A Fast and Accurate Way for API Network Construction Based on Semantic Similarity and Community Detection

Xi Yang, Jian Cao

A fast and accurate way for API network construction based on semantic similarity and community detection

Xi Yang and Jian Cao*

Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai 200240, China
{ciciyang,cao-jian}@sjtu.edu.cn

Abstract. With the rapid growth of the number and diversity of web APIs on the Internet, it has become more and more difficult for developers to look for their desired APIs and build their own mashups. Therefore, web service recommendation and automatic mashup construction becomes a demanding technique. Most of the researches focus on the service recommendation part but neglect the construction of the mushups. In this paper, we will propose a new technique to fast and accurately build a API network based on the API's input and output information. Once the API network is built, each pair of the connected APIs can be seen as generating a promising mushup. Therefore, the developers are freed from the exhausting search phase. Experiments using over 500 real API information gathered from the Internet has shown that the proposed approach is effective and performs well.

Keywords: Web API, Mashup Construction, API Network

1 Introduction

With the development of Web 2.0 and related technologies, more and more service providers publish their resources on the Internet, usually in the form of web service APIs [8]. As a result, mashup technology, which aims for software developers to use multiple individual existing APIs as components to create value-added composite services [9] becomes a promising software development method in service-oriented environment [11]. To meet the developers' demand, several online mashup and web service repositories like ProgrammableWeb are established. For example, on ProgrammableWeb, there are altogether 17422 different APIs and 7881 mushups. These APIs and mushups are either manually collected by the website from different API providers or uploaded by the user. This manual work requires a lot of time for collecting and daily maintenance.

As a result of the rapid growth of the number and diversity of web APIs, it is a difficult task for developers to construct their own mashups. If a user want to

* corresponding author

search for some suitable APIs and construct a mushup with a new functionality, he or she will need to do the next steps: first to select a few APIs from a variety of APIs based on the user's demand and the APIs' functionality. In this phase, normally a bunch of similar APIs will meet the requirement. Then the user must look into each API's documentation and figure out if the APIs can actually cooperate with each other and decide on how the data flows between each 2 individual APIs, meaning whose output can be the input of another. The whole process is so time consuming and cumbersome.

To address the above problem, some researchers adopt service recommendation techniques for service discovery to release manpower and contribute to a fast mashup construction. The main method includes semantic-based, Qos-based and network-based recommendation. The semantic-based way mainly focus on using the Natural Language Processing(NLP) tools for information retrieval. Latent dirichlet allocation (LDA) [5] and the later tagging augmented LDA (TA-LDA) model [2] are proposed to calculate the semantic similarity between the query and the API content. This method requires the WSDL document of the service, which is often uneasy to obtain nowadays with the privilege of RESTful services. Qos-based approach centers on the nonfunctional properties of web services and helps the user find suitable services among functionally equivalent services. Algorithms such as Collaborative Filtering [12] are employed. Similar as the semantic approach, Qos information is not always available. Another way is to use network analysis in service recommendation. Service usage patterns [10] and user behaviour patterns [4] are investigated to understand the correlation among the services and between the users and the services. The network-based approach does not take the users' functionality demand into account and therefore cannot return the user with useful service content information.

Even though the existing approaches show improvements in service recommendation, most of the methods concern about searching for the suitable APIs based on the user's query and ignore the subsequent steps. After the user's query, these methods return a bunch of unorganized APIs and don't show the user how exactly the mashup is constructed. The users still need to search for the documentation for each API and manually construct the mashup.

An ideal way to overcome the existing problem is to maintain an API network. Mashups can be seen as the connected APIs. These connections not only describe the matchable information, but also can indicate the direction of the data flow from one API's output to another's input. Therefore we proposed a fast and accurate way to build this API network based on the existing RESTful APIs' information and provide the user about the data flow information inside the mashup. Once the user has decide on which APIs to use, we can return if these APIs and actually work as a mushup and how it works. Or if a new web service API is published, we can quickly determine which existing services can construct a mushup with the new one.

Our rationale is that if 2 separate web service APIs can construct a mushup, there must be a data flow between these 2 APIs. One of the API's returned value can be used as the other's input data. We call it a pair of APIs that

match with each other and can be components in the same mashup. Most of the service providers also publish the API references on their websites, containing the descriptions for the API input and output information.

Based on the above facts of APIs in the same mashup and the availability of the API information, we propose a new way by calculating the semantic similarity of the input and output description to determine if the 2 APIs match with each other to automatically construct a mashup. Also, to avoid doing the match work for each pair of input and output, we adopt the algorithm of community detection and cluster the APIs into several groups. Thus for the newly incoming APIs, we can simply compare them with the group centers instead of all the the existing APIs. In this way, the computation for the matching is greatly reduced and the calculating process is accelerated.

2 Background

2.1 Motivation Example

Suppose the developer would like to build a new mashup which can recommend the tourist a complete travel route including where to visit, where to stay, where to eat, how to travel between 2 different locations and show all these information on a map. The developer may need to search separately for the key words like "map", "travel" ,"hotel" and so on. If user search for each of the key words listed above on ProgrammableWeb, the server will search in its 17440 APIs and return 1075/542/159 results separately for these 3 queries. Of course, among all these results, most of them are irrelevant and can not cooperate with each other to build the desired mashup. Such single query can not tell the developer whether these APIs can be used together. It is challenging for the developer to check for each API and decide which to chose and how to use them.

But if we already constructed an API network and have the information about the possible matchable APIs, knowing about how the data transfers inside these APIs, we can return the developer with a well-organized flow chart as shown in Fig 1. This chart contains several APIs and illustrates the usage of the APIS. The example in Fig 1 shows that the output of API a and API b can be the input of API c. The developer only need to choose from a small number of candidates and then start on building the new mushup.

2.2 Requirements and Challenges

By looking into the APIs listed on ProgrammableWeb, we can find out that by June 2017, there are 17440 services collected. For each web service, there are on average more than 30 APIs provided. For each API, the number of input parameters and the output ranges from several to several tens. Take the web service FourSquare as an example, one of the API description is shown in Fig 2. There is a API name containing the keywords to describe the functionality, a short API description, an url showing the API endpoint, a table of general information

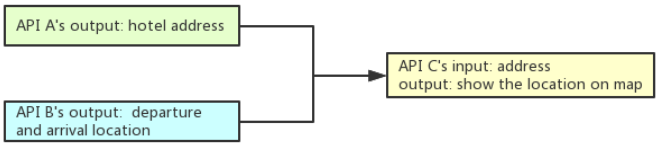

Fig. 1. An example of the API flow chart.

about the API, a table of input parameters and a table of output parameters. Coming along with all the input and output parameters, short descriptions are also provided. In our approach, we focus on the parameter descriptions since these descriptions give the semantic meaning of the parameters. If one input description and one output description from 2 separate APIs are talking about the same thing, which means the descriptions are semantically similar, we can conclude that these 2 APIs match with each other and can make up a mashup.

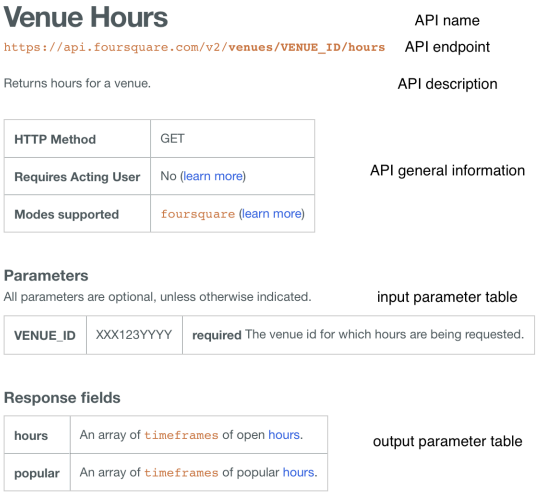

Fig. 2. FouSquare API example

In total, FourSquare contains 118 APIs, 397 input parameters and 110 output parameters. Even within this single web service, the semantic similarity calculation will take 397×110 times. If this one to one calculation needed to be done for all the 17422 APIs on ProgrammableWeb, the time consumed is unimaginable. Also, it is unrealistic to do the one to one matching every time when a new API

emerges. Therefore, how to do the match calculation fast and accurate enough becomes a demand.

Therefore, we identify the the major requirements for the mashup detection:

- **Comprehensive API combination detection.** The detection is based on the semantic similarity of the input and output. An good similarity calculation helps improving the accuracy for building the API network by finding as much matched pairs of input and output as possible and exclude the others.
- **Fast calculation.** With the rapid growth of the number of APIs, similarity calculation for each pair of input and output is unrealistic. Instead of the one to one comparison, a good pre-processing phase to narrow down will be helpful to greatly accelerate the calculation.

3 Method Overview

To meet the above requirements, we proposed an approach to fast and accurately build the API network. The whole process contains 2 main parts as shown in Fig 3:

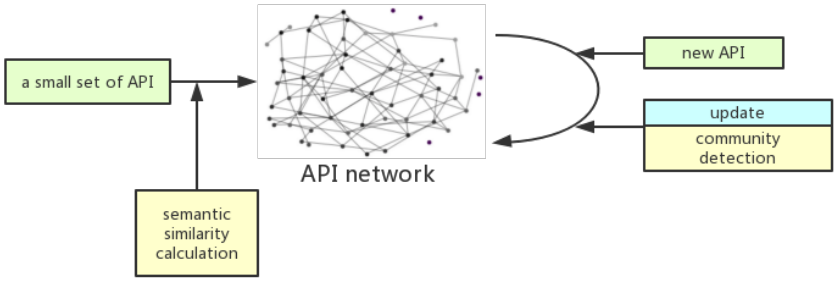

Fig. 3. Framework

Beginning with a small amount of APIs, we calculated the semantic similarity for each pair of input description and output description. This NLP technique is the first main part described in 3.1. After this calculation, all the semantically similar input and output will be connected. The results generated an undirected graph as shown in Fig 3. The nodes are sentences indicating all the input descriptions/output descriptions and the edges meaning that the linked nodes have similar semantics.

The second main part is based on this undirected graph. Here we adopted the community detection algorithm which is already widely used in social network. By using the community detection on this graph, all the nodes will be divided

into several groups with every node within the same group sharing the same semantics.

The whole construction of the API network is done in an incremental way. Every time we want to add a new API input/output into this network, in other words, find the matchable APIs for the new one, we can simply calculate semantic similarity of the new one with the center of each community. The community center should represent the whole community and extract as much common information as possible. It's defined as the most frequent words of all the nodes within the group. If the center of a group is semantically similar to the new one, we consider all the nodes in this group is possible to build mashups with the new API. So we can continue to check each node in this community. Otherwise, if one center of a group dose not mach the new one, the whole nodes in the group will be regarded as impossible to match new one and skipped. In this way, the calculation is greatly reduced. Every time a new node is added, the community detection algorithm will run again and reorganize the network.

In the following part, we will go through the details of semantic similarity calculation and community detection:

3.1 Semantic Similarity

The semantic similarity algorithm is based on WordNet [7] [3] and corpus statistics [6]. WordNet is a large lexical database of English. All nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms unordered sets(synsets), each expressing a distinct concept.

The detail is shown in Algorithm 1. The basic idea is to vectorize 2 sentences and calculate the cosine similarity between these 2 vectors. The way to vectorize the sentence is shown in function `Vectorize`. Each value in the vector indicates the corresponding word similarity calculated in the function `wordSimilarity`. 1 means that both sentences contain exact the same word and 0 means only one sentence contains the word and the other don't even contain a similar word.

The words in WordNet are constructed hierarchically on base of lexical knowledge. Each word has semantic connection to the others. The child node can be seen as a semantic subset of the ancestor node. For example, the node "boy" and "girl" might be the children nodes of node "children". The path between 2 nodes in the structure is a proxy for how similar the words are and the height of their Lowest Common Subsumer (LCS) from the root of WordNet shows how broad the concept is and less similar. Therefore, the word similarity is calculated by these 2 measurements as shown in Eq 1 with α and β being the weighted length and height:

$$similarity = e^{-\alpha}\frac{e^{\beta} - e^{-\beta}}{e^{\beta} + e^{-\beta}} \tag{1}$$

3.2 Community Detection

The community detection part used the louvain method described in [1]. The Louvain method is a simple and easy-to-implement yet efficient method for

Algorithm 1 Semantic Similarity

```
function SemanticSimilarity(sentence1, sentence2)
    w1 ← Tokenize(sentence1)        ▷ Tokenize both of the sentence into word sets
    w2 ← Tokenize(sentence2)
    j ← Union(w1, w2)                ▷ Get the join set of the above 2 word sets
    vector1 ← Vectorize(w1, j)
    vector2 ← Vectorize(w2, j)
    return CosineSimilarity(vector1, vector2)
end function
function Vectorize(w, j)
    vec ← initialized with the length equal to size(j)
    i ← 0
    for word in j do
        if word in w then
            vec[i] ← 1                          ▷ Set the value to 1 if word is in w
        else
            sim ← findSimilarWord(word, w)      ▷ If the word is not in w, then calculate the similarity between word and the most similar word in w.
            if sim ≥ threshold then  ▷ Means that there is actually a similar word in w.
                vec[i] ← sim
            else                                ▷ There is no similar word in w.
                vec[i] ← 0
            end if
        end if
        i ← i + 1
    end for
    return vec
end function
function findSimilarWord(word, w)
    maxValue ← MAX
    for element in w do
        maxValue ← max(maxValue, wordSimilarity(element, word))
    end for
    return maxValue
end function
```

identifying the communities in large networks. It is one of the most widely used method for detecting communities in large networks.

In this method, "modularity" is defined to describe the structure of a network, measuring the strength of the division of a network into groups. Intuitively, high modularity means the network has dense connection between nodes within groups but sparse connections between groups. The calculation for modularity is shown in Eq 2:

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - \frac{k_i - k_j}{2m}) \delta(c_i, c_j) \tag{2}$$

with

- m is the number of all edges in the graph;
- A_{ij} represents the edge between node i and j;
- k_i and k_j mean the sum of edges connected to node i and j;
- δ is a delta function.
- c_i and c_j are the communities of node i and j;

Louvain method is a greedy optimization method that attempts to optimize the modularity of a division of a network. The optimization is performed in 2 steps as shown in Algorithm 2. These 2 steps are iteratively repeated until a maximum modularity is gained.

Algorithm 2 Community Detection

Input: A network N.
Output: A division D of the netwrok with maximum modularity.

initialization: $D \leftarrow$ assign each node in N to its own group
initialization: $Q \leftarrow \texttt{modularity}(D)$ ▷ `modularity` calculates the modularity of division D using Eq 2
initialization: $\Delta Q \leftarrow Q$
while $\Delta Q > 0$ **do**
    for all node n **in** N **do**
        $maxQ \leftarrow \texttt{modularity}(D)$
        $maxD \leftarrow D$
        for all group g **in** neighbours of c_n **do**
            $D_{new} \leftarrow$ move n into g
            $Q_{new} \leftarrow \texttt{modularity}(D_{new})$
            if $Q_{new} > maxQ$ **then**
                $D \leftarrow D_{new}$
                $maxQ \leftarrow Q_{new}$
                $\Delta Q \leftarrow maxQ - Q$
                $Q \leftarrow Q_{new}$
            end if
        end for
    end for
end while
return D

All the input and output description data are connected based on the semantic similarity. Thus an undirected network is generated. In the following section we will show that the community detection also works well on this kind of network.

4 Experiments and Results

In out experiment, we crawled in total 580 APIs from the providers' website to see how fast and accurate when we use the method proposed above. All these APIs belong to 24 different categories and in total have 2775 input description and 638 output description. The average length for a single description is 9.67.

4.1 Accuracy of semantic similarity algorithm

To check whether the semantic similarity can denote the matchable information for the input and output pair, we applied this algorithm to every pair of input and output in our experiment dataset, in total more than 150000 pairs. The result is shown in Fig 4.

By manually checking, we can conclude that most of the input/output pairs don't match with each other(with the similarity less than 0.4). We find out that the higher the similarity is, the more likely this pair matches with each other. It is reasonable to use this semantic similarity to construct the API network. Therefore, we need to choose a threshold to decide whether a pair of input and output matches with each other. The threshold need to meet these 2 requirements:

- Small enough to include all the matchable pairs.
- Big enough to exclude all the impossible pairs.

We manually checked each threshold candidates ranging from 0.95 to 0.40 and finally choose 0.55 as the threshold. All the pairs with similarity greater than 0.55 are matchable and will be connected together.

4.2 Speed-up calculation using community detection

By checking the accuracy of semantic similarity in the former experiment, we constructed the most comprehensive API network since each pair of input and output is checked and any possible pair won't be missed. But if we take the time consumption into consideration, this one-to-one approach is not practical. The total time to finish the calculation is more than 110 hours. On average, every API takes about 0.2 hour to finish calculating. For every input and output description , more than 600 and 2500 times of calculation is needed separately. Also, this approach will become more and more slower because new APIs show up every day and the number is growing rapidly.

To check how the speed-up way with community detection works, we will compare its running time and comprehensiveness with the one-to-one method. The network this one-to-one method build is referred as the original network.

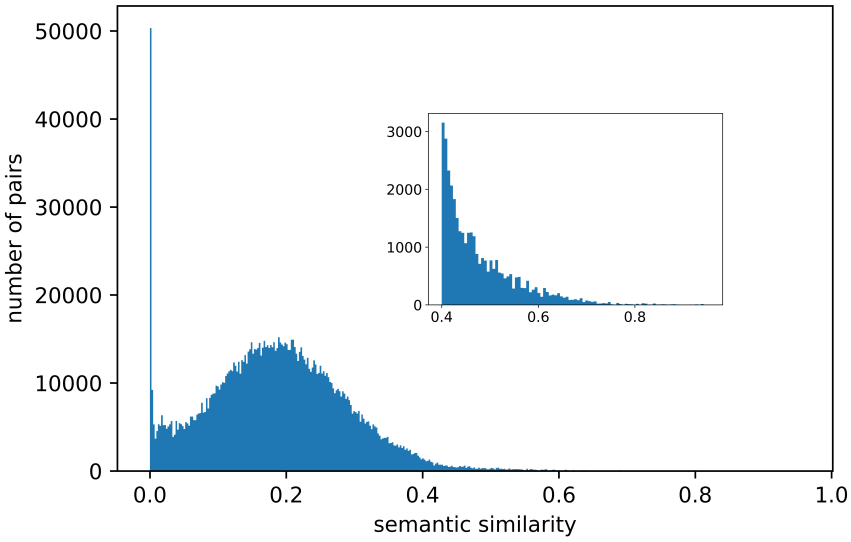

Fig. 4. The distribution of all the pairs according to their semantic similarity

To check the comprehensiveness, we randomly took out 1 API out of the original network. All the input, output and related connections belonging to this API are taken out. The other APIs are kept unchanged. This taken-out API will be added back into the network to build its own connections in the network using the speed-up way. The number of connections it builds will be compared with the connections in the original API network.

This experiment is repeated 50 times to get the convincing results as shown in Fig 5. Most of the nodes are slightly blow the standard line, meaning that the speed-up method find most of the matchable pairs with a small part missing. This is reasonable because speed-up method calculates only a subset of the APIs instead of calculating for all the pairs. Take the running time into consideration, this whole 50 times repetition takes only 2 hours, meaning that each API takes 0.04 hour on average, comparing with 0.2 hour on average for each API using one-to-one method. This method is comprehensive enough yet with a surprisingly good time-saving capability.

We also tried taking 67 APIs out of the original network and adding back all these 67 APIs one by one to see if the incremental way still remains effective. The result is shown in Fig 6. Even though the blue line remains lower than yellow line, it still finds around 72% connections comparing with the original network, indicating the speed-up method is practical and good enough to construct the API network. The average time it takes is 0.041 hour for a single API. This approach makes it possible to find connections quickly for a newly coming API.

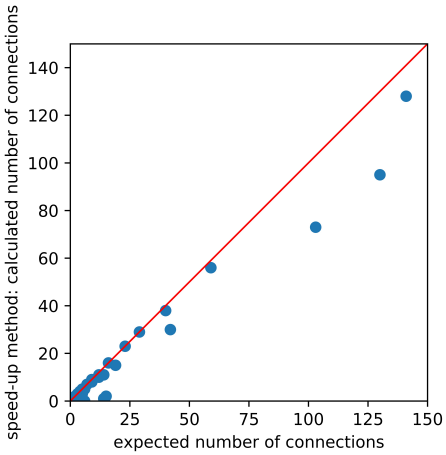

Fig. 5. Comparison for adding one API into network

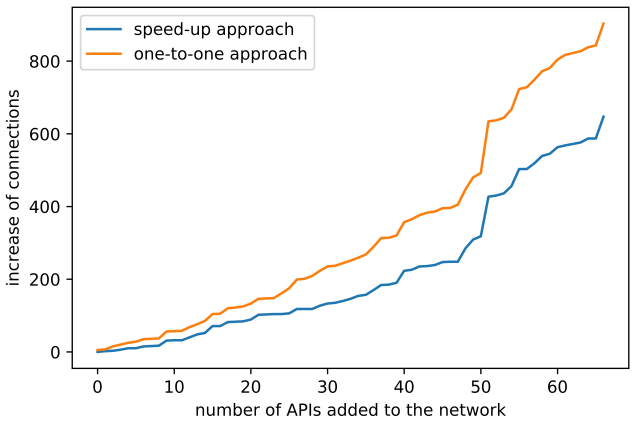

Fig. 6. Comparison for adding 67 APIs into the network

5 Conclusion

This paper present a new way of adopting semantic similarity and community detection for automatic generate the API network and assist mashup development in a incremental way. The experiment performed on hundreds of APIs demonstrates the effectiveness of the proposed method. It keeps a relatively good result yet reduces quite a lot of computation. This is very meaningful when a new API is released and to get the matchable information for this API without delay. Also, in this way, we can maintain an API network representing all the matchable pairs and assist the developer for mashup construction.

Acknowledgment This work is supported by China National Science Foundation (Granted Number 61472253)

References

1. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008), http://stacks.iop.org/1742-5468/2008/i=10/a=P10008
2. Chen, L., Wang, Y., Yu, Q., Zheng, Z., Wu, J.: Wt-lda: User tagging augmented lda for web service clustering. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC. Lecture Notes in Computer Science, vol. 8274, pp. 162–176. Springer (2013), http://dblp.uni-trier.de/db/conf/icsoc/icsoc2013.html#ChenWYZW13
3. Fellbaum, C. (ed.): WordNet: an electronic lexical database. MIT Press (1998)
4. Huang, K., Yao, J., Fan, Y., Tan, W., Nepal, S., Ni, Y., Chen, S.: Mirror, Mirror, on the Web, Which Is the Most Reputable Service of Them All?, pp. 343–357. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-45005-1_24
5. Li, C., Zhang, R., Huai, J., Guo, X., Sun, H.: A probabilistic approach for web service discovery. In: Proceedings of the 2013 IEEE International Conference on Services Computing. pp. 49–56. SCC '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/SCC.2013.107
6. Li, Y., McLean, D., Bandar, Z.A., O'Shea, J.D., Crockett, K.: Sentence similarity based on semantic nets and corpus statistics. IEEE Transactions on Knowledge and Data Engineering 18(8), 1138–1150 (Aug 2006)
7. Miller, G.A.: Wordnet: A lexical database for english. In: Proceedings of the Workshop on Human Language Technology. pp. 468–468. HLT '94, Association for Computational Linguistics, Stroudsburg, PA, USA (1994), http://dx.doi.org/10.3115/1075812.1075938
8. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: 05462 service-oriented computing: A research roadmap. In: Cubera, F., Krämer, B.J., Papazoglou, M.P. (eds.) Service Oriented Computing (SOC). No. 05462 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2006), http://drops.dagstuhl.de/opus/volltexte/2006/524
9. Sheth, A., Benslimane, D., Dustdar, S.: Services mashups: The new generation of web applications. IEEE Internet Computing 12, 13–15 (2008)
10. Tan, W., Zhang, J., Foster, I.: Network analysis of scientific workflows: A gateway to reuse. Computer 43(9), 54–61 (Sept 2010)
11. Xia, B., Fan, Y., Tan, W., Huang, K., Zhang, J., Wu, C.: Category-aware api clustering and distributed recommendation for automatic mashup creation. IEEE Transactions on Services Computing 8(5), 674–687 (2015)
12. Zheng, Z., Ma, H., Lyu, M.R., King, I.: Qos-aware web service recommendation by collaborative filtering. IEEE Transactions on Services Computing 4(2), 140–152 (April 2011)