CHAPTER - 7

PROPOSED ARCHITECTURE FOR THE WEB SERVICE RECOMMENDATION USING HYBRID RECOMMENDER SYSTEM

7.1 Preamble

In this research work, a hybrid recommender system for web service is proposed which uses query and invocation history data. Related data from query and invocation logs are extracted which is used for similarity computation. The collaborative filtering mechanism is applied in the ranking process.

In this work, invocation logs and query logs are used for usage data. From the invocation log, the user’s invocation time and invocation frequency on different services can be found. Query log saves the user’s expectation on QoS properties in each request. From these logs infer the user’s expectation and preferences on QoS properties as well as service performance are inferred. For example, if a service is invoked by a user with some sort of QoS expectations or requirements many times, it can be inferred 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.

Users who have similar QoS expectations and invocation histories are considered as similar users. If a service has been invoked by a group of similar users, this service is considered to satisfy this type of users’ similar requests. Through the invocation frequency, the level of satisfaction is inferred; through the invocation time, the user’s preference change or service performance drifting over time is inferred. Therefore, a service is ranked 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, QoS value requirements as well as preferences are used to build user profiles, and then generate the user-user similarity matrix using an offline computation routine. In this routine, the Kendall tau coefficient is used, 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 the similarities between QoS value requirements from two users are computed using Jaccard Coefficient [141] method.
7.2 Proposed Architecture for Ranking Web Services using Log Data

Based on the discussions of previous sections, a collaborative filtering based service selection system is proposed using query and invocation history data. Related data are extracted from query and invocation logs and perform similarity computation. The collaborative filtering mechanism is then applied in the ranking process. The following sections contain detailed explanation about the system architecture, the history data used for selection, the algorithms used in finding similar users, as well as the ranking algorithm.

The architecture model proposed in this work for selecting services is shown in Fig. 7.1 which consists of three parts, namely, client, server and centralized server.

Figure 7.1: Architecture of the proposed service selection system

In the client part, service requestor will request a service with or without their QoS values and preference. The browser is responsible for forwarding the client request to the centralized server and the servers. In the server part, servers are used for recording the usage history
information (i.e. invocation data’s) into log file. During the invocation, servers will store user’s QoS values and preference, invoked service name, user id and invocation month and year. The service selection and ranking mechanism along with the service description repository (WSDL) constitute the other part of the proposed system. The centralized server consists of three major data repositories and five major components. The three major data repositories are service description data, user-service matrix, and user-user matrix.

- **Service description data**: This data contains both functional and 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 servers and then processed and saved into the central repository. The other two data repositories are created by the proposed system.

- **User-Service (U-S) matrix**: The user-service matrix is extracted from central repository (log files). This matrix contains the user’s 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.

- **User-User (U-U) matrix**: The user-user matrix is created through computing user similarity based on query and invocation history information provided by U-S matrix. Details about these calculation algorithms are in later sections.

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

**Log data collection**: 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 this proposed system, the servers contain query log and invocation log files. 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.
User similarity calculation: 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 U-S matrix. Based on the U-S matrix, user similarity will be calculated and saved into the U-U matrix to improve the efficiency in finding the similar users at run time. These two matrices are computed and updated offline periodically.

Service selection: 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.

Service ranking: 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. To produce a collaborative-based ranking, service recommendation component needs the data from the two matrices.

Finding Similar Users
Finding similar users is the key part in a collaborative recommender system. In this proposed 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.

Generating the U-S matrix
The U-S matrix is to record the invocation history for all users. Its values are extracted from query logs and invocation logs. If \( m \) users and \( n \) services exists, user’s set is denoted as \( U = \{u_1, u_2... u_m\} \), and the service’s set is denoted as \( S = \{s_1, s_2... s_n\} \). For each user \( u_i \in U \), the history of call on service \( s_j \in S \) is denoted as:

\[
IH_{ij} = \left( q_{ij}^1, t_{ij}^1 \right), \left( q_{ij}^2, t_{ij}^2 \right) ... \left( q_{ij}^{lij}, t_{ij}^{lij} \right)
\]

(7.1)

where \( l_{ij} \) is the total number of calls on \( s_j \) from \( u_i \), \( q_{ij}^k \) and \( t_{ij}^k \) represents the related query and the call time of the \( k^{th} \) (\( 0 < k < l_{ij} \)) call occasion on \( s_j \) from \( u_i \). There is chance of having empty history when the user hasn’t invoked any services. A cut-off value is set up to record how
many calls are considered for saving space and to avoid the user’s call data having much effect on the ultimate result, particularly when there are more calls for service by this user so that only the latest $N$ calls are saved where $N$ is defined as a small number. When there is no related query with the call, $q^k_i$ is represented as $\emptyset$ and when there is query it is represented as $(fq^k_i, qq^k_i)$, where $fq^k_i$ and $qq^k_i$ are functional and non-functional (QoS) parts of the user query.

Term vector is used to denote the functional part of query. The QoS part of query $qq^k_i$ has requirements on several QoS. User preference is used in determining the value and weight of the QoS requirements. QoS part is defined as $qq^k_i = ((qq^k_{i1},w,qq^k_{i1}), (qq^k_{i2},w,qq^k_{i2}), \ldots, (qq^k_{ip},w,qq^k_{ip}))$ where $p$ is the number of QoS attributes considered by the system, $qq^k_{iph}.w$ and $qq^k_{iph}.v$ are the weight and value requirements on the $h^{th}$ ($1 \leq h \leq p$) attribute for the $k^{th}$ call instance. The value of $qq^k_{iph}.w$ is considered from 1 to $p+1$ subject to the order of preference, with a least number denotes an ideal attribute and the value $p+1$ is considered if the user is not anxious on this attribute. Since the value requirement could have different data types depending on the QoS attribute $qq^k_{iph}.v$ as a set. The sample user-service matrix is shown in the Table 7.1.

Table 7.1: Sample User-Service matrix

<table>
<thead>
<tr>
<th>User id</th>
<th>Service1</th>
<th>Service2</th>
<th>Service3</th>
<th>Service4</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>((10,3),(80,2), (10,1), 03/12), (α,07/12)</td>
<td>((23,2),(70,3), (10,1), 02/12), (α,07/12)</td>
<td></td>
<td>((10,1),(90,2), (20,3), 02/12)</td>
</tr>
<tr>
<td>U2</td>
<td>((10,3),(80,1), (10,2), 05/12)</td>
<td></td>
<td>((20,3),(80,2), (20,1), 04/12)</td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td>((40,2),(80,3), (10,1), 02/12)</td>
<td>((20,3),(80,2), (25,1), 02/12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U4</td>
<td></td>
<td>((20,2),(90,3), (10,1), 02/12)</td>
<td></td>
<td>((15,1),(90,2), (10,3), 02/12)</td>
</tr>
<tr>
<td>U5</td>
<td>((23,2),(70,3), (10,1), 02/12)</td>
<td></td>
<td>((15,3),(90,1), (20,3), 05/12)</td>
<td></td>
</tr>
</tbody>
</table>
Generation of U-U matrix

After the U-S matrix is generated from the logs, next the user similarity is calculated and saved it into the U-U matrix. In the WS context, recommender systems sometime needs the users to offer ratings after call of services, using the call rate as the preference and some users use the observed QoS values for prediction. In this proposed system, the recommender system is used for ranking services using QoS values. Hence, the user similarity is based on their QoS requirements and the subsequent decision-making on selection and invocation of services.

It is considered that if any two users select and invoke the same service from a list of functionally similar services, their QoS requirements may be the same to decide in selecting service and if their similarity based on QoS requirements can be found from their actual queries, the similarity level is higher. The similarity level is higher when there are more commonly invoked services for two users. Since the invocation can expose accurately about the user’s current behaviors, the current service calls rather than the previous calls are considered. The user similarity is calculated in four steps such as weight similarity, value similarity, query similarity and user similarity.

Query similarity

This work focus on the QoS query part for each service invocation. As a weight vector and a value vector for all attributes are part of each QoS query, the complete query similarity is found using both weight and value similarities. The weight similarity is calculated to check whether users $u_i$ and $u_j$ are having same preferences on QoS attributes in a service call of $s_k$, and the value similarity is calculated to check whether the same two users are having identical QoS value requirements ranges in this invocation. The weight vector of the $h^{th}$ service call from user $u_i$ on service $s_k$ is denoted as $q\hat{w}_{ih} = (qq_{ih1}, w, qq_{ih2}, w, \ldots, qq_{ihp}, w)$, and the value vector is represented as $q\hat{v}_{ih} = (qq_{ih1}, v, qq_{ih2}, v, \ldots, qq_{ihp}, v)$. The query similarity between $u_i$ and $u_j$ on service $s_k$ is calculated using Eqns. 7.2 or 7.3.

$$\text{If } I_{ih}^i = 0 \text{ or } I_{jh}^j = 0, \text{sim}_{ijk} = 0 \text{ otherwise,}$$

$$Sim_{ijk} = \frac{1}{I_{ik} \times I_{jk}} \sum_{h_1=1}^{I_{ih}} \sum_{h_2=1}^{I_{jh}} (SV(q\hat{w}_{ikh1}, q\hat{v}_{ikh2}) \times (SW(q\hat{w}_{ikh1}, q\hat{w}_{ikh2}) + c_1))$$

where $I_{ik}$ is the number of service calls on $s_k$ from $u_i$ considering QoS queries (non-empty), $I_{jh}$ is the number of service calls on $s_k$ from $u_j$ considering QoS queries (non-empty), SV(.) is for
finding the similarity between the value vectors, \( SW(.) \) is for calculating the similarity between the two weight vectors, and \( c_1 \) is a small constant value which is set to 0.1 in this work.

The weight vector is used to save the user’s preferences on \( p \) QoS attributes and it is treated as a ranked list. So instead of using Pearson Correlation Coefficient (PCC) or Cosine Similarity for calculating the similarity, the Kendall tau coefficient [138] is used to measure the arrangement between the two ranked lists. When two ranked lists \((x_1, x_2, \ldots, x_n)\) and \((y_1, y_2, \ldots, y_n)\) exists, 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 \( y_i > y_j \), then they are considered as concordant, otherwise, they are considered as discordant. The Kendall tau coefficient is computed using Eqn. 7.4.

\[
\Gamma_A = \frac{n_c - n_d}{\frac{1}{2}n^*(n-1)} \quad (7.4)
\]

where \( n_c \) is the total pairs which are concordant, \( n_d \) is the total discordant pairs, and \( n \) is the total attributes which at least one user is concerned with. Here, \( \frac{1}{2}n^*(n-1) \) is the total number of ordered pairs that an ordered set of \( n \) objects can compose. When the user has the identical preference on different QoS attributes, this definition is altered a little bit in order to allow 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 \( y_i \geq y_j \), then they are considered as concordant, otherwise, they are considered as discordant. If the result is 1, it is considered as perfect agreement and if it is -1 then it is considered as perfect disagreement. The similarity value ranges from 0 to 1, and if it is 0 then two preference orders are completely different and if the value is 1 they are the same. So the Eqn. 7.4 is converted as Eqn. 7.5.

\[
\Gamma_A' = \frac{n_c}{\frac{1}{2}n^*(n-1)} \quad (7.5)
\]

Also if the common attributes are more, the similarity value will be high. With two weight vectors \( \bar{q}w_{ikh_1} \) and \( \bar{q}w_{jkh_2} \) first all the attributes with weight values not equal to \( p+1 \) is found meaning that users are interested in 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 Eqn. 7.6 is used for calculating the final weight similarity value.

\[
SW(\bar{q}w_{ikh_1}, \bar{q}w_{jkh_2}) = \frac{na_c}{na_u} \cdot \Gamma_A' \quad (7.6)
\]
To calculate the value similarity, since the value requirement on each attribute is represented as a set, Jaccard Coefficient is used to do the calculation, which is more appropriate to measure the similarity between two sets and it is given by Eqn. 7.7.

$$SV(q^i_{vk1}, q^i_{vk2}) = \frac{1}{p} \sum_{h=1}^{p} \left| \frac{q^h_{ik1} \cap q^h_{ik2}}{q^h_{ik1} \cup q^h_{ik2}} \right|$$  (7.7)

**User similarity**

Set of services invoked by $u_i$ is denoted as $IS_i$ and the set of services invoked by user $u_j$ is denoted as $IS_j$. The set of commonly invoked services by two users is defined as $CS_{ij} = IS_i \cap IS_j$. For every service $s_k \in CS_{ij}$, the histories of service calls by the users $u_i$ and $u_j$ is denoted as $IH_{ik}$ and $IH_{jk}$ respectively. The total similarity value denoted as $sim_{ij}$ between users $u_i$ and $u_j$ is computed by totaling up $sim_{ijk}$ of all commonly invoked services and $sim_{ij}$ is computed using the Eqn. 7.8.

$$sim_{ij} = \frac{2^{|CS_{ij}|}}{|IS_i| \times |IS_j|} \times \frac{1}{|CS_{ij}|} \sum_{s_k \in CS_{ij}} sim_{ijk} + c_2$$  (7.8)

where $c_2$ is a constant number in order to assure that the common service calls are always considered. In the experiment conducted, a value of 0.2 is assigned to $c_2$. The similarity calculation will be run regularly. Depending on the frequency of invocation and query request, the user similarity matrix is updated frequently and it is computed offline. The sample user-user matrix is shown in the Table 7.2.

<table>
<thead>
<tr>
<th>User id</th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>U4</th>
<th>U5</th>
<th>U6</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>1</td>
<td>0.13</td>
<td>0.09</td>
<td>0.15</td>
<td>0.23</td>
<td>0</td>
</tr>
<tr>
<td>U2</td>
<td>0.13</td>
<td>1</td>
<td>0.35</td>
<td>0.26</td>
<td>0.21</td>
<td>0</td>
</tr>
<tr>
<td>U3</td>
<td>0.09</td>
<td>0.35</td>
<td>1</td>
<td>0.63</td>
<td>0.10</td>
<td>0</td>
</tr>
<tr>
<td>U4</td>
<td>0.15</td>
<td>0.26</td>
<td>0.63</td>
<td>1</td>
<td>0.1</td>
<td>0</td>
</tr>
<tr>
<td>U5</td>
<td>0.23</td>
<td>0.21</td>
<td>0.1</td>
<td>0.16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**7.2.1 Proposed Algorithm for Selection and Ranking**

After generating the user-user matrix, most similar user can be found easily and rank the matching services for the query.
Collaborative Filtering Ranking Based on Invocation History

In CF recommender system, usually the K number of most similar neighbors is determined with respect to the current user, and based on those K user’s earlier records, the active user’s attitude is predicted. The degree of match of a service to a particular user request is calculated using CF. The main idea of the proposed method is that if a service has been invoked several times, it has a higher chance to be selected in the current search session, and if several earlier service calls are from users having similar interests as the current user, then that service will be having a greater chance to be selected.

If a user $u_i$ is submitting a query which includes both functional and QoS requirements, the query is denoted as $(Q_f, Q_{qos})$, where $Q_f$ should be non-empty whereas $Q_{qos}$ may be empty. This is because occasionally a user may not be having QoS requirements or may not be knowing how to convey a QoS requirements. For functional requirement $Q_f$, the similar services are found by the agent of the centralized server selection and the set containing the similar services is denoted as $RS_f$. The proposed CF ranking algorithm primarily uses the set $RS_f$. For each service $s_k \in RS_f$ if the service has been invoked by some users, its score is based on CF is computed using Eqn. 7.9.

\[
S_{CF}(S_k) = \sum_{u_j \in S(u_i)} sim_{ij} \cdot \frac{1}{N} \cdot \sum_{h=1}^{t_j} \frac{t_j^h - t_s}{t_c - t_s} \cdot (SF(f q^h, Q_f) + c_3) \tag{7.9}
\]

where $S(u_i)$ is the set of similar users to user $u_i$ and it is denoted as $K$ and $N$ is the maximum number of calls on a service for one user. The value $sim_{ij}$ is obtained from the user similarity matrix. For each of the invocations on service $s_k$ from user $u_j$, $t_j^h$ denotes the time of the $h^{th}$ service call, $t_s$ denotes the starting time of the log and $t_c$ denotes the present system time. In spite of this uncertainty, the number of calls is computed by adding a constant value of $c_3$ (i.e. 0.1) to the query similarity value for the reason that the user-service matrix may be sparse. This computation is done considering the Top-K similar user’s service calls. Based on the Eqn. 7.9, the score value will be greater if the user similarity is high, if more recent call time is considered, if calls are based on similar queries and if there are more number of calls of this service. Fig. 7.2 shows how the collaborative filtering based score is calculated based on those data. In the Fig. 7.2, $RS_{fo}$ 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.
In this proposed work, in order to overcome the cold start problem for new users, the ranking score is calculated based on the call history. When \( IH_k \) is empty for all \( k = 1, \ldots, n \), meaning that the user \( u_i \) is a new user and has no calls to any services, or when \( sim_j \) is zero for all \( j = 1, \ldots, m, and j \neq i \), meaning that the user \( u_i \) has no similar users, then for every service \( s_k \in RS_f \), score of that is derived using Eqn. 7.10.

\[
S_{CF}(s_k) = \frac{1}{N} \times \sum_{j=1}^{m} \sum_{h=1}^{l_{jk}} \frac{t_{jk}^h - t_s}{t_c - t_s} + (SF(q_j^h, Q_f) + c_3)
\]  

(7.10)

Figure 7.2: Calculation of collaborative filtering-based score

In this proposed work, the cold start problem is overcome by sending a separate list consisting of newly published services which are ranked based on their publication date is presented to the current user which enables the new services to have an equal chance to be viewed and selected by users.

### 7.2.2 Overall Selection and Ranking Algorithm

The Fig. 7.3 shows the overall selection and ranking process and the steps used for service selection in this proposed work given below.
1. Given a query \((Q_f, Q_{qos})\) from user \(u_i\), \(RS_f\) has the functionally similar services. Then the services in \(RS_f\) are ranked using \(Q_{qos}\) values and the service call records.

2. If there is no \(Q_{qos}\) values from the user, go to step 3, else, go to step 4.

3. If there is no QoS requirement from the current user, the services are ranked by using the call history. As explained in the previous section, the score is calculated differently for new users and existing users. The final result presented to the user will be two separate ranked lists, one is a list of new services, \(RS_n\) 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 QoS requirements are given by the user, first the services not satisfying the QoS requirements are filtered. The remaining services will be in $R_{Shq}$ which is a subset of $R_S$. Then for all services in $R_{Shq}$, their QoS based ranking scores are calculated using the proposed approach discussed in chapter 5. Afterwards, their invocation history-based ranking scores are calculated using Eqns. 7.9 and 7.10. The final result will also be two ranked lists, one is a list of new services, $R_{Sn}$ ranked on their publication date, and the other is the rest of the services. For rest of services, they have both QoS-based and collaborative filtering-based scores. Finally, rank aggregation method such as Borda Fuse [113] is used to achieve a single ranked list.

Proposed selection system is rather generic, and any algorithm can be plugged in for the QoS score calculation and rank aggregation.

7.3 Efficiency of Proposed Algorithms

In this section, the complexity of the proposed ranking and user similarity algorithms are evaluated. The memory used by the proposed algorithms mainly depends on the number of users or services in this proposed system, and it can be a linear dependency. Hence, the time complexity alone is analysed. 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, the complexity using the pseudo codes are calculated based on the major loops that are related to the parameters such as the total number of users ($m$) and the total number of services ($n$).

Ranking Algorithm

The algorithm for service ranking is given Fig. 7.4 and the algorithm for computing the frequency is given in Fig. 7.5.

For the first step in service ranking algorithm, 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 Eqn. 7.9 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, the complexity of the proposed ranking algorithm is achieved as \( O(m*\log(m)) \).

```java
    double ranking (...) {
        ...
        find top k similar users {
            ...
            for each top similar user {
                Call the freqq method to get the invocation frequency and time value;
                Computing the ranking score for the services based on Eqn. 7.9;
            }
        }
        return the ranking score;
    }
```

Figure 7.4: Algorithm to compute service ranking

```java
    double freqq (...) {
        ...
        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;
    }
```

Figure 7.5: Algorithm to compute invocation frequency

In algorithm for computing the invocation frequency, the number of executions depends on the number of invocation instances used in the proposed algorithm, which is defined as \( N \). The number of executions in iterations 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 the algorithm is 
\[
q = (4+t)N + z
\]
where \( z \) represents the number of executions before the loop, which is a constant value.

**Similarity Algorithm**

Fig. 7.6 presents the pseudo code for computing the user similarity for every pair of users (U-U matrix) and saving the results in a file.

Fig. 7.7 gives the algorithm for computing the similarity between two users on each commonly invoked service and returning the results to the calling method. Fig. 7.8 shows the algorithm to compute the Kendall tau coefficient for user QoS preference similarity and returns the result to the calling method. Fig. 7.9 includes the algorithm to compute the concordant value. Fig. 7.10 presents the algorithm for computing the user value requirement similarity which is a Jaccard coefficient and returning the result to the calling method.

```c
void user_sim(...) {
    ...
    for (int i = 0; i < numUser; i++) {
        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 Eqn. 7.8; \[8\]
            }
            Write result to file; \[9\]
        }
    }
}
```
Figure 7.6: Algorithm to compute user similarity value

```java
void com_invoked (…)
{
    …
    if (user i and user j have history data)
    {
        …
        if (user i and j have commonly invoked services)
        {
            for (each service in the set of commonly invoked services)
            {
                …
                Computing similarity between user i and j for the service based on Eqn.
                7.2 or 7.3 and store the result in an array for the calling method;
                …
            }
        }
        …
    }
}
```

Figure 7.7: Algorithm to compute similarity between users $i$ and $j$ on each commonly invoked service

```java
double weight_sim (…)
{
    …
    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 Eqn. 7.6; [4]
    …
    Return tau;
}
```

Figure 7.8: Algorithm to compute weight similarity value
The number of executions for the first step in algorithm for computing concordant value (Fig. 7.9) 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 $2h \times \log(h)$. In the second step of the Algorithm, comparing these two lists using nested loops through h members takes $h \times h$ times. Therefore, the total amount of executions is given by $x_1 = 2h \times \log(h) + h \times h$. It is a constant value when the number of attributes of interest is pre-defined, which is the case in the proposed system.

The computation for step 1 and 2 in algorithm for computing the weight similarity (Fig. 7.8) 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. Assuming that multiple search methods are used, the execution time is less than or equal to $2h \times h$. Step 4, calculating the weight similarity value based on Eqn. 7.6 needs a small constant number of steps, i.e., $g$. So the total number of executions is in the order of: $x_2 = 2h \times h + x_1 + g$. Since $h$ is a pre-defined integer number and $x_1$ is a constant number as mentioned above, $x_2$ is a constant number.

![Figure 7.9: Algorithm to compute concordant value](image)

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

```c
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 Eqn. 7.7

} } Return the coefficient value;
}}

Figure 7.10: Algorithm to compute value similarity

The complexity level of algorithm for computing value similarity is similar to that of algorithm for computing the weight similarity (Fig. 7.8), which depends on the number of attributes users $i$ and $j$ are interested in for the invocation of the service. The execution time complexity is represented as $x_3$, which is also a constant value.

Fig. 7.7 gives the algorithm to compute the user similarity on each service and returns a list of similarity values for all services users $i$ and $j$ commonly invoked. Step 3 in this algorithm 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 Eqn. 7.2 or 7.3. Each iteration computes the similarity on each commonly invoked service for user $i$ and $j$. The number of executions is $N'N'x_2x_3$ where $N'$ is the number of invocations, each service of each user is used in the similarity computation, which is usually a pre-defined constant number. The overall execution time is in the order of $S*N'N'x_2x_3$, where $S$ is the number of commonly invoked services. In the worst case scenario when $S = n$, the complexity is $O(n)$.

Fig. 7.6 gives the algorithm to get 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 $m \times m$ iterations. Steps 2, 3, 4 and 5 take fixed steps of operation and they are denoted 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 the 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, it is seen 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.
7.4 Experiments and Evaluation

Experiments are conducted to evaluate the accuracy of the CF-based ranking algorithm and to test the effect of different parameter settings to the system performance.

Currently there is no standard dataset to evaluate the QoS-based WS selection systems, let alone for CF-based approaches. Since the proposed algorithm depends on the invocation and query logs to do the ranking, a model program is used for generating the service requests and invocation records and then the simulated dataset is used for the evaluation purpose. The dataset reported in [102] is considered in this work for conducting experiments at it contains real time data and it includes almost 2500 services.

This dataset consists of a name, uniform resource locator of its WSDL file, and the QoS values for 9 attributes for every service. By checking the name of each service and its WSDL file, 15 popular keywords are chosen as single-word functional queries such as “message”, “user”, “tourism” etc. Every query has a list of matching services, for instance, 52 services on topic “message”, 21 services on topic “user”, and 62 services on topic “tourism”. The numbers of services for all the 13 topics are shown in Table 7.3.

Implementation and Design

The proposed architecture for service selection is implemented in a simulator program using java under Windows platform, and it is used to create query and call requests. Functional and QoS requirements and the particular about the service selected and invoked from the list of

Table 7.3: Collection of services used in the experiments

<table>
<thead>
<tr>
<th>Number</th>
<th>Topic</th>
<th>Number of services</th>
<th>Number</th>
<th>Topic</th>
<th>Number of services</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Logistics</td>
<td>26</td>
<td>8</td>
<td>Communication</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>Finance</td>
<td>19</td>
<td>9</td>
<td>Weather</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Business</td>
<td>14</td>
<td>10</td>
<td>Travel</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>Marketing</td>
<td>23</td>
<td>11</td>
<td>Internet</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Location</td>
<td>107</td>
<td>12</td>
<td>Road</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Utilities</td>
<td>73</td>
<td>13</td>
<td>Miscellaneous</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>ecommerce</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
services returned as response is included as part of the query record. Sometime, the QoS particulars may not be available as part of a user query. Also there is a chance for no service call for a user query and some service calls are not because of query submitted by the user. In this experiment, keywords used for functional part are mentioned above, and reliability, availability and response time are used as QoS attributes which are included in the dataset. The priorities assigned for QoS attributes lies between 1 and 4 where 1 representing the highest priority and 4 for no QoS requirements. The requirements of QoS values are in accordance with the attribute data types and the values included in the considered dataset. In order to make the implemented program to be generic, a configuration file is created to store the various parameters values and thus different datasets can be generated by altering their values. The parameters considered in this work are given below.

- Three constant numbers c1, c2, c3 as shown in Eqns. 7.2, 7.7, 7.8 and 7.9.
- $K$ - Number of similar users considered.
- $N$ - Maximum number of calls for a service which is stored for every user.
- $NQ$ - Number of effective queries that a user might submit and its value ranges from 1 to 10. [A query is considered as effective if it is followed by a call]
- $NSI$ - Number of invocations of a service after submitting a query and this is a range value.
- $NII$ - Number of service calls without query and it is also a range value.
- $NU$ - Number of users using the system.

The queries and the call instances are distributed randomly among 12 months and the first 11 month’s data is considered as the training data which is used for calculating the user similarities whereas the last month’s data is considered as the testing data. The proposed CF algorithm is used to rank the functionally similar services for the queries submitted during the last month. The precision of the ranking algorithm to check whether the invoked service is in the top result list is calculated using the Eqn. 7.11.

$$P(TN) = \frac{1}{Q_{12}} \sum_{q \in Q_{12}} in_q(TN)$$  \quad (7.11)

where $Q_{12}$ represents the queries during 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 the proposed ranking algorithm, it
is 1, and otherwise 0. In the experiment conducted, the value of $TN$ is chosen as 5 and 10. The precision value for the Top 5 and Top 10 results is denoted as P-5 and P-10 respectively in the rest of the thesis.

**Experiment Results and Analyses**

Each experiment is run for three times and the average precision value is considered for evaluation. The values assigned for the parameters are shown in the Table 7.4.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NU</td>
<td>100</td>
</tr>
<tr>
<td>NQ</td>
<td>Between 1 and 10</td>
</tr>
<tr>
<td>NSI</td>
<td>Between 1 and 100</td>
</tr>
<tr>
<td>NII</td>
<td>Between 1 and 10</td>
</tr>
<tr>
<td>K</td>
<td>5</td>
</tr>
<tr>
<td>N</td>
<td>5</td>
</tr>
<tr>
<td>$c_1$</td>
<td>0.1</td>
</tr>
<tr>
<td>$c_2$</td>
<td>0.2</td>
</tr>
<tr>
<td>$c_3$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The value of one parameter is changed in each experiment to see the effect of the precision values. The values of $c_3$, $NQ$, $K$ and $N$ are changed in the first set of experiments. The Figs. 7.11, 7.12, 7.13 and 7.14 show the results of P-5 and P-10 when changing the parameters. The second sets of experiments are conducted with new values for NSI and NII to test their effect on the precision values and Figs. 7.15, and 7.16 show the observed result. Finally, an experiment is conducted in order to test the effect of value of $NU$ on the precision value and the result is included in Fig. 7.17. The percentage is calculated as:

$$\text{percentage} = \frac{\text{difference between two values}}{\text{average of two values}} * 100$$ (7.12)

**Changing $c_3$, $NQ$, $K$, and $N$**

From Figs. 7.11, 7.12 and 7.13, it is clear that the value of P-5 is more than 0.6 and the value for P-10 is around 0.8 which implies that the probability for users to select the desired services from the Top 5 or Top 10 results is high and this proves the effectiveness of the proposed CF-based ranking method. In the absence of the proposed ranking algorithm, the functionally matching
service approach has to be used by the users to select the service. For example, the users have to check more than 50 matching services for the functional query “communication” to select the service which they require.

Fig. 7.11 shows the result of P-5 and P-10 in which c3 is 0.1, 0.5, and 1 respectively. Fig. 7.12 shows the result of P-5 and P-10 with different NQ values per user. Figure 7.11 shows that the changes in the value of c3 have no much effect in the precision values; P-10 has only changed 18.18% when c3 changes from 0.1 to 0.5 which is the biggest change in results. Similar results for c1 and c2. Also the experiment results show that the values of K and N have no influence on the precision value. The biggest change on precision for K is 28.57% when K changes from 10 to 25, and changing of N does not change the precision based on the observed results. So in the later experiment N is assigned a value of 5 because the algorithm takes more time to run when the value for N is bigger. The precision value is increased when more number of queries is considered and it is a reasonable conclusion as RS is normally produces more accurate results when it deals with more data. Increasing NQ can significantly improve the precision, especially when NQ increases from 1 to 10 and 10 to 20, there is a 40% increase for P-5 and 28.57% increase for P-10. It is concluded that the more NQ is, the higher the precision value. But for a larger number of queries will increase the processing time and in order to balance the efficiency and accuracy, this number is set between 20 and 30 in the rest of the experiments.

**Changing c3**

![Figure 7.11: Precisions of P-5 and P-10 in which c3 is 0.1, 0.5 and 1](image-url)
Changing $NQ$

![Graph showing precision of P-5 and P-10 with different $NQ$ values per user.]

Figure 7.12: Precisions of P-5 and P-10 with different $NQ$ values per user

Changing $K$

![Graph showing precision of P-5 and P-10 in which $K$ is 5, 10, 15, 20 and 25.]

Figure 7.13: Precisions of P-5 and P-10 in which $K$ is 5, 10, 15, 20 and 25

Changing $N$

![Graph showing precision result of P-5 and P-10 in which $N$ is 5 and 10.]

Figure 7.14: Precision result of P-5 and P-10 in which $N$ is 5 and 10
If this number increases further, 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, set this number between 20 and 30 in the rest of the experiment.

In the second experiment, the precision values are tested by changing the $NSI$ and $NII$ values and the precision values are shown in Figs. 7.15 and 7.16. According to the Fig. 7.15, the precision value is not affected much when the value of $NSI$ is changed as this value is used only in the calculation of similarity, and in the ranking part, only a certain number of service call instances are considered. However, when the $NII$ value is higher, the precision decreases more than 20% on P-5 and more than 6% on P-10 which is shown in Fig. 7.16. The reason is that the proposed ranking algorithm greatly depends on user’s earlier query histories.

**Changing $NSI$**

![Figure 7.15: Precisions when changing $NSI$ values](image)

**Changing $NII$**

![Figure 7.16: Precisions when changing $NII$ values](image)
From Fig. 7.17, when the value of \( NU \) is changed, it has no influence on increasing the precision values. There is a relatively bigger decrease of 7% on P-5 when the \( NU \) changes from 15 to 25. This can be caused by larger \( NII \) values due to experiment setting which uses randomly produced numbers. Normally the RS yields better accuracy when the users are more. However, in this experiment setup, pre-defined patterns are followed when submitting the queries in order to assure that they are similar users. Owing to this, even when there are more users, the precision value does not change much.

**Changing \( NU \)**

![Figure 7.17: Precision when changing \( NU \) values](image)

From the observed results, it is concluded that the number of effective queries and the number of random invocations greatly affects the precision of proposed ranking algorithm. When the number of user queries is more, meaning that there are more usage data and when the number of random invocations is smaller, meaning that most of service calls are related to user queries, the accuracy of the recommendations will be better. When the proposed algorithm is applied to the real selection system and when there are more amounts of user data and there is a high chance of finding similar users, the accuracy of the service selection system will be increased.

Experiments are conducted to study the quality of the Top-K Web service recommendations done by the proposed approach. Two versions of an application are implemented using the architecture of proposed service selection system. First application (Application-A), using keyword matching approach to cluster WS and distance vector method for ranking WS. The second application (Application-B) using the proposed support-based
selection (TF-IDF) for semantically clustering WS and the proposed improved distance vector method for ranking WS.

Normally, the users tend to select only the top items from the returned result list and the item in the higher position in the returned result list, especially the first position, is more important than the items in lower positions. The qualities of Top-K results are measured using the Discounted Cumulative Gain (DCG) [142] approach. A large DCG value means high QoS utilities of the Top-K returned Web services. The DCG at a particular rank position \( p \) is computed using the Eqn. 7.13.

\[
DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}
\]  

(7.13)

where \( rel_i \) is the relevance grade (i.e.) the score value computed using the proposed Borda Fuse method explained in chapter 6 using the rank list based on the attributes’ QoS values and the rank list based on invocation history of the result at position \( p \). DCG places stronger emphasis on retrieving relevant documents. A larger DCG value tells that the Top-K returned WS is of good quality.

Figs. 7.18 and 7.19 shows the DCG values calculated using Eqn. 7.13 of Top-K (K=5 and K=10) recommended WS in both the applications. It is observed that the DCG values of Application-B are higher than Application-A which shows that the Top-K WS included in the recommended list generated by the Application-B includes the most appropriate WS based on the user’s requirements when compared to the Top-K WS included in the recommended list generated by Application-A. The five queries used for both the applications are Temperature conversion, Bank account, Currency conversion, Country zipcode and Company information.

![Figure 7.18: DCG values for Top 10 recommendations](image)
In this work, a new architecture for a hybrid recommender system is proposed. The proposed approach for service selection is implemented and evaluated and it is proved that the proposed approach works for different user invocation histories and different number of users. It is also seen from the experiments conducted 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 WS in a random pattern will benefit less from the recommendation.

Figure 7.19: DCG values for Top 5 recommendations

7.5 Chapter Summary

In this work, a new architecture for a hybrid recommender system is proposed. The proposed approach for service selection is implemented and evaluated and it is proved that the proposed approach works for different user invocation histories and different number of users. It is also seen from the experiments conducted 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 WS in a random pattern will benefit less from the recommendation.