

ALKINDI MOVIE RECOMMENDATION ENGINE

ALGORITHM SPECIFICATION IV:

ALKINDEX

Eugene Stern
Alkindi, Inc.

May, 2001

CONFIDENTIAL

1 Introduction

A user's *Alkindex* is meant to be a measure of how well we know the user's tastes. It ranges from 0 to 1. Intuitively, an Alkindex of 0 means we have no sense of the user's tastes and can do no better than recommend products to them based on the ratings of the entire user base. An Alkindex of 1 means that the user is perfectly clustered with other users of identical tastes. (Strictly speaking, since we have different groupings of our user base for each product cluster, an Alkindex of 1 means that the user is perfectly clustered with respect to every product cluster.)

2 Fit With Respect to One Product Cluster

2.1 Personalized and Unpersonalized Lists

To begin with, we imagine that we have one product cluster. A user is assigned to a user cluster based on the product cluster; the user cluster makes personalized recommendations of products in the product cluster to the user. In principle, the user cluster *recommends the same products to each of its member users*. This is not strictly true, because when we actually recommend products to a particular user, we filter out products that the user has already rated or that have recently been recommended to the user. (Here, and throughout this document, by "rated," we mean that the user has seen or purchased a product (e.g., seen a movie) and submitted a rating between 1 and 6 for that product.)

However, such filtering is inappropriate in our current context, because what we want to do is evaluate the quality of personal recommendations by comparing them to a user's actual ratings. Consequently, for each user cluster, we compile a single list of movies that, ignoring any filtering, we imagine the user cluster recommending to all of its members. To

construct the list, we use a single scoring function (a tunable parameter, set to S , where S is between 1 and 8, specifies which one is to be used). Given a user cluster, we evaluate the scoring function on all the products in the product cluster that are used for clustering users. The user cluster's list consists of the D top scoring products. (The list depends on the user cluster because when we evaluate the scoring function at a given product in the product cluster, we use the ratings assigned that product by users in the user cluster.)

Remark. A product cluster may consist of many products, only some of which are used in clustering users. Here we will refer to these as **coreProducts** in the product clusters. For purposes of calculating the Alkindex and all quantities it depends on, only **coreProducts** in each product cluster will be considered. This is reflected in the reference above to evaluating the scoring function only on those products in the product clusters that we use to cluster users.

A user cluster's list of "top D products" can be compiled by a function **getTopProductsInCluster**. This function takes a **userClus** as input and does the following:

```
list = products in product cluster used for clustering users;
scoringFunction = scoringFunctionArray[S];
for (int iprod = 1; iprod < list.size(); iprod++) {
    score(iprod) = evaluate(scoringFunction, iprod, userClus);
}
Sort members of list by score;
Return top D scoring members of list.
```

We intend to compare the personalized list above (compiled based on the ratings of a user cluster) with an unpersonalized list (compiled based on the ratings of all users). The unpersonalized list can be computed by a function **getTopProductsOverall()**, which can be implemented simply as **getTopProductsInCluster(bigClus)**, where **bigClus** is a "dummy cluster" made up of all users of the system.

2.2 Good and Bad Recommendations

Intuitively, we think of **getTopProductsInCluster(userClus)** as returning a list of personalized recommendations good for all members of **userClus**, and **getTopProductsOverall()** as returning an analogous list of unpersonalized recommendations. Evidently, the personalized recommendations will be more appropriate for some users in the user cluster than others. Our goal is to see how much the personalized recommendations improve on the unpersonalized recommendations for a given **user** in **userClus**. We do this by looking at cases where users have rated products on the two lists of recommendations and seeing if their ratings are high. If a user has rated a recommended product highly (R or above), we consider the recommendation good; otherwise, we consider it bad.

2.3 Proxy Subgroups

It is impractical to compare how many of the recommendations on our two lists are good or bad for a single user. Instead, we divide the user cluster up into several (8) subgroups and compute and store a percentage of bad personalized recs and a percentage of bad unpersonalized recs that applies to each user in the subgroup.

A user's subgroup membership is based on the fraction of the total number of `coreProducts` in the product cluster that the user has rated. Precisely, consider seven tunable parameters $\Theta_1, \Theta_2, \dots, \Theta_7$, all lying between 0 and 1 with $\Theta_i \leq \Theta_{i+1}$. Set $\Theta_0 = 0, \Theta_8 = 1$. We divide the user cluster up into subgroups G_1, G_2, \dots, G_8 , where the membership of G_j consists of those users in the user cluster who have rated at least Θ_{j-1} , but less than Θ_j of the `coreProducts` in the product cluster. (If $j = 8$, only the left boundary condition should apply.)

We introduce integer parameters $\nu_1, \nu_2, \dots, \nu_8$ with $\nu_j \geq \nu_{j+1}$ and $\nu_8 = 1$. (A typical set of values is $(\nu_1, \dots, \nu_8) = (15, 8, 6, 4, 3, 2, 1, 1)$.) Continuing with j as an index between 1 and 8, for a given user cluster `userClus`, we define \tilde{G}_j to be the union of:

- The subgroups G_j and G_{j+1} of `userClus`. (If $j = 8$, we take G_9 to be \emptyset , and we just include G_8 in \tilde{G}_8 .)
- The subgroups G_j and G_{j+1} in the $\nu_j - 1$ user clusters closest to `userClus`. (If $\nu_j = 1$, we consider `userClus` alone and do not include neighboring clusters. Distance between user clusters is measured as distance between their means in rating space.)

For the user cluster `userClus`, and for $j = 1, \dots, 8$, we now define two numbers b_j and B_j . We take the list `getTopProductsInCluster(userClus)` and count up good and bad recommendations on this list over all users in \tilde{G}_j . (More precisely, each time a user in \tilde{G}_j has rated one of the products on the list `getTopProductsInCluster(userClus)` as R or above, we increment a counter `goodRecs`, and each time a user in \tilde{G}_j has rated one of the products on the list `getTopProductsInCluster(userClus)` below R , we increment another counter `badRecs`.)

Remark. Note that even though the products we are on our list were recommended by `userClus`, we are sometimes considering whether they are good or bad recommendations for users outside `userClus`. This reflects the fact that we're unsure whether a typical user in the subgroup G_j of `userClus` should really belong to `userClus` or to a neighboring user cluster. As the number of products the user has rated increases, the number of neighboring clusters we imagine the user might belong to goes down.

2.4 b and B : Fractions of Bad Recommendations

Having totaled up the good recs and bad recs from `getTopProductsInCluster(userClus)` over all the users in \tilde{G}_j , we now define b_j to be fraction of bad recs (bad recs divided by good plus bad recs). This is a quantity depending on `userClus` and on j , so we can think of it as a quantity depending on the subgroup G_j of `userClus`.

Pseudocode for a function `compute_b(userClus, j)` goes as follows. (Recall again that all user clusters under consideration are with respect to a particular product cluster, which has been fixed since the second paragraph of this document.)

Let $\text{subGroup}(\text{userClus}, j)$ be a function returning the membership of the subgroup G_j of userClus , and let $\text{nu}(j)$ be a function returning ν_j . Let $\text{union}(\text{list1}, \text{list2})$ be a function returning the list of all members of either list1 or list2 .

```
list = getTopProductsInCluster(userClus);
proxyGroup = subGroup(userClus, j);
if (j < 8) {
    proxyGroup = union(proxyGroup, subGroup(userClus, j+1));
    if (nu(j) > 1) {
        Sort user clusters by distance from userClus;
        neighbors = list of nu(j) - 1 user clusters closest to userClus;
        for (int iclus = 0; iclus < neighbors.size(); iclus++) {
            proxyGroup = union(proxyGroup, subGroup( neighbors[iclus], j) );
            proxyGroup = union(proxyGroup, subGroup( neighbors[iclus], j+1) );
        } // end for (iclus)
    } // end if (nu(j) > 1)
} // end if (j < 8)
goodRecs = 0;
badRecs = 0;
for (int iuser = 0; iuser < proxyGroup.size(); iuser++) {
    for (int iprod = 0; iprod < list.size(); iprod++) {
        goodRecs++ if iuser rated iprod R or above;
        badRecs++ if iuser rated iprod below R;
    } // end for (iprod)
} // end for (iuser)
return badRecs / (goodRecs + badRecs).
```

For the subgroup G_j of userClus , we also compute a quantity B_j by replacing the list $\text{getTopProductsInCluster}(\text{userClus})$ by $\text{getTopProductsOverall}()$. Equivalently, we define a function $\text{compute_B}(\text{userClus}, j)$ by replacing the first line in the pseudocode above by

```
list = getTopProductsOverall()
```

Note that it would be incorrect to define $\text{compute_B}(\text{userClus}, j)$ as $\text{compute_b}(\text{bigClus}, j)$. This is because userClus figures in compute_b both in generating the list of products and in generating the proxy group. When we compute_B , we use bigClus to generate the list of products but still use userClus to generate the proxy group.

Given a user , we compute_B for the user by assigning the user to a userClus , a subgroup G_j within the userClus , and setting $\text{compute_B}(\text{user}) = \text{compute_B}(\text{userClus}, j)$. We $\text{compute_b}(\text{user})$ similarly, with two exceptions:

- If user has rated fewer than τ percent of the coreProducts in the product cluster, we set $b = B$.

- If $\text{compute_B}(\text{userClus}, j) < \text{compute_b}(\text{userClus}, j)$, we set $b = B$.

Thus, $\text{compute_b}(\text{user})$ is defined as follows:

```
Find userClus, j so that user belongs to subGroup(userClus, j);
if (compute_B(userClus, j) < compute_b(userClus, j) ||
    user rated less than  $\tau$  percent of coreProducts in cluster) {
    return compute_B(userClus, j);
} else {
    return compute_b(userClus, j);
}
```

Summarizing, given a **user** and a product cluster, we have associated with the **user** two numbers B and b based on the product cluster. These numbers are computed entirely in terms of prestored quantities B_j and b_j associated with subgroups of user clusters.

3 Alkindex: Fit for All Product Clusters

3.1 b_i and B_i : b and B by Product Cluster

Now let $i = 1, \dots, N$, where $N = \text{num_prod_clus}$, be an index over the product clusters we use to cluster users. Abusing notation, given a **user**, let B_i and b_i be the quantities B and b computed for the **user** as above with respect to the i -th product cluster.

3.2 Definition of the Alkindex

We then define the **user**'s Alkindex to be

$$\frac{n_1(B_1 - e^{-cf_1}b_1) + \dots + n_N(B_N - e^{-cf_N}b_N)}{n_1B_1 + \dots + n_NB_N}. \quad (1)$$

Here n_i is the number of products in the i -th product cluster, counted appropriately (this means that if a product lies in 5 product clusters, it contributes $\frac{1}{5}$ to the count of elements in each of these product clusters). f_i is the fraction of **coreProducts** in the product cluster rated by the user, and c is a tunable parameter (typically 3/2).