# Instacart Basket Prediction

Eitan Angel

June 27, 2019

## Contents

## List of Figures



## List of Tables



## List of Listings

# 1 Introduction

## 1.1 Problem: Predict the Items Instacart Costumers Purchase

Instacart is a grocery-on-demand start-up which, in 2017, released a dataset containing 3 million orders from 200,000 (anonymized) users. A now-completed Kaggle competition asked entrants to predict which previously purchased products would be in a consumer's next order. In addition, Instacart would like to develop recommender systems that could predict which products a user would buy for the first time and which products would be added to a user's cart in future orders.

Section 6 makes predictions according to the Kaggle competition, that is, from a given user's set of previously purchased products, predict which of those products the user will ultimately order. The model includes different product application variants, for instance

- more precise predictions to, for example, autopopulate a user's cart
- top-$N$ most-likely products a particular user will purchase to, for example, display on a web page of fixed results

## 1.2 Data: The Instacart Online Grocery Shopping Dataset

The Instacart public dataset release is a sample of 3 million orders from 200,000 anonymized users released in 2017. For each user, Instacart has provided between 4 and 100 of their orders, including the intraorder sequence in which products were purchased, the week and hour of the day the order was placed, the relative time between orders, and the grocery store department to which each product belongs. A blog post by Instacart provides some information about the dataset and a Kaggle competition provides additional details.

All data was obtained from the Kaggle competition website. The dataset is a relational set of `.csv` files which describe customer orders over relative times. Each entity (customer, order, department, etc.) has a unique id.

## 1.3 Outline: The Data Map Composition

The structure of the subsequent sections corresponds to the maps of the composition

$$\text{RawData} \xmapsto{\mathsf{P}} D \xmapsto{\mathsf{F}} X \xmapsto{\widehat{\mathsf{RFC}}_{\mathbf{n_0}}} \hat{y}^\star \xmapsto{\mathsf{TopN}} \hat{y}. \tag{1.1}$$

as well as the domain and image. In section 2 we perform exploratory data analysis on RawData, the dataset described in section 1.2. Section 3 describes the train/test partition $\mathsf{P}$. The feature design map $\mathsf{F}$ of section 4 assembles the partitioned raw data $D$ into a design matrix $X$ which has user-product pairs as its rows. The random forest classifier $\mathsf{RFC}_{\mathbf{n_0}}$ of section 5 makes a probabilistic prediction $\hat{y}^\star$ of user-product pairs appearing in the user's ultimate order. In particular, section 5.4 locates optimal hyperparameters $\mathbf{n_0}$ for the random forest classifier. We devise various $\mathsf{TopN}$ schemes in section 6 to assign binary predictions $\hat{y}$. Finally, we exhibit a data visualization utility in section 7 to compare the predictions $\hat{y}$ against the true values $y_{\text{test}}$.

In addition, section 8 describes a wishlist of potential improvements to the project. The list of features is in section 9.1: Appendix I; the Jupyter notebooks containing code for this project are referenced in section 9.2: Appendix II.

## 2 Exploration

Section 9.2.1 references the code described in this section.

Kaggle posts Simple Exploration Notebook - Instacart and Exploratory Analysis - Instacart, both of which the author viewed while deciding on the dataset and project, were very instructive. First, we display the `*.csv` files forming the RawData. Displayed in this section are number of plots and tables showing the most (and sometimes least) popular products, aisles, departments, and times for (re)orders:

- `order_products*.csv`
- `orders.csv`
- Dictionaries between `*_id` and `*_name`
- Merged DataFrame of RawData
- Reorder rates for products and aisles
- The most popular products, aisles, and departments
- Heatmap of order times
- Histogram of order frequency
- Histogram of order counts by user
- Histogram of basket sizes

Table 1: `order_products*.csv`

| order_id | product_id | add_to_cart_order | reordered |
|---|---|---|---|
| 1 | 49302 | 1 | 1 |
| 1 | 11109 | 2 | 1 |
| 1 | 10246 | 3 | 0 |
| 1 | 49683 | 4 | 0 |
| 1 | 43633 | 5 | 1 |

While only `order_products__train.csv` is displayed, `order_products__prior.csv` has the same structure.

Table 2: `orders.csv`

| order_id | user_id | eval_set | order_number | order_dow | order_hour... | days_since... |
|---|---|---|---|---|---|---|
| 2539329 | 1 | prior | 1 | 2 | 8 | NaN |
| 2398795 | 1 | prior | 2 | 3 | 7 | 15.0 |
| 473747 | 1 | prior | 3 | 3 | 12 | 21.0 |
| 2254736 | 1 | prior | 4 | 4 | 7 | 29.0 |
| 431534 | 1 | prior | 5 | 4 | 15 | 28.0 |

The abbreviated column headers are `order_hour_of_day` and `days_since_prior_order`.

Table 3: Dictionaries between `*_id` and `*_name`

(a) `products.csv`

| product_id | product_name | aisle_id | department_id |
|---|---|---|---|
| 1 | Chocolate Sandwich Cookies | 61 | 19 |
| 2 | All-Seasons Salt | 104 | 13 |
| 3 | Robust Golden Unsweetened Oolong Tea | 94 | 7 |
| 4 | Smart Ones Classic Favorites Mini Rigatoni Wit... | 38 | 1 |
| 5 | Green Chile Anytime Sauce | 5 | 13 |

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

(b) `aisles.csv`

| department_id | department |
|---|---|
| 1 | frozen |
| 2 | other |
| 3 | bakery |
| 4 | produce |
| 5 | alcohol |

(c) `departments.csv`

Table 4: Merged DataFrame of RawData

| | | | dow | hr | days | re | product_name | aisle | department |
|---|---|---|---|---|---|---|---|---|---|
| . | . | . | | | | | | | |
| 1 | 1 | 1 | 2 | 8 | NaN | 0 | Soda | soft drinks | beverages |
| | | 2 | 2 | 8 | NaN | 0 | Organic Unsweetened … | soy lactosefree | dairy eggs |
| | | 3 | 2 | 8 | NaN | 0 | Original Beef Jerky | popcorn jerky | snacks |
| | | 4 | 2 | 8 | NaN | 0 | Aged White Cheddar Popcorn | popcorn jerky | snacks |
| | | 5 | 2 | 8 | NaN | 0 | XL Pick-A-Size Paper Towel … | paper goods | household |
| | 2 | 1 | 3 | 7 | 15.0 | 1 | Soda | soft drinks | beverages |
| | | 2 | 3 | 7 | 15.0 | 0 | Pistachios | nuts seeds … | snacks |
| | | 3 | 3 | 7 | 15.0 | 1 | Original Beef Jerky | popcorn jerky | snacks |
| | | 4 | 3 | 7 | 15.0 | 0 | Bag of Organic Bananas | fresh fruits | produce |
| | | 5 | 3 | 7 | 15.0 | 1 | Aged White Cheddar Popcorn | popcorn jerky | snacks |

Merge of `order_products*.csv` with `orders.csv` on `order_id` upon a reindexing (and deletion of column `order_id`). The MultiIndex is `(user_id, order_number, add_to_cart_order)` while `order_dow, order_hour_of_day, days_since_prior_order, and reordered` are abbreviated `dow, hr, days, and re`, respectively. The total number of rows – that is, the total number of items ordered – is 33,819,106.

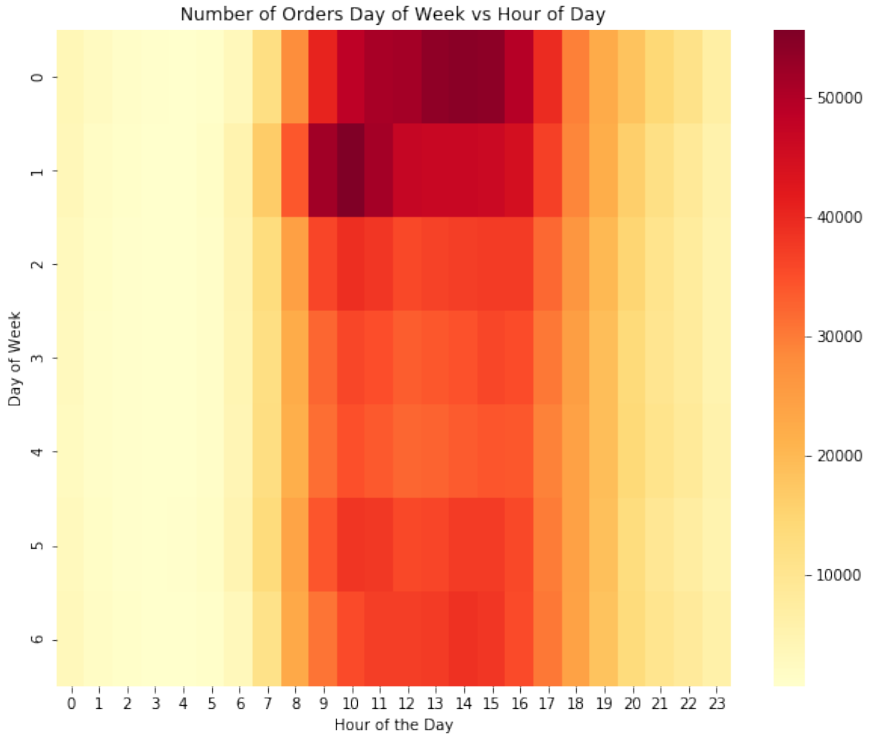



Figure 1: Saturday afternoon and Sunday morning are the most popular time to make orders.

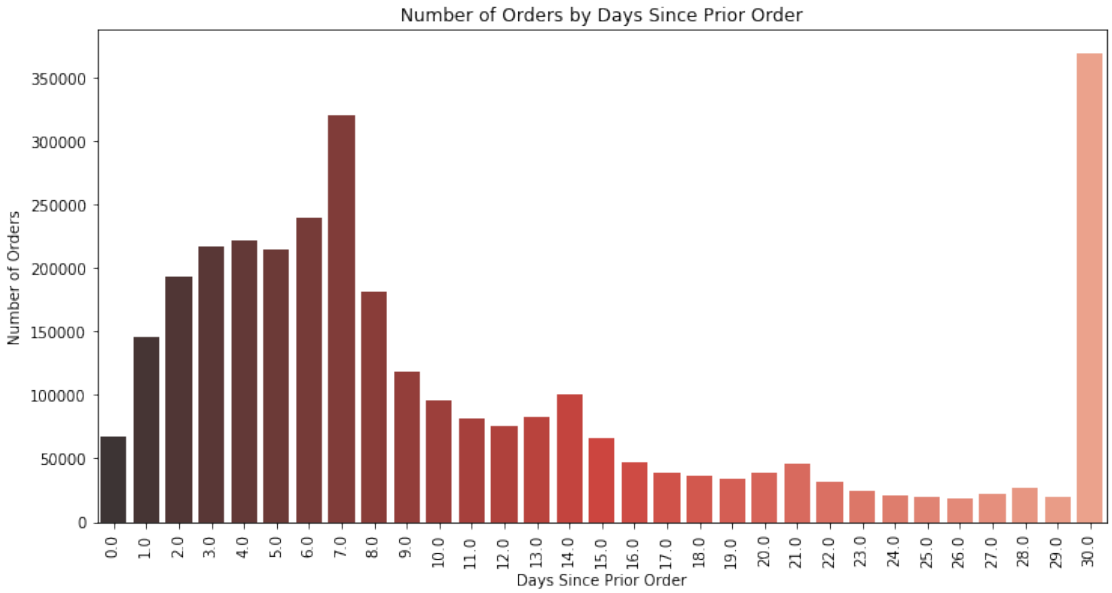



Figure 2: The most popular relative time between orders is monthly (30 days), but there are "local maxima" at weekly (7 days), biweekly (14 days), triweekly (21 days), and quadriweekly (28 days).

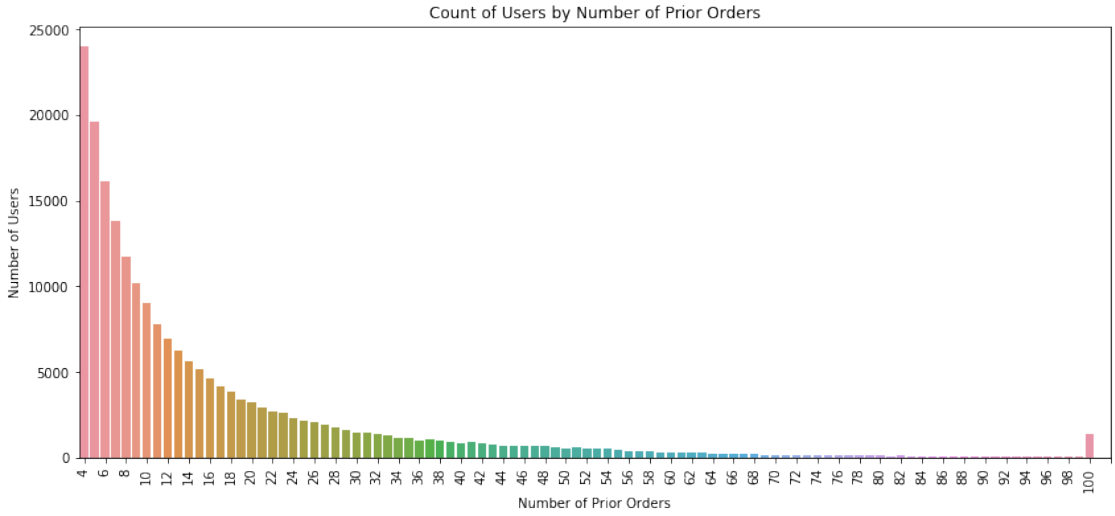



Figure 3: Users have between 4 – 100 orders. Those users in the dataset with 100 orders seem to have at least 100 orders and we have only their most recent 100 orders.

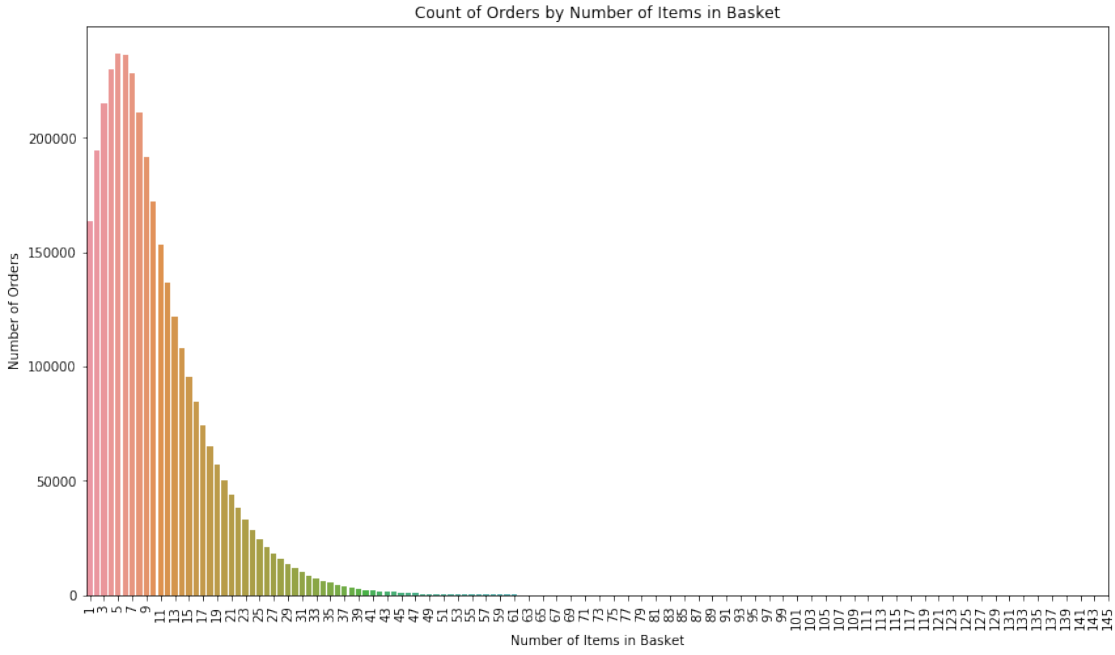



Figure 4: As one should expect, this distribution is right-skew. The mode basket size is 5.

Table 5: Reorder rates for products and aisles. 59.01% of purchases were reorders.

| product_name | reorder |
|---|---|
| Raw Veggie Wrappers | 0.9420 |
| Serenity Ultimate Extrema ... | 0.9333 |
| Chocolate Love Bar | 0.9215 |
| Bars Peanut Butter | 0.8985 |
| Soy Crisps Lightly Salted | 0.8955 |
| Maca Buttercups | 0.8942 |
| Benchbreak Chardonnay | 0.8918 |
| Organic Blueberry B Mega | 0.8888 |
| Sparking Water | 0.8870 |
| Fragrance Free Clay ... | 0.8702 |

(a) The most reordered products with at least 50 orders.

| aisle | department | reorder |
|---|---|---|
| milk | dairy eggs | 0.7818 |
| water seltzer... | beverages | 0.7299 |
| fresh fruits | produce | 0.7188 |
| eggs | dairy eggs | 0.7063 |
| soy lactosefree | dairy eggs | 0.6923 |
| ... | ... | ... |
| beauty | personal care | 0.2128 |
| first aid | personal care | 0.1958 |
| kitchen supplies | household | 0.1948 |
| baking supplies ... | pantry | 0.1675 |
| spices seasonings | pantry | 0.1529 |

(b) The most and least reordered from aisles.

| product_name | count |
|---|---|
| Banana | 491291 |
| Bag of Organic Bananas | 394930 |
| Organic Strawberries | 275577 |
| Organic Baby Spinach | 251705 |
| Organic Hass Avocado | 220877 |
| Organic Avocado | 184224 |
| Large Lemon | 160792 |
| Strawberries | 149445 |
| Limes | 146660 |
| Organic Whole Milk | 142813 |
| Organic Raspberries | 142603 |
| Organic Yellow Onion | 117716 |
| Organic Garlic | 113936 |
| Organic Zucchini | 109412 |
| Organic Blueberries | 105026 |
| Cucumber Kirby | 99728 |
| Organic Fuji Apple | 92889 |
| Organic Lemon | 91251 |
| Organic Grape Tomatoes | 88078 |
| Apple Honeycrisp Organic | 87272 |
| Seedless Red Grapes | 86748 |
| Organic Cucumber | 85005 |
| Honeycrisp Apple | 83320 |
| Organic Baby Carrots | 80493 |
| Sparkling Water Grapefruit | 79245 |

(a) The majority of the 25 most popular products are organic fruits or vegetables.

| aisle_name | count |
|---|---|
| fresh fruits | 3792661 |
| fresh vegetables | 3568630 |
| packaged vegetables fruits | 1843806 |
| yogurt | 1507583 |
| packaged cheese | 1021462 |
| milk | 923659 |
| water seltzer sparkling water | 878150 |
| chips pretzels | 753739 |
| soy lactosefree | 664493 |
| bread | 608469 |

(b) The most popular aisles.

| department_name | total_pct |
|---|---|
| produce | 0.2923 |
| dairy eggs | 0.1665 |
| snacks | 0.0888 |
| beverages | 0.0829 |
| frozen | 0.0690 |
| pantry | 0.0578 |
| bakery | 0.0362 |
| canned goods | 0.0329 |
| deli | 0.0323 |
| dry goods pasta | 0.0267 |
| household | 0.0229 |
| meat seafood | 0.0218 |

(c) The share of the most popular departments.

Table 6: The most popular products, aisles, and departments

# 3 Train-Test Split

Section 9.2.2 references the code described in this section.

It is worth taking a moment to discuss the dataset partition process. Typically, training a model, tuning its hyperparameters, and evaluating its performance are done using a three-way split of the dataset into independent training, cross-validation, and test sets. On the other hand, the random forest algorithm is a bootstrap aggregation (bagging) of decision trees technique that has a convenient property. Given a sample $x$ in the training dataset, $x$ is used to train only about $2/3$ of the decision trees, so that roughly $1/3$ of the trees are trained on data which does not include $x$. An out-of-bag (OOB) estimate uses trees which have not been trained on $x$ to make a prediction for $x$. Beyond a minor simplification in the dataset partition process, exploiting this property can lead to considerable resource advantages, without which, far less of this project could have succeeded using the resources available on the Kaggle Kernel platform. We elaborate on these details and discuss the implications for the hyperparameter search in Section 5.

The *partition map* is a transformation,

$$\begin{aligned} \mathsf{P} : \mathsf{RawData} &\to \mathsf{D}^{\mathrm{DSets}} \\ \mathsf{RawData} &\mapsto \{D_s\}_{s\in\mathrm{DSets}}, \end{aligned} \tag{3.1}$$

where

- $\mathrm{DSets} = \{\mathrm{train}, \mathrm{test}, \mathrm{kaggle}\}$,
- RawData denotes the collection of tables of user orders and baskets in `orders.csv` and `order_products*.csv`,
- RawData is the set of possible collections of tables matching the column and relational structure of RawData,
- P is the unique partition defined by a partition of the set of users, $U$, ordered by, say, `user_id`, into $\{U_s\}_{s\in\mathrm{DSets}}$ where the sklearn.model_selection.train_test_split utility determines the partition between $U_{\mathrm{train}}$ and $U_{\mathrm{test}}$, while $U_{\mathrm{kaggle}}$ consists of users whose `user_id` is in a row matching the condition `eval_set == 'test'` as follows

  $U_{\mathbf{train}}$**:** 80% of the 131,209 users whose ultimate orders are available.

  $U_{\mathbf{test}}$**:** 20% of the 131,209 users whose ultimate orders are available.

  $U_{\mathbf{kaggle}}$**:** The 75,000 users whose ultimate orders are withheld by Kaggle. This project does not explicitly use this set; predictions on kaggle merely serve as a sanity check via submission to Kaggle.

- $\{D_s\}_{s\in\mathrm{DSets}}$ is the image of RawData under P.
- $\mathsf{D}^{\mathrm{DSets}}$ are possible sets of the form $\{D_s\}_{s\in\mathrm{DSets}}$, that is, the collection of possible partitions of members of RawData over DSets.

| eval_set | user_id |
|---|---|
| prior | 206209 |
| train | 131209 |
| test | 75000 |

`eval_set` counts

The list of strings `dsets = ['train', 'test', 'kaggle']`, implements DSets. `users` is a dictionary of lists of `user_ids` keyed by `dsets`, initialized by

```
> users = dict.fromkeys(dsets)
```

so that

| representation | implementation |
|---|---|
| DSets $= \{\text{train}, \text{test}, \text{kaggle}\}$ | `dsets = ['train', 'test', 'kaggle']` |
| $s \in$ DSets | `ds in dsets` |
| $D_s$ | `orders[ds], prior[ds] # P(RawData)` |
| $U_s$ | `users[ds]` |
| $X_s$ | `X[ds]` |
| $y_s$ | `y[ds]` |

Table 8: Notation Dictionary: representation – implementation

```
> [users[ds]  for ds in dsets]
```

implements $\{U_s\}_{s\in\text{DSets}}$. Similarly, this notebook constructs matrices `X[ds]` which implement design matrices $X_s$. Aside from the analogy between dictionary keys and subscripts, the `dict` type offers a coherent way to partition the dataset $D_s$ of user orders and baskets in `orders.csv` and `order_products*.csv` into separate DataFrames `orders[ds]` and `prior[ds]`, respectively, for `ds in dsets` at the outset, so as to avoid potential data leaks. Table 8 is a partial dictionary between mathematical abstractions and instantiations.

We often suppress the dataset decoration in subsequent discussion so that, for example, $X_s$ is denoted $X$ when the objects discussed do not depend on the value of $s$. Ideas to suppress `[ds]` in code are welcome.

# 4 Feature Design

Section 9.2.2 references the code described in this section.

Feature design is a transformation

$$\begin{aligned} \mathsf{F} : \mathsf{D} &\to \mathsf{M}_{n\times m}(\mathbb{R}) \\ D &\mapsto X \end{aligned} \tag{4.1}$$

where $\mathsf{M}_{n\times m}(\mathbb{R})$ is the space of $n \times m$ real-valued matrices and $\mathsf{D}$ is the space of possible user-basket data of the form $D$.

We describe $\mathsf{F}$ explicitly as an $m$-tuple of transformations

$$\mathsf{F}(D) := (\mathsf{f}_1(D), \ldots, \mathsf{f}_m(D)), \tag{4.2}$$

or more succinctly $\mathsf{F} = (\mathsf{f}_j)_1^m$, where

$$\mathsf{f}_j : \mathsf{D} \to \mathbb{R}^n \tag{4.3}$$

calculates the $j^{\text{th}}$ column of $X \in \mathsf{M}_{n\times m}(\mathbb{R})$ via the equivalence $(\mathbb{R}^n)^m \cong \mathsf{M}_{n\times m}(\mathbb{R})$. Features may be implemented as `int`, `bool`, or `float` types though we consider them abstractly as being $\mathbb{R}$-valued.

For most $j \in \{1, \ldots, m\}$, $\mathsf{f}_j$ consists of aggregations, filtrations, arithmetic operations, though at times we employ more complex transformations. For example, some features are computed via an unsupervised learning technique, Latent Dirichlet Allocation (LDA), introduced in [BNJ03]. This is a probabilistic generative model we apply to the matrix of user-product purchase counts.

Although the random forest classifier already sees this data as a column of $X$, the distributional assumptions of LDA can yield a bit more predictive power. We can view this as a simple model-based collaborative filtering technique.

The features – columns of the design matrix $X$ – we group into a few Profiles. This overall structure is inspired by the description of feature design in [LNZ+16], in which the authors describe their approach to a related e-commerce buyer prediction competition. The User Profile, for example, consists of operations and aggregations grouped by user, so that the index for the user profile is $U$, the list of users. The rows of $X$ are not merely users, but user-product pairs, which means that the User Profile is broadcast to the user-product multi-index, $\Gamma$, via a `.join()` operation. That is, the values of the User Profile are repeated across all products in the user-product index for any given user. An analogous statement holds for the Product Profile. Therefore, the User-Product profile will have the features with the greatest information content (and the Aisle and Department profiles the least). A list of Profiles and features is located in Section 9.1.

For each $u \in U$, let $\text{Prod}(u)$ be the ordered sequence of products purchased by $u$ during prior purchases with ordering induced by, say, `product_id`. Rows of $X$ are multi-indexed by

$$\Gamma = ((u, p) \mid u \in U \;\&\; p \in \text{Prod}(u)) \tag{4.4}$$

where the ordering is the lexicographic order defined by the ordering on $U$ and $\text{Prod}(u)$. If we denote $(u, p) \in \Gamma$ by $\gamma$, then we may denote the $\gamma^{\text{th}}$ observation in $X \times y$ by $(x^\gamma, y^\gamma)$. Denote $n_s := |\Gamma_s|$ so that $X_s$ is of size $n_s \times m$.

We were able to achieve as many as $m \approx 50$ features on the Kaggle platform on which the collection of notebooks comprising this project were developed and run. The limiting resource is memory, though the limitation occurs at the `sklearn.ensemble.RandomForestClassifier` calls of subsequent notebooks. Kaggle provides instances with 16GB of memory; the instance provides no virtual memory (disk) so this is a hard limit on memory availability.

| $n_{\text{train}}$ | $n_{\text{test}}$ | $n_{\text{kaggle}}$ | $m$ |
|---|---|---|---|
| 6760791 | 1713870 | 4833292 | 47 |

The model we construct phrases the labels as a vector $y$ of binary classes

$$y^\gamma = \begin{cases} \text{True} & \text{if user } u \text{ purchased product } p \text{ in ultimate order} \\ \text{False} & \text{if user } u \text{ did not purchase product } p \text{ in ultimate order} \end{cases} \tag{4.5}$$

for all $\gamma = (u, p) \in \Gamma$. Recall that $y_{\text{kaggle}}$ is withheld by Kaggle. Our aim is in the following sections is to predict the probability $P((u, p))$ that user $u$ purchases product $p$ in their ultimate order, for all $(u, p) \in \Gamma$ – or at least to predict a rank-ordering of these probabilities.

## 5 Random Forest Classifier

While random forests are well-known, we will review some details in Section 5.1 so as to explain, motivate, and justify the strategy of Section 5.4 and our choice of train-test rather than train-cv-test split in Section 3. We proceed with a discussion of the OOB samples in section 5.2, a list of metrics we examine in section 5.3, a hyperparameter search in section 5.4, and scores for the rank metrics AUC-PR and AUC-ROC in section 5.5.

Definition 2 provides a map of probabilistic predictions

$$\begin{aligned} \widehat{\mathsf{RFC}}_{\mathbf{n_0}} : \mathsf{M}_{n\times m}(\mathbb{R}) &\to [0,1]^n \\ X &\mapsto \hat{y}^\star \end{aligned} \tag{5.1}$$

once we make an optimal choice of hyperparameters $\mathbf{n_0}$. Section 6 discusses methods for mapping

$$\begin{aligned} [0,1]^n &\to \{0,1\}^n \\ \hat{y}^\star &\mapsto \hat{y} \end{aligned} \tag{5.2}$$

## 5.1 Algorithm Review

We begin with a concise review of decision trees to fix notation; Chapters 9.2 and 10.9 of [HTF09] elaborate on the notions mentioned here.

**Definition 1** (Decision Tree)**.** Let $\mathcal{P}_\Lambda(\mathbb{R}^m)$ denote the space of partitions of the feature space $\mathbb{R}^m$ into rectangular regions, $(R_\lambda)_{\lambda\leq\Lambda}$, obtained via recursive binary splitting by hyperplanes perpendicular to coordinate axes. $\Lambda$ is the *leaf node count* and we may refer to $\lambda$ as the *leaf index*. Define the *space of binary decision tree parameters over* $\mathbb{R}^m$ by

$$\mathcal{B}(\mathbb{R}^m) := \bigcup_{\Lambda\in\mathbb{N}} \left\{ (R_\lambda, \beta_\lambda) \,|\, (R_\lambda) \in \mathcal{P}_\Lambda(\mathbb{R}^m), (\beta_\lambda) \in \{0,1\}^\Lambda \right\}. \tag{5.3}$$

Hence a *tree parameter* $\Theta \in \mathcal{B}(\mathbb{R}^m)$ is a partition $(R_\lambda) \in \mathcal{P}_\Lambda(\mathbb{R}^m)$ for some $\Lambda \in \mathbb{N}$ with a (boolean) class value $\beta_\lambda$ associated to each $R_\lambda$. A *decision tree* on $\mathbb{R}^m$ is formally

$$\begin{aligned} \mathsf{T}(\cdot\,;\Theta) : \mathsf{M}_{n\times m}(\mathbb{R}) &\to \{0,1\}^n \\ \mathsf{T}(x;\Theta) &:= \sum_\lambda \beta_\lambda \mathbf{1}_{R_\lambda}(x) \end{aligned} \tag{5.4}$$

for $x \in \mathbb{R}^m$ and $\mathbf{1}_R$ the indicator function on $R \subset \mathbb{R}^m$.

Implementations of decision trees and random forests employ many hyperparameters to control the tree topology. For example, the `sklearn.tree.DecisionTreeClassifier` and `sklearn.ensemble.RandomForestClassifier` APIs expose

- the numbers of variables to consider during splits ($n_{\texttt{max\_features}}$),
- minimum samples in leaf nodes ($n_{\texttt{min\_samples\_leaf}}$),
- maximum tree depth ($n_{\texttt{max\_depth}}$ ), etc.

Collect the hyperparameters into a vector,

$$\mathbf{n} = (n_{\texttt{max\_features}}, n_{\texttt{min\_samples\_leaf}}, n_{\texttt{max\_depth}}, \ldots). \tag{5.5}$$

The space, $\mathbf{H}$, of all such $\mathbf{n}$ parametrize a family of probability distributions, $\mathcal{T}(\mathbf{n})$, on $\mathcal{B}(\mathbb{R}^m)$, from which we sample tree parameters.

Given hyperparameters $\mathbf{n} \in \mathbf{H}$, to *grow a decision tree classifier,* $\hat{\mathsf{T}}_{\mathbf{n}}(x;\Theta)$*, on a training set* $X_{train} \times y_{train}$ is to find tree parameters $\hat{\Theta}$ via empirical risk minimization

$$\hat{\Theta} = \arg\min_\Theta \sum_{\lambda\leq\Lambda} \sum_{x^\gamma\in R_\lambda} L(y^\gamma, \beta_\lambda) \tag{5.6}$$

where $L$ is a loss function (e.g. ordinary least squares) whose precise definition depends on the implementation. In addition, computing $(R_\lambda)$ is combinatorially difficult so we must settle for $\tilde{\Theta}$, an approximation of the estimate. Practically, this means a procedure which recursively repeats the steps

1. Select $n_{\texttt{max\_features}}$ features at random from the $m$ features.
2. Use an information criterion such as Gini impurity to determine the best variable and split point among the $n_{\texttt{max\_features}}$ features.
3. Split the node into two daughter nodes.

until the components of $\mathbf{n}$ determine the procedure stops. Section 1.10.7.1 of the scikit-learn documentation has more precise implementation details.

Bootstrap aggregation, or bagging, is essentially an averaging technique which is useful for high-variance, low-bias estimators, such as decision trees. Random forests are a modification of bagging trees which builds de-correlated trees by selecting a random subset of input variables from which to form decision nodes – that is, allowing $n_{\texttt{max\_features}}$ to take values less than $m$.

Following the first definition of [Bre01] and the exposition of Chapter 15 of [HTF09], we can give a high-level overview of the random forest classifier algorithm as follows:

**Definition 2** (Random Forest)**.** Given a number of decision trees, $B$, and hyperparameters $\mathbf{n} \in \mathbf{H}$, draw bootstrap samples $(\mathbf{Z}_b^*)_{b \leq B}$ from $X_{\text{train}} \times y_{\text{train}}$ of size $n_{\text{train}}$ and grow decision tree classifiers $\left(\hat{\mathsf{T}}_{\mathbf{n}}(x; \Theta_b)\right)_B$ on $(\mathbf{Z}_b^*)_B$.

The resulting *random forest classifier* is the predictor

$$\begin{aligned} \widehat{\mathsf{RFC}}_{\mathbf{n}} &: \mathsf{M}_{n \times m}(\mathbb{R}) \to \{0,1\}^n \\ \widehat{\mathsf{RFC}}_{\mathbf{n}}(x) &:= \text{majority-vote}\left\{\left(\hat{\mathsf{T}}_{\mathbf{n}}\left(x; \Theta_b\right)\right)_B\right\}. \end{aligned} \tag{5.7}$$

In fact, the implementation details of `RandomForestClassifier` differ from the above in that scikit-learn averages probabilistic predictions instead of using majority-vote; therefore $\hat{\mathsf{T}}$, $\widehat{\mathsf{RFC}}$, and $\widehat{\mathsf{OOB}}$ defined in section 5.2 below, are probability predictions rather than class predictions. In other words, the ranges of the maps (5.4), (5.7), and (5.10) are ordered $n$-tuples of intervals $[0,1]$ instead of ordered $n$-tuples of boolean values $\{0,1\}$:

$$\hat{\mathsf{T}},\ \widehat{\mathsf{RFC}},\ \widehat{\mathsf{OOB}} : \mathsf{M}_{n \times m}(\mathbb{R}) \to [0,1]^n \tag{5.8}$$

We discuss various schemes for mapping $[0,1]^n \to \{0,1\}^n$ in Section 6. In the `sklearn.ensemble.RandomForestClassifier` API, $B$ is called `n_estimators`.

## 5.2 Out of Bag Samples

Random forests have a convenient property which we exploited in hyperparameter tuning. In Definition 2, the bootstrap samples are chosen so that approximately 1/3 of instances are left out. Therefore, for any $\gamma \in \Gamma_{\text{train}}$, the observation $(x^\gamma, y^\gamma) \in X_{\text{train}} \times y_{\text{train}}$ is left out of approximately 1/3 of $\left(\mathbf{Z}_b^*\right)$.

**Definition 3.** Given a random forest classifier $\widehat{\mathsf{RFC}}_{\mathbf{n}}$, let

$$B(\gamma) = \{b \in \{1, \ldots, B\} \mid (x^\gamma, y^\gamma) \notin \mathbf{Z}_b^*\} \tag{5.9}$$

be the set of indices in $B$ for which observation $\gamma$ is *not* in the bootstrap sample $\mathbf{Z}_b^*$.

For any $\gamma \in \Gamma_{\text{train}}$, the *out-of-bag (OOB) classifier induced by* $\widehat{\mathsf{RFC}}_{\mathbf{n}}$ is

$$\begin{aligned} \widehat{\mathsf{OOB}}_{\mathbf{n}} &: \mathsf{M}_{n \times m}(\mathbb{R}) \to \{0,1\}^n \\ \widehat{\mathsf{OOB}}_{\mathbf{n}}(x^\gamma) &:= \text{majority-vote}\left\{\left(\hat{\mathsf{T}}_{\mathbf{n}}\left(x^\gamma; \Theta_b\right)\right)_{b \in B(\gamma)}\right\} \end{aligned} \tag{5.10}$$

that is, the prediction of the random forest constructed by averaging only those trees corresponding to $\mathbf{Z}_b^*$ in which $(x^\gamma, y^\gamma)$ does *not* appear. The out-of-bag (OOB) error is the error of $\text{OOB}_\mathbf{n}$ on $X_{\text{test}} \times y_{\text{test}}$.

The motivation for using OOB estimates instead of $N$-fold cross-validation is two-fold. First, this simplifies the phases of the overall project. Second, $N$-fold cross-validation with `sklearn.model_selection.GridSearchCV` requires scikit-learn to split and copy the dataset $N$ times, which, in addition to parallelization, causes a memory spike that the 16GB Kaggle Kernel cannot handle, even using $N = 3$, for a feature set of reasonable size.

OOB error estimates are described in Chapter 15.3.1 of [HTF09], which mentions that OOB errors are nearly identical to those obtained by $N$-fold cross-validation. As well, section 3.1 of [Bre01] explains that, unlike in cross-validation, out-of-bag estimates are unbiased estimators once $n_{\text{estimators}}$ is large enough that the test error converges. Otherwise, OOB *over*estimates error. This does not directly speak to how the difference between OOB error and test error varies with different choices of $\mathbf{n}$, which is a key question for understanding the results of hyperparameter tuning, especially in the low $B$ ($\approx 40$) domain. These errors are empirically close for the purposes of this project, so overall the OOB approach seems justified, though perhaps we should not trust fine differences in scores over different hyperparameters too much.

As before, the implementation details of `RandomForestClassifier` conveniently differ from the above in that $\hat{\mathsf{T}}(x; \Theta_b)$ is a probability prediction rather than a class prediction (and scikit-learn averages probabilistic predictions instead of using majority-vote). Given the approximation between OOB estimates and test error, this feature allows us to use OOB estimates for rank metrics in addition to threshold metrics, as described in the next section.

A Stack Overflow post was insightful in devising the strategy to calculate OOB estimates using different metrics mentioned in Table 10. To use OOB estimates with the `sklearn` interface, we construct the hyperparameter search somewhat manually with `sklearn.model_selection.ParameterGrid`, as shown in Listing 2 which essentially creates a Python iterator from a dictionary of parameters, `param_grid`. Once we have trained a random forest classifier on a combination of parameters in `param_grid`, we expose the probabilistic OOB estimates with the `RandomForestClassifier` attribute `oob_decision_function_`. Then we can score all OOB estimates using the metrics described in Table 10.

### 5.3 Metrics

Sections 9.2.3 and 9.2.4 contain the code described in this section.

Not all of these metrics are necessarily relevant. In particular, Log Loss and Cohen's Kappa are included 1) out of curiosity and 2) because seven is prime; we should not pay much attention to these metrics except as an exploration into the behavior of the metrics themselves.

By threshold metrics, we mean those which score class values, e.g. precision, recall, and geometric mean (g-mean), while by rank metrics (threshold == False), we mean those which score probability orderings, e.g. AUC-ROC and AUC-PR.

One issue to be aware of when considering metrics is that the ratio between negative and positive classes is

$$\text{skew} = \frac{\text{Negative Classes}}{\text{Positive Classes}} \approx 10. \tag{5.11}$$

Perhaps this is not exactly an imbalance, but certain metrics are more robust to skew than others.

The discussion of [JCDLT13] offers guidance on metric behavior with respect to skew and classifier performance. The simulations shown in Figure 1 of [JCDLT13] suggest that, as far as

| name | function | threshold |
|---|---|---|
| AUC-ROC | `sklearn.metrics.roc_auc_score` | False |
| AUC-PR | `sklearn.metrics.average_precision_score` | False |
| Log Loss | `sklearn.metrics.log_loss` | False |
| Precision | `sklearn.metrics.precision_score` | True |
| Recall | `sklearn.metrics.recall_score` | True |
| F1 Score | `sklearn.metrics.f1_score` | True |
| Balanced Accuracy | `sklearn.metrics.balanced_accuracy_score` | True |
| Geometric Mean | `imblearn.metrics.geometric_mean_score` | True |
| Cohen's Kappa | `sklearn.metrics.cohen_kappa_score` | True |

Table 10: List of metrics

rank metrics, it could make sense to favor AUC-PR to AUC-ROC. Given the skew and the high misclassification rate in this problem relative to those the authors considered, perhaps AUC-PR scores have a greater spread than AUC-ROC scores with respect to misclassification rate.

To compute threshold metrics, we choose the probability threshold which predicts a skew for $\hat{y}$ of 10. Threshold metrics are not relevant in judging hyperparameters, though it could be informative to view them. The Kaggle competition uses F1 Score, which is the harmonic mean of precision and recall. Balanced accuracy is the arithmetic mean of these while g-mean is the geometric mean of sensitivity and specificity.

We use the same set of metrics to score the tuned classifier on $y_{\text{test}}$. Note that scores for the model describe test error which is not exactly comparable to the OOB error we compute in hyperparameter tuning.

### 5.4 Hyperparameter Tuning

Section 9.2.3 references the code described in this section.

The hyperparameters we modify from the default values of `RandomForestClassifier` are various pairs of the following:

- `max_features` $\in \{5, \ldots, 9\}$

  Within the above range, there was little variation in scores.

- `max_depth` $\in \{4, \ldots, 64\}$

  This parameter actually seemed generally harmful to performance when varied (from `None`).

- `criterion` $\in \{$`gini`, `entropy`$\}$

  The default is `gini`. Splitting criterions made no observable difference in model performance. Explanations as to why are contained in Figure 9.3 of [HTF09] as well as the following StackOverflow posts:

  - What is "entropy and information gain"?
  - When should I use Gini Impurity as opposed to Information Gain?

- `min_samples_leaf` $\in \{4, \ldots, 48\}$

**Listing 1:** The parameter grid responsible for the plot of optimal scores in Figure 5.

```
param1 = 'min_impurity_decrease'
param2 = 'min_samples_leaf'

param_grid = {
    param1: np.geomspace(1e-8, 1e-6, 5),
    param2: np.around(np.geomspace(12, 48, 3)).astype(int)
}
```

"The minimum number of samples required to be at a leaf node."

This is one of the two values adjusted from the defaults to train the model.

- `min_impurity_decrease` $\in [10^{-8}, 10^{-5}]$

  "A node will be split if this split induces a decrease of the impurity greater than or equal to this value."

  This is the other value adjusted from the defaults to train the model. In the context of decision trees, [HTF09] mentions that modifying this hyperparameter (from `None`) is near-sighted, since there there may be higher information gain splits to be found beyond this threshold. Nonetheless, tuning this hyperparameter was more effective than the above and it has the benefit of making tree size easier to control than any of the previous hyperparameters, which is helpful to control resource usage. We have not attempted to vary `max_leaf_nodes`, which could achieve these goals more effectively.

In the notation of Definition 2 we may consider a collection of random forest classifiers

$$\left\{\widehat{\mathsf{RFC}}_{\mathbf{n}}(x)\right\}_{\mathbf{n}\in\mathbf{G}} \tag{5.12}$$

where $\mathbf{G} \subset \mathbf{H}$ is a *parameter grid* instantiated by, say, the `param_grid` of Listing 1. Then Figure 5 displays the OOB scores of the $\widehat{\mathsf{OOB}}_{\mathbf{n}}$ classifier associated to $\widehat{\mathsf{RFC}}_{\mathbf{n}}$

$$\hat{y}^\star = \widehat{\mathsf{OOB}}_{\mathbf{n}}(x) \text{ for all } \mathbf{n} \in \mathbf{G}. \tag{5.13}$$

**Listing 2:** Hyperparameter search loop; `decisions` == `y_predict_proba`

```
rfc = RandomForestClassifier(n_estimators=35,
                             n_jobs=-1,
                             oob_score=True)

decisions = []

for params in ParameterGrid(param_grid):
    rfc.set_params(**params)
    rfc.fit(X['train'], y['train'].values.ravel())
    decisions.append(rfc.oob_decision_function_[:, 1])
```

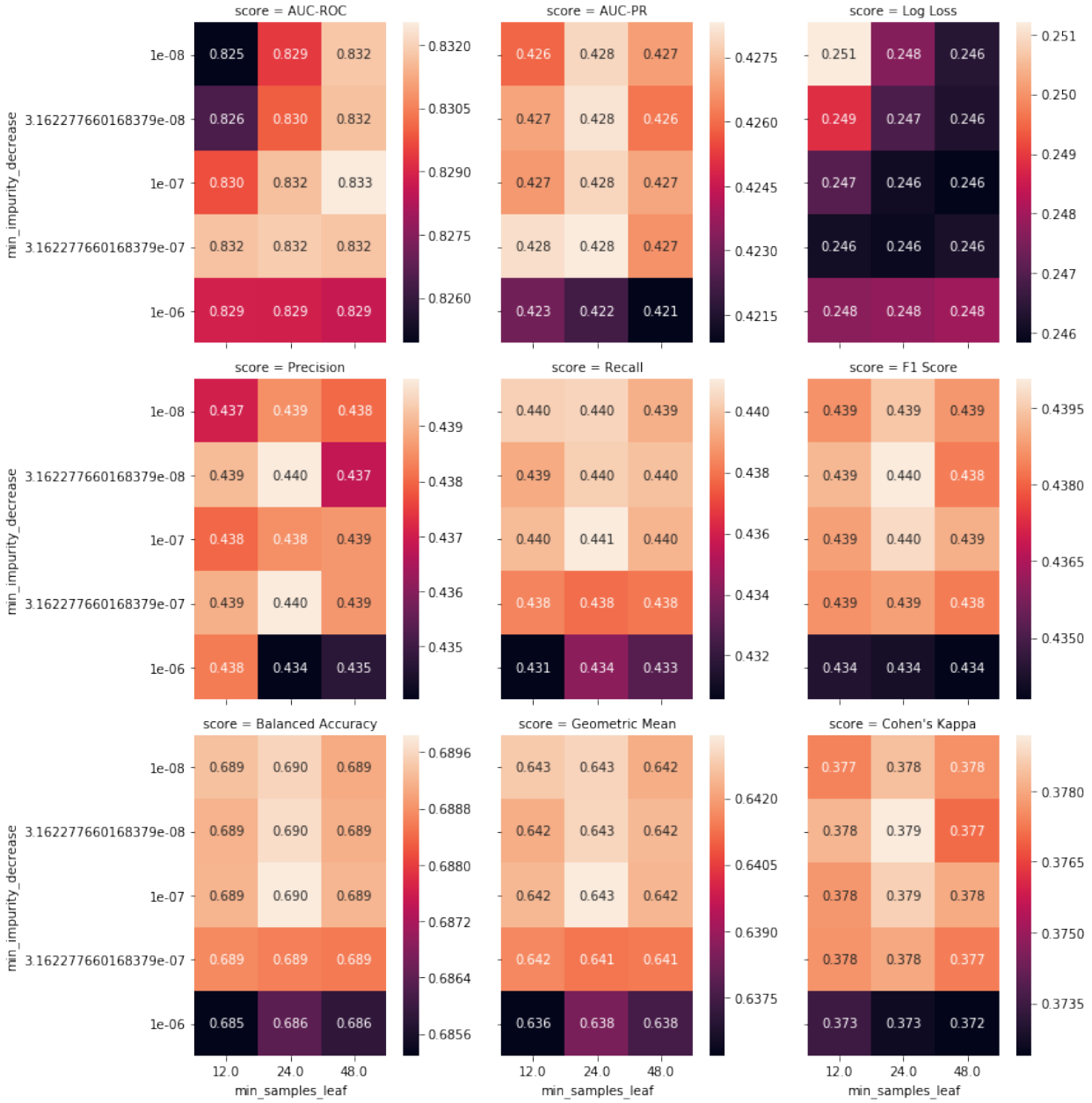



Figure 5: A `ParameterGrid` search as in Listing 2 in the neighborhood of the optimal scores defined by Listing 1 scored using the metrics of Table 10.

As mentioned in Section 5.1, the implementation details of `RandomForestClassifier` mean that $\hat{y}^\star$ is a probabilistic prediction rather than a boolean value; we develop variants to convert $\hat{y}^\star \mapsto \hat{y}$ in section 6.

Fortunately, the optimal scores across metrics seem near enough, at least at this scale. The AUC-PR arg max occurs at a value $\mathbf{n_0}$ of

- `min_impurity_decrease` $= 10^{-6.5}$,
- `min_samples_leaf` $= 24$

to the resolution displayed in Figure 5. Note that lower scores are better for Log Loss.

Those trees grown with the default values for `RandomForestClassifier` are "fully-grown" trees, in that those values for $\mathbf{n}$ do little or nothing to 'interfere' with the tree topology, aside from `max_features = 'sqrt'`, which is the default value for classification suggested by the creators. Such fully-grown trees have $\Lambda \approx 10^6$ on $X_{\text{train}} \times y_{\text{train}}$, whereas in the neighborhood defined by Listing 1, $\Lambda \approx 10^{4.5}$—$10^5$.

The search performed in Figure 5 used $B =$ `n_estimators` $= 35$ trees, which is admittedly few, though samples roughly every observation in `'train'`. We used this value for `n_estimators` to obtain a runtime less than the six hours provided by Kaggle Kernels. A search on a smaller parameter grid with roughly double the `n_estimators` indicates a bit of empirical stability in optimal hyperparameters as we increase the number of trees.

### 5.5 Rank Scores

Section 9.2.4 references the code described in this section.

**Listing 3:** Call and fit `RandomForestClassifier`

```
rfc = RandomForestClassifier(n_estimators=600,
                             max_features='sqrt',
                             min_impurity_decrease=3e-7,
                             min_samples_leaf=24,
                             n_jobs=-1)

rfc.fit(X['train'], y['train'].values.ravel())

y_predict_proba = dict.fromkeys(dsets)

for ds in dsets:
    y_predict_proba[ds] = rfc.predict_proba(X[ds])[:, 1]
```

Now that we have decided on hyperparameters, we fit the classifier as in Listing 3. The decision to increase `n_estimators` to 600 decreases error at the cost of compute time. More trees decreases variance without increasing bias; the only reason we do not use more trees in Listing 2 is the six hour compute time limit of Kaggle Kernels. Listing 3 has a runtime of approximately four hours.

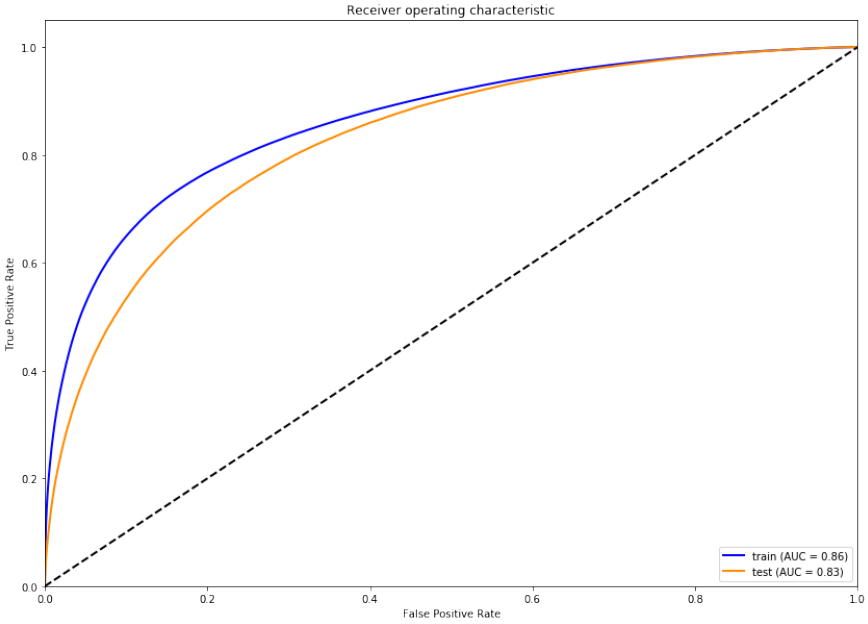



Figure 6: Receiver Operating Characteristic (ROC) curve; AUC-ROC = 0.83.

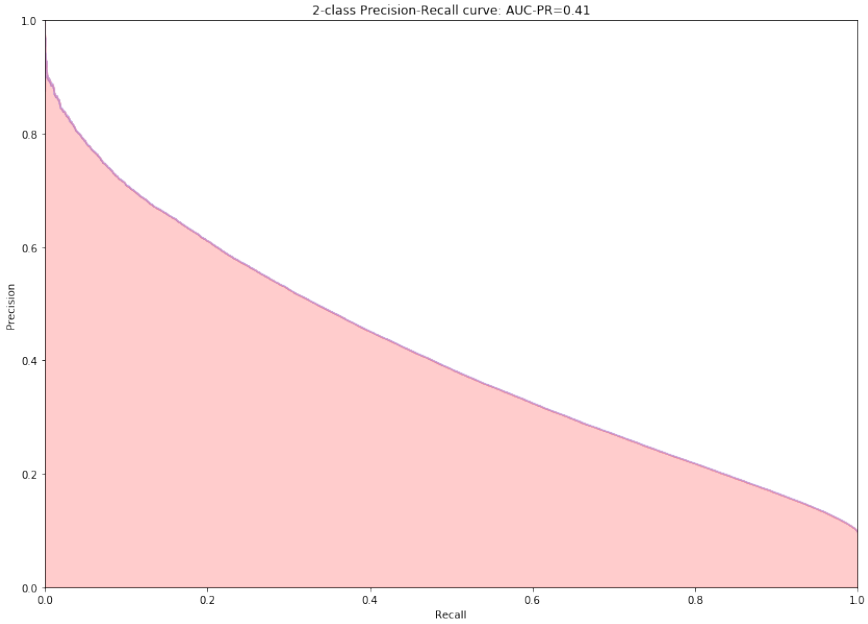



Figure 7: Precision-Recall Curve; AUC-PR = 0.41.

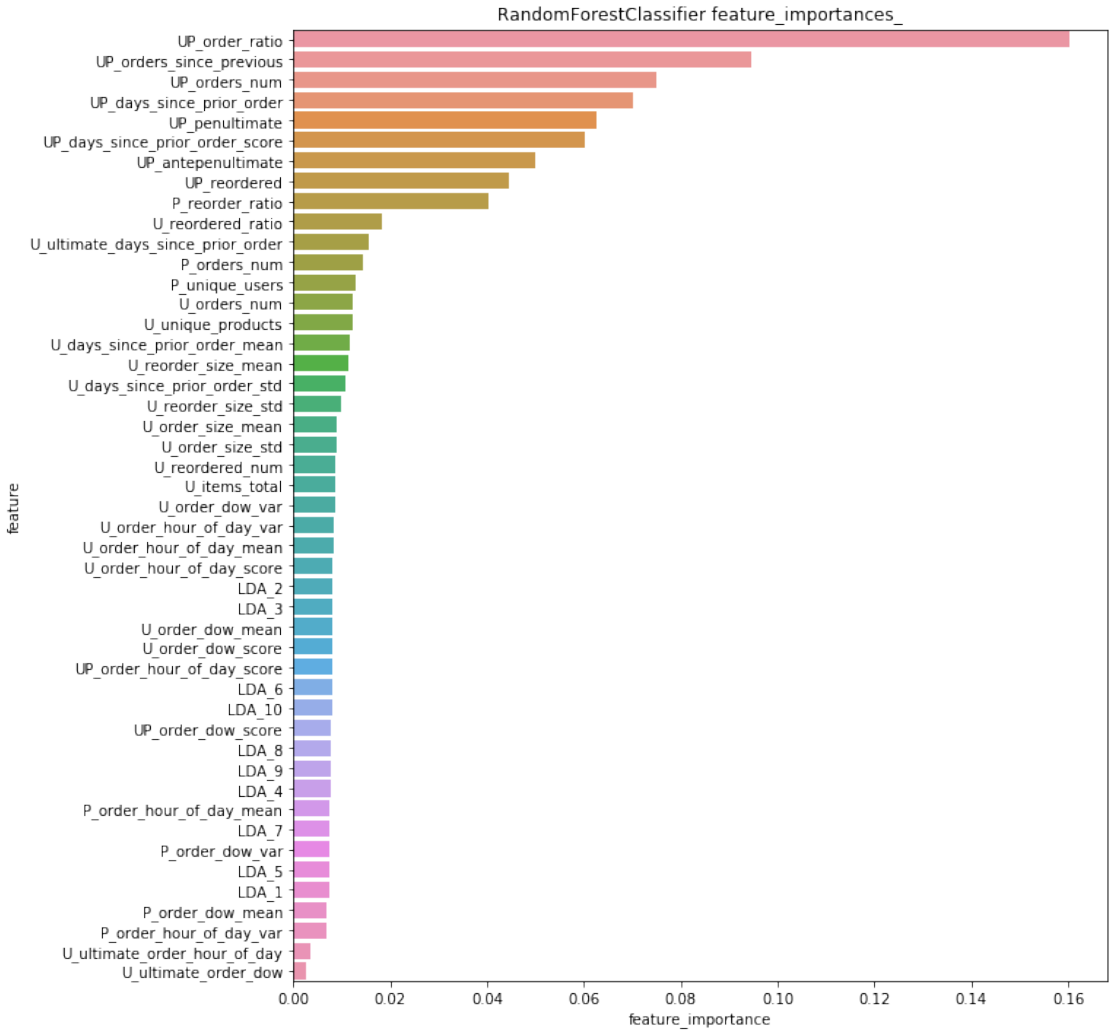



Figure 8: The `RandomForestClassifier` uses an information criterion to determine variable importances.

### 5.5.1 ROC Curve

An issue with changing `n_estimators` to a different value after hyperparameter tuning is that there is not necessarily reason to think that the optimal hyperparameters for `n_estimators` $= 35$ are the same as the optimal hyperparameters for `n_estimators` $= 600$. On the other hand, changing the value of `n_estimators` does not alter $\mathcal{T}(\mathbf{n})$, the decision tree distribution. Also, there is a small amount of empirical evidence that the optimal hyperparameters are relatively stable in the range `n_estimators` $\approx 25$—$75$. Finally, given that notebooks of both Sections 9.2.3 and 9.2.4 must run in six hours, this seems like the best strategy since the hyperparameters found in Section 5.4 are the best we are able to compute.

### 5.5.2 Precision-Recall Curve

Now we can examine the probabilistic predictions from the `RandomForestClassifier` of Listing 3. The ROC Curve shown in Figure 6 displays decent classifier performance. On the other hand, a

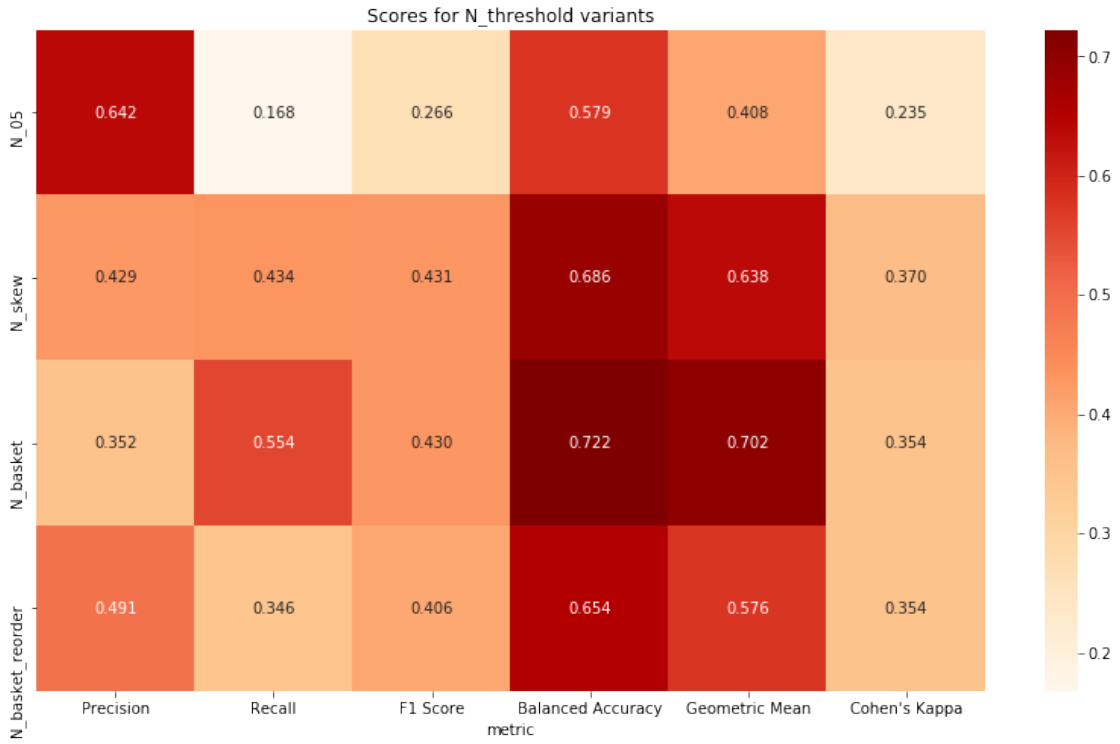



Figure 9: Scores for $\mathsf{TopN}_{\mathsf{threshold}}$ variants.

skew of 10 means that it is a bit 'easier' to find negatives and 'suppress' the false positive rate. The performance displayed in Figure 6 seems overly optimistic when compared to the Precision-Recall Curve of Figure 7. [Bro18] contains an informative review and argument for using AUC-PR rather than AUC-ROC given an imbalance.

### 5.5.3 Variable Importances

Figure 8 describes the importances of the features listed in Table 12. Variable importance is another nice feature of random forest classifiers. The importances are computed via the information criterion; the idea is described in 15.3.2 of [HTF09]. One of the most fruitful avenues to pursue to improve classifier performance is to focus on manually creating additional user-product features which capture more complex user-product interactions.

## 6 TopN Variants

Section 9.2.4 references the code described in this section.

Finally, we define various maps

$$\begin{aligned} \mathsf{TopN} : [0,1]^n &\to \{0,1\}^n \\ \hat{y}^\star &\mapsto \hat{y} \end{aligned} \tag{6.1}$$

to make binary predictions. The structure of the definitions is to define a subset of user-product

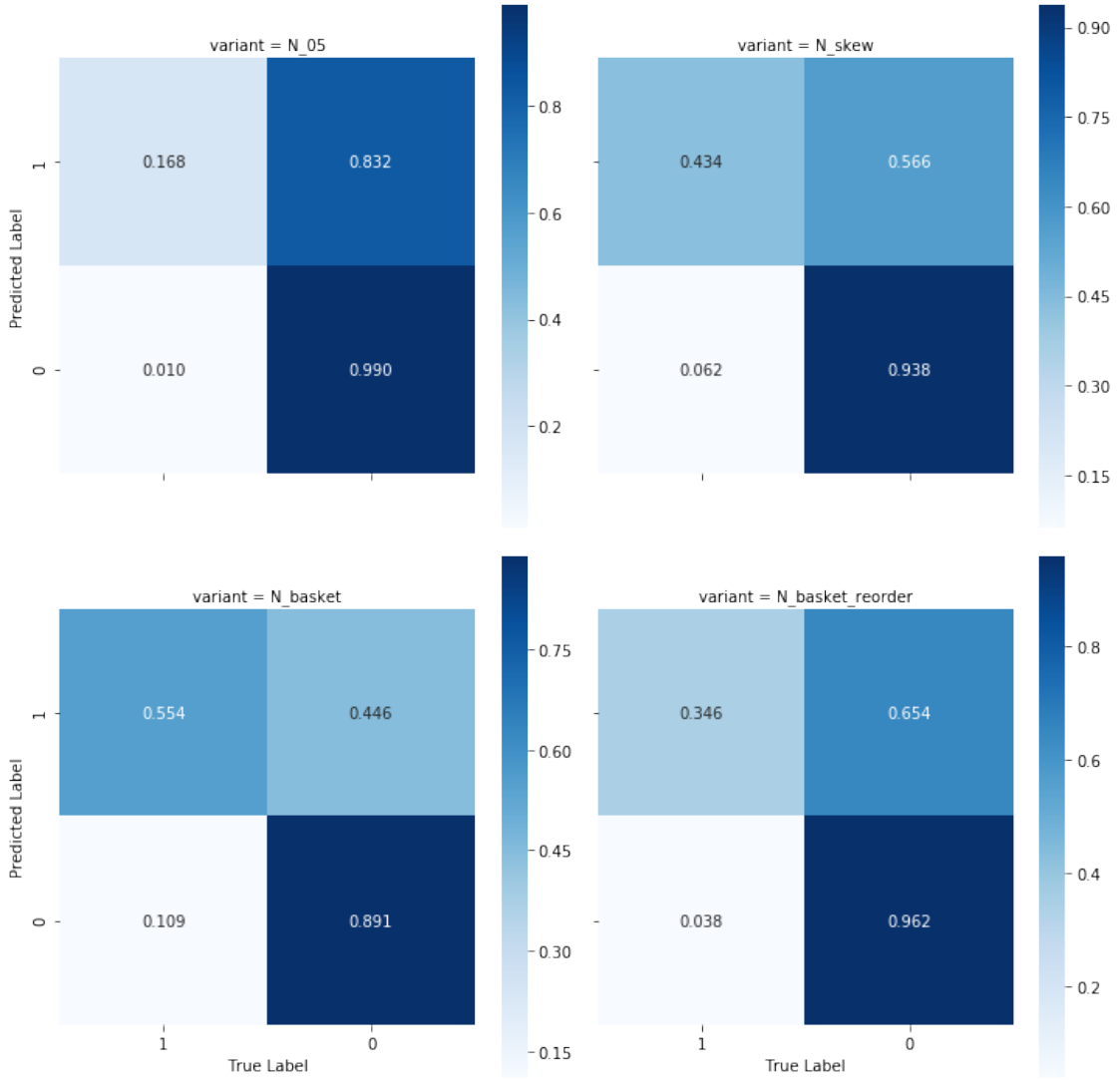



Figure 10: Normalized confusion matrices for $\mathsf{TopN}_{\text{threshold}}$ variants.

pairs $N_{\text{variant}} \subset \Gamma$ and throughout let

$$\mathsf{TopN}_{\text{variant}}((\hat{y}^\star)^\gamma) = \begin{cases} 1 & \text{if } \gamma \in N_{\text{variant}}, \\ 0 & \text{if } \gamma \notin N_{\text{variant}}. \end{cases} \tag{6.2}$$

## 6.1 $\mathsf{TopN}_{\text{threshold}}$ Variants

If we choose the $N_{\text{threshold}}$ user-product pairs $(u, p) \in \Gamma$ with the greatest $(\hat{y}^\star)^{(u,p)}$, then this top-$N$ variant is a reparamaterization of the classification threshold, $p_0$, via

$$N_{\text{threshold}} = \left\{ (u, p) \mid (\hat{y}^\star)^{(u,p)} > p_0 \right\}.$$

An advantage of this variant is that it recommends the items users are, in aggregate, most likely to purchase. Therefore, we can, overall, make the 'best' recommendations. A disadvantage is that

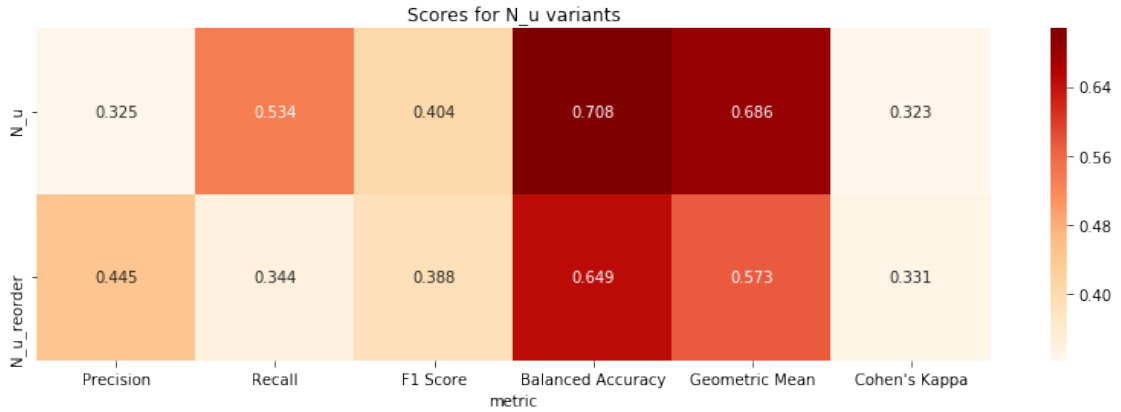



Figure 11: Scores for $\mathsf{TopN}_\mathsf{u}$ variants.

there is a lot variation in the number of recommended products. Some proposals for principled choices of $N_{\text{threshold}}$ follow.

$N_{0.5}$

A direct interpretation of $\hat{y}$ in terms of probabilities would make a threshold of $p_0 = 0.5$ the most principled choice. On the other hand, choices made in the model definition and implementation make this interpretation dubious. Nonetheless, we can define

$$N_{0.5} = \{(u, p) \mid P((u, p)) > 0.5\}$$

to explore scores. Figures 9 and 10 show scores and normalized confusion matrices.

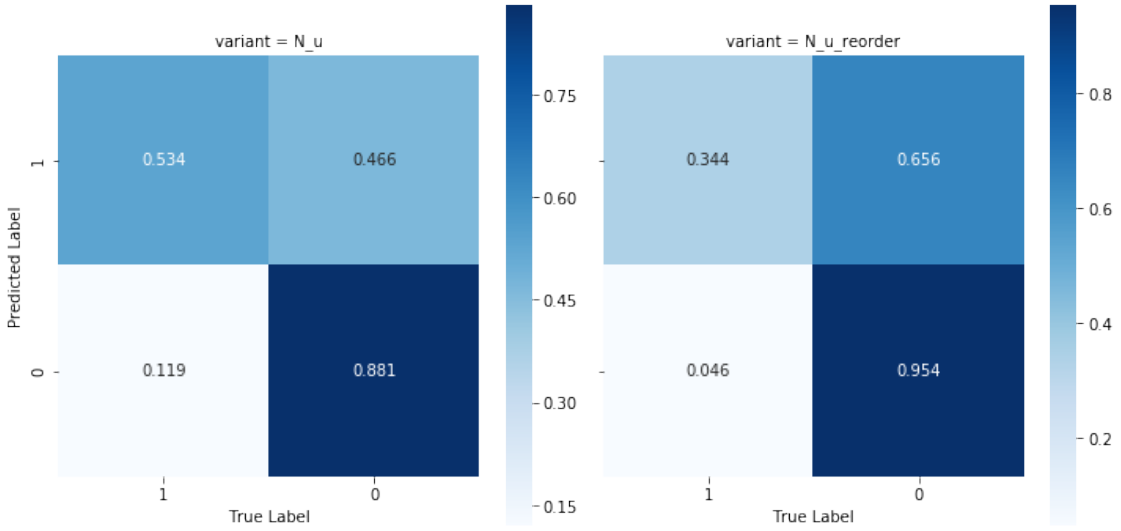



Figure 12: Normalized confusion matrices for $\mathsf{TopN}_\mathsf{u}$ variants.

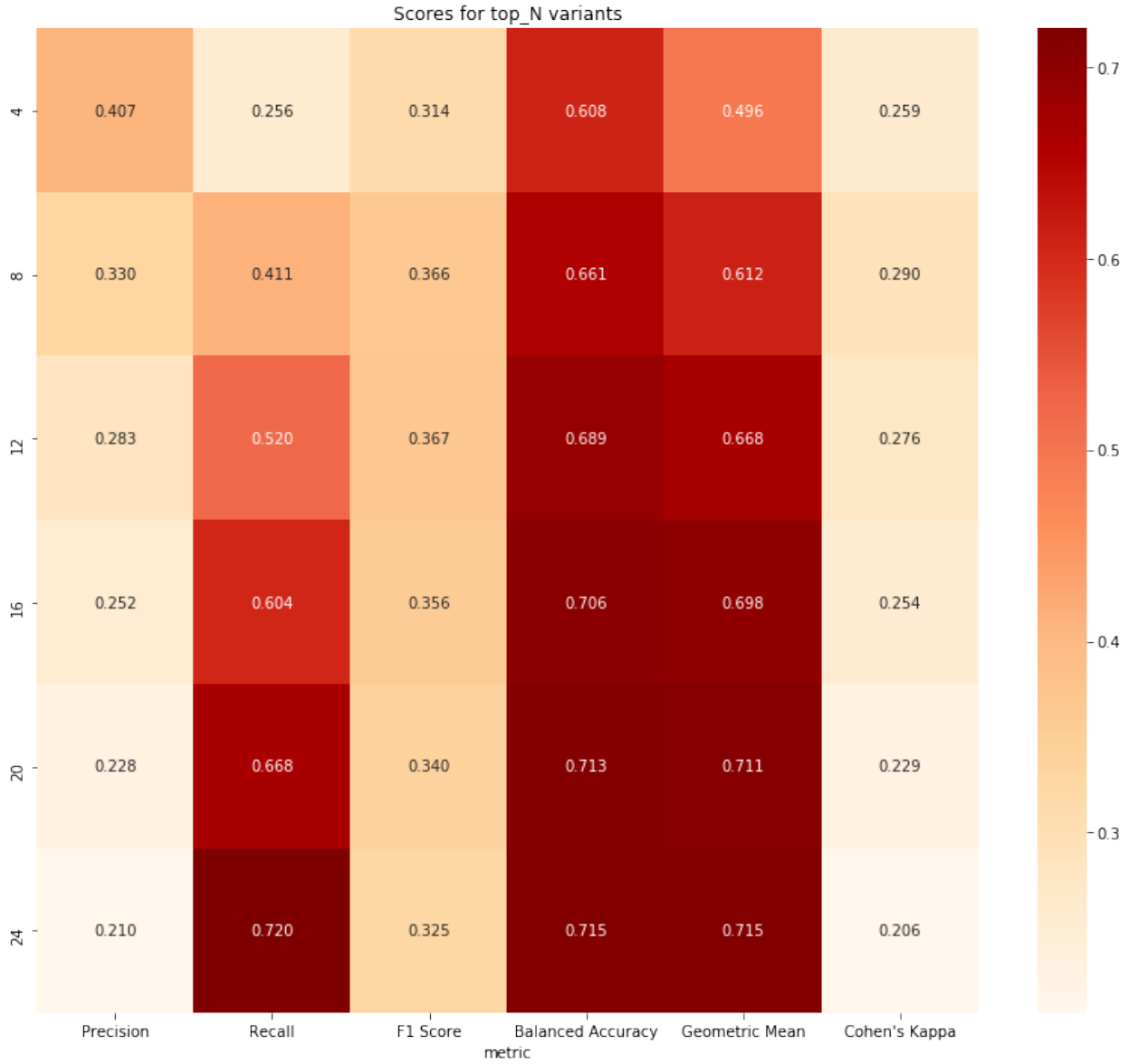



Figure 13: Scores for $\text{TopN}_\text{N}$ variants.

**$N_{\text{skew}}$**

It may instead be most principled to choose $N$ such that the skew of $\hat{y}$ equals the train skew.

**$N_{\text{basket}}$**

A couple other values we may consider to be principled are

$$N_{\text{basket}} = \sum_{u \in U} \overline{b(u)}$$

$$N_{\text{basket reorder}} = \sum_{u \in U} \overline{r(u)}$$

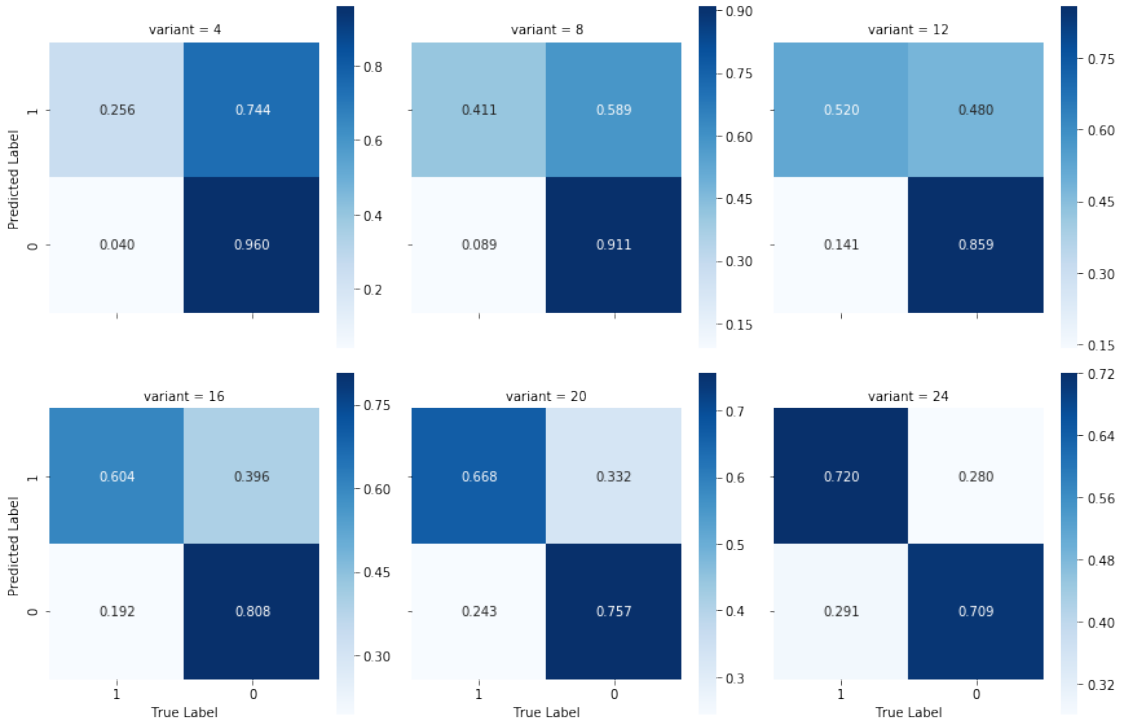



Figure 14: Normalized confusion matrices for $\mathsf{TopN}_\mathsf{N}$.

where $\overline{b(u)}$ is the mean basket size for user $u$ and $\overline{r(u)}$ is the mean reordered items per basket for user $u$.

## 6.2 $\mathsf{TopN}_\mathsf{u}$ Variants

If we want to avoid the high variation of basket sizes in the $\mathsf{TopN}_{\mathsf{threshold}}$ variant, we can define ultimate basket sizes by user. Let

$$N(u) = \left\lceil \overline{b(u)} \right\rceil$$

$$N_{\text{reorder}}(u) = \left\lceil \overline{r(u)} \right\rceil$$

for all $u \in U$, that is, the ceiling of the mean (re)orders per basket for user $u$. Using the ceiling rather than round or floor offers the advantage of recommending at least one item to users in a straight-forward way. The mean reorders per basket is biased even as a predictor of ultimate *re*orders since user's initial orders have zero reorders. On the other hand, since we are predicting fewer positives, the precision of $N_{\text{reorder}}(u)$ is better.

For the curious reader, we can be formal about this by defining a maximum-$N$ function recursively as

$$M_1(A) = \{\max(A)\}, \qquad M_{N+1}(A) = \{\max(A \setminus M_N(A))\} \cup M_N(A) \tag{6.3}$$

where $A$ and $M_N(A)$ are ordered sets. Then

$$N_u = \bigcup_{u \in U} M_{N(u)}(\Gamma|_u) \tag{6.4}$$

defines a function

$$\mathsf{TopN}_u : [0,1]^n \times \mathcal{D} \to \{0,1\}^n \tag{6.5}$$

which depends on the raw data since $N(u)$ depends on the raw data.

These kinds of variants, or those which offer control of ultimate basket size while offering more flexibility in that control, may be useful for product applications like auto-populating user carts with items we most expect users to reorder. For such an application we would prefer models with higher precision. Figures 11 and 12 show scores and normalized confusion matrices.

### 6.3 $\mathsf{TopN}_N$ Variants

While the top-$N_u$ model gives better predictions, a top-$N$ model with a fixed $N$ for all users may be useful, for example, in displaying previously purchased product recommendations on a web page of fixed size. For an application like this one would prefer higher recall models. Figures 13 and 14 show scores and normalized confusion matrices for $N \in \{4, 8, 12, 16, 20, 24\}$. Using (6.3) it is straight-forward to write explicit formulae for the $\mathsf{TopN}_N$ maps.

## 7 Prediction Explorer

Section 9.2.5 references the code described in this section.

For a user and a model variant, we construct a utility to visualize the model prediction, true order, and basket history, along with colorings for

- True Positives: Ordered and Predicted
- False Positives: Predicted but not Ordered
- False Negatives: Ordered but not Predicted
- True Negatives: neither Ordered nor Predicted

A model with greater **precision** has fewer False Positives while a model with greater **recall** has fewer False Negatives. Note: `'add_to_cart_order'` is not meaningful for the top rows of predictions and true orders.

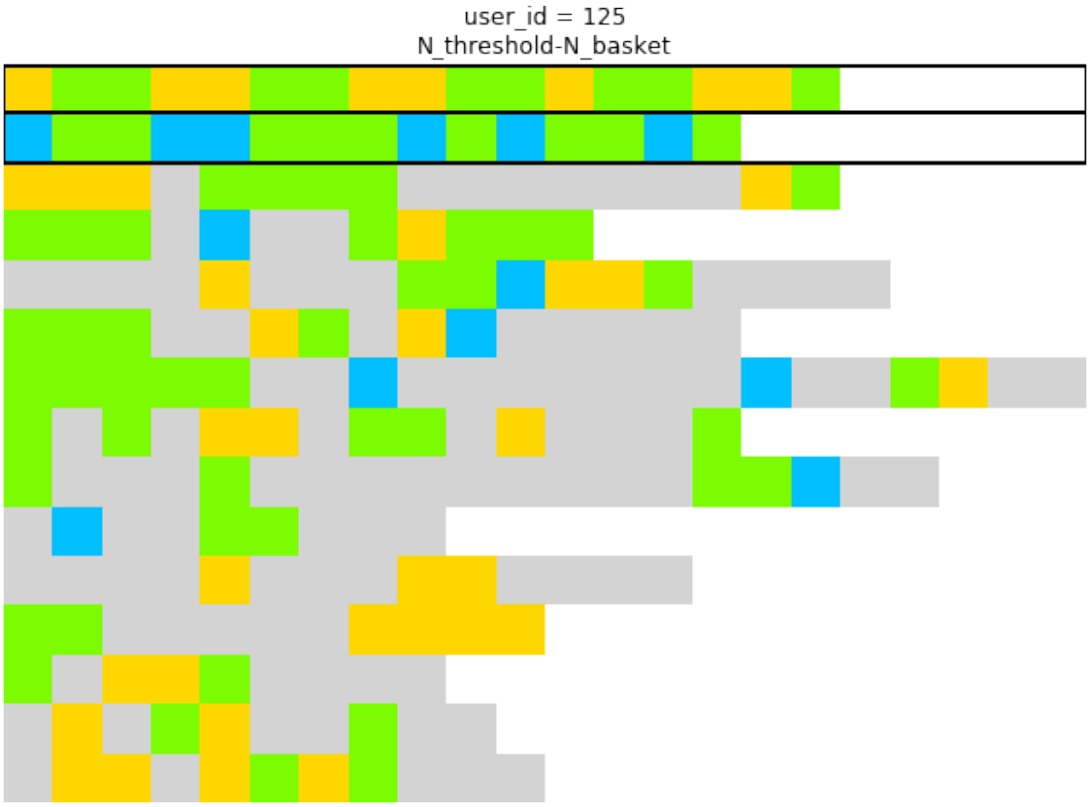
**Listing 4:** Model choices for Figure 17

```
models_list = [
    ('N_threshold', 'N_basket'),
    ('N_u', 'N_u'),
    ('top_N', '8'),
    ('top_N', '20')
]
```

| add_to_cart_order | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| order_number | | | | | | | | |
| prediction | Hampshire 100% Natural Sour Cream | Cut & Peeled Baby Carrots | Bistro Bowl Chicken Caesar Salad | Salisbury Steak with macaroni and cheese Salisbury Steak with macaroni and cheese | Multi-Grain Club Crackers | Sliced Sourdough Bread | Organic Fat-Free Milk | Blueberry on the Bottom Nonfat Greek Yogurt |
| true | Chicken Thighs | Cut & Peeled Baby Carrots | Bistro Bowl Chicken Caesar Salad | Deluxe Plain Bagels | Ritz Crackers | Sliced Sourdough Bread | Organic Fat-Free Milk | French Onion Dip |
| 13 | Strawberry on the Bottom Nonfat Greek Yogurt | Blueberry on the Bottom Nonfat Greek Yogurt | Non Fat Black Cherry on the Bottom Greek Yogurt | Colagate Total Whitening Toothpaste | Cut & Peeled Baby Carrots | Sliced Sourdough Bread | Poppycock Cashew Lovers | Butter Toffee Peanuts |
| 12 | Cut & Peeled Baby Carrots | French Onion Dip | Sliced Sourdough Bread | Original Cream Cheese | Deluxe Plain Bagels | Ultra Plush® 3 Ply Double Toilet Paper Rolls | Select-a-Size Rolls Paper Towels Tissue | Coke |
| 11 | Roma Tomato | French Bread | Organic Navel Orange | Whole Grains Oatnut Bread | Hampshire 100% Natural Sour Cream | Broccoli Crown | Cooking Beef Stock | Horseradish |
| 10 | Spiced Rum | French Onion Dip | Bistro Bowl Chicken Caesar Salad | Complete Clean Power Toilet Bowl Cleaner Value Pack | Assorted Chocolate Miniatures Chocolate Candy Bars | Multi-Grain Club Crackers | Cut & Peeled Baby Carrots | Classic Mix Variety |
| 9 | Coke | Spiced Rum | Sliced Sourdough Bread | Cut & Peeled Baby Carrots | French Onion Dip | Rigatoni Pasta | Crackers | Chicken Thighs |
| 8 | Spiced Rum | Butter | Coke | Original Citrus Sparkling Flavored Soda | Meatloaf and Mashed Potatoes | Salisbury Steak with macaroni and cheese Salisbury Steak with macaroni and cheese | 360 Dusters Refills Unscented | French Onion Dip |

Figure 15: A portion of a styled DataFrame of Predictions, Ultimate Orders, and Basket History for `user_id == 125`; `model == ('N_threshold', 'N_basket')`.

Figure 16: "Zoom out" on Figure 15 by ignoring product names.



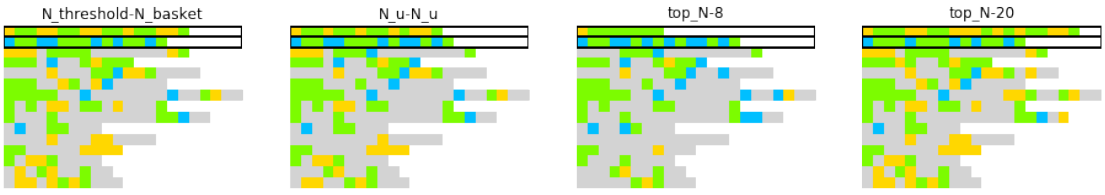




Figure 17: "Zoom out" further by displaying predictions for `user_id == 125` using the model choices of Listing 4. By fixing a user, we can inspect the behavior of different models.

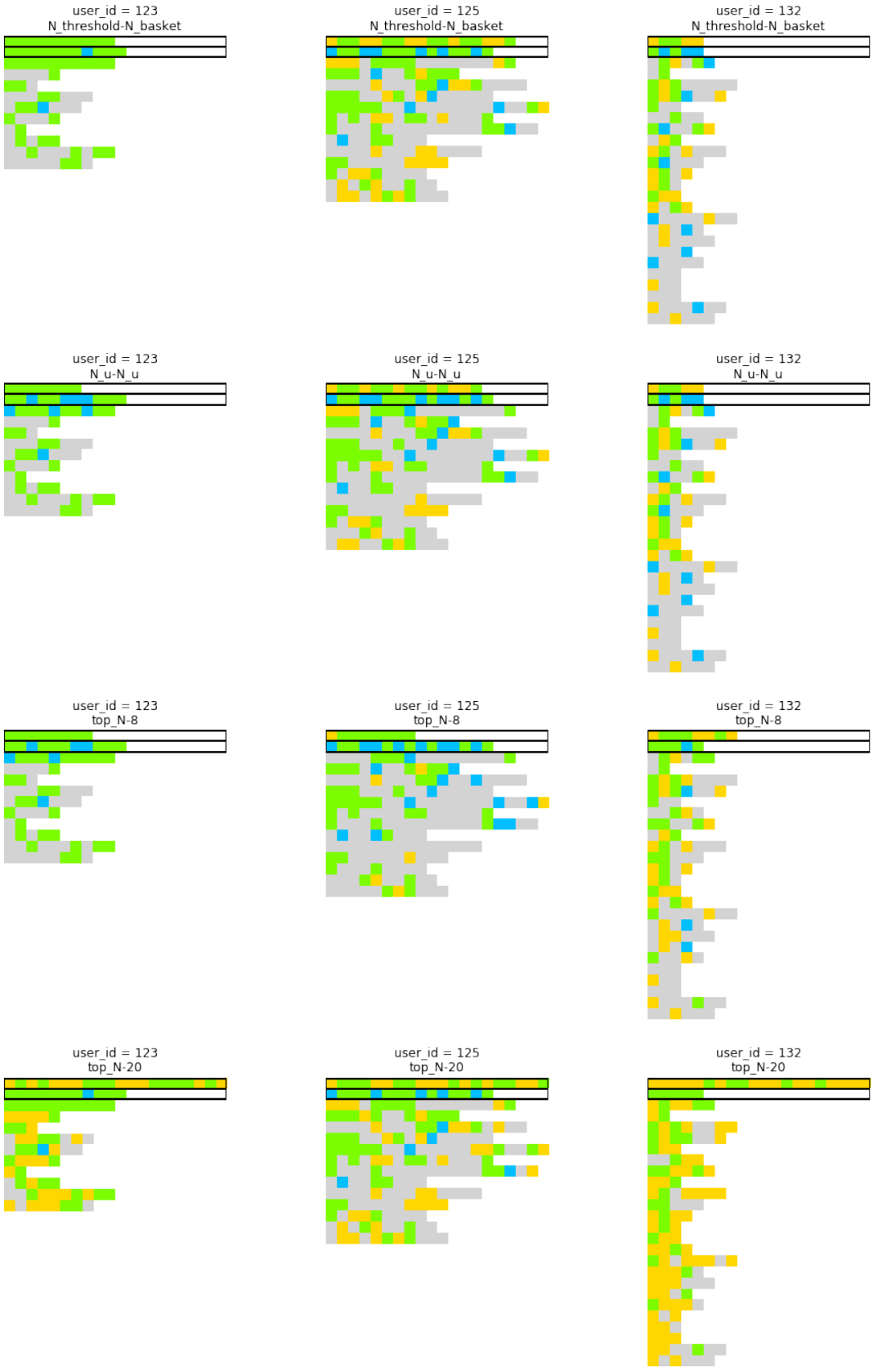



Figure 18: Zoom out on Figure 17 to include more users.

# 8 Prospects

There are a number of avenues to pursue in future versions of this project:

**Feature Design:** The first suggestions listed may be the most urgent to improve performance.

- Design more User-Product features as a first step. Figure 8 demonstrates how much more effective these features are than those which come from Profiles with lower information content.
- User-Aisle and User-Department Profiles could be the next reasonable Profiles to build if the User-Product profile is exhausted though such features would be correlated with analogous User-Product features.
- As well, there are additional sorts of complex features which require a modest increase in memory. For example, non-negative matrix factorization (NMF) techniques have traditionally been used in recommendation systems. Such techniques may be appropriate for the user-user matrix of, say, counts of common product purchases. One application of NMF to this matrix is dimensionality reduction – to create a relatively small number of user "topics" based on the common purchase count matrix. Experimentation along the lines of Listing 5 on subsets of the overall dataset suggest the matrix is tractable at a size of perhaps roughly 10GB for $s$ = train. The value of `n_components` requires a search (scored by `factor.reconstruction_err_`).

**Listing 5:** NMF Mock-up

```
from sklearn.decomposition import NMF
from itertools import combinations

def col_comb(gp, r):
    return pd.DataFrame(list(combinations(gp.values, r)),
                        columns=['row', 'col'])

product_user = dict.fromkeys(dsets)
product_similarity = dict.fromkeys(dsets)

for ds in dsets:
    product_user[ds] = (prior[ds][['user_id', 'product_id']]
                        .drop_duplicates()
                        .sort_values(by=['user_id', 'product_id']))

product_similarity = (product_user[ds]
                      .groupby('user_id').product_id
                      .apply(col_comb, 2)
                      .reset_index(level=1, drop=True).reset_index()
                      .groupby(['row','col']).count())

factor = NMF(n_components=10)
W = factor.fit_transform(product_similarity)
H = factor.components_
```

- There are many approaches to incorporate interactions between baskets. The factorizing personalized Markov chains model in [RFS10] is a tensor factorization approach which improves on matrix factorization alone by introducing a "personalized Markov chain" product-product transition matrix for each user. These papers are mentioned in [QCJ18], which provides an excellent overview of these more complex techniques.

**Random Forest Classifier:** Alterations related to the classifier itself may improve performance somewhat.

- Adjust train/test split ratio to 70%/30%; check if this lowers the difference between train and test AUC-ROC.
- Experiment with the `max_leaf_nodes` hyperparameter.
- Instead of random forests, experiment with gradient boosting or XGBoost.
- A couple variants of random forests were introduced in [CAL04] to better handle imbalanced data. These may offer marginal improvements.
  - Balanced Random Forests are an under/over-sampling technique to either oversample the minority class or undersample the majority class; the imbalanced-learn package has an implementation imblearn.ensemble.BalancedRandomForestClassifier though the results of early experiments have been difficult to interpret.
  - Weighted Random Forests penalize the majority class more or the minority class less; despite the name, they appear to be implemented in scikit-learn as `class_weight='balanced'`. Early experimentation suggests there may be marginal improvements we can gain with this.

**Prediction Explorer:** Some ideas to improve the Prediction Explorer utility are

- a product information highlight display on mouse hover (using Bokeh),
- an API allowing for interactivity in user choice, model variant, etc., and
- a simple web app front-end.

# 9 Appendices

## 9.1 Appendix I: Features List

Table 11: Profiles

| prefix | name |
|---|---|
| U | User Profile |
| P | Product Profile |
| UP | User-Product Profile |
| AD | Aisle and Department Profiles (ignored) |
| LDA | Latent Dirichlet Allocation User Features |

Table 12: List of features

| feature | dtype | description |
|---|---|---|
| U_ultimate_order_dow | float16 | dow of user's ultimate order |
| U_ultimate_order_hour_of_day | float16 | hour of user's ultimate order |
| U_ultimate_days_since_prior_order | float16 | days since user's previous order (from ultimate) |
| U_orders_num | uint8 | number of orders a given user has placed |
| U_items_total | uint16 | number of total items a given user has purchased |
| U_order_size_mean | float16 | mean basket size for a given user |
| U_order_size_std | float16 | std basket size for a given user |
| U_unique_products | uint16 | number of unique products a given user has purchased |
| U_reordered_num | uint16 | number of total items a given user has purchased which are reorders |
| U_reorder_size_mean | float16 | mean reorders per basket |
| U_reorder_size_std | float16 | std reorders per basket |
| U_reordered_ratio | float16 | proportion of items a given user has purchased which are reorders |
| U_order_dow_mean | float16 | mean order_dow |
| U_order_dow_var | float16 | var order_dow |
| U_order_dow_score | float16 | ultimate score for order_dow using circstd = sqrt(-2ln(circvar)) |
| U_order_hour_of_day_mean | float16 | mean order_hour_of_day |
| U_order_hour_of_day_var | float16 | var order_hour_of_day |
| U_order_hour_of_day_score | float16 | ultimate score for order_hour_of_day using circstd = sqrt(-2ln(circvar)) |
| U_days_since_prior_order_mean | float16 | mean days since prior order (mean user order time interval) |

| | | |
|---|---|---|
| U_days_since_prior_order_std | float16 | std days since prior order (std user order time interval) |
| P_orders_num | uint32 | number of total purchases |
| P_unique_users | uint16 | number of purchasers |
| P_reorder_ratio | float16 | reorder ratio |
| P_order_hour_of_day_mean | float16 | mean order_hour_of_day |
| P_order_hour_of_day_var | float16 | var order_hour_of_day |
| P_order_dow_mean | float16 | mean order_dow |
| P_order_dow_var | float16 | var order_dow |
| UP_orders_num | uint8 | number of times particular user has ordered particular product |
| UP_orders_since_previous | uint8 | number of orders since previous purchase of product by user |
| UP_days_since_prior_order | uint16 | days since user last ordered product |
| UP_days_since_prior_order_score | float16 | normalize above by user's days_since_prior_order |
| UP_reordered | bool | boolean indicating whether the product was ever reordered by user |
| UP_order_ratio | float16 | fraction of baskets in which a given product appears for a given user (count of orders in which product appears divided by total orders) |
| UP_penultimate | bool | products in user's penultimate (previous) order as bool (train and test sets contain ultimate order) |
| UP_antepenultimate | bool | products in user's antepenultimate order as bool |
| UP_order_dow_score | float16 | ultimate score for order_dow using (U_ultimate - P_order_dow_mean) / P_order_dow_std (intuitively, how 'far' is a user's ultimate order dow from the mean dow product is ordered) |
| UP_order_hour_of_day_score | float16 | ultimate score for order_hour_of_day using (U_ultimate - P_order_hour_of_day_mean) / P_order_hour_of_day_std (intuitively, how 'far' is a user's ultimate order hour_of_day from the mean hour_of_day product is ordered) |
| LDA_1 | float16 | Latent Dirichlet Allocation Feature 1 |
| LDA_2 | float16 | Latent Dirichlet Allocation Feature 2 |
| LDA_3 | float16 | Latent Dirichlet Allocation Feature 3 |
| LDA_4 | float16 | Latent Dirichlet Allocation Feature 4 |
| LDA_5 | float16 | Latent Dirichlet Allocation Feature 5 |

| | | |
|---|---|---|
| LDA_6 | float16 | Latent Dirichlet Allocation Feature 6 |
| LDA_7 | float16 | Latent Dirichlet Allocation Feature 7 |
| LDA_8 | float16 | Latent Dirichlet Allocation Feature 8 |
| LDA_9 | float16 | Latent Dirichlet Allocation Feature 9 |
| LDA_10 | float16 | Latent Dirichlet Allocation Feature 10 |

## 9.2 Appendix II: Notebooks

The following notebooks contain the code for this project as well as additional discussion particular to the code itself. Some of the linked notebooks repeat parts of the above discussion in order to be more readable as independent documents.

### 9.2.1 Instacart: Exploratory Data Analysis

Examines the raw data prior to making any predictions.

### 9.2.2 Instacart: Feature Engineering

Designs nearly 50 features from the raw data. These features are columns of $X$.

- Instacart: LDA GridSearchCV (Course) is a coarse parameter search for Latent Dirichlet Allocation parameters.
- Instacart: LDA GridSearchCV (Fine) is a fine parameter search for Latent Dirichlet Allocation parameters.
- Instacart: Non-negative Matrix Factorization is a work in progress aimed at constructing features derived from matrix factorization of user-user product counts and variants.

### 9.2.3 Instacart: Random Forest ParameterGrid Search

Conducts a hyperparameter search using `sklearn.model_selection.ParameterGrid`, which permits sufficient flexibility for out-of-bag (OOB) samples to be used in cross-validation. A history of previous searches is available at Kaggle.

### 9.2.4 Instacart: Top-N Random Forest Model

Trains a random forest classifier using the best parameters found in Section 9.2.3. In fact, sklearn.ensemble.RandomForestClassifier provides an array of probabilities. From the ranking determined by this array, Instacart: Top-N Random Forest Model picks the top-$N$ user-product pairs, for various meanings of $N$.

### 9.2.5 Instacart: Prediction Explorer

Constructs a visualization utility to inspect model performance for a given model and user.

## References


[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003. URL: http://jmlr.csail.mit.edu/papers/v3/blei03a.html.

[Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001. URL: https://doi.org/10.1023/A:1010933404324, doi:10.1023/A:1010933404324.

[Bro18] Jason Brownlee. How and When to Use ROC Curves and Precision-Recall Curves for Classification in Python, August 2018. URL: https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/.

[CAL04] Chao Chen, Andy Liaw, and Leo Breiman. Using Random Forest to Learn Imbalanced Data. Technical Report 666, University of California, Berkeley, Department of Statistics, July 2004. URL: https://statistics.berkeley.edu/tech-reports/666.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York, New York, NY, 2009. URL: http://link.springer.com/10.1007/978-0-387-84858-7, doi:10.1007/978-0-387-84858-7.

[JCDLT13] Laszlo A. Jeni, Jeffrey F. Cohn, and Fernando De La Torre. Facing Imbalanced Data–Recommendations for the Use of Performance Metrics. In *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, pages 245–251, Geneva, Switzerland, September 2013. IEEE. URL: http://ieeexplore.ieee.org/document/6681438/, doi:10.1109/ACII.2013.47.

[LNZ+16] Guimei Liu, Tam T. Nguyen, Gang Zhao, Wei Zha, Jianbo Yang, Jianneng Cao, Min Wu, Peilin Zhao, and Wei Chen. Repeat Buyer Prediction for E-Commerce. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, pages 155–164, San Francisco, California, USA, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2939672.2939674, doi:10.1145/2939672.2939674.

[QCJ18] Massimo Quadrana, Paolo Cremonesi, and Dietmar Jannach. Sequence-Aware Recommender Systems. *arXiv:1802.08452 [cs]*, February 2018. URL: http://arxiv.org/abs/1802.08452, arXiv:1802.08452.

[RFS10] Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. Factorizing personalized Markov chains for next-basket recommendation. In *Proceedings of the 19th International Conference on World Wide Web - WWW '10*, page 811, Raleigh, North Carolina, USA, 2010. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1772690.1772773, doi:10.1145/1772690.1772773.