The Price Is Right

Oleksandr-Patrik Yandola, Marc Bacvanski, Jason Gin
DS4400 Fall 2022

Abstract

With the vast majority of products now being sold by online retailers like eBay, Amazon, and Mercari, the market grows ever more competitive. Thus, it is more important than ever for merchants to intelligently price their products. E-commerce businesses themselves have opportunities to benefit from suggesting pricing strategies, and in this project, we seek to predict the price of online goods from their listing attributes, including name, brand name, and description, with the help of Machine Learning.

Introduction

The product price prediction problem is unique and multi-faceted, and small differences in products can mean large differences in price. Further, we must rely on data that merchants provide, which can include noise, clickbait, and exaggerated advertising. Finding a good solution requires a combination of natural language processing and machine learning concepts. A good solution will enable e-commerce companies to intelligently and accurately suggest prices for merchants on the platform, and perhaps provide insight into pricing strategies. In this project, we train a model to predict the price of a product sold online, given a featureset of only listing information provided by the seller. We used a dataset found on Kaggle, https://www.kaggle.com/competitions/mercari-price-suggestion-challenge, to train and test our model. We tested and compared Ridge Regression, Stochastic Gradient Descent Regression, MLP Regression, and DNN model built through Pytroch as predictors of the price based on product description.

Dataset

The dataset is given by the Kaggle competition, in the format of 5 files 430 MB in size. The files contain entries for a product sold on the Japanese online store website, Mercari. The training dataset features 1,482,535 samples, and the evaluation dataset contains 693,360 samples. We plan to combine these datasets and split them ourselves into a training and test set, with a distribution of roughly 80%-20% respectively. The feature vectors for each product are already available in the training data set.
Each sample's features include:

- `train_id` or `test_id`: the ID of the listing
- `name`: title of the listing, cleaned to remove text that looks like prices. Removed prices are represented as `[rm]`
- `item_condition_id`: the condition of the items
- `category_name`: category of the listing
- `brand_name`
- `price`: the target variable to predict, in USD
- `shipping`: 1 if shipping fee is paid by seller and 0 if by buyer
- `item_description`: full description of the item, also with removed prices represented as `[rm]`

While we investigate several approaches through this report, our overall strategy is to extract numerical, vectorizable features from textual data, and then use these second-order features to train a regression model.

Data Preparation

We imputed missing values as a string value “Missing”, “None”, or “Other”, depending on the field. A sampled example entry in our input dataset looks like this.

brand_name	Razer
category_name	Electronics/Computers & Tablets/Components & Parts
item_condition	3
item_description	This keyboard is in great condition and works like it came out of the box. All of the ports are tested and work perfectly. The lights are customizable via the Razer Synapse app on your PC.
name	Razer BlackWidow Chroma Keyboard
price	52
shipping	0

We observed that the category name is usually composed of a general category and two subcategories, in this case, “Electronics” and then “Computers & Tablets” and “Components & Parts”. In preprocessing, we split these into three separate fields: “general_category”, “subcategory_1”, and “subcategory_2”. Further, we take the natural log of (1 + price).

brand_name	Razer
general_category	Electronics
subcategory_1	Computers & Tablets
subcategory_2	Components & Parts
item_condition	3
item_description	This keyboard is in great condition and works like it came out of the box. All of the ports are tested and work perfectly. The lights are customizable via the Razer Synapse app on your PC.
name	Razer BlackWidow Chroma Keyboard
price	3.970291913552122
shipping	0

We found that splitting the category into its general category and subcategories yielded the best results.

Data Vectorization

After preprocessing the data, we built a vectorizer using scikit-learn's FeatureUnion class. Different fields are treated differently.

We utilized a CountVectorizer on the fields "brand_name", "general_category", "subcategory_1", and "subcategory_2", ignoring stopwords. The CountVectorizer accumulates the occurrences of words within each of these fields, which makes more sense than a simple one-hot vectorization because we may want to form some association between categories that share words — for example, a category named "Camera Filters" may be closely related to a category named "Camera Tripods".

We utilized TF-IDF vectorization for the "name" and "item_description" fields. While we initially started with simple CountVectorizers on these fields, both of these fields can be quite long and descriptive. Because TF-IDF compares word frequency rather than count as CountVectorizer does, TF-IDF enables us to compare two different descriptions with very different lengths. We observe that while some names and some descriptions are very short, others are very long, and so even though the raw counts of word appearances may be difficult to directly relate, frequencies are amenable to comparison.

We used a one-hot encoding for "item_condition_id" and "shipping", which are represented as integers in the original dataset and so it is very natural to encode them in this way.

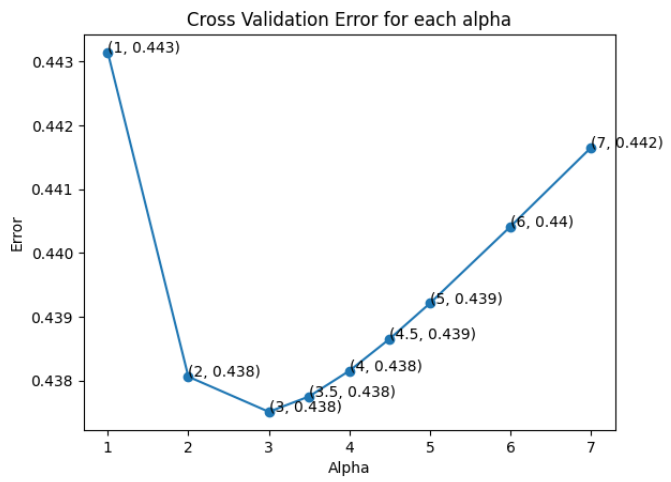
We split our training data by assigning 80% of it to training and 20% to validation. We fitted our vectorizers on the training data and applied the fitted vectorizers to both the training and test datasets.

Ridge Regression

Our overall approach with the ridge regression was to vectorize each item in our dataset and then fit a ridge regression to it. We did a line search over values for the coefficient α.

Parameter Search

The ridge regression formula is $\|A\mathbf{x} - \mathbf{b}\|_2^2 + \|\Gamma\mathbf{x}\|_2^2$. We sampled different values for Γ, and used mean squared log error as our metric for cross-validation. In the plot below, alpha is used as the parameter name instead of Γ. Plotting the values, we see that the optimal value for Γ is 3, which yields the lowest cross validation error in the test dataset of about 0.43875. This is competitive with results on our dataset's Kaggle competition, where our MSLE ranks us about 375 out of 2380 teams, in the top 15% of all teams worldwide.



Explanation

We utilized the eli5 library to explain our model's predictions. By writing custom vectorizer classes compatible with eli5 library, we were able to utilize eli5 to explain the weights our model associated with different features. Here, we use eli5 to show the top 30 features with the largest positive and largest negative weights.

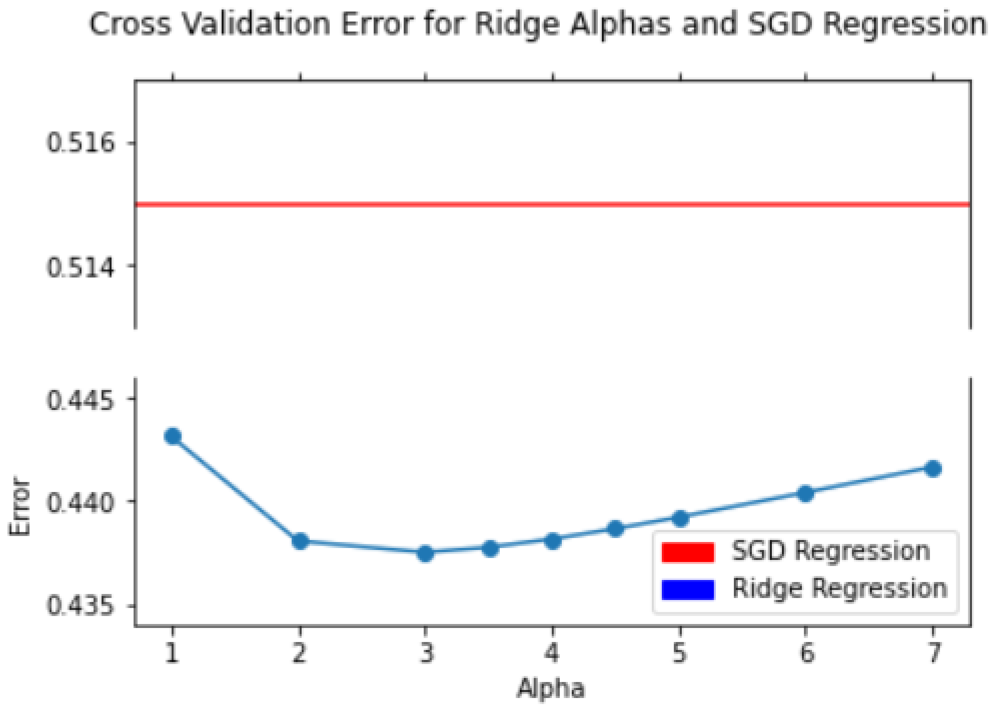
We see, for example, that a product with name "apple watch" has a very high weight, alongside other consistent indicators of expensive or high quality products: for example, when the item description contains "128gb" or "14k", when the name is "hoverboard", or when the brand name is "yurman". The features with the lowest weight seem, to the human eye, to indicate items of lesser value: names like "sticker", "controller skin", or "case". These feature weights are, to the human eye, relatively understandable.

2.111	name__tieks	-1.41	name__sticker
2.098	name__apple watch	-1.362	name__dust bag
2.031	name__14k	-1.229	name__controller skin
1.896	name__tyme	-1.208	name__dustbag
1.886	name__inr	-1.205	name__box only
1.871	item_description__128gb	-1.203	item_description__envelope
1.836	item_description__14k	-1.195	name__watch box
1.683	name__mcm	-1.156	item_description__tracking
1.645	name__hoverboard	-1.142	name__case
1.623	item_description__unlocked	-1.14	name__lululemon bags
1.604	name__tria	-1.032	name__amiibo card
1.601	item_description__carat	-1.022	name__minifigure
1.599	name__lash boost	-1.016	name__pocketbac
1.59	name__brahmin	-0.996	name__sample
1.586	name__10k	-0.978	name__apple watch band
1.556	name__dyson	-0.969	name__controller grips
1.552	name__bellami	-0.968	name__yeezy keychain
1.552	name__64gb	-0.966	name__coupons
1.539	name__gift cards	-0.961	name__hold for charlotte
1.518	brand_name__yurman	-0.959	name__stickers
1.506	name__playstation vr	-0.948	name__charger
1.502	name__nintendo switch	-0.945	item_description__13 x13
1.498	name__nerium	-0.935	name__iphone ipad
1.497	name__128gb	-0.934	name__iphone 4s
1.495	name__14kt	-0.904	name__hair ties
1.488	item_description__500gb	-0.9	name__shopping bag
1.486	item_description__16gb	-0.887	name__nose
1.449	name__james avery	-0.87	item_description__with stamp
1.437	name__tula	-0.87	name__gucci box
1.419	name__hatchimal	-0.867	name__love spell

SGD Regression

In addition to ridge regression, our team attempted to use Stochastic Gradient Descent regression. We utilized the same vectorization and data preparation that we used for the ridge regression above, as well as the same random_state variable that determines the order of the data shuffling - so that we could compare SGD and ridge regressions within the same context. Both the ridge and SGD regressions were imported from scikit-learn. We used the default loss function - ordinary least square fit for optimization.

The results for the SGD regression were worse on both the training and test sets than the Ridge regression. The cross validation error for the test dataset using SGD regression was 0.515. This error is considerably higher than ridge regression with the most efficient alpha of 3, which only had a 0.438. Not only that, the SGD regression has a greater error than even the worst alpha tested for ridge regression. At the alpha of 1, the ridge regression error is 0.443, still considerably lower than the SGD regression.



Deep Neural Network (PyTorch and sklearn)

Our final approach was to utilize a deep neural network to predict prices using a deep neural network. Originally we used MLPRegressor imported from scikit-learn to do our deep neural network training before we found out that PyTorch should have been used for DNNs instead. Once we realized we had to use PyTorch we also made a DNN using PyTorch, but since we already had the MPLRegressor data we decided to include it in the report alongside our PyTorch DNN data.

MLPRegressor DNN

With MLPRgeressor we used three hidden layers of size 64, Rectified Linear Unit (relu) as the activation function, and a maximum of 200 iterations. We made a few different attempts at using MLPRegressor by changing the number of samples we fed into the model, as well as the size of the batches to see how it affects the final error and the number of iterations. Below is the table with the data.

MLPRegressor Data

# samples	samples per batch	# iterations	Final test RMSLE
10,000	10,000	41	0.580
100,000	10,000	44	0.571
all	50,000	86	0.616

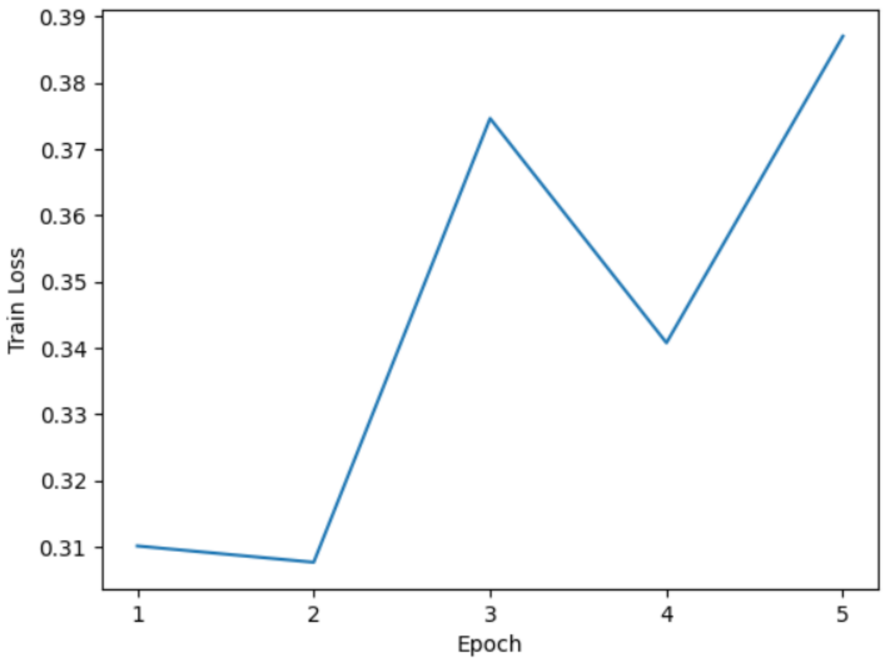
As seen in these results, MPLRegressor with 3 hidden layers of size 64, provided a worse result than both the SGD and ridge regressions.

PyTorch DNN

Now for the PyTorch implementation of the DNN model. Here we also used a relu activation function and also used 3 hidden layers. Each hidden layer was of size 256, which is 4 times the size of a hidden layer used in the MPLRegressor DNN described above. We ran the model on 10,000 data samples with batches of size 128, for a total of 5 epochs. The final RMSLE came out to 0.220, which is so low and accurate that it beggars belief. It is likely that the score was so good only because we got lucky with the 10,000 samples we randomly chose to test on. This score however is far superior to the RMSLE 0.580 for the MPLRegressor model that also ran on 10,000 samples but with no batches and smaller hidden layers.

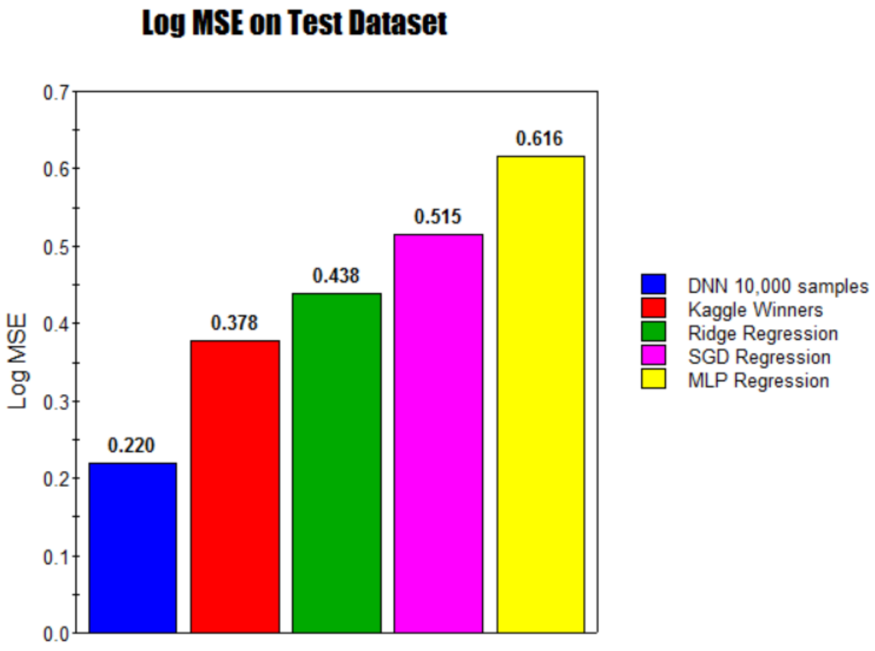
There were two reasons we ran the PyTorch DNN model on only 10,000 samples - RAM availability and compute time. The PyTorch DNN implementation was segfaulting on some computers and not others, and our current guess is that it was occurring due to a lack of RAM. The code was segfaulting on laptops with 12 GBs of RAM but wasn’t segfaulting on the computer with 32 GBs of RAM. As a result, only one person in the group was capable of running this code. The other reason was run-time - even when using a modern GPU for training on the small 10,000 sample set, it took multiple hours to run.

A strange phenomenon occurred when graphing the epoch loss of this DNN model against the number of epochs run. It was expected that with each epoch the loss of the model would decrease, but that was not the case:



We attribute this strange pattern to exploding gradient - a problem where large error gradients accumulate and then backpropagate into large updates to the model and its weights.

Conclusion



We do not account for our DNN model score, because the only reasonable explanation for why it outperformed the winner of the competition as much as it did - is that we got lucky with the 10,000 samples we used. Still, our Ridge Regression score is quite competitive, putting us in an approximate rank of #375 out of 2380 teams worldwide.

Overall, Ridge Regression coming out on top of the SGD Regression made sense to us. The big fluctuation between the Pytorch DNN and MPL Regression from scikit-learn is a bit harder to make sense of. MLP Regression had a considerably higher loss compared to Ridge and SGD regressions, and that loss seemed to increase the more samples from the data we fed into the training of the model. Possible explanations for that include too few hidden layers with too small of a hidden layer size and batch sizes that are too big. For comparison, the DNN model's total number of samples was five times smaller than the batch size of the MLP Regression model.

On the other hand, the PyTorch DNN model with its far smaller batch size and 4 times bigger hidden layers, could not be trained on two of the team members' computers, and took so long to train the team had to settle for using 10,000 samples instead of the full set. Such robust but expensive training could in part explain the huge differences between MLP Regression and the PyTorch DNN models. The other big part of course is the fact that it wasn't trained on the entire dataset, and we might have just gotten lucky.

This project was a useful learning experience on applications of Machine Learning algorithms to real-world use cases. The ability to suggest prices of internet goods based only on product listings has significant applicability in e-commerce and can find applications in intelligent price suggestions, and understanding the ways in which product listings inform pricing strategies. With the continual rise of online vendors, such algorithms are sure to become a mainstay in the future economy.

Our code can be found in our GitHub repo at https://github.com/mbacvanski/DS4400_PricePrediction.