Assisting End-users in Filling out Web Services

by

Shaohua Wang

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
May 2016

Abstract

With the quick advance of web service technologies, end-users can conduct various on-line tasks, such as shopping on-line. Usually, end-users compose a set of services to accomplish a task, and need to enter values to services to invoke the composite services. Quite often, users re-visit websites and use services to perform re-occurring tasks. The users are required to enter the same information into various web services to accomplish such re-occurring tasks. However, repetitively typing the same information into services is a tedious job for end-users. It can negatively impact user experience when an end-user needs to type the re-occurring information repetitively into web services.

Recent studies have proposed several approaches to help users fill in values to services automatically. However, prior studies mainly suffer the following drawbacks: (1) limited support of collecting and analyzing user inputs; (2) poor accuracy of filling values to services; (3) not designed for service composition. To overcome the aforementioned drawbacks, we need maximize the reuse of previous user inputs across services and end-users.

In this thesis, we introduce our approaches that prevent end-users from entering the same information into repetitive on-line tasks. More specifically, we improve the process of filling out services in the following 4 aspects: First, we investigate the characteristics of input parameters. We propose an ontology-based approach to automatically categorize parameters and fill values to the categorized input parameters. Second, we propose a comprehensive framework that leverages user contexts and usage patterns into the process of filling values to services. Third, we propose an approach for maximizing the value propagation among services and end-users by linking a set of semantically related parameters together and similar end-users. Last, 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. Our framework learns and analyzes interactions of user inputs and input parameters to rank user inputs for input parameters under different contexts.

Related Publications

Earlier versions of the work in this thesis were published as listed below:

- **An Empirical Study on Categorizing User Input Parameters for User Inputs Reuse** (Chapter 4). Shaohua Wang, Ying Zou, Bipin Upadhyaya, Iman Keivanloo and Joanna Ng. Proc. the 14th International Conference on Web Engineering (ICWE 2014) , pp. 21-39, 1-4th of July, 2014, Toulouse, France. Springer.

- **An Intelligent Framework for Auto-filling Web Forms from Different Web Applications** (Chapter 5). Shaohua Wang, Ying Zou, Iman Keivanloo, Bipin Upadhyaya, Joanna Ng. International Journal of Business Process Integration and Management (IJBPIM), Inderscience publishers, in press.

- **An Intelligent Framework for Auto-filling Web Forms from Different Web Applications** (Chapter 5). Shaohua Wang, Ying Zou, Bipin Upadhayaya and Joanna Ng. Proc. 1st International Workshop on Personalized Web Tasking (PWT 2013) on IEEE Services, pp. 175-179, June 2013 - Santa Clara, California, USA . IEEE Computer Society Press.

- **Automatic Propagation of User Inputs in Service Composition for End-users** (Chapter 6). Shaohua Wang, Bipin Upadhyaya, Iman Keivanloo, Ying Zou, Joanna Ng and Tinny Ng. Proc. the 21st IEEE International Conference on Web Services (ICWS), pp. 73-80, June 27 - July 2, 2014, Alaska, USA. IEEE.

- **Automatic Reuse of User Inputs to Services among End-users in Service Composition** (Chapter 6). Shaohua Wang, Ying Zou, Iman Keivanloo, Bipin Upadhyaya, Joanna Ng, Tinny Ng. IEEE Transactions on Services Computing (TSC), IEEE Computer Society, pp. 343-355, 8(3), 2015.

- **Learning to Reuse User Inputs in Service Composition** (Chapter 7). Shaohua Wang, Ying Zou, Joanna Ng and Tinny Ng. Proc. the 22nd IEEE International Conference on Web Services (ICWS) , Application Track, pp. 695-702, June 27- July 2, 2015, New York, USA.

The above research papers were co-authored with my supervisor Dr. Ying Zou, IBM Toronto lab researchers Ms. Joanna Ng and Ms. Tinny Ng, and colleagues (Dr. Bipin Upadhyaya and Dr. Iman Keivanloo) in our lab. In all cases, I am the primary author. More specifically, Dr. Ying Zou supervised the research related to those papers. Ms. Joanna Ng, Ms. Tinny Ng, Ms. Diana Lau, and Dr. Bipin Upadhyaya participated the meetings of my research and gave feedback and suggestions to improve the research.

Acknowledgments

Finally, I am here to write the most important part of my thesis. I would like to thank everyone who played an important role in my life to get me here. First, I would like to thank my adviser, Dr. Ying Zou, who have played a formative and significant role in my life and research development. She is a smart, creative, and remarkable person, more importantly, like a big sister to me, who helps and teaches me a lot. It is my greatest honor to know and work with her.

Second, I am grateful to have Dr. Kwei-Jay Lin, Dr. Ahmad Afsahi, Dr. Patrick Martin, and Dr. Hossam S. Hassanein on my thesis examination committee, and thank them for spending their precious time on critiquing my work. Moreover, I am grateful for the support of my PhD supervisory committee, Dr. Ahemd E. Hassan and Dr. Mohammad Zulkernine, who have offered me constructive and insightful feedback and guidance.

I am lucky to know and work with an amazing group of young researchers at the Software Re-engineering Lab. I would like to thank my colleagues for being nice and helpful to me: Dr. Foutse Khomh, Dr. Feng Zhang, Dr. Bipin Upadhyaya, Mr. Yu Zhao, Mr. Pradeep Venkatesh, Ms. Haoran Niu, Mr. Hanfeng Chen, Mr. Tejinder Dhaliwal, Mr. Hao Yuan, and Ms. Liliane Barbour. However, I am particularly grateful to Dr. Foutse Khomh, Dr. Feng Zhang, and Dr. Bipin Upadhyaya, who have been like brothers to me, offering advice and sharing their experiences.

I feel lucky to work with an amazing group of collaborators at IBM Toronto Lab. I thank the great IBMers, Ms. Joanna Ng, Ms. Tinny Ng, Ms. Dianna Lau, for their useful, constructive inputs to my research. More importantly, they have broadened my research perspective. Specially, I learned a great deal from Ms. Joanna Ng, who is a pioneer in computer industry and inspired me a lot.

Last but absolutely not the least, without the support of my family and friends, this thesis would not have been possible. My parents, the greatest parents on this planet, are always there to give me unconditional love and encourage me to pursue my dreams without reservations. I thank my wife, Chen Du, for her love, devotion, sacrifice, and patience. I also thank my two lovely boys, Luke and Mark, for making me happy everyday and lighting up my world. Like your grandparents, I will always be with you and support your dreams. A big **LOVE YOU!** to Dad, Mom, Chen, Luke, and Mark.

Dedication

To my parents, my wife, Luke, and Mark.

Statement of Originality

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, the Web is becoming indispensable in people's lives. Software organizations, such as Amazon[1], publish their services or data through Web services that are software systems created to support and facilitate the interactions between machines [79]. Web services can be published, discovered, and executed on the Web. Recently, the rapid advancement of web service technologies allows millions of users to conduct various on-line tasks, such as shopping on-line and planning a trip. Usually, a single web service is not sufficient to finish a user's task. Quite often, a set of logically related web services are aggregated to accomplish end-users' tasks. The process of aggregating web services to create new services is namely *service composition*. There are two types of service compositions:

- *Explicit*. Service-oriented architecture [65][66] utilizes Web services as basic blocks to build complex systems flexibly and rapidly. Following service-oriented

[1]http://www.amazon.com/

architecture (SOA) standards, IT professionals (e.g., Software Developers) compose services using well-defined business processes to fulfill the business operations. The order of executing Web services in a business process is strictly predefined. Such service compositions are explicit.

- *Implicit.* End-users who have no IT background or are not familiar with service standards and tools frequently re-visit websites and use on-line services to perform daily tasks [91][108]. End-users potentially compose ad-hoc processes to meet their needs. Such an ad-hoc process consists of a set of Web services performed without a strict predefined order. For example, in today's on-line user experience, planning a trip on-line, a common task that can be performed by millions of end-users, could require end-users to potentially compose ad-hoc processes repeatedly [108] by visiting different travel websites and using various on-line services.

1.1 Major Steps in Service Composition

Conducting explicit or implicit service compositions contains the following major steps [70][89][105]:

Step 1: Defining processes for a task. Usually, an end-user's activity contains a number of logically linked tasks. Accomplishing each task can involve a composition of Web services. For example, end-users may need to finish the following tasks to plan a trip on-line from Toronto to Paris: purchasing flight tickets, booking hotels, buying event tickets, and buying medical travel insurances. Purchasing flight tickets can require a Web service selling flight tickets and another Web service for payment.

End-users need specify a goal (e.g., “buy a pair of shoes on-line”) that can expressed in the following forms [105]: (1) a formal or informal language; (2) an abstract process model, for instance, a business process model; (3) a design model, such as Unified Modeling Language (UML) diagrams.

End-users need create business processes for a goal. A business process has a number of logically linked tasks to achieve a business goal. Typically, a business process defines tasks, connections, roles, and resources. Process knowledge is critical for service composition systems to conceive formal business processes in explicit service compositions or ad-hoc processes in implicit service compositions. Process knowledge can be extracted from various sources, such as business analysts, service descriptions, on-line resources (e.g., eHow[2] an on-line how-to guide)

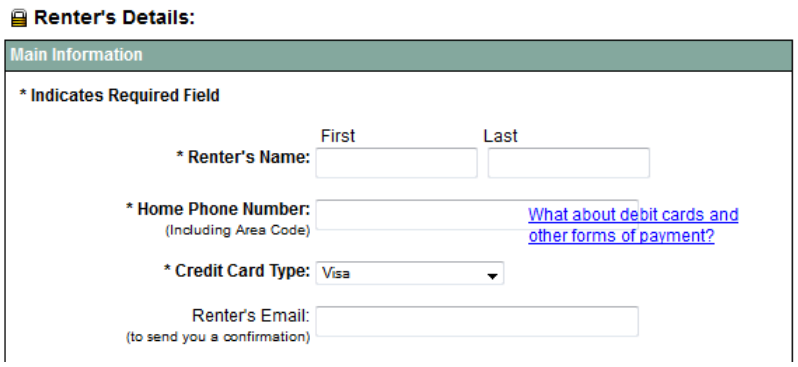
Step 2: Discovering relevant Web services. To fulfill a task, it is of importance to discover Web services that meet end-users’ requirements from a pool of Web services that are published by service providers. Service providers publish their Web services in a service repository, such as ProgrammableWeb[3] an on-line Web service repository. The published Web services are indexed by third-party engines (e.g., Google[4]) or service repository embedded search engines. To discover relevant Web services from repositories, end-users need search through these repositories by issuing queries. The engines analyze the information of Web services, such as service description, to return a set of Web services matching end-users’ requirements.

Step 3: Selecting proper Web services. Once a set of Web services that are functionally equivalent are returned, end-users need to select one service from the set of Web services for each task. A wide range of techniques have been utilized

[2]http://www.ehow.com/
[3]http://www.programmableweb.com/
[4]www.google.com

to help end-users select a proper Web service. Some research (e.g., [5][54]) analyzes Quality of Services (QoS) to rank Web services that can match with end-users' criteria. Reputation and trust-based mechanisms [94] are used to help end-users make the service selection, such as [58].

Step 4: Aggregating selected Web services. Web services are aggregated to allow end-users to create applications offering more capabilities. Composite services provide reusability of service functionality to end-users. The order of executing Web services in a composition can be obtained from process knowledge.


Renter's Details:

Main Information

* Indicates Required Field

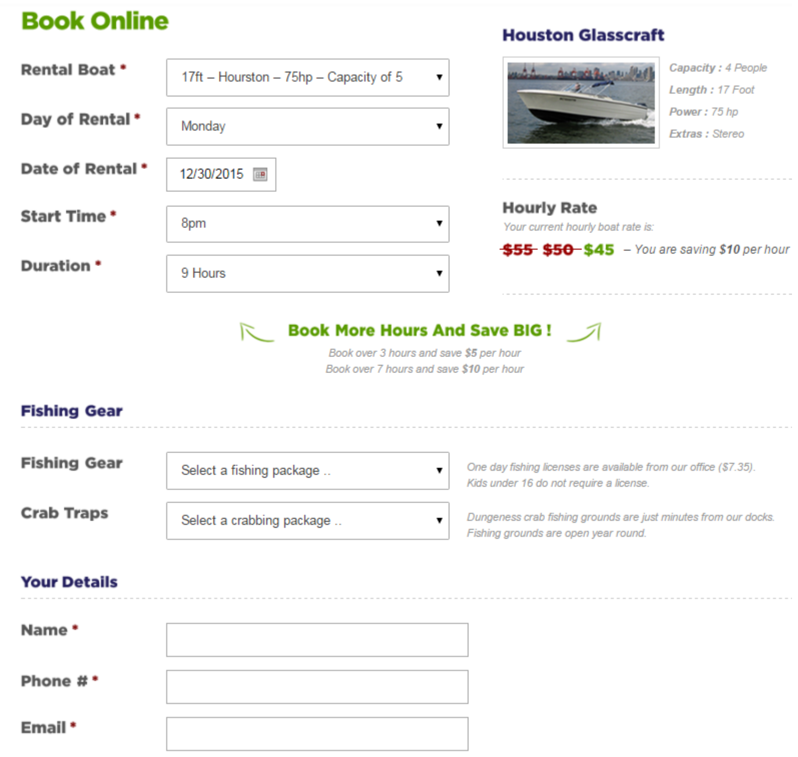
	First	Last
* Renter's Name:		
* Home Phone Number: (Including Area Code)		
* Credit Card Type:	Visa	
Renter's Email: (to send you a confirmation)		

What about debit cards and other forms of payment?

Figure 1.1: A sample screen shot of a web form requiring user's personal information to rent a car from a car rental service named Enterprise.

Filling values into Web services. Each task can be accomplished by invoking one service or composite services. Typically, a service has a set of input parameters taking values from users, such as the one illustrated in Figure 1.1. A user input is a piece of information entered into services. 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 [93].

To invoke a service, every required input parameter of the service should be filled



Book Online

Rental Boat *	17ft – Hourston – 75hp – Capacity of 5
Day of Rental *	Monday
Date of Rental *	12/30/2015
Start Time *	8pm
Duration *	9 Hours

Houston Glasscraft

Capacity : 4 People
Length : 17 Foot
Power : 75 hp
Extras : Stereo

Hourly Rate

Your current hourly boat rate is:

~~$55~~ ~~$50~~ $45 – *You are saving **$10** per hour*

Book More Hours And Save BIG !

*Book over 3 hours and save **$5** per hour*
*Book over 7 hours and save **$10** per hour*

Fishing Gear

Fishing Gear	Select a fishing package ..	*One day fishing licenses are available from our office ($7.35). Kids under 16 do not require a license.*
Crab Traps	Select a crabbing package ..	*Dungeness crab fishing grounds are just minutes from our docks. Fishing grounds are open year round.*

Your Details

Name *	
Phone # *	
Email *	

Figure 1.2: A sample screen shot of a web form requiring user's personal information to reserve a fishing boat from a boat rental website named Granville Island Boat Rentals.

with a proper value. Even though an explicit composition of Web services in a business process is well defined and connected (i.e., the input and output parameters are perfectly linked so that the values can be propagated among well defined Web services), initial user inputs are still required to run the composite services. In implicit service compositions, Web services are not linked seamlessly. End-users have to provide values to input parameters not chained with other parameters. For example, planning a fishing trip is an ad-hoc process for many end-users. Planning such a trip could involve two tasks: renting a car and reserving a boat, and each task involves at least several input parameters. For example, an end-user has to input their personal information, such as the ones illustrated in Figure 1.1, into Enterprise[5] a car rental website to book a rental car. When the end-user reserves a fishing boat on Granville Island Boat Rentals[6] a boat renting website, he or she is also required to enter his or her personal information as illustrated in Figure 1.2.

1.2 Problem Statement

Although the recently rapid advancement of service composition technologies make end-users able to conduct various complex on-line tasks, end-users are often required to enter the same information into input parameters repetitively [4]. End-users, such as a clerk filling out web forms to re-order office supplies or a graduate student booking a concert trip from Ottawa to Toronto, are wasting their precious time on entering the same information to input parameters repetitively. In most cases, input parameters can be filled with proper values by analyzing end-users' past data entry activities. Unnecessary interruptions caused by repetitively typing the same information to services decreases the efficiency of service composition and negatively impacts the user experience.

Thesis Statement: Repetitively typing the same information into services is a tedious job and impacts user experience negatively. Users' past data entry activities can be analyzed and utilized to help end-users fill in input parameters of operations in services accurately and efficiently.

A motivating example. Planning a trip can be conducted by millions of users. For instance, a trip from Toronto to Paris can involve two services: *Priceline*[7], a

[5]https://www.enterprise.ca/en/home.html
[6]https://www.boatrentalsvancouver.com
[7]*http : //www.priceline.com/*

travel-related product discount website and *ticketmaster*[8], 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. Each of the two services could require users to provide, at least, a dozen values (e.g., city name) to each of the services. Some values, such as end-user's name and payment information, are required by both services. It could be a cumbersome and annoying process for an end-user to fill the same information into web forms with multiple input fields repeatedly. Filling the same values into multiple services is tedious [101], especially when the number of services is high. Therefore, assisting end-users in filling out services, such as pre-filling or recommending values to input parameters for end-users, becomes critical to save end-users from this cumbersome process. Rukzio et al. [75] found that end-users are four times faster on smartphones when they just have to correct pre-filled form entries compared to entering the information from scratch.

We summarize the following challenges that end-users are currently facing:

- **Limited support of automatically collecting end-user's inputs**: The existing approaches do not store and organize all of the available user inputs entered previously by end-users. Some tools such as Mozilla Firefox Autofill Forms [27] need end-users to manually create their personal data and preferences records. Typically, a record contains a key and its corresponding value, such as a key "City" and its value "Toronto". The tools fill the values of records into web services (e.g., web applications) based on textual similarity between the keys of personal profile records and the input parameter of services. Indeed, some tools such as Google Chrome Autofill Forms [32] provide limited support

[8] *http : //www.ticketmaster.ca/*

on collecting user inputs. However, the existing approaches are limited to basic personal information, such as credit card information.

- **Lack of fully exploiting 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) [27]. 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 under 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.* Without the analysis of patterns and contextual information, existing approaches are not able to achieve a high accuracy and efficiency in filling out services. Although some tools, such as Google chrome auto-filling tool [32], can discover patterns of a few types of user inputs, but not contextual information.

- **Limitation on propagating end-user's inputs across different services.** Among the parameters of composed services, the semantically related parameters should be linked to facilitate the same information to be propagated among the relevant parameters. For example, buying flight tickets from *Priceline* and shopping concert tickets from *Ticketmaster* both require an end-user to enter the number of quantity of tickets to both services. Therefore, we should link the two input parameters requiring the same number in the two services. If one of the services is filled with a numeric value (i.e., the number of tickets), the value should be propagated to the other service.

- **Lack of approaches for pre-filling composed services.** End-users are often required to provide values to the input parameters of composed services. However, the information provided by the end-users can be redundant and repetitive, because the information can be previously entered into other services. For example, in the process of composing a set of RESTful services, developer account IDs issued by the service providers should be pre-filled if the end-user used the IDs before. Pre-filling input parameters of services can improve the efficiency of service invocation.

- **Limited ability of learning user input entry activities.** Different types of information, such as user contexts and patterns of user inputs, of user input entry activities can affect the performance of ranking previous user inputs. With the accumulation of user input entry activities over time, the importances of such varying types of information to end-users can change. Furthermore, users may enter different user inputs for an input parameter. The performance of ranking user inputs can be decreased, if changes in the importance of different information and the values of user inputs for input parameters are not reflected in the ranking models. However, existing approaches cannot automatically adjust the importance of a type of information and detect the changes of user inputs for input parameters.

1.3 Thesis Objectives

In a service composition, there are three types of linkages [92]: 1) User-to-Service occurs when a user invokes a service; 2) Service-to-Service occurs when a service invokes another service; and 3) Service-to-User occurs when the service needs human inputs or confirmation. In this thesis, we aim to facilitate these three linkages to overcome the aforementioned challenges. We propose a framework consisting a set of approaches to help end-users fill out services by analyzing end-users' previous data entry activities. We improve the process of assisting end-users in filling out services from the following aspects:

(1) Understanding the characteristics of input parameters.

The existing tools and approaches treat all input parameters equally, and the matching mechanism of existing approaches is only based on the string matching or semantic similarity calculation. If a match is identified, the existing approaches pre-fill a value to an input parameter. However, different user input parameters can have very varied natures and be required to be filled under different conditions. Understanding the different characteristics of parameters of different categories can help analyze, process and pre-fill the parameters differently to improve the accuracy of pre-filling. In this thesis, we conduct empirical studies to categorize input parameters and study the characteristics of them. We propose an ontology-based approach to automatically categorize parameters and fill values to the categorized input parameters.

(2) Using user contexts and usage patterns for parameter auto-filling. User contexts and usage information play an important role in the process of filling values to services. Various values could be entered for the same input parameter under different contexts. For example, an end-user could input two values for an input parameter requiring a "contact phone number", using one value for flight booking and another value for hotel reservation. User inputs can be used together to accomplish a task. When patterns of user inputs are formed, the recommendation can be improved. For example, a set of two UIs 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 user inputs as a pattern, if the set of user inputs are used together more than twice. In this thesis, we propose a framework that automatically detects user inputs and builds user profiles. The framework organizes and analyzes the collected user inputs to generate patterns of user inputs for assisting end-users in filling out services.

(3) Propagation of values among services. The existing approaches treat input parameters of services individually, and do not identify and link the relevant parameters for assigning values to input parameters of services. However, linking a set of semantically related parameters together is essential to maximize the value propagation among services and end-users. We propose an approach leveraging meta-data (e.g., task and textual information) of input and output parameters to link proper parameters together and maximize the propagation of user inputs across services. To increase the chance of discovering a suitable value for a user, the user information of other users, who share similar activities with the user, is taken into account. In [97], we redefined the pre-filling model [96] by adding the dimension of multi-user into the model. The extended approach can leverage and consume user inputs from multiple users.

(4) Ranking user inputs dynamically by learning user contexts and history. With the accumulation of user history, the interactions between users and parameters should be learned automatically to improve the accuracy of filling values to parameters of services. 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. Our framework (1) collects and organizes interactions between user inputs and input parameters with contextual information, then (2) learns and analyzes such interactions to rank user inputs for input parameters under different contexts. We propose 11 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 of user inputs and input parameters, we adopt the framework of learning-to-rank (LtR) [37][36] that consists of a set of supervised machine learning approaches to automatically build ranking models from training data built from user input entry activities. We leverage learning-to-rank algorithms initially designed for the research field of information retrieval [36] for ranking user inputs.

1.4 Thesis Overview

We now provide a brief overview of the thesis. We first introduce the background material.

Chapter 2: *Background and Definitions*

Before delving into the main body of this thesis, we first introduce background of our thesis and define key terms that will be used throughout the thesis.

Chapter 3: *Related Research*

To distinguish this thesis with prior research, we present the related research on assisting end-users in filling out services.

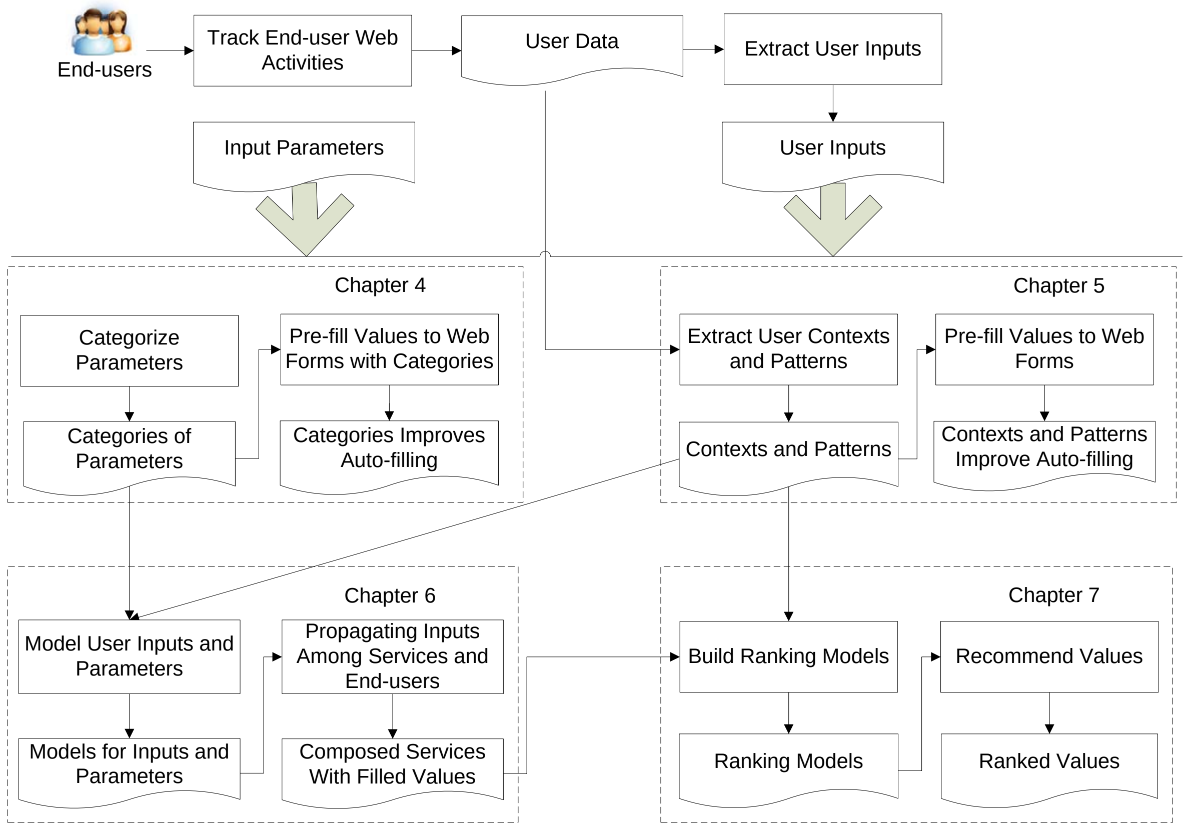

End-users
Track End-user Web Activities
User Data
Extract User Inputs
Input Parameters
User Inputs
Chapter 4
Categorize Parameters
Pre-fill Values to Web Forms with Categories
Categories of Parameters
Categories Improves Auto-filling
Chapter 5
Extract User Contexts and Patterns
Pre-fill Values to Web Forms
Contexts and Patterns
Contexts and Patterns Improve Auto-filling
Chapter 6
Model User Inputs and Parameters
Propagating Inputs Among Services and End-users
Models for Inputs and Parameters
Composed Services With Filled Values
Chapter 7
Build Ranking Models
Recommend Values
Ranking Models
Ranked Values

Figure 1.3: The main process in each chapter and the relations among four core chapters. The input parameters and user inputs are needed for each chapter.

Next, we switch to the main body of this thesis. We divide the main body into four core chapters. Each chapter sets out to achieve a thesis objective. Figure 1.3 shows the overall approach and relations among four core chapters. The four core chapters are listed as follows:

Chapter 4: *Understanding the Characteristics of Input Parameters*

In this chapter, we investigate the distinct characteristics of user input parameters. More specifically, we conduct an empirical study on the classification of input parameters. Utilizing our findings in the empirical study, we propose an automatic approach to categorize input parameters and pre-fill user inputs to web forms.

Chapter 5: *Using User Contexts and Usage Patterns for Parameter Auto-filling*

We introduce our comprehensive framework that analyzes user inputs and user contexts for assisting end-users in filling out services. More specifically, the framework tracks end-users' web activities, and extracts user contexts and usage patterns from the collected web activities. Analyzing and utilizing user contexts and usage patterns, the framework pre-fills previous user inputs to web forms. The framework lays out the technically fundamental infrastructure of this thesis.

Chapter 6: *Propagation of Values among Services*

To maximize the reuse of user inputs among services, linking semantically related parameters is necessary. Our empirical results in Chapter 4 and 5 show the importance of user contexts and categories of parameters in the process of filling in input parameters. In this chapter, we build a user input model to preserve user contexts for user inputs and a parameter context model to store information for input parameters. We introduce our approach that discovers and links parameters to propagate user inputs across different services among multiple end-users in service composition.

Chapter 7: *Ranking User Inputs*

Analyzing accumulated user history, containing interactions between users and parameters, can improve the accuracy of filling values to parameters of services. With the learned knowledge in the previous chapters, we propose a ranking-based framework to analyze and learn user history to recommend values to end-users for filling in input parameters. More specifically, we adopt the framework of learning-to-rank (LtR) [36][37] that consists of a set of supervised machine learning approaches to automatically build ranking models from training data that is built from user input entry activities. We leverage learning-to-rank algorithms initially designed for the research field of information retrieval [36] for ranking user inputs.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents background information. Chapter 3 summarizes the related literature. Chapter 4 introduces our study of categorizing input parameters. Chapter 5 describes our framework that automatically collects and analyze user inputs, user contexts and patterns of user inputs. In Chapter 6, we provide our approaches linking similar parameters and identifying end-users. Chapter 7 presents our ranking based framework that recommends user inputs to end-users. Finally, Chapter 8 concludes this thesis and outlines future work.

Chapter 2

Background

A web service is a software component that is used to support interactions among machines over a network. Various types of web services have distinct characteristics, such as interfaces. In this chapter, we first present the basic concepts of web services and service-oriented architecture in Section 2.1. Section 2.2 introduces different types of services in details.

2.1 Service-oriented Architecture

A web service is a self-contained, distributed, discoverable component that is designed to support machine-to-machine interaction over a network [81]. Service-oriented Architecture (SOA) [10][65][66] is an architectural paradigm and a logical way for designing and developing distributed software systems. In SOA, software applications are treated as services. Any new software application can be created by assembling existing distributed services across geographic locations, organizations, and platforms. SOA increases the reusability of software applications so that software organizations can develop softwares in a rapid way with low cost.

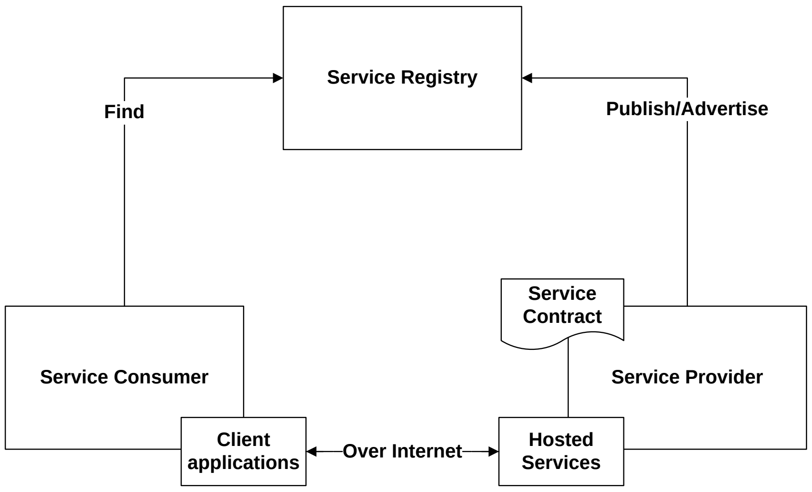


Figure 2.1: Interactions among service registry, service consumer, and service provider

In SOA, any component of a software can be converted as a service and published to a network so that the component can be reused by other software applications. Typically, SOA has three shareholders: service providers, service registry, and service consumers. Figure 2.1 shows the interactions among these three shareholders. The shareholders and their relations can be described as follows:

- **Service providers:** create services and host the services on providers' servers. Usually, the created services are maintained by service providers. Service providers are responsible for publishing and advertising the descriptions of the created services. For a service, a service contract is designed to formalize the details of the service, such as the contents, the price, and the delivery process. In particular, a service contract specifies the functional and non-functional requirements [64][89].

- **Service registry:** is a repository where service providers publish their created services. The service registry indexes and organizes the published services from service providers. Typically, a service registry is equipped with a search engine that facilitates the discovery of services.

- **Service consumers:** usually search for services in service registries, and integrate published services into service consumers' software applications.

- **Interactions among shareholders.** Typically, service providers first implement services and host the services on their servers. Next, service providers publish their web services with service contracts in service registries. To access the published services in service registries, service consumers first need search for proper web services that match consumers' requirements. Service consumers locate the services in the service registries, bind to the service over a network and execute the published services. A service is executed by sending a request that is formatted according to the service contract of the service.

2.2 Types of Web Services

Our research primarily focuses on assisting users in filling out web services that may not have user interfaces. Our research can handle the following three types of web services:

(1) *Simple Object Access Protocol (SOAP) based services* use *Web Services Description language* (WSDL) [102], an XML-based interface description language, to describe the functionality of web services. WSDL is often used in combination with SOAP.

(2) *RESTful services [71]* are proposed to simplify the development, deployment and invocation of services. Compared with SOAP-based services, RESTful services

are light weight. RESTful services use standard HTTP protocols and permit various data formats. Web Application Description Language (WADL) [61] is an XML description of HTTP-based web applications (i.e., typically REST web services). WADL is the REST equivalent of WSDL services.

(3) *Web Application services* are utilized by users to conduct various tasks using web applications (e.g., web forms). In this thesis, we consider web applications as a type of on-line services. We treat SOAP-based and RESTful services as the services with descriptions and web forms of web applications as the services without formal descriptions.

We introduce the above three types of services in detail in the following subsections.

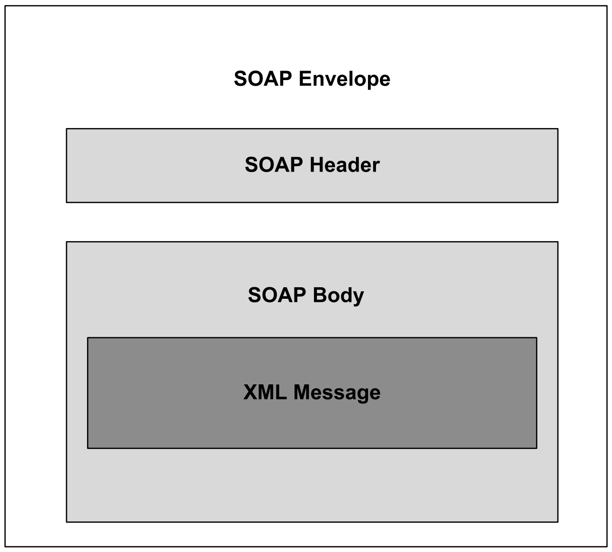


Figure 2.2: SOAP message format (edit from [89])

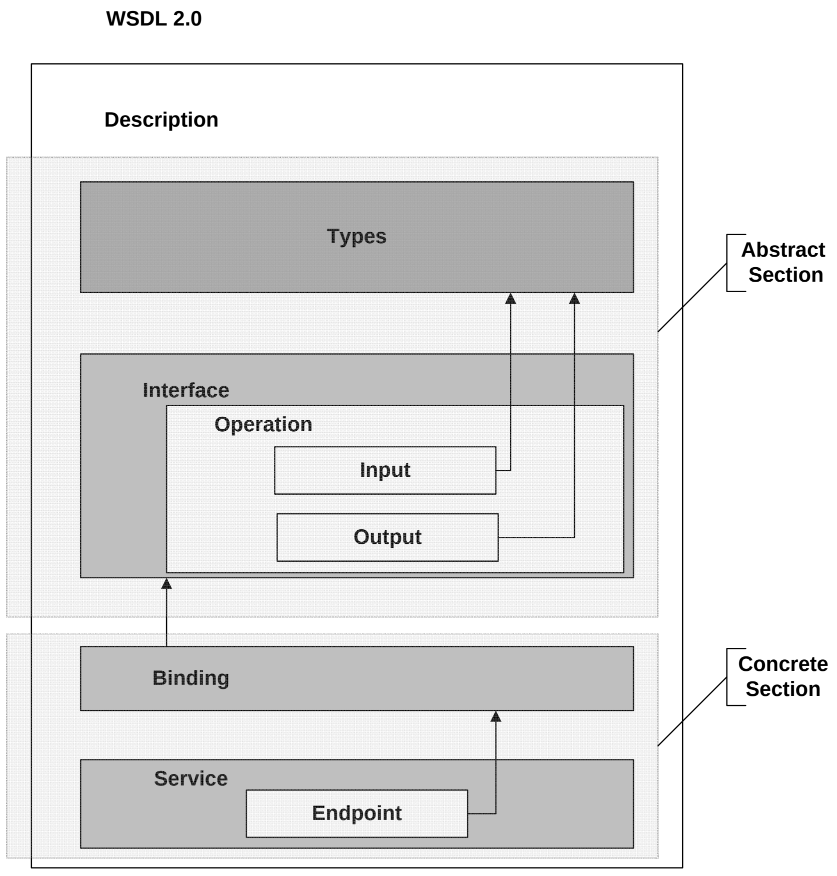


Figure 2.3: Basic structure of a WSDL 2.0 document (edit from [82])

2.2.1 SOAP-based Services

Simple Object Access Protocol (SOAP) is used for the communication between software applications of service consumers and web services of service providers. SOAP defines the message exchange paradigm between SOAP senders and SOAP receivers. SOAP can be utilized to create more complex interactions between SOAP senders and receivers. A SOAP message is written in XML. Figure 2.2 shows the basic SOAP message format. A SOAP message has a message header and a message body. The body is designed to contain mandatory information intended for the ultimate recipient of a message. The body element contains an XML document that represents return data, arguments, or error reporting.

The Web Services Description Language (WSDL) describes the functionality of a web service. Usually, WSDL is used with SOAP. Figure 2.3 shows the basic structure of a WSDL document.

The WSDL describes services as collections of network endpoints, or ports [82][89]. In a WSDL 2.0 document, the abstract section contains three elements: types, interface, operations.

- **Types.** This element describes the data types using type systems, such as XML Schema Definition (XSD) [109]
- **Interface.** It defines a web service, an abstract set of operations that are supported by the web service, and the messages which are used for performing the operations.
- **Operation.** An abstract description of an action that can be performed using the service. One operation has at least one input parameter and one output parameter.

The concrete section consists of two elements:

- **Binding.** It specifies the interface and defines the SOAP binding style. This element also defines the operations.
- **Service.** It describes a set of system functions that are open to web-based protocols.

2.2.2 RESTful Services

The software organizations open their resources (e.g., services) by defining RESTful services or Application Program Interfaces to a request-response message system. The resources can be services provided by the organizations. The RESTful APIs are usually described in plain HTML web pages. Client applications access the resources via direct HTTP requests and responses. The format of the requests and responses can be defined in various protocols.

A RESTful HTTP request must be associated with one of four standard HTTP methods [73]: GET, PUT, POST and DELETE.

- **GET.** A GET request is used to retrieve the data from service providers
- **PUT.** A PUT request indicates that the enclosed data in the request should be stored. If the enclosed data was not an already existing resource, the service provider can save the data on the servers.
- **POST.** Creating new resources requires the POST requests.
- **DELETE.** A DELETE request allows service consumers to tell a service provider to delete the data.

A typical RESTful HTTP request includes 1) a HTTP method (e.g., GET); 2) a domain address of API server; 3) a name of RESTful API method; 4) a format of return data; 5) a set of parameters of the method. The (domain + method name) can also be referred as *resource URL*. For example, a Twitter RESTful request of retrieving the 2 most recent mentions is listed as follows:

GET https://api.twitter.com/1.1/statuses/mentions_timeline.json?count=2

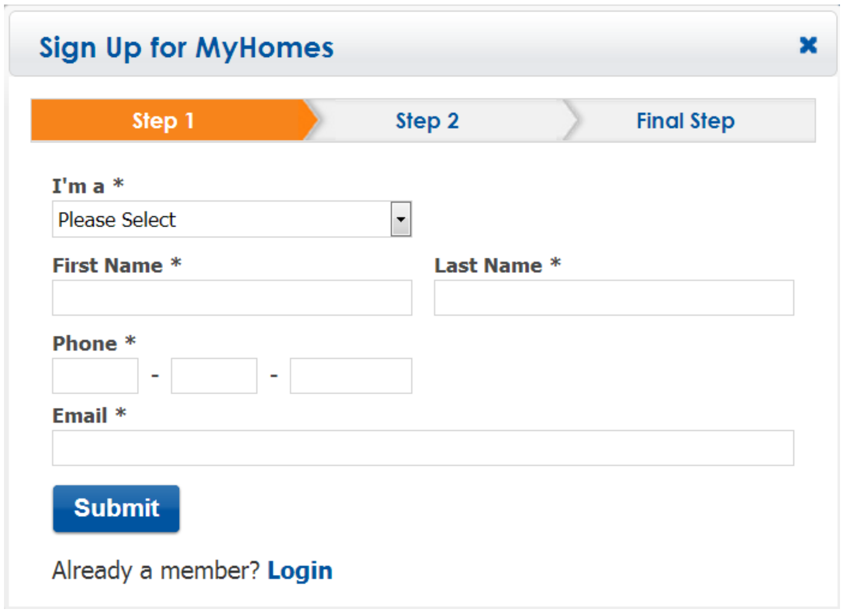
GET is the standard HTTP method. *api.twitter.com/1.1* is the address of the API server. *statuses/mentions_timeline* is the method name. *json* is the format of return data. *count* is a parameter specifying the number of tweets to be retrieved.

2.2.3 Web and Mobile Application User Interface

Nowadays, end-users with no IT background compose ad-hoc processes by integrating different tasks of web applications to meet their needs. Typically, an end-user interacts with web applications by entering data to them. Usually, the web user interface of a web application is written in HTML (or XHTML) and Cascading Style Sheets (CSS) and contains web forms. A web form consists of a set of input UI components, such as text fields, radio buttons and check-boxes. A user interface can be converted into HTML DOM tree [87]. The developers use an HTML tag <form> to define a web form. An input UI component is expressed as an <input> element in an HTML DOM tree. The <input> element has several attributes, such as *name* specifying the name of an <input> element, *type* defining the type of the <input> element (e.g., a button or checkbox), and *value* stating the value of an <input> element. Furthermore, an <input> element is associated with a human-readable label, such as "Renter's Name" illustrated in Figure 1.1. We can obtain all the information of a web form and its associated input UI components by parsing the HTML source code.

Web pages are mainly built with HTML and Cascading Style Sheets (CSS), and can be converted into an HTML Document Object Model (DOM) tree [1]. A Web form is defined by an HTML FORM tag <form> and the closing tag <*/form*>. A web form usually consists of a set of input elements, such as the text fields in the

[1]http://www.w3schools.com/htmldom/dom_nodes.asp



Sign Up for MyHomes

Step 1 Step 2 Final Step

I'm a *

Please Select

First Name *

Last Name *

Phone *

Email *

Submit

Already a member? Login

Figure 2.4: A sample screen shot of a web form requiring user's personal information to sign up a personal medical website, webMD

registration form of a personal medical website, webMD[2] illustrated in Figure 2.4, to capture user information. An input element is defined by an HTML INPUT tag <input> specifying an input field where the user can enter data. The input element can contain several attributes, such as *name* specifying the name of an <input> element, *type* defining the type <input> to display (e.g., displayed as a button or checkbox) and *value* stating the value of an <input> element. An <input> element can be associated with a human-readable label, such as First Name. This information can be accessible by parsing HTML source code.

There are three types of mobile applications:

- *Native Apps:* The native apps are developed specifically for one platform such as Android[3], and can access all the device features such as camera.

[2]http://www.webmd.com/
[3]http://www.android.com/

- *Web Apps:* The web apps are developed using standard Web technologies such as Javascript. They are really websites having *look and feel* like native apps. They are accessed through a web browser on mobile devices.

- *Hybrid Apps:* The hybrid apps are developed by embedding HTML5 apps inside a thin native container.

An Android application is encapsulated as an Android application package file (APK) [4]. An APK file can be decoded into a nearly original form which contains resources such as source code (e.g., Java) and user interface layout templates in XML files that define a user interface page or a user interface component if it is not a HTML5 application. In this study, we only study the native mobile applications because the user interface templates in XML files can be obtained by decoding APK files.

2.3 Chapter Summary

In this chapter, we introduce fundamental concepts of web services. More specifically, we define the web services and the paradigm of Service-oriented Architecture (SOA), and describe the structures of different types of web services.

[4]http://en.wikipedia.org/wiki/APK_(file_format)

Chapter 3

Related Work

In this chapter, we summarize the related work focusing on four areas related to this thesis. First, we analyze the techniques and approaches that assign values to web services. Second, we introduce prior research on identifying input-output data flow of web services. Third, we present contexts models for web services. Last, we delve into the machine learned ranking models that are adopted and utilized in our framework to rank user inputs.

3.1 Assigning Values to Web Services

Summarizing and analyzing the prior research on assigning values to web services can help us discover the shortcomings and motivate our thesis. In this section, we first introduce the approaches for assigning values to web services for end-users. Second, we present the value filling approaches for service testing and deep web crawling.

3.1.1 Filling Out Web Services for End-users

The research of assigning values to web services for end-users mainly proposes two types of techniques to aid users in filling in input parameters.

Pre-filling Values to Input Parameters

Several industrial tools (e.g., [1][27][50]) have been developed to pre-fill web forms. For example, Mozilla Firefox Add-on Autofill Forms [27] automatically fills stored values into input parameters. However, it requires users to manually create entries for filling profiles and enter values for the pre-defined entries. RoboForm [72], LastPass [50], and 1Password [1] 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 revisits a page.

Similar to the above industrial tools, another group of approaches (e.g., [7][28][48] [100][101]) improve the accuracy of filing out web forms by creating a personal space storing structured information for users or data binding schemas. The data binding schemas are essential techniques helping connect user interface elements with data objects of applications. This technology needs an ontology to perform the data integration. Instead of focusing on custom ontology, some binding schemas rely on the emergence of open standard data types, such as Microformats [45] and Micodata [60]. For example, Winckler *et al.* [101] propose an approach using a pervasive information space for storing and collecting user's personal data for filling out forms. To support data interoperability between the personal space and web forms, the web forms can be annotated with Microformats [44] used for describing data with semantics by developers or experienced web users using an annotation tool. The proposed approach was demonstrated by a beta-user. Firmenich *et al.* [28] propose an approach based on web form augmentation [29] to support a straightforward interaction between third-party web forms and users' personal information management systems for web form filling. Araujo *et al.* [7] propose a concept-based approach that mines the semantical concepts of the web form fields and uses the relations of concepts to automatically fill out web forms with user inputs collected from 6 subjects.

Some studies (e.g., [83]) require apriori ([6]) tagging of websites, or a manually crafted list that includes the labels or names of input element to describe a semantic concept. All of the above industrial and academic tools require manual work from users

Some academic studies such as [24][34][47][86] propose several approaches to automatically pre-fill web forms. For example, Toda *et al.* [86] propose a probabilistic approach using the information extracted from data-rich text as the input source to fill values into web forms. WebFeeder [24] leverages iMarcos [40] 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.* [47] 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. However, no end-users were involved in the empirical studies for testing the effectiveness of the proposed approach.

All of the above approaches and tools are not context-aware. They do not store and analyze contextual information and patterns of user inputs. However, our framework consisting of a set of approaches can track and store all of the information of the interactions between user inputs and input parameters.

Recommending Values to Users for Filling in Input Parameters

Some other industrial and academic approaches (e.g., [32][34]) recommend a list of values to users for input parameters. For example, Google Chrome Autofill forms (Chrome-fill) [32] 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.* [38] develop a non-intrusive assistant, a learning and prediction system, to provide values for blank fields in a form. CCFU [4] uses a joint probability distribution based on Bayesian network to calculate the contextual relevance for predicting a value for a target field. CCFU uses a greedy structure learning algorithm over a set of previously filled-out forms to learn the dependencies between fields in a form. However, the above approaches are not context-aware. Hartmann *et al.* [34] propose an approach to map the user contextual information with the user interfaces. The context-aware user interfaces (UI) 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 [26]). However, the above approaches utilize conventional ranking models which cannot adjust the importance of ranking features over time. However, our ranking framework utilizing machine learned ranking models can analyze and learn users' data entry activities to automatically adjust the importance of ranking features.

3.1.2 Assigning Values to Web Services for Deep Web Crawling and Testing

The deep web is a vast information repository, such as articles in a university library, that is hidden behind HTML forms and is usually not indexed by automated search engines, such as Google[1] [57]. To collect the information from the deep web, an end-user needs perform a form submission with valid input values. Automatically crawling deep web requires value assignments to web forms. For example, Kabisch *et al.* [43] propose a deep web integration system, named VisQI, to transform web interfaces into hierarchically representations classified into different domains for linking different interfaces. Kriegel *et al.* [46] introduce an approach to classify similar websites as sets of feature vectors. Nguyen *et al.* [63] propose an approach using learning classifiers to extract element labels from web form interfaces. Mapping the labels to elements correctly is the key step to link similar web interfaces. Similar to deep web crawling, automatically testing web services needs entering values to web services. AbuJarour *et al.* [3] propose an approach to generate annotations for web services by sampling their automatic invocations. They use four sources, such as random values, to automatically assign values for input parameters of web services.

3.2 Identifying Input-Output Data Flow of Services

The approaches from research studies (e.g., [15][53][85]) identify input-output data flow for chaining services to help end-users complete their tasks. For example, Thomas *et al.* [85] propose an approach to model the flow of messages and methods in a web

[1] www.google.com

service transaction to represent distributed web services. Gerede *et al.* [15] propose techniques to automate the design of composite e-services. They use Roman model [16] to represent e-services and consider the Roman model as activity-based finite state automata. Li *et al.* [53] propose an apropos to identify and verify fundamental relationships (i.e., support and correlate) in the patterns of web service message exchange flows by analyzing the abstract specification of WSDL service operations.

All of the above approaches do not take into consideration the propagation of user inputs across services and only use the name of input and output parameters for chaining services. In this thesis, we extract and analyze more information about parameters to link similar parameters to propagate user inputs across services.

3.3 Context-aware Models for Web Services

In this thesis, we propose approaches that extract different features and contexts for input parameters to link similar parameters. In this section, we summarize some research (e.g., [11][56][62][78][106]) on proposing context-aware models for web services. For example, Mrissa *et al.* [62] propose a context model for composing services. Blake *et al.* [11] propose a context model containing user's contextual information for identifying relevant services to end-users. Xiao *et al.* [106] propose a context modeling approach dynamically dealing with various context types and values. They adopt ontologies to enhance the meaning of context values and automatically discover the relations among context values. Aviv *et al.* [78] analyze various web service classification methods for composition and propose a context-based semantic matching method for ranking services. Zakaria *et al.* [56] propose an approach that utilizes four layers: policy, user, web services, and resource to manage behaviors of web services seamlessly by combining contexts and policies. However these proposed context models are not designed for propagating user inputs across services and assigning values to services. In this thesis, we propose a context-aware data model to store and organize user inputs and a parameter model for modeling input and output parameters of web services to link similar parameters.

3.4 Chapter Summary

In this chapter, we survey prior research related to our thesis. We first summarize existing approaches that aim to automatically assign values to services. Next, we present a set of approaches for linking parameters and several context models for modeling services. Last, we introduce the basic concepts of machine learned ranking algorithms and different learning-to-rank models.

Chapter 4

Understanding the Characteristics of Input Parameters

When end-users invoke web services to accomplish their tasks, input parameters can require different types of values as inputs. Varied types of input parameters should be analyzed separately. Some parameters should be pre-filled with a value and some parameters should not. However, existing approaches of filling out services do not distinguish distinct types of parameters, and quite often fill in some input parameters which should not be provided with a value in the first place. In this chapter, we conduct empirical studies on input parameters of web and mobile application interfaces to study the characteristics of different input parameters.

Chapter Organization. The rest of the chapter is organized as follows. Section 4.1 gives an overall introduction of this chapter. Section 4.2 presents the proposed approach for categorizing input parameters. Section 4.3 introduces the empirical studies.

Section 4.4 discusses the threats to validity. Finally, section 4.5 concludes the chapter and outlines some avenues for future work.

4.1 Introduction

It is crucial to improve the accuracy of automatic value pre-filling by understanding the characteristics of different user input parameters. In order to reuse user inputs effectively, we must understand both what the input parameters and user inputs are and how different input parameters can be filled or suggested with user inputs. The existing tools and approaches treat all input parameters equally, and the matching mechanism of existing approaches is only based on the string matching or semantic similarity calculation. If a match is identified, the existing approaches pre-fill a value to an input parameter. However different input parameters can have very varied natures. For example, using a voucher to buy a gift card is a common task. The voucher number can only be valid for a single purchase, therefore it may not be suitable for value pre-filling second time. In a task of booking a flight ticket, the input fields for departure and destination cities should not be pre-filled with previous user inputs without knowing user contexts, but the end-user's name can be pre-filled, since the end-user's name does not change in most cases.

In this chapter, we investigate different characteristics of various types of input parameters to improve the process of filling out services. We conduct an initial manual empirical investigation on the input parameters of 30 popular websites and 30 Android mobile applications from Google Play Android Market [1] to discover distinct types of input parameters. Based on the results of our initial study, we identify

[1]https://play.google.com/store?hl=en

and propose four categories for input parameters from web and mobile applications. The proposed categories offer a new view for understanding input parameters and provide tool developers with guidelines to identify pre-fillable input parameters and necessary conditions for pre-filling. To the best of our knowledge, we are the first to propose categories for input parameters to explore the characteristics of user input parameters. We conducted another empirical study to verify the effectiveness of proposed categories. The second study was conducted on a new dataset including 50 websites from three domains and 100 mobile applications from five different categories in Google Play Android Market.

The results of our empirical study show that our categories are effective to cover all the input parameters in a representative corpus.

To express the common parameters and their relations, we propose an ontology model. We use the proposed ontology model to carry the category information for input parameters. We propose a WordNet-based approach that automatically updates the core ontology for unseen parameters from new applications. The results of our empirical study show that our approach obtains a precision of 88% and a recall of 64% on updating the ontology to include the unseen parameters on average. Moreover, we propose an ontology-based approach to identify a category for input parameters automatically. On average, our approach for category identification can achieve a precision of 90.5% and a recall of 72.5% on the identification of a category for parameters on average. We test the effectiveness of our proposed categories on improving the existing approaches. We build a baseline approach which does not distinguish the different characteristics of input parameters, and incorporate our proposed categories with the baseline approach to form a category-enabled approach. We compare two approaches through an empirical experiment. The results show that our approach can improve the baseline approach significantly, i.e., on average 19% in terms of precision.

4.2 Our Proposed Approach for Categorizing User Input Parameters

In this section, we first present an ontology model for expressing common user input parameters and their relations. Second, we propose an automatic approach for updating the ontology. Third, we propose an ontology-based approach for categorizing input parameters.

4.2.1 An Ontology Definition Model

In this study, we build an ontology to capture the common user input parameters and the five relations among parameters. Figure 4.1 illustrates the main components of ontology definition model and their relations.

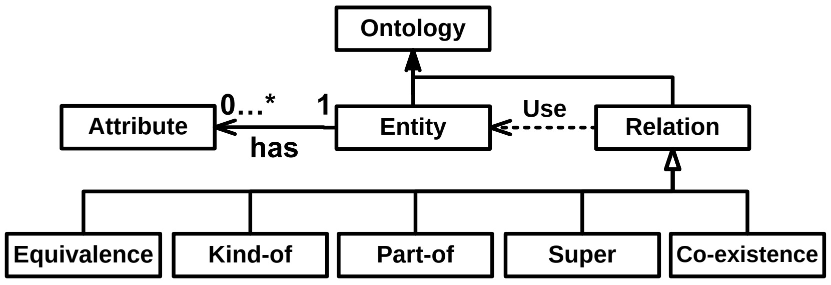


Figure 4.1: Components of ontology definition model

The components are listed as follows:

- *Entity:* is an input parameter from a website or mobile application, or a concept description of a group of resources with similar characteristics.

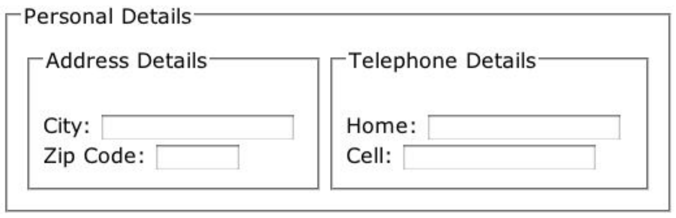


Figure 4.2: An example of 4 user inputs parameters: City, Zip Code, Home and Cell

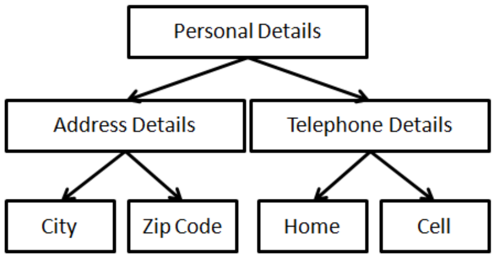


Figure 4.3: An example ontology of personal details containing four user inputs parameters: City, Zip Code, Home and Cell

- *Attribute:* is a property that a parameter can have. There are four attributes:
  - *Category.* The category defines the category information of a parameter.
  - *Label.* This attribute stores the *Label* (i.e., a human-readable description) of an input field from websites or mobile applications.
  - *Coding Information.* This attribute stores the HTML coding information (i.e., websites) or XML templates (i.e., mobile applications).
  - *Concepts.* This attribute stores the concepts related to the parameter.
- *Relation:* defines various ways that entities can be related to one another. The five relations are listed as follows:
  - *Equivalence.* Two entities are the same concept such as Zip Code and Postal Code.

– *Kind-of.* One entity is a kind-of another one. For example, Home (i.e., home phone) is a kind of Telephone Details illustrated in Figure 4.2.

– *Part-of.* One entity is a part-of another one, such as Zip Code and Address Details illustrated in Figure 4.2.

– *Super.* One entity is a super of another. The super relation is the inverse of the Kind-of relation. For example, Telephone Details is a super of Home (i.e., home phone) illustrated in Figure 4.2.

– *Co-existence.* Two entities are both a part-of an entity and they are not in the relation of equivalence, such as City and Zip Code illustrated in Figure 4.2.

Usually the user input parameters are terminal concepts [107], such as *Phone Number* and *Price*, which have no sub-concepts. During the process of ontology creation, we use the UI structure and semantic meanings of the parameters to identify the relations. If two input elements have the same parent node in an HTML DOM tree, their relation is co-existence, and the relation between the two input parameters and their parent HTML DOM node is part-of. For example, the user input parameters *City* and *Zip Code* co-exist and they have a part-of relation with Address Details in Figure 4.2. Figure 4.3 shows the visualization of the ontology that is built based on the example in Figure 4.2.

4.2.2 Our Approach of Updating Ontology

Once the initial ontology based on the proposed ontology model (Section 4.2.1) is established, an automatic approach for updating the ontology is required to add a new parameter into the ontology. We use WordNet [26], a large lexical database for

English and containing semantic relations between words, to identify the relations between the new parameter and the existing ones in the ontology. The following relations of words defined in WordNet are used to identify our 5 relations:

1. If two words have the same synsets, they have a same semantic meaning, we convert it to Equivalence relation.
2. **Hypernym** shows a kind-of relation. For example, car is a hypernym of vehicle. We convert it to a kind-of relation.
3. **Meronym** represents a part-whole relation. We convert it to a part-of relation.
4. **Hyponym** defines that a word is a super name of another. We convert it to super relation. Hyponym is the inverse of hypernym meaning that a concept is a super name of another. For example, vehicle is a hyponym of car.
5. We use the part-of relation to identify the co-existence relation. If a word with another word both have a part-of relation with a same word, this word and the other word have co-existence relation.

4.2.3 Our Approach for Category Identification

It is important for a pre-filling approach to know the category of a user input parameter automatically. In this section, we introduce our ontology-based approach for identifying a category of an input parameter in details. Our approach uses the ontology definition model proposed in Section 4.2.1. Our approach uses two strategies which are listed as follows:

- *Concept-based Strategy.* This strategy relies on an ontology of user input parameters to identify a category for a user input parameter automatically. If two input parameters have the same concepts, these two input parameters belong to the same category.

- *UI-based Strategy.* This strategy relies on the design of the UI layouts. We consider two user input parameters are the nearest neighbours to each other if their input fields are contained in the same parent UI component. Figure 4.2 shows an example of 4 user input parameters: *City, Zip Code, Home and Cell.* The *City and Zip Code* are the nearest neighbours to each other since they have the same parent UI node which has a label *Address Details*. We assume that if all the neighbours of a user input have been categorized and belong to the same category, there exists a high chance that they belong to the same category.

Given an ontology having a set of parameters, which is denoted as O = $< P_1^O, \ldots, P_n^O >$, where n is the number of parameters, and an input parameter P, our approach uses the following steps to identify a category for P:

- Step 1. We use WordNet synsets to retrieve synonyms of the concepts of the parameter P to expand the concept pool (i.e., a bag of words) of P and the concept pool of each P_i^O, where $1 \leq i \leq n$.

- Step 2. We identify the P_j^O (where $1 \leq j \leq n$) whose concept pool has the concepts which are the synonyms of (or identical to) any concepts in the concept pool of P (i.e., having the same semantic meaning).

- Step 3. If multiple parameters in the ontology are identified for P in Step 2, we choose the parameter sharing the most common concepts with the concept pool of P as the identical parameter to P. We assign the category of the chosen parameter to P

- Step 4. If no parameter is identified in Step 2, we apply the UI-strategy on the given ontology O and input parameter P. We identify the neighbors of P from its UI and repeat Step 1-3 to identify a category for every neighbor. If any neighbor of P cannot be categorized (i.e., no parameters in the ontology have the same concepts as the neighbor does) or the neighbors of P have different categories, we cannot categorize P. If all the neighbors of P belong to a same category, we assign this category to input parameter P.

4.3 Case Studies

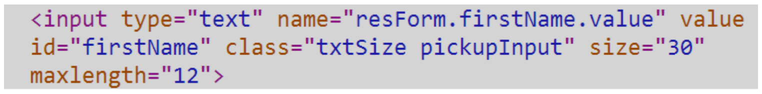
We conduct two studies on different datasets. The first study is designed to study the different characteristics of parameters and propose categories for input parameters. The second empirical study is designed to evaluate the effectiveness of the proposed categories and the approach for categorizing input parameters.

4.3.1 Manual Classification of Input Parameters

We conduct an initial study to understand the nature of user input parameters and categorize the parameters. Understanding the different characteristics of parameters of different categories can help analyze, process and pre-fill the parameters differently to improve the accuracy of pre-filling. In this section, we introduce the study setup, the analysis method and the findings of our empirical investigation.

Data Collection and Processing.

The study is conducted on 30 websites (i.e., 15 shopping websites and 15 travel planning websites) and 30 mobile applications (i.e., 15 shopping apps and 15 travel apps). We collect the user input parameters from web and mobile applications in different ways.



```
<input type="text" name="resForm.firstName.value" value
id="firstName" class="txtSize pickupInput" size="30"
maxlength="12">
```

Figure 4.4: A sample screen shot of source code of an input field.

Collecting input parameters from websites: First, we manually visit the website to bypass the login step. Second, we use *Inspect Element*[2] of Google Chrome[3] to collect the following information of a user input parameter:

- **Label:** The value of the label (i.e., the descriptive text shown to users).
- **Attributes:** The values of the attributes of the input element such as id and name. For example, Figure 4.4 shows the source code of the input field First Name in Figure 1.1.

Collecting input parameters from mobile applications: Instead of running each mobile application, we use Android-apktool [4], a tool for reverse engineering Android APK files, to decode resources in APKs to readable format. We build a tool to extract the information of a form from UI layout XML files, then collect the information related to a user input parameter in the same way as we collect from websites.

Data cleaning: The extracted information is needed to be cleaned for further processing. For example, the extracted information for the input parameter *First Name*

[2]https://developers.google.com/chrome-developer-tools/docs/elements
[3]https://www.google.com/intl/en/chrome/browser/
[4]https://code.google.com/p/android-apktool/

in Figure 1.1 is {First Name, resForm.firstName.value, FirstName, text} need to be cleaned. We remove the duplicated phrases and programming expressions. For the given example above, the output after cleaning is {First Name, res, Form, value, text}.

Data processing: We manually process the information of parameters as follows: First, we identify the concepts from the information of a user input parameter. A concept is a semantic notion or a keyword for describing a subject (e.g., "taxi" and "city"). Second, the parameters having the same concepts or same semantical meanings are merged and considered as one parameter which we save all the unique concepts for. For example, two input parameters from two different flight searching websites, one with the label *Departure* and the other one with the label *Leave* should be merged and considered as one parameter having two concept words *Departure* and *Leave*.

Results of the First Empirical Study.

We collected 76 and 32 unique user input parameters from websites and mobile applications respectively. Due to the fact that the user input parameters are repeated among different applications, we observe that the fewer unique parameters are identified when we study more applications studied. The 76 parameters extracted from the websites (i.e., shopping and travel) contain all the 32 parameters from mobile applications from the same domains. This is due to the fact that mobile applications usually are the simplified versions of corresponding websites.

After studying the user input parameters, we found that input parameters can be categorized into four categories. The four categories are listed as follows:

- *Volatile Parameters:* The values for this type of parameters cannot be reused. They can be further categorized into two sub-categories:
  - One-Time Parameters: The values of this type of parameters are valid for one interaction, such as coupon number. This type of parameters should not be used for pre-filling at all.
  - Service-Specific Parameters: The values of this type of parameters can only be used for a specific service (e.g., a website or a mobile application). For example, Member ID of the BESTBUY Reward Zone[5] in Canada can only be used for "Sign In" function or "Account Set Up" function of reward service. When end-users receive a membership card, they need enter the Member ID to set up an account in BESTBUY Reward Zone. This type of parameters can be pre-filled, however the parameters cannot be reused by different services.
- *Short-time Parameters:* The values of this type of parameters change with high frequency. For example during the course of conducting an on-line clothes shopping, an end-user may use the different colors as the search criteria to find the most suitable clothes from different services for the user, during a short period of time, the value of the color can be pre-filled. This type of parameters can be pre-filled and reused by different services, however it is extremely hard to pre-fill this type of parameters unless some conditions are met. For example, an end-user starts searching for clothes, the value of the cloth color should be pre-filled only if the end-user has not switched to a new task.

[5]https://www.bestbuyrewardzone.ca/

- *Persistent Parameters:* The values of this type of parameters do not change over a long period of time. For example, the gender of the end-user and permanent home address. The persistent parameters are suitable for pre-filling and being reused across different services.

- *Context-aware Parameters:* There exist some user input parameters which are context-dependent, such as the user's location and current local time. This type of parameters can be pre-filled based on user's contextual information. The value of a context-aware parameter can be obtained from two types of data sources:

  – *Direct Data Sources.* The context data is directly available and accessible with little computation, such as entries in Google calender, time from mobile phone clock, "my" To-do lists from task manager and a friend's preferences on Facebook.

  – *Analyzed Data Sources.* With the accumulation of user history, additional contextual data can be mined from history data such as user behaviors. For example, a user always books a round-trip business class flight ticket from Toronto to Los Angeles for business trips and a round-trip economy class flight ticket from Toronto to Vancouver for personal trips.

In this chapter, we consider only the user input parameters whose values can be obtained directly from available sources.

4.3.2 Empirical Study for Evaluating the Effectiveness of Our Approach

The goals of the second empirical study are to: 1) examine the representativeness of the proposed categories of parameters; 2) validate the effectiveness of the approach of updating ontology; 3) test the effectiveness of the proposed approach for category identification; 4) validate the effectiveness of our proposed categories on improving existing approaches of pre-filling values to web forms.

Data Collection and Processing.

We conduct our second empirical study on 45 websites from three different domains: Finance, Healthcare and Sports (i.e., 15 websites from each domain) and 100 Android mobile Applications from five different categories: Cards& Casino, Personalization, Photography, Social and Weather in Google Play Market (i.e., 20 applications from each studied category). The studied websites and mobile applications are different from the ones studied in our first empirical study. We use the same approach as discussed in section 4.3.1 to collect user input parameters from web and mobile applications. In total, there are 146 and 68 unique user input parameters from web and mobile applications respectively.

Research Questions.

We presents four research questions. For each research question, we discuss the motivation of the research question, analysis approach and the corresponding findings.

RQ1. Are the proposed categories of parameters sufficient to cover the different types of user input parameters?

Motivation. The existing tools and approaches do not distinguish the characteristics of user input parameters. The parameters are processed equally. Therefore the lack of knowledge on the user input parameters negatively impacts the accuracy of pre-filling approaches, and is the main reason for limited support of reusing parameters within one application or across multiple applications. Categorizing the users input parameters can help us in understanding the different characteristics of parameters. Each category of parameters has its unique characteristics which need to be fully understood.

In Section 4.3.1, we identified 4 categories for user input parameters via a manual empirical study. In this research question, we validate the proposed categories to see if they can explain the further unseen user input parameters collected in the empirical study.

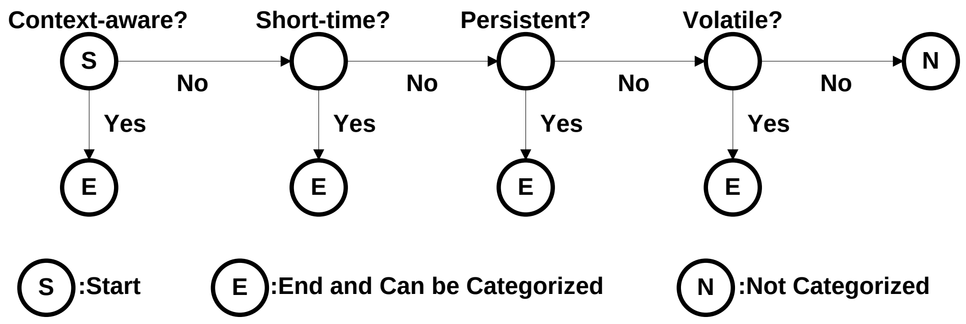


Figure 4.5: The decision tree for the judgment process

Analysis Approach. To answer this question, we study the user input parameters collected from the 45 websites and 100 mobile apps. For each user input parameter, we manually process it to see whether a user input parameter belongs to any category or not. The judgment process is based on a decision tree as shown in Figure 4.5. When we process a parameter, we first decide whether it is a context-aware parameter or not. If no, we further decide whether the parameter can be categorized

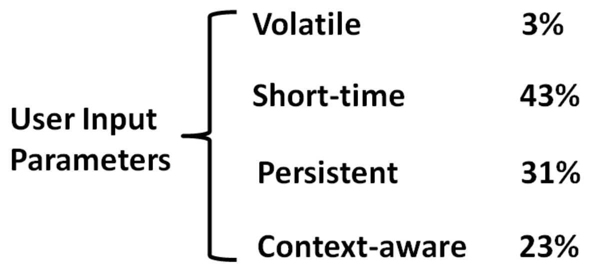


Figure 4.6: The percentage of each category

as a short-time parameter or not. If no, then we decide whether the parameter is a persistent parameter or not. If no, finally we decide whether the parameter is a volatile parameter or not. If it is still a no, the parameter cannot be categorized by using our proposed categories of input parameters.

Results. Figure 4.6 shows that all of the unseen user inputs parameters can be categorized into our proposed categories and there is no "not Categorized". More specifically, 43% of the parameters are short-time and only 3% of the parameters are volatile parameters. The results suggest that our categories are enough to describe all the unique user input parameters in our empirical study.

Summary of RQ1: The proposed four categories of parameters: Volatile, Short-time, Persistent, and Context-aware, can describe all unique user input parameters from our analyzed websites and mobile applications.

RQ2. Is the proposed approach for updating ontology effective?

Motivation. Usually the ontologies are created manually by web filling tool builders or contributors from the communities of knowledge sharing. It is helpful to have an automatic approach to update and expand the ontologies. We test the effectiveness of the WordNet-based approach of updating ontology proposed in Section 4.2.2.

Analysis Approach. We conduct two experiments to answer this question. In the first experiment, our approach for updating ontology uses the input parameters from one domain of applications to update the ontology to include the parameters from other domains of applications. In the second experiment, our approach for updating ontology uses the parameters from one domain of applications to update the ontology to include the parameters from the same domain of applications.

To conduct *experiment 1*, we construct an ontology[6], denoted as $O^{Initial}$, using the 76 user inputs parameters from the first empirical study (Section 4.3.1). Second, we use the ontology $O^{Initial}$ as the existing ontology to include the parameters of 146 parameters from web applications and 68 parameters from mobile applications (Section 4.3.2). Third, we compute the precision and the recall of the approach using Equation (4.1) and Equation (4.2) respectively. The precision metric measures the fraction of the placed user input parameters that are in the correct place (i.e., having the right relations with other parameters), while the recall value measures the fraction of all the user input parameters that are correctly placed.

$$precision = \frac{|\{Correctly\ Placed\ Params\} \bigcap \{Placed\ Params\}|}{|\{Placed\ Params\}|} \tag{4.1}$$

$$recall = \frac{|\{Correctly\ Placed\ Params\} \bigcap \{Placed\ Params\}|}{|\{All\ Params\}|} \tag{4.2}$$

The goal of *experiment 2* is to see whether we can obtain a better result if we build a domain dependent ontology for each domain in our empirical study. To conduct *experiment 2*, we conduct the following steps on the 45 websites:

[6] https://dl.dropboxusercontent.com/u/42298856/icwe2014.html

1. randomly select 5 of 15 Finance websites and construct the domain specific ontology for the user input parameters in Finance using the user input parameters from the selected websites.

2. use the domain dependent ontology of parameters in Finance as an existing ontology and perform a manual updating on the existing ontology to include the user input parameters from the rest 10 Finance websites.

3. apply the automatic ontology updating approach on the parameters from the rest 10 Finance websites to verify its performance.

We replicate the previous steps on 15 Health Care websites and 15 Sports websites. We conduct our analysis on mobile applications in the same way as we do on websites, the only difference is that we randomly select 5 mobile applications from each domain. We compute the precision and the recall of the approach using Equation (4.1) and Equation (4.2) respectively.

Table 4.1: Results of our approach for updating ontology using domain-specific ontology (Experiment 2)

Type	**Domain**	**Precision(%)**	**Recall(%)**
website	Finance	83	66
website	Health Care	78	69
website	Sports	95	65
mobile	Cards&Casino	92	74
mobile	Personalization	87	48
mobile	Photography	90	62
mobile	Social	85	42
mobile	Weather	97	85

Results. In the experiment 1, our approach for updating existing ontology obtains a precision of 74% and a recall of 42% on websites, and a precision of 82% and a recall of 30% on mobile applications. We further investigate our results to gain further insights regarding the low recalls. We found that 35 of 146 website user input parameters and 25 of 68 mobile applications user input parameters can be found in the ontology constructed using the input parameters collected during the initial empirical investigation. The low recalls indicate that it is hard to use one general ontology as the existing ontology to include the parameters from other domains automatically, although the precisions are considerably high.

In experiment 2, Table 4.1 shows that our approach for updating ontology obtain a precision of 85% and a recall of 67% on websites, and a precision of 90% and a recall of 62% on mobile applications. The results of our approach in **experiment 1** can be improved by using the domain-specific ontology as the existing ontology. On average, the approach can be improved by 11% in terms of precision and 25% in terms of recall on websites, and 8% in terms of precision and 32% in terms of recall on mobile applications.

Summary of RQ2: The results of our empirical study show that our proposed approach for updating existing ontology is effective in including new input parameters in the existing ontology.

RQ3. Is our approach of category identification effective?

Motivation. After showing the representativeness of our categories for input parameters, we propose an ontology-based approach to identify a category for an input parameter. In this section, we investigate the effectiveness of our approach for identifying a category for an input parameter.

Analysis Approach. To assess the performance of the proposed approach, we proceed as follows:

1. We use the domain-specific ontologies constructed in RQ2 as the training ontologies.
2. We categorize the parameters in ontologies by assigning a category to each parameter.
3. We locate the nearest neighbours for the input parameters which are not in the training ontologies.
4. We apply our approach on the parameters in Step 3

We compute the precision and the recall of the approaches using Equation (4.3) and Equation (4.4) respectively. The precision metric measures the fraction of the categorized user input parameter that are categorized correctly, while the recall value measures the fraction of all the user input parameters that are categorized correctly.

$$precision = \frac{|\{Correctly\ Categorized\ Params\} \bigcap \{Categorized\ Params\}|}{|\{Categorized\ Params\}|} \tag{4.3}$$

$$recall = \frac{|\{Correctly\ Categorized\ Params\} \bigcap \{Categorized\ Params\}|}{|\{All\ Params\}|} \tag{4.4}$$

Results. Table 4.2 shows the precision and recall values of our approach to identify a category for input parameters. On average, our approach can achieve a precision of 90% and a recall of 77% on websites, and a precision of 91% and a recall of 68% on mobile applications. Our approach works well on the input parameters from mobile applications in the domain of weather, because usually the weather mobile apps relatively have fewer number of input parameters and very similar functionalities.

Table 4.2: Results of our approach of category identification

Type	Domain	Precision(%)	Recall(%)
website	Finance	90	79
website	Health Care	92	82
website	Sports	89	70
mobile	Cards&Casino	90	78
mobile	Personalization	91	58
mobile	Photography	95	62
mobile	Social	77	55
mobile	Weather	100	88

Summary of RQ3: Our proposed automatic approach for assigning a category to an input parameter in our dataset is effective.

RQ4. Can the proposed categories improve the performance of pre-filling?

Motivation. The existing pre-filling approaches (e.g., [7][101]) treat all the input parameters equally. They do not take into consideration the characteristics of input parameters. These approaches are usually based on the string matching or semantic similarity calculation between the user inputs and the labels of input parameters (e.g., input fields). Essentially they fill a value to an input field as long as a match between a user input and an input field is identified, however the value required by the input parameter may not be suitable for pre-filling due to curtain conditions as mentioned in Section 4.3.1. In this research question, we evaluate the effectiveness of our categories on improving the existing techniques of reusing user inputs.

Analysis Approach. To answer this question, we build a baseline approach which adopts the approach in [7]. The baseline approach [7] pre-fills values to input parameters based on the semantic similarity between user inputs and textual information (e.g., words mined from labels and the attributes of HTML DOM elements defining the input parameters in the user interface) of input parameters. The baseline approach calculates the similarity between user inputs and labels of input parameters using WordNet [26]. The stopword removal, word stemming and non-English words removal are conducted on the textual information of input parameters to identify meaningful words. Then, WordNet is used to expand each word with synsets (i.e., a set of words or collation synonyms) to retrieve synonyms of the found terms.

We build our approach by enriching the baseline approach with the proposed categories for input parameters (Baseline + Categories). We evaluate the improvement of applying our categories with the baseline approach. Our approach uses the ontology-based approach proposed in Section 4.2.3 to identify a category for an input parameter to see whether the input parameter is suitable for pre-filling or not. If the input parameter is suitable for being pre-filled, our approach pre-fills an input parameter. We compute precision using Equation (4.5) and recall using Equation (4.6) to measure the improvement. The precision metric measures the fraction of the pre-filled user input parameters that are pre-filled correctly, while the recall value measures the fraction of all the user input parameters that are pre-filled correctly.

$$precision = \frac{|\{Correctly\ Pre-filled\ Params\} \bigcap \{Pre-filled\ Params\}|}{|\{Pre-filled\ Params\}|} \tag{4.5}$$

$$recall = \frac{|\{Correctly\ Pre-filled\ Params\} \bigcap \{Pre-filled\ Params\}|}{|\{All\ Params\}|} \tag{4.6}$$

To pre-fill input parameters using the user previous inputs, we collect user inputs through our input collector [99] which modifies the Sahi[7] tool to track user's Web activities. The author of this thesis used the tool to track his inputs on web forms from three domains: Finance, Health and Sports. We apply both the baseline approach and our approach on the 45 websites.

Table 4.3: Results of the evaluations of our categories on improving the baseline approach of filling input parameters.

Domain	**Baseline**		**Baseline+Categories**	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Finance	50	34	64	36
Health	41	22	67	30
Sports	36	16	53	20

Results. Table 4.3 shows that our approach incorporating categories of input parameters can improve the baseline approach on average 19% in terms of precision. We further inspect the results and found that our approach can reduce the number of wrong filled values compared with the baseline approach. Some of the wrong filled values should not be pre-filled to the input parameters in the fist place.

Summary of RQ4: Our empirical results show that identifying categories of input parameters can improve the current practice of filling out web forms.

4.4 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [111].

Construct validity threats concern the relation between theory and observation. In this study, the construct validity threats are mainly from the human judgment in

[7]http://sahi.co.in/

categorizing the parameters and ontology construction. We set guidelines before we conduct manual study and we paid attention not to violate any guidelines to avoid the big fluctuation of results with the change of the experiment conductor.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study.

4.5 Chapter Summary

Reusing user inputs efficiently and accurately is critical to save end-users from repetitive data entry tasks. In this chapter, we study the distinct characteristics of user input parameters through an empirical study. We propose four categories, *Volatile, Short-time, Persistent and Context-aware*, for input parameters. The proposed categories help pre-filling tool builders understand that which type of parameters should be pre-filled and which ones should not.

In this chapter, we propose an ontology model to express the common parameters and the relations among them. In addition, we propose a WordNet-based approach to update the ontology to include the unseen parameters automatically. We also propose an approach to categorize user input parameters automatically. Our approach for category identification uses the proposed ontology model to carry the category information.

Through an empirical study, the results show that our proposed categories are effective to explain the unseen parameters. Our approach for updating ontology obtains a precision of 88% and a recall of 64% on updating the ontology to include unseen parameters on average. On average, our approach of category identification achieves a precision of 90.5% and a recall of 72.5% on the identification of a category for parameters on average. Moreover, the empirical results show that our categories can improve the precision of a baseline approach which does not distinguish different characteristics of parameters by 19%. Our proposed categories of input parameters can be the guidelines for pre-filling tool builders and our ontologies can be consumed by existing tools.

Chapter 5

Using User Contexts and Usage Patterns for Parameter Auto-filling

User contexts and patterns of user inputs can make a significant impact on the accuracy of filling values to input parameters [34]. However, the existing approaches do not exploit the information related to user contexts and patterns of user inputs, such as the information of the tasks that a user input was filled in for, that is useful to facilitate the auto-filling task. In Chapter 4, our empirical results show that the categories of input parameters can improve the process of pre-filling input parameters. In this chapter, we study whether analyzing user contexts and patterns of user inputs can improve the process of pre-filling input parameters. We propose an intelligent auto-filling framework that propagates user's inputs across different web applications, collects end-user's inputs, identifies user's usage patterns, and detects end-user's contexts. Our framework clusters input user interface (UI) components to form semantically similar groups of input UI components.

Chapter Organization. The rest of the chapter is organized as follows. Section 5.1 presents an overall introduction of this chapter. Section 5.2 presents an overview of our proposed framework. Section 5.3 introduces the empirical study. Section 5.4 discusses the threats to validity. Finally, Section 5.5 concludes the chapter and outlines some avenues for future work.

5.1 Introduction

During the ad-hoc composition of web applications (i.e., web sites with intensive web forms), user contexts and patterns of user inputs can play a major role in the process of filling values to the web applications. For example, physical location, a user context, can impact the process of filling out services. An end-user may only want to enter the credit card information into a service when the end-user is at home. However, existing tools and approaches suffer from the following drawbacks:

- **Limited support of collecting and analyzing end-user's inputs automatically**: The existing approaches do not store and organize all of the available user inputs entered previously by end-users. Some tools such as Mozilla Firefox Autofill Forms [27] need end-users to manually create their personal data and preferences records. Typically, a record contains a key and its corresponding value, such as a key "City" and its value "Toronto". The tools fill the values of records into web services (e.g., web applications) based on textual similarity between the keys of personal profile records and the input parameter of services. Indeed, some tools such as Google Chrome Autofill Forms [32] provide limited support on collecting user inputs and analyzing patterns of user inputs. However they are limited to basic personal information, such as credit card information.

- **Limited propagation of end-user's previous inputs to different web applications used by an end-user**: The existing approaches do not exploit all of the available information of web services and user inputs for auto-filling. For example, a collection of web applications can be linked implicitly by end-users during the task integration of an ad-hoc process. An end-user usually purchases concert tickets from *ticket liquidator* [1] an event ticket booking website and then books bus tickets from *Greyhound* [2] a bus ticket booking website. These two web applications are linked implicitly by the end-user. If the end-user books two tickets from *ticket liquidator* and a few minutes later tries to buy bus tickets from *Greyhound*, the framework should be aware that the end-user would buy two bus tickets from *Greyhound* and pre-fill the number of required tickets (i.e., two tickets) into *Greyhound* by learning from the data entered in the previous step of the implicit ad-hoc process composition. However existing approaches cannot support this type of intelligent filling.

In this chapter, we propose a comprehensive framework to address the aforementioned limitations. The proposed framework lays the basic technological infrastructure for this thesis. Our framework automatically detects user inputs and builds user profiles containing personal and preference information. Then our framework organizes and analyzes the user inputs to generate the patterns of user inputs for auto-filling during the task integration of ad-hoc processes. The framework explores the similarity between input user interface (UI) components among different web services.

[1]http://www.ticketliquidator.com
[2]https://www.greyhound.ca/

Understanding the relationships between input UI components is a crucial step for automatically propagating user inputs between different services. Furthermore, our proposed framework is context-aware and dynamically detects the changes of user's contexts, such as user's current location. Our framework supports the information exchange across user interface (UI) components (e.g., drop-down box and text fields). In addition, the framework recommends a new list of values to end-users when the end-users modifies pre-filled values. To the best of our knowledge, we are the first to propose a comprehensive solution on service auto-filling during the task integration of ad-hoc processes. We conduct an empirical study to evaluate the effectiveness of our framework for web form filling on a large and well-known dataset. The empirical results show that our clustering approach for identifying similar input UI components achieves a precision of 89% and a recall of 83%. Furthermore, our framework achieves an average precision of 74.5% and an average recall of 58% on pre-filling web forms, and a precision of 82.25% and a recall of 68.4% on suggesting values to end-users if the end-users edit the initial pre-filled values.

5.2 Our Proposed Framework

This section presents our proposed framework. First, we introduce the major components of our framework. Second, we present the interactions among the components. Figure 6.1 shows the architecture of our framework and interactions among its components.

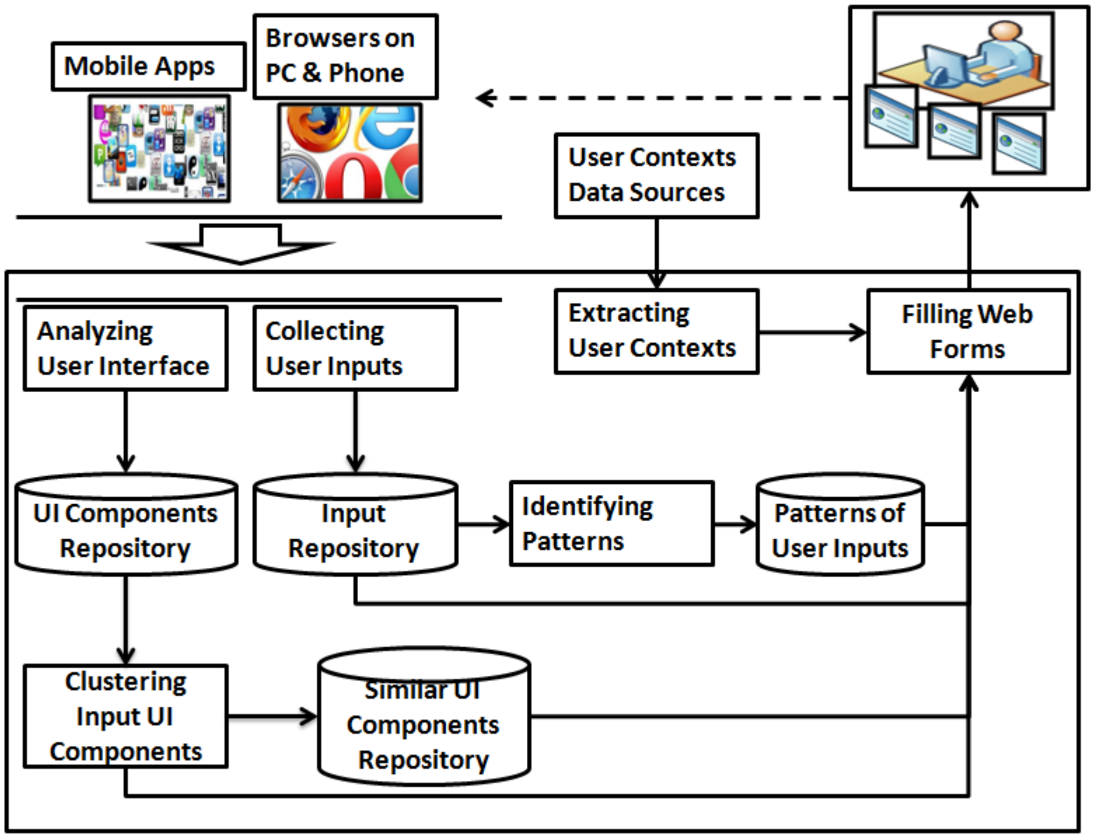


Figure 5.1: Overview of our proposed framework for filling out web forms

5.2.1 Major Components of Our Framework

Our framework consists of six major components listed as follows:

- ***Analyzing User Interfaces.*** This component parses the web pages visited by an end-user and then extracts information (e.g., a label) of the input user interface (UI) components expecting an input from an end-user. All of the collected entities are stored in *UI Components Repository*.
- ***Collecting User Inputs.*** This component detects and collects the inputs from end-users, stores the user inputs with associated UI components for further

analysis. All of the collected user inputs are stored in *Input Repository.*

- ***Identifying Patterns of User Inputs.*** The framework analyzes the collected user inputs in *Input Repository* and identifies patterns of user inputs.

- ***Identifying Similar Input UI Components.*** The collected input user interface components in *UI Components Repository* are clustered into semantic groups. The input UI components within a group are similar to each other and potentially require the same value. The semantic groups are stored in *Similar UI Components Repository.*

- ***Extracting User Contexts.*** The framework detects and extracts user contexts from external sources such as a user's calender.

- ***Filling in Web Form.*** The framework conducts web form filling using the collected user inputs, patterns of user inputs, clusters of similar input UI components, and user contexts such as entries in Google Calendar.

5.2.2 Interactions Among Components

In this sub-section, we introduce the interactions among framework components through the scenarios of end-users' interactions with input UI components of web forms. After an end-user opens a web page, we summarize three scenarios:

Scenario 1. Before the end-user enters a value into a web form: The framework conducts the following analysis:

First, the framework starts ***analyzing the user interface*** to extract the textual information of input UI components.

Second, the extracted information of an input UI component is used to check whether the input UI component is already stored in *UI Components Repository* or not.

- If it is not in *UI Components Repository*, the framework conducts the following steps:
  - Step 1. The input UI component with its extracted information is stored in *UI Components Repository.*
  - Step 2. The framework starts *identifying similar input UI components* by clustering the input UI components in *UI Components Repository* and stores the clusters of similar input components in *Similar UI Components Repository.*
  - Step 3. The framework searches for the similar input UI components, associated with some previous user inputs in *Input Repository*, from *Similar UI Components Repository.* If such input UI components are identified, the framework saves them as a *Candidate UI Set.* Otherwise the input UI component cannot be pre-filled with any previous inputs.
- If it is already stored in *UI Components Repository*, the framework checks whether the input UI component is associated with any previous user inputs in *Input Repository* or not.
  - If the input UI component is associated with a set of previous user inputs which can be entered by the end-user at different times, the framework saves the set of user inputs as a *Candidate Set.*

– If the input UI component is not associated with any previous user inputs in *Input Repository*, the framework repeats **Step 3** to identify the similar input UI components.

Third, the framework starts ***extracting user contexts*** from different data sources such as Google Calendar.

Fourth, the framework starts ***filling in web forms*** using the extracted user contexts and the *Candidate Value* or *Candidate UI Set*.

Scenario 2. When the end-user enters a value into a web form: When the end-user enters a set of inputs to a web form of the web page and submits the web form, the framework starts ***collecting the user inputs***. The framework stores the set of inputs and the information of UI components receiving the user inputs in the *Input Repository*. Then the framework starts ***identifying patterns of user inputs*** by analyzing the collected user inputs in the *Input Repository*.

Scenario 3: When the end-user is not satisfied with a pre-filled value: The end-user modifies the pre-filled value. During the modification of the initially pre-filled value, the ***filling in web forms*** component dynamically recommends a list of possible user inputs which are potentially suitable for the input UI component.

The remainder of this section elaborates on each component of our framework.

5.2.3 Analyzing User Interfaces and Collecting User Inputs

Due to the high diversity in the structure of HTML web pages, locating a label for an input user interface component is a challenging and difficult task. A label of a user interface component can be positioned in different locations. Some websites explicitly link a label and its corresponding user interface component using the HTML

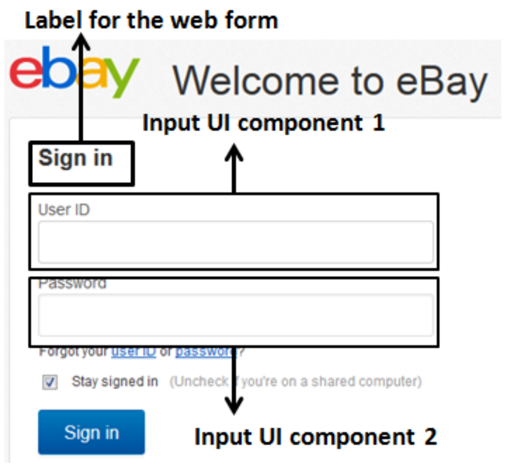


Figure 5.2: An example of a HTML page

tag <label>. A label can be assigned to an element either by using the "for" attribute, or by placing the element inside the <label> element. In this case, it is easy to identify the label for the user interface component. However, we found that most of websites do not link the label and its corresponding user interface component explicitly. We have to use heuristics to identify the label representing an input element. We adopt the approach used in [90].

The approach in [90] analyzes different parts of a web page. A web page consists of a set of opening and closing HTML tags and these tags separate the web page content into different partitions. The approach traverses and analyzes the HTML DOM tree [87] to identify the partitions with a label. When reaching a partitioning element, such as $< p >$, $< br >$, and $< hr >$, the approach creates a label and adds the text node inside the partitioning element to the label. If the partitioning element has a child partitioning element, the approach links the text nodes appearing under the child partitioning with the child partitioning element. For each input element, the approach calculates the distance between the input element and the text nodes to determine which text node is for the input element. The distance is calculated based on the number of nodes visited to reach the input element from a text node. The approach chooses the label with the least distance as the label for the input element.

Analyzing User Interfaces

Our framework extracts input user interface (UI) components of web pages visited by an end-user. The framework parses the HTML DOM [87] of a web page to extract input user interface (UI) components. For an input UI component such as component 1 in Figure 5.2, the following information is extracted:

- *Textual information of a web form.* We extract the information of a web form where the input UI component locates as follows:
  - The value of the name attribute of HTML tag $< form >$. In XHTML, we extract the id attribute instead.
  - The label or text (a human readable textual information) assigned for the web form. For example, the label "Sign in" illustrated in Figure 5.2 is a label for the sign-in form of ebay[3].
  - The URL of the web page which the web form is in.
- *Information of the Input UI Component* We store the following information:
  - *Type.* We keep the type of the UI component such as drop-down box and text field.
  - *Coding Information.* We store the values of the attributes of the input element of HTML DOM such as id, name, text and hint. We locate the

[3]www.ebay.com

HTML INPUT tag <input> specifying a user interface component (i.e., the component receiving input from end-users) to extract the coding information.

– *Label.* We extract the label associated with the user interface component.

The extracted information is stored in *UI Components Repository* which holds a collection of input UI components. We define a unique id for every input UI component and web form. A unique id of an input UI component is used to locate it in the *UI Components Repository.* The unique id of a web form is used to identify the neighbor input UI components for an input UI component. If two input UI components reside in the same web form, these two input UI components are neighbors to each other. We generate a unique id for every input UI component and every web form. We combine the values of an input component: *Label*, *Coding Information*, *Type* to be an id. We use the Textual information of a web form as its id. In the *UI Components Repository*, we also store the *Unique IDs of neighbors* for a input UI component.

Collecting User Inputs

To make a framework intelligent, it is important to track and collect user inputs from web forms. When an end-user enters a user input to an input UI component, we extract the following information:

- *User Input.* We store the value of the user input entered by an end-user.
- *Information of Input User Interface Component.* We store the exact same types

of information of an input user interface component as the ones stored in Section 5.2.3 (i.e., analyzing user interfaces).

- *Time Stamp.* After the end-user enters a user input and submits the user input through a web form, we store the time of the submission. Usually a set of user inputs entered into a web form has the same time stamp because these user inputs are submitted together by end-users.

We use the same approach described in Section 5.2.3 (i.e., analyzing user interfaces) to generate a unique id for an input user interface component. We use the generated id to locate the input component in *UI Component Repository*, then we store the user input, the id of its associated input UI component, and the *Time Stamp* in *Input Repository*.

We modify and extend an open source tool called Sahi [76] used for automating web application testing to collect end-user's information and analyze user interfaces. Sahi injects Javascripts into web pages using a proxy and the Javascripts help automate the actions of web applications. It monitors the user's actions (e.g., click, submit and search), remembers the possible user's inputs (e.g., search queries, end-user's name, age, gender), and generates a log. By default, the Sahi tool does not include the detailed information (e.g., descriptive information) of an input user interface component in the log, we extend the tool to extract more information of the input user interface components.

5.2.4 Identifying Patterns of User Inputs

When an end-user conducts web tasks such as on-line shopping for a while, a lot of user inputs can be collected. Among these user inputs, there exist possible patterns

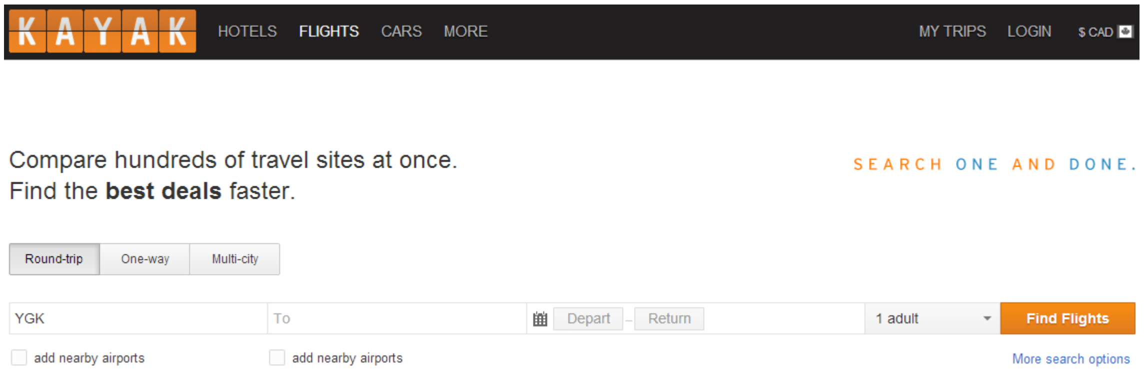

Compare hundreds of travel sites at once.
Find the **best deals** faster.

Figure 5.3: A sample screenshot of a web form on Kayak[4], a travel website.

of user inputs. The user inputs within a pattern can be filled together into a web form if any user input is selected by the end-user. For example, an end-user searches for cheap flights using the search form in Figure 5.3 of KAYAK[4], a travel website. The end-user always searches for a round-trip business class flight ticket from Toronto to Los Angeles and a round-trip economic class flight ticket from Toronto to Vancouver.

In this study, we mainly focus on recognizing the patterns of user inputs from web forms. To identify the patterns of user inputs from the *Input Repository* and *UI Component Repository*, we conduct the following steps:

- *Step 1.* Every user input in the *Input Repository* has a time stamp. We use the time to group user inputs. A group of user inputs are submitted at the same time.

- *Step 2.* Every input UI component in the *UI Component Repository* has an id of a web form. We use the web form id to group input UI components. A group of input UI components are from the same web form.

- *Step 3.* We use the unique ids of input UI components to identify the user

[4]https://www.ca.kayak.com/

inputs associated with the groups of input UI components generated in Step 2. The output of this step is the mapping between a web form (i.e., a set of input UI components) and groups of user inputs entered to the web form at different times.

- *Step 4.* We identify the frequent patterns of user inputs of a web from using the BI-Directional Extension based frequent closed sequence mining (BIDE) pattern mining algorithm proposed in [95]. The BIDE mines maximal frequent patterns. For example, a web form in Figure 5.3 takes user inputs to search for cheap flights. Over the time, for example, there could be three sets of user inputs entered into the web form. The three sets of user inputs are:
  Set 1: "round-trip, LA, NYC, Thr 13/3, Tue 22/4, 1 adult".
  Set 2: "round-trip, Toronto, Paris, Sat 26/4, Fri 27/6, 2 adults".
  Set 3: "round-trip, Toronto, Paris, Tue 9/12, Wed 31/12, 2 adults".
  We set BIDE to identify any pattern of user inputs which shows up at least twice. Therefore the pattern generated for the above example is "round-trip, Toronto, Paris, 2 adults".

5.2.5 Identifying Similar Input UI Components

The framework clusters the input UI components stored in the *UI Components Repository* to form semantic clusters using their textual information. The input UI components within a cluster are semantically similar and potentially require the same value. Each input UI component has the *Coding Information* and *Label*. We merge all of the descriptive textual information together from *Coding Information* and *Label* to construct *a bag of words* [59] for each input component.

Our clustering approach weights each word. A bag of words of an input UI component can be represented as a vector of weight values. We use the Cosine Similarity [77] to calculate the similarity between two vectors of input UI components and Quality Threshold Clustering Algorithm [39] to cluster similar UI Components.

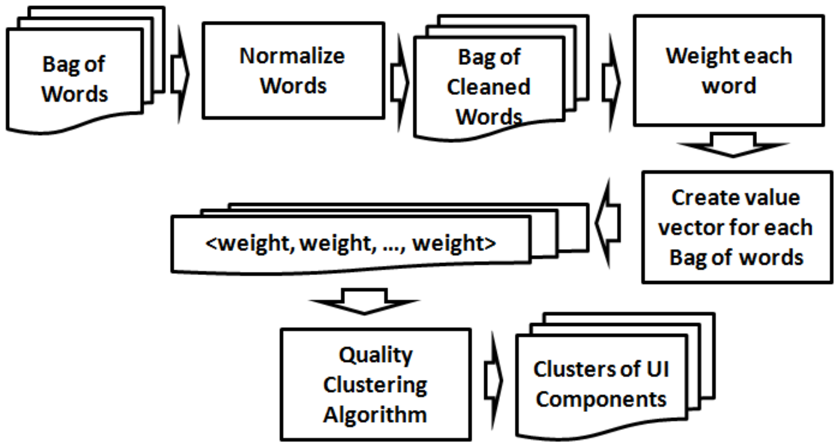

Bag of Words
Normalize Words
Bag of Cleaned Words
Weight each word
Create value vector for each Bag of words
<weight, weight, ..., weight>
Quality Clustering Algorithm
Clusters of UI Components

Figure 5.4: Overview of our approach for clustering UI components

Figure 5.4 shows the major steps of our clustering approach consisting of the following steps:

1) Words Normalization: We split the words in the bags of words stored in the *UI Components Repository* by special characters such as "_" and capital characters if applicable. We use Wordnet [26] an lexical database for English to remove non-English words. The stop words removal, word stemming are conducted in this step to normalize the words of each component. A bag of "cleaned" words is generated for each component.

2) Value Vector Creation: The Term Frequency and Inverse Document Frequency (TF-IDF) value are used to calculate the weight for each word in the bag of "cleaned" words. The *Term Frequency* (TF) is the frequency of a word appearing in a document.

We refer to a bag of words as a document. The *Inverse Document Frequency* (IDF) diminishes the weight of words that occur very frequently in the whole corpus and increases the weight of words occur rarely. We calculate the TF as shown in Equation (5.1) and the IDF as shown in Equation (5.2) for each word.

$$TF = \frac{|\{occurrences\ of\ a\ word\ in\ the\ document\}|}{|\{total\ words\ in\ the\ document\}|} \tag{5.1}$$

$$IDF = \log \frac{|\{documents\ in\ the\ corpus\}|}{|\{documents\ having\ the\ word\}|} \tag{5.2}$$

We use a TF-IDF value as the weight of a word. We calculate the weight for each word as shown in Equation (5.3).

$$weight = TF \times IDF \tag{5.3}$$

Once every word has a weight, a bag of words is expressed as a vector of values, and an input UI component is represented by a vector of values.

3) Quality Threshold Clustering [39]: We use the Quality Threshold (QT) Clustering algorithm to cluster the similar UI components. We decide to use QT because it returns the consistent results across multiple runs of clustering with the same input, and it can be used to cluster particular groups. However, requiring more computation resources is a drawback of QT. We measure the similarity $Sim(C_i, C_j)$ between UI components C_i and C_j using the cosine similarity algorithm [77]. The $Sim(C_i, C_j)$ $= Cosine(V_i, V_j)$, where V_i is the value vector of UI $component_i$ and V_j is the value vector of UI $component_j$. We calculate the similarity value between any pairs of UI components except the components from the same parent element in HTML DOM Tree. The QT algorithm clusters UI components based on the similarity values of UI components. Within a cluster, all of the components potentially require the same inputs from users.

5.2.6 Extracting User Contexts

To make it context-aware, the framework extracts end-user's contexts and uses them for web form filling. The contextual information ranges from physical data such as user's location to "virtual" data such as user's to-do list. We identify three types of data resources which can be used for extracting user contexts in the task of web form filling. The three sources are listed as follows:

- *The End-user's Computing Environment.* The end-users can use different computing environments for different web form filling tasks. The data extracted from the computing environments can help web form filling techniques identify the different web form filling scenarios of an end-user. For example, an end-user searches for the cheapest tickets using his or her computer at work and pays for the tickets using his or her own computer because his or her work computer is monitored by the corporate intra-net and payment information is too private. Furthermore, the configuration data, such as time zone and IP address can be used for filling in input UI components requiring the current contexts (e.g., current time, current location) from the end-user.

- *End-user's Applications.* End-users use various applications (e.g., Google calender and Facebook) for their daily tasks such as sending emails and checking friend's references on Facebook. The information from such applications helps the web form filling techniques infer the details of end-users. For example, an end-user has blocked a time window ranging from Mar 09th to Mar 14th on his or her calender for a business travel. He or she wants to book an appointment with an optometrist through the optometrist's website. However only the time window from 3:00PM to 3:45PM on Mar 10th is shown as open on the website within the whole month of March. Therefore the time window 3:00PM to 3:45PM on Mar 10th should not be selected automatically.

- Web browsers: End-users use web browsers to surf web pages. Web browsers cache users' bookmarks and historical activities that can reflect user's preferences. The web form filling techniques consult with the user's preferences during the process of filling web forms.

One challenge is to map the contextual data to the input UI components. In most cases, the descriptive text used in user interface differs from the one used to describe the context [34]. For example, the location information in a calender entry has the label "Location" whereas the location information for the UI in a car rental website is "pick-up". We adopt the approach proposed in [34] for mapping contextual information to UI elements. We combine string-based and semantic similarity measures.

In our current work, we only extract simple user contexts: current location, current time and calendar entries in Google calendar.

5.2.7 Filling in Web Forms

After collecting and analyzing user inputs, user's contexts and input UI components, the framework starts filling in the web forms. During the process of filling values to web forms, there are two strategies listed as follows:

Strategy 1: Pre-filling a web form before an end-user enters a value. When the framework receives current user contexts and the possible *Candidate Set* or *Candidate UI Set* as described in Section 5.2.2. The following steps are conducted to fill an input UI component, denoted as ***target component***, of a web form with a value: First, the framework detects whether the ***target component*** requires the current user contexts (e.g., current location and current time) or not. If the ***target component*** requires the current user contexts, then the framework pre-fills a current user context to the ***target component***. Otherwise, the framework processes a *Candidate Set* or *Candidate UI Set* in the following way:

- The *Candidate Set* of the ***target component*** is parsed if the ***target component*** is already stored in the *UI Component Repository.* The framework ranks the user inputs of *Candidate Set* based on their timestamps from latest to earliest and pre-fills the latest user input to the ***target component*** if the latest user input does not contradict with any calender entries.

- The *Candidate UI Set* of the ***target component*** is parsed if the ***target component*** is a new input UI component in the *UI Component Repository.* The framework chooses the latest user input of the most similar input UI component to the ***target component***.

Strategy 2: Recommending values to input UI components after an end-user adjusts any pre-filled values. The following steps are conducted to fill an input UI component, denoted as ***target component***, of a web form with a value:

- Step 1. The framework builds a set of user inputs for recommendation to the ***target component*** in the following steps:

  – If a *Candidate Set* exists, the *Candidate Set* is considered as a recommendation set.

  - If a *Candidate UI Set* exists, all of the user inputs of the input UI components in *Candidate UI Set* are extracted and built as a recommendation set.
  - Otherwise there is no set of user inputs for recommendation.

- Step 2. The framework recommends a user input to the ***target component*** based on the modified value (i.e., a string) in the ***target component*** dynamically. The framework conducts a string matching between the modified value and the values in the *recommendation set* to identify the suitable values from the *recommendation set.*
  - If no values identified, no value is recommended to the ***target component***.
  - If a set of values identified, the values are ranked based on their times from latest to earliest.
- Step 3. If the end-user selects a user input from the recommendation set, the framework searches for the patterns of user inputs containing the selected user input and fills the other values in the pattern to other input UI components which are neighbors to the ***target component***. If multiple patterns identified, the most frequent pattern is chosen.

5.3 Case Studies

In this section, we conduct several experiments to evaluate the effectiveness of our framework. First, we introduce the case study setup. Then, we present the research questions. For each research question, we state the motivation of each question, the analysis approach and the corresponding results.

Table 5.1: Descriptive statistics of TEL-8 web forms.

Domain	**# of Forms**	**# of Components**
Airfare	65	447
Auto Mobiles	47	706
Books	84	441
Car Rentals	25	204
Hotels	39	396
Jobs	49	389
Movies	73	504
Music Records	65	448

5.3.1 Case Study Setup

In this sub-section, we introduce our case study setup.

Web Forms Dataset

We conduct our study on TEL-8 Query Interfaces [41] which contain a set of web query interfaces of 447 web sources across 8 domains in the web. The 8 domains are Airfares, Automobiles, Books, Car Rentals, Hotels, Jobs, Movies, and Music Records. We chose this dataset as our testing data due to the two following reasons:

- The dataset contains web interfaces (i.e., web pages) from the typical domains of web applications that capture web form structures on the web nowadays. Moreover, the dataset is publicly available.

- The dataset has been widely used by several research studies in different research areas. For example, Zhang et al. [113] extract the query capability from this dataset for understanding the web query interfaces; He et al. [35] propose a correlation mining approach to discover complex matchings across web query interfaces using this dataset; Araujo et al. [7] propose a concept-based approach for web form filling.

The dataset of TEL-8 Query Interfaces is manually collected in [113]. We extract input UI components from the web sources. Table 5.1 shows the number of web forms of each domain and the number of extracted input UI components from the web forms of each domain. In total, we collected 3,535 input UI components.

User Input Collection and Processing

To test the effectiveness of our framework on web forms auto-filling, we collect user inputs from four end-users who are graduate students. Three of the recruited end-users are female and the other one is male. Their ages are from 25-35 and they typically spend 8-10 hours per day on-line. The author of this thesis is not involved in the user input collection.

The four end-users randomly selected web forms from TEL-8 dataset and used the approach of ***collecting user inputs*** described in Section 5.2.3. The four end-users spent two days on collecting user inputs. We collected four end-user profiles for each domain. Based on the four profiles for each domain, we generate more user inputs for each profile by randomly splitting the collected user inputs and then combining the split user inputs.

5.3.2 Research Questions

In this sub-section, we presents our two research questions. For each research question, we discuss the motivation of the research question, analysis approach and the corresponding findings.

RQ1. Is our clustering technique effective in grouping similar UI components?

Motivation. Similar user interface components may consume the same values. Our proposed framework clusters input UI components into semantic groups. The components within a cluster are considered to be similar. In this research question, we verify the effectiveness of the clustering approach on identifying similar components.

Analysis Approach. To test effectiveness of our clustering approach, we apply our clustering approach on the extracted input UI components of each domain. To be able to validate the results manually, we randomly sample input UI components with confidence level 95% [19]. In total, we randomly sampled 347 input UI components from 3,535 input UI components. Second, we perform a manual clustering step on the sampled 347 components to create our *gold standard* for validation purposes. Third, we apply our clustering approach on the sampled 347 components to automatically cluster them into different semantic groups. During the clustering process, we use 0.85 determined experimentally as the minimum similarity threshold between any pair of UI components in a cluster. Forth, we compare the results generated from the previous step with the *golden standard*. We compute precision and recall of our clustering approach using Equation (5.4) and Equation (5.6) respectively.

$$Precision = \frac{\sum_{i \in C} P_{C_i}}{Length(C)} \tag{5.4}$$

$$P_{C_i} = \frac{succ(C_i)}{succ(C_i) + mispl(C_i)} \tag{5.5}$$

$$Recall = \frac{\sum_{i \in C} R_{C_i}}{Length(C)} \tag{5.6}$$

$$R_{C_i} = \frac{succ(C_i)}{succ(C_i) + missed(C_i)} \tag{5.7}$$

where C_i is the cluster i, P_{C_i} and R_{C_i} are the precision and the recall for cluster C_i. $succ(C_i)$ is the number of input fields successfully placed in the proper cluster C_i. $mispl(C_i)$ is the number of input fields incorrectly clustered into C_i. $missed(C_i)$ is the number of input fields that should be clustered into C_i, but incorrectly placed into other clusters. Length(C) is the number of clusters.

Results. The clustering approach generates 107 clusters in total. All the components within a cluster potentially require the same user input from the end-users. If one of them has been associated with a previous user input, it is most likely that the other components of the cluster require the same user input. Our approach achieves a precision of 89% and a recall of 83%. Based on a postmortem manual investigation, we found that the major reason for missing some cases is that some of the extracted UI components, in our test data, do not have a meaningful descriptive information. This is the major reason that the values of precision and recall are not 100%.

Summary of RQ1: The results of our empirical study show that our proposed approach for identifying similar user interface components in our dataset is effective.

RQ2. Is our framework effective in pre-filling web forms?

Motivation. Pre-filling web forms improves the productivity of composing ad-hoc processes by saving end-users from repetitive inputing of values to web forms [96]. In this research question, we test the effectiveness of our framework on pre-filling web forms based on the previous user inputs, patterns of user inputs, user's contexts and clusters of similar input UI components.

Table 5.2: Results of our framework and Firefox Autofill Forms on pre-filling web forms. P stands for Precision and R stands for Recall.

Domain	**Our Framework**		**Firefox**	
	P(%)	R(%)	P(%)	R(%)
Airfare	62	55	45	30
Auto Mobiles	70	58	55	38
Books	78	64	69	55
Car Rentals	82	65	75	45
Hotels	79	53	60	45
Jobs	81	63	79	48
Movies	71	55	60	22
Music Records	73	54	48	26

Analysis Approach. To test the effectiveness of our framework on pre-filling web forms, we compare our framework with Mozilla Firefox Autofill Forms [27] used as a baseline approach. We conduct the experiment in the following steps. First, we apply our approach of ***identifying patterns of user inputs*** on our collected user inputs in Section 5.3.1 (i.e., User inputs collection and processing). Second, we correct the incorrectly placed input UI components from the clusters generated in *RQ1* and use the clusters for identifying similar input UI components in our framework. Third, we manually enter values to the predefined entries of the default profile in Mozilla Firefox Autofill Forms [27]. Fourth, we apply our framework to pre-fill the web forms which are not selected by the four end-users during the collection of user inputs in Section 5.3.1 using the collected user inputs in Section 5.3.1. Fifth, we manually use

Firefox Browser to visit the web forms used in the previous step, and Firefox Autofill Forms to fill in input UI components of the visited web forms. Last, we calculate the precision and the recall using Equation (5.8) and Equation (5.9) to measure the effectiveness of our framework and Firefox Autofill Forms on web form pre-filling. The precision metric measures the fraction of the pre-filled user input parameters that are correctly pre-filled, while the recall value measures the fraction of all the user input parameters that are correctly pre-filled.

$$Precision = \frac{|Correctly\,Filled\,Input\,Components|}{|Filled\,Input\,Components|} \tag{5.8}$$

$$Recall = \frac{|Correctly\,Filled\,Input\,Components|}{|Input\,Components\,Need\,To\,Be\,Filled|} \tag{5.9}$$

Results. Table 5.2 shows that our framework outperforms Firefox Autofill Forms on pre-filling web forms. On average, our approach can achieve a precision of 74.5% and a recall of 58%, the Firefox Autofill Forms can achieve a precision of 61.3% and a recall of 38.6%. We investigated our results and found that even simple end-user context information such as current location helps identify right values for input UI components.

The main reason of the low recall of Firefox Autofill Forms is lack of support for complex types of information. The Firefox Autofill Forms is limited to basic types of information, such as Name and City, for web form filling. Furthermore, the Firefox Autofill Forms can only keep one value for each type. However an input UI component may require different values of a specific type under different tasks. For example "contact phone number" can have two values, one value is used as a travel contact number and the other one is used as the contact number of receiving shipments from E-commerce websites. Therefore, the Firefox Autofill Forms does not distinguish the different pieces of information of one type under different tasks. In addition, the Firefox Autofill Forms solely relies on keyword string matching when identifying user inputs for input UI components. If the textual description of UI components is missing or does not contain the keywords predefined in the default user profile of Firefox Autofill Forms, the Firefox AutoFill Forms missed the opportunity for pre-filling.

Summary of RQ2: Our framework outperforms Firefox Autofill Forms on pre-filling web forms in our dataset. Firefox Autofill tool fails mainly due to the limitation in handling complex data types. Moreover, Firefox Autofill Forms can only keep one value for each type.

RQ3. Is our framework effective in recommending values to web forms?

Motivation. When an end-user is not satisfied with a pre-filled value to an input UI component, he or she can modify the pre-filled value. During the course of value modification, the framework recommends a list of values to the end-user. In **RQ2**, we evaluate the effectiveness of our approach on web form auto-filling. In this research question, we test the effectiveness for recommending values to end-users.

Analysis Approach. To test the effectiveness of our framework on recommending values to end-users, we compare our framework with Google Chrome Autofill Forms [32] used as a baseline approach. We conduct the experiment in the following steps. First, we clear out all of the input UI components which are incorrectly pre-filled in **RQ2**. Second, we build a new set of user inputs for recommending values to UI components by excluding the user inputs used in **RQ2**. Third, we apply our approach of ***filling web forms*** on recommending values to the UI components incorrectly pre-filled in **RQ2**. Fourth, we manually visit the web forms in Section 5.3.1 which do not contain a UI component incorrectly pre-filled in **RQ2** using Google Chrome Web Browser and enter the new set of user inputs into proper UI components. Fifth, we apply Google Chrome Autofill Forms on the input UI components incorrectly pre-filled in **RQ2**. Finally, we calculate precision and recall using the Equation (5.8) and Equation (5.9) based on the value recommended on the top, the top-1 value, to measure the effectiveness of our framework and Google Chrome Autofill Forms on suggesting values to end-users. We choose the top-1 value for the second time pre-fill.

Table 5.3: Results of our framework and Google Chrome Autofill Forms on recommending values to end-users. P: Precision, R: Recall.

Domain	**Our Framework**		**Chrome**	
	P(%)	R(%)	P(%)	R(%)
Airfare	73	68	65	55
Auto Mobiles	82	63	75	60
Books	86	72	80	65
Car Rentals	87	75	85	70
Hotels	82	65	80	60
Jobs	90	81	70	55
Movies	78	60	66	48
Music Records	80	63	62	52

Results. Table 5.3 shows that our framework outperforms Google Chrome Autofill Forms. When the pre-filled values are not correct in the first place (i.e., filling web forms in Scenario 2), on average, our approach can achieve a precision of 82.25% and a recall of 68.4%, and Google Chrome Autofill Forms can achieve a precision of 60.3% and a recall of 58.1%, on recommending values to end-users. We further investigate the results of our framework and Google Chrome Autofill Forms. We found that the patterns of user inputs help identify the right values for recommendation. With the accumulation of user history and more patterns generated, the results can be improved gradually. The Google Chrome Autofill Forms can only track a limited number of types of user inputs, such as name and address. The limited support of user input collection leads to the low recalls.

Summary of RQ3: Our framework outperforms Google Chrome Autofill Forms on recommending top-1 values to input parameters of web forms in our dataset. Furthermore, the patterns of user inputs help identify the right values for recommendation.

RQ4. Does our framework have qualitative advantages over existing web browser auto-filling tools?

Motivation. In **RQ2** and **RQ3**, we compare our framework with Mozilla Firefox Autofill Forms [27] and Google Chrome Autofill Forms [32] quantitatively. In this research question, we compare our proposed framework with Mozilla Firefox Autofill Forms [27] and Google Chrome Autofill Forms [32] qualitatively.

Analysis Approach. To answer this research question, we follow the analysis framework proposed in [20] to compare our framework with the mentioned two tools. Although the analysis framework in [20] is proposed for comparing web information extraction (IE) tools, we extend and apply it on the comparison of web auto-filling techniques.

We compare tools in three dimensions: *Task Difficulties*, *The Techniques Used* and *Automation Degree*. The first dimension, *Task Difficulties*, evaluates the difficulty of an auto-filling task, which can be used to answer the question "why an auto-filling technique fails to handle some input UI components?" The second dimension,

Table 5.4: Comparison between our framework with existing tools: Firefox Autofill Form and Google Chrome Autofill Form. PUI: Previous User Input, BPI: Basic Personal Information.

Features	**Firefox**	**Chrome**	**Ours**
Automation Degree	Partial	Fully	Fully
Range of UI Components	Short	Short	Medium
Data Supported	PUIs, BPI	Partial PUIs, Partial BPI, and User Usage Data	PUIs, BPI, Basic Contextual Data, and User Usage Data (maximum length of patterns)
Matching Algorithms	String Matching	String Matching	String Matching, Semantic Similarity
Filling Strategy	Pre-filling	Recommending	Pre-filling, Recommending
Learning Algorithms	String alignment	String alignment, Pattern Mining	String alignment, Pattern Mining, and Clustering

The Techniques Used, evaluates the techniques used in different auto-filling tools. The third dimension, *Automation Degree*, evaluates the effort made by end-users for providing values to input UI components. For each dimension, we include a set of features that can be used as criteria for comparing and evaluating auto-filling techniques.

We list the features of each dimension as follows: **Task Difficulties.** The range of types of user interface components which can be identified and filled in with a value is important to an auto-filling tools. In this chapter, we classify the range into three lengths based on the number of user interface components: short (1 to 5 UI components), medium (5 to 15 UI components), long (over 15 UI components). Supporting more UI components requires tremendous research effort. Due to the variety of HTML page structures, the task of extracting meaningful textual information for different UI components is not trivial. It is also a challenging job to match values with proper components, convert the values into right format for filling.

The types of data which could be used for filling in web forms can be previous user inputs, basic personal information profile (e.g., name, address, credit card information), end-users' contextual data, user usage data. Each type of data source requires auto-filling tools to make different research efforts such as approaches of collecting data. The user usage data can also be viewed as a type of end-users' contextual data, however it requires relatively huge computation and history, we follow the approach in [98] to separate the user usage data from end-user's contextual data.

The Techniques Used. Filling web forms involves a series of algorithms. In summary, the features for comparing auto-filling tools from the perspective of techniques used include: tokenization, extraction rules, features involved, and learning algorithms. Furthermore, pre-filling and recommending are the two main strategies of reusing use data for filling values to web forms [7].

Automation Degree. Speaking of the task of filling values to web forms, it is really all about how much convenience the auto-filling tools can bring to end-users in terms of reusing their data. We identify three automation degrees on the data types supported by the auto-filling tools. The three automation degrees are listed as follows:

- *Full Automation.* There is no need of human involvement on data management, such as user input collection.
- *Partial Automation.* The data management still needs some human intervene.
- *No Automation or very little.* End-users have to manually manipulate the data,

such as creating personal profiles manually for filling web forms.

Results. Table 5.4 shows that our framework supports more types of data and user interface components than the other tools do. The Google chrome auto-filling tool only supports very limited number of user inputs (e.g., credit card information and login credentials), and tracks all of the patterns of the user inputs supported by chrome (i.e., Chrome keeps all lengths of patterns of user inputs). We believe the big advantage of our tool over the other existing two tools is analyzing and linking similar user interface components which enables the propagation of user inputs across different tasks during the ad-hoc business process composition.

Summary of RQ4: The results of our qualitative analysis show that our framework outperforms Google Chrome Autofill Forms qualitatively. One big advantage of our tool over the other existing tools is analyzing and linking similar user interface components which enables the propagation of user inputs across different tasks.

5.4 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [111].

Construct validity threats concern the relation between theory and observation. In this work, the construct validity threats are mainly from Parsing HTML web forms: Due to the various structures of HTML web pages, it is a challenging task to extract information from web pages. For example, the positions of labels in web form depend on the designer of the web page. The labels can be placed above, below, to the left, or to the right of an input element. We adopt the approach in [90] to extract information from web pages. The *User Input Collector* and *User Interface Analyzer* deal with the HTML web pages.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. First, we modified the web application testing tool called Sahi [76] to trace the user inputs and extract the information we need for further process. We chose the Sahi instead of developing a new tool due to two reasons:

- We only have limited resources (e.g., man power).
- The tool covers all types of devices, which becomes an advantage if we collect user inputs from a mass of end-users. In the near future, we plan to recruit more people for our user case study.

Second, we collected user inputs through four end-users. The number of subjects may be low, the main reason is that people are reluctant to give away their personal information. However, the number of end-users profiles is not low. We collected 32 real end-user profiles for web forms from 8 domains in total. Moreover, we generated more user profiles by splitting and combining the collected real personal profiles. Third, we performed a manual clustering step to create gold standard for validation, two human experts are involved in the creation and verification of gold standard. The human judgment could threaten the validity, however it is very common to include manual study in research studies such as [7].

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The dataset of web forms used in our study are publicly available [41].

5.5 Chapter Summary

In this chapter, we propose an intelligent framework to help users fill in web forms and save them from repeatedly entering the same information to web forms in web applications. Our framework automatically collects user inputs and analyzes them for generalizing patterns. The patterns of user inputs and user context data are used to support web form auto-filling. Our framework reuses the previous user inputs by clustering the similar UI components into semantic clusters. Based on the results of our empirical study, our clustering algorithm can achieve a precision of 89% and a recall of 83%. Furthermore, the results of our empirical study show that our framework is effective in filling web forms and recommending values to UI components with the previous user inputs by exploiting usage patterns and mining context information, compared with Firefox Autofill Forms and Google Chrome Autofill Forms. On average, our framework can achieve a precision of 74.5% and a recall of 58% on pre-filling web forms, and a precision of 82.25% and a recall of 68.4% on suggesting values to end-users if the end-users are not satisfied with the pre-filled values.

Chapter 6

Propagation of Values among Services

A web service has input and output parameters. Accurately linking logically and semantically similar parameters, efficiently organizing previous user inputs for filling out services, and properly leveraging similar end-users' user inputs are essential for propagating previous user inputs among different services and end-users.

In the prior chapters, input parameters from web and mobile applications are studied. A comprehensive technological infrastructure has been proposed in the previous chapter. Furthermore, the empirical results in the prior chapters (i.e., Chapter 4 and 5) show the importance of categories of input parameters and user contexts in the process of pre-filling parameters. Reusing the results in prior chapters, in this chapter, we propose an approach that propagates user inputs across services and pre-fills values to input parameters for an end-user using his or her previous inputs. To increase the chance of identifying a proper value for an input parameter performed by one end-user, our approach also leverages the inputs from other end-users.

Chapter Organization. The rest of this chapter is organized as follows. Section 6.1 presents an introduction of this chapter. Section 6.2 introduces the proposed approach. Section 6.3 introduces the case study on our approach. Section 6.4 discusses the threats to validity. Finally, Section 6.5 concludes the chapter and outlines some avenues for future work.

6.1 Introduction

The critical step for propagating user inputs among services is linking semantically related parameters. Existing approaches heavily rely on the names of parameters for identifying similar parameters. When the names of parameters are not suitable for calculating the similarity between input parameters due to bad naming and incorrect spelling, these approaches do not perform well. Moreover, all of the existing approaches are not designed for filling values to input parameters in the context of service composition.

In this chapter, we propose an approach that analyzes and links similar parameters for propagating user inputs among services, pre-fills linked parameters with user inputs. We propose a context-aware meta-data model for capturing user's inputs. The model is designed to store task information of an input. Our approach of pre-filling uses the model for context-aware matching between the inputs and the parameters (i.e., a value to an input parameter) during the service compositions. We propose an *Input Parameter Context Model* to identify similar parameters of services in order to reuse user's inputs across services. In our *Input Parameter Context Model*, each parameter has three contexts: Textual Context, Task Context and Neighbors Context.

In some cases, one end-user's previous inputs may not be sufficient to identify a proper value for pre-filling an input parameter due to the lack of history. We collect and leverage user inputs from other end-users who share the similar activities as the end-user does. Two end-users are similar if the two end-users have performed similar activities before. We calculate the similarity between two end-users using their historical activities (e.g., services performed). Our approach identifies a set of similar end-users to an end-user, and leverages the user inputs from the identified similar end-users. Our approach allows us to maximize the reuse of user's inputs and take our best effort to prevent end-users from being interrupted by redundant data entries during the service execution.

We conduct empirical studies using real world services to evaluate the effectiveness of our proposed approach. On average, our approach using an end-users previous inputs can reduce on average 41% of repetitive typing for the execution of composed services. In 83% of the cases, our approach can identify the most similar end-user to an end-user. More importantly, including user inputs of similar end-users of an end-user can help reduce the parameter filling for the end-user.

6.2 Overview of Our Approach

Figure 6.1 shows the steps of our approach. Our approach consists of five major steps:

1. *Collecting and storing user inputs.* The user's previous inputs can be an excellent source to feed input parameters. A user input is a piece of information

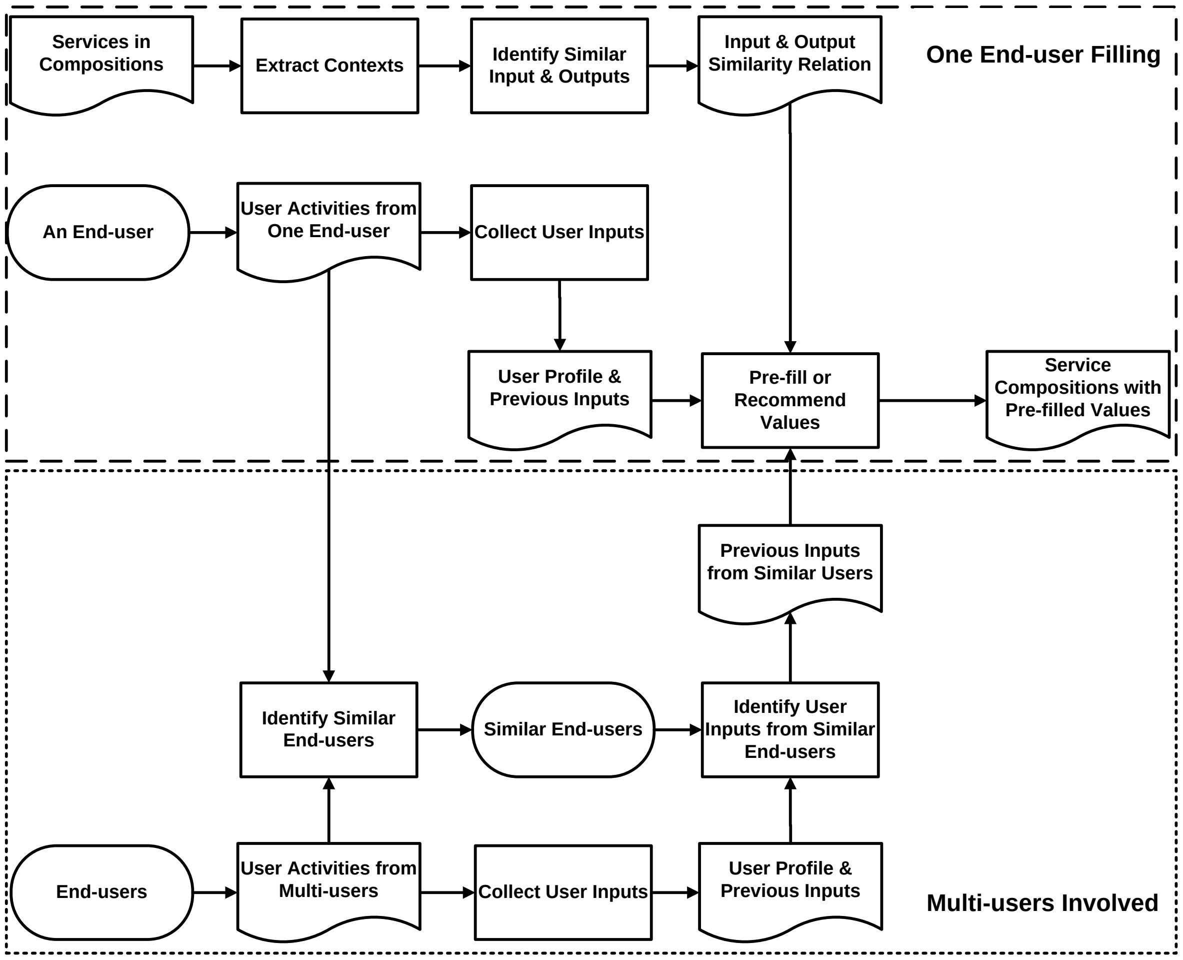

Services in Compositions
Extract Contexts
Identify Similar Input & Outputs
Input & Output Similarity Relation
One End-user Filling
An End-user
User Activities from One End-user
Collect User Inputs
User Profile & Previous Inputs
Pre-fill or Recommend Values
Service Compositions with Pre-filled Values
Previous Inputs from Similar Users
Identify Similar End-users
Similar End-users
Identify User Inputs from Similar End-users
End-users
User Activities from Multi-users
Collect User Inputs
User Profile & Previous Inputs
Multi-users Involved

Figure 6.1: Overall steps for propagating the inputs of end-users across different services during service composition.

entered by an end-user into services. To exploit such information, the first step is to capture such data properly for further usage.

2. *Collecting contexts of parameters and building input-output dependency.* We extract the contexts of parameters to identify similar parameters for building input-output parameter dependency. The three contexts of an input parameter are *Textual* describing the textual information of the parameter, *Task* defining the task information of the parameter, and *Neighbors* storing the information

of a group of other parameters that reside in the same operation as the parameter. The dependency contains the semantic relations among parameters. The dependency is used to propagate user's inputs across different services.

3. *Identifying similar end-users.* Two end-users are similar if these two end-users perform similar or identical tasks, such as services composed. Similar end-users may fill in the same value to an input parameter. We link the end-users conducting similar activities, such as the services composed and input parameters provided with values in the composed services.

4. *Collecting user inputs from similar end-users.* For an end-user required to enter a value to an input parameter of a task, we identify the similar or identical tasks from the similar end-users identified in the step of *Identifying similar end-users.* Then, we collect the user inputs associated with the identified tasks for pre-filling the input parameter.

5. *Pre-filling values for operations of services.* This step selects a proper value for an input parameter from all of the possible data sources: an end-user's personal information and his or her similar end-users' previous inputs. The end-user's personal information includes his or her profile information and his or her previous inputs.

The Steps 1), 2) and 3) are the main steps for pre-filling input parameters using one end-user's previous inputs and profile information. With the addition of the steps of *Identifying similar end-users* and *Collecting user inputs from similar end-users*, our approach can use not only one end-user's previous inputs, but also other end-user's previous inputs for pre-filling input parameters.

6.2.1 Five Scenarios of Assigning Values to Input Parameters

We identify five scenarios of supplying a value to an input parameter of a service participating in a service composition performed by an end-user. The five scenarios are listed as follows:

1. *Scenario 1:* Sharing values among the similar input parameters of operations. The operations can be from a single service or different services. If two input parameters from different operations are semantically related, the value provided to one of operations can be propagated to fill in the other one either in the same service or a different service.

2. *Scenario 2:* Reusing values of output parameters to fill in an input parameter. The output parameters are generated from an operation, and the targeted input parameter is requested from another operation. The operations can be either from a single service or different services. The output of an output parameter can be used to fill in an input parameter. For example, an operation of a service generates a city name using zip code, and an input parameter of a weather forecast service requires a city name to provide weather forecast.

3. *Scenario 3:* Using other data sources of an end-user to assign values to input parameters. The personal profile data and the previous user inputs from the end-user are used as the external data sources to fill in an input parameter of a service performed by the same end-user. The end-users could execute the similar services consuming the previous inputs or information in personal profile. For example, an end-user may conduct a business trip multiple times within a year with same inputs except for departure and return dates.

4. *Scenario 4:* Leveraging the user inputs from other end-users similar to an end-user. An end-user' previous inputs may be not sufficient to pre-fill any encountered input parameters. However, using the user inputs from the similar end-users of the end-user could increase the chance of pre-filling the right values to the encountered input parameters. The end-users who perform similar activities could share and reuse their inputs.

5. *Scenario 5:* Requiring data entry by end-users. If the aforementioned sources are not available, the end-users have to enter the values manually.

In this study, we collect user inputs from end-users when they use web interfaces and our approach uses the collected information in the above five scenarios.

6.2.2 Collecting and Storing User Inputs from End-users

A user input is a piece of information entered by end-users into services. It is usually stored as a key-value pair by existing approaches such as [7][27][32] where the key is the type of the information and the value is the information itself, for example "address" is the key and "Kingston, Canada" is the value. The key-value model lacks of information for conducting context-aware matching between the values and input parameters. For example, there are two values for a key "contact phone number", such as (647)222-3333 and (123)456-7890; the end-users use (647)222-3333 for flight booking and (123)456-7890 for hotel reservation. We need to modify and enrich the key-value pair model by attaching contextual information to the piece of information and propose a context-aware Meta-Data Model for storing user inputs.

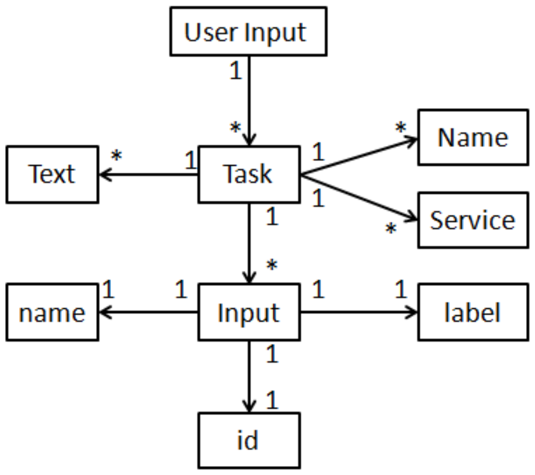
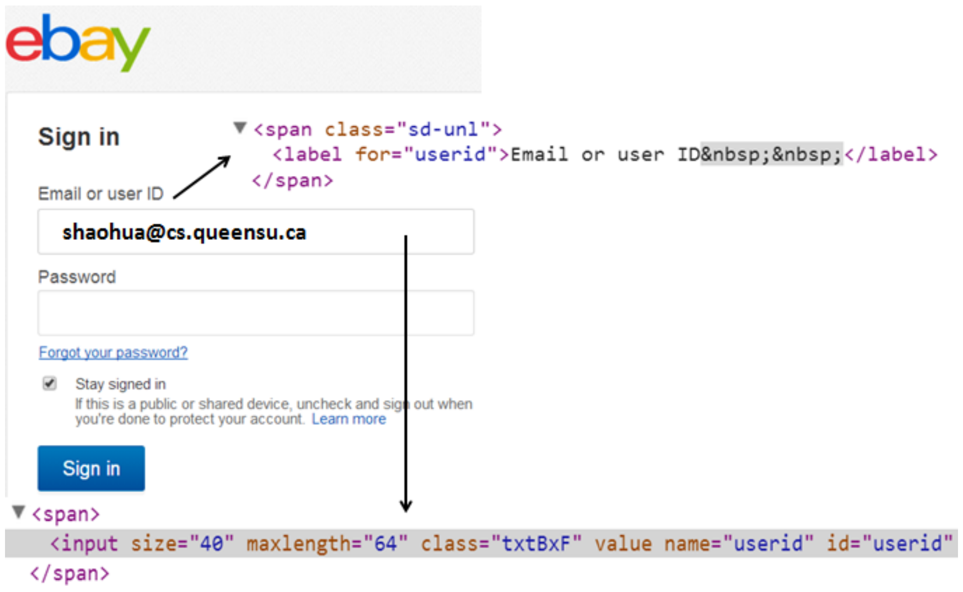


Figure 6.2: Description of Context-aware Meta-Data Model


ebay
Sign in
Email or user ID
Password
Forgot your password?
Stay signed in
If this is a public or shared device, uncheck and sign out when you're done to protect your account. Learn more
Sign in

shaohua@cs.queensu.ca

```
▼<span class="sd-unl">
   <label for="userid">Email or user ID&nbsp;&nbsp;</label>
 </span>
```

```
▼<span>
   <input size="40" maxlength="64" class="txtBxF" value name="userid" id="userid"
 </span>
```

Figure 6.3: A mapping between UI and its HTML coding.

Context-aware Meta-Data Model of User Inputs

Figure 6.2 shows the description of our proposed context-aware meta-data model. Figure 6.3 illustrates an example of storing an email address “shaohua@cs.queensu.ca” entered into the login service of Ebay[1]. A user input has two properties: Type and Task. The Type property records the data type of the information. A task can be an operation in WSDL services, or a resource of a RESTful service and the associated action (e.g., get or delete) or a web form of a web application. As shown in Figure 6.3, the data type of “shaohua@cs.queensu.ca” is email and its task is “Sign in”. A user input can be associated with multiple tasks. For example, an address 25 Union St. Kingston, Canada can be used as a shipping address, and also a home address for different tasks. The Task property describes the applicable tasks for a user input (i.e., a piece of information) and has four sub-properties listed as follows:

- ***Name Property*** states the name of a task, such as the name of an operation defined in WSDL, the resource name and its associated actions of a RESTful service, and a label (e.g., from a web form or HTML tag). As shown in Figure 6.3, the Name Property of the task of “shaohua@cs.queensu.ca” is“Sign in”.

- ***Text Property*** defines the description of a task. We retrieve a description for a task from the description of an operation in WSDL, the descriptive text of a resource in a RESTful service, or textual information of a web form. If the description is not available, we assign NULL to this property. In Figure 6.3, the Text Property of the task “Sign in” is web form name “SignInform”.

- ***Service Property*** records the name of a service in which a task is performed. We use the name of the WSDL, RESTful service, or the URL of a web form as a service property. As shown in Figure 6.3, the service property is “URL of the login Web form”.

[1] *www.ebay.com*

- ***Input Property*** stores the coding information of the input parameters. The coding information includes three properties: *label*, *name*, *id*. In the context of Web applications, *label*, *name*, *id* are the attributes of an HTML DOM element which defines the input field consuming the information. As shown in Figure 6.3, the values of input properties: *label*, *id* and *name* are "Email or user ID", "userid" and "userid". If the information is entered into a WSDL or RESTful service, we store the name of the input parameter in the property of *label*.

Collecting User Inputs

The end-user's inputs can be collected through end-user's activities, such as on-line activities and web service testing. For example, the end-user's on-line activities can be shopping on-line and filling in student registration web forms; the web service testing can be testing a RESTful service using its URL through a web browser, or testing a SOAP-based WSDL service using SOAPUI framework[2].

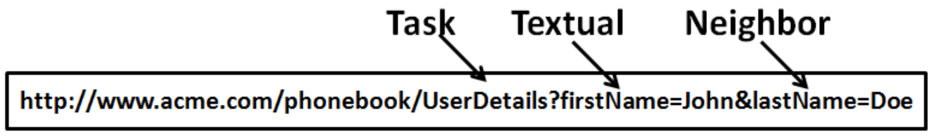
We modify and extend an open source tool called Sahi[3] to monitor end-user's web activities. Sahi injects Javascripts into web pages and monitors the user's actions (e.g., click, submit), records the user's inputs. We extend Sahi to include the detail information of a user interface component where the end-user enters a value. Then we mine the information of properties as specified in Section 6.2.2 from Sahi log and store the user inputs with their property information in the Meta-Data Model. Our tool is a standalone application running on different operating systems including Windows 7 or 8 and Linux OS. The end-users can run our tool and surf the web through web

[2]http://www.soapui.org/
[3]http://sahi.co.in/

browsers including Google Chrome[4] and Mozilla Firefox[5].

6.2.3 Collecting Contexts of Parameters and Building Input-Output Dependency

To propagate an end-user's input across services and reduce the number of input parameters requiring user inputs, it is essential to link similar inputs and output parameters across services. The two similar input parameters from different operations (i.e., **Scenario 1** as described in Section 6.2.1) could require the same values. The value of an output parameter of an operation can be consumed by an input parameter of another operation (i.e., **Scenario 2** as introduced in Section 6.2.1). We exclude fault messages, such as "error code", for services in our analysis as they seldom contribute to the data flow in the subsequent services.



http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe

Figure 6.4: An example annotated screenshot of showing the four contexts of a parameter from a RESTful service.

Input Parameter Context Model

We propose an Input Parameter Context Model to identify the linkages among parameters. We extract three contexts for a parameter denoted as P (i.e., an input or output parameter) and calculate the similarity between parameters using their contexts. Figure 6.4 illustrates an annotated screenshot showing the contexts of a

[4]http://www.google.ca/chrome/
[5]https://www.mozilla.org/en-US/

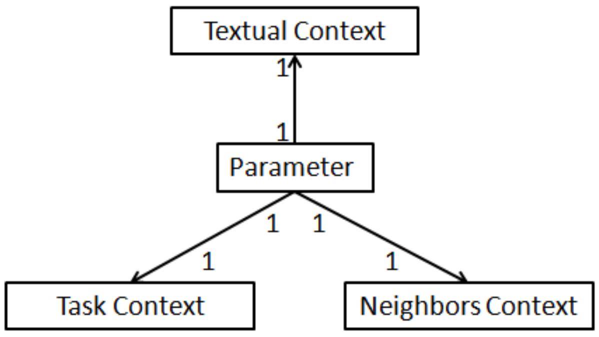


Figure 6.5: Description of Input Parameter Context Model

parameter from a RESTful service. Figure 6.5 shows the description of our proposed Input Parameter Context Model. The three contexts of a parameter are listed as follows:

- **Textual Context** (denoted as P^{Text}) defines the textual information of a parameter. In most cases, the value is the name of a parameter from an operation defined in WSDL and RESTful services. For example, the textual information of the input parameter "firstName" in Figure 6.4 is "firstName". RESTful services defined in WADL[6] usually provide some descriptive text for an input parameter. For an input parameter in Web component service, we extract text from *label*, *id* and *name* from the HTML DOM [87].

- **Task Context** (denoted as P^{Task}) stores the task information. A task is an operation in a WSDL service, a resource and its associated actions (e.g., get or delete), or a web form of a web application. We store the descriptive information along with a name for the task. For example, the task name of the input parameter "firstName" in Figure 6.4 is "UserDetails". The descriptive information of the task "UserDetails" can be obtained from the web page where the

[6]http://www.w3.org/Submission/wadl/

resource of a RESTful service is described.

- **Neighbors Context** (denoted as $P^{Neighbor}$): stores a group of other parameters required by the same operation. For example, the task "UserDetails" in Figure 6.4 has two input parameters: "firstName" and "lastName". Then, the two parameters are neighbors of each other.

After extracting contexts for a parameter, we store them in the following format $P =< P^{Text}, P^{Task}, P^{Neighbor} >$.

Building Input-Output Similarity Relation

To identify meaningful words from the contexts, we conduct word normalization. We decompose any possible compound words (e.g., FindCity). The naming of operations or parameters follows the conventions used in programming languages. We use four rules to decompose words: case change (e.g., FindCity), suffix containing a numeric number (e.g., City1), underscore separator (e.g., departure_city) and dash separator (e.g., Find-city). To remove non-English words, we use WordNet [26] which is a lexical database organizing the English words into a set of synonyms. Furthermore, we remove the possible stop words using a pre-defined list of stop words, such as "the". Finally, we use porter stemmer [68] to reduce derived words to their stem, base, or root form (e.g., "reserves" and "reserved" are mapped to the stem form "reserve").

After the word normalization, we calculate the context similarity. A context, such as task information P^{Task} of a parameter, could contain more than one word (e.g., "book car"). We formulate a context into a phrase, Ph, which is a collection of words, $Ph = \{W_1, W_2, \ldots, W_n\}$, where n is the number of words in Ph. We use WordNet to calculate the semantic similarity between two phrases Ph_i and Ph_j in the following cases:

- Case 1. We identify the common words (i.e., two words are identical or synonymous) of two phrases. If Ph_i and Ph_j contain the same number of words and all of the words are same, then Ph_i=Ph_j.
- Case 2. If the number of words contained in Ph_i is greater than the number of words contained in Ph_j, and all of the words in Ph_j are contained in Ph_i, we use Equation (6.1) to calculate the similarity between two phrases.

  $$sim(Ph_i, Ph_j) = \frac{|Ph_i \cap Ph_j|}{|Ph_j|} \tag{6.1}$$

  where $sim(Ph_i, Ph_j)$ denotes the semantic similarity value between Ph_i and Ph_j, Ph_j is a subset of Ph_i.
- Case 3. If there is no *containment* relation between Ph_i and Ph_j, we use the following steps to calculate the semantic similarity between two phrases.
  - Step 1: We calculate the number of pairs among common words of two phrases Ph_i and Ph_j.
  - Step 2: We calculate the semantic similarity between every pair of words W_a ($W_a \subset Ph_i$) and W_b ($W_b \subset Ph_j$) from two phrases after excluding the common words from calculation. Then, we sum up all of the similarity values of pairs of words, which is denoted as $Sum_{sim} = \sum_{W_a \subset Ph_i} \sum_{W_b \subset Ph_j}$ sim(W_a, W_b), where *sim(W_a, W_b) denotes the similarity value between two words, W_a and W_b; W_a and W_b are not identical.*

– Step 3: We use Equation (6.2) to calculate the similarity between two phrases. The numerator of the equation is to calculate the similarity between each pair of words of two phrases; the denominator of the equation is the total number of times of calculating the similarity between words.

$$sim(Ph_i, Ph_j) = \frac{|Ph_i \bigcap Ph_j| + Sum_{sim}}{|Ph_i \bigcap Ph_j| + |pairs\ of\ words\ without\ common\ words|} \tag{6.2}$$

where $sim(Ph_i, Ph_j)$ denotes the semantic similarity value between Ph_i and Ph_j. $|Ph_i \bigcap Ph_j|$ states the number of pairs of common words of Ph_i and Ph_j. Sum_{sim} is the sum of the similar value of every pair of words excluding the common words from Ph_i and Ph_j.

Given two parameters, P_i and P_j, for similarity calculation sim(P_i, P_j), we first calculate the similarity between each of their contexts separately using Equation (6.2), then we integrate the similarity of each context of parameters using Equation (6.3).

$$\begin{aligned} sim(P_i, P_j) = M_a * sim(P_i^{text}, P_j^{text}) + \\ M_b * sim(P_i^{task}, P_j^{task}) + \\ M_c * sim(P_i^{neighbor}, P_j^{neighbor}) \end{aligned} \tag{6.3}$$

where $sim(P_i, P_j)$ defines the similarity between two parameters.

$$sim(P_i, P_j) = \begin{cases} 1 & \text{if they are identical} \\ 0 & \text{if they are completely different} \end{cases}$$

$sim(P_i^{text}, P_j^{text})$ defines the semantic similarity between the context of Textual Information from two parameters, $sim(P_i^{task}, P_j^{task})$ is the semantical similarity between

the context of Task Information of two parameters, $sim(P_i^{neighbor}, P_j^{neighbor})$ evaluates the semantic similarity between the context of Neighbors of two parameters. M_a, M_b and M_c are the weights of a context. We assign equal weights to each context, i.e., 0.33. The weights of a context are derived empirically.

Given a set of services $S_1, S_2, \ldots, S_n$ (n is the number of services) in an on-the-fly service composition, we extract the input and output parameters of services. Then, we calculate the similarities between each input parameter and all of the other input and output parameters using contexts of the parameters. Therefore, each input parameter has a vector of similarity values. We remove the similarity values which are less than a threshold value which is derived empirically.

Given two services, S_i and S_j $(1 \leq i < j \leq n)$ of a service composition, S_j is composed after S_i. To fill in an input parameter, P of an operation in the service S_i, the parameters of operations from the service S_j cannot be used as data sources for S_i, because the parameters in the service S_j are from subsequent services. We remove such parameters from our analysis when we identify the similar parameters for the input parameter P. We rank the similarity values from the highest to the lowest within the vector for each input parameter. We build input-output similarity relation based on the vectors of similarity values among parameters.

Using the input-output similarity relation, we can reduce the number of input parameters requiring user inputs by identifying the similarity relations between parameters in **Scenario 1** and **Scenario 2**.

6.2.4 Identifying Similar End-users

An end-user's previous inputs and personal profile may not be enough to pre-fill proper values to input parameters. To increase the odds of pre-filling a proper value to an input parameter for one end-user, we leverage other end-users' historical information (i.e., previous inputs) for pre-filling input parameters. When a group of the similar end-users perform the similar tasks, they could reuse the same inputs. Linking similar end-users is a critical step to reuse the user inputs for end-users. In this section, we introduce our approach of identifying similar end-users for an end-user.

An end-user U performs a set of tasks $\{Task_1^U, Task_2^U, \ldots Task_n^U\}$, where n is the number of tasks performed by the end-user. Each task, $Task_i^U$ $(1 \leq i \leq n)$, involves a set of services and input parameters entered with values. Given two end-users U_i and U_j, we calculate the similarity between them in the following steps:

- Step 1: We identify common services and input parameters entered with a value of two end-users, U_i and U_j. We use Equation (6.4) to calculate the number of common services and input parameters between two end-users.

$$\begin{split} Common(U_i, U_j) = \\ |Common\ Services| + |Common\ Parameters| \end{split} \tag{6.4}$$

  where $Common(U_i, U_j)$ defines the number of common activities between two end-users, U_i and U_j.

- Step 2: We identify the end-user having more services and parameters, and then count the number of services and parameters. We use Equation 6.5 to calculate the number of services and parameters from the end-user having more activities.

$$Longer = |Services| + |Parameters| \tag{6.5}$$

*where **Longer** defines the total number of services and parameters from the end-user having more activities.*

- Step 3: We use Equation (6.6) to calculate the similarity value between U_i and U_j.

$$sim(U_i, U_j) = \frac{Common}{Longer} \tag{6.6}$$

where $sim(U_i, U_j)$ defines the similarity between two end-users and ranges from 0 to 1.

$$sim(U_i, U_j) = \begin{cases} 1 & \text{if they are identical} \\ 0 & \text{if they are completely different} \end{cases}$$

To identify similar end-users to an end-user U, we check all of the other end-users and calculate the similarity value between any other end-user and the end-user U. Then we select the other end-user with the highest similarity value as the similar peer to the end-user U.

6.2.5 Collecting User Inputs from Similar End-users

To identify the most likely useful user inputs from the similar end-users to pre-fill an input parameter P for an end-user, we identify the same or similar input parameters performed by both the end-user and his or her similar peers.

Given an end-user U and one of his or her similar peers U_p, we obtain the possible user inputs from a similar peer U_p in the following steps:

- Step 1: We traverse all of the tasks $\{Task_1^{U_p}, Task_2^{U_p}, \ldots Task_n^{U_p}\}$ of the similar peer U_p to verify whether the tasks are similar or identical to the task $Task_{current}^{U}$ performed by an end-user U. We calculate the similarity between each task, $Task_i^p$ $(1 \leq i \leq n)$, and the current task $Task_{current}^{U}$ using the approach of calculating the similarity value between two phrases described in Section 6.2.3.

- Step 2: We select the task $Task_i^{U_p}$ of the similar peer U_p with the highest similarity value as the matching task to the current task $Task_{current}^{U}$ of the end-user U.

- Step 3: We traverse all of the input parameters of the task, $Task_i^{U_p}$ selected in the previous step. We calculate the similarity value between each input parameter of the selected task $Task_i^{U_p}$ and the input parameter P performed by the end-user U using Equation (6.3).

- Step 4: We select the input parameter in **Step 3** with the highest similarity value, and then use the user inputs associated with the selected input parameter as the collected user inputs from U_p. We exclude user's personal information from the collected user inputs, because the personal information, such as end-user's name and shoe size, can not be reused in a different end-user context.

6.2.6 Pre-filling Values for Input Parameters of Services

In this section, we describe how our approach handles the **Scenario 3** and **Scenario 4** as discussed in Section 6.2.1 to provide values to input parameters in order to maximize the reuse of user inputs across services.

To fill in an input parameter P, we identify a proper value from the previous user inputs stored in the context-aware Meta-Data Model proposed in Section 6.2.2 and the collected user inputs from similar end-users presented in Section 6.2.5 if the end-user has no matching parameters from his or her previously used services. We traverse all of the available user inputs and calculate the similarity value between each available user input and the input parameter P. We select the user input with the highest similarity value as the input for the requested input parameter P.

Given a user input I and an input parameter P, we calculate the semantic similarity between them. We conduct word normalization on all of the properties of the user input I using the approach as discussed in Section 6.2.3. A user input can be associated with multiple tasks in different contexts. Therefore, we traverse all the possible ***Tasks***, $\{Task_1^I, Task_2^I, \ldots Task_n^I\}$, associated with the input I to verify whether the input I can be used to the input parameter P (where n is the number of tasks that the user input is used for). We calculate the similarity between a task associated with the user input I, denoted as $Task_i^I$ $(1 \leq i \leq n)$, and the input parameter P in the following steps:

1. We combine the words used in the ***Label***, ***ID*** and ***Name*** in the ***Input Property*** of $Task_i^I$ (a task associated with the user input I) as the input information of $Task_i^I$, denoted as $IP(Task_i^I)$. We use Equation (6.2) proposed in Section 6.2.3 to calculate the similarity value between the $IP(Task_i^I)$ and the ***Text Context*** of the input parameter P, denoted as $Text_p$. The similarity calculation is denoted as sim$(IP(Task_i^I), Text_p)$.

2. We merge the words from ***Text and Name properties*** of $Task_i^I$ as Textual information, which is denoted as $Text(Task_i^I)$. We use Equation (6.2) proposed in Section 6.2.3 to calculate the similarity value between $Text(Task_i^I)$ and the

Task context of the input parameter P, denoted as $Task_p$. The similarity calculation is denoted as sim($Text(Task_i^I)$, $Task_p$).

3. We use Equation (6.7) to calculate the similarity between a ***Task Property*** of input I and an input parameter P.

$$\begin{gathered} sim(Task_i^I, P) = \\ M * sim(IP(Task_i^I), Text_p) + N * sim(Text(Task_i^I), Task_p) \end{gathered} \tag{6.7}$$

M is the weight for the Input Similarity and N is the weight for the Textual Similarity between the user input and the input parameter. We assign 0.5 to M and 0.5 to N. The weights are derived empirically. $sim(Task_i^I, P)$ *defines the similarity between the* ***Task Property*** *of the user input and the input parameter.*

$$sim(Task_i^I, P) = \begin{cases} 1 & \text{if perfectly matched} \\ 0 & \text{if completely not matched} \end{cases}$$

We select a $Task_j^I$ $(1 \leq j \leq n)$ with the highest similarity value with the input parameter as the match for the input parameter.

6.3 Case Studies

We introduce our case study setup and the research questions. For each question, we present the motivation behind the question, the analysis approach and our findings.

6.3.1 Case Study Setup

We conduct our study on two types of services: SOAP-based services & RESTful with service description, and web applications without service description. In our dataset, the SOAP-based services are in WSDL; the RESTful services are described in web pages.

Table 6.1: Descriptive statistics of our collected public web services.

Domain	**# of WSDL and REST**	**# of Web forms**
Travel	235	30
E-commerce	192	15
Finance	164	10
Entertainment	109	10

Collecting public web services

In total, we download and collect 640 public available WSDL files. We use programmableWeb[7] to collect 60 URLs of RESTful services and download the web pages containing the description of APIs. We manually collect the contextual information related to RESTful services, such as the description of resources.

The 700 services with service descriptions fall into 4 domains: Travel (e.g., book flights), E-commerce (e.g., buying shoes), Finance (e.g., check a stock price) and Entertainment (e.g., search TV shows). We use Google to search for websites for each domain to download web forms. We choose the websites listed on the top of the Google result sets. In total we download 50 web forms. Table 6.1 provides a summary of the dataset used in our case study.

[7]http://www.programmableweb.com/

Collecting user inputs and user activities

To evaluate the effectiveness of our pre-filling approach, we require a set of recorded data for input fields. We collect user inputs through our input collector proposed in Section 6.2.2. We recruit five graduate students who typically spend 8-10 hours per day on-line in addition to the author of this thesis for user inputs collection. Four of the subjects are female and the other two are male. Their ages are from 25-35. The six subjects (i.e., five recruited graduate students and the author of this thesis) used the tool proposed in Section 6.2.2 to track their inputs and activities for three days (e.g., which web forms and input parameters used) on web forms requiring their information, such as name, email, address, and phone number. Furthermore, we use different values for diverse tasks. For example, we use different email addresses for various tasks such as login, contact information.

To test the effectiveness of our approach for linking similar end-users, we manually identify the most similar subject to a subject among the other five subjects and label the identified subject. Each subject has a labeled most similar subject. We use the 6 labeled subjects as the gold standard to verify the results of our approach.

6.3.2 Research Questions

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

RQ 1. Can the proposed Input Parameter Context Model help identify similar parameters?

Motivation. Linking similar parameters is critical to propagate end-user inputs across services. In this question, we measure the effectiveness of identifying similar parameters using our proposed Input Parameter Context Model (IPCM).

Approach. We build a baseline approach that uses the names of parameters to identify similar parameters. We compare the baseline approach with our approach using Input Parameter Context Model to identify similar parameters.

We collect all of the input and output parameters from 700 WSDL and RESTful services, and 60 web forms separately. Then, we calculate the similarity value between each input parameter and the other input or output parameters using our approach and the baseline approach. Our approach and the baseline approach identify the most similar parameter (i.e., input or output parameter) for each input parameter. Therefore, the pairs of parameters are generated from both our approach and the baseline approach. Each pair has an input parameter and its most similar parameter (i.e., the parameter has the highest similarity value with the input parameter).

Table 6.2: Results of the accuracy of our approach and the baseline approach for identifying similar parameters.

Approach	**WSDL and REST**	**Web Forms**
Our approach	84%	89%
Baseline	68%	61%

We randomly sample the pairs of parameters collected from our approach and the baseline approach with confidence level 95% [19] for manual verification of the results. We randomly sampled 376 pairs generated from WSDL and REST services and 230 pairs generated from web forms, and manually verify the correctness of the pairs of parameters for each approach. We use Equation (6.8) to measure the accuracy of our approach and the baseline approach.

$$Accuracy = \frac{Number\ of\ Correct\ Pairs}{Number\ of\ Parameter\ Pairs} \tag{6.8}$$

Results. Table 6.2 shows the results of our evaluation. Our approach achieves an effectiveness of 84% and 89% on WSDL and RESTful services and web applications respectively. Our approach outperforms the baseline approach. The baseline approach heavily relies on the names of input parameters. If the names are not available or the words of names are not meaningful, the baseline approach fails in linking the services. Even though the name of parameters cannot help calculate semantic similarity, our approach can still identify the similar parameters because our approach uses three contexts of input parameters.

Summary of RQ1: Our approach outperforms the baseline approach that heavily relies on the names of input parameters on linking similar input parameters in our dataset.

RQ 2. Is our proposed pre-filling approach effective to fill in input parameters?

Motivation. It is essential to save end-users from repetitive data entries by pre-filling values to input parameters of operations of services. We are interested in evaluating our pre-filling approach using the task information of user inputs and the contexts of parameters. The task information of user inputs is stored in the proposed context-aware Meta-data Model proposed in Section 6.2.2. The contexts of parameters are described in the Input Parameter Context Model proposed in Section 6.2.3.

Approach. Pre-filling requires the similarity calculation between a user input and an input parameter to determine if there exists any opportunity for pre-filling. In this chapter, the user inputs are described using the context-aware Meta-data Model (MDM) and the contexts of a parameter are in the Input Parameter Context

Model (IPCM). Our pre-filling approach uses MDM and IPCM to calculate the similarity between a user input and an input parameter. To compare with our approach, we build a baseline approach based on the key-value pair model as mentioned in Section 6.2.2 to describe an end-user's input and the name of an input parameter.

To validate the results manually, we randomly sample the input parameters with the confidence level of 95% [19]. We randomly select 376 input parameters from WSDL and RESTful services and 230 input parameters from web forms. Both our approach and the baseline approach use the collected end-users' inputs using our input collector to fill in the selected input parameters. To measure the effectiveness of our approach, we calculate precision and recall using Equation (6.9) and Equation (6.10). The precision metric measures the fraction of the pre-filled user input parameters that are correctly pre-filled, while the recall value measures the fraction of all the sampled user input parameters that are correctly pre-filled.

$$Precision = \frac{|Correctly\ Filled\ Input\ Parameters|}{|Filled\ Input\ Parameters|} \tag{6.9}$$

$$Recall = \frac{|Correctly\ Filled\ Input\ Parameter|}{|Input\ Parameters\ Need\ To\ Be\ Filled|} \tag{6.10}$$

Table 6.3: Results of the evaluation of our approach and the baseline approach of pre-filling values to input parameters.

Approach	**WSDL and REST**		**Web Forms**	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Our Approach	75	18	81	42
Baseline	42	10	68	34

Results. Table 6.3 shows the results of our evaluation. Our approach outperforms the baseline approach. The recalls of the two approaches on WSDL and REST are relatively low, because the input parameters are randomly selected from 700 services and most of them do not require the collected end-users' inputs, such as email. However, web forms are usually designed to require the basic personal information. The baseline approach cannot distinguish the information related to different tasks. For example, a key *"contact phone number"* in the key-pair model can have multiple values, one value is used as a travel contact number and the other value is used as the contact number of receiving shipments from E-commerce websites. The baseline approach can only achieve a precision of 55% on average. However, our pre-filling approach achieves a precision of 78% on average.

Summary of RQ2: The results of our empirical study show that our approach of filling in input parameters is effective and outperforms the baseline approach that fails in distinguishing the information related to different tasks.

RQ3. Can our approach using previous user inputs of an end-user reduce the number of inputs required from the end-user?

Motivation. RQ1 and RQ2 measure the effectiveness of two different sub-steps of our overall approach. In this question, we evaluate the effectiveness of our overall approach for reducing the number of inputs which an end-user is required to enter in a single user environment. In this research question, we do not leverage the user inputs from similar end-users who performed the same task during the service composition.

Approach. To answer the research question, we create a set of composed services. More specifically, we combine web forms within a web application to form service compositions. For WSDL and RESTful services, we randomly select 30 services from each domain, and use the data flow dependency to link the selected services. Then, we remove the compositions which only have one service in the service compositions.

To test the effectiveness of our approach in reducing repetitive typing required for an end-user during service composition, we randomly select 5 service compositions from the generated compositions from each domain for each type of services (WSDL and REST, and web forms). To avoid errors in the compositions generated by an automatic approach, we manually correct the errors to remove the possible bias in our observations.

To test the effectiveness of our approach for reducing the number of input parameters in a single user environment, we exclude the steps of *Identifying Similar End-users* and *Collecting User Inputs from Similar End-users* which are related to a multi-user environment. We apply our approach on the randomly selected compositions of services. Our overall approach analyzes the parameters and generates a list of input parameters which need end-user's inputs. Our approach uses the collected user inputs of each subject (i.e., discussed in Section 6.3.1) to fill in the input parameters. We use Equation (6.11) to calculate the effectiveness of our approach for each subject. The nominator of Equation (6.11) calculates the difference of the numbers of input parameters requiring user inputs between without using our approach and using our approach on the selected compositions; the denominator of Equation (6.11) is the number of input parameters requiring user inputs without using our approach. Last, we use Equation (6.12) to measure the average effectiveness of our approach for six subjects.

$$Effectiveness_i = \frac{Before - After}{Before} \tag{6.11}$$

where **Before** *stands for the number of input parameters requiring end-user's inputs* *without using our approach on the compositions of services.* **After** *stands for the number of parameters requiring end-users to enter information after using our approach on the compositions of services.* i *stands for the* i_{th} *recruited end-user,* $1 \leq i \leq n$, *and* n *is the total number of recruited end-users.*

$$Avg - effectiveness = \frac{\sum_{i=1}^{n} Effectiveness_i}{\# \text{ of Recruited End-users}} \tag{6.12}$$

Table 6.4: Average effectiveness of evaluating our approach for reducing the number of input parameters.

Domain	**WSDL and REST**	**Web Forms**
Travel	38%	62%
E-commerce	40%	46%
Finance	25%	42%
Entertainment	28%	50%

Results. Table 6.4 shows the results of our approach. Our approach reduces on average 50% and 32.75% of the number of input parameters on web forms and WSDL& REST respectively. Our approach performs better on web forms, because the number of parameters which can be linked together from the web forms within an application domain is greater than that of parameters which can be linked together from WSDL & RESTs. For example, a user is planning a trip; he or she has to visit a web form to search for cruises and another web form for booking cars, a large portion of the input parameters from both web forms can be linked, such as *departure* and *return date*. Using our approach, the user just needs input values to a pair of *departure* and *return date* within a Web form.

Summary of RQ3: Our approach can considerably save end-users from on average 37% of information typing into services during the process of composing services. More specifically, our approach reduces on average 50% and 32.75% of the number of input parameters on web forms and WSDL& REST respectively.

RQ4. Is our proposed approach effective in identifying similar end-users?

Motivation. Similar end-users may enter the same value to an similar input parameter of the same or different services. Reusing user inputs among end-users can help an end-user fill in a value to input parameters especially when the end-user has few historical inputs (e.g., no previous inputs at all) or no proper values for the input parameters. Identifying similar end-users is the first step to reuse user inputs among multiple end-users. In this research question, we evaluate the effectiveness of our proposed approach for identifying similar end-users.

Approach. To answer the research question, we apply our approach of identifying similar end-users (i.e., discussed in Section 6.2.4) on the six collected user profiles of the subjects to identify similar end-users. For each recruited subject, our approach recognizes two similar subjects from the other five subjects by calculating the similarity between subject activities. Moreover, we rank the identified two subjects based on their similarity values from the highest to the lowest. We use the labeled 6 subjects described in Section 6.3.1 as gold standard to verify the results of our approach. We use Equation (6.13) to calculate the effectiveness of our approach on each recruited end-user. The value of the Equation (6.13) is 1 or 0, because each subject only has one labeled similar subject. Finally, we use Equation (6.14) to measure the overall performance of our approach on all of the six subjects.

$$\text{k-}Effectiveness_i = \#\ of\ Labeled\ End\text{-}users\ in\ Top\text{-}k\ Results \tag{6.13}$$

$$\text{Avg-k-effectiveness} = \frac{\sum_{i=1}^{n} \text{k-}Effectiveness_i}{\#\ of\ Recruited\ End\text{-}users} \tag{6.14}$$

Where k is the number of recommended end-users from our approach by detecting the labeled similar subject for a subject. i stands for the i_{th} *recruited subject,* $1 \leq i \leq n$*, and n is the total number of recruited subjects.*

Table 6.5: Average effectiveness of evaluating our approach for identifying similar end-users with different values of k.

Top-k	**Avg-Effectiveness**
Top-1	66.6%
Top-2	83.3%

Results. Table 6.5 shows that our approach can effectively identify similar end-users for an end user using end-user's activities. In 83% of the cases, the top two user profiles recommended by our approach contain the most similar end-user which is manually labeled in Section 6.3.1. Identifying the most similar end-users to an end-user can help our approach collect suitable user inputs in order to propagate the user inputs from one end-user to another. Our approach is designed to discover the labeled subject profile with a minimum recommended similar end-user profiles.

Summary of RQ4: Our proposed approach is effective in identifying similar end-users in our dataset.

RQ5. Can the user inputs from similar end-users help reduce the

number of inputs an end-user is required to enter?

Motivation. In **RQ3**, we show that our approach only using previous user inputs from the same end-user can help reduce on average 41.38% of the number of inputs which need him or her to provide. In this research question, we evaluate the effectiveness of our overall approach to use the previous user inputs of an end-user as well as the similar end-users for reducing the number of inputs requiring the end-user to enter.

Approach. To answer the research question, we conduct two experiments:

Experiment One. Unlike the approach used in **RQ3**, we apply our overall approach to include the steps of *Identifying Similar End-users* and *Collecting User Inputs from Similar End-users* for the service compositions composed in **RQ3**. For each recruited subject, our approach collects user inputs from the recommended similar end-users in the Top 2 similar end-users as discussed in **RQ4**. Our approach uses the previous user inputs of the recruited subject and his or her similar peers to pre-fill the input parameters.

Experiment Two. For each recruited subject, our approach also collects user inputs from the human-labeled similar subject of each recruited subject. Then, we use the previous user inputs from the human-labeled similar subject to pre-fill the input parameters.

For both experiments, each recruited user verifies the pre-filled values of our approach. We use Equation (6.11) to calculate the effectiveness of our approach for each recruited user. The Equation (6.11) calculates the percentage of repetitive typing that our approach can help end-users save. Then, we use Equation (6.12) to measure the average performance of our approach for six recruited subjects on reducing number of input parameters.

Table 6.6: Average effectiveness of evaluating our approach using similar end-users' inputs for reducing the number of input parameters in experiment one.

Domain	**WSDL and REST (Improvement)**	**Web Forms (Improvement)**
Travel	38% (+0%)	66% (+4%)
E-commerce	46% (+6%)	54% (+8%)
Finance	31% (+6%)	46% (+4%)
Entertainment	32% (+4%)	52% (+2%)

Table 6.7: Average effectiveness of evaluating our approach using similar end-users' inputs for reducing the number of input parameters in experiment two.

Domain	**WSDL and REST (Improvement)**	**Web Forms (Improvement)**
Travel	38% (+0%)	66% (+4%)
E-commerce	47% (+7%)	56% (+10%)
Finance	31% (+6%)	46% (+4%)
Entertainment	32% (+4%)	56% (+4%)

Results. Table 6.6 and Table 6.7 show that the user inputs from similar end-users of an end-user can help improve the input parameter pre-filling. The percentage in parentheses in both Table 6.6 and Table 6.7 expresses the improvement of our approach using the previous inputs of an end-user and his or her similar peers over our approach only using the previous inputs of the same end-user (i.e., in **RQ3**).

We further investigate the results of our experiments. We find that most of the improvement is from the correctly filled input parameters requiring user inputs related to user preference settings, such as input parameters in a shopping search criteria page of an E-commerce website. It is beneficial for an end-user to include the user inputs of his or her similar end-users in the process of parameter filling, when his or her previous inputs are not sufficient to fill in input parameters.

Comparing the results in Table 6.6 and Table 6.7, we find that our approach using the user inputs from the human-labeled subject to pre-fill input parameters in *Experiment Two* improves the results of our approach in *Experiment One* by 1% on WSDL and REST in e-commerce, 2% on Web Forms in e-commerce, and 2% on Web Forms in entertainment. The improvement confirms the importance of the most similar end-user discovery in the process of leveraging and using user inputs of similar end-users for an end-user's input parameter filling.

Summary of RQ5: The results of our empirical study show that the user inputs from similar end-users of an end-user can help improve the input parameter pre-filling. Most of the improvement is resulted from the correctly filled input parameters requiring user inputs related to user preference settings, such as input parameters in a shopping search criteria page of an E-commerce website.

6.4 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [111].

Construct validity threats concern the relation between theory and observation. In this chapter, the construct validity threats are mainly from extracting input parameters from RESTful services and web applications. Extracting information from web pages is a challenging task. For example, the positions of labels in web forms can be various. We adopt the approach in [90] to extract information from web pages.

Internal validity threats concern our selection of subject systems, tools, and analysis method. The first threat is from manually labeling subject profiles in the section of Experiment Setup (i.e., Section 6.3.1). We set guidelines before we conduct the manual labeling process. We paid attention not to violate any guidelines to avoid big fluctuation of results with the change of the experiment conductor. It is very common to include manual processes in research studies, such as the manual process in [7]. Second, we collected user inputs through 6 end-users. The number of subjects may be low, the main reason is that people are reluctant to give away their personal information. However, other related academic research studies also collect user inputs from a small number of subjects. For example, 6 subjects were recruited in [7]

6.5 Chapter Summary

Repetitively entering the same information into composed services causes unnecessary interruptions and decreases the efficiency of service execution. In this chapter, we propose an approach to prevent end-users from repetitive data entry tasks. Our approach propagates user inputs to different services among multiple end-users by identifying similar parameters and linking end-users who perform similar tasks. We propose a context-aware Meta-data Model to store an end-user's previous inputs. Our approach pre-fills the input parameters using the previous inputs from end-users. Our approach discovers similar end-users by calculating the similarity between the previous activities of end-users. Our approach also leverages and consumes the user inputs of the similar end-users to an end-user for pre-filling input parameter.

To identify a proper value for an input parameter, our approach achieves an effectiveness of 86.5% on average and outperforms the baseline approach. We also evaluate the effectiveness of pre-filling user inputs into parameters. Our results show that on average 78% of the input parameters are filled correctly and our approach outperforms the baseline approach by filling 23% more parameters on average. Furthermore, our overall approach can reduce on average 41% of input parameters requiring an end-user to enter values for composed services using the previous user inputs of a single end-user. With the addition of user inputs from the similar end-users to the end-user, our approach can reduce on average 46.5% of input parameters.

Chapter 7

Ranking User Inputs

Recommending right values to input parameters for end-users is an effective technique to assist end-users in filling out services. In this chapter, reusing the learned knowledge in the prior chapters (i.e., Chapter 4, 5, and 6), we propose a context-aware ranking framework to rank values for input parameters. Our framework adopts learning-to-rank (LtR) algorithms to automatically construct ranking models by integrating ranking features constructed from various types of information, such as user contexts and patterns of user inputs.

Chapter organization. The rest of this chapter is organized as follows. Section 7.1 shows an introduction of this chapter. Section 7.2 introduces learning-to-rank models. Section 7.3 delves into our proposed approach. Section 7.4 discusses the case studies. Section 7.5 describes the threats to validity. Section 7.6 concludes this chapter.

7.1 Introduction

When an end-user enters a value to an input parameter, an interaction between the user input and the input parameter is established. Such an interaction is denoted as ***UI-IP interaction***. The information of such interactions is essential for analyzing and learning user's previous inputing activities. One main technique of assisting end-users in filling out services is recommending proper values to input parameters for end-users. Existing recommending approaches suffer the following two drawbacks:

- **Lack of fully exploiting user contexts and usage patterns associated with user previous inputs.** Most existing approaches do not fully analyze two types of information in recommendation:
  - *Patterns of user inputs.*
  - *Contextual information of interactions, such as time and user location.*

  Lack of the analysis on patterns and contextual information makes existing approaches cannot achieve a high accuracy and efficiency in filling out services.

- **Limited ability of learning user input entry activities.** With the accumulation of user input entry activities over time, the importance of a type of information can change. Existing recommendation approaches cannot automatically detect the changes of the importance of a type of information. Failing to reflect the changes of the importance of types of information and the values of user inputs for input parameters could lead to a decrease of performance in ranking.

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. Our framework collects and organizes interactions between user inputs and input parameters with contextual information. We propose a context-aware 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 ranking-based approach uses the model for context-aware matching between user inputs and input parameters.

Our framework learns and analyzes the collected interactions between user inputs and input parameters to rank user inputs for input parameters under different contexts. To learn and analyze interactions of user inputs and input parameters, we adopt the framework of learning-to-rank (LtR) [36][37] that consists of a set of supervised machine learning approaches to automatically build ranking models from training data built from user input entry activities. 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 propose 11 ranking features into our ranking model. The proposed ranking features capture various aspects of interactions between user inputs and input parameters. 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. Furthermore, we conduct more empirical experiments to discover the most influential ranking features among

11 ranking features on the ranking of user inputs.

We test the effectiveness of our framework on real-world services through a series of empirical studies. First, we identify the best performing learning-to-rank model (LtR) from 8 LtR models on ranking user inputs for our framework. Our results suggest that RankBoost [30], a pairwise LtR model, can outperform other LtR models on ranking user inputs. Second, we build two conventional ranking approaches (Bayesian Belief Network (BBN) [23] based ranking and frequency-based ranking) as baselines. We compare these two baselines with our framework utilizing RankBoost (denoted as framework-RankBoost). Our framework-RankBoost outperforms the baseline ranking approaches. Furthermore, we observe that our framework utilizing any LtR model can outperform the two baselines and Google chrome auto-filing tool [32] on ranking user inputs. Last, we observe that the textual similarity based features affect the ranking most.

7.2 Learning-to-Rank Models

In addition to the related research introduced in Chapter 3, we introduce a set of related studies that are only related to this chapter.

In this thesis, our ranking framework utilizing learning-to-rank models to rank user inputs for input parameters. In this section, we introduce the basic concepts of learning-to-rank (LtR) models and different types of LtR.

Learning-to-rank has been employed in a wide variety of applications in Information Retrieval [51]. We take document retrieval as an example to explain the learning-to-rank.

A learning-to-rank (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_1. 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 data. The feature vectors are ranked according to $F(x)$, then the top K results are evaluated using their corresponding relevance judgments. LtR approaches can be modified and developed into settings where no training data is available before deployment [25]. Compared with the conventional ranking models (e.g., Bayesian Belief Networks [23] 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.* Each query-document pair in the training data has a numerical or ordinal score. The training is viewed as an ordinal classification or regression problem. *(2) Pairwise approaches.* The learning-to-rank problem is approximated by a classification problem, learning a binary classifier that can tell which document is better in a given pair of documents. The goal is to minimize average number of inversions in ranking. *(3) Listwise approaches.* The listwise approaches take ranked lists of objects as instances in learning and learn the ranking model.

7.2.1 Pointwise Algorithms

Pointwise algorithms transform the ranking problem to regression or ordinal classification. For example, **PRank** [22] employs the Perceptron algorithm [74], based on Stochastic Gradient Descent [8], to simultaneously learn linear models. Given training data, PRank iteratively learns a number of parallel Perceptron models, and each model separates two neighboring grades. **OC SVM** [80] is a large margin approach for ordinal classification. OC SVM learns parallel hyperplanes to separate neighboring grades following the Large Margin Principle. Two ways of defining the margin are considered: (1) margins between the neighboring grades are the same and the margin is maximized in learning; (2) different margins are allowed for different neighboring grades, and the sum of margins is maximized.

7.2.2 Pairwise Algorithms

In this subsection, we analyze the selected representative pairwise algorithms, and discuss them in three groups based on the learning models employed.

The Support Vector Machine (SVM) [21] is a supervised machine learning model for classification and regression analysis. Some LTR approaches use the SVM as a learning model. For example, **Ranking SVM** [36][37] formalizes the ranking problem as a pairwise classification problem and employs the SVM technique to perform the learning task. **IR SVM** [17] emphasizes two important factors in ranking for document retrieval: (1) Correctly ranking documents on the top of list is crucial; (2) The number of relevant documents can vary from query to query.

Some other pairwise approaches employ Boosting [12], a supervised machine learning algorithm for reducing bias and variance, as a learning model. For example, **RankBoost** [30] is based on the gradient boosting technique. In [30], theoretical results are shown to describe the algorithm's behavior both on the training data and the test data. **GB Rank** [114] is a regression ranking framework for web search, employing a pairwise regression loss function. GB Rank utilizes a novel optimization method based on the Gradient Tree Boosting algorithm to iteratively minimize the loss function. **GB Rank** extends the boosting technique in **RankBoost** to handle substantially more complex loss functions arising from a variety of machine learning problems.

Neural networks [9], a family of statistical learning models, have been used by some pairwise approaches. For example, **RankNet** [13] is based on pairwise classification like Ranking SVM and RankBoost. The major difference lies in that RankNet employs Neural Network as a ranking model and uses Cross Entropy as a loss function. Gradient Descent is used by RankNet to learn the optimal Neural Network model. **Frank** [88] uses the Neural Network model optimized with Gradient Descent as a learning model.

7.2.3 Listwise Algorithms

The listwise approach takes ranking lists as instances in both learning and prediction. The group structure of ranking is maintained and ranking evaluation measures are more directly incorporated into the loss functions. We analyze 8 listwise algorithms.

The selected algorithms can be categorized in two different groups based on the way of optimization algorithms.

The IR measures, such as Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) [42], are used to measure the performance of IR approaches. One group of algorithms optimize objective functions that are bounds of the IR measures. For example, **SVM MAP** [112] can efficiently find a globally optimal solution to minimize an upper bound of the loss function, which bounds from 1-AP from above, based on MAP. **AdaRank** [110] minimizes an exponential loss function directly on an evaluation measure by using the Boosting technique.

Some other listwise approaches do not create surrogate functions based on IR measures. For example, **ListNet** [18] uses the Neural Network model for training and employs the Kullback-Leibler (KL) divergence [49] as a loss function. The permutation probability or top k probability of a list of objects is calculated by the Luce-Plackett model [55][67]. **LambdaRank** [69] uses an implicit listwise loss function optimized with Gradient Descent to learn the optimal ranking model. LambdaRank defines the gradient function of the loss function, referred as Lambda Function. LambdaRank employs a Neural Network model for training. **LambdaMART** [14][103] uses the Boosting technique and the Lambda function in LambdaRank. LambdaMark uses the Gradient Tree Boosting [31] technique to learn a boosted regression tree as a ranking model. Wu et al. [103] have verified that the efficiency of LambdaMART is significantly better than LambdaRank, and the accuracy of it is also higher.

7.3 Overview of Our Ranking Approach

In this chapter, we propose a ranking approach that learns and analyzes user contexts and user history to recommend values to 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 user input entry activities in order to provide accurate rankings of user previous inputs for users. We collect interactions of user inputs and input parameters (*UI-IP interactions*) through users' web activities. To store and organize *UI-IP interactions*, we propose a context-aware meta-data model that captures textual information and contexts of *UI-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 11 ranking features constructed from user previous inputing activities to rank the user inputs.

7.3.1 Collecting and Storing User Inputs with Contextual Information

To help users fill in input parameters under different contexts, we need a mechanism to organize and store user inputs efficiently. We extend our prior context-aware meta-data model proposed in [97] to include contextual information of interactions between user inputs and input parameters, in addition to the basic information (e.g., a textual human-readable label in a web form for a user input),

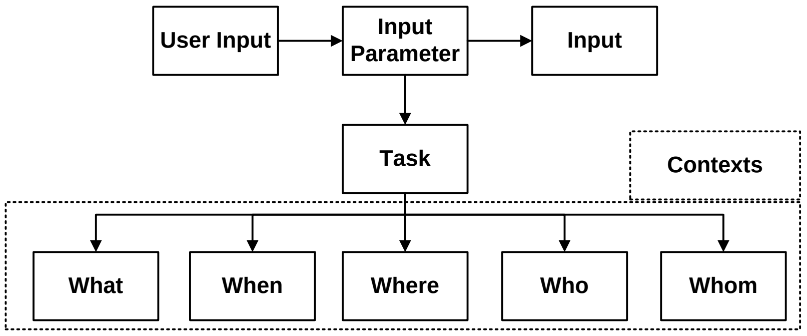


Figure 7.1: Description of context-aware meta-data model

Context-aware Meta-data Model

The description of our context-aware meta-data model is illustrated in Figure 7.1. A user input is associated with an *Input Parameter*. An *Input Parameter* has two properties: *Task* property and *Input* property.

Task Property: A *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 services. For WSDL services, we view an operation as a task. For RESTful services, we consider a resource with its associated actions as a task. For web application services, we define a web form 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 the name of an operation in a WSDL file, the resource name and its associated actions of a RESTful service, and a web label (e.g., from a web form or HTML tag).

  - ***Text*** stores the description of a task. A task description is retrieved from the description of an operation in WSDL, the web page introducing a resource and its associated actions in a RESTful service, or textual information of a web form. Since the description of a task may not always be available, the value of this property can be NULL. 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 name of a WSDL, RESTful service, or the URL of a web form.

- ***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. For example, the identity of a user can be a device's Media Access Control (MAC) address.
- ***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: An *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 mete-data 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[1] a lexical database to remove non-English words. We remove stop words using a pre-defined list[2] of stop words, such as "the". Finally, we use porter stemmer [68] to reduce derived words to their stem, base, or root form. For example, "reserves" and "reserved" are mapped to the stem form "reserve".

Collecting user inputs and their contexts

We collect user input entry activities through users' web activities (i.e., web forms). For example, users can buy a pair of shoes on Ebay[3] or register a course by filling in university registration web forms. Users can operate a simple RESTful web service using its URL through web browsers, or a SOAP-based WSDL service through SOAPUI framework[4]. User inputs can be collected through such a massive amount of user activities. We modify the tool used in [96] to collect interactions of user inputs and input parameters. The original tool in [96] extends a web testing tool named Sahi [5] to monitor users' web activities and collect the user inputs. Sahi embeds Javascripts into

[1]http://wordnet.princeton.edu/
[2]http://www.ranks.nl/stopwords
[3]http://www.ebay.com
[4]http://www.soapui.org/
[5]http://sahi.co.in/

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 the user inputs with their property information in the proposed context-aware meta-data 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 [2]. We collect four types of contexts: time, location, identity and activity, for interactions of user inputs and input parameters. We collect context values at different levels of granularity for each context type. Table 7.1 shows the contexts of each context type.

Table 7.1: User contexts collected for each context type.

Context Type	**User Contexts**	**Example**
Time	year, month, day of the month, day of the week, hour, am or pm (6 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"

When an interaction of an input parameter and an input parameter occurs, we take the follows 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[6] and parse it into contexts at different levels of granularity. To be able to know the type and name of a current user location, we allow a user to label the location for us. We assign null to any context value 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 process of typing values, 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.

[6]https://developers.google.com/maps/documentation /javascript/examples/map-geolocation

7.3.2 Recommending Values to Users for Filling in Input Parameters

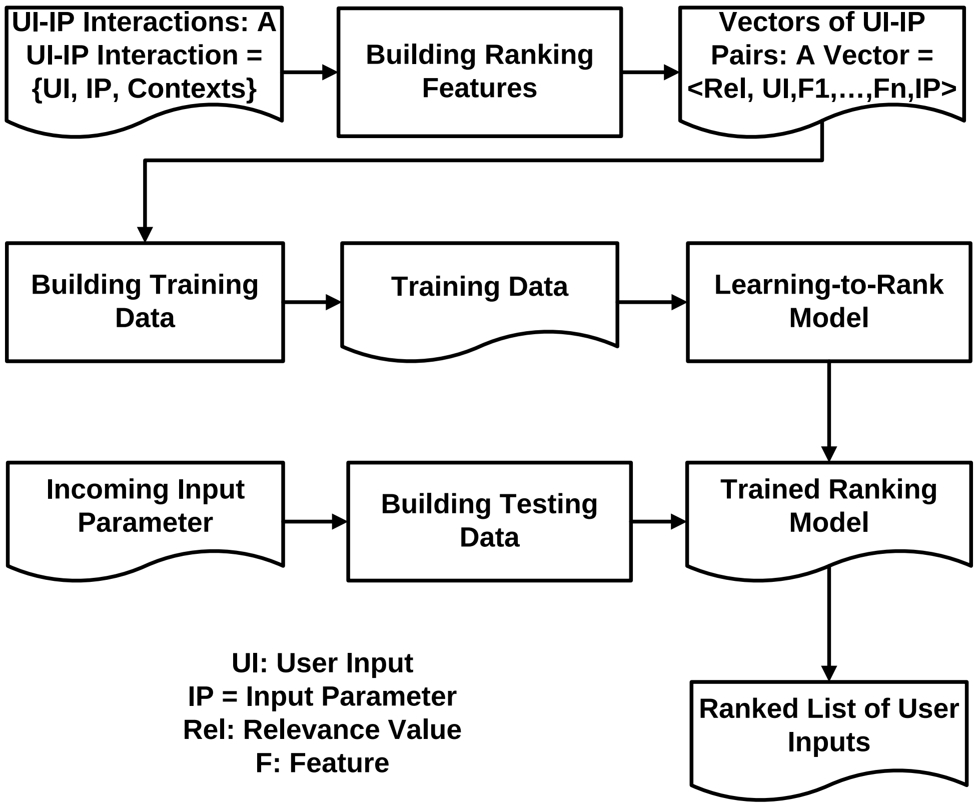


Figure 7.2: Main steps of our ranking framework.

We store all previous user inputs in the proposed context-aware meta-data model (Section 7.3.1). To help a user fill in 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 7.2 shows the main steps of our ranking framework. First, we build a vector of ranking features for each pair of user input and input parameter in UI-IP interactions. Second, we create training data based on UI-IP pairs with feature vectors. The training data is used by learning-to-rank models to generate trained ranking models. Last, we create testing data for the input parameter to rank user inputs for an input parameter.

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 and start typing a new value. Before the typing starts, a list of ranked user previous inputs are recommended to the user.

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 in the input parameter. A list of ranked user previous inputs are recommended based on what the user is typing.

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.

Our Learning-to-Reuse Model

To learn from user input entry activities for reusing user inputs (learning-to-Reuse), we adopt learning-to-rank in the task of ranking user inputs. We view an input parameter (IP) and a previous user input (UI) as a query and a document respectively, denoted as IP-UI pair. We build ranking features and calculate feature values for each IP-UI pair. The value of a relevance judgment on a user input UI with respect to an input parameter IP (IP-UI pair) can only be 0 or 1, where 1 means the UI 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 $= < Relevance, UI, feature1, \ldots, featureN, IP >$, where N is the total number of ranking features.

Table 7.2: Outline of our ranking features. UI: user input. IP: input parameter. t: text. k: task. F: Feature.

Category	**Rationale and Explanation**	**Features**
Textual similarity	A UI textually similar to an IP could be the right value and selected by a user. We adopt 3 approaches, cosine similarity [84], Jaccard's coefficient [33], and WordNet-based approach (WNA) [97] to calculate the textual similarity between the text of UI and the text of IP.	F1=Cosine(UI_t,IP_t); F2=Jaccard(UI_t,IP_t); F3=WNA(UI_t,IP_t)

Task similarity	If a task of a UI is textually similar to a task of an IP that is being performed by a user, the UI could be selected by the user. Like above, we adopt the same 3 approaches to calculate the similarity between the task of UI and the task of IP.	F4=Cosine(UI_k,IP_k); F5=Jaccard(UI_k,IP_k); F6=WNA(UI_k,IP_k)
Frequency	More frequent a UI is used for an IP or multiple IPs, more chances the UI will be used next time. We calculate two frequencies of a UI: (1) # of times the UI used for a particular IP (denoted as Freq_par); (2) # of times the UI used for all IPs (denoted as Freq_all)	F7=Freq_par; F8=Freq_all
Time length	The recently used UIs could be used again while the oldest UIs may not be used. We calculate the length of time from current time to the time of last use of a UI (denoted as T_Length).	F9=T_Length

Patterns	When a set of UIs or IPs occur together multiple times, a pattern is formed. Patterns can improve recommending. For example, a set of two UIs are always entered together. When one of them is used, the other one could also be used. Each UI has unique id in our dataset. Among all UIs, we discover the UIs having a pattern. We use # of such UIs as the number of features under this category. For a UI, we calculate the number of times the UI is used with each of discovered UIs having a pattern.	Fi=# of times used with a UI having a pattern. n is the number of UIs having a pattern, $10 \leq i \leq (n+10)$.
Contexts	An individual context or combinations of contexts could affect the ranking of user inputs. The number of context features is the number of k-combinations from a given set S of n contexts (denoted as C_n^k), k= 1,..., n. For a context feature, we calculate # of times the contexts of the feature appear in the past.	Fj = # of times a k-combination of n contexts appear in the past. $(i+1) \leq j \leq \sum_n^k C_n^k$, $1 \leq k \leq n$, i is the last feature number of the set of above pattern features.

Building Ranking Features

To learn from user input entry activities, we propose 11 ranking features that capture 6 different perspectives of interactions of user inputs and input parameters. Table 7.2 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 [84], Jaccard's coefficient [33], and WordNet-based approach (WNA) [97], 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 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 (7.1) to calculate a cosine similarity between set_i and set_j.

$$Cosine(set_i, set_j) = \frac{\sum_{m=1}^{n} v_m^{set_i} v_m^{set_j}}{\sqrt{\sum_{m=1}^{n} (v_m^{set_i})^2} \sqrt{\sum_{m=1}^{n} (v_m^{set_j})^2}} \tag{7.1}$$

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 (7.2) to calculate the textual similarity between set_i and set_j.

$$Jaccard(set_i, set_j) = \frac{|set_i \cap set_j|}{|set_i \cup set_j|} \tag{7.2}$$

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) [97] to calculate the semantic similarity between set_i and set_j in the following steps: If $set_i \subset set_j$, we use Equation (7.2) 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 $Sim_{sum} = \sum_{W_a^i \subset set_i} \sum_{W_b^j \subset set_j} \text{sim}(W_a^i, W_b^j)$, where $\text{sim}(W_a^i, W_b^j)$ denotes the similarity value between two words; $W_a^i \neq W_b^j$. Last, we use Equation (7.3) 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.

$$WNA(set_i, set_j) = \frac{|set_i \bigcap set_j| + Sim_{sum}}{|set_i \bigcap set_j| + |\# \, of \; calculation|} \tag{7.3}$$

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 calcualte the textual similarity between a user input (UI) and an input parameter (IP),

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

Task similarity based features. If one of the tasks associated with a user input (UI) can match the current task being performed by a user, the UI 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 7.3.1. 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 the input parameter (IP), denoted as $sim^{name}(IPP_{ui}, IP)$;

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

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

We use Equation 7.4 to calculate the task similarity between an IPP of a user input and an input parameter (IP).

$$\begin{gathered} sim\_task(IPP_{ui}, IP) = w_n * sim^{name}(IPP_{ui}, IP) + \\ w_d * sim^{des}(IPP_{ui}, IP) + w_s * sim^{service}(IPP_{ui}, IP) \end{gathered} \tag{7.4}$$

where w_n, w_d, w_s are weights for similarity calculation. w_n, w_d, and w_s have same value (i.e., $1/3$) derived empirically.

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 (UI) and an input parameter (IP),

- **Feature 4** uses Cosine similarity, denoted as F4 = Cosine(UI_{task}, IP_{task}).
- **Feature 5** uses Jaccard similarity, denoted as F5 = Jaccard(UI_{task}, IP_{task}).
- **Feature 6** uses our WordNet approach (WNA), denoted as F6 = WNA(UI_{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 set of stored interactions of user inputs and input parameters (UI-IP interactions), and a user input and an input parameter, we analyze the UI-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 UI-IP interactions (denoted as Freq_all). Based on two types of frequencies, we build two ranking features.

Frequency Based Features: Given a user input and an input parameter,

- **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 interactions of user inputs and input parameters (UI-IP interactions), and a user input, we analyze the UI-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

Patterns based features. User inputs can be used together to accomplish a task. When patterns of user inputs are formed, the recommendation can be improved. For example, a set of two UIs 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 user inputs as a pattern, if the set of user inputs are used together more than twice.

Given a set of stored interactions of user inputs and input parameters (UI-IP interactions), a user input (UI), we first discover patterns of user inputs. We group user inputs based on the number of times of tasks that user inputs are used together. Second, we mine sets of user inputs occur more than twice across different groups of user inputs, denoted as $patterns^{ui}$. Third, we obtain a set of unique user inputs that have a pattern, denoted as set_p^{ui}. Last, for the UI, we calculate the number of times that the UI is used together with every $UI_i \subset set_p^{ui}$. When UI = UI_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 UI and UI_i are in a pattern, denoted as # of $Patterns^{UI_i}$. In total, we build K pattern ranking featuers, where K = the number of unique user inputs that are, at least, in a pattern.

Pattern Features: Given a user input (UI) and UI-IP interactions, we first discover a set of user inputs that are, at least, in a pattern, denoted as set_p^{ui}. We build K pattern ranking features, where K = # of unique user inputs that are, at least, used in a pattern.

Feature f# = # of $Patterns^{UI_i}$, f# is feature number, $10 \leq f\# \leq the\ size\ of\ set\ set_p^{ui} + 10$.

Contexts based features. The contexts associated with interactions between user inputs and input parameters are important for recommendation. Given an input parameter and a set of user contexts, we identify user inputs whose contexts match the set of user contexts. For example, a context can be a location or device where a user enters a value to a service. An individual context, such as types of location, could help ranking models, as well as the combinations of individual contexts. Therefore, we build ranking features based on individual contexts and combinations of individual contexts.

Each UI-IP interaction is associated with a set of contexts recorded during the execution. In this chapter, we collect 18 contexts. One of them is *user's current typing*, we use it as a filter to dynamically retrieve candidate user inputs for an input parameter in runtime. Therefore, we build ranking features based on 16 contexts, excluding the context *user's current typing*.

Given a set of UI-IP interactions and each UI-IP interaction has a set of values of m contexts $C = \{c_1, c_2, \ldots, c_m\}$, we use the following steps to build ranking features based on a set of m contexts C: First, we build k-combinations of context values for each UI-IP interaction. We obtain $\sum_n^k C_n^k$ combinations of context values for a UI-IP interaction, $1 \leq k \leq m$. When k=1, 1-combination means an individual context.

Each combination is a set of k contexts. Second, we calculate the number of times each k-combination of context values appear in the set of UI-IP interactions. To utilize every k-combination of contexts, we build a ranking feature for each k-combination of contexts. In total, for a UI-IP pair, we build $\sum_n^k C_n^k$ contexts based ranking features. Our model is extensible with more types of contexts which could potentially affect ranking.

For example, we have two UI-IP interactions, each interaction has 3 contexts: city, year, and who. The two interactions can be represented as:

$I_1 = \{UI_1, IP_1, Markham, 1999, Shaohua\}$ and

$I_2 = \{UI_2, IP_2, Markham, 2000, Ying\}$.

Calculating values for contexts features for I_1, we build the combinations of k contexts, $1 \leq k \leq 3$. We obtain 6 k-combinations of context values are { "Markham", "1999", "Shaohua", "Markham 1999", "Markham Shaohua", "Markham 1999 Shaohua"}. Then, we count the number of times that each combination (i.e., a set of contexts) appears in the two UI-IP interactions. In the end, values of contexts features of I_1 is {2, 1, 1, 1, 1, 1}. Because "Markham" appears twice, so the value for context city is 2.

Contexts Features: Given a set of UI-IP interactions and each UI-IP interaction has a set of values of n contexts, we build k-combinations of n context values for each UI-IP interaction. We build $\sum_n^k C_n^k$ ranking features, where n is number of contexts of each UI-IP interaction, $1 \leq k \leq n$.

Feature j = # of times a value of a k-combination of n contexts appear in UI-IP interactions; $i \leq j \leq (\sum_n^k C_n^k + i)$, i is the last feature index of patterns based features.

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

To have a proper ranking of user inputs, we need build scenarios containing positive and negative examples for input parameters to train a learning-to-rank model to rank user inputs responding to contexts. Each input parameter can have several scenarios. A scenario can only have one positive example showing a value entered into an 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 (UI) and an input parameter (IP) that are in a UI-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 UI-IP interaction shows that the user input (UI) is entered into the input parameter (IP), the relevance value for the pair of UI and IP is 1. After the ranking features are built for the UI-IP interaction, the UI-IP interaction can be represented as a vector

$V =< Relevance, UI, Feature_1, \ldots Feature_N, IP >$

The value of Relevance is 1.

(2) *Building negative examples for the given UI-IP interaction.* For each user input in Set^{others}, denoted as UI^o, we form a pair of UI^{others} and the IP, denoted as UI^o-IP pair. We assign 0 to the relevance value of UI^o-IP pair, because the UI^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 UI^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.

7.4 Case Studies

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

7.4.1 Case Study Setup

We conducted our study on three types of services: WSDL services (i.e., SOAP-based services), RESTful services, and web applications (i.e., web forms). The RESTful services are described in web pages.

Table 7.3: Our collection of public 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

Service collection

We collect 640 public available WSDL files. We use programmableWeb[7] to collect 120 URLs of RESTful services and download the web pages containing the APIs. We manually collect the information related to RESTful services, such as the description of resources and input parameters. 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. 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 7.3 provides a summary of the dataset used in our case study. In total, we collect 11127, 792, 1024 input parameters from WSDL, Restful, and Web application services, respectively.

User input collection

To evaluate the effectiveness of our framework, we collect interactions of user inputs and input parameters (UI-IP interactions) using our input collector (Section 7.3.1). We recruit 6 subjects who are graduate students and typically spend 8-10 hours online 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 UI-IP interactions as listed in Table 7.1, 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 of user inputs and input parameters over a week. For each UI-IP interaction, we calculate a feature vector using all the 11 ranking features. We refer this automatically collected dataset as *auto-data*.

[7]http://www.programmableweb.com/

Manually-labeled training dataset

As the *auto-data* is only collected from web forms, to test our approach on the collected 960 services, 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% [19] so that we can manually identify user inputs from *auto-data* for set_{ip}^{Sample}. We randomly sample 371, 259, 280 input parameters for WSDL, Restful, and Web application services, 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 7.3.1. Last, we calculate the feature values for all the 11 ranking features. We refer the manually labeled dataset as *manually-labeled-data*.

7.4.2 Research Questions

We conduct experiments to measure the effectiveness 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. Our framework incorporating the framework of learning-to-rank (LtR) learns and analyzes user input entry activities. The learning-to-rank framework consists of a set of LtR models. To maximize the performance of our framework, it is critical to select and deploy a best suitable LtR model in our framework. In this research question, we compare 8 LtR models utilized by our framework to discover the best performing learning-to-rank model on ranking user inputs.

Approach. Once training and testing dataset is ready, our framework needs select a learning-to-rank (LtR) model for training and ranking user inputs for input parameters. Our framework can utilize any LtR model. To answer this research question, 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:

We analyze 4 pairwise LtR models:

1. **RankSVM** [36][37] formalizes the ranking problem as a pairwise classification problem and employs the SVM technique to perform the learning task.

2. **IR SVM** [17] emphasizes two important factors in ranking for document retrieval: Correctly ranking documents on the top of list is crucial; and The number of relevant documents can vary from query to query.

3. **RankBoost** [30] is based on the gradient boosting technique.

4. **RankNet** [13] is based on pairwise classification like RankSVM and RankBoost.

We analyze 4 listwise LtR models:

1. **AdaRank** [110] directly optimize an evaluation measure by using the Boosting technique.

2. **ListNet** [18] uses the Neural Network model for training and employs the Kullback-Leibler (KL) divergence [49] as a loss function.

3. **LambdaRank** [69] uses an implicit listwise loss function optimized with Gradient Descent to learn the optimal ranking model.

4. **LambdaMART** [14][103] uses the Gradient Tree Boosting [31] 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 [52]. To test 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 7.4.1. For each dataset, we build vectors of features for interactions of 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 [104]. 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 apply the trained LtR models to generate rankings of user inputs for the given set of input parameters. Last, we use Equation (7.5) and Equation (7.6) to measure k-precision and k-recall for recommending a list of user inputs to P_i $(0 < i \leq n)$. We use Equation (7.7) and Equation (7.8) to calculate the average precision and recall.

$$k - precision_i = \frac{|Correct\ Inputs\ in\ top\ k\ results|}{k} \tag{7.5}$$

$$k - recall_i = \frac{|Correct\ Inputs\ in\ top\ k\ results|}{|Correct\ Input|} \tag{7.6}$$

where $|$ *Correct Input* $|$ *is a constant and = 1, because there can only be one correct input for a* P_i. $|$ *CorrectInputsintopkresults* $|$ *= (1 or 0). When k=1, k-precision*$_i$*= k-recall*$_i$*.*

$$avg - k - precision = \frac{\sum_{i=1}^{n}(k - precision_i)}{n} \tag{7.7}$$

$$avg - k - recall = \frac{\sum_{i=1}^{n}(k - recall_i)}{n} \tag{7.8}$$

Table 7.4: Performance of different approaches with different k values on ***auto-data***. 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. Table 7.4 shows that on dataset *auto-data*, 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. The results in Table 7.5 suggest that generally RanKBoost can outperform other LtR models. Only LambdaMart can slightly work

Table 7.5: Performance of different approaches with different k values on ***manually-labeled-data***. P: Precision, R: Recall. All results of precision and recall are in percentage (%).

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

better than RanKBoost on Top-1 and Top-3 ranked user inputs for input parameters from WSDL services.

Overall, we select RankBoost as a learning-to-rank model for our framework. Moreover, the results of Table 7.4 and Table 7.5 show that the performance of a ranking built on automatically collected interactions of 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.

Summary of RQ1: RankBoost performs the best in ranking the collected user inputs for the input parameters in our dataset, 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. Ranking user inputs to input parameters is an essential step to save users from repetitive typing. Ideally, a user is satisfied with a pre-filled value, the top-1 value of a ranked list, for an input parameter. However, a new ranking of user inputs should be provided, if the user is not satisfied with the pre-filled value. The ranking should be adjusted and adapted to the dynamic changes of user contexts. In this question, we evaluate the effectiveness of our framework in ranking user inputs.

Approach. To answer this research question, 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) [23] 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 7.3.2) 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 [104]. 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 (7.5) and Equation (7.6) to measure k-precision and k-recall for recommending a list of user inputs to P_i $(0 < i \leq n)$. We use Equation (7.7) and Equation (7.8) to calculate the average precision and recall.

Table 7.6: Performance of different approaches with different 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 7.6 and Table 7.7 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 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,

Table 7.7: Performance of different approaches with different 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

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 in our dataset. 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 of 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 (7.7) and Equation (7.8) 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. The context features are generated by combining the 17 individual contexts, we cannot test all generated features. Therefore, we test the effect of each context type: time, location and identify on the ranking of user inputs. There are no features built on the context type Activity, so we exclude it from our analysis. We use the same above approach in *Experiment 1* to test a feature in a category.

Table 7.8: 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%

Results. Table 7.8 shows the effects of categories of ranking features. The percentages in Table 7.8 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, we observe that compared with other types of contextual information, user location and time make the largest impact on the rankings of user inputs.

Summary of RQ3: The results of our empirical study show that the category of textual similarity based ranking features can make the largest impact on the rankings of user inputs in our dataset, 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) [32] 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% [19] 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 of user inputs and input parameters (UI-IP interactions) of *auto-data* into these input parameters. Since UI-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 UI-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 (7.7) and Equation (7.8) to measure the effectiveness of our framework using RankBoost and Chrome-fill on ranking user inputs.

Table 7.9: Results of our framework using RankBoost and Chrome-fill on recommending user inputs. P: Precision, R: Recall.

Domain	**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 7.9 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: The results of our empirical study on our dataset suggest that 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.

7.5 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [111].

Construct validity threats concern the relation between theory and observation.

In this chapter, the construct validity threats are mainly from extracting input parameters from RESTful services and web applications. Extracting information from web pages is a challenging task. For example, the positions of labels in web forms can be various. We adopt the approach in [90] to extract information from web pages.

Internal validity threats concern our selection of subject systems, tools, and analysis method. The main threats are from manually labeling the training dataset. It is very common to include manual processes in research studies, such as the manual process in [7]. Second, we collected user inputs through 6 end-users. The number of subjects may be low, the main reason is that people are reluctant to give away their personal information. However, other related academic research studies (e.g., [7] and [101]) either collect user inputs from a small number of subjects or no subjects involved.

7.6 Chapter Summary

Unnecessary interruptions caused by repetitively typing same information to services decreases the efficiency and negatively impacts the user experience. In this chapter, we propose a learning-to-rank (LtR) model based framework to recommend user inputs for input parameters. Our framework consists of a context-aware model for storing user inputs with contextual information efficiently. Our framework can analyze and learn interactions of user inputs and input parameters by utilizing learning-to-rank models. Through a series of empirical studies, after comparing 8 LtR models, we discover that our framework using RankBoost, a pairwise LtR model, can achieve the best performance on ranking user inputs. Our empirical results show that our context-aware ranking framework utilizing any LtR model can outperform the two baseline approaches (i.e., Bayesian Belief Network and Frequency based approaches). By studying the effect of distinct features on the ranking performance, we observe that the textual similarity based ranking features affect the ranking the most compared with other categories of features. Among various types of contextual data, user location (i.e., physical locations and devices) and time matter most to ranking.

Chapter 8

Conclusions and Future Work

To invoke web services, end-users need to enter values to input parameters of web services. However, quite often, a huge chunk of previously entered values are same. Unnecessary interruptions caused by repetitively typing the same information to services decreases the efficiency and negatively impacts user experience. To save end-users from repetitive typing, analyzing users' past entry activities is critical to reuse previous user inputs. In this thesis, we propose a set of approaches to improve the process of filling out services by analyzing and reusing previous user inputs used under various conditions. In the rest of this chapter, we outline the contributions of this thesis and lay out the promising research opportunities for future research.

8.1 Contribution

The main goal of this thesis is to improve the process of filling out services by saving end-users from entering the same information into web services repetitively. Broadly speaking, we propose approaches to analyze users' past data entry activities to reuse previous user inputs among services and end-users.

We improve the process of filling out services from four aspects. We list the main findings of this thesis.

(1) **Characteristics of input parameters (Chapter 4)**. We first investigate the natures of various types of input parameters. We propose categories for input parameters and an automatic approach for categorizing unseen input parameters. We utilize the proposed categories to improve the process of filling out services

Different natures of types of input parameters (Chapter 4): The main findings are: (1) Our proposed categories can represent all of the input parameters in our corpus. (2) Our proposed approach of categorizing unseen input parameters is effective. (3) Utilizing our proposed categories in the process of filling out services can improve the accuracy of filling out services.

(2) **User contexts and usage patterns (Chapter 5)**. We study the impact of user contexts and usage patterns on the process of filling out services. We propose a comprehensive framework that discovers similar user interface components, collects user inputs, and generate patterns of user inputs for filling out services.

Incorporating user contexts and usage patterns in filling out services (Chapter 5): The main findings are: (1) Our proposed approach is effective in identifying similar user interface components. (2) Utilizing user contexts and patterns of user inputs can improve the accuracy of filling out services. (3) Our approach can outperform existing tools, such as Firefox Autofill and Chrome Autofill tools.

(3) **Maximum propagation of user inputs (Chapter 6)**. We propose approaches to maximize the reuse of previous user inputs across services and end-users.

We propose a model that stores task information of user inputs and an input parameter model that stores features of input parameters. We link semantically related parameters using the features of input parameters. We propose an approach to identify similar end-users who can share user inputs among each other.

Maximizing the propagation of user inputs across services and end-users (Chapter 6): The main findings are: (1) Our proposed approach is effective in identifying semantically related parameters and similar end-users. (2) Our approach can save end-users from, on average, 41% of typing. (3) Leveraging the user inputs from other end-users can improve the process of filling out services.

(4) **Ranking user inputs (Chapter 7)**. We propose a ranking framework that analyzes and learns user contexts and previous data entry activities to recommend proper values for input parameters. More specifically, we propose a context-aware data model to store all of the contextual information of interactions between user inputs and input parameters. We propose a ranking approach that analyzes the user inputs in the context-aware data model and recommend values to input parameters.

Ranking user inputs for filling out services (Chapter 7): The main findings are: (1) Our ranking framework outperforms Chrome Autofill tool on recommending values to services. (2) Through empirical studies, we discover that RankBoost is the best performing learning-to-rank model on ranking user inputs. (3) The most influential ranking feature is textual-based.

8.2 Future Work

Although this thesis has made a positive impact on the improvement of the process of filling out services, there is plenty room for research. We outline a few promising avenues for future work.

8.2.1 Improving the Process of Filling Out Mobile Applications

Mobile devices become indispensable in people's life. Mobile applications running on mobile devices empower people to conduct various tasks on the go. To run the functionalities of mobile applications, values need to be provided to the interfaces of the functionalities. Due to the limitations, such as screen size and computation, filing out mobile applications is even more challenging and frustrating for end-users than web forms. A tremendous effort has been made to improve the accuracy of filling out web services. However, very less research is focused on the mobile application auto-filling. Analysis of the process of filing out interfaces of mobile applications is absolutely needed.

8.2.2 Understanding Interactions Between End-users and Web services (and Mobile Applications)

Every end-user has their own style of interacting with web services or mobile applications. In end-user driven service composition, it is critical to understand how end-users interact with web services so that end-users can be recommended with the proper web services and experience a smooth composition of services. More importantly, optimizing the interactions between end-users and web services can improve the process of filling out web services or mobile applications.

8.2.3 Enhancing the Semantics of Web Services

In this thesis, our proposed approaches of identifying similar input parameters cannot achieve a precision of 100% mainly due to the lack of semantics of input parameters and web services. Semantics can help understand the input parameters better so that similar input parameters can be linked and reuse previous user inputs. There is still a plenty of room for the research that improves the semantics of web services.

8.2.4 Increasing the Scale of Our Empirical Studies

In the future, we plan to recruit more end-users in our study and collect more user data. We plan to conduct our empirical studies for evaluating the effectiveness of our framework on a large scale user data.

Bibliography

[1] 1Password. https://agilebits.com/onepassword. Last accessed on March 19th, 2015.

[2] 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.

[3] M. AbuJarour and S. Oergel. Automatic sampling of web services. In *Int'l Conf. on Web Services*, pages 291–298. IEEE, 2011.

[4] A. Ali and C. Meek. Predictive models of form filling. Technical Report, MSR-TR-2009-1, Microsoft Research, 2009.

[5] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890. ACM, 2009.

[6] Apriori. http://en.wikipedia.org/wiki/Apriori_algorithm. Last accessed on March 19th, 2015.

[7] S. Araujo, Q. Gao, E. Leonardi, and G. Houben. Carbon: Domain-independent

automatic web from filling. In *2010 International Conference on Web Engineering (ICWE 2010)*, pages 292–306. LNCS, 2010.

[8] D. P. Bertsekas. *Nonlinear programming.* Athena scientific, 1999.

[9] D. P. Bertsekas and J. N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Int'l Conf, on Decision and Control*, volume 1, pages 560–564. IEEE, 1995.

[10] P. Bianco, R. Kotermanski, and P. F. Merson. Evaluating a service-oriented architecture. 2007.

[11] M. B. Blake, D. R. Kahan, and M. F. Nowlan. Context-aware agents for user-oriented web services discovery and execution. *Distributed and Parallel Databases*, 21(1):39–58, 2007.

[12] L. Breiman. Bias, variance, and arcing classifiers. 1996.

[13] 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.

[14] C.J. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11:23–581, 2010.

[15] Ç. Automated composition of e-services: lookaheads.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[17] Y. Cao, J. Xu, T. Liu, H. Li, Y. Huang, and H. Hon. Adapting ranking svm to document retrieval. In *ACM SIGIR conference on Research and development in information retrieval*, pages 186–193, 2006.

[18] 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.

[19] R. L. Chambers and C. J. Skinner. *Analysis of survey data.* John Wiley & Sons, 2003.

[20] C. Chang, M. Kayed, M. R. Girgis, and K. F. Shaala. A survey of web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1411–1428, 2006.

[21] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[22] K. Crammer and Y. Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems*, pages 641–647. MIT Press, 2001.

[23] 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.

[24] 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.

[25] 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.

[26] C. Fellbaum. *WordNet An Electronic Lexical Database.* MIT Press, 1998.

[27] Firefox. Autofill forms add-on. https://addons.mozilla.org/en-US/firefox/addon/autofill-forms. Last accessed on March 19th, 2015.

[28] 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.

[29] 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.

[30] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *Machine Learning Research*, 4:933–969, 2003.

[31] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[32] Google. Chrome autofill forms. https://support.google.com/chrome. Last accessed on March 19th, 2015.

[33] 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.

[34] M. Hartmann and M. Muhlhauser. Context-aware form filling for web applications. In *IEEE International Conference on Semantic Computing*, 2009.

[35] B. He, K. C. Chang, and J. Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 148–157. ACM, 2004.

[36] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. *Advances in neural information processing systems*, pages 115–132, 1999.

[37] R. Herbrich, T. Graepel, and K. Obermayer. Support vector learning for ordinal regression. In *Int'l Conf. on Artificial Neural Networks*. IET, 1999.

[38] 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.

[39] L. J. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: identification and analysis of coexpressed genes. *Genome research*, 9(11):1106–1115, 1999.

[40] iMarcos. http://imacros.net/overview. Last accessed on Sep. 12, 2015.

[41] TEL-8 Query Interfaces. http://metaquerier.cs.uiuc.edu/repository/datasets/tel-8/. Last accessed on March 19th, 2015.

[42] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[43] T. Kabisch, E. C. Dragut, C. Yu, and U. Leser. Deep web integration with visqi. *VLDB Endowment*, 3(1-2):1613–1616, 2010.

[44] R. Khare. Microformats: The next (small) thing on the semantic web? *Internet Computing*, 10(1):68–75, Jan/Feb 2006.

[45] R. Khare. The next (small) thing on the semantic web? *IEEE Internet Computing*, 8, 2008.

[46] H. Kriegel and M. Schubert. Classification of websites as sets of feature vectors. In *Databases and applications*, pages 127–132, 2004.

[47] 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.

[48] T. T. Kristjansson. Assisted form filling, 2008. US Patent 7,426,496.

[49] S. Kullback. *Information theory and statistics.* Courier Corporation, 1968.

[50] LastPass. www.lastpass.com. Last accessed on March 19th, 2015.

[51] H. Li. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems*, 94(10):1854–1862, 2011.

[52] H. Li. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies*, 7(3):1–121, 2014.

[53] L. Li and W. Chou. Automatic message flow analyses for web services based on wsdl. In *IEEE International Conference on Web Services*, pages 880–887. IEEE, 2007.

[54] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 66–73. ACM, 2004.

[55] R. Ducan Luce. *Individual Choice Behavior a Theoretical Analysis*. John Wiley and sons, 1959.

[56] Z. Maamar, D. Benslimane, G. K. Mostéfaoui, S. Subramanian, and Q. H. Mahmoud. Toward behavioral web services using policies. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 38(6):1312–1324, 2008.

[57] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *VLDB Endowment*, 1(2):1241–1252, 2008.

[58] Z. Malik and A. Bouguettaya. Rateweb: Reputation assessment for trust establishment among web services. *The International Journal on Very Large Data Bases*, 18(4):885–911, 2009.

[59] C. D. Manning, P. Raghavan, and H. Schütze. Scoring, term weighting and the vector space model. *Introduction to Information Retrieval*, 100, 2008.

[60] HTML Microdata. http://www.w3.org/TR/microdata. Last Accessed on December 11, 2015.

[61] Sun Microssystems Inc. Web application description language. http://www.w3.org/Submission/wadl/. Last accessed on August 25th, 2015.

[62] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg, and S. Dustdar. A context-based mediation approach to compose semantic web services. *ACM Transactions on Internet Technology (TOIT)*, 8(1):4, 2007.

[63] H. Nguyen, T. Nguyen, and J. Freire. Learning to extract form labels. *VLDB Endowment*, 1(1):684–694, 2008.

[64] M. P. Papazoglou. The challenges of service evolution. In *Advanced Information Systems Engineering*, pages 1–15. Springer, 2008.

[65] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information System*, 17(2):223–255, 2008.

[66] R. Perrey and M. Lycett. Service-oriented Architecture. In *Symposium on Applications and the Internet Workshops*, pages 116–119. IEEE, 2003.

[67] R. L. Plackett. The analysis of permutations. *Applied Statistics*, pages 193–202, 1975.

[68] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[69] C. Quoc and V. Le. Learning to rank with nonsmooth cost functions. *Advances in Neural Information Processing Systems*, 19:193–200, 2007.

[70] J. Rao and X. Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2005.

[71] L. Richardson and S. Ruby. *RESTful web service*. O'Reilly Media, 2007.

[72] RoboForm. www.roboform.com. Last accessed on March 19th, 2015.

[73] A. Rodriguez. Restful web services: The basics. *IBM developerWorks*, 2008.

[74] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65:386–408, 1958.

[75] E. Rukzio, C. Noda, AD. Luca, J. Hamard, and F. Coskun. Automatic form filling on mobile devices. *Pervasive and Mobile Computing*, 4(2):161–181, 2008.

[76] Sahi. http://sahi.co.in/. Last Accessed on March 1, 2015.

[77] G. Salton and C. Yang. On the specification of term values in automatic indexing. *Journal of documentation*, 29(4):351–372, 1973.

[78] A. Segev and E. Toch. Context-based matching and ranking of web services for composition. *IEEE Transactions on Services Computing*, 2(3):210–222, 2009.

[79] Web services Glossary. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice. Last Accessed on December 11, 2015.

[80] A. Shashua and A. Levin. Ranking with large margin principle: Two approaches. In *Advances in neural information processing systems*, pages 937–944, 2002.

[81] D. Sprott and L. Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17, 2004.

[82] WSDL Structure. https://en.wikipedia.org/wiki/Web_Services_Description_Language. Last Accessed on January 1, 2016.

[83] J. Stylos, B. A. Myers, and A. Faulring. Citrine: providing intelligent copy-and-paste. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 185–188. ACM, 2004.

[84] S. Tata and J. M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM Sigmod Record*, 36(2):7–12, 2007.

[85] J. P. Thomas, M. Thomas, and G. Ghinea. Modeling of web services flow. In *IEEE International Conference on E-Commerce*, pages 391–398. IEEE, 2003.

[86] 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.

[87] DOM Tree. http://www.w3schools.com/htmldom/dom_nodes.asp. Last Accessed on March 1, 2015.

[88] M. Tsai, T. Liu, T. Qin, H. Chen, and W. Ma. Frank: a ranking method with fidelity loss. In *ACM SIGIR conference on Research and development in information retrieval*, pages 383–390, 2007.

[89] B. Upadhyaya. Composing heterogeneous services from end users' perspective. Canadian PHD thesis, 2014.

[90] 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.

[91] B. Upadhyaya, R. Tan, and Y. Zou. An approach for mining service composition

patterns from execution logs. *Software: Evolution and Process*, 25(8):841–870, August 2013.

[92] B. Upadhyaya, Y. Zou, S. Wang, and J. Ng. Automatically composing services by mining process knowledge from the web. In *11th International Conference on Service Oriented Computing*, pages 267–282. ACM, 2013. Berlin, Dec 2-5.

[93] 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.

[94] L. Vu, M. Hauswirth, and K. Aberer. Qos-based service selection and ranking with trust and reputation management. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 466–483. Springer, 2005.

[95] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of 20th International Conference on Data Engineering*, pages 79–90. IEEE, 2004.

[96] 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.

[97] 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.

[98] S. Wang, Y. Zou, B. Upadhyaya, I. Keivanloo, and J. Ng. An empirical study on categorizing user input parameters for user inputs reuse. In *Web Engineering*, pages 21–39. Springer, 2014.

[99] S. Wang, Y. Zou, B. Upadhyaya, and J. Ng. An intelligent framework for auto-filling web forms from different web applications. In *World Congress on Services*, pages 175–179. IEEE, 2013.

[100] Y. Wang, T. Peng, W. Zuo, and R. Li. Automatic filling forms of deep web entries based on ontology. In *International Conference on Web Information Systems and Mining (WISM)*, pages 376–380. IEEE, 2009.

[101] 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.

[102] WSDL. http://www.w3.org/TR/wsdl. Last accessed on September 12th, 2015.

[103] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.

[104] 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.

[105] H. Xiao. End-user driven service composition for constructing personalized service oriented applications. Canadian PHD thesis, 2011.

[106] H. Xiao, Y. Zou, J. Ng, and L. Nigul. An approach for context-aware service

discovery and recommendation. In *IEEE International Conference on Web Services (ICWS)*, pages 163–170. IEEE, 2010.

[107] H. Xiao, Y. Zou, R. Tang, J. Ng, and L. Nigul. An automatic approach for ontology-driven service composition. In *Int'l Conf. on Service-Oriented Computing and Applications*, pages 1–8. IEEE, 2009.

[108] 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.

[109] XML Schema Definition Language (XSD). https://www.w3.org/TR/xmlschema11-1/. Last Accessed on January 1, 2016.

[110] 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.

[111] Robert K Yin. *Case Study Research: Design and Methods.* Sage publications, 2014.

[112] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A support vector method for optimizing average precision. In *ACM SIGIR conference on Research and development in information retrieval*, pages 271–278, 2007.

[113] Z. Zhang, B. He, and K. C. Chang. Understanding web query interfaces: Best-effort parsing with hidden syntax. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 107–118. ACM, 2004.

[114] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. In *Advances in neural information processing systems*, pages 1697–1704, 2008.