Capstone Project Thesis
Instacart Market Basket Competition

Introduction:

Walmart's acquisition of Jet.com for $1.3 billion and Amazon's acquisition of Whole Foods for $13.7 billion are symptomatic of an anxiety surrounding the future of retail. No superlatives have been spared to describe the trend, despite eCommerce accounting for no more than 9% of US retail spend, as of Q1 2017 (according to the Census Bureau). At its very core, this shift falls under the domain of Supply Chain Management; products and services are experiencing a wildly different lifecycle between the point of production and the point of consumption.

The result is that retailers with lean, adaptive supply chains will win. The implicit imperative behind retail's changing landscape is the need to predict and adapt to consumer habits, which makes for some supremely interesting Supply Chain questions.

Project Selection:

I had six months to prepare for my Master's in Supply Chain Management (SCM) at MIT, and my goal was to solidify my skills in Data Science, focused on transforming data into concrete, actionable strategies. I chose Springboard based on the mentorship program, which turned out to be a great decision.

With a background in eCommerce and retail and a solid foundation in JavaScript, the Kaggle competition presented an ideal context for channeling rigorous data science into immediately impactful decisions. The top three submissions win $25,000 and the code would be on the fast-track for pushing into production. Being that my goal was not to become a Data Scientist, but to develop fluency in the process and implementation of Data Science within my own domain of eCommerce SCM, the Kaggle competition was perfect in that it offered an extremely active community as well as daily AMA with Instacart's Data Science team. While Kaggle competitions are "Data Science Light", in the sense that you are given clean and well-wrangled data, it was perfect for me given my professional aspirations, which inevitably involve managing Data Science teams and using insights from big data to propel business strategy.

Instacart is an online grocery and delivery app. Instacart's shopping experience is driven by data scientists that curate the products that you see. The current $20 billion Americans spend at online groceries represents a meager 2.5% of the estimated $800 billion US grocery market, where perishable goods make predicting consumer buying habits a matter of life or death (figuratively, except for the produce). Demand Forecasting being a cornerstone of Supply Chain Management (SCM), I see this challenge as an opportunity to explore the domains that have driven my career thus far – eCommerce and SCM – through the lens of data science.

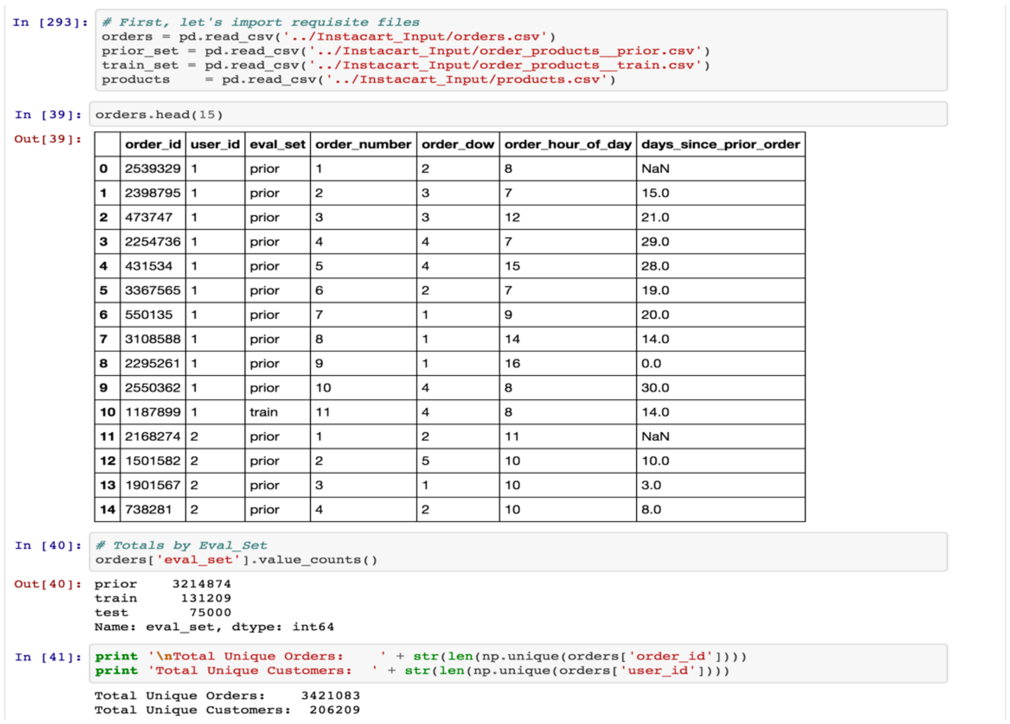
Instacart Kaggle Competition:

Goal: Predict the products that a customer will buy next (i.e. the specific products a user will include in their next order)

For the test set of 75,000 orders, the submission file consists of space-separated values of product IDs that represent predicted products corresponding to test orders.

```
order_id,products
17,39276  234 725
34,39276  37 34234
137,None
...
```

Data: Over 3.4M orders from 206,209 customers.



```
In [293]: # First, let's import requisite files
          orders = pd.read_csv('../Instacart_Input/orders.csv')
          prior_set = pd.read_csv('../Instacart_Input/order_products__prior.csv')
          train_set = pd.read_csv('../Instacart_Input/order_products__train.csv')
          products     = pd.read_csv('../Instacart_Input/products.csv')

In [39]: orders.head(15)
```

Out[39]:

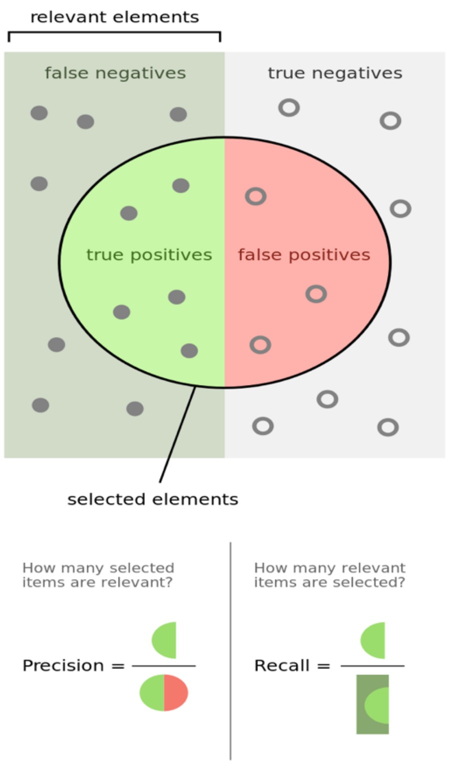
	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_order
0	2539329	1	prior	1	2	8	NaN
1	2398795	1	prior	2	3	7	15.0
2	473747	1	prior	3	3	12	21.0
3	2254736	1	prior	4	4	7	29.0
4	431534	1	prior	5	4	15	28.0
5	3367565	1	prior	6	2	7	19.0
6	550135	1	prior	7	1	9	20.0
7	3108588	1	prior	8	1	14	14.0
8	2295261	1	prior	9	1	16	0.0
9	2550362	1	prior	10	4	8	30.0
10	1187899	1	train	11	4	8	14.0
11	2168274	2	prior	1	2	11	NaN
12	1501582	2	prior	2	5	10	10.0
13	1901567	2	prior	3	1	10	3.0
14	738281	2	prior	4	2	10	8.0

```
In [40]: # Totals by Eval_Set
         orders['eval_set'].value_counts()

Out[40]: prior    3214874
         train     131209
         test       75000
         Name: eval_set, dtype: int64

In [41]: print '\nTotal Unique Orders:     ' + str(len(np.unique(orders['order_id'])))
         print 'Total Unique Customers:   ' + str(len(np.unique(orders['user_id'])))

         Total Unique Orders:     3421083
         Total Unique Customers:   206209
```

Competition website: https://www.kaggle.com/c/instacart-market-basket-analysis
Client: Instacart.

Evaluation Metric: Mean F1 Score.

```
F1 = 2 * (precision * recall) / (precision + recall)
```



F measures are a popular metric for unbalanced datasets, where relevant items are rare[1]. This is the case for this competition, where "relevant" refers to items that occur in the test order, and "irrelevant" refers to all items that do not occur.

F measures are interesting because they encapsulate the trade-off between precision and recall, where precision roughly translates to "exactness" and recall roughly translates to "completeness".

Optimizing F1 scores ended up being a crucial element in predicting product assortment / basket-sizes, so I will delve deeper into this concept later.

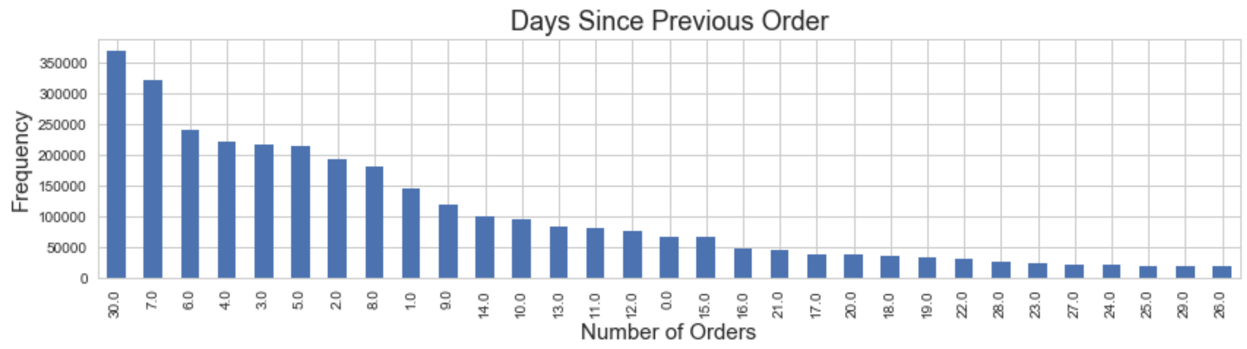
Preliminary Exploration & Initial Findings:

I initially explored the data by importing the six (6) data files into a number of Jupyter iPython notebooks, which I organized by theme. The iPython environment provided an ideal interface for convenient, clean, and visually effective exploration, as well as providing a robust story-telling framework via LaTeX and piecemeal code execution.

The data was relatively clean and well-wrangled, which presented an ideal opportunity to focus on analyzing patterns that I would later build into features for my predictive algorithm. That said, I will mention a caveat that I'll address later on: just because there were no obvious missing values or errors doesn't mean there aren't hidden distortions and misrepresentations. Instacart bundled all orders for which the previous order was greater than 30 days ago into a single bucket = 30. This became obvious once I noticed that 30 was the most frequent

[1] Ye, N., Chai, K., Lee, W., and Chieu, H. Optimizing F-measures: A Tale of Two Approaches. In ICML, 2012.

observation for the variable, and the maximum value.

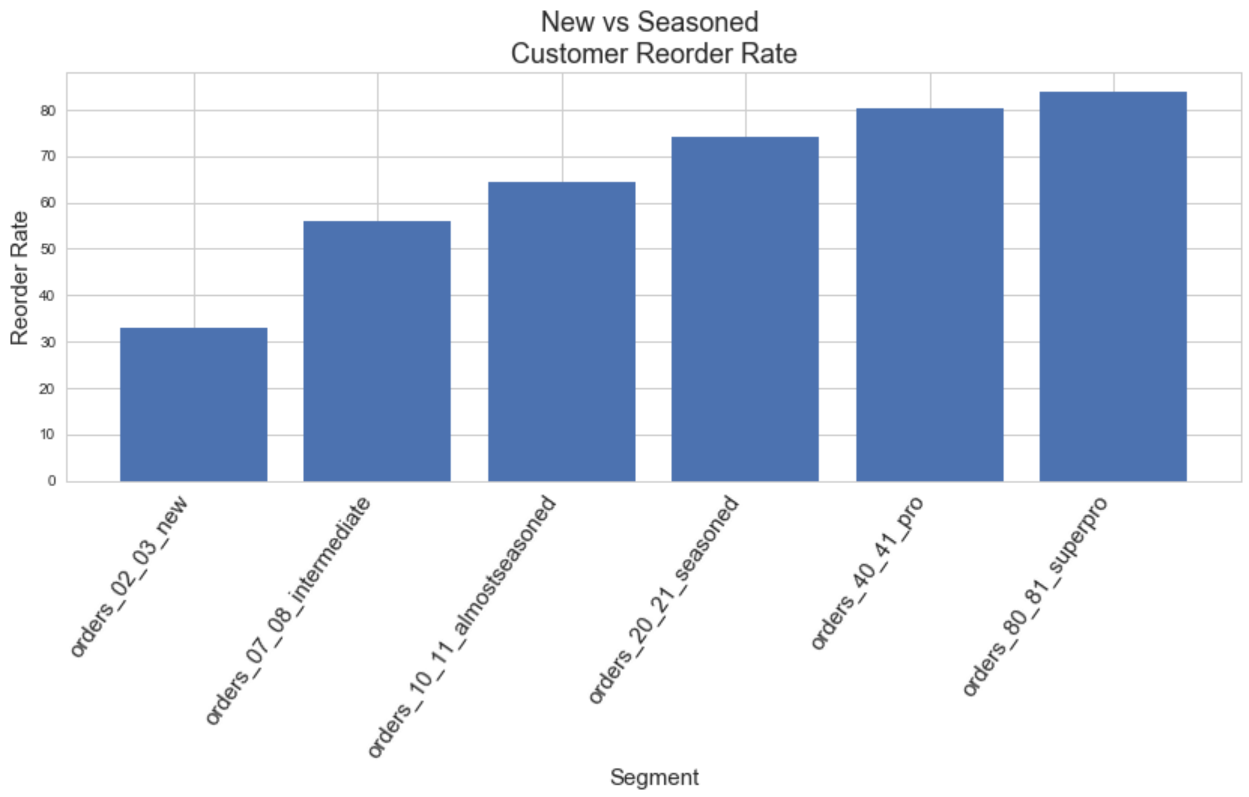


That means that if a customer order 100 years ago, it would list 30 as the days_since_prior_order variable. More on this later, but suffice it to say that the data was homogenous and assumed to be error-free, which is to say clean (for the most part).

Given my personal goal for the Springboard Data Science Immersive Course, this was ideal; I do not necessarily intend to join a data science team within an organization, but will more likely be coming from the perspective of business strategy/operations/supply chain strategy. Thus, while the data wrangling process is a critical tool to have in one's arsenal as a Data Scientist, I anticipated being short on time as I geared up to join MIT's Supply Chain Management program, and this competition suited my goal of focusing primarily on transforming data into actionable insights and production-ready models capable of advancing specific business strategies.

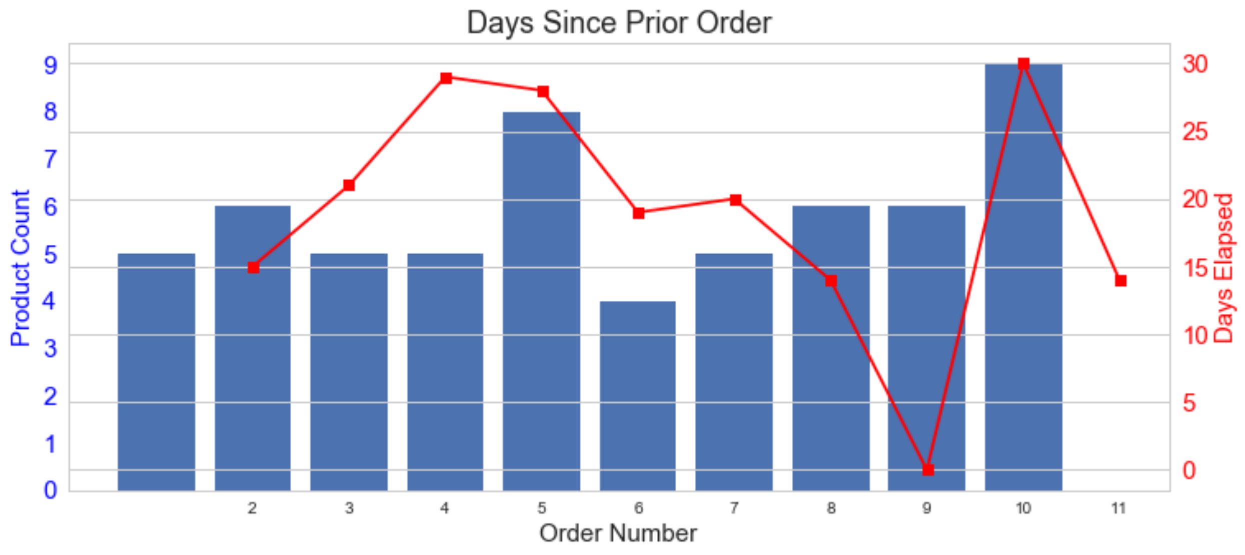
The first step, after cleaning the data, is aggregating it in a meaningful way. That means combining the six separate files into various combinations that are themselves useful for analysis. This meant, at times, combining them into a single file, but more often it meant combining them into convenient smaller files for faster processing. Processing speed is something I came to understand well in a competition where there were approximately 12M (user_id, product_id) tuples, each with 75+ features in my case.

The basic assumptions of predictive algorithms are covered well in the Springboard coursework as well as in a few papers sited at the end of this thesis. In the context of this competition, the most basic assumption is that past order behavior has some bearing on the actual products that a user will buy (assortment) and the size of their order (basket size), among other things. This foundation has its caveats, of course, but you can begin to see patterns like this one:



It also assumes a user's habits, including order timing, frequency, product loyalty, etc., are imperfect but capable indicators that influence the probabilities of ordering specific products in the future. It may also be shown that larger aspects of the shopping experience influence macro-level trends that are customer-agnostic, in the sense that they can have measurable impact on large groups of customers. For example, 24 out of the 25 most popular products late at night are ice-cream.

Concepts like cyclicality start to become evident, as you notice buying habits that occur with some degree of regularity. The following shows an example for user_id 1:



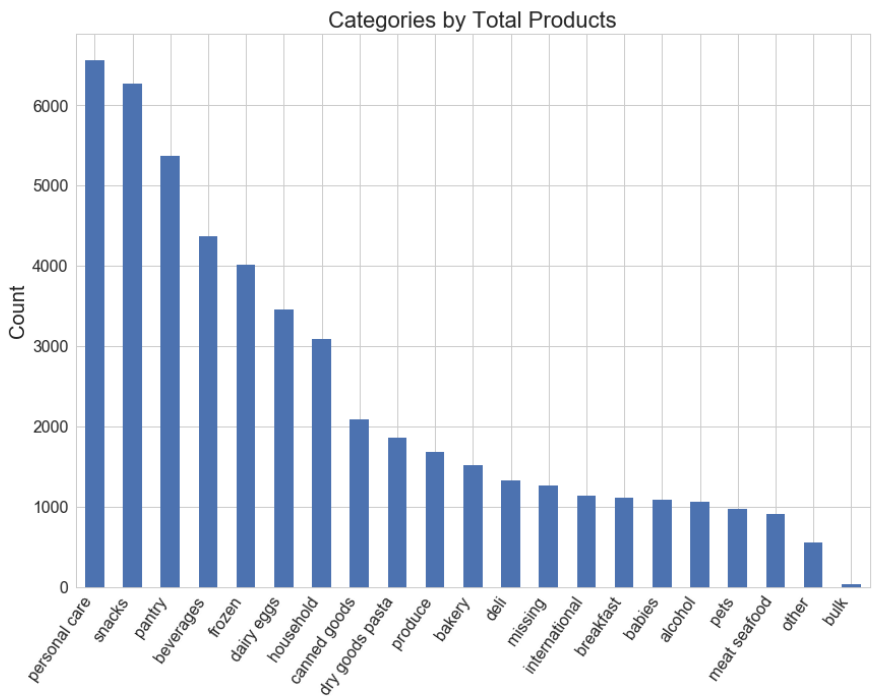
Already you might imagine how difficult it ended up being to predict basket-sizes when variability (in product count from one order to the next) can coexist with predictability (basket-sizes can be derived reliably from every nth order, in this example). This obviously varies from customer to customer, lending to the issue of an unbalanced dataset that I will address throughout this thesis.

Nonetheless, the most surprising aspect of this competition was how crucial basket-sizes are to formulating accurate predictions. Simply put, if you knew the exact number of products for every order in the test set – let's say quantity n – you could easily predict the threshold for the machine-learning algorithm that corresponds to the nth highest prediction. But the reality is that basket size is a moving target, and has critical implications for threshold-setting. If you pick an arbitrary threshold – let's say 20% for customer X – and 20 products have predicted probabilities above 20% despite an actual basket-size of 5 products, then your accuracy will be miserably low simply because you picked a terrible threshold that included 15 irrelevant products. In some ways, basket-size is the single most influential variable for this competition. This was surprising because I began the competition focused on which specific products to include, when the real issue is how many products to include, once you have an ordered list of probabilities associated with each product for a user. I will expand on this later.

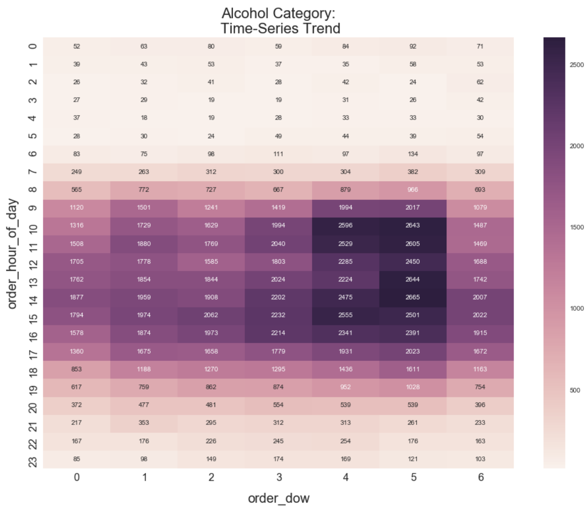
While I used the Python language as a basis for driving my exploration, I relied heavily on numpy for linear algebra and numerical manipulation, pandas for data processing and frameworks, as well as matplotlib and seaborn for visualization. I also made extensive use of several built-in libraries in Python, including os for file-handling, re for regular expressions, datetime and calendar for various time-series manipulations, and Unix from the command line to setup and run my coding environment. I'll address machine-learning algorithms later.

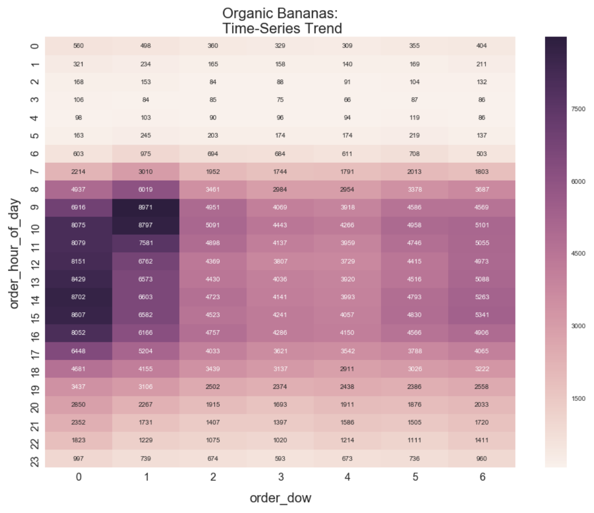
I created three notebooks with my initial findings, listed below. The notebooks provide more of a narrative than I will present here, so please consult the links for more in-depth exploration.

1. <u>Categories – General</u>
    a. This file explores products and product categories, including relationships between what Instacart calls 'departments' and 'aisles', which correspond roughly to 'category' and 'sub-category' respectively.
        i. https://github.com/RyanAlberts/Springbaord-Capstone-Project/blob/master/Instacart_EDA_Categories__General.ipynb



Some interesting trends exist; for example, the popularity of the alcohol category during Friday and Saturday (note: Instacart did not divulge which days correspond to the numbered values 0-6, which these explorations served to clarify). Seaborn's heatmap was a useful visual tool in identifying such trends.



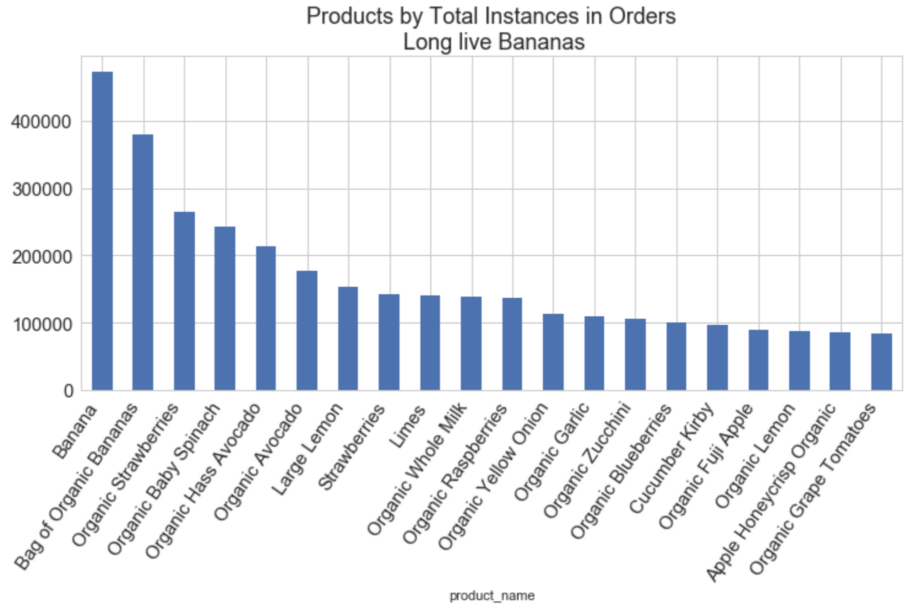


These trends would prove helpful in the feature engineering stage of this project.

2. Assortment

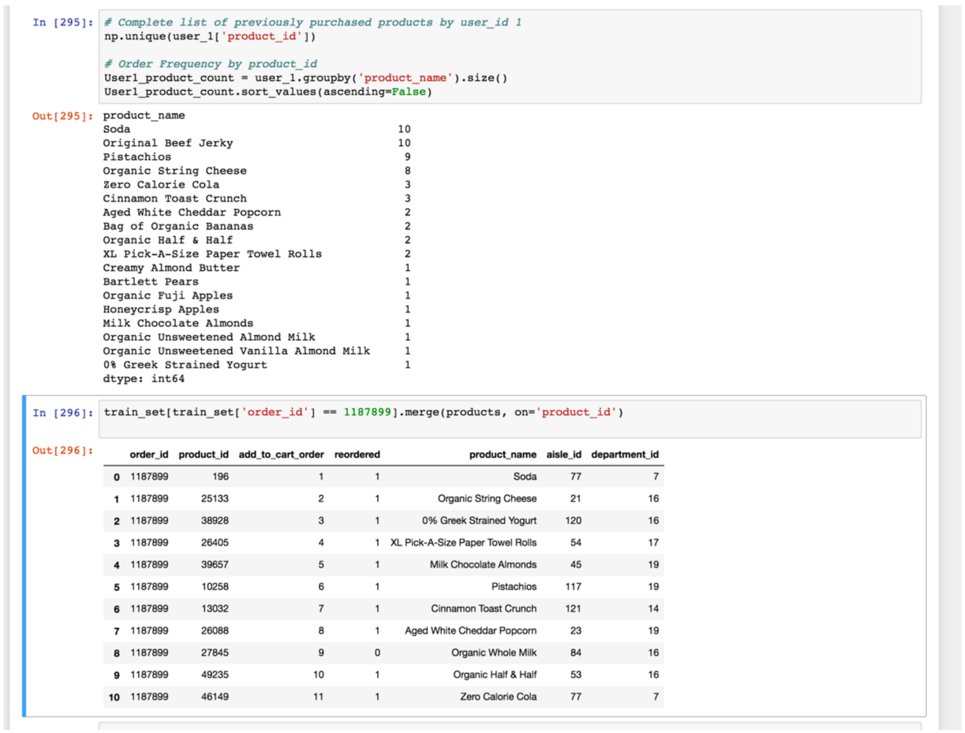
   a. This notebook dives a deeper into how a user's product assortment (the particular combination of products they include in each order) affects reorder behavior. I attempted to explore how category affinities, basket sizes, product diversity, and order frequency interact to influence user behavior.

i. https://github.com/RyanAlberts/Springbaord-Capstone-Project/blob/master/Instacart_EDA_DataStory_Assortment.ipynb



3. <u>Engagement</u>
   a. This notebook incorporates the previous two and served as a launching pad for feature engineering. It attempts to address issues like 'New' vs. 'Seasoned' customer order behavior.
      i. https://github.com/RyanAlberts/Springbaord-Capstone-Project/blob/master/Instacart_EDA_DataStory_Engagement.ipynb

<u>Deep Dive – Exploring the Data</u>:
I explored some interesting trends with a deep-dive into user_id 1.



```
In [295]: # Complete list of previously purchased products by user_id 1
          np.unique(user_1['product_id'])

          # Order Frequency by product_id
          User1_product_count = user_1.groupby('product_name').size()
          User1_product_count.sort_values(ascending=False)

Out[295]: product_name
          Soda                                          10
          Original Beef Jerky                           10
          Pistachios                                     9
          Organic String Cheese                          8
          Zero Calorie Cola                              3
          Cinnamon Toast Crunch                          3
          Aged White Cheddar Popcorn                     2
          Bag of Organic Bananas                         2
          Organic Half & Half                            2
          XL Pick-A-Size Paper Towel Rolls               2
          Creamy Almond Butter                           1
          Bartlett Pears                                 1
          Organic Fuji Apples                            1
          Honeycrisp Apples                              1
          Milk Chocolate Almonds                         1
          Organic Unsweetened Almond Milk                1
          Organic Unsweetened Vanilla Almond Milk        1
          0% Greek Strained Yogurt                       1
          dtype: int64

In [296]: train_set[train_set['order_id'] == 1187899].merge(products, on='product_id')

Out[296]:
```

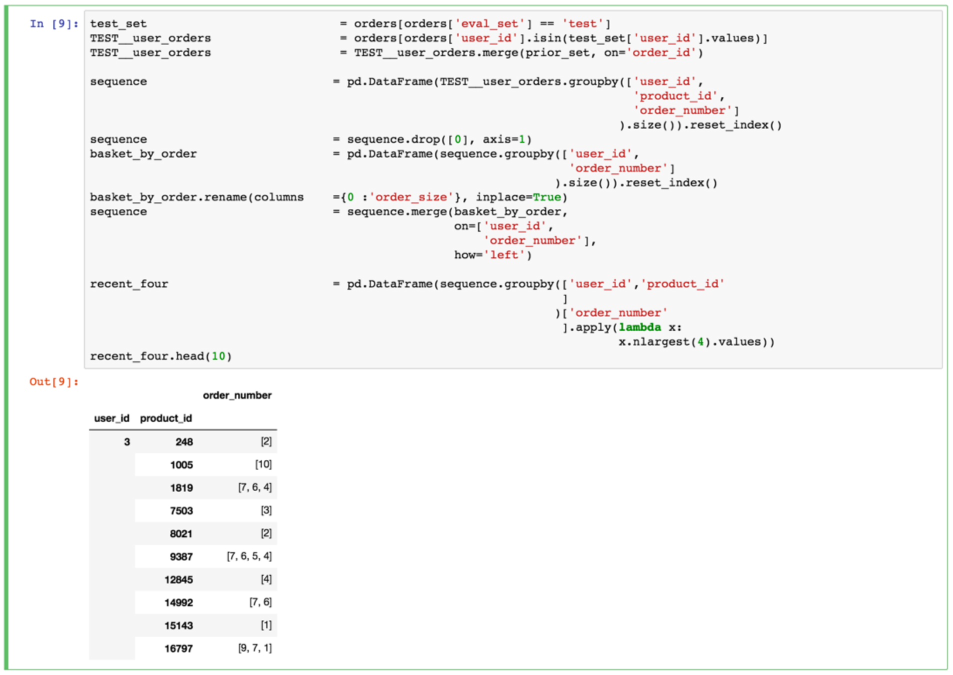
	order_id	product_id	add_to_cart_order	reordered	product_name	aisle_id	department_id
0	1187899	196	1	1	Soda	77	7
1	1187899	25133	2	1	Organic String Cheese	21	16
2	1187899	38928	3	1	0% Greek Strained Yogurt	120	16
3	1187899	26405	4	1	XL Pick-A-Size Paper Towel Rolls	54	17
4	1187899	39657	5	1	Milk Chocolate Almonds	45	19
5	1187899	10258	6	1	Pistachios	117	19
6	1187899	13032	7	1	Cinnamon Toast Crunch	121	14
7	1187899	26088	8	1	Aged White Cheddar Popcorn	23	19
8	1187899	27845	9	0	Organic Whole Milk	84	16
9	1187899	49235	10	1	Organic Half & Half	53	16
10	1187899	46149	11	1	Zero Calorie Cola	77	7

For any given customer, there is often a long-tail of products that are never reordered, and a small but formidable group of products that are consistently reordered. There was also a strong correlation between the ‘days_since_prior_order’ variable and the basket size and assortment of any particular order. The ‘EDA_FeatureSet’ Notebook provides the basis for a lot of feature engineering that I later developed for the machine-learning algorithms, and provided a cauldron for improving my skills with grouping, merging, indexing, and applying methods to various pd.DataFrame and Series objects.

Given the size of the data, it was an ideal way of learning first-hand how crucial lean, efficient, modular programming is, including list comprehensions, apply methods, and other sorting and grouping techniques. These techniques were essential to exploring the data, considering you can easily loose track of a line of thought when it takes your program hours to iterate through a for loop, or to iterate through a dataset where the dtypes haven’t been optimized for the purpose.

Despite it becoming clear, from the Kaggle discussion forums and my own experience, that there is no ‘magic’ feature capable of predicting reorder behavior across the 3M orders, there are some general trends that are worth noting.

For example, the more recent order history (e.g. last 4 orders) are more capable of predicting future order behavior than an averaging of the entire order history of a customer (maxing out at 100 orders). I was able to extract data from the most recent four orders using a number of tactics based on my application. Here is an instance of extracting the most recent order number:



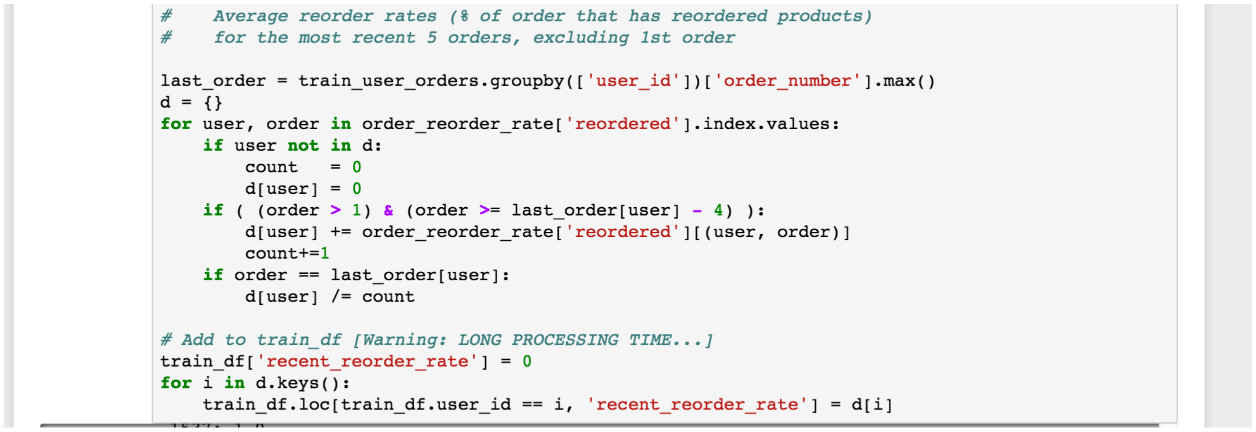
```
In [9]: test_set                           = orders[orders['eval_set'] == 'test']
        TEST__user_orders                  = orders[orders['user_id'].isin(test_set['user_id'].values)]
        TEST__user_orders                  = TEST__user_orders.merge(prior_set, on='order_id')

        sequence                          = pd.DataFrame(TEST__user_orders.groupby(['user_id',
                                                                                    'product_id',
                                                                                    'order_number']
                                                                                  ).size()).reset_index()
        sequence                          = sequence.drop([0], axis=1)
        basket_by_order                   = pd.DataFrame(sequence.groupby(['user_id',
                                                                           'order_number']
                                                                         ).size()).reset_index()
        basket_by_order.rename(columns    ={0 :'order_size'}, inplace=True)
        sequence                          = sequence.merge(basket_by_order,
                                                           on=['user_id',
                                                               'order_number'],
                                                           how='left')

        recent_four                       = pd.DataFrame(sequence.groupby(['user_id','product_id'
                                                                          ]
                                                                         )['order_number'
                                                                          ].apply(lambda x:
                                                                                  x.nlargest(4).values))
        recent_four.head(10)
Out[9]:
```

		order_number
user_id	product_id	
3	248	[2]
	1005	[10]
	1819	[7, 6, 4]
	7503	[3]
	8021	[2]
	9387	[7, 6, 5, 4]
	12845	[4]
	14992	[7, 6]
	15143	[1]
	16797	[9, 7, 1]

I was able to use this to compute, among other things:

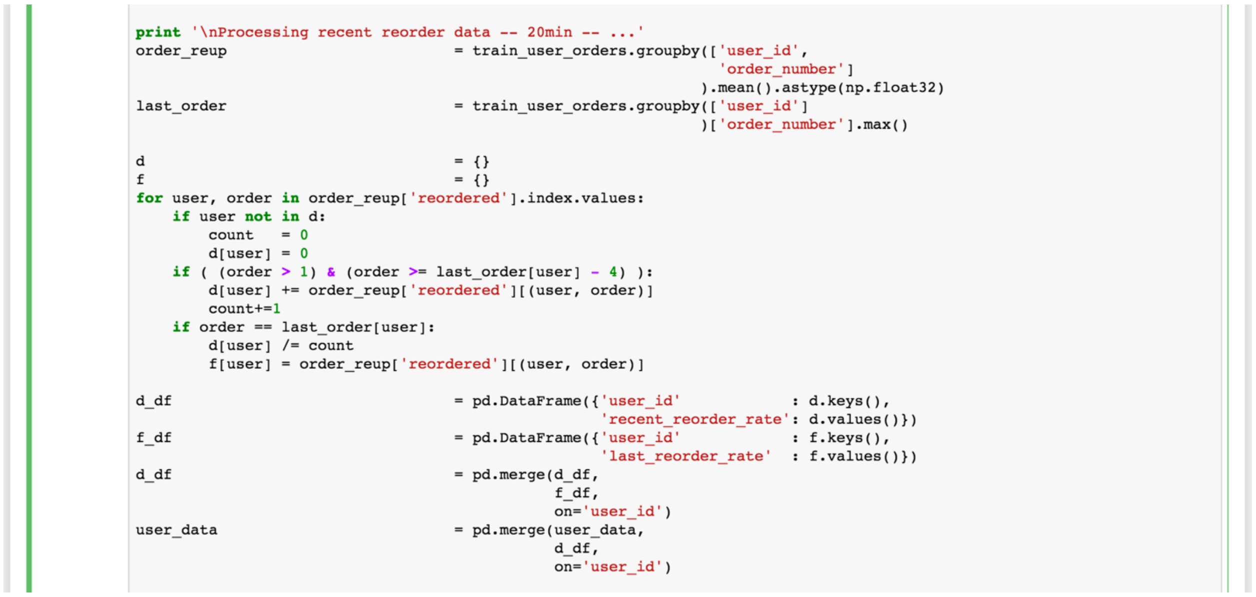
```
recency_ratio = most_recent_order_customer / most_recent_order_each_product
range = (0,1]
e.g. customer with 12 prior orders, and a particular product was most
recently ordered in the 7th order:  recency_ratio = 12 / 5 = .583
```

I realizes fairly early on that processing times when iterating over millions of observations necessitate efficient code



```python
#    Average reorder rates (% of order that has reordered products)
#    for the most recent 5 orders, excluding 1st order

last_order = train_user_orders.groupby(['user_id'])['order_number'].max()
d = {}
for user, order in order_reorder_rate['reordered'].index.values:
    if user not in d:
        count   = 0
        d[user] = 0
    if ( (order > 1) & (order >= last_order[user] - 4) ):
        d[user] += order_reorder_rate['reordered'][(user, order)]
        count+=1
    if order == last_order[user]:
        d[user] /= count

# Add to train_df [Warning: LONG PROCESSING TIME...]
train_df['recent_reorder_rate'] = 0
for i in d.keys():
    train_df.loc[train_df.user_id == i, 'recent_reorder_rate'] = d[i]
```

Note the difference between the above and below code was nearly 3.5 hours, and I was capable of coopting the below code to create two features rather than one:



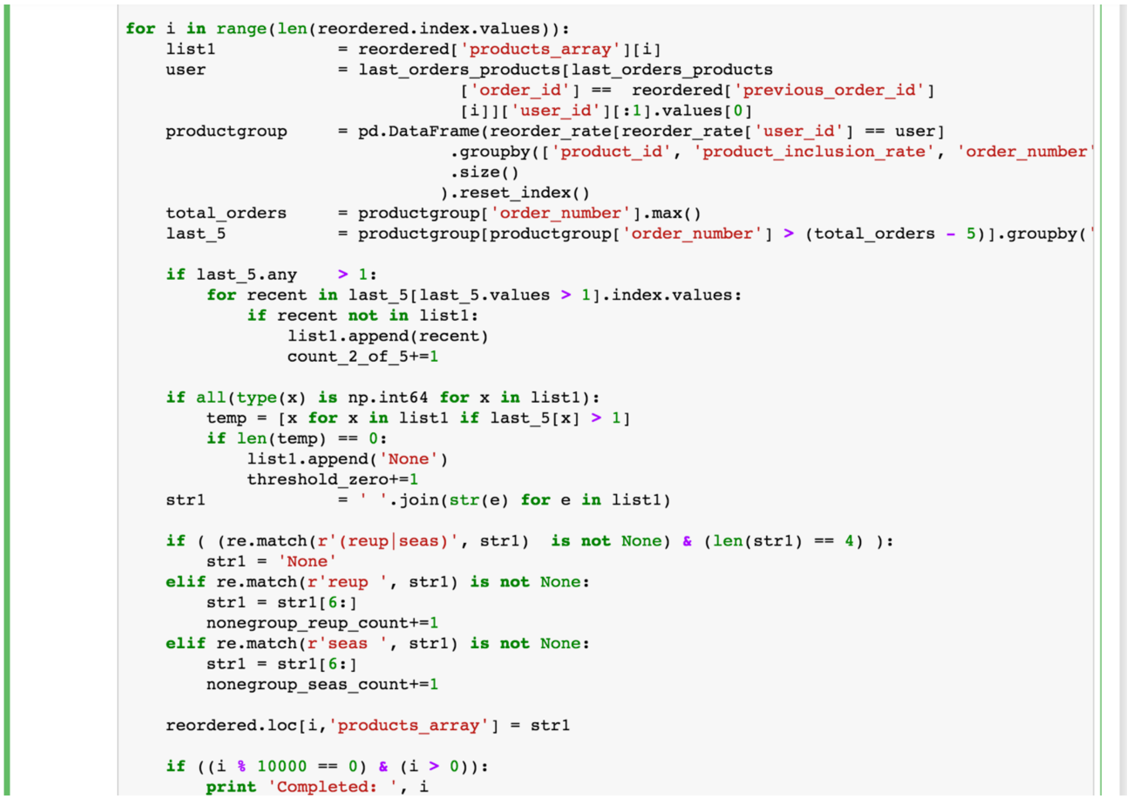
```python
print '\nProcessing recent reorder data -- 20min -- ...'
order_reup                        = train_user_orders.groupby(['user_id',
                                                                'order_number']
                                                              ).mean().astype(np.float32)
last_order                        = train_user_orders.groupby(['user_id']
                                                              )['order_number'].max()

d                                 = {}
f                                 = {}
for user, order in order_reup['reordered'].index.values:
    if user not in d:
        count   = 0
        d[user] = 0
    if ( (order > 1) & (order >= last_order[user] - 4) ):
        d[user] += order_reup['reordered'][(user, order)]
        count+=1
    if order == last_order[user]:
        d[user] /= count
        f[user] = order_reup['reordered'][(user, order)]

d_df                              = pd.DataFrame({'user_id'            : d.keys(),
                                                  'recent_reorder_rate': d.values()})
f_df                              = pd.DataFrame({'user_id'            : f.keys(),
                                                  'last_reorder_rate'  : f.values()})
d_df                              = pd.merge(d_df,
                                             f_df,
                                             on='user_id')
user_data                         = pd.merge(user_data,
                                             d_df,
                                             on='user_id')
```

Also, employing statistical analyses of the central tendencies of customer behaviors (e.g. the coefficient of variability for the basket size, or the standard deviation of the number of orders since that user bought a specific item) proved remarkably capable of establishing confidence in the probabilities associated with reordering certain products. For example, we can be more certain of the probabilities associated with certain products for a customer that shows very little variance in the time and regularity of their order (i.e. regularly ordered at 9AM on Monday morning every week) as compared to someone who orders erratically and infrequently. Thus we can weight the probabilities associated with the first user more heavily than those associated with the second user. I will address this in more depth in the Gradient Boosting Machine section, because this is something they accomplish quite well.

However, the data is highly unbalanced. With three million orders and 4.8M (user_id, product_id) pairs in the test set alone (over 12M total), there is a high degree of variability in binary reorder behavior (0 for negative observation, 1 for positive observation), which made it clear that over-fitting would be a major issue.

Initial Model: Unrefined, but Capable

I built my initial predictive model without any sort of machine-learning library, as a way of benchmarking my progress. At its core, my initial model was built to look at every single product a user ordered, and reduce that set to a smaller sub-group of products that were likely to be reordered. There was no probabilistic modeling, regression, or RMSE calculations, but rather a relatively robust, albeit unsophisticated, mapping of input (user-product pairs) to output (user-product pairs) based on static, binary-criteria-based conditional operations. For example, if the product was new in the most recent order, the product would be excluded, and if the product had been ordered in two or more of any of the previous 5 orders, it would be included.



```
for i in range(len(reordered.index.values)):
    list1           = reordered['products_array'][i]
    user            = last_orders_products[last_orders_products
                            ['order_id'] ==  reordered['previous_order_id']
                            [i]]['user_id'][:1].values[0]
    productgroup    = pd.DataFrame(reorder_rate[reorder_rate['user_id'] == user]
                           .groupby(['product_id', 'product_inclusion_rate', 'order_number'
                           .size()
                          ).reset_index()
    total_orders    = productgroup['order_number'].max()
    last_5          = productgroup[productgroup['order_number'] > (total_orders - 5)].groupby('

    if last_5.any    > 1:
        for recent in last_5[last_5.values > 1].index.values:
            if recent not in list1:
                list1.append(recent)
                count_2_of_5+=1

    if all(type(x) is np.int64 for x in list1):
        temp = [x for x in list1 if last_5[x] > 1]
        if len(temp) == 0:
            list1.append('None')
            threshold_zero+=1
    str1            = ' '.join(str(e) for e in list1)

    if ( (re.match(r'(reup|seas)', str1)  is not None) & (len(str1) == 4) ):
        str1 = 'None'
    elif re.match(r'reup ', str1) is not None:
        str1 = str1[6:]
        nonegroup_reup_count+=1
    elif re.match(r'seas ', str1) is not None:
        str1 = str1[6:]
        nonegroup_seas_count+=1

    reordered.loc[i,'products_array'] = str1

    if ((i % 10000 == 0) & (i > 0)):
        print 'Completed: ', i
```

Note that this model is lacking in many ways that GBMs are not, but for starters: this model had a strict, deterministic sequence with no randomness, and significant bias. Any given operation would inherit the bias and error of previous operations. Also, the model had no way of addressing optimal F1-scores for factoring how the inclusion of one product might influence including another. The probability of including one product depends immensely on the probability of all other products, because the probability of a customer ordering 1 product is 100% and decreases to nearly 0 as you increase basket-size. No matter how likely a product is to be included, if it is the 145th product that you are including in your prediction, you have likely made a number of errors. There are so many problems with this model, but I will mention just one more: None-handling. I have yet to discuss this in detail, but as an introduction:

Instacart allows you to include 'None' as you would any other product, in any order for which you predict there will be no reordered products. They also allow you to include 'None' in orders where you also predict products, by way of allowing partial credit (e.g. if the actual observation is 'None', and you are not sure, your F1 score would be some fraction if you include 'None' with some products, whereas it would be 0 without it). My initial model handled 'None' by identifying 'high-risk' orders using similar tactics described above, rather than using probabilistic modeling, which seeks to compare the probability of 'None' against the probability of all other items, effectively treating 'None' like any other product.

```
P(None) = P(not item 1)P(not item 2).......*P(not item k)
```

This initial model was rough. It scored as high as .365, which was enough to get me to the top 50%, but the amount of time and energy to progress became exponential at a point, because criteria for inclusion were independent (refining one might detract from another, and the net loss in accuracy increased with each new criteria, due to increasing complexity), uniform across users (meaning, unlike the data, the criteria did not vary based on user, despite the obvious fact that user behavior is idiosyncratic), and insensitive to things like basket size and statistical measures of variance. It was a great benchmark, but not enough to get to the top of the ranks.

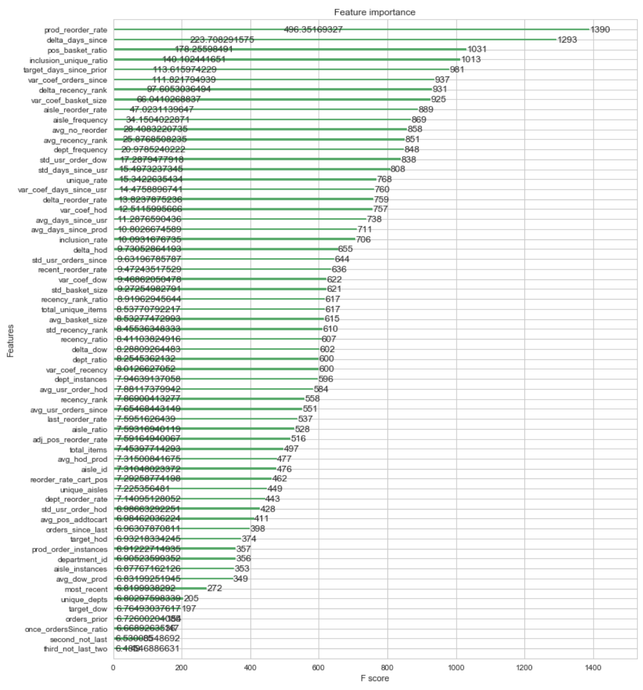
That said, I found the benchmark models extremely valuable because I was able to interact with the training and test data sets and see how certain categories of criteria are more important than others (e.g. recent order history is hugely important, but whether a product is new in the most recent order is less useful, for predictive purposes). These trends helped me immensely when it came time to employ machine-learning algorithms.

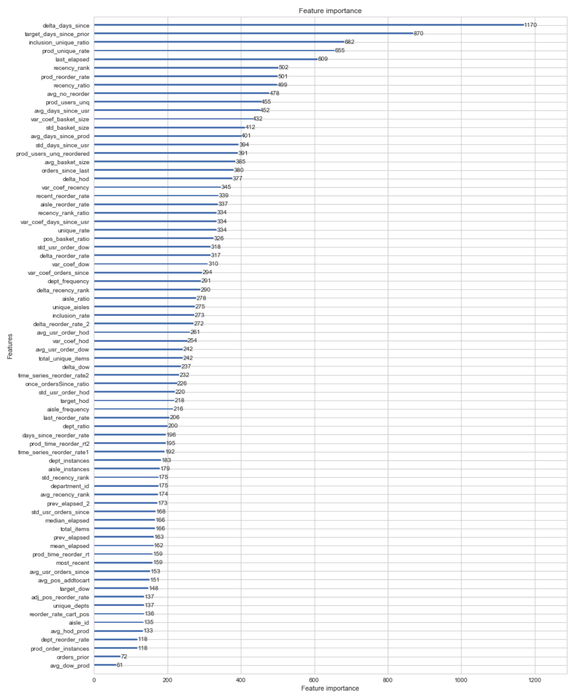
<u>Gradient Boosting: LightGBM, XGBoost, CatBoost</u>:

Gradient Boosting Machines (GBMs) are an extremely popular decision-tree machine-learning model, of which there are many algorithmic interpretations. It is popular in many disciplines, but notably in Kaggle competitions, because of its applicability and effectiveness in business strategy environments, where decisions can be derived from binary classifiers. It is also capable of computing for regression as well as boosted trees applications, which makes it useful in a range of situations from object retrieval (search engines Yahoo and Yandex) to Demand-Forecasting (Instacart).

<u>Comparative Analysis – Predictive Algorithms</u>:

How different models fare can vary widely, depending on idiosyncrasies of the data set in question, your team's objective, how the model interacts with different data types, and what statistical considerations the models emphasize, among others. Here is an illustration of how significantly the differences can be between different models' treatment of the same features:





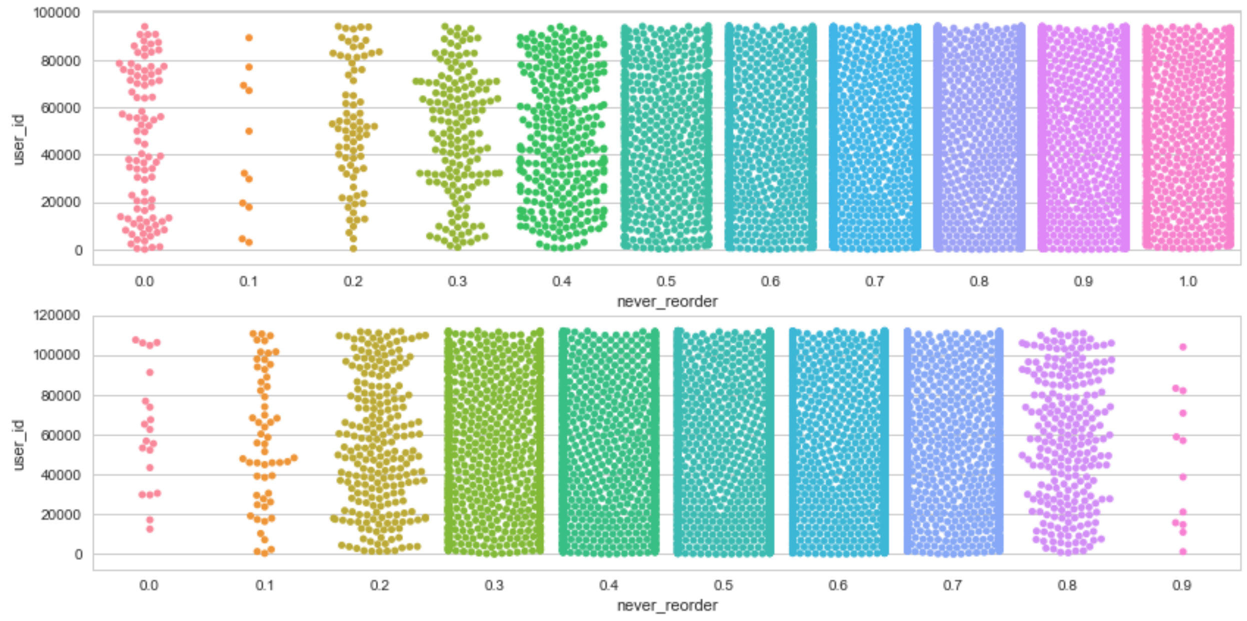
CatBoost, for example, treats categorical data with comparatively robust tuning mechanisms, which give the engineer more flexibility if defining and treating categorical data. This can be a huge benefit when you have a large feature-set of categories. LightGBM, on the other hand, is fast, which has its own merits. XGBoost often has the best predictions, because it has extensive parameters for tuning loss functions and regularization. XGBoost is more flexible with categorical feature recognition and smoothing, whereas LightGBM is very sensitive to encoding. Also, documentation exists for LightGBM, but it took me some time to figure out that categorical feature support was being deprecated, certain parameters were defunct, and early stopping for boosting rounds needs to be explicitely called in training (as opposed to setting it in the parameters). Generally speaking, they all have significantly different terminology and nuances that govern parameter-tuning, so it was helpful to operate using an ensemble method. I focused on GBMs, where there are two primary considerations:

1) Loss Function – This refers to the training loss function, which measures the performance of the model given a certain set of parameters. This can be
    a. logloss (logarithmic loss): measures the uncertainty of predictions by scoring how far your prediction diverges from actual observation.
    b. AUC (Area Under the ROC precision-recall Curve): measures probability that, from two random observations, (one negative, one positive), your classifier will assign the positive example a higher score. This is particular useful, I found, when the goal is F1 optimization, as this competition is.
    c. RMSE (Root Mean Squared Error): measures the difference between actual and predicted values, according to the root mean squared error.
    d. Many, many more.
2) Regularization – This concept refers to all the mechanisms in a model that control the complexity of the model, by addressing the bias-variance tradeoff. More complex models tend to more accurately represent the training data, but also more variable in their predictions. Regularization attempts to create a 'best fit' model that generalizes from the training data and balances the competing forces of needing to be accurate while also needing to generalize for unseen data.

How different models deal with categorical data is essential various reasons, including overfitting for unbalanced data sets, speed considerations (hot-1 encoding, i.e. HOE), and since F1 optimization algorithm assumes feature independence. The feature independence assumption means that a model must be capable of penalizing categorical data with too many children (most start with 255 max_bin_size, but nevertheless produce different predictions with the same features), lest it over-estimate the predictive power of related categories. For example, I had categories that recorded whether a product was included in the most recent order, but also whether it was included in all 4 of the last 4 most recent orders, all 3 of the last 3 most recent orders, 2 or more of 4 most recent orders, etc. These necessarily overlap to comprehensively treat all possible scenarios. What I found was that wrapping these features into a single feature, ranking each product according to a combination

Feature Engineering – Tuning Out the Noise:

It's easy to understand that the past is capable of influencing the future (Newton's third law, etc), that past orders are capable of predicting future orders with some degree of certainty. The real exciting moment is when you realize that the data used for those predictions can be engineered, or transformed from its current state into a distinct new state that more robustly represents the underlying problem; predictive algorithms can use engineered features to improve accuracy of predictions by enabling them to deal better with unseen data. I found this aspect of predictive modeling the most exciting, because it is deeply rooted in exploratory analysis and domain knowledge.



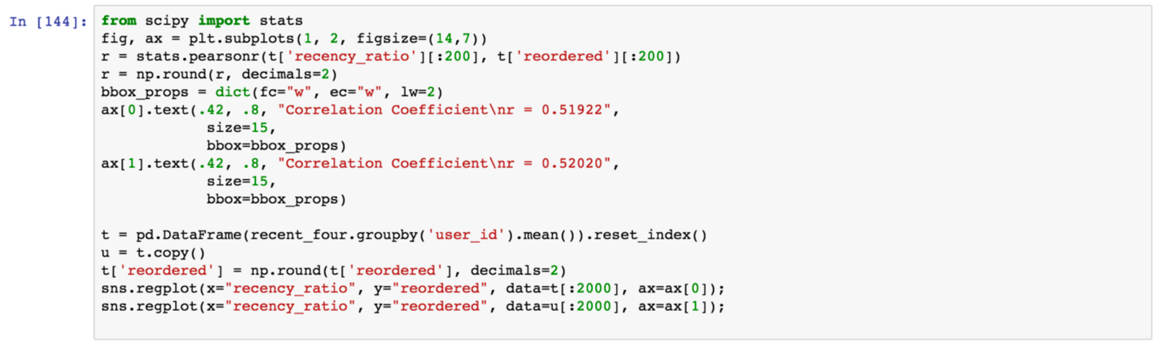
The above illustrates an engineered feature 'never_reordered', which is a binary categorical variable capturing a user's mean probability of never reordering a product chosen at random. The top plot illustrates users who have 6 or fewer orders, and the bottom illustrates users with 20 or more orders. Capturing this trend boosted my score by .003, which was significant.

For example, Instacart provided order numbers for past orders, which were each associated with products ranked according to when they were added to the customer's cart. To leverage the hidden value in these features, I created new features that captured the mean values for add_to_cart position, standard deviations of basket sizes, and reorder rates for each customer associated with each position in their basket (among others). The goal is distill the predictive power of the provided variables for use in the model. In total, there were 15 features provided by Instacart, which I turned into 103 via feature engineering – some of which made it into the final model, while others were deprecated. This is an example of my feature set along with ranked importance according to XGBoost, early in the competition:

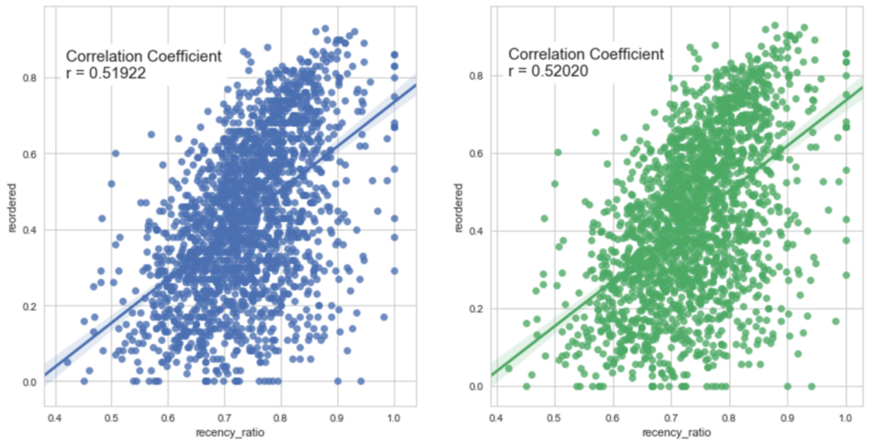
A key aspect in feature engineering is focusing on quality pattern recognition. I learned quickly that very subtle mistakes can be harmful to the predicative capacity of the model. One such mistake is "over-engineering", or "target leakage", which refers to engineering new features using the target of your learning pool (some other feature that your model is using to form its predictions). Redundancy of features can be harmful because you may given weight to a certain trend that does not actually exist.

I found that one of the most helpful things to consider when designing features is the type of feature you are creating. Notwithstanding the capacity for certain GBMs to handle certain types over others, there are intrinsic differences between ranked, numeric, and categorical variables.

There is also the matter of deciding the type of number you will use for numeric data. Some variables are better served as floats (decimals), others as integers, and others are best served by first calculating floats for feature engineering and afterwards rounding to integer values. For example, the correlation can be seen to increase, however slightly, when rounding 'recency_ratio' (detailed on page 10 of this thesis):



```
In [144]: from scipy import stats
          fig, ax = plt.subplots(1, 2, figsize=(14,7))
          r = stats.pearsonr(t['recency_ratio'][:200], t['reordered'][:200])
          r = np.round(r, decimals=2)
          bbox_props = dict(fc="w", ec="w", lw=2)
          ax[0].text(.42, .8, "Correlation Coefficient\nr = 0.51922",
                      size=15,
                      bbox=bbox_props)
          ax[1].text(.42, .8, "Correlation Coefficient\nr = 0.52020",
                      size=15,
                      bbox=bbox_props)

          t = pd.DataFrame(recent_four.groupby('user_id').mean()).reset_index()
          u = t.copy()
          t['reordered'] = np.round(t['reordered'], decimals=2)
          sns.regplot(x="recency_ratio", y="reordered", data=t[:2000], ax=ax[0]);
          sns.regplot(x="recency_ratio", y="reordered", data=u[:2000], ax=ax[1]);
```



The only way to test the predictive capacity for different formulations of feature types, you generally need to submit your predictions (test your model against unseen data) or else use local cross-validation. Both can be time-consuming, to its valuable to be able to see your progress locally. Unfortunately, I've experienced significant differences in F1 score when I tested locally using sklearn.metrics.f1_score as compared to the public leaderboard on Kaggle.

I experienced higher correlations between the two measures when I increased the size of my validation set, but only to a point. It is my impression from the Kaggle discussion forums that these measures are largely subject to how representative the training data is of the test data, and that Instacart's training set is pretty reliable. Comparing against other competitions, like the Mercedes challenge, supports this hypothesis.

When No Prediction is the Best Prediction – Predicting the Unpredictable:

Because excluding 'None' for 'None' orders gives you an F1 score of 0, the competition incentivizes you to submit partial-score answers, including 'None' with high-probability products The training set exhibits 6.556% of the 131k orders did not reorder anything, i.e. qualify for 'None'. For the 75,000 orders in the test set, that is in the neighborhood of 4,900 orders. The tricky part is that the users that might not reorder anything tend to have erratic or otherwise uncategorized behavior relative to the sample, so it makes it difficult to pin down.

I found that the most effective way was to target high-risk categories of users, where the F1 score of 'None' outweighs any previously ordered products. Still, I included 'None' with approximately 8-12k orders with varying levels of success, so my error rate was certainly high.

Optimizing F-Measures for Binary Classifier – Dynamic Thresholds:

Considering the competition is evaluated using mean F1 score, it seems obvious that you should optimize your predictions according to the very metric. Unfortunately, implementing an F1-optimization method is another thing entirely.

The first, critical step was to realize that F1 scores must consider entire orders, not just products, which means you can't realistically build F1 measures into your feature set. Intuitively, any given product's probability of being included in the user's next order depends heavily on what else the user buys, because naturally they have a limit as far as how much they are willing to spend in a single shopping trip. More rigorously, the F1 score is derived from sets of binary product predictions that cumulatively represent the order's F1 score, which means that you cannot compute optimal F1 scores on the product level, but on the order level.

With that discovery firmly settled, the task becomes: how to transform the product predictions produced by the GBM into F1 optimized order predictions. It turns out that there are a number of resources available that served as my boilerplate benchmark for this process.

With the help of the Kaggle discussion boards, I was able to figure out that an effective implementation must involve setting thresholds for product predictions that reflect optimal F1 score. I was already dynamically setting thresholds based on average basket sizes and trial-and-error approaches to prediction cutoffs, which were rough estimates.

I was able to implement F1 optimization first by sorting the list of all possible products a user had previously ordered according to their predicted probability of inclusion. Then, I calculated the F1 score of the top ranked product according to my GBM's predictions, followed by the second ranked product, whose F1 score assumes the user is also buying the top-rated product, and so on. At a certain point, the F1 score peaks and begins to go down as you go down the list, because it factors in the bias-variance tradeoff encapsulated in mean F1 scores.

I saw a drastic improvement in score using this approach. My F1 score in the public leaderboard went up from .386 to .395, which represented a huge increase in ranking. The next step was to incorporate 'None'-handling, which was the cause of much concern. I realized that treating it as if it were a product itself would allow me to compute F1 scores using 1 – p(product) and adding 'None' to orders where the probability of 'None' exceeded the highest probability of any product in an order. This also gave me a bump in score, from .395 to .398. This was enough to get me into the Top 5% of the competition.

What initially seemed a daunting task was ultimately extremely rewarding, because I was able to implement a tailored F1 optimized solution using theoretical principles to write code capable of meaningfully engaging the data set at hand. I was also able to stitch together resources from the Kaggle discussion forums, stackoverflow, academic research papers, and others, which I've been told is often representative of software development generally.

Once I had optimized basket size estimates and F2 scores for each product, including 'None', I had the basis of what became my final ensemble model, which merged the predictions of XGBoost and LightGBM. With this framework in place, I spent an enormous amount of time feature engineering and parameter tuning, which allowed me to tweak my score and land in stay in the Top 5%. Unofortunately, Instacart pushed the deadline back one week, to August 14th, and my Master's program starts August 10th, so I've decided to focus on my program and let my score stand. I imagine there will be some shake-ups in the competition, but I've done what I've come to do.

<u>**Conclusions & Recommendations**</u>:

The beauty of this competition is that the deliverable is itself an immediately actionable strategy platform. The code that ultimately created the submission file contains logic that is capable of predicting what products a user is likely to order next, and can be pushed into production in various ways:

1. As a method of refining search recommendations for users while they are shopping
2. As a method of offering discounts and promotions to more effectively manage inventory
3. As a method of consumer outreach, in email or other marketing campaigns, to more effectively engage customers with personalized content

<u>**Limitations of Investigation & Areas for Further Exploration**</u>:

I have learned an enormous amount in the process of completing this capstone project, from the wealth of resources that Springboard and my mentor have provided to the resources and inspiration that can be found in the Kaggle discussion forums. Over the course of the past six months, I've transformed my skillset into one that can propel me to the top 5% of a Kaggle competition in binary classification, having (relatively) effectively implemented several machine learning algorithms to create predictions and forecast demand for an eCommerce grocer, where demand forecasting is a crucial element of business strategy. That said, there are several limitations that offer great opportunities to further my foray into Data Science.

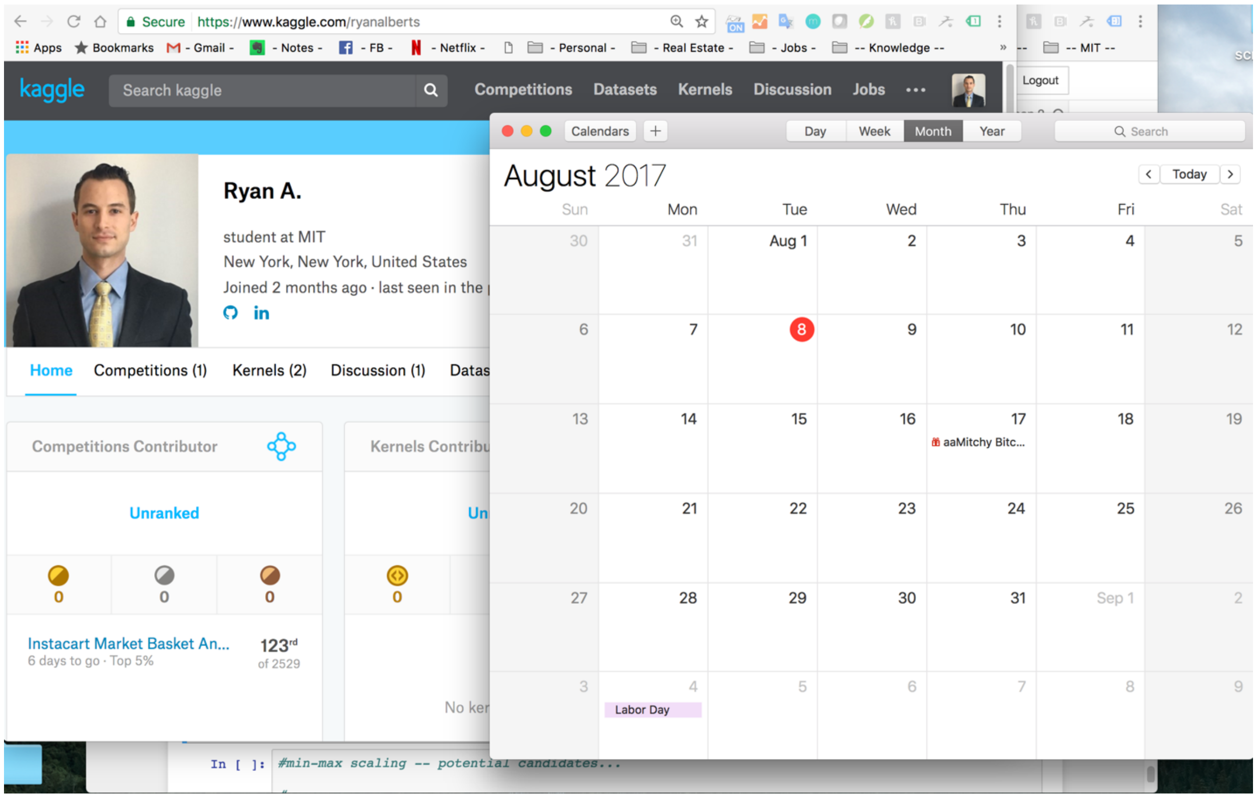
Considering my knowledge of machine learning has been framed within the specific case of Instacart's Kaggle competition, I expect to dive into other Data Science projects to add diversity to the sorts of problems I'm capable of handling. Binary classification is one of many applications of machine learning, and I look forward to exploring GBMs for regression, as well as neural networks and SVMs for more advanced topics. Also, because I focused primarily on the efficacy of the code vis-à-vis rankings in the competition, I will need to spend time investigating the current state of research as far as the logic, infrastructure, and methodology driving machine learning algorithms, which is a constantly evolving discipline.

I also hope to advance my ability to test locally, which I was having trouble doing reliably during the competition. The nuances of cross-validation, leakage, and grid-search are things that I need to become more familiar with in the future.

Evaluation:

My ultimate goal was to use this experience, including immersing myself in the Kaggle discussions and culture of collaborative open-source analysis, to be able to function at a high level within an organization both from an Operations & Strategy perspective, as well as from a Data Science perspective, focusing on applied strategy and decision-making. I have learned a great deal regarding machine learning programs, statistical applications in data science, and how to consume and adapt public code repositories for the purpose at hand, as well as iterate off of collaborative coding environments in a productive and competitive manner.

I am extremely proud to have a production-ready predictive model capable of handling intense Demand-Forecasting applications for Instacart, and the skills to transfer these capabilities to future endeavors.


Secure https://www.kaggle.com/ryanalberts
Apps Bookmarks Gmail Notes FB Netflix Personal Real Estate Jobs Knowledge MIT
kaggle
Search kaggle
Competitions Datasets Kernels Discussion Jobs
Logout
Home Competitions (1) Kernels (2) Discussion (1) Datas
Competitions Contributor
Kernels Contribu
Unranked
Calendars Day Week Month Year Search Today
Sun Mon Tue Wed Thu Fri Sat

Ryan A.
student at MIT
New York, New York, United States
Joined 2 months ago · last seen in the

Unranked 0 0 0

Instacart Market Basket An... 123rd of 2529
6 days to go · Top 5%

August 2017

17 aaMitchy Bitc...
4 Labor Day

```
In [ ]: #min-max scaling -- potential candidates...
```

Citations:

Ye, N., Chai, K., Lee, W., and Chieu, H. *Optimizing F-Measures: A Tale of Two Approaches*. In ICML, 2012.

Zachary C. Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. *Optimal thresholding of classifiers to maximize F1 measure*. In Machine Learning and Knowledge Discovery in Databases, pages 225–239. Springer, 2014.