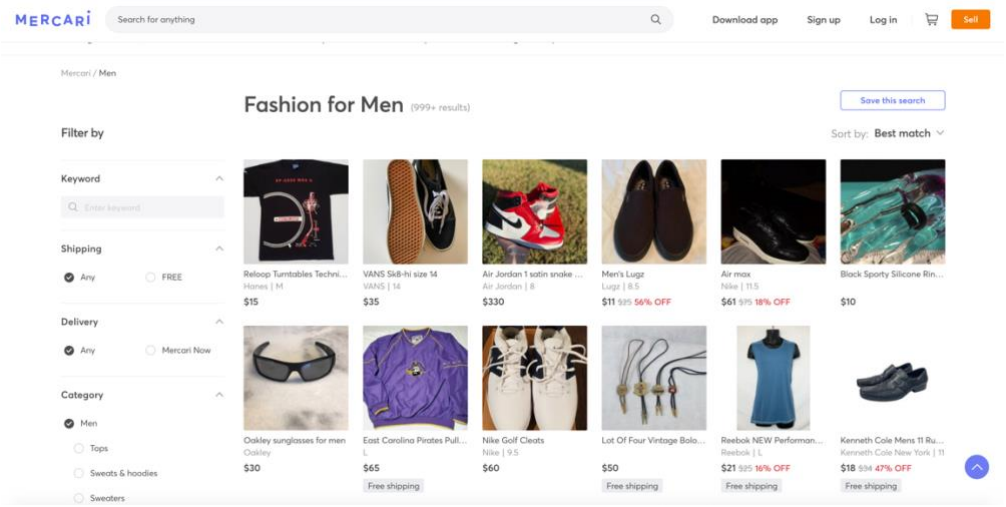
Final Report:

Predicting the Prices of Item Listings on Mercari

Jason Zhou

Problem Statement

Mercari is an online shopping website where sellers are able to freely set their own prices on products. Unlike for example Amazon, sellers on Mercari are all individuals rather than companies, who already know what prices to place on their products.



Therefore, determining prices is trickier for these individual sellers on Mercari. Not only do they need to take into account the branding and the quality of their item, but also its current condition. Our goal is to build regression models that will be able to predict the price of a new item listing, given the item's other characteristics. For example, based on the name, category, branding, and condition of an item, our model would be able to accurately predict how much this item should be charged for. This will allow sellers to set more accurate prices.

Working Data Set

Our data set is simply a large data set of item listings on Mercari, totaling up to about one million rows. It has seven columns, as described here:

1. Name: name of the listing
2. Condition: the condition that the item is in. Condition is rated on a scale from 1 to 5, with 1 being the highest rating and 5 being the lowest
3. Category: the category that the item falls under. Examples include electronics, fashion, household, with even more specific subcategories and so on
4. Brand: the brand of the item, if applicable. Examples include Nike, Sephora, lululemon, Apple, etc.
5. Price: the amount of payment that the buyer pays to the seller
6. Shipping: whether or not the seller is paying for shipping. The value of "shipping" is "0" if the seller is not paying for shipping fees, and is "1" if they are
7. Item Description: a short summary of the item, if applicable

Computational Constraints

Before we get into the main steps of the data science process, I must address some computational constraints I was facing while working on this project, as they dictated many of the decisions along the way. While I was fitting models to my training data, and tuning hyperparameters with cross validation, I noticed that the code executions of these steps were taking far too long, spanning several hours. Just for reference, after I'd one-hot encoded my categorical features, my resulting DataFrame ended up being 10 GB in file size after being exported as a .csv file. Even just reading in this file in my modeling step of the process took about 5 whole minutes, which served as a benchmark as to how long I could expect the more complicated code executions to run for. As a result, I had to make the decision to slice down

our original data set to a more workable size whenever I could, while still retaining a representative distribution. This ultimately will sacrifice performance of the final model I end up going with, but it was a tradeoff that had to be made in order to properly create and test different models at all. Keeping all this in mind, we go into each step of the data science method looking for ways we can eliminate any low quality or unhelpful data.

Data Cleaning

For starters, let's see how many null values are present in the data set:

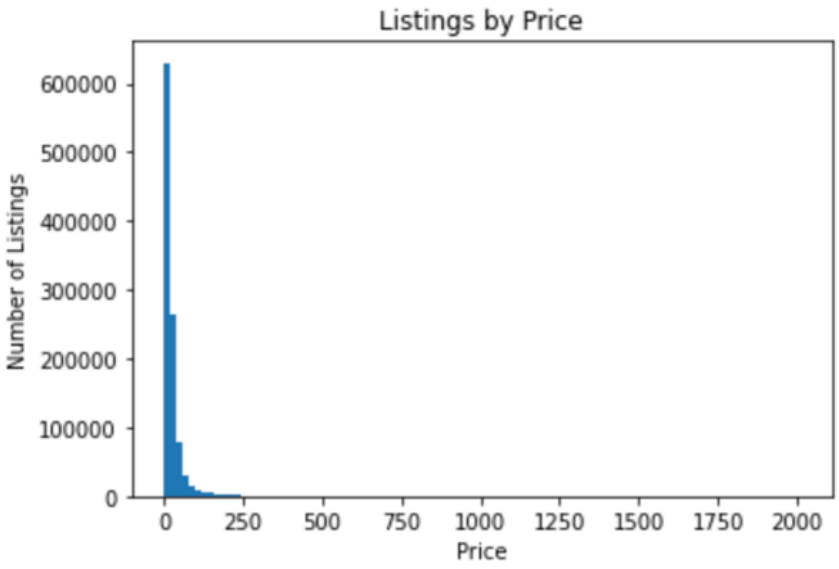
Out[18]:

	count	%
Brand	447335	42.66123
Category	4503	0.42944
Item Description	2	0.00019
Name	0	0.00000
Condition	0	0.00000
Price	0	0.00000
Shipping	0	0.00000

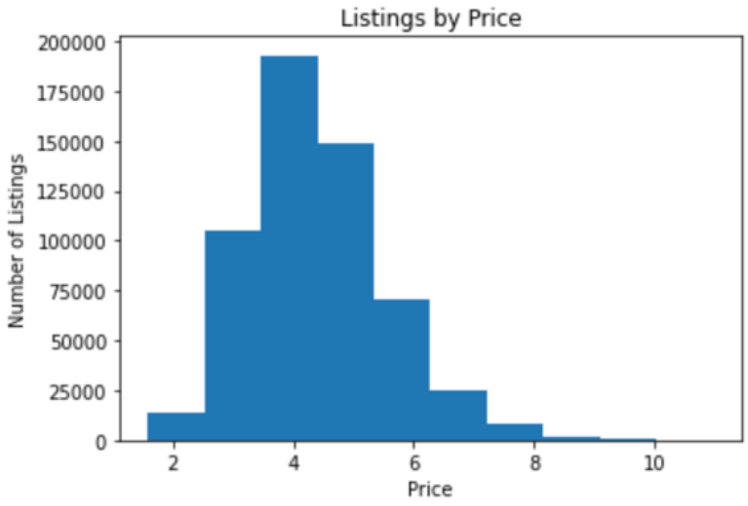
Out of 1,048,575 rows initially in total, dropping rows with missing values would filter our data set down to about a half, and so I did just that. After that, I then dropped rows of data where its Brand or Category occurred less than 50 times throughout the whole data set. For example, if there are only 30 listings that are under the "Patagonia" brand, then the corresponding rows of data would no longer be a part of the data set that I would go forward with. The main motivation of doing this was to have less additional columns generated, after the one-hot encoding that I would have to do eventually.

I then decided that the “Name” and “Item Description” columns would not be useful to include in the analysis. Out of one million listings, there were roughly 900,000 unique values in both the “Name” column and the “Item Description” column. If I were to one-hot encode my data set with these columns in the data, I would end up about a 1 million x 2 million size data set, which I doubt my entire computer would even be able to store. And because a vast majority of values in “Name” and “Item Description” are unique, I did not think that these columns would be useful as categorical features. In the end I decided to not include these two columns as they would not be worth the trouble.

While looking at a distribution of the Price column, I noticed that the data was heavily skewed.



The mean price of the entire data set is around $26, and yet the x-axis spans all the way to the $2000 mark. The majority of data points, across the entire range of price values, are mostly encapsulated on a tiny sliver on the left. This non-normal distribution would not satisfy the statistical assumptions required for analysis later on. I decided to take the log (base2 specifically) of all the Price values, and it greatly improved the distribution.



Before I took the log, I dropped every row that had a value of 0 in its Price column. Not only because you can't take the log of 0, but also because it's not possible for item listings to have a price of 0 on Mercari anyways. These values were most likely mis-entered in the first place.

My last touch was making adjustments to the Condition column. Originally, the values of Condition are set up such that "1" represents in best condition, whereas "5" represents poor condition. However numerically and quantitatively this wouldn't make sense, as 1 is not greater than 5. Therefore, I decided to invert the values so that the items' conditions would be comparable more intuitively. In other words, every instance of a 5 was replaced with a 1, every instance of a 4 was replaced with a 2 and vice versa.

Before:

```
1    204733
3    189271
2    157306
4     14748
5      1194
Name: Condition, dtype: int64
```

After:

```
5    204733
3    189271
4    157306
2     14748
1      1194
Name: Condition, dtype: int64
```

This concludes our data cleaning part of the process. To summarize the data set after our cleaning, it was down to 560,000 rows and five columns. The five remaining columns were Condition, Brand, Category, Shipping, and Price.

Exploratory Data Analysis

Of the remaining columns, this is where I needed to decide which one was my target. This was fairly straightforward, of course it was the Price column. Meaning that the remaining columns were our features. First, I investigated if there were any significant correlations between the features individually and the target.

I used a simple .corr() call for Condition vs. Price:

	Price	Condition
Price	1.00000	0.07189
Condition	0.07189	1.00000

This suggests that there is very low correlation between Condition and Price. For the remaining three categorical features, I used one-way ANOVA hypothesis testing to check for significant correlation between them and Price. The test results for all three showed that there was significant correlation between them and Price.

I however still wanted to do further testing on the Condition column. My method of this was by doing Chi-squared testing, by looking at Condition + Brand vs. Price, and Condition + Category vs. Price. The results of this were still that Condition did not seem to have much of an effect on the outcome Price. The conclusions of my EDA were that I was going to use Brand, Category, and Shipping as my features, while dropping Condition.

Preprocessing

Going into preprocessing, our current data set size was 567,252 rows by 5 columns. This was still too much to handle during the modeling stage, and so I took a random 20% sample of this data, resulting in 113,450 rows.

I defined 'X' as the Brand, Category, and Shipping columns and had 'y' as the Price column. Because all of the columns in 'X' are categorical, I used one-hot encoding to format it to be ready for models to be fitted to it. The encoding ended up adding about 1000 dummy columns to our 'X' DataFrame. Going into the train/test splitting of the data, I went with a 70/30 train/test split. That concluded the preprocessing stage and I was now ready to start fitting and testing models.

Modeling

We first started with a baseline model that simply predicts the price as the mean every time. The two metrics of model performance that I used throughout my modeling phase were R score and mean absolute error. The R score for the baseline model was 0.0 and had a mean absolute error of 0.88. These served as a "bare minimum" benchmark going into evaluating models I would create later on. Because we were using a regression approach, I planned on testing the only two regressions models I knew about: linear regression and random forest.

Beginning with linear regression, I first wanted to see how a basic version of the model would perform. The results were already much better than our baseline model:

```
r2_score(y_train, y_pred_tr), r2_score(y_test, y_pred_te)
```

```
(0.4752705658019125, 0.46275633515855996)
```

```
mean_absolute_error(y_train, y_pred_tr), mean_absolute_error(y_test, y_pred_te)
```

```
(0.6285439201081136, 0.6303507832549848)
```

Our R score was 0.46 compared to our baseline R score of 0, and our MAE was also significantly lower compared to the baseline's. This only being a baseline model, there should naturally be more room for improvement. I decided to refine the model by doing hyperparameter tuning on selecting k best features.

Because of computational limitations, I was not able to iterate through every possible k between 1 and 1100, I compromised and searched for the best k in step sizes of 20. The results were that 201 was the best amount of features to use and so I created a model that only took the best 201 features and tested it. The performance metrics were actually not as good as those from our basic linear regression model.

```
r2_score(y_train, y_pred201tr), r2_score(y_test, y_pred201te)
```

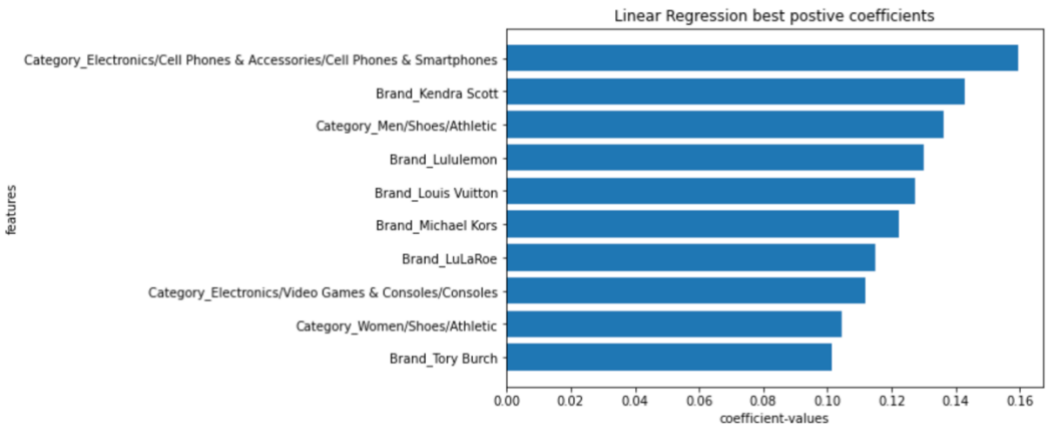
```
(0.41418313427887954, 0.4109796451152883)
```

```
mean_absolute_error(y_train, y_pred201tr), mean_absolute_error(y_test, y_pred201te)
```

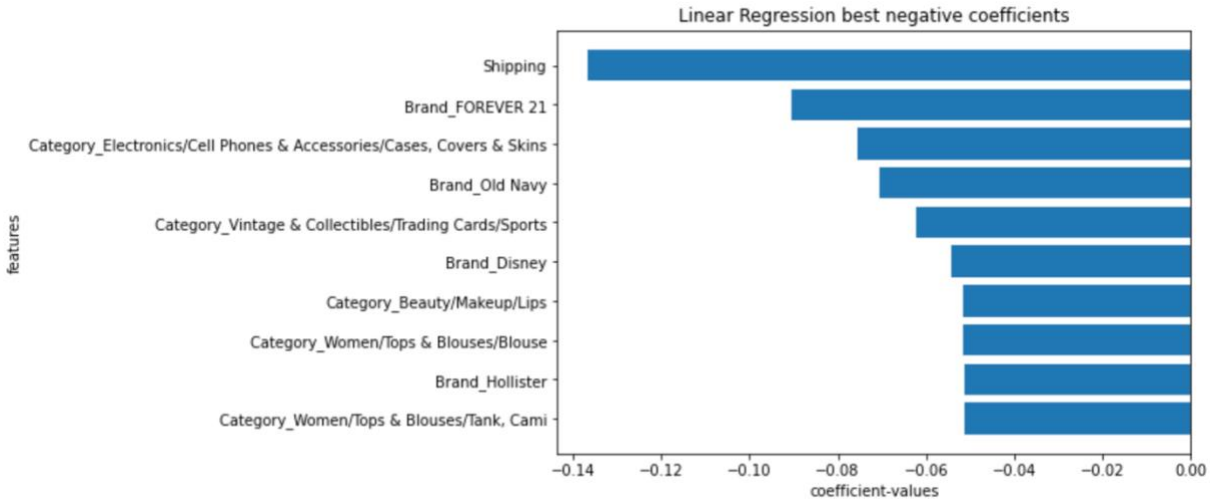
```
(0.6692841493339317, 0.6653032993029445)
```

Our R score was slightly lower and our MAE was slightly higher. However, this model is less likely to overfit and its performance metrics aren't that much worse than the base model's, and so I decided on this version of the model for our best linear regression model.

Using this model, I went on to do analysis on coefficient impact of the model by identifying the most positive and negative coefficients and their corresponding features names. In theory, the most positive coefficients should contribute the most in increasing price, and the most negative coefficients should contribute the most in decreasing price.Let's first look at the most positive coefficients:



The fact that these are our most positive coefficients makes a lot of sense. The brands pictured include some luxury ones, and the categories here are also of the more expensive ones. This is a relief because it seems that this model is correctly identifying the more expensive brands and categories as the biggest contributors to increasing the price. Let's now look at the most negative coefficients.



Likewise, seeing these as our most negative coefficients is a good sign. The feature names listed are all fairly cheap brands and categories. The one feature of interest is Shipping, as it has the most negative coefficient by far. This means that item listings where the seller is paying for shipping also tend to be lower priced. Perhaps this can be explained by sellers who are desperate to sell off their item. Not only will they lower the price to make it attractive to

buyers, but they will also offer to pay for shipping to make the deal seem extra enticing. Let's now move on to our Random Forest modeling.

Starting with a basic random forest model with a max depth of 50 and 15 trees, we obtained the following performance metrics for it.

```
r2_score(y_train, y_predRFtr), r2_score(y_test, y_predRFte)
```

```
(0.408128407615975, 0.37491119258967387)
```

```
mean_absolute_error(y_train, y_predRFtr), mean_absolute_error(y_test, y_predRFte)
```

```
(0.6721901422229208, 0.6843152360244126)
```

Not the worst, but not as good compared to either of our linear regression models. There's definitely room for improving our random forest model, most likely based on either the max depth and number of trees parameters.

I unfortunately was not able to use GridSearch and cross validation to search for the optimal amount of either of these parameters due to computational restrictions. As a compromise, I instead had to just simply use trial and error to improve the random forest model.

While adjusting the max depth and n_estimators parameters, I noticed that increasing n_estimators did not really have as much of an effect on improving performance metrics compared to increasing max depth. After testing four different versions of the random forest models with different parameters, I ended up with this one,

```
RF_pipe4 = make_pipeline(
    StandardScaler(),
    RandomForestRegressor(max_depth=200, n_estimators=50)
)
```

and obtained the following metrics for it:

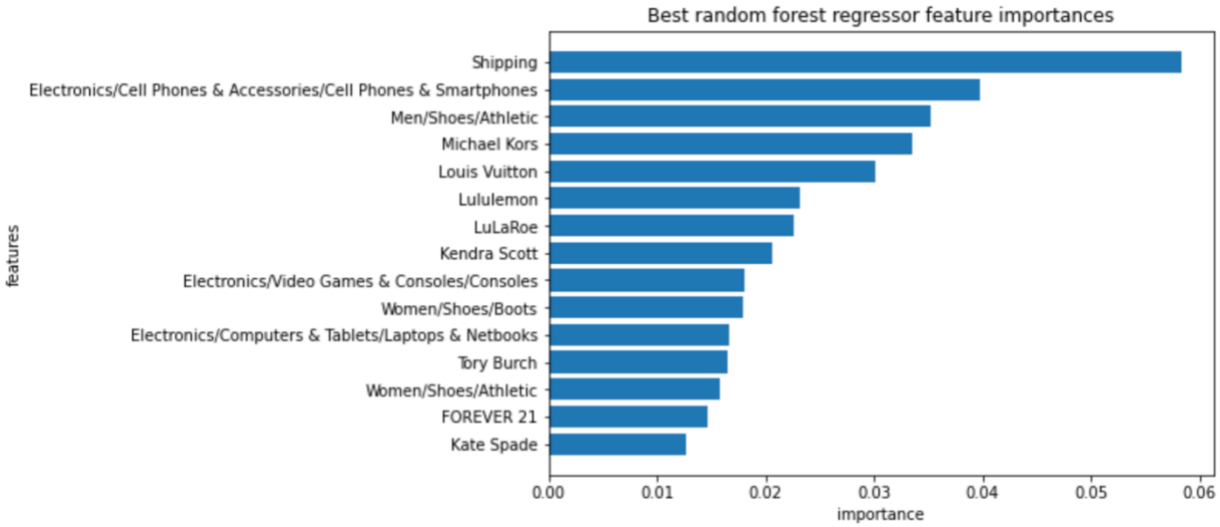
```
r2_score(y_train, y_predRFtr), r2_score(y_test, y_predRFte)
```

```
(0.5352083165172818, 0.45844209462714314)
```

```
mean_absolute_error(y_train, y_predRFtr), mean_absolute_error(y_test, y_predRFte)
```

```
(0.5880756292478646, 0.633345754478913)
```

These were our best metrics yet, so I decided on this model as the final random forest model. Following this, I conducted some analysis on feature importance.



I had seen most of these features when I was doing feature impact analysis for my linear regression model. This is a good thing, as it shows that both of the regression models were coming to the same conclusions as to which features were the biggest influencers on price. It seems that both the biggest influencers of price, either positively or negatively, were listed here.

For selecting the final overall model, all I had to do was compare the performance metrics of the two final models I had made. Their metrics are listed as following:

Model	Linear Regression	Random Forest
R score	0.4109	0.4584
Mean Absolute Error	0.6653	0.6333
Mean Absolute Percentage Error	16.3564	15.4824

From the table, it was clear that our random forest model was performing better. Moving from there, it was time to do some error and residuals analysis.

Because I took the log of the Price column initially, I would need to account for that when analyzing my error calculations. I was able to recalculate both the MAE and MAPE as such:

```
print("Mean absolute error: ", MAE)
print("Mean percentage absolute error: ", MAPE)
```

```
Mean absolute error:  14.178555136093182
Mean percentage absolute error:  0.4830132205610671
```

Having a mean absolute error of 14.18 means that on average, the model can expected to be off by around $14.18 in terms of its price predictions. Having a mean absolute percentage error of 48.3% means that the model's price predictions will be off by about that amount. This is of course not very ideal, as this is significant error to be having. Expectedly, it is most likely due to how we dropped many columns and rows of our original data set. As a conclusion, I ask that Mercari keep the expected error in mind when using our model.

Future Improvements

For future projects, I need to be able to have more methods of handling large data sets rather than simply cutting it down until it's workable with. For starters I can assign more efficient variable types to my columns. For example, my 'Condition' column only has values between 1 and 5, and yet it's being stored as an int64 when int8 would've been enough. This goes true for our one-hot encoding as well. Every value in the encoded DataFrame was stored as an int64 when int8 would've been enough, since all the values are either 0's or 1's. Once I'm able to properly handle the dataset in its entirety without having to worry about memory or runtime, I'll be able to build higher performance models for future projects.

My exploratory data analysis was also rather lacking. Instead of obtaining correlation coefficients between all features, I was only able to find pairwise correlations one at a time because I didn't know how to properly handle categorical, non-numeric data. Once I'm able to execute this, I'll be able to conduct my analysis with less bias.

I also have much studying to do when it comes to all the different types of models. For our regression problem, I only knew of two models suited for this. In reality, there may have been a more appropriate model for my type of business problem, that I wasn't aware of. During future projects, I'll be more aware of all of the different model options I have at my disposal so I can truly decide on the best one.