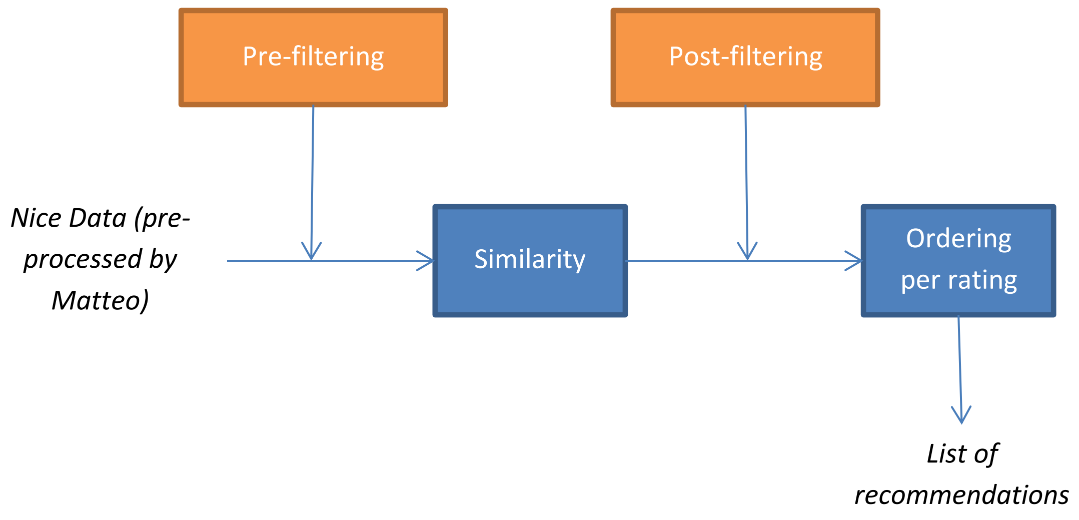
Recommender System

Alexandru Ardelean

Julia Chatain

Global Architecture:



The recommender principle is very basic: it takes as input a matrix of venues x features, and a vector of features that the user asked for. From that, it computes the similarity between the venues' features and the user's features, and returns the venues ordered by similarity.

To improve this, we add two plugins (that can be removes and modified without compromising the good execution of the recommender): a plugin for pre-filtering, and one for post-filtering.

The approach taken in our system will be quite simple: the pre-filtering component will attempt to reduce the search space as much as possible based on the current context, while the post-filtering component will use the context in order to re-evaluate the rankings and increase the chance that better recommendations will be found at the top of the list.

A few words on context hierarchies first. All context dimensions will be organised in tree hierarchies, rooted in the most general aspect and going down to finer granularities. In order to deactivate these two plugins, you should be able to provide a very loose context, using the roots of the context hierarchy trees.

Pre-filtering:

In order to provide the user with a list of recommended venues, the system has to compute the vector similarity between the user vector and each of the venue features vectors and calculate a similarity ranking for each one. Based on a context vector that is computed based on the user's current situation, preferences and whereabouts, we can attempt to remove the venues that will definitely not make it on the final recommendation list. This is a very simple way of increasing the performance of the system.

Input: The pre-filtering block takes as input the venue-feature matrix and the user vector. In addition, it will also take a context vector as a parameter, and will use this in order to filter out any venues that do not match it.

Output: The output of this component is of the same type as the input (thus this component can be unplugged without affecting the system). In particular, it will output a venue-feature matrix with a reduced *horizontal* and *vertical* dimension. This means that for all features that do not match the context, not only is the venue removed from the matrix (horizontal reduction), but also the feature itself will be removed in order to reduce the time required to calculate the similarity (this can be a simple logical removal – changing the weights of this feature to 0 – or a hard removal that actually reduces the number of columns in the matrix and in the user vector).

The context variables that will be taken into account in the pre-filtering block for the moment are:

- TimeOfDay: if the user specifies a time of day in which he would like to visit the venue, then all venues that have opening hours available and are not open in the time range the user specified will be removed from the matrix

- Location: the location filtering is two-fold. In the pre-filtering phase we will remove the venues that are, let's say, in a different town than the estimated location of the user when asking for the recommendation. In the post-filtering phase we will also attempt a re-weighting of the rankings based on proximity.

- Category: should the user ask to be recommended museums, then all venues that do not fit this category will be removed. We will make use of the venue category tree from foursquare in order to also implement a generalization step here: should the user ask for Ski Trail, we will keep in mind to offer a higher weight to such venues in the post-filtering, but wil only remove venues that are not in the parent category (in this case Ski Area).

- Price: the user has the opportunity to specify a price range in his query as a categorical value (cheap, expensive etc). All venues not in this price range will be removed.

Other features might be inserted in this component, but for the moment we want to make it as unobtrusive as possible, so as to minimize the useless calculations, but not to make exaggerated assumptions about what the user wants to see in the results page. We will leave that to the actual similarity calculator.

Post-filtering:

Input: The post-filtering block takes as an input a matrix with venues as rows and features as columns. This matrix also has an extra column that is the rating given by the similarity-computation block. Moreover, the extended post-filtering plug-in will require a context vector (with features that has to be defined, but for example, position of the user, former venue, company (boyfriend, colleagues, etc), etc).

Output: The output will be of the same type than the matrix input (as the plug-in has to be transparent for the recommender). It will just change the ratings according to some parameters.

Goal: The goal of the post-filtering plug-in is to re-evaluate the weights according to the features.

Detailed description: Let's call V the matrix of the venues. This matrix is composed of a big matrix A of venues x features, and an extra column r of ratings.

We assume that we have a vector w of the same length as the width of matrix A. This vector contains the weights we give to each feature. For example, it will have a higher value for the feature "Price" if we decide that this feature is more important than another feature, like "Distance".

We also have the context vector c, input of the whole plug-in.

We have a big function F that will take as an input A, w and c, and that will return a column vector of size equal to the height of the matrix A. This vector contains the rating modificator for each venue. The final rating will be obtained by summing this vector and the vector r of input ratings.

The function F will be composed of sub-functions, for each feature, and will return the following (if n is the number of features, and if we only consider the line i of the matrix A):

$$F\left(A_{i,*}, w, c\right) = \sum_{j=1}^{n} w_j f_j(c, A_{i,j})$$

The content of the functions f_j has to be defined according to each feature.