xStreams: Recommending Items to Users with Time-evolving Preferences

Zaigham Faraz Siddiqui
University of Magdeburg
Magdeburg, Germany
siddiqui@iti.cs.uni-magdeburg.de

Eleftherios Tiakas
Aristotle University of Thessaloniki
Thessaloniki, Greece
tiakas@csd.auth.gr

Panagiotis Symeonidis
Aristotle University of Thessaloniki
Thessaloniki, Greece
symeon@csd.auth.gr

Myra Spiliopoulou
University of Magdeburg
Magdeburg, Germany
myra@iti.cs.uni-magdeburg.de

Yannis Manolopoulos
Aristotle University of Thessaloniki
Thessaloniki, Greece
manolopo@csd.auth.gr

ABSTRACT

Over the last decade a vast number of businesses have developed online e-shops in the web. These online stores are supported by sophisticated systems that manage the products and record the activity of customers. There exist many research works that strive to answer the question "what items are the customers going to like" given their historic profiles. However, most of these works miss to take into account the time dimension and cannot respond efficiently when data are huge. In this paper, we study the problem of recommendations in the context of multi-relational stream mining. Our algorithm first separates customers based on their historic data into clusters. It then employs collaborative filtering (CF) to recommend new items to the customers based on their clustering structure. We evaluate our algorithm on two data sets, MovieLens and a synthetic data set.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications—*Data Mining*

Keywords

Stream recommenders, evolving user preferences, stream mining

1. INTRODUCTION

Data mining for recommendation engines is a mature domain, and the underpinnings of the core procedure nowadays constitute public knowledge. In the simplest scenario, the recommender matches the preferences of a peer user to the preferences of users similar to her, and then recommends items that these similar users have liked. In the most cases the user preferences are retrieved through the web, recorder in historical databases and used by the recommendation engines, which are included in internet applications, e-business systems, e-shops etc. Data mining and Web mining are responsible for learning a model on user similarity; the simplest case involves offline clustering users on similarity of preference. However, this traditional view of a recommendation engine oversees the fact that user preferences change over time: the preferences of a 22-year old girl are not the same she had when she was a teenager. In this study, we propose a method that couples data *stream* mining with a recommendation engine to learn and exploit the evolution of user preferences.

The awareness on the importance of time for a recommendation engine gained momentum during the Netflix competition (an one-million-dollar competition for making a recommender that outperforms the existing system). The competition's winner, Yehuda Koren, has demostrated how the incorporation of time into an ensemble of learners boosts predictive performance [8]. Remarkably, the approach of Koren can*not* adapt to the evolving preferences of people: it rather captures the preferences of the people during the time period of observation and learns a *static* model on them. This is not appropriate for the dynamic environment of a recommendation engine: the model learned by the miner on the basis of observed transactions must be *adapted* as new transactions arrive, because new transactions reflect the evolution of people's preferences.

To deal with this issue, we observe the activities of the users as a *stream*, and perform stream clustering upon similar entities. At each timepoint, the recommendation engine uses the clusters to identify users similar to the peer user and deliver suggestions to her. However, the task of clustering similar *users* on their evolving preferences does not agree with the conventional stream mining paradigm either! To juxtapose the task of learning the preferences of evolving users to a conventional stream mining task, consider following example of a traditional stream.

Assume an e-shop that records customer purchases. The purchases constitute a stream, an instance of which contains the identifiers of the customer and of the purchased prod-

WIMS '14 June 02-04, 2014, Thessaloniki, Greece

ACM 978-1-4503-2538-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2611040.2611051.

uct, the timepoint of the purchase, and further information such as the product's price, payment option, shipment option, delivery address etc. The shop is interested in detecting fraudulent purchases. To this purpose, transactions are clustered on similarity and outliers are inspected. As new purchases arrive, the clusters evolve, some transactions turn to be similar to earlier outliers (and are thus suspect), while new outliers show up (and must be inspected, too). In other words, the stream clustering algorithm adapts the clusters to arriving data and allows for the detection of new kinds of fraudulent transactions.

The above example describes (unsupervised) model learning on a stream of instances - the purchases. The recommendation scenario is a bit different though.

Assume again the e-shop that records customer purchases as a stream. The shop is interested to recommend to each customer X a product that she will like, knowing that her attitude to some products may have changed. To this purpose, the list of products preferred by X must be extended whenever she buys a new product, but the timepoint of the purchase must also be recorded. The miner must now cluster together users who are similar to each other because they (a) now like the same products and (b) have liked the same products in the past.

The two examples differ, because the first one involves clustering the instances on similarity, while the second one requires clustering the *users* on similarity of the *sets of instances* associated to them at different moments. This task requires multi-relational stream clustering. We build upon our earlier method [19] for clustering a stream of complex objects, and extend it to deliver insights on user similarity on the basis of their past transactions.

The paper is organized as follows. In the next section we discuss related work on recommendations over dynamic data and on multi-relational stream learning. Then, we present our approach for the formulation of recommendations towards users with evolving preferences. We report on our experiments with historical data for the MovieLens and a synthetic data set. The last section concludes our work.

2. RELATED WORK

In this Section, we will present the main work in collaborative filtering, content-based filtering and hybrid recommender systems. Moreover, we will discuss the latest progress on time-aware recommender systems, which is a rather new research area. Notice that, related work comes from two different direction: the domain of (i) temporal mining and (ii) stream mining. The difference between these research directions is that the latter does assume that all data are retained in eternity. Thus, data elements must be discarded or archived at some point and the model should be able to adapt to drifts in the underlying concept (concept drift).

2.1 Content-based, Collaborative Filtering and Hybrid Recommender Systems

Three parallel approaches have emerged in the context of recommender systems: collaborative filtering (CF), content-based Filtering (CB) and hybrid methods.

Collaborative filtering algorithms recommend those items to the target user, that have been rated highly by other users with similar preferences and tastes [16, 18, 6, 10]. In most CF approaches, only the item and users' identifiers are accessible and no additional information over items or users is provided. Websites that provide recommendations in the form, "Customers who bought item i also bought item y", typically fall under collaborative filtering approaches. Grouplens research group [16] introduced a collaborative filtering algorithm, known as user-based CF, because it employs users' similarities for the formation of the neighbourhood of nearest users. Another CF algorithm proposed by [18], is known as item-based CF algorithm, because it employs items' similarities for the formation of the neighbourhood of nearest users. A pitfall of CF is the cold start problem: new items have received only few ratings, so they cannot be recommended; new users have performed only few transactions, so there are hardly other users similar to them.

Content-based filtering assumes that each user operates independently. It exploits only information derived from documents or item features (eg. terms or attributes) [15, 14, 11]. In particular, it exploits a set of attributes which describes the items and recommend other items similar to those that exist in the user's profile. In this way, the cold start problems for new items and new users are alleviated, *provided* that users prefer items that are similar in content to those they have already chosen. However, the pitfall of CB is that there is no diversity in the recommendations. That is, the user gets recommendations that are very familiar to her, since these recommended items are similar to those in her item profile.

There have been several hybrid attempts to combine CB with CF. The Fab System [1], combines CB and CF in its recommendations, by measuring similarity between users after first computing a profile for each user. Fab initially categorizes documents by a CB filter and then recommends them to the test user based on his relevance feedback. In contrast, the CinemaScreen System [17] runs CB on the results of CF. In particular, CinemaScreen system computes predicted rating values for movies based on CF and then applies CB to generate the recommendation list. Our approach can be categorized as a hybrid method, since it combines both content-based characteristics and collaborative filtering. In contrast to the aforementioned hybrid methods, we also incorporate in our model the time dimension.

2.2 Time-aware Recommender Systems

In the last years, there are scholars who proposed methods that attempt to capture the temporal aspects of user behaviour, while others investigated the updating of recommenders to change in the user behaviour. Remarkably, these two categories of methods are distinct, in the sense that studies capturing temporal aspects produce static models, while studies capturing change produce dynamic models. Our approach belongs to the second category. In this Section, we will briefly discuss the approaches of Ding and Li [4] and of Yehuda Koren [7, 8], which belong to the first category.

Ding and Li [4] proposed a method that assigns time weights for purchases of items by decreasing the weight to old purchasing data. They have shown that the users' purchase habits vary and even the same user has quite different preferences towards the same items over time. Yehuda Koren [7, 8] considers the scenario of item rating by capturing the influence of time on ratings. It has been identified the following aspects: (i) user-bias (deviation from the average) changes over time, i.e. the user's rating assigned to an item

may vary over time and may exhibit periodicity, (ii) the item-bias changes over time, and (iii) the ratings submitted by a given user (i.e. user's preference) may vary depending on time of day, day of week or period in the year. Based on the aforementioned aspects a baseline predictor (which assigns to each item an average rating μ) can be extended as shown in Equation 1:

$$\widehat{r}_{ui} = \mu + b_u(t_{ui}) + b_i(t_{ui}), \quad (1)$$

where $\widehat{r}_{ui}$ is the predicted rating of a user u for an item i, μ is the overall average rating, b_u and b_i are the user and item bias over μ respectively, and t_{ui} denotes the rating of user u on item i at day t.

The above baseline predictor can be easily integrated into a factor model [7, 8]. Koren and Bell [9] proposed timeSVD++, a set of predictors that learn latent factors (as with traditional matrix factorization) thereby exploiting implicit information on user preferences (i.e which items users rate, regardless of their rating) and the impact of time (including day effects on a user's rating attitude). timeSVD++ was shown to offer accuracy superior to SVD++ [9].

Although the above approaches are temporal and of evolutionary nature, they are not appropriate for stream mining, because stream mining requires that a model is adapted to new data as they arrive, while the aforementioned methods learn a model that explains all data seen thus far. An excellent elaboration on this issue has been written down by Koren and Bell themselves in [9], subsection '5.3.4.1 Predicting future days' (page 160), stating among others that '... for those future (untrained) dates, the day-specific parameters should take their default value.' Although they refer explicitly to day-specific parameters, we must keep in mind that *all* parameters of the timeSVD++ have been specified with cross validation upon the whole dataset [9]. Once new untrained data arrive, cross validation must be rerun, because, as Koren and Bell state on the same page 160 '... our temporal modelling makes no attempt to capture future changes. All it is trying to do is to capture transient temporal effects, which had a significant influence on past user feedback', whereby we should like to stress the word 'past'.

Another way of formulating the statements of Koren and Bell is that timeSVD++ learns a *static* model over a finite dataset, and can be applied on future data (on a stream) if and only if all thinkable concept drift can be captured by modelling this finite dataset, i.e. if and only if future data follow exactly the same distribution as past data. Stream mining research encompasses methods for the analysis of data for which this assumption does not hold.

2.3 Stream Mining for Recommender Systems

The importance of model updating in a recommender has been demonstrated by Dias et al. [2]. They have experimentally shown that updating the checkout recommender model files consistently resulted in an increase in the number of new shoppers using the recommender system.

A truly adaptive recommendation engine for streams has been proposed by Nasraoui et al. in [13] for the prediction of the next inspected page in user sessions. The stream under inspection is the clickstream, in which sub-sessions are observed, matched to already seen (complete) sessions, from which the top-N recommendations are derived. The recommendations are formulated and evaluated immediately by reading in the users' responses (essentially, the next page click per session) and incorporating them into the model learned thus far. For model learning and updating, Nasraoui et al. consider kNN, which finds the k most similar sessions to an input sub-session, and their stream clustering algorithm TECNO-STREAMS [12], which learns and adapts profiles (as abstractions of sessions) and returns for each input sub-session the profile most similar to it.

Nasraoui et al. [13] have tested their adaptive recommender on two Web clickstreams, thereby simulating two evolution scenarios. In the "induced drastic sequential user profile evolution" (scenario D), the sessions of the data set are clustered into profiles, and the sessions of each profile are delivered to the recommender *one profile at a time*. Hence, once the session of a given profile are read through, there come sessions that do not fit to the model learned by the recommender at all - this corresponds to a shift. In the 'natural or mild chronological order' (scenario M), the sessions arrive in chronological order, hence the profiles are mixed. The evaluation on moving average of the F-measure shows that kNN is of slight advantage over TECNO-STREAMS in the scenario D, because the former relies more on model learning, while the two algorithms behave similarly under Scenario M.

Our approach follows the same philosophy as the recommender of Nasraoui et al. [13]: we anticipate that the recommender should forget old instances and learn from newly arriving instances immediately; the recommender uses profiles instead of matching individuals; the profiles are adapted over time to respond to evolving user preferences. Our evaluation is also dictated by the idea that the recommender must deal with both sudden shift (as in scenario D) and with arbitrary drift (scenario M) [1] In short, we share the core ideas of proper stream learning under drift.

Yet, the objective of Nasraoui et al. [13] makes their algorithm not comparable to ours. In particular, their recommender predicts the next page of a user session, and adapts as more and more of the session is observed. Although it is possible to express the sequence of items ever observed by a user as a session, this is not desirable: a user session is a matter of moments or hours, while the interaction of a user with an e-shop may extend arbitrarily across time; forgetting the beginning of a session as the session progresses is unintuitive, while forgetting very old purchases of a user is a reasonable option; a user revisits pages s/he has studied before, thus allowing for forms of sub-session matching that make no sense when comparing the preferences of users. Most importantly, a session in clickstream mining is a sequence of page identifiers, whereby we incorporate into user similarity and model learning also the properties of the products observed at each moment, allowing the impact of old products to fade out. A comparison of clustering algorithms might have been possible, but the experiments of [13] have shown that kNN is mostly superior and never truly inferior to TECNO-STREAMS. Hence, we compare our approach to an adaptive collaborative filtering stream recommender that essentially uses kNN.

Recently, Diaz-Aviles et al. have proposed Stream Ranking Matrix Factorization (RMFX) [3], an algorithm that is intended to perform matrix factorization and item ranking on a stream. The focus of the algorithm is on maintaining an up-to-date model on the basis of possibly small, intelligently

[1]We use more elaborate scenarios for concept drift: assuming only one profile at a time and abrupt change, as in scenario D [13] is a bit simplistic.

devised samples. Accordingly, the experimentation was done on two time slots only (one slot used for learning, the other for testing) and delivered insights on the efficiency of the algorithm but not on its adaptivity. Beyond this, RMFX must know the sets of users and of items (i.e. the dimensions of the matrix) in advance (similarly to timeSVD++); then, it can fill it gradually. This makes the algorithm inappropriate for our scenario: in a realistic long-term setting, new customers show up and new items may be put to sale at any time, hence the matrix dimensions cannot be known in advance. Note that the previously mentioned algorithm of Nasraoui et al. [13] has neither caveat: it has been designed to be adaptive and it does not need to know all users nor all pages that a user may choose to access. Hence, in our evaluation we use as baseline an algorithm that satisfies the same core properties as the one of [13].

3. THE XSTREAMS METHOD

Our stream recommender xStreams consists of two modules. (i) The back end is an incremental, adaptive learner that processes the streams of activities (purchases with ratings) and associates it with earlier obtained and updated information on the entities involved - users/customers and items/products. (ii) The front end builds on top of the incremental learner to deliver the top-N items as recommenders to each user. We first describe the process of reading *instances* from the stream of postings and combining them with earlier recorded information on the referenced *entities* (users, items). This process is called 'incremental propositionalisation' and comes from our earlier work [19]. We use the results of this process to compute the similarity of a given user to other users, as described next in subsection 3.2. The back end and the front end algorithms are presented as pseudo-code in subsection 3.3, where we also discuss their complexity. As running example we use the multi-table stream of ratings, users and items in Figure 1.

3.1 Incrementally combining information on users, items and ratings

The core source of information for the recommendation engines is the stream of activities performed by the users. In the classical stream mining scenario, stream instances are observed, processed and forgotten. In the context of learning, a stream record is used to adapt the model (here: a model of the *ratings*) and is then forgotten. However, for the recommendation scenario we study, we want to learn and adapt a model of the *users*, so that the recommender can respond to changes in a user's attitude towards items.

3.1.1 Learning Task on Multiple Streams

In Figure 1(a), we depict users and their ratings for movies by means of three tables. Model learning and updating by the recommender's back end will be performed on the table *User*, which is linked to the table *Rating*, which is in turn linked to the table *Movie*. The stationary information on users, such as name and gender, is stored in the table *User*, but learning must also exploit each user's ratings for movies, as well as the properties of the movies themselves, e.g. genre. Since ratings arrive at any time, *Rating* is a stream, the records of which are seen and forgotten. In contrast, users and movies are *perennial* entities: they are stored in the database and retrieved from it to be linked to new information (new ratings). Nonetheless, new users and new movies may also arrive at any time, hence *User* and *Movie* are also streams - streams of *perennial* entities [21]. For perennial entities, we use the terms 'table' and 'stream' interchangeably hereafter, while for a *ephemeral* records like the ratings we use solely the term 'stream'.

The back-end of our method, xStreams_backend, is an adaptive stream mining algorithm that learns a model over the table *User* - as it is extended with information from the streams *Rating* and *Movie*. However, the natural join of entities of these entity types is not appropriate for learning: as can be seen in Figure 1(b), this join result contains as many user entities as ratings - user David (a single person) appears as four *independent* entities. For model learning, we rather need an expansion of table *User* with the data from the other tables. To this purpose, we use the *incremental propositionalisation* algorithm proposed in [19]: it expands the so-called 'target stream' (here: *User*) with information from the other streams, and produces one entry per entity of the target, as can be seen in the first two rows of the propositionalised table in Figure1(c). When new stream records arrive, the vectors of the entities are updated; in Figure 1(c), we see the entries/vectors of David and Tom for June 2012 and then for July 2012 [2].

3.1.2 Incremental Propositionalisation for Learning

More formally, let $\mathcal{T}$ the *target stream*, i.e. the stream, on which we want to perform the learning: in our recommendation scenario, $\mathcal{T}$ is the stream of users (cf. *User* in Figure 1). It is a *stream*, because new users arrive at any time. It is the *target* stream, because we want to learn the user profiles and exploit them to compute user similarity. For this learning task, we combine $\mathcal{T}$ with information from further streams $\mathcal{T}_1, \ldots, \mathcal{T}_J$ referencing it [3]; in our recommendation scenario, these further streams are the stream of ratings and the items (also a stream) - cf. *Rating*, *Movie* in Figure 1. We slide a window of length W timepoints over the streams, so that the data observed at timepoint t are the entities from all streams observed in the interval $(t - W, t]$. Entities outside this interval are forgotten.

Let $schema(X)$ be the schema of any of these streams $X \in \mathcal{A} := \{\mathcal{T}, \mathcal{T}_1, \ldots, \mathcal{T}_J\}$. At each timepoint t, our incremental propositionalisation method [19] expands each entity $u \in \mathcal{T}$ with the contents of the entities that reference u - they constitute the set $matches(u)$, the elements of which belong to different streams from $\mathcal{A}$. For this *expansion operation*, our method extends $schema(\mathcal{T})$ by turning the values of the elements in $matches(u)$ into new columns/features for e_u. The set of features thus generated can change at each timepoint, we therefore denote it as $Features(t)$. In particular, at timepoint 0:

- For each numerical attribute A in $\cup_{X \in \mathcal{A}} schema(X)$ that appears in the schema of some element $y \in matches(u)$, we add to $Features(0)$ four features: the *min*, *max*, *av-*

[2] In Figure 1(b), we show each user twice to demonstrate the differences in the aggregated ratings between June and July. The learner will see only one entry/vector per user: in June 2012 it uses the June vector, in the next month this old vector is replaced by the July vector.

[3] Any of $\mathcal{T}_x$ may be a static table, but we consider the general case where all of them are streams. For example, if movie genre were not an attribute but rather an entity type *Genre*, then we could assume that all possible genres are known in advance, hence *Genre* would be a table rather than a stream.

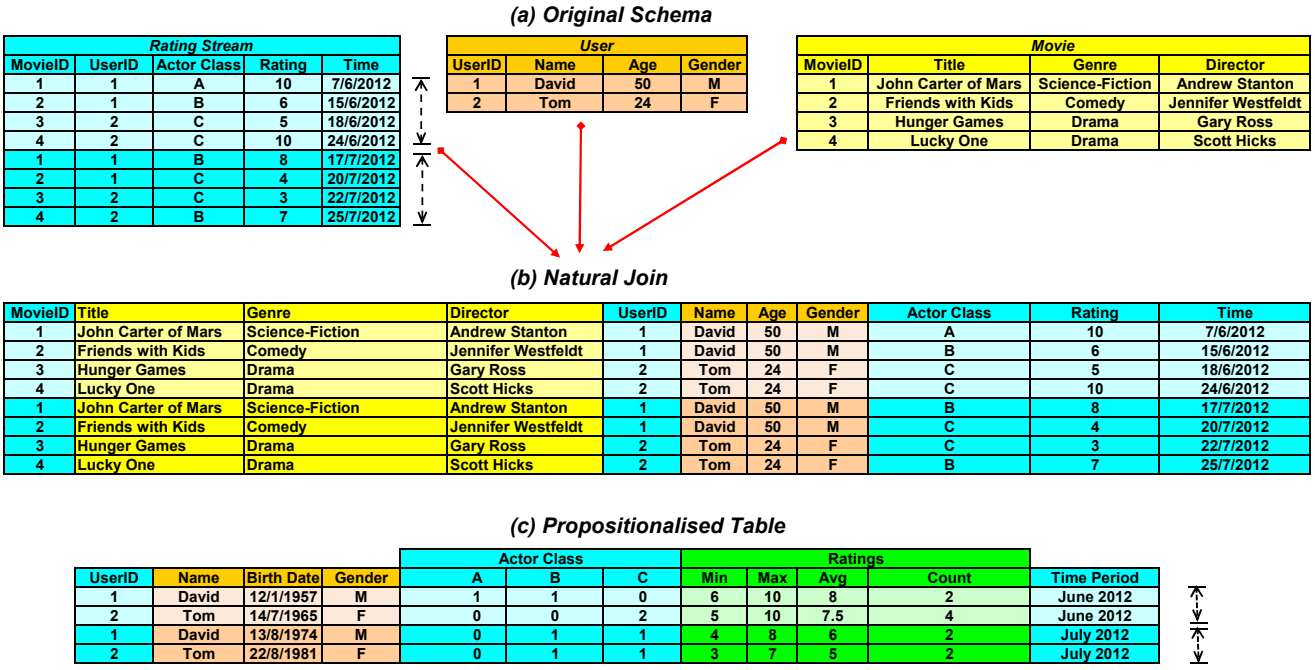


Figure 1: User and Movie entities linked to Rating entities: (a) the original schema consists of three tables, all of which are actually streams; (b) the natural join over them results in one entry/vector per rating, while (c) the propositionalisation operation produces one vector per user. We perform incremental propositionalisation to learn over the User entities, as they grow with information on ratings and movies.

erage and *count* for A, and we store in them the corresponding values seen in y. In our running example, we have calculated the *min*, *max*, *average* and *count* of ratings for each user_id per month – see Figure 1(c).

- For each nominal attribute A in $\cup_{X \in \mathcal{A}} schema(X)$, we create r_A features, one per distinct value of A observed at timepoint 0 for all entities in the target stream. For an entity x, each of those features takes the value 1(one) if the original attribute value was in an entity in $matches(x)$ and 0 (zero) otherwise.

At a later timepoint t, we update the numerical features by adding values for the arriving entities in $matches(x)$ and by subtracting values for the entities that exit the sliding window, i.e. have been seen earlier than $t - W$. For the nominal attributes, we can extend $Features(t)$ as new, previously unseen, nominal values arrive. However, it is not feasible to expand $Features(t)$ to unlimited values. Rather, we set an upper threshold size τ to the number of generated features per attribute and *encode* the values observed thus far for this attribute into τ derived features. This encoding is based on grouping values that appear in otherwise similar entities together into τ clusters. Details on this encoding are provided in [19].

As can be seen in the two entries per user of Figure 1(c), the propositionalisation algorithm delivers at each timepoint the vector of each *active* user, i.e. for each user who has performed some rating inside the sliding window. The user's vector contains the information obtained on this user from the data in the window; these data are summarized, while data that have slide outside the window are forgotten. Details on window sliding and memory management can be found in [20].

3.2 Computing the Similarity of Evolving Users

The vectors of the active users form the basis for computing user-user similarity. We consider two aspects of similarity between users: similarity on the basis of summarized past preferences and similarity on the basis of current ratings. We describe these two types of similarity below, and then explain how we combine them into a single similarity function. It must be stressed that the similarity values change as new ratings arrive, hence we need to update the similarity matrix in an incremental way or replace it with some surrogate that can be computed efficiently.

3.2.1 *Similarity on the basis of past preferences*

Let u, v be two users that are active at timepoint t, i.e. have performed ratings in the interval $(t - W, t]$, where W is the window size. Let e_u^t, respectively e_v^t be the updated vectors of these two users after propositionalisation on all information within the interval. We define the similarity between these users with the function:

$$simCB(t, u, v) = \frac{e_u^t . v_v^t}{|e_u^t| . |e_v^t|} \quad (2)$$

The postfix CB in the name of $simCB()$ stands for 'Content-Based' and reflects the fact that attributes of the users are also taken into consideration by the similarity measure.

For the efficient computation of similarity on the basis of past preferences, we couple incremental propositionalisation with stream clustering of the users' entries. In particular,

at each timepoint t, we retrieve from secondary storage all users who performed ratings within the interval $(t - W, t]$ (W is the window size), i.e. all active users. We expand the entries of these users with incremental propositionalisation, place them into K clusters and then adapt the clusters through centroid re-computation [20]. Then, for each user u we can return the k most similar users by assigning u to the cluster/profile with the closest centroid and then depicting the k nearest neighbours to u from this cluster.

It is thinkable that the cosine similarity used in Equation 2 is extended to consider only co-inspected items, in a similar way that only co-rated items are used in Collaborative Filtering to compute user similarity. In particular, for the similarity between two users in CF, items not rated by one of the users are ignored. By this, it is avoided that user similarity takes into account items that one user has never inspected. In $simCB()$, we could similarly restrict the similarity computation to skip features (derived attributes, cf. subsection 3.1) that have not perceived by one of the users being compared. For example, if a user has never seen a movie of a specific producer, we could ignore the attribute referring to this producer, when comparing this user with others. This extension has not been considered in the following; it is left as future work.

3.2.2 Similarity on the basis of ratings

Using the underpinnings of [6, 10], let the rating of a user u over an item i be denoted as $r_{u,i}$. If the user has not rated the item i we set $r_{u,i}$ to NULL. Since we perform collaborative filtering on a stream, we slide a window of length W over the stream of ratings and consider at each time point t only the set of ratings R_t inside the window $(t-W, t]$. At timepoint t, let $I_{t,u}$ be the set of items rated by u within R_t and $I_{t,v}$ the corresponding dataset for another user v. Then, the ratings-based similarity between u and v is computed as:

$$simCF(u,v) = \frac{\sum_{i \in I_{t,u} \cap I_{t,v}} (r_{u,i} \cdot r_{v,i})}{\sqrt{\sum_{i \in I_{t,u}} (r_{u,i})^2} \sqrt{\sum_{i \in I_{t,v}} (r_{v,i})^2}} \qquad (3)$$

where $I_{t,u}$ and $I_{t,v}$ are computed anew at each time point t and may have an empty intersection inside some window, although they were overlapping before that window.

Combining different aspects of user similarity: The conventional similarity function $simCF()$ exploits similarity of ratings between two users. Knowledge about each user's profile and expressed past preferences is captured by our new similarity function $simCB()$, which exploits accumulated past information from the aggregated feature profile of the users. As in conventional collaborative filtering, the expected rating is computed as the weighted average of the ratings made by users similar to u, but xStreams combines the two similarity functions when it computes the expected rating of user u for item j, $\hat{r}_{u,j}$. For rating prediction, we use Equation 4, which is explained in subsection 3.3.2.

3.3 xStreams BackEnd and FrontEnd

The xStreams BackEnd adaptive learner and FrontEnd recommendation interface are decoupled, indirectly interacting modules. The back end slides a window of width W over the stream and maintains the perennial entities of seen users and products in a database. It maintains the learned user profiles up to date. These are used by the front end for the identification of the k most similar users to a given user u, for whom the algorithm formulates n recommendations.

Algorithm 1: xStreams_BackEnd

Input : stream of ratings R,
database of perennial entities D,
window length W in timepoints

```
1  U_act ← ∅
2  foreach timepoint t do
3  |  R_t ← ratings arrived in (t − w, t]
4  |  foreach user u who performed ratings in R_t do
5  |  |  Retrieve the state of the user's vector e_u^{t'} from D
6  |  |  Expand e_u^{t'} with the ratings of R_t that were
   |  |  performed by u into e_u^t
7  |  |  if user u is already in U_act then
8  |  |  |  replace the old vector of u in U_t with e_u^t
9  |  |  end
10 |  |  else insert (u, e_u^t) to U_act
11 |  end
12 |  Remove all users that are inactive in R_t from U_act
13 |  Write (t, U_act) to Output
14 end
```

3.3.1 xStreams_BackEnd

The pseudo-code of our adaptive learner is depicted in Algorithm 1. At each timepoint t, the algorithm processes all ratings arrived in $(t-w, t]$. These ratings constitute a set R_t (line 3), from which the active users U_{act} in the interval are extracted. For a user u that has performed ratings in R_t, her vector is fetched from the database D (line 5). The vector is modified to accommodate the current ratings (line 6), as described in Sec 3.1. For a user that already exist in U_{act}, the algorithm replaces the old vector with the newer one (line 8). The vectors of new users are simply inserted it into the list (line 9). All the users that have become inactive, i.e., they have no ratings in R_t, are removed from the list of active users (line 10). Since BackEnd processes a continuous stream, hence it contains no **return** operation. It rather writes the updated list of users to the Output (line 11) before processing the new timepoint (line 2).

3.3.2 xStreams_FrontEnd

The interactive FrontEnd of our recommender predicts the items and their ratings for a specific user u. The pseudo-code is given in Algorithm 2. Note that the timepoint t is the timepoint 'now', the moment at which user u is observed. It must be passed as parameter to the interface between FrontEnd and BackEnd, so that the correct similarity values are computed on the basis of the stream chunk R_t (cf. Algorithm 1).

To build the set $TopUsers_u^k$ (line 1), we compute the similarity between u and *all other* active users by using *both* $simCB()$ and $simCF()$. We use $simCB()$ of Equation 2 to compute the similarity of u to each user $u' \in U_{act}$ (line 11, Algorithm 1), where similarity refers to *aggregated* information on these users. We use $simCF()$ of Equation 3 to compute the similarity of u to active users on the non-aggregated individual ratings in R_t. We then compute for

$$\hat{r}_{u,j} = avg_u + \frac{\sum_{v \in TopUsers_u^k \wedge r_{v,j} NOTNULL} simTotal(t,u,v) * |r_{v,j} - \overline{r_j}|}{\sum_{v \in TopUsers_u^k} simTotal(t,u,v)} \quad (4)$$

Algorithm 2: xStreams_FrontEnd

Input : user u,
k neighbours,
n recommendations, timepoint t, and
corresponding set of ratings R_t
Output: list of recommended items RI_u for user u

1 Compute $TopUsers_u^k$, the set of k most similar users to u at timepoint t, using $simCF()$ and $simCB()$
2 $avg_u \leftarrow$ average rating value of u in R_t
3 $RI_u \leftarrow \emptyset$
4 **foreach** *item j in R_t so that $r_{u,j}$ is NULL* **do**
5 compute the average rating for j within R_t, $\overline{r_j}$
6 compute $\hat{r}_{u,j}$ using Equation 4
7 add $(j, \hat{r}_{u,j})$ to RI_u
8 **end**
9 Sort RI_u on $\hat{r}_{u,j}$ and retain only the top-n positions
10 return RI_u

each of the users in $CB_u(t) \cup CF_u(t)$ the value[4]

$$simTotal(t,u,v) =$$

$$weight * simCB(t,u,v) + (1 - weight) * simCF(u,v) \quad (5)$$

Thereafter, we sort the users on decreasing similarity and depict the top-k users, forming the set $TopUsers_u^k$.

If the *weight* of $simCB()$ is set to 1, then we ignore $simCF()$. This makes sense if the current data are so volatile that we cannot draw safe conclusions from the current behaviour of the users, and should concentrate on their past profiles. If the *weight* is less than 1, then we use $simCF()$ to exploit also the current user behaviour, as reflected in their ratings in R_t. Using this we estimate the rating $\hat{r}_{u,j}$ that u would have given to item j within the current time window. Let $\overline{r_j}$ be the avg. rating for item j in R_t, considering only items not yet rated by u (line 4). Further, let $r_{v,j}$ be the rating that user $v \in TopUsers_u^k$ has given to item j within R_t. Then, the predicted $\hat{r}_{u,j}$ (line 6) is computed as shown in Equation 4. The items with the estimated rating values are accommodated in a list (line 7), and the top-n rated items are returned for recommendation (lines 9-10).

The items and their estimated ratings are added to set RI_u (line 6), which is then sorted on rating value, retaining only the top-n positions (line 8). The items at these positions are recommended to the user u (line 9).

4. EVALUATION PLAN

Evaluation for recommenders on timestamped data is a challenging issue. Diaz-Aviles et al. [3] define a timepoint t_{split} and use all objects arriving prior to this timepoint for learning and all objects arriving afterwards for testing. This approach is based on the implicit assumption that past data are adequate to predict the whole future, i.e. no concept changes occur. However, stream mining is based on the assertion that changes *do* occur.

In contrast to [3], Nasraoui et al. [13] partition the data into batches, where each batch stores the data from a single user-profile: the batches are presented to the recommender one after the other, thus enforcing a concept shift at the end of each batch. However, the evaluation of [13] caters for concept change, but only for one particular change. For a proper evaluation, we need a less heuristic approach.

In this section we discuss some issues that are critical in order to successfully evaluate a recommender over data streams. We discuss the conventional stream evaluation, some works on streaming evaluation for recommenders and juxtapose them with the challenges that arise in evaluating a recommenders on streams.

Item recommendation can be modelled as classification problem. Stream classifiers are evaluated using *hold-out evaluation* or *prequential evaluation* [5]. In hold-out evaluation, a subset of the *newly arriving* objects is reserved for evaluation. This approach has the disadvantage of completely wasting the information carried by the held-out objects. In prequential evaluation, new objects are first labelled by the model, so that the quality of the model is evaluated; the objects with their true labels are then used for model learning. We study whether prequential evaluation can be used for stream recommenders and devise an appropriate evaluation plan.

4.1 Prequential Evaluation?

In Fig. 2, we depict the interplay of learning and evaluation for a stream recommender. At each timepoint, the recommender's model is updated by incorporating the new batch of *active* ratings, while the batch of *old* ratings is forgotten. Prequential evaluation is done on *future* ratings. The evaluation is subject to following challenges.

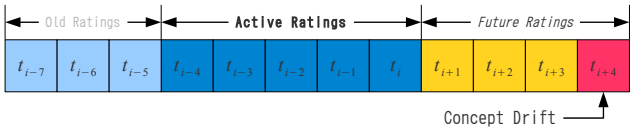


Figure 2: Data exploitation in a stream recommender: learning is done on the active ratings (those in the window), evaluation on the future ratings, which are gradually incorporated to the sliding window and used for learning.

Challenge 1 – the span of the future: In conventional prequential evaluation, all objects to be labelled appear in the very next batch, i.e., at timepoint t_{i+1}. In recommenders,

[4]This value might be perceived as the result of a similarity function $sim()$. However, this operation can be computed only over the set $CB_u(t) \cup CF_u(t)$ for some user u, since these sets are computed independently.

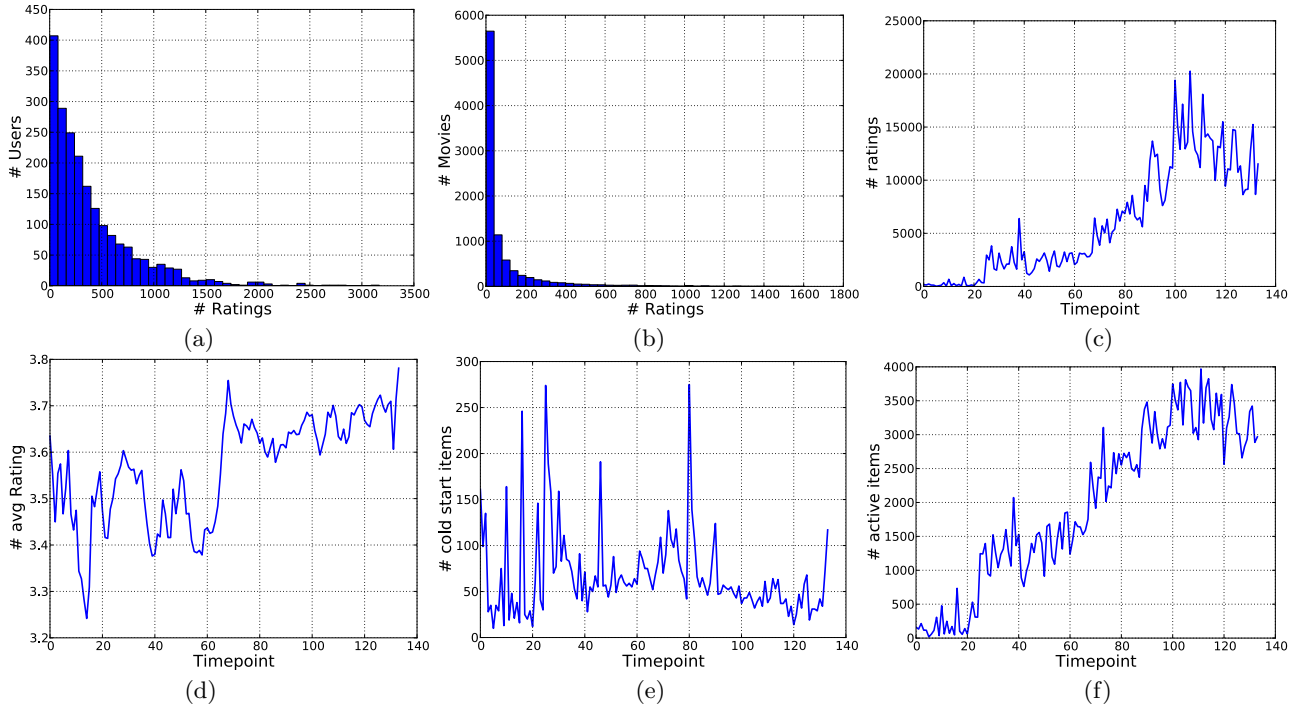


Figure 3: Statistics for MovieLens, recorded at each timepoint: from top: l-to-r, (a) #ratings per user, (b) #ratings per movie, and for each timepoint (c) #ratings, (d) avg. rating, (e) # cold start items and (f) # rated items.

the users in the current batch may not appear in the batch of t_{i+1}. For example, assume that the recommender suggests item x to user u. u indeed rates x favourably, but does so at $t' \gg t_{i+1}$. Prequential evaluation will count a miss. A naive way to alleviate this caveat is to consider all future ratings for evaluation. This leads to the second challenge.

Challenge 2 – the drift in the future: Assume that we evaluate on all future ratings. At timepoint t_{i+1}, let item x be recommended to user u, who rates it negatively at t_{i+4} (a miss). At t_{i+2} the model adapts to drift and learns that u would dislike everything like x. So, after t_{i+2}, the model would not recommend x to u, yet the recommendation is already done and counted. Hence, we must limit the horizon of the prequential evaluation to a window of predefined size.

Challenge 3 – the one-time users: In social and commercial sites, many users appear rarely or even only once. In the MovieLens dataset (Section 5.1), 300 users (ca. 14% of all users) appear only once. Prequential evaluation demands a recommendation for each user seen at t_i, but the outcome cannot be verified since some users might not show up again.

Challenge 4 – the casual users: Many users re-appear irregularly at timepoints that are far apart. When we discretized the MovieLens dataset into 140 months, we identified 400 users (ca. 19%) who appear at less than 5 timepoints that are very far apart. If we use an evaluation horizon that ends before the user's next re-appearance, then prequential evaluation cannot categorize the outcome as hit or miss (similarly to *Challenge 3*). If we specify a huge evaluation horizon to capture the next re-appearance of such users, then we would provoke again *Challenge 2*.

4.2 Prequential Evaluation with Hold-Outs

In the light of the above challenges, we propose a hybrid method that sets apart a $split_{test}$ portion of the ratings in each incoming batch for hold-out evaluation, and performs prequential evaluation on the remaining ratings. In particular, if $split_{test} > 0$, then we use $1 - split_{test}$ of the ratings first for evaluation and then for learning (i.e. for prequential evaluation), and $split_{test}$ only for evaluation. If $split_{test} = 0$, we use all data for prequential evaluation only. Based on preliminary experiments, we have used $split_{test} = 0.5$.

5. EXPERIMENTAL EVALUATION

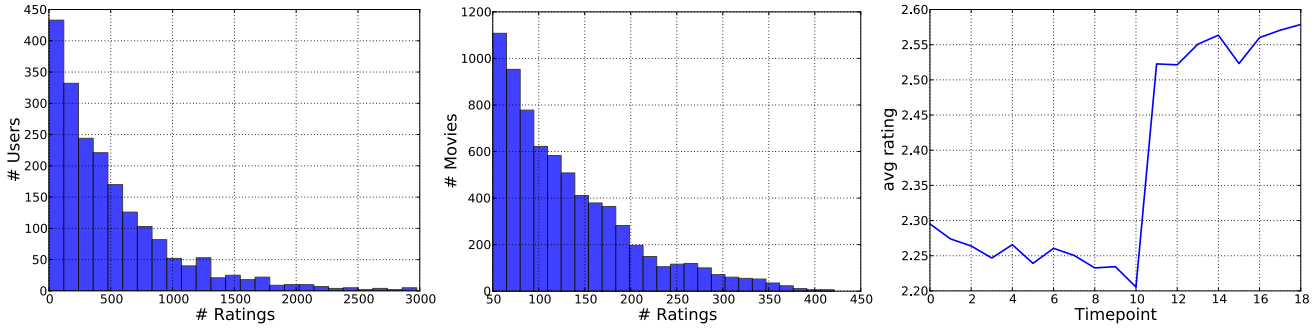
We use the evaluation plan proposed in 4.2 to study the performance of our method. We compare xStreams to a stream-based extension of the collaborative filtering (CF) algorithm of [23], which we call CFstream. Since we evaluate on recommendations to users, we adjust the conventional evaluation measures for recommenders as follows: for a test user who receives a list of n recommended items, **Precision** is the ratio of the hits (favourably rated items) among the n ones; **Recall** is the ratio of hits from the top-n list to the complete set of items rated favourably by the user; **RMSE** is the Root Mean Square Error between predicted and true rating for the test user.

5.1 Real and Synthetic Datasets

5.1.1 Real Dataset

We have used MovieLens[5] dataset. It contains 2113 users, 10,197 movies and approximately 0.85 million ratings. Additionally to the user-item rating matrix, the dataset includes information on each movie's genre, directors, actors and user

[5]GroupLens website: http://grouplens.org/datasets/hetrec-2011/; Dataset: hetrec2011-movielens-2k.zip



(a) (b) (c)

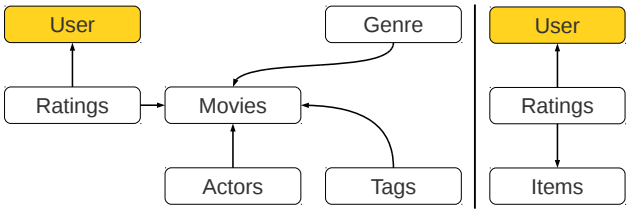
Figure 4: Statistics for synthetic dataset: (a) number of ratings vs. user, (b) number of ratings vs. movies, (c) average rating value vs. timepoint.

tags. This auxiliary information is provided as separate streams/tables. We have created a multi-stream by using the schema of Figure 5(a), whereby we grouped actors into 6 categories, by ranking them on importance of their role in a movie multiplied by the average rating of those movie. Summing up these values over all the movies an actor was involved gives us an *AScore* for an actor:

$$AScore(a) = \sum_{m \in movies(a)} (250 - Rank(a, m)) * MScore(m)$$

The six categories are based on *AScore* where, C1 accommodates first 18 actors, C2 the next 50, C3 the next 200, C4 the next 500, C5 the next 5000 and C6 all the rest.

In Figure 3 we show statistics on MovieLens. Movielens dataset follows zipf distribution for both the number of ratings provided by users and the number of ratings given to a particular movie, i.e., there is a small number of users who have rated many items (short head) and many users that have only rated a small number of items (long tail). There are only few ratings before t_{30} (Figure 3c). The bulk of rating starts arriving after t_{70}. The average rating value increases continuously with a sharp increase (concept shift) at t_{80} (Figure 3d). Around t_{80}, there also is a large influx of new cold-start items (Figure 3e). We show later that the concept shifts and the cold-start items affect the performance negatively, while the increase in #ratings and of rated items have a positive effect.



(a) (b)

Figure 5: Schema of (a) MovieLens Dataset, and (b) Synthetic Dataset.

5.1.2 *Synthetic Dataset*

To study how xStreams responds to concept changes, we build a synthetic dataset with predefined moments of drift. We use the data generator of [22] that creates streams of users, items and ratings. The schema of the generated relational data is shown in Figure 5(b). First, item descriptors and user profiles are generated as vectors. They are cluster centroids, so that items and users are generated as vectors proximal to some centroid. A user profile *u_p* is far or proximal to an item descriptor *i_d*, and thus dictates (with some variance) the rating generated by some user who adheres to *u_d* for some item that adheres to *i_d*. A user may change from one profile to another with some likelihood, since people's preferences change with time.

In our experiments with the synthetic dataset we incorporated concept drift into data. Our generator is capable of simulating the properties of real world datasets as shown in Figures 4(a) and 4(b), which they both follow the Zipf distribution similar to the results shown in Figures 3(a) and 3(b), respectively. Moreover, in Figure 4(c) we have incorporated explicit concept drift into our synthetic data. It is a shift of average rating that happens around timepoint 8, where the mean rating value jumps from around 2,25 stars to 2.55 stars. This concept drift forces the users to change their rating behaviour abruptly around timepoint 8. As will be later experimentally shown, our xStreams algorithms is able to adapt fast to the concept drift.

This generator simulates the properties of datasets like MovieLens: users, items and ratings follow power law distributions and overtime, users change their rating preferences.

5.2 Experiments on Synthetic Data

We first study the performance of xStreams for various sizes of the sliding window: $w = 2, 4, 8$. We set xStreams to return the top-n recommendations for $n = 7$ and to consider the $k = 7$ nearest neighbors (cf. Algorithm 2). We use only $simCB()$, i.e. set $weight = 1.0$ in Eq. 4 and $split_{test} = 0.0$.

In Figure 6(a) & (b), we see that the smallest window size $w = 2$ achieves the best RMSE curve: the values are low and the recovery after the concept shift at t_8 is fast. This indicates that forgetting old data soon is best if the data exhibit drifts. We have also run this experiment for Precision, Recall and F-Measure and also for $k = 30$; we observed the same trend.

In Figure 6(c) & (d), we compare xStreams to CFstream. For xStreams, settings are same except the window is set to w=2 and varying the *weight* of $simCB()$ in Equation 4, assigning the values $1.0, 0.67, 0.34$. xStreams outperforms CFstream and exhibits lowest RMSE when the weight of

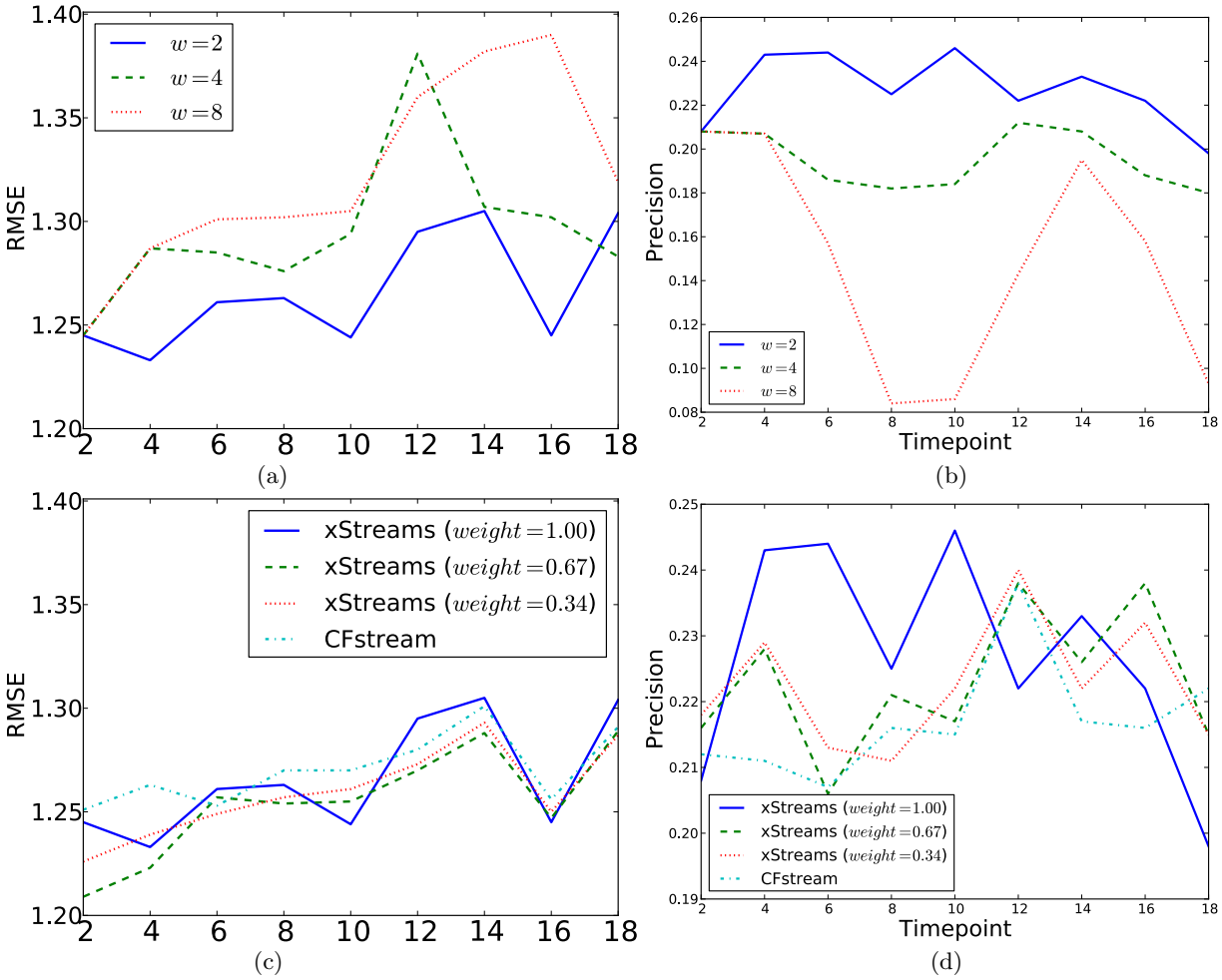


Figure 6: RMSE values for xStreams (a-b) under different windows sizes $w = 2, 4, 8$ ($simCB() = 1.0$) and (c-d) when Comparing xStreams to CFstream ($w = 2$), for $k = 7$ nearest neighbours and returning $n = 7$ recommendations; lower values are better.

$simCB()$ in the ratings is 0.67; this means that both the similarity of users on accumulated past data, as captured by $simCB()$, and the conventional similarity on the ratings should be taken into account. However, CFstream that only considers the conventional similarity has lower performance. When evaluating on F-measure, Precision and Recall (not shown), xStreams also outperformed CFstream; for Precision and Recall, the values for $weight = 1.0$ were better than for $weight$=0.67 in the first half of the timepoints but deteriorated thereafter; for $weight$=0.67, the performance was more stable.

5.3 Experiments on Real Data

We evaluate xStreams to CFstream on MovieLens. Over the 140 timepoints (months) of the MovieLens data, we slide a window w of length 12.

We show the curves on Precision and RMSE for $w = 12$, $n = 2$, and $k = 100$ in Figure 7 (a) & (b), respectively. The evaluation measures illustrate nicely how different parameter settings respond to these two counteracting measures. All the strategies had a low recall, which was around 0.04 with little variance among, therefore we omit their graphs. The algorithm allows for the exploitation of more past data because of the larger window size ($w = 12$). We have set the number of neighbours $k = 100$ and used a $split_{test} = 0.5$, i.e. half of the arriving ratings are held-out for evaluation. We start with the delivery of recommendations at t_{30}, because the numbers of ratings and rated items are very low in the first timepoints, while the number of cold-start items is very high (see Figures 3(a), (d) and (c), respectively).

It is obvious that precision (and also recall) values depend strongly on the number of recommendations. Thus, if we increase the n parameter, precision will fall (and recall will increase). Comparing the curves for the two n values, we observe that the setting $n = 2$ and $n = 7$ (see Figure 7 (d) & (d)) leads to lower performance. Returning the top-2 recommendations is more challenging, but we expected that the large window size and the use of $simCB()$ would have partially compensated it. The precision drops faster for $n = 7$ (Figure 7(a) vs (c)). A further reason for the low performance is the use of $split_{test} = 0.5$: setting aside half of the data for evaluation has a stronger negative impact in the larger window.

Among the different variants of xStream and the CF-Stream there is no clear winner. Rather, each strategy out-

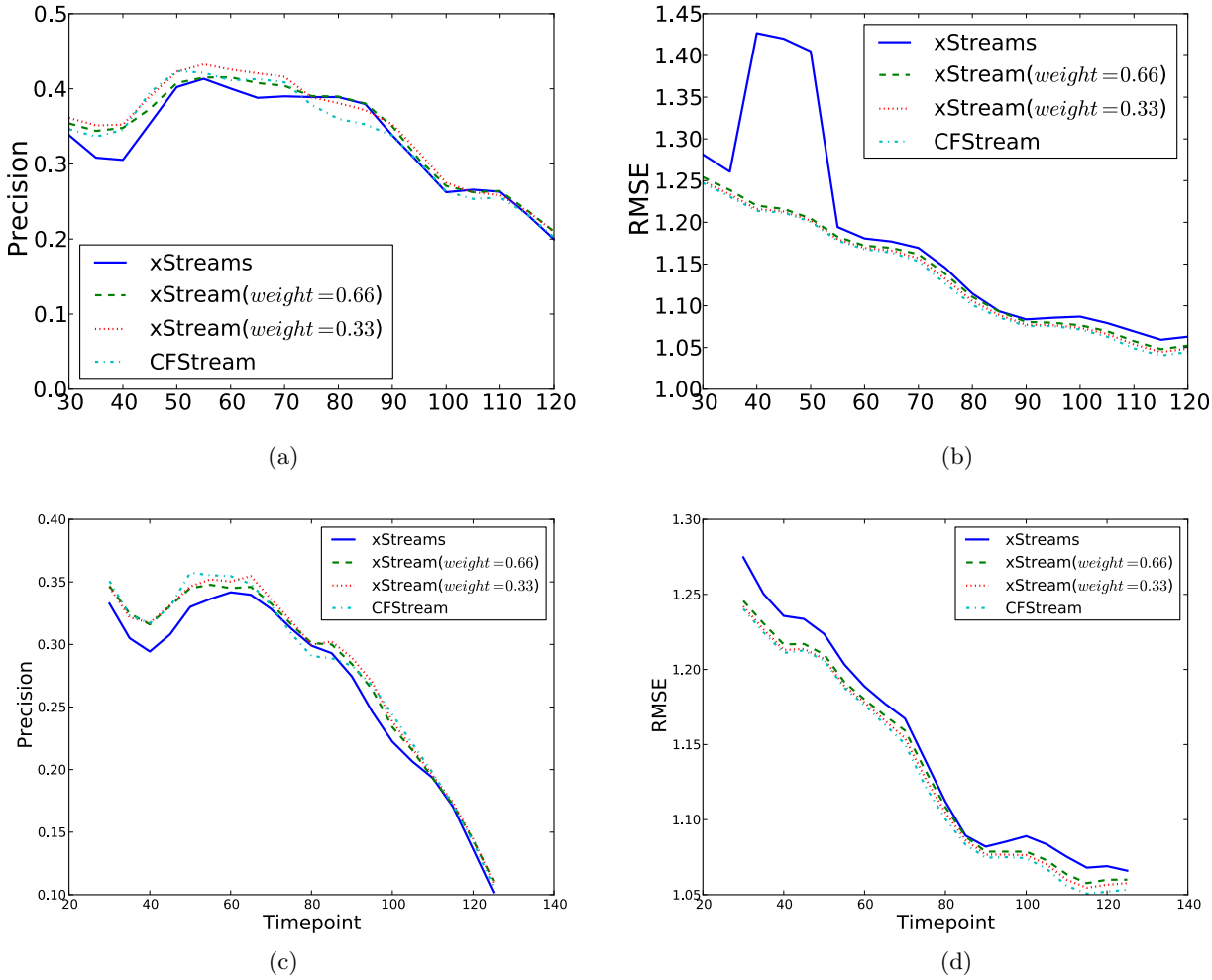


Figure 7: Comparison of xSreams & CFStream: from top l-to-r: Precision and RMSE for (a-b) $n = 2, w = 12, k = 100$**, (c-d)** $n = 7, w = 12, k = 100$**.**

performs the others at different moments. CFStream performs comparably to xStreams in the first moments, while the variants of xStreams with a weight of 67% or 0.33% for $simCB()$ show consistently better performance after $t60$, i.e. they cope well with the influx of cold start items at $t80$ (Figure 3c). Since CFStream corresponds to a weight of 0, and since xStreams with $weight = 1.0$ shows lower precision/recall values, we conclude that it is necessary to combine the similarity among recent ratings (done by $simCF()$) with the similarity of the profiles incorporating the users' past behaviour (as done by $simCB()$).

6. CONCLUSION

We have presented a stream-based recommendation method that learns and adapts to the user preferences, as these preferences evolve over time. Preferences are reflected in the ratings that users give to items; we accumulate them to user profiles, and use stream clustering to build the profiles and adjust them to change. This gives the basis for a more elaborate notion of similarity: instead of a static similarity between users, we have a dynamic similarity between user profiles. To alleviate possible negative effects caused by very large profiles (sparse clusters with large radius), we also restrict similarity further by requiring proximity within the profile. Hence, two users are similar at some moment, if they have the same profile at this moment and are proximal within the profile.

We have studied our approach on a synthetic and a real dataset. The synthetic dataset has been used to depict the effects of change in a controlled, transparent way. For this dataset, we have shown that our method experiences a performance deterioration after the imputed change, whereupon it replaces gradually but swiftly the outdated profiles with new ones, and recovers fast. The real dataset MovieLens exhibits less drastic forms of change, whereupon our approach also shows smoother performance. Strategies with larger window size achieve higher precision and average RMSE (which is apparently dominated by precision) at the cost of low recall. The best balance between precision and recall is achieved by a strategy with small window size, i.e. one that maintains little profile information and replaces the profiles as soon as performance deteriorates.

An intriguing finding was the discrepancy between precision and recall for all strategies, i.e. for both our stream-

based strategies with different window sizes and for the change-insensitive baseline. A possible explanation, which requires further investigation though, is the power law that governs the data distribution. The long tail shown in Figure 3 implies that little information is available for most of the users: perhaps, high precision is achieved by learning a lot about the few users in the short head, while the low recall is caused by the long tail. This seems to be supported by the fact that best recall is achieved by the strategy with the smallest window size, which exploits little information about the users and forgets learned profiles fast. A deeper study of this issue is our next planed task.

Related to the above is the future study of the implications of power law on the similarity among users: there are many users with many ratings, but most users have few ratings on mostly different items. This makes the computation of similarity more complicated. We want to investigate elaborate similarity measures for such users. This will allow us to tackle the problem of formulating recommendations for the many users in the long tail, whose preferences evolve no less than those of the few users in the short head.

Acknowledgements

Part of this work was funded by the German Research Foundation project SP 572/11-1 "IMPRINT: Incremental Mining for Perennial Objects".

7. REFERENCES

[1] M. Balabanovic and S. Y. Fab: Content-based, collaborative recommendation. *ACM Communications*, 40(3):66–72, 1997.

[2] M. B. Dias, D. Locher, M. Li, W. El-Deredy, and P. J. Lisboa. The value of personalised recommender systems to e-business: a case study. In *RecSys '08*, pages 291–294, 2008.

[3] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl. Real-time top-n recommendation in social streams. In *RecSys 2012*, 2012.

[4] Y. Ding and X. Li. Time weight collaborative filtering. In *CIKM '05*, pages 485–492, 2005.

[5] J. Gama, R. Sebastião, and P. P. Rodrigues. Issues in evaluation of stream learning algorithms. In *KDD '09*, pages 329–338. ACM, 2009.

[6] J. Herlocker, J. A. Konstan, and J. Riedl. An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. *Information Retrieval*, 5(4):287–310, Oct. 2002.

[7] Y. Koren. Collaborative filtering with temporal dynamics. In *KDD 2009*, pages 447–456, 2009.

[8] Y. Koren. Collaborative filtering with temporal dynamics. *Communications of ACM*, 53(4):89–97, 2010.

[9] Y. Koren and R. Bell. Advances in collaborative filtering. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors, *Recommender Systems Handbook*, chapter 5, pages 145–186. Springer Science+Business Media, 2011.

[10] M. R. McLaughlin and J. L. Herlocker. A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *SIGIR 04*, pages 329–336. ACM, 2004.

[11] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *CDL*, pages 195–204. ACM, 2000.

[12] O. Nasraoui, C. Cardona-Uribe, and C. Rojas-Coronel. Tecno-Streams: Tracking evolving clusters in noisy data streams with an scalable immune system learning method. In *Proc. IEEE Int. Conf. on Data Mining (ICDM'03)*, Melbourne, Australia, 2003.

[13] O. Nasraoui, J. Cerwinske, C. Rojas, and F. Gonzalez. Performance of recommendation systems in dynamic streaming environments. In *Proceedings of the SIAM International Conference on Data Mining (SDM 2007)*, 2007.

[14] M. J. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *ML 97*, 27(3):313–331, 1997.

[15] A. Prieditis and S. J. Russell, editors. *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, USA, July 9-12, 1995*. Morgan Kaufmann, 1995.

[16] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering on netnews. In *Proceedings of the Computer Supported Collaborative Work Conference*, pages 175–186, 1994.

[17] J. Salter and N. Antonopoulos. Cinemascreen recommender agent: Combining collaborative and content-based filtering. *Intelligent Systems Magazine*, 21(1):35–41, 2006.

[18] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proc. WWW Conf.*, pages 285–295, 2001.

[19] Z. F. Siddiqui and M. Spiliopoulou. Combining multiple interrelated streams for incremental clustering. In *SSDBM 09*, 2009.

[20] Z. F. Siddiqui and M. Spiliopoulou. Stream clustering of growing objects. In *DS 09*, 2009.

[21] Z. F. Siddiqui and M. Spiliopoulou. Tree induction over perennial objects. In *SSDBM 2010*, pages 640–657. Springer-Verlag, 2010.

[22] Z. F. Siddiqui, E. Tiakas, M. Spiliopoulou, and P. Symeonidis. A data generator for multi-stream data. In *MUSE workshop held with ECML PKDD 2011*, 2011.

[23] P. Symeonidis, E. Tiakas, and Y. Manolopoulos. Product recommendation and rating prediction based on multi-modal social networks. In *RecSys 2011*, pages 61–68, 2011.