Instacart Market Basket Analysis

PREDICT NEXT BASKET

Prajakta Gujarathi | Capstone Project | 10/5/2017

Definition

PROJECT OVERVIEW:

Instacart, a grocery ordering and delivery app, aims to make it easy to fill customer's refrigerator and pantry with their personal favorites and staples when they need them. After selecting products through the Instacart app, personal shoppers review order and do the in-store shopping and delivery for customers. But achieving this simplicity cost effectively at scale requires an enormous investment in engineering and data science.

Knowing customer's next order will be helpful for Instacart business in following way:

- Optimize algorithm which enroute Instacart shoppers, for timely and efficient delivery
- Balancing supply and demand of customers. This includes estimating Instacart's capacity to fulfill orders to create optimal customer experience.
- Provide customers with recommendation by analyzing their previous purchase history.

PROBLEM STATMENT

In this project goal is to use previous transactional data of customer to develop models that predict which products a user will buy again. Task involve are following:

1. Download data from Recently, Instacart open source data from 3 Million Instacart Orders, Open Sourced.
2. Explore, visualize and analyze important dimensions
3. Preprocess data by adding or reducing dimension as required.
4. Train a different models using preprocessed data and identify best model
5. Predict next carts products for test users.

Final result will be a text file output with order id followed by list of Product Ids that are predicted to be being bought.

Kaggle competition Link: https://www.kaggle.com/c/instacart-market-basket-analysis

Metrics:

F1 is a common metric for binary classifiers; it takes into account both true positives and true negatives with equal weight.

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

$$F_1 = 2 \cdot \frac{PRE \cdot REC}{PRE + REC}$$

TP = will be product which are actually present in next order.

FP = will be product that incorrectly predicated to be in next order.

TN = will be product which are not present in next orders.

FN = will be products which are supposed to be in next order be predicated as not.

The problem here is to approximate P(u,p | user's prior purchase history) which stands for how much likely user u would repurchase product p given prior purchase history.

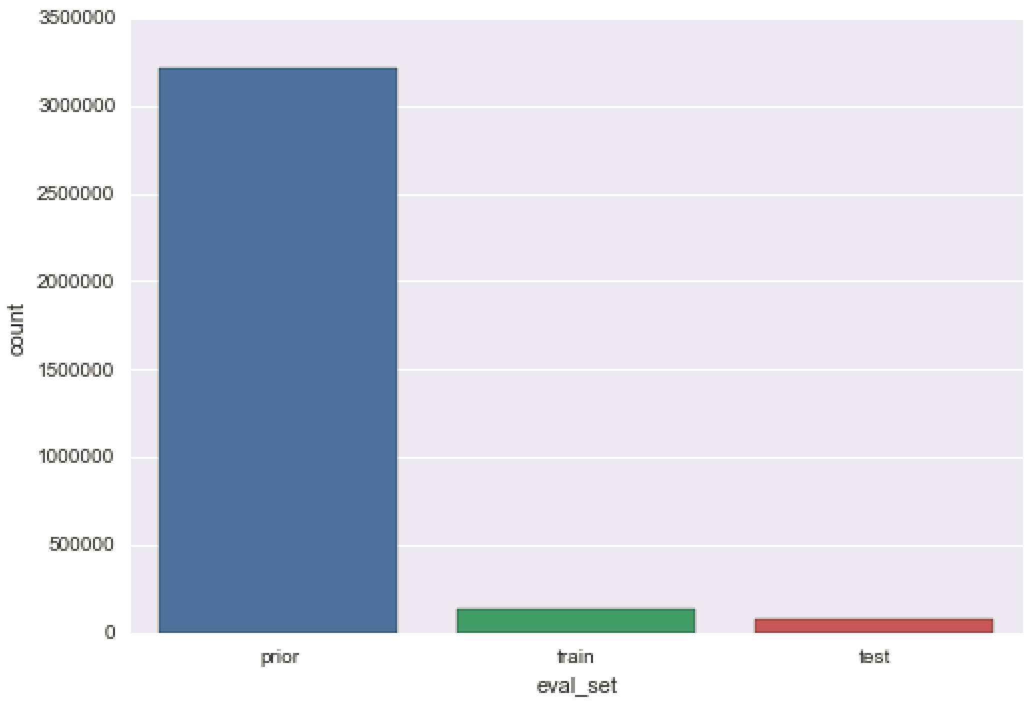
So my main model is a binary classifier. Features are created manually or automatically are fed to the classifier which will predict that given orderId ProductId pair will be purchased 1 or not 0.

Analysis:

Exploratory Data Analyst:

By Looking at Instacart data we know that we have **21 departments**, **134 aisles** and **49688** different **products**. In this dataset, 4 to 100 orders of a customer are given and we need to predict the products that will be re-ordered. So the last order of the user has been taken out and divided into train and test sets. All the prior orders information of the customers are present in order_products_prior file. We can also note that there is a column in orders.csv file called eval_set which tells us as to which of the three datasets (prior, train or test) the given row goes to.

```
<matplotlib.axes._subplots.AxesSubplot at 0xc9e51d0>
```



Orders_product*csv file has more detailed information about the products that been bought in the given order along with the re-ordered status. So We have total 206209 number of total customers. Out of which for 131209 cutomers last purchase is given in training set and we have to predict for 75000 test customers.

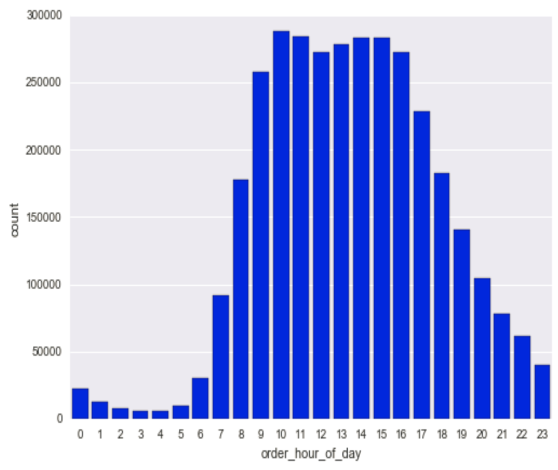
Exploratory Visualization:

Entire data visualization can be found in Instacart.ipynb file.

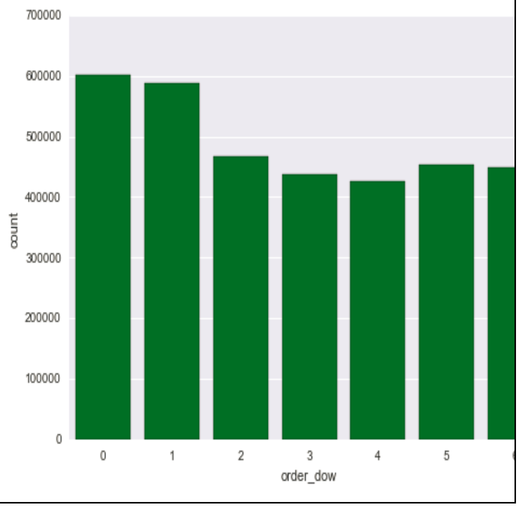
Order placed Time and Day:

Fig 1,2,3 show most orders places on sat Sunday between day time. Specially during 9AM-4PM.

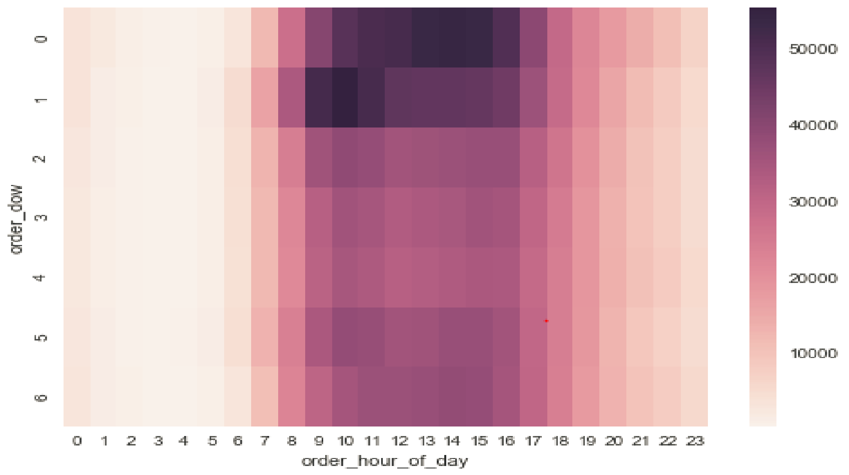
```
<matplotlib.axes._subplots.AxesSubplot at 0xca0fbe0>
```



```
<matplotlib.axes._subplots.AxesSubplot at 0xcf635c0>
```



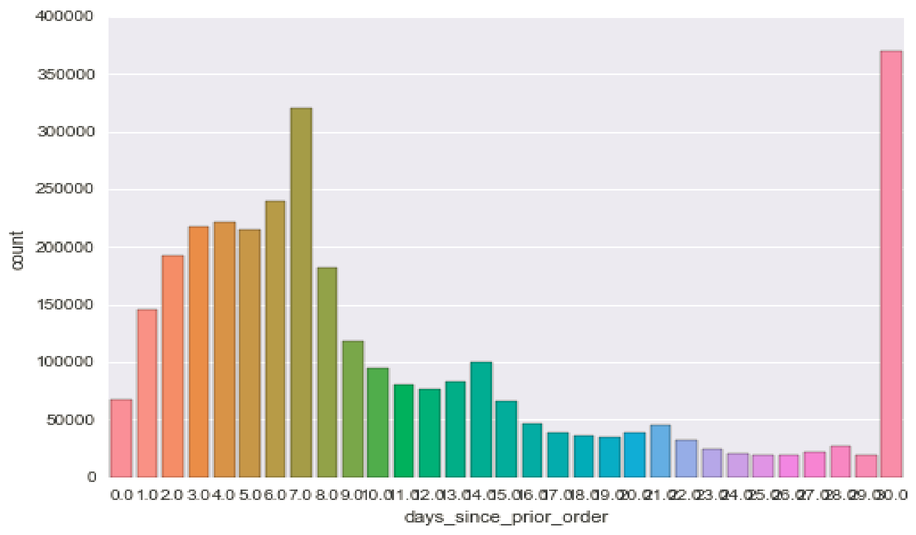
```
<matplotlib.axes._subplots.AxesSubplot at 0xc8471d0>
```



Frequecy of Orders:

We can see that people order every week, so every 7th day. There is also small peak at day 14 and 21.

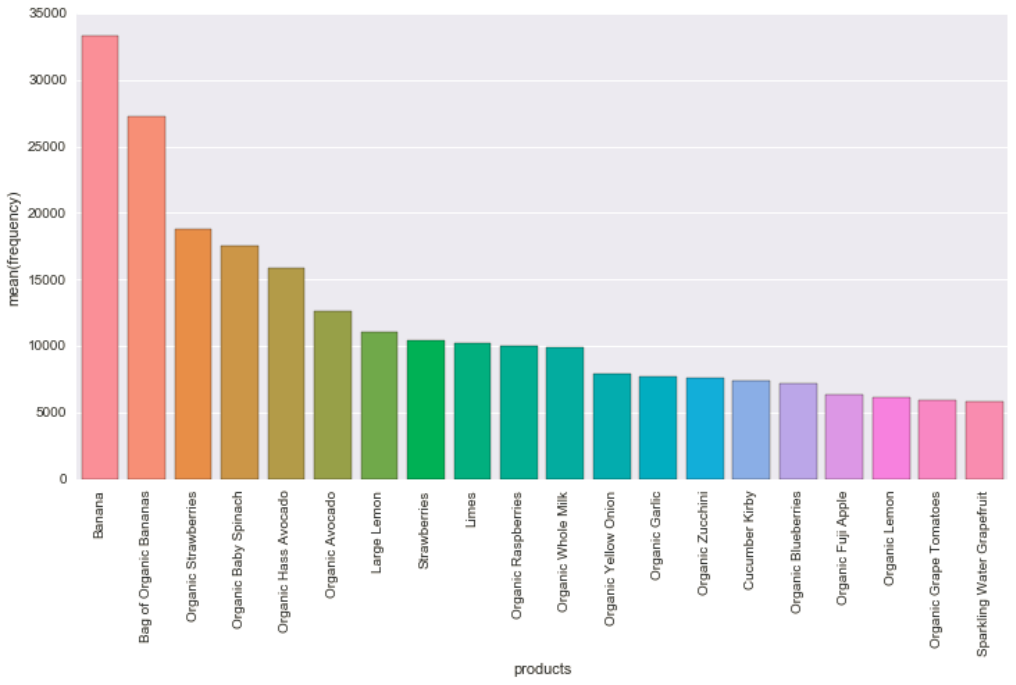
```
<matplotlib.axes._subplots.AxesSubplot at 0x2970d080>
```



<u>Frequently brought Products:</u>

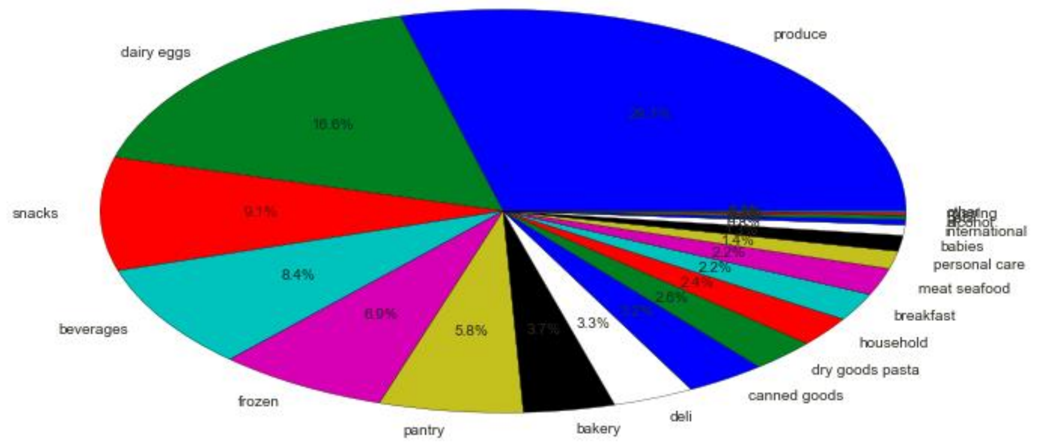
We see that Banana and most of organic fruits and vegetables are brought frequently.

```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19]), <a list of 20 Text xticklabel objects>)
```



<u>Department wise distribution of ordered products:</u>

Below pie chart shows how frequently customer purchase product from each department. Produce and egg dairy being highest of them.



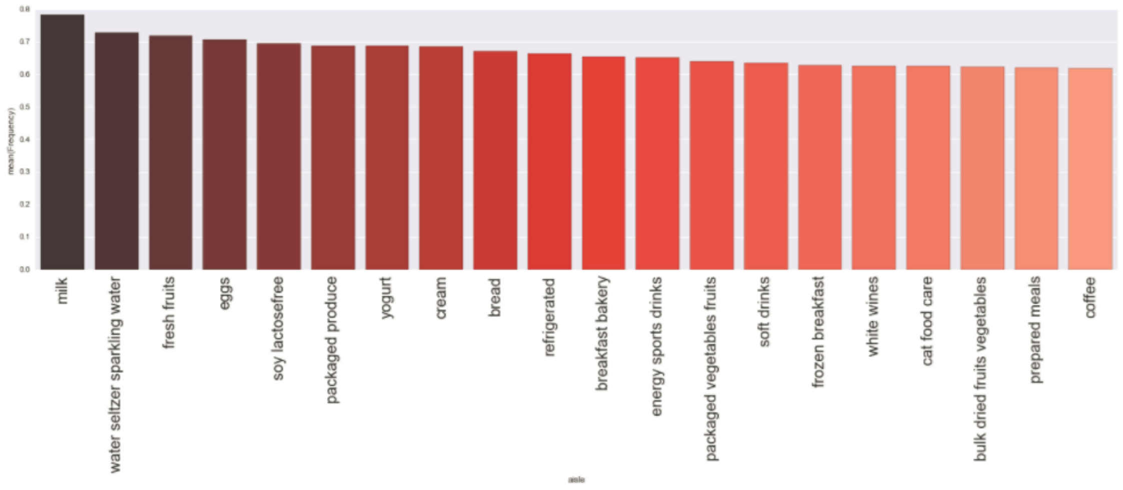
<u>Reordered Product Analysis:</u>

This is important analysis as we need to predict next cart product considering previously reorder products by each user.

Considering reorder products below chats shows most visited **aisles**:

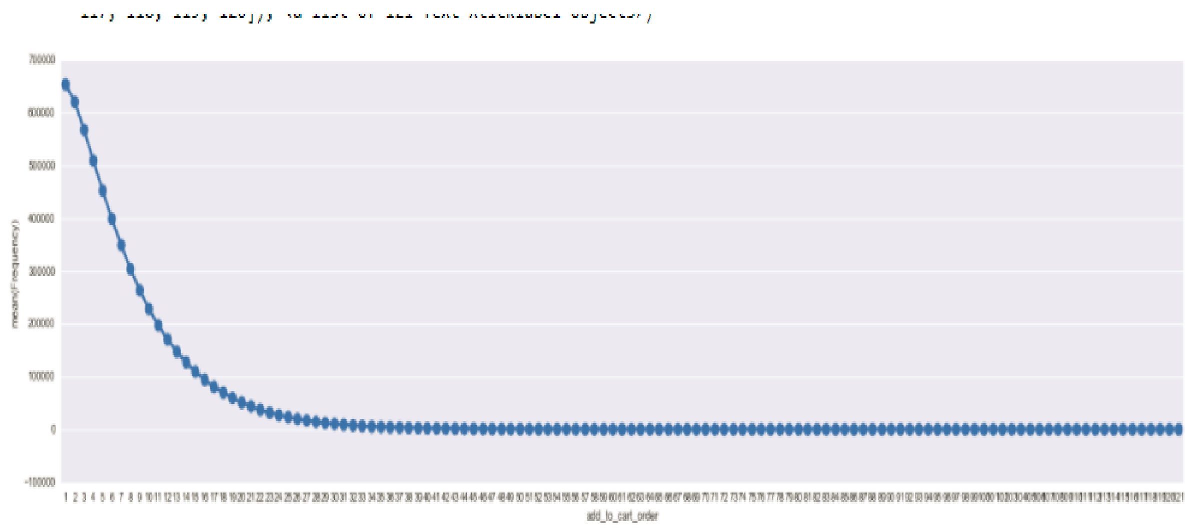
Milk, water, fresh fruits and eggs aisle has most product reordered

```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19]), <a list of 20 Text xticklabel objects>)
```



<u>Relation between Add to cart order and Reorder</u>

Looks like item which are added to cart first are the one reordered most.



Other thing that I found out using data exploration is, reorder ratio is quite high early morning that any other time of the day.

Important features:

After performing Exploratory data analysis, I found below feature which are important, this include some of the manually created features too:

- Users Related features:
    1. User total orders
    2. Users total items
    3. Total distinct items
    4. Users average days between the orders
    5. Users average basket size
- Product Related features:
    6. Aisle ID
    7. Department ID
    8. Total order count of products
    9. Total reorder count of products
    10. Reorder rate
- UserXProduct Related features:
    11. User number of order per products
    12. Users order rate

13. User product last order ID
14. User product average position in cart
15. Users product reorder rate
16. User product day since last order
17. User product order same day as last

Handling missing data:

Thankfully the data obtained it clean data with no missing values.

Algorithms and Techniques

The machine learning technique used is **Ensemble Learning.** An ensemble approach involves building a collection of models, trained on subsets of data, which are then combined to provide a robust model for classification or prediction. The basic idea is that each model provides a marginal amount of new information; when many of these models are combined, each providing some small incremental improvement to your prediction, you end up with a high-performing model.

BOOSTING:

The idea of boosting came out of the idea of whether a weak learner can be modified to become better. A weak hypothesis or weak learner is defined as one whose performance is at least slightly better than random chance. A weak hypothesis or weak learner is defined as one whose performance is at least slightly better than random chance.

HOW GRADIENT BOOSTING WORKS

Gradient boosting involves three elements:

1. A loss function to be optimized.
2. A weak learner to make predictions.
3. An additive model to add weak learners to minimize the loss function.

1. Loss Function

The loss function used depends on the type of problem being solved. It must be differentiable, but many standard loss functions are supported and you can define your own. For example, regression may use a squared error and classification may use **logarithmic loss.**

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

2. Weak Learner

Decision trees are used as the weak learner in gradient boosting.

Specifically, regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and "correct" the residuals in the predictions. Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss. It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes. This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

3. Additive Model

Trees are added one at a time, and existing trees in the model are not changed. **A gradient descent** procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss.)

Generally, this approach is called functional gradient descent or gradient descent with functions. The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model.

A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

The classifier I am using is **XGBoost (eXtreme Gradient Boosting), it** is an advanced implementation of gradient boosting algorithm.

Benchmark:

To create an initial benchmark for the classifier, I assigned zero to all Orderid Productid pair with initial Accuracy score: 0.9022, F-score: 0.9202**.** My aim is to develop a model with F1 score and accuracy greater than the initial baseline classifier.

Methodology

Data Preprocessing:

Data Preprocessing involve Data cleaning + Feature Engineering

Data we got from Instacart is clean data with no missing values or outliers. So most of the data preprocessing was in order to perform feature engineering.

Feature engineering: Feature Engineering selects the right attributes to analyze. Here I spent lot of time acquiring domain knowledge of sales and ecommerce industry to select and create attributes that make machine learning algorithms work. Feature Engineering process includes:

1. Brainstorming for new features
2. Feature selection
3. Validation of how the features work with your model
4. Improvement of features if needed
5. Return to brainstorming / creation of more features until the work is done

Feature engineering was the most important and most time consuming step. I have added **USER, PRODUCT and USERXPRODUCT features** to dataset. File used in data preparation are **create_users.py**, **create_products.py**, **create_userXproducts.py** and finally all these features are merged into **merge_features.py** file. After multiple trials final feature that I have selected to uses which creating models are:

features_to_use = ['**user_total_orders**', '**user_total_items**', '**total_distinct_items**',

'**user_average_days_between_orders**', '**user_average_basket**',

'**order_hour_of_day**', '**days_since_prior_order**', '**days_since_ratio**',

'**aisle_id**', '**department_id**', '**product_orders**', '**product_reorders**',

'**product_reorder_rate**', '**UP_orders**', '**UP_orders_ratio**',

'**UP_average_pos_in_cart**',

'**UP_reorder_rate**','**UP_orders_since_last**','**UP_delta_hour_vs_last**']

Implementation:

Implementation can be split into following stages:

1. Implementing baseline model
2. Implementing classifier training

BASELINE MODEL:

Looking at the distribution of classes (Order product pair present in next order vs not present) it's clear most products are not present in next orders. This can greatly affect accuracy, since we could simply say "product not present in next order" and generally be right, without ever looking at the data! Making such a statement would be called naive, since we have not considered any information to substantiate the claim. So I use naïve predictor as my Baseline Model. I am considering ordered productid pair absent that is class 0 for all the data. Baseline model is implemented in baselinemodel.py as Naïve Predictor.

Code:

```
dtrain_predictions = [0] * df_train.shape[0]
TP = label.count() - np.sum(label)  # Counting the ones as this is the naive case. Note that 'income' is the 'income_raw' data
# encoded to numerical values done in the data preprocessing step.
FP = label.count() - TP  # Specific to the naive case

TN = 0  # No predicted negatives in the naive case
FN = 0  # No predicted False negatives in the naive case

accuracy = float(TP) / float(label.count())
recall = 1
precision = accuracy
beta = 0.5

fscore = (1 + 0.25) * (precision * recall) / ((0.25 * precision) + recall)

# Print the results
print "Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(accuracy, fscore)
```

>> **Naive Predictor: [Accuracy score: 0.9022, F-score: 0.9202]**

CLASSIFIER:

While creating classifier I had challenge mainly with execution time. Some of the classifier I tried applying to data set like Regression classifier, Decision Tree classifier or Adaboost classifier took like days to run on my 8Gb ram and subset of data. So for this project I am using Kaggle competition winner XGBOOST classifier.

XGBOOST classifier parameter:

The overall parameters have been divided into 3 categories by XGBoost authors:

General Parameters: Guide the overall functioning

Booster Parameters: Guide the individual booster (tree/regression) at each step

Learning Task Parameters: Guide the optimization performed

GENERAL PARAMETERS

These define the overall functionality of XGBoost.

- **booster [default=gbtree]**
  Select the type of model to run at each iteration. It has 2 options:
  gbtree: tree-based models
  gbliner: linear models
- **silent [default=0]:**
  Silent mode is activated is set to 1, i.e. no running messages will be printed.
  It's generally good to keep it 0 as the messages might help in understanding the model.
- **nthread [default to maximum number of threads available if not set]**
  This is used for parallel processing and number of cores in the system should be entered.

BOOSTER PARAMETERS

Though there are 2 types of boosters, I'll consider only **tree booster** here because it always outperforms the linear booster and thus the later is rarely used.

- **eta [default=0.3]**: Analogous to learning rate in GBM.
  Makes the model more robust by shrinking the weights on each step. Typical final values to be used: 0.01-0.2
- **min_child_weight [default=1]**
  Defines the minimum sum of weights of all observations required in a child.
  This refers to min "sum of weights" of observations while GBM has min "number of observations".
  Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
  Too high values can lead to under-fitting hence, it should be tuned using CV.
- **max_depth [default=6]** The maximum depth of a tree, same as GBM.

Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

- **max_leaf_nodes**
  The maximum number of terminal nodes or leaves in a tree.
  Can be defined in place of max_depth. Since binary trees are created, a depth of 'n' would produce a maximum of 2^n leaves.
- **gamma [default=0]**
  A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
- **subsample [default=1]**
  Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
  Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
  Typical values: 0.5-1
- **colsample_bytree [default=1]**
  Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
  Typical values: 0.5-1
- **scale_pos_weight [default=1]**
  A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

LEARNING TASK PARAMETERS

These parameters are used to define the optimization objective the metric to be calculated at each step.

- **objective [default=reg:linear]**
  This defines the loss function to be minimized. Mostly used values are:
  binary:logistic –logistic regression for binary classification, returns predicted probability (not class)
- **eval_metric [default according to objective]**

  The metric to be used for validation data.

  The default values are rmse for regression and error for classification.

  Typical values are:

  rmse – root mean square error

mae – mean absolute error

logloss – negative log-likelihood

error – Binary classification error rate (0.5 threshold)

merror – Multiclass classification error rate

mlogloss – Multiclass logloss

auc: Area under the curve

- **seed [default=0]**

  The random number seed.

So I used **default parameter** to start with and train my model on it. Model took almost 2 days to train on my 8gb RAM.

Model Results: XGB Predictor: [Accuracy score: 0.9089, F-score: 0.9745]

REFINEMENT:

I used Gridsearch to tune XGBOOST classifier.

Code:

```
param_test1 = {
  'min_child_weight': range(1, 6, 2)
}
param_test2 = {
  'n_estimators: range(50, 200, 10)
}
param_test3 = {
  'gamma': [i / 10.0 for i in range(0, 5)]
}
param_test4 = {
  'subsample': [i / 10.0 for i in range(6, 10)],
  'colsample_bytree': [i / 10.0 for i in range(6, 10)]
```

}

param_test5 = {

  'max_depth': [1,3,5,7,9]

 }

gsearch1 = GridSearchCV(estimator=XGBClassifier(learning_rate=0.5, n_estimators=140, max_depth=9,min_child_weight=1, gamma=0.2, subsample= 0.9, colsample_bytree=0.6,objective='binary', nthread=4, scale_pos_weight=1,seed=27),param_grid=param_test4/3/2/1, scoring='roc_auc', n_jobs=4, iid=False, cv=5)

gsearch1.fit(X_train[:10000], y_train[:10000])

print gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_

Final Model After tuning has higher accuracy and F1 score and less time required to run the model.

XGB Predictor: [Accuracy score: 0.9117, F-score: 0.9746]

Result

Model Evaluation and Validation

During development, a validation set was used to evaluate the model.

The final model, hyperparameters and features to use were chosen because they performed the best among the tried combination.

Final model has following parameters:

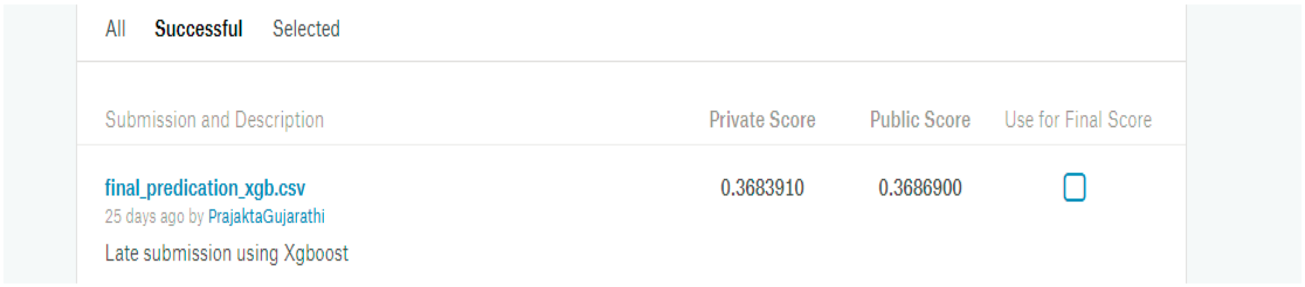
#finally after tunning all the parameters run classifier on entire training set.

xgb1 = XGBClassifier(learning_rate=0.2, n_estimators=140, max_depth=9, min_child_weight=1, gamma=0.2, subsample= 0.9, colsample_bytree=0.6, objective='binary:logistic', nthread=4, scale_pos_weight=1, seed=27)

xgb1.fit(X_train,y_train)

To verify how well the model is performing I applied model to test data from order.csv with eval as Test. Output file in the form of csv is final_prediction_xgb.csv is tested on Kaggle website https://www.kaggle.com/c/instacart-market-basket-analysis.

You can see the score for competition in fig below:



All Successful Selected

Submission and Description	Private Score	Public Score	Use for Final Score
final_predication_xgb.csv 25 days ago by PrajaktaGujarathi Late submission using Xgboost	0.3683910	0.3686900	☐

JUSTIFICATION:

Naive Predictor: [Accuracy score: 0.9022, F-score: 0.9202]

XGB Predictor un-tuned: [Accuracy score: 0.9089, F-score: 0.9745] Execution time: 1573.18 minutes

XGB Predictor: [Accuracy score: 0.9117, F-score: 0.9746] Execution time: 468.17 minutes

Naïve Predictor has accuracy greater than 90% as avg cart size is 17.16, so most of the products are not going to present in next order.

Un-tunned model has accuracy 90.89 with high F1 score of 0.9745. But execution time taken for this if too high, tuning a model I am getting 91.17% and F1 score similar as un-tuned model 0.9746. But due to lower execution time I am going to consider tuned model as my final model.

Conclusion

Reflection

The process used for this project can be summarized using the following steps:

1. An initial problem, public dataset was found using kaggle competition.

2. The data was downloaded and preprocessed (adding many manual features)

3. A benchmark was created for the classifier using naïve predicator.

4. The classifier was trained using the data (multiple times, until a good set of parameters were found)

5. By using predicated probability for each orderId productId combination, select product which has higher probability than threshold as product that will be in next order for given customer. Result are stored in csv file.

I found steps 2 and 4 the most difficult, as I had to brainstorm to create new features. As for the most interesting aspects of the project, I'm very glad that I found the xgboost model which is way faster than any other supervise learning model. I also tried using light gradient boosting model but it was overfitting in my case. I learnt concept of Ensemble learning and boosting in depth. I'm sure it will be useful for later projects/experiments.

Improvement

To achieve the accurate model, create more manual features is essential. Also running a model on cloud servers with higher RAM will reduce the processing time. Which would, in turn, enable the following features:

❖ Better Prediction results with F1 score as high as 0.99

❖ Reduction in processing time.

The model prediction accuracy and F1 score could also be improved significantly by using Model Stacking (also called meta ensemble) is a model ensemble technique used to combine information from multiple predictive models to generate a new model. Often times the stacked model (also called 2nd-level model) will outperform each of the individual models due its smoothing nature and ability to highlight each base model where it performs best and discredit each base model where it performs poorly.

Reference Links:

https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

https://www.jeremyjordan.me/ensemble-learning/

https://stats.stackexchange.com/questions/173390/gradient-boosting-tree-vs-random-forest

http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/

https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/