Classify Products into Different Virtual Shelves

By Jinjin Ge, 11/4/2016

Training Data Exploration

At first glance, there are many missing values among most of the variables, which is understandable because we can barely have perfect completed dataset from real-world business. Among these incomplete data, there are 18 items that have completely missed all features but only have the “item_ID” and “tag”, which is also acceptable comparing to the total number of 10,593. I choose to keep these 18 items because the test dataset also has the same problem.

The second issue about this training data is that, there are multiple variables indicate the same feature. For example, “Actual Color”, “Color”, “actual_color” should be aggregated as one “color” feature; “Product Short Description” and “Short Description” seem be one feature but named differently on some items, because those items whose “Product Short Description” are empty but all have values in “Short Description” column.

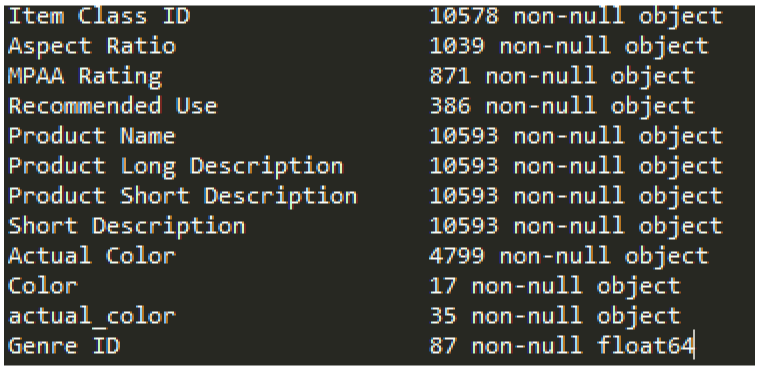
I also notice that “Genre ID” has highly correlation with “tag”; however, only the items whose “Item Class ID” is 4 have values in “Genre ID”, and only 88 of the 10,593 items have data in “Genre ID”. The same thing happens to “Aspect Ratio”, “MPAA Rating” and “Recommended Use”. Therefore I decide to include these 4 features but composite them with some other feature has less missing values, like “Item Class ID”.

Last but not least, the text contents in the training dataset are so non-standardized that it’s hard to code them as categorical variables. Also, considering almost all of the items have “Product Name” and the product description related variables, I decide to extract features from those text data (i.e., turn the text contents into numerical feature vectors).

Data Preprocessing

Predictors:

I selected 12 variables from the training dataset to composite 4 text variables for word feature extraction.



```
Item Class ID                  10578 non-null object
Aspect Ratio                   1039 non-null object
MPAA Rating                    871 non-null object
Recommended Use                386 non-null object
Product Name                   10593 non-null object
Product Long Description       10593 non-null object
Product Short Description      10593 non-null object
Short Description              10593 non-null object
Actual Color                   4799 non-null object
Color                          17 non-null object
actual_color                   35 non-null object
Genre ID                       87 non-null float64
```

The finished 4 predictors are:

"**Product Name**": Filled the missing value in "Product Name" with text "NoProductName".

"**Product Long Description**": Filled the missing value in "Product Long Description" with text "NoProductLongDescription".

"**New Product Name**": Composited by 'Product Name', 'Item Class ID', 'Genre ID', 'Aspect Ratio', 'MPAA Rating', 'Recommended Use', 'Actual Color', 'Color', and 'actual_color'.

"**Combined Short Description**": Composed by 'Product Long Description', 'Product Short Description', 'Item Class ID', and 'Genre ID'.

To extract features from the text variables, I used feature_extraction.text module from ***SciKit-Learn***. It has two steps: first, text preprocessing, tokenizing and filtering of stopwords to build a dictionary of features and transform documents to feature vectors. CountVectorizer supports counts of N-grams of words or consecutive characters:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vect = CountVectorizer()
>>> X_train_counts = count_vect.fit_transform(train_df)
```

Then, it suffices to divide the number of occurrences of each word in a document by the total number of words in the document, which is called "Term Frequencies (tf)"; or another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus, which is called "Term Frequency times Inverse Document Frequency (tf-tidf)":

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf_transformer = TfidfTransformer()
>>> X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
```

By applying the two steps to the 4 finished text variables, I have 4 bags of word information as my predictors.

Tag:

Each product can have a list of different tags, therefore this is a multi-labels classification problem. I tried two ways to code the tag list:

1. Code the tag list as one vector with 32 binary elements. Each element represent one tag, with value of 1 means having this tag, 0 means not. Then fit one classifier to predict this tag vector for the test dataset.

   For example:
   [0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] = [1229817, 1229821]

2. Code the tag list into 32 individual binary variables, each variable represent one tag, with value of 1 means having this tag, 0 means not. Then fit 32 classifiers using the 32 binary variables as label, respectively; finally use the 32 classifiers to predict the 32 binary tag variables for the test dataset.

   For example, [4483, 5065] will be represented as variable “4483” =1, variable “5065” = 1, and the rest 30 tag variables all equal to 0.

After comparing the accuracies got from above the two ways using the same predictors and model parameters, **I decide to use the 2nd method – fit 32 classifiers for the 32 tags, respectively**. Because with the same predictors and modeling parameters, the 32-classifier method always has a higher accuracy than the first one.

Model Selection

I used a supervised Nueral Network model called “**Multi-layer Perceptron classifier**” that implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation. My model has **one hidden layer** which has **50 neurons**. To avoid overfitting, I used an **alpha = 0.1** to penalize weights with large magnitudes.

As I mentioned before, I am making individual classifier for each of the 32 tags, respectively, therefore, I chose ‘**lbfgs**’ as the solver for weight optimization, because it can converge faster and perform better.

```
>>> clf = MLPClassifier(solver = 'lbfgs', alpha = 0.1, hidden_layer_sizes = (50, ), r
andom_state = 1)
```

The whole prediction procedure involves 4 modellings.

The first run uses the features extracted from “**New Product Name**” as predictors, to predict most items’ tags. The reason why I use features from “New Product Name” because this composited variable has integrated most variables’ information.

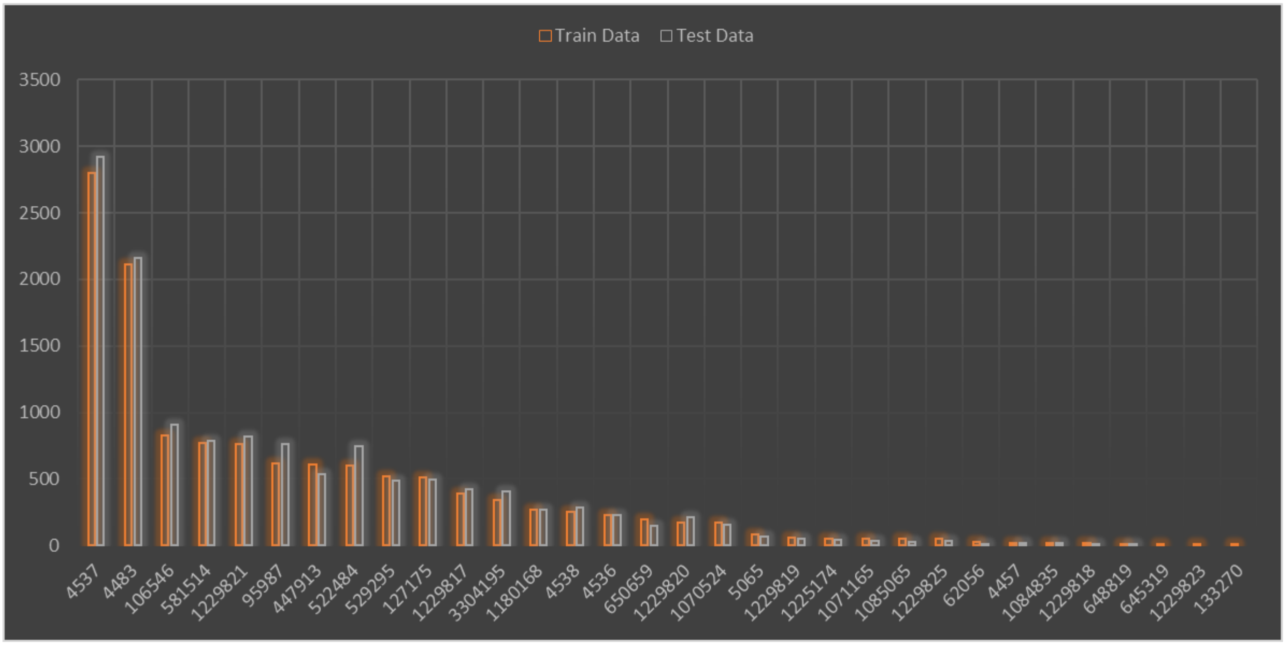
After the first run, there are still 993 items untagged. So the second run, which uses features extracted from "**Product Long Description**" as predictors, is applied to predict the tags for the 993 untagged items.

After the second run, there are still 315 items untagged. So the third run, which uses features extracted from "**Combined Short Description**" as predictors, is applied to predict the tags for the 315 untagged items.

After the third run, I have one unused feature vector, which is extracted from "**Product Name**", and there are still 178 items untagged. So I decide to do the forth and the last run, which uses the last feature vector, to predict the tags for the 178 untagged items.

Unfortunately, there are still 135 item untagged. I decide to leave as they are. And this is the biggest shortness of my model – it cannot guarantee all items have at least one tag. With large enough training samples and more features could probably solve this problem; or by tuning the alpha penalty to more regularize the classifiers (but may decrease the accuracy).

To confirm my prediction, I compared the tags distributions in the training dataset and my predicted tags for the test dataset. They look very similar! So, I am more confident that my prediction of tags is not bad ☺


Train Data
Test Data