Machine Learning Engineer Nanodegree
Capstone Project

Instacart Market Basket Analysis

Lutfur Khundkar
September 20, 2017

I. Definition

Online grocery ordering is on the verge of going mainstream. As large brands like Target, Albertsons, Costco and Walmart expand and invest in technologies to retain and grow market share in the online retail segment, they are also increasing their reliance on recommender systems to improve the end-user experience. In this mix, Instacart, a grocery ordering and delivery service operating in several large metropolitan areas, partners with local grocery and warehouse stores to offer users the convenience of online shopping from these partner stores combined with deliveries that fit their schedules. Their aim is "to make it easy to fill your refrigerator and pantry with your personal favorites and staples when you need them".[1]

A ML system that can successfully predict which products a user is likely to order the next time they shop online offers several advantages to an online retailer or a service like Instacart:

a) Balance supply and demand by managing partnerships with the local brick and mortar stores where the merchandise will be picked up by the shopper (which could let the service offer more competitive pricing on items).
b) Gain repeat customers as a result of a positive shopping experience promoted by product recommendations that align well with a user's preferences.
c) Increased sales and more consistent utilization as a result of trust built between the user and the recommender through effective recommendations.
d) Gain new customers and momentum from a social media buzz produced as a result of a good user experience.

Problem Statement

In this competition, Instacart "is challenging the Kaggle community to use anonymized data on customer orders over time to predict which previously purchased products will be in a user's next order".[1]

This problem is a binary classification one at its core. For each order in the test set, we are required to predict if the user's shopping cart will include a product from the universe of the list of products. The F1 score[2, 3] (the harmonic mean of precision and recall) will be the single-valued performance metric used to gauge the success of the proposed solution.

Strategy

The dataset) provides the history of a large numbers of users and products. We have direct information on the day of the week, time of day (binned by hour) an order was placed, and the time (number of days) between orders by each user. On the product side, we have two variables – the department and aisle. Clearly, these few variables alone are not enough to allow us to predict the behavior of 75000 customers (users)!

We will add engineered features to tease out patterns in a user's purchase history that will allow us to identify relationships between users and products. These features will be crafted from statistics of the user's buying patterns. Instead of using the full product list (50,000) we will only consider a product that a user had previously purchased in our predictions.

Resource Limitations

The size of the available data for users' order history (32M user-product rows) and the training data set (3M+ rows) will be computationally challenging. We will use fast implementations of decision trees such as random-forests[4] and boosted decision tree methods[5] and report results from our best attempts to solve this problem.

Metrics

The F1 score (a combination of precision and sensitivity or recall) is commonly used for evaluating classification problems.

$$F1 = \frac{pr}{(p+r)}$$

where *p* is the precision (true positives)/(true positives + false positives) and *r* is the recall value (true positives)/(true positives + false negatives). It provides a good balance in scenarios where the class distribution is highly skewed, i.e., where one class dominates the population. We expect that our data will be skewed heavily towards the negative class (not included in the cart) as we are going to include the entire list of products a user ever purchased which could be hundreds of items whereas an average order includes about a dozen products.

Since this project continued beyond the end of the competition, there will be no opportunity to have the final model from this project to be scored. We will therefore set aside 25% of the orders in the training data and use this as the test data for our model. To compare the results from our test data to the ones provided for the competition, we will present the F1 score for the benchmark models on our test set along with the submission scores from three submissions I made before the competition ended.

II. Analysis

Data Exploration

The dataset made available for the competition is curated and released by Instacart from actual customer data, suitably anonymized. This (minus the test set identified exclusively for the competition) has also been released as a public open source dataset.[6] The dataset consists of five relational tables released as standard csv files:

a) ***departments*** – provides a high-level categorization of the products that Instacart offers. There are 21 unique entries in this table. The *department_id* field is sequential with no gaps. One department (21) is labelled "missing". This is a factor (categorical variable).

	department_id	department
0	19	snacks
1	13	pantry
2	7	beverages
3	1	frozen
4	11	personal care

b) ***aisles*** – describes a section in a typical brick-and-mortar store where the product would be found. This corresponds to a finer categorization of the products. There are 134 entries in this table, also sequential with no gaps. One aisle (100) is labelled "missing". This is also a categorical variable.

	aisle_id	aisle
0	1	prepared soups salads
1	2	specialty cheeses
2	3	energy granola bars
3	4	instant foods
4	5	marinades meat preparation

c) ***products*** – defines a total of ~50,000 items and their association with a department and an aisle. Each product has a 1..1 relation with a department as well as an aisle. There are approx. 50,000 products. They are not distributed evenly across all aisles. The mean count of items per aisle is higher than the 50th percentile value, indicating that the distribution has a tail towards higher number of items. The table also includes a *product_name* field that identifies the product in familiar terms, e.g, bananas, Danish Butter Cookies, and Naturally Sweet Plantain Chips. The enbedded table here shows that the distribution of products in different aisles is fairly broad and not a normal distriution as the minimum (12) is less than 1.5 standard deviations (267.0) below the

	department_id	aisle_id	# items
count	134.000000	134.000000	134.000000
mean	11.373134	67.500000	370.805970
std	5.726435	38.826537	267.010165
min	1.000000	1.000000	12.000000
25%	7.000000	34.250000	179.750000
50%	12.000000	67.500000	305.500000
75%	16.000000	100.750000	497.500000
max	21.000000	134.000000	1258.000000

mean (370.8) while the maximum is almost 3 standard deviations above the mean

d) ***orders*** – a table of ≈ 3.3 million orders (*order_id*) linked to approx. 63,000 distinct users (*user_id*). The table has been curated to include only users with between 4 and 100 orders. Each order is also linked to three additional pieces of information – the day of the week (*order_dow*), time of day binned by hour (*order_hour_of_day*), and the number of days since a user's prior order (*days_since_prior_order*). The first order for a user has a missing value in the *days_since_prior_order* column. Interestingly, the maximum value for *days_since_prior_order* is 30, which may be an artificial cap.

	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_order
0	2539329	1	0	1	2	8	NaN
1	2398795	1	0	2	3	7	15.0
2	473747	1	0	3	3	12	21.0
3	2254736	1	0	4	4	7	29.0
4	431534	1	0	5	4	15	28.0

a) ***order_products*** – this table identifies the products included in a particular order. In addition, each product and order combination also includes the rank of placement in the basket and whether the product was included in a prior order for the same user.

	order_id	product_id	add_to_cart_order	reordered
index				
0	2	33120	1	1
1	2	28985	2	1
2	2	9327	3	0
3	2	45918	4	1
4	2	30035	5	0

There are 49,677 distinct products and 3,214,874 orders in this table, so each product is ordered 64.7 times (average) over the period this data was collected. Approximately 59.0% of these were reorders.

The *orders* and *order_products* table are partitioned into three sets

– a prior set, a training set, and a test set (75k orders, 1 per customer). The prior orders set contains ≈ 3.2 M orders for ~206 K unique user_ids (customers), with an average of 10 product items per order. The training set contains ≈ 130k orders, 1 order per customer, and the test set contains 75K orders, also 1 for each customer.

	eval_set	Orders
0	prior	3214874
1	train	131209
2	test	75000

The *order_products* test set for which we are expected to provide predictions can be inferred from the *order_products* table using the *eval_set* column value.

Moved down [1]: ¶
order_id

Moved (insertion) [1]

Formatted Table

The set of unique users in the test set has no overlap with the set of unique users in the train set. Also, there are no repeat orders from the same customer (user) in either the test or the training set.

Exploratory Visualization

A plot of the distribution of order counts (normalized to 100 for each *eval_set*) matches out intuition – the distribution is fairly uniform, with increased likelihood during weekends (day 0,1, 5 and 6). The trend for heavier order volume on weekends is somewhat more pronounced in the training and test sets.

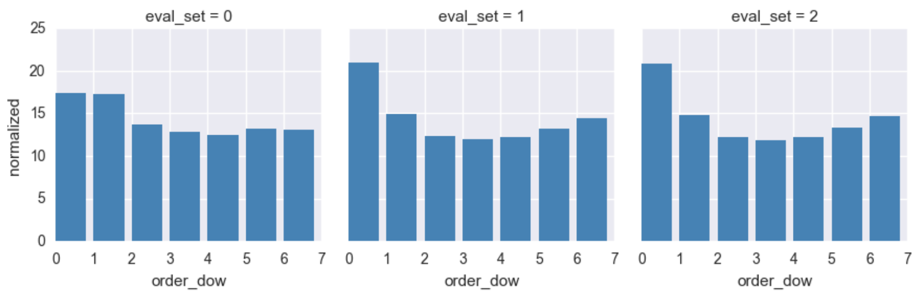

Figure 1. Frequency Distributions of orders vs. day of week

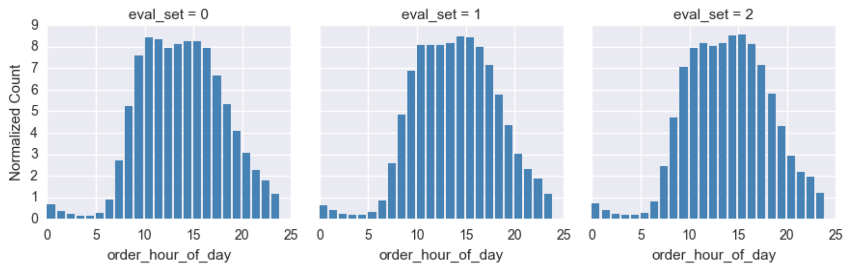

Figure 2. Frequency Distribution of orders vs. hour of day

A plot of the distribution of *days_since_prior_order* (with NA values for the first order for a user being set to 32) reveals a few interesting features. First, the prevalence of the value 30 is much higher in the training and test sets than in the prior orders set and its frequency across the dataset is much higher than 28 or 29, indicating that this value was likely capped at 30. Second, 32 does not appear in the training or test data sets. We will need to preprocess this variable for our analysis.

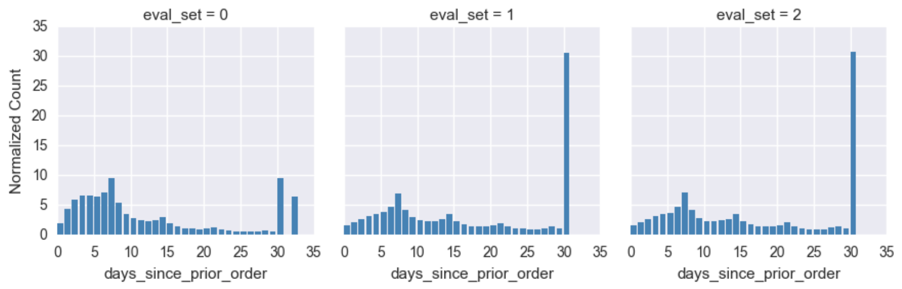

Figure 3. Frequency Distributions of orders vs. number of days between orders. The x-value of 32 represents missing data (first order).

Implementation details for this exploratory analysis are provided in the associated Jupyter notebook *Instacart – Exploration.ipynb*.

Formatted: Font: Italic

Algorithms and Techniques

Algorithms that are commonly applied to binary classification problems include decision tree, random forests, support vector machines and neural networks.[7] For this project, we will use and gradient-boosted decision trees as these are known to be robust with respect to overfitting[8] and more efficient in memory requirements as well as in computational time.[9]

Decision trees[10] are easy to understand, interpret and visualize. They can handle both numeric and categorical variables, require less data preparation and are faster to train. The fact that decision trees can capture very complex relationships in the data also make them prone to overfitting or high variance and the resulting model may not generalize well to data not in the original training set. They can be sensitive to input data and can lead to a biased model if the underlying data is dominated by a subset of the classes that the model is being trained to predict.

Deleted: T

Deleted: and

Deleted: are therefore inherently prone to

Deleted: .

Ensemble methods such as random forests or gradient-boosted trees uses averages over many trees with low bias and thereby reduce variance[11] and help mitigate the disadvantages of simple decision trees. Random-forests work by fitting a decision tree to a large number of subsets of the input data, each subset being a random sample drawn with replacement. In the process of training each tree, the algorithm also introduces "feature bagging" which chooses a subset of the available features (typically the square root of the total number for classification problems) when the tree needs to be split. For predicting the class of an unseen input, the predicted class is determined by a majority vote of all the learners. As a result, the model can handle very large feature sets without becoming susceptible to overfitting. As an useful side-effect, the training processes can provide data on the relative importance of all the input features, sometimes ignoring ones that are not statistically relevant for the model.

Deleted: .

Deleted: are relatively easy to set up and provides good results with little tuning.

Gradient boosted trees offer another way to generate a model that makes predictions based on an ensemble of decision tree learners. The model in this case is a sequence of trees in which each tree is a weak learner over a version of the input space that has been weighted to emphasize the input examples where the previous tree made an inaccurate classification. The model is a weighted sum of the ensemble of learners where the weights are selected to minimize a differentiable loss function, commonly mean squared error or logloss function. Gradient boosted trees retain the advantages of decision trees and add stronger predictability.

We chose the ensemble of tree-based weak learners as our solution because these algorithms are insensitive to outliers and redundant features, fast and generally have good predictive quality, and gradient-boosted trees specifically because they will usually give better results than random-forests with careful tuning.

Deleted: are robust with respect to overfitting and

Two variants of gradient-boosted models have been used with success in Kaggle competitions[12] -- xgboost[13] and LightGBM.[14] Both of these provide efficient implementations of gradient boosted models with LightGBM being known to be significantly faster and more memory efficient. We chose to use LightGBM for our analysis.[15]

Benchmarks

The competition provides a sample submission file of 75000 orders where each order is predicted to include the same two products – bananas and baby bananas. The mean F1 score (on 33% of the test set) is 0.0005453.

We will also include a few other intuitive baseline benchmarks that don't involve machine learning:

a) all orders in the test set include the *N* most popular products (based on the prior and train partitions) where *N* is the average number of products in an order also computed from the prior and train sets.
b) A more personalized base model is one where each user in the test set orders all items from their previous order.
c) A third baseline benchmark that we can consider is a combination of the two previous models, where each user orders their top *M* most frequently ordered products, where *M* is the average size of their prior orders.

III. Methodology

Data Preprocessing

The tables provided do not appear to have any missing data with two caveats. In the *orders* table, a user's first order has an empty value for the *days_since_prior_order*. We will impute this, as the average of the mean value of that feature for each user or across all orders if the user has only one order in the dataset. There are approximately 1250 products that are not associated with their proper department or aisle but labelled as *missing* instead. While it would be interesting to attempt to impute these values as well, we chose to leave them as their special category.

A few features, e.g., the aisle and department a product is associated with, the *order_dow* (day of the week the order was placed), and *reordered* (identifying whether a product in a basket was previously purchased by the same user) are factors and will need to set up as either binary or one-hot encoded variables. Obviously, the aisle and department features are categorical variables and will need to be one-hot encoded as well. Two other integer variables from the *orders* table – *order_hour_of_day* and *days_since_prior_order* – may be used as continuous ones in our model.

The biggest challenge lay in engineering additional variables to gather information on user average behavior. The goal was to attempt to capture user's individual purchase patterns on two levels – features tied to a user and features tied to a user-product combination.

Implementation

Missing values and outliers

We imputed the missing *days_since_prior_order* as the average for each user. The dataset included at least four orders from each user so there was no need to go beyond this.

While a value between 1 and 29 represented the true number of days between orders, 30 was over-represented in the data and was the largest value for that feature. We took this as implying that 30 meant 30 or more days. We imputed this outlier value by adding a categorical variable *30plus_days* and substituting a random number between 30 and 50 (to match the overall frequency distribution).

We computed the following features using the historical dataset (with eval_set=0, prior) using mostly pandas. In a couple of situations, the manipulations required custom logic which were too compute-intensive to implement as basic lambda functions. We resorted to using numpy data structures to achieve sufficient speedup to allow us to compute these features in a reasonably short time period.[16]

User-product features

1. *up_cart_rank* – average of *order_place_in_cart* over all orders containing this product.
2. *up_times* - the number of times user purchased this product
3. *up_reord_prob* – given user bought product, probability that they reordered it
4. *up_oprob* – the probability user purchases this product in each order
5. *up_avg_days* – the average number of days between product in order/basket
6. *up_last_reord* – binary variable to indicate whether product was reordered the last time it was in this user's cart
7. *u_aprob* – the number of times user bought a product from this aisle
8. *u_a_reord* – the probability that a product from this aisle was reordered
9. *u_a_pcount* – the average # of products from this aisle purchased by user
10. *u_a_reord* – the conditional probability that products from this aisle were reordered, given that they purchased at least one from this aisle
11. *ua_ocount* – the number of orders with products from this aisle
12. *in_last* – binary variable to indicate whether the product was in the user's last order

User features

1. *uo_count* – the # orders placed by user
2. *ubasket_avg* – the average basket size for user
3. *udays_since* – average number of days since the previous order
4. *u_dow* – the average day of week order was placed
5. *u_tod* – the average time of day (24 hr) of day the order was placed
6. *top-1 - top-10* – the aisle_id of 10 most frequently ordered products aggregated by aisle

The code used to derive these features are documented separately in the attached IPython/Jupyter notebook (*Instacart – Preprocessing*). There are additional notes on how the top-1 through top-10 were calculated in that notebook.

Preparing the training (and test) set

The training set provided by Instacart is a selection from the underlying *orders_products* table and contains only four columns. We prepared this official training set by adding a new column with all values set to 1 to indicate these products were in the orders in the training set. This is our label column (*in_order*). We split this into two sets, one for training and validating our model, the other for estimating out-of-sample errors. Operationally, it was more convenient to establish the split on the orders table using train_test_split, with 25% designated as the test set for the out-of-sample error estimates. Each of these sets needed to be augmented with the engineered features:

1. From the *orders* table (DataFrame), select the training set using *eval_set*==1 (appropriately sampled)
2. Merge with the table of engineered features (very large) on *user_id*. This expands the table from step 1 to include all products purchased for each user
3. Merge the expanded table with the products included in the train (or test) set using a left join. The number of rows in this result should be the same as in the table from step 2.
4. Fill in the missing values in the label column (*in_order*) with zeros.

Once the training data had been augmented, we applied *train_test_split* from scikit-learn with a test fraction of 0.5 to derive the train and validation sets used for our modelling.

Running the models

LightGBM comes with python wrappers (lightgbm v 2.0.5) that look and behave very much like other scikit-learn classifiers. Thus we instantiate a model with the data and a long-list of parameters including common ones like learning rate, tree size and regularization and fit the model to get the optimized classifier. We chose to run our model using *logloss* as the error metric as this had been found to be effective in scenarios with skewed class distributions in the model data.

Prediction also follows the standard scikit-learn patterns and yields a vector of predictions that can then be scored by standard sklearn functions

Refinement

Benchmark models

We calculated the three benchmark models identified in the analysis section and scored them against the competition test set (33%) while the competition was still in progress. They were scored as 0.006 (all baskets filled with the *N* most popular items where *N* is the average basket size), 0.312 (duplicate of each user's last order in the history set) and 0.328 (each basket was filled with the *M* most popular items for that user where *M* is the average basket size for the user) for the three models.

Preliminary model

We used the random-forest (*rf*) algorithm available in the LightGBM package (v2.0.5) as our initial model. The algorithm is invoked by passing `rf' as the value of a hyperparameter *boosting_type*. We used the default parameters except for the number of iterations (*n_estimators=30*), *bagging_fraction=0.9, bagging_frequency=5*, and *feature_fraction=0.9* and which were required by the model. These parameters are some of the many available to reduce overfitting and have default values that are not acceptable when the rf algorithm is selected.

Formatted: Font: Italic

Formatted: Font: Italic

Formatted: Font: Italic

Deleted: *bagging*

Formatted: Font: Italic

Formatted: Font: Italic

Deleted: *ratio*

We explored two other algorithms – *gbdt* (gradient-boosted decision trees) and *dart* (Dropouts meet Multiple Additive Regression Trees) available in LightGBM. These were invoked by passing `gbdt’ and `dart’ respectively as the value of *boosting_type*. Other hyperparameters were set to the same values as for the random-forest model. The DART algorithm has been shown to give better recall rates[17] which could be helpful in our problem with a very skewed class distribution. The following table shows the success of prediction results over the validation set for each algorithm. The threshold for calculating precision and recall values was set to 0.20 in each case.

	Precision	Recall	F1-score	logloss	AUC	Accuracy
rf	0.563	0.298	0.390	0.304	0.799	0.828
gbdt	0.501	0.347	0.410	0.261	0.813	0.859
dart	0.694	0.246	0.363	0.309	0.809	0.762

Of the three algorithms we tried, *gbdt* appeared to be the most promising as it showed a higher recall, comparable precision and a lower value for the loss function. None of the methods had converged in the 30 iterations we specified as the limit.

The next step was to do further refinements of the model. LightGBM provides simple guidelines on parameter tuning.[18] They discuss the interplay between three main parameters – *num_leaves*, *max_depth* and *min_child_samples* – for managing overfitting from tree-depth. They provide additional parameters like bagging and feature fraction to also control overfitting from having too many features. We chose to optimize the learning rate (0.05, 0.0.075, 0.15, 0.25) and the number of leaves (51, 81, 121) using GridSearchCV with n_estimators = 30.

With the best fit parameters learning_rate=0.15, num_leaves=121, we trained a model with n_estimators=200 and an early_stopping_rounds=10 (to terminate a cycle if it didn’t result in reducing the cost function within that many iterations) with 5-fold cross-validation as our final model.

The lightGBM module was relatively easy to setup and use following the examples available in the official distribution repository on GiitHub. The core training class (LGBMClasifier) has settings that allow the training to run on all available cores in a multi-core machine, over cores in different machines, and also on gpus supporting OpenCL. The default settings run on all available cores on a single machine and ignores the gpu, which worked well on the hardware I was training the model on. One thing that required a bit of hacking was that there was some mismatch between the parameters used by the sklearn wrappers (e.g., GridSearchCV) and the parameters used by LGBMClassifier that did the training. For example, GridSearchCV recognized n_estimators as the number of iterations, but the LGBMClassifier required this to be num_boost_round. This meant that to train the best classifier returned by GridSearchCV for 5-fold cross-validation, we had to remap these parameters.

Formatted: Font: Italic

Deleted: min_data_in_leaves

Formatted: Font: Italic

Formatted: Font: Italic

Deleted: In addition,

Deleted: t

Deleted: min_child,

Deleted: frequency

Deleted: number of boost cycles (iterations)

Deleted: first with the default

Deleted:

Deleted: learning

Deleted: rate of 0.1.

Finally, since the model is set up to return a probability of the product being purchased through our choice of *binary_logloss* as the metric, we varied the threshold for classification as "reordered" and picked the threshold (0.20) that gave the best f1-score.

Deleted: S

Formatted: Font: Italic

IV. Results

Model Evaluation and Validation

The features importances show a distribution over many features (see Figure 4 below). Not surprisingly, *up_last_order* (product was present in the previous order) and *up_reord* (probability of a user reordering the product) and are among the top important features. The *aisle_id* and several of the *top-k* features are also weighted significantly in the final model. The prominence of the *up_last_order* makes sense given how well benchmark 2 and 3 performed. It is also reassuring that the aisle-id and the product associations we crafted through the top-K aisle features turned out to be significant – intuitively, we would expect certain products to be purchased together and the grouping by aisle is recognizing the structure among the products that is present in the data and the real world.

The process of cross validation is known to reduce variance and out-of-sample error. We tested the model against both the validation set and the out-of-sample (test) set we set aside. Our model scores for the validation set and the test set are comparable and indicates that this model does generalize well to unseen data. The scores on the test set in particular are worth noting as there is no user overlap between the test and the training set while there is likely to be some overlap between the dataset the model was trained on and the validation set.

To assess whether the model is robust enough, we ran train and evaluation using two other training sets – one with a 40:60 spilt and one with the 50:50 split but a different random seed. In all cases the results were similar:

Model	Set	Precision	Recall	F1 score	AUC
Final model	validation	0.5051	0.3654	0.424	0.822
50:50 split, different random seed	validation	0.5186	0.3742	0.435	0.868
40:60 split	validation	0.518	0.375	0.435	0.823
Final model	OOS Test	0.4972	0.3587	0.417	0.818
50:50 split, different seed	OOS Test	0.4982	0.3593	0.418	0.863
40:60 split	OOS Test	0.4959	0.3589	0.416	0.864

We therefore conclude that our final model is robust with respect to small perturbations in the training set.

Justification

Model	Precision	Recall	F1-score	OOS (*Kaggle*)
Benchmark 1 – top N products for all, N=avg size of all orders	*0.0950*	*0.2557*		*0.1385 (0.0639)*
Benchmark 2 – same as user's previous order	*1.0000*	*0.0982*		*0.1788 (0.3118)*
Benchmark 3 – same as user's previous order, trimmed to size of user's average order	*0.4427*	*0.3254*		*0.3751 (0.3284)*
GBDT – on validation set	*0.5051*	*0.3654*	*0.4240*	
GBDT – on OOS Test set	0.4972	0.3587	0.4172	

Table 1. Summary of Results

The results of our final model show better f1-scores than any of the benchmark models. The model shows better precision and recall scores than benchmark 3 which had the highest f1-score. We therefore conclude that our model does perform better than the best baseline model. It is interesting though that the model doesn't do very much better than picking the most commonly ordered items from the user's basket. This suggests that users tend to do a lot of repeat buys and this is reflected in the feature importances – *up_last_order* (which marks whether the product was included in the user's last order) is one of the more dominant features in our model.

The model's f1-score of 0.424 is relatively low, indicating that while it has good predictive power, there may be significant gains to be made with a different set of engineered features or a different algorithm as discussed in the improvement section below.

V. Conclusion

Free-Form Visualization

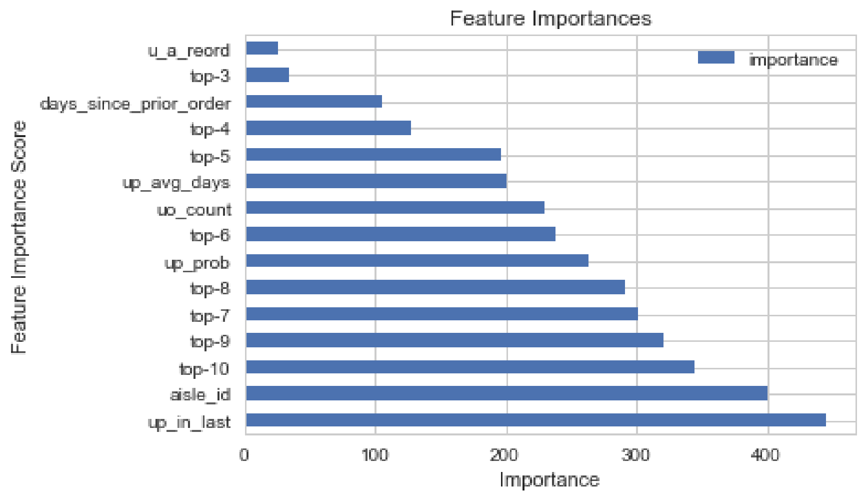

Figure 4. Plot of feature importances sorted by importance

An interesting observation from the sorted list of feature-importances showed that the aisles a user tended to shop the most frequently (*top-1* through *top-3*) were less important to the model than others where the user shopped less frequently (*top-5* through *top-10*). This is a bit counterintuitive and may simply be a reflection of a simple principle like all baskets have bread and milk so those aisles have limited use for prediction.

It is interesting that many of the statistics we calculated (with great effort in some cases) ended up being essentially ignored by the model. The *up_tod* feature is worth pointing out as it is being ignored in our model, but the blog announcing this dataset[6] shows a graphic indicating that some products are more frequently purchased at certain times of the day. Our analysis shows that the time of day has little effect on whether a product will be included in the shopping cart.

Reflection

In this project, we built a set of approximately 30 new features from statistics derived from user historical data. We crafted a sub-universe of products a user might consider placing in their cart based exclusively on their past habits and trained a gradient-boosted tree model to predict which of these a user was likely to purchase. Our final model was acceptable but had significant room for improvement.

The process of calculating these features using pandas involved many complex grouping and aggregation functions. This proved challenging on multiple levels. My experience to date had not prepared me for the complexity of the manipulations that were required for this project and I also discovered through this experience that lambda functions and the DataFrame *apply* function is very inefficient. Given the size of this dataset, I had to devise alternate ways of looping through numpy matrices to get through some of the computations in reasonable time.

The underlying motivation for crafting the set of engineered features was to create a multi-faceted profile of each user that our learning algorithms could use to detect patterns of similar behavior between users and use this collective information to make better (robust) predictions about a customer's future purchases. In a way this is a very primitive recommender system built on user-user correlations.

Improvement

There were three classes of challenges we faced in this project:

1. Working with large dataframes in pandas with complex aggregations can be painfully slow
2. Working with a large dataset and in-memory learning algorithms limited our choices of techniques.
3. Determining a useful set of features to represent the variation in user purchase patterns

As mentioned in the methods sections, we had to develop workarounds for some aggregations by resorting to looping through matrices. We found documentation of ways to speed up pandas aggregation functions by using Cython to compile python code but could not successfully implement this in time. Having this capability would have given me time to consider additional features for this problem.

The training data provided by Instacart had 1.3M rows (user-product combinations) but our solution using statistics for features meant that we were limited by each user's own history. It also meant that our training data grew almost 10-fold larger than the official training set as we had to add in many additional rows corresponding to products in the user's purchase history but not in the training data itself. This made the memory footprint even more challenging and meant we were not able to attempt more sophisticated classification algorithms like support vector machines or use PCA to reduce dimensionality. Since we had to anonymize the user and product information (i.e., leave out *user_id* and *product_id* as features), using some methods of clustering, e.g., KNN to group users and products into clusters might have boosted the predictive power of our model. We did use *aisle_id* as a cluster of products, but looking at the data in different dimensions may have provided new insight into how users buy clusters of products as opposed to how stores arrange to sell them.

We treated each order as an individual order and although we aggregated information across orders for user, we left out any information embedded in the sequence of orders for each user. This problem would have been better addressed as a collaborative filtering problem. Implementing restricted Boltzmann machines to solve this as a collaborative filtering problem[19] using a deep-learning library such as *tensorflow* holds potential for yielding a better model than the one we crafted in this project.

References

[1] Kaggle Competitions – https://www.kaggle.com/c/instacart-market-basket-analysis. This project is based on a competition on Kaggle that ended August 14, 2017.
[2] Scikit learn – classification metrics - http://scikit-learn.org/stable/modules/model-evaluation.html.
[3] "F1 score " – https://en.wikipedia.org/wiki/F1_score
[4] Random forests:"Ensemble methods" accessed from http://scikit-learn.org/stable/modules/ensemble.html
[5] https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/
[6] "3 Million Instacart Orders, Open Sourced", Accessed from https://tech.instacart.com/3-million-instacart-orders-open-sourced-d40d29ead6f2 on August 7, 2017
[7] https://en.wikipedia.org/wiki/Binary_classification
[8] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
[9] Nikam S. S. A Comparative Study of Classification Techniques in Data Mining Algorithms. Orient. J. Comp. Sci. and Technol;8(1). Available from: http://www.computerscijournal.org/?p=1592
[10] https:// http://scikit-learn.org/stable/modules/tree.html
[11] Introduction to Statistical Learning accessed from http://www-bcf.usc.edu/~gareth/ISL/
[12] http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/
[13] xgboost - https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

[14] https://github.com/Microsoft/LightGBM
[15] https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/
[16] IPython notebook detailing feature engineering code – *Instacart – Pre-processing.ipynb*
[17] ``DART: Dropouts meet Multiple Additive Regression Trees'' accessed from https://proceedings.mlr.press/v38/korlakaivinayak15.pdf September 2017.

[18] https://github.com/Microsoft/LightGBM/blob/master/docs/Parameters-tuning.md
[19] https://www.coursera.org/learn/neural-networks/lecture/r4R4E/rbms-for-collaborative-filtering-8-mins

Formatted: Font: Verdana, 11 pt

Formatted: Font: Verdana, 11 pt