

ALKINDI MOVIE RECOMMENDATION ENGINE

ALGORITHM SPECIFICATION I:

CLUSTERING MANAGER

Eugene Stern
Alkindi, Inc.

May, 2001

CONFIDENTIAL

1 Introduction

1.1 Collaborative Filtering and Clusters

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.

This document describes the algorithms that underlie the Clustering Step, which is at the core of Alkindi's engine. Other documents describe the Input and Output Steps. Conceptually, clustering should be understood first, followed by the Output Step (which generates recommendations based on the clusters), then the Input Step (which collects preference data to form the best clusters and generate the best recommendations). Consequently, assumptions are made here about data collected on users; documentation of the Input Step explains how data satisfying these assumptions is collected.

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 Clustering

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 design of the software is described in a separate document (the Software Requirements Specification).

1.2 Offline and Online Clustering

The Alkindi engine divides users up into clusters using an iterative procedure called *K*-means clustering. The computational cost of clustering makes it desirable to carry it out offline. The resulting clusters, as well as a range of statistics associated with them, are stored in a database.

Online, the clusters evolve in response to users submitting additional ratings. Users can rate products which are recommended to them, or products which are offered for rating to better characterize the user's taste. (New users can also enter the system by rating some initial products; this is a special case of better characterizing a user's taste.) This changes the user's taste profile, and we update the user's cluster membership based on the updated profile. This enables the system to respond in real time to user feedback.

1.3 Miscellaneous Notes

1.3.1 Current System Status and Future Development

The Alkindi engine is being improved on both the development and the research fronts. While the primary intention of this document is to specify all the calculations carried out in the Alkindi Production System, it also includes elements that have been prototyped and tested in research, but have not yet been introduced into the production system. These elements are marked *Prototyped; In Development*. The document also includes elements that need to be prototyped and evaluated before it can be decided that they should go into the Production system. These elements are marked *To Be Tested*.

1.3.2 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 clusters formed, or a target for the number of users per cluster, 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 clustering-related parameter named ϵ in this document would be named **CLUS.Epsilon** in the Tunable Parameters document.

Some parameters associated with the engine as a whole, but not specifically with clustering, 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_CORE1min` in the list of tunable parameters is referred to by the same name in this document.)

2 The Data and the Model

2.1 Product Clusters

Typically, users' tastes overlap on some, but not all, products. (To take an extreme example, it is unlikely that two people who like the same Western movies will also like the same computer books.) Consequently, rather than trying to group users based on all available products, the Alkindi engine groups them based on subclasses, called *product clusters*, which appear to have predictive value. (Predictive value means we can reasonably expect agreement on some products in a product cluster to indicate likely agreement on others.)

Product clusters in the Alkindi movie recommendation engine currently correspond to movie genres: action films, comedies, etc. The clustering module partitions the user base into user clusters based on *each* product cluster. For example, a user might belong to an action user cluster, which agrees with the user on action movies, and is subsequently used to recommend action movies to the user. If the user has not supplied data on any action movies, the user is not assigned to a user cluster based on action, and action movies are, at least typically, not recommended to the user.

Similarly, any user who has supplied the appropriate data belongs to a comedy cluster, a drama cluster, and so on. (What constitutes "appropriate data" will be clarified in section 2.2.2, "Core Products," below.)

2.2 Ratings Space

2.2.1 The Rating Scale

Users express their tastes by assigning ratings to products. In the current implementation of the movie recommendation engine, users rate products on an integer scale ranging from 1 (low) to 6 (high).

To Be Tested. Clearly, other rating scales are possible as well. Another rating scale, with minimum rating R_{\min} and maximum rating R_{\max} , can be rescaled to a 1-6 scale (though we can no longer assume ratings are integral). While a rating scale can theoretically be discrete or continuous, we lose no generality by assuming a discrete rating scale because ratings must be truncated after a fixed number of decimal places.

If desired, we can normalize our ratings to have a particular mean or variance. If we like, we can normalize ratings for each individual user (so that a rating of 3 by a user whose average rating is 4 is interpreted differently from a rating of 3 by another user whose average rating is 2). We can force all users to have the same mean rating, or the same mean rating and rating variance.

The most general case is to assume that ratings are on a single discrete scale ranging from R_{\min} to R_{\max} . For the rest of this document, however, we will stick to ratings on a 1-6 scale, since this corresponds to what has been implemented in Production so far.

2.2.2 Core Products

As mentioned above, for each product cluster, we cluster users based on their ratings of products in that product cluster. It would be ideal to use all products, but some products may have no rating data associated with them; others may have too little data to be useful in characterizing user tastes.

For each product cluster, we cluster users based on *core products*, which are, intuitively, products in that cluster that have enough ratings. The meaning of “enough” is set by the engine administrator, as follows. Associated with the i -th product cluster is a tunable parameter called `ENG_COREimin`. This parameter is a positive integer, and a product is defined to be a core product in the i -th product cluster if it has at least `ENG_COREimin` ratings.

Thus, for each product cluster, we divide up into user clusters *the collection of those users who have provided rating data for at least one core product in the product cluster*. Other users are not assigned to a user cluster for that product cluster.

2.2.3 Geometric Representation of Users

The engine represents users as points in a geometric space called *ratings space* and uses K -means clustering to group those points into clusters. As mentioned above, this is carried out (separately) for each product cluster. The geometric representation of each user is built upon the user’s rating data for the core products in each product cluster.

Choose a product cluster, and let P_1, P_2, \dots, P_N be the core products in this product cluster. We geometrically represent a user as a point (r_1, r_2, \dots, r_N) , where r_k is the user’s rating of the k -th product P_k , if the user has rated that product. If not, we set $r_k = 0$. (This assumes that possible ratings are all greater than 0.)

Prototyped; In Development. The representation above enables grouping together users based on tendency to rate products in a product cluster similarly. Alkindi has developed a more sophisticated model, in which users are grouped together based simultaneously on tendencies to rate products similarly and also to choose or purchase the same products (to see the same movies, in the context of the movie recommendation engine). Evidently, this model requires more data for each user. For each core product, we need to know:

1. Whether the user has been asked for rating info on the product.
2. If so, whether the user has rated the product. A failure to rate a product that has been offered for rating is taken to mean that the user has not purchased the product (e.g., seen a movie).
3. The rating, if the user rated the product.

As above, we assume there are N core products in the product cluster under consideration. We represent the customer in ratings space as a vector

$$(f_1, \epsilon_1, r_1, f_2, \epsilon_2, r_2, \dots, f_N, \epsilon_N, r_N),$$

where the triple (f_k, ϵ_k, r_k) corresponding to the k -th movie in the movie cluster is determined as follows:

- $f_k = 1$ if the user has been asked to rate the k -th product, and 0 otherwise.
- $\epsilon_k = 1$ if the user has rated the k -th product, 0 otherwise.
- r_k is the rating if the user has rated the k -th product, 0 otherwise.

We refer to this model as the “two-dimensional” model of rating space, because it combines two kinds of information: ratings, plus the choice to purchase (or see) each product. (The coordinates f_k do not really correspond to a third dimension; rather, they are a flag that tells us how to interpret the data contained in ϵ_k and r_k .)

Prototyped; In Development. Alkindi has also explored assigning users to clusters based on purchase data alone. In this case, data on users is confined to the f_k and ϵ_k . ϵ_k is set to 1 or 0 depending on whether or not the user has purchased the product. f_k is set to 1 or 0 depending on whether or not the form of purchase data on the user allows a determination of whether the user has purchased the product.

2.3 Vector Means

K -means clustering is an iterative algorithm that repeatedly updates the center of mass of each user cluster depending on the current cluster composition. The center of mass of a group of users, each represented by a vector $\vec{v} = (r_1, r_2, \dots, r_N)$, is also a vector $\vec{w} = (\bar{r}_1, \bar{r}_2, \dots, \bar{r}_n)$ of N entries, with the k -th entry corresponding to the k -th core product. The k -th entry \bar{r}_k of the center of mass is the average rating for the k -th core product, computed over those users who have rated this product (i.e., over the vectors \vec{v} that have a non-zero entry for r_k).

Prototyped; In Development. If the two-dimensional model of ratings space is used, the center of mass \vec{w} of a collection $\vec{v}_1, \dots, \vec{v}_M$ of vectors is determined as follows. We write

$$\vec{v}_i = (f_{ik}, \epsilon_{ik}, r_{ik})_{k=1}^N = (f_{i1}, \epsilon_{i1}, r_{i1}, \dots, f_{iN}, \epsilon_{iN}, r_{iN}), \quad (1)$$

$$\vec{w} = (f_k, \epsilon_k, r_k)_{k=1}^N = (f_1, \epsilon_1, r_1, \dots, f_N, \epsilon_N, r_N). \quad (2)$$

The entries of the vector mean \vec{w} of the \vec{v}_i are determined as follows. f_k represents the fraction of users in the cluster who’ve had the chance to provide data for the k -th core product (for the movie recommendation engine, this corresponds to having been asked to rate that product). (In other words, $f_k = (f_{1k} + f_{2k} + \dots + f_{Mk})/M$.) ϵ_k is the fraction of the users in the cluster who’ve rated the k -th product (out of those who have been asked to rate it). Finally, r_k represents the average rating assigned to the k -th product by the users who’ve rated it.

Alternatively, in terms of the notation used above, ϵ_k can be computed as $(\sum_i \epsilon_{ik})/(\sum_i f_{ik})$ and r_k can be computed as $(\sum_i r_{ik})/(\sum_i \epsilon_{ik})$. For increased efficiency, the sum of ϵ_{ik} can be

taken over those i for which $f_{ik} \neq 0$, and the sum of r_{ik} can be taken over those i for which $\epsilon_{ik} \neq 0$.

Prototyped; In Development. If users are being clustered based on purchase data, the center of mass of a user cluster is computed as follows. First, for each user, normalize the ϵ -part of the vector representing that user. (In other words, modify the coordinates of the vector corresponding to the user by replacing ϵ_k by $\epsilon'_k = \epsilon_k/\varepsilon$, where $\varepsilon = \sqrt{\sum \epsilon_{k'}^2}$, and the sum is taken over the ϵ -coordinates of the user.) Then, for each k , average together the ϵ'_k corresponding to those users for whom we have data for the k -th core product (i.e., who have $f_k = 1$).

2.4 Distance in Ratings Space

K -means clustering aims to assign each user to that user cluster which is located closest to the user in ratings space. Implementing this requires a way to measure distance between two points in ratings space (most typically, a point representing a user and a point representing a cluster center).

The distance between two points $\vec{u} = (r_1, r_2, \dots, r_N)$ and $\vec{w} = (r'_1, r'_2, \dots, r'_N)$ is computed as follows. Let d_k represent the contribution that the k -th core product makes to the overall distance. Then we set

$$d_k = |r_k - r'_k| \quad (3)$$

if both r_k and r'_k are non-zero; $d_k = 0$ otherwise.

We combine the single-product distances into a single distance by the usual Euclidean formula. That is, the overall distance d is given by

$$d = (d_1^2 + d_2^2 + \dots + d_N^2)^{1/2}, \quad (4)$$

where the sum is over all core products in the product cluster under consideration.

Prototyped; In Development. A generalization of Euclidean distance is the L^P -distance, which depends on a tunable parameter $P \geq 1$, and is defined by

$$d = (d_1^P + d_2^P + \dots + d_N^P)^{1/P}, \quad (5)$$

where the sum is again over all core products in the product cluster under consideration.

Prototyped; In Development. The 2-dimensional model is based on a more sophisticated measure of distance, which takes into account both the r_k and ϵ_k coordinates. Again, we consider two points $\vec{u} = (f_k, \epsilon_k, r_k)_{k=1}^N$ and $\vec{w} = (f'_k, \epsilon'_k, r'_k)_{k=1}^N$ in rating space, and define d_k , the k -th core product's contribution to the overall distance, as follows:

$$d_k = |g(\epsilon_k) - g(\epsilon'_k)| + \beta \cdot \frac{2}{R} \cdot \min(h(\epsilon_k), h(\epsilon'_k)) \cdot |r_k - r'_k| \quad (6)$$

if both f_k and f'_k are non-zero, and $d_k = 0$ otherwise. Here R represents the *rating range*: if users rate movies on a scale of 1 to 6, then $R = 5$.

In prototyping, we have used the p -th roots for the rescaling functions g and h : $g(t) = h(t) = t^{1/p}$, where $p > 1$. Here p is a tunable parameter; $p = 6$ is a typical value that has been

used in prototyping. (*To Be Tested:* Strictly speaking, g and g' could be any two functions $[0, 1] \rightarrow [0, 1]$ which are increasing, have $g(0) = g'(0) = 0$ and $g(1) = g'(1) = 1$, and which are concave down.) β is another tunable parameter, which has taken on values between $\frac{1}{2}$ and 10 in prototyping. (These two parameters do not appear in the Tunable Parameters document since they have not yet been added to the production system.)

The d_k 's are combined into a single distance by the Euclidean distance formula (or, more generally, the L^P -distance formula), as above.

Prototyped; In Development. If we are clustering users based on purchase data alone, we use a so-called *cosine metric*. Let $\vec{u} = (f_k, \epsilon_k)_{k=1}^N$ be a vector representing a user's purchase data. Let $\vec{w} = (f'_k, \epsilon'_k)$ be a vector representing the center of mass of a user cluster. (Note that the user cluster could consist of users for whom we have ratings data. However, if we are assigning a user for whom we only have purchase data, we interpret the data on the members of each user cluster as only purchase data, and compute the center of mass of the cluster as described in section 2.3 for the case of purchase data.)

The distance between \vec{u} and \vec{w} is computed as $1 - \cos \theta$, where θ is the angle between \vec{u} and \vec{w} . More precisely,

$$\cos \theta = \frac{\sum \epsilon_k \cdot \epsilon'_k}{\sqrt{\sum \epsilon_k^2} \sqrt{\sum \epsilon'_k}}, \quad (7)$$

where the sums are all over k corresponding to core products in the product cluster for which $f_k > 0$. (These are the products for which we can determine whether or not the user represented by \vec{u} has purchased them.)

3 K -Means Clustering Offline

K -means clustering is a geometric algorithm that aims to minimize the sum of the distances in ratings space between each point (representing a user) and the center of mass of the user cluster it belongs to. The underlying idea is that clusters can be expressed in terms either of two quantities, each of which can be viewed as determining the other:

- The list of users that belong to each cluster;
- The center of mass of each cluster.

Evidently, the centers of mass are computed from the cluster membership (see above). In the other direction, to minimize the distances between each user and the center of mass of the cluster to which the user belongs, each user should be assigned to the cluster whose center lies closest to that user. Thus, knowing the cluster centers allows us to recover the cluster membership. The K -means algorithm alternates between updating cluster centers and updating cluster membership: we first guess where the cluster centers are, then compute the cluster membership based on the guess, recompute the centers based on the membership, then recompute the membership based on the updated centers, and so on. We stop once the effect of repeated updating becomes negligible, or once we've reached a certain maximum number of iterations.

3.1 The K -Means Clustering Routine

The basic K -means clustering routine `kmcluster(points, vmeans)` takes as inputs one set `points` of points in rating space (the points to be clustered) and another set `vmeans` of points in rating space. (Intuitively, the input `vmeans` correspond to the initial guess at the centers of mass of the clusters). It outputs “clusters,” which may be taken to mean any set of data which specifies how `points` (our set of points) is to be partitioned into clusters.

3.1.1 A Single Step: Updating `vmeans`

The routine works by iteratively updating `vmeans`. In more detail, the updating subroutine `update_vmeans(points, vmeans)` does the following. (The first line — filling in missing data — is explained in the following subsection.)

```
Fill any missing data in the vmeans from neighbors.
Loop over points.  For each point:
    Compute its distance to each of the vmeans;
    Find the closest vmean to the point;
    Assign the point to the temporary cluster associated with this vmean.
Loop over the resulting temporary clusters.  For each temporary cluster:
    Compute the center of mass of the points assigned to it;
    Update the original vmean to be this center of mass.
Output the new set of vmeans.
```

3.1.2 The Data Filling Subroutine

The data filling step consists of filling in data (in an iterative way) each time a `vmean` has no rating data for a product (which means that nobody in the corresponding cluster has supplied data for that product). It is not a problem when a user (a point to be clustered) is missing data for some product, since that product will just be ignored when we compute the distance from the user to each of the cluster centers. (See (3), (6), or the explanation of (7).) However, suppose that a user has rated a product for which one of our user clusters (say, Cluster A) has no rating data (but the others do). Evidently, we don’t have a good way to tell, based on this product, whether the user should be in Cluster A or in one of the other clusters. The distance formulas (referred to above) punt on the question by arbitrarily setting the distance to Cluster A equal to 0, in effect forcing us to assign the user to Cluster A (at least based on this particular product).

We avoid this problem by estimating rating data missing in a particular `vmean` from the corresponding data in neighboring `vmeans`. (Clustering is based on core products, so we know at least some of the `vmeans` have data for each product.) To find “neighboring” clusters which will be used to supply data when a cluster is missing some, we have to invoke the notion of distance, which is what we’re trying to make work in the first place. Consequently, an iterative algorithm is appropriate.

We begin by assuming that all `vmeans` are the same distance apart (say 1 unit, although the precise number is unimportant). In other words, we first fill in any rating data missing for a given user cluster from all the other user clusters, without regard to their location in ratings

space. Next, since all user clusters now have rating data for all products, we have no problem measuring their pairwise distances in ratings space, using one of the formulas (3), (6), or (7). (Evidently, we must use the distance formula that corresponds to the model we are using.) Next, we repeat the filling-in step, using the distances we just measured to weigh the contribution one user cluster makes to another. (For a given user cluster, nearby user clusters contribute more.) Unless the filled-in `vmeans` just computed are the same as those computed in the first step, we continue iterating: recompute the pairwise distances based on the latest filled-in `vmeans`, repeat filling step based on these updated distances, compare results of filling with results of filling from the previous step. We stop when the filled-in points are within a constant `DATA_FILL_TOL` (for every point, for every coordinate) of the filled-in points obtained in the previous iteration. Another tunable parameter `MAX_DATA_FILL_ITER` provides a guaranteed stopping point; we stop after `MAX_DATA_FILL_ITER` iterations, regardless of whether the `DATA_FILL_TOL` criterion has been met.

To complete the specification of the data filling routine, it remains to explain how to carry out the actual data-filling step (in each iteration). To begin with, if we can measure distance between cluster centers, filling in missing data in a single cluster center is straightforward. The process can be expressed as a subroutine `fill_point(point, points, weights)`, where `point` is the `vmean` we're filling in, and `points` represents the `vmeans` from which we fill in data. `weights` is a matrix, where the row `weights[i]` of which corresponds to the i -th product. `weights[i]` represents the contribution that each of `points` makes in filling in data for the i -th product; consequently, it has length $M = \text{points.size}()$. It is given by

$$\text{weights}[i] = \left(\frac{n_{i1}}{d_1^q}, \dots, \frac{n_{iM}}{d_M^q} \right), \quad (8)$$

where n_{ik} is the number of users in cluster k who rated the i -th product, and d_k is the distance between `points[k]` and `point` (which doesn't depend on the product). Here k indexes the user clusters, or `vmeans`; in other words, `points[k]` is the center of mass of cluster k . q is a tunable parameter.

The `fill_point` subroutine loops over all (core) products. For each product, it checks if `point` has rating data for that product. If not, the data is filled in as a weighted average of the corresponding rating data from `points`, where the contribution of each member of `points` is weighed by the corresponding entry of `weights`. (Aside from the weights, finding the average is done as in section 2.3, "Vector Means.") The subroutine returns `filled_point`.

Note that weighing `points` by the number of ratings they contribute means that cluster centers that have no rating data for a product automatically get ignored. One consequence is that `point` can be included among `points`. Another is that any filled in data in any of `points` is automatically ignored in filling `point`, and we only use actual rating data.

A single step of filling in data in a family of `vmeans` can be carried out by a function `fill_points(vmeans, distances)`, which applies `fill_point` iteratively, filling in each of `vmeans` in turn, using all of `vmeans` as the data source (i.e., as the `points` argument to `fill_point`) in each case. (The family `vmeans` passed to this function should always consist of unfilled cluster centers.)

`distances` is a matrix of pairwise distances between the `vmeans`. By the above, the first time we call `fill_points`, we set all the pairwise distances to be 1; in subsequent iterations,

the distances are computed from the output of the previous call to `fill_points`. (We may assume that the function outputs `filled_vmeans`.) The distance matrix is used to compute the `weights` that are passed to `fill_point` each time that subroutine is called. Explicitly, the weights are defined by (8); we look in the distance matrix to obtain the distances d_k between the element of `vmeans` currently being filled in (i.e., the one playing the role of `point` in `fill_point`) and each of the `vmeans`. (There is one exception: the `vmean` being filled in is one of the `vmeans`, and while the distance from a point to itself is 0, the corresponding weight should be 0.)

The overall data filling routine should return a set of `filled_vmeans`; these are just the `filled_vmeans` returned by the last call to `fill_points`.

Remark. The data filling routine described above, and related routines, have a separate FRS document (FRS_DataFill.doc, titled “Data Fill Manager FRS”) devoted to them. The tunable parameters referred to in this subsection are consequently indexed by the data filling document rather than by the clustering document.

One related routine is data filling for non-core products. It is explained further down in this document, in section 3.4.1.

3.1.3 *K*-Means Clustering Iterates Updating `vmeans`

The routine `kmcluster(points, vmeans)` iterates the `update_vmeans` subroutine:

Do:

`update_vmeans (points, vmeans);`

Compare new `vmeans` with (previous) `vmeans`;

Until ($|\text{new } vmeans - vmeans| < T$) or (number of iterations = `MAX_ITER`).

Ideally, the clustering routine would run until all the points (representing users) find the right cluster and settle there — that is, until a single iteration of `update_vmeans` keeps all the points in the same user clusters. In this case, the new `vmeans` are the same as those produced by the previous iteration, and further iterations also have no effect. We approximate this by stopping iterating `update_vmeans` when the change in `vmeans` after a single iteration is smaller than some tolerance. Specifically, the stopping condition is that the absolute change in every coordinate of every `vmean` be less than some constant T . (T is a tunable parameter.)

Note that the `vmeans` we compare are *raw* (that is to say, unfilled) `vmeans`, because `fill_points` is called inside the `update_vmeans` function.

Prototyped; In Development. If we use the two-dimensional model of rating space, the stopping condition is formulated in terms of several tunable parameters T_r , T_ϵ , and T_f . We ask that the absolute change in any rating coordinate for any `vmean` be less than T_r , that the absolute change in any ϵ -coordinate for any `vmean` be less than T_ϵ , and that the absolute change in any f -coordinate for any `vmean` be less than T_f .

While we expect our clustering routine to converge in almost all cases, we need to guard against aberrant cases where it doesn’t. We do this by setting a maximum number of iterations, `MAX_ITER`. This value is again a tunable parameter. We stipulate that the clustering routine `kmcluster` should stop after the `MAX_ITER`-th iteration of `update_vmeans`, regardless

of whether it has converged. In addition to returning user clusters, `kmcluster` should return the number of iterations of `update_vmeans` it took to generate the clusters.

3.2 Wrapping K -Means Clustering in an Uberloop

3.2.1 Structure of the Uberloop

The quality of the clusters returned by `kmcluster` is heavily dependent on at least two inputs:

- The initial number of clusters (i.e., the number of initial `vmeans`),
- The location of the initial `vmeans`.

It makes sense to vary the inputs, apply the `kmcluster` function to each input set, and keep the “best” clusters. One evident prerequisite for doing this is a method to measure cluster quality. For the time being, assume that we have a routine `evaluate_clusters` that takes in a set of user clusters and returns a number (call it an *evaluation value*), with higher evaluation values corresponding to better clusterings. (This routine will be specified in section 3.3, below.)

To put this more precisely, consider a *clustering problem*: given a set of products (expressed as core products in a product cluster), and a set of users, each of whom has submitted rating data for at least one of those products, partition the users into clusters. We solve this problem by running an “uberloop,” which loops over different values for the inputs to `kmcluster`. (Strictly speaking, these are inputs for computing the initial `vmeans`, which are then passed to `kmcluster` along with the `points` to be clustered. For each iteration, we compute initial `vmeans` based on the inputs, apply `kmcluster` to the `points` and `vmeans`, computing user clusters, then apply `evaluate_clusters` to the user clusters, computing an evaluation value. After all iterations are completed, the user clusters corresponding to the highest evaluation value are kept.

The uberloop may be expressed as a function `kmcluster_loop(product_cluster)`, which does the following:

```
Find core products in the product_cluster.
Find users who have rated at least one of the core products.
Express users as points (vectors of ratings of core products).
Loop over inputs into kmcluster:
    Compute initial vmeans from inputs;
    kmcluster(points, vmeans);
    evaluate_clusters(user_clusters).
Return cluster membership corresponding to the best evaluation.
```

3.2.2 Inputs for an Iteration of the Uberloop

A surprisingly effective approach to finding initial cluster centers is to choose them randomly. Namely, if we wish to partition our users into K user clusters, we choose K of the users at random to play the role of initial `vmeans`. This may appear risky, but embedding clustering

based on these `vmeans` inside an uberloop and trying out many random choices minimizes this risk. The number `UBER_ITER` of times we iterate the uberloop, with a different set of random initial `vmeans` each time, is a tunable parameter.

The number `NUM_CLUS` of initial `vmeans` is another tunable parameter that must be set by the engine administrator (it is currently not looped over). Note that the initial number of `vmeans` passed to `kmcluster` need not be the final number of clusters, because some clusters may empty out during the K -means clustering process.

Prototyped; In Development. In addition to looping over randomly chosen initial `vmeans`, we can loop over how many of them there are. This can be expressed in terms of three tunable parameters: `NUM_CLUS_LOW`, `NUM_CLUS_HIGH`, and `NUM_CLUS_STEP`, which act as parameters (initial value, final value, and step size) for a for loop. Each value for the number of initial `vmeans` is plugged into the above routine that computes the initial `vmeans` (playing the role of `NUM_CLUS`). Once the initial `vmeans` are computed, they are passed to `kmcluster`.

3.3 Evaluating Clustering Results

Here we describe the `evaluate_clusters` submodule of the clustering engine, which associates a number called the *evaluation value* with each clustering run, so that the set of clusters with the highest evaluation value can be kept.

Since the purpose of clusters is to generate product recommendations, we evaluate them with this end in mind. The idea is to evaluate a set of user clusters by using it to generate simulated recommendations, then estimating the quality of those recommendations by comparing them with existing user ratings.

In more detail, once the user clusters are formed, we generate a list of “test recommendations” associated with each one. Intuitively, for a given user cluster, this list consists of the products liked the most by members of that user cluster, according to some criterion (Details are given below). The list of test recommendations is then used to evaluate “goodness of fit.” If a user cluster is tight (i.e., cluster members have closely overlapping tastes), the best-liked products will be liked by nearly all cluster members. Thus, we seek clusterings where the test recommendations receive top ratings as much of the time as possible. We will call the ratio of top ratings to all ratings the *fraction of good test recommendations*; the `evaluate_clusters` routine will return this fraction for a set of user clusters.

3.3.1 Generating Test Recommendations for a User Cluster

The Alkindi engine generates product recommendations for each user cluster using a *scoring function*. Given a user cluster, and a product in the product cluster with respect to which the user cluster was constructed, the scoring function assigns a number to the product based on the ratings of the members of the user cluster. We then recommend the highest scoring products.

Here is an example, which is all we need to generate test recommendations. We assume that all user ratings lie on a scale from 1 to 6, but are not necessarily integers. (Non-integer ratings can arise if we renormalize ratings from another scale, e.g., 1-10, to a 1-6 scale. See section 2.2.1 for details.) Fix a user cluster and a product as above. For $i = 1, 2, \dots, 6$, we let N_i represent the number of users in the user cluster who rated the product at least i , but

less than $i + 1$. (If we use the standard 1-6 integer rating scale, this reduces to the number of users who rated the product exactly i .) Then we define the score of the product with respect to the user cluster to be

$$\text{Score} = c_1 N_1 + c_2 N_2 + \cdots + c_6 N_6. \quad (9)$$

Here the c_i are tunable parameters. (The intuition is that many high ratings should make a product score highly, and many low ratings should make a product score low, so typical values for the c_i might be $c_6 = 2$, $c_5 = 1$, $c_4 = 0$, $c_3 = -1$, $c_2 = -2$, $c_1 = -3$.) A more general treatment of scoring functions is given in the sequel to this document (Algorithm Specification II: Recommendation Manager).

To minimize calls to the database, the clustering manager only computes scores for core products, whose ratings are readily available (they have just been used in clustering). The highest scoring core products are then chosen to be test recommendations. To be precise, we introduce another tunable parameter `NUM_TEST_REC`, and choose the `NUM_TEST_REC` products in the product cluster that have scored the highest with respect to the user cluster under consideration to be that user cluster's test recommendations.

Prototyped; In Development. If the product clusters have widely differing numbers of core products, it is better to express the number of test recommendations as a fixed fraction of the number of core products in each product cluster. We introduce another tunable parameter, `FRAC_TEST_REC` (lying between 0 and 1), and choose the top scoring `FRAC_TEST_REC` fraction of the core products in the product cluster as the test recommendations.

3.3.2 Evaluating the Test Recommendations

For each user cluster with respect to a given product cluster, we total up the good recommendations and bad recommendations on the list of test recommendations, as follows. For each of the products on the list of test recommendations, we look at all the users in the user cluster who rated the product. For each user who has rated the movie above a certain threshold T_1 , we increment a counter `GOOD_RECS` by 1. For each user who has rated the movie below a certain threshold T_2 , we increment a counter `BAD_RECS` by 1. (T_1 and T_2 are tunable parameters.) We loop over all user clusters with respect to the product cluster under consideration, and over all the entire list of test recommendations for each user cluster, without resetting the counters. This way, `GOOD_RECS` and `BAD_RECS` end up counting all possible good and bad recommendations out of all the test recommendations for all the user clusters.

After counting up all the good and bad recommendations, we define a quantity `FRACTION_GOOD_RECS`, which is the ratio of `GOOD_RECS` to `GOOD_RECS + BAD_RECS`. This quantity, associated with a particular partition of users into user clusters with respect to a product cluster, is returned by the `evaluate_clusters` function.

3.4 Statistics Associated with the Clusters

When the best user clusters (i.e., those with the best evaluation value) associated with each product cluster are calculated, they are stored in the database. In addition, we compute and store a range of statistics associated with the user clusters (average ratings for products,

fraction of users who rated each product, and so on), to be used in recommending and in other engine functions. These calculations are also made offline (typically directly in the database), allowing many quantities used by the recommendation engine to be looked up rather than having to be calculated 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`.

3.4.1 Data Filling for Non-Core Products

User cluster statistics, as well as other statistics on products, typically include more products than just the core products used in clustering. For example, statistics used in recommending products should evidently be kept on all products eligible to be recommended, and statistics used in selecting products to ask users to rate should be kept on all products eligible to be so selected. (Algorithm Specifications for the Recommendation and Rating Managers give details on which products are so eligible.)

While most of these statistics are fairly straightforward and do not need to be described in detail here, one of them — average user cluster ratings — requires the machinery developed in this document. This is because predicted ratings are computed as a weighted average of average user cluster ratings. For example, to predict how a user would rate a product that lies in three product clusters, we take the user clusters with respect to those product clusters that the user belongs to, and combine their average ratings for the product. (For details, see the Algorithm Specification for the Recommendation Manager.) If a user cluster has no members who have rated a product in the associated product cluster, that rating needs to be filled in from neighboring user clusters.

In this setting, the filling-in process is non-iterative. For each product cluster, the points to fill in again correspond to the centers of the user clusters. We are now interested in user cluster ratings for all *selectable* products (see the Algorithm Specification for the Rating Manager), rather than just the core products. Indeed, ratings for the core products have already been filled in, whenever necessary, as part of the K -means clustering routine described in section 3.

Extending data-filling to non-core products essentially consists of a single application of the `fill_point` function (see section 3.1.2) to fill in each user cluster center, using the weights given by 8. The d_i (distances between the user cluster centers) are based on core products, using filled-in data whenever necessary. (These quantities are computed as part of the clustering process and stored in the database.) We loop over points (user cluster centers); for each point, we fill in missing data for non-core selectable products from the other points, using the steps followed by `fill_point` and the weights given by 8, but looping over non-core selectable products rather than core products.

3.4.2 Alkindex Statistics

Another non-trivial collection of statistics associated with the user cluster is related to the *Alkindex*. This is the system’s measure of how well it understands the user’s tastes, which is based on an estimate of how much we expect personalized recommendations to improve

on unpersonalized recommendations (where we recommend products based on their popularity among all users, rather than just those who share our particular user’s taste). These statistics are described in detail in a separate Algorithm Specification document (“Algorithm Specification IV: The Alkindex”).

4 “Fast Clustering” Individual Users in Real Time

Anytime a user, whether new or existing, has rated some new products, we know more about that user’s tastes than we did before. Since our knowledge of the user’s tastes is encapsulated in the user’s cluster membership, we always update the user’s cluster assignments in response to any new ratings (of core products).

The first time a user rates core products in a particular product cluster, we assign them in real time to one of the user clusters associated with the product cluster. This is the cluster to which the user is geometrically closest, based on the ratings space model.

If a user has already rated some core products in a product cluster, he or she has already been assigned to a user cluster associated with that product cluster (either in the last offline batch process or by an application of this section or section 4.1 since the last batch process). A user who then rates some more core products in that product cluster may be taken out of his or her current user cluster and assigned to a new one. This occurs if the user is closer to the new cluster than to the current one based on the updated ratings data.

4.1 New Users

Suppose that a new user has just submitted their first set of ratings. We begin by deciding which product clusters we can now cluster the user with respect to. Evidently, these are the product clusters that contain core products for which the user has just submitted ratings. Loop over these product clusters. For each one, we represent the user as a point in ratings space as in section 2.2.3.

We also call on the database to provide the centers of mass of the user clusters with respect to the product cluster under consideration. Naively, these are averages of the ratings data of the members of each user cluster as in section 2.3. However, since some user clusters may be missing data for some products, we should use **vmeans** that have had any missing data filled in by the **fill_points** function described in section 3.1.2. The Alkindi engine writes filled-in data associated with the user cluster centers into the database as part of the offline compilation of user cluster statistics, so all we need to do is look up the filled-in **vmeans** in the database.

Once we have points in ratings space representing the user and each of the **vmeans**, we simply find the **vmean** to which the user is closest in ratings space, based on the distance measure described in section 2.4, and assign the user to the user cluster having that **vmean**.

4.2 Existing Users

When an existing user submits a set of ratings, we proceed similarly, identifying the product clusters for which new ratings for core products have just been submitted, then seeing if we

need to assign the user to new user clusters based on those product clusters. There are just a few extra considerations:

1. A user may have already rated some core products in one of the product clusters for which new ratings have been submitted. If this is so, the user's previous ratings in the product cluster need to be obtained from the database and combined with the just-submitted ratings to form the point in ratings space that represents the user.
2. If a user has already rated core products in a product cluster, that user should belong to a user cluster with respect to the product cluster. The purpose of the calculation being described is to decide whether the user should stay in that user cluster or move to a new one, based on the updated data just submitted. If we move the user to a different user cluster, we must also remove them from the one to which they previously belonged.

4.3 Cluster Statistics and Online Clustering

User cluster statistics are *not* updated online. For example, suppose that a user cluster had 24 members after the offline clustering process, and a 25th member — a new user who had just entered the system — was later added online. We would write to our database that the new user now belongs to this cluster. However, the database entry that records the number of users in that user cluster (which was computed as part of a general computation of statistics after the offline clustering process) remains unchanged at 24. (The reason for this is that there are so many statistics associated with the user clusters that recalculating all of them frequently is impractical.) Similarly, if a user were moved out of a user cluster with 34 members and into another user cluster, the user's user cluster membership data would be updated, but the user cluster statistics again wouldn't be, so the cluster would still be listed in the database as having 34 members, even though one was just removed.

The effect of this can be conceptually summed up as follows. The Alkindi system is based on the idea that similar users recommend products to each other. When we assign a user to a user cluster without updating that cluster's product statistics, we are saying that we'll use the cluster to recommend products to that user, without having the user contribute to the way the cluster recommends (to its other members). After the next time that the entire set of users is reclustered, the statistics associated with the clusters are calculated anew, and each user contributes to the way that each user cluster to which they belong recommends.