Gousto Data Science Task - Recipe Allocator Algorithm

Daniel Waller

October 2, 2020

1 Introduction

In this document, I describe my approach to solving the Recipe Allocator Algorithm task, and discuss relevant issues, pertaining both to business risks that may be presented by the approach, and to more technical aspects.

1.1 Usage

All functions are contained within the `algorithm.py` file. One method of usage: navigate to the file location in a Python shell and execute the command:
`>>> exec(open("algorithm.py").read())`.
Then call:
`default_orders_satisfied(<order_filepath>,<stock_filepath>)`
which will return either `True` (if all orders can be satisfied) or `False` (if not), as required.

1.2 Algorithm design

How it works. The algorithm takes all orders, sorts them (according to rules defined in the next section) and then allocates recipes to each one sequentially. For each order, the algorithm decides which recipes to allocate by choosing the N-highest remaining stocks, where N is the number of recipes-per box; this maximises possibilities for future allocation. It then removes the appropriate amount from stock and moves on. If a customer cannot be allocated recipes (due to a lack of stock) it returns False; if it processes all customers successfully, it returns True.
Sorting the orders. The key part of the algorithm is deciding how to sort customers. We have three pieces of information about each: meal type (vegetarian or gourmet), the number of recipes-per-box, and the number of portions per recipe. When allocating recipes to fulfil the orders, it was necessary to first work out the optimal order in which to do so. The following set of rules describes the approach I took:

1. Fulfil orders with **highest number of portions** first. This is to avoid cases where remaining stock levels are spread too thinly across the board to satisfy orders with larger numbers of portions. For example, a situation where:
    - 1 customer gourmet, 2 recipes, 4 portions
    - 2 customers vegetarian, 4 recipes, 2 portions
    - stocks of (6,6,2,2,2,2,2,2) (all vegetarian)

Order	Num. of portions	Meal type	Num. of recipes
1.	four	vegetarian	four
2.	four	vegetarian	three
3.	four	vegetarian	two
4.	four	gourmet	four
5.	four	gourmet	three
6.	four	gourmet	two
7.	two	vegetarian	four
8.	two	vegetarian	three
9.	two	vegetarian	two
10.	two	gourmet	four
11.	two	gourmet	three
12.	two	gourmet	two

Table 1: The order in which customers are processed by the algorithm in the default case.

is a simple one where serving either by meal type or most recipes first would not find an allocation, but serving by portions first would.

2. If 1. is equal, fulfil **vegetarian orders before gourmet orders**. This is because vegetarian customers are only able to receive a subset of the recipes that gourmet customers are. As a further precaution, gourmet orders are always filled from gourmet stocks if possible. However, if (and only if) it is not possible, the algorithm widens the recipes to include the vegetarian options.

3. If 1. and 2. are equal, fulfil orders with **highest number of different recipes** first. Clearly, orders with fewer recipes-per-box can be fulfilled more easily, and therefore should be left for last.

With the default settings of 2,3 or 4 recipes per box, and either 2 or 4 portions, this leads to customers being processed in the order shown in Table 1.

1.3 Issues for the business to consider

It is important to note that the recipe allocator is only designed for the purpose of checking if all orders can be fulfilled, and not to actually make the recipe allocations for those customers. The algorithm is usable for the latter purpose, but would have the following drawbacks:

- It does not include information on which recipes particular customers are likely to prefer, for example the similarity of the recipes to previous recipes they have enjoyed.

- It 'prioritises' some customers over others, leading to possible discrepancies in the distribution of recipes depending on what category they are in. For instance, four-portion allocations are dealt with first, meaning that two-portion customers may only get 'what's left'.

These drawbacks could impact on customer's enjoyment of their boxes, and consequently damage (amongst other things) customer retention, the reputation of the company, and future growth.

1.4 Issues - technical

- **Agnosticism.** The algorithm can comfortably adapt to an increase in the number of total recipes in stock. It can also accommodate different numbers of recipes-per-box and/or portions, with just minor additions to the dict object that converts the strings to numbers.

- **Logging.** Logging has been implemented using the logging module - the logging messages are recorded in the file recipeallocator.log. The log records the current characteristics of the orders being filled, so that in the event of `False` being returned, it can be seen which category the algorithm had reached. In addition, the log will print the number of orders that are left in the category.

- **Scalability.** Conducting tests with different numbers of orders, I recorded approximate run times (on my laptop):

  - 1,000,000 orders - 11 seconds
  - 12,000,000 orders - 140 seconds
  - 120,000,000 orders - 1517 seconds

  This seems to indicate that the computation scales linearly with the number of orders.

  To improve the run-time, it might be possible to extend the algorithm to process some customers in batches, rather than running through each order individually. At present, the stock levels are sorted using the `numpy.argpartition` function before processing each order; however, in many situations, the same recipes will be allocated for a number of consecutive orders, due to the pattern of stock levels. This possibility was noticed while scoping the task; ultimately I decided to leave it as an extension.