A Comparison of Recommender Systems for Mashup Composition

Paolo Cremonesi, Matteo Picozzi, Maristella Matera

DEI- Politecnico di Milano, Milano, Italy
firstname.lastname@polimi.it

Abstract—Web mashups are a new generation of applications created by composing contents and functions available through Web services and APIs. A central activity in mashup development is the retrieval and selection of components to be included in the composition. The adoption of recommender systems can alleviate some of the difficulties arising in this activity. Based on the results of an empirical study, this paper tries to shed light on the application of recommender systems to the mashup composition domain, and discusses the performance of different recommendation systems when applied to a very large collection of mashups and mashup components.

Keywords—Recommender systems, Web mashups, APIs.

I. INTRODUCTION

Web mashups are a new generation of applications created by composing contents and services available through Web services and APIs [19]. Mashups were initially conceived in the context of the consumer Web, as a means for end-users to create their own Web applications. More recently, tools for more critical domains, e.g., enterprise mashups, also started to emerge, thus showing the always increasing interest in this technology. Such interest is also proved by the ever increasing number of mashups available online: the ProgrammableWeb repository, the biggest mashup registry available online, counts more than six thousand mashups created by Web users and more than five thousand public APIs that can be used as mashup components. On the one hand, the interest in mashups undoubtedly lies in their capability to go beyond the single components by composing them in novel, value-adding manners; on the other hand, it also lies in the appeal they exert on non-programmers and their capacity to turn non-programmers into developers.

While initially mashups were developed manually by skilled programmers, recently new-generation tools have been proposed for the computer-assisted composition of mashups components. Current composition tools offer simple, programming-free visual editors, which can help realize the dream of a “programmable Web”. However, despite the efforts spent on the visual editors, the composition task can be still hard for non-programmer users: they not always have a clear idea of their composition goals or, even if they know which mashups they want to compose, they are required to select components from collections of available services and information sources.

This paper addresses the lack of effective paradigms for the assisted composition of mashups, and investigates the adoption of recommender systems to support the selection of mashup components based on the analysis of mashups created by the “masses”. Through a simulation experiment, we in particular compare the performance of different recommender systems when executed on a large repository of mashups and mashup components crawled by the ProgrammableWeb registry.

The paper is organized as follows. In Section II we contextualize the adoption of recommender systems to the mashup composition process and discuss some related works. Section III introduces the recommender algorithms that we have compared and describes the adopted testing methodology and the data set on which tests have been performed. Sections V and VI then summarize the experimental results and outline our conclusions.

II. RATIONALE AND BACKGROUND

In the last years we have assisted to an evolution in the way mashups are created. Initially mashups were merely developed manually by skilled programmers who took advantage of reusable components and mainly devoted their efforts to programming the composition logics. Soon the phenomenon has started inspiring a new generation of tools for the computer-assisted composition of mashups. The most prominent mashup editors (for example Yahoo Pipes) provide developers with easy to use environments where components are wrapped and ready to be used, and the composition is specified visually by coupling input/output parameters. Such editors, however, require the users to identify the “right” components, i.e., components that (i) best fits the current composition from the point view of the syntactic and semantic compatibility, and that (ii) increase the mashup added value from the point of view of additional and relevant data and functions. As found out in recent user-centric studies [16], this task could not be trivial for the average users. Some research works have therefore started proposing design-time assistance mechanisms. MashupAdvisor [13] is one of such works. It relies on a repository of mashups, and generates recommendations through a probabilistic model that ranks candidate mashup outputs retrieved in the repository based on their popularity and on AI metrics that identify a maximum utility plan. In [7] authors present algorithms to query mashup repositories.
and discover relevant composition knowledge, i.e., reusable composition patterns, based on syntactic similarity. In the context of our research on mashup development tools [5], we also defined a recommendation mechanism that ranks and suggests mashup components depending on their quality and on the contribution that they give to the aggregated quality of the overall composition [17]. To our knowledge, however, there are no approaches exploiting the potential of recommender systems in the mashup domain.

Recommender systems can be fruitful, due to the always massive increase in the variety of mashups available through online repositories, and can provide more reliable results. New recommender systems have been recently proposed for the selection of components in the construction of telcom mashups [15] and cloud applications [20]. However, their quality is arguable, and the proposed algorithms need more extensive evaluations. In this paper, we investigate the application of recommender systems to the mashup domain through an experimental, bottom-up approach in which we test and compare the performance of well-known algorithms. Our goal is to gain some lessons that can help maximize the advantages offered by recommender systems with respect to the peculiarities of the mashup domain.

III. ALGORITHMS

Our study has considered four state-of-the-art recommender algorithms. The main input to such algorithms is the user rating matrix (URM), where each element $r_{ui}$ is user $u$’s rating on item $i$ (missing ratings are set to zero). When composing mashups, users integrate two or more components. Therefore, users in the user rating matrix correspond to mashups, while items correspond to components. Ratings can be gathered either explicitly (typically in the 1–5 scale) or implicitly (typically in binary format: 1 if the user likes an item, 0 otherwise). In the mashup domain, ratings about components are often implicit, inferred by tracking the users’ activity (e.g., if a component has been used to compose a mashup).

A. Non-personalized algorithms

Non-personalized algorithms present any user a predefined list of items. Such algorithms usually serve as the baseline for the more advanced personalized algorithms. In this work, we considered a simple, non-personalized estimation rule, denoted by Top Popular (TopPop), which recommends top-N components with the highest popularity.

B. Collaborative neighborhood algorithms

Neighborhood algorithms base their prediction on the similarity relationships between either users or items [12]. We have focused our attention on two item-based neighborhood algorithms, i.e., Cosine Neighborhood (Cos) and Direct Relations (DR), as they usually perform better, while also being more scalable [18].

- **Cosine Neighborhood.** The similarity $s_{ij}$ between component $i$ and component $j$ is measured as the tendency to use both the components in the same mashup. Commonly, the similarity between components is computed via the cosine [18]

$$s_{ij} = \frac{\text{# mashups using both components}}{\sqrt{\text{# mashups using } i} \cdot \sqrt{\text{# mashups using } j}}$$

Note that, thanks to the denominator, the cosine similarity has the property of being bounded in the 0–1 range.

- **Direct Relations.** An alternative way of computing the similarity between each pair of components $i$ and $j$ is to count the number of mashups that use both the components, without any normalization factor

$$s_{ij} = \text{# mashups using both components}$$

According to [2] this metric emphasizes the similarity between popular items.

Once the similarity between components is computed, the estimated rating that user $u$ would express for component $i$ is usually computed as [8]

$$\hat{r}_{ui} = \sum_j s_{ij} \hat{r}_{uj}$$

Here $\hat{r}_{ui}$ does not represent a proper rating, but is rather a value we can use to rank the components according to user $u$’s estimated interest.

C. Collaborative latent factor algorithms

Recently, several recommender algorithms based on latent factor models have been proposed. Latent factor models - also informally known as Singular Value Decomposition (SVD) models - try to explain ratings by characterizing items and users with factors that are automatically inferred from user feedback. In this work, we have considered PureSVD, a recently proposed latent factor algorithm [8]. Its rating estimation rule is based on conventional SVD. The user rating matrix $R$ is approximated by the factorization [2]

$$R = U \cdot \Sigma \cdot Q^T$$

where $U \in \mathbb{R}^{n \times f}$ and $Q \in \mathbb{R}^{m \times f}$ are two orthonormal matrices representing, respectively, the left and right singular vectors associated to the top-$f$ singular values of $R$ with the highest magnitude. The top-$f$ singular values are stored in the diagonal matrix $\Sigma \in \mathbb{R}^{f \times f}$. Once the user rating matrix has been decomposed, the prediction rule can be written as

$$\hat{r}_{ui} = r_u \cdot Q_i \cdot q_i^T$$

where $r_u$ denotes user $u$’s vector of ratings (where unknown ratings are filled with zeros), and $q_i$ represents the $i$-th column of $Q$. Note that, similarly to (3), $\hat{r}_{ui}$ is not a proper normalized rating, but can be used to rank items according to user $u$’s interests. We have tested PureSVD with different numbers of latent factors. Cross validation suggested us that good choice is to use 10 latent factors.
IV. Testing methodology

The evaluation of recommender systems is typically performed by using classification accuracy metrics - such as recall, precision, and fall-out [4], [9], [14]. We present the results of our experiments in terms of recall, which is defined as the ratio between the number of relevant (i.e., interesting to the user) components recommended to the user and the total number of relevant components present in the catalog.

\[
    r(N) = \frac{\text{# relevant components recommended}}{\text{# relevant components in the catalog}} \quad (6)
\]

Since we are facing with an implicit, binary dataset, in our evaluation we have considered - analogously to other works such as [10], [11] - all rated components to be relevant, as we do not have any further information about the degree of user satisfaction. Known ratings in the dataset are split into two subsets: training set \( M \) is used to learn the algorithm parameters and test set \( T \) is used to test the algorithm accuracy. The splitting methodology adopted in this study is the \( k \)-fold cross validation, similar to the one described in [9]. The user-rating matrix is partitioned into \( k \) sets of users (called folds), i.e., mashups, of the same size. Then we evaluate \( k \) times, always using one fold as the test set \( T \) and all the other folds as the training set \( M \). The \( k \) results are then averaged. This approach produces more robust results and allows to estimate variance of recall. In our experiments, two folds (i.e., \( k = 2 \)) were sufficient to observe strict bounds in the recall estimation.

We have not included precision as an accuracy metric because it cannot be estimated in a reliable way unless all ratings are known for all mashups and all components. If this is not the case, the test set \( T \) will contain a large number of unrated components. These are considered irrelevant; they miss a fraction of unknown positive relevance, and lead to precision underestimation [3]. Moreover, we have not included fall-out because it cannot be computed on implicit, binary datasets as the one adopted in our experiment.

During the testing, we use the leave-one-out methodology, that consists of removing one rated component from a tested user's profile. We later provide recommendations to that same user and we analyze if the removed component is recommended. Note that mashups with one component cannot be tested since, after leave-one-out removal, they became mashups with no ratings and hence it is not possible to predict any recommendations. This is also coherent with the generally recognized definition of mashups as a composition of two or more components.

A. Dataset

The dataset has been constructed by crawling the ProgrammableWeb repository. This is the biggest mashup registry available on the Web and plays a benchmark role in the mashup community. It contains more than six thousands mashups and more than five thousands APIs contributed by the mashup developer community. Through this site, it is possible to publish mashups and their composing APIs, or just APIs. Mashups are described and tagged, and the APIs used in the mashup composition are listed. Each API is characterized by, e.g., categories, tags, data exchange formats, links to documentation, mashups in which it is used, 'how-to’’s and users’ comments. Crawling all these data has allowed us to populate a database from which we have then extracted the dataset used in the experiment.

The dataset consists of 821 components and 2786 mashups. The number of implicit ratings (i.e., the total number of times a component has been used in a mashup) is 9200. The density of the dataset is 0.004, the average number of components per mashup is 3.3, and the average number of mashups that use a component is 11.

B. Popular components vs. long-tail

According to the well known long-tail distribution of rated items applicable to many systems, the majority of mashups are using a small fraction of the most popular components [1]. Main goal of a recommender system is to suggest novel components that allow to cover the otherwise unexplored part of the set of components. Figure 1 plots the cumulative rating distributions of the ProgrammableWeb datasets. Components on the horizontal axis are ordered according to their popularity, most popular on the left. We observe that about 50% of implicit ratings involve only the 1.5% of most popular components (i.e., 12 components).

We refer to this small set of very popular components as the short-head, and to the remaining set of less popular components – about 98.5% of the total – as the long-tail [6]. Recommending popular components is trivial and does not bring much benefits to users. On the other hand, recommending less known components adds novelty but it is usually a more difficult task. In this study we aim at evaluating the accuracy of recommender algorithms in suggesting non-obvious components. To this purpose, part
of the experiments have been performed by removing from the dataset the 12 most used components. These experiments will be labeled long-tail in the Results section.

V. Results

In this section we present the quality of the four recommender algorithms presented in Section III on the ProgrammableWeb dataset\(^2\) described in Section IV-A. The first one - TopPop - is a non-personalized algorithm, and we would expect it to be outperformed by any recommender algorithm. For each algorithm, we have performed one set of experiments on the full dataset and one set of experiments on the long-tail dataset. We report the recall as a function of \(N\) (i.e., the number of components recommended) and the percentage increase (or decrease) of recall with respect to the TopPop algorithm. We have focused our attention on values of \(N\) in the range 1–20. Larger values of \(N\) can be ignored as there is no difference whether an interesting component is placed within the top 100 or the top 200, because in neither cases it will be presented to the user.

Figures 2a and 2c report the performance of the algorithms on the full dataset. It is evident that the algorithms do not have a significant performance disparity in terms of recall. For instance, the recall of any algorithm at \(N = 1\) is in the range 0.3–0.5, i.e., all the algorithms have a probability between 30% and 50% to recommend an interesting component in the first position. Non-personalized TopPop shows surprisingly good results. More surprisingly, the recall of PureSVD is constantly lower than the recall of TopPop.

The strange and somehow unexpected result of TopPop motivates the second set of experiments, accomplished over the long-tail components, whose results are drawn in Figures 2b and 2d. As a reminder, now we exclude the 12 very popular components from consideration. Here the ordering among the different recommender algorithms better aligns with our expectations. In fact, recall of the non-personalized TopPop falls down. However, even when focusing on the long-tail, the best algorithms is still DR, whose recall at \(N = 1\) is almost twice as good as the recall of TopPop. Performance of PureSVD, which is better tuned to recommend long-tail items, now becomes significantly close to that of Cos, and both are able to increase TopPop recall by a factor of 40% at \(N = 1\).

VI. Concluding Remarks

This work shows, through an extensive empirical study, that recommender systems can efficiently help users in finding interesting components for their mashup applications. Over both full and long-tail datasets, DR is consistently the top performer algorithm, beating the more detailed and sophisticated PureSVD. Given its simplicity, we did not expect this result. In fact, we would view it as a good news, as DR combines multiple advantages. First, it is very easy to code, without a need to tune any parameter. DR also has the convenience of easily providing explanations to the users about recommendations. An interesting finding is that, when moving to longer tail components, recall of PureSVD improves with respect to other algorithms. This suggests that with larger datasets, comprising thousands of components, latent factors approaches can be the better choice. As future work we plan to exploit the results of this study to extend our mashup platform with recommendation mechanisms.

References


\(^2\)The dataset is available for free download at the following address: http://home.dei.polimi.it/cremones/recsys/ProgrammableWeb.zip. When using the dataset, please cite this paper.
Figure 2: Performance of the compared algorithms.


