Aggregating Functionality, Use History, and Popularity of APIs to Recommend Mashup Creation

Aditi Jain, Xumin Liu(✉), and Qi Yu

B. Thomas Golisano College of Computing and Information Sciences,
Rochester Institute of Technology, Rochester, USA
{axj4268,xumin.liu,qi.yu}@rit.edu

Abstract. Creating mashups from existing Web APIs has provided an effective means to boost software reuse and approach the full potential of online programming resources. One of the key hindrance faced by mashup creation is to discover relevant APIs, especially due to the recent fast growth of Web APIs and the brief, unstructured API descriptions. In this paper, we propose a novel approach that recommends APIs to create a mashup given a free-form text description. We incorporate three heterogeneous but complimentary factors into the recommendation process: the functionality of an API, the usage history of the API by existing mashups, and the popularity of the API. We leverage probabilistic topic models to learn an API's functionality from its textual description and compute relevance between the API and the given mashup description. As most APIs lack a rich textual description, we extend the API discovery process by exploiting collaborative filtering to estimate the probability of an API being used by existing similar mashups. These two sources of information are then integrated through Bayes' theorem, which allows us to discover a set of functionally relevant APIs. The popularity of these APIs is then factored in to perform quality based ranking so that the best APIs can be recommended first. A comprehensive experimental study has been conducted on a real-world dataset to evaluate the efficiency and effectiveness of the proposed method. The result indicates that our method is efficient and provides better recommendation than other competitive methods.

1 Introduction

The advent and advance of the Web 2.0 paradigm have expanded the development of mashups and their use in various web and mobile applications. Web service mashup refers to the composition of several web services or APIs (as most of them are REST-ful) to augment the functionality of those APIs. For example, an application to show weather forecast on a map of a location may integrate the mapping API by Google and weather API by Weather Channel. Mashups let a developer reuse already existing APIs and save development time and provide higher quality and reliability with least effort [2]. As the interest in

© Springer-Verlag Berlin Heidelberg 2015
A. Barros et al. (Eds.): ICSOC 2015, LNCS 9435, pp. 188–202, 2015.
DOI: 10.1007/978-3-662-48616-0_12

developing mashups increases so does the number of APIs. For example, there are about 13,444 APIs as of May 2015 published on ProgrammableWeb. Selecting the best suited API for mashup creation is a strenuous task even for an experienced developer. Many of the APIs lie under the same umbrella of functionality, so picking an optimal API among them in terms of quality and user-interest further complicates the selection process. Other limitations in selecting services for mashup include the compatibility of services with each other, so analyzing the input/output parameters of the services is again laborious. Therefore, there's a need for an approach that could simplify the API selection process for mashup composition so that developers can focus on other parts of their applications.

Existing efforts on recommending services for mashup creation fall into three main categories that are based on functionality based, QoS, and social network, respectively. Each type of approaches focus on one aspect, making them vulnerable to the possible low quality and insufficient input of that particular type of information. Functionality based approaches focus on finding APIs that provide relevant functionality [6,7,9]. They leverage various information resources, such as structured and unstructured API descriptions, semantic markups, tags, topic models, and API categories, to identify those relevant APIs for a mashup. QoS-based approaches use QoS value as the main guidance for finding suitable APIs for a mashup [3]. The generation of a mashup is led by an optimization process, aiming to achieve the best QoS in the end. These approach require the input from users to identify those functionally related APIs. This could be very challenging for users given the huge number of available APIs online. Social network based approaches leverage the network among mashups, APIs, user etc. and then predict links for services and mashup [2,10]. These approaches require user information as the input for API recommendation, which is difficult to obtain in many cases.

In this paper, we propose a novel approach that addresses the above limitations when recommending APIs to create a mashup given its textual description. The approach is hybrid as it estimates the recommendation probability of an API from three perspectives: functionality, usage history, and popularity. Specifically, our contributions in this work are summarized as follows.

1. We exploit **probabilistic topic models** to derive functional features of APIs and the desired mashup specification given by the user. Both the APIs and mashup can be then represented as probabilistic distributions on latent topics. The relevance of an API to a mashup is measured by the similarity between the topic distributions of its description and the mashup specification.
2. We leverage **matrix factorization based collaborative filtering** to identify additional functionally relevant APIs. The idea of using collaborative filtering is to leverage the mashup creation efforts made before, which were recorded in descriptions of existing mashups. As our goal is to create a mashup, no APIs have been used by such a new API yet, giving rise to the long-standing cold start issue in recommender systems. We instead focus on the existing mashups that are similar to the user desired mashup based on their topic distributions computed using probabilistic topic models. These

mashups are then used in a collaborative filtering algorithm to locate additional relevant APIs.

3. We **apply Bayes' theorem to integrate** the two sources of information obtained through probabilistic topic models and collaborative filtering, which gives the posterior probability of given an API being used by the new mashup. The top-k most probable APIs are identified, which are all considered as functionally relevant to the new mashup. The popularity of these APIs are then used to implicitly **perform QoS based ranking** and recommend the best ones to the user.

The remainder of this paper is organized as follows. In Sect. 2, we give an overview of existing effort that is relevant to the proposed approach. In Sect. 3, we present in detail the proposed API discovery and recommendation approach for mashup creation. In Sect. 4, we describe our experimental result. We conclude in Sect. 5.

2 Related Work

In this section, we discuss several representative related work and differentiate them with our work.

Functionality Based Recommendation. An approach was proposed to use Relational Topic Model (RTM) to identify functionally equivalent APIs for recommendation [6]. RTM determines topic distribution in a document considering all the citations and web links present in the document. The approach assumes that mashups and APIs form a network of documents where documents consist of topic related words and tags and links between them means that the API is part of the mashup. Now, to recommend a mashup, RTM model and binary random variables are used to predict the links. An iMashup tool was proposed to compose mashup based on a data-driven approach using tag-based semantic annotations [7]. It aims toward quick and easy composition of mashup as many end-users want ease in the discovery and integration of services to get the required final mashup. The tool makes use of tags to derive semantic annotations and links them to service's inputs/outputs. Then services are linked based on similar tags and a directed acyclic graph is constructed using those links. Depth first search and then regression search is run to obtain the recommendation services from the graph. In our work, we consider more factors besides functionality for recommendation. A category-based approach was proposed to identify the latent categories of potential APIs from the development requirement of a mashup and recommend the top ranked APIs in each category [9]. This method assumes that no more than one API should be selected from one category for a mashup, which is not always the case. As an example, a social mashup that allows users to access multiple social accounts may require several social APIs, such as Twitter, Facebook, and LinkedIn.

Quality-Based Recommendation. An approach was proposed to use the quality of APIs to drive the composition of a mashup [3]. A tool was developed to allow a user to drag APIs to add to a mashup. Then the tool calculates how the added API would influence the overall quality of the mashup and provide suggestions for alternative services. To start with the quality evaluation, first the QoS is taken into account and then the role of this service in the mashup is analyzed. Naturally, a master/central service would influence quality of the mashup more as compared to any slave/side services. This method requires a user to select the appropriate APIs from the functionality perspective, which imposes a great burden on users due to the large number of available APIs.

Social-Based Recommendation. A social-aware recommendation approach was proposed to address the implicit and explicit requirements of a user [10]. The approach explores and analyzes the social relationships between tags, mashups/API topics, and users by building a coupled matrix model. Then coupled-factorization algorithm is ran on the matrices obtained after the analysis to identify latent relationships and construct a recommendation prediction matrix. Another social-based approach was proposed by A. Maaradji et al. for service discovery and selection for mashup composition [8]. It proposed a platform called Social Composer (SoCo), which leverages social interactions of composition interests of users to derive mashup recommendations. It transforms the interactions between users and services to interactions between users. It uses an implicit social graph to map implicit social relations between users and links the users depending on their composition interest and activities. These links help build recommendation confidence based on the common interest of two users. The recommendation is based on analyzing user profiles for user's interests in services and relevance of services. An approach was proposed to solve the cold-start problem and under utilization of social information [4]. It makes use of both social and functional information to accelerate service discovery and recommendation for mashups. The method first extracts semantic descriptors from user's mashup query to discover candidate APIs and social features are extracted using popularity and collaboration ratings. Then, these semantic and social features of APIs are represented using graphs. The candidate mashup chains are assessed for input/output connectivity to recommend services for mashups. This approach assumes the knowledge of service input/output, which may be difficult to obtain for RESTful services that have become the majority of online APIs. In contrast, our approach can work with both structured and unstructured service descriptions so that it can be used for both SOAP based and RESTful services. The social based approaches request user information and the corresponding social network. Such information may be hard, or impossible to locate in many real-world scenarios.

3 The Proposed Approach

In this section, we describe in detail the proposed approach that aggregates API functionality, their usage history by existing mashups, along with their

popularity to recommend relevant APIs for mashup creation. The approach is composed of three interrelated components - (1) functionality based API discovery, (2) matrix factorization based collaborative filtering to expand the candidate API space, and (3) popularity based ranking for API recommendation.

Given the limited terms in the API descriptions, discovery of functionally relevant candidate APIs should go beyond simply matching terms in the descriptions. The recent advances in topic models in natural language processing and machine learning enable us to match APIs and mashups based on their underlying topics, which are expected to cover the functionality they provide. In particular, we exploit the Latent Dirichlet Allocation (or LDA [1]) model to determine the topics in API descriptions and the desired mashup specification given by the user. Then cosine similarity is used to determine the similarity between the topic distributions of API descriptions and the given mashup specification. Topic models still inherently rely on the use of terms to derive the underlying topics, based upon which the relevant APIs will be identified. As a result, the few terms used by most API descriptions may limit the effectiveness of topic models and hence relevant APIs may be missed out. To address this limitation, matrix-factorization based collaborative filtering is employed to discover additional candidate APIs based on the way they have been used in other existing mashups. We will focus on the existing mashups that are similar to the user desired mashup based on their topic distributions computed using LDA. Each of these two components will assign a relevance score to each API. Since the relevance scores lie in the range of 0 to 1, they can be interpreted as the probability of being relevant to the user desired mashup. By making the class conditional independence assumption, we can leverage the Bayes' theorem to integrate the two relevance scores into a single one, which allows us to choose a set of top-k candidate APIs. These top-k candidates will be finally ranked and returned to the user based on how popular they have been used in the past. As the top-k candidates are mainly determined from a functional perspective, the popularity of these APIs help factor in the non-functional aspects of these services, which enable to recommend functionally relevant services with good qualities.

In sum, the first two components of the proposed approach help discover a set of functionally relevant APIs for mashup creation while the last component uses popularity to implicitly perform QoS based ranking of these selected APIs and recommend the best ones to the user. Figure 1 illustrates the overall structure of the proposed approach.

3.1 Functionality-Based Candidate API Discovery

In this component, functionally relevant APIs are identified based on the given mashup specification in the mashup query. LDA is used to represent the functionality of available APIs and mashup specification by determining their topics.

LDA is a generative model, which regards each document as generated from a collection of topics and each topic is a distribution of words. It will determine the topic distribution in API descriptions and the mashup specification in the given mashup query to analyze their functionality. The API descriptions in this study are part of the API dataset from ProgrammableWeb.com that contains API

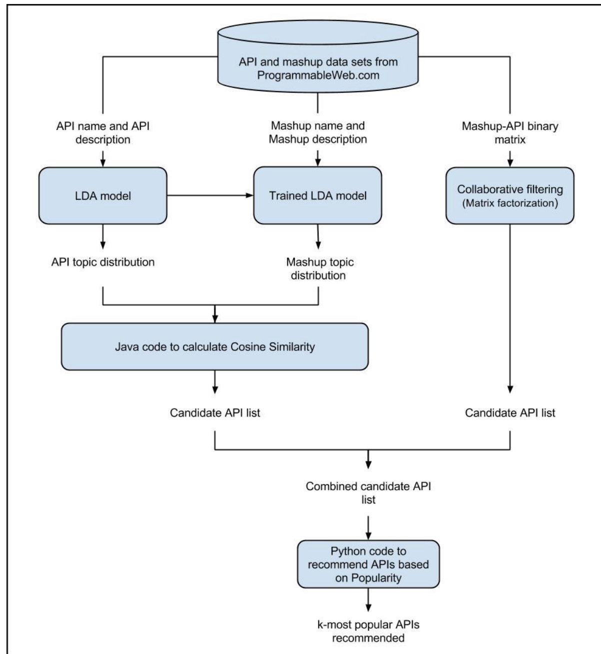

Fig. 1. Overall structure of the approach

names, descriptions of API, the number of mashups an API has been used for, API providers, and tags. Like many other model, LDA consists of two phases - learning or training phase and inference or testing phase. LDA is trained using API descriptions. In the training phase, the model takes all the API descriptions and the total number of topics as input. It tries to distribute words under different topics based upon their co-occurrences to obtain word-topic distributions. Meanwhile, it analyzes and assigns each document (or API description in this case) with probability of existence of each topic and gives topic probability distribution as the output. The topics having high probabilities contribute towards the identification of functionality of the API. In the testing phase, the topics for given mashup specification are inferred. The word-topic distributions are available from the training model obtained using API descriptions. The same topics from the training set are used to find the probability of those topics in the given mashup specification. After obtaining topic distributions for both APIs and mashup specification, the similarity between each API and the mashup specification can be calculated using the cosine similarity of their respective topic distribution vectors.

3.2 Historical Usage Based API Discovery

As stated above, the lack of rich API descriptions may limit the power of using topic models, such as LDA, to identify their underlying functionality and hence

discover APIs that are relevant to a user desired mashup. To address this issue, we propose to leverage historical usage of APIs in existing mashups and apply collaborative filtering techniques to identify additional candidate APIs.

Collaborative filtering is one of the most famous techniques for recommendation systems. It is based on the concept that if two users who previously preferred same items would prefer similar items in future. While recommending items to a user, all the users similar to an active user are found and the items preferred by those users are recommended. Alternatively, if a user prefers an item, similar items could be referred to the user. Thus, leveraging usage history of an item and a user, items could be recommended to users by analyzing the preference of users in past. Collaborative filtering could use two techniques - neighborhood-based or model-based. Neighborhood-based approaches usually use Pearson Correlation Coefficient to calculate the nearest or most similar users (and/or items) for recommendation. Though this technique is easy to understand and implement, it is usually not effective when the data is sparse and its performance decreases as the size of the dataset grows. On the other hand, model-based approaches, such as matrix factorization, are usually effective to overcome the data sparsity issue. These techniques are based on the idea that there are latent features/factors that could be discovered from user preferences and then be used for making recommendations [11].

Matrix factorization based collaborative filtering utilizes a user-item matrix, which usually represents user ratings for each item and may contain empty entries suggesting that user has not used that item and doesn't provide a rating for it. Thus, predicting the rating for these unused items would aid the recommendation process. This matrix is decomposed to learn latent factors and obtain two different matrices (corresponding to users and items, respectively) whose product recovers the original matrix. In practice, the original matrix is never recovered perfectly. The goal is to discover component matrices whose product minimizes the errors or differences between the original matrix and recovered matrix. After the errors are minimized, the new matrix would contain predicted approximate rating values for those empty entries. Early implementation of matrix factorization relies on Singular Value Decomposition (SVD), which is inefficient for large sparse data matrices. We instead exploit the Alternating Least Squares (ALS) method which works well with sparse data matrices and minimizes the squared errors by alternating between holding one of the factors fixed while computing the other [5].

The above idea can be applied to identify relevant APIs based on their usage history of existing mashups. Specifically, mashups play the role of users and APIs play the role of items. The key remaining issue is that since the goal is to create a new mashup, there is no usage history for the mashup yet. To still apply collaborative filtering, we instead seek for existing mashups that are similar to the desired mashup. We can again leverage LDA based topic models to determine the similarity between the new mashup specification and all existing mashup descriptions. In this way, the most similar existing mashups can be identified. After that, all the APIs used by these mashups will be included in the candidate list. This helps include some additional APIs but the number may still be limited

as most mashups only use a very small number of APIs. For example, after analyzing the mashups crawled from ProgrammableWeb.com, it was observed that around 85 % of the mashups use only three or less APIs. We then use each of these mashups as if it is the new API and apply matrix factorization based collaborative filtering to identify more relevant APIs. Intuitively, APIs that have been used by mashups that are similar to new mashup stand a greater chance of being used again as they would be more contextually relevant and thus could be recommended for the required mashup. As the output, a list of candidate APIs for the new mashup is returned with a probability of the API being used by the new mashup.

3.3 Popularity Based API Ranking

By using topic models in the first component, each API is assigned a cosine similarity with the new mashup that a user desires to create. Similarly, by leveraging the API usage history through collaborative filtering in the second component, each API is assigned a probability of being used by the new mashup. In fact, the outputs from the two components can be both regarded as relevance scores of an API. Since the relevance scores take values in $[0, 1]$, we can interpret them as the probability of being relevant to the user desired mashup. Let $p(a_t|m)$ denotes the probability that API a is relevant to mashup m based on their topic distributions; let $p(a_u|m)$ denotes the probability that API a is relevant to mashup m based on its usage history in existing mashups similar to m. By assuming conditional independence, we can compute

$$p(a|m) = p(a_t|m)p(a_u|m) \tag{1}$$

$p(a|m)$ essentially specifies given a mashup m, how likely API a will be used in it. We are instead interested to know given an API a, how likely it will be used by the (new) mashup m, which is given by the posterior probability $p(m|a)$. By applying Bayes' theorem, we have

$$p(m|a) \propto p(a_t, a_u|m)p(m) = p(a_t|m)p(a_u|m)p(m) \tag{2}$$

$p(m)$ is the prior probability of observing a mashup m, which can be set to $1/M$ with M denoting the total number of mashups. We can use $p(m|a)$ to select the top-k most relevant APIs.

The top-k APIs are all considered as providing relevant functionality for the new mashup. However, some of these APIs may offer identical functionality. It would be desirable if these APIs can be ranked based on their nonfunctional properties (i.e., QoS) and returned to the user. Intuitively, APIs which have higher quality are the ones frequently used for constructing existing mashups, hence, they are more popular as compared to the others. Thus, the popularity score, which is computed as the number times an API has been used in existing APIs, can serve as a good QoS indicator and be used to rank the functionally relevant APIs. In this way, the recommended APIs are not only functionally relevant for the new mashup, but also provide a good QoS guarantee [6].

4 Experiments and Evaluation

We conducted a set of experiments to evaluate the efficiency and effectiveness of the proposed approach. We collected the experimental dataset by crawling the ProgrammableWeb.com, one of the largest public web API repository. We crawled the API and mashup profiles including their names, API categories, tags, brief descriptions, and the use of APIs by mashups. There are 10,325 APIs and 6,819 mashups in the dataset, but only the mashups that use four or more APIs were selected to get more tangible results. So, about 950 mashups were selected and used.

All experiments were carried out on a Macbook Pro with 2.6 GHz Core processor and 8 GB DDR3 memory under Mac OS X 10.9.5 operating system. The evaluation focus mainly on the accuracy of the recommendation. We use the summary of a mashup as the input and evaluate how well it can predict the APIs for the mashup, comparing the actual APIs used by the mashup with the ones recommended by our approach. We compare the proposed method with functionality based, collaborative filtering based recommendations, as well as some existing competitive methods [6].

4.1 Training Probabilistic Topic Models

We used the Mallet LDA package[1] to learn latent topics and the their probabilistic distributions for terms. As mashup descriptions are treated as the input from users that reflect their requirements on creating a mashup, such information is not always available before making recommendations. Therefore, we only uses API descriptions for the topic learning process. The performance of LDA is affected by the predefined number of topics. The optimal number can be obtained through a set of trials and the observation on the resulted word-topic distribution for each trial. If many semantically unrelated words are assigned to the same topic, then the specified number of topics should be bigger. On the other hand, if many semantically related words are assigned to different topics, the specified number of topics should be smaller. Following this guidance, we trained the model with different number of topics, starting from 30 to 100 with an interval of 5 topics. We settled at 65 topic as this gave us the most readable result. The output of the LDA model training are topic-document and word-topic distributions. Figure 2 illustrates distribution of 14 topics over 10 different API descriptions. Figure 4 visualizes some example words for topics 4, 63, and 45. The font size of each word in the table is proportional to the probability of the word appearing in that topic. For example, the word *interface* appears more number of times as compared to word *developer* in topic 45 which corresponds to *web-related* topic. The relation between APIs *flickr* and *tumblr* can be seen through Figs. 2 and 4 that topic 63 has the highest probability, which consists of words like *social, friends, photos* etc. (which we know is true). Also, Fig. 3 illustrates percentage of API descriptions assigned to different topics. It shows

[1] http://mallet.cs.umass.edu/.

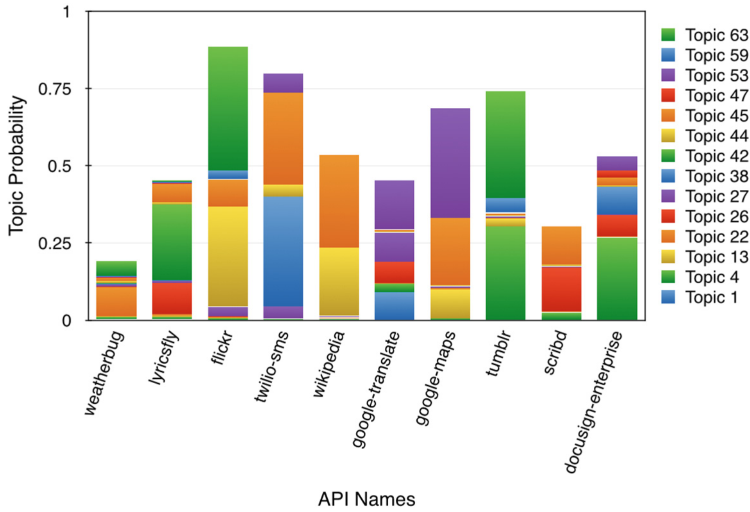

Fig. 2. Stacked bar graph illustrating topic distribution over API description

that topic 45 is assigned to approximately 21 % of the API descriptions. This is because it contains words like *interface* and *protocol*, which are common words used for API descriptions.

After training the LDA model, the model was used to infer topics in mashup description of the mashup query. To infer the topics, the word-topic distribution is piped from the model trained with API description. We use the same topics from the training set and find the probabilities of those topics in mashup description. The inference output of the mashup query is the list of topics with their probabilities. The topics in each document are ordered based on their topic probabilities. Hence, the first topic assigned for each document is the most prominent topic and so on.

4.2 Evaluation Result

We used the current API used by mashups as the ground truth to evaluate the accuracy of our recommendation method. We divided the mashups into three sets: training, validation, and testing. In the training and validation sets, the use of APIs by mashups is assumed to be known. The testing set consists of 302 mashups, where the use of APIs by these mashups were compared to the recommendation result for the evaluation. We compared our recommendation method with two other ones: topic modeling based and collaborative filtering based approaches. As most of the existing mashup recommendation methods do not tackle the cold start problem, their experimental results are not comparable to ours except for the one proposed in [6]. Therefore, we compared our method to it.

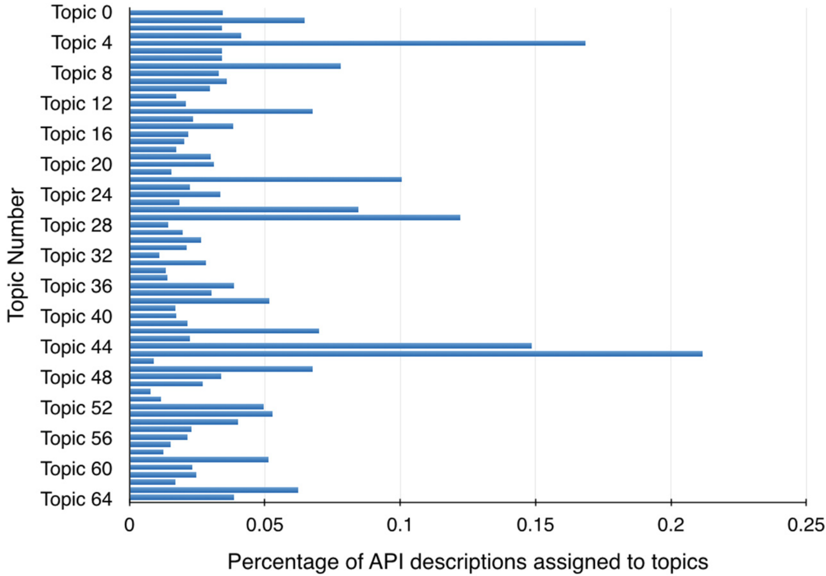

Fig. 3. Bar graph illustrating percentage of API descriptions assigned to topics

Traditional metrics for measuring prediction accuracy of a recommendation system includes *precision* and *recall*. Precision is the ratio of the total number of properly recommended APIs to the total number of recommended APIs. Recall is the ratio of the total number of properly recommended APIs to the total relevant APIs. In this work, we use recall@T to examine the impact of the number of recommended APIs on the accuracy. That is, given a constant T, a recommendation system suggests the top T APIs. We set T from 10 to 100, with an interval of 10. Recall is calculated as the ratio of the total number of properly recommended APIs in the suggested list (N_{Ti}) to the total number of APIs actually used in the mashup (N_{m_i}). Therefore, the overall performance of the recommendation is the average of all recalls evaluated in the testing set, i.e.,

$$recall@T = (1/M)\sum_{i=1}^{M}(N_{Ti}/N_{m_i}) \quad (3)$$

We also use F-score as an evaluation metric. F-score integrates both recall and precision by computing the harmonic mean of them as follows, i.e.,

$$\text{F-score} = \frac{Recall * Precision}{Recall + Precision}$$

Figure 5 shows the average recall values for top-T recommended APIs and compares each of the approach where $T \in \{10, 20, ..., 100\}$. It can be seen that the proposed hybrid recommendation method outperforms the LDA based and

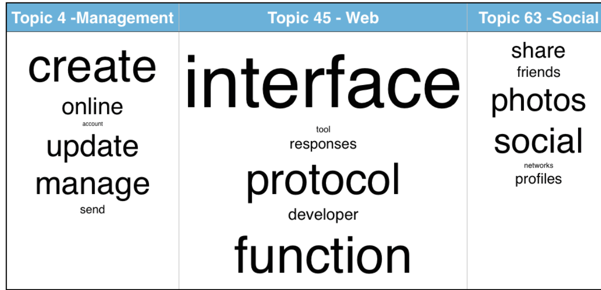

Fig. 4. Example of word-topic distribution

collaborative filtering method methods. On the one hand, LDA can find similar APIs, but doesn't consider mashup's usage history, which leads to recommending APIs that are not relevant or not preferred by the mashup. This explains why it performs poorly. On the other hand, collaborative filtering considers mashup usage history and can recommend APIs that are preferred by the mashups, thus performing much better than LDA. However, collaborative filtering suffers from the cold-start problem. Therefore, combining both approaches to give a hybrid solution seems logical to overcome these issues to some extent. Moreover, integrating popularity score of APIs adds QoS factors in the recommendation as usually most popular APIs are acknowledged as being of high quality. Figure 5 also demonstrates that the value of recall increases as the number of recommended APIs increases for all the approaches. This is a logical behavior as the probability of relevant APIs being recommended would be higher when more APIs are recommended. But when this number is higher than 80 the recall is almost constant and when recommended APIs lie between 60–80 there is not much difference in recall values. This indicates that about 60–80 APIs could be recommended in order to achieve similar performance and recommending less number of APIs would reduce mashup creator's further workload.

Table 1. Comparison of proposed approach with the ERTM Approach in [6]

	Recall	Average precision
Proposed approach	0.62	0.41
ERTM approach	0.27	0.16

Table 1 compares our work and the one proposed in [6]. In order to make a fair comparison, we used the metric in their work to evaluate the accuracy. That is, average precision gives the accuracy of ranked recommendation list i.e. it considers the order of the recommended list and computes if the results in higher rank/position in the list are relevant or not. This measure is computed as it is usually desirable in search results to know if top results are relevant to the user or not. It was computed as:

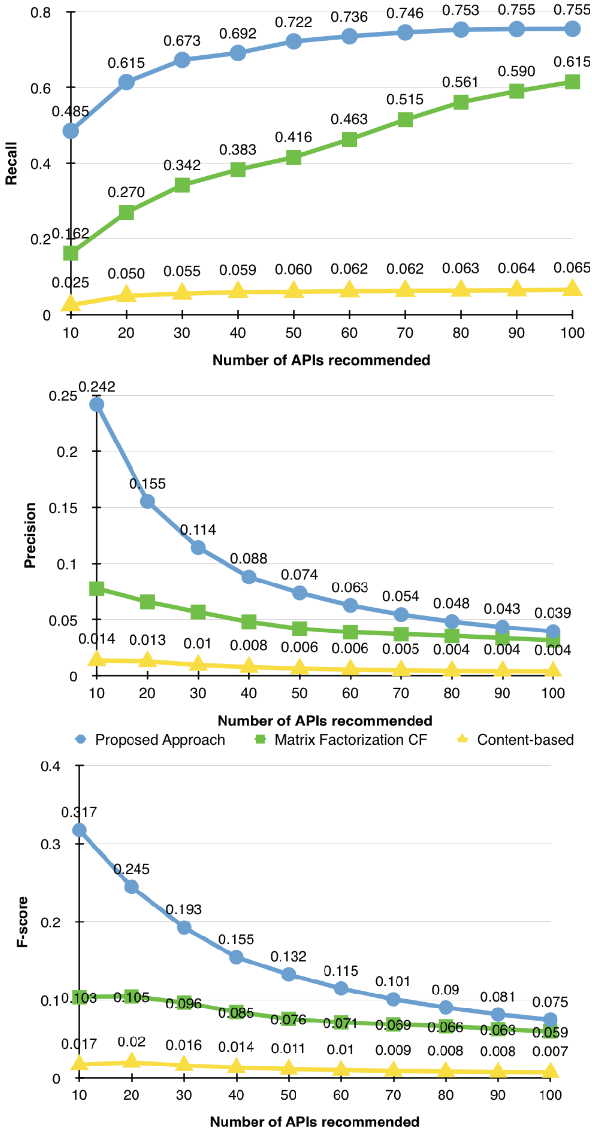

Fig. 5. Average Recall, Precision and F-score comparison for the proposed approach, individual matrix factorization CF and LDA with cosine similarity. The number of recommended APIs was increased to analyze its effect on the performance.

$$Average\ Precision = \frac{\sum_{k=1}^{n} (\ P(k) \times rel(k)\ )}{|Relevant\ APIs|}$$

where k is the rank/position of the API in the list; $P(k)$ is the precision at cut-off k and $rel(k)$ is the change in recall or in simple words, it indicates if the API at

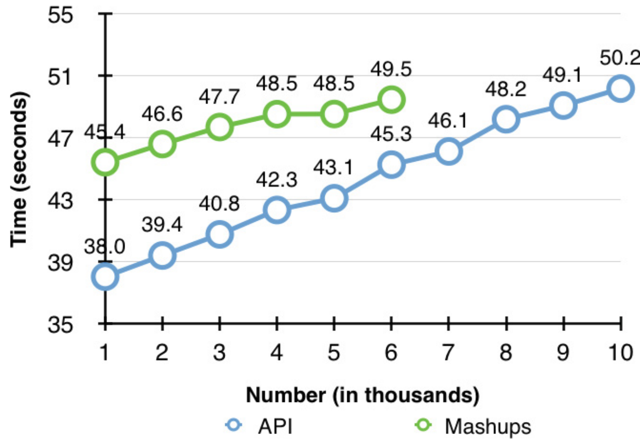

Fig. 6. Influence of increasing number of APIs and mashups on matrix factorization's performance

rank k is relevant or not. Its value is 1 if the API is relevant, otherwise it is 0. As shown in Table 1, the average recall of our approach is 0.62, which is better than the one in [6]. Our approach achieves a better precision as well, i.e., 0.27 vs 0.16.

Finally, to evaluate the performance of the approach, performance of matrix factorization phase was evaluated. This is because its performance is directly proportional to the increasing number of APIs and mashups as it increases data density. Also, performance of LDA depends on the number of topics and first phase could be performed offline, therefore, it is not considered in the evaluation of performance of the complete approach. Figure 6 shows that time required to train the matrix factorization model and predict ratings increases with the increase in the number of APIs and mashups.

5 Conclusion

In this paper, we propose a novel approach to recommend relevant APIs with good QoS for mashup creation. The proposed approach integrates functionality and usage history to discover functionally relevant APIs and then uses their popularity in existing mashups to achieve a ranked list of candidate APIs. By aggregating multiple sources of information, the proposed approach helps discover a set of functionally relevant APIs for mashup creation while performing QoS based ranking of these selected APIs so as to recommend the best ones to the user. The results of the experiments demonstrate that the proposed approach outperforms both individual matrix factorization and content-based (LDA plus cosine similarity) approaches. Moreover, it also has better accuracy as compared to other competitive methods.

Our future work will focus on improving recommendation accuracy by exploring advanced topic modeling techniques and checking the orchestration compatibility of candidate APIs.

Acknowledgments. This work was supported by US National Science Foundation under grant DUE-1141200.

References

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
2. Cao, B., Liu, J., Tang, M., Zheng, Z., Wang, G.: Mashup service recommendation based on user interest and social network. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 99–106, June 2013
3. Cappiello, C., Matera, M., Picozzi, M., Daniel, F., Fernandez, A.: Quality-aware mashup composition: issues, techniques and tools. In: 2012 Eighth International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 10–19, September 2012
4. Jung, J., Lee, K.-H.: Socially-enriched semantic mashup of web APIs. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) Service Oriented Computing. LNCS, vol. 7636, pp. 389–403. Springer, Heidelberg (2012)
5. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **8**, 30–37 (2009)
6. Li, C., Zhang, R., Huai, J., Sun, H.: A novel approach for API recommendation in mashup development. In: 2014 IEEE International Conference on Web Services (ICWS), pp. 289–296, June 2014
7. Liu, X., Zhao, Q., Huang, G., Mei, H., Teng, T.: Composing data-driven service mashups with tag-based semantic annotations. In: 2011 IEEE International Conference on Web Services (ICWS), pp. 243–250, July 2011
8. Maaradji, A., Hacid, H., Skraba, R., Lateef, A., Daigremont, J., Crespi, N.: Social-based web services discovery and composition for step-by-step mashup completion. In: 2011 IEEE International Conference on Web Services (ICWS), pp. 700–701, July 2011
9. Xia, B., Fan, Y., Tan, W., Huang, K., Zhang, J., Wu, C.: Category-aware API clustering and distributed recommendation for automatic mashup creation. IEEE Trans. Serv. Comput. **PP**(99), 1 (2014)
10. Xu, W., Cao, J., Hu, L., Wang, J., Li, M.: A social-aware service recommendation approach for mashup creation. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 107–114, June 2013
11. Yu, Q.: Cloudrec: a framework for personalized service recommendation in the cloud. Knowl. Inf. Syst. **43**(2), 417–443 (2015)