Predicting future demand for products sold in Amazon

Data science & Deep learning research conducted on behalf of FeedVisor

Introduction:

Our mission was providing a demand forecast of products sold on Amazon by merchants who are customers of FeedVisor, based on history sales details of each product.
Each product has few listings, when a listing is a product sold by a specific merchant.
We decided calculating the demand of each listing per day as defined by the fracture below:

$$\frac{\text{Ordered quantity per listing}}{Top\ Rank\ Share}$$

While,

- Ordered quantity per listing - total number of units ordered in a specific day (each order may be of multiple units)
- Top Rank Share - percent of time this competitor(merchant) has been in top rank (buy box)

In order to get daily demand per product, we will sum all of listings demands belong to that product.

Loading the Data

Out of all data csv files provided, we used the following:

- Dim Categories - provides the name of each category id.
- Fact Product Category Daily - provides a category id for each product.
- Dim Listings- provides the connection between product id and listing id.
- Fact Listings Daily - includes daily data for each listing of each of FeedVisor clients.

We used 'Pandas' and 'Dask' Python packages for loading data and doing calculations on it.

```
import pandas as pd
import dask.dataframe as dd
```

In order to ease the problem and use data that will fit best to our perdition model (will be better explain in the next section); we decided to focus on **Toys** category of products.
In the following code lines, we loaded 'Dim Categories' table in chunks (avoiding memory error) and extracted all category ids belong to category paths containing the word 'Toy'.

```
# find all category ids of the Toys category.
category_chunksize = 10000
category_col_names = ['Category_ID', 'Category_Path']
category_cols_numbers = [2,4]
# dataframe of category IDs of all toys categories
relevant_category_IDs = pd.DataFrame()
# relevant_category_IDs = []
General_Category = "/Toys"

for df in pd.read_csv(dim_categories,names=category_col_names,usecols=category_cols_numbers,
                      chunksize=category_chunksize, iterator=True):
    # delete rows with at least one NA
    df = df.dropna(axis=0, how='any')
    # Changes Path column to string values
    df['Category_Path'] = df['Category_Path'].astype(str)
    # DEFINE - Select only rows with specific string (Specific category)
    df = df[df['Category_Path'].str.contains(General_Category)]
    frames = [df, relevant_category_IDs]
    relevant_category_IDs = pd.concat(frames)
```

Next, we find all of products id's connected to each one of the category id's we have found before.

```
##########################################################################################################
# Find all toys products IDs
toys_prod_ids = dd.read_csv(category_to_product, names=['Category_ID','Product_ID'], usecols=[3,5])
toys_prod_ids = toys_prod_ids.loc[toys_prod_ids['Category_ID'].isin(cat_id_list)]
```

After having Toys product ids in a data frame, we find all listing ids of each product by joining the data frame with 'Dim Listing' table separately loaded.

```
# Find all toys listings IDs
prod_to_list = dd.read_csv(product_to_listing,names=['Listing_ID', 'Product_ID']
                           , usecols=[0, 3])
# join between prod-category and prod-listing
toys_prod_to_list = toys_prod_ids.set_index('Product_ID')\
    .join(prod_to_list.set_index('Product_ID'), how='inner')
toys_prod_to_list = toys_prod_to_list.reset_index()
# turn the dask dataframe into pandas dataframe
toys_prod_to_list = toys_prod_to_list.compute()
```

Now that we have all relevant listing id's we would like to get daily details about them.
In the next section of code, we load all rows but only necessary columns out of 'Fact Listing Daily' table into a dask dataframe. We removed all lines with some missing data or those have 'Top Rank Share' of zero, avoiding divide ordered quantity by 0 (as part demand calculation).

```
# Find daily details of toys listings
listing_col_names = ['Date', 'Listing_ID', 'Ordered_Quantity',
                     'Avg_Num_Of_Listings', 'Top_Rank_Share']
listing_col_numbers = [0, 1, 5, 10, 12]
list_det_df = dd.read_csv(listings_daily, names=listing_col_names,
                          usecols=listing_col_numbers, dtype={'Ordered_Quantity': float})
list_det_df = list_det_df.dropna(how='any')
list_det_df = list_det_df[list_det_df.Top_Rank_Share != 0]
# # Union
list_det_df = list_det_df.repartition(npartitions=list_det_df.npartitions // 5)
list_toys_daily = list_det_df.merge(toys_prod_to_list, on='Listing_ID', how='inner')
list_toys_daily = list_toys_daily.drop_duplicates()
```

Out of all toy listings we collected, we would like to have only the most frequent ones to consist our training dataset. Therefore, we grouped list_toys_daily data frame by Listing_ID and counted number of observations each group has. We took the 100 most frequent listing ids and saved them.

```
# This part of code used to calculate the 100 top frequents of toys listing IDs and send them to csv format
list_top_freq = list_toys_daily.groupby(['Listing_ID'])['Date'].count().nlargest(100)
list_top_freq = list_top_freq.compute().to_frame()
list_top_freq.to_csv('Top_Frequent_Toys_Listings.csv')
```

After reading the csv file of the frequent listing id into dataframe and then saving them as a list, we looked for rows containing thos listings ids in our joined daily list_toys_daily dataframe. Finally, we calculted the demand forecast of each listing and saved it in a new column called 'Demand'.

```
# Take only 100 top freq listings daily details
list_toys_top_freq_daily = list_toys_daily.loc[list_toys_daily['Listing_ID']
    .isin(top_freq_list_toys_list)]


list_toys_top_freq_daily['Demand'] = list_toys_top_freq_daily['Ordered_Quantity'] \
                                     / list_toys_top_freq_daily['Top_Rank_Share']
```

The aggregated data frame was save in a csv file containing 31,712 rows and this file constituted an initial training data set for the model.

Demand	Category_I	Product_ID	Top_Rank	Avg_Num_	Ordered_C	Listing_ID	Date		
1	1.66E+08	b3f49797c	1	1	1	3958751	01/02/2017	9087654	2
7	1.66E+08	b3f49797c	1	1	7	3534512	20/03/2016	897	3
3	1.66E+08	b3f49797c	1	1	3	3534512	21/03/2016	1195	4
0	1.66E+08	b3f49797c	1	1	0	3534512	22/03/2016	1493	5
0	2.94E+08	b633313fd	1	1	0	3541358	20/03/2016	1791	6
0	2.94E+08	b633313fd	1	1	0	3541358	21/03/2016	2102	7
0	2.94E+08	b633313fd	1	1	0	3541358	22/03/2016	2413	8
1	2.98E+09	aed5cb01a	1	1	1	3547846	20/03/2016	2724	9
4	2.98E+09	aed5cb01a	1	1	4	3547846	21/03/2016	3045	10
5	2.98E+09	aed5cb01a	1	1	5	3547846	22/03/2016	3366	11
0	1.66E+08	88dd6ab7c	1	1	0	3557220	20/03/2016	3687	12
0	1.66E+08	88dd6ab7c	1	1	0	3557220	21/03/2016	3998	13
0	1.66E+08	88dd6ab7c	1	1	0	3557220	22/03/2016	4309	14
0	1.66E+08	5175dd9e2	0.4	11	0	3958751	20/03/2016	11669	15
0	1.66E+08	5175dd9e2	0.47	11	0	3958751	21/03/2016	11976	16

Training the Model

We decided to build a LSTM (=Long Short Term Memory) deep learning model, that fits to this sequential forecast task.

In this point, the code reads the saved csv files and creates the relevant data frame which consist of date and demand columns.

We had to convert the date column to datetime type in order to be possible to sort the data according to the dates.

Then, we grouped by the date column, meaning we arranged the data according to it, and changed the demand column to be consisted of the average of all products which available on each specific date.

Finally, we saved the data as a Numpy array with float type demands (since it's better input for LSTM models).

```
data_set['Date'] = pd.to_datetime(data_set.Date)
data_set = data_set[['Date','Demand']]
# Arrange the data by dates - make an avg of all products demands on a specific date
data_set = data_set.groupby('Date').mean()
data_set_vec = data_set.values  # As a Numpy array
data_set_vec = data_set_vec.astype('float32')  # convert demands to float
```

Our LSTM model have a Sigmoid layer with has values between 0 to 1, therefore we used MinMaxScaler function to rescale the demand to this range, meaning we made normalization to the data.

We also tried a bit different model – we changed the Sigmoid layer to be Tanh layer and changed the rescale rang appropriately to be between -1 to 1 but this end up in worse results.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
data_set_vec = scaler.fit_transform(data_set_vec)
```

We wanted to use this prepared data for two goals:

1. Building the model – training the model and fit the best weights (weights=the model parameters).
2. Evaluate the model – checking the model quality and the demand predicting error.

Therefore, we split the data to:

1. Train = 67% of the data.
2. Test = 33% of the data.

```
# split into train and test sets
train_size = int(len(data_set_vec) * 0.67)
test_size = len(data_set_vec) - train_size
train, test = data_set_vec[0:train_size,:], data_set_vec[train_size:len(data_set_vec),:]
```

In order to use the dada array in the training procedure, we built a function to created a dataset matrix.

This function expects to get look_back parameter (is 1 by default). This parameter is the number of previous time steps to use as input variables to predict the next time period

We used look_back=5 in order to determine that each prediction of label y will be defined by 5 inputs x.

For example, for the following list of demands: 1.3, 2.3, 9.4, 5.6, 8.7, 9.0, 2.1, 5.7 .

The result is:

1.3, 2.3, 9.4, 5.6, 8.7

2.3, 9.4, 5.6, 8.7, 9.0

9.4, 5.6, 8.7, 9.0, 2.1

etc.

```
# reshape into X=t and Y=t+1
# input: look_back = 5
look_back = 5
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return numpy.array(dataX), numpy.array(dataY)
```

In this step, we built the LSTM model and started the training part – the fit part.

We used batch_size=1, but by default it means that each LSTM batch will be an individual part.

Therefore, we used LSTM model that has a stateful mode variable.

In case stateful=True, the last item on the previous batch will be the first item on the current batch.

The Dense layer with activation Sigmoid multiplies between the input and the weights (of the previous step) according to Sigmoid activation, and adds a bias.

The output of this layer is 1 and this is the prediction for the next step.

```
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1, activation='sigmoid'))
```

We used MSE loss function (mean squared error).

It calculates the sum of the squared differences between each true label (label y as explained before), and the predicted demand. Then, it divides the sum in the size of the data.

The MSE formula is: $\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$

The optimizer we chose is Adam, since it's a very common optimizer and he get better results than another one we tried (e.g. rmsprop).

Adam updates the gradients in each step t according to the mean and variance in step t-1.

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

We started with fitting the model 200 times (=200 epochs).

We saw that the results converged quickly so we changed it to be 100 and then to 50 epochs.

As we thought, it didn't ruin the results, so we fit the model that we built before with 50 epochs.

After every epoch we had to do reset_states manually since we are using LSTM with stateful mode, which turn off the default reset_states between epochs.

```
# input : num of total runs = 50
for i in range(50):
    print(i)
    model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
    i=i+1
```

The loss of the first 7 epochs decreased as below:

```
Epoch 1/1
9s - loss: 0.0877
1
Epoch 1/1
9s - loss: 0.0072
2
Epoch 1/1
9s - loss: 0.0068
3
Epoch 1/1
9s - loss: 0.0068
4
Epoch 1/1
9s - loss: 0.0068
5
Epoch 1/1
9s - loss: 0.0068
6
Epoch 1/1
9s - loss: 0.0067
7
Epoch 1/1
9s - loss: 0.0067
```

In order to check the model quality we evaluate it.

Then, we invert the demand value to the original range and calculated the RMSE (=root mean squared error).

```
# make predictions
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)

# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])

# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

The result was:

```
Train Score: 0.80 RMSE
Test Score: 1.44 RMSE
```

Finally, we added an option to save the built model and the resulted weights.

Thanks to this, it's possible to load the trained model and use it in order to predict a specific product.

```
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")


# Make Predictions:
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

loaded_model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])

# predict
obs = numpy.array([[1,4,5,0,0]])
obs_reshaped = numpy.reshape(obs, (obs.shape[0], obs.shape[1], 1))
obs_predict = loaded_model.predict(obs_reshaped, batch_size=batch_size)
loaded_model.reset_states()

print(obs_predict)
```