Multimedia Tools and Applications manuscript No.
(will be inserted by the editor)

Online Optimization for User-specific Hybrid Recommender Systems

Simon Dooms · Toon De Pessemier · Luc Martens

Received: date / Accepted: date

Abstract User-specific hybrid recommender systems aim at harnessing the power of multiple recommendation algorithms in a user-specific hybrid scenario. While research has previously focused on self-learning hybrid configurations, such systems are often too complex to take out of the lab and are seldom tested against real-world requirements. In this work, we describe a self-learning user-specific hybrid recommender system and assess its ability towards meeting a set of pre-defined requirements relevant to online recommendation scenarios: responsiveness, scalability, system transparency and user control. By integrating a client-server architectural design, the system was able to scale across multiple computing nodes in a very flexible way. A specific user-interface for a movie recommendation scenario is proposed to illustrate system transparency and user control possibilities, which integrate directly in the hybrid recommendation process. Finally, experiments were performed focusing both on weak and strong scaling scenarios on a high performance computing environment. Results showed performance to be limited only by the slowest integrated recommendation algorithm with very limited hybrid optimization overhead.

Keywords Recommender systems · Hybrid · Optimization · Online · MovieTweetings · User-interface · HPC

1 Introduction

Recommender systems have become an everyday commodity as they are integrated in many of the online (and offline) services people use on a daily basis. They are

S. Dooms - T. De Pessemier - L. Martens
Wica, iMinds-Ghent University
G. Crommenlaan 8 box 201, 9050 Ghent, Belgium
Tel.: +32-09-33-14908
Fax: +32-09-33-14899
E-mail: simon.dooms@intec.ugent.be
E-mail: toon.depessemier@intec.ugent.be
E-mail: luc.martens@intec.ugent.be

used to bridge the gap between interesting content and users lost in information overload. One of the earliest trends in the domain was using community information where the idea was that similar users could be used as guides towards interesting content i.e., collaborative filtering [35]. Since then, the recommender system domain has been constantly evolving and adapting to new trends and needs in society which resulted in the development of many types of recommendation algorithms each focusing on specific users, data or scenarios. New categories of recommendation algorithms included content-based systems [29], knowledge-based systems [8], social recommender systems [21], mobile recommender systems [36], context-aware recommender systems [2] and hybrid recommender systems [9]. The hybrid recommender systems combine multiple individual recommendation algorithms which allows to increase recommendation quality and minimize the individual drawbacks each of them might have [9,1,5].

Many recommendation algorithms exploit the idea that users in a system behave similarly [26] but more and more research is starting to focus on the individuality of a single user and experimenting with user-specific approaches for generating recommendations [5,19,28]. Assuming every user is different and given all recommendation algorithms, each user may be best served by a different algorithm, or even a different combination of algorithms (i.e., hybrid). In previous work [16] we pursued the ideal hybrid recommender which would be capable of integrating all known recommendation algorithms and auto-adapting its hybrid configuration to dynamically generate optimal recommendations for every user specifically. A hybrid recommendation strategy was devised capable of dynamically optimizing its configuration. In this work, we build on these results and try to get the recommender out of the lab by assessing and improving its ability towards meeting real-world requirements for an online recommendation scenario.

To define real-world requirements for such a hybrid recommender system, we introduce a realistic recommendation use case focused on the movie domain. Imagine we want to deploy an online recommender system called 'MovieBrain' that recommends interesting movies to users. The most important requirement for such a system would be **scalability**. For online systems it is extremely difficult to predict the number of active users since online popularity is very variable. An online system may serve a number of users ranging from just a few hundreds to many thousands and even millions. More importantly the number of users may change very quickly in short periods of time because of online viral effects. Therefore an online system should be able to dynamically scale with the workload it is presented with.

Another requirement for online systems is **responsiveness**. Nowadays users have grown accustomed to fast and responsive online services. Whether they are searching on Google, posting updates to Facebook or watching videos on YouTube, they expect an instant response from the system they interact with. Responsiveness in terms of a recommender system scenario would mean that user interactions have immediate visible effects. For example a user that rates a recommended movie as *bad*, does not want to see that movie in its recommendation list anymore (even though the system may only calculate new recommendations once a day).

While recommender systems in the past often acted as *black boxes* where ratings go in and recommendations come out [24], current-time users expect some kind of explanation about the origin of the recommendations. Online platforms

like IMDb[1] or Amazon[2] display their recommendations with accompanying titles as 'People who liked this also liked...', 'Frequently Bought Together' or 'Customers Who Bought This Item Also Bought'. Despite their simplicity, the titles succeed in explaining the users how the recommendations are calculated. Even though the explanation may be an oversimplification of the true recommendation calculation process, it may still serve to inspire user trust and loyalty [41]. Therefore it is important for an online (recommender) system to be (or at least *seem*) **transparent** to the user.

Finally, users want **control**. In typical popular online services as Facebook or Google users are able to customize their experience by manipulating settings involving notifications, privacy and many other types of configuration. Users need to be able to interact with the system and provide preference feedback. Our online movie recommender system 'MovieBrain' should therefore interactively offer some way for users to be in control of their resulting recommendations or at least have some way of influencing and guiding the system other than by merely providing ratings. In conclusion, the discussed real-world challenges for an online recommender system can be summarized in the following list of requirements (*REQs*).

- **REQ1** Responsiveness
- **REQ2** Scalability
- **REQ3** System transparency
- **REQ4** User in control

In this work we present a hybrid recommender system capable of optimizing its configuration for individual users and we evaluate and improve its ability to meet each of the discussed requirements in an online movie recommendation scenario. The remainder of this work is organized as follows. In Section 2 we discuss our previous work and the relation of this work towards the current state of the art. Section 3 presents our hybrid optimization system and covers the optimization architecture taking into account **REQ1** and **REQ2**. Section 4 illustrates how **REQ3** and **REQ4** can be satisfied in the user-interface of our 'MovieBrain' recommender. In Section 5 the results of a number of experiments are detailed aimed at determining the actual performance and scalability of the system when deployed on a cluster of computing nodes. Finally, Section 6 discusses the obtained results and Section 7 concludes the contributions of this work.

2 Related Work

This work builds on our previous work [16] where we focused on offline optimization for user-specific hybrid recommender systems. Different hybrid configurations (combining up to 10 individual recommendation algorithms) were tested and their optimization performance was evaluated. Experiments with both hybrid switching (i.e., choose the best algorithm) and a weighted hybrid strategy (i.e., combine results using a weighted formula) were performed, see [9] for a thorough analysis of different hybrid strategies. Results showed that the switching strategy was highly

[1] http://www.imdb.com
[2] http://www.amazon.com

sensitive to which individual algorithms were integrated in the hybrid configuration while the weighted approach was more robust and obtained significant better optimization results. In this work we get our hybrid recommendation strategy out of the lab and see how it can be modified to handle real-world challenges as responsiveness, scalability, transparency and user control.

Responsiveness for recommender systems translates to being able to react in almost real time to the arrival of new ratings in the system. Most recommendation algorithms need to retrain their complete model to integrate new data which can often not be done in real time. For some specific recommendation algorithms, online updating approaches have been developed such as SVD [7] or MatrixFactorization [34]. In [10], the *StreamRec* recommender system was demonstrated which allowed instant recommendation updates using an underlying item-based collaborative filtering approach. Since the online updating approaches are usually algorithm specific, few research focuses on realtime updating hybrid models. In [4], the authors describe their hybrid system, called *STREAM*, which applies a similar optimization approach as described in this work, but requires domain experts to manually define runtime metrics used in their ensemble learning method. A real time strategy for integrating new ratings in the models was not mentioned.

Scalability is often the focus of recommender systems research, but usually focused on tweaking recommendation algorithms to make them faster and more able to handle large workloads rather than allowing their implementations to be distributed over multiple computing nodes (as this work does). For neighborhood-based recommendation algorithms for example restricting the size k of the neighborhood is such a typical tweak [3,23]. When a recommender system needs distributed computing however, often a MapReduce paradigm is involved (e.g., [39, 27,43,11,12]). This requires computational steps to be rewritten as *map* and *reduce* functions which is not straightforward to do and causes overhead thus reducing parallel efficiency [13]. In this work we show how the system can embrace distributed computing without the need for MapReduce.

Providing system transparency and user control in a recommender system should prevent users from feeling trapped inside a *filter bubble*[3] of tailored information. Explanations have been known to positively increase the user perceived system transparency [40]. User control in a system is however difficult to achieve. Aside from processing ratings, recommendation algorithms usually do not provide the tools for users to allow fine-grained preference feedback. In [37], a class of recommendation interface is introduced called *meta-recommendation systems*. They experimented with a hybrid system called *MetaLens* that allowed users control over their recommendations by means of a preference screen where a number of item features could be filtered on. Their user-study confirmed that users preferred the advanced level of control offered by their system.

3 An online hybrid optimization strategy

In this section we continue from previous work [16] where an offline optimization procedure for a self-learning hybrid recommender was defined. We propose an architecture for an online environment taking into account the defined requirements

[3] http://www.thefilterbubble.com

for online recommender systems. The hybrid recommender proposed in this work is user-specific and so optimizes its model for each user in the system individually.

3.1 The hybrid optimization methodology

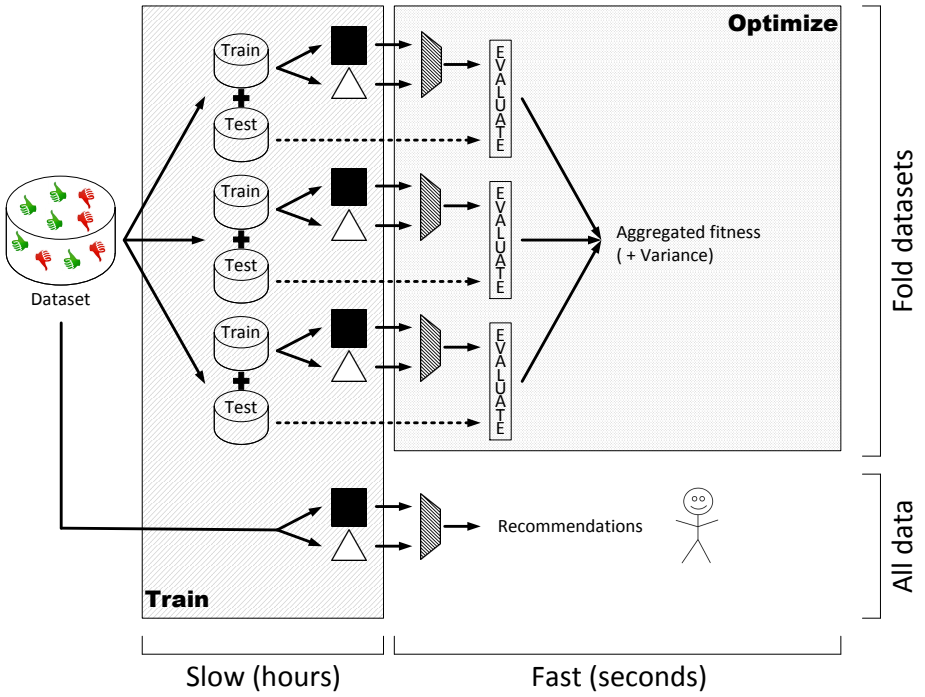

Fig. 1 The optimization process for the hybrid recommender illustrated for one user, using 3 folds and 2 (individual) recommendation algorithms.

Fig. 1 illustrates the general optimization process for one user of the hybrid recommender. The process starts with the concept of a rating dataset. We assume a user has expressed an opinion about a number of items present in the system. In this work we are developing the use case of an online movie recommender so items are movies. In a first step, from the rating dataset multiple fold datasets are created which are then split according to some pre-set ratio into train and test fold datasets. Fig. 1 illustrates the situation with 3 fold datasets. Each training subset of the fold datasets is provided as input to a number of recommendation algorithms (2 algorithms depicted in the figure as a black square and white triangle shape). At the same time the complete rating dataset is also provided as input to instances of the same algorithms. Each algorithm then, in parallel, trains its models based on the given input. In the figure, 2 algorithms are defined and so 4 instances of those algorithms (3 for the fold datasets and 1 for the complete rating dataset) will be trained, which results in the computation of 8 models. This computational step can be potentially very slow depending on which recommendation algorithms are

involved. Algorithms like MatrixFactorization are generally accepted to train fast [31], while other algorithms like KNN methods can be very slow [25] (depending on the parameters e.g., neighborhood size). Although this computation phase will be very slow, it needs to be run only once in order for the system to be able to present a user with recommendations. After this initial computation the system will be able to incorporate new rating data and react to user responses in real time as we will show later in this section.

When the training of the algorithm models has finished, the system uses the output i.e., recommendations to optimize a weight vector used for the configuration of the final hybrid recommendation list. We integrate a *weighted* hybrid approach [9] and therefore such a weight vector is needed for the aggregation of the individual recommendation list. In the figure this aggregation is presented as the vertical trapezoid shape that takes multiple recommendation results as input and outputs one hybrid recommendation list. In a first optimization step the outputs of the recommendation algorithms trained on the fold datasets are aggregated using an initial start weight vector (identical for all folds). Since the fold datasets were split in train and test sets (and models were only trained on the train sets), the remaining test sets can be used to evaluate the quality of the aggregated result. We do not specify an exact method of evaluation as this will be depend on the end goal of the recommender (e.g., user satisfaction, recommendation accuracy, item coverage, etc.). The output of the evaluation must however be quantifiable into a numeric value so that it can be compared and measured. Three evaluation values result from the scenario as depicted in the figure, one for each fold. The evaluation scores are aggregated, by some chosen aggregation function e.g., *arithmetic mean*, into a single *fitness* value indicating the quality of the current weight vector. The variance of the individual evaluation scores between the folds must also be taken into account to indicate the consistency of the performance of the weight vector over the different dataset folds.

The weight vector is then step-wise optimized by applying standard optimization procedures borrowed from the machine learning domain until a certain number of iterations has passed or a sufficient fitness value has been reached. In our previous work [16] we illustrated the offline optimization procedure using binary search and *RMSE* as evaluation criteria. The optimization method can be any chosen method as long as it has a fast convergence rate to the optimal value. The training of the algorithms will not be run often, so it does not matter if it takes a long time (i.e., hours) to complete. This step of optimizing the weight vector however will be executed frequently and therefore should be as fast a possible (i.e., complete in a matter of seconds). When the weight vector has been optimized, it can be used to generate the final recommendations by applying it to aggregate the algorithm models which were trained on the complete rating dataset (below in the figure).

When applying optimization methods, part of the data is often dedicated for the evaluation of the objective function. Because of our proposed procedure of training and testing on fold datasets, all of the rating data can still be integrated in the models of the final (non-fold) recommendation algorithms. Furthermore because of the high-level structure, the hybrid optimization procedure can be applied to different methods of recommendation strategies (e.g., rating prediction versus item prediction).

3.1.1 Overfitting

One potential problem of the optimization procedure as described above is *overfitting*. Overfitting is the phenomenon that the model performs poorly on test data, despite being able to predict the training data almost perfectly [33]. Overfitting originates from training with too few data or optimizing a model too hard so that the learned results are too specific to the training data and therefore not generalizable to new data. The first problem, i.e., training on too few data, can be easily circumvented by applying the optimization procedure only for users with a sufficient number of ratings. For users with a low number of ratings the system is prone to overfit. Say a user has provided 5 ratings to the system, and assume a train-test split ratio of 60%. Only 3 ratings would be used to train the fold datasets and the remaining 2 would be used for the evaluation (in the test datasets). Optimizing a weight vector over multiple algorithms based on only 2 data points will not yield generalizable results. Users with too few ratings could be handled in two ways: require more ratings from the user before calculating the recommendations i.e., do nothing, or train the models on the few ratings available and for weights use a default pre-computed weight vector that has shown to yield good results for many other users of the system (i.e., non-personalized approach).

The other problem, causing the models to overfit, is too train the models too specific to the input data which again prevents generalizability. One way to prevent this overfitting scenario is to not use all data for the optimization but instead only a random subsample of the dataset. Creating (and optimizing for) multiple randomly subsampled datasets at the same time is even better, and so this is implemented in the approach described in this work. By optimizing for multiple folds at the same time and taking into account the agreement of the evaluation of the models (i.e., the variance) the optimization process is forced to generalize over the complete rating dataset as well as over random subsampled subsets. If the provided ratings are good indicators of possible future ratings, then the system should be able to generalize. By changing the number of fold datasets and the train-test split ratio the process can be fine-tuned for the specific needs and properties of every use case.

Using very few folds, say in the extreme case only 1, requires very few computational effort but may cause overfitting the sampled data. Chances are that the sampled train rating data are all special cases or outliers which are bad indicators for future performance. To reduce the chance of bad subsampling, the number of folds can be increased.

The opposite extreme case where the number of folds (and train-test ratio) are so high that every data point (i.e., rating) in a certain fold serves as the test set while all other ratings make out the rating dataset is referred to in literature as *Leave-one-out* cross-validation [30]. While this method is very thorough in using all data, it is computationally very expensive. A well-accepted meet-in-the-middle approach in recommender system literature is the k-fold cross validation method where k folds are generated and used for testing, k often set to 10 for robustness [18,42]. It improves the chance of generalizability (reduces overfitting) with only limited additional computational burden. In Fig. 1, 3-fold cross validation was used.

3.1.2 A responsive online recommender

In the introduction we defined our requirements for an online recommender system, one of which was *responsiveness* (**REQ1**). For the proposed hybrid optimization process both slow and fast components were discussed i.e., the training of the models versus the optimizing of the weight vectors. By combining both components, the system can be made responsive, or at least *seem* responsive to the user.

As most recommender systems, the proposed system in this work suffers from the cold start syndrome [38]. Without any data no models can be trained, no weights optimized and therefore no recommendation can be generated. Two common ways of dealing with the cold start problem is either by presenting the user with a list of default non-personalized recommendations (e.g., most popular items), or not presenting any recommendations at all and requiring (more) data from the user before presenting any results.

As soon as data (i.e., user ratings) are available, the models can be trained. Since the optimization process requires the output of trained models, the initial training step must be completed first. While the initial training of the models may be slow, it will block the recommendation process only once i.e. when the models are trained on a new user for the first time. With the models trained, the optimization step is designed to complete almost instantly which can be leveraged to making the system feel responsive. For a system to feel responsive it must react to user input in almost real time. In the recommender system use case this translates to having the system react in real time to new ratings and so for every new item the user rates, the system must be able to present a new (or updated) recommendation list.

If we were to add new ratings to the training sets and require the models to be recomputed, the system would be too slow to react to new ratings in real time. Instead we propose to add new ratings to the test fold datasets. As shown in Fig. 2, by adding new rating directly to the test datasets, they affect the optimization of the weight vector which in its turn influences the final recommendation list. So by adding newly provided ratings to the test fold datasets and instantly re-optimizing the user's weight vector, the new rating can trigger changes in the final recommendation list.

Every now and then the individual models can be retrained offline (incorporating the new ratings since last training) and then be inserted back into the online system, all of this completely transparent to the user. That way, the system is capable of calculating powerful and complex models and at the same time respond in real time to provided user feedback (which was requirement **REQ1**).

3.2 Server-clients structure

An important requirement for online recommenders is scalability (defined as **REQ2** in the introduction). For online systems it is very hard to predict a realistic number of engaged users. There might be thousands of users or even millions depending on the popularity. Therefore, for online recommenders, scalability will be even more important than for closed environment recommenders. For a recommender to be scalable, its underlying model must be scalable i.e., able to handle a growing number of users or data without exponentially taking more time to calculate. When

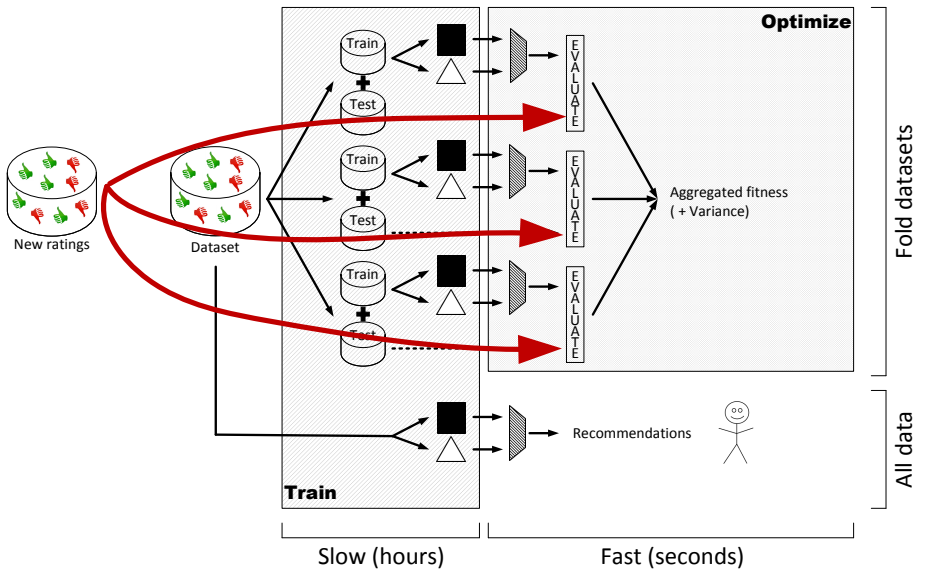

Fig. 2 The optimization process, detailing how new ratings are added to the test fold datasets where they can have an instant affect on the final recommendations without the need for retraining the individual models.

considering our hybrid model which integrates multiple fold datasets and various individual recommendation algorithms, it may seem like some compromise to scalability will have to be made. In this section however, we show that by adopting a client-server architectural design, our hybrid system parallelizes extremely well, which allows it to scale naturally to available hardware and large user bases.

Fig. 3 illustrates the architectural design applied to the scenario from Fig. 1. There are 3 fold datasets and 2 individual recommendation algorithms (the symbolical black square and white triangle). For each of the training fold datasets, instances of both algorithms are trained in addition to the instances trained on the complete ratings dataset, bringing the total number of algorithm instances for this scenario to *8*. The main principle of the client-server approach is to isolate parts of the system that can run in parallel into their own separate processes. The figure shows the main server process i.e., the *Hybrid Model*, which stores the test fold datasets, has the functionality to optimize weight vectors (*Optimizer* component) and combine the final recommendations (*Combiner* component), and communicates with the instances of the individual algorithms.

Instead of running the client instances in the same process as the server (and thus limiting their ability to parallelize), they are executed in separate processes and communication is handled by *Algorithm Proxy* components. Communicating by means of proxy components allows the *Hybrid Model* to interact with the algorithm instances transparent of their true location, which may be on the same computation node, another node in the local network, or a random PC across the Internet.

The main control flow of the recommendation process is depicted in Fig. 4. When the system is first started, the proxies are initialized by the *Proxy Generator*

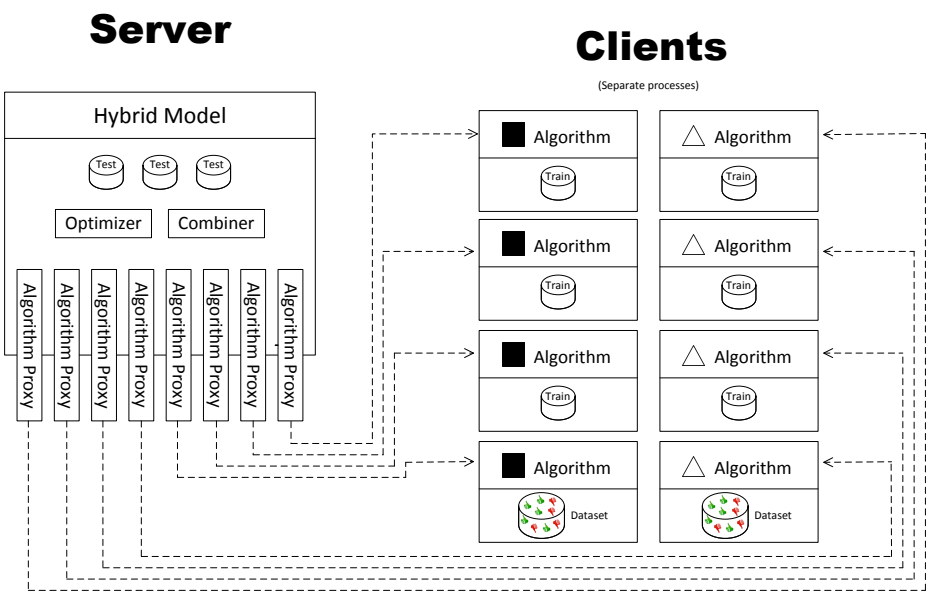

Fig. 3 The Client-server architectural design, illustrating how each individual algorithm executes in a separate process (potentially on a different machine) and communicates with the *Hybrid Model* by means of proxy objects.

component. This component initializes the processes of the individual recommendation algorithms across available computing nodes. If multiple computing nodes are available, the component attempts to distribute the processes over the nodes as equally as possible. A link to the proxy objects is provided back to the *Hybrid Model* to allow future communication. When the model is initialized, ratings can be provided, which are processed in train and test datasets and passed through the appropriate algorithm proxies (train fold datasets to the fold algorithms, full rating dataset to the non-fold algorithms).

When the *Train()* command is provided, the command is delegated to all the algorithm proxies in parallel, which causes all of them to start training at the same time in their separate processes. Since the total time will be equal to the maximum execution time over all trained algorithms, this phase may take long (i.e., hours) to complete. When all algorithms have completed training, the *Hybrid Model* is notified and may start accepting recommendation requests for specific users.

The request for recommendations for a specific user triggers a chain of events eventually leading to the final recommendations. First the weight vector for that user must be calculated (if not already available) by the optimization procedure in the *Hybrid Model*. The optimization requires the test fold datasets (which are available in the *Hybrid Model*) and the recommendations for the algorithms trained on the train fold datasets. With the weight vector available, all that remains is to apply it in the final phase which is the combination of the results of the individual recommendation algorithms trained on the complete rating dataset.

The main advantage of our client-server architecture is the deployment flexibility. Because the principal calculating components are decoupled and running in

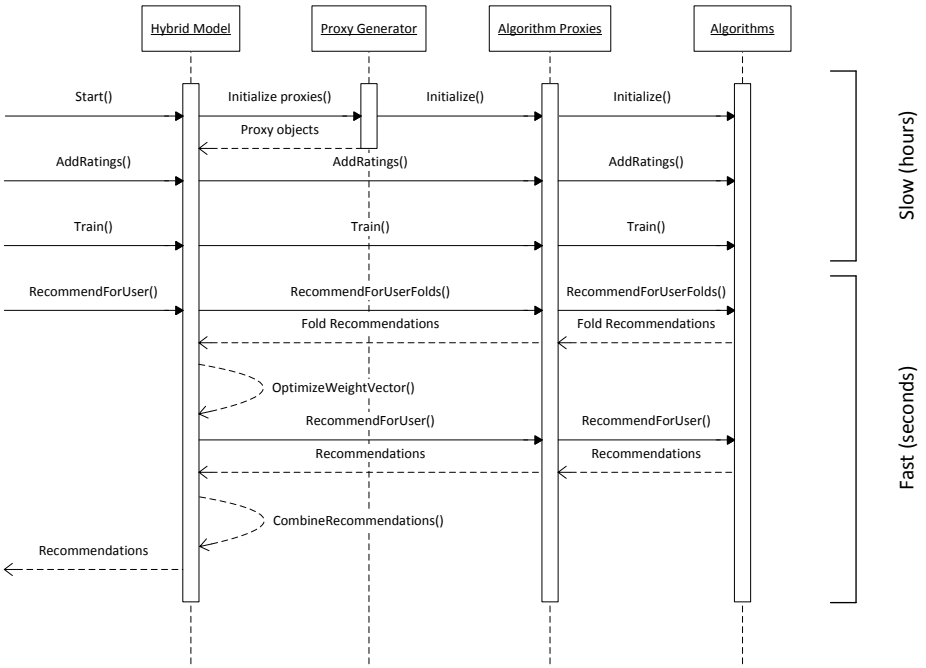

Fig. 4 Sequence diagram illustrating the execution flow of the complete recommendation process from initialization to returning the recommendation results.

their separate processes controlled by one server (i.e., the *Hybrid Model*), they can be distributed across severs as desired. This allows to take into account the specific properties of the individual algorithms. Algorithms that require a lot of RAM memory may be deployed on dedicated machines, while disk-intensive algorithms may be deployed on machines with specially equipped hard drives. Furthermore all of the algorithms execute in parallel which reduces the main scalability of the system to the scalability of the least scalable integrated individual recommendation algorithm. The only computations affected by the number of integrated individual recommendation algorithms are the optimizing and combining processes in the *Hybrid Model*. In Section 5 however, we show how the impact of these effects on the general scalability of the system is rather limited.

3.2.1 Performance optimization: prefetching

Implementing the above described approach requires some optimization to avoid bottlenecks as network speed compromising the performance of the system. To illustrate the effect of network speed on overall performance, consider the following Python code fragment. Assume we are in a rating prediction scenario and are evaluating the quality of a weight vector using the popular *RMSE* metric as objective function.

```
def eval_weights(user, weights, algorithm_proxies, test_fold_dataset):
    rmse = 0.0
    count = 0.0
    for (rating, item) in test_fold_dataset[user]:
        prediction = predict(user, item, weights, algorithm_proxies)
        error = prediction - rating
        rmse += error * error
        count += 1
    rmse = math.sqrt(rmse / count)

def predict(user, item, weights, algorithm_proxies):
    numerator = 0.0
    denominator = 0.0
    for algorithm_proxy in algorithm_proxies:
        weight = weights[algorithm_proxy]
        prediction = algorithm_proxy.get_recommendation(user, item)
        numerator += prediction * weight
        denominator += weight
    weighted_prediction_value = numerator / denominator
    return weighted_prediction_value
```

The code fragment displays two functions which are needed for the evaluation of a given weight vector *weights*. Here the *RMSE* value serves as fitness value allowing to compare (and therefore optimize) the quality of different weight vectors. *RMSE* is calculated by comparing all the ratings of the given user in the test fold dataset with the predicted score of the algorithms. The predicted score is calculated using a simple weighted average formula to aggregate the individual prediction scores of the recommendation algorithms.

Although this naive code fragment functions correctly, it will not be very efficient considering our client-server architecture. The reason for this is the following line of code.

```
        prediction = algorithm_proxy.get_recommendation(user, item)
```

While the *eval_weights* and *predict* functions will run in the server process of the *Hybrid Model* (in the Optimizer component), the above line of code requests the prediction value for a certain user and item from an algorithm proxy, triggering the request to be passed to the actual process of the recommendation algorithm which may be running on an other computer. So every time the above line of code is called, in the background (transparently to the *Hybrid Model*) a network connection may be set up and teared down for the required communication between the algorithm proxies and the actual algorithm processes. Individually such a request is considerably fast, but in the above code example the request would be called for every rating in the test fold dataset and for every algorithm proxy which will limit the performance of the optimization method in the *Hybrid Model*. We implemented the proposed approach in Python using the *XML-RPC*[4] package for the communication between the algorithm proxies and the actual algorithm processes. The *XML-RPC* package wraps every request as an XML document that

[4] http://docs.python.org/2/library/xmlrpclib.html

is transported over HTTP. While the overhead of one request is small, the accumulated overhead of many such requests greatly influenced the end performance of our system. Since the performance of the optimizing part of the *Hybrid Model* should be very high to meet our **REQ1** requirement, a prefetching strategy was devised.

Instead of requesting the prediction values at the moment they are needed in the calculation, it proved better to request them all at once before the start of the calculations. By implementing the following function in the algorithm instances, only one data request per recommendation algorithm needs to be called which transfers larger (but fewer) chunks of data over the network connection.

```
def get_recommendation_multiple_items(user, items)
```

Implementing a prefetching approach proved necessary to guarantee both the performance of the *Optimizer* and *Combiner* components of the *Hybrid model*.

3.2.2 Limitations

While the proposed model aims for flexibility and performance, its complexity imposes heavy constraints on underlying hardware configurations. For the client-server architecture to be truly effective, every process should be able to run on a dedicated processor core. Since data is replicated in multiple folds and over multiple instances, the available RAM memory of the system will also be a limiting factor. Although our proposed approach in theory can be deployed on any hardware configuration, an optimal hardware configuration would be a cluster of computing nodes with a total number of dedicated processors of at least the number of spawned processes (equation 1), linked together with a high-speed network connection.

$$required\ processor\ cores = 1 + ((\#folds + 1) \times \#algorithms) \tag{1}$$

4 Online user-interface for movie recommendations

Previous sections addressed the online requirements of responsiveness (**REQ1**) and scalability (**REQ2**), which were both focused on the system side of the recommender. The remaining requirements are the transparency of the system (**REQ3**) and user control (**REQ4**), both of which directly affect the user side of the recommender and thus need to be integrated in the interaction process between user and system i.e., the user-interface (UI). In this section we propose a UI for our online hybrid optimized movie recommender system called 'MovieBrain', we illustrate how users can browse through a movie collection, provide ratings, inspect their recommendations and most importantly interact with their recommendations (i.e., take control) in a transparent way. Since focus is on online systems, the UI was constructed with web based technologies as HTML, PHP, MySQL and Javascript, all of which are common in an online setup. The UI source code has been made available on the Github platform[5].

[5] https://github.com/sidooms/Recsys-frontend

4.1 Browsing and rating

The first and most basic function the front-end of a recommender system needs to provide, is browsing through the collection of items and allowing user preferences to be indicated. Collecting user feedback is an often neglected but very important part of the recommender system process. Users need sufficient information about the items at hand and require an intuitive method of expressing their opinions.

Fig. 5 shows a screenshot of the user-interface (UI) that was developed for the system described in this work. The UI allows to browse through a collection of movies and for each movie presents detailed information e.g., director, cast, genre, etc. Movies are presented as a list that can either be ordered randomly or by year. The option to browse the list in a random manner is important to avoid presentation bias to influence the user ratings. If only very recent and popular movies would be displayed, users would be more inclined to rate those movies which would bias their final recommendations. The MovieBrain front-end also allows to directly search for movies by entering (a portion of) their title in the search form displayed on the top of the website.

Users can express their preferences towards the movies in the collection by using the thumbs up/down feedback system provided on the right hand side of every movie information panel. A 5-star rating system is more commonly used in recommender system scenarios but often fails to produce more fine-grained ratings than a simple thumbs up/down system because users mostly use the extreme rating values [14]. Therefore in the interface we integrated a thumbs up/down system, but ultimately the type of feedback system will mostly depend on the input required by the algorithms that are actually integrated in the recommender system.

4.2 Recommendation list

4.2.1 Individual recommendation lists

When a user has provided a sufficient amount of ratings, the system can start training its underlying models (see previous sections). Our UI integrates an option to inspect the individual recommendation results in a similar way as the original movie collection was displayed (Fig. 6). A select box allows to choose any of the algorithms integrated in the system (3 in the figure). When a specific algorithm is selected, the movie collection is shown as a list but this time ordered according to the recommendation output of the algorithm. Every movie in the list can also be rated to allow users to fine-tune their preferences.

4.2.2 Hybrid recommendation list

The most interesting recommendation list in the MovieBrain system is however the hybrid recommendation list, which is the final result of the MovieBrain system after having optimized a user's weight vector and combined the individual recommendation lists. Fig. 7 shows the presentation of one item in the hybrid recommendation list. The presentation of a recommended movie is again almost identical to that of a movie while browsing the movie collection. The only difference

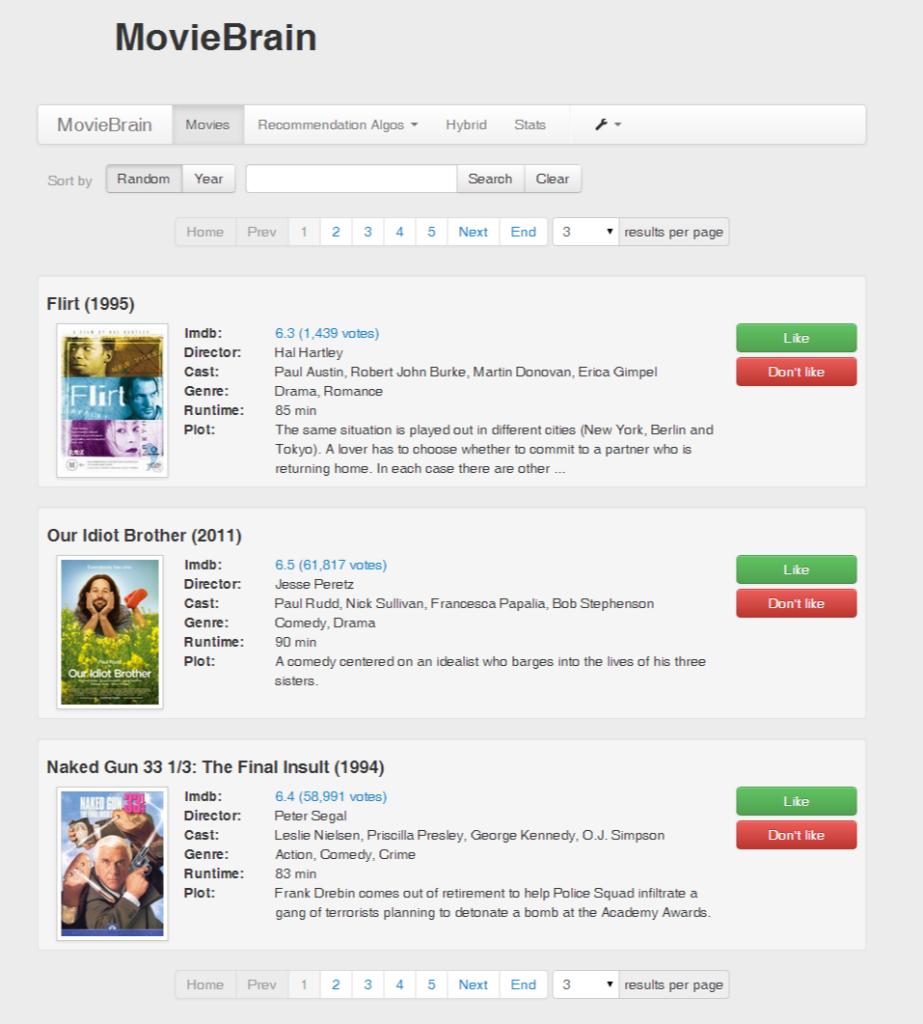

Fig. 5 A screenshot of the MovieBrain user-interface allowing users to browse through a collection of movies, inspect detailed movie information and express their movie preferences by means of a thumbs up/down feedback system.

is the addition of the '*Rec value*' property which indicates the final recommendation score calculated by the hybrid system for the concerning movie. At the end of the recommendation score label there is a *Show explanation* link which triggers information about the hybrid calculation process to become visible (Fig. 7 bottom image). The provided information details the weight vector of the user together with the individual scores for each of the recommendation algorithms and the applied aggregation function (weighted average in this case). While this kind of information is not fit to show to normal users of the system, it does illustrate an option to provide transparency (requirement **REQ3**) in to the recommendation

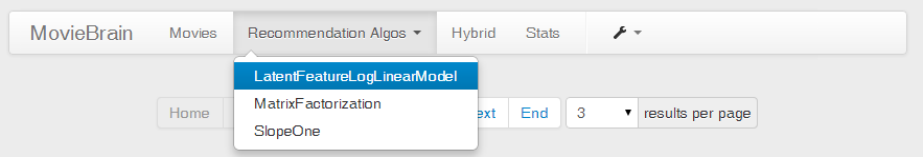

Fig. 6 A screenshot of the MovieBrain user-interface illustrating the selection feature of the individual recommendation algorithms.

process which could for example be used by system administrators to inspect and configure the hybrid model.

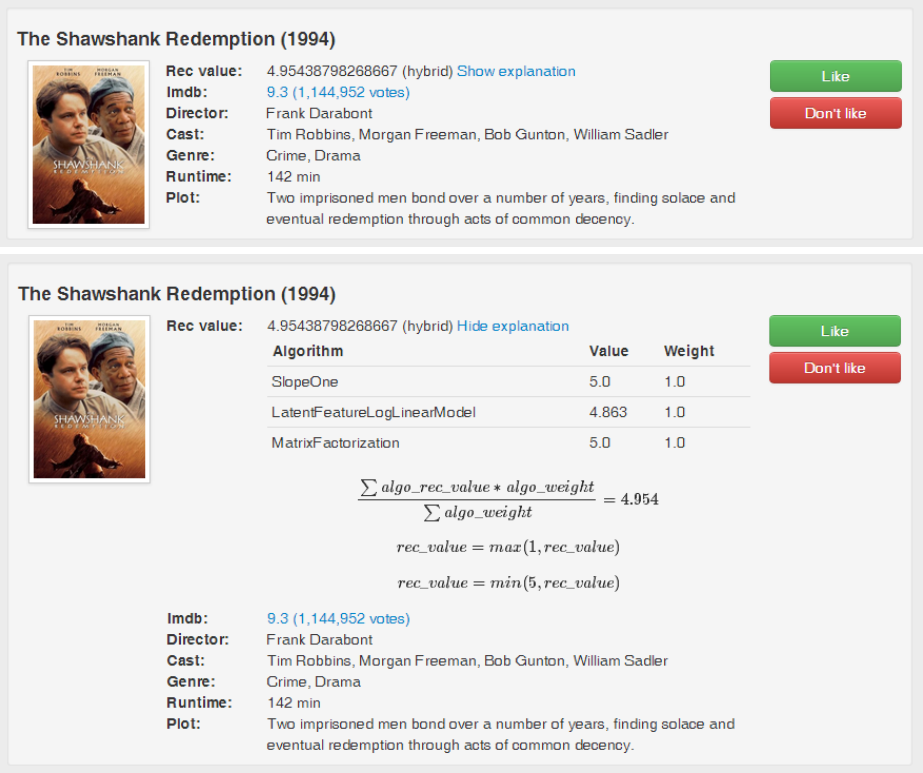

Fig. 7 A screenshot of the MovieBrain user-interface illustrating one movie item in the hybrid recommendation list. The bottom image shows the expanded text after clicking on the *Show explanation* link in the above image.

4.3 User control and transparency

The final requirement for online recommenders as defined in the introduction is to allow users control over their recommendations (**REQ4**). For most recommender

systems, the internal process of calculating the recommendations is shielded from the users (i.e., black box approach) or at least oversimplified (e.g., *People who liked this also liked...*) and users have no means of controlling or influencing the process other than by providing ratings. The MovieBrain system however, is based on a *weighted* hybrid strategy which by its very nature offers the components allowing user control: the weight vectors. The weights in the weight vector, model the contribution of each individual recommendation algorithm to the final hybrid recommendation output and thus can be used as proxies for the *importance* of the algorithm for a specific user. By allowing users not only to inspect their weight vector but also to modify the individual weights manually, users can directly influence and fine-tune their recommendation lists to their specific (and maybe contextual) interests.

At the top of the MovieBrain user-interface there is a menu item *Stats* which directs users to a webpage where their weight vectors can be visually inspected and manipulated in a very intuitive interaction process i.e., sliders (Fig. 8).

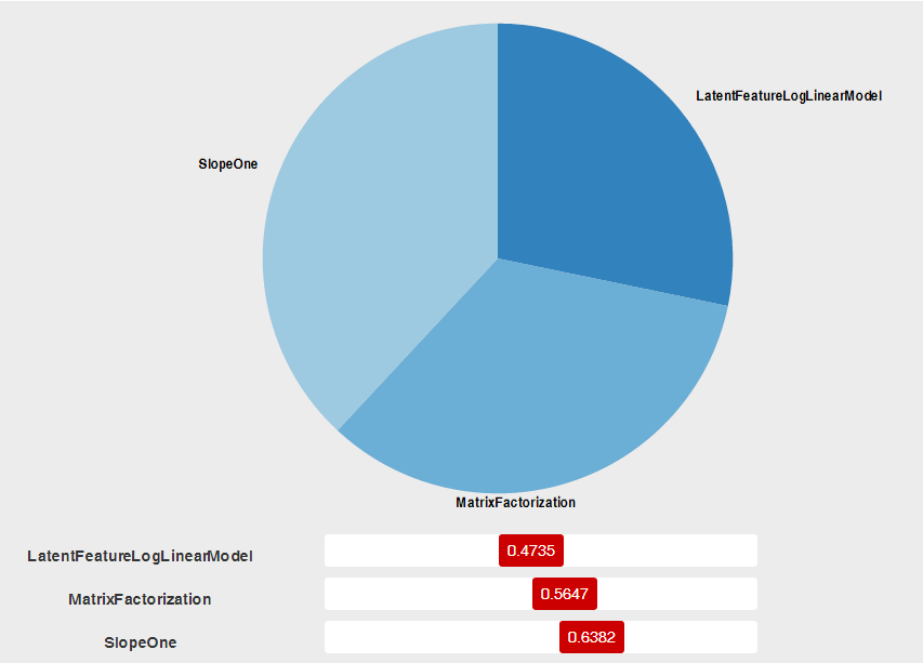

Fig. 8 A screenshot of the MovieBrain user-interface illustrating the control over the weight vector for a single user. Users can manually override their system-calculated weight vector using designated weight sliders.

For increased comparability, weights are scaled to the interval $[0, 1]$. The pie chart in Fig. 8 visualizes the normalized end result taking all weights into account. Modifying a weight value by interacting with the slider component will cause the pie chart to be redrawn in real time. This allows users to easily estimate the effect of a single algorithm, because the more algorithms are integrated in the system, the less their individual influence on the end result.

In the previous sections we detailed how the hybrid system can automatically optimize the weight vector for a specific user. This optimization will however be based on some measurable evaluation metric e.g., *RMSE* which might not correspond to the users' expectations (maybe users prefers *serendipity* instead of *recommendation accuracy*). By allowing to tweak the weight vector manually and thereby overriding the automatically determined weight vector, users are able to fine-tune their recommendations to their own specific expectations. Note that in Section 3 we explained how the process of calculating the weight vector and combining the final recommendations could (and should) be computed very fast. Therefore when a user overrides the weight vector, new hybrid recommendations can be generated instantly which provides the user-system interaction process a very natural feel.

4.4 System experts versus normal users

While inspecting and manipulating the weight vector for the individual algorithms is indeed a way of introducing control and transparency to the system, the above approach would fail for normal non-technical users. For system experts or researchers who are installing the recommender system, direct control over the weight vector will be very interesting, but for normal users who are oblivious to the technicalities of the recommender system, manually adapting the weights may be a too technical task.

What is possible however, is to simplify the algorithms to the users e.g., instead of saying 'content-based recommender' we could say 'movies similar to the ones you liked'. By translating algorithms to their most defining feature, the effect of changing the weights could be made understandable for normal users. In Fig. 9 this scenario is illustrated for three recommendation algorithms. *Novelty* could refer to an algorithm focusing on (i.e., predicting more) novel movies and the same for *Popularity* and *Similarity*.

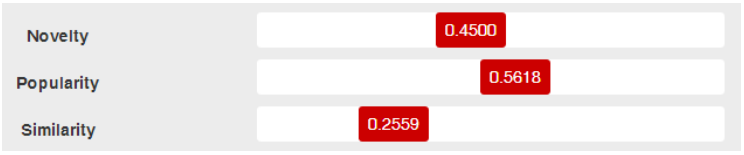

Fig. 9 A screenshot of the MovieBrain user-interface illustrating how algorithms can be translated to their most defining features to make changing the weights more interpretable by non-technical users.

Recommendation algorithms that are not easily translated to an understandable concept for normal users e.g., MatrixFactorization could simply be referred to as 'Best system guess' or 'Determined by magic' as used sometimes in Google services (e.g. Fig. 10, which was a sort option in the former Google Reader platform).

As an extension, the system could offer various post-recommendation filters like genre filters (often demanded by users for movie recommendation scenarios [17]) which can be easily implemented in the user-interface without any modifications

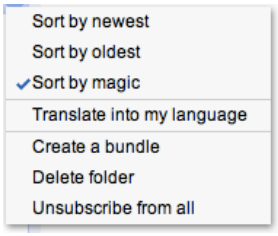

Fig. 10 Popup on the former Google Reader platform providing the option to *Sort by magic*.

to the underlying recommender system. Ultimately, the combination of both using filters and manipulating the weight vector through the user-interface provides users with the necessary tools to interactively tailor their recommendations to their own interest in a transparent way.

5 Results

In previous work [16], which focused on *offline* optimization, we evaluated the quality of our hybridization approach and experimented with different optimization techniques. In this work, since focus is on *online* optimization, we evaluate the scalability and performance of the system. The concept of scalability can focus on two scenarios: *strong scaling* or *weak scaling* [22]. In a *strong scaling* scenario, the amount of work stays constant while the number of *workers* (e.g., computing nodes, processor cores, etc.) varies. The term *weak scaling* refers to the opposite scenario where the number of workers is constant while the amount of work changes. So when a system is referred to as 'scalable' it could mean two things. Either the system is capable of scaling across multiple computing nodes thereby reducing the total execution time through parallel computing (i.e., strong scaling), or the system is capable of processing increasingly bigger workloads without exponentially increasing the execution time (i.e., weak scaling). Either scenario is interesting for our online system and so in this section we investigate both.

All the experiments detailed in this section were run on the High Performance Computing (HPC) infrastructure available for researchers at our university[6]. The computing nodes deployed in the experiments have the following specifications.

- *CPU*: dual-socket quad-core Intel Xeon L5420 (Intel Core microarchitecture, 2.5 GHz, 6 MB L2 cache per quad-core chip), thus 8 cores / node
- *memory*: 16 GB RAM (DDR2 FB-DIMM PC-5300 CL5)

Computing nodes are interconnected by an Infiniband (i.e., high-speed) network and each dispose of a local hard disk (private storage) and have access to shared storage (GPFS) as well.

In the experiments, the complete recommendation process (from initialization to the generation of the final recommendations) is deployed in various experimental configurations. We used the MovieTweetings dataset [15] as simulation data for these configurations. The MovieTweetings dataset is a collection of movie ratings

[6] http://www.ugent.be/hpc/en

presented in a similar form as the MovieLens dataset [23,6,32], but focuses more on present-day popular and recent movies. The dataset extracts ratings from posts (i.e., tweets) on the popular Twitter platform[7] and continues to grow in size. For the experiments in this section, the $200K$ snapshot was used which includes $200,000$ ratings by $25,011$ users for $14,732$ movies. A split ratio of 6:4 was set for the train-test fold datasets. For more information on the MovieTweetings dataset we refer to our previous work ([15]).

Our hybrid optimized approach integrated individual recommendation algorithms as black boxes i.e., only the input and output of the algorithms are taken into account by the system without knowledge of the internal recommendation calculation process. Because of this approach there are no restrictions towards the type of recommendation algorithms that can be integrated. To illustrate this behavior, in the following experiments we use (rating prediction) recommendation algorithms from the open source MyMediaLite[8] library [20] which provides implementations for the most common recommendation algorithms used in research. As evaluation function for the optimization process (see section 3), *RMSE* was implemented. The *StochasticHillClimber* method (parameter *MaxEvaluations*=1000) from PyBrain[9], a modular machine learning library for Python, was integrated as optimization function.

5.1 Strong scalability

To investigate the strong scaling ability of our system, we experiment with deploying the system on a varying number of computing nodes while keeping the workload constant. The experimental setup is defined in the following list.

- *Dataset*: $200K$ MovieTweetings snapshot
- *Algorithms*: *MatrixFactorization*, *SlopeOne*, *LatentFeatureLogLinearModel*
- *Computing nodes*: 1, 2, 3, 4, 5 (8 cores per node)
- *Fold datasets*: 2, 4

The 3 MyMediaLite algorithms were selected based on their divergent properties regarding complexity, execution time and RAM consumption as detailed by Table 1. Default initialization parameters where used as set in MyMediaLite version 3.10.

	Complexity	**Time**	**RAM**
MatrixFactorization	complex	fast	low
SlopeOne	simple	fast	low
LatentFeatureLogLinearModel	complex	slow	high

Table 1 The divergent properties regarding complexity, execution time and RAM consumption for 3 rating prediction algorithms from the MyMediaLite recommendation library.

The experiment begins with the startup of the system: computing nodes are initialized and algorithm proxies (see Section 3) are constructed. When the system

[7] http://www.twitter.com
[8] http://www.mymedialite.net
[9] http://pybrain.org

is ready to accept ratings, the rating dataset MovieTweetings is provided and the individual algorithm models are trained on their (fold) datasets. Then for 100 randomly selected users (each having more than 20 ratings) the system is sequentially requested to predict the recommendation value for one fixed item. Doing so, triggers the system to calculate (i.e., optimize) the weight vectors for these users and combine their final hybrid recommendation lists.

The experiment was repeated with 1, 2, 3, 4, and 5 computing nodes and for two fold dataset settings: 2 and 4. For each of these configurations, the execution times of the individual phases of the recommendation process were measured and displayed in Fig. 11 (exact numbers available in Table 2). The 4 fold, 1 PC configuration failed to complete because the required amount of RAM exceeded the available RAM in a single computing node (16GB).

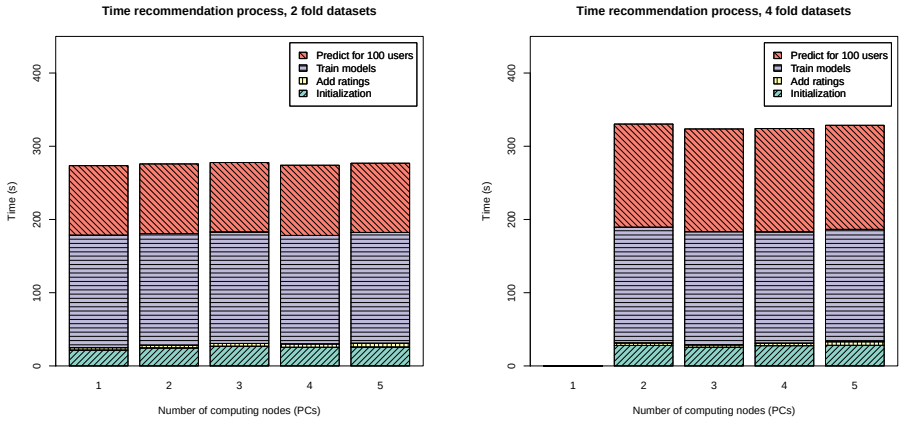

Fig. 11 The execution times of the individual phases of the complete recommendation process deployed on hardware configurations ranging from 1 to 5 computing nodes and for 2 (left) or 4 (right) fold datasets.

From the figure the initialization time for the different configurations seems identical, but closer inspection reveals a small increase for configurations of more than 1 computing node. This increase in time is caused by the required extra network communication overhead that is needed to signal the other computing nodes. For the same reason also the time for the adding of the ratings increases (although very limited). The time to train the models and the final prediction time interestingly remain unchanged for an increasing number of computing nodes (both for the 2 fold and 4 fold results). This observed behavior supports our claim that when all processes in the system are able to run in parallel (each on its own dedicated processor core), the end performance would only be limited by the slowest integrated individual recommendation algorithm. Table 3 lists for every experimental configuration the consequential number of parallel spawned processes versus the number of available processors. Only the single computing node configurations require more processors than available, and so for these conditions the execution times may be suboptimal. The effect is very limited visible in the training time which is increased by a few seconds. Among the three chosen algorithms for this

	# Folds = 2				
# PCs	**1**	**2**	**3**	**4**	**5**
Start time	21.6	24.5	26.8	25.4	25.7
Rating time	3.4	3.4	3.9	4.7	5.2
Train time	153.5	152.8	152.2	148.1	151.3
Predict time	95.0	95.2	95.0	95.9	94.6
Total time	**273.5**	**275.9**	**277.8**	**274.1**	**276.8**
	# Folds = 4				
# PCs	**1**	**2**	**3**	**4**	**5**
Start time	****	27.9	25.4	27.7	28.0
Rating time	****	3.4	3.4	3.3	5.0
Train time	****	158.0	154.5	152.0	153.4
Predict time	****	141.0	140.3	141.2	142.2
Total time	****	**330.3**	**323.6**	**324.2**	**328.6**

Table 2 The execution times of the individual phases of the complete recommendation process deployed on hardware configurations ranging from 1 to 5 computing nodes and for 2 or 4 fold datasets.

experiment, 2 of them finish fast, which means that 6 out of the 10 parallel computing processes will finish fast, allowing the 2 extra processes to start with only a few seconds delay. For the other (more than 1 PC) configurations the total training time will be equal to the time it takes for the slowest algorithm (i.e., *LatentFeatureLogLinearModel*) to complete.

Folds \PCs	**1**	**2**	**3**	**4**	**5**
2	**10/8**	10/16	10/24	10/32	10/40
4	**16/8**	16/16	16/24	16/32	16/40

Table 3 The number of spawned parallel processes versus the number of available processor cores (each PC has 8 cores) for the different experimental configurations.

While the total system execution time does not decrease with an increased number of computing nodes (as expected), it is also interesting to note that it does not increase. Scaling a software system over multiple computing nodes may often increase the communication overhead required to manage the running instances and therefore introduce some form of delay linked with the number of computing nodes. Thanks to a high-speed network infrastructure and some implementation optimizations (see prefetching in Section 3.2.1) we were able to reduce the parallel overhead to an absolute minimum.

When comparing the 2 fold configuration with the 4 fold results, very similar graphs can be noted. The time to train the models for a 4 fold configuration is equal to the time for the 2 fold configuration, again illustrating how the training time is independent of the number of folds, algorithms or available computing nodes. When a sufficient number of parallel processors are available, the training time will equal the time for the slowest individual recommendation algorithm to complete its work. The main difference between the 2 and 4 fold configurations is the difference in prediction time. Because the 4 fold configuration has more fold datasets, the optimizer will have to take more data into account to optimize the user weight vectors, which explains the increase in execution time.

5.2 Weak scalability

To experiment with the weak scaling ability of our system, we perform a similar experiment but this time the number of computing nodes (i.e., workers) stays constant while varying the dataset size (i.e., amount of work to be processed). The following list describes the experimental setup.

- *Dataset*: $40K, 80K, 120K, 160K, 200K$ MovieTweetings snapshots
- *Algorithms*: *MatrixFactorization*, *SlopeOne*, *LatentFeatureLogLinearModel*
- *Computing nodes*: 5
- *Fold datasets*: 2, 4

The algorithms used in this experiment are identical to those of the previous experiment, again using their default initialization parameters as set in MyMediaLite version 3.10. The properties of the specific MovieTweetings snapshots are detailed in Table 4.

	40K	**80K**	**120K**	**160K**	**200K**
# Ratings	40,000	80,000	120,000	160,000	200,000
# Users	9,063	14,180	19,337	22,259	25,011
# Items	6,798	9,419	11,595	13,445	14,732

Table 4 The basic properties of the different MovieTweetings snapshots used in this experiment.

Just as before, the system was instructed to run through the consecutive phases of initialization, adding ratings, training models and predicting for 100 randomly selected users with more than 20 ratings. The experiment was repeated for iteratively growing dataset sizes and for 2 and 4 fold datasets. The execution times of the individual phases were measured and displayed in Fig. 12 and detailed in Table. 5.

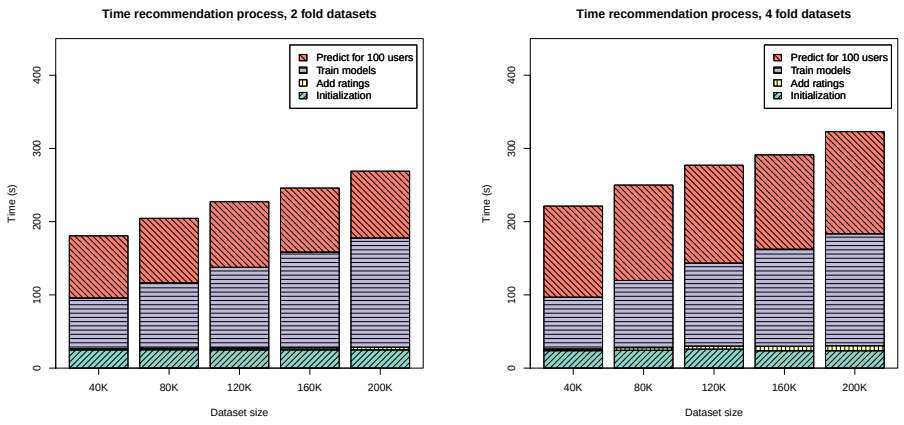

Fig. 12 The execution times of the individual phases of the complete recommendation process for different sizes of the rating dataset ($40K$ to $200K$) and for 2 (left) or 4 (right) fold datasets.

	# Folds = 2				
Dataset size	**40K**	**80K**	**120K**	**160K**	**200K**
Start time	24.9	25.1	24.9	25.0	24.9
Rating time	1.6	2.0	2.5	2.8	3.2
Train time	69.2	89.4	110.2	130.6	149.4
Predict time	85.0	88.1	89.7	87.5	91.4
Total time	**180.7**	**204.5**	**227.3**	**245.9**	**268.9**
	# Folds = 4				
Dataset size	**40K**	**80K**	**120K**	**160K**	**200K**
Start time	23.8	24.3	25.8	23.8	23.8
Rating time	2.2	3.7	4.1	6.0	6.8
Train time	70.7	91.7	113.4	132.5	152.8
Predict time	124.5	130.3	133.8	128.9	139.5
Total time	**221.3**	**250.0**	**277.1**	**291.3**	**322.9**

Table 5 The execution times of the individual phases of the complete recommendation process deployed on 5 computing nodes for varying dataset sizes ($40K$ to $200K$) and for 2 or 4 fold datasets.

For the weak scaling experiment every configuration was run on 5 computing nodes each featuring 8 processing cores and so this time the number of spawned processes did not exceed the number of available cores. Since all processes could be divided over 5 different computing nodes no RAM issues occurred and every configuration was able to complete.

The initialization time follows the same patterns as in previous results, but the time to add the ratings increases more. This makes perfect sense as the increasing datasets will require more time to process. While the time to train the models remained the same in previous results, here the training time increases linearly with the increasing dataset size. This was again to be expected since the end performance of the system will be depending on its slowest component, the *LatentFeatureLogLinearModel* algorithm, which takes linearly more time to train for increasing rating dataset sizes. The time to train for the 2 fold dataset configuration is again equal to the 4 fold dataset configuration as was observed in the strong scaling scenario.

Two observations can be noted regarding the prediction times. The time it takes to sequentially predict for 100 random users is again higher for the 4 fold configuration than the 2 fold, which is caused by the increased complexity in optimizing the user weights vector over multiple fold datasets. Secondly, the prediction time also seems to increase as the dataset size grows larger. The reason for this is linked with the selection of the random users for each dataset. While in previous experiment the 100 random users were selected and then re-used for the different configurations, here the random selection process had to be repeated for every dataset size (a user selected in the $200K$ snapshot might not be present in the $40K$ snapshot). We counted for each selection of 100 random users per dataset size the total number of ratings for those users and found it to be highly correlated with the final prediction execution time. More ratings will lead to larger cardinalities of the test fold datasets used for the optimization of the user weight vectors, which again increases the complexity of the prediction task. Because in the larger dataset sizes there are more users with >20 ratings, the chance of randomly selecting users with more ratings in total is larger than for the small dataset sizes.

6 Discussion

By means of the experimental evaluations in the previous section we tried to get the recommender of out of the lab and see how it performs in a real-world scenario by feeding it a real dataset and actually deploying it on multiple computing nodes. We have shown that the performance and scalability of the system can indeed be reduced to the performance of the slowest integrated individual recommendation algorithm as long as the underlying hardware configuration provides sufficient parallel processing power. In Section 3.2.2 equation 1 was provided to determine the number of needed parallel processor cores taking into account the number of fold datasets and individual recommendation algorithms.

The complexity of the hybrid optimizer did however turn out to be influenced by the number of used fold datasets. Because of this, a trade-off will have to be made between having many folds (e.g., 10) to reduce the chance of overfitting the model and having a small number of folds to reduce the complexity (i.e., increasing the speed) of the optimizer component. The number of fold datasets should therefore be sufficiently large while making sure the performance of the optimizer component can still be considered fast enough to guarantee instant responsiveness to new ratings (**REQ1** in Section 3.1.2).

In the end, the hybrid optimizing system presented in this work offers sufficient flexibility for a customized configuration for any specific use case. Configurable components include the individual recommendation algorithms, the number of folds, the evaluation metric (i.e., what should the system optimize for?) and the optimization method itself.

7 Conclusion

With this work, we tried to take another step towards deploying automated self-learning hybrid recommender systems in real-world scenarios. We discussed the architectural design of our hybrid optimization strategy and detailed realistic implementation issues to assure the system meets the proposed online requirements for a movie recommendation scenario: responsiveness (**REQ1**), scalability (**REQ2**), system transparency (**REQ3**) and user control (**REQ4**). By adopting a server-client architecture we showed how the system can be distributed across multiple computing nodes in a very flexible and transparent way, allowing multiple recommendation algorithms to run in parallel for optimal performance. We illustrated how the results and internal processes of the system could be visualized to users in the form of a responsive web user-interface allowing user an advanced and intuitive level of control over their recommendation lists.

Through experimental evaluation we validated the architectural design and our claim that the performance and scalability of the system can be reduced to the performance and scalability of the worst (i.e., slowest) individual recommendation algorithm integrated in the hybrid system. The added overhead of the hybrid optimization was shown to be very limited as long as a sufficient number of computing nodes (or parallel processor cores) are available.

In future work we plan on making the system available to users in either an experimental or live online environment to evaluate the perceived user satisfaction and usability of our user-specific hybrid movie recommender system.

Acknowledgements The described research activities were funded by a PhD grant to Simon Dooms of the Agency for Innovation by Science and Technology (IWT Vlaanderen). This work was carried out using the Stevin Supercomputer Infrastructure at Ghent University, funded by Ghent University, the Hercules Foundation and the Flemish Government - department EWI.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. Knowledge and Data Engineering, IEEE Transactions on **17**(6), 734–749 (2005)
2. Adomavicius, G., Tuzhilin, A.: Context-aware recommender systems. In: Recommender systems handbook, pp. 217–253. Springer (2011)
3. Anand, S.S., Mobasher, B.: Intelligent techniques for web personalization. In: Proceedings of the 2003 international conference on Intelligent Techniques for Web Personalization, pp. 1–36. Springer-Verlag (2003)
4. Bao, X., Bergman, L., Thompson, R.: Stacking recommendation engines with additional meta-features. In: Proceedings of the third ACM conference on Recommender systems, pp. 109–116. ACM (2009)
5. Bellogín, A.: Performance prediction and evaluation in recommender systems: An information retrieval perspective. Ph.D. thesis, Universidad Autonoma de Madrid (2012)
6. Bobadilla, J., Serradilla, F., Bernal, J.: A new collaborative filtering metric that improves the behavior of recommender systems. Knowledge-Based Systems **23**(6), 520–528 (2010)
7. Brand, M.: Fast online svd revisions for lightweight recommender systems. In: SDM. SIAM (2003)
8. Burke, R.: Knowledge-based recommender systems. Encyclopedia of library and information systems **69**(Supplement 32), 175–186 (2000)
9. Burke, R.: Hybrid recommender systems: Survey and experiments. User modeling and user-adapted interaction **12**(4), 331–370 (2002)
10. Chandramouli, B., Levandoski, J.J., Eldawy, A., Mokbel, M.F.: Streamrec: a real-time recommender system. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 1243–1246. ACM (2011)
11. Das, A.S., Datar, M., Garg, A., Rajaram, S.: Google news personalization: scalable online collaborative filtering. In: Proceedings of the 16th international conference on World Wide Web, pp. 271–280. ACM (2007)
12. De Pessemier, T., Vanhecke, K., Dooms, S., Martens, L.: Content-based recommendation algorithms on the hadoop mapreduce framework. In: 7th International Conference on Web Information Systems and Technologies (WEBIST-2011), pp. 237–240. Ghent University, Department of Information technology (2011)
13. Dooms, S., Audenaert, P., Fostier, J., De Pessemier, T., Martens, L.: In-memory, distributed content-based recommender system. Journal of Intelligent Information Systems pp. 1–25 (2013)
14. Dooms, S., De Pessemier, T., Martens, L.: An online evaluation of explicit feedback mechanisms for recommender systems. In: 7th International Conference on Web Information Systems and Technologies (WEBIST-2011), pp. 391–394. Ghent University, Department of Information technology (2011)
15. Dooms, S., De Pessemier, T., Martens, L.: Movietweetings: a movie rating dataset collected from twitter. In: Workshop on Crowdsourcing and Human Computation for Recommender Systems, CrowdRec at RecSys, vol. 13 (2013)
16. Dooms, S., De Pessemier, T., Martens, L.: Offline optimization for user-specific hybrid recommender systems. Multimedia Tools and Applications pp. 1–24 (2013)
17. Dooms, S., De Pessemier, T., Verslype, D., Nelis, J., De Meulenaere, J., Van den Broeck, W., Martens, L., Develder, C.: Omus: an optimized multimedia service for the home environment. Multimedia Tools and Applications pp. 1–31 (2013)
18. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern classification. John Wiley & Sons (2012)
19. Ekstrand, M., Riedl, J.: When recommenders fail: predicting recommender failure for algorithm selection and combination. In: Proc. 6th ACM Conf. Recommender systems (RecSys 2012), pp. 233–236. ACM (2012)
20. Gantner, Z., Rendle, S., Freudenthaler, C., Schmidt-Thieme, L.: MyMediaLite: A free recommender system library. In: Proc. 5th ACM Conf. Recommender Systems (RecSys 2011) (2011)

21. Guy, I., Carmel, D.: Social recommender systems. In: Proceedings of the 20th international conference companion on World wide web, pp. 283–284. ACM (2011)
22. Hager, G., Wellein, G.: Introduction to high performance computing for scientists and engineers. CRC Press (2010)
23. Herlocker, J.L., Konstan, J.A., Borchers, A., Riedl, J.: An algorithmic framework for performing collaborative filtering. In: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pp. 230–237. ACM (1999)
24. Herlocker, J.L., Konstan, J.A., Riedl, J.: Explaining collaborative filtering recommendations. In: Proceedings of the 2000 ACM conference on Computer supported cooperative work, pp. 241–250. ACM (2000)
25. Jahrer, M., Töscher, A., Legenstein, R.: Combining predictions for accurate recommender systems. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 693–702. ACM (2010)
26. Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender systems: an introduction. Cambridge University Press (2010)
27. Jiang, J., Lu, J., Zhang, G., Long, G.: Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. In: Services (SERVICES), 2011 IEEE World Congress on, pp. 490–497. IEEE (2011)
28. Kille, B., Albayrak, S.: Modeling difficulty in recommender systems. In: Workshop on Recommendation Utility Evaluation: Beyond RMSE (RUE 2011), p. 30 (2012)
29. Lops, P., de Gemmis, M., Semeraro, G.: Content-based recommender systems: State of the art and trends. In: Recommender Systems Handbook, pp. 73–105. Springer (2011)
30. Moore, A.W.: Cross-validation for detecting and preventing overfitting. School of Computer Science Carneigie Mellon University (2001)
31. Nikulin, V., Huang, T.H., Ng, S.K., Rathnayake, S.I., McLachlan, G.J.: A very fast algorithm for matrix factorization. Statistics & Probability Letters **81**(7), 773–782 (2011)
32. Peralta, V.: Extraction and integration of movielens and imdb data. Tech. rep., Technical Report, Laboratoire PRiSM, Université de Versailles, France (2007)
33. Polikar, R.: Ensemble based systems in decision making. Circuits and Systems Magazine, IEEE **6**(3), 21–45 (2006)
34. Rendle, S., Schmidt-Thieme, L.: Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In: Proceedings of the 2008 ACM conference on Recommender systems, pp. 251–258. ACM (2008)
35. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., Riedl, J.: Grouplens: an open architecture for collaborative filtering of netnews. In: Proc. 1994 ACM Conf. Computer supported cooperative work, pp. 175–186. ACM (1994)
36. Ricci, F.: Mobile recommender systems. Information Technology & Tourism **12**(3), 205–231 (2010)
37. Schafer, J.B., Konstan, J.A., Riedl, J.: Meta-recommendation systems: user-controlled integration of diverse recommendations. In: Proceedings of the eleventh international conference on Information and knowledge management, pp. 43–51. ACM (2002)
38. Schein, A.I., Popescul, A., Ungar, L.H., Pennock, D.M.: Methods and metrics for cold-start recommendations. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 253–260. ACM (2002)
39. Schelter, S., Boden, C., Markl, V.: Scalable similarity-based neighborhood methods with mapreduce. In: Proceedings of the sixth ACM conference on Recommender systems, pp. 163–170. ACM (2012)
40. Tintarev, N., Masthoff, J.: Effective explanations of recommendations: user-centered design. In: Proceedings of the 2007 ACM conference on Recommender systems, pp. 153–156. ACM (2007)
41. Tintarev, N., Masthoff, J.: Designing and evaluating explanations for recommender systems. In: Recommender Systems Handbook, pp. 479–510. Springer (2011)
42. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann (2005)
43. Zhao, Z.D., Shang, M.S.: User-based collaborative-filtering recommendation algorithms on hadoop. In: Knowledge Discovery and Data Mining, 2010. WKDD'10. Third International Conference on, pp. 478–481. IEEE (2010)

Author Biographies
Click here to download Author Biographies: Author1.txt

Simon Dooms received the bachelor degree of Informatics at the faculty of sciences (Ghent University, Belgium) in 2007 and a master degree of Computer Science Engineering at the faculty of engineering in 2009. He started working on a Ph.D. as a member of the Wireless and Cable (WiCa) research group of Professor Luc Martens at the Information Technology department of Ghent University. His research interests include (hybrid) recommender systems, rating datasets, HCI, high-performance computing and data science in general.

Author Biographies
Click here to download Author Biographies: Author2.txt

Toon De Pessemier studied at Ghent University where he received the MSc degree in computer science engineering in 2006. That same year, he joined the Wireless & Cable research group of Ghent University. From 2007 to 2011, he was a Ph.D. student of the Research Foundation - Flanders (FWO). His current research activities include recommendation algorithms and personalization systems. Besides, he specializes in Quality of Experience (QoE) and the influence of personalized recommendations on the user's experience. His other research interests include data mining, machine learning, and neural networks.

Author Biographies
Click here to download Author Biographies: Author3.txt

Luc Martens received the M.Sc. degree in electrical engineering and the Ph.D. degree from Ghent University, Ghent, in July 1986 and December 1990, respectively. From September 1986 to December 1990, he was a Research Assistant in the Department of Information Technology (INTEC), Ghent University, where he was involved in the physical aspects of hyperthermic cancer therapy. His research dealt with electromagnetic and thermal modeling and with the development of measurement systems for that application. In January 1991, he became a member of the permanent staff of the Interuniversity Microelectronics Centre, INTEC, Ghent University, where he was responsible for the research on experimental characterization of the physical layer of telecommunication systems. He also studies topics related to the health effects of wireless communication devices. Since April 1993, he has been a Professor in electrical applications of electromagnetism at Ghent University.