


AMORE: design and implementation of a commercial-strength parallel hybrid movie recommendation engine

Ioannis T. Christou¹  · Emmanouil Amolochitis^{1,2} · Zheng-Hua Tan³

Received: 21 February 2014 / Revised: 26 May 2015 / Accepted: 21 July 2015 /
Published online: 1 August 2015
© Springer-Verlag London 2015

Abstract AMORE is a hybrid recommendation system that provides movie recommendation functionality to video-on-demand subscribers of a major triple-play service provider in Greece. Without any user relevance feedback for movies available, all recommendations are solely based on the users' viewing history. To overcome such limitations as well as the extra problem of user histories that are usually the merger of the preferences of all persons in each household, we have performed extensive experiments with open-source recommendation software such as Apache Mahout and Lens-Kit, as well as with our own implementations of several user-based, item-based, and content-based recommendation algorithms. Our results indicate that our own custom multi-threaded implementation of collaborative filtering combined with a custom content-based algorithm outperforms current state-of-the-art implementations of similar algorithms both in solution quality and in response time by margins exceeding 100 % in terms of recall quality and 6300 % in terms of running time. The hybrid nature of the ensemble allows the system to perform well and to overcome inherent limitations of collaborative filtering, such as various cold-start problems. AMORE has been deployed in a production environment where it has contributed to an increase in the provider's rental profits, while at the same time offers customer retention support.

Keywords Recommender systems · Pattern recognition · Information search and retrieval · recommender ensembles

✉ Ioannis T. Christou
ichr@ait.edu.gr

Emmanouil Amolochitis
emam@ait.edu.gr

Zheng-Hua Tan
zt@es.aau.dk

¹ Athens Information Technology, 44 Kifisias Ave, Monumental Plaza, Bld. C, Marousi 15125, Greece

² CTiF, Aalborg University, Niels Jernes Vej 12, 9220 Aalborg, Denmark

³ Department of Electronic Systems, Aalborg University, Niels Jernes Vej 12, 9220 Aalborg, Denmark

1 Introduction

Recommendation systems have gained widespread popularity in recent years and are considered to have reached sufficient maturity as a technology [13,20]. The research performed in this particular field has started more than 20 years ago ([10,22] etc.), and it focuses on examining different ways that recommendation systems can better identify user interests and preferences based on the knowledge of the users' behavior as well as on the characteristics of the items that they have consumed. In order to achieve this, several approaches have been established; nearest-neighbor-based approaches were among the first to be explored with good results. Both user-based and item-based neighborhood exploration strategies met huge early success (for the first, the name "collaborative filtering" was coined early in the 1990s). User-based recommender algorithms attempt to identify user groups with common characteristics in order to promote to specific user content items that they have not yet consumed, but other similar users within the group did (or more generally, expressed an interest for). On the other hand, item-based recommender algorithms were first explored because of scalability concerns: Establishing user neighborhoods can become very expensive when the number of users increases in the order of several millions, whereas the number of distinct items a particular e-commerce site is expected to promote and sell is not usually expected to increase beyond a few tens of thousands (even though Amazon.com and a few other e-commerce giants carry hundreds of millions of items!). Item-based recommenders then usually establish for each item a "neighborhood" of similar items; then, when the system is asked to recommend to a user some items, it computes the most similar items to the user's entire history of purchases that have not already been purchased by the user and recommends them to the user. Often, similarity between two items is established by looking at how often both the two items appear in the purchase history of the users (i.e., items are represented as vectors in a very high-dimensional user space, and the cosine similarity formula or some other function inspired from information retrieval vector-space models is used to compute the similarity between them).

Another approach to recommendation systems is the so-called content-based approach [19]: The system recommends an item to a user based on the degree to which the item's content (usually in the form of meta-data, e.g., in the context of recommending movies, the film's genres, the film's crew including actors, directors, and producers.) matches the user's purchase history. One major advantage of content-based approaches is that a new item that has not been purchased by anyone yet can still be recommended to those users whose purchase history matches the item's content meta-data, thus solving one type of cold-start problems.

Recommending content to video-on-demand service subscribers is a challenging problem that is often made much more so in the absence of any user relevance judgments and even worse when the same user account is used by multiple persons in the same household, thus resulting in a purchase/viewing history that is the union of the transactions made by different people. To address the first issue, it is some times, but not always, possible to use the average session length of a particular video by a user to infer indirectly the degree of interest of the user for the particular item [5]. On the other hand, [25] detected a low but significant inverse correlation between the percentage of a movie watched by users and its popularity. We discuss several experiments we have conducted to address both issues in the later sections in this paper.

1.1 Related work

The literature in the field of recommender systems is so vast that the field has its own conference (ACM *Conference on Recommender Systems*). Therefore, in this section, we only mention those few papers that have either exerted the most influence in our own work, or have otherwise found them to have a lot in common with our work.

A recurring problem in many production settings of recommender systems is the fact that popular items usually appear on top of recommendation result lists placing less popular items (which might have a higher degree of relevance to the interests of a user) at lower ranks. This behavior is especially evident in the recommendation functionality of YouTube [3], as well as general-purpose search engines that may promote popular pages of less relevance to the expense of a relevant, but less popular, page [4]. User-based and item-based collaborative filtering approaches attempt to minimize this effect by using special formulae that promote less popular items when computing the user or item neighborhoods (see [14]).

Being able to evaluate the performance of recommenders is also a very demanding task. As the authors in Herlocker et al. [11] suggest, different recommenders perform better or worse based on the different datasets used or differently structured datasets. Shani and Gunawardana [21] present a property-directed evaluation of recommendation systems attempting to explain how recommenders can be ranked with respect to these properties (diversity of recommendations, scalability, robustness etc). In their work, they rank recommenders based on specific properties under the assumption that an improved handling of the property at focus will improve the overall user experience.

Li et al. [16] introduce a method that uses collaborative filtering approaches in e-commerce based on both users and items alike. They also show that collaborative filtering based on users is not successfully adaptive to datasets of users with different interests. Collaborative filtering algorithms have already been used in different implementations of movie recommendation systems: [9] present FilmTrust a system that examines a way to generate movie recommendations by combining information such as a user's Semantic Web social network along with trust knowledge about the user's peers in the network.

In Christou et al. [5], the authors present a system that uses a content-based recommendation approach (without using any collaborative filtering algorithms) in order to address the problem of finding interesting TV programs for users without requiring previous explicit profile setup, but by applying continuous profile adaptation via classifier ensembles trained on sliding time windows to avoid topic drift. Similarly, Pazzani et al. [19] focus on content-based recommenders and review different classification algorithms based on the idea that certain algorithms perform better when having specific data representation. The algorithms are used to build models for specific users based on both explicit information submitted by users and by relevance judgments submitted by them.

Mild et al. [18] claim that the number of users available affects the choice of the category of recommenders that should be used. Also, they present their findings showing that for a large dataset, linear regression with simple model selection provides improved results compared to collaborative filtering algorithms.

Finally, Li et al. [17] present a novel one-class collaborative filtering recommender system that utilizes rich user information other than user ratings which they assume to be unavailable and show that such rich user information can significantly enhance recommendation accuracy. Their basic hypothesis is a perfect fit for our case as well, where our dataset includes no ratings at all; unfortunately, however, in our case, we do not have any other historical data (e.g., search queries, or simple demographic information) about our users either.

1.2 Our contribution

1.2.1 Addressing real-world complications

In this work, we address the issues mentioned above (multiple users associated with a single account, no user relevance judgments of seen items, cold-start issues) as well as issues of robustness and efficiency. Other issues that a real-world recommender system has to deal with include the *continuously changing database of available content* to subscribers: The content a triple-play service provider makes available to their subscribers is based on various limited-time deals they make with content producers. In our case, the traffic of content items for a particular time period is shown in Fig. 1 that reveals how the available content for the subscribers to view is in constant flux.

This fact makes the recommendation task much more difficult because items that a user would appreciate may not be available for viewing. In fact, the system has to also post-filter various other types of recommendations (such as “adult” movies) that might have otherwise been recommended by the system, and it also has to maintain a strong variety in its recommendations as a function of time, so that the user does not constantly see the same recommendations for more than 2 weeks in a row.

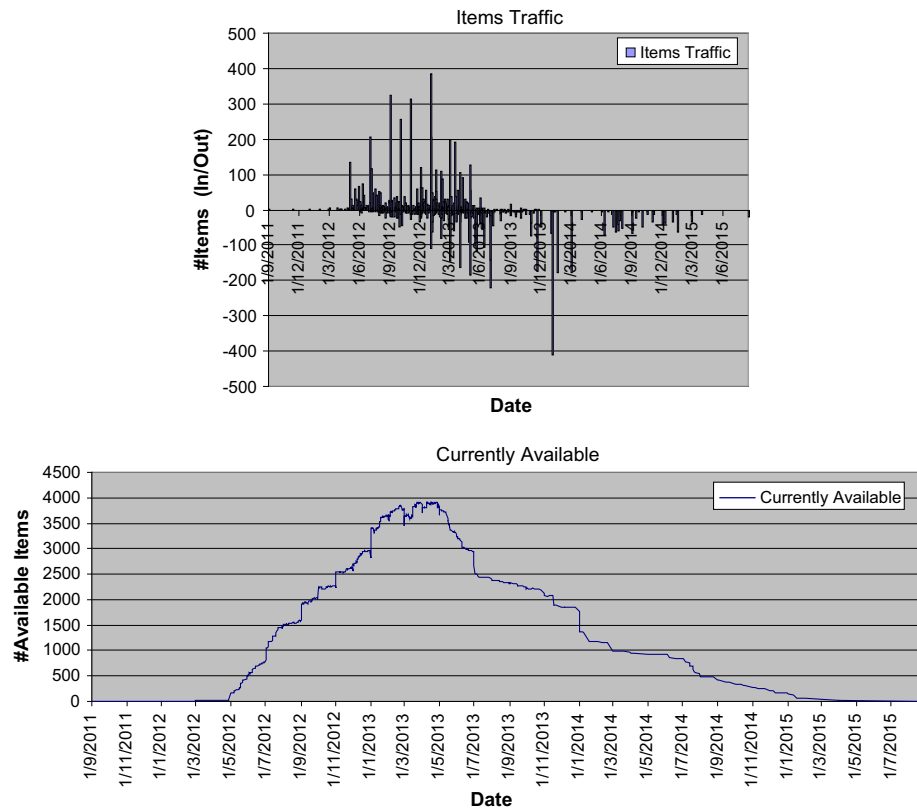


Fig. 1 a Fluctuation of available content by date captured on April 2013: *positive bars* indicate the number of content items made available since that day, *negative bars* indicate the number of content items expired that day. b Total item availability on April 2013. This chart is derived directly from the data in a

Even more importantly, the items available for viewing are not offered at the same constant price: Items that recently played in theaters are usually offered at a premium price, while older items are usually offered either for free or for a very small price. In our case, there are *nine* distinct price levels for content items, in the interval [€0, €7.99]. Pricing of course plays a major role in the viewing decisions of subscribers, and in our case, the vast majority of items viewed belongs to the “free items” category, showing that *demand for items is most of the time, for most of the users, very elastic* (as related evidence, the histories of many users often include the same, usually free, item purchased multiple times!). And to further complicate matters, a business deal with a pizza restaurant chain dictates that anyone purchasing a movie at a nonzero price is entitled to a 50% discount for pizza delivery at their house from that particular pizza chain for the duration of the movie; this fact has led many people to purchase a movie for the cheapest possible price (which is €0.01) without being actually interested in the item, simply for the 50% discount in pizza offered by the company. Unfortunately, such offers add significant further *noise* into the purchase histories of the users based on which the recommender engine has to provide high-quality recommendations to the users. On top of all that, then, as an extra marketing requirement, the system has to maintain at least a certain percentage of non-free items in its top 10 recommendations.

Finally, the recommendation engine needs to be able to instantly respond to recommendation requests at any given point in time and at the same time be able to renew user recommendations as frequently as possible. The aforementioned requirement becomes even more difficult to tackle with, when considering the fact that our system was meant to be deployed in a production environment with limited hardware resources.

Our contribution involves the design and implementation of a hybrid movie recommendation system that address all aforementioned issues. We have designed, implemented, and successfully deployed live in production, a system that uses an ensemble of *user-*, *item-*, and *content-*based recommenders. Specifically, the ensemble consists of a k -NN-user-based collaborative filtering algorithm, a k -NN-item-based recommender as well as a third, custom content-based recommender; all algorithms are multi-threaded and take full advantage of the many cores available in all server hardware today. The hybrid configuration of the ensemble allows the system to be able to use content-based information in terms of content item metadata, as well as user- and item-based collaborative filtering techniques which combined together, boost the overall system performance.

At a second level, in order to address the issue of having different users bound to a specific account and in order to be able to offer recommendations with a higher probability of being interest to the specific user, that is, currently requesting recommendations, we provide special features that allow recommendations to be generated taking into consideration only the watching history of user accounts during *specific time windows*. By doing so, we are narrowing down the recommendation results based on a specific watching behavior, in terms of watching hours within a day for different categories of users.

Finally, in order for the system to be able to instantly reply to movie recommendation requests and at the same time renew the user recommendations as frequently as possible, we have chosen an architectural design that contains two schemas following the exact same data model. Our architecture allows the system to serve recommendation requests by computing *on-the-fly* or retrieving *cached recommendations from one schema*, while at the same time the system updates user histories, item catalogs, and the cached recommendations according to the latest user watching histories on the auxiliary schema. When the caching process completes, the system switches the data source reference from the main schema to the auxiliary in order to continue serving recommendation requests from the schema containing the most recent user recommendations and, at the same time, renews the cached recommendations

stored in the schema containing the most outdated recommendations. This process allows the system to have updated recommendations on an hourly basis by using limited hardware resources in an extremely cost-effective way.

2 System architecture

The AMORE system is designed as a SOA (service-oriented architecture) in such a way so as to be as independent as possible from all other, external components that are part of the movie rental platform. From the triple-play service provider's point of view, AMORE is treated as a *black box*; it receives recommendation web service requests and provides responses, without exposing any implementation details. Similarly, the recommendation system retrieves via exposed services only the minimum amount of user and content item information that is deemed necessary for the recommendation process.

The AMORE system has been designed in such a way that it performs a pre-caching of all latest recommendation results generated for all active users in order to speed up system responsiveness to the client invocations. This database-caching of results is implemented in a batch job (called AMORE batch job) that performs a number of tasks for the "off-line" updating and caching of the recommendation results. To facilitate the uninterrupted running of both—the web service and the batch job in parallel—the system uses two databases (schemas) that we will refer to as the *original* and the *alternative* that follow the exact same data model. Having two schemas allows our system to serve web service requests by retrieving cached results from one schema (for instance, the *original*) while the batch job uses the second schema (in this case, the *alternative*) to store the updated results.

As a business rule, the system treats the "new user" cold-start problem by returning the top- n recommendations for all users, computed by lexicographically ordering the vectors associated with each item that have as their i -th component ($i = 1, 2, \dots, n$) the number of times the item appeared in the i -th position in the top- n recommendations list for some user. As a second business rule, to add *diversity* to the recommendations made to the subscribers and to make them more interesting to them, a post-processing mechanism has been implemented that gradually discounts the value of a recommended item in the list of recommendations to a user for every time the user sees the recommendation but does not decide to proceed to a purchase, thus causing recommendations that linger on the user's screen for too long eventually "fall off the charts" (see [15] for a detailed evaluation of methods for solving exactly this kind of *temporal diversity issue in recommender systems*; see also [12] for a thorough review of different approaches to the related problem of maximizing diversity to item recommendations to users; [26] formulate an optimization problem for maximizing diversity in recommendation lists subject to maintaining high relevance of the recommended items).

Before the batch job instance completes its operation, having populated the database with the newly generated recommendations, the job calls a special web service method that instructs the AMORE web service to switch the active data source from the current schema to the alternative; after doing so, the web service returns cached recommendation results by retrieving data from the schema that contains the most updated, recent results. Similarly, the AMORE job updates its data source reference in order to perform the next batch run using the schema containing the most outdated data. In order to make sure that the data retrieved every time from persistence are consistent with the currently active data source, we have implemented a mechanism that ensures and protects the system resources from "dirty"

reads/writes. This mechanism is based on a *fast reentrant global read/write-lock* with the following properties:

- A thread owning the write-lock may request (and gets) the same lock in read- or write-mode any number of times, but must call the corresponding release method for every time it has called the request method in order for the locks to be eventually released
- A thread owning a read-lock may request an upgrade to the write-lock, and the method will grant the new type lock, unless at the time of request is at least one other thread having the read-lock, in which case there is a danger of deadlock; in such a situation, a checked exception is thrown
- Threads executing a request for a read-lock will yield the *first* time if there exists a thread waiting for the write-lock so as to avoid any possible live-lock issues

Given this global lock, we implement a simple pattern in all related methods for creating, maintaining, and/or updating the in-memory caches: Whenever a method needs to access the in-memory caches, it must first obtain the global read-lock, whereas methods that need to update the in-memory caches must first obtain the global write-lock. Upon start-up of the AMORE web service, the first thread started, spawns a new thread that obtains the global write-lock and starts loading the data from the database into the in-memory caches, while the first thread waits for the new thread to complete (calling the thread's *join()* method). Once the new thread has loaded the latest snapshot of the database, it releases the write-lock and finishes, returning control to the first thread to continue its operation. Coordination between the AMORE batch job and the AMORE web service (two distinct processes residing in distinct address spaces) is obtained as follows: When the AMORE batch job is about to complete, as a last step, it calls the special AMORE web service method mentioned above, which in turn first obtains the global write-lock of the system, then switches the db pointer to the current active db schema, then refreshes all in-memory caches of the system, and finally, releases the global write-lock, allowing pending recommendation requests (waiting to obtain the global read-lock) to proceed using the most updated data.

Figure 2 provides a visual representation of the overall system architecture, as discussed above.

This configuration allows vice to reply instantly by performing the minimum number of operations for most operations using the cached recommendations, thus being able to serve a large number of concurrent requests very fast and with the lowest possible processing overhead.

3 Recommender ensembles

3.1 Mahout-based initial ensemble

Our first implementation of the AMORE system revolved around two of Apache Mahout's most used recommenders: the Boolean log-likelihood-based recommender and the Boolean Tanimoto-based recommender [24]. Boolean recommenders require only a user's transaction history recorded as a list of pairs (*user ID*, *item ID*), indicating the items a user has purchased without any further rating information, and in our case, this is the only information available to us regarding the interests of the users. Mahout's Boolean recommender algorithm first retrieves the user's entire user neighborhood that corresponds to the collection of users with the strongest degree of similarity with respect to the common items consumed. Different methods for the computation of the degree of similarity between any two users give rise to

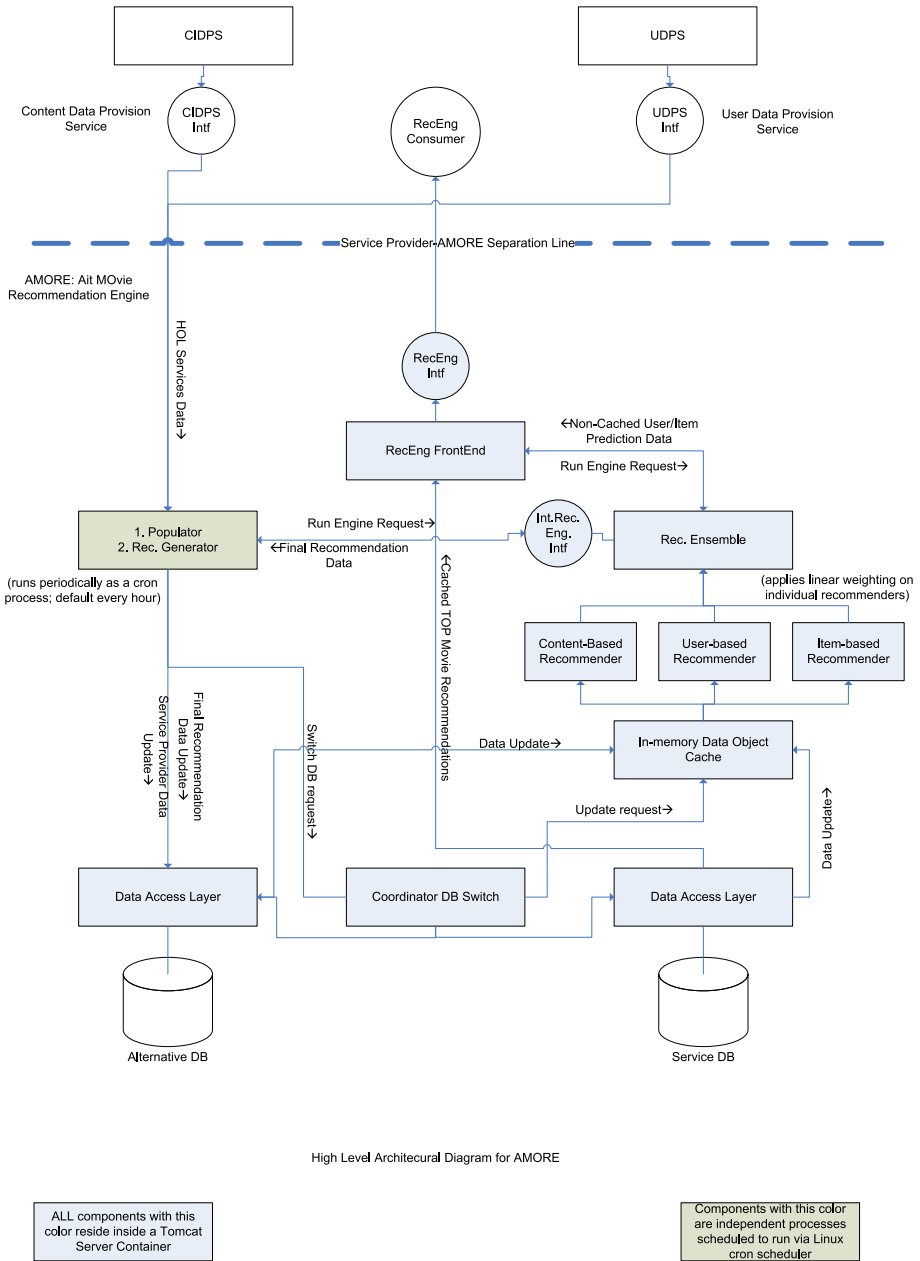


Fig. 2 System architecture overview diagram

the different Boolean Mahout recommenders. Both recommenders adopt a fixed-threshold size set at 0.3 (instead of a fixed neighborhood size) as a criterion for determining the user neighborhood (implying that any user whose similarity to the given user is above this threshold is considered as a member of the given user’s neighborhood).

The initial ensemble contained the above-mentioned Mahout-based recommenders plus a custom content-based recommender that works using only the meta-data of the content items in the user's history of transactions, described in detail in Sect. 3.2 below. The ensemble worked using a voting system in order to come up with a final recommendation score for a specific user-item pair, with each recommender having different "voting power" as measured by an assigned *voting weight*. Each of the recommenders included in the ensemble provides a score for a specific user-item pair that the system stores in persistence. The score value generated by the content-based recommender is normalized to provide values in the closed interval $[0, 1]$, but the score values generated by the two Mahout-based recommenders are not initially normalized since there are no specific ceiling value based upon which the normalization can be applied. In order to overcome this, the system performs the normalization of the score values for these two recommenders based on maximal *ceiling* values calculated after each run of the AMORE batch job, after the step of individual user recommendation generation.

The normalization of the scores generated by each individual recommender was performed by dividing the score value of each user-based Boolean recommender with the *ceiling(max)* value calculated during the latest run. After the system performed the normalization of the recommendation score generated by each recommender for a specific user-item pair, the ensemble applied the previously specified voting weights to each of the recommendation scores to come up with a final ensemble score for the specific pair. To optimize the performance of the ensemble, we needed an objective criterion to define as performance measure. We chose the recall metric $R(10)$ as defined in Karypis [14] and detailed immediately below to be the optimization criterion for this process; many other criteria are described in Shani and Gunawardana [21], but recall (together with so-called *precision-at-n* metric) is deemed as particularly appropriate when evaluating top- n recommendation results as in our case. Having an early snapshot of the database of all users at a particular point in time (November 2012), we removed all purchases of a *particular, randomly chosen, item* from the list of purchases of each user having more than one different items purchased and then run the ensemble for each user in the modified database to produce the top 10 recommendations for that user. If the top 10 recommendations include the removed item, the objective function value is increased by one, else it remains the same. The final objective function value, forming the $R(10)$ value, is the resulting sum divided by the total number of users in the database that had an item removed. Notice that with the given definition of recall and test-bed construction, the average *precision-at-n* $P(n)$ satisfies $P(n) = R(n)/n$. We used the `popt4jlib` open-source library (developed by the first author, available for download at <http://sourceforge.net/projects/popt4jlib/>) to optimize this objective function using a standard *alternating variables* optimization process, also known as coordinate ascent, whereby at each iteration the objective function is restricted to a single variable (voting weight) and is optimized with respect to this variable. The process repeats until in a cycle none of the function's variables changes its value.

From the optimization process, we extracted a number of different configurations tested and tabulate them in Table 1. These early results, though not unacceptable, when combined with very long running times (*in the order of 36h of wall-clock time for each run of the AMORE batch job*) convinced us that performance in terms of quality as well as speed of execution could be significantly enhanced.

Table 1 Initial ensemble recall performance under different voting weights of the individual recommenders on an early dataset (November 2012)

Ensemble voting weights			R(10)
Mahout user-based (Tanimoto)	Mahout user-based (log-likelihood)	Content-based	
0.5	0.5	0.0	0.099
0.25	0.25	0.5	0.073
0.0	1.0	0.0	0.100
1.0	0.0	0.0	0.026
0.33	0.33	0.34	0.082
0.25	0.5	0.25	0.077

3.2 Content-based recommender

In Christou et al. [5], the authors discuss a content-based recommender that takes a pure machine learning approach for recommending movies to a subscriber of a content delivery network: They use the percentage of time the subscriber devoted to watching any of the content they have chosen to determine class membership of the show into two possible classes (“like” or “dislike”), an idea that was also used in Bambini et al. [2], and use 4 weeks’ historical data to create an initial training set. Then, they form an online classifier ensemble based on the Hedge-β algorithm that decides class membership of previously unseen content, to recommend shows in the category “like.”

The above-mentioned approach, while rather successful in its particular setting, cannot be applied in our own setting, since there is no information available whatsoever as to whether the user liked an item they purchased or not; there is neither any user-feedback feature available in the system, nor is there any indication for how long the user actually watched the purchased item. For this reason, we resort to a simple algorithm that offers the minimal guarantee that *the more common attributes an item shares with the items in the history of the user, the higher the score for that item will be.*

To explain the algorithm behind the content-based recommender we implemented, let $H_u = \{p_{u,i_1}, \dots, p_{u,i_k}\}$ denote the set of item purchases for user u so far. With each item-purchase $p_{u,i}$, we have the following associated meta-data:

- An ordered list of actors A_i that appear in the content item i (in order of appearance). Each element in this list is an element of the full set of actors A known to the system.
- An ordered list of directors D_i that directed the content item. Each element in this list is an element of the full set of directors D known to the system.
- An ordered list of producers P_i that produced the content item. Similarly, each element of the list is a member of the set P of producers known to the system.
- An ordered list of the genres G_i of the content item, each element of which is a member of the full set of genres G that the service provider has defined.
- The year y_i the content item was produced.
- An ordered list of the countries C_i that participated in the production of the content item.
- An ordered list of the languages L_i in which the content item is available.
- An ordered list of the languages S_i in which subtitles in the content item is available.
- The total duration of the content item d_i (in seconds).
- The price $m_{u,i} \geq 0$ the user paid to view the item.
- The exact date and time $t_{u,i}$ the user started viewing the content item.

Given the above information, our custom content-based recommender is able to compute the following functions:

$$\begin{aligned}
 F^A(x, u) &= \sum_{i: x \in A_i \wedge p_{u,i} \in H_u} [1 + m_{u,i}] \\
 F^D(x, u) &= \sum_{i: x \in D_i \wedge p_{u,i} \in H_u} [1 + m_{u,i}] \\
 F^P(x, u) &= \sum_{i: x \in P_i \wedge p_{u,i} \in H_u} [1 + m_{u,i}] \\
 F^G(x, u) &= \sum_{i: x \in G_i \wedge p_{u,i} \in H_u} [1 + m_{u,i}] \\
 F^Y(x, u) &= \sum_{i: |x - y_i| < l^y \wedge p_{u,i} \in H_u} [1 + m_{u,i}]
 \end{aligned}
 \tag{1}$$

and similarly, the functions F^L, F^S, F^C are defined; all are cached in appropriate hash tables in memory so that the computations are only performed once, right after the system’s databases are updated. Each of the above functions provides an estimate of the degree of “matching” of a user u with the value of the appropriate attribute x : for example, F^A (“Tom Cruise”, “S668275”) represents the system’s estimate of the matching of user “S668275” with actor “Tom Cruise,” and the estimate is essentially the sum of euros the user has paid to see movies starring Tom Cruise plus the total number of times the user saw movies starring that actor; as another example, F^D (“Alfred Hitchcock”, “S668275”) is the sum of euros the user “S668275” has paid to see movies directed by “Alfred Hitchcock” plus the total number of times the user saw movies directed by this film-maker.

The prediction score of a content item i that has not been already viewed by user u then is computed according to the following formula:

$$R_{u,i} = \sum_{j \in \{A, D, P, G, L, S, C, Y\}} w^j R_{u,i}^j / M_u
 \tag{2}$$

where the quantities $R_{u,i}^j, M_u$ are defined as follows:

$$\begin{aligned}
 M_u &= \sum_{j \in \{A, D, P, G, L, S, C, Y\}} w^j \cdot \sum_{i: p_{u,i} \in H_u} |j_i| \cdot \left[\max_x \left\{ F^j(x, u) : x \in j_i \right\} \right]^{k^j} \\
 R_{u,i}^j &= w^j \cdot \sum_{x \in j_i} \left[F^j(x, u) \right]^{k^j}
 \end{aligned}
 \tag{3}$$

and the set Y_i which measures time proximity is defined as $Y_i = \{x \in \mathbb{N} : |x - y_i| < l^y\}$ where l^y , empirically set to the value 5, is a nonnegative parameter: Essentially, this set helps define an estimate for the interest of a user to movies made in a particular time period, expressed by the function $F^Y(x, u)$, e.g., $F^Y(1949, “S669758”)$ is calculated as the total sum of money the user “S669758” has paid to see movies produced in the years [1944, 1954] plus the total number of items the user has watched that were produced in that time period. The score $R_{u,i} \in [0, 1]$ is therefore a weighted nonlinear combination of the “likeness” of the user toward each of the content item attributes as measured by the total percentage of the amount of money the user has paid to view items with such an attribute as well as by the

number of times the user has viewed such items. The top- n recommendations are the n items currently available for viewing having the highest $R_{u,i}$ score for each user u .

The parameters l^y, w^j as well as the exponents k^j for $j = A, D, P, G, L, S, C, Y$ were considered to be independent nonnegative variables to be optimized, with objective criterion the recall metric R (10) specified in the previous section. Different values for the l^y, w^j, k^j produce different recall metric values. We optimized these parameters, using again the popt4jlib open-source library via a standard genetic algorithm process.

3.3 Final hybrid parallel recommender ensemble

Given the long running times of the initial ensemble (that were mostly attributed to the running times of the two Apache Mahout recommenders), and the quality of results that was deemed suboptimal (since on other movie-related datasets, such as the well-known movielens dataset, most systems achieve an $R(10)$ value above 0.25, see [14]), we decided to replace the two Apache Mahout recommenders with two custom multi-threaded implementations of k -nearest-neighborhood item-based and user-based recommenders as well as a modified fusion technique.

Our custom implementation of the k -NN-item-based recommender is as follows (we simplify somewhat our description to avoid discussing issues that are not essential to the algorithm such as availability of content items and filtering of the user histories according to certain time windows) Let U denote the set of all users who have subscribed at some point to the video-on-demand service; for every user $u \in U$, let their unique sequential user ID be $sid(u) \in \{1, \dots, |U|\}$, and similarly, let I be the set of all content items known to the system, and for every item $i \in I$, let its unique sequential item ID be $sid(i) \in \{1, \dots, |I|\}$. For every user $u \in U$, we compute and store the (sparse) vector $h_{(u)}$ with dimensions equal to $|I|$, whose j -th component ($j = 1 \dots |I|$) is defined according to the equation

$$(h_{(u)})_j = \sum_{p_{u,i} \in H_u: sid(i)=j} [m_{u,i} + 1] \tag{4}$$

In the above equation, the price the user has paid for an item is also taken into account as partial evidence of the ‘‘appeal’’ of the item to the user; in our setting, this is about the closest thing to a user-rating for an item that we have available, but since prices are solely determined by the service provider, they cannot be expected to correlate very well with the true relevance judgment the user would have in mind; still, the computational results showed that this formula improves the quality of results.

Using these vectors, we build for each item $i \in I$ another (sparse) vector $g_{(i)}$ with dimensions equal to $|U|$ whose j -th component ($j = 1 \dots |U|$) is defined to be

$$(g_{(i)})_j = \begin{cases} \frac{1}{\sqrt{|H_u|}}, & sid(u) = j \wedge p_{u,i} \in H_u \\ 0, & \text{else} \end{cases} \tag{5}$$

where $|H_u|$ denotes the number of purchases user u has made so far. Having these data structures available in shared memory, a number of threads are then spawned and execute in parallel without any further synchronization required, to compute for each item they have been assigned to, the k most similar items to it, together with their corresponding similarity values. Following loosely the SUGGEST recommendation library implementation [7, 14], we define the similarity $sim(i_1, i_2)$ between two items i_1, i_2 to be the following quantity:

$$\text{sim}(i_1, i_2) = \frac{\sum_{j:(g_{(i_1)})_j > 0} (g_{(i_2)})_j}{|g_{(i_1)}| \cdot \sqrt{|g_{(i_2)}|}} \tag{6}$$

where $|g|$ denotes the number of nonzero components of the vector g (notice how the similarity relationship between two items fails to be reflective, i.e., $\text{sim}(i_1, i_2) \neq \text{sim}(i_2, i_1)$ for $i_1 \neq i_2$ in general). This computation is fully parallelized in an “*embarrassingly parallel*” loop since no communication or synchronization between the threads is required.

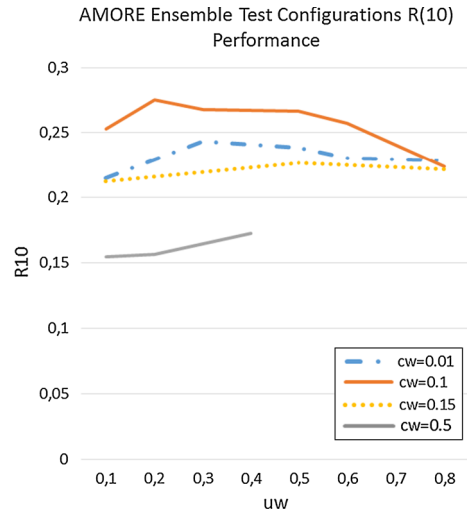
Having computed (in parallel) and stored for each item, the k most similar items’ indices and their corresponding similarity values, the k -NN-item-based recommender computes the top- n recommendations for a user u , using the following procedure: For each nonzero element of the vector $h_{(u)}$, i , the k most similar items to i are examined, and those that are available and not already purchased by the user are added to a hash table C_u whose keys are items j and values the sum of the quantities $(h_{(u)})_{\text{sid}(j)} / \sqrt{q}$ where q denotes the position in the list of k most similar items to j that item i is found in. Once all the nonzero elements of $h_{(u)}$ have been examined, the n key value pairs in C_u with the highest values are proposed as the top- n recommendations for the user u .

Our custom multi-threaded implementation of the k -NN-user-based recommender is completely analogous to our custom implementation of the k -NN-item-based recommender. For every user $u \in U$, we define the (sparse) vector $\hat{h}_{(u)}$ in $|I|$ dimensions, whose j -th component ($j = 1 \dots |I|$) is simply defined to be 1 if item i satisfying $\text{sid}(i) = j$ was purchased by the user and zero otherwise. Having obtained these vectors in a global shared memory, a number of threads are spawned that independently and concurrently execute in another *embarrassingly parallel* loop that does not require any synchronization or communication among them. The loop in each thread computes for each of a set of users it has been assigned to, the similarity between this user and every other user in the database, according to the cosine similarity formula $\text{sim}(u_1, u_2) = \hat{h}_{(u_1)} \cdot \hat{h}_{(u_2)} / (\|\hat{h}_{(u_1)}\| \|\hat{h}_{(u_2)}\|)$ (notice the reflective relationship that holds in this definition of similarity between users: The “amount of similarity” that u_1 has with u_2 is the same as that of u_2 to u_1 .) Once the $k(= 150)$ most similar users to the given user u have been computed along with their similarity scores, these top- k similarity scores are normalized to sum up to unity (by dividing each score by the sum of the k scores). These k most similar users to u define the k -nearest neighbors of u .

Having created the above data structures in shared memory, our k -NN-user-based recommender computes the top- n recommendations for a given user u by computing for each (available and not already purchased) item in the history of the k most similar users to u , the sum of the (normalized) similarity scores of the users that purchased that item; the algorithm then simply recommends the n highest scoring items to user u .

The final top- n recommendations for a particular user u are computed by first asking each of the three recommenders (in parallel) to compute the top $5n$ recommendations for u and then computing for each recommended item (by any of the individual recommenders), a linear-weighted combination of the recommendation values of all three recommenders, whereby if an item is not in the top $5n$ list of some recommender, it assumes by default the value zero for this recommender. The weights w^i, w^u, w^c of the item-based, user-based, and content-based recommenders were set (using the same optimization process that was employed for the computation of optimal weights for the parameters of the content-based recommender) to values approximately equal to 0.75, 0.15, and 0.1 respectively. The actual $R(10)$ value is a highly nonlinear function of the ensemble weights, and this is why an optimization process is needed to determine the optimal values for these weights. In Fig. 3, we show some indicative results of the AMORE $R(10)$ value as a function of these weights.

Fig. 3 Plot of the recall metric $R(10)$ as a function of the weights given to the three recommenders of the AMORE ensemble: each *curve* is drawn for a constant value of the weight given to the content-based recommender cw , and varying the weight uw given to the user-based recommender (the item-based recommender's weight is then 1 minus the sum of the other two recommenders' weights)



The resulting values are sorted in descending order, and the top- n items are returned. The same linear-weighted combination process (with the same weights) applies when the recommender ensemble is asked to produce the final value of a (*user ID*, *item ID*) pair recommendation (see [1] for a detailed discussion of fusing ordered lists of search results of various heuristics in an ensemble to produce superior final ordered result lists).

3.4 Experiments with other base recommender algorithms

Two quite different base recommender algorithms are also very popular today. The first is the so-called *SlopeOne* recommender algorithm [24], and the second is reduced-dimensionality-based recommenders using *Singular Value Decomposition* (SVD, originally proposed as a method to make recommender systems more scalable in the face of very large datasets). The first technique, unfortunately, is not applicable in our case as it only works with datasets containing explicit user ratings of items. Regarding the second one (SVD-based recommenders), since our dataset is more than 99% sparse (see discussion in next section), we expected that SVD-based top- n recommendation results on this dataset would be inferior to the results of k -NN-based algorithms, as Sarwar et al. [23] had reported previously. Indeed, the results produced by Apache Mahout's SVDRecommender implementation were quite worse than the results obtained by the other Boolean user-based recommender implementations available in Mahout (similar quality results were produced using the open-source FunkSVD implementation [8]). The performance of the SVD-based recommendations is shown in Fig. 7 in Sect. 4.

4 Computational results

In this section, we compare the results of our final three-recommender ensemble to the results produced by Apache Mahout's Boolean recommender algorithm with log-likelihood similarity measure and threshold-based definition of user neighborhood, with threshold set at 0.3 (that we found to be the optimal threshold level for our dataset), as well as the individual

performance of each of the three recommenders participating in our ensemble. The reason for choosing this particular configuration for Apache Mahout's user-based recommenders is that it provided the best performance on the dataset used to produce the results shown in Table 1.

All test runs reported below were performed on a Dell Opti-Plex 755 equipped with an Intel Core-2 Quad CPU running at 2.4 GHz having 2 GB RAM running Windows. The testing dataset, being a snapshot of the database taken on April 2013, comprises more than 20,000 users in total, with a little more than one million purchases (views) in total. The total number of items in the database are a little less than 7000, but it is worth noting that the service provider's database contains a significant number of duplicate entries (entries with different item IDs for items with the same title, year of production, actors, directors, etc., with the possible exception that the genres in one entry are sometimes a subset of the genres in the other entry) that we had to keep track of, so that we never recommend an item that the user has already purchased, even though *it is quite common in this dataset for the same user account to have purchased the same item many times* (often 10 times or more); this holds especially true for items that belong to genres such as "Mickey Mouse' Fun Club" and others that are available free of charge. The user-item matrix's nonzero entries are less than 0.9% of the total number of cells in the matrix.

Table 2 provides the recall $R(n)$ values and associated running times T_n for the final ensemble, its individual recommenders acting alone, and Apache Mahout, for $n = 10, 20, 30, \dots, 100$, for recommendations produced using the entire history of each user, except a single item randomly chosen from each user's history to act as the "hidden" item to measure recall against [14].

A graphical illustration of the above results is shown in Figs. 4 and 5, showing recall and response times of the various recommenders. Quite surprisingly, Apache Mahout's user-based recommender lags very significantly behind both the AMORE ensemble and our custom implementation of the user- and item-based recommenders in terms of recall (and equivalently, precision), as well as response times, and it is better than our content-based recommender only in terms of recall (but is much slower). This pattern holds for all values of n . As it can be easily verified, *our AMORE ensemble is more than 80% better than Mahout in terms of the $R(10)$ metric, and is about 15 times faster than Mahout*.

The ensemble's $R(10)$ value for those users whose history of purchases includes 50 or more items is 0.316, quite above the overall recall value of 0.28575, implying that for low values of n , the system is able to better understand the preferences of users with a large history of purchases. However, the ensemble $R(30)$ value is 0.475 for those users having made 50 purchases or more, which is now much closer to the overall $R(30)$ value of 0.45995, showing that as n gets larger, the recall value for the ensemble is approximately the same between users with small purchase histories and those with large ones.

Notice that the recall values obtained compare well with the best values obtained for much more controlled datasets, such as the *movie-lens* dataset where the ratings information that is made available for each user is the true rating the particular user has given to the item, as opposed to our dataset that only contains the purchase history of each user account (that is often used by all members of the household). To alleviate this additional problem with our dataset, we have provided an additional feature to our algorithms, namely the ability to train them using only those content items seen by the user within a particular time window. The rationale behind this choice is that by narrowing the user history to items seen for example during prime time, the chances that this user history is the union of more than one actual person in the household should be reduced, and therefore, the accuracy of the system should be increased. In Table 3, we show the results of running the various recommenders trained

Table 2 Comparing recommenders' quality and response times given the entire user histories (April 2013)

<i>n</i>	AMORE ensemble			Apache Mahout			AMORE item-based			AMORE user-based			AMORE content-based		
	R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)	
		Load	Rec.		Load	Rec.		Load	Rec.		Load	Rec.		Load	Rec.
10	0.286	238	1076	0.158	238	17,821	0.263	238	325	0.250	238	446	0.046	238	362.4
20	0.388	238	1176	0.232	238	17,558	0.365	238	324	0.339	238	427	0.069	238	358.8
30	0.460	238	1325	0.286	238	17,061	0.433	238	327	0.400	238	431	0.085	238	361.6
40	0.518	238	1452	0.326	238	17,068	0.490	238	322	0.447	238	640	0.100	238	366
50	0.565	238	1618	0.367	238	17,083	0.531	238	321	0.484	238	462	0.115	238	369.2
60	0.602	238	1729	0.403	238	17,063	0.566	238	323	0.515	238	462	0.128	238	379.2
70	0.633	238	1752	0.435	238	17,047	0.591	238	371	0.542	238	476	0.140	238	382.8
80	0.659	238	1989	0.465	238	17,089	0.613	238	375	0.566	238	471	0.151	238	392.8
90	0.683	238	2161	0.489	238	17,112	0.635	238	325	0.588	238	475	0.160	238	401.6
100	0.706	238	2814	0.512	238	17,160	0.653	238	323	0.605	238	485	0.170	238	436.4

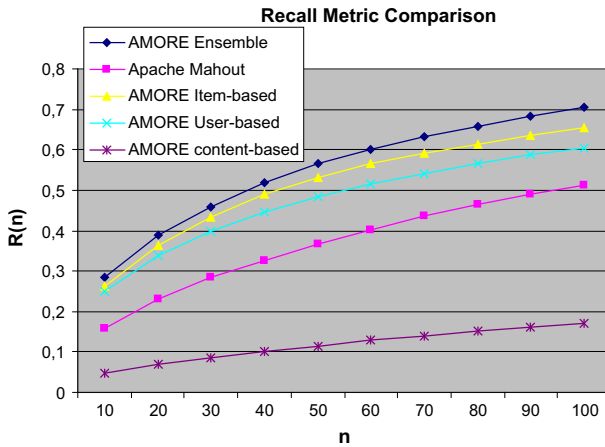


Fig. 4 Plot of the recall metric $R(n)$ as a function of n for various recommenders trained on the entire user purchase histories

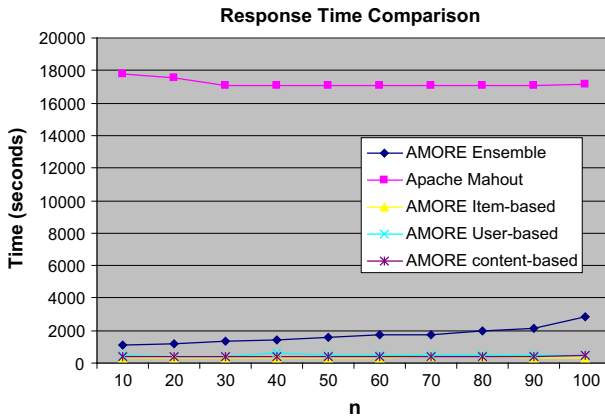


Fig. 5 Plot of the response time T_n as a function of n for various recommenders trained on the entire user purchase histories

using only items that were seen by the users during a time that overlaps with the “prime time” window between 9 p.m. and 1 a.m. The results again show a very clear superiority of our ensemble, even though they do *not* improve upon the results obtained when training the classifiers with the entire history of user purchases, therefore the hypothesis that time windows can help narrow down the persons using the service from each user account does not have statistical support. The quality of the results is visualized in Fig. 6. Regarding running times, our ensemble is between 1.98 and 6.46 times faster than Apache Mahout.

We attribute the much faster response times of our system to two main reasons:

1. A *sophisticated multi-threading design and implementation* that allows the software to utilize 100% of the available cores of the CPU and obtain essentially linear speedups. To achieve this performance, each running thread never creates any objects on the heap (that dramatically reduce parallel performance) using the operator *new* and of course does not

Table 3 Comparing recommenders' quality and response times given the history of user purchases that occurred between 9 p.m. and 1 a.m. (dataset of April 2013)

<i>n</i>	AMORE ensemble			Apache Mahout			AMORE item-based			AMORE user-based			AMORE content-based		
	R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)		R(n)	Time (s)	
		Load	Rec.		Load	Rec.		Load	Rec.		Load	Rec.		Load	Rec.
10	0.243	241	498	0.165	241	4531	0.223	241	243	0.202	241	259	0.062	241	999
20	0.334	241	578	0.237	241	4529	0.314	241	242	0.282	241	401	0.089	241	1009
30	0.401	241	704	0.293	241	4431	0.379	241	253	0.335	241	399	0.111	241	1026
40	0.453	241	872	0.339	241	4482	0.428	241	243	0.381	241	396	0.133	241	1008
50	0.495	241	1043	0.373	241	4459	0.466	241	246	0.416	241	399	0.151	241	1061
60	0.534	241	1040	0.409	241	4489	0.496	241	240	0.444	241	401	0.167	241	1042
70	0.566	241	1272	0.439	241	4434	0.521	241	244	0.471	241	404	0.183	241	1054
80	0.593	241	1507	0.470	241	4435	0.543	241	247	0.495	241	403	0.195	241	1076
90	0.620	241	1798	0.499	241	4444	0.562	241	246	0.514	241	405	0.208	241	1094
100	0.642	241	2170	0.518	241	4524	0.578	241	247	0.533	241	405	0.217	241	1128

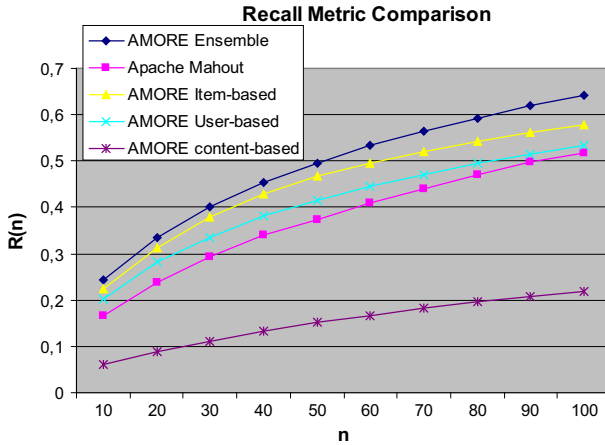


Fig. 6 Plots of the recall metric $R(n)$ as a function of n for various recommenders trained on user histories on the interval 9 p.m. to 1 a.m

have to obtain any synchronization locks as they only write data in different areas of the same arrays and do *not* require any data computed in parallel from the other threads

2. A better-suited implementation of sparse vectors for k -NN-based implementations of recommender algorithms than the one available in the Colt numeric library that was adopted for Apache Mahout’s core numeric computations, combined with a very fast implementation of thread-local object pools for lightweight objects that make it possible for the computing threads never to call the *new* operator as stated in reason #1 above.

As another experiment, we have deleted from the snapshot of our database taken on April 2013, all user purchases that occurred during the last 2 weeks recorded in the system, and have trained the system with the remaining older data, to see the levels of the precision and recall metrics on this differently constructed test dataset. The plots in Fig. 7 show how average *precision*, *recall*, and the combined *F-metric* vary with different recommendation list lengths (measured in points that are multiples of 5 and 8). The reduced recall values are expected since the system must now be able to find not just one of the items the user has selected at any random point in the past, but the items the user saw in the last 2 weeks: but within the last 2 weeks, items made available within that time frame may have not been seen yet by a statistically significant number of users so that the system can “understand” to what other items they are similar with, thus the drop in the recall values. Again, the AMORE ensemble is far superior to Apache Mahout user-based recommender or the SVD-based recommendation.

We have performed an *empirical* small-scale test where we asked eight volunteer users (other than the authors) to explicitly state the relevance (like/dislike) of the top 10 recommendations the systems produced for them, *after declaring just five of their favorite movies*. The precision of the results is shown in Fig. 8 and is much more encouraging. The significant difference between *explicitly stated user relevance* and *calculated system accuracy from user histories* can be attributed to many factors, the most prominent of which would be the fact that users are very likely to have already seen in the theaters their favorite movies that the system calculates for them, or the sometimes high pricing of specific content items available for viewing.

We believe that the noteworthy fact that *precision-at-n* is above 60 % for all values of n up to 10 for users having stated only five movies they like can be attributed to the combination

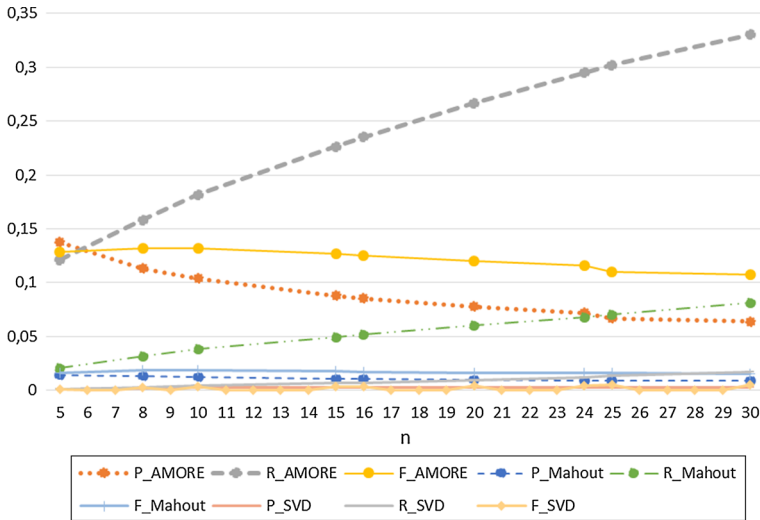


Fig. 7 Plots of precision (P_AMORE, P_Mahout, P_SVD), recall (R_AMORE, R_Mahout, R_SVD), and F-metric (F_AMORE, F_Mahout, F_SVD) for the AMORE ensemble, Apache Mahout, and SVD-based recommender when the test data are the last 2 weeks of user purchases. The F-metric is maximized at $n = 10$ for the AMORE and Apache Mahout engines

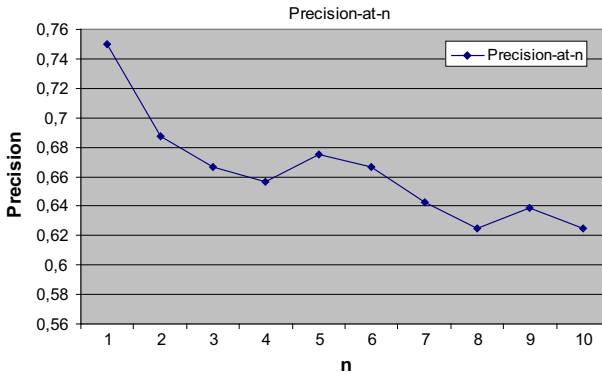


Fig. 8 Empirical average AMORE precision-at- n measured after users have stated exactly five of their most favorite movies

of user- and item-based nearest-neighbor recommenders and content-based recommendation in our ensemble, which can be particularly helpful when only a small number of preferences is known for a user.

Finally, in Fig. 9, we show how AMORE performance has evolved over time.

The latest experimental results on system recall and response times (September 2013, on a database of more than 26,000 users and more than 1.9 million views) show that *AMORE outperforms Apache Mahout by more than 100% in terms of the R(10) metric, and more than 6300% in terms of speed! AMORE has been increasing its performance as time passes by, by more than 13.8% between April and September 2013. Mahout's user-based recommender (using the log-likelihood metric) on the other hand, dropped its performance by more than 10% in the same time interval.*

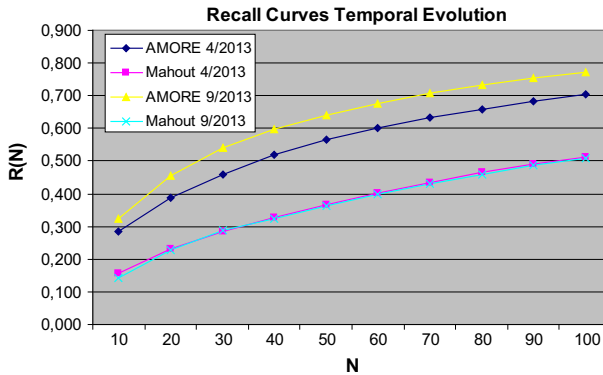


Fig. 9 Temporal evolution of AMORE and Mahout performance

In Cremonesi and Turrin [6] and Bambini et al. [2], the authors showed that in their own production environments, the recall rate of item-based recommenders may deteriorate as time passes by, due to cold-start issues and the fact that once new users view so-called easy-to-recommend items (i.e., blockbusters), the task of the recommender engine becomes much more difficult. In contrast, our results indicate that the combination of our custom item-based recommender, user-based, and content-based leads to a system that evolves so that it improves its recall rate as time passes, and the improvement is significant. We reason that this improvement is mainly due to the fact that k -NN-user-based recommenders attain a higher level of personalization than k -NN-item-based recommenders as argued in Karypis [14], and therefore, after a while, and once a user has seen the “easy” items, the user-based part of the ensemble, together with the equally highly personalized content-based recommender, “kick-in” to provide more relevant recommendations to the user than the item-based recommender alone. This “kicking-in” happens as the item-based recommender starts providing recommendations with mostly low ratings, so that the recommendations provided by the user-based and the content-based recommenders dominate the final recommendation list for each user. On the other hand, it should be noted that the improvement comes with the associated cost of having to maintain the user model of the user-based recommender, which can become much more expensive than the model of the item-based recommender when the number of users exceeds the number of items by orders of magnitude (again, for an estimation of the complexity of building and updating k -NN-user- or item-based recommenders see [14]). In our setting, the subscriber base of our triple-play service provider customer is not expected to exceed one hundred thousand, and this is comparable to the number of items the provider currently offers: On September 2014, there were approximately 39,000 users in the database and a little less than 20,000 items available. Also notice that the algorithms’ main loops are “embarrassingly parallel,” and further increases in user or content size can be easily tackled by simply “throwing more processor cores at the problem,” or in other words by increasing the number of threads spawned in a computer having more cores. This strategy (theoretically) should let AMORE handle subscriber bases of up to a few millions of subscribers without any problem. To tackle problems above this size, the algorithms could become distributed as well, to utilize distributed memory clusters and clouds of computers.

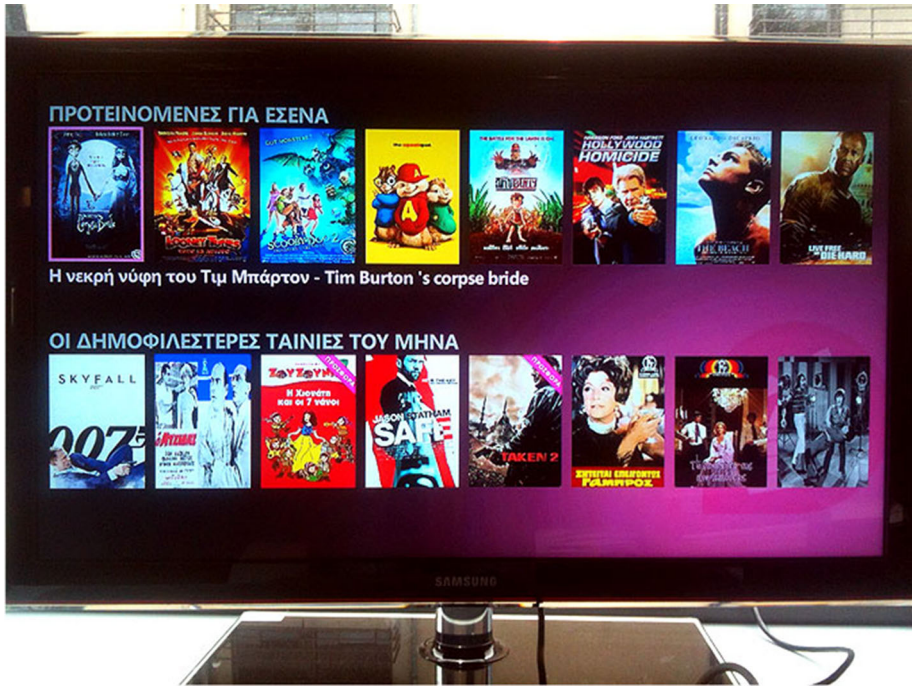


Fig. 10 AMORE End User-On-TV Screen Interface

5 User interfaces

AMORE recommendations are shown to the service subscribers on their TV screen in a special screen shown in Fig. 10. The first row shows the recommended movies for the user account, whereas the second row immediately below it shows the month's most popular movies.

While Fig. 10 shows a screen shot of the user interface as seen by the end user (the service subscriber), AMORE offers a variety of other interfaces. As almost any other commercial system, AMORE comes equipped with a set of Business Intelligence/Analytics functions to help operators appreciate the effectiveness of the recommendations engine, as well as help them in their decision-making processes regarding item promotions, etc. In Fig. 11, the user has chosen to see the time evolution of the recommendations made by the system for children's movies, as well as how many of them were actually followed (the periodicity of the graph is of course due to the fact that most items are viewed during weekends).

And finally, in Fig. 12, we show how the user may choose to see the *contribution in terms of sales and views of action movies featuring Tom Cruise as an actor*. These are just a few of the functionalities provided by the AMORE BI/BA tool.

6 Conclusions and future directions

We have presented AMORE, a commercial, hybrid movie recommendation system. The system addresses a number of issues that originate from business requirements and limitations

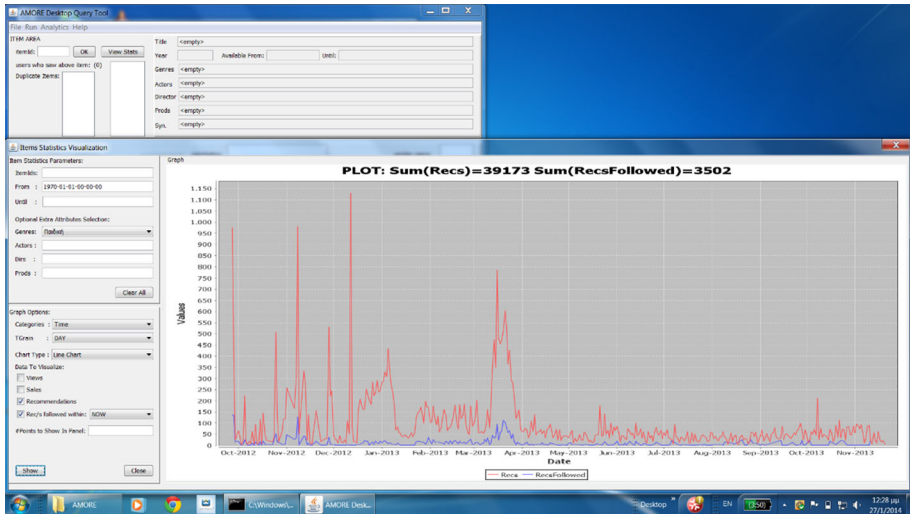


Fig. 11 AMORE business intelligence UI

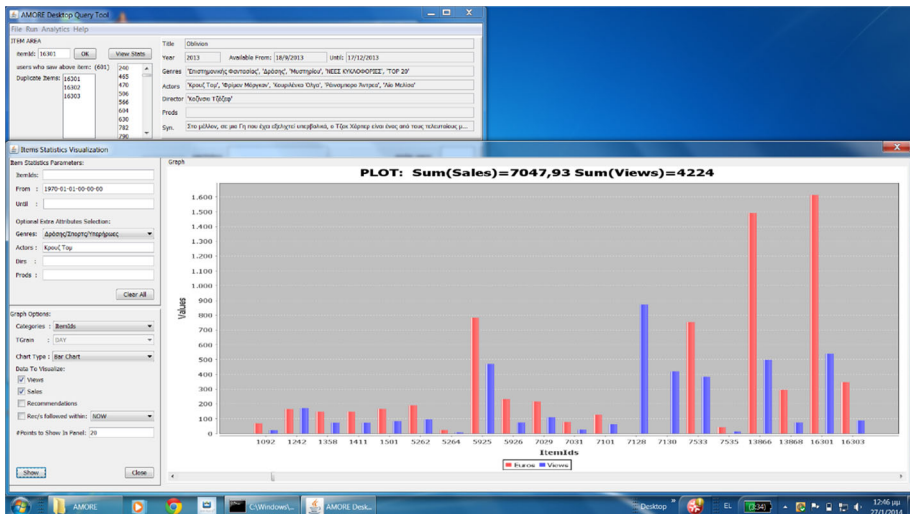


Fig. 12 AMORE business analytics UI final screenshot

similar to those that other commercial systems face. One of the most significant challenges in our case is the requirement to accurately determine user preferences given only the information of consumed items from all members in the household to which a given user account belongs, in the absence of any relevance judgments. Relevant to that is the fact that a large percentage of all available content is currently offered for free, making it easier for anyone to “purchase” items that they would not otherwise choose to purchase. Without any information as to the length of time that the user actually watched any item (and especially a “free” item), user histories can easily become “tainted” with items they do not really like but the system has no way of knowing that.

By using different types of recommenders including item-based, user-based as well as content-based, the ensemble is able to handle cases where a significant number of users have consumed a significant number of items, thus taking advantage of the benefits of collaborative filtering, as well as cases where new content items have not been yet consumed by any user but by using its content-based recommender, the ensemble can still provide meaningful recommendations (thus, at least partially, solving the “new content” cold-start-type problem).

The system also deals with the “new user” cold-start-type problem (when new users are added to the system), using the following *business rule*: whenever a new user is inserted into the system, and before they have purchased any items, the system simply recommends the top recommendations made for all other users in the system at that time. In the corner case of new users having seen yet only very few *and*rarely watched movies, the content-based recommender is still able to provide other similar movies to the content the user has opted for, which is essentially optimal in such context.

Finally, the system addresses hardware infrastructure constraints; we have introduced a cost-effective way with which the system is able to provide instant replies to web service requests and at the same time renew user recommendations as frequently as possible (in the order of 15 min or less). This was made possible by the architecture of the system, as well as the two databases following the same data model that are used to separate the updates of the system (performed by the AMORE batch job) from the response to web service requests for recommendations (performed by the AMORE web services that live in a web application server).

AMORE is currently the only live commercial recommender system for video-on-demand in Greece and has been successfully deployed in the production environment of a prominent Greek triple-play services provider and has already contributed to an increase in the provider’s profits in terms of movie rental sales while at the same time offers customer retention support allowing the company’s Marketing Department to offer more interesting subscription offers to both old and new customers alike.

We have experimented with the application of various algorithms implemented in the Apache Mahout suitcase (upon which myrrix is also based, see <http://myrrix.com>) but the results were *not* deemed satisfactory neither in terms of quality nor in terms of response times, thus necessitating the development of our own parallel multi-threaded custom implementation of the well-known k -NN-item-based and k -NN-user-based recommenders and variants thereof. Various other experimental recommendation systems have already shown the superiority of hybrid systems incorporating tens or even hundreds of individual recommender algorithms over schemes incorporating only a single algorithm (the best Netflix prize contestants belong in this category). AMORE results have shown that a very small number of different types of recommender algorithms (that can be updated very fast) are sufficient to produce high-quality recommendations that users enjoy: Currently, the *users make a rental from the proposed recommendations once for every two visits to the AMORE recommendations screen*. In the immediate future, we are aiming to introduce novel algorithms which take into consideration additional information about user behavior patterns including the prices that users are willing to pay in order to provide improved recommendation services to them.

Acknowledgments The authors would like to thank Hellas On Line S.A. for providing the industrial grant that made this research possible.

References

1. Amolochitis M, Christou IT, Tan Z-H, Prasad R (2013) A heuristic hierarchical scheme for academic search and retrieval. *Inf Process Manag* 49(6):1326–1343

2. Bambini R, Cremonesi P, Turrin R (2011) A recommender system for an IPTV service provider: a real large-scale production environment. In: Ricci et al (eds) *Recommender systems handbook*. Springer, New York, NY
3. Cha M, Kwak H, Rodriguez P, Ahn YY, Moon S (2007) I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In: *Proceedings of the 7th ACM SIGCOMM conference on internet measurement*
4. Cho J, Roy S (2004) Impact of search engines on page popularity. In: *Proceedings of World Wide Web conference*
5. Christou IT, Gekas G, Kyrikou A (2012) A classifier ensemble approach to the TV-viewer profile adaptation problem. *Int J Mach Learn Cybern* 3(4):313–326
6. Cremonesi P, Turrin R (2009) Analysis of cold-start recommendations in IPTV systems. In: *Proceedings of ACM recommender systems conference RecSys'09*, pp 233–236
7. Deshpande M, Karypis G (2004) Item-based top-n recommendation algorithms. *ACM Trans Inf Syst* 22(1):143–177
8. Ekstrand MD, Ludwig M, Konstan JA, Riedl JT (2011) Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit. In: *Proceedings of ACM recommender systems conference RecSys'11*
9. Golbeck J, Hendler J (2006) Filmtrust: movie recommendations using trust in web-based social networks. In: *Proceedings of the IEEE Analysis of consumer communications and networking conference*
10. Goldberg D, Nichols D, Oki BM, Terry D (1992) Using collaborative filtering to weave an information tapestry. *Communications of the ACM*. December
11. Herlocker JL, Konstan JA, Terveen LG, Riedl JT (2004) Evaluating collaborative filtering recommender systems. *ACM Trans Inf Syst* 22(1):5–53
12. Hurley N, Zhang M (2011) Novelty and diversity in top-n recommendation—analysis and evaluation. *ACM Trans Internet Technol* 10(4):14 1–30
13. Jahrer M, Toscher A, Legenstein R (2010) Combining predictions for accurate recommender systems. In: *Proceedings of the ACM conference on knowledge discovery in databases (KDD 2010)*
14. Karypis G (2001) Evaluation of item-based top-n recommendation algorithms. In: *Proceedings of the 10th conference on information and knowledge management (CIKM 01)*, pp 247–254
15. Lathia N, Hailles S, Capra L, Amatriain X (2010) Temporal diversity in recommender systems. In: *Proceedings of the SIGIR 2010*, July 19–23, Geneva, Switzerland
16. Li Y, Lu L, Xuefeng L (2005) A hybrid collaborative filtering method for multiple-interests and multiple-content recommendation in E-Commerce. *Exp Syst Appl* 28(1):67–77
17. Li Y, Zhai CX, Chen Y (2014) Exploiting rich user information for one-class collaborative filtering. *Knowl Inf Syst* 38:277–301
18. Mild A, Natter M (2002) Collaborative filtering or regression models for Internet recommendation system? *J Targeting Meas Anal Market* 10(4):304–313
19. Pazzani M, Billsus D (2007) Content-based recommendation systems. *Lect Notes Comput Sci* 4321:325–341
20. Ricci F, Rokach L, Shapira B, Kantor P (eds) (2011) *Recommender systems handbook*. Springer, New York, NY
21. Shani G, Gunawardana A (2011) Evaluating recommendation systems. In: Ricci et al (eds) *Recommender systems handbook*. Springer, New York, NY
22. Shardanand U, Maes P (1995) Social information filtering: algorithms for automating 'word of mouth'. In: *Proceedings of the human factors in computing conference (CHI '95)*. Denver, CO
23. Sharwar B, Karypis G, Konstan J, Riedl J (2000) Application of dimensionality reduction in recommender system—a case study. University of Minnesota Computer Science and Engineering, Technical Report 00-043
24. Owen S, Anil R, Dunning T, Friedman E (2011) *Mahout in action*. Manning, New York
25. Yu H, Zheng D, Zhao BY, Zheng W (2006) Understanding user behavior in large-scale video-on-demand systems. *ACM SIGOPS Operat Syst Rev* 40(4):333–344
26. Zhang M, Hurley N (2008) Avoiding monotony: improving the diversity of recommendation lists. In: *Proceedings of the ACM recommender systems conference RecSys 08*, pp 123–130



Ioannis T. Christou holds a Dipl. Ing. Degree in Electrical Engineering from the National Technical University of Athens, Greece (1991), an MSc (1993) and PhD (1996), both in Computer Sciences from the University of Wisconsin at Madison (Madison, WI, USA), and an MBA from NTUA and Athens University of Economics and Business (2006). He has been with Delta Technology Inc., as Senior Developer, with Intracom S.A. Development Programmes Dept. as Area Leader in Data and Knowledge Engineering, and with Lucent Technologies Bell Labs as Member of Technical Staff. He has also been an adjunct Assistant Professor with the University of Patras, Greece, and an Adjunct Professor at Carnegie-Mellon University, Pittsburgh, PA, USA. He is currently an Associate Professor at Athens Information Technology, Athens, Greece, and is the CTO and co-founder of IntelPrize, a Big-Data start-up company, and has published more than 70 articles in scientific journals and peer-reviewed conferences.



Emmanouil Amolochitis received the BSc degree in Information Technology in 2005 from Deree, The American College of Greece, the MSc degree in Information Technology and Telecommunications in 2009 from Athens Information Technology, Greece, and the PhD degree from Aalborg University, Denmark, in 2014. He works as research engineer for Voice-Web, Athens, Greece. His research interests include Data Mining, Machine Learning, and Information Retrieval.



Zheng-Hua Tan received the BSc and MSc degrees in 1990 and 1996, respectively, both in Electrical Engineering from Hunan University, China, and the PhD degree in Electronic Engineering from Shanghai Jiao Tong University, China, in 1999. He is an Associate Professor in the Dept. of Electronic Systems at Aalborg University (AAU), Denmark. Before joining AAU, he was a postdoctoral fellow in the Dept. of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST), Korea, and was also a visiting scientist at the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA. He was also an Associate Professor in the Dept. of Electronic Engineering at Shanghai Jiao Tong University. His research interests include Computational Intelligence, Speech Recognition, and Machine Learning. He has published extensively in these areas in refereed journals and conference proceedings, and is a member of the editorial board of many scientific journals.