

ALKINDI MOVIE RECOMMENDATION ENGINE ALGORITHM SPECIFICATION II: RECOMMENDATION MANAGER

Eugene Stern
Alkindi, Inc.

May, 2001

CONFIDENTIAL

1 Introduction

Alkindi's recommendation engine is based on an approach to recommending known as *collaborative filtering*. Collaborative filtering simulates system *users* collaborating to recommend *products* to each other, using each other's opinions to filter out all but the most relevant content. In the case of the movie recommendation engine, the products are movies, or videos, which are identified using a unique correspondence between product identifications and titles.

More specifically, Alkindi's engine works by:

1. **Input Step:** Building up an individual taste profile, consisting of *evaluations* of *products*, for each user;
2. **Clustering Step:** For each user, identifying other users with similar taste profiles;
3. **Output Step:** Recommending to each user products that users similar to the user have tended to enjoy.

1.1 Role of This Document

This document specifies the Output Step — the routines Alkindi's Recommendation Manager uses in making recommendations to users based on clustering data. It assumes the machinery developed in "Algorithm Specification I: Clustering Manager."

This is an Algorithm Specification document; it specifies *what* quantities need to be computed by a component of Alkindi's recommendation engine (in this case, by the Recommendation Manager), but does not specify *how* those quantities are to be computed from a software design standpoint. For the sake of concreteness and specificity, this document

describes a number of computational steps as subroutines, with particular inputs and outputs. The intention is only to specify underlying algorithmic ideas in an understandable way, and in common language, and not to insist on a particular software design, submodules, or interfaces between the modules. The actual functionality and design of the software are described in separate documentation (the Functional Requirements Specification and the Software Requirements Specification). The FRS, titled “Recommendation Manager Functional Requirements Specification,” is in a file called `FRS_Recommendation.doc`. Occasionally, the present document leaves out details which are given in the FRS.

1.2 Tunable Parameters

There are many fixed parameters that play a role in Alkindi’s recommendation engine, which can be tuned in attempt to improve performance. (For example, the number of days that pass from the time a product was recommended to the time that product can be recommended again could be one such parameter.) We call these *tunable parameters*.

Tunable parameters are pointed out in this document whenever they appear. All the tunable parameters are compiled in a separate document, which also lists current values for the parameters. Since this document contains parameters for all the modules of the engine (clustering, recommending, etc.), the notation identifies both the module and the name of the parameter in the specification of that module. For example, a recommending-related parameter named *A* in this document would be named `REC_A` in the Tunable Parameters document.

Some parameters associated with the engine as a whole, but not specifically with the Recommendation Manager, also appear in this document. These parameters are referred to here by the names assigned to them in the Tunable Parameters document. (For example, a general, engine-related parameter called `ENG_RECmin` in the list of tunable parameters is referred to by the same name in this document.)

1.3 Assumptions and Data Used

We assume that the products that may be recommended are grouped into *product clusters*. For each product cluster, the Clustering Engine specified in “Algorithm Specification I: Clustering Engine” partitions the users who rated core products in that product cluster into user clusters. (The user clusters are constructed so that the members of a single user cluster rate products in the corresponding product cluster similarly.) A single product can belong to more than one product cluster, and a user belongs to a user cluster corresponding to each product cluster that contains at least one core product that the user has rated. (Definitions, and more details, can be found in the algorithm specification of the Clustering Manager.)

The clustering of the user base occurs offline, and is followed by a process of calculating and storing in the database a broad range of statistics associated with the clusters. This takes place offline as well, and allows the Recommendation Manager (as well as the Rating Manager, which is specified in the document that follows this one) to simply look up in the database many statistics that would be difficult to calculate in real time. The calculation and storage of these statistics are documented separately. The database tables and their columns are described in an Excel spreadsheet called `pkgs_deps.xls`. The relationships

between the tables are documented in an Entity Relationship Diagram stored in the file `Alkindi_ERD.jpg`.

1.4 Structure of This Document

The core routine returns a single recommended product. It is described in section 2.

Typically, the system is asked for several recommendations (perhaps 4 or 8) at once. This is essentially done by calling the routine of section 2 the appropriate number of times, but section 3 briefly supplies any necessary details.

Recommended products come with a predicted rating — the system’s best guess for how the user would rate the product. This applies more generally than just to recommended products; for example, products that come up as search results also come with predicted ratings. Section 4 describes how the predicted ratings are computed.

2 Making a Single Recommendation

2.1 Overview

For concreteness, we refer to the routine that generates a single recommended product as `get_recommendation`. It uses the the user’s cluster membership and the cluster statistics described above to output a single recommended product to a user.

Very broadly, the routine does the following:

Pick product cluster to recommend from.

Pick a particular method of recommending.

Identify and prioritize recommendable products in the product cluster.

Recommend a product using the chosen method.

By “method of recommending,” we mean a scoring function that associates a number (a score) with each recommendable product.

A product is *recommendable* if it has a certain number of ratings. This number, called `ENG_RECmin`, is a tunable parameter.

The prioritization of recommendable products will be explained in more detail below; the basic idea is that we prefer to recommend some products rather than others, regardless of what their scores are. In other words, lower priority products are never recommended if any higher priority products are available. For example, we prefer to recommend products that we have not recommended to the user in the past to those we have, so products that have not already been recommended get higher priority.

The final step, recommending, consists of selecting the highest priority recommendable product with the highest score.

2.2 Making a Stochastic Choice

Evidently, the key steps are (1) and (2) — the steps at which the routine chooses something, either a product cluster or a scoring function. This is done by assigning a weight to each possibility (cluster or function) and selecting a possibility at random according to the weights.

In more detail, this kind of stochastic choice can be carried out by a function called `choose_by_weights(choices, weights)`, which works as follows. Here `choices` refers to a list of the possibilities from which we are choosing, which could be product clusters, scoring functions, or something else entirely if the function is being used in another context.

```
Check that choices.size() equals weights.size().
Divide each component of weights by sum of entries of weights.
Define intervals[i] = weights[1] + ... + weights[i].
Choose r, a random number between 0 and 1.
Find i so r < intervals[i] && r !< intervals[i-1].
Output choices[i].
```

Note that here we index our arrays starting from 1. Thus the routine normalizes the weights (so that their sum is 1), and associates with each normalized weight (equivalently, with each entry of `choices`) a subinterval in $[0, 1]$ whose length is that normalized weight. We choose a random number between 0 and 1, see which subinterval it falls in, and choose the element of `choices` corresponding to that subinterval.

2.3 Selecting a Product Cluster

This step breaks up into two substeps: computation of the `weights` associated with the product clusters, and a call to `choose_by_weights`, where the arguments are the `choices` corresponding to all the product clusters, and the just-computed `weights`. The second substep already having been explained, we focus here on the first.

Recommendations may come from all product clusters, but the user may also request that recommendations come from a particular subset of product clusters. (In the current implementation of the Alkindi movie recommendation engine, product clusters correspond to genres, and a user may request movies from one genre only, or perhaps several.) Product clusters not requested automatically have weight 0. In addition, product clusters that have already been demonstrated to have no products suitable for recommending also have weight 0. For an explanation of this, see sections 2.5.1 and 3. We assume that at least one product cluster has been requested and has non-zero weight. (If all requested product clusters have zero weight, this means that they have all been examined and found to have no suitable products. In this case, and only in this case, the routine terminates, returning an indication that it was unable to come up with a recommendation. See section 2.5.1 for details.)

2.3.1 Unclustered Users and Counting Products

We prefer to recommend from product clusters in which the user has been assigned to a user cluster (these are the clusters in which we can make personalized recommendations). These are the product clusters in which the user has rated at least one core product (see the Algorithm Specification for the Clustering Manager for details). However, it is possible that the user has not been clustered with respect to *any* of the requested product clusters.

In this case, we weigh the requested product clusters by the number of recommendable products they contain. (Intuitively, product clusters with more products are preferred since there are more products there to recommend from.) Thus, if the i -th product cluster has

been requested, its weight is defined to be n_i , the number of recommendable products in the cluster. (Again, unrequested product clusters have weight 0.) However, products should be counted carefully, since some products can belong to more than one cluster. Let the i -th cluster contain N products P_1, P_2, \dots, P_N . Let $m_{ij} = 1/c_j$, where c_j represents the number of product clusters the product P_j belongs to. Then we set $n_i = m_{i1} + m_{i2} + \dots + m_{iN}$. (Intuitively, we spread out each product's contribution among all the clusters it belongs to.)

2.3.2 All Product Clusters Requested

Assuming the user is clustered with respect to at least one of the requested product clusters, we consider the case where the user has, perhaps implicitly, requested all product clusters. ("Implicitly" means that the user hasn't requested any subset of the product clusters, freeing us to make recommendations from all of them.)

Let S_i be the i -th product cluster. If the user is clustered with respect to S_i , the weight w_i associated to S_i is

$$w_i = n_i \cdot \exp \left(A \cdot r_i + B \cdot \epsilon_i + K\delta_i + \frac{g \cdot I_i - h \cdot N_i}{M_i} \right). \quad (1)$$

Many terms need explanation. To begin with, n_i is the "normalized" number of recommendable products in S_i , defined in section 2.3.1. r_i and ϵ_i are measures of the user's tendency to rate and like products in the i -th product cluster. ϵ_i is the fraction of products in the i -th cluster that the user has rated (out of those the user has been asked about), and r_i is the average rating the user has assigned to those products. (Note that this is computed over all products, not core or even recommendable products.) A and B are tunable parameters; intuitively they are weights that determine the contribution that r_i and ϵ_i make to the overall weight w_i .

δ_i is a measure of the user's well-clusteredness with respect to the i -th product cluster. Specifically, the Algorithm Specification for the Alkindex (an overall measure of how well the system knows the user's tastes) contains a quantity called b_i , which measures the fraction of bad recommendations the user is expected to get from the i -th product cluster. We set $\delta_i = b_i$. The coefficient K , another tunable parameter, should evidently be *negative*.

The final term, $(gI_i - hN_i)/M_i$, measures the interest which the user has indicated in products in the i -th product cluster. When users are shown products, in addition to rating them, they can click on one of "Interested," "Not Interested," or "Don't Know." I_i is the number of products in the i -th product cluster to which the user has responded "Interested," and N_i is the number of products in the product cluster to which the user has responded "Not Interested." g and h are tunable parameters. M_i is defined to be the maximum of $I_i + N_i + D_i$ and Ψ , where U_i is the number of "Don't Know" responses in the i -th product cluster, and Ψ is another tunable parameter. We have $I_i/M_i \leq I_i/(I_i + N_i + U_i)$ and $N_i/M_i \leq N_i/(I_i + N_i + U_i)$, where the right sides of the inequalities are the fraction of Interested and Not Interested responses out of all interest-related responses. Thus, Ψ should be regarded as a lower bound for the number of Interested/Not Interested responses we get before these fractions play a full role.

2.3.3 Only Some Product Clusters Requested

The case where the user has requested only some product clusters is treated analogously, with one difference. If the user has requested a product cluster with respect to which he or she is not clustered, we do not automatically rule that cluster out (after all, the user has asked for it). However, we cannot use the formula 1 for the weight of a product cluster in which the user isn't clustered, because the user has not rated any core products in that product cluster, and may not have any ratings in that cluster at all.

Thus, if the user is not clustered with respect to the i -th product cluster S_i but that user's requested clusters include S_i , the weight w_i of S_i is computed by starting with the formula 1 and replacing $(r_i, \epsilon_i, D_i, I_i, N_i, M_i)$ by corresponding terms $(r_{\text{all}}, \epsilon_{\text{all}}, D_{\text{all}}, I_{\text{all}}, N_{\text{all}}, M_{\text{all}})$. Here:

- r_{all} is the user's average rating computed over all products.
- ϵ_{all} is the fraction of products rated by the user out of all products shown to the user.
- D_{all} is the weighted average of b_i over all product clusters with respect to which the user is clustered; each cluster is weighed by n_i , the normalized number of products in it.
- I_{all} and N_{all} count the user's total "Interested" and "Not Interested" responses to all products (over all product clusters).
- $M_{\text{all}} = \max(I_{\text{all}} + N_{\text{all}} + U_{\text{all}}, \Psi)$, where U_{all} counts the user's total "Don't Know" responses to all products (over all product clusters).

2.4 Selecting a Recommendation Method

2.4.1 Details on Scoring Functions

As indicated above, a recommendation method, or scoring function, is a function that assigns a number, or score, to each recommendable product in a product cluster based on the ratings of a particular user cluster associated with the product cluster. We go on to output the highest scoring product as a recommendation to a user in that user cluster. Two methods of computing a score for each product are

1. Score = average rating given the product by members of the user cluster,
2. Score = number of people in the user cluster who've rated the product.

A generalization of (2) can be given as follows. Our users typically rate products on a scale from 1 to 6. If we let N_i be the number of users who rated the product an i , we can write down a scoring function of the form

$$\text{Score} = c_1 N_1 + c_2 N_2 + \cdots + c_6 N_6, \quad (2)$$

where c_1, \dots, c_6 are constants that determine the weight given to each instance of a particular rating. For example, the second scoring method above corresponds to all c_i being set to 1.

Any formula of the form (2) can be rewritten in the form

$$\text{Score} = d_1 M_1 + d_2 M_2 + \cdots + d_6 M_6, \quad (3)$$

where M_i now represents the number of products rated i or above, and the d_i are a different set of constants. (For example, our second scoring function above corresponds to $d_1 = 1$, all other $d_i = 0$.) Writing scoring functions in the form (3) is convenient because we may have ratings in our database that are not on a 1 to 6 scale. For example, some users may have rated products on a different scale (for example, 1 to 10); when their ratings are rescaled to fit our 1 to 6 scale, they cease being integers.

No formula of the form (2) or (3) will allow us to compute the average rating of a product. However, using the notation in (2) and (3), we can write the average over all users in a cluster as

$$\text{Average} = \frac{c_1 N_1 + c_2 N_2 + \cdots + c_6 N_6}{N_1 + N_2 + \cdots + N_6} \quad (4)$$

$$= \frac{d_1 M_1 + d_2 M_2 + \cdots + d_6 M_6}{M_1}, \quad (5)$$

where $c_i = i$ and $d_i = 1$ for all i . In general, a formula of the form (4) or (5) represents a *weighted average* of user ratings; the c_i and d_i can be viewed as weights.

In addition, notice that formulas of the form (3) and (5) are not restricted to six terms. We can let k be the number of terms; as i runs from 1 to k , we set $M_i =$ number of ratings above r_i , where we choose the thresholds r_i .

A *scoring function* will be any function of the form (3) or (5). The data specifying each scoring function should be viewed as a set of tunable parameters, in the sense that the recommendation engine administrator may add new scoring functions, or adjust already existing ones. To add a new function, one first specifies the form (2 or 4, corresponding to equations (2) and (5)). Once the form is specified, one specifies the number of terms (called k above), the thresholds (called r_i above) and then the weights (the d_i). To modify an existing scoring function, one modifies those parameters.

The parameters for the scoring functions currently being used are given in the Recommendation Manager FRS.

2.4.2 Scoring Function Weights

Each time we compute a recommendation for a user (in real time), we stochastically choose a scoring function to use to generate the recommendation. The choice is made stochastically, as described in section 2.2; here we specify how the weights for each scoring function are computed.

To begin with, each scoring function has an *initial*, or base, weight. If the scoring functions are indexed by $1, 2, \dots, n$, we let SFW_j (Scoring Function Weight) be the initial weight of the j -th scoring function; this is a tunable parameter. Scoring functions for a user's first round of recommendations are always chosen using the initial weights.

Once a user has gotten recommendations, the weights of the scoring functions evolve in response to the user's feedback to recommendations received so far. This works as follows.

As described in section 2.3.2, when we recommend a product, the user has a chance to give feedback: rate the product in case they’ve seen it, and indicate interest (as Interested/Not Interested/Don’t Know) if they haven’t. An answer of Interested/Not Interested/Don’t Know, is also treated as feedback on the recommendation method.

More precisely, we introduce three more tunable parameters a_1, a_2, a_3 , with $a_2 < a_3 < 0 < a_1$. Recalling that we have already selected a product cluster from which to recommend, count the number of times the user has indicated “Interested,” “Not Interested,” and “Don’t Know” to previous recommendations *in the given product cluster*. Letting η_1, η_2 , and η_3 denote the number of “Interested,” “Not Interested,” and “Don’t Know” responses, respectively, set the weight ω_j of the j -th scoring function to be

$$\omega_j = SFW_j \cdot \exp(a_1\eta_1 + a_2\eta_2 + a_3\eta_3). \quad (6)$$

2.5 Exclusion and Prioritization

Once the product cluster and scoring function have been selected, it remains to select a product from the recommendable products in the product cluster using the scoring function. First, we restrict and order the universe from which we may recommend. Some recommendable products may be excluded from recommendation. (For example, in the case of movies: we may exclude R-rated movies if the user is below 17.) The remaining products are be divided up into priority categories, where, for example, recommending a Priority 1 product is always preferable to recommending a Priority 2 product, and so on. Details on the criteria for excluding products, and for the formation of priority categories are given in the Recommendation Manager FRS. (Note that these criteria are given in terms of a range of tunable parameters which are not described in this document, but which are labeled in the tunable parameters document according to the rules outlined in section 1.2.)

2.5.1 No Unexcluded Recommendable Products

It is possible that the selected product cluster will have *no* unexcluded recommendable products. In this case, *and only in this case*, we will be unable to recommend a product from the product cluster.

If this occurs, the routine returns to the beginning (i.e., to the selection of the product cluster). The modification we make is to set to 0 the weight of the product cluster just discovered to have no suitable products to recommend. The weight remains 0 for the rest of the “get a recommendation” routine (see also section 2.3). Moreover, getting a single recommendation is typically a subroutine in a larger routine which compiles a list of recommendations (see section 3). If this is the case, and the weight of a product cluster is set to 0 because of a lack of suitable products during one call to the “get a recommendation” routine, the weight remains at 0 for all subsequent calls to “get a recommendation” made by the “get a recommendation list” routine. For example, if eight recommendations are desired, and a particular product cluster is found to have no suitable products in the course of generating the fifth recommendation, its weight remains at 0 when we generate the sixth, seventh, and eighth recommendations.

If it develops that *all* requested product clusters (eventually) have weight 0 before a recommendation is made, it is impossible to generate a recommendation. In this case, the

“get a recommendation” routine terminates, returning an indication that it could not come up with a recommendation. (In this case, we have looked through all the product clusters, and found no products suitable for recommendation.)

2.6 Recommending

The final step is to generate the recommendation. It will be the recommendable product in the highest non-empty priority group in our chosen product cluster that scores highest under the chosen scoring function. For example, if there are no priority one or priority two recommendable products in our product cluster, but there exist some recommendable priority three products, we recommend the priority three product that scores highest, ignoring lower priority products.

3 Generating a Recommendation List

In general, we return several recommendations to a user at a single time. For example, in the Alkindi movie recommendation engine, four recommended movies are returned on the “My Alkindi” page, and eight recommended movies are returned on the “Alkindi Recommends” page.

To generate a recommendation list with k items, we call the routine that generates a single recommendation k times. We remark on several odds and ends:

- We must guarantee that a single product doesn’t appear more than once in a single list of recommendations. Thus, if a product appears on the list currently being generated, it becomes an excluded product for all future calls to get a single recommendation to add to the current list.
- It was remarked in section 2.5.1 that an attempt to get a recommendation from a particular product cluster might reveal that the cluster has no products to recommend to the user, either because there are no recommendable products, or because all recommendable products are excluded. If so, we set the weight of that product cluster to 0, and keep it at 0 over the rest of the process of compiling the recommendation list. (See also sections 2.3 and 2.5.1.)
- It is possible that eventually *all* product clusters will have weight 0, before we have generated the full list of recommendations. Once all the weights become 0, the “get a recommendation” routine terminates with an indication that it failed to find a product to recommend (see section 2.5.1). In this case, the routine generating a recommendation list terminates as well, returning the recommendations computed so far, along with an indication that it was not possible to generate any further ones.

4 Predicted Ratings

Along with each recommendation, the recommendation manager also outputs a “user’s predicted rating” for the product. (We compute predicted ratings for a product in other contexts

beside recommendations; for example, products returned to a user as the results of a search come with predicted ratings.)

Essentially, the predicted rating is the average rating in the user’s cluster relative to the product cluster containing the product. The complication is that a given product may be in several product clusters, and the user is likely to lie in different user clusters with respect to each of these product clusters.

Thus, let a product lie in several product clusters. To contribute to the predicted rating, a product cluster must satisfy two properties:

1. There must be at least one user clustered with respect to that product cluster who has rated the product. (Note that this means that every user cluster associated with that product cluster has an average rating associated with the product. If no users in the user cluster have rated the product, the average rating is “filled in” from the user clusters that *do* have ratings for the product. See the Algorithm Specification for the Clustering Manager for details.)
2. The user for whom the predicted rating is being computed must be clustered with respect to the product cluster.

If no product clusters satisfy this, the predicted rating will just be the product’s average rating over all users.

Let $\{P_i\}$ be the product clusters satisfying these two conditions, and let U_i represent the user’s user cluster with respect to the product cluster P_i . Let r_i represent the average rating for the product in the user cluster U_i . (If nobody in U_i has rated the product, r_i is the “filled in” average rating.) The predicted rating for the product is a weighted average $\sum w_i r_i / \sum w_i$ of the r_i , where the weight w_i associated with the i -th product cluster is given by

$$w_i = n_i \cdot \exp(A' \delta_i). \quad (7)$$

Here:

- n_i is the number of users in U_i who have rated the product. If no users in U_i have rated the product and we are using a “filled in” average rating, we set $n_i = 1$.
- A' is a tunable parameter.
- $\delta_i = b_i$ (see section 2.3.2), the measure of the user’s well-clusteredness with respect to the product cluster P_i . As remarked in that section, this quantity *decreases* as the user becomes better clustered, so the parameter A' should be negative.

Once we’ve computed the weighted average rating for the movie from all of the user’s relevant clusters, the last step is to round it up or down, as appropriate, to obtain an integer. Thus a weighted average rating of 5.65 yields a predicted rating of 6; a weighted average rating of 4.41 yields a predicted rating of 4.