Instacart Model

Joey Sham

November 15, 2017

1 Introduction

The following document is a detailed description on the methods used to build the model for the Instacart Kaggle Competition, as well as motivation for features.

2 How It Works

2.1 Given Files

There were 7 files given:

- `aisles`
- `departments`
- `order_products__prior`
- `order_products__train`
- `orders`
- `products`
- `sample_submission`

The `orders` file lists every user, and every order each user has made. There are detailed information such as the day-of-the-week the order was placed, and the sequence of the orders. This allows insight on which products were ordered recently, and which were ordered some time ago.

The files `order_products__prior` and `order_products__train` are the exact same format, as they are from the same set of data that has been divided up to *prior* and *train*. The `order_products__*` files has detailed information on each order, including what products were ordered in each order, the order in which they were placed into the cart, and a binary value on whether each product has been ordered previously. These files can be joined with the `orders` file, as `orders` has column `eval_set` with values *prior* or *train* to indicate how the join should happen.

In this analysis, we will not be using `aisles` and `departments`.

To get all of the files, a new method called **readFiles** from the file `pandabase.py` was created to read all the files from the **/data** folder. This **/data** folder was not included in the repository due to memory limits.

2.2 Initial Analysis

The files are arranged in a very tabular manner, with strong similarity to relational database tables. That means if further analysis is requirement, it may be wise to load the information into a SQL database. Due to time constrant, the files were just loaded in memory into **DataFrames** via **pandas** in **python3**.

Even after loading the data, it is obviously that the given fields are not enough to create a model. Thus, feature engineering is required which involves creating new columns in the dataframe based on existing data. Features are divided into two sections: user-based features, and product-based features.

2.3 Feature Engineering

Before any new columns are created, a new DataFrame called `orders_details__prior` was created by joining/merging `order_products__prior` and `orders` on `order_id`. This dataframe is a detailed view of all users and all their orders, with information on which products occured in which order.

A new feature called `no_of_times_user_bought_item` is created in `orders_details__prior` that counts the total number of times a user has ordered that specific item.

2.3.1 User-based Features

The following is a list of each feature generated based on user behaviour, with a description of the meaning of each feature to each user. A SQL interpretation of this is *GROUP BY user_id*.

total_orders
The total number of orders each user has made. Instead of counting, since `order_number` exists, this is the equivalent to the **MAX(**`order_number`**)**

total_days_between_orders
Sums up the number of days between orders, to get a number representing the total number of days between orders for each user

avg_days_between_orders
Takes the mean of number of days between orders, to understand on average how long it takes for each user to make a new order

reorder_ratio_user
For each user, the `reorder_ratio_user` is calculated with

$$\begin{aligned} &\frac{number\ of\ reorders\ in\ orders}{number\ of\ products\ ordered\ that's\ not\ in\ the\ first\ order} \\ &= \frac{count\ all\ orders\ where\ reordered\ == 1}{count\ all\ where\ order\_number > 1} \end{aligned} \tag{1}$$

total_products
Counts the number of products each user has purchased. This is the equivalent to a **COUNT()** in SQL

distinct_products
Counts the number of distinct/unique products each user has purchased.

This is the equivalent to a **COUNT(DISTINCT())** in SQL

average_no_items_per_order
For each user, the `average_no_items_per_order` is calculated by

$$\frac{total\_products}{total\_orders} \tag{2}$$

which calculates the average number of items in the cart when the user orders

2.3.2 Product-based Features

The following is a list of each feature generated based on products, with a description of the meaning of each feature to each product. A SQL interpretation of this is *GROUP BY product_id*.

no_purchased
The number of times this product has been purchased by users. Note that this count is non-unique, but rather it is the equivalent to a **COUNT()** in SQL

no_reordered
The number of times this order has been reordered. Since `reordered` is a binary of 0 or 1, this sums the column to get the count.

no_bought_first_time
The number of times a product is bought the first time by a user. An interpretation of this is the number unique users has bought this item. This counts the number of times `no_of_times_user_bought_item` == 1

no_bought_second_time
The number of times a product is bought the second time by a user. An interpretation of this is the number of unique users that has bought this item a second time. This counts the number of times `no_of_times_user_bought_item` == 2

reorder_prob

This describes the probability that the product will be reordered for a second purchase after a user has bought it the first time. This is calculated by

$$\begin{aligned} &\frac{number\ of\ times\ this\ item\ has\ been\ bought\ a\ second\ time}{number\ of\ times\ this\ item\ has\ been\ bought\ the\ first\ time} \\ =&\frac{no\_bought\_first\_time}{no\_bought\_second\_time} \end{aligned} \tag{3}$$

reorder_ratio_prod
This is the ratio of how many reorders for each item, compared to number of orders. This is calculated by This describes the probability that the product will be reordered for a second purchase after a user has bought it the first time. This is calculated by

$$\begin{aligned} &\frac{number\ of\ reorders}{number\ of\ purchases} \\ =&\frac{no\_reordered}{no\_purchased} \end{aligned} \tag{4}$$

avg_no_times_ordered
This is the average number of times an item is ordered. This is calculated by

$$\begin{aligned} &\frac{no\_purchased}{no\_bought\_first\_time} \\ = 1+&\frac{no\_reordered}{no\_bought\_first\_time} \end{aligned} \tag{5}$$

2.3.3 User-based and Product-based Features

The following is a list of each feature generated based on both users and products, with a description of the meaning of each feature to each user and each product. A SQL interpretation of this is $GROUP\ BY\ user\_id, product\_id$.

no_of_orders
This shows the number of times each user has ordered each product

order_number_of_first_purchase

For each user, this displays which order number in their history that they first ordered an item. For example, user with id 1 ordered product 10326 the first time during user 1's 5th order. This can be calcaulted by taking the MIN of `order_number`.

order_number_of_last_purchase
For each user, this displays which order number in their history that they last ordered an item. For example, user with id 1 ordered product 10326 the last time during user 1's 5th order. As this is the same user from the example above, it can seen that he/she has only ordered this item once. This can be calcaulted by taking the MAX of `order_number`.

avg_cart_order_number
For each user, this describes the average order the product is added to cart.

order_rate
This describes how often the user orders this product. This is calculated by

$$\frac{no\_of\_orders}{total\_orders} \tag{6}$$

no_of_orders_since_last_purchase
This describes how many order it has been since the user has last ordered this specific product. This is calculated by

$$total\_orders - order\_number\_of\_last\_purchase \tag{7}$$

order_rate_since_first_purchase
This describes how often this product is ordered since the first purchase of this specific product. This is calculated by

$$\frac{no\_of\_orders}{total\_orders - order\_number\_of\_first\_purchase + 1} \tag{8}$$

2.4 Model

Due to the nature of the data, XGBoost was selected to be used to train and generate the model. This falls inline with Instacart's data science team, who also admits to currently using XGBoost.

Before training occurs, the data has to be split up. By merging the various dataframes, there exists one dataframe with all relevant information and features that was described in section 2.3. This dataframe also inherited the `eval_set` column from `orders` file that only has two values: *train* and *test*. From hereon, training data refers to the dataset with $eval\_set = train$ and testing data refers to the dataset with $eval\_set = test$.

2.4.1 Training

The idea for training this model is the same as training any other model. Given x, y, there exist a process to continuous update A such that $y = Ax$, where x is the input and y is the output. In this scenario, the output is the `reordered` column, where it has a binary result of 0 or 1. The training input data should only contain relevant columns as to eliminate noise and not confuse the learning method. Therefore, columns such as `eval_set`, `user_id`, `product_id`, and `order_id` are dropped since their values do not directly contribute to the binary result of `reordered`.

At this point, there are two dataframes: `X_train` for training data x, and `y_train` for training `reordered` data y. These two dataframes are fed into the XGBoost training object, with different hyperparameters. Selection of hyperparameters is sometimes more of an art than a science, and can be time consuming since each epoch takes quite a bit of time. The most important part is to know the objective is "reg:logistic" as this is a classification problem. It is common to use logloss as an evaluation metric, as rmse and mae can incurr L1 and L2 errors.

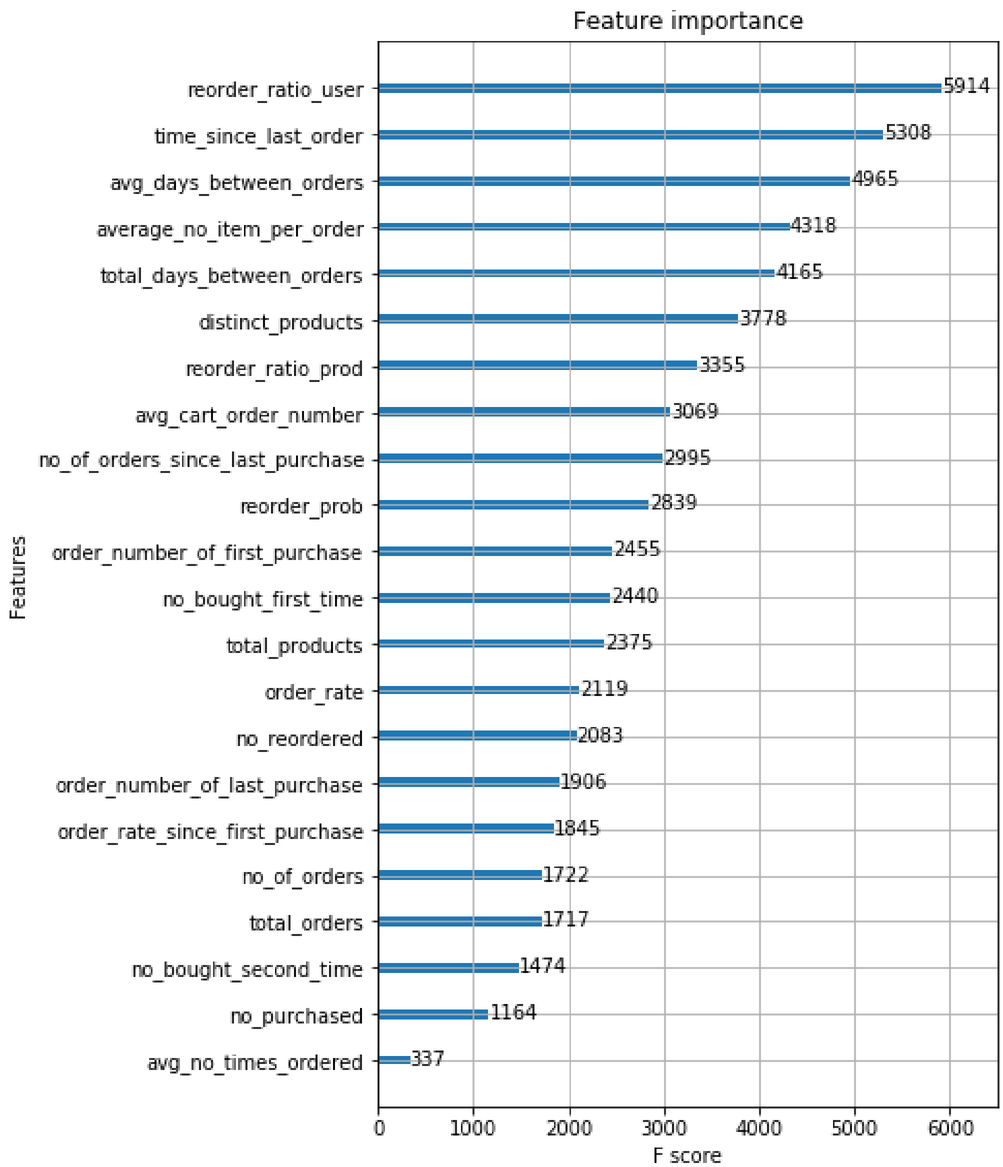

Figure 1: XGBoost Training Feature Importance

After training, it is often beneficial to view which features contribute to the training model the most, especially when there is an increasing number of features. Figure (1) above shows a graph comparing the features described above, ordered from highest attribution to the model to the least.

2.4.2 Predicting

Similar to creating a dataframe for training, the test input data has to drop irrelevant columns. This include columns such as `eval_set`, `user_id`, `product_id`, `reordered`, and `order_id`. `reordered` is dropped in the testing dataframe because this is the desired output, and should not be a part of the input. The resulting dataframe can be input into the XGBoost trained model.

From the testing data, all predicted score above 0.21 are considered to be reordered, and below is not. The results are aggregated and appended to the `sample_submission` file. The dataframe is aggregated in such a way that for every `order_id`, `products` column has the `product_id` where `reordered` was calculated to be 1. For example, the dataframe would be of the form

order_id	products
17	13107 21463
34	47792

3 Result

While there is some variability in the submissions, the submission score is **0.3812513**.

4 Future Improvements

There are two major ways to improve the model. The first is to create new features, and the second is to tune the hyperparameters. There are a lot of literature out there that describes how better to tune hyperparameters, so it will not be discussed here. Creation of new features would be an interesting task, and the feature importance plot can be used to determine how well various engineered features fare. One of these possible features could be one that integrates recency, such that if products were heavily ordered in orders 1-5, but not ordered in orders 6-9, then it can help determine such a product would be less likely to be reordered.

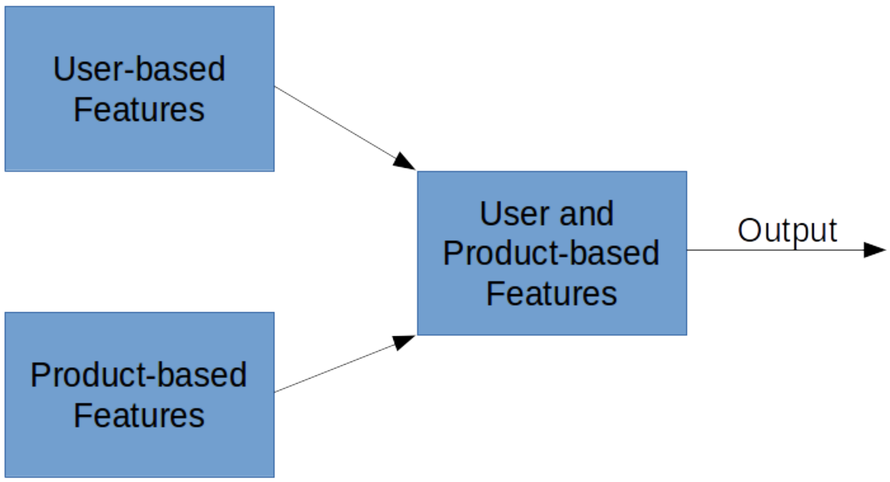

Figure 2: Future Model Design

As described above, the features are divided into user-based (see Section 2.3.1) and product-based (see Section 2.3.2). A possible next step could be to divide up the features into two separate dataframes, each fed into its own model, and combine the results. Then there would be a user-specific model, and a product-specific model, and both would merge into a user-product-model with user-product features (see Section 2.3.3). An example of that can be seen in the figure 2 above.