1-1-2011

Collaborative filtering based service ranking with invocation histories

Qiong Zhang
Ryerson University

COLLABORATIVE FILTERING BASED SERVICE RANKING WITH INVOCATION HISTORIES

By

Qiong Zhang

B.Sc. in Computer Science, Ryerson University, Canada, 2007

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Computer Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2011

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

QIONG ZHANG

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

QIONG ZHANG

BORROWER'S PAGE

Ryerson University requires the signatures of all persons using or photocopying this thesis.

Please sign below, and give address and date.

Name	Signature	Address	Date

COLLABORATIVE FILTERING BASED SERVICE RANKING WITH INVOCATION HISTORIES

Qiong Zhang
Master of Science, Computer Science, 2011
Ryerson University

ABSTRACT

Collaborative filtering based recommender systems have been very successful on dealing with the information overload problem and providing personalized recommendations to users. When more and more web services are published online, this technique can also help recommend and select services which satisfy users' particular Quality of Service (QoS) requirements and preferences. In this thesis, we propose a novel collaborative filtering based service ranking mechanism, in which the invocation and query histories are used to infer the users' preferences, and user similarity is calculated based on invocations and queries. To overcome some of the inherent problems with the collaborative filtering systems such as the cold start and data sparsity problem, the final ranking score is a combination of the QoS-based matching score and the collaborative filtering based score. The experiment using the simulated data proves the effectiveness of the proposed algorithm.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Dr. Cherie Ding for her valuable support and guidance in helping me to go through all the difficulties in my work. Her precious suggestions and guidance have greatly enhanced my knowledge and skills in research and have significantly contributed to the completion of this thesis.

In addition, I would like to thank Dr. Alireza Sadeghian, Dr. Marcus Santos, and Dr. Isaac Woungang who have reviewed my thesis and have given me valuable comments which enabled me to improve my thesis.

Also, I would like to acknowledge the support of the Computer Science Department of Ryerson University and my fellow students. Specially, I would like to give my sincere thanks to Delnavaz who has helped me in preparing the dataset.

Finally, I would like to express my deep appreciations to my family, relatives, and friends who have motivated and supported me during these years of study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

AHP: Analytical Hierarchy Process

CP: Constraint Programming

DL: Description Logic

HTTP: Hypertext Transfer Protocol

IA: Iterative Algorithm

IR: Information Retrieval

MCDM: Multi-Criteria Decision Making

MIP: Mixed Integer Programming

PCC: Pearson Correlation Coefficient

QoS: Quality of Service

SICS: System for Implicit Culture Support

SOA: Service Oriented Architecture

SOAP: Simple Object Access Protocol

UDDI: Universal Description, Discovery and Integration

URL: Uniform Resource Locator

WSDL: Web Service Description Language

XML: Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Background and the Problem Statement

1.1.1 Background

Web services are self-contained software components that can be described by Web Service Definition Language (WSDL), published into a web service registry such as a UDDI (Universal Description Discovery Integration) registry and discovered through certain discovery functions. They are based on standards such as XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and HTTP (Hypertext Transfer Protocol) to support interoperability, platform independency, and reusability for enterprise application integration and cross-organizational integration. Web service is an implementation of SOA (Service Oriented Architecture) which defines architecture of designing a software system through services. There are three primary roles in the SOA: service provider, service requester (client), and service registry.

Service providers describe their services using WSDL files and publish them with a service registry (e.g. UDDI). WSDL files describe web services as network endpoints that are available for public access. A client can submit a query to the registry with the discovery agent to find the services that satisfy their requirements. Then, based on the information returned from the registry, the client can select a most suitable web service and invoke it through a provider with a binding operation.

1.1.2 Problem Statement

Nowadays, web service has become a more and more popular way of implementing business solutions. More and more web services satisfying the same or similar functionalities are available online. With the proliferation of web services, it could be very hard for users to choose among a list of functionally matching services returned from a service registry. Therefore, how to effectively rank services to satisfy a user's personal preferences becomes one of the key challenges for the service oriented computing community.

Service selection is generally considered as a two-step process: matching based on functional requirements, and then filtering and ranking based on non-functional, e.g. Quality of Service (QoS) requirements. Researchers are trying to improve the performance of the service selection systems using different approaches and algorithms – collaborative filtering is one of them.

Collaborative filtering based recommender systems [1] [2] have been very helpful and successful on dealing with the information overload problem and providing personalized recommendations to users for their online browsing or shopping activities. The main idea behind is that if a user shares similar interests/tastes/opinions with other users on some items (e.g. books, movies, web pages), it is very likely that this kind of similarity will hold for items which are new to this user. Based on this rationale, the items selected by similar users will be recommended to the user.

So far there are only a limited number of research works using the collaborative filtering techniques in web service selection systems. A few examples include predicting the QoS values which might be experienced by service users [3] [4], recommending services based on user feedbacks [5] [6] or previous usage history data [7], etc. In this thesis, we propose to use the collaborative filtering techniques in the service selection process, with a focus on the second step

– QoS-based service ranking and selection. After the functionally matching services are identified, we could rank them based on the current QoS requirements, as well as how they were selected and invoked previously by other users who have similar QoS requirements.

1.2 The Proposed Methodology

QoS properties describe non-functional aspects of a web service. In addition to functional properties, providers can also advertise QoS properties of their services using a QoS description language in an extended UDDI registry for QoS-aware discovery [8]. Requesters can specify their desired QoS criteria in their queries. QoS properties are often used to evaluate the degree that a web service meets the specified criteria in a service request and they have become some of the most important factors in selecting a web service.

There can be many QoS attributes for a web service, such as response time, latency, availability, reliability, and security. Some researchers have studied their characteristics and have proposed different ways of categorization [9]. Many of the QoS attributes' values are changing over time, such as response time, latency, availability. Many factors can cause this change, such as performance improvement made by providers**,** network condition, and so on. Moreover, providers' offers may differ with the actual performance, and different users can have different QoS experiences because of their different expectations and different conditions. Hence, web service selection can involve decision-making based on multiple QoS metrics under changing and diversified environments.

Collaborative filtering systems use user feedback, such as ratings, to reflect users' opinions or experiences on performance or quality, which would be a major factor to be considered in a recommender system. The explicit user feedback systems, such as reputation

based or community feedback based systems, usually involve human efforts to provide feedback or ratings [10][11].

However, not all users are willing to provide feedback or ratings after each usage [1], and furthermore, the user ratings might not be accurate or trustworthy [12]. Since explicit user feedbacks are often impractical or unable to completely reflect users' true opinions or experiences, we propose to use implicit user feedback information which can be extracted from usage logs. For example, in a collaborative filtering system, based on other similar users' experiences, we can know that a user who searches for the flight information may also be interested in the hotel information. Or, if many users bought tickets from a travel agency, it indicates that people are satisfied with this agency's ticket booking service, and as a result, this agency should be recommended to new users.

Different types of usage data have been used in web search recommender systems. The click-through data [13] and users' searching history data [14] have been used to improve the search engine performance. Server log analysis is a common technique to discover the user interest and recommend web pages [15].

In this thesis, we are going to use invocation logs and query logs for usage data. From the invocation log, we can find user's invocation time and invocation frequency on different services. Query log saves user's expectation on QoS properties in each request. From these logs we can infer the user's expectation and preferences on QoS properties as well as service performance. For example, if a service was invoked by a user with some sort of QoS expectations or requirements many times, we can infer that this service can satisfy these QoS requirements or it is the most suitable service among other available services for this user, even

if the offered QoS values are not exactly matching with the query. Different users can have different criteria on suitability and expected performance for the same QoS requirements.

We consider users who have similar QoS expectations and invocation histories as similar users. If a service has been invoked by a group of similar users, we consider this service can satisfy this type of users' similar request. Through the invocation frequency, we can infer the level of satisfaction; through the invocation time, we can infer user's preference change or service performance drifting over time. Therefore, we can rank a service based on its level of popularity, which is a function of the frequency and time, among a group of similar users.

To find similar users, we propose to use QoS value requirements as well as preferences to build user profiles, and then generate the user-user similarity matrix using an offline computation routine. In this routine, we use Kendall tau coefficient [16], which is to measure the agreement between two ranked lists, to calculate the similarity between the preferences on various QoS attributes from two users, and then we use Jaccard Coefficient [17] to calculate the similarity between QoS value requirements from two users.

1.3 Objectives

In this thesis, we have three main objectives. Firstly, we use the invocation and query history data, especially the QoS query part to build a collaborative filtering system. Secondly, we propose a unique selection and ranking algorithm which could take advantages of existing QoS-based selection models and overcome some of the shortcomings (e.g. cold start problem) of traditional collaborative filtering systems. Finally, we introduce a practical architecture model for the web service selection system using collaborative filtering techniques.

1.4 Thesis Outline

The remainder part of the thesis is organized as follows:

Chapter 2 first reviews different QoS-based service selection models. Then, the research efforts that are more closely related to our work such as the researches on log analysis, and a number of recommender systems that use collaborative or content based approach are reviewed.

Chapter 3 explains the system architecture, the history data used for selection and ranking, the algorithms used in finding similar users, as well as the ranking algorithm in details. Then, we assess the efficiency of our algorithms using the time complexity analysis. We also use a few use case scenarios to illustrate the similarity computing and ranking algorithm steps.

In Chapter 4, we explain the experiment used to evaluate our system, by discussing the details about the experiment design, dataset used, as well as the result analyses.

Finally, in Chapter 5, we conclude our thesis with a summary of results and analysis. Our future research directions are also discussed in this chapter.

CHAPTER 2
RELATED WORKS

In this chapter, we will first review various QoS-based service selection models proposed in recent researches. Then, we will review some of the literatures that are more closely related to our work: server log analysis in IR (Information Retrieval) systems, and recommender systems that use collaborative filtering or content-based filtering mechanisms in their service selection processes.

2.1 QoS-based Service Selection Models

QoS-based web service selection is usually considered as an optimization problem. Researches in this area have proposed a number of different approaches. These approaches include Description Logic (DL), Constraint Programming (CP), Mixed Integer Programming (MIP), Multi-Criteria Decision Making (MCDM), skyline computation, etc. These approaches have provided optimized matching and selection based on users' requirements and providers' offers on QoS parameters. However, one of the major drawbacks of these approaches is that they did not consider the actual service performance in a dynamic execution environment in their selection processes, i.e. different users' experiences on QoS performance is different based on their individual conditions.

CP and MIP have been used in semantic matching and selection where multiple requirements on QoS parameters are treated as constraints [18] [19]. In [19], a semantic QoS aware framework was proposed to deal with multiple constraints on QoS requirements. DL reasoning was used to ensure the semantic matching on functional requirements. CP was

employed in the selection process where QoS constraints were converted into Constraint Satisfaction Problems (CSP). Combining these two technologies, the best service was found based on optimization of a global utility function which is a weighted combination of all interested QoS attributes. In this paper, a weighed combination of QoS attributes based on different user's interests was used in its selection algorithm. However, it failed to deal with different user's actual performance under different execution conditions. In [18], semantic matchmaking based on multiple QoS attributes was explored with CP and MIP approaches. And the experiment results showed that MIP outperformed CP. Similar approach was adopted in [20] where a broker used a global utility function and a cost function to select services which can optimize the global utility function for a client based on the client's cost constraint and the provider's cost function. Both [18] and [20] focused on optimized selection without considering personalized selection based on their different requirements and conditions.

Users' QoS preferences or priorities were further considered in [21] where the MCDM techniques were adopted for quality optimization. This paper also used weights to express the priorities on QoS parameters such as response time, latency, and availability. However, same as what we mentioned before, this approach did not consider different users' experiences either. In [22], an MCDM approach has been used to solve requester's dynamic preference on service configuration in large value spaces. A different approach, AHP (Analytical Hierarchy Process), which is also a method used in solving MCDM problems, was applied in [9] where AHP was used as an underlying mechanism in QoS based ranking with the proposed QoS property model. Instead of using simple aggregation of different QoS attributes to get an overall evaluation for optimization, AHP was used in modeling the multiple QoS criteria into a hierarchical structure which consists of multiple phases that fine-tune each level of QoS properties. The ranking was

obtained by sorting the final ranking vector which was an aggregation of each level of ranking vectors. This work focused on the QoS ontology model where most of the data were assumed to be provided by client or determined by the system. User experience was not the major focus of this research. Both approaches, [22] and [9], had the similar problems as [21].

The major limitations on above optimization approaches include: (1) they are based on the assumption that QoS values do not change over time; (2) different users have the same QoS performance.

Service skyline is another approach in dealing with multi-criteria problem in either functional level [23] or QoS level [24] to find an optimal service offer. The method proposed in [24] could address the problems mentioned in (1) by computing service skyline and *p*-dominant service skyline respectively. Here, the concept of dominance is defined as: for two objects *X* and *Y* which are described by a set of parameters or properties, *X* dominates *Y* if and only if *X* is better or equal to *Y* in all parameters and *X* is better than Y in at least one of the parameters. The *p*-dominant service skyline is defined as a set of providers with the probability of dominance by any other providers less than *p*. In this case, each individual transaction was looked at in detail regarding all QoS properties under consideration. And the probability which is a percentage of dominance on all transactions was computed for the providers. A transaction log was used to capture actual QoS performance for each transaction. However, this mechanism suffered from the following drawbacks: 1) It cannot capture a user's preference or priority on QoS attributes. All parameters are considered as equally important in computing skyline and users' experiences are considered as equally important. Hence, it cannot provide personalized recommendation regarding a user's individual preference. 2) There might be no *p*-dominant providers found based on the definition of dominance when there are many parameters and none of the transactions of a

service of a provider p_1 is equal or better than all transactions of a service of a provider p_2 in all parameters and is better in at least one parameter and vice visa.

In this thesis, our collaborative filtering based approach is to provide ranking of a service based on user's preference as well as other similar users' implicit experiences by employing the log data. We also consider differences among different users' experiences, as well as variance on QoS expectation and performance over time.

2.2 Log Data Used in Web Search Personalization

There are many researches on web usage mining for web personalization [15]. Web usage data records users' interaction with the Web and could usually be stored in a log file which is located at either client side or server side. Client side log can collect usage data from an individual user who often interacts with multiple web sites, whereas server side log can collect usage data from multiple users who access the web site hosted on the server. Hence, client side log is often used in content-based recommender systems, and server side log can be used in both content-based and collaborative filtering based recommender systems.

There are different types of usage data, including the browsing history, the click-through data, time spent on a page, actions applied on a page (such as printing and saving), the transaction history, and so on. Search engines normally rank the level of relevance of a web document based on the frequency of the query keywords appeared in the content of the document. However, for short and ambiguous queries, search engine performance will be deteriorated. The click-through data was used in [13] to improve the performance of the search engine under such ambiguous conditions. The click-through data was extracted from a large amount of log data collected by search engine servers. The log normally contains search queries,

followed by the URL of the web page clicked by the user. The user's click stream reflects the user's opinion about the page relevance. An Iterative Algorithm (IA) was proposed to compute the page similarity as well as the query similarity based on the following concept: web pages visited by similar queries are similar; and search queries visiting similar web pages are similar. The experiment showed a big improvement on the search performance.

The effectiveness of using implicit user feedbacks in web search ranking has been studied in [14]. This paper modeled the user search behaviour as a combination of "background" information and "relevance" information where "background" information represents noise information. User actions for each search result were represented as a vector of features. It could include any type of user interaction that is collected by search engine logs and these data are categorized into click-through features, browsing features, and query-text features. Then, a ranker was trained to discover feature values that are relevant on search results to produce a trained user behaviour model which was used to help the ranking process. It used a simple merge algorithm which computed the merge score of a document based on its rank from the implicit feedback-based ranks and the original ranks. The result showed a significant improvement in the final performance.

2.3 Recommendation Systems and Web Service Selection

Since mid-1990's, recommender system has become an active area of research. Various systems and algorithms have been proposed. Some well-known recommender systems include MovieLens [11], eBay [25], and Amazon.com [26]. Early recommender systems are mostly based on user ratings and recommendation problems are usually reduced to predicting ratings for unknown items for the user [1] [2]. There are three main categories of recommender systems:

content-based, collaborative filtering based, and hybrid system which combines these two approaches.

Content-based system recommends items based on similar features in the items that the user has preferred in the past. For example, in a movie recommender system, it tries to find the common features (directors, actors, subject matters, etc.) contained in the movies that the user preferred and/or rated in the past. Then, these features will be considered as user preference which will usually be stored in a user profile [10]. When a user searches for movies, a list of movies matching the search criteria such as keywords will be returned. Among these returned movies, the movies that have the most similar features as the profile based on certain criteria will be recommended. Some of the problems with the content-based system are: (1) it cannot distinguish the quality of the items if they possess the same or very similar features; (2) it is difficult to recommend an item that is not similar to any item the user has ever selected.

Collaborative filtering based systems try to overcome these weaknesses of the content-based systems. Instead of analyzing contents, it predicts whether a new item will be preferred by the user or how much the user will like the item based on other similar users' interests or preferences. Rating is often used in collaborative system to reflect the level of satisfaction or quality [1]. The similar users are found based on previous feedbacks (e.g. ratings) on commonly interested items by applying a similarity algorithm, such as Pearson Correlation Coefficient (PCC). The bigger the similarity value, the higher the level of similarity. The collaborative recommendation process usually has three steps: finding a list of items that matches the user query, finding the similar users, and ranking the items based on similar users' feedbacks.

The problems for the collaborative systems are cold start problems for new users, new items, and the data sparsity problem. Since the recommender system recommends items based on

similar users' preference, for new users who have no or very limited feedback information recorded in the system, it is difficult to find their similar users and hence the system is not able to provide recommendations. Similarly, for new items, there is no or very few number of users who have selected them before, and hence the feedback information on these items is very limited. Again, the recommender system is not able to recommend these new items. The data sparsity problem happens when there is a large number of users and items in the system, however each user only uses a few items and each item is only used by a few users, so that we will end up with a sparse user-item matrix. This is a common problem in many recommender systems [1].

Recommendation algorithms have been used in web service selection in different ways. The first usage is the QoS prediction, through which a user could select services based on their predicted QoS values. In [3], collaborative filtering was used and both positive and negative correlations were taken into consideration when calculating predicted values. A hybrid approach which combined collaborative and content-based filtering was proposed in [4] in order to solve the data sparsity problem in the user-item matrix. A confidence value was also calculated for each prediction. In both papers, individual observed QoS performance data were explicitly collected.

The second usage is service selection and ranking. In [5], the recommendation score for a service was calculated based on ratings on this service from similar users and ratings on similar queries. In [27], service selection was based on both objective and subjective QoS values. Collaborative filtering was used in subjective QoS information process. In both papers, [5] and [27], the ratings were assigned explicitly by the users. In [12], service usage data were implicitly collected. It was computed as the invocation frequency of a service among all services within the same service class. Then the user similarity was calculated based on the average frequency for

each service class using Pearson Correlation Coefficient (PCC). The final service ranking was determined by both user group finding through collaborative filtering and service dependency identification through association rule mining.

A community was set up in [7] to collect the implicit user information such as the service invocations following the user requests. The recommendation process was based on the query to find the possible actions (invocations) depending on the "culture" [7]. Content-based filtering was used in [6] to predict the user rating on a service for the dynamic selection.

User feedbacks are fundamental in collaborative recommendation and personalization. Either explicit or implicit feedbacks have been used in those systems. User feedbacks were explicitly collected in systems such as [3], [4], [5], [6] and [27]. In the experiments of [3], volunteers collected QoS invocation performance values and reported the data. In [4], users input observed QoS performance into an input handler in the WSRec system. In [5], [6], and [27], user ratings were explicitly provided by the users. The drawbacks about explicit data collection have been mentioned previously as it is impractical in many cases. Implicit data collection was adopted in [7] and [12]. In [12], service invocation data were collected for building the user profile, but it did not explain how these data were collected. In [7], QoS and invocation data were observed and collected from the client side as the historical data used to identify which services were relevant to which requests. These historical data were stored in its System for Implicit Culture Support (SICS) core. It used the SICS Remote Module to support information exchange with a client where a SICS Remote Client was installed.

2.4 Summary

From these reviews, we know that server logs have been used in web information retrieval for recommendations and have shown significant improvement in web search [13] [14]. But, in web service selection, there are not many researches using log data as described above. In our approach, query log and invocation log are used in a collaborative filtering based recommender system. Both QoS requirements in queries and invocation data are implicitly collected through a centralized mechanism.

CHAPTER 3

COLLABORATIVE FILTERING BASED SERVICE RANKING USING LOG DATA

Based on the discussions of previous chapters, we propose a collaborative filtering based service selection system using query and invocation history data. We extract related data from query and invocation logs and perform similarity computation. The collaborative filtering mechanism is then applied in the ranking process. In the following sections, we will explain the system architecture, the history data used for selection, the algorithms used in finding similar users, as well as the ranking algorithm in details.

3.1 System Architecture

The architecture model of our service selection system is shown in Figure 1, which consists of two parts: the client, and the extended UDDI registry.

In the client part, a client-side proxy is installed. It is responsible for forwarding the client requests to the registry and recording the usage history information into log files: query and invocation logs. It also provides the mechanism to collect the actual QoS data through monitoring the invocation processes. The monitored data is stored in the QoS data file.

The service selection and ranking mechanism along with the service description repository (WSDL) constitute the other part of our system - the extended UDDI registry, which consists of three major data repositories and five major components.

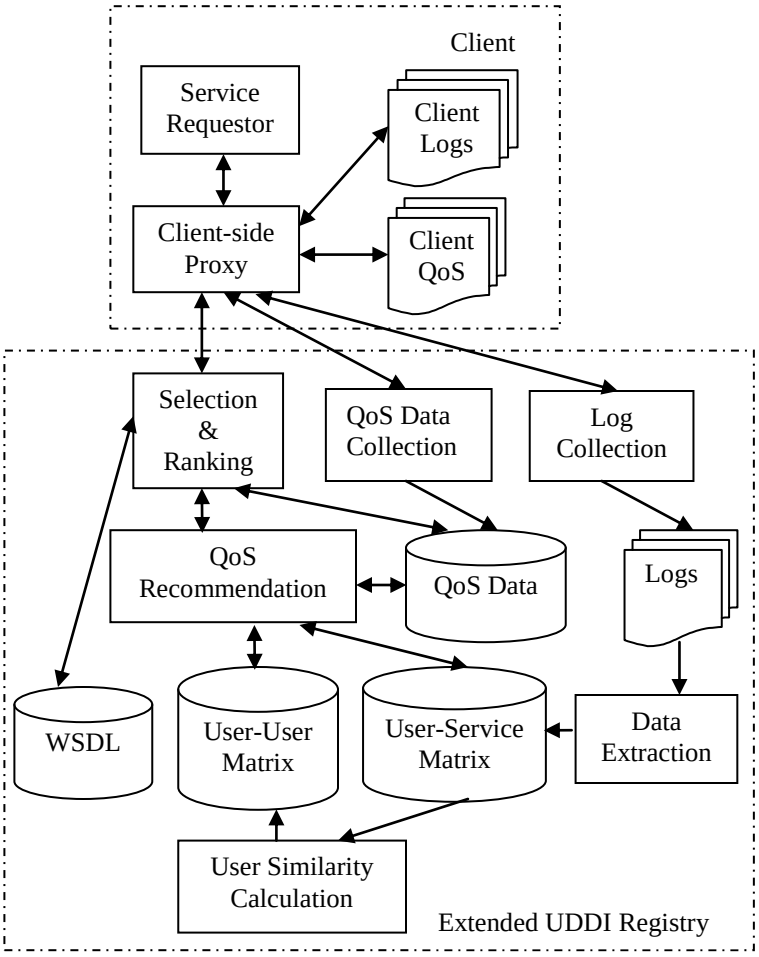


Figure 1. Architecture of our service selection system

The three major data repositories are: service description data, user-service matrix (U-S), and user-user matrix (U-U). Service description data includes both functional and non-functional (QoS) descriptions of the services. A typical example of the functional description is the WSDL file of a web service. The QoS description data can be collected from all the connecting clients and then processed and saved into the central repository. The other two data repositories are created by our system. The user-service matrix is extracted from log files. This matrix contains users' query and invocation records such as their functional and non-functional query requirements, as well as the invocation information such as the invocation time. Each item in the

matrix is represented by a vector and its representation will be explained in details in later sections. The user-user matrix is created through computing user similarity based on query and invocation history information provided by user-service matrix. Details about these calculation algorithms are in later sections.

The five major components of our system are: log data collection, QoS data collection, user similarity calculation, service selection and ranking, and QoS recommendation component.

There are standard formats for web server logs or search engine logs. However, no standard has been defined about how and where to record the service requests and invocation information. In our proposed system, the query log contains the information on the complete query (both functional and QoS parts), the user who submitted the query, the query submission time, and the related invocation (i.e. which service being invoked) if there is any. The invocation history records the information of the user, the invoked service, the invocation time, and the associated query if there is one.

The log collection mechanism can be similar to [7]. A proxy component can be placed in the client machine. The proxy is responsible for sending queries to the extended UDDI registry and redirecting invocation requests to the provider. It also records all the requests and invocation information as well as the QoS data for the client. The collected data can be stored in the local logs and QoS file. Then the log collector and QoS data collector in our system will pull the data from each proxy log periodically and aggregate all of the data into the centralized invocation and query logs as well as the QoS data repository. Before storing the QoS data, the QoS data collected from all the clients are processed to get the overall value.

The similarity calculation is done offline and on a regular basis. Some data pre-processing should be done on the two logs and the service description data, and then the useful

information will be extracted and saved into the user-service matrix. Based on the user-service matrix, user similarity will be calculated and saved into the user-user matrix to improve the efficiency in finding the similar users at run time. These two matrices are computed and updated offline periodically.

After the two matrices are generated or updated, the system is ready to accept the service request from the user. Upon receiving the service query, the service selection and ranking component will process the query, discover the matching services and rank them, and finally present the results to the user. During the ranking process, it will consult the service recommendation component to get the collaborative filtering based ranking, and then it will aggregate QoS-based ranking and collaborative-based ranking using an aggregation algorithm such as Borda Fuse [28]. To produce a collaborative-based ranking, service recommendation component needs the data from the two matrices. Figure 2 briefly illustrates the roles of these three components: similarity computation, service recommendation, and service selection and ranking, in our service selection process. The offline part is inside the shadowed box in figure 2.

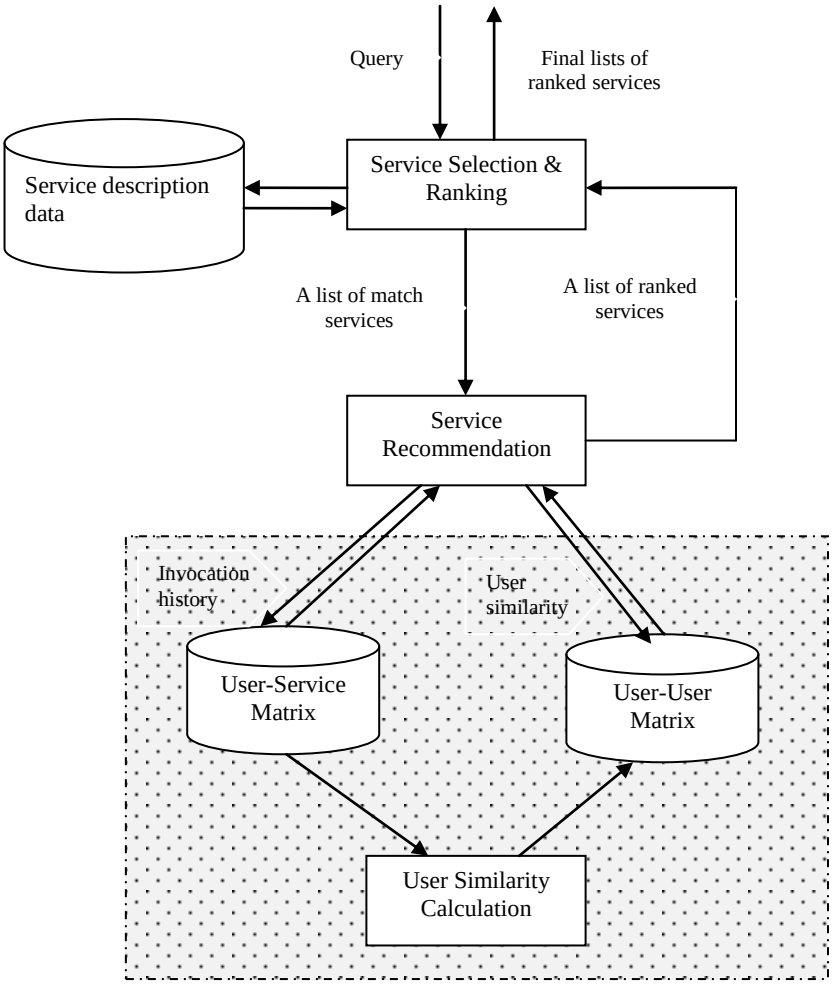


Figure 2. Roles of three components in service selection

3.2 Finding Similar Users

Finding similar users is the key part in a collaborative recommender system. In our system, similar users are computed offline periodically based on User-Service matrix and stored in the User-User matrix where each cell in the matrix contains a similarity value of the correspondent pair of users.

3.2.1 Generating the User-Service Matrix

The user-service matrix is to record the invocation history for all users. Its values could be extracted from query logs and invocation logs. Suppose there are *m* users and *n* services, the set of users could be represented as $U = \{u_1, u_2 \ldots u_m\}$, and the set of services could be represented as $S = \{s_1, s_2 \ldots s_n\}$. For each user $u_i \in U$, the invocation history on service $s_j \in S$ is represented as $IH_{ij} = ((q_{ij}^1, t_{ij}^1), (q_{ij}^2, t_{ij}^2), \ldots, (q_{ij}^{l_{ij}}, t_{ij}^{l_{ij}}))$, where l_{ij} is the number of invocations on s_j from u_i, q_{ij}^k and t_{ij}^k represent the associated query and the invocation time of the k^{th} ($0 \le k \le l_{ij}$) invocation instance on s_j from u_i. The invocation history could be empty if the user hasn't invoked any services yet. In order to save the space and avoid that one user's invocation data may influence too much on the final result, especially when this user has invoked the service many times, we could set up a cut-off value about how many invocations we want to keep so that only the latest *N* invocations will be saved, with *N* defined as a small number. Each q_{ij}^k is either Ø, which means there is no associated query with the invocation, or represented as (fq_{ij}^k, qq_{ij}^k), where fq_{ij}^k and qq_{ij}^k refer to the functional and non-functional (QoS) part of the query.

The functional part of query could be represented as a term vector. The QoS part of query qq_{ij}^kcontains requirements on multiple QoS or non-functional attributes such as response time, reliability, cost, etc. Each attribute has a value requirement and a weight which is determined by the user preference. We define $qq_{ij}^k = ((qq_{ij1}^k.w, qq_{ij1}^k.v), (qq_{ij2}^k.w, qq_{ij2}^k.v), \ldots, (qq_{ijp}^k.w, qq_{ijp}^k.v))$, where p is the number of QoS attributes supported by the system, $qq_{ijh}^k.w$ and $qq_{ijh}^k.v$ represent the weight and value requirement on the h^{th} ($1 \le h \le p$) attribute for the kth invocation instance. The value of $qq_{ijh}^k.w$ would be from 1 to $p+1$ depending on the order of preference, with a lower number inferring a more preferred attribute, and if the user is not

interested on this attribute, the value would be *p+1*. Since the value requirement could have different data types depending on the QoS attribute [29], we generalize $qq_{ijh}^{k}.v$ as a set. For instance, if the requirement is "reliability > 95%", which is an interval data, $qq_{ijh}^{k}.v$ would be $\{x \mid x \epsilon R, 0.95<x<1\}$, or if the requirement is "authentication: yes", which is a single Boolean value, $qq_{ijh}^{k}.v$ would be {1}.

3.2.2 Generating the User-User Matrix

After the user-service matrix is generated from the logs, the next step is to calculate the user similarity and save it into the User-user matrix. In a classical collaborative filtering recommender system such as MovieLens [2], users need to give a rating to each item. However it is known that explicit feedback information is hard to elicit from users and sometimes the accuracy of the ratings also cannot be guaranteed if there is no proper trust management mechanism being enforced. To overcome this, various implicit feedback measurements have been proposed, including the click-through data from the search engine, the duration of time a user spends on a web page and some follow-up actions such as printing or saving or bookmarking, the actual transaction record of purchasing a product, etc. Although the user opinion/interest can only be inferred from these observable behaviors and the inference may not be precise sometimes, it is generally believed that the uncertainty could be handled when the amount of feedback data is adequate.

In the web service domain, some recommender systems require users to provide ratings after they invoke the services [6], some use invocation rate as the preference indicator [12], and some others just use the monitored QoS values to predict the future values [4]. In our system, we want to use the recommender system in the QoS-based ranking step. Therefore, the user

similarity should be based on their QoS requirements and the subsequent decision-making on selecting and invoking services. The main idea is that: if two users select and invoke the same service from a list of functionally similar services, it indicates that they may have similar QoS requirements or considerations so that they make the same decision of choosing this service; if their similarity on QoS requirements can be confirmed from their actual queries, the similarity level should be higher. When two users have more commonly invoked services, the similarity level should be higher. Also we prefer the more recent invocations than the earlier ones because they could reflect users' current behaviors more accurately. The detailed similarity calculation algorithm is explained below.

Let IS_i be a set of services that u_i invoked and IS_j be a set of services that u_j invoked. The set of commonly invoked services by two users is defined as $CS_{ij} = IS_i \cap IS_j$. For each service $s_k \epsilon CS_{ij}$, the invocation history of u_i and u_j would be represented as IH_{ik} and IH_{jk} respectively. Since we focus on the QoS query part for each service invocation and each QoS query includes a weight vector and a value vector for all attributes, the overall query similarity will be determined by both weight and value similarities. The weight similarity is to measure whether u_i and u_j have similar preferences on QoS attributes in an invocation of s_k, and the value similarity is to measure whether these two users have similar requirements on QoS value ranges in this invocation. The weight vector of the h^{th} invocation from user u_i on service s_k could be represented as $\overrightarrow{qw}_{ikh} = (qq^h_{ik1}.w, qq^h_{ik2}.w, \ldots, qq^h_{ikp}.w)$, and the value vector can be represented as $\overrightarrow{qv}_{ikh} = (qq^h_{ik1}.v, qq^h_{ik2}.v, \ldots, qq^h_{ikp}.v)$. We calculate the query similarity between u_i and u_j on service s_k as follows,

$if\ l'_{ik} = 0\ or\ l'_{jk} = 0,\ sim_{ijk} = 0$ otherwise,

$$sim_{ijk} = \frac{1}{l_{ik}^{'} * l_{jk}^{'}} \sum_{h_1=1}^{l_{ik}^{'}} \sum_{h_2=1}^{l_{jk}^{'}} (SV(\overrightarrow{qv}_{ikh_1}, \overrightarrow{qv}_{jkh_2}) * (SW(\overrightarrow{qw}_{ikh_1}, \overrightarrow{qw}_{jkh_2}) + c_1)) \quad (1)$$

where $l_{ik}^{'}$ is the number of invocations on s_k from u_i with non-empty QoS queries, $l_{jk}^{'}$ is the number of invocations on s_k from u_j with non-empty QoS queries, $SV(.)$ is a function to calculate the similarity between the two value vectors, $SW(.)$ is a function to calculate the similarity between the two weight vectors, and c_1 is a small constant value which is set to 0.1 in the current implementation.

The weight vector saves user's preferences on p QoS attributes, and it could be considered as a ranked list. So instead of the commonly used Pearson Correlation Coefficient (PCC) or Cosine Similarity [1], for the similarity calculation, we use the Kendall tau coefficient [16] which could measure the agreement between the two ranked lists. Suppose we have two ranked lists ($x_1, x_2, \ldots, x_n$) and ($y_1, y_2, \ldots, y_n$), given any pair on i^{th} and j^{th} position, if $x_i < x_j$ and $y_i < y_j$, or $x_i > x_j$ and $y_i > y_j$, we say they are concordant, otherwise, they are discordant. The Kendall tau coefficient is defined as:

$$\Gamma_A = \frac{n_c - n_d}{\frac{1}{2} * n * (n-1)} \quad (2)$$

where n_c is the number of pairs which are concordant, n_d is the number of discordant pairs, and n is the number of attributes which at least one user is interested in. Here, $\frac{1}{2} * n * (n-1)$ is the total number of ordered pairs that an ordered set of n objects can compose. Since sometimes a user may have the same preference on different QoS attributes, we modify this definition a little bit to allow for the non-strict ordering. So for any pair, if $x_i \leq x_j$ and $y_i \leq y_j$, or $x_i \geq x_j$ and $y_i \geq y_j$, we say that they are concordant, otherwise, they are discordant. The result is in the (-1, 1) range with 1 referring to the perfect agreement and -1 referring to the perfect disagreement. We want the similarity to be a value between 0 and 1, and the value 0 means two preference orders are

completely different and the value 1 means they are the same. So we convert the above formula to the new one as shown below:

$$\Gamma_A{}' = \frac{n_c}{\frac{1}{2} * n * (n-1)} \quad (3)$$

Also the more the common attributes, the higher the similarity value. With two weight vectors $\overrightarrow{qw}_{ikh_1}$ and $\overrightarrow{qw}_{jkh_2}$, we first need to find out all the attributes with weight values not equal to *p+1*, which means users are interested on them. Let na_c be the number of common attributes both users are interested in and na_u be the number of attributes at least one user is interested in, the final weight similarity is calculated only on the attributes at least one user is interested in, and the formula is shown below:

$$SW\left(\overrightarrow{qw}_{ikh_1}, \overrightarrow{qw}_{jkh_2}\right) = \frac{na_c}{na_u} * \Gamma_A{}' \quad (4)$$

To calculate the value similarity, since the value requirement on each attribute is represented as a set, we use Jaccard Coefficient [17] to do the calculation, which is more appropriate to measure the similarity between two sets,

$$SV\left(\overrightarrow{qv}_{ikh_1}, \overrightarrow{qv}_{jkh_2}\right) = \frac{1}{p}\sum_{h=1}^{p} \frac{\left|qq_{ikh}^{h_1} \cap qq_{jkh}^{h_2}\right|}{\left|qq_{ikh}^{h_1} \cup qq_{jkh}^{h_2}\right|} \quad (5)$$

Finally, the overall similarity sim_{ij} between u_i and u_j can be computed by adding up sim_{ijk} over all commonly invoked services. Also if among all invoked services by two users, there are many common ones, the similarity score will be higher than the case when there are only a few common ones. Its value is zero if there is no common service, otherwise, it is calculated as,

$$sim_{ij} = \frac{2*|CS_{ij}|}{|IS_i|+|IS_j|} * \frac{1}{|CS_{ij}|}\sum_{s_k \epsilon CS_{ij}} (sim_{ijk} + c_2) \quad (6)$$

where c_2 is a constant number to make sure the common invocation is always counted. In the current implementation, c_2 is set to 0.2. The similarity calculation will be run regularly. Depending on the frequency of invocation and query request, we could set up a proper frequency to update the user similarity matrix. Since it is done offline, the efficiency is not a big concern, whereas the accuracy is more important.

3.3 Selection and Ranking Algorithm

After we get the User-User matrix, we can easily find most similar users and rank the matching services for the query.

3.3.1 Collaborative Filtering Ranking Based on Invocation History

In a typical collaborative filtering recommender system, we need to first find out the most similar *K* neighbors to the current user, and then predict the user's opinion (e.g. a rating on a movie) based on those *K* users' previous records. Here we use collaborative filtering to calculate the matching degree of a service to a particular user request. Our ranking algorithm is similar to the click-through data based ranking for the search engines [13]. The main idea is that if a service has been chosen and invoked many times before, it has a higher chance to be selected in the current search session, and if many of the previous invocations come from users who have similar opinions or interests as the current user, it has an even higher chance to be selected.

Suppose a user u_i submits a query including both functional and QoS requirements. Let the query be (Q_f, Q_{qos}), in which Q_f has to be non-empty whereas Q_{qos} might be empty because sometimes a user may not have a particular QoS requirement or may not know how to formulate a QoS requirement properly. Based on the functional requirement Q_f, the discovery agent of the

registry finds all the matching services. Let the result set be RS_f. Our collaborative filtering ranking algorithm mainly works on RS_f. For each service $s_k \in RS_f$, if the service has been invoked by some users, we calculate its collaborative filtering based score as follows:

$$S_{CF}(s_k) = \sum_{u_j \epsilon S(u_i)} sim_{ij} * \frac{1}{N} * \sum_{h=1}^{l_{jk}} \frac{t_{jk}^h - t_s}{t_c - t_s} * (SF(fq_{jk}^h, Q_f) + c_3) \qquad (7)$$

where $S(u_i)$ is a set of users most similar to user u_i and the size of this set is K, and N is the maximum number of invocations on a service we would save for one user. The value sim_{ij} could be obtained from the user similarity matrix. For each of the invocations on service s_k from user u_j, t_{jk}^h represents the time of the h^{th} invocation, t_s represents the starting time of the log, t_c represents the current system time, and $SF(fq_{jk}^h, Q_f)$ is to calculate the similarity between the current query and the functional query associated with this particular invocation. For the similarity function $SF(.)$, we choose the commonly used Cosine Similarity formula [30]. When fq_{jk}^h is $\emptyset$, the query similarity value would be zero, which means that we are not sure whether the invocation is resulted from the same functional query. Despite of this uncertainty, we still count this invocation by adding a constant value c_3 (i.e. 0.1) to the query similarity value because of the concern on the possible sparse user-service matrix. This calculation considers the invocations from the top K similar users. Based on the above formula, if the user similarity is higher, the score will be higher; if the invocation time is more recent, the score will be higher; if the invocation is due to similar queries, the score will be higher; if there are more invocations of this service, the score will be higher. Figure 3 shows how the collaborative filtering based score is calculated based on those data. In the figure, RS_{fo} includes the services of the functionally

matching services returned from the service description data repository, excluding those services that have never been invoked by any user.

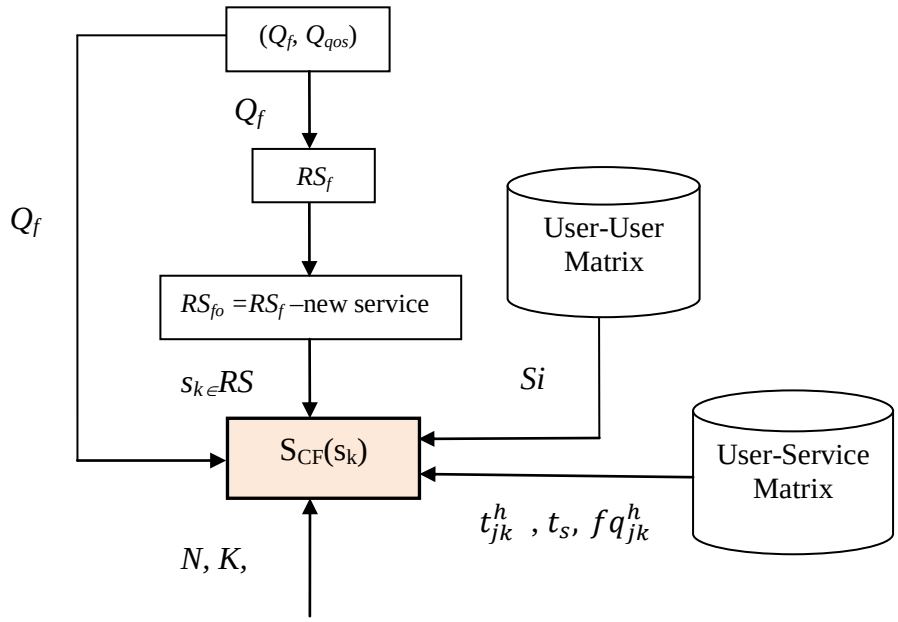


Figure 3. Calculation of collaborative filtering based score

To overcome the cold start problem for new users, we propose that in this case we will calculate the ranking score based on the invocation history only. If IH_{ik} is empty for all k (k=1, … , n), which means user u_i is a new user and hasn't invoked any services yet, or if sim_{ij} is zero for all j (j=1, … , m, and $j \neq i$), which means user u_i doesn't have any similar users, then for each service $s_k \in RS_f$, we calculate its score using a different formula:

$$S_{CF}(s_k) = \frac{1}{N} * \sum_{j=1}^{m} \sum_{h=1}^{l_{jk}} \frac{t_{jk}^h - t_s}{t_c - t_s} * (SF(fq_{jk}^h, Q_f) + c_3) \qquad (8)$$

To overcome the cold start problem for new services, we propose to present the newly published services in a separate list, which is simply ranked on their publication date and invocation frequencies, so that the new services will have an equal chance to be viewed and selected by users.

3.3.2 Overall Selection and Ranking Algorithm

Having explained our invocation history based ranking algorithm, the complete service selection process can be described in the following steps:

(1) Given a query (Q_f, Q_{qos}) from user u_i, RS_f contains all the functionally matching services. For all the services in RS_f, we will rank them based on Q_{qos} and the service invocation history.

(2) If Q_{qos} is empty, go to step 3, otherwise, go to step 4.

(3) When there is no QoS requirement from the user, the ranking of the services is purely decided by the invocation history. As explained in the previous section, the score $S_{CF}(s_k)$ is calculated differently for new users and existing users. The final result presented to the user will be two separate ranked list, one is a list of new services, RS_n, published in the past T (a predefined threshold) days, which is ranked on their publication date, and the other is the rest of the services, ranked on their calculated scores S_{CF}.

(4) When there are QoS requirements, we first filter out services which couldn't satisfy the QoS hard constraints. The remaining services will be in RS_{hq}, which is a subset of RS_f. Then for all services in RS_{hq}, we calculate their QoS-based ranking scores using an approach such as the one described in [18], [31], [32], etc. Afterwards, we calculate their invocation history based ranking scores using formula (7) and (8). The final result will also

be two ranked list, one is a list of new services, RS_n, ranked on their QoS-based ranking scores, and the other is the rest of the services. For latter services, they have both QoS-based and collaborative filtering based scores. In order to achieve a single ranked list, we could use any rank aggregation methods such as Borda Fuse [28] to combine the two.

Figure 4 shows the overall selection and ranking process. Our selection system is rather generic, and we could plug in any algorithms for the QoS score calculation and rank aggregation.

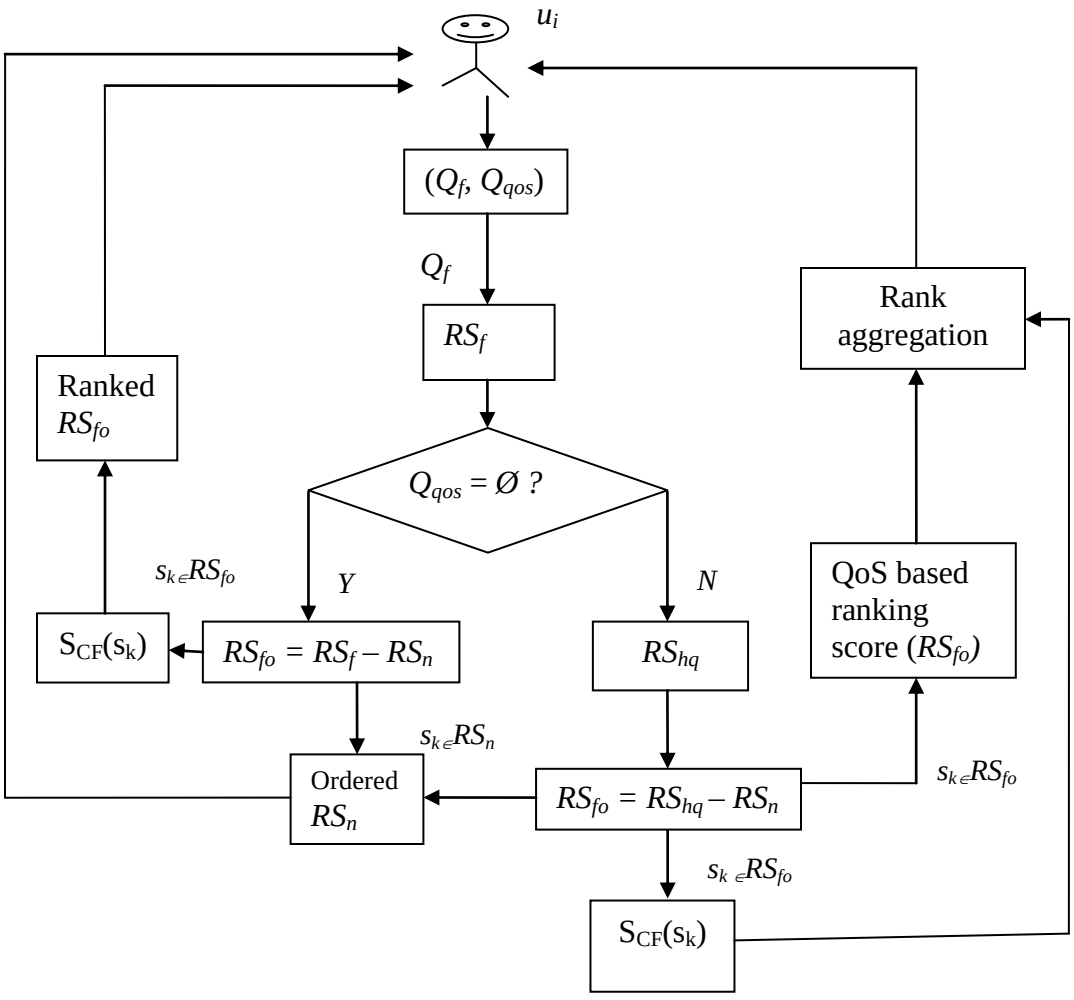


Figure 4. Overall selection and ranking process

3.4 Efficiency of Our Algorithms

In this section, we will evaluate the complexity of our ranking and user similarity algorithms. The memory used by our algorithms mainly depends on the number of users or services in our system, and it can be a linear dependency. Here we only analyze the time complexity. Since different implementation approaches can be adopted and the time complexity analysis only considers commands with their numbers of executions vary with the problem size, we evaluate the complexity using the pseudo codes based on the major loops that are related to the following parameters: number of users and number of services. We use the same notations as the previous section: *m* to be the number of users and *n* to be the number of services.

3.4.1. Ranking Algorithm

We list the ranking algorithm as Algorithm 1 and the algorithm computing the frequency as Algorithm 2. For each major step of an algorithm, e.g., step *i*, we label it as [i], for the convenience of the later reference in our discussion.

Algorithm 1. Pseudo code for the ranking algorithm

```
double rank (…) {
        …
        find top k similar users;                    [1]
        …
        for each top similar user                    [2]
        {
                Call the freq method to get the invocation frequency and time value;
                Computing the ranking score for the service s based on formula (7);
        }
        return the ranking score;
}
```

Algorithm 2. Pseudo code for computing the invocation frequency

```
double freq(…) {
        …
        while (numInvocation < N)
        {
                if (the instance was invoked with same query)
-                       f += (invocationTime - startTime)/(currentTime – startTime) *
                        (1+c3);
                else
                        f += (invocationTime - startTime)/(currentTime – startTime) * (c3);
                numInvocation++;
        }
        return f;
}
```

Algorithm 2 shows that the number of executions depends on the number of invocation instances used in our algorithm, which is defined as *N*. In each iteration, the number of executions is 4 for the calculation of the *f* value. There are a few other commands in the loop. Therefore, the number of executions for the loop is: *(4 + t)*N*, where *t* represents the number of executions of other commands in the loop. The total number of executions for Algorithm 2 is $q = (4+t)*N + z$, where *z* represents the number of executions before the loop, which is a constant value.

For the first step of Algorithm 1, since it involves sorting the user similarity list and then choosing the top *K* users, the time complexity is $O(m*log(m))$. For the second step, the algorithm loops *K* times, and each time it calculates *f* value and the ranking score. The ranking score is calculated based on formula (7) which takes a small number of executions, i.e., *r*. Therefore the number of executions for the loop is in the order of $K*(r+q)$. The time complexity for the ranking algorithm is $O(m*log(m)) + O(K*(r+q))$. Since *K* is a pre-defined integer and *r* are small

integers and q depends on N, t and z which are fixed numbers, we get the complexity of our ranking algorithm as $O(m*log(m))$.

3.4.2 Similarity Algorithm

In the following boxes, Algorithm 3 lists the pseudo code for computing the user similarity for every pair of users (*U*-*U* matrix) and saving the results in a file. Algorithm 4 is for computing the similarity between two users on each commonly invoked service and returning the results to the calling method – Algorithm 3. Algorithm 5 computes Kendall tau coefficient for user QoS preference similarity and returns the result to the calling method – Algorithm 4. Algorithm 6 computes the concordant value for Algorithm 5. Algorithm 7 is the pseudo code for computing the user value requirement similarity which is a Jarccard coefficient and returning the result to the calling method – Algorithm 4.

Algorithm 3. Pseudo code for user similarity computing

```
void profiling(…) {
        …
        for (int i = 0; i < numUser; i++)                                   [1]
        {
                for (int j = 0; j < numUser; j++)
                {
                        …
                        Get history data for user i;                        [2]
                        Get history data for user j;                        [3]
                        if (history data exist for both user i and j)
                        {
                                Get services user i invoked;                [4]
                                Get services user j invoked;                [5]
                                Get commonly invoked services of user i and j;  [6]
                                Call similarity computing between user i and j for each
                                commonly invoked service and get the result in an array; [7]
                        }
                        Computing user similarity using the similarity array based on
                        formula (6);                                        [8]

                }
                Write result to file;                                       [9]
                …
        }
        …
}
```

Algorithm 4. Pseudo code for computing the similarity between user *i* and *j* on each commonly invoked service

```
Void qWVSim(…) {
        …
        if (user i and user j have have history data)                    [1]
        {
                …
                if (user i and j have commonly invoked services)         [2]
                {
                        for (each service in the set of commonly invoked services)   [3]
                        {
                                …
                                Computing similarity between user i and j for the service
                                based on foumula (1) and store the result in an array for the
                                calling method;
                                …
                        }
                }
        }
…
}
```

Algorithm 5. Pseudo code for computing the weight similarity

```
double simWins_Kendall(…) {
        …
        Get the commonly interested QoS attribute for user i and j on service k;        [1]
        Get the union of the interested QoS attribute for user i and j on service k;    [2]
        Call concordant method to get the concordant value;                             [3]
        Computing the similarity between user i and j on service k using formula (4);   [4]
        …
        Return tau;
}
```

Algorithm 6. Pseudo code for computing the concordant

```
int concordant(…) {
        …
        Make the attributes lists either user i or user j interested for service k
        in order;                                                          [1]
        Compare the two ordered list to get the concordant value;          [2]
        Return concordant value;
}
```

The number of executions for the first step of Algorithm 6 depends on the number of attributes under consideration, which is usually a small pre-defined integer, i.e., *h*. Sorting two lists of this number of integers takes the time in the order of *2h*log(h)*. In the second step of the Algorithm, comparing these two lists using nested loops through *h* members takes *h*h* times. Therefore, the total amount of executions is: $x_1 = 2h*log(h) + h*h$. It is a constant value when the number of attributes of interest is pre-defined, which is the case in our system.

The computation for step 1 and 2 of Algorithm 5 involves calculating the intersection and the union of the sets of attributes users *i* and *j* are interested in for service *k*. The number of attributes is denoted as *h* as mentioned above. Assume we use multiple search method, the execution time is less than or equal to *2h*h*. Step 4, calculating the *tau* value based on formula (4) needs a small constant number of steps, i.e., *g*. So the total number of executions is in the order of: $x_2 = 2h*h + x_1 + g$. Since *h* is a pre-defined integer number and x_1 is a constant number as mentioned above, x_2 is a constant number.

Algorithm 7. Pseudo code for computing the value similarity

```
Double simVIns_Union(...) {
        ...
        Get the commonly interested attributes
        ...
        for each interested attribute
        {
                ...
                Compute the union of the range of the values
                Compute the intersection of the range of the values
                Compute the Jaccard coefficient based on formula (5)
        }
        Return the coefficient value;
}
```

The complexity level of Algorithm 7 is similar to that of Algorithm 5, which depends on the number of attributes users *i* and *j* are interested in for the invocation of the service. We are not going to explain this method in details. We use x_3 to represent the execution time complexity, which is also a constant value.

Algorithm 4 computes the user similarity on each service and returns a list of similarity values for all services users *i* and *j* commonly invoked. In Algorithm 4, step 3 is the major one that contributes the most to the final complexity. Step 3 loops through all commonly invoked services and computes the similarity values based on each service using formula (1). Each iteration computes the similarity on each commonly invoked service for user *i* and *j*. The number of executions is $N'*N'*x_2*x_3$ where N' is the number of invocations we keep for each service of each user in our similarity computation, which is usually a pre-defined constant number. The overall execution time is in the order of $S*N'*N'*x_2*x_3$, where S is the number of commonly invoked services. In the worst case scenario when $S = n$, the complexity is $O(n)$.

Algorithm 3 gets the final user similarity values for each pair of users and stores the results in a user-user matrix. The main loop going through all users needs $m*m$ iterations. Steps 2, 3, 4 and 5 take fixed steps of operation. We can denote them as y_1 for step 2, 3, and y_2 for step 4, 5. Step 6 computes the union of the sets of services users i and j invoked. In the worst case scenario when users have invoked all of the services in the system, it takes $n*n = n^2$ times of execution. Based on our previous analyses, the complexity of step 7 is $O(n)$. Step 8 involves iteration through the similarity array, which is the size of the commonly invoked services. In the worst case, it is in the order of $O(n)$. Step 9, writing the result to the file, takes $O(m)$ time, because it writes the array of similarity values between current user and all the other users. Hence, the overall complexity level is: $O(m*(m*(2*y_1+2*y_2+n^2+n+n)+m)) = O(m^2n^2)$.

Based on above analyses, we can see that the complexity of the ranking algorithm is $O(m*log(m))$, whereas the complexity of the user similarity computation is $O(m^2n^2)$. Both complexity levels are practical for the system to work efficiently.

3.5 Case Studies Illustrating the Ranking Algorithm

In this section, we use case scenarios to illustrate the ranking algorithm. Suppose there are seven users who want to find hotel reservation services. They are using our system hoping to get better recommendations. Our system has collected some information about the previous queries and invocations from them. The three QoS attributes we are considering are response time, rating, and cost. The rating is defined as a scale from 1 to 5, with 5 referring to the best rating, and cost is defined as the subscription fee per year of usage. There are five services, s_1, s_2, s_3, s_4, s_5 and their QoS parameters are shown in Table 1.

Table 1. QoS parameters of 5 hotel reservation services

	s_1	s_2	s_3	s_4	s_5
Response time	1.5s	2s	1s	0.8s	0.8s
Rating	3	2	3	5	5
Cost	$100/year	$80/year	$110/year	$175/year	$220/year

After the data extraction step based on the two log files, the QoS part of the user-service matrix is shown in Table 2.

Table 2. A sample user-service matrix

	Name	s_1	s_2	s_3	s_4	s_5
u_1	Alice	(((3,1s), (2, 3), (1, $100)), 03/09), (Ø, 07/09)	(((2,1.5), (3, 3), (1, $90)), 08/09), (Ø, 09/09) (Ø, 12/09)	(((3,1s), (1, 3), (2, $110)), 12/09),		
u_2	Bob	(((3,2s), (1, 3), (2, $110)), 05/09)		(((3,1s), (2, 3), (1, $130)), 12/09), (Ø, 02/10)		
u_3	Cindy	(((3,1s), (1, 3), (2, $120)), 03/09), (Ø, 08/09)		(((2,1s), (1, 3), (3, $120)), 10/09), (Ø, 11/09) (Ø, 12/09) (Ø, 01/10)		
u_4	David			(((2,1s), (1, 3), (3, $150)), 05/09), (Ø, 08/09)	(((2,1s), (1, 5), (4, Ø)), 06/09), (Ø, 07/09)	(((2,1s), (1, 5), (4, Ø)), 01/10),
u_5	Emma	(((2, 1.2s), (3, 3), (1, $100)), 2/09), (Ø, 4/09),	(((2,2s), (3, 3), (1, $85)), 06/09), (Ø, 10/09) (Ø, 03/10)			
u_6	Flora	(Ø, 5/09)		(((3,1s), (1, 3), (2, $150)), 10/09), (Ø, 10/09)	(((2,1s), (1, 5), (3, $200)), 12/09), (Ø, 12/09) (Ø, 04/10)	
u_7	Henry					

In the table, we simplify the QoS value requirement to a single value, which will be converted to a set later after processing the data. For instance, if an invocation record is (((3,1s), (2, 3), (1, $100)), 03/09), it means that in the associated query, the preferences on the 3 attributes

are 3, 2 and 1 respectively, and the value requirements are $\{x \mid x \in R,\ 0<x\leq 1\}$, $\{x \mid x \in Z,\ 3\leq x\leq 5\}$, and $\{x \mid x \in R,\ 0<x\leq 100\}$ respectively, and the invocation time is March 2009. The time unit in this example is month. In the actual implementation, it can be more fine-grained. The empty cell indicates that the user has never invoked the corresponding service and Ø indicates that the user invokes the service without a query. u_i represents the i^{th} user.

The weight similarity is calculated using formula 3 and 4. For instance, both u_1 and u_2 invoked s_1 and s_3. The weight similarity between u_1 and u_2's first invocations on s_1 would be:

$$SW(\overrightarrow{qw}_{111}, \overrightarrow{qw}_{211}) = SW([3,2,1],[3,1,2]) = \frac{3}{3} * \frac{2}{\frac{1}{2} * 3 * 2} = 0.67$$

The weight similarity between u_1 and u_2's first invocations on s_3 would be:

$$SW(\overrightarrow{qw}_{131}, \overrightarrow{qw}_{231}) = SW([3,1,2],[3,2,1]) = \frac{3}{3} * \frac{2}{\frac{1}{2} * 3 * 2} = 0.67$$

The value similarity is calculated using formula 5. The value similarity between u_1 and u_2's first invocations on s_1 would be:

$$SV(\overrightarrow{qv}_{111}, \overrightarrow{qv}_{211}) = SV([1,3,100],[1,3,110])$$

$$= \frac{1}{3} * \left(\frac{1}{1} + \frac{6-3}{6-3} + \frac{100}{110}\right) \quad = 0.97$$

The value similarity between u_1 and u_2's first invocations on s_3 would be:

$$SV(\overrightarrow{qv}_{131}, \overrightarrow{qv}_{231}) = SV([1,3,100],[3,3,130])$$

$$= \frac{1}{3} * \left(\frac{1}{1} + \frac{6-3}{6-3} + \frac{110}{130}\right) \quad = 0.95$$

To calculate the similarity between u_1 and u_2 on s_1, s_3 we can use formula 1:

$$sim_{121} = \frac{1}{1*1} * (0.97 * (0.67 + 0.1)) = 0.746$$

$$sim_{123} = \frac{1}{1*1} * (0.95 * (0.67 + 0.1)) = 0.732$$

We can see that the similarity between u_1 and u_2 on s_3 is slightly lower than on s_1, because the similarity on QoS value requirements is slightly lower. And finally the similarity between u_1 and u_2 would be calculated with formula 6:

$$sim_{12} = \frac{2*2}{2+3} * \frac{1}{2} * (sim_{121} + 0.2 + sim_{123} + 0.2) = 0.4 * (0.746 + 0.732 + 0.4)$$

$$= 0.751$$

Similarly, both u_1 and u_3 invoked s_1 and s_3. The weight similarity between u_1 and u_3's first invocations on s_1 would be:

$$SW(\overrightarrow{qw}_{111}, \overrightarrow{qw}_{311}) = SW([3,2,1],[3,1,2]) = \frac{3}{3} * \frac{2}{\frac{1}{2}*3*2} = 0.67$$

The weight similarity between u_1 and u_3's first invocations on s_3 would be:

$$SW(\overrightarrow{qw}_{131}, \overrightarrow{qw}_{331}) = SW([3,1,2],[2,1,3]) = \frac{3}{3} * \frac{0}{\frac{1}{2}*3*2} = 0$$

The value similarity is calculated using formula 5. The value similarity between u_1 and u_3's first invocations on s_1 would be:

$$SV(\overrightarrow{qv}_{111}, \overrightarrow{qv}_{311}) = SV([1,3,100],[1,3,120])$$

$$= \frac{1}{3} * \left(\frac{1}{1} + \frac{6-3}{6-3} + \frac{100}{120}\right) \quad = 0.94$$

The value similarity between u_1 and u_3's first invocations on s_3 would be:

$$SV(\overrightarrow{qv}_{131}, \overrightarrow{qv}_{331}) = SV([1,3,110],[3,3,120])$$

$$= \frac{1}{3} * \left(\frac{1}{1} + \frac{6-3}{6-3} + \frac{110}{120}\right) \quad = 0.97$$

To calculate the similarity between u_1 and u_3 on s_1, s_3 we use formula 1:

$$sim_{131} = \frac{1}{1*1} * (0.94 * (0.67 + 0.1)) = 0.724$$

$$sim_{133} = \frac{1}{1*1} * (0.97 * (0 + 0.1)) = 0.097$$

We can see that the similarity between u_1 and u_3 on s_3 is much lower than on s_1, because the similarity on QoS weight is 0, but it still contributes to the similarity value on s_3 because a constant value 0.1 is added. And finally the similarity between u_1 and u_3 is calculated using formula 6:

$$sim_{13} = \frac{2*2}{2+3} * \frac{1}{2} * (sim_{131} + 0.2 + sim_{133} + 0.2) = 0.4 * (0.724 + 0.097 + 0.4)$$

$$= 0.488$$

We calculate all the similarity values between each pair of users. The final User-User matrix for all users is shown in Table 3:

Table 3. User-User matrix

	u_1	u_2	u_3	u_4	u_5	u_6	u_7
u_1	1	0.751	0.488	0.197	0.846	0.48	0
u_2	0.751	1	0.854	0.244	0.28	0.47	0
u_3	0.488	0.854	1	0.489	0.292	0.147	0
u_4	0.197	0.244	0.489	1	0	0.406	0
u_5	0.846	0.28	0.292	0	1	0	0
u_6	0.48	0.47	0.147	0.406	0	1	0
u_7	0	0	0	0	0	0	1

Based on above User-User similarity matrix, we use the following case studies to explain our ranking algorithm.

Case I:

Suppose u_2 (Bob) submits a query to find a hotel reservation service with cost <= \$120, after doing the matchmaking, the result set RS_f is $\{s_1, s_2, s_3\}$. Suppose the value of t_s is 01/09 and t_c is 06/10. To simplify the calculation, we assume that *SF(.)* value would be 1, which means that they all have the same query. Suppose K = 3, based on the user similarity matrix in Table 3, we

know u_3, u_1, and u_6 are the top three neighbors of u_2 with similarity value of 0.854, 0.751, 0.47 respestively. However, u_2 has invoked s_1 too and its own experience should be the most significant factor in the ranking. So, we compute the similarity based on u_2, u_3, and u_1 for the rank of s_1. Based on formula 7, the collaborative filtering based ranking scores for s_1, s_2, s_3 are:

$$S_{CF}(s_1) = 0.854 * \frac{1}{5} * \left(\frac{3-1}{12+6-1} + \frac{8-1}{17}\right) * (1+0.1)$$

$$+0.751 * \frac{1}{5} * \left(\frac{3-1}{17} + \frac{7-1}{17}\right) * (1+0.1) + 1 * \frac{1}{5} * \frac{5-1}{17} * (1+0.1) = 0.228$$

$S_{CF}(s_2)$ = 0.272

$S_{CF}(s_3)$ = 0.88

So the ranking order of these three services would be: s_3, s_2, s_1.

Case II:

Suppose u_2 (Bob) submits a query to find a hotel reservation service with only functional part, after doing the matchmaking, the result set RS_f is $\{s_1, s_2, s_3, s_4, s_5\}$. The value of t_s, t_c and *SF(.)* are assumed the same as in Case I. Based on the user similarity matrix in Table 3 and formula 7, the collaborative filtering based ranking scores are:

$S_{CF}(s_1) = 0.228$

$S_{CF}(s_2)$ = 0.272

$S_{CF}(s_3)$ = 0.88

$S_{CF}(s_4)$ = 0.158

$S_{CF}(s_5)$ = 0

The ranking order of the services is s_3, s_2, s_1, s_4, s_5. We could see from this case, the recommendation for top three services doesn't change. The system could implicitly capture the user's preference from his previous experiences.

Case III:

Suppose u_7 (Henry) submits a query to find a hotel reservation service with only functional part, after doing the matchmaking, the result set RS_f is $\{s_1, s_2, s_3, s_4, s_5\}$. The value of t_s, t_c and *SF(.)* are assumed the same as in Case I. Since Henry doesn't have any similar users, based on formula 8, the collaborative filtering based ranking scores are:

$$S_{CF}(s_1) = \frac{1}{5} * \left(\frac{3-1}{12+6-1} + \frac{7-1}{17}\right) * (1 + 0.1) + \frac{1}{5} * \left(\frac{5-1}{17}\right) * (1 + 0.1) +$$

$$\frac{1}{5} * \left(\frac{3-1}{17} + \frac{8-1}{17}\right) * (1 + 0.1) + \frac{1}{5} * \left(\frac{2-1}{17} + \frac{4-1}{17}\right) * (1 + 0.1) + \frac{1}{5} * \left(\frac{5-1}{17}\right) * (1 + 0.1)$$

$= 0.376$

$S_{CF}(s_2) = 0.763$

$S_{CF}(s_3) = 2.66$

$S_{CF}(s_4) = 0.466$

$S_{CF}(s_5) = 0.155$

For new user Henry, the ranking order of the services would be s_3, s_2, s_4, s_1, s_5. Although as we can see from user-service matrix, s_1 has more invocations than s_2 and s_4, the invocations of s_2 and s_4 are more recent than s_1. As a result, the rank scores of s_2 and s_4 are higher than that of s_1. If Henry submits a query to find a hotel reservation service with cost <= \$120, the result set RS_f is $\{s_1, s_2, s_3\}$ and the order is s_3, s_2, s_1.

3.6 Summary

In this chapter, we have explained the system architecture, the history data used for selection, the algorithms used in finding similar users, as well as the ranking algorithm. The efficiency of our algorithms has been assessed by the time complexity analysis. Three use cases are used to explain how the computation is done on similarity and ranking. From the use cases, we can see how the user similarity value, invocation frequency, and invocation time can affect the ranking score. In next chapter, we will do further evaluation on this algorithm.

CHAPTER 4

EXPERIMENTS

The main purpose of our experiments is to evaluate the accuracy of the collaborative filtering based ranking algorithm, and in the mean time test the impact of different parameter settings to the system performance.

4.1 Dataset

Currently there is no standard dataset to evaluate the QoS-based web service selection systems, let alone for collaborative filtering based approaches. Since our algorithm depends on the invocation and query logs to do the ranking, we use a simulation program to generate the service requests and invocation records, and then use the simulated dataset to do the evaluation.

The dataset reported in [33] was from the actual crawling and monitoring of the online web services. Each service in this dataset has a name, URL of its WSDL file, and the values of 9 QoS attributes. There are altogether 2507 services. We used this dataset as our service repository. By checking the name of each service and its WSDL file, we chose 15 most popular keywords as single-word functional queries such as “genome”, “weather”, “sequence”, and “map”. For each query, there is a list of matching services, for instance, 52 services on topic “protein”, 21 services on topic “user”, and 24 services on topic “genome”. The numbers of services for all the 15 topics are shown in table 4.

Table 4. The collection of services used in the experiments

	Topic	Number of services		Topic	Number of services
1	Protein	52	9	Development	19
2	User	21	10	Business	21
3	Genome	24	11	Weather	19
4	Amazon	31	12	Management	23
5	Commerce	27	13	Match	21
6	Sequence	28	14	Google	19
7	Net	17	15	Code	24
8	Map	21			

4.2 Implementation and Design

Our simulator program was implemented in Java using NetBeans IDE under Windows XP platform, and it could generate query and invocation requests. Each query record includes a user's functional and QoS requirement, as well as the service selected from the returned results and invoked afterwards. It is possible that a query may not have the QoS part. It is also possible that there is no invocation after the query, and some invocations are not resulted from query submissions. In the current setting, the functional query is one of the 15 keywords as listed above. The QoS attributes are reliability, availability and response time, which are supported by QWS dataset [33]. The QoS priorities are randomly produced from the range 1~4 where 1 represents the highest priority and 4 represents no requirement for that QoS attribute. The QoS value requirement is in accordance with the attribute data type as well as its actual values as in the QWS dataset, and we also make sure there are a reasonable number of matching services. To make our simulator more generic, we use a configuration file to save the values of different

parameters, and by adjusting their values we could generate different datasets. The major parameters are listed and explained below:

- The values of constant numbers c_1, c_2, c_3 in equation (1), (6), (7), (8).
- The value of *K*, which measures how many similar users we are going to consider.
- The value of *N*, which measures the maximum number of invocations of a service saved for each user.
- *NQ*: the number of effective queries a user might submit, which is a range value, e.g. 1~10. An effective query is a query which is followed by an actual invocation. The queries without any follow-up actions are not considered. For each user, the number of actual queries submitted would be a random number within this range.
- *NSI*: the number of invocations of a service after submitting a query and selecting this service from the matching results. This is also a range value.
- *NII*: the number of invocations which is not the result of a query. Again, it is a range value.
- *NU*: the number of users who are using the system.

The generated queries and invocation instances were randomly distributed among 12 months. We used the first 11 months' data as the training data to calculate the user similarities, and then the last month's data as the testing data. For queries submitted in the last month, we applied our collaborative filtering algorithm to rank their functionally matching services. The precision of the ranking algorithm is measured afterwards. By checking whether the actually invoked service is in the top result list, we define our measurement as:

$$P(TN) = \frac{1}{|Q_{12}|}\sum_{q \epsilon Q_{12}} in_q(TN) \qquad (9)$$

where Q_{12} contains the queries in the last month, and in_q is a Boolean value, if the service user selected for query q is in the top TN results returned by our ranking algorithm, it is 1, and otherwise 0. In our experiment, the value of TN is chosen as 5 and 10. We denote the precision on top 5 and 10 as *P-5* and *P-10* respectively in the rest of the thesis.

4.3 Experiment Results and Analyses

We ran each experiment 3 times, and the final precision value was averaged on 3 runs. By default, the parameters are set as: NU = 100, NQ = 1~10, NSI = 1~100, NII = 1~10, K=5, N=5, c_1 = 0.1, c_2 = 0.2, c_3 = 0.5. Then in each experiment we change one parameter at a time to different values to see how it affects the precision values. There are altogether 3 groups of experiments. In the first group of experiments, we changed the values of $\boldsymbol{c_3}$, K, N and NQ. Figure 5, 6, and 7 show the results of *P-5* and *P-10* when we change these parameters. In the second experiment, we changed the values of NSI and NII to test their impact on the precision values. The results are presented in Figure 8, 9, 10, and 11. The last experiment is to test how the value of NU affects the precision value. Figure 12 and 13 show the results.

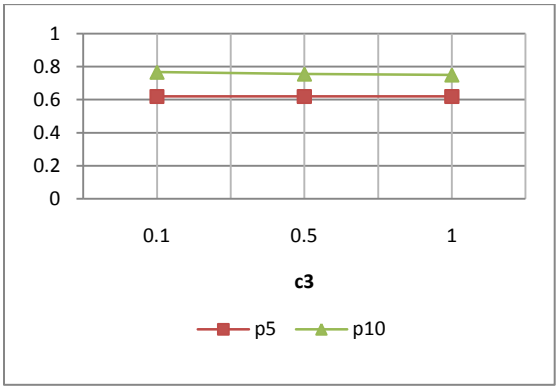
4.3.1 Changing $\boldsymbol{c_3}$, $\boldsymbol{N}$, $\boldsymbol{K}$, and $\boldsymbol{NQ}$

From Figure 5 ,6, and 7, we could see that the values of *P-5* are normally above 0.6 and the values of *P-10* are around 0.8, which means that the probability for users to find their desired services from the top 5 or 10 results is pretty high. This result indicates the effectiveness of our collaborative filtering based ranking algorithm. Without our ranking algorithm, users have to select through the functionally matching services. For instance, for the functional query

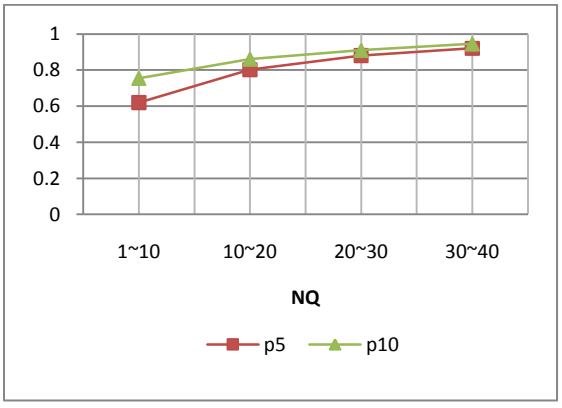
"protein", users may need to check 52 matching services to select the one they want. With our algorithm, it is very likely users could locate the service from the top 10 results.

4.3.1.1 Changing c_3 and *NQ*

In Figure 5(a), we show the result of *P-5* and *P-10* in which c_3 is 0.1, 0.5, and 1 respectively. Figure 5(b) shows the result of *P-5* and *P-10* with different *NQ* values per user.



(a) Changing c_3 values



(b) Changing *NQ* values

Figure 5. Precisions when changing c_3 and *NQ* values

4.3.1.2 Changing *K*

Similarly, we do the experiments on the different values of *K*. The results are shown in Figure 6.

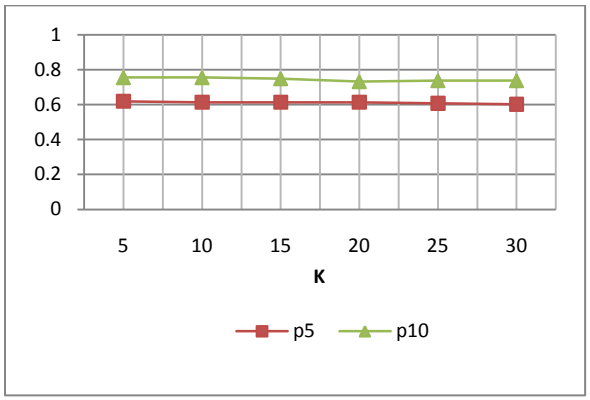


Figure 6. Precisions when changing *K* values

4.3.1.3 Changing *N*

Figure 7 shows the result of changing *N* values.

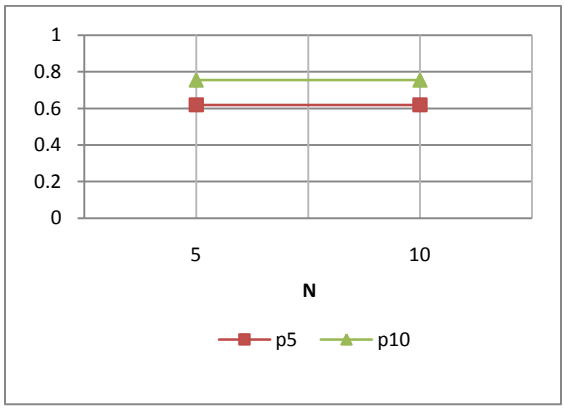


Figure 7. Precisions when changing *N* values

We use the following tables to show the percentage of change (increase or decrease) on *P-5* and *P-10* with different experiment values for parameter *c3*, *K*, *N*, and *NQ* corresponding to the above figures. A positive value indicates an increase in precision and a negative value indicates a decrease in precision. The percentage is calculated as the difference between the second and the first value divided by the first value.

Table 5. Comparison of *P-5* and *P-10* on different c_3 values

Precision	Precision change (%) (c_3:0.1 → 0.5)	Precision change (%) (c_3:0.5→1)
P-5	0	0
P-10	-1.4	-0.5

Table 6. Comparison of *P-5* and *P-10* on different *K* values

Precision	Precision change (%) (*K*:5 → 10)	Precision change (%) (*K*:10 → 15)	Precision change (%) (*K*:15 → 20)	Precision change (%) (*K*:20 → 25)	Precision change (%) (*K*:25 → 30)
P-5	-0.9	0	0	-0.9	-0.9
P-10	0	-0.7	-2.2	0.8	0

Table 7. Comparison of *P-5* and *P-10* on different *N* values

Precision	Precision change (%) (*N*:5 → 10)
P-5	0
P-10	0

Table 8. Comparison of *P-5* and *P-10* on different *NQ* values

Precision	Precision change (%) (*NQ*: 1~10 → 10~20)	Precision change (%) (*NQ* :10~20 → 20~30)	Precision change (%) (*NQ* : 20~30 → 30~40)
P-5	30	8	4.5
P-10	14	5.6	3.9

Figure 5(a) and Table 5 show that changing the value of c_3 doesn't affect the precision values much; *P-10* has only changed 1.4% when c_3 changes from 0.1 to 0.5 which is the biggest change in our results. We got similar results for c_1 and c_2. The experiment results also show that the values of *K* and *N* also have no obvious impact on the precision value. The biggest change on precision for *K* is 2.2% when *K* changes from 15 to 20, and changing of *N* does not change the precision based on our results. So in the later experiment we fix their values to 5 because when this number is bigger, it takes longer time to run the algorithm. Figure 5(b) and Table 8 show that when there are more queries recorded for users, the precision level is higher, which is a reasonable conclusion because normally the recommender system is more accurate when there is more user data collected. Increasing *NQ* can significantly improve the precision, especially when *NQ* increases from 1 ~ 10 to 10 ~ 20, there is a 30% increase for *P-5* and 14% increase for *P-10*. If we keep increasing this number, the precision value will be even higher, however, the improvement is becoming less and less. Since a larger number of queries definitely increase the processing time, to balance the efficiency and accuracy, we set this number to 20~30 in the rest of the experiment.

4.3.2 Changing *NSI* and *NII*

In the second experiment, we changed the values of *NSI* and *NII* to test their impact on the precision values. The results are presented in Figure 8, 9, 10, and 11.

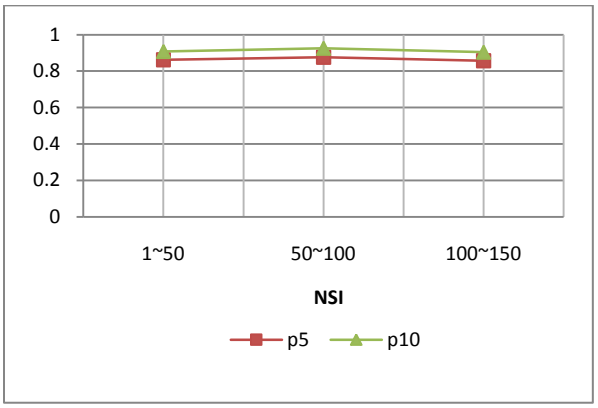


Figure 8. Precisions when changing *NSI* values

Table 9. Comparison of *P-5* and *P-10* on different *NSI* values

Precision	Precision change (%) (*NSI* :1~50 → 50~100)	Precision change (%) (*NSI* : 50~100 → 100~150)
P-5	1.6	-2.2
P-10	1.9	-2.3

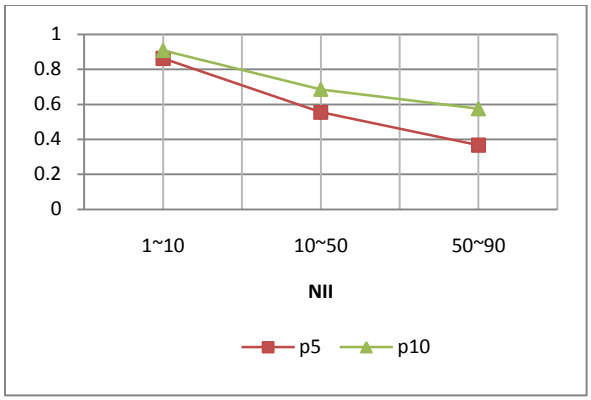


Figure 9. Precisions when changing *NII* values (*NSI* = 1~50)

Table 10. Comparison of *P-5* and *P-10* on different *NII* values (*NSI* = 1~50)

Precision	Precision change (%) (*NII* : 1~10 → 10~50)	Precision change (%) (*NII* : 10~50 → 50~90)
P-5	-35.6	-33.8
P-10	-24.4	-16.1

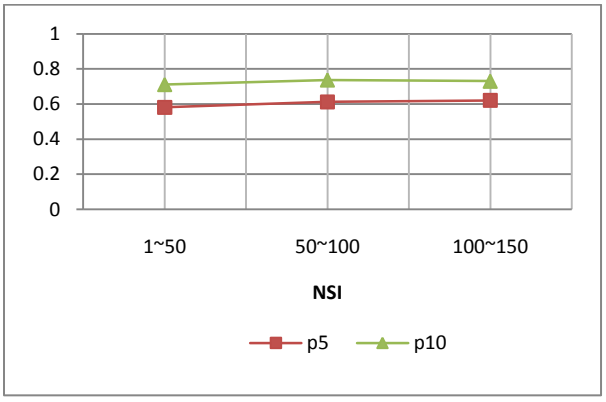


Figure 10. Precisions when changing *NSI* values (*NQ* = 20~30 and *NII* = 20~30)

Table 11. Comparison of *P-5* and *P-10* on different *NSI* values

(*NQ* = 20~30 and *NII* = 20~30)

Precision	Precision change (%) (*NSI* :1~50 → 50~100)	Precision change (%) (*NSI* :50~100 → 100~150)
P-5	5.3	1.4
P-10	3.6	-0.9

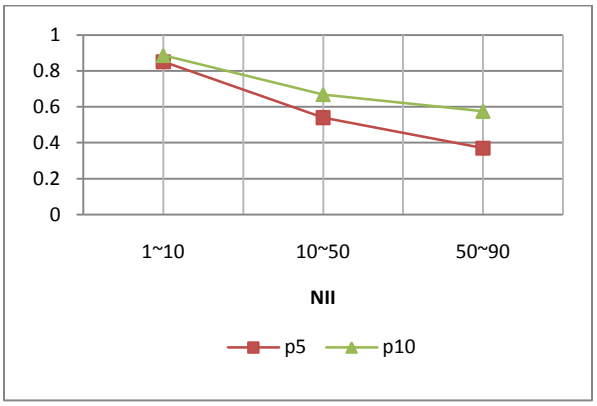


Figure 11. Precisions when changing *NII* values

(*NQ* = 20~30 and *NSI* = 50~100)

Table 12. Comparison of *P-5* and *P-10* on different *NII* values

(*NQ* = 20~30 and *NSI* = 50~100)

Precision	Precision change (%) (*NII* :1~10 → 10~50)	Precision change (%) (*NII* :10~50 → 50~90)
P-5	-36.5	-31.4
P-10	-24.5	-14

According to the results shown in previous figures and tables, changing *NSI* value doesn't affect the precision value much. The biggest percentage change is only around 5% on *P-5* when *NSI* increases from 1~50 to 50 ~100. It is understandable since this value is only used in the similarity calculation, and in the ranking part, we only consider a certain number of invocation instances. However, *NII* value has a very obvious impact on the precision value as shown in Figure 9 and 11. When it is higher, the precision is lower. The precision decrease can be more than 30% on *P-5* and more than 20% on *P-10*. The main reason is that our ranking algorithm heavily depends on users' previous query histories, and if there are more invocations which are not results of query submissions, the ranking accuracy will definitely be lower. From above results, we can also see that changing *NQ*, *NSI* and *NII* have more influence on *P-5* than *P-10*.

4.3.3 Changing NU

The last experiment is to test how the value of *NU* affects the precision value. Figure 12 and 13 show the results.

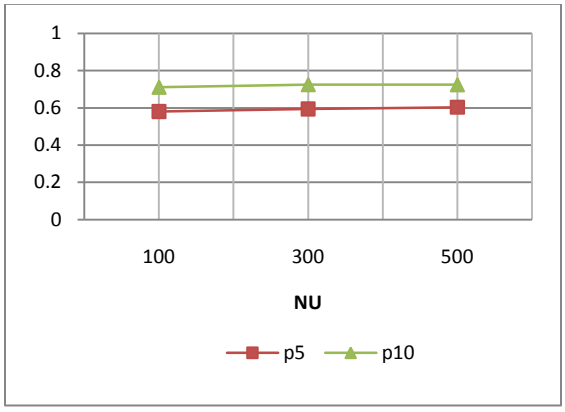


Figure 12. Precisions when changing *NU* values(NII = 20~30)

Table 13. Comparison of *P-5* and *P-10* on different *NU* values

Precision	Precision change (%) (*NU* :100 → 300)	Precision change (%) (*NU* :300 → 500)
P-5	2.2	1.6
P-10	1.9	-0.08

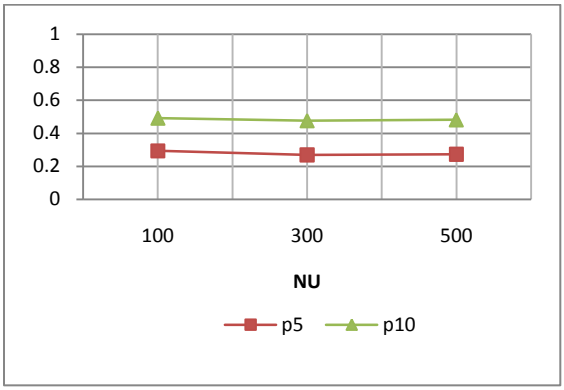


Figure 13. Precisions when changing *NU* values (NII = 50~90)

Table 14. Comparison of *P-5* and *P-10* on different *NU* values (*NII* = 50~90)

Precision	precision change (%) (*NU* :100 → 300)	precision change (%) (*NU* :300 → 500)
P-5	-8.1	1.5
P-10	-3.1	1

From these figures and tables, we could see that the value of *NU* has no obvious effect on increasing the precision values. There is a relatively bigger decrease of 8.1% on P-5 when the *NU* changes from 100 to 300. This can be caused by larger *NII* values due to our experiment

setting which uses randomly produced numbers. Normally for the recommender system, when more users are using the system, the accuracy is higher. However, in our experiment setup, all users followed one of a few pre-defined patterns when they submitted queries, which is to make sure there are similar users in the generated dataset. Due to this configuration, even when there are more users, the precision value does not change much. Since a higher *NU* value means a longer processing time, this value is set as 100.

From these experiment results, we could see that the precision of our ranking algorithm is mainly affected by the number of effective queries and the number of random invocations. When the number of queries is larger, which means there is more usage data, the accuracy is better. When the number of random invocations is smaller, which means most of invocations are connected to queries, the accuracy is better. When we apply our algorithm to the real selection system, as long as we could collect a reasonable amount of user data, and there is a high chance of finding similar users, the accuracy of the service selection system could be largely improved.

4.4 Summary

In this chapter, we explained the dataset we used for experiments as well as the experiment design. We have evaluated and analyzed our system and proved that our system worked for different user invocation histories and different number of users. We could see from these experiments that the increased number of initial invocation can positively affect the precision, whereas the increased random invocations can negatively affect the precision. It could be easily understood that users who have clear preferences will get better recommendations, and users who select web services in a random pattern will benefit less from the recommendation.

CHAPTER 5
CONCLUSIONS AND FUTURE WORKS

5.1 Conclusions

In this thesis, we proposed a collaborative filtering based service selection algorithm. The user similarity is calculated mainly based on the past QoS queries and the actual invocations. Users are considered similar if they invoked same services and submitted similar queries. During the selection process, the service will be recommended to users depending on its matching degree with the QoS requirement and its collaborative filtering ranking score calculated on its invocation history from similar users.

Our system overcomes the cold start problem for new users and new services, using the following strategies: if the user is a new user, the system makes recommendation based on all other users' interests; for services which have never been invoked before, we provide a separate list ordered by their publication dates. Our system is flexible to support as many QoS attributes as the user prefers.

The effectiveness of our system is proved by our experiments and use case scenarios. We could see from the experiments that the recommendation precisions depend on the number of invocations. The increased number of invocations due to queries can increase the precision of the recommendation and the increased number of random invocations would decrease the precision. From use case scenarios, we can see that our system can make recommendation based on users' previous experiences; it can infer a user's preference from previous query and invocation history even when the user doesn't submit QoS requirements in the current query. The use case scenario

also shows how invocations, user QoS preferences, and required values, can affect user similarity and how invocation frequency and time can change the ranking scores.

The major contributions of this research work are:

- To the best of our knowledge, it is a novel idea of using invocation and query history, especially the QoS query part to build the collaborative filtering system. We compute the user similarity based on multiple factors: invocation, QoS preference and QoS required values.
- With the implicit feedback, our selection and ranking algorithm considers the changing requirement and service performance over time and it supports the personalized recommendation. Our algorithm could take advantages of the existing QoS-based selection models and overcome some of the shortcomings (e.g. cold start problem) of the traditional collaborative filtering systems.
- To implicitly collect data, we introduce a unique and practical architecture model which includes a centralized data collecting mechanism for the web service selection system using collaborative filtering techniques.

5.2 Future Works

A few directions we would like to work on in the future include:

Firstly, a statistical analysis on our experimental results should be conducted to evaluate the confidence level of the result.

Secondly, a larger-scaled experiment could be conducted with more user records, and a few different data distribution patterns could be tested. We would also like to try the method as proposed in [3] [4] for the data collection.

Finally, we may expand our work to the hybrid approach to check whether it will further improve the performance. Also, implementing the actual QoS-based ranking algorithm and using a rank aggregation method to combine it with the recommendation algorithm will be helpful to evaluate the overall system performance.

REFERENCES

[1] Adomavicius, G., and Tuzhilin, A.: Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. IEEE Transactions on Knowledge and Data Engineering, Vol. 17, Issue 6, 734--749 (2005)

[2] Schafer, J.B., Frankowski, D., Herlocker, J., and Sen, S.: Collaborative Filtering Recommender Systems. In: Brusilovsky, P., Kobsa, A., and Nejdl, W. (Eds.) The Adaptive Web, pp. 291--324 (2007)

[3] Shao, L., Zhang, J., Wei, Y., Zhao, J., Xie, B., and Mei, H.: Personalized QoS Prediction for Web Services via Collaborative Filtering. In: 5th International Conference on Web Services, pp. 439--446 (2007)

[4] Zheng, Z., Ma, H., Lyu, M.R., and King, I.: WSRec: A Collaborative Filtering Based Web Service Recommender System. In: 7th International Conference on Web Services, pp. 437--444 (2009)

[5]Averbakh, A., Krause, D., and Skoutas, D.: Recommend me a Service: Personalized Semantic Web Service Matchmaking. In: 17th Workshop on Adaptivity and User Modeling in Interactive Systems (2009)

[6] Manikrao, U.S., and Prabhakar, T.V.: Dynamic Selection of Web Services with Recommendation System. In: International Conference on Next Generation Web Services Practices, pp. 117 (2005)

[7] Birukou, A., Blanzieri, E., D'Andrea, V., Giorgini, P., and Kokash, N.: Improving Web Service Discovery with Usage Data. IEEE Software, Vol. 24, Issue 6, 47--54 (2007)

[8] Kokash, Natallia (University of Trento) PhD dissertation: Engineering Service-Oriented Systems: Modeling, Discovery and Quality, http://homepages.cwi.nl/~kokash/index.html, Last retrieved at November 2010.

[9] Tran, V.X., Tsuji, H., and Masuda, R.: A New QoS Ontology and its QoS-based Ranking Algorithm for Web Services. Simulation Modeling Practice and Theory, Vol. 17, Issue 8, 1378--1398 (2009)

[10] Alspector, J., Koicz, A., and Karunanithi, N.: Feature-based and Clique-based user model for Movie selection: A Comparative Study. User Modeling and User-Adapted Interaction, Vol 7, Issue 4, 279--305 (1997)

[11] MovieLens: http://www.movielens.org, Last retrieved at November 2010.

[12] Rong, W., Liu, K., and Liang, L.: Personalized Web Service Ranking via User Group combining Association Rule. In: 7th International Conference on Web Services, pp. 445--452 (2009)

[13] Xue, G.R., Zeng, H.J., Chen, Z., Yu, Y., Ma, W.Y., Xi, W.S., and Fan, W.G.: Optimizing Web Search Using Web Click-through Data. In: 13th ACM International Conference on Information and Knowledge Management, pp. 118--126 (2004)

[14] Agichtein, E., Brill, E., and Dumais, S.T.: Improving Web Search Ranking by Incorporating User Behavior. In: 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 19--26 (2006)

[15] Mobasher, B.: Data Mining for Web Personalization. In: Brusilovsky, P., Kobsa, A., and Nejdl, W. (Eds.) The Adaptive Web, pp. 90--135 (2007)

[16] Abdi, H.: Kendall Rank Correlation. In Salkind, N.J. (Eds.) Encyclopedia of Measurement and Statistics, pp. 508--510 (2007)

[17] Cross, V. and Cabello, C.: A Mathematical Relationship Between Set-Theoretic and Metric Compatibility Measures, In: 3rd International Symposium on Uncertainty Modeling and Analysis and Annual Conference of the North American Fussy Information Processing Society, pp.169-174 (1995)

[18] Kritikos, K., and Plexousakis, D.: Mixed-Integer Programming for QoS-based Web Service Matchmaking. IEEE Transactions on Services Computing, Vol. 2, Issue 2, 122--139 (2009)

[19] Ma, Q., Wang, H., Li, Y., Xie, G., and Liu, F.: A Semantic QoS-aware Discovery Framework for Web Services. In: the 6th IEEE International Conference on Web Services, pp.129--136 (2008)

[20] Menasce, D. A., and Dubey, V.: Utility-based QoS Brokering in Service Oriented Architectures. In: 5th IEEE International Conference on Web Services, pp. 422--430 (2007)

[21] Herssens, C., Jureta, I.J., and Faulkner, S.: Dealing with Quality Tradeoffs during Service Selection. In: 5th International Conference on Autonomic Computing, pp. 77--86 (2008)

[22] Lamparter, S., Ankolekar, A., Studer, R., and Grimm, S.: Preference-based Selection of Highly Configurable Web Services. In: 16th International Conference on World Wide Web, pp. 1013--1022 (2007)

[23] Skoutas, D., Sacharidis, D., Simitsis, A., Kentere, V., and Sellis, T.: Top-k Dominant Web Services Under Multi-Criteria Matching. In: 12th International Conference on Extending Database Technology: Advances in Database Technology, pp. 898--909 (2009)

[24] Yu, Q., and Bouguettaya, A.: Computing Service Skyline from Uncertain QoWS. IEEE Transactions on Services Computing, Vol. 3, Issue 1, 16--29 (2010)

[25] eBay: http://www.ebay.com, Last retrieved at November 2010.

[26] Amazon: http://www.amazon.com, Last retrieved at November 2010.

[27] Wang, H.C., Lee, C.S., and Ho, T.H.: Combining Subjective and Objective QoS Factors for Personalized Web Service Selection. Expert Systems with Applications, Vol. 32, Issue 2, 571--584 (2007)

[28] Aslam, J.A., and Montague, M.: Models for Meta-search. In: 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 276--284 (2001)

[29] Ding, C., Sambamoorthy, P., and Tan, Y.: QoS Browsing for Web Service Selection. In: 7th International Conference on Service Oriented Computing, pp. 285--300 (2009)

[30] Manning, C.D., Raghavan, P., and Schütze, H.: Introduction to Information Retrieval, Cambridge University Press (2008)

[31] Ruiz-Cortés, A., Martín-Díaz, O., Toro, A.D., and Toro, M.: Improving the Automatic Procurement of Web Services Using Constraint Programming. International Journal on Cooperative Information Systems, Vol. 14, Issue 4, 439--468 (2005)

[32] Yan, J., and Piao, J.: Towards QoS-based Web Service Discovery. In: 6th International Conference on Service Oriented Computing, pp. 200--210 (2008)

[33] Al-Masri, E., and Mahmoud, Q. H.: QoS-based Discovery and Ranking of Web Services. In: 16th International Conference on Computer Communications and Networks, pp. 529--534 (2007)

APPENDIX A – Computing similarity matrix

```
void profiling(String historyRandom, String history, String fileProfile)
     {
         profile = new double[numUsers];

         double[] simWVij = new double[totService];
         ArrayList CSij, ISi, ISj;
        //lists of commonly invoked services and user i invoked and user j invoked services

         Object[][] hisUi, hisUj;
         int rowi, rowj;
         FileOutputStream fos;
         PrintWriter out;
         try
         {
            fos = new FileOutputStream(fileProfile);
            out= new PrintWriter(new OutputStreamWriter(fos));
            out.close();

            fos= new FileOutputStream(fileProfile, true);
            out = new PrintWriter(new OutputStreamWriter(fos));

            int i, j;
            for (i = 0; i < numUsers; i++)
            {
                for (j = i; j < numUsers; j++)
                {

                    CSij = new ArrayList();
                    ISi = new ArrayList();
                    ISj = new ArrayList();

                    rowi = getNumLines(i, history);
                    rowj = getNumLines(j, history);
                    if (rowi > 0 && rowj > 0)
                    {
                        hisUi = new Object[rowi][his_col];
                        hisUj = new Object[rowj][his_col];

                        hisUi = getRecordsU(historyRandom, i);
                        hisUj = getRecordsU(historyRandom, j);
                        if (hisUi != null && hisUj != null)
                        {
                            CSij = cInvok(hisUi, hisUj, i, j);
                            ISi = getServiceU(hisUi, i);
                            ISj = getServiceU(hisUj, j);

                            qWVSim(hisUi, hisUj, CSij, history, simWVij, i, j);
                        }

                        profile[j] = 0;

                        if (CSij != null && ISi != null && ISj != null)
                        {
                            if (simWVij !=  null)
                            {
                                int k;
                                for (k = 0; k < CSij.size(); k++)
                                {
                                    profile[j] += (simWVij[k] + c2);
                                }
                            }
                            profile[j] = (2.0/(ISi.size() + ISj.size()))*profile[j];
                        }
                    }
                }
```

```
                writeLine(profile, out);
                size = new int[] {numUsers, numUsers};
                writeSize(size, "F:\\Coding\\configure_rank.dat");
            }
            out.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
```

```
void qWVSim(Object[][] hisUi, Object[][] hisUj, ArrayList CijQoS,
            String hisFile, double[] simWVij, int i, int j)
    {
        ArrayList simKij = new ArrayList();

        double simK;
        int numLineSerik = 0;
        int numLineSerjk = 0;

        Object[][] serUik, serUjk;

        int rowi = getNumLines(i, hisFile);
        int rowj = getNumLines(j, hisFile);

        if (rowi > 0 && rowj > 0)
        {

            serUik = new Object[rowi][his_col];

            serUjk = new Object[rowj][his_col];

            int s, k;
            if (CijQoS != null)
            {
                for (s = 0; s < CijQoS.size(); s++)
                {
                    simK = 0.0;

                    k = toInt(CijQoS.get(s));

                    serUik = getServiceHis(hisUi, k);
                    numLineSerik = numLineSer;
                    serUjk = getServiceHis(hisUj, k);
                    numLineSerjk = numLineSer;

                    /* Kendall and Jacard coefficient for all instances for service k
                    of user i and j */
                    if (numLineSerik != 0 && numLineSerjk != 0)
                    {
                        int m, n;
                        try {
                            m = 0;
                            while (serUik[m][0] != null && m < serUik.length)
                            {
                                n = 0;
                                while (serUjk[n][0] != null && n < serUjk.length)
                                {
                                      simK += (simWIns_Kendall(serUik, serUjk, m,
                                      n)+c1) * (simVIns_Union(serUik, serUjk, m, n));
                                    n++;
                                }
                                m++;
                            }
```

```
                        // similarity value of user i and j on service k
                        simK = simK/(numLineSerik*numLineSerjk);
                    }
                    catch (ArrayIndexOutOfBoundsException e)
                    {
                        e.getStackTrace();
                    }

                }
                simKij.add(simK);
            }
        }
    }
    if (simKij != null)
     {
        int l;
        if (i != j)
        {
            for (l = 0; l < simKij.size(); l++)
            {
                simWVij[l] = toDouble(simKij.get(l));
            }
        }
        else
        {
            for (l = 0; l < simKij.size(); l++)
            {
                simWVij[l] = 1.0;
            }
        }
     }
     else
     {
         simWVij= null;
     }
}
```

```
static void writeLine(double[] array, PrintWriter out)
   {
       if (out != null)
       {
           int i;
           for (i = 0; i < array.length; i++)
           {
              out.print(array[i] + ",");
           }
           out.println();

       }
   }
```

APPENDIX B – Computing weight similarity

```
/* return Kendall coefficient for service k on instance m and n of user i and j*/
double simWIns_Kendall(Object[][] serUik, Object[][] serUjk, int m, int n)
     {
        double tau1 = 0.0;
        double tau = 0.0;
        int p = 0;
        int Nc = 0;

        ArrayList atti = new ArrayList();
        ArrayList attj = new ArrayList();

        ArrayList unionAtt= new ArrayList();
        ArrayList interAtt = new ArrayList();

        atti = getAttrUkh(serUik, m);
        attj = getAttrUkh(serUjk, n);

        unionAtt = unionAttr(atti, attj);
        interAtt = interAttr(atti, attj);

        if (unionAtt != null && interAtt != null)
        {
           Nc = concordant(atti, attj, unionAtt);
           p = unionAtt.size();
           if (p > 1)
                tau1 = (double)2*Nc/(p*(p-1));
           else if (p == 0)
                tau1 = 0;
           else
                tau1 = 1.0;

           tau = tau1*((double)interAtt.size()/p);
        }
        return tau;
     }
```

```
/* return the number of concordant pairs*/
int concordant(ArrayList atti, ArrayList attj, ArrayList unionAtt)
     {
        int concord = 0;

        ArrayList orderAttm = new ArrayList();
        ArrayList orderAttn = new ArrayList();

        orderAttm = orderPre(unionAtt, atti);
        orderAttn = orderPre(unionAtt, attj);

        if (orderAttm != null && orderAttn != null )
        {
            int i, j;

            for (i = 0; i < orderAttm.size() - 1; i++)
            {
                for (j = i + 1; j < orderAttm.size(); j++)
                {
                    if (toInt(orderAttm.get(i)) > toInt(orderAttm.get(j)) &&
                            toInt(orderAttn.get(i)) > toInt(orderAttn.get(j)))
                        concord++;
                    if (toInt(orderAttm.get(i)) < toInt(orderAttm.get(j)) &&
                            toInt(orderAttn.get(i)) < toInt(orderAttn.get(j)))
                        concord++;
                }
            }

        }
        return concord;
     }
```

APPENDIX C – Computing value similarity

```
/* return Jaccard coefficient for service k on instance m and n of user i and j*/
double simVIns_Union(Object[][] serUik, Object[][] serUjk, int m, int n)
     {
        double simV = 0.0;
        int index = 0;
        double[] range1 = new double[2];
        double[] range2 = new double[2];

        ArrayList attikm = new ArrayList();
        ArrayList attjkn = new ArrayList();

        ArrayList interAtt = new ArrayList();

        attikm = getAttrUkh(serUik, m);
        attjkn = getAttrUkh(serUjk, n);

        if (attikm != null && attjkn != null)
        interAtt = interAttr(attikm, attjkn);

        if (interAtt != null)
        {
            int i;
            for (i = 0; i < interAtt.size(); i++)
            {

                index = toInt(interAtt.get(i));
                if (index == 0)
                {
                    range1[0] = toDouble(serUik[m][6]);
                    range1[1] = toDouble(serUik[m][7]);
                    range2[0] = toDouble(serUjk[n][6]);
                    range2[1] = toDouble(serUjk[n][7]);
                }
                else if (index == 1)
                {
                    range1[0] = toDouble(serUik[m][9]);
                    range1[1] = toDouble(serUik[m][10]);
                    range2[0] = toDouble(serUjk[n][9]);
                    range2[1] = toDouble(serUjk[n][10]);
                }
                else if (index == 2)
                {
                    range1[0] = toDouble(serUik[m][12]);
                    range1[1] = toDouble(serUik[m][13]);
                    range2[0] = toDouble(serUjk[n][12]);
                    range2[1] = toDouble(serUjk[n][13]);
                }
                else
                    System.out.println("Error");
                if (max(range1[0], range1[1]) <= min(range2[0],range2[1]) ||
                        min(range1[0], range1[1]) >= max(range2[0], range2[1]))
                {
                    simV += 0;
                }
                else
                    simV += interRange(range1, range2)/unionRange(range1,range2);
            }
            return simV/interAtt.size();
        }
         return 0;
     }
```

APPENDIX D – Ranking

```
double rank(int qId, int ui, int s)
    {
        double score = 0;

        Object[][] topSims = new Object[numTopUsers][2];
        topUsers(ui, topSims);

        int uj;
        int j = 0;
        while (j < topSims.length && topSims[j][0] != null)
        {
            uj = toInt(topSims[j][0]);
            score += toDouble(topSims[j][1])*(freq(qId, uj, s)/N);
            j++;
        }

        return score;
    }
```

```
double freq(int qId, int uj, int s)
    {
        int f = 0;
        int row = getNumLines(uj, fnt);
        Object[][] hisUj = new Object[row][his_col];
        Object[][] serUjk = new Object[row][his_col];

        hisUj = getRecordsU(fntRandom, uj);
        serUjk = getServiceHis(hisUj, s);
        int numLineSerjk;
        if (N > numLineSer)
            numLineSerjk = numLineSer;
        else
            numLineSerjk = N;

        int insk, tjk;
        insk = 0;
        while (insk < numLineSerjk)
        {
            tjk = toInt(serUjk[insk][3]);
            if (qId != -1)
                f += ((double)(tjk- startTime)/(double)(currentTime -
                      startTime))*(1+c3);
            else
                f += ((double)(tjk- startTime)/(double)(currentTime - startTime))*(c3);
            insk++;
        }

        return f;
    }
```