"Strength Lies in Differences" – Diversifying Friends for Recommendations through Subspace Clustering

Eirini Ntoutsi
LMU, Munich
ntoutsi@dbs.ifi.lmu.de

Kostas Stefanidis
ICS-FORTH, Heraklion
kstef@ics.forth.gr

Katharina Rausch
LMU, Munich
rausch.katharina@gmail.com

Hans-Peter Kriegel
LMU, Munich
kriegel@dbs.ifi.lmu.de

ABSTRACT

Nowadays, WWW brings overwhelming variety of choices to consumers. Recommendation systems facilitate the selection by issuing recommendations to them. Recommendations for users, or groups, are determined by considering users similar to the users in question. Scanning the whole database for locating similar users, though, is expensive. Existing approaches build cluster models by employing full-dimensional clustering to find sets of similar users. As the datasets we deal with are high-dimensional and incomplete, full-dimensional clustering is not the best option. To this end, we explore the fault-tolerant subspace clustering approach. We extend the concept of fault tolerance to density-based subspace clustering, and to speed up our algorithms, we introduce the significance threshold for considering only promising dimensions for subspace extension. Moreover, as we potentially receive a multitude of users from subspace clustering, we propose a weighted ranking approach to refine the set of like-minded users. Our experiments on real movie datasets show that the diversification of the similar users that the subspace clustering approaches offer results in better recommendations compared to traditional collaborative filtering and full-dimensional clustering approaches.

1. INTRODUCTION

With the growing complexity of WWW, users often find themselves overwhelmed by the mass of choices available. Shopping for DVDs, books or clothes online becomes more and more difficult, as the variety of offers increases rapidly and gets unmanageable. To facilitate users in their selection process, recommendation systems provide suggestions on items, which might be interesting for the respective user. In particular, recommendation systems aim at giving recommendations to users or groups of users by estimating their item preferences and recommending those items featuring the maximal predicted preference. The prerequisite for determining such recommendations is historical information on the users' interests, e.g., the users' purchase history.

CIKM'14, November 3–7, 2014, Shanghai, China.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2598-1/14/11 ...$15.00.
http://dx.doi.org/10.1145/2661829.2662026.

Typically, user recommendations are established by considering users sharing similar preferences as the query user. Scanning the whole database to find such like-minded users, though, is a costly process. More efficient approaches build user models for computing recommendations. For example, [14] applies full-dimensional clustering to organize users into clusters and employs these clusters, instead of a linear scan of the database, for predictions. Full-dimensional clustering is not the best option for the recommendation domain due to the high dimensionality of the data. Typically, there exist hundreds to thousands or millions of items in a recommendation application. Feature reduction techniques, like PCA, tackle the high dimensionality problem by reducing the initial high dimensional feature space into a smaller one, and working upon the reduced feature space. Such a global reduction is not appropriate for cases where different dimensions are relevant for different clusters; for example, there might be a group of comedy fans, part of which might belong to another group of drama fans, but there might be no group of both comedy and drama fans. Due to high dimensionality, such cases are actually more common than finding, e.g., users similar with respect to the whole feature space.

To deal with the high dimensionality aspect of recommendations, we employ *subspace clustering* [11], that extracts both clusters of users and dimensions, i.e., items, based on which users are grouped together. As users possibly belong to more than one subspace cluster (each cluster defined upon different items), this approach broadens our options for selecting like-minded users for a query user. Employing in the recommendation process users that differ qualitatively in terms of the items upon which their selection was made, makes the set of like-minded users more diverse. Traditional subspace clustering methods [11] cannot be applied in our settings as our data are characterized by sparsity and incompleteness (users rate only few items, resulting in a lot of missing values). To deal with these issues, recently, the so called *fault tolerant subspace clustering* has been proposed [7]. We extend the original grid-based fault tolerant subspace clustering algorithm and introduce two *density-based fault tolerant* approaches that, as we will show, perform better in terms of both quality and efficiency. To speed up the algorithms, we introduce the *significance threshold*, a heuristic for finding the most significant dimensions for extending subspace clusters instead of considering all dimensions. Subspace clustering might result in overlapping clusters; we propose a *weighted ranking* approach to combine these results and select the most prominent users for recommendations.

In brief, our contributions are as follows:

- We explore *subspace clustering* for recommendations and introduce two *density-based fault tolerant subspace clustering* approaches.
- We propose the *significance threshold* as a criterion to prune non-promising for subspace extension dimensions.
- We refine via *weighted ranking*, the selection of like-minded users that result from the combination of subspace clusters that a user belongs to.
- We experimentally show that employing fault tolerant subspace clustering with the weighted ranking of like-minded users results in predictions of higher quality, while the proposed significance threshold reduces the runtime of our algorithms.

The rest of the paper is organized as follows. Section 2 presents the basics of recommendations and the limitations of existing approaches. Section 3 overviews fault tolerant subspace clustering, while Section 4 extends fault tolerant subspace clustering from grid-based to density-based, and introduces the significance threshold for reducing the search space. Section 5 describes the weighted ranking approach for exploiting the subspace clustering results for recommendations. Section 6 presents our experimental results. Related work is in Section 7, and conclusions are given in Section 8.

2. BACKGROUND AND LIMITATIONS

2.1 Background on Recommendations

Assume a recommendation system, where I is the set of items to be rated and U is the set of users in the system. A user $u \in U$ might rate an item $i \in I$ with a score $rating(u, i)$ in $[0.0, 1.0]$; let R be the set of all ratings recorded in the system. Typically, the cardinality of the item set I is high and users rate only a few items. The subset of users that rated an item $i \in I$ is denoted by $U(i)$, whereas the subset of items rated by a user $u \in U$ is denoted by $I(u)$.

For the items unrated by the users, recommendation systems estimate a relevance score, denoted as $relevance(u, i)$, $u \in U$, $i \in I$. There are different ways to estimate the relevance score of an item for a user. In the content-based approach (e.g., [13]), the estimation of the rating of an item is based on the ratings that the user has assigned to similar items, whereas in collaborative filtering systems (e.g., [9]), this rating is predicted using previous ratings of the item by similar users. In this work, we follow the collaborative filtering approach. Similar users are located via a *similarity function* $simU(u, u')$ that evaluates the proximity between $u, u' \in U$ by considering their shared dimensions. We use F_u to denote the set of the most similar users to u, hereafter, referred to as the *friends* of u.

Definition 1. *Let $\mathcal{U}$ be a set of users. The friends $\mathcal{F}_u$, of a user $u \in \mathcal{U}$ consists of all those users $u' \in \mathcal{U}$ which are similar to u w.r.t. a similarity function $simU(u, u')$ and a threshold δ, i.e., $\mathcal{F}_u = \{u' \in \mathcal{U} : simU(u, u') \geq \delta\}$.*

Given a user u and his friends $\mathcal{F}_u$, if u has expressed no preference for an item i, the relevance of i for u is estimated as:

$$relevance_{\mathcal{F}_u}(u, i) = \frac{\sum_{u' \in \mathcal{F}_u} simU(u, u') rating(u', i)}{\sum_{u' \in \mathcal{F}_u} simU(u, u')}$$

After estimating the relevance scores of all unrated user items, the top-k rated items are recommended to the user.

Most previous works focus on recommending items to individual users. Recently, *group recommendations* that make recommendations to groups of users instead of single users (e.g., [3, 14]), have received considerable attention. Our goal is to test our methods for both user and group recommendations. Here, for group recommendations, we follow the approach of [14]: first, estimate the relevance scores of the unrated items for each user in the group, then, aggregate these predictions to compute the suggestions for the group.

Definition 2. *Let $\mathcal{U}$ be a set of users and $\mathcal{I}$ be a set of items. Given a group of users $\mathcal{G}$, $\mathcal{G} \subseteq \mathcal{U}$, the group relevance of an item $i \in \mathcal{I}$ for $\mathcal{G}$, such that, $\forall u \in \mathcal{G}$, $\nexists rating(u, i)$, is:*

$$relevance(\mathcal{G}, i) = Aggr_{u \in \mathcal{G}}(relevance_{\mathcal{F}_u}(u, i))$$

As in [14], we employ three different designs regarding the aggregation method $Aggr$: (i) the *least misery design*, capturing cases where strong user preferences act as a veto (e.g., do not recommend steakhouses to a group when a member is vegetarian), (ii) the *fair design*, capturing more democratic cases where the majority of the group members is satisfied, and (iii) the *most optimistic design*, capturing cases where the more satisfied member of the group acts as the most influential one (e.g., recommend a movie to a group when a member is highly interested in it and the rest have reasonable satisfaction). In the least misery (resp., most optimistic) design, the predicted relevance score of an item for the group is equal to the minimum (resp., maximum) relevance score of the item scores of the members of the group, while the fair design, that assumes equal importance among all group members, returns the average score.

2.2 Limitations of Existing Approaches

One of the key issues in collaborative filtering approaches is the identification of the friends set $\mathcal{F}_u$ of a user $u \in U$. Below we discuss two approaches towards this direction: (i) the *naive approach* that scans the whole database of users to select the most similar ones and the (ii) *full dimensional clustering approach* that partitions users into clusters and employs cluster members for recommendations.

2.2.1 Naive Approach

A straightforward approach for finding the set $\mathcal{F}_u$ for a user $u \in \mathcal{U}$, is to compute $simU(u, u')$, $\forall u' \in \mathcal{U}$, and select those with $simU(u, u') \geq \delta$, where δ is the similarity threshold. Such an approach though would be inefficient in large systems, since it requires the online computation of the set of friends for each query user. The problem is aggravated in case of group recommendations, where the sequential scan of the database should be performed for each user in the query group. The execution time increases linearly with the number of group members; the larger the query group, the slower the approach.

2.2.2 Full-dimensional Clustering Approach

One way to overcome the limitations of the naive approach, is to build some users model and directly employ this model for recommendations. Full-dimensional clustering has been used towards this direction to organize users into clusters of similar ones. The pre-computed clusters are then employed to speed up the recommendation process; the friends of a given user u correspond approximately to the users that belong to the same cluster as u.

In [14], a bottom-up hierarchical clustering algorithm has been employed to build the users model. The similarity between two clusters is defined in terms of the complete linkage, i.e., as the minimum similarity between any two users of these clusters. The algorithm terminates when the similarity of the closest pair of clusters violates the user similarity threshold δ. Thus, the resulted clusters fulfill the similarity criterion in Definition 1, i.e., all users within a cluster have a similarity of at least δ. However, the set of friends might be incomplete, i.e., for a user u belonging to a cluster C there might be users with whom u has a similarity of at least δ but they do not belong to C. The reason is in the clustering process: at each step the two most similar clusters are merged, resulting in a larger cluster. The order of merging plays an important role on the final clusters setup. Although all members of a cluster will have a similarity of at least δ, there might be users with similarity greater or equal to δ being assigned to different clusters.

The recommendation time in this case is reduced, since the clusters are pre-computed and the set of friends is easily deliverable; the decrease is rapid for group recommendations, especially as the group size increases. Concerning quality, the naive approach outperforms the full clustering approach, as it can locate the extensive set of friends for each user whereas in full clustering, the set of friends for a user is its corresponding cluster members. The smaller the cluster is, the more restricted the set of friends for a user within the cluster is and therefore, the lower the quality of recommendations for this user.

Both naive and full dimensional clustering approaches "suffer" from the so called curse of dimensionality since both operate in the original high dimensional item space. In high dimensional spaces, the discriminative power of the distance functions lowers and moreover, it is more difficult to find similar users in the whole (high dimensional) feature space, whereas it is easier to locate such users in a subspace of dimensions. To deal with these issues, subspace clustering [11] tries to detect both the objects (users in our case) that belong to a cluster and the dimensions (items in our case) that define this cluster. Therefore, a user might belong to more than one subspace clusters, each defined upon a different subset of items. Employing subspace clustering for recommendations serves a three-fold purpose: (i) improves the clustering quality by providing a better partitioning of the users based on different subsets of items, (ii) expands the set of friends for a user by allowing users to belong to several clusters, and (iii) diversifies the set of friends as different friends might be chosen based on different items.

3. SUBSPACE CLUSTERING AND FAULT-TOLERANT SUBSPACE CLUSTERING

Subspace clustering approaches aim at detecting clusters embedded in subspaces of a high-dimensional dataset [11]. Clusters may be comprised of different combinations of dimensions, while the number of relevant dimensions may vary strongly. To restrict the search space, only axis-parallel subspaces are searched through for clusters. A *subspace S* describes a subset of items, $S \subseteq I$; $|S|$ is the subspace cardinality. A subspace cluster C is then described in terms of both its members $U \subseteq \mathcal{U}$ and subspace of dimensions $S \subseteq I$ upon which it is defined as $C = (U, S)$.

The vast majority of subspace clustering algorithms works on complete datasets. However, our data is characterized by many missing values, since users rate only a few items. Recently, *fault tolerant subspace clustering* [7] has been proposed to handle sparse datasets. The main idea of this approach is that clusters including missing values can still be valid, as long as the amount of missing values does not have a negative influence on the cluster's grade of distinction.

To restrict the number of missing values in a subspace cluster, thresholds w.r.t. the number of missing items, the number of missing users and their combination, are used. These thresholds are adapted from the original work [7] to the recommendations domain. Users featuring a missing value for item $i \in I$ are included in $U_?(i) = \{u \in U | rating(u,i) =?\}$, whereas $I_?(u) = \{i \in I | rating(u,i) =?\}$ holds those items having a missing value for user $u \in U$.

User Tolerance: Each user in a subspace cluster must not contain more than a specific number of missing item ratings. That is, $\forall u \in U : |I \cap I_?(u)| \leq \epsilon_u \cdot |I|$, where $\epsilon_u \in [0,1]$ is the user tolerance threshold.

Item Tolerance: Each item in a subspace cluster should not contain too many missing values. That is, $\forall i \in I : |U \cap U_?(i)| \leq \epsilon_i \cdot |U|$, where $\epsilon_i \in [0,1]$ is the item tolerance threshold.

Pattern Tolerance: The total number of missing values in a subspace cluster must not exceed the *pattern tolerance* threshold $\epsilon_g \in [0,1]$. That is, $\sum_{u \in U} |S \cap I_?(u)| \leq \epsilon_g \cdot |U| \cdot |S|$.

Thus, a cluster $C = (U, S)$ is a *valid fault tolerant subspace cluster* if the number of missing items per user does not violate ϵ_u, the number of missing users per item does not violates ϵ_i and the total number of missing values is bounded w.r.t. ϵ_g.

Bottom-up subspace clustering approaches make use of the *monotonicity property*: if $C = (U, S)$ is a subspace cluster, in each subset S', $S' \subseteq S$, there exists a superset of users, so that, this set is a subspace cluster as well. The fault tolerance model defined so far, does not follow the monotonicity property. [7] suggests *enclosing cluster approximations*, which form supersets of the actual clusters, i.e., they include more users than the actual subspace clusters do. These approximations follow the monotonicity property. A subspace cluster can be extended by adding some dimensions up to a dimensionality of value mx. Thus, each fault tolerant subspace cluster $C = (U, S)$, with $|S| \leq mx$, is *mx-approximated* by a maximal fault tolerant subspace cluster $A = (U_A, S)$, established by the thresholds $\epsilon_u = min\{\epsilon_u \cdot \frac{mx}{|S|}, 1\}$ and $\epsilon_i = \epsilon_g = 1$. The rationale is to extend the subspace clusters by some dimensions, in order to create enclosing approximations, which fulfill these thresholds.

4. SUBSPACE CLUSTERING FOR USER RECOMMENDATIONS COMPUTATION

4.1 Grid-based Fault Tolerant Subspace Clustering (gridFTSC)

[7] introduces the *grid-based fault tolerant subspace clustering* algorithm *FTSC* by integrating the fault tolerance concepts to the grid-based subspace clustering algorithm CLIQUE [2]. As in CLIQUE, the data space is partitioned into non-overlapping rectangular grid cells by partitioning each item into g equal-length intervals and intersecting the intervals. Clusters consists of dense cells containing more than a density threshold *minPts* points.

In *FTSC*, users with missing values/ratings might also be part of the clusters. To this end, an extra interval is allocated for each item, where users with missing values for the respective item are placed in. To generate cluster approximations, except for the item intervals with existing values, item intervals for missing values are also considered. Thus, a cluster approximation consists of the users in the respective cluster cells plus the users obtained by considering the intersection with the missing values intervals.

For an efficient generation of cluster approximations, a set of users U_A is partitioned according to the amount of missing values per user, i.e., $(U_A, S) = ([U_A^0, U_A^1, \cdots, U_A^x], S)$, where U_A^i consists of the users with exactly i missing values. To avoid analyzing all possible subspaces, the monotonicity of approximations and the fault tolerance thresholds are exploited (Algorithm 1). To generate the actual clusters from the approximations, the approximation list of users is traversed and users are added to the cluster. Users with no missing values (at the top of the list) are first added to the cluster, whereas those with missing values are gradually added if they do not violate the fault tolerance thresholds.

Algorithm 1 Candidate Approximations (gridFTSC) [7]

```
1: function GETCANDIDATE($(U_A, S), d, I, I_?$)
2:     if ($firstRun$) then
3:         $S' \leftarrow d$
4:         $U_B^0 \leftarrow I$
5:         $U_B^1 \leftarrow I_?$
6:     else
7:         $S' \leftarrow S \cup \{d\}$
8:         for ($i$ from 0 to $size(U_A)$) do
9:             $existingList^i \leftarrow I \cap U_A^i$
10:            $missingList^i \leftarrow I_? \cap U_A^i$
11:        end for
12:        $U_B^0 \leftarrow existingList^0$
13:        $n \leftarrow size(existingList)$
14:        for ($i$ from 1 to $(n-1)$) do
15:            $U_B^i \leftarrow existingList^i \cup missingList^{i-1}$
16:        end for
17:        $U_B^n \leftarrow missingList^{n-1}$
18:    end if
19:    return $(U_B, S')$
20: end function
```

Figure 1 (left) shows an example with the 1-item intervals for item/dimension 1. There are 4 dense intervals/cells (red), whereas the users with missing values are allocated to their own interval (blue)[1]. The candidate approximation is a list: users with no missing values are stored first (marked with (a)) followed by users with 1 missing value (marked with (b)). Figure 1 (right) shows the result when extending the candidate approximation based on item 2. For illustrative purposes, we choose the interval [0.75,1] for item 2. The extension of the candidate approximation list would be as follows: users with values in both items 1 and 2 for this range (marked with (1)), are assigned to the first position of the candidate approximation list. Users with missing values in either item 1 or 2 (marked with (2) or (3), resp.), are stored in the second position of the list. The last position of the list stores the interval marked with (4) for users with missing values in both items 1 and 2. This part can be directly pruned, since it contains only missing values.

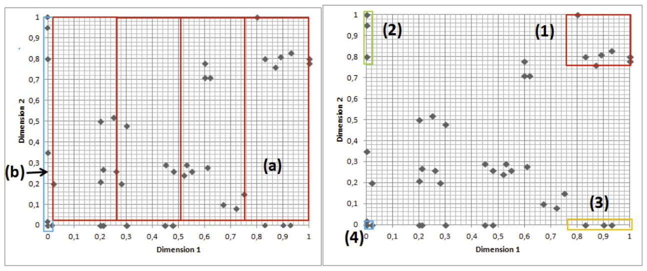


Figure 1: Grid-based cluster approximations ($g = 4$, $minPts = 3$).

4.2 Density-based Fault Tolerant Subspace Clustering

The quality of grid-based clustering heavily depends on the positioning of the grid in the data space. Density-based approaches are more flexible in detecting arbitrarily shaped clusters. Interestingly, we can adapt the candidate approximations construction to density-based clusters, instead of grid-cells. The difficulty that emerges when transferring the fault tolerance concept to density-based subspace clustering is that density-based approaches use a distance function and so, we need to evaluate distances between users, even though they might contain missing values.

We propose two approaches for building candidate approximations for density-based fault tolerant subspace clustering: (i) a hybrid approach, called hybridFTSC, that combines grid- and density-based ideas and (ii) a pure density-based approach inspired by the subspace clustering algorithm SUBCLU [8], called denFTSC.

4.2.1 *Hybrid Fault Tolerant Subspace Clustering (hybridFTSC)*

Instead of splitting the data space into intervals, as in $gridFTSC$, the hybrid approach determines 1-item density-based clusters for each item/dimension of the dataset. For the 1D clustering, we employ DBSCAN [6]. However, as DBSCAN cannot handle missing values, for each item, the users with missing values on it are "isolated" into a so called *pseudo cluster* and DBSCAN is applied on the remaining users. Thus, for each item, we get one or several density-based clusters with users featuring no missing values and a pseudo-cluster with users having no ratings for the specific item. As in gridFTSC, we extend each of those 1-item candidate approximations with additional items to receive broader subspaces, c.f., *getCandidate* method (Algorithm 2).

We aim at generating candidate approximation lists sorted according to the users' numbers of missing values. Thus, in the first run of *getCandidate*, we assign each 1-item density-based cluster to the first position of the candidate approximation list (line 4). In the second position, we save the corresponding pseudo cluster with users featuring missing values for the respective item (line 5). To generate an n-item candidate approximation, we combine all the users from the candidate approximation list, generated in the $(n-1)$th run, to get a set of users for clustering (line 9-10). When examining this set of users, we filter out users with a missing value in item d and assign them to a pseudo cluster (line 13). Afterwards, we call an 1-item DBSCAN (considering item d only) on the remaining users (line 16). This DBSCAN-call may result in one or several density-based clusters, which represent new candidates for extension. We discard the noise points and generate a new candidate approximation for each of the clusters. To do this, each resulting cluster is combined with all users from the pseudo cluster (line 18) and sorted ascendingly according to the users' numbers of missing values in the current subspace to generate the new candidate approximation list (line 19-20). The algorithm continues as the grid-based one. Generally, our focus is on creating density-based grid-cells by executing 1-item DBSCAN-runs in order to extend the subspaces to a higher dimensionality. Since the positioning of these grid-cells is flexible and not static, we are able to find broader and tighter grid-cells.

Figure 2 (left) displays the 1-item density-based clusters for item 1. We consider the candidate approximation, which includes the density-based cluster marked by (a), as well as the pseudo cluster, which contains the users with missing values for item 1 (marked by (b)). To extend this candidate approximation by item 2, i.e., to the subspace spanned by items 1 and 2, we combine the users of both clusters into one set. Afterwards, we filter out the users with missing values for item 2 (marked by (2) and (3) in the right part). We call DBSCAN on the remaining users and obtain two clusters,

[1]For visualization reasons, we assume that missing values are represented by 0 in the respective item and the lower bound for ratings is greater than 0.

Algorithm 2 Candidate Approximations (hybridFTSC)

```
 1: require density-based 1-item cluster $C$ from item $d$, pseudo cluster $C_?$ including missing values for item $d$
 2: function GETCANDIDATE($(U_A, S), d, C, C_?$)
 3:     if ($firstRun$) then
 4:         $S' \leftarrow d$
 5:         $U_B^0 \leftarrow C$
 6:         $U_B^1 \leftarrow C_?$
 7:         add $(U_B, S')$ to $candidateApproximations$
 8:     else
 9:         $S' \leftarrow S \cup \{d\}$
10:         for $i$ from 0 to $size(U_A)$ do
11:             add $U_A^i$ to $allUsers$
12:         end for
13:         $missingList \leftarrow filterMissing(allUsers, d)$
14:         $existingList \leftarrow filterExisting(allUsers, d)$
15:         // dist(): distance function based only on item d
16:         $clusters \leftarrow DBSCAN(existingList, dist, d)$
17:         for all cluster in clusters do
18:             $users \leftarrow cluster \cup missingList$
19:             $U_B \leftarrow sortByMissingValues(users, S')$
20:             add $(U_B, S')$ to $candidateApproximations$
21:         end for
22:     end if
23:     return $candidateApproximations$
24: end function
```

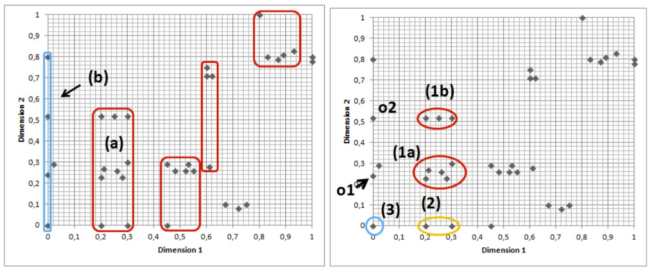


Figure 2: Hybrid-based cluster approximations ($\epsilon = 0.04$, $minPts = 3$).

marked as (1a), (1b). User $o1$ is included in cluster (1a), as we consider just the second item. This is because it belongs to the ϵ-neighborhood of some of the cluster users of (1a) in item 1. User $o2$ belongs to cluster (1b). We create two new candidate approximations by combining each of the clusters with both (2) and (3). For each new candidate, we create a list, which includes users ordered according to their number of missing values within the current subspace. For example, the candidate approximation list based on (1a) holds (1a) at its first position, as it does not contain any missing values, (2) and user $o1$ at its second position, as they contain one missing value per user, and (3) at its last position with only missing values in the current subspace.

4.2.2 *Density-based Fault Tolerant Subspace Clustering (denFTSC)*

SUBCLU [8] is a density-based bottom-up clustering approach, which is based on DBSCAN [6]. In SUBCLU, the notions of neighborhood, reachability, connectivity and cluster from DBSCAN are related to a specific subspace. The DBSCAN parameters ϵ (for defining the neighborhood of a point) and $minPts$ (for deciding on core points) are inherited. SUBCLU starts in 1-item subspaces and applies DBSCAN to every subspace to generate 1-item clusters. Next, it checks, for each cluster, in a bottom-up way, whether the cluster or part of it still exists in higher-item subspaces. SUBCLU considers each k-item candidate subspace and selects those of the remaining k-item subspaces, which share (k - 1) attributes with the former. The algorithm joins them in order to determine (k + 1)-item candidate subspaces.

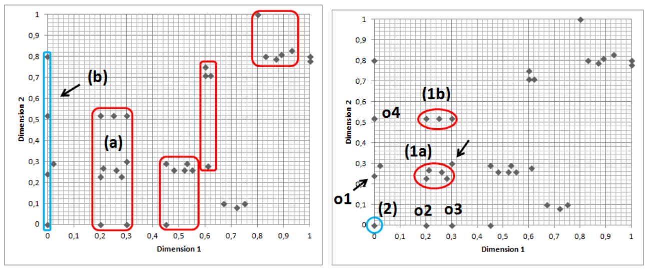


Figure 3: SUBCLU-based cluster approximations ($\epsilon = 0.04$, $minPts = 3$).

The SUBCLU-based approach aims at determining density-based candidate approximations by focusing on the complete current subspace. Our goal is to transfer the SUBCLU clustering paradigm to fault tolerant clustering. Therefore, for computing distances for the candidate approximations, we employ a function, which ignores items with missing values.

As in the hybrid approach, we generate 1-item density-based clusters by applying DBSCAN for each item. The users with missing values for the respective item are again assigned to a pseudo cluster. In the first run of *getCandidate* (Algorithm 3), every 1-item cluster is saved in the first position of the candidate approximation list (line 4), whereas the pseudo cluster for the item is assigned to the second position (line 5). For the nth run of *getCandidate*, we merge all users in the candidate approximation list of the $(n-1)$th run (line 9-10). They pose the basis for the DBSCAN-call with respect to the current candidate's subspace (line 15). For calculating distances, we use a function that considers the objects in the current subspace and ignores items with missing values. Again, we receive one or several clusters and discard the noise points. We then create a candidate approximation list for each of the resulting clusters by sorting the clustered users according to their numbers of missing values in the current subspace (line 17). The rest of the algorithm's processing is similar to the grid-based approach.

Figure 3 (left) depicts the state after the call of an 1-item DBSCAN on item 1. The algorithm retrieves 4 clusters and a pseudo cluster. To extend the candidate approximation consisting of (a) and (b) to the two-item subspace (items 1 and 2), we merge (a) and (b) at the second run of *getCandidate*. The call of DBSCAN on the 2-item subspace results in two new candidate approximations: (i) the combination of (1a), o_1, o_2, o_3 and (2), and (ii) the combination of (1b), o_2, o_3, o_4 and (2). Users o_2 and o_3 are assigned to both approximations, as we ignore items not exist in both users' feature vectors. Users included in (2) feature missing values in both items 1 and 2 and therefore, it has not been determined yet, whether they belong to one of the clusters, and if so, to which one. So, they are assigned to both approximations.

Again, the candidate approximation list is generated by building sets of users sorted according to their number of missing values in the current subspace.

In contrast to the hybrid approach, the SUBCLU-based approach does not assign the user right from cluster (1a) (marked with an arrow) to cluster (1a), because it considers the 2-item distance, which exceeds the value of ϵ. As the hybrid approach, however, considers just the 1-item distance (based on item 2), the user is assigned to cluster (1a), as he features a distance below ϵ to nearest cluster object.

4.2.3 *Reducing the search space through the significance threshold*

The runtime of the algorithms, highly depends on the dimensionality of the datasets since we are looking for clusters in subspaces of the original high dimensional feature space.

Algorithm 3 Candidate Approximations (denFTSC)

```
1: require density-based 1-item cluster $C$ from item $d$, pseudo cluster $C_?$ including missing values in item $d$
2: function GETCANDIDATE($(U_A, S), d, C, C_?$)
3:     if ($firstRun$) then
4:         $S' \leftarrow d$
5:         $U_B^0 \leftarrow C$
6:         $U_B^1 \leftarrow C_?$
7:         add $(U_B, S')$ to $candidateApproximations$
8:     else
9:         $S' \leftarrow S \cup \{d\}$
10:        for $i$ from 0 to $size(U_A)$ do
11:            add $U_A^i$ to $allUsers$
12:        end for
13:        // dist is a distance function ignoring missing values
14:        // dist is based on the current subspace S'
15:        $clusters \leftarrow DBSCAN(allUsers, dist, S')$
16:        for all cluster in clusters do
17:            $U_B \leftarrow sortByMissingValues(cluster, S')$
18:            add $(U_B, S')$ to $candidateApproximations$
19:        end for
20:    end if
21:    return $candidateApproximations$
22: end function
```

To reduce the runtime, we exploit the fact that we are only interested in extending our candidate subspaces and looking for broader clusters, which are highly distinctive and contain significant information. For example, if most users rated in the same way a particular item (resulting in one big user cluster and noisy users for this item), the item does not need to be considered for extension. Following this rationale, for subspace extension, we consider only those items which contain at least *clusterThreshold* 1-item clusters featuring at least *dataThreshold %* of the overall number of users in the dataset. The value for *clusterThreshold* should be larger than 1 (i.e., an item should be part of at least two 1-item clusters) and approximately half of the amount of possible ratings (in order to express significance relatively to the characteristics of the dataset). The *dataThreshold* expresses how much of the population these clusters should cover.

5. WEIGHTED RANKING FOR LOCATING SIGNIFICANT FRIENDS

Through subspace clustering, we possibly receive many like-minded users for a user u, as u might be a member of several subspace clusters. Combining all users from the clusters u belongs to is advantageous, as we gain an extensive and diverse selection of like-minded people for u, since it is based on different subsets of items. Thus, we are able to reflect on different characteristics u might feature to calculate the most promising recommendations for him/her.

To enhance the quality of recommendations, we refine the set of like-minded users based on their common ratings to u. In particular, we order these users according to their *full-dimensional distance* (based on their common dimensions) to u. Using full-dimensional distance instead of subspace distance for user ranking, allows us to capture the overall similarity of preferences between users. By employing a subspace distance function during clustering, we are able to detect like-minded users, which might share just part of their interests with u. When ranking the like-minded users, however, we are interested in finding the most promising ones, i.e., those that also agree or at least do not disagree too strongly with u in the rest of their commonly rated items.

A distance based on a higher number of common items should be more significant than a distance based on considerably less or just one item, therefore we weight the distances between u and his/her friends based on the number of their common items. Formally, the *weighted distance* between u and his/her friend $v \in U$ is given by:

$$dist_{weighted}(u,v) = \frac{1}{c_{u,v}} \sqrt{\sum_{i=1, i \in I_{uv}}^{n} (u_i - v_i)^2}$$

I_{uv} is the set of common items between u and v and $c_{u,v}$ is their normalized share items. To compute $c_{u,v}$, min-max normalization is employed:

$$\forall u, v \in U : c_{u,v} = \frac{it_{u,v} - it_{min}}{it_{max} - it_{min}}$$

where $it_{u,v}$ is the number of common ratings between u and v and it_{min} (it_{max}) is the minimum (maximum) number of common ratings between u and the like-minded users of u.

Finally, the weighted top friends (shortly, friends) used for recommendations are those featuring a distance to u, which is below a *weighted distance threshold* β. More formally:

DEFINITION 3. *Let $\mathcal{U}$ be a set of users and $\Theta = \{\theta_1, \ldots, \theta_\kappa\}$ the subspace clustering model upon $\mathcal{U}$, such that $\Theta = \cup\theta_i$. The friends $\mathcal{F}_u$ of a user $u \in \mathcal{U}$ are the users $u' \in \mathcal{U}$ that are members of the same clusters θ_i as u and their weighted distance is below the weighted distance threshold β, i.e., $\mathcal{F}_u = \{u' \in U | \exists \theta_i \in \Theta : u, u' \in \theta_i, dist_{weighted}(u, u') \leq \beta\}$.*

6. EXPERIMENTAL EVALUATION

We evaluate the efficiency and quality of the (i) *naive*, (ii) *fullClu*, (iii) *gridFTSC*, (iv) *hybridFTSC*, and (v) *denFTSC* approaches using two MovieLens datasets[2]. The *ML-100K* dataset contains 100,000 ratings given by 983 users for 1,682 movie items. The *ML-1M* dataset includes 1,000,000 ratings of 6,040 users over 3,952 movie items. For *efficiency*, we study how the runtime of the algorithms is affected by different parameters. For *quality*, we use accuracy measures that directly compare the predicted user ratings with the actual ones. The *Mean Absolute Error* (MAE) signifies the average of absolute errors of the predictions compared to the actual given ratings for a user: $MAE = \frac{1}{n}\sum_{i=1}^{n} |predicted_i - actual_i|$. The *Root Mean Squared Error* ($RMSE$) expresses the average of squares of absolute errors of the prediction compared to the existing rating for a user: $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(predicted_i - actual_i)^2}$. The smaller the values, the better the quality of recommendations. As there are ratings only for single users, for group recommendations, we experiment with different characteristics of query groups, choosing from heterogeneous to homogeneous groups, and report on the average MAE and RMSE over all group members. We also study the *number* and *size* of generated clusters as an indirect measure of quality. Intuitively, when a user belongs to a very small cluster, his friends selection is limited and the recommendations are worse compared to a user that gets recommendations from a larger pool of friends.

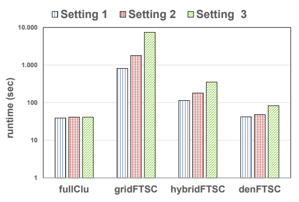
Experiments run on a 2.5 GHz Quad-Core i5-2450M architecture featuring 8.00 GB RAM and a 64-bit operating system. The distance between two users is evaluated as the Euclidean distance over their commonly rated items. When a specific subspace is considered, the distance relies only on the items that comprise the subspace.

6.1 Parameter Settings and Efficiency

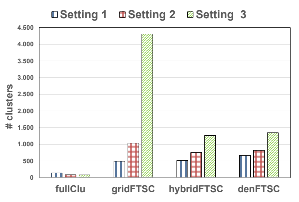
We study the execution time of our methods under different settings, and the number and dimensionality of the resulting clusters. We do not report on the *naive* approach here; according to [14], it takes 4 times longer than *fullClu*[3].

[2]`http://www.grouplens.org/node/73`

[3]Although there are more efficient methods for kNN acquisition, here we refer to the naive approach that does not use any special index structure, does not produce approximate results [5], neither it refers to a distributed environment [4].

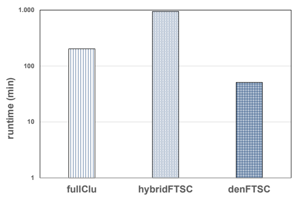


(a) Runtime

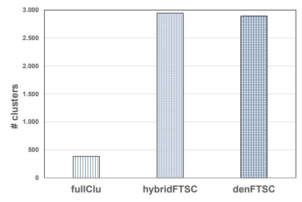


(b) # Clusters

Figure 4: ML-100K dataset: Runtime (in logarithmic scale) and #clusters for different parameter settings (c.f. Table 1).



(a) Runtime



(b) # Clusters

Figure 5: ML-1M dataset: Runtime (in logarithmic scale) and #clusters, for the parameters of Table 2.

For fullClu, the smaller the number of clusters, the better, since this indicates clusters of large cardinality, which is what we need for a broad selection of friends. For subspace clustering, a good clustering features many subspace clusters, because each user potentially belongs to several clusters, offering a wider and more diverse selection of friends.

6.1.1 *Dataset* ML-100K

We experimented with different parameter settings (Table 1). The runtime and number of generated clusters for each approach are depicted in Figure 4.
fullClu: The user similarity threshold *delta* has a strong impact on the algorithm performance. The higher its value, the smaller the number of clusters and the bigger (on average) the clusters. For example, for $\delta = 0.2$, there are 3-15 users per cluster, and 3 big clusters containing 27, 38 and 96 users, respectively. For $\delta = 0.7$, the range is 3-28 users and there is one big cluster of 138 users. The execution time also depends on δ; the lower it is, the longer the algorithm takes. fullClu has the smaller runtime, however its clustering might be problematic for determining users' friends, since it consists of a small number of clusters with imbalanced cluster cardinalities. This way, the selection of friends for a user of one of the (many) small clusters, is very narrow and therefore, the quality of the recommendations might be poor. The problem is not so severe for a user of a (usually one) big cluster, as his friends selection is more broad.
gridFTSC: The lower the density threshold $minPts$, the larger the number of clusters and the more higher-dimensional the clusters, since more grid cells are considered as dense. This way, the runtime increases with a decreasing value of $minPts$, as the number of clusters and the number of items to be considered for extension increases. For instance, for $minPts = 50$, the algorithm detects 494 subspace clusters, whose dimensionality lies in [1-4] range. The higher-dimensional subspace clusters are based mostly on the same subset of items. This implies that the dataset features some prominent items, which "derive" big clusters. When we lower the threshold, the number of clusters as well as the running time increase drastically, however the maximum subspace dimensionality of the clusters not; the maximum dimensionality is 5 for both $minPts = 40$ and $minPts = 50$. Higher-dimensional subspace clusters are desirable, as they demonstrate a higher agreement in terms of the included users preferences. Therefore, the choice of $minPts$ is a trade-off between algorithm run-time and clustering result quality. Compared to the other approaches, gridFTSC generates the larger number of clusters and is the slowest method.
hybridFTSC: Similarly to gridFTSC, with a decreasing value of $minPts$, the number of clusters and the dimensionality of subspaces increase. However, due to the significance threshold, the runtime is only a fraction of gridFTSC's runtime, while being able to detect a similar amount of clusters. For $minPts = 40$, the dimensionality of the detected clusters is in the [1-3] range, with mostly 1-item clusters. The large number of 1-item clusters comes from the comparatively high significance threshold used to speed up the algorithm. Nevertheless, the quality of recommendations, as we will see later, does not suffer from this. The algorithm also finds all the prominent items which have been determined by the grid-based approach. The dimensionality of the detected clusters increases for $minPts = 30$, while the increase in runtime is considerably low. For $minPts = 20$, the number of clusters significantly increases but still, the run-time is far below that of gridFTSC (5 vs 29 minutes). As the dimensionality of subspace clusters ranges in [1-4], the algorithm is able to compete with gridFTSC in terms of clustering quality at a significantly lower run-time.
denFTSC: This approach is superior in both runtime and clustering quality. For all parameter settings, it determines high-dimensional subspace clusters, while its runtime is unequaled. For $minPts = 35$, the dimensionality of the subspaces lie in the [1-5] range. All prominent items retrieved by the previous approaches are detected by the extracted subspace clusters, but also new items are added to the subspace cluster definitions. The variety in the items of the subspace clusters of denFTSC is significantly higher compared to the other approaches. Without an observable increase in runtime, the algorithm determines a noticeable higher number of subspace clusters comparing to hybridFTSC. For $minPts = 20$, even more clusters were detected at almost 1 minute, whereas comparative results by gridFTSC and hybridFTSC were achieved in approximately 30 and 6 minutes.

6.1.2 *Dataset* ML-1M

The parameter settings are depicted in Table 2, whereas the results are shown in Figure 5.
fullClu: Compared to ML-100K, the runtime increases drastically. In general, the overall behavior is analogous to the observations made so far: we receive plenty of small clusters and few clusters that are exceptionally big.
gridFTSC: Due to heap space limitations, we were not able to examine its performance on this dataset.
hybridFTSC: The performance deteriorates drastically. As it differs from denFTSC at the point where users are divided in two sets, we conclude that this operation is highly expensive. The number of discovered subspace clusters is similar to denFTSC. The dimensionality of the resulting subspace clusters for $minPts = 100$ and a significance threshold of 0.17 ranges between 1 and 3 items.
denFTSC: Although there is an increase in runtime compared to ML-100K dataset, it is not as drastic as with the other approaches. The performance is stable and superior to the other approaches in efficiency and effectiveness. For $minPts = 100$ and a threshold of 0.15, the algorithm determines 2,894 subspace clusters in 23 minutes. The dimensionality of the subspace clusters ranges from 1 to 4 items.

Concluding, denFTSC is superior to the other approaches in both datasets, as it exhibit a runtime comparable to fullClu, while deriving many subspace clusters and the larger subspace variety among the subspace clustering methods. Both gridFTSC and hybridFTSC face major difficulties when

Table 1: ML-100K dataset: Parameter settings and results

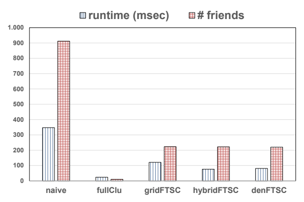
	Parameter settings	Runtime	# Clusters	Setting ID
fullClu	$\delta = 0.2$	39s 585ms	141	1
	$\delta = 0.5$	41s 115ms	87	2
	$\delta = 0.7$	41s 237ms	83	3
gridFTSC*	minPts = 50, grid = 3	13min 33s 55ms	494	1
	minPts = 40, grid = 3	29min 53s 364ms	1038	2
	minPts = 30, grid = 3	2h 3min 41s 655ms	4308	3
hybridFTSC (*)(+)	minPts = 40, $\epsilon = 0.1$	1min 54s 555ms	516	1
	minPts = 30, $\epsilon = 0.1$	3min 0s 345ms	754	2
	minPts = 20, $\epsilon = 0.1$	5min 51s 895ms	1265	3
denFTSC (*)(+)	minPts = 35, $\epsilon = 0.1$	42s 399ms	667	1
	minPts = 30, $\epsilon = 0.1$	48s 586ms	819	2
	minPts = 20, $\epsilon = 0.1$	1min 23s 15ms	1349	3

(*) parameters for FTSC: $\epsilon_o = 0.4$, $\epsilon_s = 0.3$, $\epsilon_g = 0.4$, (+) parameters for significance threshold: $\beta = 0.13, c = 2$

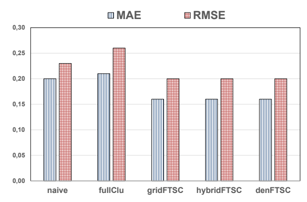
Table 2: ML-1M dataset: Parameter settings and results

	Parameter Settings	Run-time	#Clusters
fullClu	$\delta = 0.5$	3h 22min 57s 869ms	384
hybridFTSC (*)	minPts = 100, $\epsilon = 0.1$, d = 0.17, c = 2	15h 43min 50s 50ms	2946
denFTSC (*)	minPts = 100, $\epsilon = 0.1$, d = 0.15, c = 2	51min 5s 957s	2894

(*) parameters for FTSC: $\epsilon_o = 0.4$, $\epsilon_s = 0.3$, $\epsilon_g = 0.4$)



(a) Runtime & #friends



(b) MAE & RMSE

Figure 6: Top-10 user recommendations statistics (ML-100K dataset, Setting 1 from Table 1).

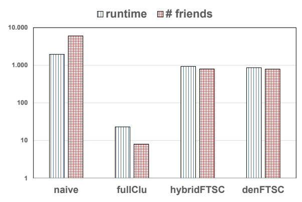
they come to cluster large datasets. hybridFTSC outperforms gridFTSC in runtime, while determining a clustering result that is qualitatively comparable. The fullClu approach, in general, is not a good option for determining user clusters, as it produces too many small clusters.

6.2 Quality of User Recommendations

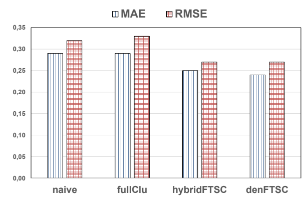
For examining the qualitative differences of our approaches, we randomly choose users with different demographics (occupation, age and sex) and a number of ratings equal to the average number of ratings per user. We issue the 10 most promising recommendations to users. For subspace clustering, we use $\beta = 0.15$, and, for the naive approach, we employ the distance threshold of the fullClu.

ML-100K: Next, we present results for a 34 years old educator with 70 ratings. According to his ratings, he seems to be interested in Drama, Action, Comedy and Romance movies (30, 18, 18 and 14 ratings), and not in Documentary, Fantasy, Horror or Western movies (0 ratings). For calculations, we employed the parameters settings 1 (Table 1), which give the less promising clustering results for all approaches. fullClu results in the smallest clusters and therefore, in a limited choice of friends. Subspace clustering approaches also result in the lower number of generated subspace clusters and the lower number of considered dimensions for these clusters. The results are shown in Figure 6.

The runtime is a great deal lower when employing user clusters, since *naive* scans the whole database to determine the friends of the query user. *fullClu* is the fastest method, however its quality of predictions suffers heavily from the small set of friends considered. The set of friends generated by the subspace clustering approaches is larger compared to fullClu, but still small compared to the naive ap-



(a) Runtime & #friends



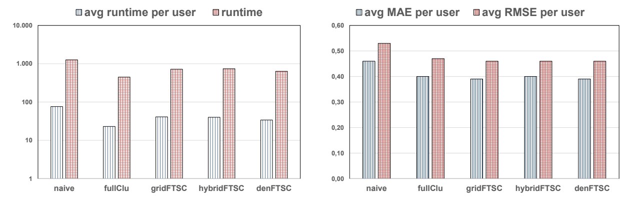
(b) MAE & RMSE

Figure 7: Top-10 user recommendations statistics (ML-1M dataset, Figure 7 (a) is depicted in logarithmic scale).

proach. MAE and RMSE, however, show that the predictions quality of the subspace clustering approaches increases when compared to naive, due to the careful selection of friends. gridFTSC is the slowest among the fault tolerant approaches due to the employed significance threshold, while all subspace clustering approaches issue the same suggestions, though their ranking might differ.

ML-1M: Here, we present results for a female scientist at the age of 56. She has submitted 148 ratings, preferring movies from the genres Drama, Comedy, Romance and Thriller (95, 33, 24, 20 ratings), whereas she does not seem to be interested in Western, Documentary, Sci-Fi, Film Noir or Fantasy movies (1, 1, 2, 2, 2 ratings).

Calculating recommendations on this dataset further amplifies the effects already observed (Figure 7). The runtime increases, except for fullClu which requires approximately the same time as before, since the cluster of our user contains only 9 users. Naive considers more than half of the overall users as friends, which is obviously too wide. Thus, both fullClu and naive suffer from a poor selection of friends, as reflected in the corresponding MAE and RMSE scores. On the other hand, hybridFTSC and denFTSC performed quite well. Their runtime is smaller than the one required by naive, thanks to the significance threshold, while the quality of recommendations does not suffer from this pruning heuristic, as MAE and RMSE show. hybridFTSC and denFTSC agree in all recommended items; there is a slight difference in their rankings. Naive calculated similar recommendations and gives the same top-2 items as the density-based approaches. fullClu though, totally disagrees with the other approaches in its recommendations. This confirms that the narrow selection of friends leads to poor recommendations.



(a) Runtime (b) MAE & RMSE

Figure 8: Runtime, avg runtime and avg quality values for a homogeneous query group (ML-100K dataset, Setting 1 from Table 1, Figure 8 (a) is depicted in logarithmic scale).

To conclude, fault tolerant subspace clustering approaches overcome the friends selection problems of naive and fullClu at a reasonable runtime. Although, subspace clustering leads to a larger selection of friends, the weighted ranking based on full dimensional distance and number of globally shared dimensions, refines the selection of friends and leads to more qualitative recommendations comparing to naive and fullClu.

6.3 Quality of Group Recommendations

Since there is no ground truth for group recommendations, for the quality evaluation, we rely on the quality of the individual group members recommendations. We report results for the fair design for groups with different user demographics. For the naive approach, we used the distance threshold of fullClu, whereas for subspace clustering, a weighted ranking with $\beta = 0.15$. The group has 10 members, and we issue the 10 most promising recommendations in each experiment.

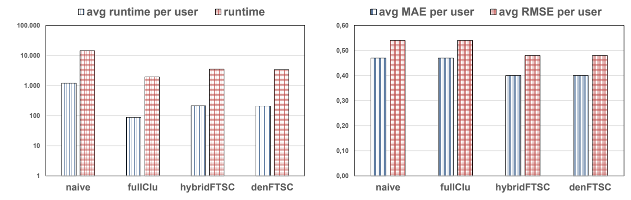
6.3.1 Homogeneous Query Group

We randomly generate query groups with users that exhibit homogeneity w.r.t. their demographics. In particular, we choose users sharing the same occupation, age range and sex, assuming that due to these similar characteristics their movie taste will be also similar.

ML-100K: For clustering, we used the parameters setting 1 (Table 1). We choose 10 young male programmers with age between 28 and 30. Figure 8 displays the total runtime, the average runtime per user, and the MAE and RMSE scores averaged over all group members. The runtime of the naive approach increases rapidly, since a sequential scan of the database is required for each group member to detect the set of friends. denFTSC is the best among the subspace clustering approaches, while the best runtime is achieved by fullClu. Regarding quality, fullClu is the worst and denFTSC the best. All subspace clustering approaches agree in their top-5 recommendations, although their rankings slightly differ. denFTSC and hybridFTSC even comply with each other in their top-7 recommendations.

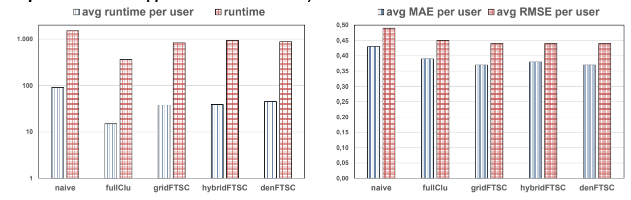
ML-1M: Here, we consider a group with 10 female homemakers aged between 35 and 44. Figure 9 displays the runtime, average runtime and quality scores per user. The observations are similar to the ones for ML-100K. The difference in runtimes is even greater now, due to the dataset size. Naive requires double time compared to denFTSC. fullClu is the fastest and the one with the worst quality, and denFTSC the one with the best quality. hybridFTSC and denFTSC agree in 8 out of 10 items; their rankings slightly differ.

In overall, denFTSC offers the best trade-off between runtime and quality. Although fullClu is the fastest, its quality is the worst, since it depends on the positioning of the group members in clusters; small clusters result into narrow friends sets and poor recommendations. denFTSC offers a wide selection of friends due to the different subspace clusters that a group member belongs to. Also, due to the weighted ranking filtering of these friends, the resulting set of friends upon which the recommendations are based is highly qualitative.



(a) Runtime (b) MAE & RMSE

Figure 9: Runtime, avg runtime and avg quality values for a homogeneous query group (ML-1M dataset, Figure 9 (a) is depicted in logarithmic scale).



(a) Runtime (b) MAE & RMSE

Figure 10: Runtime, avg runtime and avg quality values for a heterogeneous query group (ML-100K dataset, Setting 1 from Table 1, Figure 10 (a) is depicted in logarithmic scale).

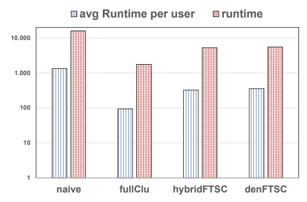
6.3.2 Heterogeneous Query Group

Next, we examine groups of users that have been randomly selected by considering different demographics.

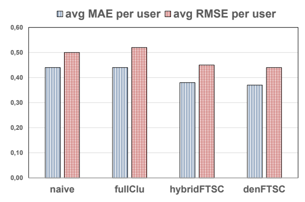
ML-100K: For the clustering approaches, we used the parameters setting 1 (Table 1). The query group consists of 5 women (an executive, an artist, a student, an engineer and a retired lady) and 5 men (a librarian, two students, a scientist and an educator). Figure 10 displays the runtime, the average runtime and quality measures per user. For efficiency, the conclusions drawn so far hold. The worst runtime is that of naive. fullClu is the fastest with the worst quality. In general, subspace clustering offers good quality at acceptable runtimes; the best quality is achieved by denFTSC. Concerning the actual results, hybridFTSC and denFTSC agree in 8 out of 10 recommendations and in their top-3 items, while fullClu shares on average 6 out of 10 items with the subspace clustering approaches.

ML-1M: Here, the females are a grad student, a writer, a doctor, an executive, and a self-employed lady. The males are a programmer, an engineer, an artist, a tradesman, and a retired man. The results are shown in Figure 11. Again, naive is the slowest and fullClu is the fastest with the lowest quality scores. hybridFTSC and denFTSC finish significantly faster than naive and their quality scores are better from those of naive and fullClu. Therefore, they compromise a good trade-off. Regarding the actual recommendations, hybridFTSC and denFTSC agree in 9 out of 10 recommendations, and they even agree in their rankings.

To conclude, the effects we observed for single user recommendations are stronger in case of a group. The runtime of naive increases rapidly, since the database needs to be scanned for each user in the group. The quality, which depends on the quality of the individuals recommendations, relies on the selection of the friends set. Neither a very narrow friends selection, like in fullClu, nor a very wide one, as in naive, perform well. Our experiments show that a broad pool of diverse friends achieved by subspace clustering and a qualitative selection among them based on weighted ranking, offer the best recommendations at a fair runtime.



(a) Runtime



(b) MAE & RMSE

Figure 11: Runtime, avg runtime and avg quality values for a heterogeneous query group (ML-1M dataset, Figure 11 (a) is depicted in logarithmic scale).

7. RELATED WORK

Typically, recommendation approaches are distinguished between content-based and collaborative filtering. Content-based approaches recommend items similar to those the user previously preferred (e.g., [13]), while collaborative filtering approaches recommend items that users with similar preferences liked (e.g., [9]). Several extensions have been proposed, such as time-aware recommendations (e.g., [18, 17]) and group recommendations (e.g., [3, 14]). Lately, there are also approaches on extending database queries with recommendations [10, 16].

To facilitate the selection of similar users to a query user, clustering has been employed to pre-partition users into clusters of similar users and rely on cluster members for recommendations. For example, [14] employ full-dimensional clustering; as explained though, full dimensional clustering is not the best option due to the high dimensionality and sparsity of data. Dimensionality reduction techniques, like PCA, could be applied to reduce dimensionality, however clusters existing in subspaces rather than in the original (or reduced) feature space will be missed.

[1] proposes a research paper recommender system that augments users through a subspace clustering algorithm. However, only binary ratings are considered, and therefore, the problem is simplified, since typically ratings lie in a value range with higher values indicating stronger preferences. On the contrary, we use full range of ratings. [12] also uses subspace clustering to improve the diversity of recommendations, the dimensions considered though, are not the items but rather more general information extracted upon these items (like movie genres). This way, neither the high dimensionality of the data nor the missing values problem is confronted. In our approach, we use subspace clustering upon item ratings to diversify the resulting set of friends and a fault-tolerant approach to deal with missing ratings. Initial ideas on employing subspace clustering for recommendations appeared in [15]. In this work, we proceed further by proposing two density based fault tolerant subspace clustering approaches which, as we show, perform better. Moreover, we introduce the significance threshold pruning to reduce the runtime and the weighted ranking approach to combine subspace friends into a final friends set for recommendations. We also provide an extensive experimentation for both users and groups of users.

8. CONCLUSIONS

In this work, we integrate subspace clustering into the recommendation process to improve its efficiency and effectiveness. We examine a fault tolerant approach, which relaxes the cluster model, so that missing ratings can be tolerated. We extend grid-based fault tolerance to the density-based clustering paradigm and propose the *hybridFTSC* and *denFTSC* approaches. To improve runtime, we introduce the notion of *significance threshold* that identifies prominent dimensions for subspace cluster extension. To improve the quality of predictions, we combine the wide selection of diverse friends offered by subspace clustering with a *weighted ranking* based on full-dimensional distances between users. Such a concurrent consideration of both local and global distances (in clustering and raking, respectively) allows for a wide selection of friends and a fine selection of the best ones for computing recommendations. Our experiments show that fault tolerant subspace clustering approaches outperform, in terms of runtime and quality, both the naive and the fullClu approaches.

Acknowledgments

This work was partially supported by the European project DIACHRON (IP, FP7-ICT-2011.4.3, #601043).

9. REFERENCES

[1] N. Agarwal, E. Haque, H. Liu, and L. Parsons. Research paper recommender systems: A subspace clustering approach. In *WAIM*, 2005.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*, 1998.

[3] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group recommendation: Semantics and efficiency. *PVLDB*, 2(1):754–765, 2009.

[4] A. Boutet, A.-M. Kermarrec, D. A. Frey, R. Guerraoui, and A. Jegou. Whatsup: A decentralized instant news recommender. In *IPDPS*, 2013.

[5] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[7] S. Günnemann, E. Müller, S. Raubach, and T. Seidl. Flexible fault tolerant subspace clustering for data with missing values. In *ICDM*, 2011.

[8] K. Kailing, H.-P. Kriegel, and P. Kröger. Density-connected subspace clustering for high-dimensional data. In *SIAM*, 2004.

[9] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, 1997.

[10] G. Koutrika, B. Bercovitz, and H. Garcia-Molina. Flexrecs: expressing and combining flexible recommendations. In *SIGMOD*, 2009.

[11] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, Mar. 2009.

[12] X. Li and T. Murata. Using multidimensional clustering based collaborative filtering approach improving recommendation diversity. In *Web Intelligence/IAT Workshops*, 2012.

[13] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *ACM DL*, 2000.

[14] E. Ntoutsi, K. Stefanidis, K. Nørvåg, and H.-P. Kriegel. Fast group recommendations by applying user clustering. In *ER*, 2012.

[15] K. Rausch, E. Ntoutsi, K. Stefanidis, and H.-P. Kriegel. Exploring subspace clustering for recommendations. In *SSDBM*, 2014.

[16] K. Stefanidis, M. Drosou, and E. Pitoura. *You May Also Like* results in relational databases. In *PersDB*, 2009.

[17] K. Stefanidis, E. Ntoutsi, M. Petropoulos, K. Nørvåg, and H.-P. Kriegel. A framework for modeling, computing and presenting time-aware recommendations. *T. Large-Scale Data- and Knowledge-Centered Systems*, 10:146–172, 2013.

[18] L. Xiang, Q. Yuan, S. Zhao, L. Chen, X. Zhang, Q. Yang, J. Sun, and J. Sun. Temporal recommendation on graphs via long- and short-term preference fusion. In *KDD*, 2010.