A Smart Agent Framework for Personalized Service Composition

by

Yu Zhao

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2019

Abstract

End-users face a massive amount of services when selecting and composing the desired services to meet their personal preferences. For example, people frequently use the Internet to perform on-line activities (*e.g.*, on-line shopping and banking) and control their IoT devices (*e.g.*, turning off home light). The ever increasing amounts of services available could be overwhelming to end-users. To relieve end-users' cognitive overloading when performing on-line activities, agents can be designed and developed to act autonomously on end-users' behalf.

However, the autonomous and cooperative natures of agents make it complex to design and implement by developers. General speaking, developers face the following challenges to adopt agents for service composition: 1) difficult to program agents; 2) lack of domain knowledge to identify necessary tasks for composite services; and 3) lack of a standard interface to integrate various services (*e.g.*, IoT devices) with web services by agents.

In this thesis, we present a smart agent framework that facilitates the design and implementation of agents for service composition. More specifically, our approach consists of 5 aspects: 1) we propose an easy-to-understand semi-natural language syntax that allows developers to specify the functionalities of agents; 2) we use natural language processing techniques and machine learning algorithms to identify service related tasks from on-line sources, *e.g.*, on-line how-to instructions; 3) we propose an approach to transform functionalities of IoT devices to web services using the standard web service infrastructure; 4) we propose a deep learning based model (*i.e.*, DeepCont) to predict end-users' ratings on services, using user reviews and service descriptions; and 5) we use a ranking algorithm (*i.e.*, RankBoost) to automatically learn user preferences from the service usage history, and integrate the learned user preferences to recommend a collection of services for end-users. A series of case studies demonstrate that our approaches are effective to develop agents and recommend personalized services for end-users. Our approaches relieve the required efforts from developers to design agents for personalized service composition.

Related Publications

Earlier versions of the work in this thesis have been published in the following papers.

- **Mining User Intents to Compose Services for End-Users** (Chapter 4). Yu Zhao, Shaohua Wang, Ying Zou, Joanna Ng, and Tinny Ng. In Proceedings of the 2016 IEEE International Conference on Web Services (ICWS), San Francisco, CA, USA, 2016.

- **An Automatic Approach for Transforming IoT Applications to RESTful Services on the Cloud** (Chapter 5). Yu Zhao, Ying Zou, Joanna Ng and Daniel Alencar da Costa. In Proceedings of the 15th IEEE International Conference on Service-Oriented Computing (ICSOC), Malaga, Spain, November 13-16, 2017.

- **Automatically Learning User Preferences for Personalized Service Composition** (Chapter 7). Yu Zhao, Shaohua Wang, Ying Zou, Joanna Ng and Tinny Ng. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, Hawaii, USA.

For each of the aforementioned papers, I am the first author. My contributions include proposing research ideas and methods, investigating background and related work, conducting experiments, and writing and polishing manuscripts. My co-authors participated discussion meetings of the research to give suggestions on improving the research, and provided advice to help polishing manuscripts.

Acknowledgments

Finally, it comes to the last and the most important part of this thesis. Three years ago, when Dr. Feng Zhang finished his PhD defense, he told me that “soon this day will come to you”. It’s hard to believe that the day comes so fast. I felt so fortunate to have this wonderful five years journey as a Master and PhD student that was full of happiness, excitement, pressure and challenges. I learned from this PhD degree, not only fancy technical skills, but more importantly, the research methods that can solve any problems in a systematic way.

Without many people, I could not accomplish this great milestone. First and foremost, I would express my sincere gratitude to my supervisor and great friend, Dr. Ying Zou. She brought me to Queen’s University, Canada that changed the course of my life. Dr. Ying Zou is an intelligent, knowledgeable, responsible and kind-hearted person who is always available and supportive to me. I remembered that as a fresh graduate student, my writing was awful. She kept revising my writing with great patience and encouraged me towards perfectiveness. I learned a great deal of technical and research skills from her and appreciated so much her advice on my job hunting and other personal decisions.

I would also like to thank my thesis committee: Dr. Yuan Tian, Dr. Tom Dean, Dr. Michael Korenberg and Dr. Yan Liu for their constructive and insightful feedback on my thesis.

I am grateful and lucky to work with an amazing group of talented researchers during my PhD studies, who bring me so much joys: Dr. Feng Zhang, Dr. Shaohua Wang, Dr. Ehsan Noei, Mariam El Mezouar, Hanfeng Chen, Haoran Niu, Yonghui Huang, Pradeep Venkatesh, Yongjian Yang, Dr. Daniel Alencar da Costa, Guoliang Zhao, Taher Ahmed Ghaleb, Aidan Yang, Dr. Safwat Hassan, Osama Ehsan, and Omar El Zarif. Each of our group members have their shining points that I can learn from. Dr. Feng Zhang provided me tremendous support during my early days in Kingston, Canada. Dr. Shaohua Wang's passion in research gave me great motivations. Special thanks goes to Dr. Daniel Alencar da Costa, with whom I shared precious moments doing research, exchanging views in world affairs and personal values, and working out. I also want to thank my best friends for the past five years: Ke Xu, Dayuan Wang, Dr. Heng Li, Dr. Wenjie Li, Wenbo Liu, Pei Jin, Liang Zhuge and Chengguang Yu. I enjoyed a lot staying with you guys.

I would never be able to make this achievement without the understanding, support and love from my parents. In my most difficult times, the first person I want to talk with was my father. He always gave me valuable suggestions from a friend's perspective that would relieve my pressure and cheer me up. My mom's expectation was that I can receive a good education. Now she must be proud of me. I dedicate this thesis to my beloved mom. My deepest love gives to my wife and soulmate, Qiao Lu. I can always be myself in front of you. You touched my heart when you cook for me, sit by my side and accompany me catching deadlines. Thanks for your love, dedication and sacrifice. I cherish every moment with you.

Last but not least, I want to thank Ed Sheeran for writing great songs (such as Photograph), which give me a peaceful mind to do creative research and write this thesis. I am ready to embrace the next challenges!

Statement of Originality

Contents

Abstract **i**

Related Publications **iii**

Acknowledgments **v**

Statement of Originality **viii**

Contents **ix**

List of Tables **xiii**

List of Figures **xv**

Chapter 1: Introduction **1**

1.1 Background 2

1.1.1 Agents 2

1.1.2 Web Services 3

1.1.3 IoT Devices 4

1.1.4 Service Composition Process 5

1.2 Research Challenges 7

1.3 Thesis Statement 9

1.4 Thesis Objectives 9

1.5 Thesis Overview 14

Chapter 2: Related Work **16**

2.1 Agent-Oriented Software Engineering 16

2.2 Applications of the Multi-Agent Framework for Service Composition . 19

2.3 Natural Language Processing 20

2.4 Service Discovery 21

2.5 User Rating Prediction on Services 22

2.6 Service Composition 25

Chapter 3: Composing Services Using a Multi-Agent Framework 27
3.1 Introduction . . . 28
3.2 Agent Programming Using the Jadex Platform . . . 30
3.3 Overview of Our Approach . . . 31
3.3.1 An Overview of Our Semi-Natural Language Syntax . . . 32
3.3.2 Specifying Plans . . . 36
3.3.3 Composing Services . . . 43
3.3.4 Generating Agent Code . . . 45
3.4 Experiments . . . 45
3.4.1 Experiment Setup . . . 45
3.4.2 Research Questions . . . 47
3.5 Discussion . . . 53
3.6 Conclusion . . . 55

Chapter 4: Identifying Service Related Tasks from On-line How-to Instructions 56
4.1 Introduction . . . 57
4.2 Natural Language Processing . . . 59
4.3 Overview of Our Approach . . . 60
4.3.1 Predicting Valid How-to Instructions . . . 61
4.3.2 Identifying Tasks . . . 62
4.3.3 Predicting Service Related Tasks . . . 67
4.3.4 Generating Agent Specifications . . . 70
4.4 Case Study . . . 72
4.4.1 Collecting How-to Instructions . . . 73
4.4.2 Research Questions . . . 73
4.5 Chapter Summary . . . 79

Chapter 5: Automatically Transforming IoT Applications to RESTful Web Services 81
5.1 Introduction . . . 82
5.2 Programming Structure of the Source Code of IoT Applications . . . 84
5.3 Overview of Our Approach . . . 85
5.3.1 Identifying External Methods . . . 86
5.3.2 Extracting Service Specifications . . . 90
5.3.3 Representing External Methods in a Service Schema . . . 92
5.3.4 Transforming External Methods to IoT Services . . . 94
5.4 Case Study . . . 97
5.4.1 Case Study Setup . . . 97
5.4.2 Research Questions . . . 99

5.5 Chapter Summary 102

Chapter 6: Predicting User Ratings Using User Reviews and Service Descriptions 103

6.1 Introduction 104
6.2 Convolution Operations 107
6.3 Variational Autoencoder Network (VAE) 108
6.4 The Proposed Deep Content Model 109
6.4.1 Architecture 110
6.4.2 Training DeepCont 117
6.5 Case Study Setup 118
6.5.1 Datasets 119
6.5.2 Evaluation Settings 122
6.5.3 Baselines 123
6.6 Case Study 124
6.7 Chapter Summary 141

Chapter 7: Automatically Learning User Preferences for Personalized Service Composition 143

7.1 Introduction 144
7.2 Overview of Our Approach 147
7.2.1 Extracting Service and End-User Information 147
7.2.2 Inferring User Preferences and the Priority Orders of the User Preferences 150
7.2.3 Composing Services 160
7.3 Case Study 162
7.3.1 Case Study Setup 162
7.3.2 Case Study Results 165
7.4 Chapter Summary 171

Chapter 8: Conclusions and Future Work 173

8.1 Contribution 174
8.2 Future Work 177
8.2.1 Generating Executable Agent Code for Other Agent Platforms 177
8.2.2 Analyzing Source Code of IoT Applications Written in Other Programming Languages 177
8.2.3 Dynamically Recommending Next Tasks to Perform 178
8.2.4 Learning User Preferences from More Categories of Learning Features 178

Bibliography 179

Appendix A: Research Ethics Approval **202**

List of Tables

3.1 The semi-natural language syntax for specifying beliefs. The blue keywords represent the syntax (the same applies for other tables). 32

3.2 The semi-natural language syntax for specifying goals 33

3.3 The semi-natural language syntax for specifying Jadex services 35

3.4 The semi-natural language syntax for specifying plans 36

3.5 The *ConditionLanguage* syntax for specifying conditions 37

3.6 The results of our user study. *Number*: number of subjects who understand our syntax or prefer our approach (*i.e.*, subjects choose 4 or 5 in questions); *P*: *Number* divided by the total number of subjects. . 52

4.1 Rules to extract intents in various linguistic patterns (VB stands for Verb, NP denotes Noun Phrase and PP represents Prepositional Phrase) 62

4.2 Rules to extract constraints from Noun Phrases, Prepositional Phrases and Clauses 66

4.3 The performance comparison of our model with three baseline models on the testing set. 78

5.1 Service specification that is extracted from the external methods in Figure 5.3 85

5.2 A summary of available operations for an IoT service 94

5.3 The distribution of projects and python methods in each domain. *Avg LOC* denotes the average lines of code for each project. 98

6.1 Statistical description of out datasets 120

6.2 MSE Comparison of our model with four baseline models on 21 datasets (Best results are marked as bold) 131

6.3 MSE Comparison of our model with four variants of our model on 21 datasets 134

7.1 Twenty learning features. Learning features marked with * indicate unique features based on our dataset. 152

7.2 The distribution of services and tasks in each domain. We also show the average number of tasks performed by the subjects in each domain. 165

7.3 Performance of the three approaches with different sizes of training dataset. **Abbreviation for method names**: OA, B1, and B2: our approach, Equal-Weights-MORL, and No-Order-MORL. We show the actual precision of our approach, and demonstrate the percentage of precision our approach improves or decreases the Equal-Weights-MORL and No-Order-MORL. "+" and "-" means improvement and decrease, respectively. Ui means the i^{th} end-user. The number colored with dark gray represents the highest number in the column, while the light gray denotes the lowest (same for other tables). 168

7.4 Performance of our approach with different threshold values (%). "T" in the headers represents threshold. Ui means the i^{th} end-user. 170

7.5 Performance of our approach with different precision metrics (%). Ui means the i^{th} end-user. 171

List of Figures

1.1 An overview of our proposed approaches in the thesis. 10

1.2 An example how-to instruction for buying a laptop 11

3.1 An example of a code snippet for two agents to exchange messages. The red code is the *business logic code*. 30

3.2 An overview of our approach 31

3.3 The *BuyLaptop* agent specification 31

3.4 A pop up window to insert *business logic code* 38

3.5 A pop up window to implement the code for invoking services 42

3.6 The three approaches to compose services using the semi-natural language syntax. 44

4.1 Two examples of POS tagger and Parser analyzed sentences (POS tags: determiner (DT), verb (VB), noun (NN), preposition (IN), to (TO), conjunction (CC), adjective (JJ), comparative adjective (JJR), adverb (RB), number (CD) and personal pronoun (PRP)) 59

4.2 An overview of our approach to identify service related tasks. 60

4.3 The structure of the MLP model 69

4.4 An example how-to instruction and two examples of agent specifications for buying a laptop 71

4.5 Number of hidden layers 78

4.6 Number of neurons 78

4.7 *F-Measure* results on the validation set with respect to different hyper-parameters (black dots represent the optimal *F-Measure* values). . . . 78

5.1 External method 1 84

5.2 External method 2 84

5.3 Examples of external methods 84

5.4 An overview of our approach 86

5.5 The service schema 92

5.6 The URL schema for accessing an operation. 95

5.7 An annotated screenshot of a web form that is used to control a led. The blue text highlights the data that is extracted from our service schema. 95

5.8 An annotated screenshot of the *profile* operation, which obtains the temperature. The blue text highlights the data that is extracted from our service schema. 96

5.9 A screenshot of our tool that shows the available operations of IoT services to end-users 99

6.1 The architecture of our proposed DeepCont model (n and p denote the number of words in a user review and a service description, respectively; c is the dimension of a word vector; and m_1 represents the number of filters) 111

6.2 Dimension of latent features k 126

6.3 Number of filters m_1 126

6.4 Number of layers L 126

6.5 Batch size M 126

6.6 Text cut-off percentage p 126

6.7 Analysis of MSE with respect to different hyper-parameters (Black dots represent the optimal MSE values) 126

6.8 MSE results for predicting new services 137

7.1 An overview of our approach. MORL refers to the the multi-objective reinforcement learning algorithm. 146

7.2 An example of task description and service description. 148

7.3 A screenshot of our tool for selecting services. 164

7.4 Average precision of our approach and the two baseline approaches using 70% of the whole dataset for training. The x-axis denotes subject ids, Ui means the i^{th} end-user. y-axis is the average precision of recommending services. 167

A.1 Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 3. 203

A.2 Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 3. 204

A.3 Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 7. 205

Chapter 1

Introduction

With the pervasiveness of the Internet, people perform various on-line activities daily. For example, end-users frequently visit E-commerce websites (*e.g.*, Amazon.com) to buy various preferred items. The Internet is also widely used to control and access information of various Internet-of-Things (IoT) devices. For example, end-users use smart phones to remotely turn off lights at home. Although the use of the Internet greatly eases our daily lives, end-users are overloaded with the ever increasing information. According to the statistics,[1] Amazon has more than 606 million items for sell in the United States in 2017. To buy a preferred item, an end-user has to manually look through the large amount of items to select preferred items, and compare the price of the selected items in different websites in order to get the best deal. To relieve end-users' cognitive overloading of performing such repetitive and tiresome on-line activities, agents [152] can be designed as a service composition infrastructure to act autonomously on behalf of end-users to perform on-line activities using web

[1]https://www.scrapehero.com/how-many-products-does-amazon-sell-worldwide-october-2017/

services. In the following subsections, the background of agents, web services, IoT devices and service composition process is briefly discussed.

1.1 Background

1.1.1 Agents

Agents are software entities that can make independent decisions on which actions to perform, and proactively perform the selected actions to achieve their designed goals. For example, agents can act on behalf of end-users to frequently check the price of end-user preferred items in different websites and select the best deal. Agents are designed in multi-agent frameworks (*e.g.*, Jadex [101] and Jason [12]) that follow the Belief-Desire-Intention (BDI) architecture model [13]. The BDI architecture endows the abilities of agents to make decisions on the performed actions to achieve goals. An agent that follows the BDI architecture has three essential components, *i.e.*, beliefs, goals and plans. Beliefs are the knowledge that an agent has about the environment and its internal state. For example, the price of an item can be viewed as a belief. Goals are the target states that an agent tries to achieve, *e.g.*, selecting an item with the best deal. A goal may have multiple associated plans that specify the actions to achieve the goal. At run time, an agent can decide the performed plan from the set of associated plans to achieve the goal. An agent can be reactive to context changes, *e.g.*, a change of a belief value, to decide whether triggering a plan or dropping the current performed plan. Moreover, agents have the social ability that they can cooperate with other agents to achieve a common goal.

1.1.2 Web Services

Web services are software modules that are used for machine-to-machine communication over the World Wide Web. This communication can be implemented using two kinds of architecture styles, *i.e.*, Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) [38]:

- *SOAP-based web services.* SOAP provides the message protocol layer for exchanging machine-readable information in the implementation of web services. It uses XML schema as its message format and transmits messages through Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP). SOAP is typically used in combination with Web Services Description Language (WSDL) [25]. WSDL is an XML-based language that describes the functionalities of web services, such as the data structures, parameters and operations that a service provides. A client program can read a WSDL file in a web service to determine available functionalities on the server. The client can then send a SOAP request to the server to call one of the listed operations in the WSDL file. Due to the messages are formatted with XML schema, it demands more computational resources to handle SOAP messages.

- *RESTful web services.* REST is designed to provide a lightweight mechanism to interact between machines. Services conform to the constraints of REST can be called RESTful web services [114]. RESTful services communicate over Hypertext Transfer Protocol (HTTP) with a fixed set of HTTP verbs: GET, PUT, DELETE and POST. Uniform Resource Identifiers (URIs) are used to identify web services. The URIs can be operated upon with standard HTTP verbs (such as GET). The exchanged message is often described by a lightweight data format – JSON. When exchanging request and response messages, each of which contains all the necessary information to describe the exchanging process. Thus, compared with the SOAP architecture, the REST architecture is lightweight and stateless.

A majority of web applications (*e.g.*, Amazon.com) are built on top of web services, mainly because the two main advantages of web services: 1) web services have standard interfaces and structured data formats that allow web services to be easily integrated with existing applications; and 2) web services can be independently developed and deployed.

1.1.3 IoT Devices

IoT devices are gaining popularity among our daily lives. The functionalities provided by IoT devices can be used to specify actions in agents. For example, IoT devices can sense home temperature and react upon an end-user's request to turn off the light. An IoT device is a physical item that is embedded with a computing system and can be controlled remotely through Bluetooth or Wi-Fi. In general, there are three classes of IoT devices: sensors, actuators and composite devices [45]. A sensor can measure the physical properties of a physical environment at a constant frequency. For example, a sensor can sense the temperature. An actuator is an IoT device that receives control requests from end-users to change the physical environment. For example, an actuator can receive a command from an end-user to turn on the light. Finally, a composite device is composed of both sensors and actuators. For example, a thermostat is a composite IoT device. It can sense the temperature and change the temperature upon requests. An IoT device may have multiple functionalities. For instance, an indoor sensor may sense both temperature and humidity. Each functionality of an IoT device can be published as an IoT service. An IoT service exposes a functionality of an IoT device with an interface that confirms to the web service standard. An IoT device may use various networking protocols to exchange data with an IoT service, such as MQTT [57], a lightweight publish-subscribe messaging protocol designed for exchanging real-time IoT data.

1.1.4 Service Composition Process

In existing web applications, a single service (*i.e.*, a web service or an IoT service) is not sufficient to achieve a goal. A set of logically related services are composed together to provide a functionality in an application. Generally speaking, developers use three steps to compose services: *designing abstract processes, discovering services* and *composing services*. We describe the three steps as follows:

- **Designing Abstract Processes.** An *abstract process* contains a number of tasks that need to be performed by services to fulfill a functionality. For example, to design an abstract process for an end-user to buy a laptop, the relevant tasks may include checking laptop specifications, checking user reviews, making payments and shipping.

- **Discovering services.** An identified task in an abstract process may be implemented by a large number of functionally similar services. A service is described by functional description and non-functional attributes. The functional description contains the formal specification of functionalities that a service provides. Non-functional attributes capture the performance and qualitative/quantitative aspects of services, such as price and user rating. The process of *discovering services* aims to automatically identify the service that performs the functionality of a task.

- **Composing services.** The discovered services are linked and composed to provide a functionality that can be used to achieve end-users' goals (*e.g.*, buy laptops). The composite service can be reused by many other web applications to provide reusable functionalities to end-users.

Services are passive components that can only be executed once called upon [3][56]. Because of the passive attribute, the developed composite services in existing web applications have to hard-code the control flow and data flow among the contained services. For example, a composite service designed for shopping may specify that a service used for searching for items is followed by a payment service. Once services contained in the composite services are defined, these services cannot communicate with other services. To avoid hard-coding the relations of services, agents can be designed to compose services. Agents can proactively perform services contained in plans to achieve the assigned goals and autonomously react upon the results of services to determine the next better alternative services. For example, based on the price obtained from the item searching service, an agent can decide whether to use an email notification service to notify end-users the price, or use a payment service to directly place an order. Moreover, agents can cooperate with each other, thus enabling the communication among the contained services. For example, different agents can check the price of items in different websites.

1.2 Research Challenges

Despite the advantages of using agents to compose services, developers face the following challenges to develop agents:

- **Difficult to develop agents to compose services.** Existing studies [14][18] show that the development of agents is complex due to the internal behaviors of agents, *i.e.*, the proactive, autonomous and reactive characteristics. In addition, composing services by cooperating agents can be more complex. Developers may spend a huge amount of efforts to learn the domain knowledge of agents and develop codes to meet various requirements from different groups of end-users. Therefore, it is essential to provide an easy-to-use agent programming approach for developers to develop composite services in agents.

- **Lack of expertise to identify the necessary tasks for service composition.** To design an agent for service composition, developers need to understand a specific domain to identify service related tasks in an abstract process. For example, to design an agent for buying a preferred laptop, it is necessary to implement a task to place an order for the laptop. However, the task to explore discount codes for a group of end-users, such as university students, is also important, since students may have a tight budget. A well-designed agent needs to consider various needs of end-users to achieve the design objectives. Nevertheless, developers typically do not possess sufficient knowledge on the vast diversity of different domains to identify tasks. Consequently, it is important to provide an automatic approach to suggest the tasks for developers.

- **Lack of standard interface to integrate IoT devices with web services.**

IoT devices can be combined with web services to provide rich functionalities for end-users. For instance, an on-line grocery order can be made based on a food consumption alert that is triggered by analyzing the data read from a fridge sensor. In the current practices, to control and access the information of IoT devices, end-users have to install proprietary applications (*e.g.*, mobile applications) on smart phones or computers for each individual IoT device. The diverse end-user applications lack a standard interface to allow the communication among various IoT devices and web services. To allow agents to control IoT devices and combine IoT devices with web services, it is crucial to transform the functionalities of IoT devices to IoT services that confirm to the web service standard. Thus, IoT services and web services are services that can be composed in agents to provide more diverse functionalities.

- **Limitation on recommending end-user preferred services.** A task can be implemented by a vast number of functional similar services. End-users may have distinct preferences when selecting services. For example, end-users with a low budget may prefer to buy a laptop with a low price. While other end-users may care more about the quality when buying a laptop. To select a preferred service, an end-user needs to carefully examine service descriptions, the corresponding user reviews and non-functional attributes, such as user ratings. To relieve the cognitive overloading when selecting services for end-users, a recommendation system offered by agents is essential to predict the preferred services for end-users.

- **Limited ability to recommend personalized composite services.** Usually, end-users need to use multiple services in web applications to achieve their goals. Many web applications allow end-users to enter searching requirements to search for services. For example, an end-user can specify the price range and brand names to search for items on the Amazon website. However, the number of predefined options is limited and can not be customized for the individuals' needs. Moreover, the predefined options are designed for searching preferred services for a single task, rather than searching for preferred composite services. To better satisfy end-users' personal needs, it is crucial to support automated service composition by efficiently recommending personalized composite services.

1.3 Thesis Statement

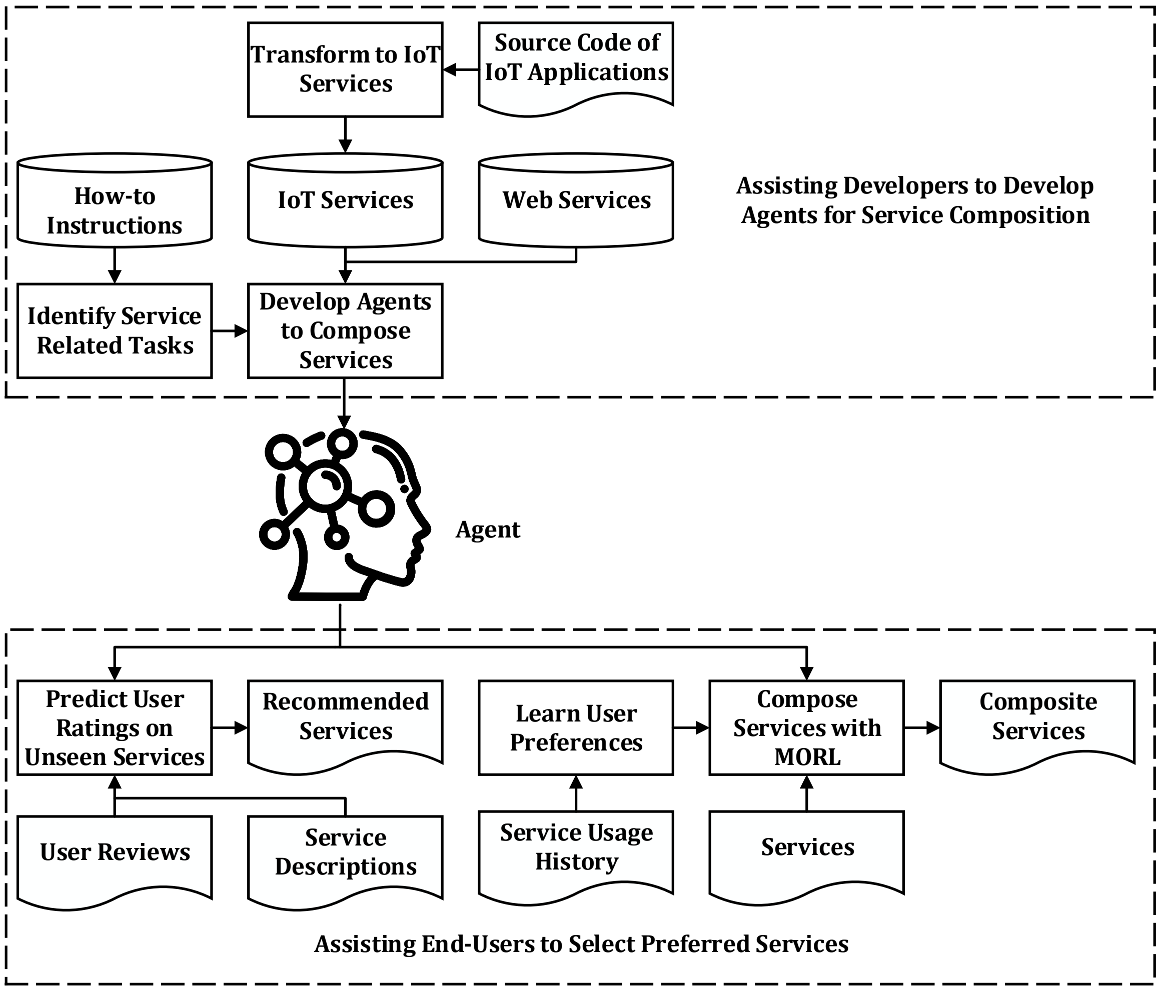
End-users face a massive amount of services to meet their personal preferences when selecting services to perform daily activities. It is essential to provide an easy-to-use agent programming approach for developers to develop agents that can act autonomously on behalf of end-users to perform on-line activities. Agents can be designed to analyze end-users' past activities using machine learning algorithms to perform personalized service recommendation.

1.4 Thesis Objectives

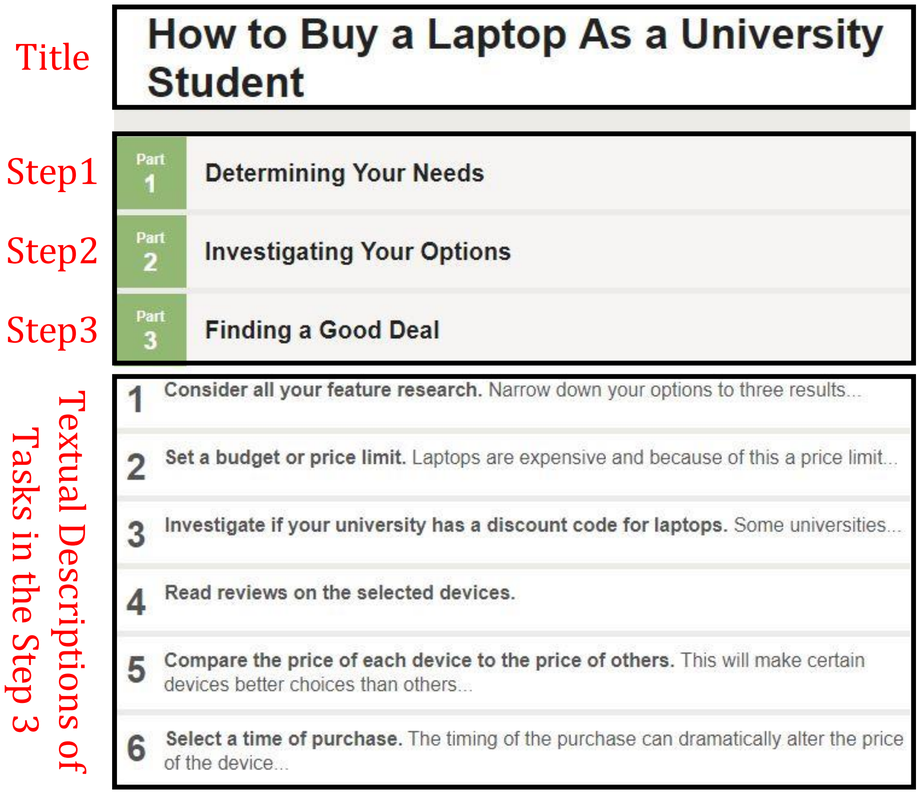
Figure 1.1 presents an overview of our proposed approaches in the thesis. To address the challenges that are listed in Section 1.2, our approach improves the process of service composition from the following aspects:

- **Using a semi-natural language approach to develop agents for service composition.** We use the Jadex platform as our multi-agent framework

Figure 1.1: An overview of our proposed approaches in the thesis.


Transform to IoT Services
Source Code of IoT Applications
How-to Instructions
IoT Services
Web Services
Assisting Developers to Develop Agents for Service Composition
Identify Service Related Tasks
Develop Agents to Compose Services
Agent
Predict User Ratings on Unseen Services
Recommended Services
Learn User Preferences
Compose Services with MORL
Composite Services
User Reviews
Service Descriptions
Service Usage History
Services
Assisting End-Users to Select Preferred Services

for developers to build agents. To alleviate the required efforts from developers to develop agents using the Jadex platform, we design a semi-natural language approach for developers to develop agents. Our semi-natural language approach abstracts the complex agent programming syntax to an abstract easy-to-understand semi-natural language syntax. Developers can use our proposed syntax to specify the functionalities of agents. Our proposed syntax can automatically generate agent specific code, and guide developers to develop customized code to implement business logics. Our semi-natural language approach automatically generates executable agent code from our proposed syntax and the customized code to compose services.



How to Buy a Laptop As a University Student

Part 1 Determining Your Needs

Part 2 Investigating Your Options

Part 3 Finding a Good Deal

1 **Consider all your feature research.** Narrow down your options to three results...

2 **Set a budget or price limit.** Laptops are expensive and because of this a price limit...

3 **Investigate if your university has a discount code for laptops.** Some universities...

4 **Read reviews on the selected devices.**

5 **Compare the price of each device to the price of others.** This will make certain devices better choices than others...

6 **Select a time of purchase.** The timing of the purchase can dramatically alter the price of the device...

Figure 1.2: An example how-to instruction for buying a laptop

- **Identifying service related tasks from on-line how-to instructions.** To enrich developers' knowledge on designing composite services in agents, we leverage on-line how-to instructions to identify service related tasks. On-line how-to instructions are written in a natural language (*i.e.*, in English) by domain experts to teach people to conduct various activities, such as buying laptops with deals. Figure 1.2 shows an example of a how-to instruction for buying a laptop.[2] Typically, a how-to instruction consists of a title and a list of steps. The title describes the intention of the instruction, while each step represents a goal that needs to be performed to fulfill the intention of the instruction. Each step has a name and a textual description of the tasks. These tasks may contain service related tasks that can be performed by services.

To identify the tasks in on-line how-to instructions, we use natural language processing techniques to analyze the grammatical structures of each sentence. Many extracted tasks specified in how-to instructions are manual activities and cannot be performed by services, *e.g.*, "inspect laptop damages".[3] We apply a deep learning based model, called multilayer perceptron (MLP), to filter out tasks irrelevant to services.

- **Automatically transforming IoT applications to RESTful web services.** To transform functionalities of IoT devices to IoT services, our approach analyzes the source code of IoT applications to identify the methods that can be potentially transformed to IoT services. From the identified methods, we extract service specifications that can describe the corresponding IoT services, *e.g.*, service name and HTTP function. Our approach instantiates IoT services on the cloud from the identified methods. To automatically instantiate IoT services, we propose a service schema that describes the service specifications of IoT services.

- **Predicting user ratings using user reviews and service descriptions.**

[2]https://www.wikihow.com/Buy-a-Laptop-As-a-University-Student
[3]https://www.wikihow.com/Buy-Used-Laptops

To relieve end-users' cognitive overloading when selecting services, a recommendation system is essential to predict end-user preferred services. User reviews and service descriptions are important factors for end-users to select services in addition to user ratings. Existing approaches use either user reviews or service descriptions for service recommendation. In this thesis, we propose a deep learning based model, *i.e.*, *Deep Content Model* (DeepCont), to predict user ratings on unseen services and leverage both user reviews and service descriptions. Our proposed DeepCont model learns user latent features and service latent features from user reviews and service descriptions, respectively, using two coupled generative convolutional neural networks, *i.e.*, convolutional variational autoencoder (VAE) network [62][113]. The two networks are jointly trained to simultaneously generate effective latent features and user rating prediction.

- **Automatically learning user preferences for personalized service composition.** To recommend personalized composite services for end-users without much manual specification, we automatically learn user preferences using machine learning approaches. More specifically, our approach extracts various learning features from end-users' service selection history. We apply a learning-to-rank algorithm, *i.e.*, RankBoost, to learn user preferences and the priority orders on the extracted learning features. To enable automated service composition, our approach integrates the learned user preferences to the multi-objective reinforcement learning (MORL) algorithm to recommend personalized composite services.

1.5 Thesis Overview

We present an overview of this thesis in the following.

- Chapter 2 **Related Work.** We give an overview of the related work of the thesis to differentiate our work from the related research.
- Chapter 3 **Composing Services Using a Multi-Agent Framework.** We describe our proposed semi-natural language approach for developers to develop agents for service composition.
- Chapter 4 **Identifying Service Related Tasks from On-line How-to Instructions.** We describe our natural language processing technique to extract tasks from on-line how-to instructions, and present our MLP model to identify service related tasks from the extracted tasks.
- Chapter 5 **Automatically Transforming IoT Applications to RESTful Web Services.** We illustrate our approach that identifies methods in IoT applications and transform them to IoT services. We use a designed service schema to instantiate IoT services from the identified methods.
- Chapter 6 **Predicting User Ratings Using User Reviews and Service Descriptions.** We present a deep learning based model, *i.e.*, DeepCont, to predict end-users' ratings on services.
- Chapter 7 **Automatically Learning User Preferences for Personalized Service Composition.** We present our approach that can learn user preferences from the learning features, and integrate the learned user preferences to recommend a collection of end-user preferred services.

- Chapter 8 **Conclusions and Future Work.** In this Chapter, we conclude the thesis and discuss future directions.

Chapter 2

Related Work

In this section, we summarize the related work on six areas that are relevant to the thesis: 1) agent-oriented software engineering; 2) the applications of the multi-agent framework for service composition; 3) natural language processing techniques; 4) service discovery; 5) user rating prediction on services; and 6) service composition.

2.1 Agent-Oriented Software Engineering

The BDI architecture model is implemented in many agent platforms. We summarize five agent platforms that implement the BDI architecture model, *i.e.*, JAM [54], Jason [12], SPARK [97], JACK [52], and Jadex [101]. We choose these five agent platforms because they have programming languages and toolset support, and are still in adoption [90]. JAM [54] is an early attempt that provides a Domain Specific Language (DSL) to implement beliefs, goals, plans in the BDI architecture. The JAM platform supports meta-level reasoning that decides an applicable plan to realize a given goal. The SPARK [97] platform uses combinations of keywords (*e.g.*, *achieve*) and symbols (*e.g.*, ::) to construct the syntactic and semantics of the language to design BDI agents. A key difference between the SPARK platform with other agent platforms is that SPARK uses the finite state machines (FSMs) to declare actions in a plan. Using FSMs can track the current execution state of a plan. However, the introduced semantics of the language and FSMs make it hard for developers to learn the SPARK agent programming. The Jason [12] platform uses the AgentSpeak, a logic programming language, to describe the architecture of BDI agents. The use of the logic programming language, *i.e.*, AgentSpeak, facilitates the formal verification [11] of the correctness of developed agents in Jason. However, developers are more familiar with general programming languages (*e.g.*, Java), and may get confused on the syntax (*e.g.*, ! and !!) in the logic programming language. JACK [52] extends the Java language that incorporates agent-oriented concepts to design BDI agents. Many applications are designed under the JACK platform, *e.g.*, unmanned air vehicle control [36]. However, the commercial license of JACK impedes the widely usage in the industry and research.

The aforementioned platforms use either existing agent communication protocols, such as Knowledge Query and Manipulation Language (*i.e.*, KQML) [39] and Agent Communication Language (*i.e.*, FIPA-ACL) [8], or use proprietary agent communication protocols (*e.g.*, the JACK platform) to enable the communications between agents. However, the various incompatible protocols impede the communication between agents in different platforms. Moreover, it is cumbersome to design and maintain the interaction between agents and applications, because of the incompatible protocols used in agent platforms [102]. Our approach uses the Jadex platform [14][101] to compose services. Jadex uses an extension of the Java language to program BDI agents that have similar architecture compared to the aforementioned platforms. The major advantage of using the Jadex platform is that agents use Jadex services to communicate with each other. The Jadex services can be published as SOAP-based or RESTful web services. The standard interface of Jadex services makes it easier for Jadex agents to be integrated with existing applications.

The area of agent-oriented software engineering has been actively researched to support the development of agents. Generally speaking, the development of agents can be divided into four phases, *i.e.*, *requirement analysis, architecture design, detailed design* and *implementation* [15]. Specifically, the *requirement analysis* phase [155] identifies the roles of involved agents, the goals of each agent and the basic interactions among agents. The *architecture design* phase defines the global architecture of agents and designs agent coordination models [55][157]. For example, the Gaia methodology [157] views multiple agents as an organization and describes an interaction between two agents as a set of proprietary defined protocols. Unlike the above approaches, our approach to develop agents focuses on the *detailed design* and *implementation* phases.

The *detailed design* phase mainly uses model-driven development approaches to specify complete internal behaviors (*e.g.*, plans) of an agent with high-level abstractions. For example, Bergenti *et al.* [9] propose a DSL with grammers extended from Xtend (a language similar to Java),[1] to model agents in the JADE platform. Tezel *et al.* [136] develop a metamodel that is composed of graphical notations for modeling agents in the Jason platform. SEA_ML [18][19] designs a DSL to model agents in the Jadex platform. Other approaches [46][117] present metamodels or DSLs that are

[1]https://www.eclipse.org/xtend/

independent of agent platforms. Developers may be unfamiliar with the syntax proposed in the aforementioned approaches. Same as the aforementioned approaches, our approach proposes abstract syntax to enable modularized agent development. Differently, our proposed semi-natural language syntax is easy to be understood by developers.

The *implementation* phase concerns with programming agents using specific agent platforms. Many approaches transform the design of agents in the *detailed design* phase to the agent code. For example, the DSLs developed by Bergenti *et al.* [9] and SEA_ML [18][19] can be used to generate agent code that is specific to the JADE and Jadex platform, respectively. Hahn *et al.* [46] apply two model transformations to transform the designed metamodel to agent code that is used in the JACK and JADE platform. Moraitis *et al.* [96] transform Gaia models [157] to agent code for the JADE platform. However, developers cannot implement customized *business logic code* using the proposed metamodels and DSLs. Unlike the aforementioned approaches, our approach of programming agents tightly integrates the *detailed design* phase with the *implementation* phase. Thus, the abstract semi-natural language syntax guides developers to generate executable agent code.

2.2 Applications of the Multi-Agent Framework for Service Composition

Many researchers combine services with agents to achieve goals of agents. Corchado *et al.* [29] and Maamar *et al.* [87] use agents as controllers and coordinators to discover services and communicate with services. Greenwood *et al.* [42] focus on designing message translators to enable agents to communicate with remote services. The above approaches mainly work with services that are described with WSDL.[2] Other approaches facilitate the dynamic cooperation between agents with semantic web services [92]. For example, Klusch *et al.* [64] design algorithms for agents to discover semantic web services. SEA_ML[18][19] pre-defines several actions in agents to discover, negotiate and execute semantic web services. The aforementioned approaches support agents to dynamically interact with services at run time. However, the effectiveness of the dynamic interaction is not thoroughly evaluated. In contrast, our approach supports developers to statically compose services using agents at design time.

2.3 Natural Language Processing

Natural language processing is studied and applied in various aspects, such as information extraction (IE), question-answer (QA) systems and semantic parsing. An IE system utilizes natural language processing techniques to extract predefined types of information from unstructured text [6][35][82][122]. QA systems analyze end-user questions written in natural languages and search for direct answers from knowledge bases [60][68]. Thomason *et al.* [137] and Chen *et al.* [22] utilize semantic parsing techniques to translate natural languages to a formal representation form that is understood by machines.

Many existing natural language processing techniques build probabilistic models to tag part-of-speech of each word in a sentence. For example, Toutanova *et al.* [139] present a part-of-speech tagger that uses tags of preceding and following words to tag current words. Huang *et al.* [53] build deep learning based models, *i.e.*, bidirectional

[2]https://www.w3.org/TR/2001/NOTE-wsdl-20010315

Long Short-Term Memory (LSTM) models, to perform part-of-speech tagging. A substantial research efforts focus on parsing grammatical structures of sentences, *i.e.*, grouping words into phrases. Collins [28] uses the lexical dependency information to parse sentences. However, the lexical dependency information could be sparse, as many lexical dependencies (*e.g.*, *stocks skyrocketed*) occur infrequently. Instead of using lexical dependency information, Klein and Manning [63] exploit structure context to analyze grammatical structures of sentences. Socher *et al.* [126] build a recursive neural network to perform grammatical parsing. In our work, we use existing natural language processing techniques to identify tasks from on-line how-to instructions in the context of service composition. Our approach to identify tasks is similar to Upadhyaya *et al.* [142] who mine process knowledge to compose services. Unlike their approach that only identifies verbs and nouns, our work identifies tasks by analyzing grammatical structures of sentences that extract various phrases (*e.g.*, prepositional phrase and subordinate clause).

2.4 Service Discovery

Service discovery aims to rank services based on the similarity degree between the task query and service descriptions. Two categories of approaches are used to discover services, *i.e.*, supervised and unsupervised. In the supervised approaches, a training dataset needs to be constructed manually to label the relevance between services and queries from end-users [59][130]. The manual labeling process is time-consuming and tedious. The unsupervised approaches use keyword matching [47][140][153] and clustering techniques [34][78][150] to discover services. Upadhyaya *et al.* [140] use

WordNet [94] and ConceptNet [84] to identify representative concepts in a service description. An end-user's query is used to match the identified representative concepts from a service to discover services. Hao *et al.* [47] propose an approach to augment service descriptions with mashup descriptions for service discovery. Xiong *et al.* [153] match the inferred semantic relations in the service descriptions and user queries to rank services. Elgazzar *et al.* [34] present an approach to cluster WSDL documents into functionally similar groups to bootstrap the discovery of SOAP-based services. Wang *et al.* [150] and Lin *et al.* [78] use unsupervised machine learning techniques (*e.g.*, Support Vector Machine) to cluster services. Given an end-user's query, the approach firstly ranks service clusters and then ranks services from the highest ranked service clusters. Other approaches expand end-users' queries with user preferences for service discovery. For example, Balke and Wagner [5] allow end-users to specify soft constraints, *e.g.*, low price, in the query to sort functional similar services. Yang *et al.* [154] perform service discovery by expanding an end-user's queries with the end-user's contextual information, *e.g.*, social relations and performed activities.

2.5 User Rating Prediction on Services

A task can be implemented by a vast number of functional similar services. To reduce end-users' workload to select preferred services, many researchers propose to use machine learning techniques (*e.g.*, deep learning) to predict user ratings on services. The predicted user ratings represent end-users' preferences and can be used to recommend services. We summarize the related research to predict user ratings in three categories: 1) collaborative filtering approaches; 2) approaches using user reviews; and 3) approaches using service descriptions.

1) Collaborative filtering approaches. In recent years, the collaborative filtering (CF) approaches [129] used in recommendation systems have gained popularity due to the achieved good performance. Many of the CF approaches are based on the matrix factorization (MF) [65][95]. Matrix factorization learns user and service latent features from the past user ratings. The inner product of the user and service latent features generates the predicted user ratings. Many variants of MF are introduced to enhance the model. For example, factorization machine [111] generalizes the MF model that maps the interactions of latent features to a low dimensional space. Instead of the inner product function used in MF, Neural Collaborative Filtering (NCF) [51] learns the interaction function between latent features. The aforementioned approaches only use the user rating information for prediction. The performance degrades significantly when the user rating matrix is increasingly sparse. Differently from the aforementioned approaches, our model learns latent features from user reviews and service descriptions to address the sparsity problem.

2) Approaches using user reviews. Many researchers resort to the user review as additional input to apply collaborative filtering. The Hidden Factors and Hidden Topics (HFT) model and the Ratings Meet Reviews (RMR) model learn latent topics related to services based on user reviews by applying topic modelling techniques, *e.g.*, Latent Dirichlet Allocation (LDA) [10]. The learned latent topics are associated with service latent features. Explicit Factor Model (EFM) [159] and Aspect-Aware Latent Factor Model (ALFM) [23] analyze user reviews to extract different aspects of services and user preferences about the aspects of services. In particular, EFM analyzes sentences in user reviews and uses a rule-based engine to extract service aspects and end-users' opinions. In ALMF, user preferences and service aspects are learned by a multinomial distribution over the latent topics of user reviews. Moreover, many researchers apply deep learning based approaches to predict user ratings using user reviews [17][165]. DeepCoNN [165] uses two convolutional neural networks (CNN) to jointly model user and service latent features from user reviews. The user and service latent features are connected in a shared latent space to predict user ratings. The DeepCoNN model is extended in the TransNet [17] and the NARRE [21] models. Specifically, TransNet adds an additional latent layer to represent the user review for the target user-service pair. NARRE assigns weights to the input user reviews to model the usefulness of each user review. The other approach, *i.e.*, TARMF [86], combines the attention based recurrent neural networks (RNN) with matrix factorization to predict user ratings. Although using user reviews for recommendation achieves promising results, the previously mentioned approaches cannot learn latent features from newly added services, since such services have not received any user ratings or user reviews yet. However, our model can infer service latent features from new services because we learn the features from service descriptions.

3) Approaches using service descriptions. Many other approaches for recommendation use service information as additional information for collaborative filtering. The Collaborative Topic Modelling (CTR) model [146] combines topic modelling with collaborative filtering to recommend scientific articles. In particular, CTR uses the topic modelling technique to discover the topics from service descriptions. The service latent features are associated with the topic distributions. Instead of learning latent features from topics, the Collaborative Deep Learning (CDL) model [147] learns service latent features using a Bayesian formulation of stacked denoising autoencoders (SDAE). Different from CDL, the Collaborative Variational Autoencoder (CVAE) model [75] proposes a Bayesian generative model to learn probabilistic distributions of service latent features. However, the aforementioned approaches view a service description as bag-of-words, and therefore cannot capture the order and semantic meanings of words. Thus, two similar service descriptions that are described with synonymous words could have different service latent features. Moreover, Dong *et al.* [32] use an extended stacked denoising autoencoder (SDAE) to learn latent features from side information, *e.g.*, release data of services. None of the aforementioned approaches use service descriptions and user reviews jointly. Our model leverages both types of information to learn user latent features from user reviews and learn service latent features from service descriptions.

2.6 Service Composition

Various algorithms are proposed in the literature to recommend composite services for end-users to meet end-users' functional and non-functional requirements.

Shiaa *et al.* [124] and Tong *et al.* [138] use the graph theory for service composition. Services are the nodes in the graph. The association between two services is established based on the semantic similarity between the input and output parameters of the two services. Chattopadhyay and Banerjee [20] add probability values to services in the graph theory approach to capture dynamic availabilities of services. However, the aforementioned approaches fail to consider end-users' preferences in service composition. Sohrabi and McIlraith [127], and Lin *et al.* [79] allow end-users to specify preferences in the form of PDDL3, the Planning Domain Definition Language. Ardagna and Pernici [2], and Zeng *et al.* [158] use linear programming to compose services by optimizing a linear objective function. The objective function is a weighted sum of non-functional attributes (*e.g.*, price) that are extracted from services. End-users' preferences are represented as weight values in the objective function. Canfora *et al.* [16] propose to use genetic algorithms (GAs) to satisfy end-users' constraints in service composition. Instead of optimizing a linear objective function in the linear programming approach, GAs aim to maximize some non-functional attributes (*e.g.*, rating), while minimizing the others (such as price).

The service composition process can be modeled as a Markov Decision Process (MDP) [98][110][148][149] and be solved using the multi-objective reinforcement learning (MORL) algorithm [132]. MORL conducts a trial-and-error search to learn optimal or near-optimal composite services by maximizing the expected sum of rewards, which are received when selecting services. The reward is calculated by a reward function. Wang *et al.* [148][149] use a linear weighted sum function to aggregate multiple objectives (*e.g.*, non-functional attributes) into a single one. However, the linear weighted sum method can only generate a limited number of solutions (*i.e.*, composite services) [98]. Moustafa and Zhang [98] utilize a multiple-policy MORL approach to derive a complete set of optimal composite services. Such approach may consume considerable responsive time and memory space to find large size of optimal solutions [100]. Ren *et al.* [110] consider the uncertainty of service behaviors in the MORL algorithm and guarantee the successful execution of the selected optimal composite services. Yu and Bouguettaya [156] leverage indices on service operations and an aggregate function to compute optimal composite services. In the above mentioned approaches, end-users are required to specify preferences manually. Unlike the above approaches, we automatically identify user preferences and the priority orders of the preferences to compose services for end-users.

Chapter 3

Composing Services Using a Multi-Agent Framework

Developers need to invest substantial efforts to design and implement agents due to the autonomous and cooperative natures of agents. In this chapter, we propose an easy-to-understand semi-natural language syntax to relieve the effort from developers to develop agents for service composition. Our proposed syntax can automatically generate executable agent code .

Chapter Organization. The rest of the chapter is organized as follows. Section 3.1 introduces this chapter. Section 3.2 describes the agent programming in the Jadex platform. Section 3.3 gives an overview of our proposed semi-natural language approach to develop agents. Section 3.4 evaluates our approach through an empirical study and a user study. Section 3.5 discusses the limitation and generality of our approach. Finally, Section 3.6 concludes this chapter.

3.1 Introduction

Services are employed as a significant part of software systems in various domains, such as e-commerce or Internet Banking. A set of logically related services are composed together to fulfill a functionality of a software system [162]. However, services are passive because they can be only executed once called upon. Because of the passive attribute, the developed composite services have to hard-code the control flow and data flow of the contained services. For example, a composite service designed for shopping may specify that a service used for searching for items is followed by a payment service. Thus, services cannot react upon context changes (e.g., the changes from outputs of services) to autonomously decide the next better alternative services.

To avoid hard-coding the execution logics of services, agents can be designed. Agents are software entities that can proactively perform services to achieve the assigned goals and autonomously react upon the results of services to determine the next better alternative services [152]. Generally speaking, the usage of agents has four main advantages for service composition: 1) pro-activeness, which allows an agent to take the initiative to perform services to satisfy their designed goals; 2) autonomy, *i.e.*, a failure of a service may incur an agent to choose alternative services; 3) reactiveness, which endows the ability of an agent to constantly monitor context changes and trigger services correspondingly; and 4) social ability, *i.e.*, agents are capable of cooperating with other agents to perform composite services.

Agents are mainly developed on agent platforms (*e.g.*, Jadex [101]) using general purpose programming languages (*e.g.*, Java). However, existing studies show that the design and implementation of agents is complex due to the proactive, autonomous, reactive and social characteristics of agents [9][18]. Developers need to invest a substantial effort to learn the agent programming due to the complexity of agent programming [41]. Moreover, Java does not offer the needed high-level abstractions to describe agents for modularized development [9]. Thus, there is an immediate need to provide developers with friendly abstract syntax for developing agents. To help the developing of agents, existing research mainly applies model-driven approaches to develop agents [120]. For example, approaches proposed by Bergenti *et al.* [9] and Hahn *et al.* [46] design metamodels to describe components of agents and their relationships. However, these metamodels are not designed for service composition. SEA_ML [18][19] is a domain specific language (DSL) that is used to design agents to dynamically interact with semantic services[92]. However, developers cannot implement customized code for their business logic using the DSL, thereby limiting the applicability of agents.

To address the aforementioned challenges, we propose an approach that uses a multi-agent framework to facilitate service composition. To alleviate the complexity of agent programming, we abstract the agent programming syntax to a semi-natural language syntax. Our semi-natural language syntax can be automatically transformed to the agent specific code. Our approach converts the previously identified service related tasks to an agent specification that is specified using our proposed syntax. Furthermore, developers can focus on developing customized code to implement the business logic apart from the agent specific code. Our approach generates executable agent code from the agent specification and the customized code to compose services.

We build a prototype tool as a proof of concept for our approach. We evaluate our approach through an empirical study and a user study. We design five agent

```
final String[] brandArray = new String[1];
SServiceProvider.getServices(agent, IPlaceOrderService.class, RequiredServiceInfo.SCOPE_PLATFORM)
.addResultListener(new IntermediateDefaultResultListener<IPlaceOrderService>(){
    public void intermediateResultAvailable(IPlaceOrderService is){
        is.getBrand(model).addResultListener(new IResultListener<String>(){
            public void resultAvailable(String brand){
                brandArray[0] = brand;}
            public void exceptionOccurred(Exception exception){
                exception.printStackTrace();}});
    }
});
String returned_brand = brandArray[0];
```

Figure 3.1: An example of a code snippet for two agents to exchange messages. The red code is the *business logic code*.

specifications using our approach for two practical service composition scenarios. Our results show that our approach can correctly generate executable agent code from these agent specifications. The results of our user study reveal that our semi-natural language syntax is easier to understand and adopt when compared with the current agent programming approach that uses general purpose programming languages (*e.g.*, Java).

3.2 Agent Programming Using the Jadex Platform

Agents are developed using Java in the Jadex platform. We classify the code of agent programming into two types, *i.e.*, *boilerplate code* and *business logic code*. The *boilerplate code* is a standard code template that implements beliefs, goals, plans and Jadex services, while the *business logic code* is the code customized by developers for achieving business goals. Figure 3.1 shows an example of agent code for sending messages between two agents. As illustrated in Figure 3.1, the *boilerplate code* in grey accounts for a large portion of the agent programming. In the example, the agent sends a laptop model to the *getBrand* Jadex service provided by the *PlaceOrder* agent, and receives the brand of a laptop.

3.3 Overview of Our Approach

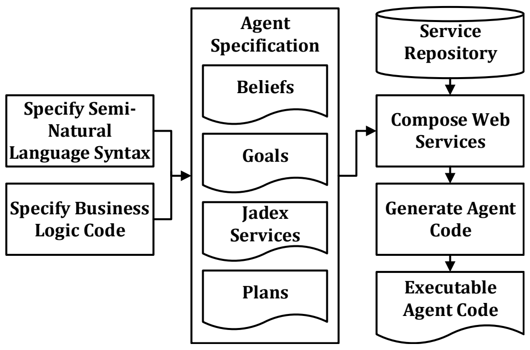


Figure 3.2: An overview of our approach

```
Parameter:
long update
double pricelimit
double price
Agent: BuyLaptop
Monitor $update
Every 2 hours interval, change $update
Plan: FindDealPlan
If $update changes, perform the plan
PlanBody: <"Get price limit and model from consumers">
Call service $getBrand located in $PlaceOrder
Invoke web service $getLaptopPrice
PlanBody: <"Extract price from the output of the service">
If $price is less than $pricelimit, perform the goal $PlaceOrderGoal located in $PlaceOrder
```

Figure 3.3: The *BuyLaptop* agent specification

Figure 3.2 presents an overview of our approach. A developer specifies the semi-natural language syntax and the *business logic code* to form an agent specification. An example of an agent specification is shown in Figure 3.3, which checks the price of a laptop in every 2 hours interval, and places an order if the price is lower than a price limit. An agent specification may contain four components, including beliefs, goals, Jadex services and plans. The components can be used to compose services in agents. Our approach generates executable agent code from the agent specification. The code generated from the agent specification listed in Figure 3.3 is shown in the appendix.[1] We describe each step of our approach in the following subsections.

[1]https://drive.google.com/open?id=1dEVZl9haeKgP1cK6z1GOrte9rrS4jkS7

Table 3.1: The semi-natural language syntax for specifying beliefs. The blue keywords represent the syntax (the same applies for other tables).

	Semi-Natural Syntax	**Example**	**Agent Code**
Agent Name	Agent: agent_name	Agent: BuyLaptop	@Agent public class BuyLaptopBDI{}
Parameter	Parameter: type name	long update	
Belief	Monitor $Beliefname	Monitor $update	protected long $update @Belief *getter* and *setter* method
Belief with Update Rate	Every NUMBER (seconds| minutes |hours) interval, change $Beliefname	Every 2 hours interval, change $update	@Belief (updaterate=7,200,000) protected long update= System.currentTimeMillis();

3.3.1 An Overview of Our Semi-Natural Language Syntax

We use Xtext,[2] a widely used framework for building domain-specific languages (DSLs), to build our semi-natural language syntax. Xtext contains Integrated Development Environment (IDE) features, *e.g.*, syntax coloring, code completion and code folding, to help developers specify agent specifications using our semi-natural language syntax. Our proposed syntax contains four essential components for describing an agent, including beliefs, goals, Jadex services and plans. Our proposed syntax generates executable agents that can run on the Jadex platform. Figure 3.3 shows an example of an agent specification for buying a laptop. The agent specification checks the price of a laptop in every 2 hours interval, and purchases the laptop (*i.e.*, places an order) if the price is lower than a price limit. We describe the approach to specify each of the four components below.

Specifying Beliefs. In the agent programming paradigm, a belief is used as a condition to trigger a plan, thus invoking the contained web services. We define the semi-natural language syntax to specify beliefs. Table 3.1 shows our syntax to

[2]https://www.eclipse.org/Xtext/

Table 3.2: The semi-natural language syntax for specifying goals

	Semi-Natural Syntax	**Example**	**Agent Code**
Goal Name	Goal: goal_name	Goal: PlaceOrderGoal	@Goal public class PlaceOrderGoal{}
Goal Input	GoalInput: $input_name	GoalInput: $model	@GoalParameter protected String model;
Goal Output	GoalOutput: $output_name	GoalOutput: $success	@GoalResult protected boolean success
Associated Plan	GoalPlan: $plan_name	GoalPlan: $PlaceOrderPlan	@Plan(trigger=@Trigger (goals=PlaceOrderGoal.class)) protected boolean PlaceOrderPlan (String model){}

specify beliefs. The lines 1-7 in Figure 3.3 give examples of our defined syntax for beliefs. More specifically, the *Agent Name* syntax (*i.e.*, *Agent: agent_name*) declares the name of an agent. The syntax generates an agent Java class that implements the components of an agent, *e.g.*, beliefs and plans. To specify parameters of an agent, we define the *Parameter* syntax that declares a parameter with a type and a name (*e.g.*, *long update*). A specified parameter can be referenced throughout the agent specification using the '$' symbol. A parameter can be set as a belief. We define the *Belief* syntax, *i.e.*, *Monitor $Beliefname*, to specify a parameter as a belief. For example, *Monitor $update* sets the parameter *update* as a belief. In the agent programming, a belief value can be changed to the current timestamp at certain time intervals (*e.g.*, 2 hours). To reduce the developers' effort of programming the belief values, we define the *Belief with Update Rate* syntax, which can generate the agent code to change belief values periodically. For example, the syntax *Every 2 hours interval, change $update* in line 7 of Figure 3.3 changes the belief *update* every 2 hours. The syntax can be used to trigger plans repetitively, *e.g.*, to check price repetitively.

Specifying Goals. Goals are realized by executing the associated plans, thus invoking the related services. Table 3.2 shows our semi-natural language syntax for specifying goals. We use the example of the *PlaceOrderGoal* in the *PlaceOrder* agent specification (see Figure 3.3 line 14) to illustrate the syntax. Readers can refer to the appendix[1] for the details of the *PlaceOrder* agent specification. We define the following four syntaxes to specify a goal: (1) the *Goal Name* syntax (*i.e.*, *Goal: goal_name*) is used to declare a goal (*e.g.*, *Goal: PlaceOrderGoal*). The *Goal Name* syntax generates a Java class to represent the goal; (2) The *Goal Input* syntax (*i.e.*, *GoalInput: $input_name*) declares input parameters of a goal; (3) The *Goal Output* syntax (*i.e.*, *GoalOutput: $output_name*) declares output parameters of a goal. The *Goal Input* syntax and the *Goal Output* syntax generate input parameters and output parameters in the Java class, respectively. Our approach adds the necessary import statements and the constructor once a Java class that represents the goal is created; and (4) The *Associated Plan* syntax (*i.e.*, *GoalPlan: $plan_name*) declares an associated plan for a goal. In our generated agent code, these plans take the input parameters of the goal as plan inputs and output parameters of the goal as plan outputs. A generated agent shows the proactive characteristic when actively performing an associated plan to achieve a goal. A developer can declare multiple associated plans for a goal in an agent specification. Thus, the autonomous characteristic of the generated agent can be reflected at run time, in which a failure of a web service could incur the generated agent to perform alternative plans.

Specifying Jadex Services. Jadex services receive messages from agents. Such messages can be served as inputs to the services in a plan. In the agent programming, a Jadex service is defined as a Java method that is declared in a Java interface

Table 3.3: The semi-natural language syntax for specifying Jadex services

	Semi-Natural Syntax	Example	Agent Code
Service Name	Service: service_name	Service: getBrand	public IFuture<String> getBrand (String model)
Service Input	ServiceInput: $input_name	ServiceInput: $model	
Service Output	ServiceOutput: $output_name	ServiceOutput: $brand	
Service Body	ServiceBody: <STRING>	ServiceBody: <"return brand">	{*customized code for "return brand"* return new Future<String>(brand);}
Service Changes Belief	The service changes $Beliefname	The service changes $modelbelief	{setModelbelief(model); return new Future<String>("");}

and implemented in the agent class of an agent. To alleviate the effort of developers to manually implement Jadex services, we design a syntax to specify Jadex services. Table 3.3 uses the example Jadex service *getBrand* in Figure 3.3, line 11 to show the syntax to generate Java methods of Jadex services. More specifically, the *Service Name* syntax (*i.e.*, *Service: service_name*) declares a Jadex service (*e.g.*, *Service: getBrand*). The *Service Input* syntax (*i.e.*, *ServiceInput: $input_name*) and the *Service Output* syntax (*i.e.*, *ServiceOutput: $output_name*) define input parameters and output parameters of a Jadex service, respectively. The above- mentioned three syntaxes initialize a Jadex method in a Java interface. To implement the initialized Jadex method in the agent class, we define the *Service Body* syntax, *i.e.*, *ServiceBody: <STRING>*. The syntax uses a *<STRING>* (discussed in Section *Specifying Plans*) to allow developers to specify a customized service implementation. Since Jadex services can be used to compose services in agents, we define the *Service Changes Belief* syntax, *i.e.*, *The service changes $Beliefname*. The syntax generates a routine code that sets a belief (*i.e.*, *$Beliefname*) with the input parameter of the Jadex service. The change of the belief value may trigger plans to perform services.

Table 3.4: The semi-natural language syntax for specifying plans

		Semi-Natural Syntax	Example	Agent Code
	Plan Name	Plan: plan_name	Plan: FindDealPlan	protected void FindDealPlan(){}
Plan Condition	**Plan Precondition**	(If|When) ConditionLanguage, (perform|apply) (the|its) plan	If $update changes, perform the plan	@Plan(trigger=@Trigger (factchangeds="update"))
	Plan Context Condition	(If|When) ConditionLanguage, (stop performing|aborting) (the|its) plan	If $context becomes false, aborting the plan	@PlanContextCondition (beliefs="context") public boolean checkCondition(){ return context!=false;}
Plan Body	**Plan Body Language**	PlanBody: <STRING>	<"Extract price from the output of the service">	*customized code for plan body*
	Perform Goals (Same)	((If|When) ConditionLanguage,)? (Perform|perform) the goal $goal_name	If $price is less than $pricelimit, perform the goal $PlaceOrderGoal	if (price <pricelimit){ final boolean success = (boolean)bdiFeature. dispatchTopLevelGoal (new PlaceOrderGoal (model)).get();}
	Perform Goals (Diff)	((If|When) ConditionLanguage,)? (Perform|perform) the goal $goal_name (located|available) in $agent_name	If $price is less than $pricelimit, perform the goal $PlaceOrderGoal located in $PlaceOrder	
	Send Messages	(Invoke|Call) service $service_name (located|available) in $agent_name	Call service $getBrand located in $PlaceOrder	See Figure 3.1
	Invoke Web Services	(Invoke|Call) web service $task_name	Invoke web service $PlaceLaptopOrder	*customized code to invoke web services*

3.3.2 Specifying Plans

A plan may activate the contained services and cooperate with plans encapsulated in other agents to compose services. In the agent programming, a plan can be represented as a Java method or an inner class. A plan may contain three components, *i.e.*, a plan name, plan conditions and a plan body. The plan body contains actions of a plan (*e.g.*, invoking services).

To allow developers to specify plans, we define three categories of syntax: *Plan Name*, *Plan Condition* and *Plan Body*. Table 3.4 demonstrates our syntax to specify plans using the example *FindDealPlan* plan (see Figure 3.3 lines 8-14). In Table 3.4, the third column defines our semi-natural language syntax, the forth column gives examples of our syntax and the fifth column shows the generated agent code from the examples. The *Plan Name* syntax determines the name of a plan. In the following,

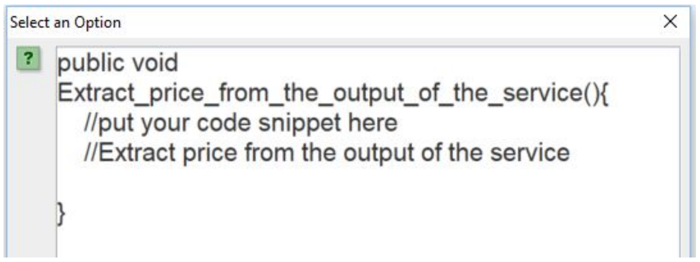
Table 3.5: The *ConditionLanguage* syntax for specifying conditions

	Semi-Natural Syntax	**Example**
Change	$Beliefname changes	$price changes
Less than	$Beliefname (is)? (less\|lower\|fewer\|below\|under\|<) (than)? (NUMBER\|$para_name)	$price is less than 4
Greater than	$Beliefname (is)? (greater\|higher\|more\|over\|larger\|>) (than)? (NUMBER\|$para_name)	$price is greater than 4
Target	$Beliefname (equals|reaches|drops|increases) (to)? (STRING|NUMBER)	$price equals to 4
Boolean	$Beliefname becomes BooleanLiteral	$success becomes true
Contain	$Beliefname contains (STRING\|$para_name)	$model contains “laptop”

we discuss 1) the *Plan Condition* syntax and 2) the *Plan Body* syntax.

1) The **Plan Condition** specifies under which conditions a plan should be executed (*i.e.*, the *Plan Pre-condition* syntax) or dropped (*i.e.*, the *Plan Context Condition* syntax). The *Plan Condition* syntax endows the reactive characteristic of generated agents, in which agents can perceive context changes (*i.e.*, belief changes) to decide to trigger or drop plans.

The conditions are specified using the *ConditionLanguage* syntax. The *Condition-Language* syntax is shown in Table 3.5. We allow developers to specify 6 categories of conditions, including *Change, Less than, Greater than, Target, Boolean* and *Contain.* In Table 3.5, *NUMBER* denotes an integer number and *STRING* represents a string literal. The *BooleanLiteral* has two values, *i.e.*, true or false. The *ConditionLanguage* syntax is contained in the *Plan Pre-condition* syntax and the *Plan Context Condition* syntax (see Table 3.4). Specifically, our *Plan Pre-condition* syntax is implemented as a plan header. The header declares that a change of a belief value triggers actions in the plan body. The plan body generates an *if-statement* based on the specified



```
public void
Extract_price_from_the_output_of_the_service(){
   //put your code snippet here
   //Extract price from the output of the service

}
```

Figure 3.4: A pop up window to insert *business logic code*

condition, to determine if the plan needs to be executed. The *Plan Context Condition* syntax generates a Java method in a plan inner class to implement the syntax. During the execution of a plan, once a belief meets the specified condition, the plan will be aborted. To follow the agent programming paradigm, we initialize an inner class as a plan if the plan contains a *Plan Context Condition.* Otherwise, we initialize a Java method as a plan.

2) The **Plan Body** contains *boilerplate code* and *business logic code* to specify actions. The *business logic code* includes the code to invoke services. We describe our syntax to specify the *Plan Body* below.

The *Plan Body Language* syntax allows developers to insert *business logic code* which they can develop separately in Java. A <*STRING*> in the syntax generates a pop up window with a Java method that is named with *STRING*, as shown in Figure 3.4. Developers can insert the *business logic code* into the generated Java method. The method is then saved to a file named with the method name.

The *Perform Goals* provides a syntax to contain sub-goals and execute the associated plans of the sub-goals. Thus, plans are connected and the contained services are composed. For example, the sub-goal *PlaceOrderGoal* is performed in the *FindDealPlan* plan (*i.e.*, line 14 in Figure 3.3) to place an order. There are two syntaxes for performing sub-goals: (1) the *Perform Goals (Same)* denotes that the sub-goal and the plan that contains the sub-goal belong to the same agent; and (2) the *Perform Goals (Diff)* indicates that the sub-goal and the plan belong to two different agents. Using the *Perform Goals (Diff)* syntax, the generated agents cooperate with each other to achieve a goal, which reflects the social characteristic of agents.

The *Send Messages* syntax generates the agent code to allow an agent to call the provided Jadex services of another agent to send messages. Figure 3.1 shows a generated code example from our syntax: *Call service $getBrand located in $PlaceOrder*, for sending messages. To generate the code, developers use the syntax to specify an agent name, *i.e.*, *agent_name* (*e.g.*, *PlaceOrder*) and the needed Jadex service name, *i.e.*, *service_name* (*e.g.*, *getBrand* shown in line 5 of Figure 3.1). The Java interface (*e.g.*, *IPlaceOrderService.class* shown in lines 2-4 of Figure 3.1) that contains the Jadex service is identified using the agent name. The *Send Messages* syntax pops up a window to allow developers to specify the message to be sent to a Jadex service. The message may be specified as a variable (*e.g.*, *model* shown in line 5) or as a statement (*e.g.*, *getText()*). Using the specified Jadex service, we retrieve the type and name of the output parameter (*e.g.*, *String brand* shown in line 6). The returned message from the Jadex service is assigned to an array that can be used in the plan body (*e.g.*, *brandArray* shown in lines 1, 7 and 12 of Figure 3.1). The other *boilerplate code* in Figure 3.1 (*i.e.*, the grey code) is automatically generated from our proposed syntax.

The *Invoke Web Services* provides a syntax to assist developers in identifying services and generating the code to invoke services. We use RESTful services described in the OpenAPI specification, a human and machine readable service description language.[3] We choose OpenAPI, since it is a widely used and structured specification that has tool set[4] support to design, document, test and deploy services. To identify services, we use the task t_i that is specified in the *Invoke Web Services* syntax (*i.e.*, *task_name* in Table 3.4). We use the Vector Space Model [89], an unsupervised ranking model to rank the most similar services for the task t_i. The Vector Space Model measures the textual similarity between services and a task. We use the following four steps to build the Vector Space Model.

Step 1. Extracting Service Description: For each service, s_i, described in OpenAPI, we extract its service description. An OpenAPI specification contains three pieces of information: the title of a service; the summary of a service; and the descriptions of multiple operations within a service. We concatenate such three pieces of information to form a service description.

Step2. Preprocessing Words: An extracted service description is represented as a bag of words. We preprocess the words in a bag of words using an approach similar to the one used by Zhao *et al.* [160]. Firstly, we split words containing the "-" and "_" characters (*e.g.*, "find_city" is split into "find" and "city"). Secondly, we split words containing capital cases, *e.g.*, "findCity" is split into "find" and "City". We lowercase each word and remove stop words. Finally, we perform word stemming (*e.g.*, "reduced", "reducing" and "reduces" are normalized to "reduce") [162].

Step 3. Building TF-IDF Vectors: We calculate the Term Frequency and Inverse Document Frequency (TF-IDF) values to weigh each word in the bag of words of a service. The Term Frequency (TF) is the count of a word appearing in a service

[3]https://www.openapis.org/
[4]https://swagger.io/

description. The Inverse Document Frequency aims to increase the weight of a word that appears less frequently in other service descriptions. Thus, the task that contains the larger weighted word may give the service a higher rank. Given a word w in a service description, s_i, the TF-IDF value is computed using Equation 3.1.

$$\begin{aligned} TF\text{-}IDF(w, s_i) &= TF(w, s_i) \times IDF(w) \\ TF(w, s_i) &= \textit{the frequency of } w \textit{ in } s_i \\ IDF(w, s_i) &= log(\frac{1+N}{1+df(w)}) + 1 \end{aligned} \tag{3.1}$$

where N denotes the number of services in a service repository and $df(w)$ represents the number of services that has the word, w.

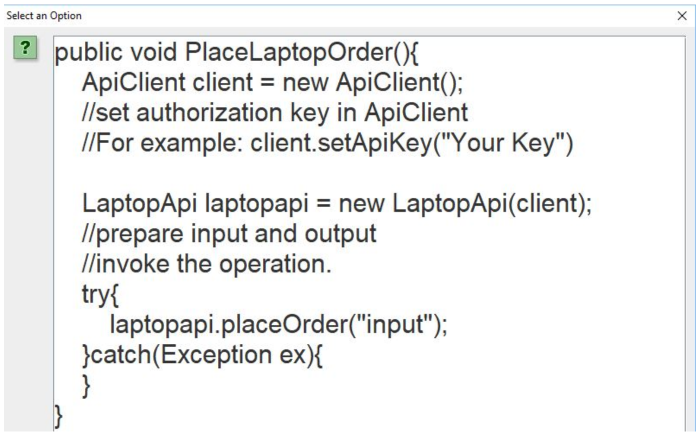
A service, s_i, is represented as a TF-IDF vector. Each value in the vector denotes a TF-IDF value of a word in the bag of words of the service description.

Step 4. Ranking Services: We compute the cosine similarity [89] between the task t_i and each service s_i in the service repository. Equation 3.2 computes the cosine similarity.

$$Similarity(t_i, s_j) = \frac{TF\text{-}IDF(t_i) \times TF\text{-}IDF(s_j)}{|TF\text{-}IDF(t_i)| \times |TF\text{-}IDF(s_j)|} \tag{3.2}$$

where $TF\text{-}IDF(t_i)$ and $TF\text{-}IDF(s_j)$ denote the TF-IDF vectors for the task t_i and the service s_j, respectively.

We use the Vector Space Model to rank the five most similar services for a task t_i. Many existing approaches are proposed to recommend services for tasks, such as topic modeling [78], deep learning [130] and quality of service [134] based approaches. Designing novel approaches to recommend services is not the focus of this thesis. We use the Vector Space Model to rank services, because the model has been proven



```
public void PlaceLaptopOrder(){
   ApiClient client = new ApiClient();
   //set authorization key in ApiClient
   //For example: client.setApiKey("Your Key")

   LaptopApi laptopapi = new LaptopApi(client);
   //prepare input and output
   //invoke the operation.
   try{
      laptopapi.placeOrder("input");
   }catch(Exception ex){
   }
}
```

Figure 3.5: A pop up window to implement the code for invoking services

effective for information retrieval [108][160], and can be simply implemented. Our results show that our Vector Space Model achieves an accuracy of 81% to identify services from the ranked five services for a task.

From our ranked services, a developer then selects a service that can perform the task. Our approach pops a window up for the developer to choose the operation of the service. The pop up window allows developers to specify: 1) the URL of the selected service; 2) an operation name; and 3) an HTTP method. The URL of a service locates the service specification. An operation name and an HTTP method together identify the operation. From the service specification of the identified operation, our approach uses the tool *Swagger Codegen*[5] to generate the client libraries for developers to consume the operation. We build the client libraries into a JAR file using *Apache Maven*,[6] a tool to build Java projects, to allow developers to use the client libraries. In a JAR file, the project and package are named with the HTTP method concatenated with the operation name (*e.g.*, postplaceOrder), to avoid conflict with other client

[5]https://swagger.io/swagger-codegen/
[6]https://maven.apache.org/

libraries. Once the libraries are built, our approach pops up a window for developers to specify customized code to invoke the operation using the generated libraries, as shown in Figure 3.5. We prefill the invocation method (*e.g.*, *laptopapi.placeOrder*) that can be identified from the service specification, and the object that sets an authorization key (*e.g.*, *client*). Developers need to specify input messages in order to invoke the operation and the code to process output messages. The service invocation code is then saved to a file named as *task_name* that is specified in the *Invoke Web Services* syntax.

3.3.3 Composing Services

In an agent platform, a single agent can compose services using plans. Moreover, multiple agents can cooperate with each other to compose services to achieve a common goal. Developers can use our proposed semi-natural language syntax to specify agents for service composition with the following three methods, as illustrated in Figure 3.6.

1) Plans (shown in Figure 3.6a). Several services can be composed in a plan by specifying multiple *Invoke Web Services* syntaxes. Output messages from a service may be served as input messages to invoke another service in a plan. Therefore, the plan would have two composed services.

2) Plans and Beliefs (shown in Figure 3.6b). The output of a service in a plan may be sent to another agent using the *Send Messages* syntax. The Jadex service in the receiver agent may use the *Services Changes Belief* syntax to change a belief value. The belief may be served as a plan pre-condition to execute a plan, thereby invoking the contained services.

3) Plans and Goals (shown in Figure 3.6c). In a plan, a sub-goal can be performed

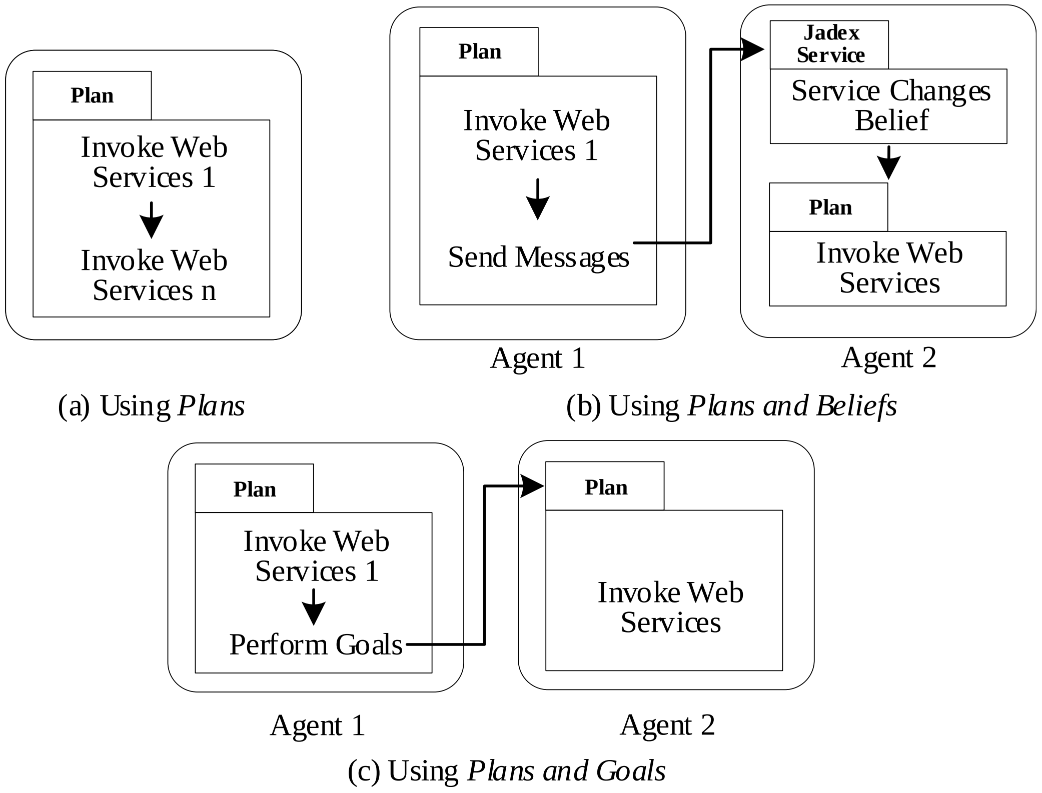


Figure 3.6: The three approaches to compose services using the semi-natural language syntax.

using the *Perform Goals* syntax. The output from a service in the plan can be used as a condition or used as an input to perform a sub-goal. The sub-goal may be performed by executing services contained in the associated plans of the sub-goal.

The above-mentioned three methods describe basic interactions among agents to compose services. In complex application scenarios, more agents need to be involved and complex interactions among agents can be built on top of the three methods. In such scenarios, we can use other researcher proposed agent coordination models [4][157] to design interactions among agents for service composition. For example, mediated agents or artifacts that embed coordination functionalities [55] can be designed to facilitate the interactions among the developed agents. Many approaches [24][157] design protocols for structured interactions, among which the commitment-based protocols [4][24] are widely accepted. We can use commitment-based protocols to design agents, so that agents commit to each other to perform a course of actions to compose services.

3.3.4 Generating Agent Code

Our approach traverses the syntax described in the agent specification to generate executable agent code. A syntax (*e.g.*, the *Goal Name* syntax) can automatically generate the *boilerplate code*. The *business logic code* can be specified either in a plan body or a service body (*e.g.*, using the *Invoke Web Services* syntax) and is saved to a Java method. To extract the *business logic code* from the Java method, we analyze the Abstract Syntax Tree (AST) [164] of a Java method, by parsing and extracting the code in the Java method. A plan body or a service body is formed by concatenating the extracted *business logic code* and the generated code from the specified syntax. Moreover, the necessary class headers and import statements are automatically generated based on the declared components in the agent specification.

3.4 Experiments

In this section, we present the setup of our experiments to evaluate our approach and the obtained results.

3.4.1 Experiment Setup

Collecting Services. To identify service related tasks and compose services using our multi-agent framework, we collect services that are described in the OpenAPI specification. The services are collected from two sources, *i.e.*, SwaggerHub[7] and APIs.guru[8] on December 4th, 2018.

SwaggerHub is a platform to design and index services. Our approach collects *published* services in the platform. An *unpublished* service cannot be consumed, while a *published* service is ready to be consumed. In total, we collect 3,140 services from *SwaggerHub*. We take the following steps to filter the collected services. Firstly, specifications with an empty URL (*i.e.*, the URL equals to "/" or null) or a test URL (*i.e.*, the URL contains "test" or "localhost") are removed. Empty and test URLs indicate that the specification is used for testing purposes. Secondly, duplicate specifications are removed. We identify duplicates by checking whether the service names are the same. Thirdly, we remove specifications without operations. A specification with an empty "paths" object has no operations.

APIs.guru is an open-source project that collects public available services. The project performs the following three steps to process services. Firstly, services that are described in different specifications are converted to the OpenAPI specification. Secondly, services are manually examined to remove private and non-reliable services. Thirdly, errors in the OpenAPI specification are manually corrected. In total, we collect 1,012 services from *APIs.guru*.

Prototype Tool. We implement a prototype tool of our approach for developers to develop composite services using the proposed semi-natural language syntax. Our tool is a plugin that can be used in popular Integrated Development Environments (*i.e.*, IDEs), such as Eclipse.[9]

[7]https://app.swaggerhub.com/home
[8]https://apis.guru/browse-apis/
[9]https://www.eclipse.org/

User Study. We setup a user study to compare our approach with the Java programming approach, *i.e.*, the agent programming approach in Jadex. In our user study, subjects follow our provided tutorial[10] to develop agents for service composition using the two approaches, *i.e.*, our semi-natural language approach and the Java programming approach. To allow subjects to develop agents using the two approaches, we use the Eclipse Rich Client Platform[11] to build a standalone Eclipse application[10] that includes our prototype tool and Java Development Kit (JDK). The subjects are not aware of which approach is ours in the user study. After developing agents using the two approaches, we ask our subjects to fill a survey[12] to evaluate the two approaches. The subjects can use Mac or Windows Operating System to perform our user study. Our user study takes approximately one hour. We recruit subjects by sending emails. We draw two $50 Amazon gift cards for our subjects as a compensation for taking part in our user study. Our user study is approved from the General Research Ethics Board (GREB) for ethical compliance. In total, we recruit 22 subjects, *i.e.*, 11 graduate students and 11 professional software developers. The subjects do not have background on agent programming, but have at least two years of Java experience. Our experiments answer the following two research questions.

RQ3.1. Can our approach correctly generate agent code?

RQ3.2. Are developers satisfied with our approach to develop agents?

3.4.2 Research Questions

RQ3.1. Can our approach correctly generate agent code?

[10]https://drive.google.com/open?id=1tp8mTTzqqfqgIDaqm5J4XJ6a_sIBudCs
[11]https://wiki.eclipse.org/Rich_Client_Platform
[12]https://forms.gle/WmcUbjTxD26G6ysW9

Motivation. Our approach generates executable agent code using the proposed syntax and the *business logic code* provided by developers. Correctly generating agent code proves the applicability of our approach. In this research question, we evaluate the correctness of the generated agent code.

Approach. We apply our approach to run two real service composition scenarios, *i.e.*, *Going Shopping* and *Purchasing a Pet*. For each scenario, we use services collected from Section 3.4.1 and use the *Invoke Web Services* syntax to generate the code for invoking services. To test the correctness of our approach to generate agent code, we design five agent specifications for the two scenarios that exercise the three approaches for service composition (see Section 3.3.3). The five agent specifications use all the semi-natural language syntax defined in Section 3.3.1. The generated agents show the proactive, autonomous, reactive and social characteristics. We describe the five agent specifications as follows.

The first two agent specifications, *i.e.*, *Case 1* and *Case 2*, implement the *Going Shopping* scenario. In the scenario, a boolean parameter *isRaining* is set as a belief. A *CheckWeather* goal has a plan that uses a weather forecasting service to check if a specified city is raining and set the *isRaining* belief. If *isRaining* is false, a *CheckOpeningHour* plan is then triggered. The *CheckOpeningHour* plan uses a service to check the opening hours of a local shop. The scenario uses plans and beliefs (shown in Figure 3.6b) to compose services. For the generated agent code of each agent specification, we test two times that input two cities with different *isRaining* values (*i.e.*, true and false) to perform the *CheckWeather* goal, respectively.

- *Case 1* (using *Plans and Beliefs* in one agent). The *CheckWeather* goal and the *isRaining* belief are implemented in the same agent.

- *Case 2* (using *Plans and Beliefs* in two agents). Different from *Case 1*, the *CheckWeather* goal and the *isRaining* belief are implemented in two different agents. The *CheckWeather* plan passes the *isRaining* value to a Jadex service that belongs to another agent. The agent receives the value and sets the *isRaining* belief.

The other three agent specifications implement the *Purchasing a Pet* scenario. In this scenario, a *CheckAvaiByID* service checks the availability of a pet. If the pet is available (*i.e.*, the pet is not sold), we place an order for the pet using the *PlacePetOrder* service. The input to the composite service is a pet ID, and the output is the order status. We identify an available pet and take the pet ID as input to the composite service.

- *Case 3* (using *Plans* in one agent). The agent composes the two services in a single plan by specifying two *Invoke Web Services* syntaxes in the plan body.

- *Case 4* (using *Plans and Goals* in one agent). We design and implement three goals in the same agent to compose services. The *CheckAvaiByID* and *PlacePetOrder* services are in two separate plans of two goals, to check availability and place an order, respectively. A *PurchasePet* goal has a plan that uses the *Perform Goals (Same)* syntax to coordinate these two goals to purchase a pet.

- *Case 5* (using *Plans and Goals* in three agents). The agent specification contains three goals that are same as *Case 4*. However, the three goals are implemented in three different agents. The *PurchasePet* goal has a plan that uses the *Perform Goals (Diff)* syntax to coordinate the other two goals.

Results. Our approach correctly generates executable agent code from the five agent specifications. We deploy the generated agent code in the Jadex platform. We consider that the code is correctly generated if the agents generate the expected outputs from composite services. Results show that the deployed agent code correctly generates outputs. For *Case 1* and *Case 2* in the *Going Shopping* scenario, a false *isRaining* value outputs the opening hours, while a true *isRaining* value terminates the plan. The three agents in the *Purchasing a Pet* scenario returns the order status. Moreover, we calculate the average lines of code written in our approach and the generated agent Java code for the 5 agent specifications. We do not count the comment lines and empty lines. Results show that the lines of code written in Java (*i.e.*, 181 lines) are 3.3 times more than the lines of code written in our approach (*i.e.*, 55 lines). Our approach requires less coding, since our approach abstracts the *boilerplate code* to a concise semi-natural language syntax. The *boilerplate code* accounts for 78% of the generated agent Java code, indicating that our syntax can generate a large portion of agent code.

RQ3.2. Are developers satisfied with our approach to develop agents?

Motivation. To evaluate whether experienced Java developers with no agent programming knowledge can easily adopt our proposed semi-natural language approach to design and develop agents, we conduct a user study to investigate the perceived usefulness of our approach according to our subjects.

Approach. The subjects follow our tutorial[10] to develop the *Going Shopping* scenario with *Case 1* (described in **RQ3.1**) using the Java programming approach and our semi-natural language approach. In the tutorial, we follow the description of the Java programming approach as specified in the Jadex documentation[13] to describe the steps to perform the Java programming approach. In our user study, the code that is implemented in the Java programming approach and the code that is generated from our semi-natural language approach are same and executable. The subjects are able to see the outputs, *i.e.*, weather and opening hours, when running the code of the two approaches. Our survey[12] contains seven single choice questions, as shown in Table 3.6. A question has a Likert-scale from 1 to 5 to be chosen. A scale 5 represents subjects understand our syntax (questions 1-2) or prefer our semi-natural language approach (questions 3-7). Moreover, we provide open-ended questions for subjects to explain their choices.

Results. Developers prefer to use our approach to build agents. Table 3.6 shows that more than 82% of subjects understand our syntax and believe our syntax is easy-to-understand. More than 86% of subjects believe that our semi-natural language approach is easier to understand, learn and develop agents, compared with the Java programming approach. We conclude that our syntax is understandable. Our approach saves the required efforts from developers to develop agents and learn to develop agents. 59% of subjects consider our approach is easier to inspect, verify and maintain the code, compared with the Java programming approach. The 59% subjects prefer our approach because 1) the agent specification described with our proposed syntax provides high-level abstractions and logics of agents; 2) they can focus on debugging the customized code rather than the generated agent specific code; and 3) the code written in our approach is shorter. The other subjects favor the inspection, verification and maintenance using the Java programming approach, because 1) the

[13]https://download.actoron.com/docs/releases/latest/jadex-mkdocs/

Table 3.6: The results of our user study. *Number*: number of subjects who understand our syntax or prefer our approach (*i.e.*, subjects choose 4 or 5 in questions); *P*: *Number* divided by the total number of subjects.

#	**Questions**	**Number (#/22)**	**P(%)**
1	To what degree do you understand the syntax in the semi-natural language approach for specifying the functionalities of agents?	18	82%
2	To what degree do you think the syntax provided in the semi-natural language approach is easy-to-understand?	20	91%
3	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to understand?	19	86%
4	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to learn?	20	91%
5	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to develop agents?	19	86%
6	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to inspect, verify and maintain the code?	13	59%
7	Do you prefer to develop agents using the Java programming approach or using the semi-natural language approach	15	68%

subjects are familiar with Java; 2) the pop-up window in our approach cannot be used to inspect Java compilation errors; and 3) subjects may feel a hassle to locate and inspect various files that contain the customized code. To improve the maintainability of our approach, in the future, we can provide user-friendly interface for developers to easily navigate between our syntax and the corresponding customized code. Moreover, we can automatically generate unit test for the customized code to ease the efforts of testing [99]. Overall, 68% of subjects prefer to use our approach when compared with the Java programming approach. The other 23% of subjects equally like the two approaches. Our semi-natural language approach separates the agent design and implementation. Developers can focus on the implementation rather than learning the agent programming domain knowledge. The automatic code generation from the designed agent specifications requires less coding efforts.

3.5 Discussion

In this section, we discuss the limitation and generality of our proposed approach.

Semi-Natural Language Syntax. Our proposed semi-natural language syntax provides high-level abstractions to specify beliefs, goals, plans and Jadex services that are essential to describe agents for service composition. Except that our syntax for specifying Jadex services is specific to the Jadex platform, the other syntaxes are general to agent platforms, *e.g.*, Jason and JACK. Thus, our designed syntax can be transformed to other agent programming languages, *e.g.*, AgentSpeak in Jason.

Generated Agent Code. The generated agent code from our proposed syntax follows the standard principle of the agent programming as described in the Jadex documentation[13]. It indicates that our generated code is not more complex compared to the code written with Jadex. Our approach concatenates the customized code implemented by developers and our generated code to form a plan body or a service body, as described in Section 3.3.4. To make the plan body or service body executable, developers need to make sure that the customized code is functional correct and satisfies the desired business logics. Nevertheless, developers can always modify the customized code.

Types of Services. Our proposed *Invoke Web Services* syntax (described in Section 3.3.2) eases the effort from developers to identify and consume services that are described with the OpenAPI specification. For services described with other types of specifications, *e.g.*, WSDL and Hydra,[14] developers can use the *Plan Body Language* syntax (described in Section 3.3.2) to implement customized code to consume the services. Our approach does not increase the complexity for consuming the services, since the customized code is also required in the Java programming approach. In future, we can provide semi-natural language syntax that facilitates developers to consume other types of services.

User Study. In our user study, subjects develop agents for the *Going Shopping* scenario (described in **RQ3.2**). Implementing more scenarios may test full capabilities of our approach. However, it requires substantial efforts from subjects to perform the user study, which may discourage the participation of subjects. We choose the *Going Shopping* scenario, since subjects compare our approach with the Java programming approach by composing services using agents. We recruit 22 subjects in our user study. Although our results show that the majority of subjects prefer our approach to develop agents, involving more subjects is desirable to make our conclusions stronger.

[14]http://www.hydra-cg.com/spec/latest/core/#introduction

3.6 Conclusion

To relieve developers from the effort on developing agents, we present an approach that uses a semi-natural language to generate executable agent code to perform service composition. More specifically, our approach abstracts the *boilerplate code* to a semi-natural language syntax and automatically generates code from the syntax. Moreover, developers can implement *business logic code* without knowing the agent programming paradigm using our approach. Our approach merges the *boilerplate code* and the *business logic code* to generate executable agent code. The user study shows that subjects without agent programming knowledge understand the semi-natural language syntax, and prefer to use our approach. The case study also demonstrates that our approach can correctly generate agent code.

Chapter 4

Identifying Service Related Tasks from On-line How-to Instructions

To design agents, developers need to possess knowledge in various domains to identify the necessary tasks for service composition. In this chapter, we leverage on-line domain knowledge, such as on-line how-to instructions, to identify the necessary tasks for developers. Our approach automatically transforms the identified tasks to an agent specification that is specified using our proposed semi-natural language syntax (described in Chapter 3).

Chapter Organization. In Section 4.1, we introduce this chapter. In Section 4.2, we describe the natural language processing tools used in our approach. In Section 4.3, we give an overview of our approach to identify service related tasks. In Section 4.4, we introduce our empirical studies. Finally, Section 4.5 concludes this chapter.

4.1 Introduction

Services are used in various domains of our daily lives, such as e-commerce or Internet Banking. Generally speaking, a single service is not sufficient to fulfill a functionality of an application. To design an agent (described in Chapter 3), developers have to decompose a functionality of an agent into a set of relevant tasks at a lower granularity to discover the relevant services that can carry out the corresponding tasks, and compose a set of logically related services. However, to identify the relevant tasks for service composition, developers need to possess knowledge in various domains. For example, to design an agent that helps end-users to buy a laptop, it is necessary to implement a task to place an order for the laptop. However, the task to explore discount codes for a group of end-users, such as university students, is also important, since students may have a tight budget. Developers that are unfamiliar with the education domain may easily ignore the task. Nevertheless, developers typically do not possess sufficient knowledge on the vast diversity of different domains to identify tasks [161]. Consequently, it is important to provide an automatic approach to suggest the necessary tasks for service composition that can enrich developers' knowledge to design agents.

To relieve developers from the effort of designing composite services in agents, we leverage the existing knowledge available on-line and written by domain experts, such as on-line how-to instructions to identify tasks that can be performed by services (*i.e.*, service related tasks). How-to instructions describe tasks on how to conduct daily activities, *e.g.*, booking restaurant reservations or composing a song. A large number of how-to instructions, *e.g.*, "How to Cook Dumplings", are invalid how-to instructions that are not relevant to service composition. In the thesis, a how-to instruction that contains at least one service related task is considered a valid how-to instruction for service composition, whereas a how-to instruction that contains no service related tasks is considered as an invalid how-to instruction. An invalid how-to instruction is not relevant to service composition and needs to be filtered out. However, manually filtering out the invalid how-to instructions to identify service related tasks can require extensive efforts. Thus, we build a simple logistic regression model to predict whether a how-to instruction is valid or not.

From the collected valid how-to instructions, we use natural language processing techniques to analyze grammatical structures of each sentence in a how-to instruction to extract a set of tasks. A task consists of an intent that expresses the main goal of the sentence, and constraints are linked with the intent to represent the non-functional requirements [158]. For example, the sentence "plan a Disney World trip to Orlando" has an intent "plan Disney World trip" and a constraint "location: Orlando". Thus, the sentence has a task that is concatenated by the intent and the constraint: "plan Disney World trip Orlando". Although we have filtered out invalid how-to instructions, a valid how-to instruction may specify many tasks that are manual activities and unrelated to services, such as "inspect laptop damages".[1] Such unrelated tasks are not suitable for service composition since they cannot be performed by services. To accurately identify service related tasks from how-to instructions, we build a multilayer perceptron (MLP) model [48], a deep learning based model, to predict whether a task described in a how-to instruction is related to services or not. Our approach converts the identified service related tasks to our proposed semi-natural language syntax to specify agents (see Section 3.3.1 in Chapter 3).

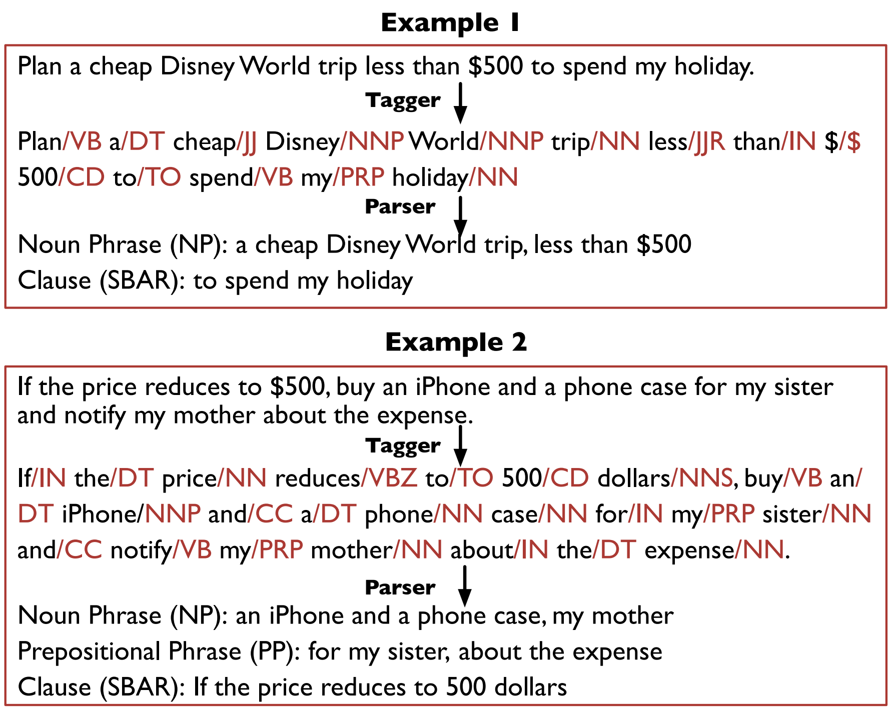
We evaluate the effectiveness of our approach through two case studies. The results

[1]https://www.wikihow.com/Buy-Used-Laptops

of our case studies show that our approach can identify valid how-to instructions with an Area Under Curve (AUC) of 0.85. Moreover, the performance of our approach for identifying service related tasks can achieve an AUC of 0.88.

4.2 Natural Language Processing

In this thesis, we analyze textual descriptions of tasks, such as on-line How-to instruction web pages to extract tasks using natural language processing (NLP) techniques. Here, we discuss the NLP tools used by our framework.


Example 1

Plan a cheap Disney World trip less than $500 to spend my holiday.

Tagger

Plan/VB a/DT cheap/JJ Disney/NNP World/NNP trip/NN less/JJR than/IN $/$ 500/CD to/TO spend/VB my/PRP holiday/NN

Parser

Noun Phrase (NP): a cheap Disney World trip, less than $500
Clause (SBAR): to spend my holiday

Example 2

If the price reduces to $500, buy an iPhone and a phone case for my sister and notify my mother about the expense.

Tagger

If/IN the/DT price/NN reduces/VBZ to/TO 500/CD dollars/NNS, buy/VB an/DT iPhone/NNP and/CC a/DT phone/NN case/NN for/IN my/PRP sister/NN and/CC notify/VB my/PRP mother/NN about/IN the/DT expense/NN.

Parser

Noun Phrase (NP): an iPhone and a phone case, my mother
Prepositional Phrase (PP): for my sister, about the expense
Clause (SBAR): If the price reduces to 500 dollars

Figure 4.1: Two examples of POS tagger and Parser analyzed sentences (POS tags: determiner (DT), verb (VB), noun (NN), preposition (IN), to (TO), conjunction (CC), adjective (JJ), comparative adjective (JJR), adverb (RB), number (CD) and personal pronoun (PRP))

We analyze the grammatical structure of a sentence using the state-of-the-art

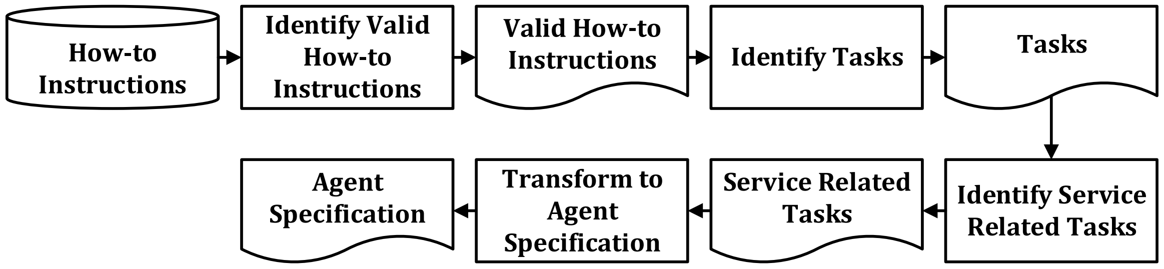

How-to Instructions
Identify Valid How-to Instructions
Valid How-to Instructions
Identify Tasks
Tasks
Identify Service Related Tasks
Service Related Tasks
Transform to Agent Specification
Agent Specification

Figure 4.2: An overview of our approach to identify service related tasks.

Stanford Part-of-Speech (POS) Tagger [139] and Stanford Natural Language Parser [63]. Figure 4.1 shows two examples of using the POS tagger and the Parser for analyzing sentences. POS tagger tags each word in a sentence with part-of-speech markers. For example, in Figure 4.1, "plan" is tagged as a verb and "cheap" is tagged as an adjective. The Parser groups words into phrases with brackets. For example, "a cheap Disney World trip" is grouped as a noun phrase (*i.e.*, NP) in the Example 1 of Figure 4.1. Another noun phrase "less than $500" expresses a requirement about the goal. The Example 2 shows a more complex sentence that contains three tasks: "buy iPhone", "buy phone case" and "notify mother expense".

4.3 Overview of Our Approach

Figure 4.2 shows an overview of our proposed approach to identify service related tasks. Our approach consists of four main steps: 1) automatically identify valid how-to instructions; 2) automatically extract tasks from each sentence in a valid how-to instruction; 3) automatically identify service related tasks from the extracted tasks; and 4) automatically transform the identified service related tasks to agent specifications that are described with our proposed semi-natural language syntax. In the following, we describe each step of our approach in more detail.

4.3.1 Predicting Valid How-to Instructions

We collect how-to instructions from the *wikiHow*[2] website (*i.e.*, a directory of how-to instructions). To predict whether a how-to instruction is related to service composition (*i.e.*, valid how-to instruction) or not (*i.e.*, invalid how-to instruction), we build a simple logistic regression model.

The logistic regression model has training and testing phases. In the training phase, a how-to instruction ht_i in the training set has a learning feature vector F_i and a label y_i. To build a learning feature vector F_i, we use the *doc2vec* [72] model, an unsupervised model, to learn a fixed-length numerical representation of a how-to instruction ht_i. We use the learned numerical represention of a how-to instruction ht_i as the learning feature vector F_i. The numerical representation is trained to infer words in a how-to instruction. Thus, the learned numerical representation captures the semantic meaning of a how-to instruction. A label y_i is either 0 or 1, denoting whether ht_i is a valid how-to instruction or not. To create a training set, we manually label the how-to instructions as valid or invalid (described in Section 4.4.2 **RQ4.1**). A valid how-to instruction that contains at least one service related task is labeled as 1. Otherwise, an invalid instruction is labeled as 0. Our approach of labeling service related tasks is described in Section 4.3.3. In the testing phase, the logistic regression model takes the learning feature vector F_i of a how-to instruction ht_i in the testing set as the input and predicts a label y_i to ht_i.

[2]https://www.wikihow.com/

Table 4.1: Rules to extract intents in various linguistic patterns (VB stands for Verb, NP denotes Noun Phrase and PP represents Prepositional Phrase)

#	Linguistic Pattern	Rules to Extract Intent	Example
1	VB+NP	Action: verb (VB); Object: nouns in a noun phrase (NP)	In "buy an iPhone", the intent is "buy iPhone".
2	VB+PP	Action: verb (VB); Object: nouns in a prepositional phrase (PP)	In "invest in the company", the intent is "invest company".
3	VB+NP+PP	Action: verb (VB); Object: nouns in a noun phrase (NP) and nouns in a prepositional phrase (PP) that has a preposition "of" or "about"	In "notify my mother about the expense", the intent is "notify mother expense".

4.3.2 Identifying Tasks

From the collected valid how-to instructions, we identify tasks for each sentence. To identify tasks, we normalize words in sentences to their stem, base or root form using the Porter stemmer [103] (*e.g.*, "reduced", "reducing" and "reduces" are normalized to "reduce"). A task is formed by concatenating an intent and constraints that are extracted from a sentence. We describe our approach to extract intents and constraints in the following.

Identifying Intents

A sentence has at least a verb and a noun to express intents. Verbs and nouns from noun phrases or prepositional phrases are formed into action-object pairs [80]. An action-object pair shows how an action can be performed on an object. For example, "plan a Disney World trip" has an action "plan" and an object "Disney World trip". We identify intents from action-object pairs.

A sentence can have multiple intents depending on the number of action-object pairs. For example, "buy iPhone", "buy phone case" and "notify mother expense" are three intents for the Example 2 in Figure 4.1. We separate a sentence into multiple segments, and identify intents in each segment. A valid segment should have a verb and an object. We identify segments in the following two cases:

- *Segments separated by conjunctions:* As shown in the Example 2 of Figure 4.1, the sentence segment before the conjunction (*i.e.*, and) contains a verb "buy" and a noun phrase "an iPhone and a phone case". "Notify my mother about the expense" following the conjunction (*i.e.*, and) is another segment.
- *Segments in a prepositional phrase or a subordinate clause:* A prepositional phrase (*e.g.*, after buying an iPhone) or a subordinate clause (*e.g.*, to spend my holiday) can have sentence segments. We identify action-object pairs and extract sentence segments.

Segments, declaring facts about various situations, are irrelevant to service composition, *e.g.*, "The hotel is in Toronto". We filter out such sentence segments based on phrasal verbs and subject noun phrases:

- *Phrasal verb:* If only one verb exists and the verb is one of {be, is, am, are, was, were, 's, 're}, we filter out the segment since the verb is not an action.
- *Subject noun phrase:* If a subject noun phrase contains a determiner, such as "a", "the" and "this", we consider that the segment does not include a intent, since the segment declares facts about the referenced subject.

Table 4.1 shows our rules for extracting intents using various linguistic patterns from segments. In pattern #1, we skip a noun phrase without a noun and identify objects from the next noun phrase. For example, "buy her an iPhone" contains two noun phrases (*i.e.*, "her" and "an iPhone"), and has an intent "buy iPhone". A segment with a pattern #1, 2 or 3 can contain multiple intents, since objects in a phrase (*i.e.*, a noun phrase or a prepositional phrase) can be separated by a comma or a conjunction. For instance, "buy an iPhone and a phone case" contains two intents "buy iPhone" and "buy phone case". In an extracted intent description (using rules from Table 4.1), an object "this" or "it" is replaced with the last mentioned object [142].

We identify intents from segments in a subordinate clause. intents in a subordinate clause, such as an infinitive-clause and a *that*-clause (*i.e.*, a clause starts with a *that*), can be used to explain the main clause. We merge intents in the main clause and the subordinate clause in such a case. For example, Example 1 in Figure 4.1 has an intent "plan Disney World trip spend holiday".

Identifying Constraints

Constraints can be used to 1) fill in service parameters (*e.g.*, "to Orlando" for a destination); 2) identify a target person (*e.g.*, "for my sister"); and 3) restrict on the selected services (*e.g.*, "less than $500").

We identify constraints from prepositional phrases, noun phrases and clauses in a segment. Table 4.2 shows our rules for extracting constraints. We store constraints in a key-value pair form. A key defines a label for a constraint and a value shows the details of a constraint. For example, in pattern #3 of Table 4.2, the key is an object (*i.e.*, sister) and the value contains the object and a preposition (*i.e.*, {sister, for}).

Constraints can have an upper or a lower range to restrict the selected services. When phrases contain a numerical number or a preposition "than" (*e.g.*, pattern #

2, 4, 5 and 6 in Table 4.2), we detect such constraints with the following steps:

- The key of the constraint is a measurement unit, such as $ and dollars. Symbols (*e.g.*, $) are transferred into natural languages (*e.g.*, dollars). If no measurement unit is detected, we use the last mentioned object as the key (such as pattern #6 in Table 4.2).

- The value contains an operator that defines either a "less than" or a "greater than" relation. An operator can be inferred from adjectives (*e.g.*, less), prepositions (*e.g.*, under) and phrasal verbs (*e.g.*, reduce to). We define a bag of words that denote a "less than" relation: *less, lower, fewer, most, maximum, below, under, reduce, decrease*, and a "greater than" relation: *more, higher, least, minimum, over, increase, raise.* A word in a constraint that is synonymous with the bag-of-words determines an operator. We use WordNet [37], a large English database, to identify synonymous words. WordNet groups synonymous words into sets called *synsets* and connect synsets by different relations, such as hypernym and meronym. Hypernym denotes a "kind of" relation. For example, *bed* is a hypernym of *furniture.*

To make the extracted constraints meaningful, we recognize four types of constraints: *location* (*e.g.*, in Toronto), *time* (*e.g.*, at Christmas), *price* (*e.g.*, less than $500), and *relative* (*e.g.*, for my sister). To determine the type of a constraint, we use WordNet to identify ***hypernym trees*** for each word in the extracted ***key*** of the constraint. A hypernym tree represents a sequence of synsets, and each of synsets has a hypernym relation with the succeeding synset. The synsets of "location", "time", "monetary" and "relative" represent the conceptual semantic meaning of location, time, price and relative respectively. We traverse all the paths of the hypernym trees

Table 4.2: Rules to extract constraints from Noun Phrases, Prepositional Phrases and Clauses

	#	**Linguistic Pattern**	**Example**	**Key-Value pair (Key=Value)**
Noun Phrase	1	Adjective	cheap	Key:Adjective Value:{Adjective}; cheap={cheap}
	2	Quantifier Phrase	less than \$500	Key:Measurement Unit Value:{Number,Operator}; dollars={500, less than}
Prepositional Phrase	3	Without Number	for my sister	Key:Noun Value:{Noun,Preposition}; sister={sister,for}
	4	With Number	over 500 dollars	Key:Measurement Unit Value:{Number,Operator}; dollars={500, greater than}
	5	Preposition is “than”	price lower than iPhone	Key:Last Mentioned Object Value:{Noun, Operator}; price={iPhone, less than}
Clause	6	Verb + “to” + Number	if the price reduces to 500	Key:Measurement Unit Value:{Number, Operator}; price={500, less than}

to record the number of occurrences of the four synsets. The synset with the highest occurrence number wins and we classify the type of the constraint as the semantic meaning of the synset. The key of the constraint that is initially identified using rules in Table 4.2 is replaced with the constraint type. For example, “dollars” is replaced with “price”.

Our approach identifies intents and constraints in each segment of a sentence. The identified intent and constraint in a segment are concatenated to form a task. The number of tasks in a sentence depends on the number of identified intents.

4.3.3 Predicting Service Related Tasks

Many tasks extracted in Section 4.3.2 are related to human activities, *e.g.*, "inspect laptop damage". These tasks cannot be performed by services. To identify service related tasks from the extracted tasks, we use a deep learning based model, *i.e.*, the MLP model. Although we employ a simple logistic regression model to collect valid how-to instructions, we use the MLP model to accurately identify service related tasks. We decide to use the MLP model because we need a strong prediction power for developers to identify service related tasks and design composite services. Compared with the logistic regression model that combines the learning features linearly, the MLP model learns non-linear interactions among the learning features. Thus, the MLP model captures more complex feature interactions and may have more accurate task prediction results.

The MLP model has training and testing phases. In the training phase, a task t_i contains a learning feature vector X_i and a label l_i. X_i describes the semantic meaning of the task. The label l_i denotes whether the task is a service related task or not. A $l_i = 1$ denotes that a task is a service related task, while a $l_i = 0$ denotes, otherwise. In the testing phase, the model takes the feature vector of a task t_i as the input and predicts a label to the task t_i. We follow the three steps below to build our MLP model.

1) Building learning feature vectors. Our approach uses *word2vec* [93] to build learning feature vectors X_i. *word2vec* is a natural language processing (NLP) tool that analyzes the semantic meanings of words. The tool takes a corpus of text as the input and outputs an m dimensional numerical vector for each word in the corpus. The numerical vector captures the semantic meanings of a word, so that the word vectors of two similar words (*e.g.*, "laptop" and "computer") are similar to each other [93].

In our approach, we use word vectors that are pre-trained on more than 100 billion words from the GoogleNews corpus [93].[3] We set the dimension m of a word vector v_j as 300 (*i.e.*, $v_j \in \mathcal{R}^{300}$), similar to the work by Zheng *et al.* [165]. For an extracted task t_i, we remove stop words (*e.g.*, "a", "an" and "the") and lowercase each word. We map the sequence of words $(w_1, w_2, ...w_n)$ in the task to their corresponding word vectors $(v_1, v_2, ...v_n)$. For example, each word w_j in the task "plan Disney World trip Orlando" is mapped to a word vector $v_j \in \mathcal{R}^{300}$. We then average the word vectors as the feature vector $X_i \in \mathcal{R}^{300}$ for the task t_i.

2) Labeling tasks. To label a task t_i, we need to examine if there exist services that can perform the task. We have collected in total of 4,152 services in Section 3.4.1 of Chapter 3. Manually examining this large number of services is tedious and time-consuming. Therefore, we use the Vector Space Model built in Section 3.3.2 of Chapter 3 to rank the five most similar services for a task t_i. Afterwards, we perform a manual analysis (described in Section 4.4.2 **RQ4.2**) to label if the task t_i is a service related task or not. If at least one of the ranked services can perform the task t_i, the task is labeled as 1, Otherwise, the task is labeled as 0.

3) Building the MLP Model. The MLP model, as shown in Figure 4.3, contains an input layer and a prediction layer. The input layer consists of a learning feature vector X_i of a task, *i.e.*, the word vector. The prediction layer generates the predicted label $\hat{l_i}$ of the task. Between the input layer and the prediction layer, there are a number of hidden layers. A hidden layer h contains a number of neurons.

[3]https://code.google.com/archive/p/word2vec/

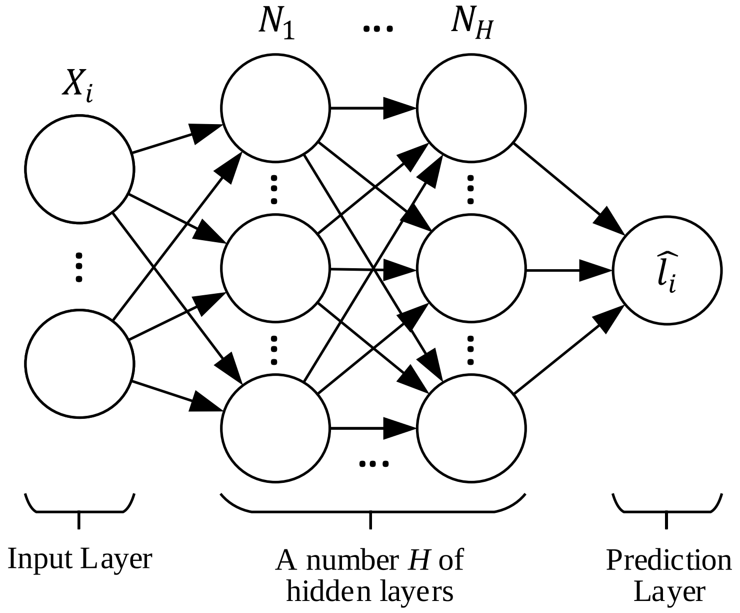


Figure 4.3: The structure of the MLP model

Each neuron N_h^j performs a weighted sum over the values on the previous layer. Next, the neuron applies a non-linear activation function, *i.e.*, Rectified Linear Units (*ReLU*) [66], on the weighted summation. The ReLU function is commonly used in the neural networks because it quickens the training process compared with other non-linear activation functions, *e.g.*, *tanh* and *sigmoid*. Equation 4.1 shows the operations performed by a neuron N_h^j.

$$O_h^j = ReLU(W_h^j * O_{h-1} + b_h^j) \tag{4.1}$$

where W_h^j *is the weight matrix;* b_h^j *is the bias term;* O_{h-1} *are the values of the previous layer; and* O_h^j *is the output.*

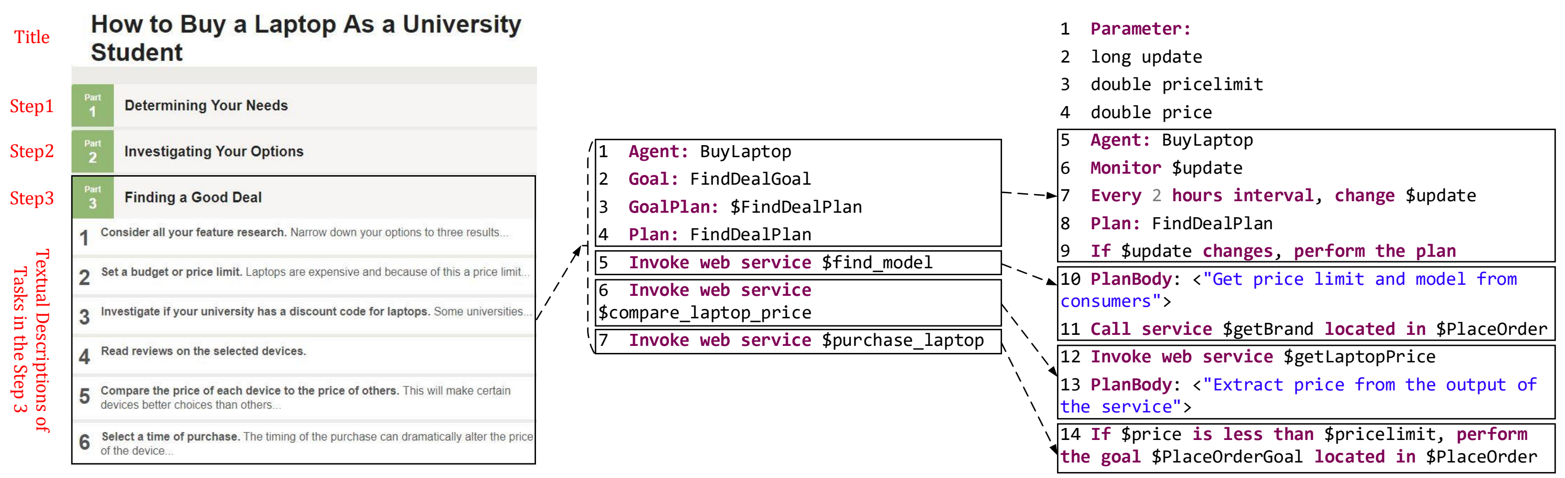
The prediction layer combines the outputs of the last hidden layer, *i.e.*, O_h, using a *sigmoid* function to generate the predicted label $\hat{l}_i$. We use the *sigmoid* function, since our task label prediction is a binary classification problem. The objective of our

MLP model is to minimize the task label prediction error, *i.e.*, the error between the actual label l_i and the predicted label $\hat{l}_i$. The task label prediction error is trained with the cross entropy loss function [116] for solving the binary classification problem. We adopt the Adam [61] algorithm, a stochastic gradient-based optimization algorithm, to optimize our loss function. The main benefit of Adam is that the learning rates of parameters are adapted in the training phase, thus alleviating the efforts to manually tune the learning rates. A task is identified as a service related task if the predicted label of the task is 1.

4.3.4 Generating Agent Specifications

To aid developers to specify agent specifications using the proposed semi-natural language syntax (see Chapter 3 Section 3.3.1), our approach generates agent specifications from valid how-to instructions. Figure 4.4b shows the generated agent specification for the *Finding a Good Deal* step of the how-to instruction shown in Figure 4.4a. We describe our approach to generate agent specifications below.

As various how-to instructions describe distinct intentions, our approach transforms each valid how-to instruction to an agent specification. The title of a valid how-to instruction is the name of an agent. A step of a valid how-to instruction may contain a goal and service related tasks that describe the plan to achieve the goal. We use our approach described in Section 3.3 to identify the service related tasks from the textual description of a step. Tasks that are unrelated to services are filtered out.



How to Buy a Laptop As a University Student

Part 1 Determining Your Needs

Part 2 Investigating Your Options

Part 3 Finding a Good Deal

1 **Consider all your feature research.** Narrow down your options to three results...

2 **Set a budget or price limit.** Laptops are expensive and because of this a price limit...

3 **Investigate if your university has a discount code for laptops.** Some universities...

4 **Read reviews on the selected devices.**

5 **Compare the price of each device to the price of others.** This will make certain devices better choices than others...

6 **Select a time of purchase.** The timing of the purchase can dramatically alter the price of the device...

(a) An annotated screenshot of a how-to instruction

```
Agent: BuyLaptop
Goal: FindDealGoal
GoalPlan: $FindDealPlan
Plan: FindDealPlan
Invoke web service $find_model
Invoke web service
$compare_laptop_price
Invoke web service $purchase_laptop
```

(b) A generated agent specification

```
Parameter:
long update
double pricelimit
double price
Agent: BuyLaptop
Monitor $update
Every 2 hours interval, change $update
Plan: FindDealPlan
If $update changes, perform the plan
PlanBody: <"Get price limit and model from
consumers">
Call service $getBrand located in $PlaceOrder
Invoke web service $getLaptopPrice
PlanBody: <"Extract price from the output of
the service">
If $price is less than $pricelimit, perform
the goal $PlaceOrderGoal located in $PlaceOrder
```

(c) A modified agent specification

Figure 4.4: An example how-to instruction and two examples of agent specifications for buying a laptop

The filtered out tasks may contain manual activities that serve as preconditions for performing service related tasks. For example, the filtered out manual task *set price limit* can be implemented before the service related task to place an order for a laptop (see Figure 4.4c). If developers consider such filtered out tasks important, they can modify the generated agent specification to include the filtered out tasks, as shown in Figure 4.4c. If all of the tasks in a step are not service related tasks, the step is identified as irrelevant to compose services. Thus, we filter out the step.

The step name (*e.g.*, *Finding a Good Deal*) is the name of the corresponding goal and is transformed to the *Goal Name* syntax (*e.g.*, *Goal: FindDealGoal*). A goal has an associated plan to perform the goal. We use the step name as the plan name, *e.g.*, *Plan: FindDealPlan* (the *Plan Name* syntax in Chapter 3 Table 3.4). The goal is linked with the corresponding plan using the *Associated Plan* syntax (*e.g.*, *GoalPlan: $FindDealPlan*). From the *Finding a Good Deal* step, our built MLP model identifies three service related tasks, *i.e.*, *find model, compare laptop price* and *purchase laptop.* Each service related task is transformed to the *Invoke Web Services* syntax (*e.g.*, *Invoke Web Service $find_model*, described in Chapter 3 Section 3.3.2). The plan body consists of a number of *Invoke Web Services* syntaxes, depending on the number of tasks. The generated agent specification serves as guidelines for developers to design agent specifications. A developer may modify the generated agent specification to the agent specification shown in Figure 4.4c.

4.4 Case Study

In this section, we present the setup of our case studies to evaluate our approach and the obtained results.

4.4.1 Collecting How-to Instructions

We collect 2,279 how-to instructions from the *wikiHow* website. The collected how-to instructions fall into 20 categories that are listed in the *wikiHow* website,[4] *e.g.*, *Arts and Entertainment*, *Computers and Electronics*, and *Education and Communications*. Our case studies answer the following two research questions.

RQ4.1. Is our approach effective to extract valid how-to instructions?

RQ4.2. Can our approach effectively identify service related tasks?

4.4.2 Research Questions

RQ4.1. Is our approach effective to extract valid how-to instructions?

Motivation. Instead of the extensive manual efforts of labeling all the collected 2,279 how-to instructions, we build a simple logistic regression (LR) model to extract valid how-to instructions (see Section 4.3.1). Accurately extracting valid how-to instructions reduces manual efforts in the labeling process. In this research question, we evaluate the performance of our model to extract valid how-to instructions.

Approach. To build the LR model, we construct training and testing sets. We apply the statistical sampling technique that randomly samples 329 from the 2,279 how-to instructions, with 95% confidence level and 5% confidence interval, as our training set. The statistical sampling technique indicates that the sampled instructions can represent the 2,279 how-to instructions [85]. The remaining 1,950 (*i.e.*, 2,279 - 329 = 1,950) how-to instructions are used as the testing set. To evaluate the performance of our model, we randomly sample 321 how-to instructions from the testing set. The labels of the sampled 321 how-to instructions are used as the ground truth. Two first

[4] www.wikihow.com

two authors independently perform a manual analysis to label the sampled how-to instructions (*i.e.*, 329 from the training set and 321 from the testing set) as valid or invalid, by examining the textual descriptions of the how-to instructions. We compute the Cohen's kappa [27] to measure the agreement between the two co-authors. The value of Cohen's kappa is 0.77, indicating a substantial agreement [69]. The two co-authors have a discussion session to reach a consensus on the disagreement. We evaluate the performance of our LR model using *Precision*, *Recall*, *F-Measure* and *Area Under Curve (AUC)*.

- *Precision* measures the ratio of correctly predicted valid how-to instructions from the set of how-to instructions that are predicted as valid (shown in Equation 4.2).
- *Recall* measures the fraction of how-to instructions that are correctly predicted as valid over the total valid how-to instructions (shown in Eqaution 4.3).
- *F-Measure* computes the harmonic average of the *Precision* and *Recall* to quantify the accuracy of our prediction results, as shown in Equation 4.4.
- *Area Under Curve (AUC)* measures the area under the curve that plots the true positive rates against the false positive rates [104]. The *AUC* provides an overview of the discriminatory power of a model. The values of AUC range from 0.5 to 1. A value of 0.5 represents random guessing, while 1 denotes that all the valid how-to instructions can be properly distinguished by a model.

$$Precision = \frac{\#relevant\ items \cap \#retrieved\ items}{\#retrieved\ items} \tag{4.2}$$

$$Recall = \frac{\#relevant\ items \cap \#retrieved\ items}{\#relevant\ items} \tag{4.3}$$

$$F\text{-}Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.4}$$

Results. Our approach can effectively extract valid how-to instructions. Our results reveal that our approach obtains a *Precision* of 72%, a *Recall* of 60%, an *F-Measure* of 0.65 and an *AUC* of 0.85 for identifying valid how-to instructions. Thus, instead of manually examining the 1,950 how-to instructions in the testing set, using the built LR model reduces our labeling efforts since we only need to manually verify 335 (out of 1,950) how-to instructions that are predicted as valid. In total, we verified and collected 275 valid how-to instructions from the training set and testing set. Our *Recall* of 60% suggests that our LR model may miss some valid how-to instructions. However, we have collected enough valid how-to instructions to identify service related tasks.

RQ4.2. Can our approach effectively identify service related tasks?

Motivation. To relieve developers from the effort of designing composite services, we propose an approach that automatically identifies service related tasks from how-to instructions. We provide guidelines for developers to build agent specifications using the identified tasks. A good task prediction performance helps developers design composite services using agents. We are interested to evaluate the performance of our approach on identifying service related tasks.

Approach. We use the approach described in Section 4.3.2 to extract tasks from valid how-to instructions that are collected in **RQ4.1**. In total, we extract 21,441 tasks. To build the MLP model (as described in Section 4.3.3), we randomly sample 2,680 tasks (*i.e.*, 12.5% of the extracted tasks). The first two co-authors independently perform a manual analysis to label whether the sampled tasks are service related tasks or not. Our manual labeling results show that we achieve the Cohen's kappa value of 0.64 between the two co-authors, indicating a substantial agreement. The two co-authors have a discussion session to reach a consensus on the disagreement. To train and test the performance of the proposed MLP model, we randomly split the sampled tasks into training, validation and testing sets using a ratio of 80:10:10 [17][165]. We use the validation set to tune the hyper-parameters of our model. Hyper-parameters are the variables that may decide the architecture of a MLP model (*e.g.*, the number of hidden layers). Different settings of hyper-parameters could influence the performance of the proposed model. Thus, we empirically examine the effects of two hyper-parameters: 1) the number of hidden layers and 2) the number of neurons. The number of hidden layers is searched in three settings, *i.e.*, {1, 2, 3}. The number of neurons in the first hidden layer determines the number of neurons in the consecutive hidden layers. For example, if the model has 3 hidden layers and 32 neurons in the first hidden layer, the model has the following neuron sizes: $32 \rightarrow 16 \rightarrow 8$. Thus, we test the number of neurons in the first layer among four numbers, *i.e.*, {32, 16, 8, 4}.

We evaluate the performance of our model using *Precision*, *Recall*, *F-Measure* and *Area Under Curve (AUC)* (described in the approach section of **RQ4.1**). Specifically, *Precision* measures the ratio of correctly retrieved tasks from the set of predicted service related tasks. *Recall* calculates the fraction of the service related tasks that our approach could retrieve. As *F-Measure* measures the accuracy of a prediction model, we use *F-Measure* as the evaluation metric to select hyper-parameters. Our results show that using AUC as the evaluation metric has the same effect as using *F-Measure* to select hyper-parameters. We test all the combinations of the two hyper-parameters on the validation set and compare the performances of the *F-Measure* to find the best setting. To evaluate the performance of our model on the testing set, we compare our model with the following 3 baselines, *i.e.*, *Logistic Regression*, *Random Forest* and *Naïve Bayes Classifier*.

- *Logistic Regression* is a well-known binary classification model that combines learning features linearly.
- *Random Forest* is an ensemble of decision trees [77] to perform classification or regression tasks. The prediction results consider each individual decision tree, thus preventing over-fitting.
- *Naïve Bayes Classifier* [115] is a simple probabilistic model for classification. The model assumes the conditional independence among the learning features.

The three baseline models and our model are built with the same learning features and labels.

Results. Our MLP model obtains the best performance compared with the three baseline models. Figure 4.7 shows the *F-Measure* obtained on the validation set by varying the number of hidden layers and the number of neurons. Our proposed MLP model obtains the best performance when there is 1 hidden layer and 8 neurons in the hidden layer. Figure 4.5 shows that the larger the number of hidden layers, the worse the performance. The model with 3 hidden layers performs predictions in which all the tasks are not relevant to services (*i.e.*, the *F-Measure* is 0). We empirically set the number of hidden layers as 1. Figure 4.6 shows that the model obtains the best performance when the number of neurons in the first hidden layer is set as 8. Further increasing the number of neurons decreases the performance of our model. The number of hidden layers and the number of neurons have the effect on the number of parameters to be trained in our model. A larger number of parameters could result in over-fitting.

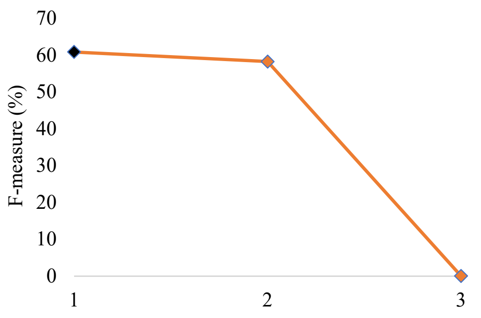


Figure 4.5: Number of hidden layers

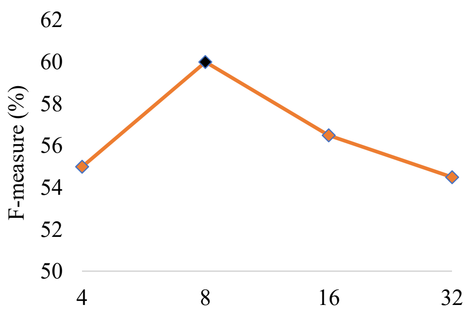


Figure 4.6: Number of neurons

Figure 4.7: *F-Measure* results on the validation set with respect to different hyper-parameters (black dots represent the optimal *F-Measure* values).

Table 4.3: The performance comparison of our model with three baseline models on the testing set.

Model	**Precision**	**Recall**	**F-Measure**	**AUC**
MLP	0.67	0.56	**0.61**	**0.88**
Logistic Regression	0.74	0.43	0.54	0.88
Random Forest	**0.86**	0.13	0.22	0.79
Naïve Bayes Classifier	0.29	**0.60**	0.39	0.71

Table 4.3 shows the performance comparison between our model and the 3 baseline models. Overall, our model achieves the best *F-Measure* (*i.e.*, 0.61) and *AUC* (*i.e.*, 0.88). AUC shows true capability of a model to distinguish between service

related tasks and other tasks, because AUC is not dependent on arbitrary threshold. Compared with our model, the *Logistic Regression* model achieves the same AUC value, but decreases the *Recall* by 13%. The lower *Recall* value of the *Logistic Regression* model produces lower *F-Measure* values. These observations suggest that our MLP model has a better prediction accuracy compared with the three baseline models. Our results show that, on average, 18% of extracted tasks (using the approach described in Section 4.3.2) are service related tasks. The results indicate that the manual identification has a maximum precision of 18%. In contrast, our approach achieves a precision of 67%, which improves the efficiency of developers to manually identify tasks by nearly four times. Moreover, our approach can identify a majority of service related tasks (*i.e.*, 56%).

4.5 Chapter Summary

Developers typically do not have complete knowledge when designing composite services. To help developers design composite services, we propose an approach to automatically identify service related tasks from on-line how-to instructions. To identify service related tasks, we build a logistic regression model to collect valid how-to instructions that are related to service composition. From the collected valid how-to instructions, we use natural language processing techniques to analyze the grammatical structures of sentences to extract tasks. Moreover, we build a MLP model to identify service related tasks from the extracted tasks. To help design agents using our proposed semi-natural language syntax (see Chapter 3), our approach transforms the identified service related tasks to our syntax. Our case study results show that our approach achieves an Area Under Curve (AUC) of 0.85 for predicting valid how-to instructions. Moreover, our proposed MLP model achieves an overall better performance compared with the three baseline models for identifying service related tasks.

Chapter 5

Automatically Transforming IoT Applications to RESTful Web Services

The diverse IoT applications in IoT devices lack a standard interface to allow the communication among various IoT devices and services. To use our multi-agent framework (described in Chapter 3) to compose various types of services, in this Chapter, we propose an automatic approach to transform the functionalities provided by IoT applications to IoT services that confirm to the RESTful paradigm.

Chapter Organization. In Section 5.1, we introduce the chapter. In Section 5.2, we present the programming structure of the source code of IoT applications. In Section 5.3, we give an overview of our approach to generate IoT services. In Section 5.4, we describe our case studies. Finally, we conclude this chapter in Section 5.5.

5.1 Introduction

The inter-connected physical devices, *i.e.*, the Internet of Things (IoT) devices, are prevalent in several aspects of our lives. For example, IoT devices may sense nearby environments (*e.g.*, obtain the temperature) and react upon an end-user's request to change the physical environment (*e.g.*, turn on the light). IoT applications are designed by application developers to provide functionalities in IoT devices, *e.g.*, to sense the temperature. In the meanwhile, the Internet has turned into a global infrastructure to host heterogeneous web services. End-users may use web services to perform various on-line activities, such as on-line shopping and banking. With web services and IoT devices combined, the possibilities to ease our daily lives increase in magnitude. For instance, an on-line grocery order can be made based on a food consumption alert that is triggered by analyzing the data read from a fridge sensor. However, this combination is not without its limitations. For example, end-users must install a large number of proprietary end-user applications (*e.g.*, mobile applications) on smart phones or computers to access the information of IoT applications in IoT devices. In addition, the diverse end-user applications lack a standard interface to allow the communication among various IoT devices and web services. Therefore, it is not trivial to integrate IoT devices with existing applications [105].

To ease the integration of IoT applications, we are interested in transforming IoT applications to IoT services, using the service-oriented architecture (SOA) to provide the functionalities offered by IoT devices. Our approach that transforms IoT applications to IoT services mainly works for smart-home IoT devices (*e.g.*, smart light and temperature sensor) that end-users can access and control. In particular, SOA based IoT services have two main advantages: (1) interoperability, which allows IoT services to exchange information with web services using a structured data format; (2) easy integration with existing applications due to the uniform interface of IoT services. Research effort has been invested on approaches to provide IoT services for end-users [31][43][105]. Nonetheless, most of these approaches run the IoT services on the IoT devices [31][43][105], which are not optimal, since IoT devices are typically designed with limited resources, *e.g.*, low battery capacity and processing power [105]. In addition, the complexity of SOA standards (*e.g.*, the verbose data format) generates energy and latency overheads in IoT devices that lead developers to spend extra effort when designing IoT services.

To overcome these practical limitations, we focus on automatically transforming the functionalities of IoT devices to IoT services. IoT services are designed using the RESTful paradigm. We use the cloud platform to host IoT services. In contrast to the resource limited IoT devices, the cloud platform has massive storage, high speed network and huge computing power. Furthermore, the cloud platform has the potential to host numerous IoT services and connect IoT devices as well as processing IoT data [109]. Additionally, the functionalities of IoT devices may be managed by standard APIs over the cloud, which may be accessed by end-users from any place.

More specifically, we analyze the source code of IoT applications to identify methods that can be controlled or accessed by end-users. Our approach further extracts the service specifications of the corresponding IoT services. A service specification describes the interface of an IoT service and is composed of three parts: service name, HTTP function and input (or output) parameters. To allow developers to modify the generated service specifications, we also propose a service schema that describes the service specifications of IoT services. The service schema identifies which data of IoT devices that should be stored in the cloud. Moreover, we use the service schema to instantiate IoT services with friendly user interfaces.

5.2 Programming Structure of the Source Code of IoT Applications

In this section, we provide background material about the programming structure of the source code of IoT applications.

The methods in the source code of IoT applications can be classified into two types: *internal methods* and *external methods*. An *internal method* is related to the set up of an IoT device and is only consumed by methods within the IoT device (*e.g.*, an *init* method to set up an IoT device). An *external method* works as an IoT device interface that can communicate with the cloud. The input variables of an external method may represent the input commands of an actuator (*e.g.*, to turn on the light), while the returned variables may represent the sensed data of a sensor (*e.g.*, the sensed temperature). Since an external method allows end-users to control an IoT device or obtain information from an IoT device, it is possible to transform such a method to an IoT service.

```
def led_control(status):
    if status == "ON":
        turn_led_on()
    elif status == "OFF":
        turn_led_off()
```

Figure 5.1: External method 1

```
def getTemperature():
    temp ← methods to
get temperature
    return temp
```

Figure 5.2: External method 2

Figure 5.3: Examples of external methods

Figure 5.3 shows examples of external methods that are extracted from the hackster.io website.[1] Hackster.io is a website that shares projects on embedded devices (*e.g.*, Raspberry Pi). In Figure 5.3, the names of the methods describe the methods' intent (*i.e.*, `led_control` and `getTemperature`). The `led_control` method[2] (Figure 5.1) can receive commands from the cloud (*i.e.*, by using the `status` variable). This method uses an *if-else statement* to identify whether the led has to be turned on or off depending on the `status` variable. The `getTemperature` method[3] (Figure 5.2) retrieves the temperature from a sensor. A developer can define methods within the external method to send the sensed temperature values to the cloud, *e.g.*, `send(temperature, url)`. Table 5.1 shows the service specification that may be extracted from the two example methods.

Table 5.1: Service specification that is extracted from the external methods in Figure 5.3

Method Name	**Service Name**	**HTTP Function**	**Input Parameters**	**Output Parameters**
led_control	led_control	POST	status	
getTemperature	getTemperature	GET		temp

5.3 Overview of Our Approach

In this section, we present our approach to automatically generate IoT services from IoT applications. Figure 5.4 shows an overview of our approach. Our approach has four activities. Each activity is explained in a subsection below.

[1]https://www.hackster.io/
[2]https://www.hackster.io/user3424878278/pool-fill-control-119ab7
[3]https://www.hackster.io/dexterindustries/add-a-15-display-to-the-raspberry-pi-b8b501

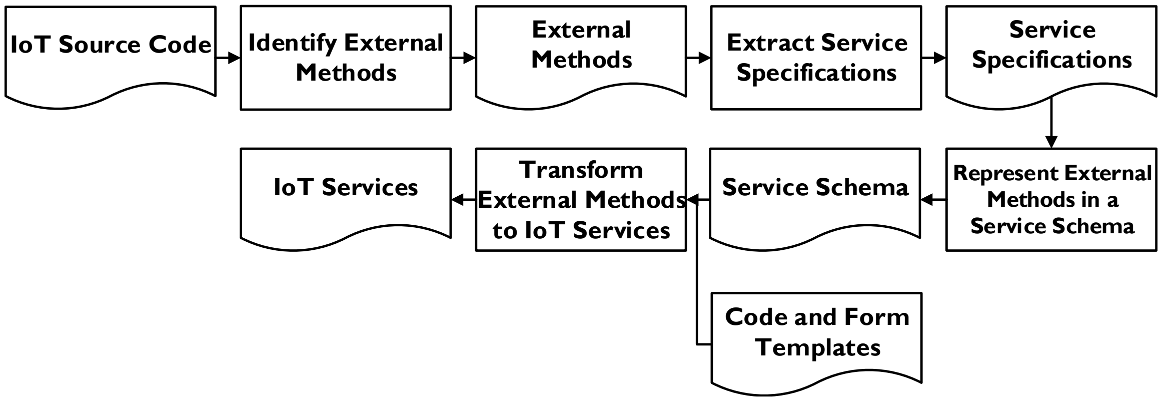

IoT Source Code
Identify External Methods
External Methods
Extract Service Specifications
Service Specifications
Represent External Methods in a Service Schema
Service Schema
Transform External Methods to IoT Services
IoT Services
Code and Form Templates

Figure 5.4: An overview of our approach

5.3.1 Identifying External Methods

To save developers' effort on manually finding code methods that should be transformed, we analyze the source code of IoT applications written in Python to investigate whether external methods can be automatically identified. We choose the Python language, since it is suitable for developing IoT applications due to its portability and easy-to-learn syntax [135]. Although our approach is language-independent, we use Python examples to explain our approach implementation. We explain the steps that are involved in this code analysis below.

STEP1: Parsing Source Code of Methods

To identify external methods, we first analyze the Abstract Syntax Tree (AST) of the source code. An AST is a tree structure that represents the syntax of the source code. Each node in the tree describes a construct (*e.g.*, method name) that is present in the source code. We traverse the tree to identify the following constructs in a method:

- *method name, e.g.*, `getTemperature` shown in Figure 5.2.
- *input variables, e.g.*, `status` shown in Figure 5.1.

- *returned variables, e.g.*, `temp` shown in Figure 5.2.
- *method calls in a method body, e.g.*, `turn_led_on()` shown in Figure 5.1.
- *if-else statements, e.g.*, `if status == "ON"` shown in Figure 5.1.

STEP2: Filtering Internal Methods

An internal method can be identified based on its extracted constructs. Methods with the following *internal features* (IF) are considered as internal methods. Internal methods are filtered out and are not investigated further.

- ***IF1****: method name containing the keywords "init, setup, debug, test".* Method names containing the keywords "init" and "setup" are initialization methods and are used to configure the initial settings, *e.g.*, setting the voltage level of GPIO pins. Method names containing the keywords "debug" and "test" are testing methods, which are used to test the different functionalities of an IoT device. Such testing methods are internal methods in an IoT device.
- ***IF2****: method name starting with "_".* The leading underscore in a method name denotes that the method is for internal use or reserved for the programming language (*e.g.*, an *__init__* method) [1].
- ***IF3****: methods that are called within internal methods or defined in internal files.* File names containing the keywords "init, setup, debug, test" or starting with "_" are internal files. Methods that are called within internal methods or defined in internal files are used for initialization and testing.

STEP3: Processing Method Names

A method name may convey the intent of the method, which can be used to distinguish external methods from internal methods. To identify the semantics of method names, we use the following steps to normalize these names. We split CamelCase words (*e.g.*, *getTemperature* is split into *get* and *temperature*). We remove the punctuation, *e.g.*, "_" and "-". We also remove the suffixes that contain numbers (*e.g.*, *led1* is normalized into *led*). Finally, we remove stop words (*e.g.*, "a", "the" and "is"). We use natural language processing (NLP) techniques to identify the part-of-speech (POS) tag of each word. For example, "get" is tagged as a verb and "temperature" is tagged as a noun. Finally, we perform word stemming to find the root words (*e.g.*, "reduced", "reducing" and "reduces" are normalized to "reduce"). These words are used to extract features for identifying external methods.

STEP4: Extracting Features for External Methods

We extract the following *external features* (EF) based on the constructs of the methods that are identified in STEP 1.

- ***EF1****: method calls.* If the methods that are called within a method body contain *send* related keywords in their names, *i.e.*, "push, post, publish, send, notify", these methods likely send data to the cloud, *e.g.*, `send(temperature, url)` and are considered as external methods.

- ***EF2****: if-else statements.* In case a method contains if-else statements that react to the input variables of the method when receiving commands from the cloud, such a method has a high probability of being an external method. For example, the `led_control` method in Figure 5.1 contains if-else statements that react to changes in the `status` variable.

- ***EF3***: *semantic of verbs.* Verbs in method names may represent the action that is performed in a method. For instance, *control* and *get* are the verbs in the examples of Figure 5.3. We identify the semantic of verbs to infer external methods. For example, if a verb has keywords that are related to sending and receiving messages (*i.e.*, "push, post, publish, send, notify, subscribe, get, sense, set, receive, control"), we infer that its respective method transmits data to the cloud. These methods are likely external methods.

- ***EF4***: *semantic of nouns.* Nouns in method names denote the objects of interest of these methods, *e.g.*, *led* and *temperature* are nouns in the examples of Figure 5.3. If these nouns match with IoT service names, their respective method is likely an external method. We identify IoT services by using the iotlist.co[4] website. This website lists various IoT devices, *e.g.*, security cameras and smart lights. For an IoT device, we manually extract their functionalities, each one corresponding to an IoT service name. For example, the Elgato Eve Room Wireless Indoor Sensor[5] will have the *sense air quality, sense temperature* and *sense humidity* IoT service names. In total, we extracted 190 IoT service names. Next, we use the approach described in STEP 3 to extract nouns from the extracted IoT service names. We form a bag of words containing the nouns and match them with the nouns that we find in method names (see STEP 3). For instance, the Wireless Indoor Sensor has a bag containing the *air, quality,*

[4]http://iotlist.co/
[5]https://www.elgato.com/en/eve/eve-room

temperature and *humidity* words that we match with the *temperature* word in the `getTemperature` method (see Figure 5.2).

In our approach, we assume that a method is an external method if it has at least two of the features that we identify in STEP 4. For example, the method in Figure 5.1 is an external method, since it has the *if-else statements* (*i.e.*, *EF2*) and *semantic of nouns* (*i.e.*, *EF4*) features.

5.3.2 Extracting Service Specifications

Based on the analyzed external methods, we extract the service specifications for their respective IoT services (see Table 5.1), *i.e.*, *service name, HTTP function* and *input (or output) parameters.*

We use the method name as the service name. For instance, *led_control* is the service name for the method in Figure 5.1. Then, we use the *external features* described in STEP 4 to distinguish HTTP GET and POST functions. Each HTTP function is associated with two *external features.* Among these *external features*, we split the *semantic of verbs* into *semantic of send* and *semantic of receive* for GET and POST functions, respectively. We explain the details below.

- *HTTP GET:* is associated with the *semantic of send* and *method calls* features. The *semantic of send* denotes that a verb in a method name contains *send* related keywords, *i.e.*, “push, post, publish, send, notify, get, sense”. Such methods send data to the cloud, so that GET-based IoT services can identify and retrieve this data.

- *HTTP POST:* is associated with the *semantic of receive* and *if-else statements* features. The *semantic of receive* denotes that a verb in a method name contains *receive* related keywords, *i.e.*, “set, receive, control, subscribe”. An IoT device receives commands from POST-based IoT services.

To determine which HTTP function should be associated with an external method, we count the number of features that belong to an external method. If an external method has a given feature, that feature has a counter of 1 (one). We derive a score for the HTTP GET function (*i.e.*, S_{get}) using Equation 5.1 and a score for the HTTP POST function (*i.e.*, S_{post}) using Equation 5.2.

$$S_{get} = C_{semantic\ of\ send} + C_{method\ calls} \tag{5.1}$$

$$S_{post} = C_{semantic\ of\ receive} + C_{if-else\ statements} \tag{5.2}$$

where $C_{semantic\ of\ send}$, $C_{method\ calls}$, $C_{semantic\ of\ receive}$ and $C_{if-else\ statements}$ denote the counters for the respective features.

We use the S_{get} and S_{post} scores to determine whether the HTTP function should be GET or POST, i.e., whichever has the highest value. In case S_{get} is equal to S_{post}, we calculate the fan-in and fan-out of an external method [143]. Fan-in represents the number of input variables of an external method, while fan-out denotes the number of returned variables of an external method. When a fan-in to fan-out ratio is larger than one, the POST function is chosen, since such a ratio indicates that an external method is written to receive data (see `led_control` in Figure 5.1). The GET function is chosen otherwise.

Finally, the parameters of IoT services are extracted based on the identified HTTP functions. For example, the returned variables of a GET-based external method are extracted as the output parameters of the corresponding IoT service. Comparatively, the input variables of a POST-based external method are extracted as the input parameters of the corresponding IoT service. As an example, the `status` variable of the POST-based `led_control` method (shown in Figure 5.1) is extracted as a service input parameter.

5.3.3 Representing External Methods in a Service Schema

To transform external methods to IoT services, we need a structured data format that describes the extracted service specifications of IoT services. We design a service schema using the Web Ontology Language (OWL) [7]. In the service schema, the identified service name, HTTP function and parameters of a service specification are prefilled. A developer may validate, modify and complete the service schema. Figure 5.5 shows how we use OWL to define our service schema.

The service schema is composed of four main components: *classes, individuals, relations* and *attributes.* A *class* represents a group of objects with similar properties. For example, an *IoT device* is a class. A *relation* is used to connect the components of our service schema (e.g., an *IoT service hasOperations*). A *class* can be inherited by *sub-classes.* For instance, a *reading* operation, which is used to get the latest value of a sensor, is a sub-class of *operation.* An *individual* is an instance of a class. Finally,

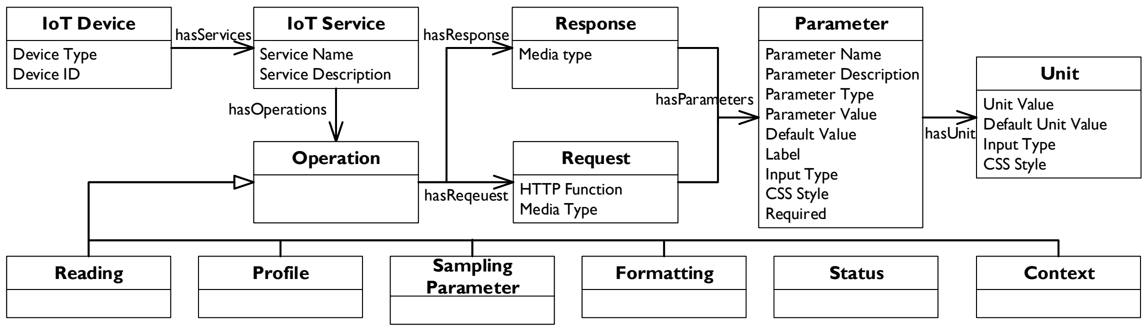


Figure 5.5: The service schema

attributes declare the properties of a class. For instance, the *IoT device* class has the *device type* and *device id* attributes. The *device type* groups a number of IoT devices that provide similar functionalities. For example, *temperature sensor* may be a device type. The *device id* attribute is unique for each IoT device and is used to distinguish one IoT device from another. The MAC address of an IoT device can be used as a device id.

A functionality of an IoT device publishes a single stream of scalar values (*e.g.*, temperature values) to a channel on the cloud [31]. The stream of scalar values is considered as a resource of an IoT service. This resource is stored in a resource database on the cloud. An IoT service identifies its resources using the *service name, device type* and *device id* attributes. An IoT service provides multiple operations to perform different actions onto a resource. For instance, the IoT service "sense temperature" can obtain the latest reading of the temperature and modify the frequency at which the temperature should be sensed. We identify six operations of IoT services based on the approach proposed by Haggerty *et al.* [31], *i.e.*, *reading*, *profile*, *sampling parameter*, *formatting*, *status* and *context*. Each external method falls in one of the operations specified in Table 5.2. The *reading* operation is used to get the latest value of a resource. This operation listens to an IoT service's resource until a new value of that resource is received. Then, the listened value and a timestamp of the value update are returned to end-users. The *status* operation returns the state of a given IoT service (*e.g.*, whether it's on or off). For actuators, an end-user may send a POST request to the *status* operation, which changes the physical state of an IoT device (*e.g.*, to turn on the light). An operation is identified by the URL pattern (see Figure 5.6).

IoT devices with the same *device type* value correspond to one unique service schema that is used to describe their respective IoT services. We use the service schema to instantiate IoT services as we describe in Section 5.3.4.

Table 5.2: A summary of available operations for an IoT service

Operation Name	**HTTP Function**	**Device Class**	**Description**	**Example**
Reading	GET	sensor	the latest reading	temperature
Profile	GET	sensor	a number of recent history readings	history temperature readings
Sampling Parameter	GET/POST	sensor	the sampling frequency of sensing	100Hz
Formatting	GET/POST	sensor	the unit of the sensed value	°C, °F
Status	GET/POST	sensor/actuator	the state of the IoT service	turn light on/off
Context	GET/POST	sensor/actuator	the location of the measurement	the location that the temperature is being sensed

5.3.4 Transforming External Methods to IoT Services

In this section, we describe how our approach automatically transforms external methods to IoT services.

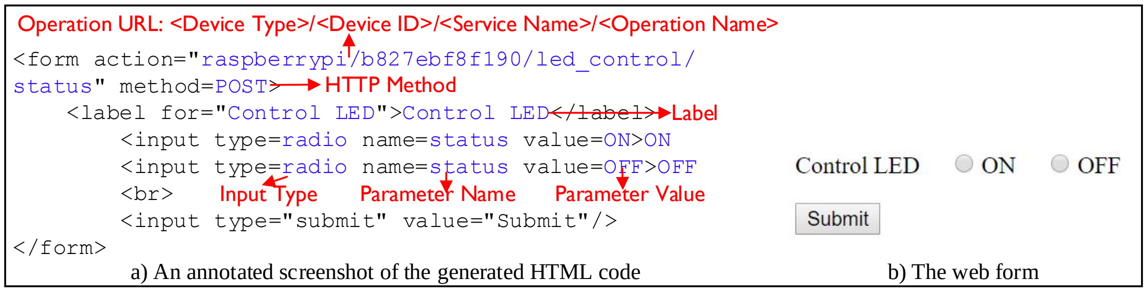
STEP1: Generating Web Forms

Since end-users may not be familiar with SOA, it is important to provide friendly user interfaces for accessing and controlling IoT services. In this regard, our approach automatically generates web forms by using our proposed service schema and form templates. A template uses the data of a service schema to generate text output, *e.g.*, source code or HTML forms. These generated forms are used to send POST requests to IoT services.

```
<Device Type>/<Device ID>/<Service Name>/<Operation Name>
```

Figure 5.6: The URL schema for accessing an operation.

We design our form templates using the FreeMarker template engine.[6] The essential components of a web form are the HTTP function, the operation URL and the parameters to be filled by end-users. We traverse the parameters in our service schema to identify which ones have an *input type* attribute. The *input type* attribute can assume one of the HTML input elements, *i.e.*, *text, radio* and *select*. The *parameter value* attribute (see Figure 5.5) defines the available options of a parameter, which end-users can choose, *e.g.*, ON or OFF. A developer may define a CSS style for an input parameter using the *CSS style* attribute. Figure 5.7 shows an HTML form example for controlling a led. Once an end-user clicks on the "Submit" button, a POST request is submitted to the operation URL.



Operation URL: <Device Type>/<Device ID>/<Service Name>/<Operation Name>

```
<form action="raspberrypi/b827ebf8f190/led_control/
status" method=POST>
    <label for="Control LED">Control LED</label>
        <input type=radio name=status value=ON>ON
        <input type=radio name=status value=OFF>OFF
        <br>
        <input type="submit" value="Submit"/>
</form>
```

Figure 5.7: An annotated screenshot of a web form that is used to control a led. The blue text highlights the data that is extracted from our service schema.

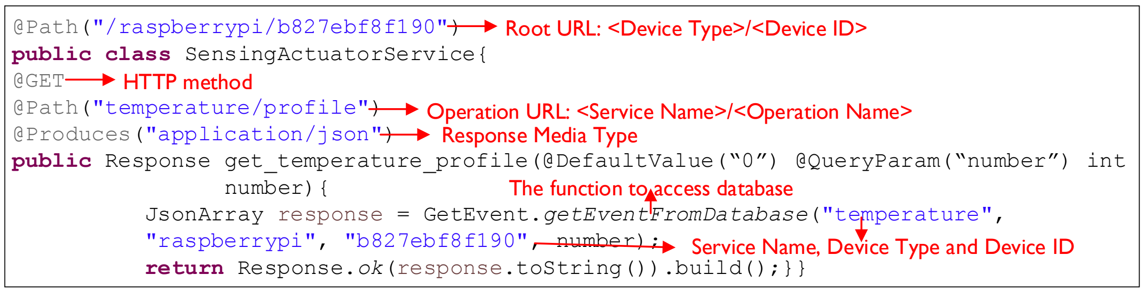
STEP2: Instantiating IoT Services

Our approach automatically generates source code to instantiate IoT services using the proposed service schema and code templates. The instantiated IoT services follow

[6]http://freemarker.org/

the Jersey[7] syntax standard.

To instantiate an IoT service, a code template needs to be filled with the information from a service schema. The required information are the HTTP function, an operation URL, a request media type, a response media type and the filled parameters from a web form. We provide code templates for each kind of operation. In a *reading* template, a function is provided to listen to a resource of an IoT service, which then responds end-users with the real-time value. The resources of a given IoT service are located by the *service name*, *device type* and *device id* that are extracted from the URL of the respective service request (see Figure 5.6). Once the *profile* operation is requested, the IoT service fetches the last N values of a resource from the database. The N value is specified by end-users as a URL parameter. We also build databases on the cloud for each of the other four operations, *i.e.*, *sampling parameter, formatting, status* and *context.* A GET request of an operation locates the respective database and retrieves the data value. Figure 5.8 shows an example of an instantiated *profile* operation.



```
@Path("/raspberrypi/b827ebf8f190")
public class SensingActuatorService{
@GET
@Path("temperature/profile")
@Produces("application/json")
public Response get_temperature_profile(@DefaultValue("0") @QueryParam("number") int
                number){
          JsonArray response = GetEvent.getEventFromDatabase("temperature",
          "raspberrypi", "b827ebf8f190", number);
          return Response.ok(response.toString()).build();}}
```

Root URL: <Device Type>/<Device ID>
HTTP method
Operation URL: <Service Name>/<Operation Name>
Response Media Type
The function to access database
Service Name, Device Type and Device ID

Figure 5.8: An annotated screenshot of the *profile* operation, which obtains the temperature. The blue text highlights the data that is extracted from our service schema.

For POST-based operations (*e.g.*, POST status of light), the filled parameters

[7]https://jersey.java.net/

(*e.g.*, ON) must be sent to the respective IoT device. We traverse the parameters of a service schema to identify which parameters end-users should fill. Instantiated POST-based operations use their parameter names (*e.g.*, status) as variables that will retrieve the values that are filled in web forms.

Once end-users invoke an instantiated operation, the generated source code of that operation is accessed by the operation URL and the HTTP function. The data that is transmitted between an IoT device and the cloud follows the JSON standard (*i.e.*, JavaScript Object Notation), a lightweight data-interchange format [30].

5.4 Case Study

We conduct a case study to evaluate our approach. In this section, we introduce the setup of our case study and we present the obtained results.

5.4.1 Case Study Setup

To test the effectiveness of our approach on identifying external methods, we analyze IoT applications written in Python. We collect the source code of IoT projects in the "Raspberry Pi" category on the hackster.io website.[8] The Raspberry Pi is a credit-card-sized embedded device, which is widely used to develop IoT solutions for home and industrial automation.

In total, we collect 1,039 projects, of which 177 contain python methods (17%). We collect a total of 17,178 python methods from these projects. The collected IoT projects have different domains, *e.g.*, living (*e.g.*, light control), communication (*e.g.*, radio receiver) and transportation (*e.g.*, parking system). Table 5.3 summarizes our

[8]https://www.hackster.io/

Table 5.3: The distribution of projects and python methods in each domain. *Avg LOC* denotes the average lines of code for each project.

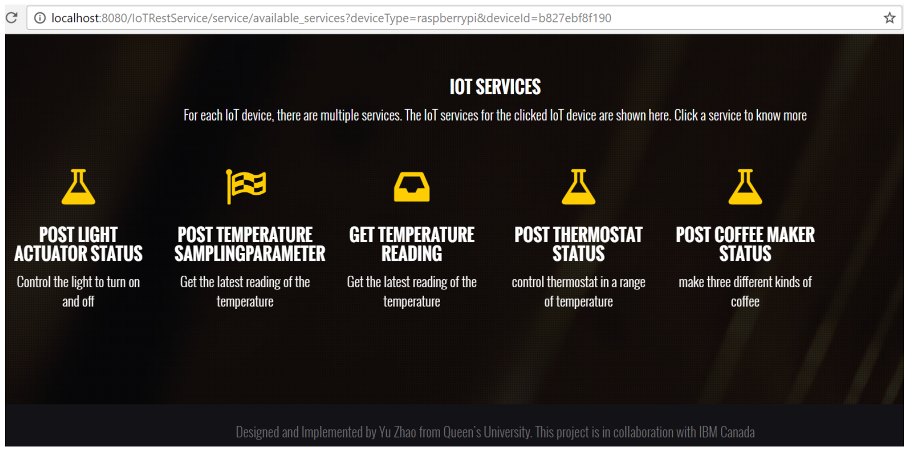
Domain	**# Projects**	**# Methods**	**Avg LOC**	**Domain**	**# Projects**	**# Methods**	**Avg LOC**
Living	41	7,035	6,349	Environmental Sensing	24	763	553
Transportation	10	387	756	Health	21	599	728
Entertainment	41	4,660	1,781	Security	22	1,122	870
Communication	18	2,612	3,891	**Total**	177	17,178	2,520

collected data. We built a prototype tool as a proof of concept for our approach. Our tool automatically analyzes IoT applications and generates the corresponding IoT services based on the identified external methods. We use the Raspberry Pi 3 Model B as our IoT device (denoted as *RPi*). The IoT device has a quad-core processor running at 1.2GHz. We use the IBM cloud platform, which uses the MQTT protocol to communicate with IoT devices. Since we are not allowed to build customized IoT services in such a commercial cloud platform, we use our approach to generate IoT services in our local server (see Section 3.4). Our server transmits data of IoT devices with the IBM cloud platform using the MQTT protocol. Figure 5.9 shows a screenshot of our prototype tool. An end-user may click on an operation to send a GET request or retrieve a web form to submit a POST request. Our case study answers the following research questions:

RQ5.1. How effective is our approach to identify external methods and extract service specifications?

RQ5.2. How accurate is our approach to generate IoT services?

We manually evaluate all the results in our case study. Our evaluator has three years' experience on building RESTful services for the service-oriented architecture.


localhost:8080/IoTRestService/service/available_services?deviceType=raspberrypi&deviceId=b827ebf8f190

IOT SERVICES

For each IoT device, there are multiple services. The IoT services for the clicked IoT device are shown here. Click a service to know more

POST LIGHT ACTUATOR STATUS
Control the light to turn on and off

POST TEMPERATURE SAMPLINGPARAMETER
Get the latest reading of the temperature

GET TEMPERATURE READING
Get the latest reading of the temperature

POST THERMOSTAT STATUS
control thermostat in a range of temperature

POST COFFEE MAKER STATUS
make three different kinds of coffee

Designed and Implemented by Yu Zhao from Queen's University. This project is in collaboration with IBM Canada

Figure 5.9: A screenshot of our tool that shows the available operations of IoT services to end-users

5.4.2 Research Questions

RQ5.1: How effective is our approach to identify external methods and extract service specifications? To measure the effectiveness of our approach, we randomly sample 376 methods from the extracted 17,178 python methods with a 95% confidence level and a 5% confidence interval [163]. We apply our approach (as described in Section 5.3.1) to identify external methods and extract service specifications from the sampled 376 methods. We use precision and recall as shown in Equations 5.3 and 5.4 to evaluate our approach. *Precision* measures the ratio of correctly retrieved external methods (or service specification parts) from the set of external methods (or service specification parts) that are retrieved by our approach [160]. On the other hand, *recall* measures the ratio of external methods (or service specification parts) from the dataset that our approach could retrieve [160].

$$Precision = \frac{\{relevant\ items\} \cap \{retrieved\ items\}}{\{retrieved\ items\}} \tag{5.3}$$

$$Recall = \frac{\{relevant\ items\} \cap \{retrieved\ items\}}{\{relevant\ items\}} \tag{5.4}$$

Our results reveal that our approach has an average precision of 75% and a recall of 72% for identifying external methods. As for service specifications, our approach has an average precision of 82% and a recall of 81%. The main reasons for the misidentified external methods and service specifications are the following: (1) We are not able to extract semantic meanings from method names. For instance, the method `getdoorstatus` is an external method to get the door status. However, we could not find the *semantic of nouns* and *semantic of verbs* because this name does not follow the CamelCase pattern. (2) Internal methods that transmit messages within an IoT device may have *external features*. For example, the method `SendParameter` is an internal method that sends parameters using the I2C (Inter-Integrated Circuit) protocol. However, we identify such a method as an external method, since it contains the *semantic of verbs* and *if-else statements* features. (3) The input (or output) parameters are defined in the code method body. For instance, the method `get_ph_reading` has a service parameter *ph_value*. However, the method uses a print function to display the parameter, rather than returning it.

RQ5.2. How accurate is our approach to generate IoT services? Our approach uses a service schema to generate IoT services on the cloud platform. The accuracy of transforming external methods to IoT services is what represents the practical usefulness of our approach. To evaluate the accuracy of generating IoT services, we use the 190 extracted IoT services described in Section 5.3.1. We design external methods on *RPi* depending on the type of an IoT service. For example, we design four possible external methods for an IoT service that is generated for a sensor. These methods are: *reading, sampling parameter, formatting* and *context.*

As for IoT services generated for actuators, we design two external methods: *status* and *context*. We do not design an external method for the *profile* operation, since a *profile* operation is instantiated to fetch a resource from a resource database. We use the approach described in Section 5.3.3 to generate service schemas for the designed external methods. We automatically generate IoT services using our approach (see Section 5.3.4). Equation 5.5 shows how we measure the accuracy of our approach.

$$Accuracy = \frac{\{\#correctly\ generated\ IoT\ services\}}{\{\#IoT\ services\}} \tag{5.5}$$

The accuracy is the ratio of the number of correctly generated IoT services to the total number of IoT services. Since an IoT service is composed of several operations, we evaluate whether an operation is correctly instantiated. A GET-based operation is correctly instantiated if, for example, a GET request for the *temperature reading* operation returns the values that match the temperature values sent from *RPi*. A POST-based operation is correctly instantiated if, for example, the external method on *RPi* that is used for receiving light status receives the commands from the *light status* operation. An IoT service is correctly generated when all the operations of such an IoT service are correctly instantiated. Our approach achieves an accuracy of 96% (182 out of 190 IoT services) when generating IoT services. Nonetheless, our approach fails to generate IoT services regarding streaming media. A streaming media IoT service constantly delivers and presents multimedia, *e.g.*, video and audio, to end-users. We do not find support in the IBM cloud platform for streaming media of IoT devices.

5.5 Chapter Summary

To integrate multiple IoT devices in a uniform environment, we provide an approach that automatically transforms functionalities from IoT devices to SOA based IoT services. We automatically generate web forms for end-users to have a friendly experience when interacting with IoT services. We use the designed service schema and templates to generate IoT services. Our case studies show that we can identify external methods from IoT applications with a precision of 75% and a recall of 72%. We can also extract service specifications from these external methods with a precision of 82% and a recall of 81%. Our approach can generate IoT services with an accuracy of 96%.

Chapter 6

Predicting User Ratings Using User Reviews and Service Descriptions

Due to the massive amount of services available, an end-user faces numerous choices to meet their personal preferences when selecting the desired services from the services with the similar functionalities. To relieve end-users' cognitive overloading when selecting services, it is essential for our designed agents (described in Chapters 3, 4 and 5) to predict the preferred services for end-users. In this chapter, we propose a deep content model (*i.e.*, DeepCont) to predict user ratings on unseen services, using user reviews, service descriptions and user ratings.

Chapter Organization. Section 6.1 describes the introduction of this chapter. Section 6.2 introduces the convolution operation used in the Convolutional Neural

Networks (CNN) [73]. Section 6.3 presents the background of the variational autoencoder network. Section 6.4 shows an overview of our proposed DeepCont model. Section 6.5 and Section 6.6 show the case study setup and the obtained results of our case studies, respectively. Finally, Section 6.7 concludes this chapter.

6.1 Introduction

Services are prevalent in our daily lives. In 2017, the Amazon.com website[1] had over 606 million shopping items available for end-users in the United States.[2] In this Chapter, we take on-line shopping items (*e.g.*, laptops) as services and a category of shopping items (*e.g.*, electronics) as a task. While choosing a service (*e.g.*, a laptop) under a task (*e.g.*, electronics), end-users come across a vast amount of similar services to choose from. To select a preferred service, an end-user needs to carefully examine the service descriptions (*e.g.*, laptop configurations) and user reviews in order to compare the user ratings of similar services before making a decision.

However, the service selection process can be tedious, since end-users need to read several service descriptions and user reviews of different services. To assist end-users in the service selection process, recommendation systems have emerged to suggest services that best fit the end-users' needs. Many existing approaches focus on predicting user ratings for the services that have been rated or reviewed by other end-users. For example, collaborative filtering (CF) approaches [95][112] use past user ratings to predict future user ratings. The assumption of the CF is that similar end-users that share the same preferences tend to provide similar user ratings for services. Despite the feasibility of the approach, the performance of user rating prediction decreases

[1]Amazon.com

[2]https://www.scrapehero.com/how-many-products-does-amazon-sell-worldwide-october-2017/

when very few services are rated in the past (*i.e.*, the sparsity problem). Other approaches leverage user reviews as additional information of collaborative filtering to predict user ratings [21][81][91][165]. These approaches extract user latent features from user reviews. A user latent feature is an inferred vector in which each number may represent an end-user's preference score. Compared with the CF approaches, using user reviews improves performance in predicting user ratings. However, service descriptions, which describe various service properties, are disregarded in these approaches. Incorporating service descriptions may further improve the prediction performance.

New services are continuously added in E-commerce websites and applications to attract more end-users and stimulate consumption. A recommendation system should not only recommend existing services with a rich set of user reviews, but also new services without any user reviews and user ratings. Recommending new services is the so-called cold start problem [119]. To alleviate the cold start problem, many approaches [75][146][147] extract service latent features from service descriptions available in new services, regardless the presence of user reviews and user ratings. The extracted service latent features are combined with collaborative filtering for service recommendation. In these approaches [75][146][147], the service descriptions are processed as bag-of-words, which fail to consider the order of words and the semantic meanings expressed in the descriptions. Therefore, we conjecture that the learned service latent features may not accurately represent the services. Moreover, these approaches do not use user reviews to extract end-users' preferences.

To accurately predict user ratings while alleviating the cold start problem, we propose the *Deep Content Model* (DeepCont), which is a neural network based probabilistic generative model. Our model takes both user reviews and service descriptions as inputs. Our model relieves the sparsity problem by extracting user latent features from user reviews, and alleviates the cold start problem by mining service latent features from service descriptions. Instead of learning latent features from bag-of-words, our model converts each word into a pre-trained word vector [93], forming a word embedding matrix for a particular user review or a service description. Such word embedding matrix preserves the order of words and captures semantic mearnings [165]. Our model jointly learns the latent features for end-users and services using two coupled generative convolutional neural networks, *i.e.*, convolutional variational autoencoder (VAE) network [62][113]. Li *et al.* [75] and Serban *et al.* [121] demonstrate that the VAE network can generate robust latent features. Moreover, the two convolutional VAE networks are learned in a jointly manner, so that user and service latent features are combined to predict user ratings. Therefore, the user rating information can guide the learning of latent features. In turn, the latent features contribute to the model prediction power.

We evaluate the performance of our model with two tasks: 1) predicting user ratings of existing services and 2) predicting user ratings of new services. We conduct experiments on 21 real-world sparse datasets from Amazon [91], with a total of more than 6.3 million services (*i.e.*, shopping items) and 55 million user reviews. We compare the performance of our model with four state-of-the-art baselines, which are the collaborative filtering based model, *i.e.*, PMF [95]; the model using user reviews, *i.e.*, DeepCoNN [165]; and two models that use service descriptions, *i.e.*, CDL [147] and CVAE [75]. Our results show that our model significantly improves the performance of the four baselines for predicting user ratings of existing services. On average, our model improves the prediction accuracy of the best performing baseline (*i.e.*, CDL) by a margin of 3.08%. For predicting user ratings of new services, compared with the best baseline model (*i.e.*, CVAE), DeepCont performs significantly better or equally well in 11 out of 21 datasets.

6.2 Convolution Operations

The convolution operation is commonly used in the Convolutional Neural Networks (CNN) [73] for processing three dimensional inputs (*e.g.*, images), which contain width, height and depth. Compared with the traditional neural network [48], CNN reduces the number of trained parameters, thus improving computation efficiency. We apply CNN to learn latent features from the word embedding matrix of a user review or a service description.

CNN contains multiple convolutional layers. Each layer contains a number of filters to transform the input to a higher level of abstracted features. Each filter is a weight matrix with a size of $t \times t$ (where t is called the window size), plus a bias term. A filter F performs the convolution operation on the input. The convolution operation computes the dot product of the weights of a filter with the region of the input, which has the same window size t. To walk through the input, the convolution operation moves the filter with a stride number s to compute the dot product with different regions of the input. We empirically set the stride number as 2. Given an input with three dimensions, *i.e.*, width=w, height=h and depth=d, a filter generates an output with two dimensions ($\frac{w}{2} \times \frac{h}{2}$). By applying a number of filters (*i.e.*, m), a convolutional layer concatenates the outputs of the filters to generate an output with three dimensions $(\frac{w}{2} \times \frac{h}{2} \times m)$.

Transposed Convolution Operations. The transposed convolution operation (also named as backward convolution [70] and fractionally strided convolution [107]) can be considered as the inverse of the convolution operation. This operation upsamples the width and height of the input by a factor of the stride number. Similar to the convolution operation, we empirically set the stride number as 2. After applying the operation, a filter transforms an input with the shape of $(\frac{w}{2} \times \frac{h}{2} \times m)$ to the output with the shape of $(w \times h)$.

6.3 Variational Autoencoder Network (VAE)

An autoencoder is a generative model that is used to learn a latent representation z_i from an input x_i (*i.e.*, a user review or a service description). The learned latent representation z_i is a compressed vector that can represent the input x_i. In an autoencoder, the latent representation z_i is learned by an encoder network and a decoder network. The encoder network estimates an input x_i as a latent representation z_i and the decoder network reconstructs the input x_i from the latent representation z_i.

The traditional autoencoder network [145] estimates a latent representation as a single vector (*i.e.*, a point). Wang *et al.* [147] take the learned representations as the service latent features to represent services. Different from the point estimation, a VAE network estimates a latent representation z_i as a variational distribution $q_\phi(z_i|x_i)$, where ϕ is the parameter for the encoder network. Instead of mapping an input x_i to a single point, VAE maps x_i to a region of a space. $q_\phi(z_i|x_i)$ is set as a diagonal Gaussian distribution [76] as shown in Equation 6.1.

$$q_\phi(z_i|x_i) \sim \mathcal{N}(\mu_\phi(x_i),\, \text{diag}(\sigma^2_\phi(x_i))) \tag{6.1}$$

where $\mu_\phi(x_i)$ and $\sigma^2_\phi(x_i)$ are the mean and variance, respectively, which are learned by the encoder network.

Optimizing VAE. To ensure that the decoder network can generate valid and diverse reconstructed outputs from any sampled latent representation z_i, the Kullback-Leibler (KL) divergence [67] is used to regularize the distribution of the encoder $q_\phi(z_i|x_i)$. Equation 6.2 shows the loss function of the VAE network, which consists of the reconstruction loss (*i.e.*, negative log-likelihood to construct an input) and the KL divergence.

$$\mathcal{L}(x_i;\theta,\phi) = -\mathbb{E}_{q_\phi(z_i|x_i)}[\text{log}p_\theta(x_i|z_i)] + \text{KL}(q_\phi(z_i|x_i)||p(z_i)) \tag{6.2}$$

where ϕ is the parameters of an encoder network and θ is the parameters of a decoder network.

6.4 The Proposed Deep Content Model

In our model, a data entry, *i.e.*, $(u, i, r_{ui}, rev_u, con_i)$ contains a user u, a service i, a user rating r_{ui} given by the end-user u, a user review rev_u written by the end-user u and a service description con_i. A user review rev_u is a concatenation of all the user reviews written by the user u in the training set, similar to the work by Zheng *et al.* [165]. A service description con_i is a concatenation of the title and the content description of the service i, similar to the approach used by Li *et al.* [75]. Our recommendation task is to predict the user rating $\hat{r}_{ui}$ given by the user u for an unseen service i, using the user review rev_u and the service description con_i. In this section, we present the details of our proposed deep content (DeepCont) model. First, we present the architecture of DeepCont. Then, we discuss the objective function used to train DeepCont.

6.4.1 Architecture

The architecture of our proposed model is shown in Figure 6.1. Our model contains two parallel convolutional VAE networks, *i.e.*, Net_u and Net_i. The convolutional VAE network Net_u learns k-dimensional user latent features from user reviews, and the convolutional VAE network Net_i learns k-dimensional service latent features from service descriptions. Since the architectures of Net_u and Net_i are the same, we only describe the details of Net_u. Our architecture consists of five main components: 1) word embedding layer; 2) convolutional network; 3) latent representation layer; 4) transposed convolutional network; and 5) prediction layer (in between Net_u and Net_i in Figure 6.1). We describe each component in the following subsections.

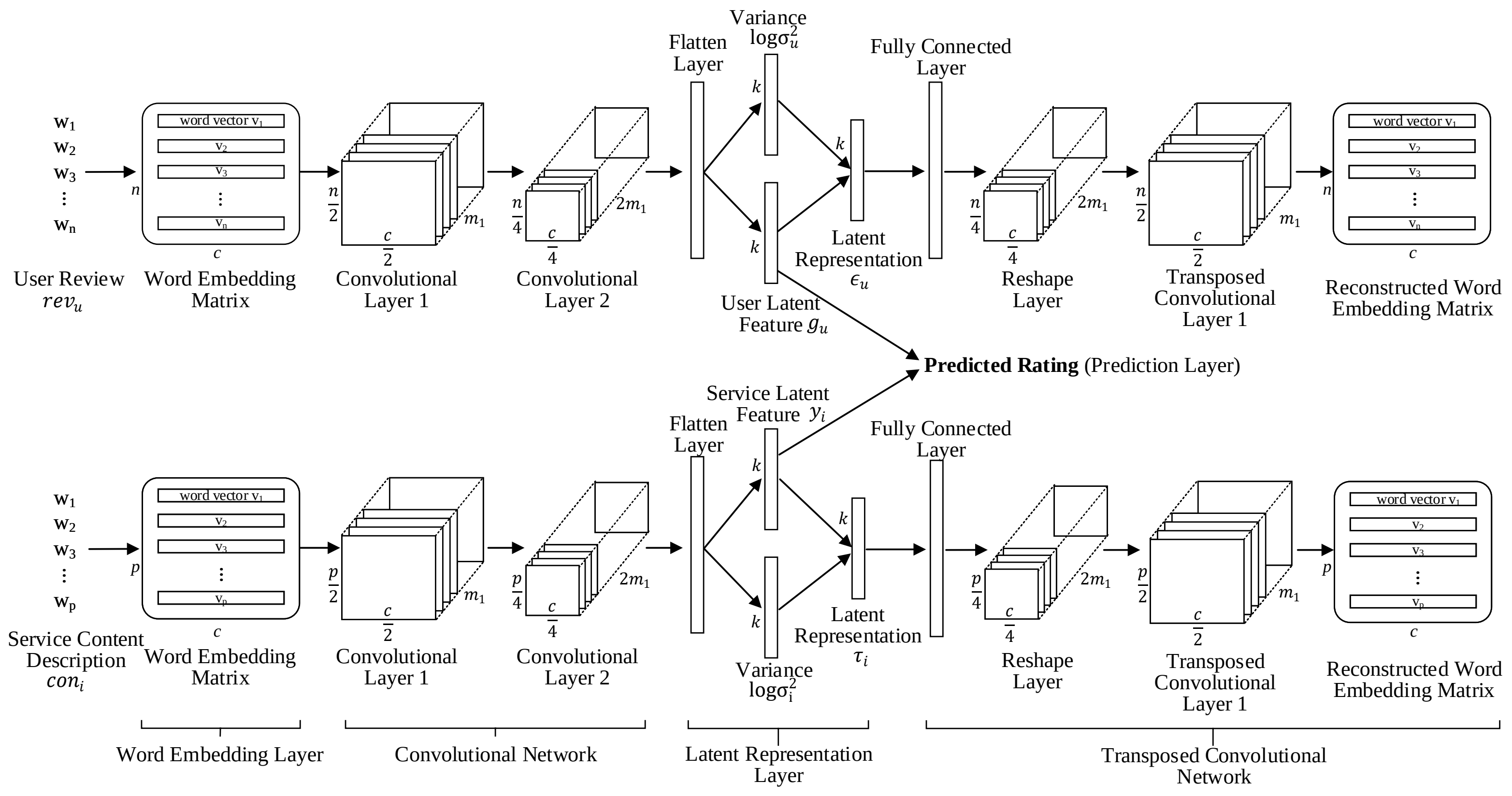

w1
w2
w3
wn
User Review rev_u
Word Embedding Matrix
word vector v1
Convolutional Layer 1
Convolutional Layer 2
Flatten Layer
Variance $\log\sigma_u^2$
User Latent Feature g_u
Latent Representation ϵ_u
Fully Connected Layer
Reshape Layer
Transposed Convolutional Layer 1
Reconstructed Word Embedding Matrix
Predicted Rating (Prediction Layer)
Service Latent Feature y_i
wp
Service Content Description con_i
Variance $\log\sigma_i^2$
Latent Representation τ_i
Word Embedding Layer
Convolutional Network
Latent Representation Layer
Transposed Convolutional Network

Figure 6.1: The architecture of our proposed DeepCont model (n and p denote the number of words in a user review and a service description, respectively; c is the dimension of a word vector; and m_1 represents the number of filters)

Word Embedding Layer

The word embedding layer converts a user review rev_u to the corresponding word embedding matrix V_u, as shown in Figure 6.1. We use word vectors pre-trained on the GoogleNews corpus by *word2vec* [93], to convert a word w_i to the corresponding word vector v_i with c dimensions, *i.e.*, $v_i \in \mathcal{R}^c$. We set the dimension c as 300, similar to the work by Zheng *et al.* [165]. Given a user review rev_u, we map the sequence of words $<w_1, w_2, ..., w_n>$ to their corresponding word vectors, *i.e.*, $<v_1, v_2, ..., v_n>$. The word vectors are concatenated to form the word embedding matrix $V_u \in \mathcal{R}^{n \times c}$ (where n is the number of words and c denotes the dimension of a word vector). The word vectors follow the order of the words that appear in the text. These word vectors are fixed during the training process, similar to the work by Catherine and Cohen [17].

Convolutional Network

The convolutional network takes a word embedding matrix V_u as an input to learn the user latent feature g_u. This convolutional network consists of a number L of convolutional layers (see Section 6.2). A convolutional layer l has a number m_l of filters.

Each filter F_l^j at the l^{th} layer applies the following operations. First, the convolution operation (see Section 6.2) is performed on the output O_{l-1} from the previous layer. Second, we perform the Batch Normalization [58] to normalize the output from the convolution operation by fixing the mean and variances of the output. The normalization process stabilizes and accelerates the learning of the convolutional network by preventing the saturation problem [58]. Batch normalization is critical for deep generation networks to learn latent representations [70][107]. Third, we perform a non-linear function, *i.e.*, Rectified Linear Units (*i.e.*, *ReLU*) [66] on the output from the batch normalization. Compared with other functions, *e.g.*, *tanh* and *sigmoid*, *ReLU* accelerates the training of the convolutional network [66]. Fourth, we apply the dropout [128] technique to reduce overfitting. In the training phase, the dropout technique randomly drops some of the values in the output from the *ReLU* function with a dropout ratio d. Thus, the parameters that are connected to the dropped values are not trained. In the testing phase, the dropout technique is disabled to use all the learned parameters for user rating predictions. Finally, in the testing phase, a filter F_l^j produces the output O_l^j using Equation 6.3:

$$O_l^j = ReLU(BNorm(O_{l-1} * F_l^j + b_l^j)) \tag{6.3}$$

where $*$ *denotes the convolution operation;* b_l^j *is the bias for the filter;* $BNorm$ *denotes the batch normalization operation.*

Given a number m_l of filters at the l^{th} convolutional layer, the output O_l of the convolutional layer concatenates the outputs from each filter (see Section 6.2), *i.e.*, $O_l = (O_l^1, O_l^2, ..., O_l^{m_l})$. Thus, the output O_l has three dimensions, with depth=m_l. As commonly used in the literature for convolutional networks [49], in our architecture, the number of filters m_l at the l^{th} convolutional layer doubles the number of filters in the previous layer, *i.e.*, $m_l = 2 \times m_{l-1}$.

Latent Representation Layer

The latent representation layer estimates the latent representation ϵ_u (see Section 6.3) of a user review rev_u. The input to the latent representation layer is the output from the convolutional network O_L (see Section 6.4.1). The convolutional network and the latent representation layer form the encoder of the convolutional VAE network. The latent representation layer contains a flatten layer and two fully connected layers. We describe each layer in detail in the following.

The flatten layer flattens the output O_L from three dimensions to one dimension. Next, the two fully connected layers estimate the mean $\mu_u \in \mathcal{R}^k$ and the variance $\sigma_u^2 \in \mathcal{R}^k$ from the output O_L, using Equation 6.4 and 6.5, respectively.

$$\mu_u = ReLU(O_L W_\mu + b_\mu) \tag{6.4}$$

$$\log \sigma_u^2 = ReLU(O_L W_\sigma + b_\sigma) \tag{6.5}$$

where W_μ and b_μ are the weights and bias in a fully connected layer to estimate mean, respectively; and W_σ and b_σ are the weights and bias to estimate the variance, respectively.

Finally, the latent representation $\epsilon_u \in \mathcal{R}^k$ is sampled from the obtained mean and the variance using the reparametrization trick [62][113]. In the next subsection, we describe the decoder of our network Net_u, *i.e.*, the transposed convolutional network.

Transposed Convolutional Network

The transposed convolutional network reconstructs the word embedding matrix V_u from the sampled ϵ_u in the latent representation layer. The transposed convolutional network consists of a fully connected layer, a reshape layer and a number L of transposed convolutional layers [44][107]. We describe each layer in detail as follows.

The fully connected layer transforms the sampled latent representation ϵ_u to the output O'_L, to be served as the input for the reshape layer (shown in Equation 6.6).

$$O'_L = ReLU(\epsilon_u W_z + b_z) \tag{6.6}$$

where W_z and b_z are the weight matrix and bias term for the fully connected layer to estimate the output O'_L.

We apply the dropout technique on the output O'_L during training. The reshape layer reshapes O'_L from one dimension to three dimensions that have the same shape as O_L (*i.e.*, the output of the convolutional network described in Section 6.4.1).

A transposed convolutional layer contains a number of filters that perform the transposed convolution operation (see Section 6.2) to up-sample the width and height of an input. The number of filters in a transposed convolutional layer halves the depth of the previous layer. For example, if a reshaped output O'_L has the depth of 32, the connected transposed convolutional layer applies 16 filters on the O'_L to keep the architecture of the transposed convolutional network symmetrical to the architecture of the convolutional network. The final transposed convolutional layer applies one filter. Thus, the output is the reconstructed word embedding matrix with two dimensions.

Similar to the convolutional network, after the transposed convolution operation, we apply the batch normalization, the *ReLU* function and dropout. Since the value of each element in the word embedding matrix V_u ranges from -1 to 1, we apply the *tanh* function at the final transposed convolutional layer to ensure that the values in the reconstructed word embedding matrix have the same range as the input matrix V_u.

By modeling end-users and services using the architecture as described above, we have two convolutional VAE networks Net_u and Net_i. The two networks are combined by the prediction layer to predict user ratings.

Prediction Layer

The prediction layer connects the two convolutional VAE networks *i.e.*, Net_u and Net_i, to make predictions of user ratings. The network Net_u (or Net_i) learns the probabilistic distribution of the latent representation ϵ_u (or τ_i) from a user review (or a service description). In our work, we take the mean of the latent representation ϵ_u as the user latent feature g_u, *i.e.*, $g_u = \mu_u$, and take the mean of the latent representation τ_i as the service latent feature y_i. The user latent feature g_u captures the overall preferences of end-users from the user review rev_u, and the service latent feature y_i represents the overall service properties from the service description con_i.

The prediction layer combines the two latent features g_u and y_i to generate the predicted user rating $\hat{r}_{ui}$. On one hand, the latent features are learned through reconstructing word embedding matrix in convolutional VAE networks. On the other hand, the user rating predictions guide the learning of the latent features. To estimate the predicted user rating $\hat{r}_{ui}$ from the latent features g_u and y_i, we concatenate g_u and y_i to a vector $e = <g_u, y_i> \in \mathcal{R}^{2k}$. We use the Factorization Machines (FM) [111] to predict the user rating $\hat{r}_{ui}$ as shown in Equation 6.7, similarly to the study by Zheng *et al.* [165].

$$\hat{r}_{ui} = w_0 + \sum_{i=1}^{2k} w_i e_i + \sum_{i=1}^{2k} \sum_{j=i+1}^{2k} \langle \hat{v}_i, \hat{v}_j \rangle e_i e_j \tag{6.7}$$

where $w_0 \in \mathcal{R}$ *is the global bias term;* $w_i \in \mathcal{R}$ *is the weight for* e_i *(*i.e.*, the* i^{th} *variable in the concatenated vector* e*);* $\hat{v}_i$ *is the vector associated with* e_i*;* k *is the dimension* *of a latent feature;* $\langle\cdot,\cdot\rangle$ *denotes the inner product between two vectors.*

6.4.2 Training DeepCont

Our DeepCont model contains two convolutional VAE networks for modelling end-users and services, and a prediction layer that connects the two networks to predict user ratings. Thus, the objective of our model is to minimize the reconstruction loss and the KL divergence of the two convolutional VAE networks (as shown in Equation 6.2), and to minimize the user rating prediction error, *i.e.*, the error between the actual user rating r_{ui} and the predicted user rating $\hat{r}_{ui}$. The user rating prediction error can be trained with different loss functions, such as the absolute error loss (L_1) or the squared error loss (L_2). Our experimental results show that training with L_1 loss produces better performance than the L_2 loss (similarly to the work by Zheng *et al.* [165] and Catherine and Cohen [17]). Therefore, we use L_1 loss as the loss function to train $\hat{r}_{ui}$. Given a training data entry, the DeepCont model minimizes the loss function as shown in Equations 6.8 and 6.9.

$$\mathcal{L}_{DeepCont} = \mathcal{L}_{Net_u} + \mathcal{L}_{Net_i} + \mathcal{L}_r \tag{6.8}$$

$$\begin{aligned}\mathcal{L}_{Net_u} =& -\mathbb{E}_{q_{\phi_u}(\epsilon_u|rev_u)}[\log p_{\theta_u}(rev_u|\epsilon_u)]\\ &+ \mathrm{KL}(q_{\phi_u}(\epsilon_u|rev_u)||p(\epsilon_u))\\ \mathcal{L}_{Net_i} =& -\mathbb{E}_{q_{\phi_i}(\tau_i|con_i)}[\log p_{\theta_i}(con_i|\tau_i)]\\ &+ \mathrm{KL}(q_{\phi_i}(\tau_i|con_i)||p(\tau_i))\\ \mathcal{L}_r =& |r_{ui} - \hat{r}_{ui}|\end{aligned} \tag{6.9}$$

where ϕ_u and θ_u are the parameters of the encoder and decoder networks of Net_u, respectively; Similary, ϕ_i and θ_i are the parameters of the encoder and decoder network of Net_i, respectively.

We use a batch of M shuffled training data entries to train our models. The loss function of the batch is obtained by averaging the loss function $\mathcal{L}_{DeepCont}$ for all the training data entries in the batch. At each batch, the two convolutional VAE networks (*i.e.*, Net_u and Net_i), and the prediction layer are jointly trained to minimize the total loss $\mathcal{L}_{DeepCont}$. The optimization of the loss functions $\mathcal{L}_{Net_u}$ and $\mathcal{L}_{Net_i}$ generates representative user and service latent features. Meanwhile, the minimization of the loss $\mathcal{L}_r$ guides the learning of the latent features so as to minimize the prediction error. To optimize the loss function $\mathcal{L}_{DeepCont}$, we adopt the Adaptive Gradient (Adagrad) [33] algorithm, a stochastic gradient-based optimizer. The main benefit of Adagrad is that the learning rates of parameters are adapted based on the updated frequency of the parameters, which reduces the manual effort to tune the learning rates.

6.5 Case Study Setup

We perform extensive experiments on 21 categories of the Amazon Product Data [50].[3] Each category contains user reviews and service descriptions on the services from one domain, *e.g.*, office products. We compare the performance of our DeepCont model with four state-of-art baseline models. In this section, we firstly discuss the datasets of our experiments. Then, we describe our evaluation settings. Finally, we discuss the four baseline models.

[3]http://jmcauley.ucsd.edu/data/amazon/

6.5.1 Datasets

We conduct experiments on the Amazon Product Data, which is a publicly available dataset. The dataset is collected during the time period of May 1996 - July 2014, from the Amazon website,[4] a large E-commerce shopping platform. The dataset has already removed the duplicated user reviews. The collected data has 24 product categories. Since the categories *Apps for Android* and *Amazon Instant Video* do not have service descriptions, we remove the two categories from our evaluation. We also remove the category *Books* because this category alone contains more than 22 million user reviews. We do not have enough computing resources to allocate such a large scale of dataset in the memory at once. The compared baselines (described in Section 6.5.3) and our DeepCont model cannot predict user ratings for the *Book* category. The statistical description of our used 21 categories of datasets is shown in Table 6.1. Our datasets are sparse. On average, an end-user provides 1.73 ratings. Zheng *et al.* [165] show that for end-users and services with fewer reviews and user ratings, a model that leverages user reviews gains more performance improvement compared to models that do not leverage user reviews. The categories of our dataset fall into different domains, such as clothing and office products. In total, our studied datasets contain more than 6.3 million services and 55 million user reviews.

For each category, we have two repositories, *i.e.*, a user review repository and a service information repository. The user review repository has a number of tuples. Each tuple (u, i, r_{ui}, rev_{ui}) contains an end-user u, a service i, a user rating r_{ui} and a user review rev_{ui} , which is given by the end-user u for the service i. The service information repository describes various service information, such as the service title

[4] Amazon.com

Table 6.1: Statistical description of out datasets

Datasets	#users	#services	#ratings
Grocery and Gourmet Food	768,438	166,049	1,297,156
Clothing Shoes and Jewelry	3,117,268	1,136,004	5,748,920
Home and Kitchen	2,511,610	410,243	4,253,926
Kindle Store	1,406,890	430,530	3,205,467
Sports and Outdoors	1,990,521	478,898	3,268,695
Cell Phones and Accessories	2,261,045	319,678	3,447,249
Health and Personal Care	1,851,132	252,331	2,982,326
Toys and Games	1,342,911	327,698	2,252,771
Video Games	826,767	50,210	1,324,753
Tools and Home Improvement	1,212,468	260,659	1,926,047
Beauty	1,210,271	249,274	2,023,070
Office Products	909,314	130,006	1,243,186
Pet Supplies	740,985	103,288	1,235,316
Automotive	851,418	320,112	1,373,768
Patio Lawn and Garden	714,791	105,984	993,490
Baby	531,890	64,426	915,446
Digital Music	478,235	266,414	836,006
Musical Instruments	339,231	83,046	500,176
Movies and TV	2,088,620	200,941	4,607,047
CDs and Vinyl	1,578,597	486,360	3,749,004
Electronics	4,201,696	476,002	7,824,482
Total	30,934,098	6,318,153	55,008,301

(*i.e.*, item title) and the content description of the service. We cross reference the two repositories to generate data entries $(u, i, r_{ui}, rev_u, con_i)$ in three types of sets (*i.e.*, training, validation and testing sets) for 1) predicting user ratings of existing services and 2) predicting user ratings of new services, as described below.

Generating datasets for predicting user ratings of existing services. An existing service is a service that may have received user ratings and user reviews from other end-users. Given a category, we randomly split the user review repository into training, validation and testing sets using a ratio of 80:10:10 [17][165]. We simulate the real-world recommendation settings, in which user reviews in the validation and testing sets are not accessed [23].

Generating datasets for predicting user ratings of new services. A new service does not have any user ratings or user reviews before making user rating predictions. Thus a service in the testing set never appears in the training set. To provide the training, validation and testing sets, we randomly split the services in the service information repository with a ratio of 80:10:10. Thus, the services in the validation and testing sets do not appear in the training set. To assign a tuple in the user review repository to a given set (*i.e.*, training, validation or testing), we check the set to which the service of the tuple is assigned. For example, if the service of a given tuple is assigned to the testing set, then we assign this tuple to the testing set. User reviews are not accessed in the validation and testing sets. Thus, the services in the validation and testing sets are new services that have not received any user reviews.

We preprocess user reviews and service descriptions in the two recommendation tasks using the following two steps: 1) words are lowercased; and 2) numbers and punctuations are removed. Moreover, we set the maximum length of a user review rev_u as ℓ. The rationale to set the maximum length of a user review rev_u is that the number of words of user reviews in a dataset follows a long tail distribution [17][21]. Thus a small number of user reviews have an extremely large number of words. ℓ is selected by setting the text cut-off percentage p, using a similar approach to the one used by Chen *et al.* [21]: 1) Given a dataset, we sort the user reviews based on the number of words in an ascending order; 2) We take the user review at the p percent quantile of the sorted list as the reference user review; and 3) ℓ is set as the number of words in the reference user review. We use the first ℓ words in a user review rev_u and cut off words that are longer than ℓ. The maximum length of a service description is set using the same approach as setting a user review. The user review rev_u and the service description con_i that do not have text are represented as empty strings.

6.5.2 Evaluation Settings

We use the Mean Squared Error (MSE) metric to evaluate the performance of our model, as shown in Equation 6.10. MSE measures the average differences between the actual user ratings r_{ui} and the predicted user ratings $\hat{r}_{ui}$. A lower MSE value denotes a more accurate user rating prediction.

$$MSE = \frac{1}{N}\sum_{u,i}(r_{ui} - \hat{r}_{ui})^2 \tag{6.10}$$

where N denotes the total number of data entries in the validation set or testing set; r_{ui} denotes the actual user rating; and $\hat{r}_{ui}$ denotes the predicted user rating.

Our proposed DeepCont model is implemented in Python with Keras[5], which uses

[5]https://keras.io/

Tensorflow[6] as the backend. Keras is a high level framework for implementing neural networks. Tensorflow is a widely used framework for numerical computation. Our model is trained and tested on NVIDIA Tesla P100 GPUs with 12 GB memory.

6.5.3 Baselines

To test the effectiveness of our DeepCont model that uses both user reviews and service descriptions, we compare the performance of our model with four baseline models that either use user reviews or service descriptions. The four baseline models fall in three categories: 1) collaborative filtering based models that neither use user reviews nor service descriptions. We choose the Probabilistic Matrix Factorization (PMF) [95] as the representative model; 2) Collaborative filtering based models that only use user reviews. We choose the state-of-the-art model, *i.e.*, DeepCoNN [165]. DeepCoNN is claimed to achieve a better performance compared with other models that use user reviews (*e.g.*, the HFT model [91]); and 3) Collaborative filtering based models that use only service descriptions. We compare with two state-of-the-art models in this category, *i.e.*, Collaborative Deep Learning (CDL) [147] and Collaborative Variational Autoencoder (CVAE) [75]. We briefly describe each baseline below:

- **PMF:** uses a probabilistic approach that models user and service latent features with Gaussian distributions.

- **DeepCoNN:** uses two coupled deep neural networks to jointly model end-users and services from user reviews. A user latent feature is learned from user reviews written by the end-user and a service latent feature is learned from user reviews written for the service. The learned latent features are combined to perform

[6]https://www.tensorflow.org/

user rating predictions.

- **CDL:** uses a Bayesian based stacked denoising autoencoder (SDAE) to learn the service latent features, through the process of reconstructing service descriptions from noised inputs. The learned service latent features are integrated with collaborative filtering to predict user ratings.
- **CVAE:** is a Bayesian probabilistic model that employs variational autoencoder and collaborative filtering. Service latent features are learned from the variational autoencoder network by reconstructing service descriptions. Both CDL and CVAE interpret service descriptions as bag-of-words.

6.6 Case Study

We answer five research questions to evaluate the performance of our model. For each research question, we describe the motivation, approach and the obtained results.

RQ6.1. What are the best hyper-parameters for using the DeepCont model?

Motivation. Hyper-parameters are the variables that may decide the architecture of a model (*e.g.*, the number of convolutional layers) and the training process (*e.g.*, the batch size of training data entries). Hyper-parameters should be assigned before training a model [26]. Different settings of hyper-parameters could influence the performance of our proposed DeepCont model. In this research question, we study the effect of the settings of hyper-parameters on the performance of DeepCont.

Approach. We empirically examine the effects of five hyper-parameters of DeepCont: 1) the dimension of latent features k; 2) the number of filters m_l; 3) the number of layers L in the convolutional network and the transposed convolutional network; 4) the batch size M of training data entries; and 5) the text cut-off percentage p. We use the validation set for predicting user ratings of existing services to tune the hyper-parameters of our model. The dimension of latent features k is searched in four settings, *i.e.*, {16, 32, 64, 128}. The number of filters in the first layer of the convolutional network determines the number of filters in the consecutive layers. As an example, if the first convolutional layer has $m_1 = 16$ filters with $L = 3$ layers, the convolutional VAE network has the following filter sizes: $16 \rightarrow 32 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 1$. Thus, we test the number of filters m_1 among three numbers, *i.e.*, {4, 8, 16}. We tune the number of layers L using three setups, *i.e.*, {1, 2, 3}. We test the batch size M in four settings, *i.e.*, {32, 64, 128, 256}. Moreover, we search the text cut-off percentage p in five settings, *i.e.*, {70%, 75%, 80%, 85%, 90%}. For the other hyper-parameters, *i.e.*, the stride number s, the window size t and the learning rate of Adagrad, we use the grid search method [147] that tests all the combinations of these hyper-parameters on the validation set to find the best setting.

Results. The optimal hyper-parameter values remain the same in our evaluated datasets. Figure 6.7 shows the MSE values obtained on the validation sets of *Grocery and Gourmet Food* and *Musical Instruments* by varying the dimension of latent features k (Figure 6.2), the number of filters m_1 (Figure 6.3), the number of layers L (Figure 6.4), the batch size M (Figure 6.5) and the text cut-off percentage p (Figure 6.6).

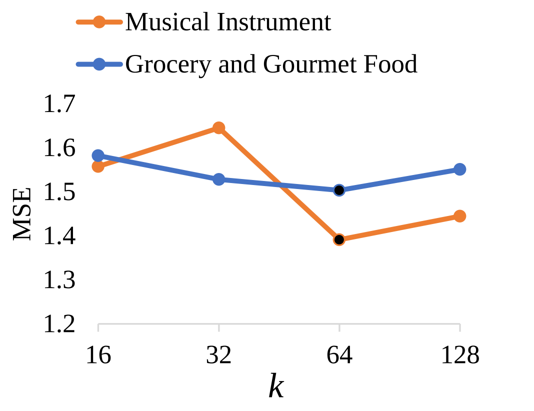


Figure 6.2: Dimension of latent features k

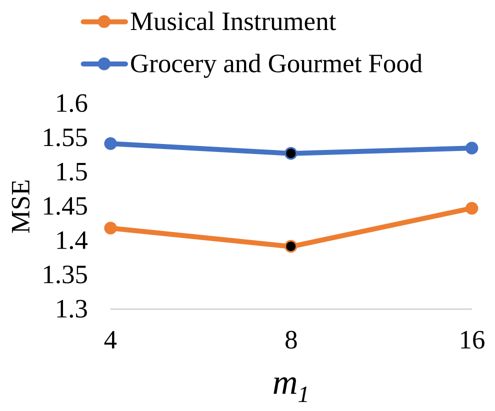


Figure 6.3: Number of filters m_1

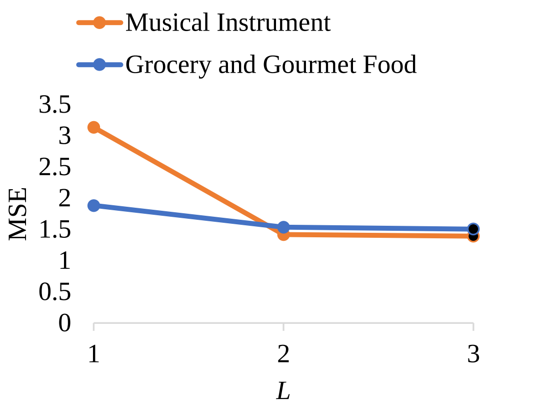


Figure 6.4: Number of layers L

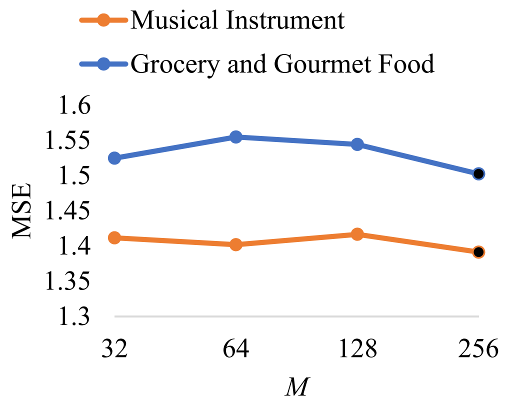


Figure 6.5: Batch size M

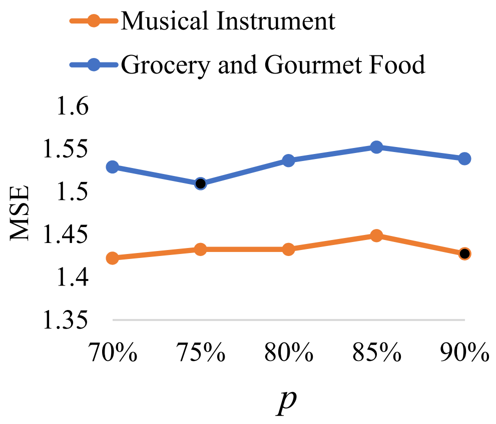


Figure 6.6: Text cut-off percentage p

Figure 6.7: Analysis of MSE with respect to different hyper-parameters (Black dots represent the optimal MSE values)

We empirically set k as 64, since the performance decreases with a lower k value and increasing k does not substantially improve the performance. Figure 6.3 shows that the two datasets obtain the best results when the number of filters in the first layer is set as 8. Figure 6.4 shows that the larger the number of layers, the better the performance. Further increasing the number of layers (*i.e.*, $L \geq 4$) decreases the dimension of the flatten layer, producing information loss from the input text. We set the number of layers as 3 for our experiments. m_1 and L have the effect on the number of parameters to be trained in DeepCont. Too many parameters could result in overfitting, while a small number of parameters may result in a lack of modelling capability to predict user ratings. Figure 6.5 shows that the batch size M does not have a significant influence on the performance. We set the batch size M as 256 because the two datasets obtain the best MSE values at this batch size. Finally, Figure 6.6 illustrates that the performance fluctuates as the increase of the text cut-off percentage p. We set the cut-off percentage p as 90%, since the *Musical Instruments* dataset achieves the best performance at 90% and the larger the cut-off percentage, the more information from user reviews and service descriptions could be kept. Through a grid search, we empirically set the stride number s as 2, the window size t as 3 and the learning rate as 0.001. We use the dropout ratio d as 0.5.

Our DeepCont model sets the dimension of latent features k=64, the number of filters m_1=8, the number of layers L=3, the batch size M=256 and the text cut-off percentage p=90%.

RQ6.2. Can our DeepCont model accurately predict user ratings of existing services?

Motivation. Our proposed DeepCont model is a deep learning based model that learns latent features from user reviews and service descriptions. Existing models have not integrated both of the user reviews and service descriptions. We evaluate the performance of our model for predicting user ratings of existing services, by comparing four baseline models (Section 6.5.3).

Approach. To fairly compare our model with the baseline models, we carefully tune the hyper-parameters of the baseline models for the task of predicting existing services.

For PMF, we use the grid search method to test the dimension of latent features k in the settings $\{10, 50, 100, 150, 200\}$ and the regularization parameter λ in the settings $\{0.001, 0.01, 0.1, 1.0\}$. PMF achieves the best validation performance when k=50 and λ=1.0. Thus, we use this setting to evaluate the performance of PMF on the testing set.

For DeepCoNN, we search the batch size in four settings, *i.e.*, $\{32, 64, 128, 256\}$ and the learning rate in six setups, *i.e.*, $\{0.001, 0.002, 0.005, 0.01, 0.02, 0.05\}$. The dimension of latent features k is tested among three numbers, *i.e.*, $\{32, 64, 128\}$. After the tuning process, we set the batch size as 256, the learning rate as 0.001 and the dimension of latent features k as 64. Consistent with the findings by Chen *et al.* [21], Adam [61] achieves the best validation result for optimizing the model. The other parameters are set similarly to the DeepCoNN model [165], *i.e.*, the number of filters and the window size are set as 100 and 3, respectively. Changing these parameters did not improve the validation performance.

The CDL model predicts user ratings as either 1 or 0. Thus, the precision parameter is set differently for the two rating values. Similar to the approach used by Ling *et al.* [81] and Zheng *et al.* [165], in our experiments, we set the precision parameter as 1 for the different observed integer user ratings. The CDL model is trained using user ratings and service descriptions. We set the dimension of latent features k=64 in CDL. We tune the regularization terms λ_u, λ_v, λ_n and λ_w. The best validation performance is achieved when the regularization terms are set as λ_u=0.1, λ_v=100, λ_n=10,000 and λ_w=0.0001. The masking noise is set with the noise level 0.3 to corrupt the input. We adopt a two-layer SDAE network architecture, similarly to the architecture used by Wang *et al.* [147].

For CVAE, we set the precision parameter as 1 and the dimension of latent features k=64, similarly to the CDL model. We test the regularization terms λ_u, λ_v and λ_r. We set λ_u=0.1, λ_v=100 and λ_r=1. The network architecture of CVAE is set to be the same as the one used by Li *et al.* [75]. To train the CVAE model, we firstly pre-train the weights of the variational autoencoder network using service descriptions. Then the weights are set as the initial points for the CVAE model for fine tuning.

Results. Our proposed DeepCont model achieves the best performance compared with the four baseline models. Table 6.2 shows the comparison of performance of our DeepCont model with baseline models on 21 datasets. Overall, our model achieves better MSE results on 17 out of 21 datasets. We use the Wilcoxon signed-rank test [123] to examine if the performance improvements are significant. The Wilcoxon test makes no assumption about the distribution of samples. Our results reveal that, except the performance comparison with CDL on the *Automotive* category, all the other improvements are significant at the 1% confidence level.

Compared with PMF, the DeepCont model improves the performance on 19 datasets with a margin ranging from 1.29% to 13.23%. PMF only uses rating information. In a sparse setting, where end-users only rate a small number of services, PMF could not capture user and service latent features effectively. Our model improves DeepCoNN in 20 datasets with an average margin of 6.26%. Although DeepCoNN is a deep learning based model and uses user reviews for predicting user ratings, we find that in some of the datasets, DeepCoNN achieves worse performances than PMF. As Catherine and Cohen [17] indicate, DeepCoNN achieves the best performance when the user review for a service at test time is available. However, in real-world settings, end-users do not provide user reviews before rating a service.

Our model significantly outperforms CDL in 17 out of 21 datasets with an average improvement of 3.08%, and significantly improves the performance of CVAE in 19 out of 21 datasets with an average improvement of 3.98%. CDL and CVAE propose two different models, *i.e.*, SDAE and VAE, to learn service latent features from service descriptions. Interestingly, the two models consistently achieve the same or better performances compared with the PMF model on the 21 datasets. It corroborates that learning service latent features from service descriptions can boost the performance of user rating predictions. However, CDL and CVAE fail to capture semantic meanings of service descriptions, since the descriptions are represented as bag-of-words. Moreover, the user latent features are learned from the collaborative filtering and user ratings in these models. Conversely, our model learns user latent features from user reviews that express user preferences and learns service latent features from service descriptions that describe various service properties. Therefore, we can achieve a better performance.

Table 6.2: MSE Comparison of our model with four baseline models on 21 datasets (Best results are marked as bold)

Datasets	PMF (a)	DeepCoNN (b)	CDL (c)	CVAE (d)	DeepCont (e)	Improvements (%)			
						e v.s. a	e v.s. b	e v.s. c	e v.s. d
Grocery and Gourmet Food	1.592	1.681	1.574	1.588	**1.535**	3.60%	8.66%	2.45%	3.32%
Clothing Shoes and Jewelry	1.530	1.702	1.508	**1.508**	1.539	-0.56%	9.61%	-2.02%	-2.04%
Home and Kitchen	1.779	1.861	**1.762**	1.776	1.782	-0.17%	4.22%	-1.14%	-0.38%
Kindle Store	1.284	1.217	**1.174**	1.201	1.182	7.92%	2.82%	-0.68%	1.52%
Sports and Outdoors	1.534	1.620	1.523	1.531	**1.511**	1.53%	6.76%	0.80%	1.30%
Cell Phones and Accessories	2.164	2.294	2.158	2.161	**2.136**	1.29%	6.87%	1.02%	1.17%
Health and Personal care	1.785	1.845	1.758	1.779	**1.679**	5.94%	9.00%	4.52%	5.61%
Toys and Games	1.642	1.596	1.601	1.633	**1.563**	4.85%	2.09%	2.39%	4.31%
Video Games	1.898	**1.741**	1.866	1.896	1.816	4.34%	-4.29%	2.70%	4.22%
Tools and Home Improvement	1.707	1.776	1.695	1.704	**1.681**	1.55%	5.39%	0.83%	1.34%
Beauty	1.717	1.840	1.692	1.714	**1.663**	3.17%	9.61%	1.70%	2.96%
Office Products	2.010	1.992	1.981	2.002	**1.895**	5.69%	4.85%	4.31%	5.35%
Pet Supplies	1.773	1.849	1.775	1.772	**1.640**	7.50%	11.28%	7.57%	7.40%
Automotive	1.645	1.711	1.638	1.641	**1.614**	1.86%	5.64%	1.48%	1.65%
Patio Lawn and Garden	1.958	1.925	1.935	1.953	**1.832**	6.44%	4.82%	5.34%	6.21%
Baby	1.660	1.643	1.684	1.657	**1.562**	5.92%	4.93%	7.28%	5.76%
Digital Music	0.923	0.939	0.886	0.898	**0.867**	6.09%	7.75%	2.18%	3.46%
Musical Instruments	1.449	1.464	1.453	1.440	**1.387**	4.24%	5.25%	4.50%	3.63%
Movies and TV	1.524	1.414	1.481	1.477	**1.323**	13.23%	6.48%	10.70%	10.46%
CDs and Vinyl	1.113	1.139	1.100	1.076	**1.015**	8.77%	10.84%	7.72%	5.59%
Electronics	1.909	2.056	1.894	1.903	**1.874**	1.81%	8.84%	1.03%	1.49%
Average	1.647	1.681	1.626	1.634	**1.576**	5.05%	6.26%	3.08%	3.98%

On average, our proposed DeepCont model improves the four baselines, i.e., *PMF, DeepCoNN, CDL and CVAE by a margin of 5.05%, 6.26%, 3.08% and 3.98%, respectively.*

RQ6.3. Is the use of the word embedding layer and the VAE network beneficial to our DeepCont model?

Motivation. The word embedding layer converts user reviews or service descriptions to word embedding matrices. The word embedding matrix captures the order of the words and the semantic meanings. The variational autoencoder network takes the word embedding matrices as inputs to learn user and service latent features. We are interested to evaluate if the word embedding layer and the VAE network are essential components of our DeepCont model.

Approach. We compare the performance of our DeepCont model with four variants of the model. The four variants are *DeepCont-Random, DeepCont-Train, DeepCont-CNN* and *DeepCont-Autoencoder*. The first two variant models (*i.e.*, *DeepCont-Random* and *DeepCont-Train*) are used to evaluate the importance of the word embedding layer, while the other two variants (*i.e.*, *DeepCont-CNN* and *DeepCont-Autoencoder*) are used to evaluate the importance of the VAE network. We describe the four variant models below.

- **DeepCont-Random:** Different from our model, the *DeepCont-Random* model randomly initializes the word vectors in the word embedding layer (see Section 6.4.1) with a uniform distribution.

- **DeepCont-Train:** Instead of fixing the word vectors in the word embedding layer that is used in our model, the *DeepCont-Train* model fine-tunes the word vectors in the training process.

- **DeepCont-CNN:** To evaluate if the VAE network improves the performance of our DeepCont model, the *DeepCont-CNN* model is constructed without the decoder network, *i.e.*, the transposed convolutional network (see Figure 6.1). Specifically, the latent representation layer in the *DeepCont-CNN* model consists of a flatten layer (described in Section 6.4.1) and a fully connected layer. The fully connected layer estimates the user latent feature g_u (or service latent feature y_i) from the output O_L, using Equation 6.4. Finally, the prediction layer combines the two latent features, *i.e.*, g_u and y_i, to generate the predicted user rating $\hat{r}_{ui}$, using the approach described in Section 6.4.1.

- **DeepCont-Autoencoder:** Instead of using the VAE network to estimate a latent representation z_i as a variational distribution (see Section 6.3), the *DeepCont-Autoencoder* model uses a traditional autoencoder network to estimate a latent representation z_i as a single vector (*i.e.*, a point). Specifically, the latent representation layer of the *DeepCont-Autoencoder* model consists of a flatten layer and a fully connected layer. The fully connected layer estimates the latent representation z_i (*i.e.*, ϵ_u for Net_u and τ_i for Net_i) from the output O_L of the convolutional network, using Equation 6.4. The prediction layer takes the latent representation ϵ_u as the user latent feature g_u, and takes the latent representation τ_i as the service latent feature y_i. The two latent features (*i.e.*, g_u and y_i) are combined to generate the predicted user rating $\hat{r}_{ui}$ (described in Section 6.4.1).

Table 6.3: MSE Comparison of our model with four variants of our model on 21 datasets

Datasets	**DeepCont -Random (a)**	**DeepCont -Train (b)**	**DeepCont -CNN (c)**	**DeepCont -Autoencoder (d)**	**DeepCont (e)**	**Improvements (%)**			
						e v.s. a	**e v.s. b**	**e v.s. c**	**e v.s. d**
Grocery and Gourmet Food	1.565	1.558	1.733	1.579	**1.535**	1.89%	1.48%	11.42%	2.77%
Clothing Shoes and Jewelry	1.581	**1.527**	1.565	1.570	1.539	2.69%	-0.79%	1.65%	1.99%
Home and Kitchen	1.853	**1.717**	1.762	1.767	1.782	3.79%	-3.78%	-1.13%	-0.89%
Kindle Store	1.217	**1.177**	1.270	1.281	1.182	2.80%	-0.46%	6.90%	7.70%
Sports and Outdoors	1.544	1.587	1.522	1.541	**1.511**	2.16%	4.83%	0.72%	1.99%
Cell Phones and Accessories	2.346	2.177	2.180	2.342	**2.136**	8.93%	1.88%	2.00%	8.79%
Health and Personal care	1.716	1.748	1.770	1.733	**1.679**	2.19%	3.94%	5.13%	3.15%
Toys and Games	1.587	1.589	1.621	1.670	**1.563**	1.51%	1.64%	3.61%	6.46%
Video Games	**1.722**	1.820	1.903	1.877	1.816	-5.46%	0.26%	4.61%	3.28%
Tools and Home Improvement	1.730	1.708	1.681	1.755	**1.681**	2.84%	1.58%	0.02%	4.22%
Beauty	1.696	1.684	1.694	1.703	**1.663**	1.97%	1.26%	1.86%	2.33%
Office Products	**1.853**	1.962	2.000	1.990	1.895	-2.28%	3.42%	5.26%	4.77%
Pet Supplies	1.698	1.689	1.740	1.803	**1.640**	3.39%	2.88%	5.72%	9.04%
Automotive	1.705	1.652	1.636	1.627	**1.614**	5.31%	2.32%	1.32%	0.78%
Patio Lawn and Garden	1.897	1.849	1.892	1.927	**1.832**	3.41%	0.91%	3.19%	4.92%
Baby	1.658	1.699	1.634	1.651	**1.562**	5.82%	8.11%	4.41%	5.43%
Digital Music	0.876	**0.863**	0.902	0.906	0.867	1.11%	-0.37%	3.94%	4.32%
Musical Instruments	1.441	**1.382**	1.421	1.444	1.387	3.69%	-0.39%	2.34%	3.91%
Movies and TV	1.361	**1.320**	1.450	1.522	1.323	2.81%	-0.18%	8.80%	13.11%
CDs and Vinyl	1.057	1.016	1.257	1.116	**1.015**	3.94%	0.10%	19.25%	9.02%
Electronics	1.928	2.034	1.932	2.011	**1.874**	2.78%	7.87%	3.00%	6.81%
Average	1.620	1.608	1.646	1.658	**1.576**	2.63%	1.74%	4.70%	4.95%

Results. The word embedding layer and the VAE network are essential components of our DeepCont model. Table 6.3 shows the performance comparison of our DeepCont model with four variants of our model. Overall, our model achieves the best average MSE result (*i.e.*, MSE=1.576). Compared with *DeepCont-Random*, the Wilcoxon signed-rank test shows that our model significantly improves 19 out of 21 datasets. *DeepCont-Random* randomly initializes word vectors, thus losing semantic meanings in user reviews and service descriptions. The performance improvement of our model shows that our word embedding layer can effectively represent user reviews and service descriptions. Compared with *DeepCont-Train*, our model slightly improves the performance by 1.74%. Though fine-tuning word vectors may result in overfitting, *DeepCont-Train* initializes word vectors with semantic meanings. The small performance difference indicates that the word embedding layer should be initiated with word vectors that embed semantic meanings.

Our model significantly improves the performance of *DeepCont-CNN* on 19 datasets (except the *Home and Kitchen* and the *Baby* datasets), and improves *DeepCont-Autoencoder* on 20 datasets (except the *Home and Kitchen* dataset). *DeepCont-CNN* trains user and service latent features by generating predicted user ratings, while our model not only uses user ratings, but also the VAE network to train the latent features. Compared with the *DeepCont-Autoencoder* model that estimates a latent representation as a single point, the VAE network in our model samples latent representations from a region of a space (see Section 6.3). Through the process that reconstructs an input from every sampled latent representation, the learned user or service latent feature could capture diverse user preferences and service properties. Thus, the VAE network has a vital importance in our model for learning user and service latent features.

> *On average, our model outperforms the four variant models,* i.e., *DeepCont-Random, DeepCont-Train, DeepCont-CNN and DeepCont-Autoencoder by a margin of 2.63%, 1.74%, 4.70% and 4.95%, respectively.*

RQ6.4. Can our DeepCont model accurately predict user ratings of new services?

Motivation. A recommendation system should not only recommend user ratings for existing services, but also for newly added services. Since new services do not have any user reviews, traditional models such as PMF and DeepCoNN cannot learn representative service latent features. However, the service descriptions are still available for new services. We are interested to evaluate the performance of our model for predicting user ratings of new services.

Approach. We use the 21 datasets generated for predicting user ratings of new services (see Section 6.5.1) to evaluate the performance. We compare the performance of our DeepCont model against the DeepCoNN, CDL and CVAE models. We do not include the PMF model in our comparisons because the PMF model cannot perform user rating predictions for new services [146]. The hyper-parameters of the compared models are set as the same hyper-parameters used in RQ6.2. We apply our model and the baseline models on the testing set to compare the performances.

Results. Generally, we observe that the four models achieve worse performances for predicting user ratings of new services when compared to predicting user ratings of existing services (*i.e.*, RQ6.2). On average, DeepCoNN, CDL, CVAE and our proposed DeepCont model get MSE values of 1.690,

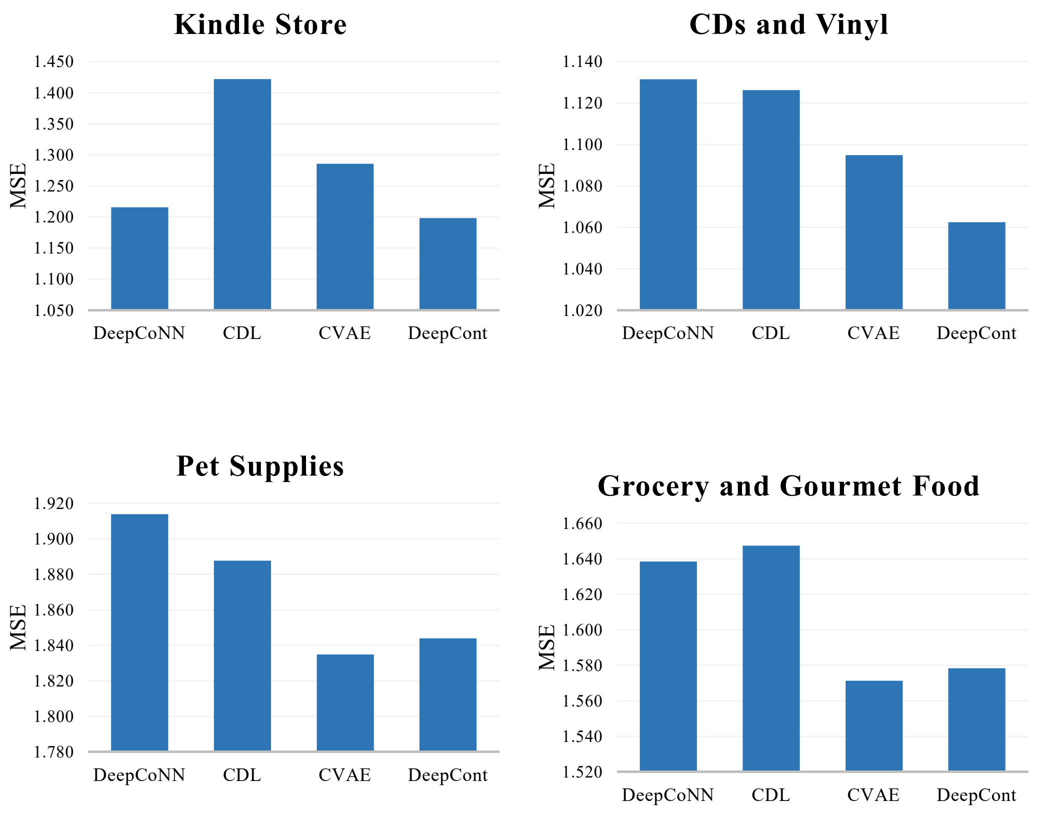


Figure 6.8: MSE results for predicting new services

1.680, 1.627 and 1.656 on the 21 datasets, respectively. Overall, our model performs better than the DeepCoNN and CDL models, and obtains a higher MSE value as compared to CVAE. Figure 6.8 shows the MSE values obtained on four of our datasets. In Figure 6.8, our model has the best performance among the three baselines in the category *Kindle Store* and *CDs and Vinyl*.

Compared to DeepCoNN, our model significantly improves performances in 17 datasets. As new services have no user reviews, the service latent features are represented by the learned bias terms of the DeepCoNN model. These bias terms cannot represent different new services, thus DeepCoNN obtains the worst performance. Instead of using user reviews for modeling services, using service descriptions can improve the performance of user rating predictions. Specifically, CDL performs slightly better than DeepCoNN, since CDL learns latent representations for new services from service descriptions. CDL learns a latent representation as a single vector (*i.e.*, a point). Differently from CDL, our model learns probabilistic representations for new services. The learned probabilistic representations can better represent new services. Our model significantly outperforms the CDL model in 15 out of 21 datasets.

In the CVAE model, the performance for predicting user ratings of new services (*i.e.*, MSE=1.634) is similar to the performance for predicting user ratings of existing services (*i.e.*, MSE=1.627). Similar to our model, CVAE learns probabilistic representations for new services and use the mean of the representations as service latent features. In CVAE, the user rating information does not guide the learning of the latent features for new services. Differently from CVAE, in our model, the service latent features for new services tend to include user ratings and service information of existing services that are trained in the training phase, which may incur noise for representing new services. Nonetheless, compared with CVAE, our model performs significantly better or equally well in 11 datasets. Our model achieves similar performance compared with CVAE. Moreover, our model performs the best for predicting user ratings of existing services (see RQ6.2).

To predict user ratings for new services, our DeepCont model outperforms the two baselines, i.e., *DeepCoNN and CDL, and achieves similar performance compared with CVAE.*

RQ6.5. Do similar services have similar service latent features?

Motivation. Until now, we mainly focus on using the MSE of predicted user ratings to evaluate the effectiveness of our model. To demonstrate the effectiveness of the learned latent features in our model, we are interested to investigate whether the latent features integrate the semantic meanings in the text (*i.e.*, user reviews and service descriptions). Specifically, we examine whether similar services have similar latent features or not. The similarity of services is decided based on the similarity of service descriptions. Since it is difficult to determine the similarity of users based on user reviews, we focus on investigating service latent features.

Approach. To determine the similarity of services, we use the *doc2vec* model [72], an unsupervised model, to learn a fixed-length numerical representation of a service description con_i. The numerical representation is trained to infer words in a service description. Therefore, the learned numerical representation captures the semantic meaning of a service description. The similarity of two services i and j is computed as the cosine similarity between the two numerical representations that are learned from the corresponding service descriptions con_i and con_j. The cosine similarity measure is commonly used in the *doc2vec* model to compute similarities of the learned numerical representations [71]. To determine the similarity between the learned service latent features of services i and j in our DeepCont model, we calculate the Euclidean distance of the learned service latent features, similarly to the measure used by Catherine and Cohen [17]. A lower Euclidean distance represents a higher latent feature similarity. We use the Euclidean distance instead of the cosine similarity to measure the latent feature similarities, because an end-user could give different user ratings for two services that have a high cosine similarity value between the corresponding service latent features. In contrast, two services that have low Euclidean distance between the respective service latent features could receive similar user ratings from the same end-user.

To examine whether similar services have similar service latent features or not, we use the Fisher's exact test [163]. In particular, we randomly sample 384 services from each category of the dataset, with 95% confidence level and 5% confidence interval. For a sampled service i, we use the *doc2vec* model to compute the service similarity values between the service i with all the other services in the dataset, and get a ranked list R_i by ranking the service similarity values. We separate services in R_i into two groups along a threshold value. We choose the median similarity value as the threshold. The two separated groups are: 1) the control group that contains services with lower similarity values (*i.e.*, less than the threshold); and 2) the experimental group that contains the remaining services with higher similarity values. Similarly, we compute the Euclidean distance between the service latent feature of the service i with the service latent features of all the other services, and get a ranked list L_i by ranking the Euclidean distances. We split services in L_i into two groups along the median distance value: 1) services with higher latent feature similarities; and 2) services with lower latent feature similarities. We test the following null hypothesis $H0_1$ for the service i.

$H0_1$*: the proportion of services with higher latent feature similarities in the control group and the experimental group has no difference.* We evaluate the hypothesis using the Fisher's exact test with 99% confidence level (*i.e.*, p-value < 0.01). The Fisher's exact test examines if there exists non-random associations between the service similarities and the latent feature similarities. We further compute the odds ratio (denoted as OR) [123], which measures the likelihood that services in the experimental group (*i.e.*, services with higher similarity values) have higher latent feature similarities with the service i. Particularly, if $OR > 1$, services with higher similarity values are more likely to have higher latent feature similarities.

Results. Similar services are more likely to have similar service latent features. Our results of the Fisher's exact test on 21 datasets show that, on average, a majority of services (*i.e.*, 56%) have significant associations between service similarities and latent feature similarities. Specifically, services that have higher similarity values with the 56% of services are 2.24 times more likely to have higher latent feature similarities. On average, 18% of services have significant Fisher's exact test result and have odds ratio lower than 1. The other 26% of services do not show significant associations between service similarities and latent feature similarities.

As shown in Figure 6.1, the service latent features are generated from service descriptions and guided by the user rating information. Thus, services with similar service descriptions do not necessarily have similar service latent features, since the received user ratings of the services may be different. However, the result that similar services are more likely to have similar service latent features demonstrate that our model incorporates service content descriptions to learn service latent features.

56% of services in our dataset show significant associations between service similarities and latent feature similarities.

6.7 Chapter Summary

User reviews typically explain the reasons behind user ratings. Moreover, service descriptions describe various properties of services. In this chapter, we propose a model, named DeepCont, to predict user ratings on services by jointly using user reviews and service descriptions. The proposed DeepCont model contains two parallel convolutional variational autoencoder networks (VAE) to learn end-users' preferences from user reviews and service properties from service descriptions. The two convolutional VAE networks are connected by a prediction layer to effectively predict user ratings. We conduct extensive experiments on 21 real world datasets to evaluate the performance of our DeepCont model with two tasks, *i.e.*, 1) predicting user ratings of existing services and 2) predicting user ratings of new services. Our results are summarized as follows.

- **Predicting user ratings of existing services.** Compared with the state-of-the-art baseline models, *i.e.*, PMF, DeepCoNN, CDL and CVAE, our model achieves the best user rating prediction accuracy. Specifically, our model improves the best baseline model (*i.e.*, CDL) by a margin of 3.08%.

- **Predicting user ratings of new services.** Our model ourperforms the two baseline models, *i.e.*, DeepCoNN and CDL, and achieves similar performance as compared to CVAE. Our model alleviates the cold start problem, where new services have not received any user reviews.

By comparing variants of our DeepCont models, our results show that the word embedding layer captures semantic meanings and the convolutional VAE network is crucial for our model to learn representative user and service latent features. The result that similar services have similar service latent features demonstrates that our model incorporates service descriptions to learn latent features.

Chapter 7

Automatically Learning User Preferences for Personalized Service Composition

Usually, a single service is not sufficient to fulfill an end-user's goal. End-users may perform multiple services for their re-occurring activities. In this chapter, we propose an automated service composition approach that recommends a collection of services to satisfy end-users' preferences. Our approach learns user preferences from end-users' past activities. The learned user preferences are integrated into our multi-objective reinforcement learning (MORL) algorithm to recommend services that achieve the highest objectives of end-users.

Chapter Organization. Section 7.1 describes the introduction of this chapter.

Section 7.2 illustrates our proposed approach to compose personalized services. Section 7.3 introduces our case study setup and the results of our case study. Finally, Section 7.4 concludes this chapter.

7.1 Introduction

The massive amount of services enables end-users to conduct various daily activities, such as on-line shopping. Generally speaking, a single service can not satisfy an end-user's goal. End-users often need to visit multiple services to fulfill their goals. For instance, to buy a pair of boots, an end-user may browse various E-commerce websites to compare the products and check user reviews of the products. Essentially, the end-user implicitly composes services for purchasing boots. However, an end-user faces numerous choices when selecting services. Various service information, such as price and user rating, can affect end-users' decisions of selecting services for achieving end-users' goals. Therefore, it becomes tedious and cumbersome tasks for end-users to discover and compose services. To reduce end-users' cognitive burden, it is critical to support automated service composition by efficiently recommending personalized services to achieve end-users' overall goals. However, in the current practices, end-users face the following challenges:

- *Limited and rigid options available for end-users to specify the personalized preferences.* Many service providers allow end-users to enter searching requirements for services to meet user preferences. For example, an end-user can specify the price range and brand names to search for boots on the Amazon[1] website. However, the number of pre-defined options is limited and such options are designed

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

to address the interests of a group of end-users. The pre-defined options can not be customized for the individuals' needs.

- *Conflicting preferences.* An end-user might have several preferences in selecting services. However, such preferences can be contradictory. For instance, end-users who have a low budget for shopping for clothes may prefer boots with a lower price but a high user rating. However, a low price service may likely receive a low user rating. Therefore, an end-user needs to make trade-offs among the conflicting preferences before choosing a service.

- *No prioritization of various preferences.* An end-user can have a priority order among several preferences. For example, an end-user may prefer high user ratings than low prices. However, the priority order of preferences is not taken into account in the existing approaches [98][158] for the service recommendation.

To address the aforementioned challenges, the multi-objective reinforcement learning algorithm (MORL) [133] is used to solve the conflicting preferences in service composition [98][148][149]. Existing approaches, such as [148][149], use the MORL to model the user preferences for services as numeric weight values, and require end-users to manually assign the weight values. However, end-users can be reluctant to enter any additional information, since the meaning of the weight values is opaque for them.

In this Chapter, we propose an approach for personalized service composition. Our approach does not require end-users to manually specify preferences. Instead, user preferences and the priority orders of the preferences are automatically inferred from past service selection history. We automatically generate weight values for the MORL algorithm to compose services based on the identified user preferences. As a

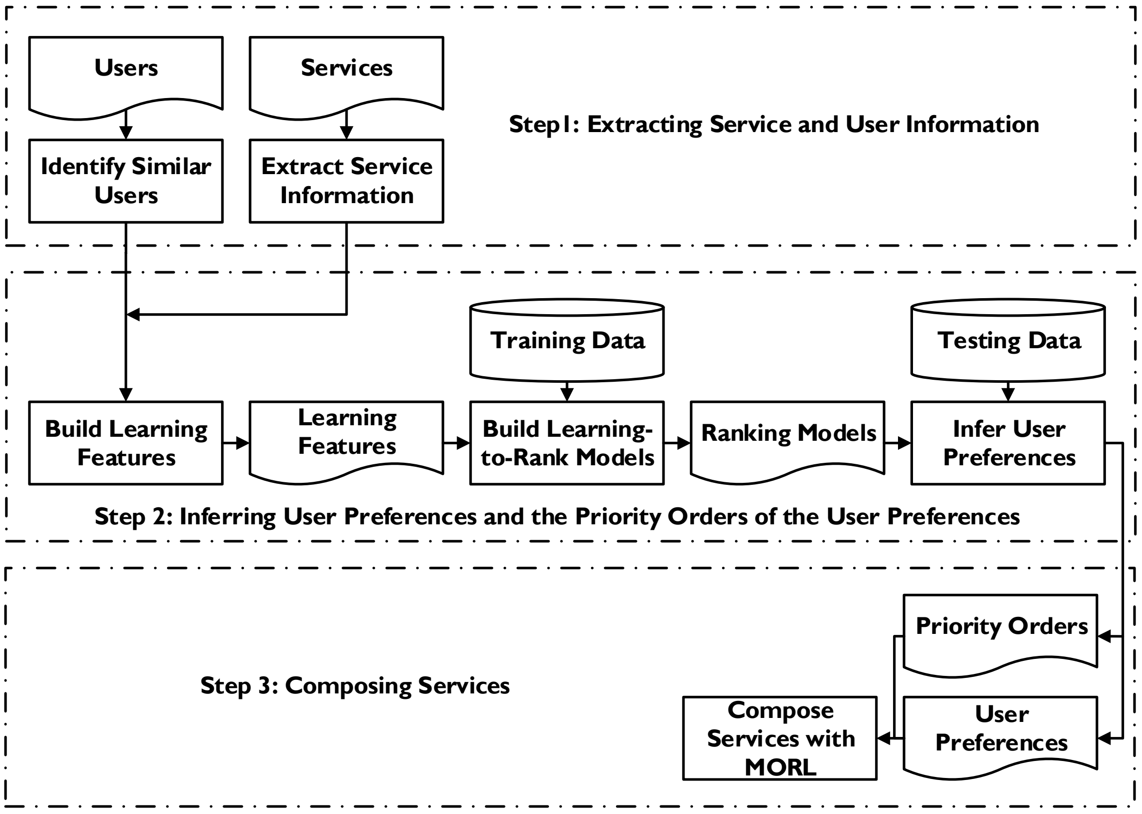

Users
Services
Identify Similar Users
Extract Service Information
Step1: Extracting Service and User Information
Training Data
Testing Data
Build Learning Features
Learning Features
Build Learning-to-Rank Models
Ranking Models
Infer User Preferences
Step 2: Inferring User Preferences and the Priority Orders of the User Preferences
Priority Orders
Step 3: Composing Services
Compose Services with MORL
User Preferences

Figure 7.1: An overview of our approach. MORL refers to the the multi-objective reinforcement learning algorithm.

consequence, our approach can derive personalized composite services with minimal effort from end-users. In particular, our approach applies a machine learning algorithm, *i.e.*, a learning-to-rank algorithm, namely RankBoost [40], in order to identify the user preferences. We extract various learning features from service and end-user information to build ranking models that can infer the prioritization of user preferences. The preference learning is conducted offline to ensure the minimal time delay for the real-time service recommendation. We conduct case studies on real dataset from end-users. Comparing with the two well-established MORL baselines, our empirical results show that our approach can outperform the MORL baselines by 100%

to 200% in terms of precision.

7.2 Overview of Our Approach

To save end-users from manually specifying their preferences to compose services, our approach automatically infers user preferences utilizing a learning-to-rank (LtR) algorithm, RankBoost. Figure 7.1 gives an overview of our approach. Our approach consists of three main steps: 1) we extract service and end-user information from end-users' service composition history; 2) We identify learning features from the extracted service and end-user information to build ranking models. We use the built ranking models to infer user preferences; and 3) We apply the MORL algorithm to optimize the service composition by incorporating the identified user preferences into the service recommendation process. In the following subsections, we discuss each step in more details.

7.2.1 Extracting Service and End-User Information

We extract service and end-user information to build learning features to create a ranking model for various user preferences. An end-user can perform a small amount of services. The data related to services selected can be limited. The sparse data can make it hard to infer user preferences. Moreover, similar end-users sharing the same interests with an end-user can have the same preferences [88][131]. In our approach, we leverage the similar end-users' service selection history to infer the end-user's preferences. We examine the following information from various end-users.

Service Information

includes task description, service description, service context and end-user choices.

- **Task Description** states the intent of an end-user's activity. Usually, a service is associated with the textual description of a task (*e.g.*, buy men's chukka boots). The task description can be provided by an end-user when an end-user specifies search criteria to describe the intents. Otherwise, it can be retrieved from the task description as specified in the business process definitions.

- **Service Description** specifies the functionality of a service. It can be retrieved from the description of operations in Web Services Description Language (*i.e.*, WSDL) and web pages describing RESTful services or resources. Figure 7.2 shows an example of the associated task description and service description.

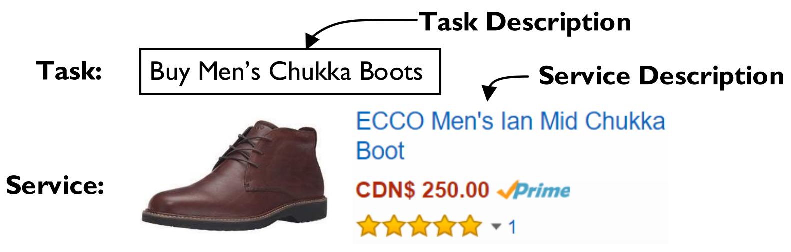


Figure 7.2: An example of task description and service description.

- **Service Context** describes the circumstances that a service interacts with an end-user. The context includes *time* and *location*. Specifically, the time is when an end-user performs a task and can be obtained from the operating system. The location can be acquired from the IP address of the computer or the GPS in a mobile device.

- **End-User Choices** record the selection of services by an end-user. The value for a choice can be 0 or 1. 1 means that the end-user selects the service and 0 denotes the service is not selected by the end-user. End-user choices can be mined from various sources, such as clicks on a URL link or invocation of a service.

Identifying Similar End-Users

To utilize similar end-users' service selection history, we need to measure the similarity between end-users. Our approach identifies the end-user similarity by leveraging end-users' personal information, such as hobbies and education background. The end-user information can be extracted from the existing profiles of end-users created in social networks (*e.g.*, Facebook[2]). We store end-user information in a key-value pair form. A key defines a label for end-user information (*e.g.*, hobby) and a value describes the contents (*e.g.*, swimming). We construct a bag of words (denoted as *bag*) for an end-user using the contents of the end-user information. We calculate the similarity of two end-users by comparing the differences of the corresponding two *bags*. A *bag* is normalized using the approach proposed by Zhao *et al.* [160]. We decompose words using special characters, such as "@", and capital cases if applicable. Suffixes containing numbers are removed. Non-English words are removed if the words are not contained in WordNet [94]. Moreover, we remove stop words (such as "a", "is" and "the") and perform word stemming ("reduced", "reducing" and "reduces" are normalized to "reduce") [103]. Then the end-user similarity can be calculated using

[2]www.facebook.com

Jaccard index [89] between the two bags of words:

$$Sim(U_i, U_j) = \frac{|bag_i \cap bag_j|}{|bag_i \cup bag_j|} \tag{7.1}$$

where $|bag_i \cap bag_j|$ denotes the number of common words in the two bags. $|bag_i \cup bag_j|$ states the number of words in the union sets of the two bags.

7.2.2 Inferring User Preferences and the Priority Orders of the User Preferences

End-users can have multiple preferences when they compose services (*e.g.*, price and user rating). The priority order of the preferences exists for end-users. To save end-users from manually specifying their preferences, our approach automatically infers user preferences and the priority orders.

Building Learning Features

We utilize a learning-to-rank (LtR) algorithm, named RankBoost [40], to build ranking models based on a set of learning features. A ranking model learns from different learning features to suggest an ideal ranking of services. Each learning feature describes a characteristic regarding a service and an end-user. We build the learning features from service and end-user information described in Section 7.2.1. To build a ranking model, the LtR needs training and testing phases. The training dataset contains tasks and services. Given a task t^i, there are a number of services $S^i = \{s_1^i, s_2^i, ..., s_n^i\}$ associated with the task for performing the task, where s_j^i denotes the j^{th} service. A task-service pair (t^i, s_j^i) contains a relevance judgment representing if an end-user chooses the service s_j^i respect to the task t^i. We take the *end-user* *choices* for the services as the relevance judgment. A learning feature vector $\mathbf{X}_j^i$ is constructed for each task-service pair (t^i, s_j^i). $\mathbf{X}_j^i$ contains a set of learning features to describe the characteristics of the task-service pair. The training process aims to train the ranking model $f(\mathbf{X})$ that can assign a score to a given task and a respective service. The top ranked services have higher possibility that end-users choose.

We propose four categories of learning features as listed in Table 7.1. We briefly discuss them as follows.

Table 7.1: Twenty learning features. Learning features marked with * indicate unique features based on our dataset.

Category	Features	Description
Service Attributes	(F1) Price*	The price of using the service S_k
	(F2) Discount*	The discount of using the service S_k
	(F3) User Rating*	The user rating of the service S_k
	(F4) Relevance*	To what extent that the service S_k is related to the task T_a
Service Usage History	(F5) Service Usage Frequency	The frequency of the service S_k used by the end-user U_i in the past for T_a
	(F6) Service Usage Frequency by the Similar End-Users	The frequency of the service S_k used by n similar end-users of the end-user U_i in the past for T_a
	(F7) Service Composition Pattern*	The frequency of two services S_g and S_k occurring together in the past by the end-user U_i
	(F8) Service Composition Pattern by the Similar End-Users*	The frequency of two services S_g and S_k occurring together in the past by n similar end-users of the end-user U_i
User Context	(F9) Usage of the Day	The frequency of the service S_k used by the end-user U_i in the past for T_a under the day D_t
	(F10) Usage of the Hour	The frequency of the service S_k used by the end-user U_i in the past for T_a under the hour H_q
	(F11) Usage of the Place	The frequency of the service S_k used by the end-user U_i in the past for T_a under the place P_x
	(F12) Usage of the Day by the Similar End-Users*	The frequency of the service S_k used by n similar end-users of the end-user U_i under the day D_t

Category	Features	Description
User Context	(F13) Usage of the Hour by the Similar End-Users*	The frequency of the service S_k used by n similar end-users of the end-user U_i under the hour H_q
	(F14) Usage of the Place by the Similar End-Users	The frequency of the service S_k used by n similar end-users of the end-user U_i under the place P_x
Textual Feature	(F15) Word Usage	The frequency of the words in the S_k used in the past by the end-user U_i
	(F16) Average Word Usage	The average number of times that the words in the S_k used in the past by the end-user U_i
	(F17) Word Usage by the Similar End-Users	The frequency of the words in the S_k used in the past by n similar end-users of the end-user U_i
	(F18) Average Word Usage by the Similar End-Users	The average number of times that the words in the S_k used in the past by n similar end-users of the end-user U_i
	(F19) Task Similarity*	The textual similarity between the task description T_a and the service description S_k
	(F20) Brand*	Whether the service S_k is provided by brand-name corporations

Service Attributes. Each service has a set of attributes. Such attributes serve as indicators for end-users to select services [158]. We can obtain service attributes from web pages of service providers and the runtime monitoring information [141]. Our approach identifies four types of service attributes: *price*, *user rating*, *discount* and *relevance* of a service to a task. The details of each learning feature are described in Table 7.1. Specifically, the *discount* learning feature is the original price of using the service S_k divided by the current price of using the service S_k. The *relevance* learning feature is the returned page number of the service S_k for the task T_a. For example, end-users can search services (*i.e.*, shopping items) for a desired task T_a (*i.e.*, a category of services) on the Amazon website[3]. The returned page number of a service S_k is the relevance of the service S_k to the task T_a.

Service Usage History. The history of end-users' service composition tracks the interests of end-users over time [131]. The past service usage is leveraged to recommend services for end-users. The category of service usage history includes the following learning features:

- *Service Usage Frequency* and *Service Usage Frequency by the Similar End-Users.* are calculated using Equation 7.2 and 7.3, respectively. Service frequency denotes how frequently a service is used in the past. The more frequently a service is used previously by an end-user U_i or by the similar end-users in the past, the higher chance that the service can be used again by the end-user U_i.

  $$Freq(S_k, U_i) = \#\, of\ times\ S_k\ used\ for\ T_a\ by\ U_i \tag{7.2}$$

  where S_k is a service; T_a is a task; and U_i is an end-user (same for other Equations).

  $$Freq\_SimUser(S_k, U_i) = \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Freq(S_k, U_j) \tag{7.3}$$

  where $Sim(U_i, U_j)$ is the end-user similarity calculated using Equation 7.1; $Freq(S_k, U_j)$ is the service usage frequency calculated using Equaiton 7.2.

- *Service Composition Patterns* and *Service Composition Patterns by the Similar*

[3]www.amazon.com

End-Users. are shown in Equation 7.4 and 7.5, respectively. Service composition patterns capture re-occurrences of a set of services that are frequently composed together. The service composition patterns can be re-used by an end-user [118]. Moreover, similar end-users can have the similar or same composition behaviors.

$$Pat(S_g, S_k, U_i) = \#\ of\ times\ S_g\ and\ S_k\ used\ together\ by\ U_i \qquad (7.4)$$

$$Pat\_SimUser(S_g, S_k, U_i) = \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Pat(S_g, S_k, U_j) \qquad (7.5)$$

where $Pat(S_g, S_k, U_i)$ *is the service composition pattern calculated using Equaiton 7.4.*

User Contexts associated with the past selection of services can be used for ranking services [151]. For instance, the end-user could choose the hotel frequently booked for business trips for their own personal trips. In our work, the user contexts include the *usage of the day* (*i.e.*, *Day*), the *usage of the hour* (*i.e.*, *Hour*) and the *usage of the place* (*i.e.*, *Place*). We use the approach proposed by Lee *et al.* [74] to abstract numerical context values to categorical values. For example, the *Day* (*i.e.*, D_t) has seven values, *i.e.*, from Monday to Sunday in a week. The *Hour* (*i.e.*, H_q) is categorized into four levels, *i.e.*, morning, noon, afternoon and evening. The number of categories of the *Place* (*i.e.*, P_x) depends on the number of places that end-users have been to. Equation 7.6 shows the calculation of the user contexts that include the *usage of the day* (*i.e.*, *Day*), the *usage of the hour* (*i.e.*, *Hour*) and the *usage of the place* (*i.e.*, *Place*). The context of using services by the similar end-users is also included, since the similar end-users can use services in the same context. Equation 7.7 calculates the *usage of the day by the similar end-users* (*i.e.*, $Day\_SimUser$), *usage of the hour* *by the similar end-users* (*i.e.*, $Hour\_SimUser$) and *usage of the place by the similar end-users* (*i.e.*, $Place\_SimUser$).

$$\begin{aligned} Day(D_t, S_k, U_i) &= \#\ of\ times\ S_k\ used\ for\ T_a\ by\ U_i\ under\ D_t \\ Hour(H_q, S_k, U_i) &= \#\ of\ times\ S_k\ used\ for\ T_a\ by\ U_i\ under\ H_q \\ Place(P_x, S_k, U_i) &= \#\ of\ times\ S_k\ used\ for\ T_a\ by\ U_i\ under\ P_x \end{aligned} \tag{7.6}$$

$$\begin{aligned} Day\_SimUser(D_t, S_k, U_i) &= \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Day(D_t, S_k, U_j) \\ Hour\_SimUser(H_q, S_k, U_i) &= \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Hour(H_q, S_k, U_j) \\ Place\_SimUser(P_x, S_k, U_i) &= \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Place(P_x, S_k, U_j) \end{aligned} \tag{7.7}$$

where $Day(D_t, S_k, U_i)$, $Hour(H_q, S_k, U_i)$ *and* $Place(P_x, S_k, U_i)$ *are the usage of the day, usage of the hour and usage of the place, respectively, calculated using Equation 7.7.*

Textual Feature is related to the textual description of services and tasks. We leverage the textual description to recommend services for end-users. To calculate the textual feature, we normalize the task description of T_a and the service description of S_k into two bags of words, denoted as Bag_T_a and Bag_S_k, respectively. We use term frequency and inverse term frequency ($tf - idf$) [89] to weight each word in the two bags. The category has the following six features.

- *Word Usage* and *Average Word Usage* are calculated using Equation 7.8 and 7.9, respectively. Sugiyama *et al.* [131] find that adapting service search results based on the word usage can achieve better accuracy for retrieving services. Service description containing more frequently used words can attract more attention from end-users.

$$Word\_Usage(S_k, U_i) = \sum_{p=1}^{m} \omega_{T_p}^{U_i} \tag{7.8}$$

where $\omega_{T_p}^{U_i}$ is the frequency of the word T_p used previously by the end-user U_i.

$$Ave\_Word\_Usage = Word\_Usage(S_k, U_i)/m_k; \tag{7.9}$$

where m_k is the number of distinct words in the Bag_S_k.

- *Word Usage by the Similar End-Users* and *the Average Word Usage by the Similar End-Users* are shown in Equation 7.10 and 7.11, respectively. The learning features rank the importance of a service for an end-user based on the words used in the previously selected services of the similar end-users.

$$Word\_Usage\_SimUser = \sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Word\_Usage(S_k, U_j) \tag{7.10}$$

where $Word\_Usage$ is calculated using Equation 7.8 (same below)

$$Ave\_Word\_Usage\_SimUser = \frac{\sum_{j=1, j\neq i}^{n} Sim(U_i, U_j) \times Word\_Usage(S_k, U_j)}{m_k} \tag{7.11}$$

- *Task Similarity.* A service sharing the highest textual similarity value with a task T_a could be selected by end-users for the task. We use the cosine similarity algorithm [89] (denoted as *cos*) to calculate the similarity value of the two weighted bags of words, *i.e.*, $Task\_Similarity = cos(Bag\_T_a, Bag\_S_k)$.

- *Brand.* End-users can choose services with a well-known brand. The learning feature has a binary value. We use Equation 7.12 to calculate the *Brand* learning feature.

$$Brand = \begin{cases} 1 & \text{if the service description } S_k \text{ contains famous brand name} \\ 0 & \text{otherwise} \end{cases} \tag{7.12}$$

Inferring User Preferences and the Priority Orders

The user preferences are the learning features (listed in Table 7.1) that are important to end-users. We determine the importances of learning features by evaluating the impact of each learning feature on the performance of our ranking models. The rational is that the higher impact of a learning feature on the performance of the ranking model, the higher is the priority order of the corresponding user preference.

We build an original ranking model R_{all} including all of the four categories of learning features. Since learning features can be correlated and redundant, we conduct the Spearman's rank correlation test and the redundancy analysis [144] to remove the correlated and redundant learning features, respectively. We exclude one correlated learning feature in the original ranking model R_{all}, if the correlation value of two learning features is higher than 0.8 [163]. Learning features that can be predicted and expressed by other features are also excluded in the R_{all}. Our approach consists of the following two steps to identify user preferences and their priority orders.

[**Step 1**]. *Identifying user preferences.* As learning features in one category can share the same properties, we study the effect of each category of learning features on the performance of a model [106]. We build alternative ranking models to analyze the impact of learning features in a category on the ranking performance. An alternative ranking model is built including all of the learning features used in the original model R_{all} **but excluding the learning features in a category**. The rational is that the impact of a category of learning features on the ranking performance can be observed when the learning features in the category are excluded. The first alternative ranking model R_{-sa} considers all the learning features used in the original ranking model R_{all} except for the learning features in the category *service attributes* (*i.e.*, price, user rating, discount and relevance). In total, we generate four alternative ranking models R_{-sa}, R_{-suh}, R_{-uc} and R_{-tf} where they include all the learning features except for the learning features in the category *service attributes*, *service usage history*, *user context* and *textual feature*, respectively.

We compare the performance of an alternative ranking model with the performance of the original model R_{all} to obtain the performance disparity. If the performance of the alternative ranking model is worse than that of the original model R_{all}, and the performance disparity is larger than a threshold T, we consider that the learning features in the category are all important to the end-user. Since the performance disparity can be contributed by an individual learning feature, we can not determine the priority order of the learning features within a category. Therefore, we temporarily assign the lowest priority order to all of the learning features within an identified category. In the next step, we identify the priority order for individual learning features.

[**Step 2**]. *Identifying the priority order of user preferences.* We determine the priority order of user preferences by analyzing the impact of each learning feature on the ranking performance. We build an individual ranking model for a learning feature F^{Test} by including all of the learning features used in the original model R_{all} **but**

excluding the learning feature F^{Test}. Then, we compare the performance of the learned individual ranking model with the performance of the original model R_{all} to obtain the performance disparity for determining the significance of the learning feature F^{Test}. The larger is the performance disparity, more important is the learning feature F^{Test}. To compare the effect of two learning features F^a and F^b, we compare their performance disparity. If the difference of their performance disparity is larger than the threshold T, it indicates that an end-user prefers F^a over F^b. We compare every two individual ranking models and update the priority order of the learned important learning features in **step 1**.

7.2.3 Composing Services

To derive personalized composite services for end-users, our approach composes services using multi-objective reinforcement learning algorithm (MORL), and integrates the inferred user preferences and the priority orders identified in Section 7.2.2.

Multi-Objective Reinforcement Learning (MORL)

The service composition process is a sequential decision-making process, which can be modeled as a Markov Decision Process (MDP) [98][149]. A MDP involves choosing multiple tasks and services. The output of the service composition MDP is a composite service that maps from each task to a service. The MDP problem can be optimized by the MORL algorithm [133]. We use a reinforcement learning approach, Q-learning [133], to learn optimal composite services. For a task and a service, Q-learning initializes an arbitrary Q-value, such as 0. Q-learning conducts an exploration and exploitation strategy to update the Q-value recursively, until a convergence point is achieved. The Q-value update function considers the current reward as well as the long-term reward, as represented in Equation 7.13. More details of the algorithm can be referred to [133].

$$Q(t,a) \leftarrow Q(t,a) + \alpha[r + \gamma \max_{a'} Q(t',a') - Q(t,a)] \tag{7.13}$$

where r is the reward function; t' is the succeeding task to t; a' is the service under the task t'; α is the learning ratio and γ is the discount factor.

The reward is calculated by a reward function. We use the weight summation method as the reward function: $r = \sum_{i=1}^{N} \omega_i \times r_i$, where ω_i and r_i are the weight value and learning feature value for the i^{th} preference, respectively. A higher weight value of a preference means that an end-user puts more emphasis on the corresponding preference. We use the well-known ϵ-greedy policy [133] to explore and exploit services. Under a task, Q-learning chooses to perform the optimal service with a (1-ϵ) probability and perform a random service with a probability of ϵ. When the learning process ends, the optimal services for each task form the optimal composite services.

Composing Services with the Inferred User Preferences and the Priority Orders

Rather than learning all the possible optimal composite services as the existing approach, we aim to identify the subset of the composite services that maximize the overall user preferences.

Given a new service composition job, we build learning features for services as described in Section 7.2.2. To ensure the learning feature values in the same range, we use the approach proposed in [158] to scale the values for learning features to a range of 0 to 1. Our approach optimizes composite services with user preferences. The reward function computes the weighted sum of the important learning features. We determine the weight values for user preferences based on the priority order identified in Section 7.2.2. Given the priority order $O = \{O_1, O_2, ..., O_N\}$ (N is the number of important learning features), we randomly assign weight values based on the following constraints:

$$\begin{cases} \omega_1 + \omega_2 + ... + \omega_N = 1 \\ \omega_i > \omega_j & \text{if } O_i > O_j \end{cases} \tag{7.14}$$

where ω_i is the weight for the i^{th} preference, and O_i is the priority order for the i^{th} preference.

The constraints guarantee that we always assign a larger weight value for an important user preference if the learning feature has a higher priority order.

As end-users tend to make choices from multiple recommended results [125], it is desirable to derive multiple possible composite services for end-users. The multiple composite services are derived by performing the MORL algorithm multiple runs with different weight settings [83]. In each run, the weight values are unchanged.

7.3 Case Study

We conduct case studies to evaluate our personalization approach. In this section, we introduce our case study setup and experimental results.

7.3.1 Case Study Setup

Collecting Services. To test the effectiveness of our approach, we collect service and end-user information from the E-commerce domain in which end-users can perform their tasks on regular basis. We collect shopping items sold on Amazon[4]. We take the category name of shopping items (*e.g.*, headset and men jacket) as a task name and all of the shopping items sold under the category as the associated services. All of the extracted learning features are generic and applicable to other types of services (*i.e.*, web services and IoT services). We mine the web pages of Amazon website to collect services.

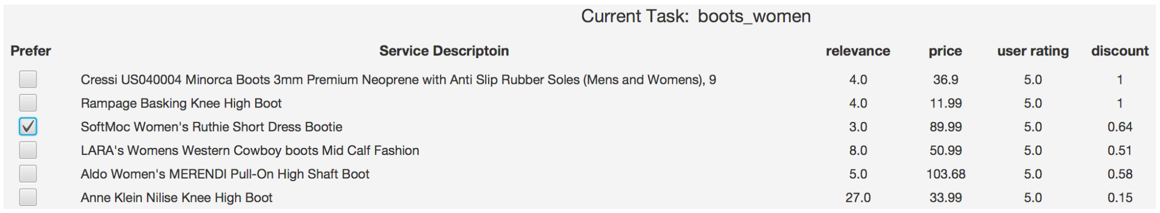
Our mined services fall into different domains: Clothing, Shoes & Handbags, Office Products, Sports & Outdoors, and Electronics. In total, we extract 64,888 services associated with 27 tasks. Table 7.2 describes the distribution of the services and the tasks in each domain. *Service descriptions* and *service attributes* (*i.e.*, price, discount, user rating, and relevance) are extracted for each service. For each category of products, Amazon provides "Top Brands" web pages to list famous brand names. We mine famous brand names from Amazon to determine the *brand* learning feature, as calculated in Equation 7.12.

Collecting Historical Service Usage Data. To collect end-users' historical service usage dataset, we develop a tool for end-users to compose services. An end-user enters various types of end-user information as described in Section 7.2.1. Our tool records four categories of learning features (described in Section 7.2.2). Thus, the collected dataset can be used in our ranking models to infer user preferences. Other datasets that record the four categories of learning features can also be used in our ranking models to infer user preferences. In our tool, when an end-user submits a goal to describe the composed services in our tool, the end-user can choose a set of tasks from our extracted tasks in order to achieve the goal. For instance, to buy gifts, an end-user can select the tasks recommended by the tool: buying candy chocolates,

[4]https://www.amazon.com

buying activity trackers and buying fountain pens.

In the initial stage, there are no sufficient historical data to use our approach. Therefore, the tool composes relevant services using the learning features known to end-users without historical information (*i.e.*, price, discount, user rating, and relevance). Our tool composes services using the MORL algorithm described in Section 7.2.3. The returned composite services are ranked by the expected Q-values derived from the MORL algorithm. As end-users tend to go through a limited number of returned services [125], we list 20 services relevant to carry out each task and show the related *service descriptions* and *service attributes* to end-users. An end-user can choose multiple preferred services by clicking the respective services. Our tool records service information as described in Section 7.2.1. Figure 7.3 illustrates a screenshot of our tool for selecting services.


Current Task: boots_women

Prefer	Service Descriptoin	relevance	price	user rating	discount
☐	Cressi US040004 Minorca Boots 3mm Premium Neoprene with Anti Slip Rubber Soles (Mens and Womens), 9	4.0	36.9	5.0	1
☐	Rampage Basking Knee High Boot	4.0	11.99	5.0	1
✓	SoftMoc Women's Ruthie Short Dress Bootie	3.0	89.99	5.0	0.64
☐	LARA's Womens Western Cowboy boots Mid Calf Fashion	8.0	50.99	5.0	0.51
☐	Aldo Women's MERENDI Pull-On High Shaft Boot	5.0	103.68	5.0	0.58
☐	Anne Klein Nilise Knee High Boot	27.0	33.99	5.0	0.15

Figure 7.3: A screenshot of our tool for selecting services.

User Study. To evaluate our approach, we recruit 12 subjects to participate in our user study (*i.e.*, 4 females and 8 males). All of the subjects are graduate students. They are not familiar with service composition techniques, but perform on-line tasks daily. The subjects spend approximately 8-10 hours on-line each day. We provide instructions for using the tool. The subjects use our tool for a period of two months. We do not set any restriction on the number of goals and tasks that a subject needs to perform. On average, the subjects have performed 51.5 tasks over a period of two

Table 7.2: The distribution of services and tasks in each domain. We also show the average number of tasks performed by the subjects in each domain.

Domain	# Tasks	# Services	# Tasks Performed by Subjects
Clothing	10	34,725	14.5
Shoes & Handbags	3	9,066	4.2
Office Products	4	5,894	7.7
Sports & Outdoors	3	6,286	6.1
Electronics	7	8,917	19.0

months. The average number of tasks performed in each domain is listed in Table 7.2.

7.3.2 Case Study Results

RQ7.1 Is our approach effective to compose services?

Motivation. To save end-users from manually specifying their preferences, it is critical to automatically learn user preferences and the priority orders. We are interested to evaluate the effectiveness of our approach that composes services by considering the inferred user preferences.

Approach. To compare the performance of our approach, we build two baseline approaches. In our approach, we use an end-user previously performed m goals as the training dataset, and compose services for the rest of the goals for the end-user using our proposed MORL algorithm (see Section 7.2.3). We apply the approach to extract the learning features and identify the priority orders of user preferences. We follow the same data splitting strategy used by Wang *et al.* [151]. For the m goals, the learning-to-rank algorithm takes half of the dataset for training and validation, the other half for testing. We use Equation (7.15) to evaluate the performance of the learning-to-rank models. P@k is the ratio of the number of end-user selected services in the top k returned services to the number k.

$$P@k = \frac{\textit{\#Services Selected in Top k Results}}{k} \tag{7.15}$$

We compare the P@k to determine the preferences and the priority orders (see Section 7.2.2). If the difference between the P@ks of two individual feature models is higher than a threshold, these two learning features are considered to have different orders.

We design two baseline approaches, named Equal-Weights-MORL and No-Order-MORL. The two baseline approaches use the MORL algorithm, the weighted sum Q-learning algorithm [149], to compose services. Different from our approach, both baselines assume that user preferences and the priority orders are pre-defined. Both baseline approaches optimize the preferences under the category *service attributes* (*i.e.*, price, user rating, discount and relevance), since the learning features are visible and known to end-users.

Equal-Weights-MORL. The weights are the same for each preference and are set to $\boldsymbol{\omega} = (0.25, 0.25, 0.25, 0.25)$.

No-Order-MORL. This approach hypothesizes that end-users do not have specific priority orders over the preferences. The summation of the weight values is 1 and the weight values are randomly assigned in each run.

To ensure the efficiency of the MORL algorithm, we set the learning rate to 1, discount factor to 0.8, the ϵ to 0.7 and the number of iteration to converge as 1,000, same as [98]. To compose services for a goal, our approach and the two baseline approaches run the MORL algorithm for 1,000 times. We compute the precision

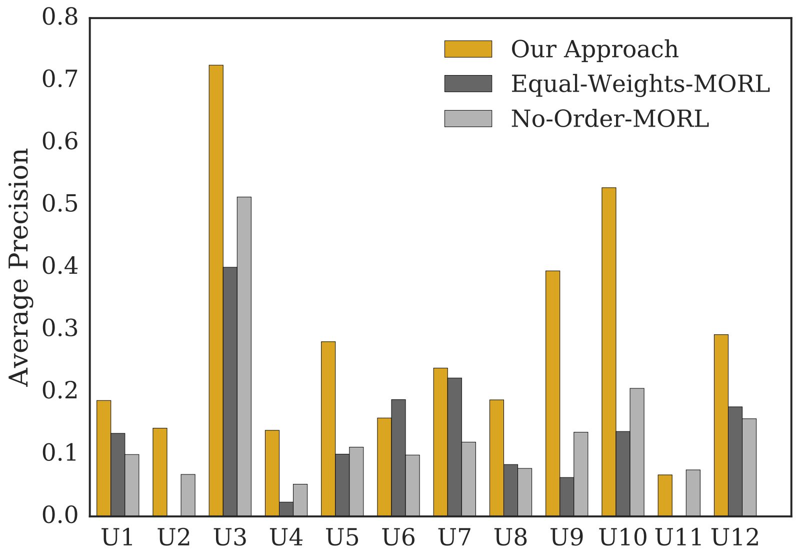


Figure 7.4: Average precision of our approach and the two baseline approaches using 70% of the whole dataset for training. The x-axis denotes subject ids, Ui means the i^{th} end-user. y-axis is the average precision of recommending services.

using Equation (7.16). The precision is the ratio of the number of end-user selected services in the retrieved ones to the total number of retrieved services. For the goals that we compose services, we compute the precision for each goal and compare the average performances of the three approaches. The size of the training dataset (*i.e.*, m) can affect our results. Therefore, in our experiments, we take 50%, 60%, 70% and 80% of the number of end-user performed goals as our training dataset separately.

$$Precision = \frac{\#Correctly\ Identified\ Services}{\#Retrieved\ Services} \tag{7.16}$$

Results. **Our approach is more effective to help end-users select and compose services.** Figure 7.4 shows the average precision of recommending services for the three approaches with 70% of the whole dataset for training. We use P@1 to

Table 7.3: Performance of the three approaches with different sizes of training dataset. **Abbreviation for method names**: OA, B1, and B2: our approach, Equal-Weights-MORL, and No-Order-MORL. We show the actual precision of our approach, and demonstrate the percentage of precision our approach improves or decreases the Equal-Weights-MORL and No-Order-MORL. "+" and "-" means improvement and decrease, respectively. Ui means the i^{th} end-user. The number colored with dark gray represents the highest number in the column, while the light gray denotes the lowest (same for other tables).

UID	**Different Sizes of Training Dataset**											
	50%			**60%**			**70%**			**80%**		
	OA	B1	B2	OA	B1	B2	OA	B1	B2	OA	B1	B2
U1	25	+69	+77	28	+101	+131	18	+39	+87	16	-15	+31
U2	19	+35	+48	12	+∞	+34	14	+∞	+110	11	+∞	+65
U3	63	+16	+42	64	+28	+31	72	+81	+41	66	+50	+40
U4	14	+899	+251	13	+682	+214	13	+501	+167	15	+332	+177
U5	20	+93	+74	37	+195	+212	28	+180	+152	22	+∞	+301
U6	29	0	+146	20	-19	+84	15	-15	+60	18	-10	+60
U7	20	0	+87	17	-31	+53	23	+7	+99	41	+150	+308
U8	17	+221	+97	4	-29	-48	18	+124	+142	11	+∞	+75
U9	29	+673	+189	36	+626	+223	39	+530	+191	40	+302	+296
U10	50	+285	+158	42	+221	+109	52	+287	+156	47	+154	+112
U11	0	0	-100	24	+∞	+165	6	+∞	-10	23	+∞	+281
U12	18	-26	-0.4	25	+9	+43	29	+65	+86	37	+40	+92

evaluate our ranking models and set the threshold to determine the significance of learning feature as 0. If the performance disparity of two ranking models is higher than the threshold value 0, we consider that the learning feature is significant or has a higher priority order. Compared with *Equal-Weights-MORL* and *No-Order-MORL*, our approach achieves a better precision for 10 out of 12 subjects (*i.e.*, 83% of subjects, excepts for U6 and U11). The highest average precision our approach obtained is 72% on the subject U3. The subject U3 is a male graduate student majoring in the computer science.

In Table 7.3, we show a breakdown of Figure 7.4, and the precision of the three approaches with different sizes of training data. We calculate the improvement of our approach compared with the baseline approaches using Equation (7.17). On average, our approach improves *Equal-Weights-MORL* by 205%, 176%, 180% and 125%, and *No-Order-MORL* by 89%, 103%, 107% and 153%, when we use 50%, 60%, 70% and 80% of the whole dataset for training, respectively. We exclude the $+\infty$ to calculate the average, as the average would be infinity. With only 50% of the whole dataset for training, we improve *Equal-Weights-MORL* and *No-Order-MORL* by 205% and 89%, respectively. As the increase of the size of training dataset, the improvement of our approach compared with the *No-Order-MORL* is more significant. Among the 12 subjects, the subject U4 achieves the highest improvement using our approach, compared with *Equal-Weights-MORL* (604% improvement) and *No-Order-MORL* (202% improvement).

Why our approach works? Our approach achieves a better performance for recommending services. It demonstrates that the inferred user preferences and the priority orders are valid for end-users. The *Equal-Weights-MORL* assumes that end-users view all of the user preferences equally. Thus end-user preferred services may not be contained in the retrieved services. The *No-Order-MORL* locates all the optimal (or sub-optimal) composite services. However, end-users can only prefer a small subset of the possible service compositions. The inferred user preferences and the priority orders help our approach to accurately locate the end-user preferred optimal composite services.

$$Improvement = \frac{Precision(OA) - Precision(B)}{Precision(B)} \tag{7.17}$$

Table 7.4: Performance of our approach with different threshold values (%). "T" in the headers represents threshold. Ui means the i^{th} end-user.

UID	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10	U11	U12
T0.00	21.6	14.3	66.6	14.2	26.3	21.0	25.7	13.0	35.2	48.3	13.6	27.7
T0.01	21.8	17.9	63.9	11.5	26.0	21.2	25.0	12.2	35.6	45.2	12.1	27.3
T0.03	16.7	18.0	64.5	8.2	19.9	17.8	23.7	12.7	32.8	26.5	11.2	32.4
T0.05	14.7	17.8	60.6	6.4	14.7	16.3	20.9	12.2	29.1	27.6	8.6	28.2

where B means a baseline and OA means our approach.

RQ7.2 What is the best configuration for using our approach?

Motivation. The user preferences and the priority orders of the preferences can be affected by two parameters, *i.e.*, the threshold of ranking models, and the precision metrics used to evaluate the models. In this question, we study the effect of the two parameters on the performance of our approach.

Approach. We conduct two experiments to study the two parameters separately. The threshold is used to compare ranking models, so as to determine the significant learning features. The precision metrics is used to evaluate the performance of ranking models to rank services. We test various threshold values from 0 to 0.05 and precision metrics P@1, P@3 and P@5. To test the effect of the threshold, we use P@1 as the precision metric to evaluate ranking models. For each threshold value, we use our approach to determine the priority orders of learning features and compose services.

Results. The performance of our approach is affected by the threshold of ranking models. Table 7.4 shows the results of our approach with different threshold values. Each value in the table is the average value of the results of our approach using four sizes of training datasets (*i.e.*, 50%, 60%, 70% and 80%). For threshold values 0, 0.01, 0.03 and 0.05, the average precisions over all the subjects are 27.2%, 26.6%, 23.7% and 21.4%, respectively. The lower is the threshold value, the better is the performance. A high threshold value can ignore important learning features. Hence, we set the threshold value as 0 for the rest of the experiments. We determine the important learning features and the priority orders by comparing the performances of the ranking models. If the performance disparity is higher than the threshold value 0, we conceive the learning feature is significant or has a higher priority order.

Table 7.5 shows the results of the effect of precision metrics on our approach. Each value in the table is the average results from the four training datasets. The average precision of all the subjects is 27.2%, 28.2% and 27.0% for P@1, P@3 and P@5, respectively. We do not observe significant differences with various precision metrics. It shows that the our approach is not affected by the precision metrics.

Table 7.5: Performance of our approach with different precision metrics (%). *Ui* means the i^{th} end-user.

UID	**U1**	**U2**	**U3**	**U4**	**U5**	**U6**	**U7**	**U8**	**U9**	**U10**	**U11**	**U12**
P@1	21.6	14.3	66.6	14.2	26.3	21.0	25.7	13.0	35.2	48.3	13.6	27.7
P@3	22.1	13.4	66.8	15.5	24.5	19.5	30.7	14.8	31.2	53.4	18.1	28.9
P@5	19.9	11.1	65.9	13.4	26.4	12.4	23.6	22.0	27.7	53.8	19.3	29.0

7.4 Chapter Summary

To overcome the complexity of manually specifying user preferences by end-users in the service composition process, we propose a learning-to-rank approach to automatically learn user preferences and the priority orders of the preferences using the historical service composition dataset. Our learned priority orders of user preferences are integrated into the multi-objective reinforcement learning (MORL) algorithm, to efficiently derive optimal composite services. We conduct experiments to evaluate the effectiveness of our approach. The results of our case study show that the inferred user preferences are helpful for selecting services. Compared with two well-established baseline approaches, our approach improves them by 100%-200% on precision for service selection.

Chapter 8

Conclusions and Future Work

The massive amount of services, *i.e.*, web services and IoT services, are composed to provide various functionalities to ease people's daily lives. However, end-users face numerous choices to meet their personal preferences when selecting the desired services to perform on-line activities. In this thesis, we propose to develop autonomous agents to help end-users perform on-line activities. Specifically, we propose a semi-natural language syntax for developers to develop agents for service composition. The developed agents can pro-actively perform services, and autonomously react upon the results of services to decide the next performed services. To relieve end-users from the effort of selecting services, the agents can recommend personalized services by learning user preferences from historical data. In the following subsections, we summarize the contributions of this thesis and outline the future research that may enhance our work.

8.1 Contribution

The main goal of the thesis is to relieve the effort from developers to design agents for service composition, and reduce manual efforts from end-users to select preferred services. The major contributions of the thesis are listed below.

(1) Proposing a semi-natural language syntax to generate agents semi-automatically for service composition (Chapter 3). The design and implementation of agents using the Jadex platform require substantial efforts from developers. To reduce the required efforts from developers to develop agents, we propose an easy-to-understand semi-natural language syntax. Specifically, the proposed syntax supports service discovery, and generates libraries and code for consuming services. Using our proposed syntax, developers may spend less efforts to learn the developing of agents, thereby developing agents for service composition more effectively.

The results of our case study show that 1) our approach can correctly generate agent code from the proposed semi-natural language syntax; and 2) developers prefer to use our semi-natural language syntax to develop agents.

(2) Identifying service related tasks to design composite services (Chapter 4). Developers may lack domain knowledge to design abstract processes for service composition. To broaden the knowledge of developers to design abstract processes, we propose an automatic approach to identify service related tasks from on-line how-to instructions. We use a logistic regression model to identify valid how-to instructions that are relevant to service composition. From the identified valid how-to instructions, we use natural language processing techniques and build a Multi-Layer Perceptron (MLP) model (a deep learning based model) to identify service related tasks. Our approach reminds developers of possible service related tasks to design composite services.

Our results show that 1) our approach can effectively identify valid how-to instructions that can be used for service composition; and 2) the built MLP model outperforms three baseline models, e.g., *Random Forest, to identify service related tasks.*

(3) Transforming IoT applications to RESTful web services to be integrated with existing web services deployed on the cloud (Chapter 5). To enable the communication among various IoT devices and integrate IoT devices with web services, we propose an automatic approach that transforms the functionalities of IoT applications in IoT devices to IoT services that confirm to the RESTful paradigm. Our automatic approach may save the effort from developers to instantiate IoT services and design user-friendly interface that is used to invoke IoT services.

Our case study results show that 1) we can effectively identify external methods from IoT applications; 2) our approach can extract service specifications from external methods accurately; and 3) our approach can correctly instantiate IoT services on the cloud.

(4) Predicting user ratings to recommend end-user preferred services (Chapter 6). A recommendation system is essential to predict end-user preferred services, since there are a vast number of functional similar services. To accurately predict user ratings on services, we propose a deep learning based model, *i.e.*, DeepCont, which incorporates user reviews and service descriptions in additional to user ratings for recommendation. The accurate prediction results of our model can help end-users to effectively select their preferred services.

Our main findings are: 1) the word embedding matrices and the VAE networks are essential components of our model; 2) our proposed DeepCont model outperforms four state-of-the-art baseline models, e.g., *CDL, for predicting user ratings of existing services; and 3) our model achieves similar performance compared with the best performing baseline model,* i.e., *CVAE, for predicting user ratings of new services.*

(5) Automatically learning user preferences to recommend personalized composite services for end-users (Chapter 7). Preference based service composition focuses on automatic recommendation of personalized composite services that satisfy end-users' goals. We propose to use a ranking algorithm, *i.e.*, RankBoost, to automatically learn user preferences using historical service usage data. The learned user preferences are integrated into the Multi-Objective Reinforcement Learning (MORL) algorithm to actively recommend a collection of services to end-users. Our approach relieves the cognitive abilities of end-users to manually specify their preferences, *e.g.*, preferred brands, to search for services.

Our case study results demonstrate that our preference based MORL approach improves the performance of two baseline MORL approaches that assume predefined preferences by 100% - 200% in terms of precision for recommending services.

8.2 Future Work

Our proposed approaches make a positive impact on the process of composing services and selecting presonalized services. However, the thesis can be extended from multiple perspectives. In this section, we outline the promising extensions of the thesis for future work.

8.2.1 Generating Executable Agent Code for Other Agent Platforms

Our generated agent code from our proposed semi-natural language syntax is specific to the Jadex platform. However, various BDI agent platforms use different programming languages for developing agents. For example, the Jason platform uses a Domain Specific Language (DSL), *i.e.*, AgentSpeak, and the JACK platform uses a programming language that extends Java. It is interesting for our proposed syntax to generate agent code for other BDI platforms, *e.g.*, Jason.

8.2.2 Analyzing Source Code of IoT Applications Written in Other Programming Languages

Our approach that identifies external methods and extracts service specifications is specific to IoT applications written in the Python programming language. However, IoT applications may be written in other programming languages, *e.g.*, Java and Javascript. Therefore, future work is encouraged to extend our approach to analyze IoT applications implemented in other programming languages. To instantiate IoT services, our approach fills the extracted service specifications to the proposed service schema. Our proposed service schema is language independent. Developers can manually input service specifications into the service schema. The filled service schema can transform IoT applications to IoT services, regardless of implemented programming languages.

8.2.3 Dynamically Recommending Next Tasks to Perform

Our approach in Chapter 4 is a static approach to identify tasks from on-line how-to instructions. In practice, the subsequent tasks may be dynamically decided based on previously performed tasks to serve end-users, since a set of tasks may frequently occur together. For example, when a task is implemented to buy T-shirts, the next performed task by end-users could be buying jeans. Our future work could develop approaches to dynamically recommend next tasks for developers based on the tasks that are implemented previously.

8.2.4 Learning User Preferences from More Categories of Learning Features

In Chapter 7, we build 20 categories of learning features from services and end-user information to automatically learn user preferences. More categories of learning features can be explored to capture more aspects of user preferences. For example, our approach uses end-users' personal information to identify end-user similarity. Another promising approach to identify end-user similarity is to use collaborative filtering based approaches to learn user latent features and calculate the similarity among the learned user latent features.

Bibliography

[1] Style guide for python code.

[2] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *Software Engineering, IEEE Transactions on*, 33(6):369–384, 2007.

[3] Costin Bădică, Nick Bassiliades, Sorin Ilie, and Kalliopi Kravari. Agent reasoning on the web using web services. *Computer Science and Information Systems*, 11(2):697–721.

[4] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technology (TOIT)*, 14(4):23, 2014.

[5] Wolf-Tilo Balke and Matthias Wagner. Towards personalized selection of web services. In *WWW. ACM*, pages 20–24, 2003.

[6] Michele Banko, Michael J Cafarella, Stephen Soderland, Matthew Broadhead, and Oren Etzioni. Open information extraction for the web. In *IJCAI*, volume 7, pages 2670–2676, 2007.

[7] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009.

[8] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[9] Federico Bergenti, Eleonora Iotti, Stefania Monica, and Agostino Poggi. Agent-oriented model-driven development for jade with the jadel programming language. *Computer Languages, Systems & Structures*, 50:142–158, 2017.

[10] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[11] Rafael H Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking agentspeak. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416. ACM, 2003.

[12] Rafael H Bordini and Jomi F Hübner. Bdi agent programming in agentspeak using jason. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 143–164. Springer, 2005.

[13] Michael Bratman. Intention, plans, and practical reason. 1987.

[14] Lars Braubach, Winfried Lamersdorf, and Alexander Pokahr. Jadex: Implementing a bdi-infrastructure for jade agents. 2003.

[15] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[16] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[17] Rose Catherine and William Cohen. Transnets: Learning to transform for recommendation. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, pages 288–296. ACM, 2017.

[18] Moharram Challenger, Sebla Demirkol, Sinem Getir, Marjan Mernik, Geylani Kardas, and Tomaž Kosar. On the use of a domain-specific modeling language in the development of multiagent systems. *Engineering Applications of Artificial Intelligence*, 28:111–141, 2014.

[19] Moharram Challenger, Marjan Mernik, Geylani Kardas, and Tomaž Kosar. Declarative specifications for the development of multi-agent systems. *Computer Standards & Interfaces*, 43:91–115, 2016.

[20] Soumi Chattopadhyay and Ansuman Banerjee. A variation aware composition model for dynamic web service environments. In *International Conference on Service-Oriented Computing*, pages 694–713. Springer, 2018.

[21] Chong Chen, Min Zhang, Yiqun Liu, and Shaoping Ma. Neural attentional rating regression with review-level explanations. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1583–1592, 2018.

[22] David L Chen and Raymond J Mooney. Learning to interpret natural language navigation instructions from observations. In *Proc. 25th AAAI Conf. Artificial Intell.*, pages 859–865, 2011.

[23] Zhiyong Cheng, Ying Ding, Lei Zhu, and Mohan Kankanhalli. Aspect-aware latent factor model: Rating prediction with ratings and reviews. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 639–648. International World Wide Web Conferences Steering Committee, 2018.

[24] Amit K Chopra, Fabiano Dalpiaz, F Başak Aydemir, Paolo Giorgini, John Mylopoulos, and Munindar P Singh. Protos: Foundations for engineering innovative sociotechnical systems. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 53–62. IEEE, 2014.

[25] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[26] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015.

[27] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[28] Michael Collins. Head-driven statistical models for natural language parsing. *Computational linguistics*, 29(4):589–637, 2003.

[29] Juan M Corchado, Dante I Tapia, and Javier Bajo. A multi-agent architecture for distributed services and applications. *IJICIC*, 8(4):2453–2476, 2012.

[30] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[31] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. smap: a simple measurement and actuation profile for physical information. In *sensys*, 2010.

[32] Xin Dong, Lei Yu, Zhonghuo Wu, Yuxia Sun, Lingfeng Yuan, and Fangxi Zhang. A hybrid collaborative filtering model with deep structure for recommender systems. In *AAAI*, pages 1309–1315, 2017.

[33] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[34] Khalid Elgazzar, Ahmed E Hassan, and Patrick Martin. Clustering wsdl documents to bootstrap the discovery of web services. In *2010 IEEE International Conference on Web Services*, pages 147–154. IEEE, 2010.

[35] Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intell.*, 165(1):91–134, 2005.

[36] Rick Evertsz, Martyn Fletcher, Richard Jones, Jacquie Jarvis, James Brusey, and Sandy Dance. Implementing industrial multi-agent systems using jack tm.

In *International Workshop on Programming Multi-Agent Systems*, pages 18–48. Springer, 2003.

[37] Christiane Fellbaum. *WordNet*. Wiley Online Library, 1998.

[38] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technology*, 2(2):115–150, 2002.

[39] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[40] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003.

[41] Sinem Getir, Moharram Challenger, and Geylani Kardas. The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems. *International Journal of Cooperative Information Systems*, 23(03):1450005, 2014.

[42] Dominic Greenwood and Monique Calisti. Engineering web service-agent integration. In *SMC*, volume 2, pages 1918–1925. IEEE, 2004.

[43] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *TSC*, 3(3):223–235, 2010.

[44] Ishaan Gulrajani, Kundan Kumar, Faruk Ahmed, Adrien Ali Taiga, Francesco Visin, David Vazquez, and Aaron Courville. Pixelvae: A latent variable model for natural images. *arXiv preprint arXiv:1611.05013*, 2016.

[45] Sara Hachem, Thiago Teixeira, and Valérie Issarny. Ontologies for the internet of things. In *Proceedings of the 8th Middleware Doctoral Symposium*, page 3. ACM, 2011.

[46] Christian Hahn, Cristián Madrigal-Mora, and Klaus Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 2009.

[47] Yushi Hao, Yushun Fan, Wei Tan, and Jia Zhang. Service recommendation based on targeted reconstruction of service descriptions. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 285–292. IEEE, 2017.

[48] Simon S Haykin et al. *Neural networks and learning machines/Simon Haykin.* New York: Prentice Hall,, 2009.

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[50] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016.

[51] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, pages 173–182. International World Wide Web Conferences Steering Committee, 2017.

[52] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.

[53] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[54] Marcus J Huber. Jam: A bdi-theoretic mobile agent architecture. In *Proceedings of the third annual conference on Autonomous Agents*, pages 236–243. ACM, 1999.

[55] Jomi F Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous agents and multi-agent systems*, 20(3):369–400, 2010.

[56] Michael N Huhns. Agents as web services. *IEEE Internet computing*, 6(4):93–95, 2002.

[57] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[59] Anup K Kalia, Jin Xiao, Muhammed F Bulut, Maja Vukovic, and Nikos Anerousis. Cataloger: Catalog recommendation service for it change requests. In *International Conference on Service-Oriented Computing*, pages 545–560. Springer, 2017.

[60] Boris Katz. From sentence processing to information access on the world wide web. In *AAAI Spring Symp. Natural Language Process. for the World Wide Web*, volume 1, page 997. Stanford, CA, 1997.

[61] D Kinga and J Ba Adam. A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, volume 5, 2015.

[62] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.*, 2013.

[63] Dan Klein and Christopher D Manning. Accurate unlexicalized parsing. In *Proc. 41st Annual Meeting on Assoc. for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[64] Matthias Klusch, Benedikt Fries, and Katia Sycara. Automated semantic web service discovery with owls-mx. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM, 2006.

[65] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[66] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[67] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[68] Cody Kwok, Oren Etzioni, and Daniel S Weld. Scaling question answering to the web. *ACM Trans. Inform. Syst.*, 19(3):242–262, 2001.

[69] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[70] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pages 1558–1566. JMLR. org, 2016.

[71] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

[72] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[73] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[74] Sangkeun Lee, Juno Chang, and Sang-goo Lee. Survey and trend analysis of context-aware systems. *Information-An International Interdisciplinary Journal*, 14(2):527–548, 2011.

[75] Xiaopeng Li and James She. Collaborative variational autoencoder for recommender systems. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 305–314. ACM, 2017.

[76] Dawen Liang, Rahul G Krishnan, Matthew D Hoffman, and Tony Jebara. Variational autoencoders for collaborative filtering. *arXiv preprint arXiv:1802.05814*, 2018.

[77] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[78] Chen Lin, Anup Kalia, Jin Xiao, Maja Vukovic, and Nikos Anerousis. Nl2api: A framework for bootstrapping service recommendation using natural language queries. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 235–242. IEEE, 2018.

[79] Naiwen Lin, Ugur Kuter, and Evren Sirin. Web service composition with user preferences. In *ESWC*, pages 629–643. Springer, 2008.

[80] Thomas Lin, Patrick Pantel, Michael Gamon, Anitha Kannan, and Ariel Fuxman. Active objects: Actions for entity-centric search. In *Proc. 21st Int'l. Conf. WWW*, pages 589–598. ACM, 2012.

[81] Guang Ling, Michael R Lyu, and Irwin King. Ratings meet reviews, a combined approach to recommend. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 105–112. ACM, 2014.

[82] Bing Liu, Chee Wee Chin, and Hwee Tou Ng. Mining topic-specific concepts and definitions on the web. In *Proc. 12th Int'l. Conf. WWW*, pages 251–260. ACM, 2003.

[83] Chunming Liu, Xin Xu, and Dewen Hu. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(3):385–398, 2015.

[84] Hugo Liu and Push Singh. Conceptnet practical commonsense reasoning toolkit. *BT technology J.*, 22(4):211–226, 2004.

[85] Sharon L Lohr. *Sampling: Design and Analysis: Design and Analysis.* Chapman and Hall/CRC, 2019.

[86] Yichao Lu, Ruihai Dong, and Barry Smyth. Coevolutionary recommendation model: Mutual learning between ratings and reviews. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 773–782. International World Wide Web Conferences Steering Committee, 2018.

[87] Zakaria Maamar, Soraya Kouadri Mostefaoui, and Hamdi Yahyaoui. Toward an agent-based and context-oriented approach for web services composition. *TKDE*, 17(5):686–697, 2005.

[88] Abderrahmane Maaradji, Hakim Hacid, Johann Daigremont, and Noël Crespi. Towards a social network based approach for services composition. In *ICC IEEE*, pages 1–5, 2010.

[89] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press, 2008.

[90] Viviana Mascardi, Daniela Demergasso, and Davide Ancona. Languages for programming bdi-style agents: an overview. In *WOA*, volume 2005, pages 9–15, 2005.

[91] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[92] Sheila A McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE intelligent systems*, 16(2):46–53, 2001.

[93] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[94] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[95] Andriy Mnih and Ruslan R Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2008.

[96] Pavlos Moraitis and Nikolaos Spanoudakis. The gaia2jade process for multi-agent systems development. *Applied Artificial Intelligence*, 20(2-4):251–273, 2006.

[97] David Morley and Karen Myers. The spark agent framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-agent Systems-Volume 2*, pages 714–721. IEEE Computer Society, 2004.

[98] Ahmed Moustafa and Minjie Zhang. Multi-objective service composition using reinforcement learning. In *ICSOC*, pages 298–312. Springer, 2013.

[99] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA Companion*, pages 815–816, 2007.

[100] Patrice Perny and Paul Weng. On finding compromise solutions in multiobjective markov decision processes. In *ECAI*, pages 969–970, 2010.

[101] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A bdi reasoning engine. In *Multi-agent programming*, pages 149–174. Springer, 2005.

[102] Alexander Pokahr, Braubach Lars, and Kai Jander. Unifying agent and component concepts. In *German Conference on Multiagent System Technologies*. Springer, 2010.

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

[104] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[105] Nissanka B Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. 2008.

[106] Jiezhong Qiu, Yixuan Li, Jie Tang, Zheng Lu, Hao Ye, Bo Chen, Qiang Yang, and John E Hopcroft. The lifecycle and cascade of wechat social messaging groups. In *WWW. ACM*, pages 311–320, 2016.

[107] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[108] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

[109] BB Prahlada Rao, Paval Saluia, Neetu Sharma, Ankit Mittal, and Shivay Veer Sharma. Cloud computing for internet of things & sensing based applications. In *ICST*, pages 374–380, 2012.

[110] Lifang Ren, Wenjian Wang, and Hang Xu. A reinforcement learning method for constraint-satisfied services composition. *IEEE Transactions on Services Computing*, 2017.

[111] Steffen Rendle. Factorization machines. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 995–1000. IEEE, 2010.

[112] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, pages 452–461. AUAI Press, 2009.

[113] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*, 2014.

[114] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008.

[115] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.

[116] Christian Robert. Machine learning, a probabilistic perspective, 2014.

[117] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. Sarl: a general-purpose agent-oriented programming language. In *WI-IAT*, volume 3, pages 103–110. IEEE, 2014.

[118] Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel, and Fabio Casati. Baya: assisted mashup development as a service. In *WWW. ACM*, pages 409–412, 2012.

[119] Andrew I Schein, Alexandrin Popescul, Lyle H Ungar, and David M Pennock. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th*

annual international ACM SIGIR conference on Research and development in information retrieval, pages 253–260. ACM, 2002.

[120] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[121] Iulian Vlad Serban, Alessandro Sordoni, Ryan Lowe, Laurent Charlin, Joelle Pineau, Aaron C Courville, and Yoshua Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. In *AAAI*, pages 3295–3301, 2017.

[122] Mehrnoush Shamsfard and Ahmad Abdollahzadeh Barforoush. Learning ontologies from natural language texts. *Int'l. J. of human-computer studies*, 60(1):17–63, 2004.

[123] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures.* crc Press, 2003.

[124] Mazen Malek Shiaa, Jan Ove Fladmark, and Benoit Thiell. An incremental graph-based approach to automatic service composition. In *2008 IEEE International Conference on Services Computing*, volume 1, pages 397–404. IEEE, 2008.

[125] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. In *SIGIR Forum*, volume 33, pages 6–12. ACM, 1999.

[126] Richard Socher, John Bauer, Christopher D Manning, et al. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of*

the Association for Computational Linguistics (Volume 1: Long Papers), pages 455–465, 2013.

[127] Shirin Sohrabi and Sheila A McIlraith. Preference-based web service composition: A middle ground between execution and search. In *International Semantic Web Conference*, pages 713–729. Springer, 2010.

[128] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[129] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009, 2009.

[130] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. Building natural language interfaces to web apis. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 177–186. ACM, 2017.

[131] Kazunari Sugiyama, Kenji Hatano, and Masatoshi Yoshikawa. Adaptive web search based on user profile constructed without any effort from users. In *WWW. ACM*, pages 675–684, 2004.

[132] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.

[133] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[134] Mingdong Tang, Yechun Jiang, Jianxun Liu, and Xiaoqing Liu. Location-aware collaborative filtering for qos-based service recommendation. In *2012 IEEE 19th International Conference on Web Services*, pages 202–209. IEEE, 2012.

[135] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. Coapthon: easy development of coap-based iot applications with python. In *WF-IoT*, pages 63–68. IEEE, 2015.

[136] Baris Tekin Tezel, Moharram Challenger, and Geylani Kardas. A metamodel for jason bdi agents. In *OASIcs-OpenAccess Series in Informatics*, volume 51, 2016.

[137] Jesse Thomason, Shiqi Zhang, Raymond Mooney, and Peter Stone. Learning to interpret natural language commands through human-robot dialog. In *Proc. 2015 Int'l. Joint Conf. Artificial Intell. (IJCAI)*, pages 859–865, 2011.

[138] Hongxia Tong, Jian Cao, Shensheng Zhang, and Minglu Li. A distributed algorithm for web service composition based on service agent model. *IEEE Transactions on Parallel and Distributed Systems*, 22(12):2008–2021, 2011.

[139] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. Conf. North American Chapter of the Assoc. Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[140] Bipin Upadhyaya, Foutse Khomh, Ying Zou, Antonio Lau, and Jason Ng. A concept analysis approach for guiding users in service discovery. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–8. IEEE, 2012.

[141] Bipin Upadhyaya, Yao Zou, Iman Keivanloo, and Jason Ng. Quality of experience: User's perception about web services. *IEEE. TSC*, 8.

[142] Bipin Upadhyaya, Ying Zou, Shaohua Wang, and Joanna Ng. Automatically composing services by mining process knowledge from the web. In *Service-Oriented Computing*, pages 267–282. Springer, 2013.

[143] Bipin Upadhyaya, Ying Zou, Hua Xiao, Joanna Ng, and Alex Lau. Migration of soap-based services to restful services. In *WSE*, pages 105–114. IEEE, 2011.

[144] Arnold L Van Den Wollenberg. Redundancy analysis an alternative for canonical correlation analysis. *Psychometrika*, 42(2):207–219, 1977.

[145] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408, 2010.

[146] Chong Wang and David M Blei. Collaborative topic modeling for recommending scientific articles. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 448–456. ACM, 2011.

[147] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, pages 1235–1244. ACM, 2015.

[148] Hongbing Wang, Xin Chen, Qin Wu, Qi Yu, Zibin Zheng, and Athman Bouguettaya. Integrating on-policy reinforcement learning with multi-agent techniques for adaptive service composition. In *ICSOC*, pages 154–168. Springer, 2014.

[149] Hongbing Wang, Xuan Zhou, Xiang Zhou, Weihong Liu, and Wenya Li. Adaptive and dynamic service composition using q-learning. In *22nd International Conference on Tools with Artificial Intelligence*, volume 1, pages 145–152. IEEE, 2010.

[150] Jian Wang, Neng Zhang, Cheng Zeng, Zheng Li, and Keqing He. Towards services discovery based on service goal extraction and recommendation. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 65–72. IEEE, 2013.

[151] Shaohua Wang, Ying Zou, Joanna Ng, and Tinny Ng. Learning to reuse user inputs in service composition. In *ICWS*, pages 695–702. IEEE, 2015.

[152] Michael Wooldridge. *An introduction to multiagent systems.* John Wiley & Sons, 2009.

[153] Wei Xiong, Zhao Wu, Bing Li, Qiong Gu, Lei Yuan, and Bo Hang. Inferring service recommendation from natural language api descriptions. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 316–323. IEEE, 2016.

[154] Stephen JH Yang, Jia Zhang, and Irene YL Chen. A jess-enabled context elicitation system for providing context-aware web services. *Expert Systems with Applications*, 34(4):2254–2266, 2008.

[155] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011.

[156] Qi Yu and Athman Bouguettaya. Multi-attribute optimization in service selection. *World Wide Web*, 15(1):1–31, 2012.

[157] Franco Zambonelli, Nicholas R Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.

[158] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.

[159] Yongfeng Zhang, Guokun Lai, Min Zhang, Yi Zhang, Yiqun Liu, and Shaoping Ma. Explicit factor models for explainable recommendation based on phrase-level sentiment analysis. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 83–92. ACM, 2014.

[160] Yu Zhao, Shaohua Wang, Ying Zou, Joanna Ng, and Tinny Ng. Mining user intents to compose services for end-users. In *ICWS*, pages 348–355. IEEE, 2016.

[161] Yu Zhao, Shaohua Wang, Ying Zou, Joanna Ng, and Tinny Ng. Mining user intents to compose services for end-users. In *Web Services (ICWS), 2016 IEEE International Conference on*. IEEE, 2016.

[162] Yu Zhao, Shaohua Wang, Ying Zou, Joanna Ng, and Tinny Ng. Automatically learning user preferences for personalized service composition. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 776–783. IEEE, 2017.

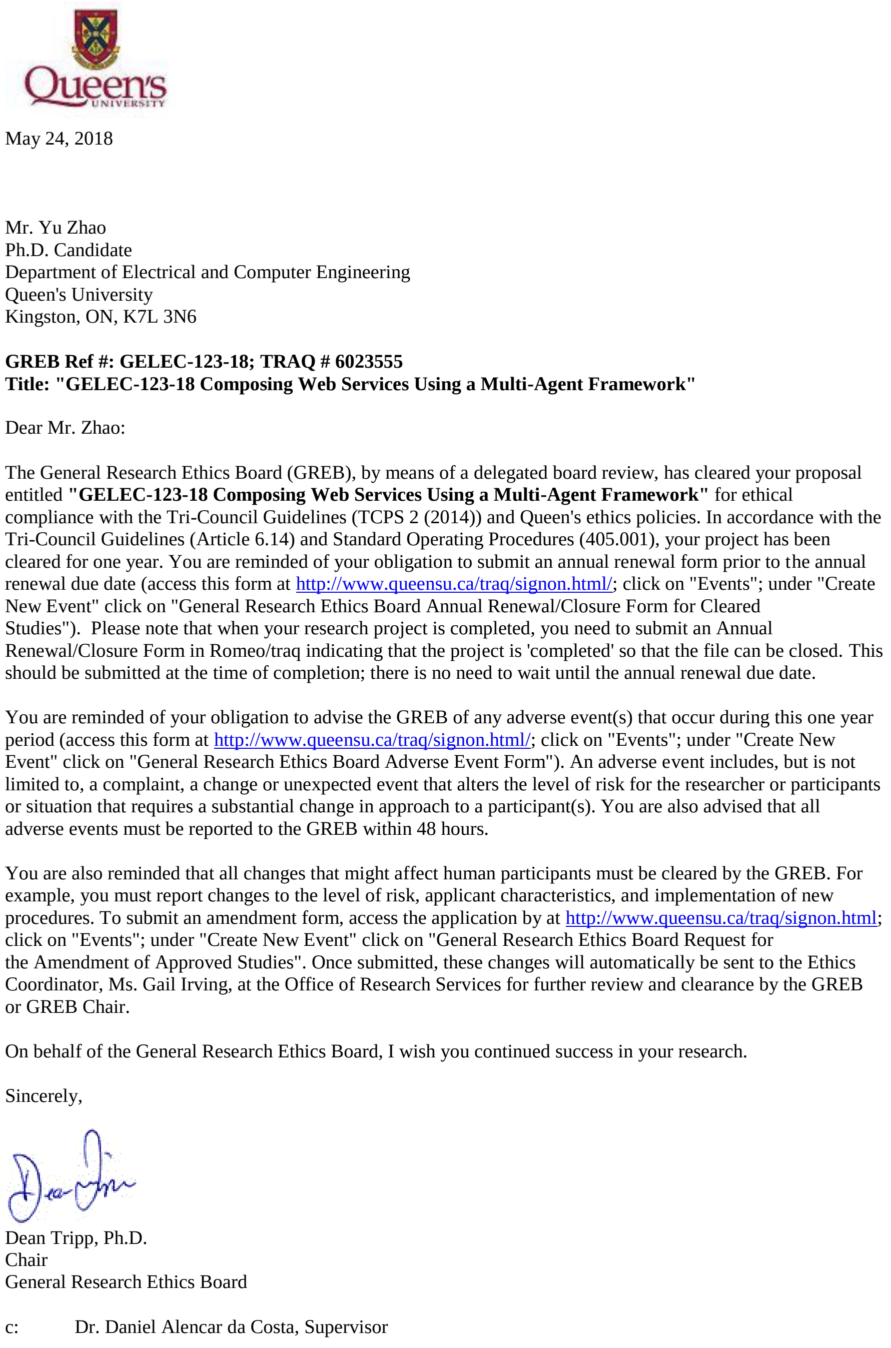
[163] Yu Zhao, Feng Zhang, Emad Shihab, Ying Zou, and Ahmed E Hassan. How are discussions associated with bug reworking?: An empirical study on open source projects. In *ESEM, ACM*, page 21, 2016.

[164] Yu Zhao, Ying Zou, Joanna Ng, and Daniel Alencar da Costa. An automatic approach for transforming iot applications to restful services on the cloud. In *ICSOC*. Springer, 2017.

[165] Lei Zheng, Vahid Noroozi, and Philip S Yu. Joint deep modeling of users and items using reviews for recommendation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 425–434. ACM, 2017.

Appendix A

Research Ethics Approval



May 24, 2018

Mr. Yu Zhao
Ph.D. Candidate
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6

GREB Ref #: GELEC-123-18; TRAQ # 6023555
Title: "GELEC-123-18 Composing Web Services Using a Multi-Agent Framework"

Dear Mr. Zhao:

The General Research Ethics Board (GREB), by means of a delegated board review, has cleared your proposal entitled **"GELEC-123-18 Composing Web Services Using a Multi-Agent Framework"** for ethical compliance with the Tri-Council Guidelines (TCPS 2 (2014)) and Queen's ethics policies. In accordance with the Tri-Council Guidelines (Article 6.14) and Standard Operating Procedures (405.001), your project has been cleared for one year. You are reminded of your obligation to submit an annual renewal form prior to the annual renewal due date (access this form at http://www.queensu.ca/traq/signon.html/; click on "Events"; under "Create New Event" click on "General Research Ethics Board Annual Renewal/Closure Form for Cleared Studies"). Please note that when your research project is completed, you need to submit an Annual Renewal/Closure Form in Romeo/traq indicating that the project is 'completed' so that the file can be closed. This should be submitted at the time of completion; there is no need to wait until the annual renewal due date.

You are reminded of your obligation to advise the GREB of any adverse event(s) that occur during this one year period (access this form at http://www.queensu.ca/traq/signon.html/; click on "Events"; under "Create New Event" click on "General Research Ethics Board Adverse Event Form"). An adverse event includes, but is not limited to, a complaint, a change or unexpected event that alters the level of risk for the researcher or participants or situation that requires a substantial change in approach to a participant(s). You are also advised that all adverse events must be reported to the GREB within 48 hours.

You are also reminded that all changes that might affect human participants must be cleared by the GREB. For example, you must report changes to the level of risk, applicant characteristics, and implementation of new procedures. To submit an amendment form, access the application by at http://www.queensu.ca/traq/signon.html; click on "Events"; under "Create New Event" click on "General Research Ethics Board Request for the Amendment of Approved Studies". Once submitted, these changes will automatically be sent to the Ethics Coordinator, Ms. Gail Irving, at the Office of Research Services for further review and clearance by the GREB or GREB Chair.

On behalf of the General Research Ethics Board, I wish you continued success in your research.

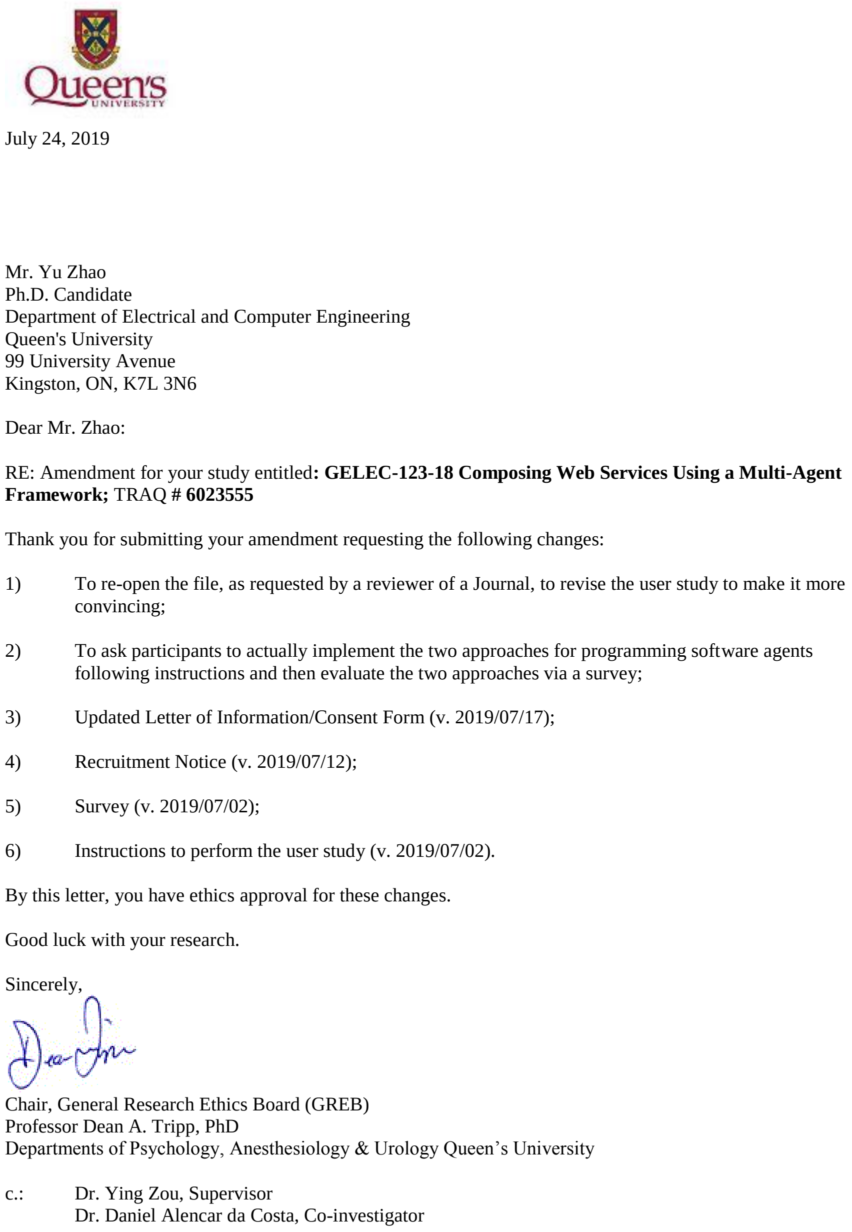
Sincerely,

[illegible]

Dean Tripp, Ph.D.
Chair
General Research Ethics Board

c: Dr. Daniel Alencar da Costa, Supervisor

Figure A.1: Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 3.



July 24, 2019

Mr. Yu Zhao
Ph.D. Candidate
Department of Electrical and Computer Engineering
Queen's University
99 University Avenue
Kingston, ON, K7L 3N6

Dear Mr. Zhao:

RE: Amendment for your study entitled**: GELEC-123-18 Composing Web Services Using a Multi-Agent Framework;** TRAQ **# 6023555**

Thank you for submitting your amendment requesting the following changes:

1) To re-open the file, as requested by a reviewer of a Journal, to revise the user study to make it more convincing;

2) To ask participants to actually implement the two approaches for programming software agents following instructions and then evaluate the two approaches via a survey;

3) Updated Letter of Information/Consent Form (v. 2019/07/17);

4) Recruitment Notice (v. 2019/07/12);

5) Survey (v. 2019/07/02);

6) Instructions to perform the user study (v. 2019/07/02).

By this letter, you have ethics approval for these changes.

Good luck with your research.

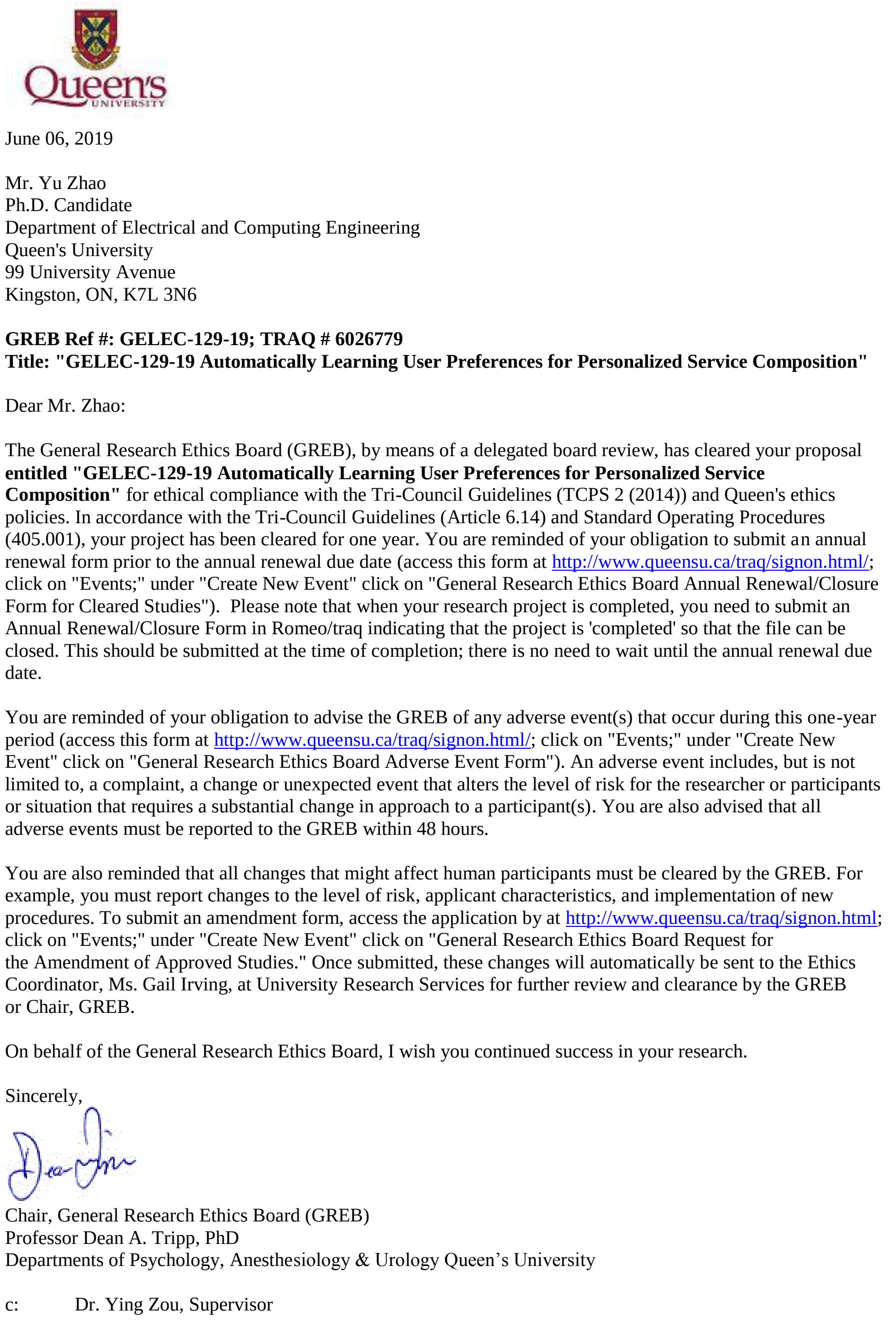
Sincerely,

[illegible]

Chair, General Research Ethics Board (GREB)
Professor Dean A. Tripp, PhD
Departments of Psychology, Anesthesiology & Urology Queen's University

c.: Dr. Ying Zou, Supervisor
Dr. Daniel Alencar da Costa, Co-investigator

Figure A.2: Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 3.



June 06, 2019

Mr. Yu Zhao
Ph.D. Candidate
Department of Electrical and Computing Engineering
Queen's University
99 University Avenue
Kingston, ON, K7L 3N6

GREB Ref #: GELEC-129-19; TRAQ # 6026779
Title: "GELEC-129-19 Automatically Learning User Preferences for Personalized Service Composition"

Dear Mr. Zhao:

The General Research Ethics Board (GREB), by means of a delegated board review, has cleared your proposal **entitled "GELEC-129-19 Automatically Learning User Preferences for Personalized Service Composition"** for ethical compliance with the Tri-Council Guidelines (TCPS 2 (2014)) and Queen's ethics policies. In accordance with the Tri-Council Guidelines (Article 6.14) and Standard Operating Procedures (405.001), your project has been cleared for one year. You are reminded of your obligation to submit an annual renewal form prior to the annual renewal due date (access this form at http://www.queensu.ca/traq/signon.html/; click on "Events;" under "Create New Event" click on "General Research Ethics Board Annual Renewal/Closure Form for Cleared Studies"). Please note that when your research project is completed, you need to submit an Annual Renewal/Closure Form in Romeo/traq indicating that the project is 'completed' so that the file can be closed. This should be submitted at the time of completion; there is no need to wait until the annual renewal due date.

You are reminded of your obligation to advise the GREB of any adverse event(s) that occur during this one-year period (access this form at http://www.queensu.ca/traq/signon.html/; click on "Events;" under "Create New Event" click on "General Research Ethics Board Adverse Event Form"). An adverse event includes, but is not limited to, a complaint, a change or unexpected event that alters the level of risk for the researcher or participants or situation that requires a substantial change in approach to a participant(s). You are also advised that all adverse events must be reported to the GREB within 48 hours.

You are also reminded that all changes that might affect human participants must be cleared by the GREB. For example, you must report changes to the level of risk, applicant characteristics, and implementation of new procedures. To submit an amendment form, access the application by at http://www.queensu.ca/traq/signon.html; click on "Events;" under "Create New Event" click on "General Research Ethics Board Request for the Amendment of Approved Studies." Once submitted, these changes will automatically be sent to the Ethics Coordinator, Ms. Gail Irving, at University Research Services for further review and clearance by the GREB or Chair, GREB.

On behalf of the General Research Ethics Board, I wish you continued success in your research.

Sincerely,

[illegible]

Chair, General Research Ethics Board (GREB)
Professor Dean A. Tripp, PhD
Departments of Psychology, Anesthesiology & Urology Queen's University

c: Dr. Ying Zou, Supervisor

Figure A.3: Research ethics approval from the General Research Ethics Board (GREB) for the user study performed in Chapter 7.