Item Pricing Recommender

1. Presentation

Item Pricing Recommendation has one of its applications in e-commerce, to suggest sellers a price for the items they intend to sell. The recommender aims at several points: *inform* customers about a reasonable price that is supposed to increase the probability of selling their items, an *automatic* price feature avoid unrealistic intentionally prices and be a *traction* factor for users (new or already present).

The problem is treated as a *regression problem*, that of predicting the price of an item based on its features. In this particular case, the models will take as input the following features: *item name*, *item description*, *item brand*, *item category*, *item condition* and *shipping mode*, and will produce as output the **predicted price**.

2. Datasets and Inputs

The dataset is provided from the Kaggle competition [link](https://www.kaggle.com/c/mercari-price-suggestion-challenge/data)
 (https://www.kaggle.com/c/mercari-price-suggestion-challenge/data)
and consists essentially of:

- a *training* dataset:

train_id	item id
name	item title
item_condition_id	item condition
category_name	item category (enumeration of a hierarchy of categories separated by slashes)
brand_name	item brand name
price	the price the item was sold for [target variable]
shipping	1 or 0 (1 shipping fee paid by seller, 0 by buyer)
item_description	item full description

- a *test* dataset: same fields save for the price that is the target variable to be predicted for the items in this dataset.

3. Structure of the Project

I created a repo in GitHub **tonicava/capstone-octa** with the structure depected below:

📁	DataExploration	DataExploration
📁	RNNTorch	big files
📁	XGBoost	XGBoost
📁	data	big files

Specifically, in the *DataExploration* folder, there is the corresponding notebook: *DataExploration.ipynb* where I perform a detailed data analysis, described in **Section 4**.

The folder *XGBoost* comprises the *XGBoostModel.ipynb* notebook, where the model is built and trained in order to get the price predictions. It is discussed in **Section 5**.

In *RNNTorch*, there is the *RNNTorchModel.ipynb* notebook used to build the neural network based on PyTorch. Although we can find there a *resources* folder where I put the py files needed for building the *model*, the *training* and the *prediction* processes.

At the upper level, there is also a *data* folder, where I download the *tsv* file with the original data as well as the processed data files, to be uploaded to S3, in the case of the XGBoost model.

Besides, there are the other files required: the initial *proposal*, the *README* as well as the present file.

4. Data Analysis

The code is given in **DataExploration/DataExploration.ipynb**

A sample of the training data is given below:

```
df.head(5)
```

	train_id	name	item_condition_id	category_name	brand_name	price	shipping	item_description
0	0	MLB Cincinnati Reds T Shirt Size XL	3	Men/Tops/T-shirts	NaN	10.0	1	No description yet
1	1	Razer BlackWidow Chroma Keyboard	3	Electronics/Computers & Tablets/Components & P...	Razer	52.0	0	This keyboard is in great condition and works ...
2	2	AVA-VIV Blouse	1	Women/Tops & Blouses/Blouse	Target	10.0	1	Adorable top with a hint of lace and a key hol...
3	3	Leather Horse Statues	1	Home/Home Décor/Home Décor Accents	NaN	35.0	1	New with tags. Leather horses. Retail for [rm]...
4	4	24K GOLD plated rose	1	Women/Jewelry/Necklaces	NaN	44.0	0	Complete with certificate of authenticity

All of the 1482535 items are different.

```
df.shape
```
```
(1482535, 8)
```

```
df['train_id'].nunique()
```
```
1482535
```

There are four features with *null* values (the three below plus the equivalent "No description yet" in *item_description*). Only the 6237 items with no category may be dropped.

It is interesting to analyse the hierarchy structure of the categories, we get there are at most 5 levels

```
ddf = df.copy()
ddf = ddf.join(ddf['category_name'].str.split('/', expand=True).add_prefix('cat_').fillna(np.nan
ddf.head()
```

dition_id	category_name	brand_name	price	shipping	item_description	cat_0	cat_1	cat_2	cat_3	cat_4
3	Men/Tops/T-shirts		NaN	10.00000	1	No description yet	Men	Tops	T-shirts	NaN	NaN
3	Electronics/Computers & Tablets/Components & P...	Razer	52.00000	0	This keyboard is in great condition and works ...	Electronics	Computers & Tablets	Components & Parts	NaN	NaN
1	Women/Tops & Blouses/Blouse	Target	10.00000	1	Adorable top with a hint of lace and a key hol...	Women	Tops & Blouses	Blouse	NaN	NaN
1	Home/Home Décor/Home Décor Accents		NaN	35.00000	1	New with tags. Leather horses. Retail for [rm]...	Home	Home Décor	Home Décor Accents	NaN	NaN
1	Women/Jewelry/Necklaces		NaN	44.00000	0	Complete with certificate of authenticity	Women	Jewelry	Necklaces	NaN	NaN

with the following number of elements:

```
cat_0        10
cat_1       113
cat_2       870
cat_3         6
cat_4         2
```

The first level category (cat_0) is too coarse regarding the price distribution to be used alone, as the below histogram shows:
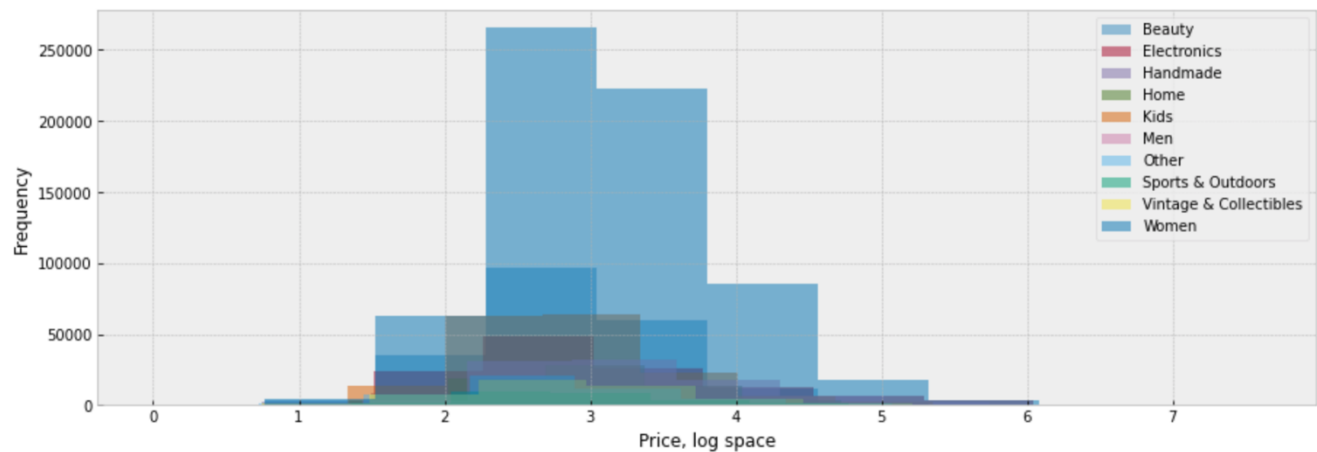
The idea is to go deeper, and I will choose the level 2 category (we start counting at 0) that is the deepest one common to all of the items.

In the notebook mentioned above (**DataExploration/DataExploration.ipynb**), I explore other statistical aspects, such as the distribution of the log of the price by category or the number of the items per category. The latter is interesting to see for the ones with at least 10,000 items, with a special note for *Others*, in second place.
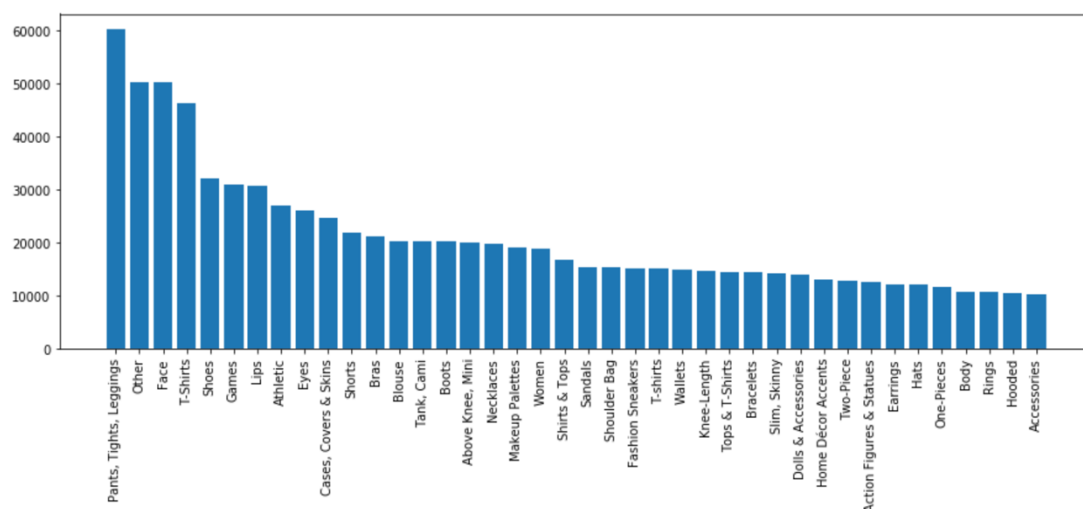
The former (log price by category) is shown with violin plots that combine the box and kernel density plots. It is interesting to note that there are not big variations (in terms of median value and interquartile range).
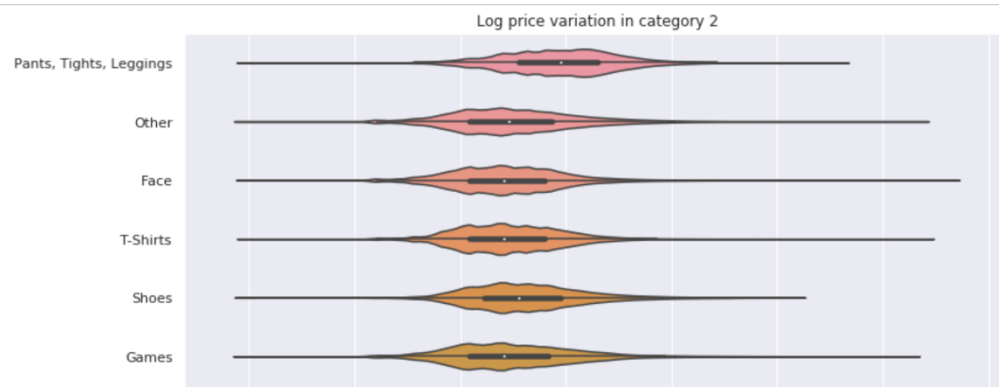
Further preprocessings I will apply directly when building the corresponding models.

5. XGBoost Model

It is the first model I build: an extreme gradient boosting (**XGBoost**) model (the implementation provided by AWS).

The code is provided in **XGBoost/XGBoostModel.ipynb** and it uses, as data repository, the folder **data/**.

5.1. Features used
I first preprocess the data, doing the following transformations:

- Drop the 198 items with no price (I consider them irrelevant for the suggestion)

- Replace nulls/nan's with empty strings in the rest (*brand_name*, *item_description*).

  name, *item_condition_id* and *shipping* are defined for all of the items.

- **Category**: as previously said, I consider the category of level 2 (third level), defined for almost all of the items (6,327 lack cat_2, out of 1,482,535). As such I split the initial hierarchy string and extract cat_2.

  Eventually I *binary-encode* it, so I get 11 columns for the 870 cat_2 categories.

- **Item condition:** I apply, too, a binary encoder, getting 4 columns (since there are only 5 values, I could have applied, too, a one-hot encoder, getting one more column).

- **Shipping:** possible values 0 or 1. I use one-hot encoding, thus getting 2 columns.

- **Brand name:** There are 4790 different values, so I use, too, a binary encoder, getting thus 14 columns.

- **Name:** I derive two features, first the polarity of its sentiment analysis, then its scaled length.

- **Item description:** I perform the same as for *name*.

I then drop the string columns, and *train_id*, thus remaining with **35** features, plus the *price* column.

Finally I compute the correlations between the features to make sure no strong correlation is present, which is the case.

5.2. Model Training

I split the data consisting of rows 1,475,347 in 10% test and 90% rest that, in turn, is split in 90% train and 10% dev, thus I get: 1,195,030 train, 132,782 validation and 147,535 test data. (For this data magnitude, the most common split is 90:10)

Next I store each of the data in csv files (locally in the *data* folder) and uplode to S3.

I use the latest image of the XGBoost model (it is noteworthy to point out that we have now to indicate the desired version, so I use *latest*).

For the instance to be run, I use:

```
"InstanceCount": 1,
"InstanceType": "ml.m4.xlarge",
"VolumeSizeInGB": 5
```

and for the static parameters, the following values:

```
"gamma": "4",
"subsample": "0.8",
"objective": "reg:linear",
"early_stopping_rounds": "10",
"num_round": "200"
```

Since I wanted to make use of the HyperParameter tuning given by SageMaker, it is important to notice that I could not use as metric RMSLE for evaluation, because as far as I could investigate, it is not possible to use it (it is about the *MetricName* key, inside *HyperParameterTuningJobObjective*), so I used along the project **validation:rmse**. This is also why, although starting the analysis also of the log of price, I kept using *price* as is.

Still, I computed, in the evaluation part, also the RMSLE metric, as well as, MAE and MAPE.

The parametrizable parameters are thus the following, within the min-max ranges:

```
"ContinuousParameterRanges": [
   {
      "MaxValue": "0.5",
      "MinValue": "0.05",
      "Name": "eta"
   },
 ],
 "IntegerParameterRanges": [
   {
      "MaxValue": "12",
      "MinValue": "3",
      "Name": "max_depth"
```

```
          },
          {
             "MaxValue": "8",
             "MinValue": "2",
             "Name": "min_child_weight"
          }
       ]}
```

I consider a maximum of *20* training jobs with at most *3* in parallel.

The best performing found has the following values:

'eta': '0.10124202006911284',
'max_depth': '12',
'min_child_weight': '5'

that I will use subsequently, for an RMSE value of **31.423200607299805**.

5.3. Model Testing and Analysis

I create the model and then use the SageMaker BatchTransform to create the transform job. The resulted output I download and perform the analyses below.

By means of a **scatter plot**, we can see how the predicted price is aligned with the test price.

As it can be seen, higher values in the test price are not well predicted.

Actually we can get a better hint, by measuring the Root Mean Square Error, which is **30.127954014320473**.

We can see that there is quite a considerable spread out of the predicted price value with regard to the test one.

Further metric values are:

RMSLE = 0.6193182449232817
MAPE = 0.7027854194275958
MAE   = 13.54716844990529

RMSLE is particularly interesting for the case study, since it tends to predict prices slightly superior to the test ones (it is actually a asymmetric metric). Its value is quite high.

As a general conclusion, the metric used to evaluate the XGBoost model shows we have *underfitting*.

6. PyTorch RNN Model

The second model I build is a PyTorch RNN.
The code is provided in **RNNTorch/RNNTorchModel.ipynb** and it uses, as data repository and py files for the deployment and inference, the folder **RNNTorch/resources/**.

First I apply the preprocessing steps that I presented in the *Subsection 5.1* above.

Then I proceed to create the model.

6.1. Model
The layers that compose the model are depicted below.
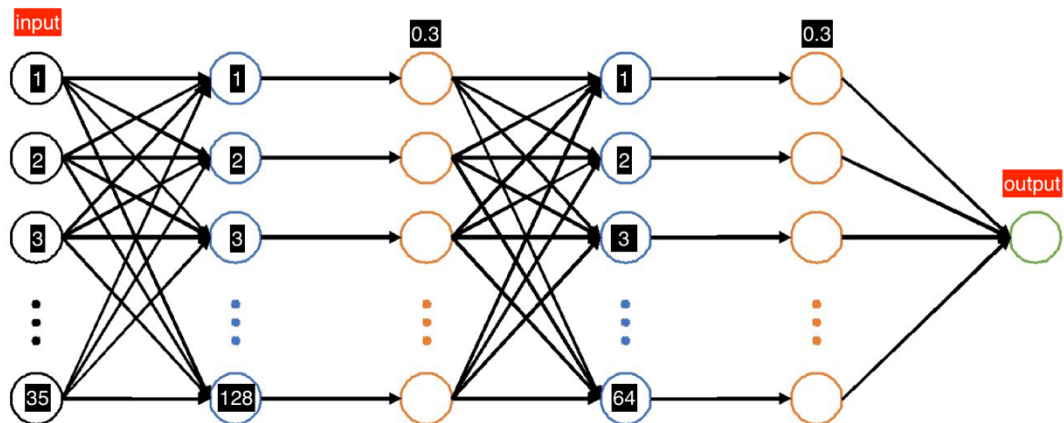
Basically the are the 35 input features and then 2 hidden layers, one of 128 and the other of 64 nodes, and the output.

We apply a dropout of 0.3 after each hidden layer and a rectified linear activation function (ReLU).

The model is coded in **RNNTorch/resources/model.py**

6.2. Training and Deployment
I split the data in train and test: 1,180,277 and 295,070 (80%:20%).

Then, in **RNNTorch/resources/train.py**, I implement the training algorithm with batches, using the model described in the previous subsection.

Next, I create an estimator, using train.py.
Important note: because I worked with a ml.c4.4xlarge and no GPU, I limited the training to 20 epochs.

Although in this case I could have applied, for evaluation, RMSLE, I kept using RMSE (that I implemented in the same train.py file - PyTorch doesn't have an implementation for it) since I wanted to be able to compare the results against the "baseline" XGBoost model. The final value was **21.855074466870626**, a lot smaller compared to the one of the XGBoost **31.423200607299805**)

The next step was to deploy the model, creating a SageMaker endpoint.

6.3. Model Evaluation
In **RNNTorch/resources/predict.py**, I wrote down the functions allowing to predict the result, deserialize the request object and serialize the response.

Then I performed the evaluation on the test data, getting for RMSE a value of **27.18863965354332**. It shows *overfitting*, but we have to be careful when interpreting because of the limited number of training epochs.
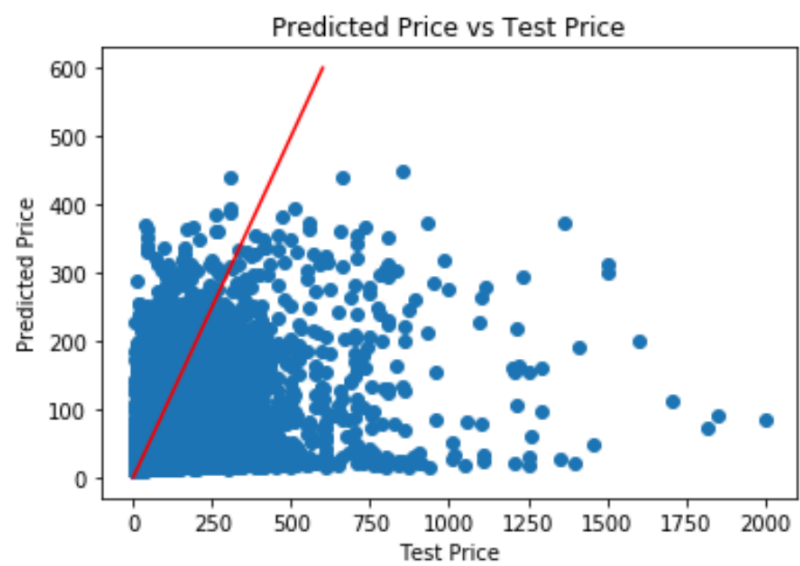
For the rest of the metrics, the following values were found:
RMSLE = 0.8581451773643494
MAPE = 1.0561008314855789
MAE   = 19.6236273608

Finally, the **scatter plot** shows that there is a lot of room for improvement, by: playing with the hyperparameters, number of epochs, etc.

Predicted Price vs Test Price

7. Evaluation Metrics

I computed the following metrics, using for evaluation RMSE (I explained the reason in the *Subsection 5.2*, when talking about the XGBoost model).

Root Mean Squared Error (RMSE)

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{p}_i - p_i)^2$$

Root Mean Squared Logarithmic Error (RMSLE)

$$\sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(log(\hat{p}_i + 1) - log(p_i + 1))^2}$$

Mean Absolute Error (MAE)

$$\frac{1}{n}\sum_{i=1}^{n}|\hat{p}_i - p_i|$$

Mean Absolute Percentage Error (MAPE)

$$\frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{p}_i - p_i|}{p_i}$$

n = number of observations in the dataset,
$\hat{p}$ = price prediction,
p - actual price.