APPENDIX : Recommendation algorithm

Definitions and Assumptions

Denote set of users U, set of labels L, set of ingredients I and set of recipes as R.
Let $p_L(u,l) : U \times L \to [0,1]$ function, which we can denote with single updatable 2D array. Let $p_I(u,i) : U \times I \to [0,1]$ function, also managed as a 2D array. Let $g(r) : R \to \mathcal{P}(I)$, indicating subset of all ingredients used for recipe r. In same way define $f(r) : R \to \mathcal{P}(L)$.
Let $p(r,i) : R \times I \to \mathbb{Z}^+$, indicating quantity of ingredient i used in r, $q(u,i) : U \times I \to Z^+$, indicating quantity of ingrdient i which user u holds. Note that every mathematical function call here is actually a database query.
Let $s(u,r) : U \times R \to \overline{R}$ function, which denotes score of recipe r in user u's perspective. Our recommendation is essentially computing the score function, and randomly drawing an element with unequal probability depending on score of each element.
For loose measure, we assume we have :

- 1e4 recipes
- 20 labels for each recipe
- 20 ingredients for each recipe

Which is all slightly larger estimation of data we have.

Formula

Define $s(u,r) = s_1(u,r) + s_2(u,r) + c$. These functions respectively computes

- s_1 : score of 'labels' in recipe r.
- s_2 : score of 'ingredient' in recipe r
- c : constant, will be discussed later

Feasibility of recipe must first be checked to determine if $s(u,r)$ is $-\infty$. This is done by ruling out the recipes which does not meet our criterion, such as minimum-rating or maximum-calories. This can be efficiently and easily done with PostGreSQL.

s_1 function

s_2 function is managed exactly the same way, using ingrdients.

We define $s_1(u, r)$ as

$$s_1(u, r) = \max \left(\sum_{l \in f(r)} \left(\frac{p_L(u, l)}{\sum_{t \in L} p_L(u, t)} \right), 0 \right)$$

To make $s_1(u, r) \in [0, 1]$, we have to normalize each p_L. Instead of storing normalized p, for performance we manage norm $\|p_L(u)\| = \sum_{t \in L} p_L(u, t)$ seperately with some abusing of notation.

Norm can be queried at $O(1)$ time. After querying whole p_L, again everything will be computed in $O(\sum |f(r)|)$ time, -1e5 operations. This norm-managing is done by storing sum scores as a seperate database entry, and updating it every time p_L is increment/decremented.

Update

If user like/dislike our recommendation r^*, we increment/decrement $p_L(u, l)$ for all $l \in f(r^*)$ by 1. $\|p_L(u)\|$ can be efficiently computed during every update.

Choice

Always choosing the maximum-score element will lead to significant bias, especially on first few recommendations. Therefore it is more reasonable to assign $t(u, r)$, the probability function, computed from each score $s(u, r)$, and randomly choose with probability $t(u, r)$ for recipe r. Our formula for this task is as following.

$$t(u, r) \propto \left(1 + e^{\frac{1}{s(u,r)^2}} \right)^{-1} + 0.01$$

For $s(u, r) = 0$, which indicates user has "noticable dislike" to this recipe, we assign probability of zero.[1]

There are reasons why we use such formula. Since s is in range $[0, 1]$, curve of the given formula is very flat in the low-score range. Note that the curve being flat is saying that probability difference is relatively smaller. We do not want to introduce a large gap within small number of choices. If the function is linear, we have observed that first few random recommendations governing the whole process for very long time, since when they are upvoted several times they will have significantly larger probability to be recommended next time. By using such function, we smooth out the early-advantage effect, while introducing large enough probability gap. The value 0.01 is that lowest-possible probability is about 1/40 of highest probability, which we thought is reasonable.

The idea between numbers is that we want our users to discover exciting new recipes which he/she wouldn't think of, but still might be enjoyable. We don't want our recommendation to converge too early, which will lead to user seeing same recipes over and over again.

[1]This is possible with use of base score, which we discuss right after

Base score

We introduce a small base probability c to each recipe. The purpose of this c is to prevent recipes from being lost by user too soon. Note that ingredient/label score will be propagated to other recipes. Again, since high-scoring recipes have higher probability to be chosen in the first place, this will make those foods even harder to be chosen.
For example, without c, one dislike to beef stew will decrement beef, onion, carrot and whatever ingredient used. This dislike can be critical in a sense that now onion-related foods are less chosen, user cannot express his/her preference toward onion through like/dislike on the recipe. Small constant score can combat this problem. There are three criterion we used to decide c.

- c should be small such that c should not be dominant factor of computing probability.
- c should not be too small to serve it's own purpose.

By considering all those factors and experimenting with different values, we chose $c = 0.2$ (about 1/10 of maximum score). This will be translated to about 0.09 of s-score.

Fallback

As stated above, we agreed that recommending something new is very crucial to our app. We therefore programmed such that if recommendation algorithm has 'converged' too much, i.e, if single recipe has too high chance of being chosen, we simply discard it and draw some recipe randomly from filtered recipes. If we have too small number of filtered recipes, we also discard filters. If filter is too strict that most recipes cannot pass, we believe that what user actually want is to 'search' with that filter instead of being recommended with such recipes over and over again. We provide recipe search feature, so there is basically no point of recommending 3 or 4 recipes in a cycle.
As stated, if something is wrong, we resort in random. This is reasonable behavior only because we have thousands of recipe datas, and all filters applied to recommendation should be considered "preference" instead of "strict inclusion". In contrast, search recipe works in strict inclusion manner.

Performance

As stated, we can bound our algorithm's time complexity as $O(\sum |f(r)| + |g(r)|)$ which is about 1e5 operations. Since SQL queries are very slow relative to internal data structures, we optimize by initially fetching all the required data with SQL query, storing into python internal data structures such as list, dict and set, and update at the last to minimize the cost of database operations. Also we filter our recipe set as soon as we can, to reduce cost of actual probability computation. These methods of optimization has improved our performance significantly, which we were able to run recommendation on Amazon free-tier instance with very limited computing power.