rf	5	11	standard	0.5355538	0.0027495	NA	NA
rf	5	21	standard	0.5354797	0.0027260	NA	NA
rf	5	30	standard	0.5345976	0.0027054	NA	NA
rf	5	40	standard	0.5338385	0.0027030	NA	NA
rf	5	2	standard	0.5335632	0.0026669	NA	NA
rf	4	2	standard	0.5329929	0.0026814	NA	NA

The first six rows seem to line up exactly in the same order as seen when comparing by RMSE. With r-squared, a higher value means a higher correlation, so the model with the highest value is the best in this case. An r-squared of 0.536 is actually not terrible. It is not great, but it is suggestive of moderate correlation. This certainly refutes the idea that there is no correlation between the predictors and price whatsoever. I should also note that the linear fit was not part of either of these tables since it had no parameters to tune and had a different recipe. However, its RMSE was 0.218 and its r-squared was 0.474, so it was by no means the best model, which is unsurprising.

Fitting on entire training set

Now that we have chosen our winning model (random forest with `mtry` of 5 `min_n` of 11), we are ready to finalize the workflow and fit the data on the entire training set. Unfortunately the output is somewhat messy, but the highlights can be summarized as follows: the RMSE is 0.2066715 and the r-squared is 0.538. This is fantastic because the entire training set has a lower RMSE and higher r-squared than any of the cross-validated v-folds.

Fitting on testing set

Our last step is to finally apply the winning model to the testing set, which the model have not yet seen. This will be a good indication of whether our winning random forest model severely overfit the training data or if it can easily adapt to new data. To do this, we can simply use a predict function to run the model with the testing data. We get the following results: an RMSE of 0.2043358 and an r-squared of 0.538. Again, this is really great. The RMSE is significantly lower than that of the entire training set, and the r-squared is still just as good as before. This means that the model, while it may still have overfit in some ways, did not do a terrible job of running itself on new data. In fact, it appears to have done better. This performance is not great relative to what the RMSE could be (even lower in theory), but it's also not all that bad. With the logged data, 0.204 is fairly decent.

Conclusion & next steps

We can definitively say that the random forest model with `mtry` of 5 and `min_n` of 11 appears to be our winning model! This is great because it means the tuning parameters were well-chosen by the machine. Furthermore, we actually ended up having better results on our testing dataset than our aggregate or cross-validated training sets.

It is very interesting to see that the random forest model seemingly outperformed the boosted trees model. In fact, the boosted trees model took significantly less time to run than the random forest model, which is also unusual. It is possible that the random forest model might do better than the boosted trees model due to noise in the data.

It is important to realize that this model is by no means close to perfect. While the log-transformation done initially was certainly helpful, it did not take away all the skewness or outliers, so this certainly contributes to no model being perfect.

Going forward, there are a number of next steps that I would take. In terms of the data, I think that it would be interesting to gather observations from places outside of New York City. Perhaps there are some strange

features of pricing or rentals in general in NYC that create noise. Chicago would be a great location to try this.

I would also like to try some more models. Elastic net models penalize a linear regression model. While linear regression was not close to the best model in this project, penalties would likely make it better. How much better cannot be determined without actually doing it, but this would be interesting.

Additionally, it would be worthwhile with more time to run more tuning parameters (wider ranges than I included, for example, in my random forest and boosted trees with `mtry`). I would also like to modify some of the feature engineering. I specifically would want to put some interactions (perhaps all interactions) in the linear regression model.

Overall, I am pleased with the outcome of this project. The results are somewhat puzzling, especially in the tuning plots for the random forest and boosted trees models, where the RMSE curve slopes up as the minimal node size increases. That being said, the random forest did perform the best and had a decent RMSE and r-squared, so this was a good sign.