Context-aware Service Input Ranking by Learning from Historical Information

Shaohua Wang, *Member, IEEE*, Ying Zou, *Member, IEEE*, Joanna Ng, and Tinny Ng

Abstract—Users visit on-line services and compose them to accomplish on-line tasks, such as shopping on-line. Quite often, users enter the same information into various on-line services to finish on-line tasks. However, repetitively typing the same information into web forms is a tedious job for users. In this paper, we propose a context-aware ranking framework to rank values for input parameters. We propose 6 categories of ranking features constructed from various types of information, such as user contexts and patterns of user inputs. Our framework adopts learning-to-rank (LtR) algorithms that consist of a set of machine learned models to automatically construct ranking models by integrating the ranking features. When a user enters a value to an input parameter, an interaction between the user input and the input parameter is established. Our framework learns information relevant to such interactions and ranks user inputs in different contexts. Through empirical studies on the real-world on-line services, we obtain the following main results: (1) Among the 8 state-of-the-art learning-to-rank models, RankBoost can outperform other LtR models on ranking user inputs for input parameters; (2) Our framework using IRSVM that performs the worst among the LtR models outperforms the two baseline conventional ranking models and Google Chrome Autofilling, an industrial tool, on ranking user inputs to input parameters; and (3) We observe that the textual information of user inputs and input parameters is the most influential factor on ranking user inputs. Among the various types of contextual data, user locations and time matter the most to the ranking of user inputs.

Index Terms—Information reuse, learning-to-rank, input parameter value recommendation, web forms.

1 Introduction

NOWADAYS, service community has proposed various types of services, such as cloud services, platform as a service, and on-line services using Service Oriented Architecture (SOA) principles. The above aforementioned types of services often have an interface demanding users' inputs. With the rapid advancement of Web and SOA technologies, millions of users can conduct various on-line tasks using on-line services designed using SOA principles to potentially compose ad-hoc processes repeatedly [1], such as shopping on-line. Typically, an on-line service using SOA takes user inputs through web forms and invoke underlying web services (e.g., RESTful services), such as *Priceline*[1] a travel-related product discount website.

Typically, a web form has a set of input parameters intaking values from users. A user input is a piece of information entered into web forms. To invoke a web service, every required input parameter should be filled with a proper value in web forms. Approximately 70 million professionals (*i.e.*, approximately 59% of all professionals in the United States) often need fill out on-line forms for their daily jobs [2]. However, users are often required to enter the same information into input parameters from varied on-line services repetitively [3]. For example, planning a trip from Toronto to Paris can be conducted by thousands of users. The trip planning can involve two on-line services: *Priceline* and *ticketmaster*[2], a ticket sales website. A user purchases an airplane ticket on *Priceline* and buys discount concert tickets on *ticketmaster* for his or her stay in Paris. Both on-line services could require users to provide, at least, a dozen of values (e.g., city name) using web forms and pass values to underlying RESTful services which use forms as user interfaces. Some values, such as user's name and payment information, are required by both services. Filling the same values into multiple web forms can be tedious [4], especially when the number of web forms is high. Therefore, it is essential to help users fill out web forms which take user's inputs for underlying web services and share user's inputs among different web services requiring the same inputs in order to reuse inputs to prevent users from repetitive typing [21]. For example, users can be four times faster on a smart phone when just correcting pre-filled form entries compared to entering the information from scratch [5].

To assist users in filling out web forms and share values among multiple web services requiring the same input values, a set of user inputs previously entered into web forms can be an excellent source for discovering and pre-filling a proper value for an input parameter [6]. When a user enters a value to an input parameter, an **Interaction** between the **U**ser **IN**put and the **I**nput **P**arameter (denoted as ***UIN-IP interaction***) is established. The information of such interactions is essential for analyzing and learning previous user input entry activities.

Recently, several approaches (*e.g.*, [3] [6] [7] [8] [9]) have been developed to aid users filling out on-line services using previous inputs. The research mainly proposes two types of techniques: (1) *Pre-filling user inputs to input parameters*

• *Shaohua Wang is with the Department of Informatics, New Jersey Insit
tute of Technology, New Jersey, USA. E-mail: davidsw@njit.edu*
• *Ying Zou is with the Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada. E-mail: ying.zou@queensu.ca*
• *Joanna Ng and Tinny Ng are with the IBM CAS Research, Markham, Ontario, Canada. E-mail: {jwng, tng}@ca.ibm.com*

1. *http://www.priceline.com/*
2. *http://www.ticketmaster.ca/*

before users start typing (*e.g.*, [7] [8] [6]); and (2) *Recommending values to input parameters for users* (*e.g.*, [3] [9]). However, the existing approaches mainly suffer the following limitations:

- **Limited exploitation of the information associated with previous user inputs.** Most existing approaches only store previous user inputs with an attached textual description (*e.g.*, a human-readable label in a web form) [7]. The attached textual description of a user input shows the type of the user input. For example, a user input "25 Union street, kingston" can be attached with "Address". In most cases, the attached textual description of a user input is not sufficient to describe the possible usages of the user input in different contexts. For example, a user may enter different values to an input parameter under distinct contexts (*e.g.*, different locations). When a set of user inputs are used together multiple times, a pattern of user inputs forms. There are two types of information not fully analyzed and utilized by most existing approaches in value auto-filling and recommendation: (1) *Patterns of user inputs*; and (2) *contextual information of interactions, such as time and user location*. Lack of the analysis on patterns and contextual information makes existing approaches unable to achieve a high accuracy and efficiency in filling out on-line services.
- **Limited ability of learning user input entry activities.** Different types of information, such as user contexts and patterns of user inputs, related to user input entry activities can affect the accuracy of recommending previous user inputs for filling out on-line services. With the accumulation of the information related to user input entry activities over time, the importance of a type of information can change as user contexts could change dramatically. Furthermore, users may enter different user inputs for an input parameter. Failing to reflect the evolution of information or detect multiple user inputs for input parameters could lead to a decrease of performance in ranking. However, existing approaches cannot automatically adjust the importance of a type of information and detect the changes of user inputs for input parameters.

To address the aforementioned limitations, we propose a ranking-based framework that ranks a list of previous user inputs for an input parameter to save a user from unnecessary data entries. As a complement to our prior work in TSC [21] taking filled values in web forms and sharing them among web services to reduce the amount of user inputs, this work focuses on ranking values for input parameters. More specifically, our framework (1) collects and stores interactions between user inputs and input parameters along with the contextual information, then (2) learns and analyzes such interactions to rank user inputs for input parameters in different contexts. We propose 6 categories of ranking features derived from various types of information that can affect the ranking of previous user inputs, such as user contexts and user's past activities. To learn and analyze interactions between user inputs and input parameters, we adopt the framework of learning-to-rank (LtR) [10] [11] that consists of a set of supervised machine learning approaches to automatically build ranking models from the training data built from user input entry activities. We leverage learning-to-rank algorithms initially designed for the research field of information retrieval [11] for ranking user inputs.

In our prior work [12], we proposed a learning-to-rank based approach that pre-fills and recommends previous user inputs to users. We proposed 5 ranking features for our proposed ranking approach. In this paper, we extend the earlier work [12] in the following aspects:

1) We re-build the original 5 ranking features and expand our set of ranking features into 6 categories of ranking features. The added ranking features capture more aspects of interactions between user inputs and input parameters. Furthermore, we conduct more empirical experiments to discover the most influential ranking features on the ranking of user inputs.
2) In our earlier version [12], we only used RankSVM, a pairwise learning-to-rank model, in our framework. In this paper, we compare 8 learning-to-rank (LtR) models on the performance of ranking previous user inputs in order to find the most suitable ranking algorithms. We also test each LtR model with different variable settings to identify the best setting of the model.
3) We conduct an additional empirical study to compare our framework with Google Chrome auto-filing tool qualitatively and quantitatively. The empirical results show that our framework adopting any LtR model is more effective than Google Chrome auto-filing tool in ranking previous user inputs, and have qualitative advantages over Google Chrome auto-filling tool.

Paper organization. Section 2 presents the background of this study. Section 3 introduces our proposed approach. Section 4 discusses the case studies. Section 5 describes the threats to validity. Section 6 summarizes the related literature. Section 7 concludes the paper and outlines the future work.

2 Background

In this section, we first introduce the services that our approach aims to work on. Then we present the terminology of learning-to-rank algorithms.

On-line Services are utilized by users to conduct various tasks. In this paper, we focus on optimizing the filling of web forms taking values as input to underlying Web services. As a consequence, we provide more accurate values to be propagated among web services that require the same input values. Additionally, we test the applicability of our approach on Web services directly, since the shareholders, such as business process or prototype designers, can work with Web services directly and fill out the composite services [13].

Web Services can be classified into two types:

(1) *Simple Object Access Protocol (SOAP) based services* use *Web Services Description language* (WSDL) [14], an XML-based description language, to describe the functionality of web services. WSDL is often used with SOAP.

(2) *RESTful services [15]* are proposed to simplify the development, deployment and invocation of services. RESTful services use standard HTTP protocols and permit various data formats. Web Application Description Language (WADL) [16] is an XML description of HTTP-based web applications. WADL is the REST equivalent of WSDL services.

Quite often, the SOAP-based and RESTful services are treated as building blocks for building web applications. Moreover, SOAP-based and RESTful services do not have user interfaces for users. Web forms are used to serve as interfaces to capture user inputs.

Learning-to-Rank. Learning-to-rank (LtR) has been employed in a wide variety of applications in Information Retrieval [17]. We introduce the LtR using the task of document retrieval as an example.

A LtR task has training and testing phrases. In the training phrase, given a set of queries $Q = \{q^1, q^2, \dots q^m\}$, where m is the number of queries, and a collection of documents. Each query q^i ($1 \leq i \leq m$) is mapped to a list of documents $D^i = \{d^i_1, d^i_2, \dots, d^i_n\}$, where d^i_j denotes the j^{th} document. Each list of documents D^i is mapped to a list of relevance judgments $Y^i = \{y^i_1, y^i_2, \dots, y^i_n\}$, where y^i_j is the relevance judgment on document d^i_j with respect to query q^i. Therefore, each query-document pair (q^i, d^i_j) has a y^i_j. Relevance judgments are usually conducted at five levels, for example, perfect, excellent, good, fair, and bad. The higher grade a document has, the more relevant the document is. For each query-document pair, a feature vector is created. Features are defined as functions of a query-document pair. The learning task is to automatically learn a function $F(x)$ given a training dataset. The feature vectors are ranked according to $F(x)$, then the top K results are evaluated using their corresponding relevance judgments.

The process of making relevance judgments can be conducted manually or learned by user implicit feedbacks (e.g., clickthrough data in web search) [18]. By automatically collecting user data and inferring relevance judgments, LtR approaches can be modified and developed into settings where no training data is available before deployment [19]. Compared with the conventional ranking models (*e.g.*, Bayesian Belief Networks [20] based ranking model), LtR models automatically learn and combine different factors affecting a ranking to suggest the ideal ranked list.

The learning-to-rank algorithms can be categorized into three groups based on the way of creating training data: *(1) Pointwise approaches* assign a numerical or ordinal score to each query-document pair in the training data. The training process can be viewed as an ordinal classification or regression problem. *(2) Pairwise approaches* approximate the learning-to-rank problem as a classification problem, learning a binary classifier to determine which document is better in a given pair of documents. *(3) Listwise approaches* take ranked lists of objects as instances in learning and learn the ranking model.

3 Overview of Our Ranking Approach

In this paper, we propose a ranking approach that learns and analyzes user contexts and user history to recommend values for input parameters of services. Our approach consists of two major steps:

1) *Collecting and storing user inputs with contextual information*. It is critical to learn from previous user input entry activities in order to provide accurate rankings of previous user inputs for users. We collect interactions between user inputs and input parameters (*UIN-IP interactions*) through users' web activities. To store and organize *UIN-IP interactions*, we propose a meta-data model that captures textual information and contexts of *UIN-IP interactions*.
2) *Ranking values for filling in input parameters*. When a user needs provide a value to an input parameter of a service, a list of possible user inputs should be ranked to aid users in selecting the most suitable value for the input parameter. We propose a learning-to-rank based framework using 6 categories of ranking features constructed from user previous inputing activities to rank the user inputs.

3.1 Collecting and Storing User Inputs with Contextual Information

In this subsection, we first present our meta-data model for storing interactions between user inputs and input parameters (UIN-IP interactions). Second, we introduce our process of collecting UIN-IP interactions and user contexts.

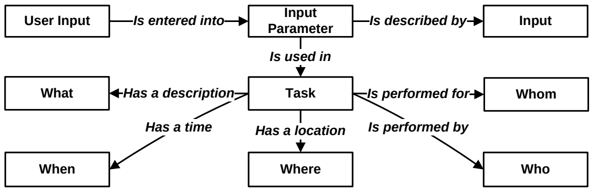


Fig. 1: Ontology for describing the semantics of our context-aware meta-data model

3.1.1 A Meta-data Model for Storing User Contexts

To help users fill in input parameters in different contexts, we need a mechanism to organize and store user inputs efficiently. We extend our prior meta-data model proposed in [21] to include the contextual information of interactions between user inputs and input parameters, in addition to the basic information defined in [21] (*e.g.*, a textual human-readable label in a web form). The description of our meta-data model is illustrated in Figure 1.

User Input: is a value entered into an *Input Parameter*. An *Input Parameter* has two properties: *Task* and *Input* property.

Task Property: has five sub-properties to store textual description and contextual information of a task where the input parameter is entered with a user input. We take different strategies to extract task information for each type of software artifacts. For web forms, we define a web form as a task. For WSDL services, we view an operation as a task. For RESTful services, we consider a resource with its associated actions as a task. The five sub-properties are listed as follows:

- ***What*** stores the information of what a task is. This property has three sub-properties:
  - ***Name*** records the name of a task, such as a web label (*e.g.*, from a web form or HTML tag), the name of an operation in a WSDL file, and the resource name and its associated actions of a RESTful service.
  - ***Text*** stores the description of a task. A task description is retrieved from the textual information of a web form, the description of an operation in WSDL, or the web page introducing a resource and its associated actions in a RESTful service. Since the description of a task

may not always be available, the property may have no value. We assign NULL for the property if it has no value. For example, the description of an operation in WSDL may not be available all the time.
  - ***Service*** records the name of a service where the task is performed. The value of this property can be the URL of a web form, the name of a WSDL or RESTful service.
- ***When*** records the time when a task is performed by a user.
- ***Where*** remembers the physical locations and computing devices of a user where he or she performs a task. The physical locations can be retrieved and calculated from IP, WiFi access points or GPS (*e.g.*, Mobile devices).
- ***Who*** stores the identity of a user who performs the task. The way of obtaining the identity is dependent on the implementation choices. For example, the identity of a user can be a device's Media Access Control (MAC) address or obtained from the user profile on social media.
- ***Whom*** records the information whom a task is performed for. The value of ***Whom*** can be equal to the one of ***Who***. For example, a user buys a phone for himself or herself.

Input Property: stores the textual information of an input parameter, and has three sub-properties: *label*, *name*, and *id*. The textual information is mined from coding information. In web applications, *label*, *name*, and *id* are the attributes of the HTML DOM elements defining the input fields consuming user inputs. In the context of WSDL and RESTful services, we store the name of the input parameter in the *label* property.

For each property stored in the context-aware metedata model, we conduct word normalization to identify meaningful words. We decompose any possible compound words (*e.g.*, BookHotel) following the conventions used in programming languages. We use four rules to decompose words: case change (*e.g.*, BookFlight), suffix containing a numeric number (*e.g.*, Flight1), underscore separator (*e.g.*, departure_city) and dash separator (*e.g.*, Book-flight). We use WordNet[3] a lexical database to remove non-English words. We remove stop words using a pre-defined list[4] of stop words, such as "the". Finally, we use porter stemmer [22] to reduce derived words to their stem, base, or root form. For example, "reserves" and "reserved" are mapped to the stem form "reserve".

3.1.2 Collecting user inputs and their contexts

We collect user input entry activities through users' web activities (*i.e.*, web forms). For example, users can shop a pair of shoes on Ebay[5] or register a course by filling in university registration web forms. Users can operate a simple RESTful web service testing using its URL through web browsers, or a SOAP-based WSDL service through SOAPUI framework[6]. User inputs can be collected through such a massive amount of user activities. We modify the tool used in [23] to collect interactions between user inputs and input parameters. The original tool in [23] extends a web testing tool named Sahi [7] to monitor users' web activities and collect the user inputs. Sahi embeds Javascripts into web pages and monitors user actions (*e.g.*, click, submit). Our original tool is a cross-platform and standalone application. Users run the tool and surf the web through web browsers. We extend the original tool to include contextual information, such as time and location, of the interactions between user inputs and input parameters. We store user inputs with their property information in the proposed context-aware metadata model.

Collecting contexts of interactions between user inputs and input parameters. To aid users in filling values to input parameters under different circumstances, it is essential to understand the dynamicity of users' needs. The definitions of user contexts are varied from different academic studies. We adopt the definition and taxonomy of user contexts in [24]. We collect four types of contexts: time, location, identity and activity, for interactions between user inputs and input parameters. We collect context values at different levels of granularity for each context type. Table 1 shows the contexts of each context type.

When an interaction of a user input and an input parameter occurs, we take the following steps to obtain user contexts:

Time. We collect the user's operating system time and parse it to extract user contexts for the context type *time*.

Location. Collecting accurate user Geo-location data is a difficult task. We automatically collect user's current location from web browsers using Google Geo-location Maps Javascript API[8] and parse it into contexts at different levels of granularity. To know the type and name of a current user location, we allow a user to label the location. A context type or name may have no value if a user does not enter a value. We assign NULL to the type and name without a value. In addition to the physical locations, we automatically detect the type of computing device a user uses.

Identity. We automatically detect the Media Access Control (MAC) address of a user's device. We allow users to manually enter their names to be attached with the MAC addresses. We also rely on the manual input from the user to obtain the context value for "whom". If a user does not enter a name for "who" and "whom", we use a MAC address for "who" and assign null to "whom".

Activity. The context value of user's current typing captures the case of user typing, when a user starts typing. Before a user finishes typing his or her intended value into an input parameter, the unfinished typing could be a perfect indicator for discovering the intended value. During the course of value typing, our framework retrieves user inputs containing user's current typing for an input parameter, then dynamically recommends a set of values based on what users are typing into input parameters.

We store UIN-IP interactions of a user in JavaScript Object Notation (JSON[9]). Each UIN-IP interaction with related context values is organized and stored in a Meta-data Model that is implemented as a JSON object.

3.2 Recommending Values to Users for Filling in Input Parameters

We store all previous user inputs in the proposed context-aware meta-data model (Section 3.1.1). To help a user fill in

3. http://wordnet.princeton.edu/
4. http://www.ranks.nl/stopwords
5. http://www.ebay.com
6. http://www.soapui.org/
7. http://sahi.co.in/
8. https://developers.google.com/maps/documentation/javascript/examples/map-geolocation
9. http://www.json.org/

TABLE 1: User contexts collected for each context type.

Context Type	User Contexts	Example
Time	year, month, day of the month, day of the week, hour in am or pm (5 contexts)	2015,Dec,1,Tues,10 am
Location	country, province or state, city or county, street, name of the location, types of location: home or work, ip address, types of devices used (8 contexts)	Canada, Ontario, Markham, Warden street, IBM, Work, 192.1.1.36, mobile
Identity	who enters a value for whom, Media Access Control (MAC) address of a user's device (3 contexts)	Shaohua Wang, Ying Zou, 00-0C-29-9C
Activity	user's current typing (1 context)	for example, typing "boo"

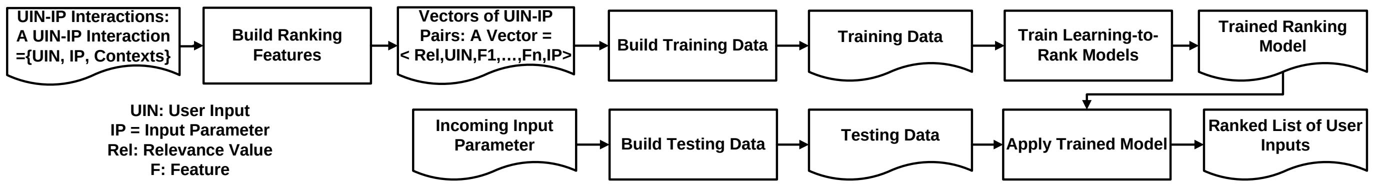


Fig. 2: Main steps of our ranking framework.

an input parameter, we propose a learning-to-rank model based framework to incorporate user contexts and history as features to recommend a personalized list of previous user inputs to users. Figure 2 shows the main steps of our ranking framework for ranking previous user inputs for a target input parameter (T-IP). First, we build a vector of ranking features for each pair of user input and input parameter in the stored UIN-IP interactions. Second, we create a training dataset based on the UIN-IP pairs with feature vectors. The training dataset is used by learning-to-rank models to generate trained ranking models. Third, we create testing data for the T-IP by building feature vectors for the T-IP and its previous user inputs. Last, we apply the trained ranking model on the testing data to rank user inputs for the T-IP.

3.2.1 Scenarios of Users Interacting with Input Parameters

We summarize four main scenarios of how users interact with input parameters of services with the help of our approach. When a user tries to enter a value to an input parameter, four scenarios can occur:

Scenario 1. A value is pre-filled to the input parameter when no typing action is performed from the user.

Scenario 2. When the user is not satisfied with the pre-filled value in *Scenario 1*, the user would remove the pre-filled value. Before the typing starts, a list of ranked previous user inputs are recommended to the user. This scenario captures the situation when there is no typing action performed by users.

Scenario 3. If no value is selected by the user from the recommended list of values in *Scenario 2*, the user starts typing a new value into the input parameter. A list of ranked previous user inputs are recommended dynamically based on what the user is typing. For example, we dynamically recommend values starting with a "m" if the user is typing a "m" in the input parameter.

Scenario 4. If no previous user inputs are suitable for recommendation or the user does not select any value recommended in the above three scenarios. The user has to finish typing a new value into the input parameter.

Our proposed context-aware ranking approach helps users in *Scenario 1*, *Scenario 2* and *Scenario 3*. Especially, we aim to maximize the help for users in *Scenario 1* and *Scenario 2* that do not require any typing from users.

3.2.2 Our Learning-to-Reuse Model

To learn from user input entry activities for reusing user inputs (learning-to-Reuse), we adopt learning-to-rank algorithms in the task of ranking user inputs. We view a previous user input (UIN) and an input parameter (IP) as a document and a query respectively, denoted as UIN-IP pair. We build ranking features and calculate feature values for each UIN-IP pair. The value of a relevance judgment on a user input UIN with respect to an input parameter IP (UIN-IP pair) can only be 0 or 1, where 1 means the UIN is perfect for the IP and 0 means the opposite. During the process of creating training dataset for LtR models, we create a set of vectors modeling relations between user inputs and input parameters. Each vector can be modeled as V = $< Relevance, UIN, feature_1, \ldots, feature_N, IP >$, where N is the total number of ranking features.

3.2.3 Building Ranking Features

To learn from user input entry activities, we propose 6 categories of ranking features that capture 6 different perspectives of interactions between user inputs and input parameters. Table 2 outlines our proposed ranking features.

Textual-similarity-based Features. A previous user input that is textually similar to an input parameter could be the right value and selected by a user.

Given a user input and an input parameter, we traverse all *input parameter* properties of the user input. For each *input parameter* property (denoted as IPP), we calculate the textual similarity between the *input* property of IPP and the *input* property of the input parameter. We obtain a set of textual similarity values between the *input parameter* properties of user inputs and the input parameter. We choose the highest value as the textual similarity between the user input and the input parameter.

Both the *input* property of an IPP and the textual information of the input parameter can have more than one word (*e.g.*, "book flight"). We formulate them into two sets of words. We adopt three algorithms: cosine similarity [25], Jaccard's coefficient [26], and WordNet-based approach (WNA) [21], to calculate the textual similarity between two sets of words.

Given two sets of words, set_i and set_j $(i \neq j)$,

(1) Cosine similarity. We merge two sets of words in a new set, set_k. We count the number of times a word in set_k

TABLE 2: Outline of our ranking features. UIN: user input. IP: input parameter. t: text. k: task. F: Feature.

Category	Rationale and Explanation	Features
Textual similarity	A UIN textually similar to an IP could be the right value and selected by a user. We adopt 3 approaches, cosine similarity [25], Jaccard's coefficient [26], and WordNet-based approach (WNA) [21] to calculate the textual similarity between the text of UIN and the text of IP.	F1=Cosine(UIN_t,IP_t); F2=Jaccard(UIN_t,IP_t); F3=WNA(UIN_t,IP_t)
Task similarity	If a task of a UIN is textually similar to a task of an IP that is being performed by a user, the UIN could be selected by the user. Like above, we adopt the same 3 approaches to calculate the similarity between the task of UIN and the task of IP.	F4=Cosine(UIN_k,IP_k); F5=Jaccard(UIN_k,IP_k); F6=WNA(UIN_k,IP_k)
Frequency	More frequent a UIN is used for an IP or multiple IPs, more chances the UIN will be used next time. We calculate two frequencies of a UIN: (1) # of times the UIN used for a particular IP (denoted as Freq_par); (2) # of times the UIN used for all IPs (denoted as Freq_all)	F7=Freq_par; F8=Freq_all
Time length	The recently used UINs could be used again while the oldest UINs may not be used. We calculate the length of time from current time to the time of last use of a UIN (denoted as T_Length).	F9=T_Length
Patterns	When a set of UINs or IPs occur together multiple times, a pattern is formed. Patterns can improve recommending. For example, a set of two UINs are always entered together. When one of them is used, the other one could also be used. Each UIN has unique id in our dataset. Among all UINs, we discover the UINs having a pattern. We use # of such UINs as the number of features under this category. For a UIN, we calculate the number of times the UIN is used with each of discovered UINs having a pattern.	Fi=# of times used with a UIN having a pattern. n is the number of UINs having a pattern, $10 \leq i \leq (n+10)$.
Contexts	Contexts could affect the ranking of user inputs. The number of contexts-based ranking features is the number of contexts (denoted as C^n, n is the number of contexts.) associated with UIN-IP interactions. For a context, we label unique values of the context with consecutive integer numbers. The value of the ranking feature for a context is the labeled integer number of a unique value of the context.	Fj = = the labeled number of a unique value of the j^{th} context in C^n, $i \leq j \leq n$

appear in both sets, set_i and set_j so that set_i and set_j can be represented as vectors v^{set_i} and v^{set_j}, . We use Equation (1) to calculate a cosine similarity between set_i and set_j.

$$Cosine(set_i, set_j) = \frac{\sum_{m=1}^{n} v_m^{set_i} v_m^{set_j}}{\sqrt{\sum_{m=1}^{n} (v_m^{set_i})^2}\sqrt{\sum_{m=1}^{n} (v_m^{set_j})^2}} \quad (1)$$

where n is the number of words in set_k, $v_m^{set_i}$ and $v_m^{set_j}$ are components of vector v^{set_i} and v^{set_j} respectively.

(2) Jaccard's coefficient. We use Equation (2) to calculate the textual similarity between set_i and set_j.

$$Jaccard(set_i, set_j) = \frac{|set_i \cap set_j|}{|set_i \cup set_j|} \quad (2)$$

where $|set_i \cap set_j|$ is # of same elements of two sets, $|set_i \cup set_j|$ is # of unique elements of two sets.

(3) WordNet-based. We adopt a WordNet-based approach (WNA) [21] to calculate the semantic similarity between set_i and set_j in the following steps: If $set_i \subset set_j$, we use Equation (2) to calculate the textual similarity between set_i and set_j. Otherwise, we count the number of common words in the two sets (denoted as $|set_i \bigcap set_j|$). We calculate the semantic similarity between every pair of words W_a^i ($W_a^i \subset set_i$) and W_b^j ($W_b^j \subset set_j$) of two sets after removing the same words from calculation, denoted as *# of calculation*. Then, we add up all the similarity values of pairs of words, denoted as $Sim_{sum} = \sum_{W_a^i \subset set_i} \sum_{W_b^j \subset set_j}$ sim(W_a^i, W_b^j), where sim(W_a^i, W_b^j) denotes the similarity value between two words; $W_a^i \neq W_b^j$. Last, we use Equation (3) to calculate the similarity between two sets. The numerator of the equation is to calculate the similarity between each pair of words of two bags; the denominator is the total number of times of calculating the similarity between words.

$$WNA(set_i, set_j) = \frac{|set_i \bigcap set_j| + Sim_{sum}}{|set_i \bigcap set_j| + |\# of\ calculation|} \quad (3)$$

where Sim_{sum} is the sum of the similar value of every pair of words excluding the common words from set_i and set_j. We propose three ranking features based on approaches used to calculate the textual similarity.

Textual-similarity-based Features: To calculate the textual similarity between a user input (UIN) and an input parameter (IP),

- **Feature 1** uses Cosine similarity, denoted as F1 = Cosine(UIN_{text}, IP_{text}).
- **Feature 2** uses Jaccard similarity, denoted as F2 = Jaccard(UIN_{text}, IP_{text}).
- **Feature 3** uses our WordNet approach (WNA), denoted as F3 = WNA(UIN_{text}, IP_{text}).

Task-similarity-based Features. If one of the tasks associated with a user input can match the current task being performed by a user, the user input could be selected by the user. Given a set of *input parameter* properties (IPPs) of a user input and an input parameter, we calculate the textual similarity between the *task* property of each IPP and the *task* property of the input parameter.

A *task* property has a *what* property describing the information of a task. The *what* property has three sub-properties: *name* storing task name, *text* saving task description and *service* keeping the name of a service where the task is performed as proposed in Section 3.1.1. We extract the task name, the task description and the service name for each IPP and the input parameter. We calculate the following three similarities between an IPP of the user input (denoted as IPP_{ui}) and the input parameter (IP):

(1) textual similarity between the task names of IPP_{ui} and IP, denoted as $sim^{name}(IPP_{ui}, IP)$;

(2) textual similarity between the task description of IPP_{ui} and IP, denoted as $sim^{des}(IPP_{ui}, IP)$;

(3) textual similarity between the service names of IPP_{ui} and IP, denoted as $sim^{service}(IPP_{ui}, IP)$.

We use Equation 4 to calculate the task similarity between an IPP of a user input and an input parameter using a similarity algorithm.

$$sim\_task(IPP_{ui}, IP) = w_n * sim^{name}(IPP_{ui}, IP) + w_d * sim^{des}(IPP_{ui}, IP) + w_s * sim^{service}(IPP_{ui}, IP) \quad (4)$$

where sim_task() can use the Cosine, Jaccard, or WordNet similarity algorithm. w_n*,* w_d*,* w_s *are weights for similarity calculation.* w_n*,* w_d*, and* w_s *have a same value (i.e.,* $1/3$*) for simplicity.*

Since a user input can have a set of *input parameter* properties, we obtain a set of similarity values. We choose the highest value as the task similarity value between a user input and an input parameter.

Similar to the textual similarity features, we adopt Cosine similarity, Jaccard coefficient, and WordNet based approach to calculate the textual similarity between two sets of words. We propose three ranking features based on the approaches used for calculating the textual similarity.

Task-similarity-based Features: To calculate the task similarity between a user input (UIN) and an input parameter (IP),

- **Feature 4** uses Cosine similarity, denoted as F4 = Cosine(UIN_{task}, IP_{task}).
- **Feature 5** uses Jaccard similarity, denoted as F5 = Jaccard(UIN_{task}, IP_{task}).
- **Feature 6** uses our WordNet approach (WNA), denoted as F6 = WNA(UIN_{task}, IP_{task}).

Frequency-based Features. When a user input is used more frequently than others, it is very likely that the user input will be consumed again.

Given a user input, an input parameter, and a set of stored interactions between user inputs and input parameters (UIN-IP interactions), we analyze the UIN-IP interactions and calculate: (1) the number of times that the user input is provided to the input parameter historically (denoted as Freq_par); (2) the total number of times that the user input is used for all input parameters in UIN-IP interactions (denoted as Freq_all). Based on two types of frequencies, we build two ranking features.

Frequency-based Features: Given a user input (UIN) and an input parameter (IP), **Feature 7** = Freq_par; **Feature 8** = Freq_all.

Time-length-based Features. A user input can decay and is not likely to be used. However, the most recently used user inputs could be use-prone compared with old user inputs.

Given a set of stored UIN-IP interactions and a user input, we analyze the UIN-IP interactions and calculate the length of time from a time point when an input parameter requires a value to the time point when the user input was most recently used (denoted as T_Length). Based on the time length, we build one ranking feature.

Time-length-based Feature: Given a user input, **Feature 9** = T_Length.

Pattern-based Features. User inputs (UINs) can be used together to accomplish a task. When patterns of UINs are formed, the recommendation can be improved. For example, a set of two UINs are always entered together. When one of them is used, the other one could also be used. A task can have several input parameters. For example, a web form can have multiple input fields. To execute a task, all input parameters of the task should be filled with values and run at the same time. We consider a set of UINs as a pattern if the set of UINs are used together more than twice.

Given a set of stored UIN-IP interactions and a target user input (T-UIN), we first discover patterns of stored UINs. We group the stored UINs based on the number of times the UINs are used together for tasks. Second, we mine sets of UINs occur more than twice across different groups of UINs, denoted as $patterns^{uin}$. Third, we obtain a set of unique user inputs that have a pattern, denoted as set_p^{uin}. Last, for the T-UIN, we calculate the number of times that the T-UIN is used together with every $UIN_i \subset set_p^{uin}$. When UIN = UIN_i, we assign the number of times to 0. To utilize every possible pattern, we build a ranking feature based on the number of times every pair of UIN and UIN_i are in a pattern, denoted as # of $Patterns^{uin_i}$. In total, we build K pattern ranking features, where K is the number of unique user inputs that are, at least, in a pattern.

Pattern-based Features: Given a user input and UIN-IP interactions, we first discover a set of user inputs that are, at least, in a pattern, denoted as set_p^{uin}. We build K pattern ranking features, where K = # of unique user inputs that are, at least, used in a pattern; **Feature f#** = # of $Patterns^{uin_i}$, f# is feature number, $10 \leq f\# \leq the\ size\ of\ set\ set_p^{uin} + 10$.

Contexts-based Features. The contexts associated with interactions between user inputs and input parameters are important for recommendation. Each interaction between a user input and an input parameter (*i.e.*, UIN-IP interaction) is associated with a set of contexts recorded during the execution. Given an input parameter and a set of user contexts ($set^{Contexts}$), we identify user inputs associated with contexts matching the $set^{Contexts}$. For example, a context can be a location or device where a user enters a user input to an on-line service. In this paper, we collect 17 contexts as described in Section 3.1.2. One of them is *user's current typing*, to boost the performance of our approach, we simply use it as a filter rule to dynamically retrieve candidate user inputs for an input parameter in runtime. In this paper, all of the user inputs are collected from subjects in Canada. Therefore, we build ranking features based on 15 contexts, excluding two contexts *user's current typing* and *country*.

Given a set of UIN-IP interactions $UP_{set} = \{up_1, up_2, \ldots, up_N\}$ where N is the number of UIN-IP interactions and UP_{set} is sorted by timestamps in a chronological order, and a UIN-IP interaction ($up_i, 1 \leq i \leq N$) has a set of m contexts $C = \{c_1, c_2, \ldots, c_m\}$, we collect all the values of every context in C and keep the unique ones. Then we label the unique values using consecutive integer numbers from 1 to M (*i.e.*, M is the number of unique values of the context). In the end, each context has a set of unique values that are labeled with integer numbers. Each unique value is mapped to an integer number.

Given a target UIN-IP pair with a set of values of m contexts C^t, we build a contexts-based ranking feature for a context in C^t. We verify whether the value of the context in C^t is already labeled in the set of unique values of the context. The value of the ranking feature for the context is *the labeled integer number* if yes, or *the number of existing unique values + 1* if no. In total, for a UIN-IP pair, we build 15 contexts-based ranking features.

For example, we have three UIN-IP interactions, each interaction has 3 contexts: city, month, and who. The three interactions can be represented as: $up_1 = \{UIN_1, IP_1, Markham, Jan, Shaohua\}$, $up_2 = \{UIN_2, IP_2, Markham, Feb, Ying\}$, and $up_3 = \{UIN_3, IP_3, Markham, Feb, Ying\}$. The up_1 and up_2 occur before the up_3. We use the up_1 and up_2 as the previous interactions, and the up_3 is the target UIN-IP pair. The three contexts (*i.e.*, city, month, and who) have only one, two, and two values in the previous interactions, respectively. Therefore, the city Markham is labeled with 1. The month Jan and Feb are labeled with 1 and 2, respectively. The names, Shaohua and Ying, are labeled with 1 and 2, respectively. Therefore, the values of the contexts-based ranking features for I_3 is {1, 2, 2}.

Contexts-based Features: Given a set of UIN-IP interactions and each UIN-IP interaction with n contexts, we build a ranking feature for each context. For a context, we label unique values of the context with consecutive integer numbers. The value of the ranking feature for a context is the labeled integer number of a unique value of the context. **Feature j** = the labeled number of a unique value of the j^{th} context, $i \leq j \leq n$.

3.2.4 *Building Training and Testing Data for Learning-to-Rank Models*

To have a proper ranking of user inputs, we need automatically build scenarios containing positive and negative examples for input parameters to train a learning-to-rank model to rank user inputs responding to contexts. Over the time, a user can enter multiple user inputs to an input parameter in different contexts. Each input parameter can have several scenarios. A scenario can only have one positive example that shows a value entered into the input parameter under a set of contexts C, but can have several negative examples showing a set of values which are not chosen for the input parameter under contexts C.

Given a user input (UIN) and an input parameter (IP) that are in a UIN-IP interaction with a set of contexts C, and a set of other user inputs (denoted as Set^{others}), we build a scenario in the follow steps:

(1) *Creating a positive example*. Since the UIN-IP interaction shows that the user input (UIN) is entered into the input parameter (IP), the relevance value for the pair of UIN and IP is 1. After the ranking features are built for the UIN-IP interaction, the UIN-IP interaction can be represented as a vector $V =< Relevance, UIN, Feature_1, \ldots Feature_N, IP >$. The value of Relevance is 1.

(2) *Building negative examples for the given UIN-IP interaction*. For each user input in Set^{others}, denoted as UIN^o, we form a pair of UIN^{others} and the IP, denoted as UIN^o-IP pair. We assign 0 to the relevance value of UIN^o-IP pair, because the UIN^o is not chosen for the IP under the contexts C. Then, we build a feature vector with a relevance value 0 for each pair of UIN^o and the IP.

We feed the created scenarios to a learning-to-rank model so that a ranking model can be trained and saved. The trained and saved model is used to rank user inputs for an input parameter that requires a value.

Using a trained ranking model to rank user inputs for an input parameter under a set of contexts C, we build a vector, with Relevance=0, for every pair of an available user input and the input parameter using the contexts C. Then the obtained feature vectors are feed into the trained ranking model. The ranking model generates a score for each user input. We use the scores to rank user inputs.

4 Case Study

We introduce our dataset and four research questions. For each question, we present the motivation of the question, the analysis approach and our findings.

4.1 Case Study Setup

We conducted our study on on-line services (*i.e.*, web forms) and Web services: WSDL (*i.e.*, SOAP-based) and RESTful services. The RESTful services are described in web pages.

TABLE 3: Our collection of public Web forms and services.

Domain	# of WSDL	# of REST	# of Web forms
Travel	215	30	50
E-commerce	190	30	50
Finance	135	30	50
Entertainment	100	30	50
Total	640	120	200

We use Google to search for web form based websites to download web forms. We choose websites listed on the top 50 of the Google results. We download 200 Web forms. We collect 640 public available WSDL files. We use programmableWeb[10] to collect 120 URLs of RESTful services and download the web pages containing the APIs. We manually collect the information of RESTful services, such as the description of resources and input parameters. The total 960 services fall into 4 domains: Travel (*e.g.*, book flights), E-commerce (*e.g.*, buy shoes), Finance (*e.g.*, check a stock price) and Entertainment (*e.g.*, search TV shows). Table 3 provides a summary of the dataset used in our case study. In total, we collect 1024, 11127, 792 input parameters from Web forms, WSDL, and Restful services, respectively.

User input collection: To evaluate the effectiveness of our framework, we collect interactions between user inputs and input parameters (UIN-IP interactions) using our input collector (Section 3.1). We recruit 6 subjects who are graduate students and typically spend 8-10 hours on-line per day to use the input collector tool to track their inputs on web forms. When the subjects conduct a task, in addition to automatically collecting most contexts for UIN-IP interactions as listed in Table 1, the subjects are requested to optionally and manually clarify a few contexts, such as the type and name of a user location. We collect 12,584 interactions between user inputs and input parameters over a week. For each

10. http://www.programmableweb.com/

UIN-IP interaction, we calculate a feature vector using all the proposed ranking features. We refer this automatically collected dataset as *auto-data* (i.e., Web forms). Like prior studies for personalized web form filling, such as [6] [8], we recruit a small group of subjects (i.e., 6 subjects) as users are usually reluctant to share their personal data or share limited data with some types of information removed (e.g., bank information). However, unlike prior studies, we manage to collect a large volume of UIN-IP interactions from each recruited subject without any information deleted.

Manually-labeled training dataset: The *auto-data* is collected from web forms that the recruited subjects chose. To test our approach on all of the collected Web services and Web forms, we create a training dataset in the following steps: (1) We randomly sample input parameters of each type service (denoted as set_{ip}^{Sample}) with the confidence level 95% [27] so that we can manually identify user inputs from *auto-data* for set_{ip}^{Sample}. We randomly sample 280, 371, 259 input parameters for Web forms, WSDL, and Restful, respectively. (2) For each input parameter in set_{ip}^{Sample}, we retrieve a list of user inputs from *auto-data* whose textual information matches the textual information of the input parameter. If the number of retrieved user inputs is greater than 10, we sort the retrieved user inputs in a descending order based on the similarity values. We choose top 10 user inputs. Third, the six subjects judge the relevance of a user input to the input parameter. There are two degrees of relevance: 1 (*i.e.*, best value for the input parameter) and 0 (*i.e.*, not suitable for the input parameter). The contexts of each pair of user input and input parameter are collected using our input collector 3.1. Last, we calculate the feature values for all the 11 ranking features. We refer the manually labeled dataset as *manually-labeled-data*.

Computation Environment: We conduct our experiments on a Lenovo windows machine having a 8GB RAM and an Intel Core (TM) i7-2620M CPU 2.70GHz.

4.2 Research Questions

We conduct experiments to measure the performance of our approach and answer the following research questions.

RQ 1. Which Learning-to-Rank models are the most suitable for the task of ranking user inputs?

Motivation. We use the learning-to-rank (LtR) framework, consisting of a set of LtR models, to learn and analyze user input entry activities. Therefore, we need to select a learning-to-rank (LtR) model for training and ranking user inputs for input parameters.

To maximize the performance of our framework, it is critical to select and deploy a best suitable LtR model.

Approach. We study 8 well-known and widely adopted LtR models to discover the best performing LtR model on ranking user inputs for input parameters. The 8 LtR models are listed as follows:

Four pairwise LtR models: (1) **RankSVM** [10] [11] formalizes the ranking problem as a pairwise classification problem and employs the SVM technique to perform the learning task. (2) **IR SVM** [28] emphasizes two important factors in ranking for document retrieval: It is crucial to rank correct documents on the top of list; and different queries have varying numbers of relevant documents. (3) **RankBoost** [29] is based on the gradient boosting technique. (4) **RankNet** [30] is based on pairwise classification like RankSVM and RankBoost.

Four listwise LtR models: (1) **AdaRank** [31] directly optimize an evaluation measure by using the Boosting technique. (2) **ListNet** [32] uses the Neural Network model for training and employs the Kullback-Leibler (KL) divergence [33] as a loss function. (3) **LambdaRank** [34] uses an implicit listwise loss function optimized with Gradient Descent to learn the optimal ranking model. (4) **LambdaMART** [35] [36] uses the Gradient Tree Boosting [37] technique to learn a boosted regression tree as a ranking model.

We only select and compare pairwise and listwise LtR models, because it has been empirically proved that pairwise and listwise LtR models can consistently outperform pointwise LtR models [38]. To test the above 8 LtR models on ranking user inputs, our framework utilizes each of them on the collected two datasets, *auto-data* and *manually-labeled-data* in Section 4.1. For each dataset, we build vectors of features for interactions between user inputs and input parameters in the dataset.

We conduct our experiment in the following steps: First, we apply each of 8 LtR models on *auto-data* and *manually labeled data*. More specifically, we follow the same data splitting strategy in [39]. For each dataset, we use half of a dataset for training and validating a LtR model, and the remaining half for testing. After a LtR model is trained, a ranking model is generated. Each LtR model is tuned on the validation data and the final ranking model of each LtR model is the one that performs best on the validation data. Second, given a set of input parameters $P_1, P_2, \ldots, P_n$ (n is the number of input parameters) in a testing dataset, we retrieve at most 10 candidate values for each input parameter using the textual similarity calculation approach of building manually-labeled training dataset in Section 4.1. Third, we apply the trained LtR models to generate rankings of 10 user inputs for the given set of input parameters. Last, we use Equation (5) and Equation (6) to measure k-precision and k-recall for recommending user inputs to P_i $(0 < i \leq n)$. We aim to help users select the desired value from a short list of user inputs. Therefore, we set k to 1, 3, and 5.

We use Equation (7) and Equation (8) to calculate the average precision and recall.

$$k - precision_i = \frac{|Correct\ Inputs\ in\ top\ k\ results|}{k} \tag{5}$$

$$k - recall_i = \frac{|Correct\ Inputs\ in\ top\ k\ results|}{|Correct\ Input|} \tag{6}$$

$$avg - k - precision = \frac{\sum_{i=1}^{n}(k - precision_i)}{n} \tag{7}$$

$$avg - k - recall = \frac{\sum_{i=1}^{n}(k - recall_i)}{n} \tag{8}$$

where $|$ $Correct\ Input$ $|$ *is a constant and = 1, because there can only be one correct input for a* P_i. $|$ $Correct\ Inputs\ in\ top\ k\ results$ $|$ *= (1 or 0). When k=1, k-precision*$_i$*= k-recall*$_i$*. Where n is the number of input parameters we rank lists of user inputs for.*

Run-time Complexity. We record the duration of time needed to train a LtR model. Moreover, we record the duration of time our approach takes to recommend a list of 5 relevant user inputs for an input parameter. We report the average run-time over all of the input parameters.

TABLE 4: Performance of different learning-to-rank approaches with different top-k values on ***manually-labeled-data***. P: Precision, R: Recall. All results of precision and recall are in percentage (%).

K	Services	RanKSVM		IRSVM		RanKBoost		RanKNet		AdaRank		ListNet		LambdaRank		LambdaMart	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
K=1	WSDL	82.5	82.5	81.2	81.2	87	87	85	85	84.3	84.3	82	82	84.5	84.5	**87.5**	**87.5**
	REST	84.3	84.3	82.5	82.5	**86.5**	**86.5**	84.3	84.3	82.2	82.2	80.5	80.5	84	84	85	85
	Web Forms	83	83	82	82	**88.2**	**88.2**	85.5	85.5	86.2	86.2	81	81	85	85	87	87
K=3	WSDL	29.2	87.6	27.5	82.5	30	90	29	87	29.1	87.3	28	84	28.7	86.1	**31**	**93**
	REST	28.5	85.5	28	84	**30.5**	**91.5**	28.7	86.1	29.5	88.5	27.5	82.5	28.3	84.6	30	90
	Web Forms	29	87	28.3	84.9	**31**	**93**	29.3	87.9	30	90	28	84	29.3	87.9	**31**	**93**
K=5	WSDL	18	90	18.2	91	**19.2**	**96**	18.2	91	18.2	92.5	18	90	18.5	92.5	19	95
	REST	18.2	91	17	85	**19**	**95**	17.5	87.5	18.5	92.5	18.5	92.5	18.2	91	**19**	**95**
	Web Forms	18.5	92.5	17.5	87.5	**19.5**	**97.5**	18	90	19	95	18.5	92.5	18	90	**19.5**	**97.5**

TABLE 5: Performance of different learning-to-rank approaches with different top-k values on ***auto-data*** (i.e., only Web forms). P: Average precision, R: Average recall.

Approach	k=1		k=3		k=5	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
RankSVM	84	84	30	90	19	95
IRSVM	80	80	27.3	82	17	85
RankBoost	87	87	**30.3**	**91**	**19.4**	**97**
RankNet	85.5	85.5	29.3	87.9	18.3	91.5
AdaRank	86.5	86.5	30.1	90.3	19	95
ListNet	82	82	28.3	84.9	18.6	93
LambdaRank	84.5	84.5	28.7	86.1	18.2	91
LambdaMart	**87.5**	**87.5**	30	90	19.2	96

Results. The results in Table 4 suggest that generally RanKBoost can outperform other LtR models on manually-labeled-data. Only LambdaMart can slightly work better than RanKBoost on Top-1 and Top-3 ranked user inputs for input parameters from WSDL services. Table 5 shows that on dataset *auto-data* (i.e., only Web forms), RanKBoost can outperform other LtR models on Top-3 and Top-5 ranked user inputs, but LambdaMart performs best on Top-1. Even though IRSVM does not work well compared with other LtR models, it sill obtains a precision over 80%. Generally, any LtR model can achieve an average recall of 92.9% on Top-5 ranked user inputs, meaning that in most cases, users are not required to type any values into input parameters, they can just choose a value from our ranked list of user inputs.

Overall, we select RankBoost as a learning-to-rank model for our framework. Moreover, the results of Table 5 and Table 4 show that the performance of a ranking built on automatically collected interactions between user inputs and input parameters from users (*i.e.*, auto-data) can achieve the same level of performance as a ranking built on manually picked user inputs for input parameters.

Running Time Complexity: On average, the learning-to-rank (LtR) algorithms take 10–20 mins to build a ranking model on our training dataset. Building a trained model on a dataset is usually off-line. The trained model is applied in real time to rank user inputs for an input parameter in our testing dataset. On average, our framework using any LtR model can generate a list of 5 user inputs within 0.2 second.

Summary of RQ1: RankBoost performs the best in ranking user inputs for input parameters compared with other LtR models. As shown in our results, our framework that learns and analyzes the automatically collected user inputs can achieve the same performance in ranking user inputs as manually ranking user inputs.

RQ 2. Is our framework effective in ranking user inputs?

Motivation. After selecting a best working learning-to-rank (LtR) model in our framework, we need verify the effectiveness of our proposed LtR framework by comparing it with existing approaches that are not learning-to-rank algorithms.

Approach. We build two baseline approaches and compare them with our framework using RankBoost that achieves the best performance in **RQ1**. The two baselines are listed as follows:

*Frequency-based (denoted as **Rank-F**).* The first baseline ranks user inputs in a descending order based on the frequencies of user inputs. The user input with the highest frequency ranks on the top.

*Bayesian Belief Network (BBN) [20] based ranking approach (denoted as **Rank-BBN**).* Given an input parameter and a set of user contexts, we build a vector of features (*i.e.*, 11 ranking features in Section 3.2.3) for each user input in *Rank-BBN*. The vectors of user inputs are used to build a BBN. The output node is the probability of a user input being used for an input parameter. The probabilities of user inputs are used by *Rank-BBN* for ranking.

We conduct our experiment in the following steps: First, we apply *Rank-F*, *Rank-BBN*, and *RanKBoost* on the collected datasets, *auto-data* and *manually-labeled-data*. Second, both *Rank-BBN* and *RanKBoost* require training datasets. We follow the same data splitting strategy in [39]. We use half of a dataset for training and validation, and the remaining half for testing. Third, given a set of input parameters $P_1, P_2, \ldots, P_n$ (n is the number of input parameters), we use Equation (5) and Equation (6) to measure k-precision and k-recall for recommending a list of user inputs to P_i ($0 < i \leq n$). We use Equation (7) and Equation (8) to calculate the average precision and recall.

TABLE 6: Comparison of RankBoost and conventional approaches with different top-k values on *auto-data*. P: Precision, R: Recall.

Approach	k=1		k=3		k=5	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
RanKBoost	**87**	87	30.3	91	19.4	**97**
Rank-BBN	65	65	23	69	14	70
Rank-F	43	43	15	45	11	55

Results. Table 6 and Table 7 show that our approach *RanKBoost* outperforms *Rank-BNN* and *Rank-F* on both datasets. More specifically, *RanKBoost* achieves an average top-1 precision of 87% and top-1 recall of 87%. In most cases, the users can be saved from repetitive typing without

TABLE 7: Performance of different learning-to-rank approaches with different top-k values on *manually-labeled-data*. P: Precision, R: Recall.

Approach	WSDL		REST		Web Forms	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
RanKBoost (k=1)	87	87	86.5	86.5	88.2	88.2
Rank-BBN (k=1)	58	58	55	55	62	62
Rank-F (k=1)	42	42	46	46	40	40
RanKBoost (k=3)	30	90	30.5	91.5	31	93
Rank-BBN (k=3)	23	69	21	63	20	60
Rank-F (k=3)	17	51	19	57	15	45
RanKBoost (k=5)	19.2	96	19	95	19.5	97.5
Rank-BBN (k=5)	14	70	13	65	14	70
Rank-F (k=5)	11	55	12	60	10	50

taking actions using our *RanKBoost*. Furthermore, the high top-3 and top-5 recalls of *RanKBoost* suggest that users can identify a proper user input within top-3 or top-5 ranked values, when the users are not satisfied with the top-1 value.

Consolidating the results in **RQ1** and **RQ2**, we observe that even IRSVM that performs the worst among all LtR models can outperform the two baseline approaches, meaning that learning-to-rank models can perform better than conventional ranking approaches.

Summary of RQ2: RankBoost outperforms the frequency based and Bayesian Network based baseline approaches in ranking user inputs. We observe that our framework utilizing any one of the 8 LtR models can outperform the two baseline approaches.

RQ 3. Which ranking features affect the ranking most?

Motivation. Our proposed ranking features capture different aspects of interactions between user input and input parameters. Distinct features can affect the ranking of user inputs in different ways. In this research question, we investigate the effect of various features on ranking user inputs and discover the most influential features in ranking.

Approach. We conduct two experiments:

[Experiment 1.] We investigate the effect of each category of features on ranking user inputs to find the category of ranking features having the largest impact on the performance of ranking. To test a category of features, we first exclude the tested category of features from the training dataset *auto-data*. Second, we apply *RanKBoost* on the *auto-data* and use Equation (7) and Equation (8) to calculate the average precision and recall for *RanKBoost*. We test 6 categories of features and compare the results with the ones of RQ1 to identify the effect of different features on ranking.

[Experiment 2.] We test the effect of each feature in a category on ranking. We can only test pattern features as a group, as the number of pattern features can be very high and patterns of user inputs are not fixed. We test the effect of each context type on the ranking of user inputs. We use the same above approach in *Experiment 1* to test a feature.

Results. Table 8 shows the effects of categories of ranking features. The percentages in Table 8 mean the decreases in the performance of RanKBoost without using the test feature. The larger the percentage is, the larger effect a ranking feature has. Without using the 1st category of ranking features, textual similarity based ranking features, the performance of our approach decreases most (*e.g.*, 22% off in top-1 precision) compared with other features. Furthermore,

TABLE 8: Effects of different types of ranking features on user input ranking. d-P and d-R stand for the decrease in average precision and recall respectively.

Test Feature	k=1		k=3		k=5	
	d-P	d-R	d-P	d-R	d-P	d-R
1 Textual	22%	22%	9%	27%	7%	35%
2 Task	7%	7%	4%	12%	4%	20%
3 Frequency	5%	5%	4%	12%	3%	15%
4 Time-length	2%	2%	1%	3%	2%	10%
5 Patterns	10%	10%	6%	18%	5%	25%
6 Context	14%	14%	7%	21%	6%	30%

TABLE 9: Effects of different contexts-based ranking features on user input ranking. We list top-5 significant features. d-P and d-R stand for the decrease in average precision and recall respectively.

Test Feature	k=1		k=3		k=5	
	d-P	d-R	d-P	d-R	d-P	d-R
1 Types of the location	5%	5%	2%	6%	2%	10%
2 Types of the devices	3%	3%	1%	3%	1%	5%
3 AM or PM	3%	3%	1%	3%	0%	0%
4 Day of the week	1%	1%	1%	3%	0%	0%
5 Hour	1%	1%	0%	0%	0%	0%

we observe that compared with other types of contextual information, user location and time make a larger impact on the rankings of user inputs in Table 9. In particular, the context, the type of the user location among the context types of location, matters the most to the ranking.

Summary of RQ3: The category of textual similarity based ranking features can make the largest impact on rankings of user inputs, compared with other categories. Among the contexts based ranking features, user location and time make the largest impact on the rankings of user inputs.

RQ 4. Can our approach outperform Google Chrome Auto-filling tool?

Motivation. Google Chrome is a popular web browser used by millions of people every day. The Google Chrome auto-filling tool (Chrome-fill) [9] is an add-on of Google Chrome browser. When a user clicks or starts typing values into an input parameter, such as an input field in a web form, Chrome-fill recommends a list of user inputs to the user based on the user's typing. To test the effectiveness of our ranking framework against industrial tools, we compare our framework adopting RankBoost with Chrome-fill.

Approach. Since Google does not publish any details of algorithms used by Chrome-fill, to compare our framework using RanKboost with Chrome-fill on ranking values, we conduct the experiment in the following steps:

(1) We clear out the auto-filling history in our Chrome-fill so that our previous history does not bias our analysis.

(2) Since Chrome-fill can only support web applications, we randomly sample input parameters in *auto-data* (denoted as *Sample*) with the confidence level 95% [27] so that we can manually compare the results of our framework with the ones of Chrome-fill.

(3) We split *Sample* into 80% for training (denoted as *Sample-80%*) and 20% for testing (denoted as *Sample-20%*) for both our framework using RanKBoost and Chrome-fill.

(4) We manually visit the web forms that contain the input parameters in *Sample-80%* using Google Chrome and

fill the corresponding user inputs stored in interactions between user inputs and input parameters (UIN-IP interactions) of *auto-data* into these input parameters. Since UIN-IP interactions are stored in a descending order based on the time from newest to oldest, we enter the user inputs in the same order as they are stored in UIN-IP interactions.

(5) We visit the web forms containing the input parameters in *Sample-20%* using Google Chrome and record the ranking of user inputs from Chrome-fill. We calculate precision and recall using Equation (7) and Equation (8) to measure the effectiveness of our framework using RankBoost and Chrome-fill on ranking user inputs.

(6) We repeat the above steps 5 times (5-fold cross-validation) and calculate the average precision and recall for evaluating our framework and Chrome-fill.

TABLE 10: Results of our framework using RankBoost and Chrome-fill on ranking user inputs. P: Precision, R: Recall.

Top-K	Our Framework		Chrome	
	P(%)	R(%)	P(%)	R(%)
Top 1	92	92	56	56
Top 3	32	96	20	60
Top 5	19.5	97.5	13	65

Results. Table 10 shows that our framework outperforms Chrome-fill. For example, our approach can achieve a precision of 92% and a recall of 92% on Top 1 ranked user input, while Chrome-fill can only achieve a precision of 56% and a recall of 56% on Top 1 ranked user input. Low recalls of Chrome-fill suggest that in 40% of cases, users have to physically type values into input parameters. We further investigate the results of our framework and Chrome-fill. We found that the patterns of user inputs (*i.e.*, user inputs are used to fill in input parameters together.) help identify the right values for recommendation. With the accumulation of user history and more patterns generated, the results can be improved gradually.

In addition to quantitative comparison, we also summarize our comparison qualitatively by comparing features of our framework and Chrome-fill. Chrome-fill has the following disadvantages compared with our framework: (1) Chrome-fill can only track a limited number of input parameters, such as name and address. The limited support of user input collection leads to low recalls. (2) Chrome-fill is not context-aware, and cannot automatically learn and analyze user input entry activities. Lack of ability to learn and context-awareness leads to low precisions.

Summary of RQ4: Our framework using RankBoost can outperform Chrome-fill quantitatively and qualitatively. Compare with our framework quantitatively, Chrome-fill achieves low recalls, because it can only track a limited number of input parameters. Furthermore, Chrome-fill achieves low precisions, due to the lack of ability to learn user input entry activities and context-awareness.

5 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [40].

Construct validity threats concern the relation between theory and observation. In this paper, we believe that extracting input parameters from RESTful services and web applications (*i.e.*, web forms) mainly causes the construct validity threats. It is challenging to extract information from web pages as the structures of web forms can be very different. For example, the labels in web forms can be in various positions. To extract information from web pages, we implement the approach in [41] .

Internal validity threats are mainly from the following sources: First, we manually labeled subject profiles in the section of Case Study Setup (*i.e.*, Section 4.1). However, we have guidelines before we do the manual labeling process. A serious attention is paid not to violate any guidelines so that a big fluctuation of results with the change of the experiment conductor can be avoided. It is common to include manual processes in research studies, such as the manual process in [6]. Second, we use a MAC address as the identify for a user for the simplicity of implementation, even though using MAC addresses could cause security concerns. We value privacy and security issues very much. However, privacy and security issues are not the main focus of this paper. In the future, we plan to use users' profiles on social medias upon users' permission.

6 Related Work

Here, we summarize the related work on filling out Web forms and web services.

Pre-filling values to input parameters of on-line services. Several industrial tools (*e.g.*, [7] [42] [43]) have been developed to pre-fill web forms. For example, Firefox Add-on Autofill Forms [7] can automatically fill stored values into input parameters. However, it requires users to manually create entries for filling profiles and enter values for the pre-defined entries. LastPass [42] and 1Password [43] are specialized in password management and provide form auto-filling function. These tools store user's information in central space, and automatically fills in the fields with stored values once the user revisit the page.

Similar to the above industrial tools, another group of approaches (*e.g.*, [4] [8] [6]) improve the accuracy of filing out web forms by creating a personal space storing structured information for users. For example, Winckler *et al.* [4] propose an approach using a pervasive information space for storing and collecting user's personal data for filling out forms. Firmenich *et al.* [8] propose an approach based on web form augmentation [44] to support a straightforward interaction between third-party web forms and users' personal information management systems for web form filling. Araujo *et al.* [6] mine the semantical concepts of the web form fields and use the relations of concepts to fill out web forms. All of the above industrial and academic tools require manual work from users.

Some academic studies such as [21] [23] [45] [46] propose several approaches to automatically pre-fill web forms. For example, Wang *et al.* [21] [23] propose an approach identifying and using the relations between similar input parameters to fill out web forms. Toda *et al.* [47] propose a probabilistic approach using the information extracted from data-rich text as the input source to fill values into web forms. WebFeeder [45] leverages iMarcos [48] from managing script code to script models to automatically fill the form-intensive websites requiring a large quantify of data with external data sources already digitalized in terms of documents, spread-sheets or databases. Kristjansson *et al.* [46] propose an interactive form filling system to help data entry workers with the tedious and error-prone database form filling. Their system extracts text segments from the unstructured text and populates the fields with the corresponding values from the text segments. All of the above approaches and tools are not context-aware. They do not store and analyze contextual information and patterns of user inputs.

Recommending values for filling in input parameters of on-line services. Some other industrial and academic approaches (*e.g.*, [9] [49]) recommend a list of values to users for input parameters. For example, Google Chrome Autofill forms (Chrome-fill) [9] can record a limited number of types of user inputs and recommend a list of previously collected user inputs to users when the users click or choose an input field of a web form. Chrome-fill can generate patterns of user inputs and recommend user inputs in patterns to users. Hermens *et al.* [50] develop a non-intrusive assistant, a learning and prediction system, to provide values for blank fields in a form. CCFU [3] uses a joint probability distribution based on Bayesian network to calculate the contextual relevance for predicting a value for a target field. To learn the dependencies between fields in a form, CCFU adopts a greedy structure learning algorithm on a set of previously filled-out forms . However, the above approaches are not context-aware. Hartmann *et al.* [49] propose an approach to map the user contextual information with the user interfaces. The context-aware user interfaces facilitate the user interaction by suggesting or pre-filling data derived from the user's current contexts. The mapping algorithm is based on string-based measure and semantic measures (*i.e.*, using wordnet [51]). However, the above approaches utilize conventional ranking models which cannot adjust the importance of ranking features over time.

Assigning values and user interface generation for Web services. Auto-testing and invoking Web services require the automatic value assignments for the generated user interfaces of Web services. For example, AbuJarour *et al.* [52] investigate the approaches to generate annotations for web services by sampling their automatic invocations. Four sources, such as random values, are used to automatically assign values for input parameters of web services. Spillner *et al.* [13] propose a new Web Services Graphical User Interfaces Engine that accepts multiple web service formats and pre-fill the auto-generated user interfaces. He *et al.* [53] propose an adaptive user interface generation framework that takes WSDLs with interface design related information for adaptive GUI generation. Zhou *et al.* [54] develop a customized interface generation model based on knowledge and template for Web services.

7 Conclusion

Unnecessary interruptions caused by repetitively typing same information to services decreases the efficiency and negatively impacts the user experience. In this paper, we propose a learning-to-rank (LtR) model based framework to recommend user inputs for input parameters. We summarize the major contributions of this paper as follows:

- We propose a meta-data model for capturing and storing interactions between user inputs and input parameters with contextual information, such as time and user location. The stored contextual information makes our framework context-aware.
- Our framework exploits the user information (*e.g.*, user contexts), input parameters and user inputs for learning past interactions between user inputs and input parameters in order to reuse user inputs for filling and ranking values for input parameters.
- We test the effectiveness of our framework on real-world on-line services and Web services through a series of empirical studies. Our results show that our framework can work well on web forms, and as an addition, perform well on Web services. Our results suggest that RankBoost [29] can outperform other LtR models on ranking user inputs. Furthermore, our framework utilizing any LtR model outperforms the two conventional ranking approaches (*i.e.*, Bayesian Belief Network [20] and frequency-based approaches) and Google Chrome auto-filing tool [9] on ranking user inputs. Moreover, we observe that the textual similarity based features affect the ranking most.

In the future, we plan to recruit more subjects to conduct user case study and add a module addressing user privacy issues in our framework for users' daily uses. Furthermore, we want to expand the capabilities of our framework on other software artifacts or systems, such as recommending next Java methods in building Web services.

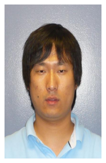
Acknowledgments

This research is partially supported by IBM Canada Centers for Advance Studies.

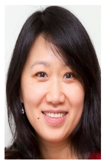
References

[1] H. Xiao, Y. Zou, R. Tang, J. Ng, and L. Nigul. Ontology-driven Service Composition for End-users. *Journal of Service Oriented Computing and Applications*, 5(3):159–181, 2011.
[2] P. Viola and M. Narasimhan. Learning to extract information from semi-structured text using a discriminative context free grammar. In *ACM SIGIR conference on Research and development in information retrieval*, pages 330–337, 2005.
[3] A. Ali and C. Meek. Predictive models of form filling. Technical Report, MSR-TR-2009-1, Microsoft Research, 2009.
[4] M. Winckler, V. Gaits, D. Vo, F. Sergio, and G. Rossi. An approach and tool support for assisting users to fill-in web forms with personal information. In *Int'l Conference on Design of Communication*, pages 195–202. ACM, 2011.
[5] E. Rukzio, C. Noda, A. De Luca, J. Hamard, and F. Coskun. Automatic form filling on mobile devices. *Pervasive and Mobile Computing*, 4(2):161–181, 2008.
[6] S. Araujo, Q. Gao, E. Leonardi, and G. Houben. Carbon: Domain-independent automatic web from filling. In *International Conference on Web Engineering (ICWE)*, pages 292–306. LNCS, 2010.
[7] Firefox. Autofill forms. https://addons.mozilla.org/en-US/firefox/addon/autofill-forms. Last accessed on Jul. 09, 2016.
[8] S. Firmenich, V. Gaits, S. Gordillo, G. Rossi, and M. Winckler. Supporting users tasks with personal information management and web forms augmentation. In *International Conference on Web Engineering*, pages 268–282. Springer, 2012. July 23-27, 2012, Berlin.
[9] Google. Chrome autofill forms. https://support.google.com/chrome. Last accessed on Jul. 09, 2016.
[10] R. Herbrich, T. Graepel, and K. Obermayer. Support vector learning for ordinal regression. In *Int'l Conf. on Artificial Neural Networks*. IET, 1999.

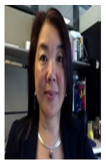
[11] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. *Advances in neural information processing systems*, pages 115–132, 1999.
[12] S. Wang, Y. Zou, J. Ng, and T. Ng. Learning to reuse user inputs in service composition. In *Int'l Conf. on Web Services*, pages 695–702. IEEE, 2015.
[13] J. Spillner, M. Feldmann, I. Braun, T. Springer, and A. Schill. Ad-hoc usage of web services with dynvoker. In *European Conference on a Service-Based Internet*, pages 208–219. Springer, 2008.
[14] WSDL. http://www.w3.org/TR/wsdl. Last accessed on Jul. 09, 2016.
[15] L. Richardson and S. Ruby. *RESTful web service*. O'Reilly Media, 2007.
[16] Sun Microssystems Inc. Web application description language. http://www.w3.org/Submission/wadl/. Last accessed on August 25th, 2015.
[17] H. Li. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems*, 94(10):1854–1862, 2011.
[18] T. Joachims. Optimizing search engines using clickthrough data. In *ACM SIGKDD Int'l Conf. on Knowledge discovery and data mining*, pages 133–142, 2002.
[19] G. E. Dupret and B. Piwowarski. A user browsing model to predict search engine click data from past observations. In *Int'l ACM SIGIR conference on Research and development in information retrieval*, pages 331–338. ACM, 2008.
[20] M. P. de Cristo, P. P. Calado, M. L. da Silveira, I. Silva, R. Muntz, and B. Ribeiro-Neto. Bayesian belief networks for ir. *Approximate Reasoning*, 34(2):163–179, 2003.
[21] S. Wang, Y. Zou, I. Keivanloo, B. Upadhyaya, J. Ng, and T. Ng. Automatic reuse of user inputs to services among end-users in service composition. *IEEE Transactions on Services Computing*, 8(3):343–355, May 2015.
[22] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
[23] S. Wang, B. Upadhyaya, Y. Zou, I. Keivanloo, J. Ng, and T. Ng. Automatic propagation of user inputs in service composition for end-users. In *International Conference on Web Services*, pages 73–80. IEEE, 2014.
[24] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.
[25] S. Tata and J. M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM Sigmod Record*, 36(2):7–12, 2007.
[26] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. In *15th International Conference on Data Engineering*, pages 512–521. IEEE, 1999.
[27] R. L. Chambers and C. J. Skinner. *Analysis of survey data*. John Wiley & Sons, 2003.
[28] Y. Cao, J. Xu, T. Liu, H. Li, Y. Huang, and H. Hon. Adapting ranking svm to document retrieval. In *ACM SIGIR conf. on Research and development in information retrieval*, pages 186–193, 2006.
[29] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *Machine Learning Research*, 4:933–969, 2003.
[30] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Int'l Conf. on Machine learning*, pages 89–96. ACM, 2005.
[31] J. Xu and H. Li. Adarank: a boosting algorithm for information retrieval. In *ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, 2007.
[32] Z. Cao, T. Qin, T. Liu, M. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Int'l Conf. on Machine learning*, pages 129–136. ACM, 2007.
[33] S. Kullback. *Information theory and statistics*. Courier Corporation, 1968.
[34] C. Quoc and V. Le. Learning to rank with nonsmooth cost functions. *Advances in Neural Information Processing Systems*, 19:193–200, 2007.
[35] C.J. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11:23–581, 2010.
[36] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.
[37] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.
[38] H. Li. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies*, 7(3):1–121, 2014.
[39] B. Xiang, D. Jiang, J. Pei, X. Sun, E. Chen, and H. Li. Context-aware ranking in web search. In *SIGIR conference on Research and development in information retrieval*, pages 451–458. ACM, 2010.
[40] Robert K Yin. *Case Study Research: Design and Methods*. Sage publications, 2014.
[41] B. Upadhyaya, F. Khomh, and Y. Zou. Extracting restful services from web applications. In *Int'l Conf. on Service-Oriented Computing and Applications*, pages 1–4. IEEE, 2012.
[42] LastPass. www.lastpass.com. Last accessed on March 19th, 2015.
[43] 1Password. https://agilebits.com/onepassword. Last accessed on Jul. 09, 2016.
[44] S. Firmenich, M. Winckler, G. Rossi, and S. Gordillo. A framework for concern-sensitive, client-side adaptation. In *11th International Conference on Web Engineering*, pages 198–213. Springer, 2011.
[45] O. Diaz, I. Otaduy, and G. Puente. User-driven automation of web form filling. In *International Conference on Web Engineering*, pages 171–185. Springer, 2013. July 8-12, Aalborg, North Denmark.
[46] T. Kristjansson, A. Culotta, P. Viola, and A. McCallum. Interactive information extraction with constrained conditional random fields. In *AAAI International Conference*, pages 412–418, 2004.
[47] G. A. Toda, E. Cortez, A. S. da Silva, and E. de Moura. A probabilistic approach for automatically filling form-based web interfaces. *VLDB Endowment*, 4(3):151–160, 2010.
[48] iMarcos. http://imacros.net/overview. Last accessed on Sep. 12, 2015.
[49] M. Hartmann and M. Muhlhauser. Context-aware form filling for web applications. In *IEEE International Conference on Semantic Computing*, 2009.
[50] L. A. Hermens and J. C. Schlimmer. A machine-learning apprentice for the completion of repetitive forms. *IEEE Expert: Intelligent Systems and Their Applications*, 9(1), 1994.
[51] C. Fellbaum. *WordNet An Electronic Lexical Database*. MIT Press, 1998.
[52] M. AbuJarour and S. Oergel. Automatic sampling of web services. In *Int'l Conf. on Web Services*, pages 291–298. IEEE, 2011.
[53] Jiang He and I-Ling Yen. Adaptive user interface generation for web services. In *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pages 536–539. IEEE, 2007.
[54] R. Zhou, J. Wang, G. Wang, and J. Li. Customized interface generation model based on knowledge and template for web service. *Journal of Networks*, 9(12), 2014.



Shaohua Wang is an assistant professor in the Department of Informatics at New Jersey Institute of Technology. He received his Ph.D from Queen's University, Canada. His research interests include: Software Engineering, service-oriented computing, machine learning, and mining software repositories.



Ying Zou is a Canada Research Chair in Software Evolution, and an associate professor in the Electrical and Computer Engineering at Queen's University in Canada. She is a visiting scientist at IBM Canada. Her research interests include software engineering, software re-engineering and maintenance.



Joanna Ng is currently a Cognitive IoT Strategy and Research Chair, Innovator and Master inventor in IBM. She was the Head of Research at IBM Canada Software Laboratories, Center for Advanced Studies.

Tinny Ng is currently a Research Staff Member for IBM CAS Research Canada at IBM Canada Lab. Her primary focus is to manage research portfolios and transfer advanced research into strategic product solutions for IBM Software Group.