Using machine learning to calculate the daily rental price of Airbnb listings

CS 221 Fall 2016
Rodrigo Grabowsky, Abiel Gutiérrez, Gerardo Rendón

1. Task Definition

1.1 Introduction

Our system uses machine learning to calculate the daily rental price of Airbnb listings in different cities around the world. The input-output functionality is simple: our system allows the user to input characteristics about a residence, such as the number of bedrooms and bathrooms it has and the number of guests it accommodates, and it outputs the predicted rental price of that residence on a given day. This day is already set and varies by city -- it corresponds to the date in which the data set for that particular city was scraped from the Airbnb website. For example, for Amsterdam, this date is July 4, 2016; for New York City, it is December 6, 2016. The complete list of features that were used to predict the price of residences will be discussed later in the report.

1.2 Motivation

The main motivation behind our project was to take advantage of the many practical applications that this tool has. One of these applications is helping Airbnb expand into locations where it isn't yet well established, and we believe this could be done by recommending daily rental prices to hosts in these locations that would enable them to profit the most from their listings.

Recall that the given date for rental predictions in each city is already set. If we had more time to work on our system, we would work to factor in the rental date as a variable input feature. In order to achieve this, the first step would be to set up the infrastructure to scrape the Airbnb website ourselves, create a data set for every day of the year, and train the model using date as a feature as well. The second step would be a little more complex: in order to use the pricing model on cities that do not have enough training data, we could find a way to map neighborhoods of new locations to similar, corresponding neighborhoods in existing locations for which Airbnb already has many listings. There are perhaps other more efficient ways of accomplishing this task that we haven't fully evaluated yet; these ideas are simply what fueled us to get going in the first place. The system we have developed now could be thought of as the essential building block for this more developed system that we strive to get to.

1.3 Initial development and evaluation

Initially, we built our model for New York City because we had access to a very comprehensive data set consisting of over 40,000 data points. After refining our system there, we tested it in Los Angeles, Berlin, and Amsterdam. We continued refining our system based on performance we achieved in the validation set for all those four cities. Finally, we tested on two extra cities, Paris and Antwerp, to verify we that we could achieve a similar performance in places where the system was not specifically refined.

We evaluated the performance of the system by comparing our results to those of the baseline. Our baseline is the average daily rental price of Airbnb listings in the training data for a particular city. To evaluate performance, we first calculated the average absolute prediction error of our baseline against the actual prices, and then calculated the average absolute prediction error of our predicted prices against the actual ones. We then computed a percentage

decrease in prediction error from the baseline to our approach in order to be able to fairly compare the system's performance across cities where demographics differ greatly. A more comprehensive discussion of our baselines and oracles, and our performance metrics analysis, will be given later in the report.

2. Infrastructure

We got our data set from the website *Inside Airbnb*; it is an independent, non-commercial site with data that allows individuals to explore how Airbnb listings are being used to compete with the residential housing market [1]. The main motivation behind the website is to publish data that can feed the debate surrounding the impact of Airbnb on housing prices in urban areas. Lately, many tourists have preferred Airbnb residences over hotels, consequently reducing the availability of Airbnb properties for long term rental and thus increasing the prices of housing in general -- either because of a shortage of supply or because prices are raised to match the earnings that could be achieved with Airbnb.

In our data set, each data point represents an Airbnb listing, and each listing has 68 entries of information -- most of this information is about the listing itself (such as number of bedrooms, bathrooms, and the like), but some of it is metadata about the web scrape, and some of it is irrelevant for our price prediction (i.e. host account number). The data attributes that we found most useful and ended up using as features in our machine learning model were:

- Number of people the listing accommodates
- Number of bathrooms
- Number of bedrooms
- Number of beds
- Number of guests included in the price
- Score for the listing's reviews
- Property type (Apartment, Loft, House, Dorm, etc.)
- Room type (Entire home/apt, private room, shared room)
- Neighbourhood
- ZIP code
- Number of amenities included
- Borough/neighborhood group
- Amount of security deposit
- Cleaning fee

Our python script uses a command line argument that allows the user to choose the city or data set they want to use for learning and prediction. The cities we support are all the ones whose data sets are included in the data subdirectory in our project's work folder: Antwerp, Paris, New York City, Los Angeles, and Amsterdam.

After learning the linear model parameters used to predict prices for one city and printing out the evaluation metrics for that city's predictions, our python script also allows users to choose whether to test/use our model by manually inputting features. That is, users may choose to input the features manually (number of bathrooms, bedrooms, etc.) and receive a daily rental price prediction if a listing would meet those exact features.

The results of a sample prediction using the script's manual input feature have been included below. The results shown are from a model that was trained in New York City. These are sample input features, and they do not correspond to a real apartment in New York City:

```
> Want to enter an example listing and test our predictor for this city? [Type yes or
no]: yes
> Enter the number of people the listing accommodates: 8
> Enter the number of bathrooms: 4
> Enter the number of bedrooms: 4
> Enter the number of beds: 4
```

```
> Enter the number of guests included in the price: 4
> Enter the property type (e.x.: Apartment, Loft, House, Dorm, etc.): Apartment
> Enter the room type (1 for Entire home/apt, 2 for Private room or 3 for Shared
room): 1
> Enter the neighbourhood: Upper West Side
> Enter the zipcode: 10023
> Enter the borough/neighborhood group: Manhattan
> Enter the security deposit amount: 500
> Enter the cleaning fee: 200
> Enter the score for the listing's reviews: 93
> Suggested daily price for this listing: 517.38
```

For all the terminal lines included above (except the last), the text to the right of the colon is user input.

The website *Inside Airbnb* has data sets that include all Airbnb listings for 16 cities in the U.S., 5 in Canada, 18 in Europe, and 4 in Asia Pacific. We specifically used the data sets from New York, Los Angeles, Berlin, and Amsterdam to refine our model, and they have 39553, 26080, 13849, and 15373 data points, respectively.

Given that we improved our model on cities with very disparaging demographics, we think that any other city listed in *Inside Airbnb* can work well with our system. In order to test whether this was true, we downloaded the data sets for Paris and Antwerp: two cities with a drastic difference in their number of listings. Antwerp only has 747 listings and Paris has 52,725. The performance of our prediction system in each city was reasonably similar to the performance of the system in cities whose validation data we used to optimize the model. For example, in Antwerp, the improvement over the baseline approach was 28.94% and the percentage prediction error was 30.58%, and in Paris, the improvement over the baseline was 39.28% and the percentage error was 30.11%. These results are comparable to New York City, where the improvement over the baseline was 34.45% and the percentage error was 37.0%. More of this will be on “Error Analysis”.

3. Approach

3.1 Baseline and Oracle

Our baseline prediction for any listing of a particular city was the average of the prices of all listings within the data set of said city. Our oracle was the actual price. We think that the gap between the two was perfect for the project. We were able to slightly beat the baseline with the first model we built, and after multiple tweaks and optimizations, we achieved an improvement of about 35% over the baseline. We are satisfied with the improvement given the scope of this class, but we think there’s still a lot of room for improvement using more advanced machine learning techniques.

3.2 Model

Our model for predicting and recommending listing prices was a weighted linear regression model, in which the estimated price was calculated as the dot product of learned weights and features of Airbnb listings. For example, for n input features, we could predict the listing price y with a formula of the type:

$$y(w, x) = w_0 + w_1 x_1 + \ldots + w_n x_n$$

The symbol y above is an estimate of the true value and not the true value itself, whilst w is the weight vector and x is the feature vector. We first used the LinearRegression class in the SciKit Learn python library, under their linear_model module, to learn the weight vector for the equation above. This class tries to fit a linear model with

coefficients given by the weight vector by minimizing the squared error over all training examples. Hence, using this class helps us solve the mathematical problem: $min_w ||x \cdot w - y||_2^2$. That is why the documentation in SciKit Learn refers to this method as Ordinary Least Squares regression.

Although we tested with different modalities in our features (value vs indicator), the best results were obtained when we made all features be indicators. The features that had the highest range in possible value, namely security deposit and cleaning fee, were discretized into buckets, given that their values varied anywhere from $0 to more than $1000. This helped make the weights more significant. The Error Analysis section discusses this further.

As we looked through the prediction results of individual listings in our first iteration, we noticed that several of them had a negative price prediction. This led to us to believe that we were overfitting the data, perhaps due to the inclusion of too many features. So we tested our model with SciKit's Lasso, a linear regression with L1 regularization. The regularization consists of adding $\alpha \sum |w_i|$ as a penalty to the loss minimization function, thus causing that features with very little impact have a weight of almost zero. In order to select the best value of α possible, we created a hyperparameter tuning python script that ran our model on alpha values ranging from 0.05 to 0.95 with a 0.05 step increase. Our results showed that the initial value of 0.05 was optimal. Lasso improved our errors by 1 or 2 dollars depending on our data set, and made us feel more secure about adding features to our model, given that the added penalization to the error minimization function resulted in an implied form of feature selection that would disregard unnecessary additions. Despite this advantage, Lasso doesn't improve issues stemming from multicollinearity as much as are other classes in the SciKit library do (i.e. Ridge regression), but it was nevertheless the one that produced the most optimal results for our model.

3.3 Challenges

Indicator Features

We faced multiple challenges when building our system. One of them was adding features like ZIP codes to our feature extractor, for they had to be factored in as indicator variables. ZIP code values had to be treated differently because they don't measure value by themselves -- they are just a set of numbers that correspond to a designated geographic area. Thus, we couldn't treat them in the same way as we treated other features such as number of bathrooms, number of bedrooms, number of guest accommodations, and the like, where the magnitude of the values was correlated with the rental price. We referred to these features (bathrooms, bedrooms, etc.) as "value features".

We used a function in the SciKit library called DictVectorizer that facilitated converting distinct ZIP code values into indicator features. All we had to do was pass in the results from our feature extractor as a vector of dictionaries, and the library created a dictionary containing an indicator feature for every possible value of the different features that we passed in. After all, introducing ZIP codes was a worthwhile change, for all the variables in our model ended up being converted to indicator variables later in our development, and this actually improved our results.

Outlier values

Another challenge that we faced were outlier values -- that is, values that appeared significantly above the average rental price of residences in a given city. For example, in New York, a few number of residences were priced at around $10,000 and above. It was hard to predict these outlier values, especially because the set of listings for New York City became dramatically sparse after $1,000; in other words, the number listings in the price range above $1,000 was significantly smaller than the number of listings in the price range below $1,000. This is intuitive, but because of this sparsity, our predictions were significantly divergent from these outlier housing prices at $10,000 and above. This is because we had few samples above $1,000 that allowed us to train our algorithm to make predictions that neared the magnitudes of the $10,000 price listings.

Multicollinearity

Multicollinearity problems occur when two variables in a linear regression are correlated with each other. In our case, we believe that there were multiple features that were correlated with one another. For instance, there is a high chance that the number of beds and the number of bathrooms in residences were correlated: as one goes up, the other goes up as well. The more bedrooms there are, the more people a house will probably accommodate, and the more bathrooms the residence will need. When this is the case, then data interpretation becomes harder because small changes in the data can lead to large changes in the model [2]. SciKit's Ridge regression does L2 normalization to account for this, but we found that Lasso regression had a better impact on our results; apparently feature selection was a greater problem in our model than multicollinearity.

Choosing the features

Choosing the weighted features was a challenge, not only due to potential issues with multicollinearity, but also because it was difficult narrowing down to the most essential ones that avoided issues with overfitting or underfitting. In order to choose the features, we first conducted some research to see what were the factors that most strongly influence housing prices. Given this information, we narrowed down to our final choices through trial and error; we adjusted features, changed some, and added others based on the results we got, and we conducted multiple iterations of this process until we converged on the features that minimized the error on our four validation sets.

4. Literature Review

4.1 Everbooked

Everbooked is a startup in Oakland that was founded in 2014 and whose goal is to enable Airbnb hosts to increase their revenue by "giving them the same tools that hotels and airlines have had for years" [3]. They have an automatic nightly pricing feature that looks at local demand factors to set the user's property to the price that will maximize revenue, which they claim that they have increased by 40% in some cases.

The technical details of their product are not disclosed, but there are some significant differences between their approach and ours:

- Their adjustments are done at an hourly rate, while we use data from a specific date that does not change and is not updated as time passes by. This allows Everbooked to gather enough data to make accurate price predictions for any day of the year, making their system more useful.
- The data that Everbooked looks at goes beyond the Airbnb environment to include hotel rates and event details from around the area. This allows their users to compete against all types of residential rentals. Our approach only looks at Airbnb data.

*There are quite a few other companies that do exactly this, among them *Beyond Pricing* and *Price Labs**

4.2 Airbnb's Dynamic Pricing

Airbnb has built a dynamic pricing feature that shows the hosts the probability of getting a property booked at a specific date with a specific price. The feature was built using their open source machine learning library called Aerosolve [4]. Their approach is different from ours in the following ways:

- Aerosolve allows them to input prior beliefs of feature information into their model, in order to avoid a "cold start" in which weights or coefficient variables are initialized to zero or random values.

- Rather than relying on textbook-defined neighborhoods (i.e. Soho, Lower East Side, Harlem, etc.), their algorithms automatically create them in several layers. With this, they can construct a tree and use information of parent nodes (larger enclosing neighborhoods) to construct more specific features for child neighborhoods. This creates different features in different demographical areas -- a bedroom in Manhattan is likely to be very different from a bedroom in Brooklyn.
- The model uses image analysis to factor in the listing's pictures into the price calculation (the nicer the picture looks, the higher the price). The training for this was done with two sets of scores on the interior design of the images: a ranking by professional photographers, and a ranking based on number of bookings.
- Additional information beyond the description of the listing is also used, like upcoming events in the area.

5. Error Analysis

5.1 Simple Error Metrics (printed terminal output)

We set up our regression python script to output multiple results, such as average absolute and percentage error. For example, if you run "`python regression.py nyc`" in the terminal, with the final version of our code, you will see the following output:

```
City: New York City
Datasize: 40227
Learning data size: 25746, validation data size: 6436, test data size: 8045
Average error: $55.43
Baseline error: $86.84
Improvement over baseline: 36.16%
Percentage error (avg. err / avg. price): 37.92%
Number of input features used: 629
```

The reason we printed all this information after every run of the machine learning algorithm is twofold: i) to sanity check our results, for instance, by verifying that the number of inputs we are using matches what we expected and that our input file for the data set was not corrupted by checking the size of the data; ii) to analyse simple error metrics such as improvement over baseline and percentage error and see the incremental improvements that each change to our code caused in the results. This approach allowed us to quickly iterate on changes we made to the model.

As mentioned under "Challenges", we started by only using what we called "value features": attributes of the data for one listing which are numerical in nature, such as the security deposit amount or the number of guests included. These features were the most simple to implement since you can simply read them off the data set and include a key-value mapping such as "numberOfGuestsIncluded : 4" in the feature dictionary representing one datapoint. By simply using these features and the simplest linear model we could find (Ordinary Least Squares regression) our improvement over the baseline was already somewhat significant for New York City:

```
Average error: $69.41
Baseline error: $83.08
Improvement over baseline: 16.45%
Percentage error (avg. err / avg. price): 47.48%
```

5.2 Errors with Indicator Features

We added indicator features such as: “zipCodeIs_10023 : True”, corresponding to a listing that has zip code 10023. Other features of this type are property type, room type and neighborhood. With the final list of features described above, the number of features used for New York City (this number changes for every city because every city has a different number of unique neighborhoods, zip code, etc.) was 824. Depending on the actual combination of features we used our errors actually became much, much worse. For example, for New York City:

```
Average error: $33850645823.0
Baseline error: $83.08
Improvement over baseline: 40746068372.9%
Percentage error (avg. err / avg. price): 23157447526.5%
```

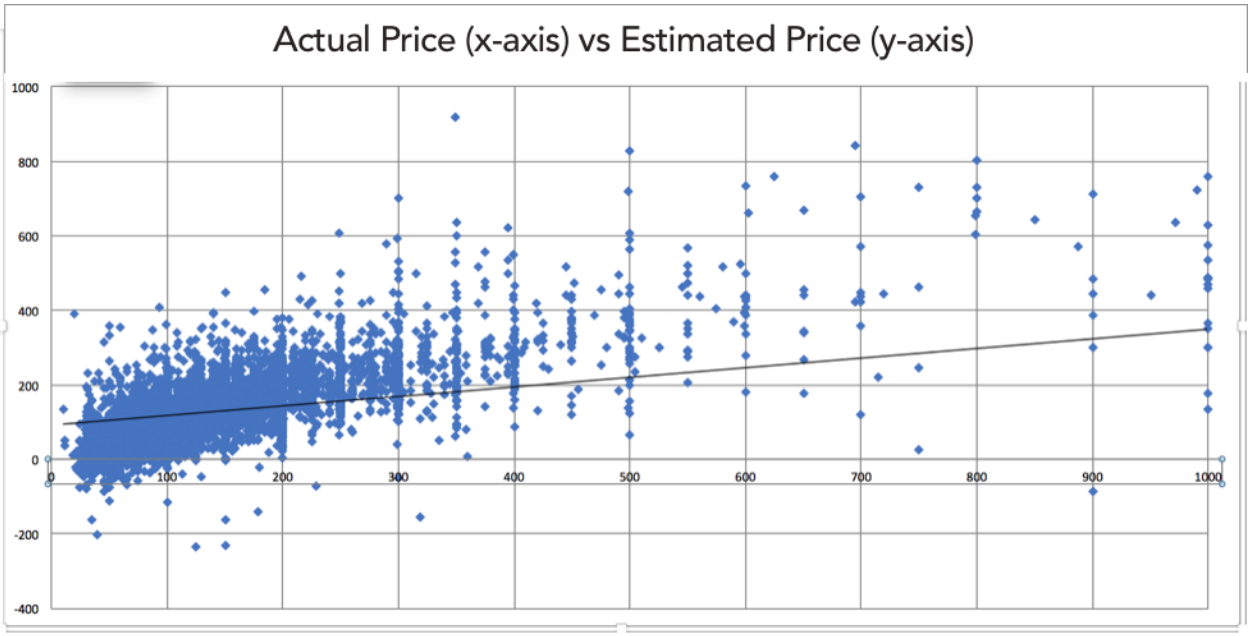
We had to stop and do some research to understand how adding more features, which theoretically add more useful information to our model, could actually make our predictions fail horribly like that. We found that a lot of features were deeply correlated (such as number of beds and number of bedrooms), meaning that simple linear regression may actually learn the weights of these features in a way that produces skewed results; that is, even though these weights could make the model produce small errors in the training data, the model could still be prone to large errors with small variations in the validation set. This is the “Multicollinearity” issue referred to in Challenges, but to be specific, we will provide an example: if we only had two inputs *X1* and *X2*, where $X1 \approx X2$, and the value we are trying to estimate is in reality very close to $Y = X1 + X2$ [2], then because of noise in the training data, the model could end up learning the weights +2 and -1 for the input variables respectively and not their true weights +1 and +1. Under these circumstances, the model could still be able to minimize the objective function, but the predictions for Y could be way off given a significantly different data set for validation.

We found two different linear models, Ridge and Lasso, which helped us solve the above problem to a certain extent. Ridge coefficients minimize a penalized residual sum of squares. There is an extra term, $\alpha||w||^2_2$ in the objective function, where α is a hyperparameter in the model. Similarly, and as mentioned before in “Approach”, Lasso also adds a penalty, $\alpha||w|_1$, where $||w|_1$ is the l_1 - norm of the weight vector. Lasso also uses a coordinate descent algorithm to fit the coefficients. This was a useful algorithm for us, due to its tendency to prefer solutions with fewer parameter values by recognizing features that have little effect on the output and setting their weight to 0. Lasso effectively reduces the number of input variables upon which our prediction function is dependent, which reduces the variability of our estimated prices and thus, reduces our average error. Both Lasso and Ridge gave similar error levels, and we included the output for Lasso, which was slightly better, below:

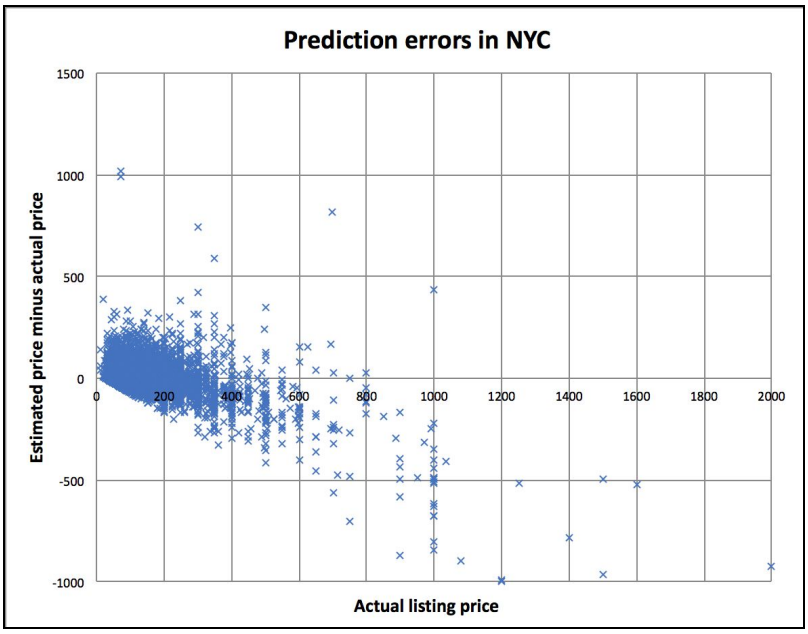
```
Average error: $56.83
Baseline error: $86.84
Improvement over baseline: 34.56%
Percentage error (avg. err / avg. price): 38.88%
```

5.3 Plotting error charts

At this point, the gains we were achieving in our predictor’s performance (in terms of decreasing error) per time spent optimizing the model were diminishing. We therefore spent some time figuring out a way to analyze our prediction errors in more detail. We included code in the python script to create an output CSV file where every row contained the actual price for a listing in the validation set and the price predicted by our model. By doing so, we could create charts in Excel to plot our predictions against the real values and see where we were miscalculating the most. One of the first charts we created for New York City is the following:



From the chart above, we quickly noticed one quick improvement we could achieve in our model. We noticed that for a non trivial number of listings we were somehow predicting negative prices. In terms of the real world situation we were trying to model, nonpositive numbers make no sense as an output, so we knew we had to restrict the range of outputs of our predictor function. Additionally, a price of just a few cents (just above 0) for an Airbnb listing is also extremely unlikely -- thus, we decided that the lower bound would be the first percentile of daily rental prices in the training data. This improved our absolute average error in NYC by about $2. We also experimented with an upper bound (on the 99th percentile) and that actually made our errors slightly worse. We realized though that this bound made no sense for the model because if a listing in the test set happened to have features that added a lot of value to the listing (for example, if the listing can fit a very large number of people), we would want to be able to predict a very high daily rental for said listing.



After this, we plotted errors against actual price to see if we were performing very differently for different price ranges. As we suspected, we tended to underestimate the prices of very expensive listings. Moreover, we saw that for a fair number of listings, our errors were on the order of a few hundreds of dollars. Note that for both graphs above, we capped the x-axis to $1000 and $2000 respectively. In reality, there are some listings in New York City whose daily prices go up to $10,000. But the data becomes much sparser in that region and capping the x-axis allowed us to analyze this denser part of the chart in more detail.

5.4 Final improvements: Transforming 'Value Features' into 'Indicator Features'

We received feedback from course staff in the poster session suggesting a further step we could take to reduce variability in our predictions and hence decrease our errors. The feedback was to transform all input features, even our so-called 'value features', into indicator variables using feature templates of the type "numberOfGuestsListingAccommodates_[integer]" (where we substituted the integer part with an actual number from the 'accommodates' column in our data set). For features whose domain was a subset of the real numbers, such as security deposit (for which the number 205.05 is a valid, albeit, uncommon value) we discretized those values splitting them into ranges and assigning increasing integer values to each subsequent range. For example, bucket 1 contained values between US$0 and US$50. Before coming up with final values for widths of buckets for each of the features we wanted to discretize (security deposit, cleaning fee, and review scores), we used a python library called Plotly [6] to create histograms to show us the distribution of the values of these features. The widths we decided upon for the features mentioned above were $50, $30 and $5, respectively. We tested with a few other widths that made intuitive sense given the range of values for these features and found that the ones we ended up with were a great balance between generalization and overfitting.

5.5 Final results

At this point it is also worth pointing out that, even though we have only been mentioning New York City in this "Error Analysis" section for conciseness, we worked with three other Airbnb cities in parallel (Amsterdam, Los Angeles, and Berlin), analyzing errors on those ones as well (from subsection 5.2 onwards). Compared to NYC, these cities had differing data sets in terms of total data size and variance of listings' daily rental price (this caused the initial baseline error for LA to be much larger than that for Berlin, for example). Hence, we believed that together they provided an appropriate size and variety of data to prevent overfitting our model to one city or a particular type of city. Please refer to appendix A for the results that we achieved with the incremental improvements referenced in subsection 5.4; the average prediction error in NYC, for instance, went down to roughly $49. However, we still had to finally test our model against our test set -- the subset of our data which we hadn't even seen up to this moment. For those purposes we added another two cities, Antwerp and Paris. Antwerp particularly was a good testing scenario because it only had a total of 747 listings, which gave us significantly less training data (478 data points), and we wanted to see how our system fared under these circumstances. Thankfully, the percentage error for Antwerp (30.58%) was very similar to that of the other cities we used. Please refer to Appendix B for our final results for all six cities.

Appendix A

Final Results for Validation Data Set	
City: New York City Average error: $49.59 Baseline error: $83.58 Improvement over baseline: 40.68% Percentage error (avg. err / avg. price): 33.49%	**City: Los Angeles** Average error: $75.58 Baseline error: $129.05 Improvement over baseline: 41.43% Percentage error (avg. err / avg. price): 41.08%
City: Amsterdam Average error: $32.05 Baseline error: $49.96 Improvement over baseline: 35.85% Percentage error (avg. err / avg. price): 24.1%	**City: Berlin** Average error: $16.84 Baseline error: $25.04 Improvement over baseline: 32.73% Percentage error (avg. err / avg. price): 28.13%

Appendix B

Final Results for Test Data Set

City: Antwerp Datasize: 747 Learning data size: 478, validation data size: 120, test data size: 149 Average error: $23.4 Baseline error: $32.93 Improvement over baseline: 28.94% Percentage error (avg. err / avg. price): 30.58% Number of input features used: 154	**City: Paris** Datasize: 52725 Learning data size: 33744, validation data size: 8436, test data size: 10545 Average error: $29.05 Baseline error: $47.83 Improvement over baseline: 39.28% Percentage error (avg. err / avg. price): 30.11% Number of input features used: 312
City: New York City Datasize: 40227 Learning data size: 25746, validation data size: 6436, test data size: 8045 Average error: $55.43 Baseline error: $86.84 Improvement over baseline: 36.16% Percentage error (avg. err / avg. price): 37.92% Number of input features used: 629	**City: Los Angeles** Datasize: 26080 Learning data size: 16691, validation data size: 4173, test data size: 5216 Average error: $78.79 Baseline error: $130.56 Improvement over baseline: 39.65% Percentage error (avg. err / avg. price): 42.83% Number of input features used: 773
City: Berlin Datasize: 15373	**City: Amsterdam** Datasize: 13849

Learning data size: 9838, validation data size: 2460, test data size: 3075 Average error: $16.64 Baseline error: $25.64 Improvement over baseline: 35.12% Percentage error (avg. err / avg. price): 27.78% Number of input features used: 491	Learning data size: 8863, validation data size: 2216, test data size: 2770 Average error: $33.78 Baseline error: $51.46 Improvement over baseline: 34.37% Percentage error (avg. err / avg. price): 25.4% Number of input features used: 3893

Sources

[1] Inside Airbnb - http://insideairbnb.com/get-the-data.html
[2] “Diving into Data” - http://blog.datadive.net/selecting-good-features-part-ii-linear-models-and-regularization/
[3] Everbooked - https://www.everbooked.com/howitworks
[4] Airbnb Aerosolve - http://nerds.airbnb.com/aerosolve/
[5] SciKit - http://scikit-learn.org/stable/modules/linear_model.html
[6] Plotly Python library - https://plot.ly/