Towards a Scalable Social Recommender Engine for Online Marketplaces: The Case of Apache Solr

Emanuel Lacić
Graz University of Technology
Graz, Austria
elacic@know-center.at

Dominik Kowald
Know-Center
Graz, Austria
dkowald@know-center.at

Denis Parra
Pontificia Universidad Catolica
Santiago, Chile
dparra@ing.puc.cl

Martin Kahr
BLANC-NOIR GmbH
Graz, Austria
martin.kahr@blanc-noir.at

Christoph Trattner
Know-Center
Graz, Austria
ctrattner@know-center.at

ABSTRACT

Recent research has unveiled the importance of online social networks for improving the quality of recommenders in several domains, what has encouraged the research community to investigate ways to better exploit the social information for recommendations. However, there is a lack of work that offers details of frameworks that allow an easy integration of social data with traditional recommendation algorithms in order to yield a straight-forward and scalable implementation of new and existing systems. Furthermore, it is rare to find details of performance evaluations of recommender systems such as hardware and software specifications or benchmarking results of server loading tests.

In this paper we intend to bridge this gap by presenting the details of a social recommender engine for online marketplaces built upon the well-known search engine Apache Solr. We describe our architecture and also share implementation details to facilitate the re-use of our approach by people implementing recommender systems. In addition, we evaluate our framework from two perspectives: (a) recommendation algorithms and data sources, and (b) system performance under server stress tests. Using a dataset from the SecondLife virtual world that has both trading and social interactions, we contribute to research in social recommenders by showing how certain social features allow to improve recommendations in online marketplaces. On the platform implementation side, our evaluation results can serve as a baseline to people searching for performance results of server loading tests.

1. INTRODUCTION

Recommender systems aim at helping users to find relevant information in an overloaded information space [11]. Although there are well known methods (Content-based [1], Collaborative Filtering [16, 18], Matrix Factorization [10]) and libraries to implement, evaluate and extend recommenders (Apache Mahout1, Graphlab2, MyMediaLite3, among others [8]), the deployment of a real-time recommender from scratch which considers a combination of algorithms and data sources, the effect of large volumes of data and hardware configuration, or the impact of model updates in the recommender performance remains unsolved or at least not publicly available for the research community. In this paper, we contribute to bridge this gap between research on recommender algorithms and system deployment by presenting in detail our approach to implement a social marketplace recommender. We describe our solution in terms of the data model that allows modularity and extensibility, as well as the system architecture that relies on the Apache Solr project to facilitate the scaling of our approach to big data.

We support our decision of using Solr on recent work that shows the strong relation between memory-based recommendation approaches and ready-to-use text analytic techniques [3]. Since Solr has already most of these techniques implemented, documented and optimized by a well established open-source community, we believe that it provides not only a good basis for a large-scale search engine but it also provides a good foundation to implement an efficient and scalable social recommender engine for online marketplaces. We appeal for Apache Solr since one might have to consider several dimensions of the data or already existing indices based on Apache Lucene, the kernel search engine Apache Solr is built upon.

To evaluate our implementation we consider diverse metrics – accuracy and ranking metrics along diversity, and user coverage – to unveil not only the performance of each recommender algorithm isolated but also to show the importance of each single feature and data source. In addition, a performance benchmarking experiment was conducted to show the scalability of our approach.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—Data mining; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—Information filtering

Keywords

social recommender engine; scalability; online marketplaces; Apache Solr

1http://mahout.apache.org
2http://graphlab.org
3http://www.mymedialite.net
In detail, the paper is structured as follows: we begin by describing our system architecture and implementation details in Section 2. Section 3 describes the experimental setup we chose to evaluate our framework, while in Section 4 we present the results in terms of recommendation quality and also system’s performance. After that, in Section 5, we discuss related work in the area and how it differs from our approach. Finally, Section 6 concludes the paper and provides an outlook to future work.

2. APPROACH

In the following sections we describe the architecture and highlight some implementation details of our approach towards a scalable social recommender engine for online marketplaces. The engine described below was implemented in Java as a joint effort with the Austrian start-up company Blanc Noir and was designed in a modular way based on Apache Solr as an highly efficient data processing and storing unit.

2.1 Implementation

The overall system architecture of our framework is illustrated in Figure 1. It consists of the following four main components:

The **Recommender Engine** consists of the implemented recommender algorithms (e.g., Most Popular, Content-based, Collaborative Filtering, etc.) that can be attached to the Recommendation Workflow component. The algorithms can be called separately, in a specific sequence or combined (e.g., as a hybrid approach). This design gives our framework not only the flexibility that new algorithms can easily be implemented and instantiated (see Listing 1) but also that new recommendation workflows can be defined based on a given use case or domain.

Moreover, the Recommender Engine component contains an Evaluator that can be used to test and tune the different algorithms (and the combinations of those) based on various evaluation metrics (see Sections 3.2 and 4.1).

The **Solr Service Container** acts as an abstraction layer for the Solr core functionalities to encapsulate the different queries and methods (e.g., facets or MoreLikeThis) into data-driven services (e.g., Marketplace or social services) that can be consumed by the recommender engine. Furthermore, this modular abstraction allows the Solr backend to be replaced by another data store or search strategy (e.g., ElasticSearch) if needed.

The **Solr Cores** contain the indexed data used to generate recommendations. Each core is described by its own Solr schema that specifies a data structure and that can be extended easily by simply adding new data fields to it and calling Solr’s RELOAD function. Currently, we store four types of data structures in the Solr cores: (1) user profiles, (2) item profiles, (3) user actions (e.g., purchases) and (4) social interactions (e.g., comments). New Solr cores can be added if new data structures are needed in the data model. For adding a new Solr core, a new data schema needs to be defined and registered in the solr.xml configuration file, which can be automatically done using Solr’s CREATE function.

Another major reason for using Solr is its support for horizontal scaling. Since version 4.0, full automatic index distribution and searching across multiple machines (either shards or replicas) is supported. Under a scenario where the maximum capacity for handling queries per second is reached, horizontal scaling with additional replicas can be performed. On the other hand, with sharding on multiple machines, Solr supports the need to store large amounts of data distributively.

The **REST API** is the interface to our framework that a client can either use to request item recommendations, based on a specific algorithm or workflow, or to update the data model (e.g., if a user has purchased an item) via JSON messages. These JSON messages are fully configurable and let the client, for example, define user-specific data filters to, e.g., request only recommendations for a given item category.

The data updates are handled by data connectors, where each connector is responsible for a different type of data. Currently there are two connectors in the system, one for social data (e.g., Facebook, G+ and Twitter streams) and the other one for marketplace data (e.g., purchases).

Listing 1: Example of how to implement and run a new recommender strategy.

```
// Implement the recommender strategy
public interface RecommendStrategy |
    public RecommendResponse recommend(RecommendQuery q, Integer maxResults, SolrServer SolrServer);

// Run the new recommender strategy
RecommendStrategy strategyToUse = new MyStrategyImpl();
Filter filter = new ContentFilter(); // optional
RecommendationService.getRecommendations("some_user", "some_product", 10, filter, strategyToUse);
```

2.2 Recommender Algorithms

Currently, our framework implements four algorithms types to recommend items (in our case products) to users. This set of algorithms can easily be extended or adapted as explained in Section 2.1.

**MostPopular (MP):** This approach recommends for any user the same set of items, which are weighted and ranked by purchase frequency.

**Collaborative Filtering (CF):** Consists of recommending items to a target user that have been previously favorited, consumed or liked by similar users, the neighbors. This method is also known as K-NN because it is usually accomplished in two steps: first, find the K nearest neighbors based on some similarity metric, and second, recommend items that the neighbors have liked that the target user still has not consumed [20].
In our case, we construct the neighborhood of a user based on two types of features: marketplace features (purchases and categories), and social features (interests, groups and interactions) as shown in Table 2. As an example: In the case of purchases ($CF_p$), we get all purchased items of the target user and query all users that have also bought these items through the Solr data model in order to recommend their purchased items to the target user. The resultant lists of users and items are ranked and weighted using Solr’s facet queries. The necessary queries for this process are the following:

```
// Find similar users based on purchased items using Solr's facet queries
/select?q=id:('some_product_1')+OR+id:('some_product_2')&facet=true&field=my_users_field
// Find items purchased by those similar users that are new to the target user
/select?q=my_users_field:('user_1'='5'+OR+'user_2'='3')&fq:id:('some_product_1')+OR+id:('some_product_2')
```

### Content-based Recommendations (C): Content-based recommendation systems analyse item meta-data to identify other items that could be of interest for a specific user. This can be done based on user profile data or on the meta-data of the items that the user has liked or purchased in the past [15]. Our implementation of a content-based recommender is based on the second method and uses the built-in MoreLikeThis functionality of Solr that finds similar items for one or multiple given items by matching their content. We use two different types of meta-data features, namely the title and the description of items (see Table 2). There are several parameters for the MoreLikeThis function that can be set e.g., the minimum document frequency (mindf), the minimum term frequency (mintf), minimum word length (minwl), etc. In the current implementation, both frequency parameters are set to 1 and the word length to 4, which gives us a good trade-off between accuracy and scalability. However, our implementation allows the application developer also to set the parameters herself, if needed. New content-based recommendation algorithms with different features can be developed by implementing the aforementioned RecommendStrategy Interface. The listing below shows how a content-based recommender can be called and customized in terms of the field (mlt.fl) used to match items with similar content:

```
/select?q=id:('some_product_id')&mlt=true&mlt.fl=description
```

### Hybrid Recommendations (CCF): All three mentioned recommender algorithms have unique strengths and weaknesses, e.g., CF suffers from sparse data and cold start problems, while content-based approaches suffer from item meta-data to be utilized [4]. Hybrid recommenders combine different algorithms to tackle this issue in order to produce more robust recommendations [5]. Considering that we want to favor items recommended by more than one method, we chose to implement the hybrid approach called Cross-Source Hybrid defined in [4]:

$$W_{reci} = \sum_{s_j \in S} (W_{reci,s_j} \cdot W_{s_j}) \cdot |S_{reci}|$$  \hspace{1cm} (1)

where the combined weighting of the recommended item $i$, $W_{reci}$, is given by the sum of all single weightings for each recommender source $W_{reci,s_j}$ multiplied by the weightings of the recommender sources $W_{s_j}$. Furthermore, it uses the number of recommender sources where $i$ appears $|S_{reci}|$ to strongly favor items that have been identified by more than one recommender. We use this approach to combine the different features and algorithms shown in Table 2 where each recommender source can be weighted accord-

### Table 1: Basic statistics of the SL dataset.

<table>
<thead>
<tr>
<th>Marketplace (Market)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>72,822</td>
</tr>
<tr>
<td>Number of purchases</td>
<td>265,274</td>
</tr>
<tr>
<td>Mean number of purchases per user</td>
<td>3.64</td>
</tr>
<tr>
<td>Number of products</td>
<td>122,360</td>
</tr>
<tr>
<td>Mean number of purchases per products</td>
<td>2.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Online Social Network (Social)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>64,500</td>
</tr>
<tr>
<td>Number of likes</td>
<td>1,492,028</td>
</tr>
<tr>
<td>Number of comments</td>
<td>347,755</td>
</tr>
<tr>
<td>Mean number of likes per user</td>
<td>14.91</td>
</tr>
<tr>
<td>Mean number of comments per user</td>
<td>3.47</td>
</tr>
<tr>
<td>Number of groups</td>
<td>260,137</td>
</tr>
<tr>
<td>Mean number of groups per user</td>
<td>8.91</td>
</tr>
<tr>
<td>Number of interests</td>
<td>88,371</td>
</tr>
<tr>
<td>Mean number of interests per user</td>
<td>1.57</td>
</tr>
</tbody>
</table>

Table 3.1: Experimental setup

In the following sections we describe in detail the dataset and the evaluation method and metrics used for our evaluation.

### 3.1 Dataset

In order to evaluate our social recommender architecture, we relied on two different sources of data to predict future product purchases (see also [25]) – online Marketplace data and an online social network data obtained from the virtual world SecondLife (SL). The reason for choosing SL over other real world sources is the lack of freely available datasets that combine both social network with marketplace data from the same set of users. The overall statistics of whole dataset can be found in Table 1.

Similar to eBay, every seller in the SL marketplace owns her own sub-page – called the seller’s store – where all items offered are presented to the general public. As with other trading platforms such as Amazon, sellers in the SL Marketplace have the possibility to apply meta-data information such as price, title, or description to their products. Customers in turn are able to provide reviews or ratings to products. In order to crawl all stores and corresponding meta-data information as well as interactions from the SL marketplace, we exploited the fact that every store has a unique URI built from the URL pattern http://marketplace.secondlife.com/stores/STORE_ID, where STORE_ID is an integer starting at 1. With this exploit at hand, we were able to download 72,822 complete user profiles with corresponding 265,274 purchases from the stores.

The online social network MySecondLife was introduced by Linden Labs, in July 2011. It can be compared to Facebook regarding postings and check-ins but aims only at residents of the virtual world. Hence, users can interact with each other by sharing text messages, and commenting or liking (= loving) these mes-

6https://secondlife.com/
8https://my.secondlife.com/
The evaluation of the performance of the recommender algorithms and data sources has been conducted in two steps, first we compared the different approaches and features on their own (see Table 2) and then we compared the combinations of those (see Table 4). The results for the different algorithms are calculated related to their user coverage and so are based only on the users where they were able to calculate recommendations as suggested in related work [9, 2]. Furthermore, also the values based on all users in the datasets are shown in parenthesis.

Table 2 shows that the best results for the accuracy metrics (F1, MRR, MAP and nDCG) are reached by CF_{\text{p}} followed by CF_{\text{d}}, the CF approaches based on social interactions and purchases. However, the results also reveal that CF_{\text{p}} only provides a small user coverage (UC) value, where CF_{\text{p}} performs much better and CF_{\text{d}} (CF based on groups) performs best. The best diversity (D) values are reached by the two content-based approaches based on title and description (C_{t} and C_{d}). Another thing that comes apparent is, that all shown approaches clearly outperform CF, (CF based on interests). Although the user’s interest seems conceptually a good metric to assess user similarity, in the SL social network it is defined by free-written keywords and phrases of the user, which would require additional validation or processing steps in order to exploit it as an efficient source of similarity.

This pattern of results also shows that the different algorithms and features have their unique strengths and weaknesses and that a hybrid combination of those should increase the overall recommender quality in terms of accuracy, diversity and user coverage [5]. Table 4 proofs this assumption and shows the combination of the marketplace-based approaches (CCF_{\text{m}} = CF_{\text{p}} + CF_{\text{d}} + C_{t} + C_{d}), the combination of the social based approaches (CF_{\text{s}} = CF_{\text{p}} + CF_{\text{d}} + C_{t} + C_{d} = CF_{\text{s}}) and the combination of both together with MP (All = CCF_{\text{m}} + CF_{\text{s}} + CF_{\text{s}}) to also address the issue of cold-start users. It can be seen that our hybrid approach not only outperforms the other approaches on all metrics but also provides a UC of 100% and so it can provide recommendations for all users in the datasets.

### 4.2 Framework Scalability

The recommender scalability has been evaluated in two ways, first we compared the runtime of the different approaches and features, as well as the hybrid combinations of those, and second we compared the mean response time of the algorithms in form of a stress test with an increasing number of requests in three scenarios. The results of the runtime comparison are shown in Table 3. The table reveals the mean test time (T_{\text{test}}) that is needed to calculate recommendations for a user and the overall time (T_{\text{test} + \text{train}}) that is needed to process all the users from the test set together with the training time (711 seconds) for building the data model.
in Solr (i.e., indexing the data). In general these results reveal that Solr is capable of providing real-time recommendations for users as the maximum mean test time is only 0.197 seconds for our hybrid approach.

Figure 2 shows the results of the stress test with an increasing number of requests in three scenarios, first without data updates during the recommendation process, the second one is similar but includes a 10% rate of data updates (i.e., randomly generated purchases), and the third scenario shows the time needed to update data. It can be seen in the first plot (without data updates) that the mean response time follows a near linear progress for our combined hybrid approach which clearly shows the scalability of Apache Solr and our framework. Most surprisingly this is also the case in the second plot that also takes data updates during the recommendation time into account and so shows the capability of Solr in maintaining its data index in near real-time. The third plot shows that Solr is also designed to handle a high number of update requests as there is a much sharper increase in the mean update time for a small number of update requests than for a high number.

This shows that our framework based on Solr already contains algorithms that not only provide a good trade-off between recommendation accuracy, diversity and user coverage, but also provide and calculate recommendations in real-time and at scale. Furthermore, there are additional ways to optimize Apache Solr (e.g., soft commits, using an SSD disk, ...) to even better tackle the performance of committing new or existing data.

5. RELATED WORK

There are already multiple frameworks and approaches out there that focus on scalable recommendation mechanisms. Most of these approaches are based on Collaborative Filtering techniques to predict the user’s ratings for items, such as movies or products, based on the user’s preferences in the past. However, the computational complexity of these calculations is typically very high, especially in the case of real-time streams.

To tackle this issue, previous work focused on distributed and scalable data processing frameworks such as Apache Hadoop or Mahout based on the map/reduce paradigm (e.g., [26] or [23]). In contrast to our framework based on Apache Solr, these approaches lack the mechanisms that enable near real-time updates of the data model (data indexing) in case of new user interactions (e.g., a user purchased an item) and updates of the data schema in case of new data sources that have to be plugged in (e.g., data from HBase tables). Furthermore, it is not trivial to handle social- and content-based data with these frameworks, whereas this functionality comes directly out-of-the-box with Solr (e.g., with the MoreLikeThis function) together with powerful full text search functionalities. An alternative method to improve Collaborative Filtering is based on Matrix Factorization as for example proposed by Diaz-Aviles et al. [6]. However, in this work the authors focus on the near real-time processing of Twitter streams for topic recommendations and not on item recommendations in social online marketplaces as it is done with our framework.

Other approaches use database systems in order to “query” the recommendations from a data model or to simply cache the already calculated recommendations. One example for a database-driven online recommender framework is the RecDB project by Sarwat et al. [19] which is built on the basis of a PostgreSQL database with an extended SQL statement set. The authors show that RecDB can provide near real-time recommendations for movies, restaurants and research papers. Although these approaches perform fairly good, it has been shown that relational database management systems are insufficient for full text searches, that are the basis for content-based recommendations, where information retrieval software like Solr greatly speed up the response time of the requested queries [24].

To date, there is only few research available that focus on the usage of search engines and information retrieval systems to implement recommendation services. In [22] a method is presented to implement a k-nearest neighbor-based recommendation system on top of full text search engines (MySQL and SphinxSearch) that provides linear scalability relative to the data size. Another work in this context is a recent contribution by Parra et al. [12] who implemented a recommender system for scientific talks based on Apache Solr. Although the latter mentioned contribution provides insights on how to implement a near real-time recommender system based on Apache Solr, they lack of extensive explanations and evaluations of how such an approach performs in a big data scenario.

6. CONCLUSIONS

In this paper we have presented the implementation details and evaluation of an online social marketplace recommender with a focus on two kind of readers: researchers and professionals in the area of recommender systems. On the research side, we provided results that highlight the importance of social features (interactions in the form of likes and comments) in order to improve the accuracy, diversity and coverage of product recommendations. From the side of professionals, we provided a description of our framework based on Apache Solr with detailed results in terms of performance and scalability in order to serve as a baseline for people interested in implementing a recommender system, information rarely found in current literature. Our framework evaluation considers dimensions such as hardware configuration, model training and testing trade-offs, real-time recommendation performance and the impact of model updates over the whole system performance.

We plan different tasks to extend our current study. In terms of algorithms, we would like to explore whether other hybridization techniques (weighted, mixed, etc.) can provide us alternative ways to combine methods and data sources, in order to produce

<table>
<thead>
<tr>
<th>Test</th>
<th>CF/5</th>
<th>CF/10</th>
<th>CF/20</th>
<th>CF/50</th>
<th>CF/100</th>
<th>CF/200</th>
<th>CF/500</th>
<th>CF/1000</th>
<th>MP</th>
<th>CCF/5n</th>
<th>CCF/10n</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.020</td>
<td>0.097</td>
<td>0.029</td>
<td>0.094</td>
<td>0.024</td>
<td>0.023</td>
<td>0.011</td>
<td>0.013</td>
<td>0.021</td>
<td>0.016</td>
<td>0.194</td>
<td>0.024</td>
<td>0.197</td>
</tr>
</tbody>
</table>

Table 3: Results of the runtime experiment (in seconds) for each single recommendation approach and feature together with the hybrid approaches and MP as a baseline.

<table>
<thead>
<tr>
<th>Measure</th>
<th>MP</th>
<th>CCF/5n</th>
<th>CCF/10n</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>nDCG</td>
<td>.0078</td>
<td>.0678</td>
<td>.0316</td>
<td>.0182</td>
</tr>
<tr>
<td>MRR</td>
<td>.0054</td>
<td>.0420</td>
<td>.0196</td>
<td>.0126</td>
</tr>
<tr>
<td>MAP</td>
<td>.0054</td>
<td>.0485</td>
<td>.0226</td>
<td>.0133</td>
</tr>
<tr>
<td>F1</td>
<td>.0032</td>
<td>.0354</td>
<td>.0165</td>
<td>.0115</td>
</tr>
<tr>
<td>D</td>
<td>.3801</td>
<td>.4877</td>
<td>.2274</td>
<td>.3770</td>
</tr>
<tr>
<td>UC</td>
<td>100%</td>
<td>46.63%</td>
<td>56.47%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 4: Results of the performance experiment for the hybrid approaches together with MP as a baseline (normalized to the actual UC in the row). Values in brackets represent the results normalized to 100% UC.
7. REFERENCES


