Data-Driven Quality Management of Online Service Systems

ZHU, Jieming

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
January 2016

Thesis Assessment Committee

Professor JIA Jiaya (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor YOUNG Fung Yu (Committee Member)

Professor CAO Jiannong (External Examiner)

Abstract of thesis entitled:

Data-Driven Quality Management of Online Service Systems

Submitted by ZHU, Jieming
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in January 2016

Nowadays, a wide variety of online services are emerging on the Internet, serving many aspects of our daily life including Web searching, social networking, online shopping, etc. Unlike traditional shrink-wrapped desktop software, most of these online services are designed as large-scale distributed systems operating on a 24x7 basis. The success of an online service is highly dependent on the delivered quality of service (QoS), which has direct effects on user experience and revenues of service providers. However, the ever-increasing scale and complexity of online service systems, coupled with the use of Web services, make it an enormous challenge for service providers to engineer their systems with high quality guarantees. In this thesis, we propose data-driven approaches to quality management of online service systems.

Firstly, we propose a Web service positioning framework for response time prediction of Web services. Response time, as one of the most important QoS attributes, is critical for many performance optimization tasks. But it is difficult to acquire comprehensive response time information in practice. By leveraging network coordinate systems, our Web service positioning framework achieves joint response time monitoring and prediction for Web services, which alleviates the limitation of data sparsity in existing approaches.

Secondly, we propose an online QoS prediction approach for runtime service adaptation. To meet QoS guarantees, online service

systems have to become resilient against the QoS variations of underlying Web services. Runtime service adaptation has been recognized as a key solution to achieve this goal. To make timely and accurate adaptation decisions, an effective and efficient QoS prediction approach is desired to obtain the QoS values of Web services. Inspired from the matrix factorization model used in recommender systems, we propose an adaptive matrix factorization approach to achieve online, accurate, and scalable QoS predictions.

Thirdly, we propose a privacy-preserving QoS prediction framework for Web service recommendation. To enable QoS-based Web service recommendation, existing studies make use of collaborative filtering techniques for personalized QoS prediction. However, the requirement to collect users' historical QoS data likely puts user privacy at risk, thus making them unwilling to contribute their usage data. To cope with this issue, we propose a simple yet effective privacy-preserving framework by applying data obfuscation techniques, and further develop two representative privacy-preserving QoS prediction approaches under this framework.

Lastly, we propose a "learning to log" framework to help developers make informed logging decisions during development. Logging is a common practice used for runtime service monitoring. However, currently there is a lack of rigorous specifications for developers to guide their logging behaviours. Logging has become an important yet tough decision which mostly depends on the domain knowledge of developers. Our "learning to log" framework is able to automatically learn the common logging practices from existing code repositories and further leverage them for actionable suggestions to developers.

In summary, this thesis targets at the use of data analytics to gain actionable insights from service-generated data and further enable data-driven quality management of online service systems. Extensive experiments on real-world datasets validate the effectiveness and efficiency of our proposed approaches.

論文題目：以數據為驅動的在線服務系統質量管理

作者　　：朱杰明

學校　　：香港中文大學

學系　　：計算機科學與工程學系

修讀學位：哲學博士

摘要　　：

現如今，互聯網上湧現出各式各樣的在線服務，并服務於我們生活的方方面面，包括網頁搜索，社交網絡，在線購物等等。與傳統的桌面軟件不同，大多數的在線服務都設計成 7x24 小時運行的大規模分佈式系統。在線服務的成功非常依賴于所提供的服務質量，因為它對用戶體驗和服務提供商的效益有直接影響。然而，在線服務系統日益增長的規模與複雜度，伴隨著 Web 服務的使用，使服務提供商構建有高質量保證的在線服務系統成為一個極大的挑戰。在本論文中，我們提出以數據為驅動的在線服務系統質量管理方法。

首先，我們提出了一個用於 Web 服務響應時間預測的 Web 服務定位框架。響應時間作為一項最重要的服務質量指標，對很多性能優化任務都很關鍵。但是，實際中很難獲取全面的響應時間信息。我們的 Web 服務定位框架利用網絡座標系統，能夠實現對 Web 服務響應時間的聯合監測及預測，並且緩解了已有方法中的數據稀疏性問題。

其次，我們提出了一種應用于服務實時自適應調整的在線服務質量預測方法。為了滿足服務質量保證，在線服務系統一定要對底層 Web 服務的服務質量變動有容忍性。服務實時自適應調整被認可為一種能達到這一目標的關鍵性方案。為了作出及時而準確的自適應調整決策，需要有效的服務質量預測來獲取 Web 服務的服務質量信息。受到推薦系統中使用的矩陣分解技術的啟發，我們提出了一種自適應的矩陣分解方法來實現在線、準確、及可擴展的服務質量預測。

再次，我們為 Web 服務提出了一種含隱私保護的服務質量預測方法。為了促進基於服務質量的 Web 服務推薦，已有研究使用協同過濾技術來完成個性化服務質量預測。但是，這些方法需要收集用戶的歷史服務質量數據，對用戶隱私造成威脅，所以用戶往往不願意貢獻他們的歷史數據。為了解決這一問題，我們基於數據混淆技術提出了一種簡單而有效的隱私保護框架，并根據這一框架開發了兩種具有代表性的含隱私保護的服務質量預測方法。

最後，我們提出了一個基於學習的日誌記錄框架來幫助程序員在程序開發過程中做出有效的日誌記錄決策。日誌記錄是一種用於實時服務監測的慣常做法。然而，目前缺乏嚴格的說明文檔來指導程序員如何進行日誌記錄。日誌記錄已經成為開發過程中一個重要但困難的決策，往往依賴於程序員的領域知識。我們提出的基於學習的日誌記錄框架能夠從已有的代碼庫里自動化地學習出常用的日誌記錄規則，然後利用它們來為程序員提供可行的日誌記錄建議。

綜上所述，本論文旨在利用數據分析技術來獲取可行的信息，進而達到以數據為驅動的在線服務系統的質量管理的目的。大量基於真實數據的實驗驗證了我們提出方法的有效性和高效性。

Acknowledgement

I feel highly privileged to take this opportunity to express my sincere gratitude to the people who have been instrumental and helpful on my way to pursuing my PhD degree.

First and foremost, I would like to thank my supervisor, Prof. Michael R. Lyu, for his kind supervision of my PhD study at CUHK. He has provided inspiring guidance and incredible help on every aspect of my research. From choosing a research topic to working on a project, from technical writing to paper presentation, I have learnt so much from him not only on knowledge but also on attitude in doing research. I will always be grateful to his advice, encouragement and support at all levels.

I am grateful to my thesis assessment committee members, Prof. Jiaya Jia and Prof. Fung Yu Young, for their constructive comments and valuable suggestions to this thesis and all my term reports. Great thanks to Prof. Jiannong Cao from The Hong Kong Polytechnic University who kindly served as the external examiner for this thesis.

I would like to thank my oversea supervisor, Prof. Kin K. Leung, for his support of my visit to Imperial College London. During this visit, Prof. Leung and his student, Shiqiang Wang, have provided insightful ideas and constructive feedback to my research. I also thank Dr. Qiang Fu, my mentor during the internship at Microsoft Research Aisa. With his kind guidance, we have published two great papers.

I thank Zibin Zheng, Yangfan Zhou, Haiqin Yang, Yilei Zhang,

Yu Kang, and Pinjia He, for their valuable guidance and contribution to the research work in this thesis. I am also thankful to my other groupmates, Chao Zhou, Qirun Zhang, Baichuan Li, Shouyuan Chen, Guang Ling, Chen Cheng, Hongyi Zhang, Shenglin Zhao, Junjie Hu, Hui Xu, Cuiyun Gao, Yuxin Su, and Jianlong Xu, who gave me encouragement and kind help.

My special thanks go to my dear friends, Huanhuan Wu, Lin Ma, Yuxiang Shi, Junjie Ye, Zhiqing Liu, Jiali Xiang, Ning Zou, Yan Wu, Yanyan Xu, Silu Huang, Jia Wang, and many others at CUHK for all the wonderful memories of cooking, hiking, and travelling we made together over these years. Life would not be so enjoyable without them.

Last but not least, I would like to thank my dear family. Without their deep love and constant support, this thesis would never have been completed.

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents our research towards data-driven quality management of online service systems, which is currently an important field of study and practice in software development and maintenance. We provide a brief overview of the research problems under study in Section 1.1, and highlight the main contributions of this thesis in Section 1.2. Section 1.3 outlines the thesis structure.

1.1 Overview

Nowadays, online services play an indispensable role in our daily life, by providing a wide variety of services including Web searching, social networking, online chatting, online shopping, etc. Unlike traditional shrink-wrapped desktop software, most of these online services are designed as large-scale distributed systems operating on a 24x7 basis to serve millions of users globally. The success of an online service is highly dependent on the delivered quality of service (QoS), including availability, reliability, responsiveness, etc. QoS has direct effects on user experience and revenues of service providers. Any service outage or degradation of service quality can lead to user dissatisfaction and cause significant revenue loss. For example, it is reported that in August 2013, Amazon.com probably lost $4.8 million after going down for 40 minutes [5], while Google's

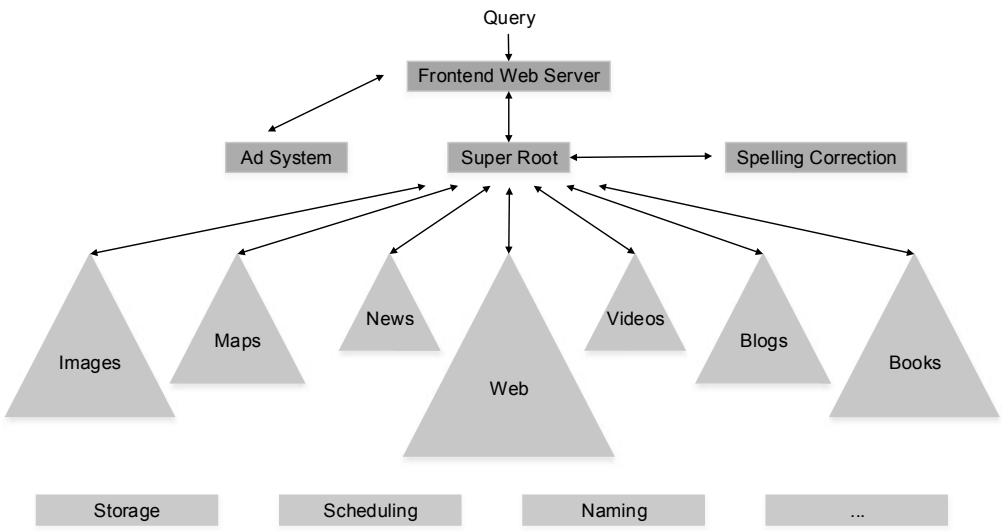


Figure 1.1: A Prototype of Google Search Service

5-minute outage caused about $545K revenue loss and 40% drop in global website traffic [11]. The high cost of such service incidents reveals the critical importance of quality management of online service systems.

With the ever-increasing demand to enable more functionality, online service systems are becoming more and more large-scale in size and complex in structure. To allow for flexible development and maintenance, large-scale online service systems are commonly built on service-oriented architecture, where invocations of underlying component services are applied to fulfill complex application logic. Even more, some services may be developed by third parties and be provisioned as black-box Web services, which have no access to source code. As a typical example of online services, Figure 1.1, provided by Google [4], illustrates the prototype structure of Google search. As shown in the figure, the Google search service actually comprises a wide variety of component services. Infrastructure services such as storage, scheduling, and naming are managed to support the delivery of various application services including images, maps, news, Web, videos, blogs, and books. A super root

service further coordinates these component services and provides the final aggregated search results to users. Moreover, a spelling correction service is employed to enhance query inputs of users, while an ad system is engaged to present relevant ads to users. The whole online service system involves the composition of all these component services.

On the other hand, cloud computing has gained much popularity for provisioning a pool of computational resources on demand to deliver various large-scale online services over the Internet. Many cloud providers like Amazon, Google and Microsoft have built large data centers in geographically distributed locations. For example, Amazon EC2[1] currently provisions cloud services over nine geographically dispersed regions, where users have options to deploy their applications in different data centers from Virginia, Oregon, California, Ireland, Singapore, Tokyo, Sydney, São Paulo and GovCloud. With the prevalence and benefit of cloud computing, more and more online services (e.g., Twitter, Netflix) are migrated to cloud computing platforms to achieve reliability and better performance. To better serve millions of users globally, component services of a large-scale online service system have to be deployed across multiple geographically distributed data centers (*e.g.*, [24, 148, 176]), so that resources are located closer to end-users. The ever-increasing scale and complexity of online service systems, coupled with the geographical-distribution of different component services, make it a notoriously challenging task for service providers to engineer their systems with high quality guarantees.

With all these new challenges, traditional engineering techniques, such as software testing and modelling in the lab, would not suffice to deal with service problems in production settings. In many cases, it is extremely difficult or even impossible to simulate production environment in the lab. Therefore, even with heavy lab testing, dynamic production environment may still lead to service failure or

[1] `http://aws.amazon.com/ec2`

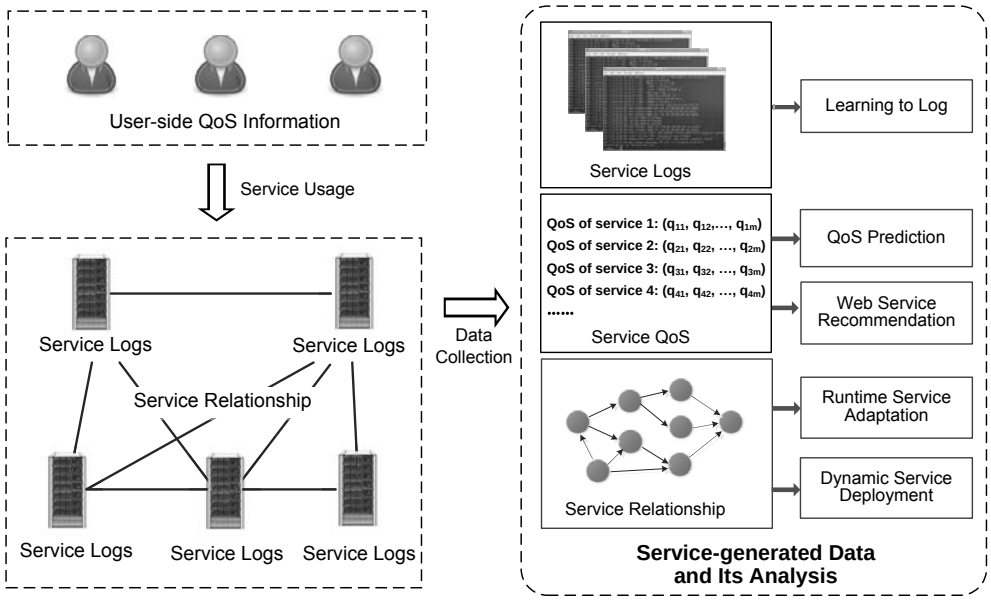


Figure 1.2: An Overiew of Data-Driven Service Quality Management

degradation of service quality. Besides, it is also hard to predict the effects of potential performance optimizations (e.g., dynamic service deployment) based on lab modelling, without enough performance data acquired in the field.

In this context, data-driven service quality management has recently emerged as a promising solution to deal with practical service quality problems. Online service systems, with the continuous operation, are generating a variety of service data. As illustrated in Figure 1.2, some typical examples of service-generated data include service logs, quality of service (QoS) information of Web services, service dependency graphs [169]. These data contain a wealth of valuable information that can aid in service quality management. For instance, service logs are usually used as a principal tool of runtime service monitoring to perform anomaly detection and failure diagnosis tasks. QoS information, on the other hand, is critical to many performance optimization tasks, such as latency-aware service deployment, QoS-aware fault tolerance of Web services, QoS-driven runtime service adaptation, and so on. Moreover, service relation-

ships, which can be denoted as service dependency graphs, also play an important role in many tasks such as dynamic service deployment and fault localization. As a result, the objective of this thesis is to establish a data-driven engineering framework, under which data-driven approaches are leveraged to gain actionable information and uncover powerful insights for service quality management of online service systems. The research of this thesis comprises four parts. In the first three parts, we focus on the study of QoS prediction of Web services. In particular, in the first part, we present response time prediction with a case study on latency-aware service deployment. In the second part, we propose online QoS prediction for runtime service adaptation. In the third part, privacy-preserving QoS prediction is investigated in the scenario of QoS-aware Web service recommendation. Finally, we propose a learning to log framework for runtime service monitoring in the fourth part.

1.2 Thesis Contributions

In this thesis, we make contribution to data-driven quality management of online service systems in the following ways:

1. **Response time prediction of Web services**

   Response time is one of the most important QoS attributes, which is critical for many performance optimization tasks. In practice, it is difficult to acquire comprehensive response time information from users due to the large overhead for active measurement. Recent work proposes the use of collaborative filtering for QoS (and response time) prediction. But these methods suffer from the data sparsity of available historical QoS data, which greatly degrades the prediction accuracy. To address this problem, we propose a Web service positioning (WSP) framework [174] for response time prediction, by combining both advantages of collaborative filtering and

network coordinate systems. To evaluate our approach, we collect a response time dataset from real-world Web services, which involves 359,400 response time records between 200 users and 1,597 Web services. The experimental results show that our WSP framework alleviates the data sparsity problem and significantly enhances the prediction accuracy. Besides, this reusable research dataset is publicly released to allow for reproducing our experiments and to promote future research.

2. **Online QoS prediction of Web services**

   To meet QoS guarantees, online service systems have to become resilient against the QoS variations of underlying Web services. Runtime service adaptation has been recognized as a key solution to achieve this goal. To make timely and accurate adaptation decisions, effective QoS prediction is desired to obtain the QoS values of Web services. To achieve this goal, we propose an online, accurate, and scalable QoS prediction approach [172]. The main contributions are three-fold: (1) This is the first work to address the problem of QoS prediction to guide candidate service selection for runtime service adaptation; (2) A novel QoS prediction approach, adaptive matrix factorization (AMF), is proposed by employing techniques of data transformation, online learning, and adaptive weights; (3) Comprehensive experiments are conducted based on a real-world large-scale QoS dataset of Web services to evaluate our proposed approach in terms of accuracy, efficiency, and scalability. Moreover, for ease of reproducing our results, we publicly release our source code and dataset of our study.

3. **Privacy-preserving QoS prediction of Web services**

   To facilitate QoS-based Web service recommendation, existing studies employ collaborative filtering techniques for personalized QoS prediction. These approaches leverage partially observed QoS values from users to make predictions on the

unobserved QoS values. However, the requirement to collect users' QoS data likely puts user privacy at risk, thus making them unwilling to contribute their historical usage data. To address the privacy issue in developing practical Web service recommender systems, we propose a simple yet effective privacy-preserving framework for QoS prediction of Web services [173]. Specifically, we make the following key contributions: (1) This is the first work to cope with the privacy issue for QoS-based Web service recommendation; (2) We propose a privacy-preserving collaborative filtering framework, and further develop two representative privacy-preserving QoS prediction approaches, P-UIPCC and P-PMF, under this framework; (3) We conduct experiments on a real-world large-scale QoS dataset of Web services to evaluate the effectiveness of our proposed approaches. Moreover, we have publicly released both of our source code and dataset for future study.

4. **Learning to log for runtime service monitoring**

   Logging is a common programming practice of practical importance to collect system runtime information for postmortem analysis. However, in current practice, there is a lack of rigorous specifications for developers to guide their logging behaviours. Logging has become an important yet tough decision which mostly depends on the domain knowledge of developers. To reduce the effort on making logging decisions, we propose a "learning to log" framework [171], which aims to provide informative guidance on logging during development. As a proof of concept, we provide the design and implementation of a logging suggestion tool, LogAdvisor, which automatically learns the common logging practices on where to log from existing logging instances and further leverages them for actionable suggestions to developers. We evaluate LogAdvisor on two industrial software systems from Microsoft

and two open-source software systems from Github (totally 19.1M LOC and 100.6K logging statements). The encouraging experimental results, as well as a user study, demonstrate the feasibility and effectiveness of our logging suggestion tool.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2**

  In this chapter, we review some background knowledge and related work on data-driven quality management of online service systems. First, we briefly introduce online service systems, with focuses on the characteristics of service-oriented architecture, the use of Web service, and the importance of quality of service. We then review some related studies on data-driven service quality management, which is the main objective of this thesis. Finally, we provide some fundamental background on the techniques of data analytics that we have used in our work.

- **Chapter 3**

  In this chapter, we present a Web service positioning framework for response time prediction of Web services. The framework is based on the use of network coordinate systems, and also incorporates the historical data that is originally used by existing collaborative filtering based approaches. More specifically, we first introduce the response time prediction problem in Section 3.1, and then provide an overview of Web service positioning framework in Section 3.2. The detailed response time prediction approach based on WSP framework is further described in Section 3.3, which comprises four steps: (1) landmark coordinate computation, (2) Web service coordinate computation, (3) service user coordinate computation,

and (4) response time prediction. Section 3.4 presents the data collection from real-world Web services, and provides the evaluation results of WSP approach compared to existing approaches. Furthermore, a case study of latency-aware service deployment is conducted in Section 3.5, which demonstrates the practical use of response time prediction. Finally, Section 3.6 summarizes this chapter.

- **Chapter 4**

  In this chapter, we present online QoS prediction of Web services, which is crucial for accurate and timely decision making in runtime service adaptation. More specifically, we first introduce the problem and motivation of online QoS prediction in Section 4.1, and then provide the framework of QoS-driven service adaptation in Section 4.2. The detailed online QoS prediction algorithm is further described in Section 4.3, which extends conventional matrix factorization into an online, accurate, and scalable model, namely adaptive matrix factorization (AMF). In Section 4.4, we conduct evaluation of AMF based on a real-world large-scale QoS dataset of Web services, with detailed results provided in terms of accuracy, efficiency, and scalability. In addition, we perform a case study on dynamic request routing in Section 4.5 to demonstrate the practical use of online QoS prediction for runtime service adaptation. Finally, Section 4.6 summarizes this chapter.

- **Chapter 5**

  In this chapter, we present a simple yet effective privacy-preserving Web service recommendation framework and further develop two representative privacy-preserving QoS prediction approaches under this framework. In particular, we introduce the privacy issue in QoS-based Web service recommendation in Section 5.1, and present the proposed privacy-preserving Web service recommendation framework in Section

5.2. By use of data obfuscation techniques, in Section 5.3 we describe two representative privacy-preserving QoS prediction approaches, namely P-UIPCC and P-PMF. Section 5.4 provides the evaluation of P-UIPCC and P-PMF with comparison to UIPCC and PMF, two counterpart QoS prediction approaches. Finally, Section 5.5 summarizes this chapter.

- **Chapter 6**

  This chapter presents our work of learning to log for runtime service monitoring. Learning to log is a data-driven framework that we propose, with the aim to provide informative suggestions to developers by automatically learning common logging practices from existing logging instances. More specifically, in Section 6.1, we provide some key observations that motivate our work. We further present our learning to log framework with implementation details of our logging suggestion tool LogAdvisor in Section 6.2. Section 6.3 provides the comprehensive evaluation of LogAdvisor conducted on two industrial online service systems from Microsoft and two open-source software systems from Github. We also provide a user study with developers to validate the effectiveness of LogAdvisor in Section 6.4. We discuss the limitations in Section 6.5 and conclude this chapter in Section 6.6.

- **Chapter 7**

  The last chapter summarizes this thesis and provides some future directions that deserve for further exploration.

To make each chapter self-contained, we may briefly reiterate the critical contents, such as model definitions and motivations, in some chapters.

□ **End of chapter.**

Chapter 2

Background Review

This chapter briefly reviews some background knowledge and related work of our research, including online service systems in Section 2.1, data-driven service quality management in Section 2.2, and data analytics in Section 2.3.

2.1 Online Service Systems

Online services are typically used to describe the provision of online access by users to software services and data over the Internet. Nowadays, online services play an indispensable role in our daily life, by providing a wide variety of services including Web searching, social networking, online chatting, online shopping, email, online entertainment, e-banking, e-health, just to name a few [147]. Unlike traditional shrink-wrapped desktop software, most of these online services are designed as large-scale distributed systems operating on a 24x7 basis to serve millions of users globally. Quality of service (QoS) is of vital importance to the successful delivery of online services. A high-quality online service can offer pleasant user experience to service customers, and thus contributes to revenue generation by keeping old customers as well as attracting new ones. In contrary, any service outage or degradation of service quality can lead to user dissatisfaction and cause significant revenue

loss. Furthermore, building high-quality online service systems can save the cost of service maintenance. As a result, there is a high demand for effective quality management of online service systems.

However, as the user demand to enable more software features and functions is ever increasing, modern online service systems are becoming very complex in structure, large-scale in size, and highly dynamic during operation, which make them more vulnerable to service incidents (*e.g.*, service outage and QoS degradation) compared to traditional software systems. On one hand, large-scale online service systems are commonly built on service-oriented architecture, where some components may rely on invocations of underlying Web services to fulfill complex application logic. Some of them may be developed and maintained by third-parties, often without any source code, and invoked over unpredictable Internet connections. On the other hand, cloud computing has gained increasing prevalence in recent years for hosting and delivering various online services over the Internet. As an online service system scales up, it is desired to provision the online service not only from single data centers but also spanning across multiple geographically distributed data centers, so as to extend functions and services with global expansion [148]. The use of Web services, coupled with the geographical-distribution of different service components, makes it a significant challenge for service providers to engineer their systems with high quality guarantees, which thus motivates our research in this thesis.

2.1.1 Service-Oriented Architecture

Service-oriented architecture (SOA) is an increasingly popular architectural style for software system development [18], in which component services are coordinated in a loosely-coupled way to fullfil complex application logic. Typically, a service-oriented system is built on SOA with a number of component services and a workflow. Individual component services are composed together to

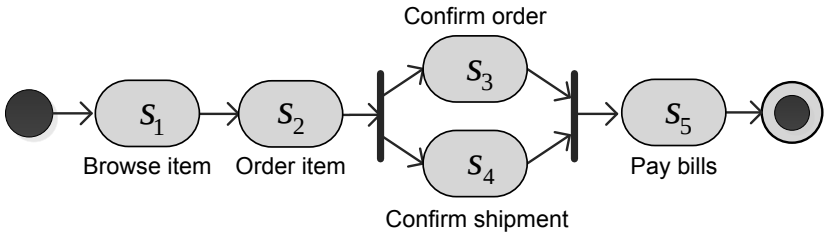


Figure 2.1: An Illustrative Example of Service-Oriented System

form a higher-level functionality, where the interaction relationship between these component services are defined by the workflow. The workflow can be specified in BPEL (Business Process Execution Language) and runs on a BPEL engine, like Apache ODE[1].

Figure 2.1 provides a concrete example of a service-oriented system, which shows a prototype application for online shopping. It comprises a simplified workflow with five abstract tasks. A workflow may consist of sequential, branch, parallel, and loop compositional structures [143]. For simplicity, we only consider sequential and parallel structures in this example. In detail, a customer begins with item search and browsing (s_1). Then the user can make an order (s_2) and the request is sent to the corresponding supplier to check the availability of the item. The supplier reserves the item for the customer and waits for further confirmation. Next the customer confirms the order (s_3), and confirms the shipment method (s_4). Once the order is confirmed, a payment transaction via online payment service (*e.g.*, paypal) will be launched (s_5). After the payment, the item purchase is completed.

Service-oriented architecture promotes flexibility and resilience to changes, as service invocations can be accomplished without considering how the underlying services have been implemented. Changes to the implementation details by the service provider will not affect the service user, while the service interface remains stable. In addition, the use of SOA makes it feasible to build new business

[1] `http://ode.apache.org`

functionality based on existing services not from scratch, which thus enables features of high re-usability, fast development, and reduced cost. Due to these attractive features provided by SOA, many online service systems are currently designed in the SOA style. For example, Amazon's e-commerce platform is built on SOA by composing hundreds of component services hosted world-wide to deliver functionalities ranging from item recommendation to order fulfillment to fraud detection [51]. In particular, we target at online service systems built on SOA in this thesis.

2.1.2 Web Service

Web service is an essential building block of service-oriented architecture. It is typically designed as a black-box software component, providing a well-defined functionality to users over the Internet via some standard interfaces (*e.g.*, XML-RPC, SOAP, REST) [19]. Recent advances in cloud computing as well as dramatic increase in user demand promote the abundance of available Web services. Typical examples range from simple operations like retrieving currency exchange rates, to various Web APIs such as Google Map API and Dropbox API, to complex processes running customer relationship management (CRM) systems. The use of such Web services, integrated with SOA, significantly eases the development and maintenance of online service systems.

According to the report from *seekda.com*[2] in 2012, there are totally 7,739 service providers and over 28,600 public Web services on the Internet. With the widespread of Web services, more and more providers begin to offer Web services even with equivalent or similar functionalities. For example, both providers, CDYNE.COM and WebserviceX.NET, offer equivalent Web services for querying global weather information [172]. In this context, quality of service has become an important factor in distinguishing those functionally-

[2] `http://seekda.com`

equivalent Web services. There is an urgent need for QoS assessment of Web services.

2.1.3 Qualtiy of Service

To build high-quality online applications, it is a critical task for application designers to select appropriate services that fulfill both functional requirements and non-functional requirements. While functional requirements specify what a service does, non-functional requirments are conerned with quality-of-service (QoS) attributes. The importance of QoS can be also well identified by its various usages in the literature, such as QoS-based service discovery [151], QoS-based service selection [143, 25], QoS-based service adaptation [40, 67], and so on. To be specific, we identify the most representative QoS attributes as follows:

1. **Availability**: This attribute measures the percentage of time that a Web service is normally operating during a certain time interval.
2. **Reliability**: This attribute measures the percentage of successful service invocations out of all the service invocations requested [158].
3. **Response time**: This attribute measures the time duration between a user sending out a service request and receiving a response.
4. **Failure probability**: This attribute measures the percentage of failed service invocations out of all the service invocations requested, which equals to $1 - reliability$.
5. **Throughput**: This attribute measures the data transmission rate (*e.g.*, kbps) of a service invocation.
6. **Price**: This attribute measures the cost that a user needs to pay for a service invocation.

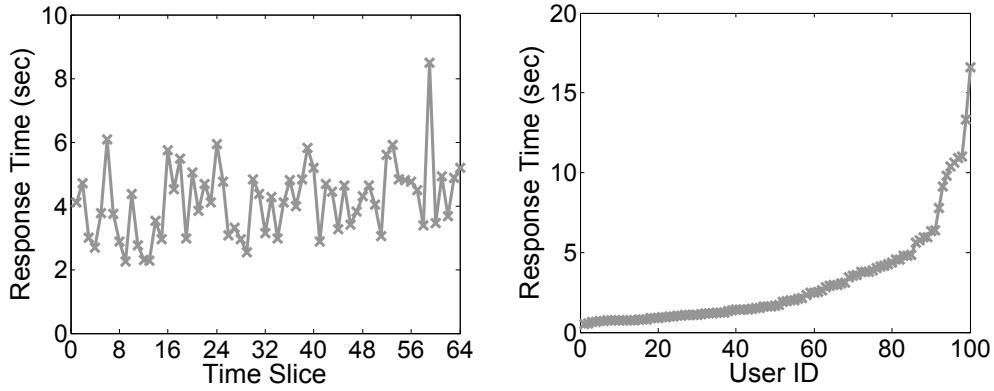
7. **Popularity**: This attribute measures how often a service is requested by users. For example, we can measure popularity by the number of invocations of a service during a certain time interval.

8. **Reputation**: This attribute measures the certainty that a service behaves as expected from the service description. The reputation of Web service is concerned with many aspects such as user ranking, compliance, and verity.

In general, the set of QoS attributes can be divided into two types: positive QoS attributes (*e.g.*, availability, throughput) and negative QoS attributes (*e.g.*, response time) [25]. Values of positive attributes need to be maximized, whereas values of negative attributes should be minimized.

Ideally, the QoS values of services can be directly specified in the service-level agreements (SLAs) by service providers. However, except attributes like price, most of the QoS values are time-varying, which at a large extent depends on the network conditions and server status. Consequently, many QoS attributes should be evaluated at runtime. In particular, evaluating real-time QoS values is notoriously difficult due to the following characteristics.

- **Time-varying**: QoS attributes like response time and throughput are time-varying. For instance, due to the impact of varying server workload and dynamic network conditions, QoS delivered to users may vary widely during different time periods. Figure 2.2(a) depicts a real-world example of the response times of a user at Pittsburgh (IP: 12.108.127.138) invoking a Web service located at Iran (`http://profiles.roshd.ir/security.asmx?WSDL`) over 64 consecutive time slices (at 15-minute interval), where the data is extracted from a real-world QoS dataset collected in [152]. The curve confirms that the user-perceived response time fluctuates around an



(a) Response Time v.s. Time Slice (b) Response Time v.s. User ID

Figure 2.2: Real-world Response Time Observations

average QoS value along the time. Therefore, some QoS attributes should be evaluated at runtime.

- **User-specific**: The online service systems built on SOA are typically loosely coupled, where users and services are distributed at different locations. For instance, the shopping cart services provided by Amazon.com [51] may be hosted at different data centers located worldwide. With the increase of geographic distribution of services, the impact of the network on user-perceived QoS becomes non-negligible. Thus, users from different locations may observe different QoS values even on the same service. Figure 2.2(b) confirms such observation by a real-world example, which presents the response times (sorted in ascending order) perceived by 100 randomly-selected users that invoke the same service. The large variation of the curve implies that QoS attributes like response time are user-specific and should be evaluated independently from each user side.

- **Measurement overhead**: As mentioned before, real-time QoS can only be observed at runtime. However, it is infeasible for

each user to actively measure the real-time QoS values of all the candidate services by periodical invocations, since it will incur prohibitively large overhead. First, some Web service invocations may be charged which heavily increase the cost of service users. Second, it is time-consuming and resource-consuming to invoke all the services periodically, as there may exist a huge number of candidate services.

Due to the unique characteristics of QoS attributes, it becomes a critical challenge to obtain real-time QoS data without causing much overhead. In this context, QoS prediction has been emerged as a key solution to estimate the unknown QoS values by employing the historical usage data, while requiring no additional service invocations.

2.2 Data-Driven Service Quality Mangement

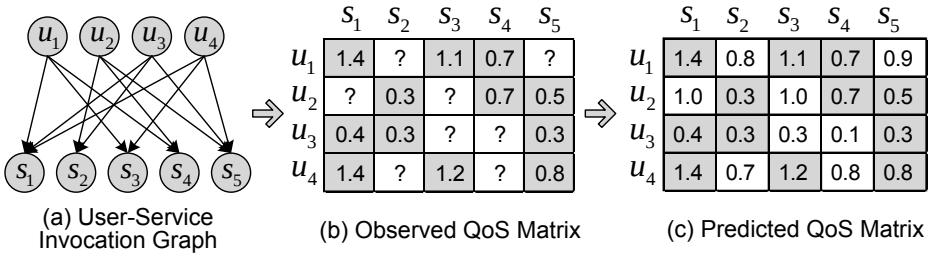
In recent years, there is a new trend towards data-driven software system management. Various data are generated throughout the software lifecycle, including source code, revision histories, bug reports, runtime logs, and so forth. These data contain a wealth of information that can aid software system management tasks. The goal to explore the potential of such rich data motivates a large body of research topics related to mining software engineering data[133, 134], software intelligence [63], and software analytics [145, 146]. Typical examples include defect prediction [77], method specifications mining [99], bug localization [170], system failure diagnosis[47], log analysis [58, 98], and so on. All these studies leverage techniuqes of data analytics (*e.g.*, data mining and machine learning) to gain actionable information and uncover powerful insights for better software development and maintenance. Following this line of research, our work in this thesis focuses mainly on data-driven quality management of online service systems, with a set of unique research problems.

2.2.1 QoS Management of Web Services

Currently, Web services have widespread use in building loosely-coupled distributed systems, especially for large-scale online service systems. Web services aim for black-box service delivery to users, thus must be designed for both availability and stability. Quality of service (QoS), as a widely-used criterion to evaluate the non-functional aspects of Web services, has become a key to the success of service provision. Any QoS degradation may harm the reputation of a Web service, and further cause revenue loss of the service provider. With the prevalence of Web services, QoS management has become a task of vital importance for Web services. In the following, we will introduce some specific issues towards this goal.

QoS Evaluation and Prediction

QoS evaluation aims to measure the service quality delivered to users. With comprehensive QoS evaluations on Web services, service users are able to obtain accurate information on the non-functional characteristics of all the Web service candidates, based on which they can examine whether a service meets their requirements or make good selection among the services with identical or similar functionalities. As described in Section 2.1.3, to acquire the QoS information needed, it is necessary to perform QoS evaluation at runtime from user side. To achieve so, the most straightforward way is to directly measure the QoS value of each Web service. However, it is usually infeasible in practice due to the following reasons: 1) Active measurements of Web services can bring large overhead to users, especially when there are a large number of candidate Web services. 2) Most of service invocations may not be free, which further increases the cost of service users. 3) Intensive service invocations for measurement will also consume additional resources of service providers. As a result, effective and efficient QoS prediction approaches are highly desired to estimate the QoS



	s_1	s_2	s_3	s_4	s_5
u_1	1.4	?	1.1	0.7	?
u_2	?	0.3	?	0.7	0.5
u_3	0.4	0.3	?	?	0.3
u_4	1.4	?	1.2	?	0.8

(b) Observed QoS Matrix

	s_1	s_2	s_3	s_4	s_5
u_1	1.4	0.8	1.1	0.7	0.9
u_2	1.0	0.3	1.0	0.7	0.5
u_3	0.4	0.3	0.3	0.1	0.3
u_4	1.4	0.7	1.2	0.8	0.8

(c) Predicted QoS Matrix

Figure 2.3: An Illustrative Example of QoS Prediction

values of Web services without the need of any real-world service invocations.

Inspired from the rating prediction problem encountered in recommender systems (*e.g.*, movie recommendation in Netflix), recent studies [161, 150, 45] suggest the use of collaborative filtering for QoS prediction. By collaborative filtering [123], the observed user ratings are leveraged to learn user preferences on the unrated movies or items and further make predictions on the unknown ratings. As with the user-movie rating matrix collected in a movie recommender system, users invoking services can produce a user-service QoS matrix with respect to each QoS attribute. We denote a QoS matrix by R, whose entry R_{ij} represents the observed QoS value (*e.g.*, response time) of user u_i invoking service s_j. Figure 2.3(b) illustrates a QoS matrix with four users (u_1, ..., u_4) and five services (s_1, ..., s_5), produced by the user-service invocation graph in Figure 2.3(a). In practice, the QoS matrix is very sparse (*i.e.*, most of the entries are unknown), since each user usually invokes only a few services. As shown in Figure 2.3(b), the grey entries are observed QoS values (*e.g.*, $R_{11} = 1.4$) and the blank entries are unknown QoS values (*e.g.*, $R_{12} = ?$). As a result, the QoS prediction problem can be modelled as a collaborative filtering problem. The intuition is that similar users may observe similar QoS values on the same service because of their shared network, service resources, etc. Figure 2.3(c) shows the predicted QoS matrix from the observed QoS matrix in Figure 2.3(b), where the unknown values are approximately

reconstructed.

In recent literature, many efforts have been invested to achieve the goal of QoS prediction. According to the techniques and data used for QoS prediction, we broadly classify existing studies into the following categories:

- **Neighbourhood-based QoS prediction**: This type of collaborative filtering approaches use the observed QoS data to compute the similarity values between users or services, and further leverage them for QoS prediction. Typical examples include user-based approaches (*e.g.*, UPCC [115]) that leverage the QoS information of similar users for prediction, item-based approaches (*e.g.*, IPCC [111]) that employ the QoS information of similar items (*i.e.*, services) for prediction, and their hybrids (*e.g.*, UIPCC [161, 162]) that combine user-based and item-based approaches together for accuracy improvement. These approaches are easy to implement, but they fail to deal with the data sparsity problem, which limits their performance in practice.

- **Model-based QoS prediction**: Model-based collaborative filtering approaches provide a predefined model to fit the observed QoS data, and then the trained model can be used to predict the unknown QoS values. Matrix factorization (*e.g.*, PMF [109]) is one of the most popular model-based CF approaches, which was first introduced to address the QoS prediction problem in [79]. Matrix factorization model handles the sparsity problem well and usually achieves better performance than neighbourhood-based approaches. More recently, some studies such as CloudPred [152], NIMF [164], and LN-LFM [138] integrate neighbourhood-based and model-based CF approaches to further improve prediction accuracy.

- **Ranking-based QoS prediction**: Both neighbourhood-based and model-based QoS prediction approaches target at estimat-

ing every QoS value in a user-service QoS matrix as accurately as possible. However, accurate QoS value predictions do not necessarily result in accurate QoS ranking of Web services. Therefore, some recent studies [166, 165] propose to address the QoS ranking prediction problem instead of direct QoS value prediction. The ranking prediction results enable users to make natural selection of top-ranking services.

- **Context-aware QoS prediction**: Most of previous studies only make use of historical QoS data for QoS predictions. However, the historical QoS is usually sparse in practice, because each user only invokes a few out of all the available Web services at each time. The data sparsity problem limits the performance of existing approaches. More recent studies have attempted to alleviate the data sparsity problem by employing additional contextual information, such as location information [45], time information [137, 172], and reputation information [105]. It has been shown that context-aware QoS prediction approaches usually provide better prediction accuracy than previous approaches.

The QoS information obtained by QoS prediction approaches can be employed in a set of tasks for QoS management of Web services, such as Web service selection [28, 25, 72, 143], Web service recommendation [43, 44, 45, 162], fault tolerance of Web services [159, 168, 160], runtime service adaptation [40, 39, 172], and so on.

QoS-based Web Service Selection and Recommendation

QoS-based Web service selection and recommendation has recently attracted much attention from the service computing community, for providing a promising way to help users select high-quality services out of all the candidate services according to the user-perceived QoS values.

Specifically, Web service selection aims to optimize the overall QoS of a given composite service by selecting proper component services from a set of candidate Web services with different QoS values. Because QoS is a multi-dimensional vector with different QoS attributes, including availability, response time, and etc. Optimize one QoS attribute may lead to degradation of another. Zeng et al. [143] propose a QoS-aware middleware for Web service selection, in which integer programming is applied to deriving the optimal solution. Alrifai et al., further improve the efficiency of QoS-based Web service selection by either combining global optimization with local selection [25], or applying skyline operators for fast computation [26]. More recent work [64] proposes the use of iterative multi-attribute combinatorial auction, which achieves the current state-of-the art results in QoS-based Web service selection.

On the other hand, Web service recommendation [82] aims to leverage recommendation techniques to help users find the appropriate service that they need timely. A typical example is the application of collaborative filtering for QoS prediction, which thus facilitates QoS-based Web service recommendation [162]. More recent studies further consider location-aware Web service recommendation [45], time-aware Web service recommendation [127, 150], domain-aware Web service recommendation [132], topic-aware Web service recommendation [78].

To enable effective QoS-based selection and recommendation, a major challenge lies in the difficulty in achieving accurate QoS predictions, which is the goal of our partial research in this thesis.

QoS-aware Fault Tolerance of Web Services

With the ever-increasing complexity and scale of online services, it is almost impossible to build fault-free online service systems. Fault tolerance is one of the major techniques used to build reliable software systems [81]. With the prevalence of Web services, it is common that a number of Web services with identical or similar

functionalities are provisioned by different service providers. Many studies (*e.g.*, [159, 168, 160]) have suggested the use of these alternative Web services for fault tolerance.

In the literature, a number of fault tolerance strategies have been proposed. Typical examples include retry, recovery block, N-version programming, and so on [156]. Traditionally, software fault tolerance strategy is determined at design time. Each fault tolerance strategy may have a different effect on the overall service quality. However, because of the highly dynamic opertional environment of online service systems, there is no single fault tolerance strategy that can handle all situations [156]. In recent work, adaptive fault tolerance strategies for Web services have been investigated [159], with the aim to make optimal strategy determination based on QoS information of alternative services and QoS requirements from users.

2.2.2 Dynamic Service Deployment

Recently, much attention has been paid on the service deployment problem in the cloud. Some studies [128, 55, 54, 130] consider the service deployment problem in a single cloud, where services are dynamically replicated and mapped into different servers with various resources to optimize a set of proposed metrics, such as performance, resource utilization and operational cost.

The service deployment model shares some similar properties with the popular facility location problem (*a.k.a.* the k-center problem), which concerns optimal placement of facilities to minimize transportation costs. An extensive review of this problem can be found in [86]. Some classical algorithms (*e.g.*, k-median model) are also introduced to solve the replica placement problem [104]. Although traditional replica placement algorithms (*e.g.*, [104, 125]) have been extensively investigated in recent literature, these approaches primarily focus on the scenario of content replicas place-

ment in content delivery networks with static topology and thus fail to take the dynamics into consideration in current cloud systems.

With the growing of data center infrastructures, more research work has been conducted to address the challenge of deploying services across geo-distributed clouds. However, in most of the work (*e.g.*, [148, 70, 121, 24, 41]), individual services are independently deployed to optimize the corresponding performance and operational cost. Service dependencies, which is necessary for online service systems. are not taken into consideration. Zhang et al. [148] propose a dynamic service placement strategy to adapt to the fluctuating service demand of different graphical regions so as to minimize the operational cost while assuring the key performance requirements. The work [142] identifies the composite service placement problem with service dependencies in the cloud and solve the model by evolutionary algorithms. However, it focuses on the resource optimization from the perspective of cloud providers, and does not characterize the service deployment across geo-distributed data centers. Moreover, each service component is only deployed as one service instance, which is not the case in reality. In [69], the service co-deployment problem is addressed by modeling it as an integer programming formulation. However, the integer programming model fails to scale to large-scale service-oriented applications deployment problem across multiple data centers, since its complexity grows exponentially with the number of services and candidate data centers.

2.2.3 Runtime Service Adaptation

Self-adaptation is a key solution for cloud applications to cope with the changing operational environments [39]. In contrast to the well-studied traditional adaptive software systems [110], the dynamic cloud environment imposes a number of new challenges to the adaptation of cloud applications. In service-based cloud applications, the

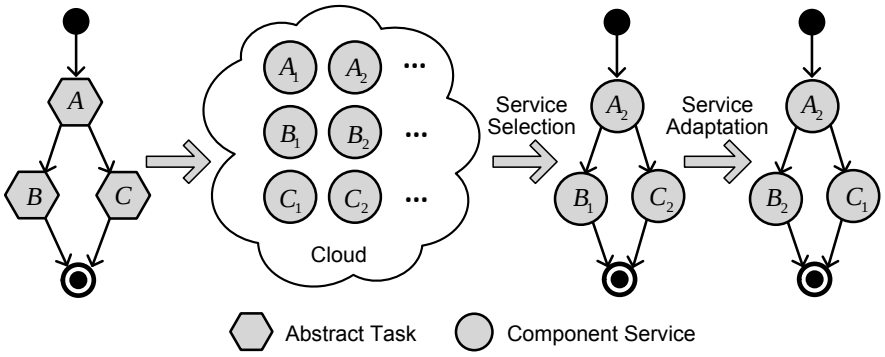


Figure 2.4: An Illustrative Example of Service Adaptation

application logic is typically expressed as a workflow with a set of abstract tasks, as shown in the leftmost panel in Figure 2.4. These abstract tasks (*e.g.*, A, B, C) are then implemented by invocations to the underlying component services (*e.g.*, A_2, B_1, C_2) provided in the cloud. It is expected that the proliferation of cloud computing will bring substantial deployment of services into the cloud, so that for each abstract task there are a set of functionally-equivalent candidate services. Conventional service composition approaches (*e.g.*, [143]) focus on how to make optimal service selection from those candidate services at design time. However, due to the dynamic nature of cloud environment, original services may become unavailable, new services may emerge, and the QoS values of services may change from time to time, thus leading to violations of SLA (service-level agreement). In such a setting, QoS-driven service adaptation is desired. Figure 2.4 presents such an illustrative example, where services B_1, C_2 are replaced with services B_2, C_1 respectively in an adaptation action, in the cases that the invocation to B_1 fails and the QoS of C_2 degrades.

To achieve this goal, a large body of research work has been conducted in recent literature. For example, the work [90] and [31] extend BPEL (Business Process Execution Language) engines with an interception and adaptation layer to enable monitoring and recovery of services. The work [36] employs autonomic configuration

of performance parameter settings to achieve self-adaptation for online Web applications. Some other work, such as [92] and [39], provides feasible adaptation mechanisms (*e.g.,* replacing the component services, or re-structuring the workflows) to support QoS-driven service adaptation. While most of these studies focus on adaptation mechanism design, our work targets at another key challenge, namely online QoS prediction [87], which is also fundamental for service adaptation.

Accurate QoS prediction is fundamental for QoS-driven service adaptation. The predicted QoS values directly impact the service adaptation decisions. For example, inaccurate predictions may cause improper adaptations and thus lead to SLA violations. For this purpose, online monitoring and prediction approaches, as presented in [129, 27], have been proposed to detect service failures and QoS deviations of the working services, but QoS prediction on candidate services is still not well explored. The approach introduced in [67] proposes to collect QoS values by sampling and invoking the candidate services, which is heavily limited by the incurred overhead. Thus, our work is motivated to address the QoS prediction problem for candidate services.

2.2.4 Service Logs Mangement

Logs are immensely useful in failure diagnosis. When software systems fail in the field, in many cases, developers can only rely on the logged messages to pinpoint the root causes, because it is notoriously challenging to reproduce production failures in the lab. Sometimes, it is also infeasible to get access to the failure-triggering input data due to privacy concerns [140]. Online service systems are typically deployed in large-scale data centers, logs are used as a principal tool for troubleshooting the failures because in production settings. Thus, it has become an industry common practice for service providers (*e.g.*, Microsoft, Google, IBM, etc.) to actively

collect their logs for postmortem analysis [140].

Log Management

With the systems scaling up, the volume of logs is rapidly growing, for example, at a rate of about 50 gigabytes (around 120∼200 million lines) per hour [88]. Therefore, log management on such huge volume of log data becomes a challenging problem. The difficulty may be further exacerbated by the situation that logs are distributed in different machines or even different data centers. There is a high demand for powerful log management infrastructures to assist log analysts in searching, filtering, analyzing, and visualizing a mountain of logs. Towards this end, some promising solutions, such as commercial Splunk[3], and open-source Logstash[4], Kibana[5], have been provided. All these solutions focus on the management of exiting logs.

Log Analysis

Logs contain a wealth of information that are useful in aiding software system maintenance, and, as such, have become an important data source for postmortem analysis [98]. For instance, logs have been widely analyzed for various tasks, such as anomaly detection [58, 135], problem diagnosis [91, 139], program verification [112], security monitoring [89], usage analysis [74], etc. In addition to the usage of logs, Shang et al. [114] studied how to automatically enrich the produced log messages with development knowledge (*e.g.*, source code, commits, issue reports) and further assist users in log understanding. Instead, our work in this thesis aims to improve the underlying logging practice, thus can potentially benefit these tasks on log analysis and log understanding.

[3] `http://www.splunk.com`
[4] `http://logstash.net`
[5] `http://kibana.org`

Logging Practices

Current research has mostly focused on the usage of logs, but little on logging itself. Two empirical studies [59, 140] have recently been conducted to characterize the logging practices. Yuan et al. [140] reported the characteristics of logging modifications by investigating the revision histories of open-source software systems. Our previous work [59] focused on studying where developer log through both code analysis and developer survey at Microsoft, and summarized five typical categories of logging strategies. Additionally, Shang et al. [113] studied the relationship between logging characteristics and the code quality of platform software. All these studies provide comprehensive logging characteristics that shed insights into our design of *LogAdvisor* in Chapter 6.

Towards improving the logging quality, Yuan et al. have recently pioneered two prior studies: LogEnhancer [141] and ErrLog [139]. LogEnhancer [141] aims to enhance the recorded contents in existing logging statements by automatically identifying and inserting critical variable values into them. ErrLog [139] summarizes a set of generic exception patterns (*e.g.*, exceptions, function-return errors) that potentially cause system failures, and then suggests conservative logging to automatically log all of them (*e.g.*, log all exceptions). Their work takes the first step towards automatic logging and provides promising results in reducing diagnosis time of system failures. In our work, we make an initial attempt to help developers make informed logging decisions.

2.2.5 Privacy Issue

Privacy is an important issue that has raised particular concerns among many research areas. In the following, we review the privacy studies related to our research.

Privacy in Service Composition

In service-oriented architecture, applications are typically built by composing Web services offered by different service providers. User information often needs to be shared across the providers to fulfil an overall application task. This can raise privacy issues between users and service providers when the selected Web services for composition have privacy policies that are not compliant with users' privacy requirements. In this regard, privacy-aware Web service selection and composition (*e.g.*, [136, 118, 48, 126]) have been studied. For example, Costante et al. [48] propose an approach to rank the candidate Web services with respect to the privacy level they offer. Tbahriti et al. [126] further provide a mechanism to verify and negotiate privacy constraints between users and service providers to enable privacy-compatible service composition. Different from these studies, our work in this thesis aims to address privacy issues for Web service recommendation.

Privacy in Recommender Systems

In recommender systems [106], users want to gain useful recommendations without compromising their privacy. To achieve so, a variety of privacy-preserving collaborative filtering approaches [33] have been proposed by using techniques such as randomization [100], cryptography [96], anonymization [66], and so on. Privacy is also of vital importance to the realization of QoS-based Web service recommendation, where users might not be willing to disclose their private usage data. However, there is currently a lack of studies on how to cope with the privacy issues for QoS-based Web service recommendation. Existing privacy-preserving collaborative filtering approaches are not directly applicable because of the unique challenges posed by Web service recommendation. For example, most of these approaches (*e.g.*, [37, 38, 144]) require multi-party or peer-to-peer collaboration between users, which is inapplicable to

service users. Some other approaches (*e.g.*, [85, 96]) are developed based on trusted server settings, thus cannot be applied to the Web service recommendation problem. In this thesis, we make the first attempt to build a privacy-preserving QoS prediction framework for Web service recommendation.

2.3 Data Analytics

In this section, we briefly introduce some techniques of data analytics that are closely related to our research in this thesis.

2.3.1 Collaborative Filtering

Collaborative filtering (CF) techniques are currently widely used in commercial recommender systems, such as movie recommendation in Netflix[6] and item recommendation in Amazon[7]. The CF model has been extensively studied in recent years. In recommender systems, CF works through the rating prediction problem. Specifically, users likely rate the items that they know about, such as 1$\sim$5 stars for the moives they have watched or books they have read. As illustrated in Figure 2.5, the values in grey entries are observed rating data, and the blank entries are unknown values. For example, the rating value between user u_1 and item i_1 is 5, while the rating value between user u_1 and item i_5 is missing, because u_1 has not rated i_5. In practice, each user usually rate only a small set out of all of the items, due to the large number of items. As a result, the user-item rating matrix is very sparse.

The basic idea of CF is to exploit and model the observed data to predict the unknown values, based on the insight that similar users may have similar preferences on the same item, and thus have similar ratings. To achieve this goal, two types of CF techniques have

[6] http://www.netflix.com
[7] http://www.amazon.com

	i_1	i_2	i_3	i_4	i_5
u_1	5	?	4	3	?
u_2	?	2	?	3	2
u_3	5	1	?	?	1
u_4	4	?	2	?	4

Figure 2.5: An Example of Rating Matrix

been studied in recent literature: neighbourhood-based approaches and model-based approaches [123].

Neighbourhood-based approaches: Neighbourhood-based approaches include user-based approaches (*e.g.*, UPCC) that leverage the similarity between users, item-based approaches (*e.g.*, IPCC) that employ the similarity between items, and their fusions (*e.g.*, UIPCC [83]). However, neighbourhood-based approaches are incapable of handling the data sparsity problem and have high time complexity.

Model-based approaches: Model-based approaches provide a predefined compact model to fit the training data, which can be further used to predict the unknown values. Matrix factorization [109] is one of the most popular model-based approaches used for collaborative filtering. In addition, matrix factorization model can usually achieve better performance than neighbourhood-based approaches.

In recent literature, CF has been introduced as a promising technique for various system engineering tasks, such as service recommendation [162, 79], system reliability prediction [157], and QoS-aware datacenter scheduling [52].

2.3.2 Matrix Factorization

Matrix factorization (MF) is a popular model to address the above collaborative filtering problem, which constrains the rank of the

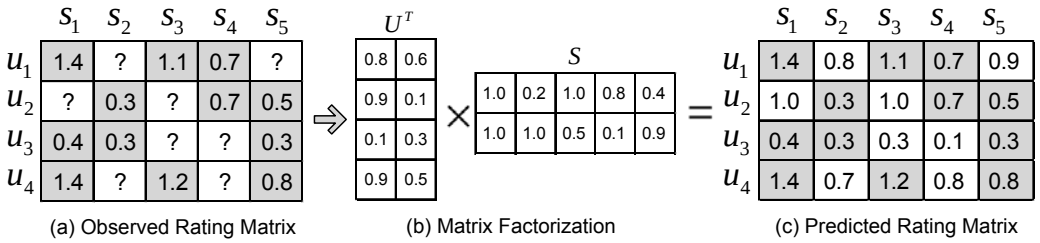


Figure 2.6: An Illustrative Example of Matrix Factorization

rating matrix, *i.e.*, $rank(R) = d$. The low-rank assumption is based on the fact that the entries of R are largely correlated, thereby resulting in a low effective rank in R. For instance, close users may have similar network conditions, and thus experience similar QoS on the same service. Figure 2.6 illustrates an example that makes use of matrix factorization for rating prediction. Concretely, factoring a matrix is to map both users and items into a joint latent factor space of a low dimensionality d (*e.g.*, $d = 2$ in Figure 2.6(b)), such that values of the rating matrix can be captured as inner products of latent factors in that space. Then the latent factors can be employed for further prediction on unknown rating values. For example, as shown in Figure 2.6(c), the predicted rating value between user u_1 and item s_2 is 0.8.

Formally, latent user factors are denoted as $U \in \mathbb{R}^{d \times n}$ and latent item factors as $S \in \mathbb{R}^{d \times m}$, which are used to fit the rating matrix R, *i.e.*, $R \approx U^T S$. To avoid overfitting, regularization terms that penalize the norms of the solutions (*i.e.*, U and S) are added. Thus we aim to minimize the following loss function:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (R_{ij} - U_i^T S_j)^2 + \frac{\lambda_U}{2} \|U\|_F^2 + \frac{\lambda_S}{2} \|S\|_F^2 , \quad (2.1)$$

where I_{ij} acts as an indicator that equals to 1 if R_{ij} is observed, and 0 otherwise (*e.g.*, $I_{11} = 1$ and $I_{12} = 0$ in Figure 2.6(a)). λ_U, λ_S are two parameters to control the extent of regularization, and $\|\cdot\|_F$ denotes the *Frobenius norm*. Frobenius norm $\|\cdot\|_F$ is a matrix norm.

Given a matrix $A \in \mathbb{R}^{n \times m}$, its Frobenius norm is defined as follows:

$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij}^2} \tag{2.2}$$

where a_{ij} is the element of A. When A reduces to a vector, the Frobenius norm is equivalent to the Euclidean norm.

It is worth noting that Salakhutdinov et al. has provided proper probabilistic interpretation [109] to the matrix factorization model in Equation 2.1, which is also known as probabilistic matrix factorization (PMF).

Gradient Descent Algorithm

Gradient descent is a widely used algorithm to find a local minimum of an objective function in an iterative way. For matrix factorization, gradient descent works by updating U_i and S_j simultaneously from random initialization using the following updating rules:

$$U_i \leftarrow U_i - \eta \frac{\partial \mathcal{L}}{\partial U_i}, \quad S_j \leftarrow S_j - \eta \frac{\partial \mathcal{L}}{\partial S_j}, \tag{2.3}$$

In particular, the derivatives of U_i and S_j can be derived from Equation 2.1 as follows:

$$\frac{\partial \mathcal{L}}{\partial U_i} = \sum_{j=1}^{m} I_{ij}(U_i^T S_j - R_{ij}) S_j + \lambda_U U_i, \tag{2.4}$$

$$\frac{\partial \mathcal{L}}{\partial S_j} = \sum_{i=1}^{n} I_{ij}(U_i^T S_j - R_{ij}) U_i + \lambda_S S_j. \tag{2.5}$$

Hence, the updating rules in Equation 2.3 can be rewritten as follows:

$$U_i \leftarrow U_i - \eta\big(\sum_{j=1}^{m} I_{ij}(U_i^T S_j - R_{ij}) S_j + \lambda_U U_i\big), \tag{2.6}$$

$$S_j \leftarrow S_j - \eta\big(\sum_{i=1}^{n} I_{ij}(U_i^T S_j - R_{ij}) U_i + \lambda_S S_j\big). \tag{2.7}$$

Algorithm 1: Gradient Descent for MF

Input: The collected rating matrix R, the indication matrix I, and the model parameters: η, λ_U and λ_S. /* $I_{ij} = 1$ if R_{ij} is known; otherwise, $I_{ij} = 0$ */

Output: The rating prediction results: $\hat{R}_{ij}$, where $I_{ij} = 0$.

Initialize $U \in \mathbb{R}^{d \times n}$ and $S \in \mathbb{R}^{d \times m}$ randomly;
repeat /* Batch-mode updating */
  foreach *(i, j)* **do** /* Compute $\frac{\partial \mathcal{L}}{\partial U_i}$ and $\frac{\partial \mathcal{L}}{\partial S_j}$ */
    $\frac{\partial \mathcal{L}}{\partial U_i} \leftarrow \sum_{j=1}^{m} I_{ij}(U_i^T S_j - R_{ij})S_j + \lambda_U U_i$;
    $\frac{\partial \mathcal{L}}{\partial S_j} \leftarrow \sum_{i=1}^{n} I_{ij}(U_i^T S_j - R_{ij})U_i + \lambda_S S_j$;
  foreach *(i, j)* **do** /* Update each U_i and S_j */
    $U_i \leftarrow U_i - \eta \frac{\partial \mathcal{L}}{\partial U_i}$;
    $S_j \leftarrow S_j - \eta \frac{\partial \mathcal{L}}{\partial S_j}$;
until *converge*;
foreach *(i, j)* $\in \{I_{ij} = 0\}$ **do** /* Make prediction */
  $\hat{R}_{ij} = U_i^T S_j$;

Gradient descent works on batch-mode, which needs all the data to be available. The latent factors U_i and S_j move iteratively by a small step of the average gradient, *i.e.*, $\frac{\partial \mathcal{L}}{\partial U_i}$ and $\frac{\partial \mathcal{L}}{\partial S_j}$, where the step size is controlled by η.

The detailed algorithm of gradient descent for MF is presented in Algorithm 1.

Stochastic Gradient Descent Algorithm

The scheme of stochastic gradient descent (SGD) is to update the stochastically using the sequentially coming data. At each step, the model can be adjusted by only considering the current data sample. Thus, SGD naturally provides an online algorithm, where we can adjust the model using each data sample from the data stream in an online fashion.

Formally, The loss function $\mathcal{L}$ in Equation 2.1 can be seen as the

Algorithm 2: Stochastic Gradient Descent for MF

Input: Sequentially observed data samples: (u_i, s_j, R_{ij}), and the model parameters: η, λ_u and λ_s.

Output: The rating prediction results: $\hat{R}_{ij}$, where $I_{ij} = 0$.

Initialize $U \in \mathbb{R}^{d \times n}$ and $S \in \mathbb{R}^{d \times m}$ randomly;
repeat `/* Online-mode updating */`
  foreach *(u_i, s_j, R_{ij})* **do**
    $\frac{\partial \ell}{\partial U_i} \leftarrow (U_i^T S_j - R_{ij}) S_j + \lambda_u U_i$;
    $\frac{\partial \ell}{\partial S_j} \leftarrow (U_i^T S_j - R_{ij}) U_i + \lambda_s S_j$;
    $U_i \leftarrow U_i - \eta \frac{\partial \ell}{\partial U_i}$;
    $S_j \leftarrow S_j - \eta \frac{\partial \ell}{\partial S_j}$;
until *converge*;
foreach *(i, j)* $\in \{I_{ij} = 0\}$ **do** `/* Make prediction */`
  $\hat{R}_{ij} = U_i^T S_j$;

sum of pairwise loss functions:

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} \ell(U_i, S_j), \tag{2.8}$$

and the pairwise loss function $\ell(U_i, S_j)$ with respect to $(U_i, S_j, R_i j)$ is defined as

$$\ell(U_i, S_j) = \frac{1}{2}(R_{ij} - U_i^T S_j)^2 + \frac{\lambda_u}{2} \|U_i\|_2^2 + \frac{\lambda_s}{2} \|S_j\|_2^2, \tag{2.9}$$

Note that the regularization parameters λ_u and λ_s are on different scale from those in Equation 2.1 . Similarly, we can derive the following updating equations for each iteration:

$$U_i \leftarrow U_i - \eta\big((U_i^T S_j - R_{ij}) S_j + \lambda_u U_i\big), \tag{2.10}$$
$$S_j \leftarrow S_j - \eta\big((U_i^T S_j - R_{ij}) U_i + \lambda_s S_j\big). \tag{2.11}$$

The detailed algorithm of stochastic gradient descent for MF is presented in Algorithm 2.

2.3.3 Network Coordinate System

Network coordinate system is originally proposed in [95] to estimate the network distances (*i.e.*, round-trip times) between pairwise Internet hosts in peer-to-peer (P2P) distributed networks. Due to its simplicity and effectiveness, network coordinate system has been widely studied in recent years. To date, the adoption of network coordinate systems has benefited a variety of applications, such as file sharing via Bit-Torrent [120], content distribution networks (CDN) [30], P2P multimedia streaming [75], shortest distance estimation in massive social networks [101, 154], etc. For more applications of network coordinate systems, we refer the reader to a recent survey [53].

Among various network coordinate systems, triangulated heuristic and global network positioning (GNP) are two widely employed approaches, due to their simplicity and generality. Triangulated Heuristic [95] employs a kind of relative coordinates based on the triangle inequality. A fixed set of landmarks are deployed in the network as references. Then each ordinary host is assigned an n-tuple relative coordinate, composed of the network distances between the ordinary host and the landmarks. Given the relative coordinate of each host, we can obtain the upper bound U and the lower bound L of the network distance between two hosts by triangle inequality. The network distance can be estimated by the convex combination of U and L (*e.g.*, $\frac{\mathrm{U}+L}{2}$). It is reported in [95] that taking the upper bound U as the network distance prediction result can achieve better performance. The triangulated heuristic approach is widely used in online shortest path distance prediction in large graphs [103].

GNP [95] is a typical landmark-based network coordinate system, which embeds the Internet hosts into an Euclidean space for network distance estimation. After obtaining the coordinate of each host, the network distance between two Internet hosts can be well

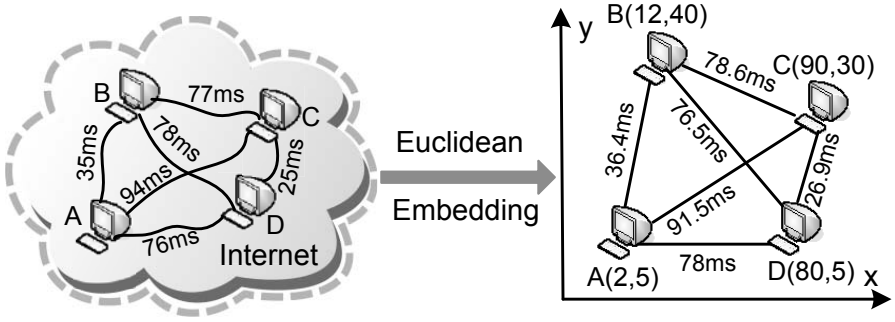


Figure 2.7: A Prototype of Network Coordinate System

approximated by the corresponding Euclidean distance. First of all, the landmarks first measure the network distances between each other and compute their own coordinates by minimizing the sum of the squared error between the computed Euclidean distances and the measured distances. Then each ordinary host has to measure the network distances to the landmarks and minimize the sum of the squared error to obtain its coordinate. After the coordinates computation, the Euclidean distance between two Internet hosts can be used to predict the corresponding unknown network distance. Figure 2.7 illustrates a prototype of the network coordinate system. As we can see from the figure, the four Internet hosts are embedded into a 2-dimensional Euclidean space by assigning each host a coordinate, and then the original network distances obtains good estimation results using the corresponding Euclidean distances.

□ **End of chapter.**

Chapter 3

Response Time Prediction of Web Services

For many online services, response time is one of the most important QoS attributes to measure the user experience on service quality. Response time information of Web services is crucial to conduct performance optimization tasks (*e.g.*, service selection and service deployment) of large-scale online service systems. In this chapter, we focus on the study of joint response time monitoring and prediction of Web services. Specifically, we first introduce the research problem and motivation of this work in Section 3.1. Then, we present the framework of Web service positioning in Section 3.2, and describe the detailed response time prediction algorithm in Section 3.3. Evaluation results and a case study on our collected real-world Web service dataset are provided in Section 3.4 and Section 3.5, respectively. Finally, Section 3.6 concludes this chapter.

3.1 Problem and Motivation

Large-scale online service systems are commonly built on service-oriented architecture (SOA) [147]. The adoption of SOA enables flexible system management with its features of loose coupling and dynamic binding, where invocations of underlying Web services are applied to fulfilling complex application logic. The use of such

Web services significantly eases the development and maintenance of online service systems. However, it may also bring non-negligible influence to the overall system quality, because of its dependence on the quality of underlying Web services. To build high-quality online service applications, it is a critical task for application designers to select appropriate services that fulfill both functional requirements and non-functional requirements. While functional requirements specify what a service does, non-functional requirements are concerned with QoS attributes such as availability, reliability, and performance measures like response time and throughput [163].

With the widespread proliferation of Web services, providers begin to offer more and more services even with equivalent or similar functionalities. For example, both providers, CDYNE.COM and WebserviceX.NET, offer equivalent Web services for querying global weather information [172]. QoS is an important factor in distinguishing those similar services, according to which services can be ranked and selected by application designers. As a result, there is a high demand for obtaining QoS information of available services. For many online services, response time, which stands for the time duration between user sending out a request and receiving a response, is one of the most important QoS attributes for service quality management. For example, according to the report in [122], a half-second delay will cause a 20% drop in Google's traffic, and a tenth of a second delay can cause a drop in one percent of Amazon's sales. In this chapter, we focus primarily on response time assessment of Web services.

Unlike quality assessment for traditional shrink-wrapped software, it is more challenging for user-perceived service quality assessment. Service invocations, especially for third-party Web services, usually rely on the Internet for connectivity. Due to the heavy influence of dynamic network conditions, users at different locations may have different QoS experiences even on the same Web service [72]. Therefore, QoS is user-specific, whereby traditional

software quality models become ineffective and inappropriate for QoS assessment of Web services. It is straightforward to measure the QoS values through direct invocations of Web services. However, it is infeasible in practice due to the prohibitive overhead incurred when each user actively invokes a large number of service candidates over the Internet. As a result, effective QoS prediction approaches are desired to provide accurate predictions for user-perceived QoS values of different Web services, without the need of additional service invocations.

In recent literature, a number of QoS prediction approaches have been proposed [44, 68, 116, 163]. These approaches apply collaborative filtering (CF) techqniues to achieve QoS predictions based on the historical QoS data collected from different users. With evaluations on real-world QoS datasets, these approaches have been shown to achieve good overall prediction accuracy under dense historical QoS data. However, the CF-based approaches suffer from a major problem that is the sparsity of the available historical QoS data. For a response time matrix, each row represents a user, each column represents a Web service, and each entry denotes the response time of a certain user invoking a certain Web service. As reported in [163], the performance of CF-based approaches is significantly degraded when the response time matrix is very sparse. This is, however, the case in practice, since a user usually invokes only a few number of the numerous Web service candidates each time. Therefore, the collected historical response time matrix is usually sparse. Moreover, influenced by unpredictable network conditions and dynamic service workload, it is common that user-perceive response time of a Web service is changing from time to time. In this case, we cannot make use of out-of-date historical QoS data for QoS prediction, which further exacerbates the data sparsity problem.

On the other hand, network coordinate systems (*e.g.*, GNP [95]) are widely used in P2P networks to estimate the network distance

between pairwise Internet hosts. The basic idea of network coordinate systems is to embed the Internet hosts into a high dimensional Euclidean space by assigning each host a coordinate in that space, such that the measured network distances (*i.e.*, latencies) between hosts can be well approximated by the corresponding Euclidean distances. In this scheme, after obtaining the coordinates of different hosts, we can use a simple Euclidean distance to accurately predict the unknown network distance between any two Internet hosts in constant time.

Inspired by the success of network coordinate based prediction approaches, we propose a Web service positioning (WSP) framework [174] to address the data sparsity problem of CF-based approaches. Our WSP framework combines the advantages of network coordinate based approaches and CF-based approaches. More specifically, we first re-design the traditional GNP algorithm (typically employed in peer-to-peer scenarios) to fit the response time prediction of Web services in our client-server scenario. Then, the available historical data of users (these data are employed in CF-based approaches for making prediction) are adopted to optimize the coordinate computation of users, which further enhances the prediction accuracy. Finally, comprehensive experiments are conducted based on our collected response time data from real-world Web services. The experimental results show that our WSP-based response time prediction approach achieves significant accuracy improvements over both the existing network coordinate based approaches and CF-based approaches.

3.2 Web Service Positioning Framework

To address the data sparsity problem of CF-based prediction approaches, we propose a Web service positioning (WSP) framework to make response time prediction for Web services, as illustrated in Figure 3.1. Built on network coordinate systems, the WSP

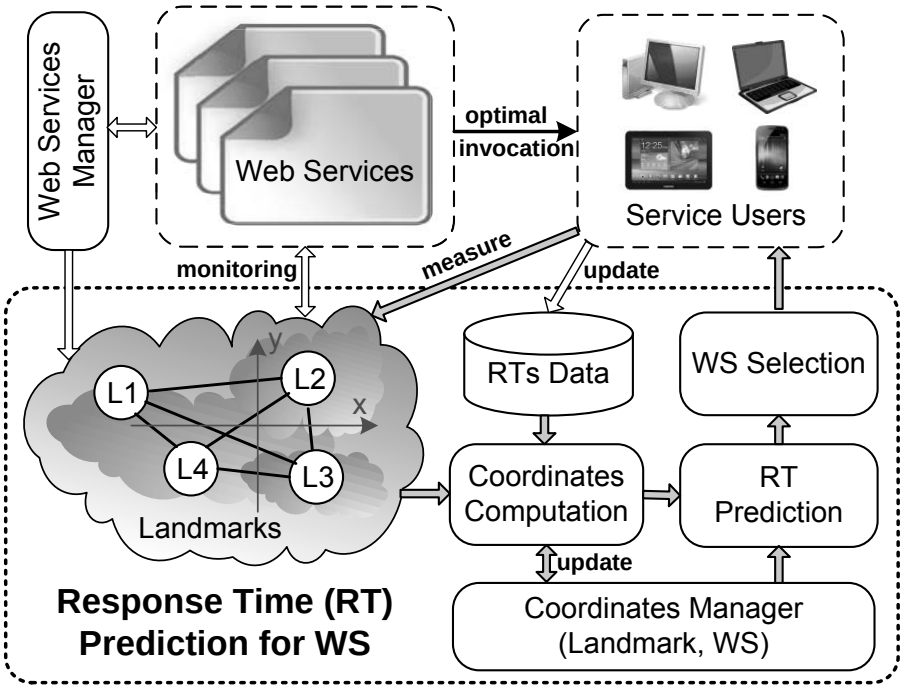


Figure 3.1: A Web Service Positioning Framework for Response Time Prediction

framework targets at collective monitoring and prediction, whereby the data sparsity issue is mitigated. In the framework, a small number of landmarks are deployed to periodically monitor the available Web services and further construct a network coordinate system. By integrating the available historical QoS data used in CF-based approaches with the network coordinate model, our proposed WSP approach combines the advantages of network coordinate based approaches and CF-based approaches. As a result, our WSP framework can not only serve for users without valid historical data (*e.g.*, mobile users) but also enhance the prediction accuracy for users with sparse historical data.

The WSP framework involves the following procedures of offline coordinates updating and online response time prediction.

1) Offline Coordinate Updating: *a)* The deployed landmarks measure the network distances between each other (*e.g.*, use ping to

measure the round-trip time), and then construct a network coordinate system by embedding the landmarks into an high-dimensional Euclidean space such that each landmark obtains a coordinate in that space. *b)* The landmarks monitor the available Web services with periodical invocations. The coordinate of each Web service is obtained by taking the landmarks as references and embedding each Web service as a coordinate in the same space.

2) Online Response Time Prediction: *a)* When a service user requests for a Web service invocation, it first measures the network distances to the landmarks (*e.g.*, by ping). Then the measurement results are used to compute the user's coordinate. To optimize the coordinate computation, we also combine the available historical response time data of this user. *b)* The user-perceived response times of all the available Web service candidates can be easily predicted as the corresponding Euclidean distances between the user's coordinate and each Web service's coordinate. *c)* Based on the response time prediction results of Web service candidates, optimal Web service can be selected for the user. *d)* The user invokes the optimal Web service for service invocation, and also obtains the real response time data of this Web service. *e)* The response time database is updated with the new observation to contribute to the coordinate updating for next Web service selection.

3.3 WSP-based Response Time Prediction

According to the above Web service positioning framework, in this section, we describe the response time prediction algorithm in detail. The algorithm involves four steps: landmark coordinate computation, Web service coordinate computation, service user coordinate computation, and response time prediction.

3.3.1 Landmark Coordinate Computation

In our WSP framework, a small number of landmarks are deployed to build a network coordinate system. Suppose n landmarks are set up, denoted by $L = \{l_i \mid i = 1, 2, \cdots, n\}$. The network distances between landmarks are measured using ping messages, and then transmitted to a central node for coordinate computation. The pairwise network distances can be expressed as an $n \times n$ distance matrix as follows:

$$D_L = \begin{bmatrix} 0 & d(l_1, l_2) & \cdots & d(l_1, l_n) \\ d(l_2, l_1) & 0 & \cdots & d(l_2, l_n) \\ \vdots & \vdots & \cdots & \vdots \\ d(l_n, l_1) & d(l_n, l_2) & \cdots & 0 \end{bmatrix}, \tag{3.1}$$

where the entry $d(l_i, l_j)$ denotes the network distance between landmarks l_i and l_j. The distance matrix D_L is assumed to be symmetric along the diagonal, which is set as 0 where $l_i = l_j$.

The goal of our first step is to build a network coordinate system by embedding these n landmarks into an m-dimensional Euclidean space R^m, such that each landmark obtains a coordinate. We denote it as $x_{l_i} = (x_{l_i}^1, x_{l_i}^2, \cdots, x_{l_i}^m)$, where $x_{l_i}^k \in R, 1 \leq k \leq m$. To optimize the embedding, we define the objective function to represent the sum of squared errors between predicted distances and real distances.

$$f_L(x_{l_1}, \cdots, x_{l_n}) = \sum_{l_i, l_j \in L, i > j} [\hat{d}(l_i, l_j) - d(l_i, l_j)]^2 , \tag{3.2}$$

where $\hat{d}(l_i, l_j)$ denotes the predicted network distance that is computed as the Euclidean distance between l_i and l_j as follows:

$$\hat{d}(l_i, l_j) = \left\| x_{l_i} - x_{l_j} \right\|_2 = \sqrt{\sum_{k=1}^{m} \left(x_{l_i}^k - x_{l_j}^k\right)^2} . \tag{3.3}$$

However, directly minimizing Equation 3.2 usually suffers from the overfitting problem. In other words, optimal solutions can lead to an accurate model with small errors on the landmarks embedding while having large errors on the unseen data, *i.e.*, the unknown response time values to be predicted. To address this problem, we make use of regularization to penalize the norms of the solutions, expressed as follows:

$$\begin{aligned} f'_L(x_{l_1}, \cdots, x_{l_n}, \lambda_l) &= \sum_{l_i, l_j \in L, i>j} [\hat{d}(l_i, l_j) - d(l_i, l_j)]^2 \\ &+ \lambda_l \sum_{i=1}^{n} \|x_{l_i}\|_2^2 . \end{aligned} \tag{3.4}$$

Note that there are many solutions for minimizing Equation 3.2, because any rotation or translation of the landmark coordinates will not influence the inter-landmark distances [95]. In addition to overcoming the overfitting issue, the regularization term can also help avoid the coordinate drift of the solution by choosing the coordinates with the smallest norm. We will further study the impact of the regularization term in Section 3.4.7.

With this formulation, the optimal coordinates of landmarks can be obtained by minimizing Equation 3.4, as a generic multi-dimensional global minimization problem. As with GNP [95], in this work, we apply *simplex downhill algorithm* [94] to solve this minimization problem.

Finally, the coordinates of the n landmarks are obtained and stored in a management node to provide references to the coordinate computation of Web services and users. To track the changes of network conditions, the coordinates of landmarks should keep updating periodically. For ease of periodical re-computation, we can simply input the old coordinates as the initialization solution each time, which can greatly accelerate the convergence of the minimization problem.

Note that for an m-dimensional Euclidean space, at least $m + 1$

landmarks should be used for coordinate computation, as it is impossible to construct an unique Euclidean space with less landmarks. But it still remains an open question about how to deploy the landmarks. In our study, we select the landmarks from a set of candidate nodes using the spectral clustering based approach as described in [84]. The impact of number of landmarks will be studied in Section 3.4.5.

3.3.2 Web Service Coordinate Computation

In the WSP framework, a small number of landmarks monitor the available Web services by periodically invoking them. Suppose there are w available Web services, denoted by $S = \{s_i \mid i = 1, 2, \cdots, w\}$. Then an $n \times w$ matrix composed of network distances between n landmarks and w Web services can be obtained, which is expressed as follows:

$$D_{LS} = \begin{bmatrix} d(l_1, s_1) & d(l_1, s_2) & \cdots & d(l_1, s_w) \\ d(l_2, s_1) & d(l_2, s_2) & \cdots & d(l_2, s_w) \\ \vdots & \vdots & \cdots & \vdots \\ d(l_n, s_1) & d(l_n, s_2) & \cdots & d(l_n, s_w) \end{bmatrix}, \tag{3.5}$$

where the entry $d(l_i, s_j)$ denotes the network distance between landmark l_i and Web service s_j .

The network distances to Web services are measured by landmarks and then transmitted to a management node to compute the coordinate for each Web service. Therefore, each Web service is embedded into the same Euclidean space with the landmarks by by taking the coordinates of the landmarks as references. Given a Web service s_j ($1 \leq j \leq w$), the m-dimensional coordinate x_{s_j} can be obtained by minimizing the following objective function:

$$f_S(x_{s_j}, \lambda_s) = \sum_{l_i \in L} [\hat{d}(l_i, s_j) - d(l_i, s_j)]^2 + \lambda_s \left\| x_{s_j} \right\|_2^2 , \tag{3.6}$$

where $\hat{d}(l_i, s_j)$ denotes the predicted network distance between landmark l_i and Web service s_j. Likewise, $\lambda \left\| x_{s_j} \right\|_2^2$ is the regularization term.

The minimization of Equation 3.6 can also be cast as a generic multi-dimensional global minimization problem so that we can solve it with *simplex downhill algorithm*. Meanwhile the coordinates of the available Web services should also update periodically.

3.3.3 Service User Coordinate Computation

Any service user can request our WSP system for optimal Web service selection. At the beginning, the user measures the network distances to the landmarks using ping messages, and then transmit the results to the management node for coordinate computation. The measured network distances can be denoted as a vector in the following:

$$D_{uL} = [d(u, l_1), d(u, l_2), \cdots, d(u, l_n)] \,, \tag{3.7}$$

where $d(u, l_i)$ denotes the network distance between the user u and the landmark l_i.

To enhance the prediction accuracy, we also incorporate the advantage of CF-based approaches by making effective use of the available historical usage data. Suppose the available response time data between the user u and the Web services is denoted as $\{d(u, s_i) \mid s_i \in S_A\}$. S_A is the Web service set that user u has used before, where historical usage data are collected. With available historical data, we propose to minimize the following objective function to obtain the coordinate of the user, *i.e.* x_u.

$$\begin{aligned} f_u(x_u, \lambda_u) &= \sum_{l_i \in L} [\hat{d}(u, l_i) - d(u, l_i)]^2 \\ &+ \sum_{s_i \in S_A} [\hat{d}(u, s_i) - d(u, s_i)]^2 + \lambda_u \|x_u\|_2^2 \,, \end{aligned} \tag{3.8}$$

where $\hat{d}(u, l_i)$ denotes the predicted network distance between user u and landmark l_i, and $\hat{d}(u, s_i)$ denotes the predicted response time value between user u and the Web service s_i in S_A. The first part of the objective function $f_u(x_u, \lambda_u)$ employs the reference information of landmarks, while the second part takes advantage of the available historical data. Additionally, the regularization term is integrated in the third part. The same optimization algorithm, *simplex downhill*, is used to solve the problem.

3.3.4 Response Time Prediction

When a new user requests for Web service invocation, it first measures the network distances to the landmarks, and then send the results to the central management node, whereby the user's coordinate can be obtained according to the above step. After obtaining the coordinate of the user, as well as the coordinates of all the monitored Web services, the response time prediction can be easily made as the Euclidean distance between the coordinates as follows:

$$\hat{d}(u, s_i) = \|x_u - x_{s_i}\|_2, \quad s_i \in S, s_i \notin S_A \ , \tag{3.9}$$

where $\hat{d}(u, s_i)$ denotes the prediction value between user u and Web service s_i. The conditions $s_i \in S$, $s_i \notin S_A$ denote the set of Web services with unknown response time data.

With the prediction results for all available Web services, the Web services can be ranked and selected for the user according to user-perceived response time performance. However, we target primarily on response time prediction in this section, and refer to Section 3.5 for a detailed case study on latency-aware service deployment to show the practical use of response time information.

3.4 Evaluation

This section presents the collection of our Web service dataset and the experimental results obtained based on this dataset for evaluation purpose.

3.4.1 Data Collection and Description

To evaluate the response time prediction approach, real-world Web service data is needed. Although several QoS datasets for Web services have been collected in the previous work [163], they are not applicable to this work because of the lack of network distances among landmarks to construct a network coordinate system. As a result, we collect a new response time dataset for our evaluation by use of the PlanetLab platform. PlanetLab[1] is an open platform for system and networking research, currently consisting of 1,353 nodes at 717 global sites. The collected dataset involves response time records between 200 users (simulated by PlanetLab nodes) and 1,597 Web services. In particular, the pairwise network distances between the 200 distributed PlanetLab nodes are also collected, which can be set as landmarks in our WSP approach.

To collect the real-world data of Web services, we first get a list of 588 active PlanetLab nodes via CoMon[2] service, since it is common that some nodes shut down or lose connection of the Internet. Meanwhile, about 5,800 Web services are obtained by crawling Web service information from the Internet. To obtain response time data, we use ping messages to measure the round-trip time (RTT) from each PlanetLab node to each Web service, assuming that the service-running time is equivalent for each Web service of the same functionality. We send 32-byte ping packets continually for ten times and take the average RTT from all replies as the response time. Similarly, the network distances among the PlanetLab nodes

[1] http://www.planet-lab.org
[2] http://comon.cs.princeton.edu

Table 3.1: Descriptions of Web Service Dataset

Statistics	**Values**
Number of records	359,400
Number of service users	200
Number of Web services	1,597
Minimal response time	0.008 ms
Maximal response time	2,976.714 ms
Mean of response time	71.984 ms
Standard deviation of response time	64.746 ms

are obtained as well. The raw data is then post-processed to retain the nodes and Web services that are all reachable. Finally, we are left with 200 PlanetLab nodes and 1,597 Web services. The relatively low yield is partially due to the case that some Internet hosts are ping unavailable, and partially due to the failure of the Internet connection. Consequently, a 200-by-1597 matrix of response times and a 200-by-200 matrix of network distances are obtained.

The data statistics of our QoS dataset are summarized in Table 3.1. The minimal and maximal values of the response time data are 0.008 ms and 2.98 s, respectively. The mean and standard deviation values are 71.984 ms and 64.746 ms, respectively, which implies that the observed response time values for different service users have a great variation. For ease of reproducing our experimental results, and to facilitate future research, we have publicly released our dataset on our project page[3].

3.4.2 Evaluation Metrics

In our experiments, we employ two metrics, *Mean Absolute Error (MAE)* and *Median Relative Error (MRE)*, to evaluate the prediction accuracy of our proposed WSP approach in comparison with other existing approaches. The two metrics are defined as follows:

- *MAE*: This metric is widely employed to measure the average

[3] `http://wsdream.github.io/WSP`

prediction accuracy [153], where smaller MAE means better prediction accuracy.

$$MAE = \frac{\sum_{i,j} \left| \hat{d}(u_i, s_j) - d(u_i, s_j) \right|}{N}, \tag{3.10}$$

where $\hat{d}(u_i, s_j)$ and $d(u_i, s_j)$ denote the predicted value and the measured value, respectively, between service user u_i and Web service s_j. N is the number of predicted records.

- *MRE*: This metric is median value of all the pairwise relative error values.

$$MRE = \underset{i,j}{Median} \frac{\left| \hat{d}(u_i, s_j) - d(u_i, s_j) \right|}{d(u_i, s_j)}, \tag{3.11}$$

which means 50% of the relative errors are below *MRE*. While MAE focuses on absolute error measurement, MRE evaluates the relative error. The selection of the evaluation metrics is dependent on the specific application scenario.

3.4.3 Accuracy Comparison

In this section, in order to evaluate the prediction accuracy of our proposed WSP approach, we compare our approach with other existing approaches in the following:

- **UPCC**: This is a user-based collaborative filtering approach, which was first introduced to Web service QoS prediction in [116]. UIPCC exploits the similarity between users to predict the response time values.
- **IPCC**: This is a item-based collaborative filtering approach, which exploits the similarity between Web service items for Web service QoS prediction in [163].

- **UIPCC**: This is a hybrid method, proposed in [163], by combing both user-based and item-based collaborative filtering approaches. UIPCC can make full use of the similarity between users and the similarity between Web service items.
- **Triangulation**: This is a heuristic approach based on the triangle inequality in the metric space, which is has been described in Chapter 2. Specifically, we take the the upper bound as the response time prediction result as indicated in [95].
- **GNP**: GNP is proposed in [95] as a landmark-based network coordinate system to estimate network distances between Internet hosts for P2P networks.

In this experiment, we choose 16 nodes as landmarks from our dataset as did in GNP [95] (the impact of the number of landmarks will be discussed in Section 3.4.5), while the remaining 184 computer nodes are taken as service users. Therefore, the measured response time data between 184 service users and 1,597 Web services can be denoted as a 184×1597 matrix. As we mentioned in the previous section, the available historical data is very sparse. In order to simulate the sparse situation in real world, we randomly remove entries from the data matrix such that each user only keeps a few available historical values. In this way, we vary the matrix density as 0, 5%, 10%, 15%. Particularly, matrix density = 0 means no historical data of users are employed, such as for mobile users, whose historical data may vary significantly due to their high mobility and are not applicable for response time prediction. Matrix density = 5%, for example, indicates that each user has 5% (*i.e.* about 80) response time data out of all the Web services. The removed entries are used as the expected values to verify the prediction quality. In the sequel, for simplicity, we set $\lambda_l = \lambda_s = \lambda_u = \lambda$, and denote n as the number of landmarks, m as the coordinate dimensionality. In this experiment, the parameter settings are $\lambda = 0.1, m = 10$. Each approach is performed 100 times

Table 3.2: Prediction Accuarcy w.r.t. MAE

Methods	Matrix Density			
	0	5%	10%	15%
UPCC	N/A	33.4439	25.4751	21.2634
IPCC	N/A	51.9462	31.3841	26.2708
UIPCC	N/A	33.7476	25.3795	21.0852
Triangulation	33.9315	33.9315	33.9315	33.9315
GNP	29.2793	29.2793	29.2793	29.2793
WSP	**28.3502**	**20.6982**	**20.3531**	**19.9709**
Improvements(%)	3.17%	29.31%	19.80%	5.28%

Table 3.3: Prediction Accuarcy w.r.t. MRE

Methods	Matrix Density			
	0	5%	10%	15%
UPCC	N/A	0.1842	0.1329	0.1051
IPCC	N/A	0.3064	0.1674	0.1335
UIPCC	N/A	0.1856	0.1317	0.1027
Triangulation	0.1733	0.1733	0.1733	0.1733
GNP	0.1375	0.1375	0.1375	0.1375
WSP	**0.1316**	**0.0972**	**0.0927**	**0.0922**
Improvements(%)	4.29%	29.31%	29.61%	10.22%

and the average values are reported. In contrast, we set Top-K = 10, $\lambda = 0.1$ for CF-based approaches [163]. The prediction accuracy of different approaches under two evaluation metrics are shown in Table 3.2 and Table 3.3.

As we can see from the experimental results, our WSP approach obtains smaller MAE and MRE results consistently under different data densities, which indicates that our approach outperforms the others. The last row of each table shows the percentages of the accuracy improvements of our WSP approach (marked in bold), compared with the best of other existing methods (marked with underline). While CF-based approaches are heavily influenced by the matrix density, our WSP approach is less sensitive to the matrix density and obtains good prediction accuracy even under sparse historical data. For instance, the WSP has about 20% improvement

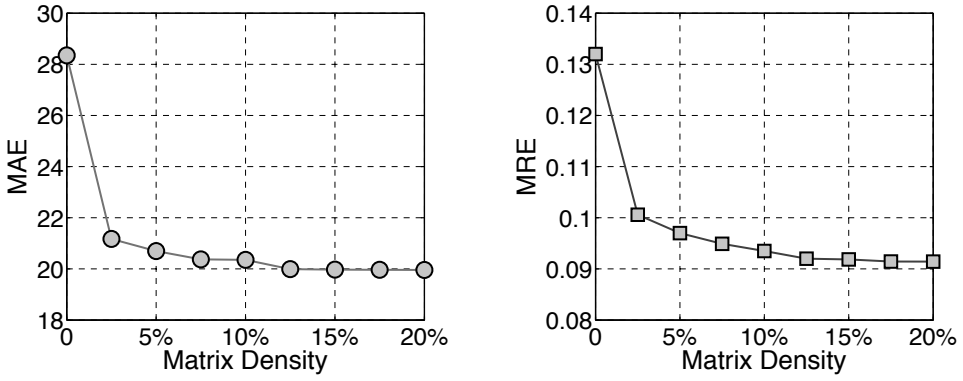
compared with the UIPCC method, with 10% historical data. Especially, for matrix density = 0, the CF-based approaches (UPCC, IPCC and UIPCC) run into a malfunction (denoted as N/A) while landmark-based approaches (Triangulation, GNP, and WSP) achieve good overall prediction accuracy. It implies that our WSP approach can also serve well for newly joining users or mobile users without available historical data.

On the other hand, our WSP approach outperforms the traditional network coordinate based approaches, triangulated heuristic and GNP, even at matrix density = 0, indicating that our WSP approach makes improvement based on GNP. We can also see that the IPCC approach performs worse than the UPCC approach in our experiments. This is because the IPCC method cannot find enough similar neighbours as the number of users is much smaller than the number of Web services. In other words, it is the data sparsity that significantly degrades the performance of the IPCC method. The results shown that with the increase of matrix density, the triangulated heuristic approach and GNP approach have no performance improvement since they make no use of the historical data.

To sum up, our proposed WSP approach combines the advantages of network coordinate based approaches and CF-based approaches, and achieves better accuracy compared to the existing prediction methods.

3.4.4 Impact of the Matrix Density

As shown in Table 3.2 and Table 3.3, both the CF-based approaches and landmark-based approaches are influenced by the matrix density, that is, the sparsity of the historical data. To further study the impact of the matrix density on the prediction accuracy, we vary the matrix density from 0 to 20% at the step of 2.5%, where matrix density = 0 means making response time predictions without using



(a) MAE (b) MRE

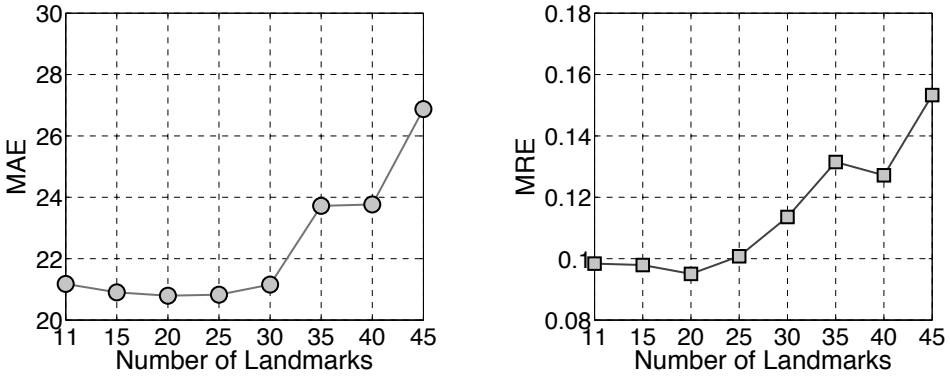
Figure 3.2: Impact of the Matrix Density

historical data, *i.e.* $S_A = \varnothing$ in Equation 3.8. In this experiment, without loss of generality, we set $n = 16$, $m = 10$, and $\lambda = 0.1$.

The experimental results are shown in Figure 3.2, composed of MAE and MRE values under different data densities. We can observe that dense historical data can benefit the prediction performance. However, after significantly decreasing when the matrix density varies from 0 to 2.5%, MAE and MRE both keep steady with a little reduce when the matrix density becomes denser. In other words, the prediction performance of WSP approach is less sensitive to the sparsity of data matrix, as a result, addresses the limitations of practical data sparsity problem of CF-based approaches.

3.4.5 Impact of the Number of Landmarks

The landmark deployment (*e.g.*, the position and number) is very essential to the performance of our WSP approach. In this experiment, we select the landmarks from the candidate nodes using the spectral clustering based approach in [84]. To characterize the impact of the number of landmarks, we conduct the experiment by varying the number of landmarks from 11 to 45. We also set $m = 10$,



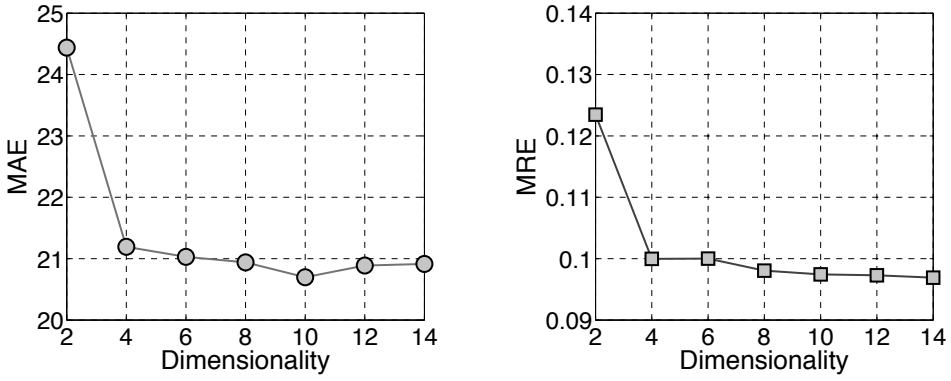
(a) MAE (b) MRE

Figure 3.3: Impact of the Number of Landmarks

$\lambda = 0.1$ and matrix density $= 5\%$. Note that there should be more than 11 landmarks for 10-dimensional Euclidean space construction. The results of MAE and MRE are illustrated in Figure 3.3. We can observe that the MAE and MRE values both decrease slightly when the number is less than 20, and then rise when the number is larger than 25. We can find that the large number of landmarks may not make for the prediction performance improvement, since there exist larger errors when embedding the Web services and users into Euclidean space with too many reference nodes, as a result of the triangle inequality violations (TIV) of network latencies [84]. In practice, we can deploy enough landmarks while each Web service and user only choose n landmarks as references, which can also avoid the single point of failure of landmarks.

3.4.6 Impact of the Coordinate Dimensionality

Dimensionality is a key factor when embedding the Internet hosts into an Euclidean space. We may wonder how many dimensions should be used to construct the coordinate system in WSP. Intuitively, higher dimensionality contributes to more accurate coordinate



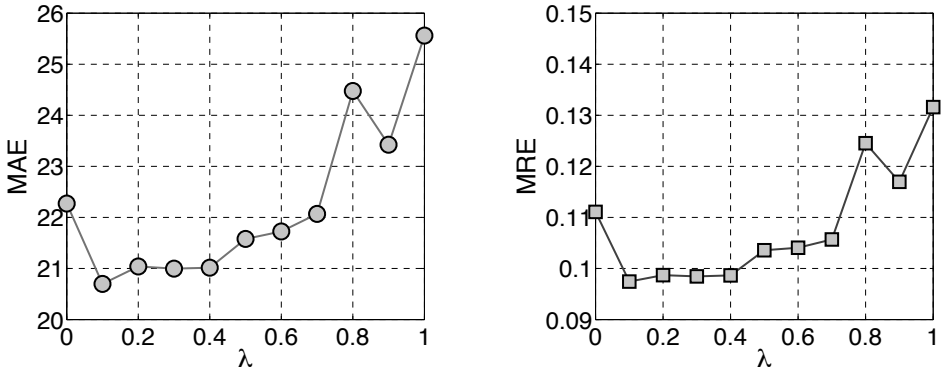
(a) MAE (b) MRE

Figure 3.4: Impact of the Coordinate Dimensionality

computation. To characterize the impact of the dimensionality, we conduct experiments with our dataset and vary the dimensionality from 2 to 14 at the step of 2. We also set $n = 16, \lambda = 0.1$ and matrix density = 5%. The experimental results are illustrated in Figure 3.4. We can observe that both MAE and MRE values decrease with the increase of the dimensionality of coordinates, but the accuracy improvement diminishes when the dimensionality is larger than 8. As a result, higher dimensionality beyond a certain point only makes little performance improvement.

3.4.7 Impact of the Regularization Term

For accurate response time prediction, we are supposed to minimize the prediction errors between users and Web services. To address the overfitting problem when computing coordinates, we introduce a regularization term to penalize the norms of the solutions which is widely adopted in machine learning area. In addition, the regularization term can also avoid the coordinate drift due to the non-uniqueness of the solution by choosing the coordinates with the smallest norm.



(a) MAE (b) MRE

Figure 3.5: Impact of the Regularization Term

In this experiment, we vary the λ from 0 to 1, while $\lambda = 0$ means no regularization term is used. For other parameters, we set $n = 16, m = 10$, and matrix density $= 5\%$. The experimental results are shown in Figure 3.5. As is shown in the figure, when $\lambda = 0.1$, smaller MAE and MRE values are obtained compared with $\lambda = 0$, indicating that the regularization term can contribute to the prediction accuracy improvement. However, MAE and MRE rise with the increase of λ. Therefore, the prediction accuracy is sensitive to λ, and we set $\lambda = 0.1$ in our experiments, which has been shown to achieve good prediction accuracy.

3.5 Case Study

In this section, we provide a case study on latency-aware service deployment in geographically-distributed clouds [176] to illustrate the practical use of response time information obtained by our WSP framework.

3.5.1 Latency-Aware Service Deployment

With the prevalence and benefit of cloud computing, many cloud providers like Amazon, Google and Microsoft have built large data centers in geographically distributed locations to achieve reliability and serve millions of users world-wide. For example, Amazon EC2[4] nowadays provisions cloud services over nine geographically dispersed regions, where service providers have options to deploy their applications in data centers from Virginia, Oregon, California, Ireland, Singapore, Tokyo, Sydney, São Paulo and GovCloud. Moreover, as the ideas of InterCloud and cloud federation [60, 80] become mature, more and more geo-distributed data centers will be cooperatively utilized for the tasks of performance optimization, operational cost minimization, traffic load balancing, demand spikes accommodation, and catastrophic recovery.

Among these tasks, a key challenge faced by service providers is how to scale their applications across these geographically distributed data centers. That is how to optimize the deployment strategies to take full advantage of the geo-diversity to achieve better performance (*e.g.*, response time) and minimize the operational cost when serving globally dispersed users. Although service management framework in a single data center has been well studied, such as auto scaling and elastic load balancing, there is an absence of efficient service deployment and management framework for geo-distributed data centers. In addition to the conventional resource constraints (*e.g.*, resource capacity, CPU and memory requirements of virtual machines) and on-demand resource assignment of service deployment in a single data center, the dynamic pricing [148] in different data centers and non-negligible time-varying communication latencies between data centers also need to be considered when deploying applications across multiple data centers. Moreover, the dynamic service demand and geographical distribution of end users [102]

[4] `http://aws.amazon.com/ec2`

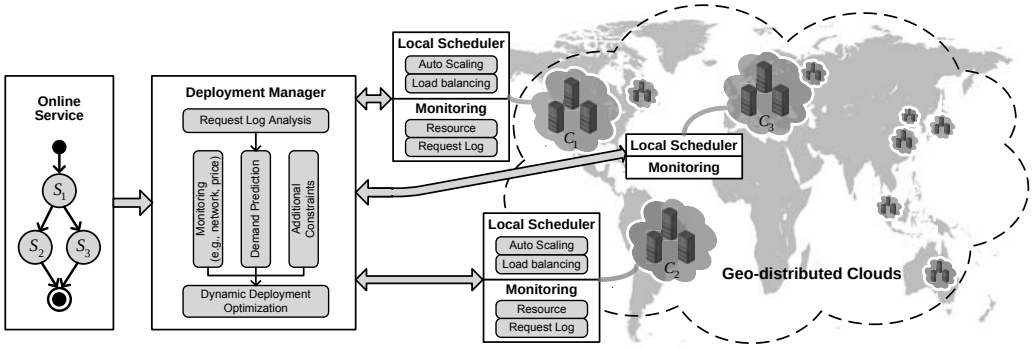


Figure 3.6: The Framework of Service Deployment in Geo-distributed Clouds

further increase the difficulty of this task. As a result, there is high demand for a dynamic and adaptive deployment strategy to scale cloud applications into geo-distributed clouds.

In this case study, we present a dynamic service deployment framework to cope with the deployment of online service systems across geo-distributed clouds. Online service systems are typically built ton SOA, which consist of a number of dependent service components and data components. Therefore, the deployment should be aware of these dependencies, since the deployment strategy of each service will directly impact the performance (*e.g.*, response time) of the application. To achieve so, in our framework, we focus on the data center selection problem for each service that takes service dependencies into account, which is formulated as an optimization problem. While the original model is a NP-hard problem, we tailor the genetic algorithm to solve it, which provides a good trade-off between the computational efficiency and the quality of the result.

3.5.2 Deployment Framework

Figure 3.6 illustrates our dynamic deployment framework, which aims to periodically optimize the deployment strategies of services while taking service dependencies into consideration. The process to deploy service-oriented applications across geo-distributed data

centers typically involves the following two phases: 1) deciding the deployment strategy for each service component, *i.e.*, selecting data centers to deploy each service components; and 2) automatically deciding the number of service instances for each service running in the data center. Our framework comprises a two-level management framework to solve these problems, including a deployment manager in data center level and a local scheduler in service instance level (either physical server or virtual machine). In the high level (*i.e.*, the data center level), the deployment manager is employed to analyze the service dependencies and predict the request demand to decide the number and locations of each service across data centers. In the low level (*i.e.*, the service instance level), the local scheduler is used to automatically scale the service instances according to the workload assigned to this data center.

- **Deployment Manager:** The deployment manager is a key component in our framework to determine the locations of each service in the geo-distributed clouds. Generally, the service provider makes an initial deployment and then iteratively improve the deployment. Towards this end, the request logs are collected and analysed to capture the user demand, and the network latencies can be measured periodically to adapt to the network dynamics. As such, the service deployment can be optimized periodically to improve the application performance, while service dependencies are considered in addition to the network condition, cloud service prices, and other additional constraints (*e.g.*, some data components are required to be placed in certain data centers for privacy concern). It is worth mentioning that there is a large body of work demonstrating the effectiveness of network coordinate systems for network performance prediction, which can also be incorporated into our framework to reduce the measurement overhead.

- **Local Scheduler:** After deploying the services into the s-

elected data centers, each local scheduler will automatically scale service instances according to the dynamic request workload and route the service requests with load balancing. For example, we can employ the auto scaling[5] and elastic load balancing[6] services in Amazon EC2 to implement the local scheduler. Furthermore, the request logs will be collected in each data center (*e.g.*, with the approach proposed in [22]) to facilitate the log analysis in the deployment manager. In each local scheduler, the platform-level resource allocation can be achieved, where the resource constraints of virtual machines are considered.

For simplicity of presentation, we focus only on the deployment manager component in this case study. In other words, we address how to select the candidate data centers for deployment, whereas the local scheduler problem is well studied in the literature (*e.g.*, [34]) as a resource allocation problem in a single data center.

3.5.3 Deployment Model and Algorithm

Application-level latency (*i.e.*, response time) typically denotes the time duration starting with a user request sent out and ending with a response to the user finally received, during which multiple invocations across services are performed. Generally, it can be computed as a summation of three elements: the involved communication delays between user and data centers and also those between data centers, the involved communication delays inside data centers, and the processing time for the service request. The second element, *i.e.*, the communication delays inside data centers, are negligible compared with the other elements, since machines in a data center are all connected by high-speed links. In addition, as the processing time of each service request is only affected by the computing capa-

[5] `http://aws.amazon.com/autoscaling`
[6] `http://aws.amazon.com/elasticloadbalancing`

Table 3.4: Notations of Deployment Model

Notations	Descriptions
N	Number of users
M	Number of service components in an application
f_{ij}	Frequency of invocations between user i and service j
c_j^m	The m-th datacenter deployed by service j
$d(i, c_j^m)$	Latency between user i and datacenter c_j^m
f_{jk}^i	Frequency of invocations between service j and service k for user i
C_j	Deployment strategy of servie j (the seleted data centers for service j)
C	The set of candidate datacenters
K_j	The number of instances of service j

bility of a service instance, we assume each service instance is the same and the total processing time is constant for each application. Consequently, for simplicity, we only study the relationship between the fist element and the service deployment strategy. Note that the processing time can also be easily incorporated into our following formulation.

$$\mathbf{min} \quad \sum_{i=1}^{N} \Big(\sum_{j=1}^{M} f_{ij} \cdot \min_{c_j^m \in C_j} d(i, c_j^m) + \sum_{j=1}^{M} \sum_{\substack{k=1 \\ k \neq j}}^{M} f_{jk}^i \cdot \min_{\substack{c_j^m \in C_j \\ c_k^n \in C_k}} d(c_j^m, c_k^n) \Big) \tag{3.12}$$

$s.t.$

$$C_j \subseteq C, \qquad \forall j = 1, 2, \cdots, M \tag{3.13}$$

$$|C_j| = K_j, \quad \forall j = 1, 2, \cdots, M \tag{3.14}$$

Equation 3.12∼3.14 present the constraint minimization formulation of our deployment model. In this model, the objective function aims at minimizing the total latencies of all requests including both user requests and cross-service requests. The notations in the

model are summarized in Table 3.4.

The intuition of our model is how to decide the deployment strategy (*i.e.*, the number and locations of selected data centers for each service) so as to minimize the user-perceived latencies. In detail, the frequency f_{ij} and f^i_{jk} indicates the dependencies between user-service requests and cross-service requests, as we jointly consider the sum of their latencies in our objective function. $\min_{c^m_j \in C_j} d(i, c^m_j)$ denotes the lowest network latency that user i can experience when invoking service j. $\min_{c^m_j \in C_j, c^n_k \in C_k} d(c^m_j, c^n_k)$ describes the lowest latency between dependent services j and k in the invocations. Moreover, the first constraint guarantees that for each service j the data centers C_j are selected from the whole candidate set C, while the second constraint ensures each service j will be deployed into K_j data centers. K_j keeps a trade-off between the user-perceived latency and the operational cost.

We can see that such a formulation is actually an NP-hard problem, which can be reduced to a set k-cover problem [56]. In [69], a service co-deployment model based on integer programming has been proposed to optimize the deployment strategy with potential service dependencies. However, the integer programming based approach suffers from poor scalability, as its complexity grows exponentially with the number of services and candidate data centers. Next, we will show how it can be solved efficiently with a genetic algorithm (GA).

Genetic algorithm is a popular search heuristic originated from the natural genetic systems to solve optimization problems. The basic idea of GA is to survive the fittest. Generally, the genetic algorithm works with a set of genomes called a population, where each genome is a feasible solution encoded with a string of integers. And each genome is associated with a fitness value, which indicates the possibility of survival and reproduction in the next generation for each individual genome. At each generation, the population goes through a set of operations, including selection, crossover,

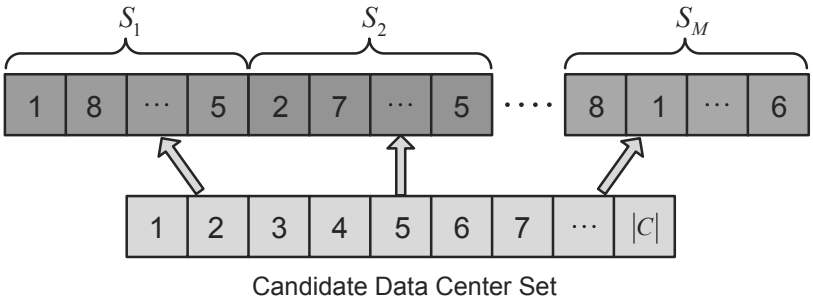


Figure 3.7: Genome Encoding of Deployment Strategy

mutation and evaluation, to evolve to the next generation. At the beginning, individuals in the population are selected in pairs as parents to take the crossover operation for each pair. The crossover operation randomly cuts off the original genome and swaps the parts to generate offsprings. Then the offsprings are placed back into the population to replace the weaker individuals. After the crossover operation, each genome will be mutated with some probability to change the genome into a new one. Finally, the evaluation operation is performed to update the fitness value of each genome. This process is repeated until some stop criteria are met (*e.g.*, until the best fitness value remains unchanged for a given number of generations, or the maximum number of generations has been reached).

In particular, we tailor the genetic algorithm by encoding the deployment strategy as shown in Figure 3.7. As we can see, each service j consists of a set of genes with size K_j, which can be selected from the candidate data center set. We jointly consider all the service components in a service-oriented application as a genome, and we can get different genomes by varying the values of the genes, as the indicator of each data center. For the parents selection, we use the *roulette wheel* method, which selects individuals of higher fitness values with higher probability. We use the classical *single-point crossover* operator and *real-value mutation* operator.

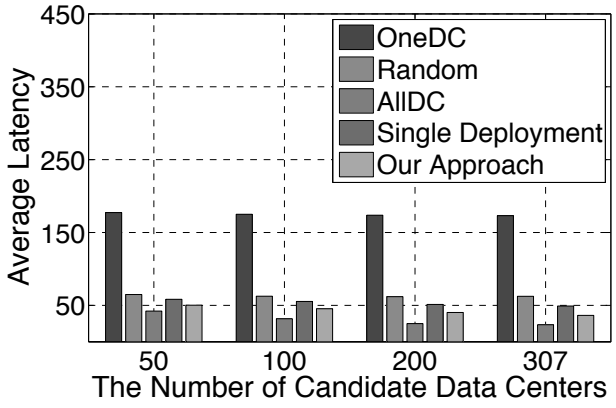


Figure 3.8: Comparison of Different Deployment Approaches

3.5.4 Results Analysis

To evaluate our proposed deployment model, we conduct experiments based on our newly collected response-time dataset, which includes 307 geo-distributed PlanetLab nodes and 1,881 public Web services. For evaluation purpose, we simulate user requests by randomly generating the request logs from our collected response-time dataset. By default, we set $N = 1881$, $M = 10$, and $|C| = 100$. For simplicity, we set all the values of K_j equally and the default setting is $K = 10$ for each j. A user of a service s would have 5 request logs. One request of a service would involve on average 5 requests of other services. In addition, the collected latency data are used to make the simulations realistic. In our evaluation, we compare our approach to the following four heuristics for service deployment. The experimental results show that our algorithm substantially outperforms these heuristic methods.

- **Random:** In *random* deployment, the services are deployed randomly in K data centers.
- **OneDC:** *OneDC* is proposed in [22], which deploys all the services in one data center with the highest performance of all the candidates. It is commonly employed by many companies

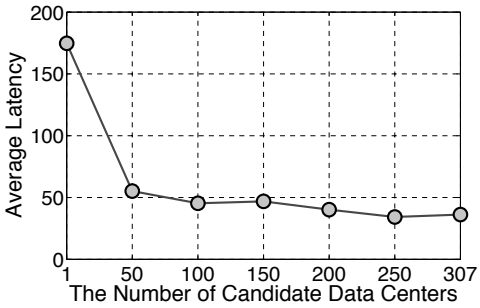


Figure 3.9: Impact of $|C|$

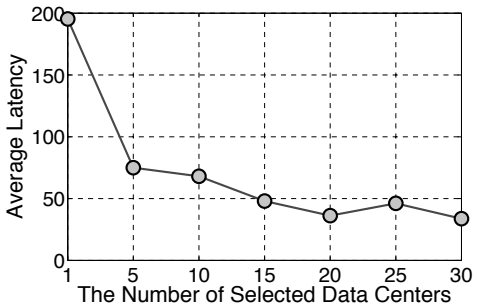


Figure 3.10: Impact of K

due to its simplicity, but the advantages of geo-distributed data centers are not considered.

- **AllDC:** *AllDC* simply deploys each service in every data center. It can be regarded as an extreme baseline, where the user-perceived latency is minimized with the operational cost ignored.
- **Single Deployment:** *Single Deployment* is proposed in [70], which optimizes the deployment strategy independently for each single service. Hence, it does not take service dependencies into account.

We compare the performance of different approaches with the results shown in Figure 3.8. We can observe that the OneDC heuristic performs worst, although it is widely employed due to its simplicity. AllDC can achieve the lowest average latency, yet with the the highest cost by selecting all the candidate data centers. Compared with AllDC, our approach obtains competitive performance result while saving cost by about 15$\times$. The experimental results show that the end-to-end response time information is immensely useful in performing service deployment optimization.

Impact of $|C|$

To study the impact of $|C|$, *i.e.*, the number of candidate data centers, we vary it from 1 to 307, and obtain the latency values. The results are shown in Figure 3.9. We can see a decreasing trend of the curve. But such reduction of latency gets smaller as the number of the candidate data centers increases. As a result, we can improve the application performance by deploying each service across multiple data centers, while the cost can be limited by using only a part of the candidate data centers. The reason is that with more candidates, our approach can find a better deployment strategy by considering service dependencies.

Impact of K

In this experiment, we vary the value of K from 1 to 30, to investigate the impact of K on user-perceived average latency. The experimental results are shown in Figure 3.10. We can see that the average latency decrease dramatically with the increasing of K (The little fluctuation at $K = 25$ may be caused by the random initial deployment). This is because with more data centers selected for service deployment, our approach can take advantage of the geo-diversity of distributed data centers to improve the performance for the dispersed users.

3.6 Summary

In this chapter, we propose a network coordinate based Web service positioning framework for response time prediction, which is one of the most important QoS properties. A small set of landmarks are deployed on the Internet to monitor the response times of all available Web services and provide references to the numerous service users. By combining the advantages of network coordinate based approaches and collaborative filtering based approaches,

our WSP framework is able to make accurate predictions on user perceived response times of Web services. Experimental results obtained on our collected real-world Web service dataset show that our proposed WSP approach alleviates the data sparsity problem of existing approaches and significantly enhances the prediction accuracy. Besides, our WSP approach can also serve for users without any available historical data, such as mobile users and new users, where existing approaches are not applicable.

This work focuses on response time prediction of Web services. However, there are some other QoS attributes that also deserve for future investigations to facilitate QoS-driven optimizations of online service systems. In addition, the limitation of Euclidean network coordinate systems lies in the triangle inequality violation problem of response times. Further exploration to mitigate this issue motivates our study in next chapter.

□ **End of chapter.**

Chapter 4

Online QoS Prediction of Web Services

With the scale and complexity of online service systems growing exponentially, it has become a significant challenge to maintain quality-of-service (QoS) guarantees. To mitigate service outage and degradation of service quality, online service systems have to become resilient against the QoS variations of their component services. Runtime service adaptation has been recognized as a key solution to achieve this goal. To aid in timely and accurate adaptation decisions, effective QoS prediction is desired to obtain the QoS values of component Web services. To achieve so, in this chapter, we focus on the study of online QoS prediction of Web services. In detail, we introduce the research problem in Section 4.1, and present the framework of QoS-driven service adaptation in Section 4.2. Then, we describe our online QoS prediction approach in Section 4.3. The evaluation results are reported in Section 4.4, and a case study is provided in Secton 4.5. Finally, Section 4.6 concludes this chapter.

4.1 Problem and Motivation

Cloud computing has gained increasing prevalence in recent years for providing a promising paradigm to host and deliver various

online services over the Internet. However, as these online service applications scale up, for example, spanning across multiple geographically distributed data centers [148], a significant challenge faced by application designers is how to engineer their applications with self-adaptation capabilities in response to the constantly changing operational environments, whereby the quality of service (QoS) can be guaranteed.

Many online service systems have employed service-oriented architecture (SOA) as a mechanism for achieving self-adaptation [92], where component services are composed in a loosely-coupled way to fulfill complex application logic. For example, Amazon's e-commerce platform is built on SOA by composing hundreds of component services hosted world-wide to deliver functionalities ranging from item recommendation to order fulfillment to fraud detection [51]. The features of SOA such as loose coupling and dynamic binding enable applications to switch component services without going offline, and thus make it particularly amenable to the introduction of service adaptation [90]. On the other hand, with the proliferation of cloud computing, many service providers begin to offer more and more services in the cloud that provide equivalent (or similar) functionalities through a well-defined interface (*e.g.*, Web service) [147]. Such redundant services can thus be utilized for service adaptation by replacing the current working services with the corresponding candidate services in response to unexpected QoS changes (e.g., unacceptable response time). To achieve so, knowledge about QoS values of the services is required to make timely and accurate adaptation decisions, such as when to trigger an adaptation action, which working services to be replaced, and which candidate services to employ. In particular, we refer to working services as the services that are being used by a cloud application, and candidate services as the alternative services that have equivalent functionalities.

For an online service system, the working services are frequently

invoked, thus their QoS values can be collected via monitoring. In recent literature, existing QoS prediction approaches (*e.g.*, [129, 76, 27]) for service adaptation focus mostly on monitoring (or predicting) QoS values of the working services, which can help determine when to trigger an adaptation action and which working services to be replaced. However, to the best of our knowledge, there is no work explicitly addressing the problem of QoS prediction on candidate services for service adaptation, thus making it difficult in determining which candidate services to employ for an adaptation action. It is challenging to obtain QoS values of the candidate services due to the prohibitive overhead for actively measuring a large number of candidate services at runtime. Besides, some service invocations may be charged, which further increases the cost of service invocations. Therefore, it is highly desired to employ QoS prediction approaches to accurately estimate the QoS values of candidate services without requiring direct invocations, which is exactly the goal of our work. In particular, effective QoS prediction on candidate services needs to fulfill the following requirements.

- ***Online***: The changing and evolving operational environment introduces a high degree of variability and uncertainty to user-perceived service quality. For instance, due to the impact of dynamic network conditions and varying server workload, the QoS values may vary significantly during different time periods. Therefore, in order to identify high-quality candidate services for service adaptation, QoS prediction needs to be performed in an online fashion.

- ***Accurate***: Ensuring the accuracy of QoS prediction is fundamental for service adaptation. Inaccurate predictions may lead to the execution of improper adaptations or missed adaptation opportunities. For example, a working service may be wrongly replaced by a low-quality service. Consequently, we need accurate QoS prediction approaches, as well as proper metrics

to evaluate the prediction accuracy.

- ***Scalable***: In the dynamic cloud environment, new services with different QoS may become available, and existing services may be discontinued by their providers. Likewise, service users may often join or leave the environment. In face of the high churning rate of users and services, QoS prediction approaches need to scale well to new services and users, and perform robustly to make accurate predictions.

To achieve these goals, in this chapter, we propose a novel QoS prediction approach to estimate the QoS values of candidate services by leveraging historical QoS data collaboratively from different users. The approach is inspired from the collaborative filtering model used in recommender systems, with the insight that different users may use a common set of services and some users may observe similar QoS on the same service. However, different from the conventional matrix factorization (MF) model applied in recommender systems, our problem is more specific to the QoS prediction problem due to the aforementioned stringent requirements. As a result, we extend the conventional MF model into an online, accurate, and scalable QoS prediction approach, namely adaptive matrix factorization (AMF) [172], by employing techniques of data transformation, online learning, and adaptive weights. To evaluate our AMF approach, comprehensive experiments including accuracy comparison, efficiency analysis, and scalability analysis are conducted based on a real-world large-scale Web service QoS dataset, which consists of response time and throughput data between 142 users and 4,500 services over 64 continuous time slices (at an interval of 15 minutes). The evaluation results provide good demonstration of our approach in achieving accuracy, efficiency, and scalability. Finally, a case study is presented to illustrate the use of online QoS prediction.

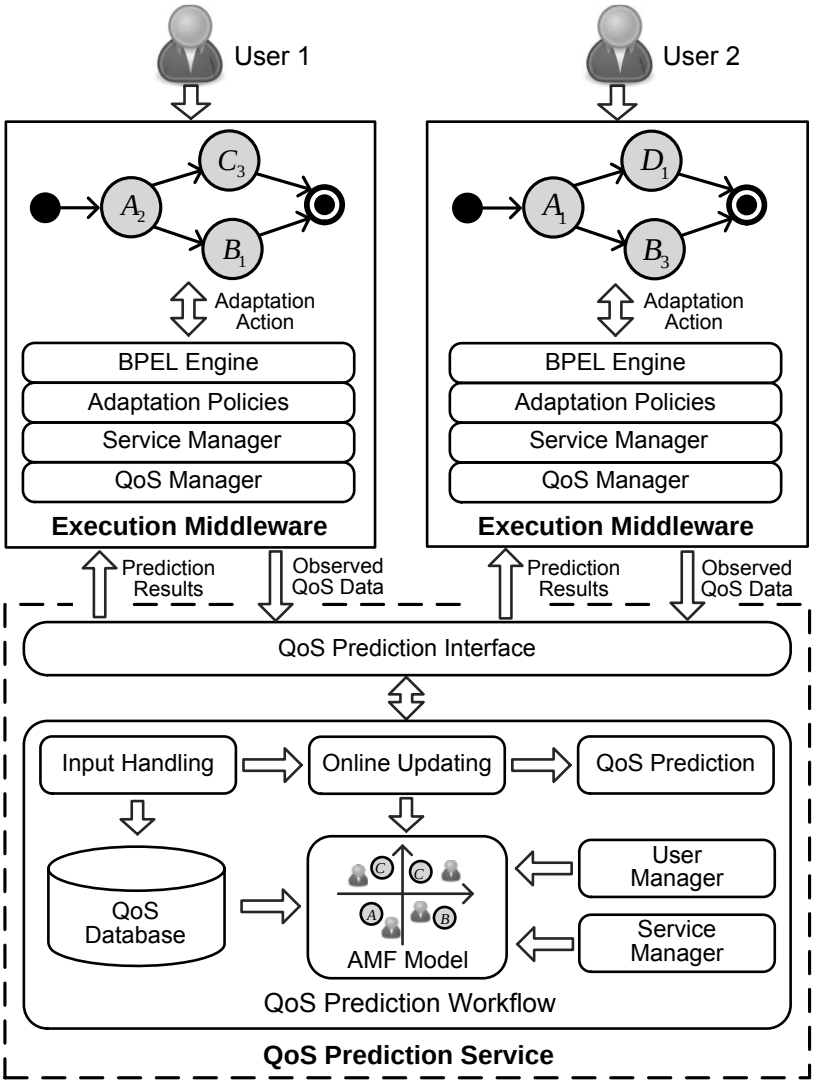


Figure 4.1: Framework of QoS-Driven Service Adaptation

4.2 Framework of QoS-Driven Service Adaptation

To build high-quality online service systems, we propose a basic framework for QoS-Driven service adaptation, as illustrated in Figure 4.1. In this framework, two modules are integrated to support QoS-Driven service adaptation.

Execution middleware: An service-oriented system typically comprises a workflow specified in BPEL and runs on a BPEL

engine, like *Apache ODE*[1]. In order to support QoS-Driven service adaptation actions (*e.g.*, replacing component services or restructuring workflows), BPEL engines can be enriched with sophisticated functionalities like QoS manager, candidate service manager, and user-specified adaptation polices. Concretely, candidate service manager discovers all available candidate services that match their needs, while QoS manager monitors the QoS values of service invocations, uploads the observed QoS data, and then obtains the related QoS prediction results through the interface of *QoS prediction service*. Based on the QoS prediction results, various adaptation polices (*e.g.*, when to trigger an adaptation action and, if necessary, which candidate services to employ) can be plugged in and executed automatically without causing any downtime of the overall system.

QoS prediction service: This module is designed as a service working by collaboratively collecting the observed QoS data from different users and then providing accurate QoS prediction results for these users transparently through a standard interface. More specifically, the QoS prediction service works as follows: 1) *Input handling*: The observed QoS data are collected and processed as formatted stream data. The QoS database can be updated accordingly. 2) *Online updating*: The AMF model can be updated online by using the sequentially observed QoS data. 3) *QoS prediction*: The QoS prediction results by our AMF model can be provided to users on demand through the QoS prediction interface. Additionally, a service manager is desired to provide utilities like service discovery and service management of available services. Likewise, a user manager is set up to manage the joining or leaving activities of users.

The framework shows that effective QoS prediction is fundamental for successful service adaptation executions in service-based online service systems, because the performance of service adaptation is heavily influenced by the QoS prediction results. Thus, QoS prediction is the main focus of this work.

[1] `http://ode.apache.org`

4.3 Online QoS Prediction

In the previous work, QoS prediction approaches (e.g., [129, 27]) focus primarily on QoS prediction for working services (that are being used by a cloud application) by employing techniques such as time series analysis on historical QoS data. According to the prediction results, potential SLA violations can be detected, thereby facilitating adaptation decisions such as when to trigger an adaptation action and which component services to be replaced. In contrary, our work focuses on QoS prediction for candidate services to help determine which candidate services to employ for an adaptation action.

Specifically, as with rating prediction in recommender systems, historical service invocations can produce a user-service QoS matrix with respect to each QoS attribute (*e.g.*, response time). This QoS matrix can be collected from user side in the form of user collaboration through our framework. In this matrix, each row denotes a service user (*i.e.*, a cloud application), each column denotes a candidate service in the cloud, and each entry denotes the QoS value observed by the a user when invoking a service. In practice, the QoS matrix is very sparse, since each user usually only invokes a handful of services. As in Figure 4.2(b), values in grey entries are observed QoS data from the user-service invocation graph in Figure 4.2(a), and the blank entries are unknown QoS values to be predicted. For example, the response time between user u_1 and service s_1 is $1.4s$, while the response time between user u_1 and service s_2 is unknown because u_1 has never invoked s_2.

Our goal of QoS prediction is to employ the observed QoS data to estimate the other unknown values. Formally, suppose there are n users and m services, we can obtain a sparse QoS matrix $R \in \mathbb{R}^{n \times m}$ with respect to each QoS attribute, where R_{ij} denotes the QoS value between user u_i and service s_j. As such, the QoS prediction problem can be modelled as a collaborative filtering problem that approximately reconstructs the unknown values from

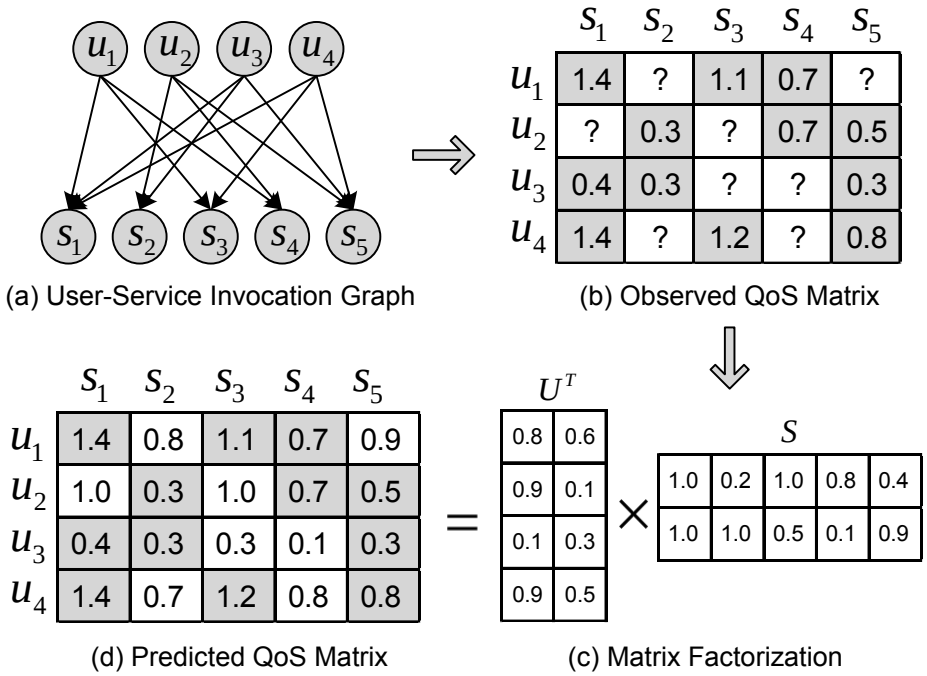


Figure 4.2: An Example of QoS Prediction by Matrix Factorization

a small number of observed entries [123]. In addition, the QoS prediction approach should be performed in an online, accurate, and scalable manner.

4.3.1 Matrix Factorization and Its Limitations

Matrix factorization [109] is a classic model to address the above collaborative filtering problem, which constrains the rank of the QoS matrix, *i.e.*, $rank(R) = d$. The low-rank assumption is based on the fact that the entries of R are largely correlated, thereby resulting in a low effective rank in R. For instance, close users may have similar network conditions, and thus experience similar QoS on the same service. Figure 4.2 illustrates an example that makes use of matrix factorization for QoS prediction. Concretely, factoring a matrix is to map both users and services into a joint latent factor space of a low dimensionality d (*e.g.*, $d = 2$ in Figure 4.2(c)), such that values of the user-service QoS matrix can be captured as inner products of

latent factors in that space. Then the latent factors can be employed for further prediction on unknown QoS values. For example, as shown in Figure 4.2(d), the predicted response time value between user u_1 and s_2 is $0.8s$.

Formally, latent user factors are denoted as $U \in \mathbb{R}^{d \times n}$ and latent service factors as $S \in \mathbb{R}^{d \times m}$, which are used to fit the QoS matrix R, *i.e.*, $R \approx U^T S$. To avoid overfitting, regularization terms that penalize the norms of the solutions (*i.e.*, U and S) are added. Thus we aim to minimize the following loss function:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (R_{ij} - U_i^T S_j)^2 + \frac{\lambda_U}{2} \|U\|_F^2 + \frac{\lambda_S}{2} \|S\|_F^2 , \quad (4.1)$$

where I_{ij} acts as an indicator that equals to 1 if R_{ij} is observed, and 0 otherwise (*e.g.*, $I_{11} = 1$ and $I_{12} = 0$ in Figure 4.2(b)). $\|\cdot\|_F$ denotes the *Frobenius norm* [109], and λ_U, λ_S are two parameters to control the extent of regularization. As introduced in Chapter 2, *gradient descent* [109] is usually employed to derive the solutions U and S, by iterating in the following form until convergence:

$$U_i \leftarrow U_i - \eta \frac{\partial \mathcal{L}}{\partial U_i}, \quad S_j \leftarrow S_j - \eta \frac{\partial \mathcal{L}}{\partial S_j}, \quad (4.2)$$

where η is the learning rate controlling how much change to make at each iteration. After obtaining the latent factors U and S, the unknown QoS values can then be predicted by their corresponding inner products: $\hat{R}_{ij} = U_i^T S_j$, where $I_{i,j} = 0$.

Although this conventional matrix factorization model performs well for rating prediction problem in recommender systems, it is insufficient to address our QoS prediction problem for service adaptation, due to the following limitations:

- ***Limitation 1***: Due to our observation on a real-world QoS dataset, we find that different from the coherent value range of ratings (*e.g.*, 1∼5) in recommender systems, the QoS values

vary widely (*e.g.*, $0\sim20s$ for response time and $0\sim7000kbps$ for throughput). Moreover, the distributions of QoS data are highly skewed with large variances (as shown in Figure 4.6(a)) compared with the rating distribution, which mismatches with the probabilistic assumption for matrix factorization [109]. Consequently, directly applying the original MF model to QoS data may significantly degrade its prediction accuracy.

- ***Limitation 2***: Our QoS prediction problem differs from recommender systems mainly in that QoS values are time-varying while rating values keep unchanged once being rated. In other words, existing QoS values will be continuously updated with newly observed values, or become expired after a time period without updating. However, conventional MF model primarily works offline on collected data. Therefore, to adapt to a newly observed QoS value, the MF model has to be entirely retrained, which will incur large computation overhead and make it infeasible to be performed online.

- ***Limitation 3***: Due to the dynamic nature of cloud environment, both users and services may continuously join or leave the environment (*i.e.*, churn occurs). However, the MF model focuses on the user-service QoS matrix with a fixed size (*w.r.t.* users and services), thus is not easily scalable to handle new users and new services without retraining the whole model.

4.3.2 Adaptive Matrix Factorization

To address the above limitations, we propose our new QoS prediction approach, adaptive matrix factorization (AMF), which aims to be online, accurate, and scalable. To achieve this goal, our AMF approach integrates three techniques: data transformation, online learning, and adaptive weights.

1) Data Transformation

To address *Limitation 1* (*i.e.*, skewed QoS value distributions), we apply a classic data transformation method, *Box-Cox* transformation [108], to QoS data. This technique is used to stabilize data variance and make the data more normal distribution-like in order to fit the matrix factorization assumption. The transformation is rank-preserving and performed by using a continuous power function defined as follows:

$$boxcox(x) = \begin{cases} (x^{\alpha} - 1)/\alpha & \text{if } \alpha \neq 0\ , \\ \log(x) & \text{if } \alpha = 0, \end{cases} \tag{4.3}$$

where the parameter α controls the extent of the transformation. For simplicity, we denote $\tilde{R}_{ij} = boxcox(R_{ij})$. Note that $\tilde{R}_{max} = boxcox(R_{max})$ and $\tilde{R}_{min} = boxcox(R_{min})$ due to its monotonously non-decreasing property of Box-Cox transformation. R_{max}, R_{min} are the maximal and minimal QoS values respectively, which can be specified by users (e.g., $R_{max} = 20$s and $R_{min} = 0$ for response time in our experiments). Similarly, $\tilde{R}_{max}$ and $\tilde{R}_{min}$ are the maximal and minimal values after data transformation. Then we map the data into the range $[0, 1]$ by linear normalization,

$$r_{ij} = (\tilde{R}_{ij} - \tilde{R}_{min})/(\tilde{R}_{max} - \tilde{R}_{min}). \tag{4.4}$$

Especially, when $\alpha = 1$, the data transformation is relaxed to a linear normalization, where the effect of Box-Cox transformation is masked.

To fit the normalized QoS data r_{ij}, we employ the sigmoid function $g(x) = 1/(1 + e^{-x})$ to map the value $U_i^T S_j$ into the range of [0, 1], as described in [109]. Therefore, the loss function in Equation 4.1 can be transferred to:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (r_{ij} - g_{ij})^2 + \frac{\lambda_U}{2} \|U\|_F^2 + \frac{\lambda_S}{2} \|S\|_F^2 , \tag{4.5}$$

where g_{ij} denotes $g(U_i^T S_j)$ for simplicity.

However, conventional matrix factorization model minimizes the sum of squared errors and employs the absolute error metrics (*e.g.*, MAE as defined in Section 4.4.2) to evaluate the prediction results. In practice, absolute error metrics are not suitable for evaluation of QoS prediction due to the large value range of QoS values. For instance, given two services with QoS values $s_1 = 1$ and $s_2 = 100$, the corresponding thresholds for adaptation action are set to $s_1 > 5$ and $s_2 < 90$. Suppose there are two sets of prediction results: (a) $s_1 = 8$ and $s_2 = 99$, (b) $s_1 = 0.9$ and $s_2 = 92$, we would choose (a) with smaller MAE if using the MAE metric. However, prediction (a) will cause a wrong adaptation action due to $s_1 > 5$, while prediction (b) is more reasonable. Consequently, we propose to employ relative error to evaluate the prediction results, where the corresponding loss function is derived as follows:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} \big(\frac{r_{ij} - g_{ij}}{r_{ij}} \big)^2 + \frac{\lambda_U}{2} \left\| U \right\|_F^2 + \frac{\lambda_S}{2} \left\| S \right\|_F^2 , \qquad (4.6)$$

2) Online Learning

To address *Limitation 2* (*i.e.*, time-varying QoS values), online learning algorithms are required to keep continuous and incremental updating using the sequentially observed QoS data. For this purpose, we employ a classic online learning algorithm, stochastic gradient descent (SGD) [117] to train our AMF model. For each QoS value observed by user u_i for invoking service s_j, we have the following pairwise loss function:

$$\ell(U_i, S_j) = \frac{1}{2} (\frac{r_{ij} - g_{ij}}{r_{ij}})^2 + \frac{\lambda_u}{2} \left\| U_i \right\|_2^2 + \frac{\lambda_s}{2} \left\| S_j \right\|_2^2 , \qquad (4.7)$$

such that $\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} \ell(U_i, S_j)$. $\left\| \cdot \right\|_2$ denotes the *Euclidean norm*.

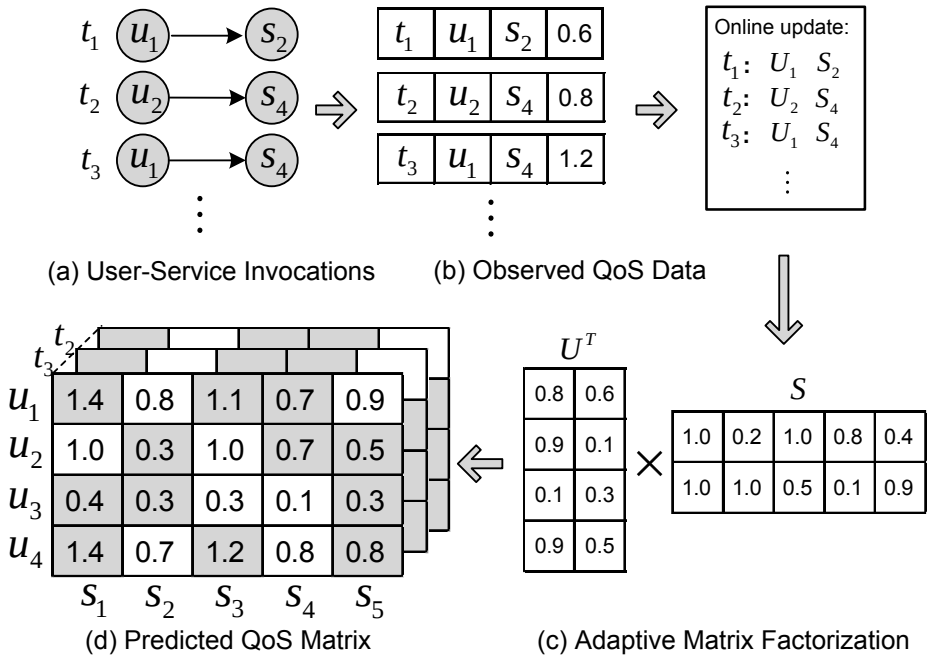


Figure 4.3: An Example of QoS Prediction by Adaptive Matrix Factorization

Instead of directly minimizing $\mathcal{L}$, SGD relaxes to minimize the pairwise loss function $\ell(U_i, S_j)$. By replacing $\mathcal{L}$ with ℓ in Equation 4.2, we can derive the following update equations regarding each data sample (u_i, s_j, R_{ij}):

$$U_i \leftarrow U_i - \eta((g_{ij} - r_{ij})g'_{ij}S_j/r_{ij}^2 + \lambda_u U_i), \tag{4.8}$$

$$S_j \leftarrow S_j - \eta((g_{ij} - r_{ij})g'_{ij}U_i/r_{ij}^2 + \lambda_s S_j), \tag{4.9}$$

where g'_{ij} denotes $g'(U_i^T S_j)$, and $g'(x) = e^x/(e^x+1)^2$ is the derivative of $g(x)$. η is the learning rate.

As illustrated in Figure 4.3(a)(b), every time when a new data sample is observed, online updating can be performed on its corresponding factors using Equation 4.8 and 4.9. In other words, at each iteration, user u_i can take a small change on feature vector U_i and service s_j can have a small change on feature vector S_j, given a newly observed data sample (u_i, s_j, R_{ij}) after user u_i invoke service s_j.

3) Adaptive Weights

To address *Limitation 3* (*i.e.*, scalability on new users and services), we make use of the above online learning algorithm, which can update the feature vectors incrementally without retraining the whole model. However, the above online learning algorithm may not perform well under the high churning rate of users and services (*i.e.*, continuously joining or leaving the environment). The convergence is controlled by the learning rate η, but a fixed η will lead to problems for new users and services. For example, for a new user u_1, if its feature vector U_1 is at its initial position, larger η is needed to help it move quickly to its correct position. But for an existing service s_2 that user u_1 invokes, its feature vector S_2 may have already been converged. Adjusting the feature vector (S_2) of service s_2 according to the user u_1 is likely to increase prediction error rather than to decrease it, since user u_1 itself has large prediction error with its initial feature vector (U_1) not converged. Thus, our approach, if performed online, need to be robust towards the churning of users and services.

To achieve this goal, we propose to employ adaptive weights in our AMF model. Although the weighted matrix factorization has also been studied in [119], our approach differs from it in that we use adaptive weights instead of fixed weights in the iteration process. Specifically, we design an adaptive weight to control the step size at each iteration, depending on the accuracy achieved by the corresponding user or service. The goal is to mitigate the impact of new users or services that have high errors with their feature vectors not converged. Intuitively, an accurate user should not move much according to an inaccurate service while an inaccurate user need to move a lot with respect to an accurate service, and vice versa. For example, if service s_1 has an inaccuracy of 10% and service s_2 with inaccuracy 1%, when a user invokes both s_1 and s_2, it should move less for its feature vector to service s_1 compared with service

s_2. As a result, we have two weights w_{u_i} and w_{s_j} for user u_i and service s_j respectively. Then we derive the following loss functions corresponding to U_i and S_j:

$$\ell_w(U_i) = \frac{1}{2} w_{u_i} (\frac{r_{ij} - g_{ij}}{r_{ij}})^2 + \frac{\lambda_u}{2} \|U_i\|_2^2, \tag{4.10}$$

$$\ell_w(S_j) = \frac{1}{2} w_{s_j} (\frac{r_{ij} - g_{ij}}{r_{ij}})^2 + \frac{\lambda_s}{2} \|S_j\|_2^2, \tag{4.11}$$

where $w_{u_i} + w_{s_j} = 1$, such that $\ell(U_i, S_j) = \ell_w(U_i) + \ell_w(S_j)$.

We denote the average error of user u_i as e_{u_i} and the average error of service s_j as e_{s_j}. Then we compute the weights w_{u_i}, w_{s_j} to control the credence between each other, as follows:

$$w_{u_i} = e_{u_i}/(e_{u_i} + e_{s_j}), \;\; w_{s_j} = e_{s_j}/(e_{u_i} + e_{s_j}). \tag{4.12}$$

To update e_{u_i}, e_{s_j}, we compute the exponential moving average [10] at each iteration, which is a weighted average with more weight (controlling by β) given to the latest data.

$$e_{u_i} = \beta w_{u_i} e_{ij} + (1 - \beta w_{u_i}) e_{u_i}, \tag{4.13}$$

$$e_{s_j} = \beta w_{s_j} e_{ij} + (1 - \beta w_{s_j}) e_{s_j}, \tag{4.14}$$

where e_{ij} denotes the relative error between g_{ij} and r_{ij}:

$$e_{ij} = |r_{ij} - g_{ij}|/r_{ij}. \tag{4.15}$$

We also find that similar weights have been used for controlling the credence of node in network coordinate system [50], but our approach is the first to incorporate such weights into matrix factorization. After obtaining the updated weights w_{u_i} and w_{s_j} at each iteration, we can derive the following updating equations by computing the gradients in Equation 4.10 and 4.11:

$$U_i \leftarrow U_i - \eta w_{u_i} ((g_{ij} - r_{ij}) g'_{ij} S_j / r_{ij}^2 + \lambda_u U_i), \tag{4.16}$$

$$S_j \leftarrow S_j - \eta w_{s_j} ((g_{ij} - r_{ij}) g'_{ij} U_i / r_{ij}^2 + \lambda_s S_j), \tag{4.17}$$

With U_i and S_j, we can predict the unknown QoS value R_{ij} (where $I_{ij} = 0$) for the service invocation between user u_i and service s_j. Finally, a backward data transformation of $g(U_i^T S_j)$ is required, which can be computed according to the inverse functions for Equation 4.3 and 4.4.

4) AMF Algorithm

After analyzing the ingredients of our AMF model, we can have a big picture of the algorithm. Figure 4.3 presents an illustrative example for QoS prediction by using AMF. Different with MF in Figure 4.2, our AMF approach collects each observed QoS value in a stream way (Figure 4.3(a)(b)), and keeps online updating accordingly (Figure 4.3(c)). Then the current QoS valuse can be predicted using the updated model (Figure 4.3(d)). The pseudo code of our online updating algorithm for AMF is provided in Algorithm 3. Specifically, at each iteration, the newly observed QoS data are collected to update the model (Line 2 $\sim$ 9), or else existing data are randomly selected for model updating (Line 11 $\sim$ 15) until convergence. Especially, the online updating operations are defined as a function $OnlineUpdate(t_{ij}, u_i, s_j, R_{ij})$ given a data sample $(t_{ij}, u_i, s_j, R_{ij})$, according to the steps described in 1) $\sim$ 3). Note that for a newly observed data sample, we first check whether the corresponding user or service is new, so that we can add it to our model (Line 5 $\sim$ 7) and keep updating its feature vector using more observed data on this user or service (Line 8 $\sim$ 9). As such, our model can scale to new users and services without retraining the whole model. Another important point is that we check whether an existing QoS value has become expired (Line 12), and if so, discard this value (*i.e.*, in Line 15, set $I_{ij} = 0$). In our experiment, for example, we set the expiration time interval to 15 minutes.

Algorithm 3: Adaptive Matrix Factorization Algorithm

Input: Sequentially observed QoS data samples: $(t_{ij}, u_i, s_j, R_{ij})$, and all the model parameters.

Output: QoS prediction results: $\hat{R}_{ij} \leftarrow (U_i, S_j)$, where $I_{ij} = 0$.

```
repeat        /* Continuous and incremental updating */
    Collect newly observed QoS data;
    if receive a new data sample (t_ij, u_i, s_j, R_ij) then
        I_ij ← 1;
        if u_i is a new user or s_j is a new service then
            Randomly initialize U_i ∈ R^d, or S_j ∈ R^d;
            Initialize e_{u_i} ← 1, or e_{s_j} ← 1;
        Update t_ij, R_ij corresponding to u_i, s_j;
        OnlineUpdate(t_ij, u_i, s_j, R_ij);
    else
        Randomly pick an existing data sample (t_ij, u_i, s_j, R_ij);
        if t_now − t_ij < TimeInterval then
            OnlineUpdate(t_ij, u_i, s_j, R_ij);
        else
            Existing data sample is obsolete: set I_ij ← 0;
    if converged then
        Wait until observing new QoS data;
until forever;

OnlineUpdate(t_ij, u_i, s_j, R_ij):                    /* Function */
Normalize R_ij by Equation 4.3 and 4.4: r_ij ← R_ij;
Update w_{u_i}, w_{s_j} by Equation 4.12: w_{u_i} ← (e_{u_i}, e_{s_j}),  w_{s_j} ← (e_{u_i}, e_{s_j});
Compute e_ij by Equation 4.15: e_ij ← (r_ij, g_ij);
Update e_{u_i}, e_{s_j} by Equation 4.13 and 4.14:
e_{u_i} ← (w_{u_i}, e_ij, e_{u_i}),   e_{s_j} ← (w_{s_j}, e_ij, e_{s_j});
Update U_i, S_j simutaneously by Equation 4.16 and 4.17;
```

4.4 Evaluation

In this section, we conduct a set of experiments based on a real-world Web service QoS dataset to evaluate our AMF approach from various aspects, including accuracy comparison, impact of parameters, efficiency analysis, and scalability analysis. All the

experiments were conducted on a machine with a 3.2 GHz Intel CPU and 4 GB RAM, running Win7. For ease of reproducing or applying our approach to future research, we publicly release our source code and dataset on our project page[2].

4.4.1 Data Description

In our experiments, we focus primarily on two QoS attributes: response time (RT) and throughput (TP). Response time stands for the time duration between user sending out a request and receiving a response, while throughput denotes the data transmission rate (*e.g.*, $kbps$) of a user invoking a service.

The data used in our experiment are extracted from a real-world Web service QoS dataset [152], including both response time and throughput values. These QoS values are collected by 142 users invoking 4,500 Web services for 64 consecutive time slices, at an interval of 15 minutes. The users are 142 machines (PlanetLab nodes) located in 22 countries, and the services are 4,500 publicly available real-world Web services from 57 countries [152]. Figure 4.4 provides some basic statistics of our dataset. For example, the range of response time is $0\sim20s$, and the throughput ranges $0\sim7000kbps$. Furthermore, we plot the data distributions of response time and throughput in Figure 4.6(a). For better visualization, we cut off the response time beyond $10s$ and the throughput more than $150kbps$. It is shown that the data distributions are highly skewed. In contrast, as shown in Figure 4.6(b), we obtain more normal data distributions through our data transformation in Section 4.3.2.

In addition, we investigate the singular values of the data matrices of response time and throughput between users and services. The singular values are computed by a singular value decomposition (SVD) [17] and then normalized so that the largest singular value is equal to 1, as illustrated in Figure 4.5. We can observe that except

[2] `http://wsdream.github.io/AMF`

Statistics	Values
#Users	142
#Services	4,500
#Time slices	64
#Time interval	15*min*
RT range	$0 \sim 20s$
RT average	$1.33s$
TP range	$0 \sim 7000kbps$
TP average	$11.35kbps$

Figure 4.4: Data Statistics

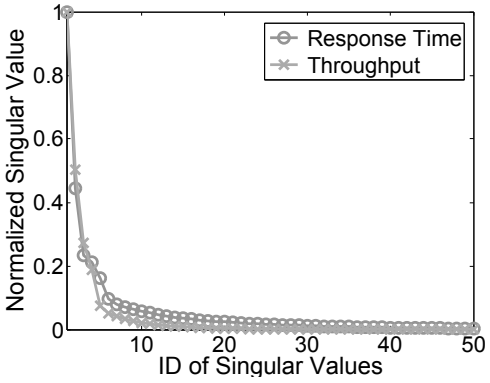
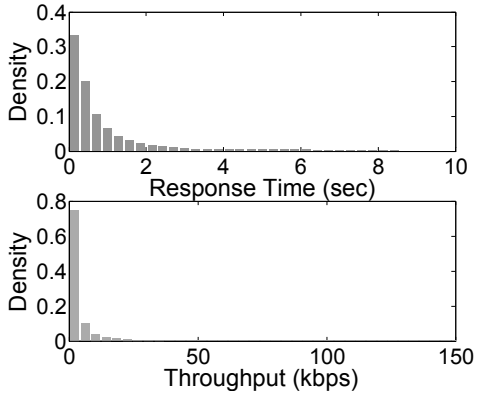
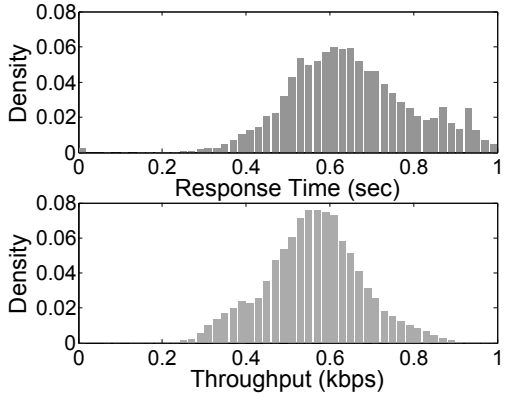


Figure 4.5: Sorted Singular Values



(a) Before Data Transformation



(b) After Data Transformation

Figure 4.6: Data Distribution Before and After Data Transformation

the first few largest singular values, most of them are close to 0. This observation indicates that both data matrices are approximately low-rank, which conforms to the low-rank assumption of matrix factorization. In our experiment, we set rank $d = 10$ (*i.e.*, the dimensionality of U_i, S_j).

4.4.2 Evaluation Metrics

We evaluate the prediction accuracy of our proposed approach in comparison with other existing approaches by using the following

metrics.

- **MAE** (Mean Absolute Error). MAE is widely employed to measure the average prediction accuracy in recommender systems, defined as follows:

$$MAE = \sum_{I_{ij}=0} \left| \hat{R}_{ij} - R_{ij} \right| \Big/ N, \tag{4.18}$$

  where R_{ij} is the measured value and $\hat{R}_{ij}$ is the corresponding predicted value. N is the number of samples that satisfy $I_{ij} = 0$.

- **MRE** (Median Relative Error). MRE takes the median value of all the pairwise relative errors:

$$MRE = \underset{I_{ij}=0}{median} \left| \hat{R}_{ij} - R_{ij} \right| \Big/ R_{ij}. \tag{4.19}$$

- **NPRE** (Ninety-Percentile Relative Error). NPRE takes the 90th percentile of all the pairwise relative errors.

Due to the large variance of QoS values, in this work, we focus more on relative error metrics, *i.e.*, MRE and NPRE, which are more appropriate for QoS prediction evaluation. Many papers report on MAE, so it is also included for comparison purpose. Nevertheless, our optimization efforts are not focused on MAE.

4.4.3 Accuracy Comparison

In order to evaluate the prediction accuracy, we compare our AMF approach with the following approaches that have been introduced for QoS prediction [162, 164]. It is worth noting that although these approaches are included for comparison purpose, they cannot be directly used for runtime service adaptation in practice, due to the aforementioned limitations.

- **UPCC**: This is a user-based collaborative filtering approach [162] that employs the similarity between users to predict the QoS values.
- **IPCC**: This is an item-based collaborative filtering approach [162] that employs the similarity between services to predict the QoS values.
- **UIPCC**: This is a hybrid approach proposed in [162], by combing both UPCC and IPCC approaches to make full use of the similarity between users and the similarity between services for QoS prediction.
- **PMF**: This is a widely-used implementation of matrix factorization model [109], which we have introduced in Section 4.3.1.

As we mentioned before, the available QoS data matrix is sparse in practice, because each user typically only uses a small number of candidate services out of all of them. To simulate the sparse situation, we randomly remove entries from the data matrix at each time slice so that each user only keeps a few available historical values. In this way, we vary the matrix density from 10% to 50% at a step increase of 10%. Matrix density = 10%, for example, indicates that each user invokes 10% (*i.e.* about 450) of the services, and each service is invoked by 10% (*i.e.* about 14) of the users. For AMF approach, the preserved data entries are randomized as a QoS data stream for training. Then the removed entries are used as the testing data to evaluate the prediction accuracy. In the sequel, for simplicity, we set $\lambda_u = \lambda_s = \lambda$ for AMF. Specifically, in this experiment, we set $d = 10$, $\lambda = 0.001$, $\beta = 0.3$, $\eta = 0.8$, $\alpha = -0.007$ for RT, and $\alpha = -0.05$ for TP. Note that the parameters of the other approaches are also optimized accordingly to achieve their optimal accuracy.

At each time slice, each approach is performed 20 times for each matrix density (with different random seeds). Then the average

Table 4.1: Prediction Accuracy w.r.t. MAE

QoS	Approach	Matrix Density				
		10%	20%	30%	40%	50%
RT	UPCC	0.8500	0.7696	0.7313	0.7050	0.6862
	IPCC	0.9460	0.8977	0.8573	0.8238	0.7888
	UIPCC	0.8482	0.7719	0.7332	0.7057	0.6843
	PMF	0.8332	0.7731	0.7443	0.7265	0.7104
	AMF	**0.7288**	**0.7034**	**0.6936**	**0.6892**	**0.6863**
	Improve.(%)	12.5%	8.9%	5.2%	2.2%	-0.3%
TP	UPCC	9.5011	8.4699	7.8835	7.5548	7.3504
	IPCC	9.6634	8.9234	7.9731	7.4345	7.0241
	UIPCC	9.3104	8.3855	7.5166	7.0149	6.6556
	PMF	6.0431	5.6822	5.3076	5.0687	4.8068
	AMF	**5.5427**	**5.4356**	**5.2974**	**5.1901**	**5.1809**
	Improve.(%)	8.3%	4.3%	0.2%	-2.4%	-7.8%

Table 4.2: Prediction Accuracy w.r.t. MRE

QoS	Approach	Matrix Density				
		10%	20%	30%	40%	50%
RT	UPCC	0.6484	0.5425	0.5054	0.4801	0.4610
	IPCC	0.7761	0.7525	0.7109	0.6807	0.6446
	UIPCC	0.6431	0.5510	0.5181	0.4944	0.4739
	PMF	0.5283	0.5269	0.5237	0.5205	0.5099
	AMF	**0.3096**	**0.2807**	**0.2667**	**0.2587**	**0.2542**
	Improve.(%)	41.4%	46.7%	47.2%	46.1%	44.9%
TP	UPCC	1.6503	1.4134	1.2571	1.1595	1.0909
	IPCC	0.7859	0.7124	0.6255	0.5855	0.5556
	UIPCC	1.4363	1.2611	1.0947	1.0172	0.9628
	PMF	0.4699	0.4477	0.4253	0.4012	0.3863
	AMF	**0.3551**	**0.3178**	**0.3007**	**0.2916**	**0.2854**
	Improve.(%)	24.4%	29.0%	29.3%	27.3%	26.1%

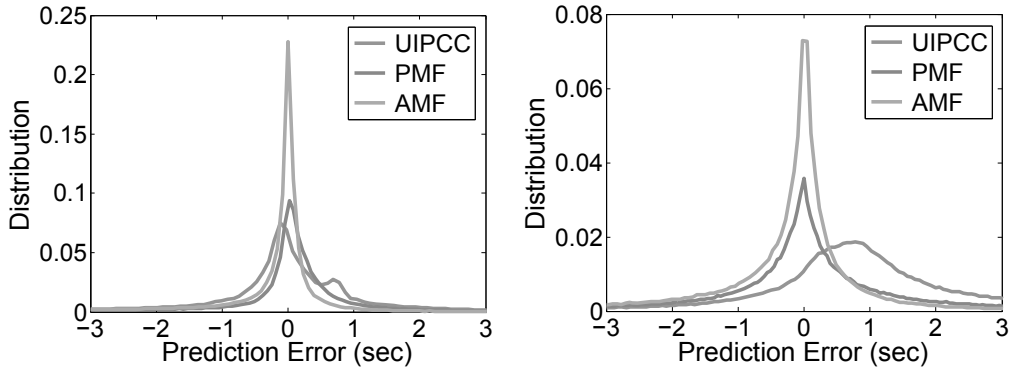
values over all the time slices (*i.e.*, 20×64 times) are reported. Table 4.1$\sim$4.3 provides the comparison results over three metrics, but we focus more on relative error metrics, *i.e.*, MRE and NPRE. As we can observe, our AMF approach significantly outperforms the other approaches over MRE and NPRE, while still achieving comparable (or best) results on MAE. The accuracy improvements of our

Table 4.3: Prediction Accuracy w.r.t. NPRE

QoS	Approach	Matrix Density				
		10%	20%	30%	40%	50%
RT	UPCC	5.4251	4.1452	3.7130	3.4341	3.2375
	IPCC	5.7514	5.5029	5.2877	5.0301	4.7026
	UIPCC	5.3820	4.3172	3.9556	3.6991	3.4904
	PMF	2.8231	3.0672	3.1161	3.3160	3.0427
	AMF	**0.9728**	**0.8994**	**0.8667**	**0.8502**	**0.8414**
	Improve.(%)	65.5%	70.7%	72.2%	74.4%	72.3%
TP	UPCC	17.3322	16.8860	16.8194	16.8934	16.9664
	IPCC	11.4606	10.4361	8.8113	8.0981	7.6114
	UIPCC	15.0760	14.2780	13.2519	12.8740	12.6269
	PMF	2.1754	2.4413	2.4966	2.4129	2.3976
	AMF	**1.3506**	**1.0622**	**0.9607**	**0.9244**	**0.9013**
	Improve.(%)	37.9%	56.5%	61.5%	61.7%	62.4%

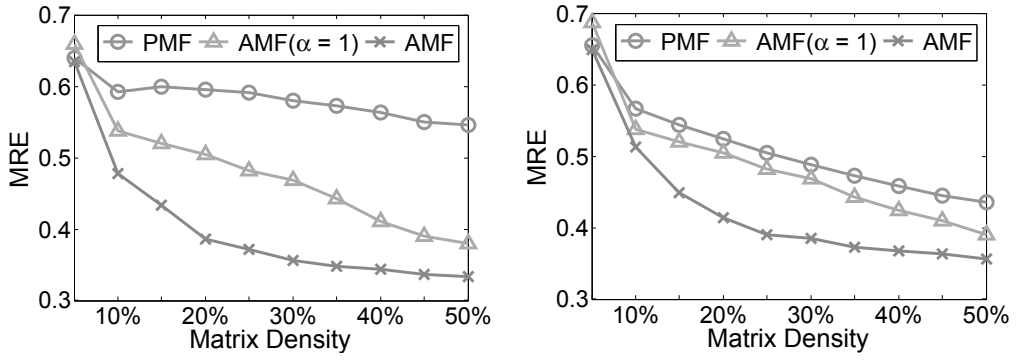
AMF approach (marked in bold) against the best of other existing approaches (marked with underline) are shown in each table. Concretely, for response time (RT) data, AMF achieves 41.4%$\sim$47.2% improvement on MRE and 65.5%$\sim$74.4% improvement on NPRE at different matrix densities. Similarly, for throughput (TP) data, AMF has 24.4%$\sim$29.3% MRE improvement and 37.9%$\sim$62.4% NPRE improvement. Note that all improvements are computed as the percentage of how much AMF outperforms the other most competitive approach. We also find that although UIPCC achieves higher accuracy over MAE than UPCC and IPCC as reported in [162], and PMF achieves better performance compared with the first three approaches as reported in [164], all these approaches have large errors over MRE and NPRE. Thus, only minimizing the absolute error may lead to large relative error, which is not suitable for QoS prediction problem.

To further analyze the benefit of our AMF approach, we plot the distributions of prediction errors in Figure 4.7. We can observe that AMF achieves denser distribution around the center 0, while UIPCC and PMF have flat error distributions, which indicates the



(a) Response Time (b) Throughput

Figure 4.7: Distribution of Prediction Error



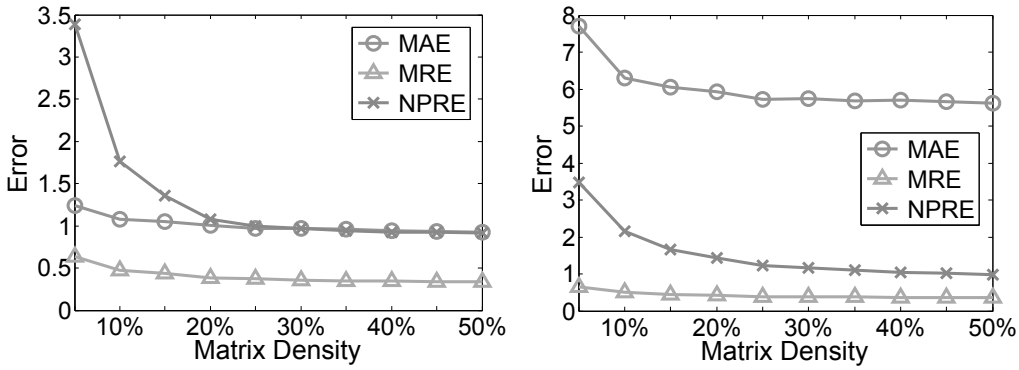
(a) Response Time (b) Throughput

Figure 4.8: Impact of Data Transformation

better performance of AMF.

4.4.4 Impact of Data Transformation

The impact of data transformation on data distributions has been illustrated in Figure 4.6. To further evaluate the impact of data transformation on prediction accuracy, we compare the prediction accuracy among three approaches, including PMF, AMF($\alpha = 1$), and AMF. In AMF($\alpha = 1$), $\alpha = 1$ indicates that the data



(a) Response Time (b) Throughput

Figure 4.9: Impact of Matrix Density

transformation is relaxed to a linear normalization procedure, since the effect of the function $boxcox(x)$ is masked. In contrast, AMF is our approach with a well-tuned α (*e.g.*, $\alpha = -0.007$ for response time and $\alpha = -0.05$ for throughput). In this experiment, we also vary the matrix density and then compute the corresponding MRE values. The results are illustrated in Figure 4.8. We can observe that the data transformation method has a significant impact on improving prediction accuracy over MRE. Especially, the PMF approach aggressively minimizes the absolute error, resulting in large MRE as shown in Figure 4.8. Besides, AMF improves a lot in MRE compared with AMF($\alpha = 1$) due to the effect of Box-Cox transformation on QoS data distributions.

4.4.5 Impact of Matrix Density

To present a comprehensive evaluation on the impact of the matrix density, we vary the matrix density from 5% to 50% at a step increase of 5%. Besides, we set the other parameters as in Section 4.4.3. Figure 4.9 illustrates the evaluation results. We can observe that as the matrix density increases, better prediction accuracy can be achieved. In particular, the error decreases dramatically with

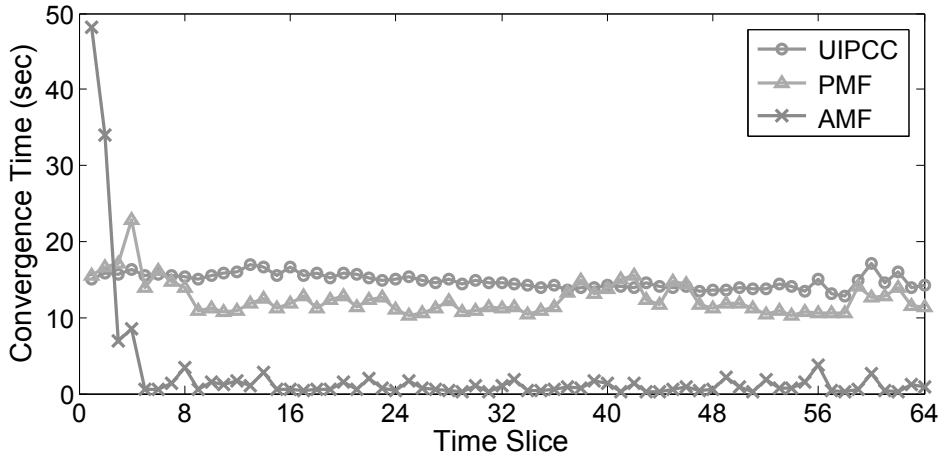


Figure 4.10: Efficiency Result

the increase of matrix density, when the QoS matrix is excessively sparse (*e.g.*, matrix density = 5%). It shows that the model can fall into the overfitting problem due to data sparsity. With more data collected, the overfitting problem can be alleviated, thus further improving QoS prediction accuracy.

4.4.6 Efficiency Analysis

To evaluate the efficiency of our approach, we compare the convergence time of AMF with two other approaches, UIPCC and PMF. As we can see in Figure 4.10, despite the long convergence time for the first time slice, our AMF approach becomes quite fast in the following time slices because AMF incrementally updates the model by online learning using sequentially observed data samples. Note that AMF has a long booting time because it needs a large number of iterations for convergence from randomly initialized values. After initial convergence, the online updating on converged values is fast. In contrast, UIPCC and PMF are more computationally expensive, since they need to re-train the whole model at each time slice, which incurs high computational overhead compared to our online algorithm. Thus, they are more appropriate for one-time training as

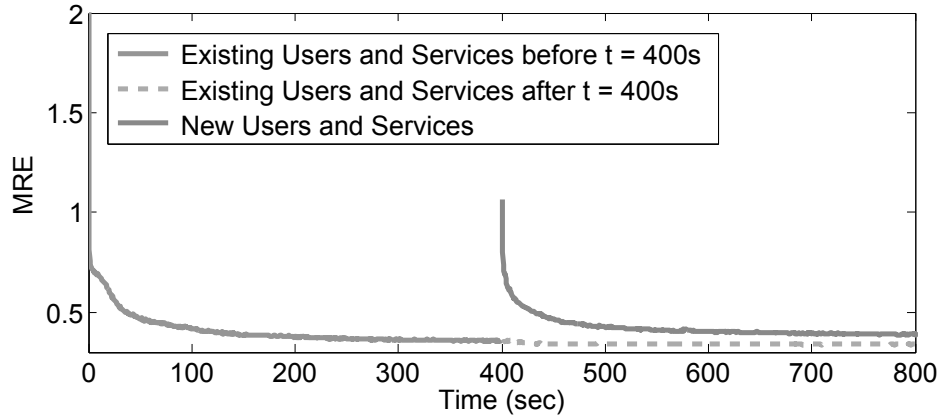


Figure 4.11: Scalability Result

used in traditional recommender system.

4.4.7 Scalability Analysis

To analyze the scalability of our AMF model on new users and services, we evaluate the prediction accuracy on these new users and services, as well as the robustness of the prediction results. For this purpose, we simulate the new users and services from our dataset. Specifically, we randomly select 80% of users and services from our dataset at time slice 1 as existing users and services, and then train the AMF model using their data. After the model converges, we add the remaining 20% of users and services into the model at time $t = 400s$. Ideally, by using our algorithm 1, AMF can scale well to the new users and services, and perform robustly by keeping updating the feature vectors of existing users and services with small weights, and the feature vectors of new users and services with large weights. Figure 4.11 presents the results, where we can see that the MRE for the new users and services rapidly decreases after their joining. However, the MRE for existing users and services still keep stable, which indicates the robustness of our model under the churning of users and services. Therefore, our AMF approach shows good scalability on new users and services.

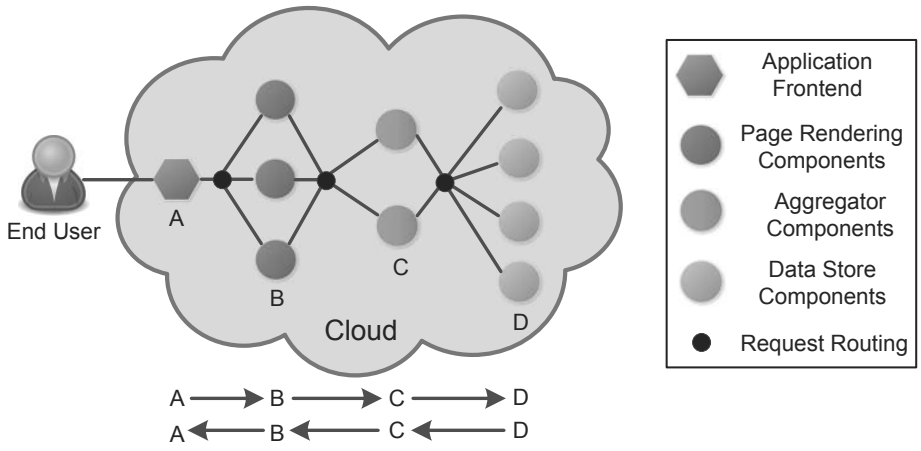


Figure 4.12: A Page Request Example in Amazon

4.5 Case Study

In this section, we present a case study of dynamic request routing for online service systems [175], which demonstrates the practical use of our online QoS prediction approach.

4.5.1 Dynamic Request Routing

Modern online service systems are typically large-scale and complex in system structures. Each service request encompasses a large number of interactions between component services, in some cases across hundreds of machines. Figure 4.12 depicts an example, taken from [51], of a page request to one of the e-commerce sites in Amazon. To generate the dynamic Web content, each request typically requires the page rendering components to invoke other aggregator components, which in turn query some other data store components to construct a composite response. These components are dependent between each other, and thus it is common to have a large call graph for an online service system [51]. To ensure that the page rendering engine can provide fluid response for maintaining seamless page delivery, the call chain between components must

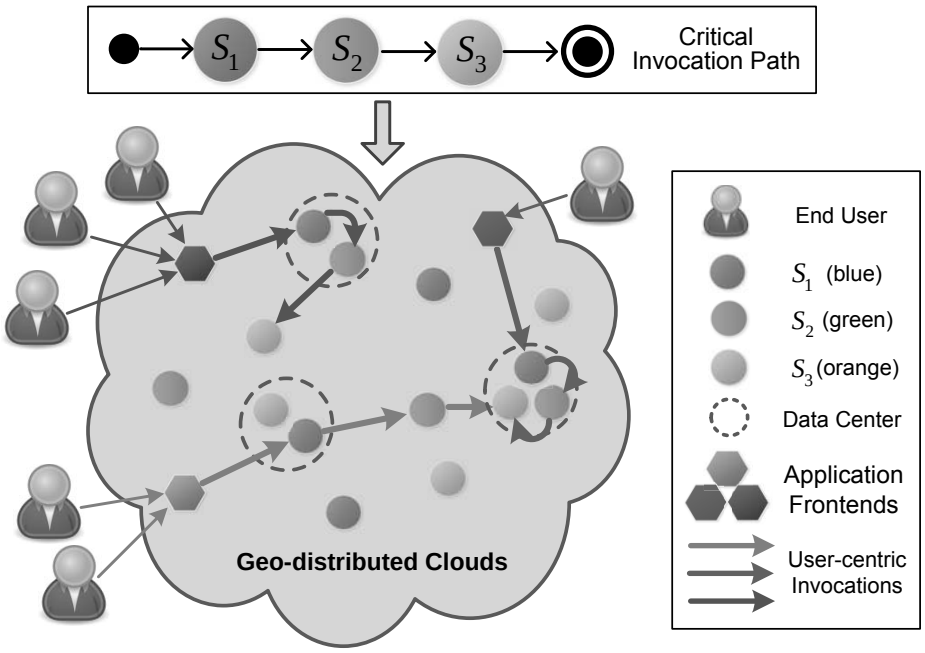


Figure 4.13: A Prototype of Dynamic Request Routing

have consistently low latency. As the system scale up, the components in an online service system have to be deployed across multiple geographically distributed data centers (*e.g.*, [148, 24, 176]), in order to locate resources closer to end-users. However, such geographical distribution can bring significant latency variations to the interactions between components. To build responsive online services, it is desired to leverage existing redundancy to tolerate latency variability.

To clarify our motivation in detail, we illustrate a prototype of dynamic request routing in Figure 4.13. As shown in the figure, a critical invocation path is given to define the invocation dependencies between a set of tasks (S_1, S_2, S_3), where each task can be completed by a corresponding cloud component. A critical path is typically the invocation path that determines the whole application latency, and can be identified from the application logic manually by engineers or using some automatic detection techniques. For simplicity, we do not elaborate how to construct a critical path here.

For each component, there are usually many redundant instances deployed in the cloud, which may be across multiple data centers. To improve the user experience when serving globally-distributed users, both application frontend servers and cloud components are placed at multiple geographically distributed locations to make the resources closer to end users. End users from different regions are usually directed to the closest application frontend server when sending an application request. Given a stream of application requests from different end users, our objective is to minimize the user-perceived application latency and tolerate the latency variability by performing dynamic request routing for each specific request from each frontend server. For example, the invocation paths, depicted in arrows with different colors, illustrate different request routing strategies while minimizing the length of each path.

In this case study, we present the use of dynamic request routing framework (namely DR^2) to achieve runtime adaptation of service invocations. The basic idea of DR^2 is to make dynamic selection among available redundant component when routing a traffic of application requests from different end-users. In this way, although the latencies between cloud components present high variability, we aim at minimizing the application latency of the whole call graph for each request, thus tolerating the low-level component-component latency variability.

4.5.2 DR^2 Architecture

In order to build seamlessly responsive online services, we propose a dynamic request routing framework by taking advantage of the redundant components to tolerate the latency variability. Our framework is adaptive, user-centric and scalable, which addresses all the challenges mentioned above.

The framework is illustrated in Figure 4.14, which includes the following two phases:

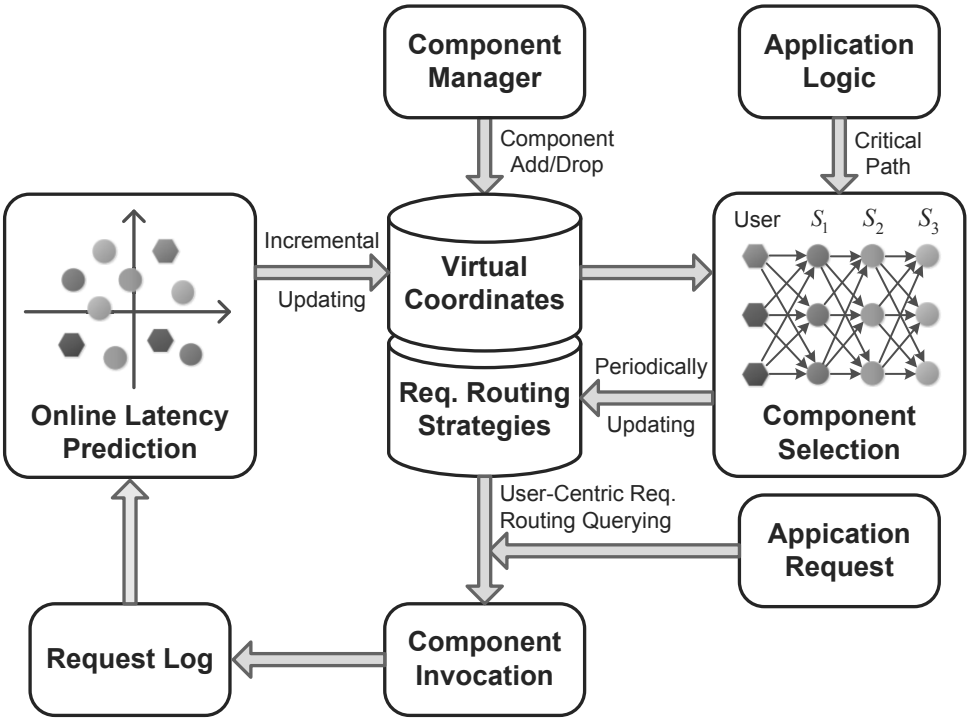
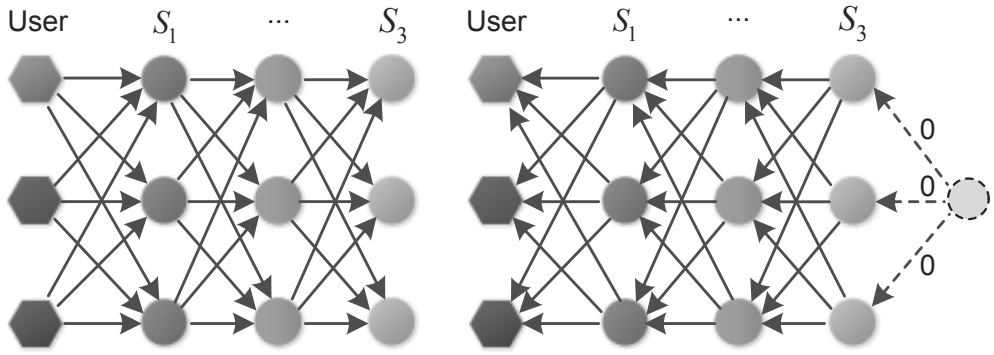


Figure 4.14: The Framework of Dynamic Request Routing (DR^2)

1) ***Online Latency Prediction****:* In order to make dynamic request routing, the precondition is to obtain real-time latency data between components. To overcome the overhead of active measurement, we resort to online latency prediction. First, request logs are collected to passively obtain the historical latency data. Then a matrix factorization model is trained using newly updated historical data in an online machine. That is, we assign each user or component a virtual coordinate, and incrementally update the corresponding coordinate through an online learning algorithm by using each data sample. In this way, our approach can adapt to the changes over time. The updated virtual coordinates can then be used to predict the latency.

2) ***Adaptive Component Selection****:* Given a critical invocation path, performing dynamic request routing is to make adaptive component selection for each task. First, we build an invocation graph of



(a) Invocation Graph (b) Virtual Graph

Figure 4.15: Graph Construction

available candidate components using the predicted pairwise latencies. Then, we propose an efficient shortest path to find the optimal component selection strategy for each user, which takes advantage of the graph characteristics (*e.g.*, DAG). Our algorithm works on a virtual graph which is transformed from the original invocation graph, thus reducing much overhead for multiple-user scenarios. The component selection needs to be performed periodically to adapt to the changing latency, and the results are stored in the database as request routing strategies.

When receiving an application request, the end user will be directed to one of the frontend server, and then a request routing strategy can be obtained by querying the database. After completing the request, the historical data of component invocations can be collected to update the matrix factorization model again.

4.5.3 DR^2 Approach

Given a critical invocation path, we can construct an invocation graph based on the redundant components for each task. Figure 4.15(a) illustrate an invocation graph example of the critical path in Figure 4.13. Notice that the number of redundant components for

Algorithm 4: Adaptive Component Selection

Input: Critical Path, Virtual coordinates: U_i, S_j, V_k
Output: Component selection strategy

```
1  Construct the virtual graph VG based on the critical path;
2  Topologically sort VG to VG_list;
3  foreach node v in VG_list do /* Initialization                    */
4      if v ∈ user then
5          v.out ← U_i; v.in ← none;
6      else v.out ← V_k; v.in ← S_j;
7      if v is in the last level of the critical path then
8          v.latency ← 0;
9      else v.latency ← inf;
10     v.parent ← none;
11 foreach node v in VG_list do
12     foreach node w in adjacency of v do
13         if w.latency lg v.latency + w.out'*v.in then
14             w.latency ← v.latency + w.out'*v.in;
15             w.parent ← v;
16 foreach node v ∈ user do /* Output the selection strategy
   v.path                                                           */
17     add v.parent to v.path;
```

each task may be different.

To tolerate the high latency variability and improve the user experience, it is vital to make optimal component selection to minimize the application latency in each time slice and periodically re-optimize the selection strategy. The problem is to find the shortest path in an invocation graph from the user to the last level of components of the critical path. As the shortest path experiences the minimal latency, the components in the shortest path are the optimal request routing strategy.

Conventionally, for example in service computing, only one user is considered for each service composition [28]. However, with the prevalence of cloud computing, the application frontend servers are usually deployed across many locations to serve end users around

the world. Hence, there are many users in the invocation graph. The approaches in the literature do not suffice to deal with this problem efficiently, since they make component selection for each user independently. In this way, the complexity will be n times as a single run for one user, which is quite inefficient.

To address this problem, we propose an efficient algorithm, which make the component selection collaboratively for all users by taking full advantage of the structure of invocation graph. To avoid finding the shortest path from each user, we construct a virtual graph based on the original invocation graph, as shown in Figure 4.15(b). In the virtual graph, we reverse the edge direction while keeping the original weight. Then a virtual node is added as the source node, thus we can employ the single-source shortest path algorithm. The objective is to find the shortest path from the virtual node to all the user nodes. In this way, we can collaboratively find all the needed shortest paths, which only needs one-time updating.

The detailed algorithm is shown in Algorithm 4. Our algorithm takes advantage of the property that the virtual graph is a directed acyclic graph (DAG), where linear-time shortest path algorithm exists [124].

4.5.4 Results Analysis

In our experiment, we use three datasets: two real-world latency datasets and one synthetic dataset.

- **Dataset1**: This dataset collected in [175] contains a 1350×460 user-to-component latency matrix and a 460×460 component-to-component latency matrix.
- **Dataset2**: This dataset is extracted from a time-aware Web service QoS dataset [152], which contains the latency data from 4,532 users and 142 components for 64 time slices.

Table 4.4: Performance Comparison

Approach	Density = 10%					
	10%	20%	30%	40%	50%	100%
Random	6.444	6.446	6.435	6.431	6.405	6.436
Greedy-M	0.888	0.613	0.517	0.506	0.496	0.656
DR^2	**0.412**	**0.269**	**0.163**	**0.129**	**0.089**	**0**

- **Dataset3**: The first two datasets are used to evaluate the accuracy of DR^2. However, the scale of components is small. To evaluate the efficiency of our approach on different scales, we also randomly generate a latency dataset.

Performance Comparison of Component Selection

To evaluate the performance of component selection, we compare our method with some other approaches in the following:

- **Random:** This approach is proposed as a baseline, in which the components are selected randomly from the redundant candidates.
- **Greedy-M:** The greedy approach is to route each user request to the closest component at each step. If there are n users, this greedy algorithm must be run for n times to get request strategy for every user, thus is denoted as Greedy-M. (M for multiple)
- **Dijkstra-M:** This is the most classic algorithm to find the shortest path, which is also used in [72] as a baseline approach. Given a original invocation graph, this method also needs to run multiple times if there are multiple users. We denote it as Dijkstra-M.

To compare the performance, we use the average relative error (ARE) metric. Relative error is the predicted error of the application latency divided by the true latency, while ARE is the average over

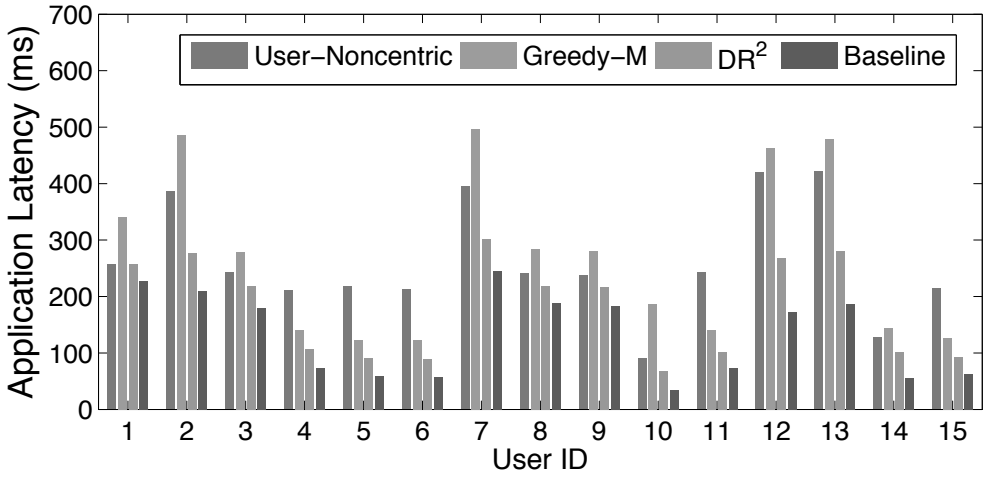


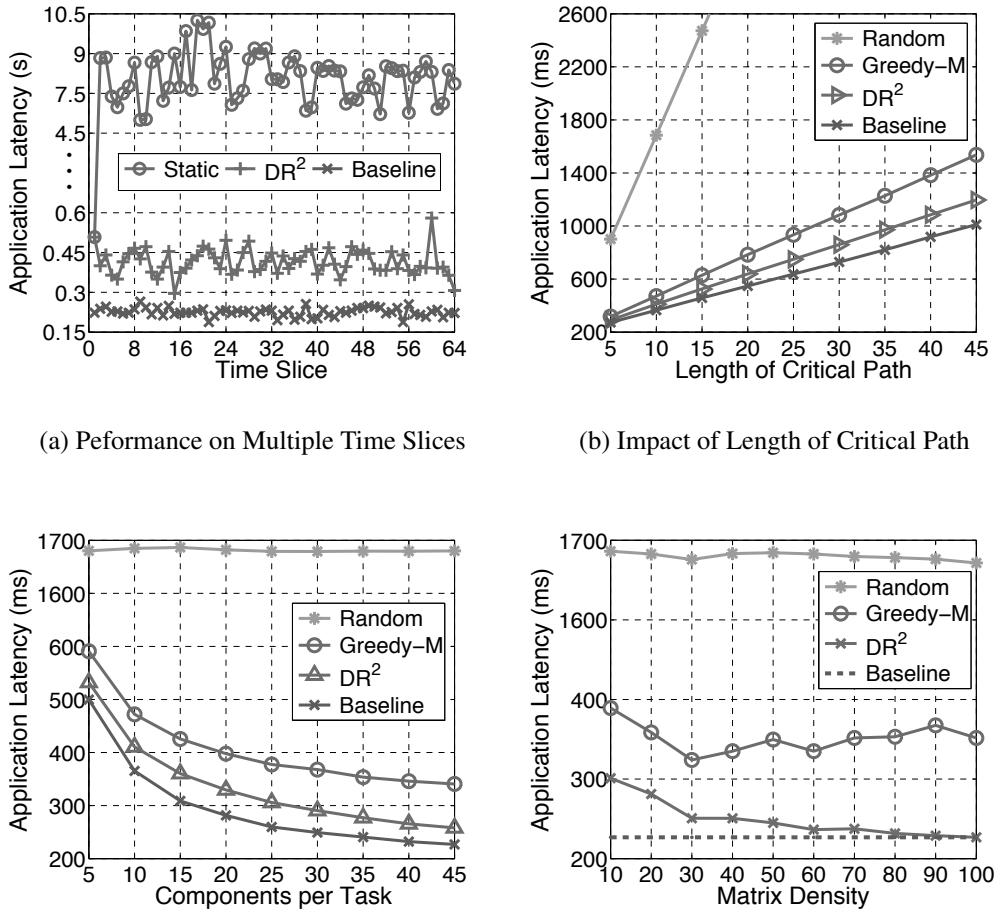
Figure 4.16: Performance on Multiple Users

100 randomly generated critical paths. For each method, we use the same 100 critical paths.

In this experiment, we set the length of critical path to 10, and each task has 45 components. The results with different densities are shown in Table 4.4, while both the average and the stand deviation are reported, since each experiment runs 20 times for each density. Our performance significantly outperforms the others. As Dijkstra-M has the same accuracy with DR^2, it is omitted for report here. With the increasing of density, the improvement of the performance diminishes, especially when density $\geq 30\%$. Density $= 100\%$ is also listed out since it is the result on the exact pairwise latency. Our method is totally correct. This column has no std since each run has the same result.

Performance on Multiple Users

A key feature of dynamic request routing is that there are usually multiple users, and user-centric request routing is in demand. In this experiment, we randomly select 15 users and get the average application latency. *User-Noncentric* is the method that does not consider the user-centric property and make the same request routing



(a) Peformance on Multiple Time Slices

(b) Impact of Length of Critical Path

(c) Impact of Components per Task

(d) Impact of Matrix Density

Figure 4.17: Performance Evaluation

for each user. Here it uses the same strategy with DR^2 for user 1. *Baseline* employs the exact pairwise latency for component selection, and provides a lower bound. The results are shown in Figure 4.16. We can observe that the application latency has high variability over users. And our method DR^2 can be user-centric and obtains good performance, while User-Noncentric and Greedy-M experiences high latency.

Performance on Multiple Time Slices

To evaluate the performance for tolerating the latency variability, experiments are conducted on *dataset2*. *Static* means the static request routing strategy for all time slices. Here it uses the same routing strategy with the one of DR^2 at time slice 1. From Figure 4.17(a), we can see that our approach can nearly adapt to the changes and keep consistently low latency, while the static request routing always experience high latency.

Impact of the Length of Critical Path

To evaluate the impact of the length of critical path, we vary it from 5 to 45 at a step of 5. In this experiment, we set the number of components to 10, the data density to 30%. From the result shown in Figure 4.17(b), we can see that the application latency is linear with the length of critical path.

Impact of the Number of Components per Task

To evaluate the impact of the number of component per task, we vary it from 5 to 45 at a step of 5. And also we set the number of components per task to 10, and the data density to 30%. We can see from Figure 4.17(c) that the application latency decreases as the number of components per task becomes larger, except for the random approach, which does not take advantage of the redundant components.

Impact of the Matrix Density

To present a comprehensive evaluation on the impact of the matrix density, we vary the density from 10% to 100% at the step of 10%. Besides, we set the length of critical path to 10 and the number of components per task to 45. For each density, 20 runs are performed and the average is reported, as shown in Figure 4.17(d). We can

observe that the application latency drops significantly when density increases and approach to the baseline when density $\geq 80\%$. The baseline is the application latency of routing request on exact latency data (density $= 100\%$). However, the random and greedy-M keep independent with the matrix density.

The experimental results of this case study show that, the use of AMF to online latency prediction, DR^2 is an effective framework for tolerating latency variability in online service systems. DR^2 has fast convergence in online latency prediction and high scalability in adaptive component selection, which substantially outperforms other approaches.

4.6 Summary

QoS information is fundamental for runtime service adaptation. This is the first work to address the problem of QoS prediction on candidate services to guide candidate service selection for runtime service adaptation. To achieve this goal, a novel QoS prediction approach, adaptive matrix factorization (AMF), is proposed by formulating the problem as a collaborative filtering model inspired by recommender system. AMF extends traditional matrix factorization model with techniques of data transformation, online learning, and adaptive weights, and thus fulfills the stringent requirements to be online, accurate, and scalable. Comprehensive experiments are conducted based on a real-world large-scale QoS dataset of Web services to evaluate our proposed approach in terms of accuracy, efficiency, and scalability. In addition, a case study on dynamic request routing demonstrates the practical use of our online QoS prediction approach for runtime service adaptation.

However, this work only considers the use of historical usage data for QoS prediction. In practice, user-perceived QoS values are influenced by a variety of contextual factors, such as user's location, invocation time, and service workload. Future investigations will fo-

cus on the incorporation of such information for context-aware QoS prediction, which may potentially achieve further improvements in prediction accuracy.

□ **End of chapter.**

Chapter 5

Privacy-Preserving QoS Prediction of Web Services

QoS-based Web service recommendation has recently gained much attention for providing a promising way to help users find high-quality services. To facilitate such recommendations, existing studies suggest the use of collaborative filtering techniques for personalized QoS prediction. These approaches, by leveraging partially observed QoS values from users, can achieve high accuracy of QoS predictions on the unobserved ones. However, the requirement to collect users' QoS data likely puts user privacy at risk, thus making them unwilling to contribute their usage data to a Web service recommender system. As a result, privacy becomes a critical challenge in developing practical Web service recommender systems. In this chapter, we make the first attempt to cope with the privacy issue in Web service recommendation. Specifically, we introduce the research problem in Section 5.1, and propose a simple yet effective privacy-preserving framework in Section 5.2. Under this framework, we develop two representative privacy-preserving QoS prediction approaches in Section 5.3. The evaluation results are reported in Section 5.4. Finally, Section 5.5 concludes this chapter.

5.1 Problem and Motivation

In recent years, Web service recommendation [161, 150, 45] has been proposed as a promising approach to help developers quickly find desirable services during development of online services. Effective service recommendation needs to fulfil both functional and non-functional requirements of users. While functional requirements focus on what a service does, non-functional requirements are concerned with the quality of service (QoS), such as response time, throughput, and failure probability, etc. QoS plays an important role in Web service recommendation, according to which similar services can be ranked and selected for users. Service invocations usually rely on the Internet for connectivity and are heavily influenced by the dynamic network conditions. Therefore, users at different locations typically observe different QoS values even on the same Web service. To enable personalized Web service recommendation, QoS evaluation from user side is desired. However, it is a challenge to acquire user-perceived QoS values of all the services because each user only has observed QoS values on a few used services. It is also impractical for each user to actively measure these QoS values due to the expensive overhead of invoking a large number of services.

To address this issue, collaborative QoS prediction has recently been proposed, and becomes a key step to QoS-based Web service recommendation. By applying collaborative filtering (CF) techniques [123] that are widely used in commercial recommender systems, unknown QoS can be predicted based on historical usage data collected from users, eliminating the need of additional service invocations. In other words, users can contribute their historical QoS data on the services they have used and receive prediction results on the QoS values of the services that they have never used before. In recent literature, a number of collaborative filtering approaches have been proposed for QoS prediction. Among them, neighbourhood-based CF approaches (e.g., UIPCC [161]) leverage the similarity

between users and/or the similarity between services calculated on the observed QoS data for unknown QoS prediction. Model-based approaches (e.g., PMF [164], EMF [79]) fit the observed QoS data with a pre-defined model (e.g., low-rank matrix factorization), and then utilize the trained model for QoS prediction. Recent studies have shown that these approaches achieve high accuracy of QoS predictions and yield encouraging results on Web service recommendation.

Despite the potential benefits provided by Web service recommender systems, a major impediment to the practical deployment of such systems lies in their threats to user privacy. To receive effective recommendations, users are required to supply their observed QoS values. However, there is currently no policy to protect users from privacy issues. Malicious recommender systems, for example, may abuse the data, infer private information from the data, or even resell the data to a competing user for profits [96]. Even if the recommender system is not malicious, an unintentional leakage of such data can expose users to a broad set of privacy issues (e.g., QoS data may reveal the underlying application configurations). This is why application providers are not willing to disclose their private usage data to the public or a third party. Such privacy threats limit the QoS data collection from users and hence degrade the accuracy of Web service recommendation. To encourage broader user participation, it is desired to consider privacy-preserving approaches for Web service recommendation that can be made without revealing private user data. Encryption is a straightforward way to achieve privacy. However, encryption techniques usually involve large computational overhead and typically work for distributed collaborative filtering problems (e.g., homomorphic encryption used in [38]) where multi-party communication is necessary. This is inapplicable to our problem because user-to-user communication is infeasible.

In this chapter, we propose a simple yet effective privacy-preserving

framework for QoS-based Web service recommendation [173]. Specifically, users are enabled to obfuscate their private data by data randomization techniques [100] before they expose the data to a recommender system. In this way, the recommender system can only collect obfuscated QoS data from users, and thus reduce the risk to expose user privacy. Our privacy-preserving framework is generic and can be applied to both the neighbourhood-based collaborative filtering approach, *i.e.*, UIPCC [162], and the model-based collaborative filtering approach, *i.e.*, PMF [164], which are two most common QoS prediction approaches in recent literature. We further revamp these two existing QoS prediction approaches based on our framework, and develop their corresponding privacy-preserving variants: P-UIPCC and P-PMF. We evaluate these approaches on WS-DREAM dataset [167], a publicly-available QoS dataset that has been widely employed for QoS prediction evaluation in the literature. The experimental results show that while preserving user privacy, our proposed approaches (P-UIPCC and P-PMF) can still attain decent prediction accuracy with comparison to the baseline approaches (UEAN and IMEAN) and the counterpart approaches (UIPCC and PMF). We also show the tradeoff between the achieved prediction accuracy and the preserved user privacy.

5.2 Framework of Privacy-Preserving Web Service Recommendation

Figure 5.1 presents our privacy-preserving Web service recommendation framework. The workflow of this framework can be separated into two parts executed at user side and server side respectively. At user side, the observed QoS data of each user undergo a data obfuscation process in order to protect user privacy as well as preserve the information required for performing collaborative QoS prediction. The obfuscated user data are then submitted to the server for QoS prediction. After receiving the prediction results

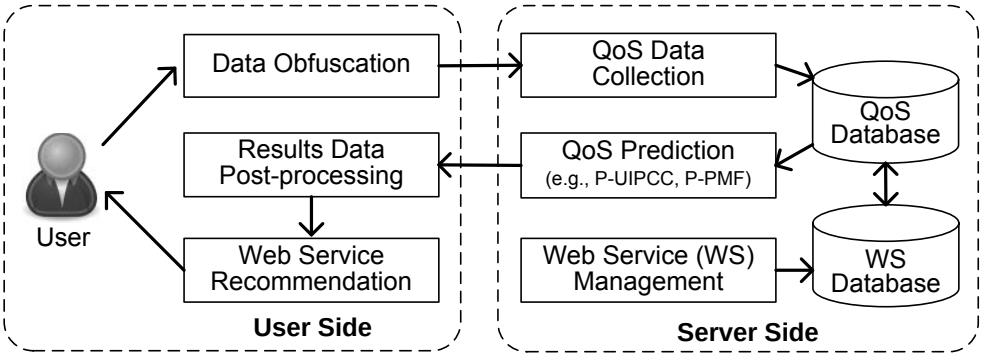


Figure 5.1: Framework of Privacy-Preserving Web Service Recommendation

from the server, a post-processing step is performed to recover the obfuscated results to the true QoS prediction values. At last, according to the recovered QoS values, candidate Web services can be ranked and recommended for the user. On the other hand, at server side, obfuscated QoS data are collected from different users in a collaborative way, through which a obfuscated QoS matrix can be acquired and stored in a QoS database. QoS prediction is then performed on the obfuscated QoS matrix by using our proposed privacy-preserving techniques such as P-UIPCC and P-PMF. At the same time, a list of the available Web services is maintained at a Web service database, which allows for service ranking and recommendation for the users.

We propose a general privacy-preserving framework such that two extended QoS prediction approaches (*i.e.*, PP-UIPCC and PP-PMF) under this framework can provide accurate recommendation results and also preserve user privacy. User privacy is preserved by our framework because: 1) For each user, user data are obfuscated before being submitted to the server, and the obfuscation settings are only known to the user itself; 2) For the server, collaborative QoS prediction is performed based solely on the obfuscated user data, whereby user-observed real QoS values cannot be inferred. In this way, our framework enables users with greater control on their private usage data and less dependence on the server for

privacy preservation. The privacy-preserving framework is generic such that both of the representative QoS prediction approaches (*i.e.*, UIPCC and PMF) can work well without the need of significant modifications.

5.3 Privacy-Preserving QoS Prediction

The above framework enables data obfuscation for preserving privacy, but also poses a challenge in accurate QoS prediction. In this section, we describe the data obfuscation process in detail, and then extend two representative QoS prediction approaches (UIPCC and PMF) into their privacy-preserving variants (P-UIPCC and P-PMF) accordingly. They are representatives of the two types of QoS prediction approaches and serve as a basis to develop many more sophisticated approaches.

5.3.1 Data Obfuscation

The need for privacy preservation has led to the development of a number of data obfuscation techniques, such as data randomization [100], data encryption [96], data anonymization [73]. Due to the sparse nature of our data, in this work, we make use of data randomization [100], a simple yet effective way to obfuscate the data.

The basic idea of data randomization is to add a random value (*i.e.*, noise) to the true value so that the resulting value becomes disguised. In this way, when the obfuscated QoS data undergo further processing, user information regarding real QoS values can be preserved. Fortunately, although each individual QoS value becomes disguised, we find that some approximate computations (*e.g.*, scalar product) on the aggregated data of users can still be done with decent accuracy.

To make it clear, we now describe the scalar product property of

data randomization [100] in detail. Let $a = (a_1, ..., a_n)$ and $b = (b_1, ..., b_n)$ be true vectors with a mean of zero. We obfuscate these vectors as $a' = a + \epsilon$ and $b' = b + \delta$, where $\epsilon = (\epsilon_n, ..., \epsilon_n)$ and $\delta = (\delta_n, ..., \delta_n)$ are random noises generated from a uniform distribution in $[-\alpha, \alpha]$. Next, we show that the scalar product between a and b can be approximated by using the obfuscated vectors a' and b': *i.e.*, $\mathbf{a'b'} \approx \mathbf{ab}$. To this end, we have

$$a'b' = \sum_{i=1}^{n}(a_i + \epsilon_i)(b_i + \delta_i) = \sum_{i=1}^{n}(a_i b_i + a_i\delta_i + b_i\epsilon_i + \epsilon_i\delta_i).$$

Because a and δ are independent vectors and each has a zero mean, we have $\sum_{i=1}^{n} a_i\delta_i \approx 0$. Likewise, we have $\sum_{i=1}^{n} b_i\epsilon_i \approx 0$, and $\sum_{i=1}^{n} \epsilon_i\delta_i \approx 0$. Hence, we derive the following approximation:

$$a'b' \approx \sum_{i=1}^{n} a_i b_i = ab. \tag{5.1}$$

With this observation, we find that data randomization can potentially preserve user privacy as well as the usability of the data for collaborative analysis. Therefore, it is appealing to study how to apply this data obfuscating technique to performing collaborative QoS prediction in a privacy-preserving way. To achieve this goal, we propose a two-step data obfuscation procedure for QoS data processing. We emphasize that, as shown in our framework in Figure 5.1, each user performs data obfuscation individually at user side before contributing the QoS data to the server.

Z-score normalization

To facilitate better randomization of the data, we perform z-score normalization on the observed QoS data as the first step. Z-score normalization is a standard normalization method to adjust the data average and data variance. The normalized data have a zero mean and unit variance. More specifically, for user u, we denote

$R_u = (R_{u1}, ..., R_{um})$ as a vector of observed QoS values on m Web services. $R_{us} > 0$ indicates that user u has invoked service s; otherwise, $R_{us} = 0$. We compute the mean ($\bar{R}_u$) and standard deviation (σ_u) of this QoS vector R_u:

$$\bar{R}_u = \sum\nolimits_{s \in I_u} R_{us}/|I_u|, \;\; \sigma_u = \sqrt{\sum\nolimits_{s \in I_u} (R_{us} - \bar{R}_u)^2/|I_u|}, \quad (5.2)$$

where $I_u = \{s \mid R_{us} > 0\}$ denotes the set of Web services that has been invoked by user u. Then z-score normalization is performed on the QoS values with the following equation:

$$r_{us} = (R_{us} - \bar{R}_u)/\sigma_u. \quad (5.3)$$

The normalization step results in a zero-mean data vector that is well suited for the following data randomization process.

Data Randomization

As the second step, we perform randomized perturbation on the normalized QoS vector by:

$$r'_{us} = r_{us} + \epsilon_{us}, \quad (5.4)$$

where ϵ_{us} is a random value generated from a specified distribution, for example, uniform distribution in $[-\alpha, \alpha]$. Especially when $\alpha = 0$, the overall data obfuscation process reduces to a z-score normalization. We further study the effect of different distributions (*e.g.*, uniform distribution, Gaussian distribution) of random noises on QoS prediction accuracy in Section 5.4.5.

After data obfuscation, users can submit their obfuscated QoS data to the server. Given n users and m services, the server can collect a QoS matrix denoted as $r' \in \mathbb{R}^{n \times m}$ with each entry (r'_{us}) being obtained via Equ. (5.4). Since such data obfuscation process is performed at user side, the private information such as $\bar{R}_u$ and σ_u are kept at user side. As a result, the server cannot infer the true QoS values of the users, and user privacy is preserved.

Next, we will show how we extend the two representative approaches (UIPCC and PMF) to perform privacy-preserving QoS prediction based on the obfuscated QoS matrix r'. Note that UIPCC and PMF have been carefully reported in the related work [162, 164], so we do not intend to provide the original descriptions but the necessary extensions from them.

5.3.2 Privacy-Preserving UIPCC (P-UIPCC)

UIPCC (a.k.a. WSRec), first proposed in [161], has been a widely-studied QoS prediction approach. The key of UIPCC is to compute the similarity between users and the similarity between services, after which QoS values contributed by similar users and similar services can be leveraged to compute the prediction value. Existing work usually employ Pearson correlation coefficient (PCC) as the similarity measure. For example, the PCC similarity between user u and user v is defined as follows:

$$sim(u,v) = \frac{\sum_{s \in J}(R_{us} - \bar{R}_u)(R_{vs} - \bar{R}_v)}{\sqrt{\sum_{s \in J}(R_{us} - \bar{R}_u)^2}\sqrt{\sum_{s \in J}(R_{vs} - \bar{R}_v)^2}}, \quad (5.5)$$

where $J = I_u \cap I_v$ is the set of Web services that are invoked by both user u and user v. R_{us} is the true QoS value of user u invoking service s. $\bar{R}_u$ and $\bar{R}_v$ are the average QoS values observed by user u and user v, respectively. From this definition, we have $sim(u,v) \in [-1,1]$, where a larger PCC value indicates higher user similarity.

However, due to the obfuscation of QoS data, at server side we only have obfuscated QoS value r'_{us}, rather than its true value R_{us}. Therefore, we consider to employ r'_{us} to approximately compute the

similarity value $sim(u, v)$ as follows:

$$sim(u,v) = \sum\nolimits_{s \in I_u \cap I_v} r'_{us} r'_{vs} / \sqrt{|I_u||I_v|} \tag{5.6}$$

$$\approx \sum\nolimits_{s \in I_u \cap I_v} r_{us} r_{vs} / \sqrt{|I_u||I_v|} \tag{5.7}$$

$$= \frac{\sum_{s \in I_u \cap I_v} (R_{us} - \bar{R}_u)(R_{vs} - \bar{R}_v)}{\sigma_u \sigma_v \sqrt{|I_u||I_v|}} \tag{5.8}$$

$$= \frac{\sum_{s \in I_u \cap I_v} (R_{us} - \bar{R}_u)(R_{vs} - \bar{R}_v)}{\sqrt{\sum_{s \in I_u} (R_{us} - \bar{R}_u)^2} \sqrt{\sum_{s \in I_v} (R_{vs} - \bar{R}_v)^2}}. \tag{5.9}$$

By applying the scalar product property in Equ. (5.1) to Equ. (5.6), substituting Equ. (5.3) to Equ.(5.7), and substituting Equ. (5.2) to Equ. (5.8), we derive Equ. (5.9), which is exactly the similarity measure used for collaborative filtering in the related work [100, 65]. Note that this similarity measure differs slightly from Equ. (5.5) in the denominator part, but provides a good approximation to it (as the experiments shown in Section 5.4). Therefore, by using the obfuscated QoS data, we employ Equ. (5.6) as the approximation of the similarity between user u and v.

After similarity computation between users, we can identify a set of top-k similar neighbours (T_u) for each user u. Then the unknown QoS value, for each entry where $r'_{us} = 0$, can be estimated as the weighted average of the QoS values observed by similar neighbours, *i.e.*,

$$\hat{r}^U_{us} = \sum\nolimits_{v \in T_u} sim(u,v) r'_{vs} \Big/ \sum\nolimits_{v \in T_u} sim(u,v). \tag{5.10}$$

In a similar way, we can also leverage the information of similar services to make QoS prediction:

$$\hat{r}^S_{us} = \sum\nolimits_{g \in T_s} sim(s,g) r'_{ug} \Big/ \sum\nolimits_{g \in T_s} sim(s,g), \tag{5.11}$$

where T_s is the set of top-k similar services of service s. The similarity $sim(s, g)$ is further calculated by employing the cosine

similarity between service s and service g:

$$sim(s,g) = \frac{\sum_{u \in I_s \cap I_g} r'_{us} r'_{ug}}{\sqrt{\sum_{u \in I_s \cap I_g} (r'_{us})^2} \sqrt{\sum_{u \in I_s \cap I_g} (r'_{ug})^2}}, \tag{5.12}$$

where $I_s \cap I_g$ represents the set of users that invoke both service s and service g. Note that the cosine similarity here equals to the original PCC similarity in UIPCC, because the QoS vectors have already been normalized during data obfuscation.

At last, as with UIPCC, a convex combination between user-based QoS prediction and service-based QoS prediction is employed to enhance the prediction accuracy.

$$\hat{r}_{us} = \lambda \hat{r}^U_{us} + (1-\lambda)\hat{r}^S_{us}, \tag{5.13}$$

where λ controls the combination weight between $\hat{r}^U_{us}$ and $\hat{r}^S_{us}$. Especially, when $\lambda = 0$, $\hat{r}_{us} = \hat{r}^S_{us}$, and when $\lambda = 1$, $\hat{r}_{us} = \hat{r}^U_{us}$.

However, this prediction result $\hat{r}_{us}$ is a normalized value that cannot reveal the prediction on the true QoS. When the user receives the prediction results from the server, a post-processing step, which is a re-normalization operation of the z-score normalization, can be taken to get the final prediction value $\hat{R}_{us}$:

$$\hat{R}_{us} = \bar{R}_u + \sigma_u * \hat{r}_{us}. \tag{5.14}$$

Note that the post-processing step can be only performed at user side because $\bar{R}_u$ and σ_u are only known to the user.

5.3.3 Privacy-Preserving PMF (P-PMF)

PMF, or probabilistic matrix factorization [109], as a popular model-based collaborative filtering approach, has been suggested for QoS prediction by prior work [79, 164]. PMF works on an essential assumption of the low-rank structure of the QoS matrix. A matrix has a low rank when the entries of the matrix are largely correlated.

In our case, as reported by the related work [162, 164], similar users usually have similar QoS values on the same Web service. The goal of PMF is to map n users and m services into a joint latent factor space with dimensionality d such that each observed entry of the QoS matrix can be captured as the inner product of the corresponding latent factors.

Formally, we denote the latent user factors as $U \in \mathbb{R}^{d \times n}$ whose u-th column represents the latent factor of user u, and the latent service factors as $S \in \mathbb{R}^{d \times m}$ whose s-th column represents the latent factor of service s. Accordingly, we use $U_u^T S_s$ to approximate the observed QoS value R_{us} between user u and service s, *i.e.*, $R_{us} \approx U_u^T S_s$, or more precisely,

$$R_{us} = U_u^T S_s + \delta_{us}, \tag{5.15}$$

where U_u^T is the transpose of U_u and δ_{us} denotes the approximation error. The goal is to minimize all of the approximation errors. By taking δ_{us} as Gaussian noise [109], the loss function can be formulated as follows:

$$\mathcal{L} = \frac{1}{2} \sum_{u=1}^{n} \sum_{s=1}^{m} I_{us} (R_{us} - U_u^T S_s)^2 + \frac{\gamma}{2} (\sum_{u=1}^{n} \|U_u\|^2 + \sum_{s=1}^{m} \|S_s\|^2). \tag{5.16}$$

The first part measures the sum of squared approximation errors between R_{us} and $U_u^T S_s$, where I_{us} acts as an indicator that equals to 1 if R_{us} is observed, and 0 otherwise. The second part are regularization terms used to avoid the overfitting problem, where $\|\cdot\|$ denotes the Euclidean norm, and γ is a parameter to control the extent of regularization.

According to the basic PMF model as specified in Equation (5.15), the specific QoS of user u invoking service s can be effectively captured by the interaction between U_u and service S_s. However, some other effects known as biases for determining the QoS values are independent of user-service interactions. For example, the users with high network bandwidth tend to experience

fast network connections and the services equipped with abundant system resources likely provide short request-processing time. To capture these factors associated with either users or services, there is a suggestion for biased matrix factorization model in [138]:

$$R_{us} = \mu + b_u + b_s + U_u^T S_s + \delta_{us}, \tag{5.17}$$

where μ is a global bias, and b_u and b_s measure the user bias and service bias respectively.

While preserving user privacy, the application of data obfuscation poses new challenges in modelling the obfuscated QoS data. To compromise the effect of data obfuscation, we set $\mu = 0$ and $b_u = \bar{R}_u$. Accordingly, we derive the following model:

$$r'_{us} = b'_s + {U'_u}^T S'_s + \delta'_{us} + \epsilon_{us}. \tag{5.18}$$

For ease of presentation, we further denote it as:

$$r'_{us} = b_s + U_u^T S_s + \delta_{us} + \epsilon_{us}. \tag{5.19}$$

This model naturally compromises the effect of z-score normalization at user side. By taking both δ_{us} and ϵ_{us} as Gaussian noise [109], the loss function can be expressed as:

$$\begin{aligned} \mathcal{L}' &= \frac{1}{2}\sum_{u=1}^{n}\sum_{s=1}^{m} I_{us}(r'_{us} - b_s - U_u^T S_s)^2 \\ &+ \frac{\gamma}{2}(\sum_{u=1}^{n} {b_s}^2 + \sum_{u=1}^{n} \|U_u\|^2 + \sum_{s=1}^{m} \|S_s\|^2). \end{aligned} \tag{5.20}$$

The minimization of this loss function can typically be solved by the gradient descent algorithm used in [164] or the stochastic gradient descent algorithm used in [138]. Due to space limits, we omit the algorithmic description here and refer interested readers to our supplementary report (see our project page). After obtaining the solutions with respect to b_s, U_u, and S_s, we can make the following QoS prediction:

$$\hat{r}_{us} = b_s + U_u^T S_s. \tag{5.21}$$

At last, as with P-UIPCC, a post-processing step in Equation (5.14) is required to recover the prediction result $\hat{r}_{us}$ to the true prediction value $\hat{R}_{us}$. For both P-UIPCC and P-PMF, after obtaining the predicted QoS values of all the available Web services, we can recommend to users those services with top-ranked QoS values.

5.4 Evaluation

This section describes the experiments and the corresponding results of evaluating our privacy-preserving QoS prediction approaches. In particular, we intend to answer the following research questions.

RQ1: What is the effect of data obfuscation?

RQ2: What is the accuracy of P-UIPCC and P-PMF?

RQ3: What is the tradeoff between accuracy and privacy?

RQ4: What is the effect of distribution of random noises on prediction accuracy?

For reproducibility, we release the source code on our project page[1].

5.4.1 Experimental Setup

In our experiments, we focus mainly on two representative QoS attributes: response time (RT) and throughput (TP). Response time measures the time duration between user sending out a request and receiving a response, while throughput stands for the data transmission rate of a user invoking a service.

The experiments are conducted based on a publicly-available QoS dataset of real-world Web services [167]. The dataset was collected in August 2009, providing a total of 1,974,675 response

[1] `http://wsdream.github.io/PPCF`

Table 5.1: Statistics of QoS Data

QoS	#Users	#Services	Range	Average	Std.
RT (sec)	339	5,825	$0 \sim 20$	0.909	1.973
TP ($kbps$)	339	5,825	$0 \sim 1000$	47.562	110.797

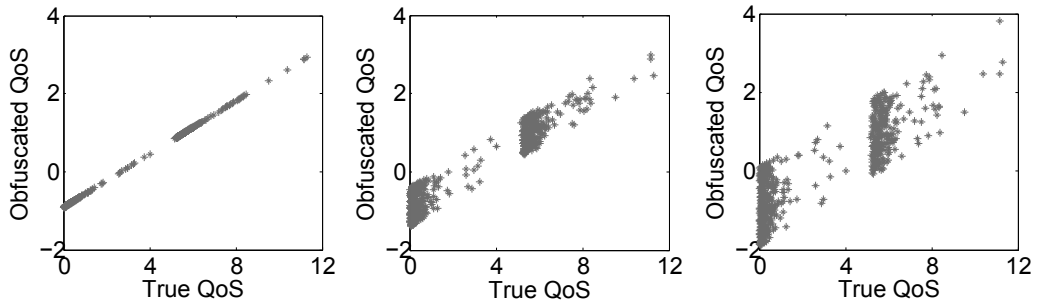
time and throughput records of service invocations between 339 users and 5,825 Web services. The 339 users are simulated by PlanetLab nodes distributed at 30 countries, while the 5,825 real-world Web services are crawled from the Internet and are deployed at 73 countries. Table 5.1 provides a summary of the statistics of the data.

In our experiments, we represent each type of QoS data by a 339-by-5825 QoS matrix with each entry denoting the observed response time/throughput of a specific invocation. In practice, the QoS matrix is very sparse because each user usually invokes only a handful of services. To simulate such data sparsity in our experiments, we randomly remove entries from the full data matrix and only keep a small density of historical QoS values. Data density = 10%, for example, indicates that each user invokes 10% of the services, or each service is invoked by 10% of the users. We leverage the preserved data entries for QoS prediction, and then use the removed QoS values as testing data for accuracy evaluation.

To quantize the accuracy of QoS prediction, we employ a standard error metric, MAE (Mean Absolute Error), which has been widely used in the existing work (*e.g.*, [79, 164]).:

$$MAE = \sum\nolimits_{I_{us}=0} \left|\hat{R}_{us} - R_{us}\right| / N \ , \tag{5.22}$$

where R_{us} and $\hat{R}_{us}$ denote the observed QoS value and the corresponding predicted QoS value of the invocation between user u and service s. N is the total number of testing samples to be predicted, *i.e.*, entries with $I_{us} = 0$. A smaller MAE value indicates better prediction accuracy.



(a) $\alpha = 0$ (b) $\alpha = 0.5$ (c) $\alpha = 1$

Figure 5.2: Obfuscated QoS (r'_{us}) v.s. True QoS (R_{us})

5.4.2 Effect of Data Obfuscation

The aim of data obfuscation is to perturb the QoS data such that user privacy regarding the true QoS values can be preserved when performing collaborative analysis on the server. To understand the effect of data obfuscation made on QoS data (*RQ1*), we compare the obfuscated QoS (r'_{us}) against the corresponding true QoS data (R_{us}). As an example, we randomly select a user from our dataset and provide three scatter plots by using the response time data of this user. The plots present the relationships between r'_{us} and R_{us} under different α settings. α is a parameter to determine the range of noises ϵ_{us} used to obfuscate the data. Especially, when $\alpha = 0$, the data obfuscation reduces to a z-score normalization process. Thus, Figure 5.2(a) shows linear dependence between r'_{us} and R_{us}. Z-score normalization is able to provide basic protection for user data where the mean and variance properties of QoS data are eliminated. The data after z-score normalization have a zero mean and unit variance. As α increases, the obfuscated data become more and more disordered, As shown in 5.2(a) and (b), the linear correlation between r'_{us} and R_{us} is further eliminated. Consequently, a larger α indicates better protection for user data. Note that we have similar observations on the throughput data and thus omit the details here.

5.4.3 Prediction Accuracy

Data obfuscation is useful to perturb the QoS data for preserving user privacy, but it makes no sense without providing accurate prediction results. We evaluate the accuracy of our privacy-preserving QoS prediction approaches (P-UIPCC and P-PMF) based on the obfuscated QoS data, and compare them against the following baselines and counterpart approaches (*RQ2*). We emphasize that these existing approaches require users' true QoS data and do not consider privacy issues.

- **UMEAN** [161]: This is a baseline approach that employs the average QoS value observed by a user (*i.e.*, the row mean of R) to predict the unknown QoS of this user invoking other unused Web services.
- **IMEAN** [161]: Likewise, this baseline approach employs the observed average QoS value of a Web service (*i.e.*, the column mean of R) to predict the unknown QoS of other users invoking this Web service.
- **UIPCC** [161, 162]: This is a hybrid approach that combines both user-based CF approach (UPCC) and item-based CF approach (IPCC) to make full use of the historical information from similar users and services for QoS prediction. UIPCC typically performs better than either UPCC or IPCC.
- **PMF** [164]: This is a widely-used implementation of the matrix factorization model [109], which have been introduced to QoS prediction in [164].

For fair comparisons, we use the original parameters for the counterpart approaches, as specified in the related work, because we experiment on the same dataset. To make it consistent with these settings, most parameters of our approaches are set the same with them (*e.g.*, $k = 10$ for top-k neighbours in UIPCC and P-UIPCC).

Table 5.2: Parameter Settings

Approach	RT			TP		
UIPCC	k : 10	λ : 0.1	–	k : 10	λ : 0.9	–
P-UIPCC	k : 10	λ : 0.9	α : 0.5	k : 10	λ : 0.9	α : 0.5
PMF	d : 10	γ : 40	–	d : 10	γ : 800	–
P-PMF	d : 10	γ : 12	α : 0.5	d : 10	γ : 12	α : 0.5

However, since both P-UIPCC and P-PMF work on obfuscated (normalized) data, we set different λ and γ values. The detailed parameters are specified in Table 5.2. We use $\alpha = 0.5$ in this experiment and study the effect of α in Section 5.4.4. Additionally, we vary the data density from 10% to 30% at a step increase of 5%. Each approach is performed 20 times under each data density (with different random seeds), and the average MAE results are reported.

Table 5.3 provides the results of prediction accuracy with comparisons among different approaches. The results show that, while both of our approaches preserve decent privacy by data obfuscation ($\alpha = 0.5$), they still perform much better than the baselines including UMEAN and IMEAN, and achieve comparable accuracy with the counterpart approaches including UIPCC and PMF. In particular, P-UIPCC sometimes performs better than UIPCC (*e.g.*, 0.569 vs 0.582), which can be attributed to the use of z-score normalization. Moreover, we observe that even working on obfuscated data, P-PMF mostly performs better than UIPCC. These encouraging results indicate the effectiveness of privacy-preserving approaches. In addition, we can see that the accuracy of these QoS prediction approaches improves with the increase in data density.

5.4.4 Tradeoff between Accuracy and Privacy

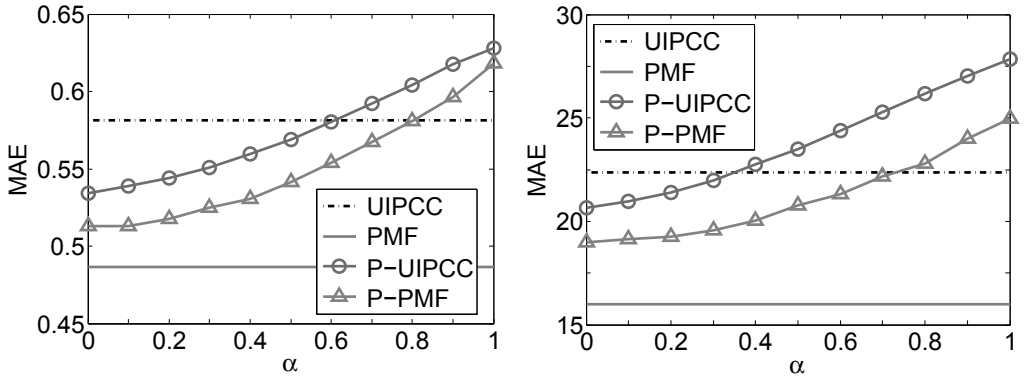
Whereas the goal of our work is to achieve both accuracy and privacy, there is indeed a tradeoff between them. At one extreme, users can provide true QoS data to obtain the most accurate QoS prediction results yet they lose privacy. At another extreme, users

Table 5.3: Prediction Accuracy w.r.t. MAE

QoS	Approach	Data Density				
		10%	15%	20%	25%	30%
RT	UMEAN	0.875	0.875	0.875	0.875	0.875
	IMEAN	0.688	0.683	0.681	0.680	0.679
	UIPCC	0.582	0.501	0.450	0.427	0.411
	PMF	0.487	0.452	0.431	0.418	0.409
	P-UIPCC	0.569	0.537	0.512	0.495	0.482
	P-PMF	0.540	0.504	0.478	0.458	0.443
TP	UMEAN	53.835	53.816	53.801	53.804	53.799
	IMEAN	26.860	26.716	26.641	26.593	26.571
	UIPCC	22.370	20.219	18.928	17.891	17.080
	PMF	15.994	14.670	13.924	13.405	13.117
	P-UIPCC	23.572	21.324	19.754	18.681	17.953
	P-PMF	20.702	18.451	17.351	16.634	16.063

can submit totally false QoS data to preserve privacy but bad prediction results will be returned. To study such tradeoff between accuracy and privacy (*RQ3*), we consider the effect of noise range α on prediction accuracy, because a larger α indicates better protection of privacy. Specifically, in this experiment, we set data density = 10% and vary α from 0 to 1 at a step increase of 0.1. Accordingly, we obtain the prediction accuracy under each α value.

Figure 5.3 presents the experimental results corresponding to response time and throughput, respectively. We can observe that both P-UIPCC and P-PMF degrade in accuracy (*i.e.*, MAE increases) when α becomes larger, because the utility of data is less preserved. However, when α is small, *e.g.*, less than 0.6 in Figure 5.3(a), our privacy-preserving approaches are more accurate than UIPCC. Even α is as large as 1, which is the variance of data after z-score normalization, the prediction accuracy is much better than the baselines (UMEAN and IMEAN). As a result, a balance needs to be made between the accuracy and privacy that a user wants to achieve. Additionally, we find that PMF and P-PMF consistently outperform UIPCC and P-UIPCC. This suggests the superior effectiveness of



(a) Response Time

(b) Throughput

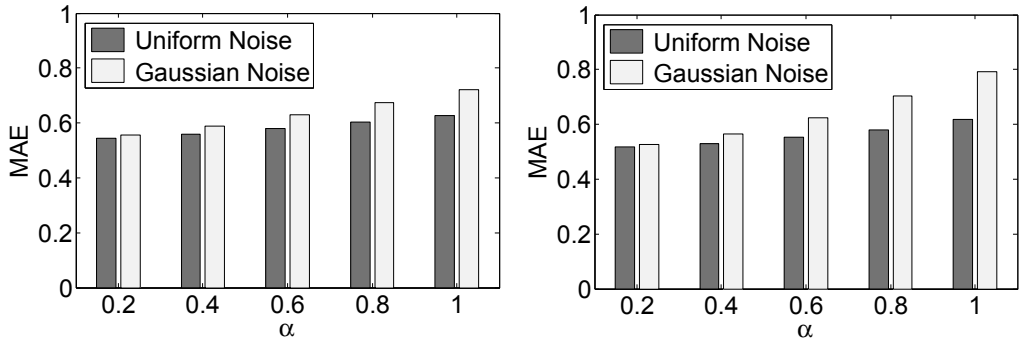
Figure 5.3: Tradeoff between Accuracy and Privacy

model-based approaches in capturing the latent structure of the QoS data, which conforms to the results reported in [164].

5.4.5 Effect of Distribution of Random Noises

In addition to the impact of noise range, a data randomization scheme is also subject to the choice of the distribution of random noises that are used for data obfuscation. In all of the above experiments, the random noises are generated from a uniform distribution located in $[-\alpha, \alpha]$. In contrast, in this experiment, we consider a Gaussian distribution $\mathcal{N}(0, \alpha)$ with a mean of zero and a standard deviation of α. Compared to a uniform distribution, random noises generated from a Gaussian distribution are unevenly distributed. To investigate the effect of distribution of random noises (*RQ4*), we vary the α value and compare the prediction accuracy of P-UIPCC and P-PMF with different settings on the distribution of random noises.

Figure 5.4 presents the results of the accuracy comparison. We can observe that, for both P-UIPCC and P-PMF, the randomization scheme with uniform noises performs better than the scheme with Gaussian noises. In particular, the performance differs significantly



(a) P-UIPCC (b) P-PMF

Figure 5.4: Effect of Distribution of Random Noises

between the two randomization schemes under a large α setting. The results imply that the distribution of random noises is a crucial factor for determining the performance of our privacy-preserving approaches.

5.5 Summary

Privacy is a practical issue to be addressed for QoS-based Web service recommendation. This work makes an initial effort to deal with the privacy-preserving Web service recommendation problem. We propose a generic privacy-preserving framework with the use of data obfuscation techniques, under which users can gain greater control on their data and rely less on the recommender system for privacy protection. We further develop two privacy-preserving QoS prediction approaches based on this framework, namely P-UIPCC and P-PMF, as representatives of neighbourhood-based CF approaches and model-based CF approaches respectively. To evaluate the effectiveness of P-UIPCC and P-PMF, we conduct experiments on a publicly-available QoS dataset of real-world Web services. The experimental results show that our privacy-preserving QoS prediction approaches can still descent prediction accuracy compared with

the counterpart approaches. We hope that the encouraging results achieved in this initial work can inspire more research efforts on privacy-preserving Web service recommendation.

□ **End of chapter.**

Chapter 6

Learning to Log for Runtime Service Monitoring

Logging is a common programming practice in software development, typically issued by inserting logging statements (*e.g.*, *printf*(), *Console.Writeline*()) in source code. As in-house debugging tools (*e.g.*, debugger), all too often, are inapplicable in production settings, logging has become a principal way to record the key runtime information (*e.g.*, states, events) of software systems into logs for postmortem analysis. To facilitate such log analysis, the underlying logging that directly determines the quality of collected logs is a matter of vital importance.

Due to the criticality of logging, it would be bad to log too little, which may miss the runtime information necessary for postmortem analysis. For example, systems may fail in the field without any evidence from logs, thus significantly increasing the difficulty in failure diagnosis [139]. However, it is also not the case that the more logging, the better. As the practical experiences reported in [6, 16], logging too much can yield many problems too. First, logging means more code, which takes time to write and maintain. Furthermore, logging consumes additional system resources (*e.g.*, CPU and I/O) and can have noticeable performance impact on system operation, for example, when writing thousands of lines to a log file per second [6]. Most importantly, excessive logging can

produce numerous trivial and useless logs that eventually mask the truly important information, thus making it difficult to locate the real issue [16]. As a result, strategic logging placement is desired to record runtime information of interest yet not causing unintended consequences.

To achieve so, developers need to make informed logging decisions. However, in our previous developer survey [59], we found that even in a leading software company like Microsoft, it is difficult to find rigorous (i.e., thorough and complete) specifications for developers to guide their logging behaviours. Although we found a number of online blog posts (*e.g.*, [1, 2, 3, 6, 12, 14]) sharing best logging practices of developers with deep domain expertise, they are usually high-level and application-specific guidelines. Even with logging frameworks (*e.g.*, Microsoft's ULS [15] and Apache's log4net) provided, developers still need to make their own decisions on where to log and what to log, which in most cases depend on their own domain knowledge. Therefore, logging has become an important yet tough decision during development, especially for new developers without much domain expertise.

Current research has seldom focused on studying how to help developers make such logging decisions. To bridge this gap, we propose a "learning to log" framework [171], which aims to automatically learn the common logging "rules" (*e.g.*, where to log, what to log) from existing logging instances, and further leverage them to provide informative guidance for new development. It is straightforward to build a rule set that can guide developers on logging, but such a rule set usually requires a large effort to produce and maintain, which may finally kill its usefulness. Motivated by our observations (detailed in Section 6.1.2), we extract a set of contextual features from the source code to construct a learning model for predicting where to log. Our logging suggestion tool built on this model, named *LogAdvisor*, can thus provide actionable suggestions for developers and reduce their effort on logging. As an

initial step towards "learning to log", this work focuses on studying where to log (or more specifically, whether to log a focused code snippet), while leaving other aspects (such as what to log) of this research for future work.

We have conducted both within-project evaluation and cross-project evaluation on *LogAdvisor* using two industrial software systems from Microsoft and two open-source software systems from GitHub. Additionally, a user study is performed to evaluate whether the suggestions provided by *LogAdvisor* can help developers in practice. The comprehensive evaluation results have demonstrated the feasibility and effectiveness of our logging suggestion tool.

The remainder of this chapter is organized as follows. Section 6.1 introduces our studied software systems and the motivation of this work. Section 6.2 provides the overview and the detailed techniques of learning to log. Section 6.3 reports on the evaluation results, and Section 6.4 presents our user study. We discuss the limitations in Section 6.5, and finally conclude this chapter in Section 6.6.

6.1 Problem and Motivation

In this section, we first introduce the subject software systems under study. Then we provide some key observations on logging practices and present the motivation of our study.

6.1.1 Subject Software Systems

In our study, we investigate four large software systems, including two industrial systems from Microsoft (denoted as System-A and System-B for confidentiality) and two open-source systems from GitHub (SharpDevelop and MonoDevelop). Each of these systems contains millions of lines of code (LOC) written in C# language. C# is one of the popular programming languages that supports the exception-handling mechanism, where errors are commonly

Table 6.1: Summary of the Studied Software Systems

Software Systems	Time	Description	Version	LOC
System-A	–	Online service	–	2.5M
System-B	–	Online service	–	12.7M
SharpDevelop	2001	.NET platform IDE	5.0.2	1.4M
MonoDevelop	2003	Cross-platform IDE	4.3.3	2.5M
Total				19.1M

Software Systems	#Logging		#Commits		
	Logging Statements	LOC of Logging	Total	#Commits with Logging	#Patches with Logging
System-A	23,624	77,945	–	–	–
System-B	69,057	240,395	–	–	–
SharpDevelop	2,896	9,261	13,886	4,593 (33.1%)	724 (15.8%)
MonoDevelop	4,996	13,043	29,357	9,437 (32.1%)	1,157 (12.3%)
Total	100.6K	327.6K	43.2K	14.0K (32.4%)	1.9K (13.6%)

handled by using exception constructs (*e.g.*, try-catch, throw) instead of returning error codes. Note that the source code of all dependent external submodules is also included for our study.

Table 6.1 provides the summary information of our studied software systems. Both industrial systems are online service systems developed by Microsoft, serving a huge number of users globally. These two systems were also used as subjects in our empirical study on logging practices [59]. To allow for reproducing and applying our approach to future research, we choose another two open-source software systems as subjects. They are two IDE projects, with SharpDevelop working on .NET platform and MonoDevelop allowing for cross-platform development. Both of them are selected due to their popularity (well-known C# projects), active updates (10000+ commits) and long history of development (10+ years).

Our targeted systems are supposed to have reasonably good logging implementation, because the produced logs by these systems have mostly met the requirements of usage analysis, troubleshooting, and operating, after undergoing more than 10 years of evolution. This is especially true for the industrial software systems, because

Table 6.2: Logging Statistics of Different Software Entities

Software Systems	Source Files		Classes	
	Total	#Logged	Total	#Logged
System-A	4,706	2,027 (43.1%)	10,349	2,788 (26.9%)
System-B	50,036	9,380 (18.7%)	62,954	10,098 (16.0%)
SharpDevelop	8,853	666 (7.5%)	10,869	704 (6.5%)
MonoDevelop	11,567	999 (8.6%)	18,724	1,177 (6.3%)
Total	75.2K	13.1K (17.4%)	102.9K	14.8K (14.4%)
Software Systems	Methods		Catch Blocks	
	Total	#Logged	Total	#Logged
System-A	57,578	9,128 (15.9%)	7,580	3,320 (43.8%)
System-B	324,167	30,988 (9.6%)	25,441	5,307 (20.9%)
SharpDevelop	69,108	1,618 (2.3%)	1,346	252 (18.7%)
MonoDevelop	121,982	2,390 (2.0%)	4,041	771 (19.1%)
Total	572.8K	44.1K (7.7%)	38.4K	9.7K (25.3%)

each of them is implemented by a group of experienced developers at Microsoft, where the code quality has been strictly controlled by a circle of development activities including code design, code implementation, code refactoring, peer review, various testing, etc. Consequently, the source code of these software systems is well suited for our study on logging practices. All of our code analysis is conducted based on an open-source C# code analysis tool, *Roslyn* [13]. By using Roslyn, we can perform both syntax analysis and semantic analysis on the source code.

6.1.2 Observations

Pervasiveness of logging

Logging is pervasively used in software development. As we can see in Table 6.1, our studied systems have a total of 100.6K logging statements (containing 327.6K lines of logging code) out of 19.1M LOC. That is, there is a line of logging code in every 58 LOC, as similarly reported in [135, 140]. By drilling down according to the type of software entities, as shown in Table 6.2, we find that about

17.4% of the source files, 14.4% of the classes, 7.7% of the methods, and 25.3% of the catch blocks are logged, respectively. In addition, by examining the revision histories of the systems, we find that, on average, 32.4% of the commits involve logging modifications, and further, 13.6% of them are modified along with patches[1]. Both its pervasive existence and active modifications reveal that logging plays an indispensable role in software development and maintenance.

Where to log

The logging decisions can resolve to where to log and what to log. *Where to log* determines the locations to place logging statements, while *what to log* denotes the contents recorded by these logging statements. Whereas the goal of "learning to log" is to handle them both, we study where to log in this work. Our previous empirical study on where developers log [59] has shown that there are some typical categories of logging strategies for recording error sites and execution paths. Error sites indicate some unexpected situations where the system potentially runs into a problem, including exceptions and function-return errors. As two typical ways for error reporting, exception mechanisms are widely used in modern programming languages (*e.g.*, C#) to handle abnormal situations, and function-return errors indicate the situation where an unexpected value (*e.g.*, -1/null/false/empty) is returned from a function call. We denote their associated code snippets as exception snippets and return-value-check snippets respectively (as examples shown in Fig. 6.1(a)(b)). They are the two most common logging strategies [59] and thus become our *focused code snippets*. Although recording information of execution path is crucial for tracking down root causes from the error sites, existing studies (*e.g.*, control-flow instrumentation [49, 97]) have been conducted to achieve this goal,

[1]We identify patches by searching commit logs for keywords such as "fix", "bug", "crash" or issue ID like "#42233", the same as in [71].

```
/* A code example taken from MonoDevelop (v.4.3.3), at file:
 * main\external\mono-tools\gendarme\console\Settings.cs,
 * line: 116. Some lines are omitted for ease of presentation.
 */
private int LoadRulesFromAssembly (string assembly, ...)
{
    Assembly a = null;
    try {
        AssemblyName aname = AssemblyName.GetAssemblyName(
            Path.GetFullPath (assembly));
        a = Assembly.Load (aname);
    }
    catch (FileNotFoundException) {
        Console.Error.WriteLine ("Could not load rules
            from assembly '{0}'.", assembly);
        return 0;
    }
    ...
}
```

(a) Exception Snippet

```
/* A code example taken from MonoDevelop (v.4.3.3), at file:
 * main\src\core\MonoDevelop.Ide\MonoDevelop.Ide\ImageService.cs,
 * line: 302.
 */
// Converts an image spec into a real stock icon id
string stockid = GetStockIdForImageSpec (name, size);
if (string.IsNullOrEmpty (stockid)) {
    LoggingService.LogWarning ("Can't get stock id for " +  name
        + " : " + Environment.NewLine + Environment.StackTrace);
    return CreateColorBlock ("#FF0000", size);
}
```

(b) Return-value-check Snippet

Structural features:
Exception Type: 0.39 (System.IO.FileNotFoundException)
Containing method: Gendarme.Settings.LoadRulesFromAssembly
Invoked methods: System.IO.Path.GetFullPath,
System.Reflection.AssemblyName.GetAssemblyName,
System.Reflection.Assembly.Load

Textual features:
Exception type/methods: gendarme(1), settings(1), load(2),
rules(1), from(1), assembly(4), system(3), io(1), path(1),
get(2), full, path(1), reflection(2), name(2), file(1), not(1),
found(1), exception(1)
Vaiables: aname(2), assembly(1), a(1) **Comments**: N/A

Syntactic features:

SetLogicFlag: 0	EmptyCatchBlock: 0	
Throw: 0	OtherOperation: 0	
Return: 1	LOC: 3	
RecoverFlag: 0	NumOfMethods: 3	**Label:** Logged

(c) Extracted Contextual Features from Exception Snippet in (a)

Figure 6.1: Code Examples and Contextual Features

which are orthogonal to our work.

```
/* Example 1: An exception used to determine logic branch*/
void AccountConfig(MONOAccount user, string propertyName) {
    ...
    bool userHasRights = true;
    try {
        user.DeleteAccountProperty(propertyName);
    }
    catch (UnauthorizedAccessException) {
        userHasRights = false;
    }
    if (userHasRights) {
        ...
    }
}
```

```
/* Example 2: An exception re-thrown */
try {
    Type t = Type.GetTypeFromID(guid);
    object instance = Activator.CreateInstance(t);
}
catch (Exception e) {
    throw new TestFailedException("Fail to create Com interface.\t: "
        + Tester.GetExceptionDetails(e));
}
```

```
/* Example 3: An exception recovered by retrying */
void DWAppOverride(...) {
   ...
   Uri UriNew = null;
   try {
      UriNew = new Uri(wApp);
   }
   catch (UriObjectFormatException) {
    // Assume http is the scheme and the URL param is the machine name
      if (UriNew == null) {
         try {
            UriNew = new Uri("http://" + wApp);
         }
      }
   }
}
```

Figure 6.2: Code Examples of NOT Logging

Why not log everything

Log information is immensely useful in maintaining software systems. So the question "why not log everything?" (*e.g.*, StackOverflow questions [7, 9]) does sound reasonable. Yuan et al. also proposed conservative logging (ErrLog) [139], which logs all the generic exceptions (*e.g.*, exceptions and function-return errors) for failure diagnosis. However, as the logging statistics shown in

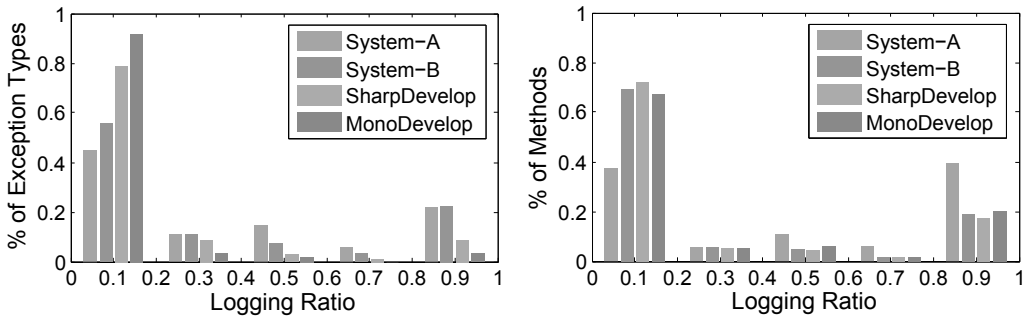
Table 6.3, we observed that, in our studied systems, the majority of exceptions (74.7% on average) and return-value-check snippets (90.7% on average) are actually not logged. To understand this fact, we posted our questions on "why not log all exceptions?" to the mail lists and websites of MonoDevelop [20], SharpDevelp [21] and StackOverflow [9], and received some valuable feedback from the developers. According to their feedback, "logging all exceptions would produce a ton of garbage and make it hard to zoom in on real issues", which conforms with our argument (not logging too much). According to our empirical study in [59], there are many situations for not logging an exception. Some examples are illustrated in Figure 6.2. Some exceptions are "expected in normal operation", while some others are satisfactorily handled or "recovered without impacting the user". In Example 1, when an "UnauthorizedAccessException" exception is caught, the program sets the flag "userHasRights" to *false* and then directs the execution to the subsequent logic branch. In Example 2, the caught exception is re-thrown to its caller as a "TestFailedException", then its caller would determine whether to log the exception at a higher level. Example 3 illustrates an example of an exception recovered by retry. When the program fails to create a *Uri* object, it uses the default scheme *http* to create the object again. In a word, not all exceptions are "unexpected" (or errors) [6]. Strategic logging needs to "determine whether or not an exception is worth reporting" [8].

Logging decision and the context

To understand this tradeoff in practice, we attempt to study how developers make decisions on whether to log a focused code snippet. Fig. 6.1(a) presents a real-world example of an exception snippet (*i.e.*, try-catch block). The operations enclosed in the try block attempt to load the rules from the input string, "*assembly*". If this assembly file cannot be found, an exception with type of "FileNotFoundException" will occur and then be caught by the

Table 6.3: Logging Statistics

Software Systems	Exception Snippets		
	#Exception Types	#Instances	#Logged Instances
System-A	188	7,580	3,320 (43.8%)
System-B	1,657	25,441	5,307 (20.9%)
SharpDevelop	106	1,346	252 (18.7%)
MonoDevelop	220	4,041	771 (19.1%)
Total		38.4K	9.7K (25.3%)
Software Systems	Return-value-check Snippets		
	#Call Types	#Instances	#Logged Instances
System-A	5,400	43,443	5,127 (13.5%)
System-B	21,624	131,870	15,081 (11.4%)
SharpDevelop	3,221	17,937	476 (2.7%)
MonoDevelop	5,821	37,360	750 (2.0%)
Total		230.6K	21.4K (9.3%)



(a) Exception Types (b) Methods

Figure 6.3: Distribution of Exception Types/Methods

catch block. Here, the exception has been logged with an error message by "*Console.Error.WriteLine*()". Intuitively, from this example, we can see that the logging decision is highly dependent on the context of this code snippet, including the *exception type* (*e.g.*, FileNotFoundException), the invoked *methods* (*e.g.*, GetFullPath, GetAssemblyName, Load) in a try block, etc. The contextual information is crucial because each exception type generally denotes one specific type of exceptional conditions while the invoked

methods indicate the functionality of operations. Driven by this intuition, we measure the logging ratio of each exception type and each method. Specifically, the logging ratio, with an exception type (or an invoked method) is measured by the number of logged exceptions divided by the number of all the exception snippets with this exception type (or containing this method). Figure 6.3 illustrates the bar plot of the distribution of exception types/methods across the corresponding logging ratios. The results show that a significant portion of exception types (82%) and methods (86%) have either high ($> 80\%$) or low ($< 20\%$) logging ratios, which suggests their high correlations (*i.e.*, either positive or negative correlations) with logging decisions of developers.

6.1.3 Motivation

With the ever growing scale and complexity of software systems, it is common that each developer is only responsible for a part of a system (*e.g.*, one or several components). Logging under this situation is notoriously challenging, because developers may not have full knowledge of the whole system. For example, in our user study (see Section 6.4), 68% of the participants have logging difficulties. However, there is a lack of rigorous specifications or tool support for developers to aid their logging decisions. Without a well-structured logging strategy, it is difficult for developers to know how to make informed logging decisions, and thus, quite often, the decisions are made based on their own domain knowledge (*e.g.*, understanding of system behaviours, logging experience). Such domain knowledge is seldom documented and it is also hard to do so, since the logging behaviours of developers may vary widely, not only from project to project, but also from developer to developer. Indeed, the pervasively-existing logging instances together can provide strong indication of the developers' domain knowledge embedded with their logging decisions. Thus, we intend

to explore whether the logging decisions of developers, such as where to log, can be learnt automatically from these existing logging instances. If so, the constructed model can represent the common knowledge of logging and be further built into tool support to provide valuable suggestions (*e.g.*, whether to log an exception snippet) for developers. Such a tool can improve the logging quality as well as reduce the effort of developers. Following this motivation, we propose "learning to log".

6.2 Learning to Log

In this section, we present the overview as well as the detailed techniques of "learning to log".

6.2.1 Approach Overview

Our goal, referred to as "learning to log", is to automatically learn the common logging practice as a machine learning model, and then leverage the model to guide developers to make logging decisions during new development. We further implement the proposed "learning to log" approach as a tool, *LogAdvisor*. Figure 6.4 presents the overview of "learning to log", which can be described as the following steps:

Instances collection

As the first step, we need to extract data instances (focused code snippets) from our target projects. There are two types of frequently-logged code snippets: exception snippets and return-value-check snippets. As shown in Figure 6.1(a) and Figure 6.1(b), exception logging records the exception context (*e.g.*, exception message) after an exception is captured in the catch block, while return-value-check logging is used to log the situation where an unexpected value (*e.g.*, -1/null/false/empty) is returned from a function call. By employing

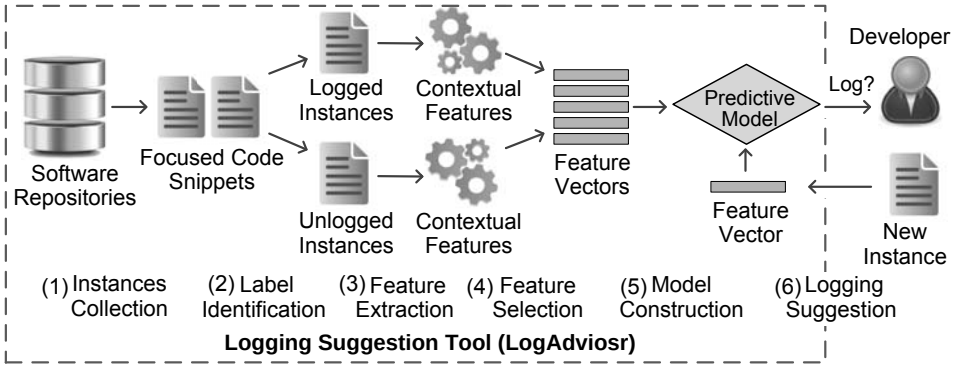


Figure 6.4: The Overview of Learning to Log

Roslyn, we extract all these focused code snippets, and use them as training data to learn the logging practices of developers.

Label identification

As a key step of preparing training data, each data instance (a code snippet) is labelled "logged" if it contains a logging statement; or "unlogged", otherwise. A logging statement denotes a statement that has an invocation to a logging method (*e.g.*, *Console.Writeline*()). We identify logging methods by searching some keywords in all method names, such as *log/logging*, *trace*, *write/writeline*, etc. The logging statement identification and labelling procedures are automatically performed based on Roslyn.

Feature extraction

In our study, we need to extract useful features (e.g., exception type) from the collected code snippets for making logging decisions, which is one of the most important steps to determine the performance of the prediction model. The details on feature extraction are described in Section 6.2.2.

Feature selection

Although features are of vital importance to learn a predictive model, they can also become an encumbrance. When there are too many features, some of them are likely redundant or irrelevant since they provide little useful information or even act as noises to degrade the prediction performance. Feature selection [61] is a key technique to remove such redundant or irrelevant features to enhance the prediction performance as well as shorten the training time.

Model training

Through feature extraction and selection, we can generate a corpus of feature vectors, where each denotes a vector of feature values from a data instance. With these feature vectors and their corresponding labels, we can apply a set of machine learning models (*e.g.*, Decision Tree [131]) to learn the common logging practice. In our study, we learn the decision on whether to log a focused code snippet as a classification model.

Logging suggestion

Through the above learning process, we can obtain a predictive model to perform accurate logging predictions. This predictive model can be trained offline and further be built into a logging support tool (namely *LogAdvisor*) to provide online logging suggestions for developers. For example, when a developer composes a new piece of code containing a *try-catch* block, *LogAdvisor* can detect and extract its feature vector in a transparent way. Then LogAdvisor can predict on whether to log, and provide a logging suggestion for the developer through IDE (*e.g.*, like the warning message). By using *LogAdvisor*, developers can make informed logging decisions.

The above learning workflow is generic and works similarly to many other machine learning applications in software engineering (*e.g.*, defect prediction [71, 93, 177]). But the following techniques

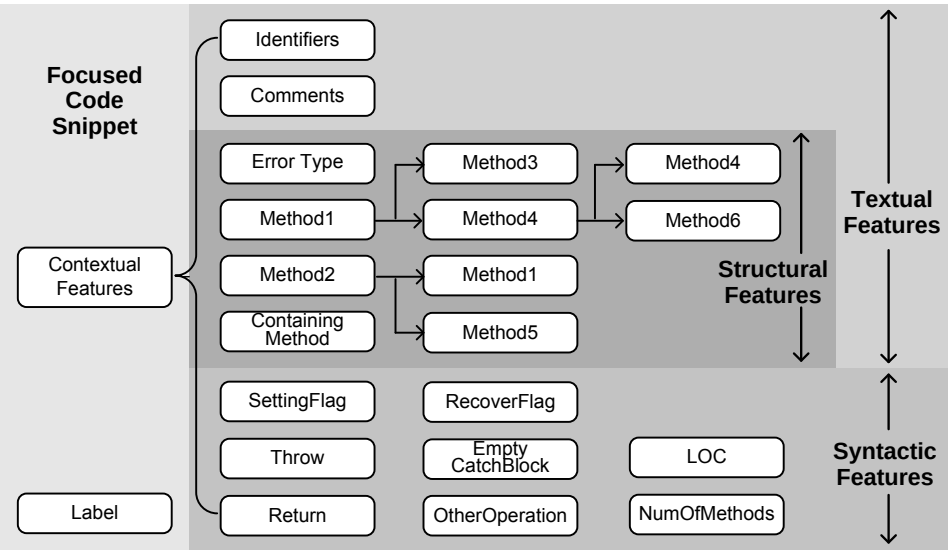


Figure 6.5: Framework of Contextual Feature Extraction

used in our approach are unique in achieving the goal of "learning to log".

6.2.2 Contextual Feature Extraction

Feature extraction lies in the core of "learning to log", because the quality of extracted features directly determines the performance of the model. The context information (*e.g.*, the functionality of operations, the impact of exceptions) of logging points are crucial for developers to make logging decisions. However, it is challenging to effectively extract such context information, because the target code snippet is usually short and linguistically sparse compared to natural language text. To address this issue, we propose a novel feature extraction framework, as illustrated in Figure 6.5, which involves three types of features: structural features, textual features, and syntactic features.

Structural features

Source code has a well-defined structure. It is desired to leverage the structure information of source code to help extract context information. To achieve this goal, we extract two types of structural features: error type and associated methods.

Error Type: The error type, such as exception type or call type, can largely reveal the context of our focused code snippets, which is highly correlated with logging decisions of developers (as indicated in Section 6.1.2). For an *exception snippet*, the exception type generally denotes one specific type of exceptional conditions with informative semantic meanings, *e.g.*, "*FileNotFoundException*" in Figure 6.1(a). For a *return-value-check snippet*, the call type is denoted as the prototype of the checked function, e.g., *string GetStockIdForImageSpec*(*string, int*) in Figure 6.1(b), which indicates one specific type of potential function-return errors. Therefore, we extract error type as a key feature. In particular, there are a number of catch blocks (approximately 12% in our study) without exception type explicitly specified by the developer. In this case, we categorize them into an "*UnspecifiedException*" type.

Each instance has a single error type, but there exist a wide variety of error types among the training data. We avoid directly using each error type as a feature dimension, which can lead to highly sparse and ineffective feature vectors. Instead, we construct only one feature dimension as the logging ratio of each error type, that is, the ratio of logged instances against all the instances within that error type. Figure 6.1(c) presents an illustration of the contextual features extracted from the code example in Figure 6.1(a). In this example, the "*FileNotFoundException*" type has a logging ratio of 39% regarding training instances in MonoDevelop, so we take the feature value of error type as 0.39.

Methods: The associated methods of a focused code snippet also provide indicative information to help understand the functionality

of the operations. For example, we can figure out the intention of developers (*i.e.*, to load an assembly file) in the example of Figure 6.1(a) according to the method names, including *LoadRulesFromAssembly*, *GetFullPath*, *GetAssemblyName*, and *Load*. Therefore, we extract these methods as important contextual features.

Specifically, there are two types of methods: the containing method and the invoked methods. The former is the method that contains the focused code snippet (*e.g.*, *LoadRulesFromAssembly* in Figure 6.1(a)), while the latter includes all the methods that are invoked by the snippet. The operations can be seen as a sequence of API method invocations. Thus, instead of using only the methods within the code snippet, we also track the callee methods. Figure 6.5 provides a prototype of our approach, where the arrows represent the invocation relationships between methods. For example, *Method1* and *Method2* are invoked by the focused code snippet, where *Method1* invokes *Method3* and *Method4*, and *Method4* further invokes itself and *Method6*. The extraction of methods continues tracking down until the invoked method is a system API or external library API method (*e.g.*, *System.IO.Path.GetFullPath*) or until a certain number of levels has been attained. The extraction process is implemented as a breadth-first search (BFS) variant, where all the recorded (visited) methods will be skipped. In particular, all the logging methods are excluded in this process.

Algorithm 5 provides the description of our methods extraction process in detail. Specifically, line 1$\sim$3 describe that the containing method of $focusedSnippet$ is identified and its method name is added to $methodList$. In line 4, all of the syntax nodes with method declaration type are obtained and stored into a list, $allMethodDeclarations$. We put the syntax node $focusedSnippet$ (with its level number) into a queue in line 5$\sim$6. Then line 7$\sim$20 describe a while loop to extract all of the invoked methods. In line 8$\sim$10, a $methodNode$ is taken out from the queue, and then extract the directly-invoked methods. Especially, in line 10, any other

focused snippet in $methodNode$ will be skipped during extraction, and therefore, the methods enclosed in other focused snippets will be ignored for the input $focusedSnippet$. In line 11$\sim$20, we examine each of the invoked methods, and check whether a method has already been recorded or it is a system method or external API method (*i.e.*, not contained in $allMethodDeclarations$). If yes, we will skip to next iteration; otherwise, we put the syntax node of its caller method into the queue for the subsequent extraction. In particular, we set a parameter $maxLevel$, which can control the maximal levels we trace back to extract the invoked methods. For example, in Figure 6.5, the levels of *Method1*, *Method3*, *Method6* are 0, 1, and 2, respectively. In our experiment, $maxLevel$ is set to 5 by default, which has shown good results in performance evaluation. At last, a list of methods ($methodList$) related to the context of $focusedSnippet$ is returned in line 21.

After extracting the list of associated methods, we obtain the full qualified name (*e.g.*, System.IO.Path.GetFullPath) of each method as a feature dimension, which contains namespace, class name, and its (short) method name. Figure 6.1(c) provides an example for these features.

Textual features

Source code is also text. Using code as flat text has been widely employed in the field of mining software repositories, and its effectiveness are demonstrated and reported in tasks such as API mining [155], code example retrieval [29], etc. Driven by these encouraging results, we also employ the similar approach to extract textual features from source code text.

More specifically, we extract all the texts in the focused code snippet excluding method names, such as variables and types. Then we combine them with the extracted list of structural features (*i.e.*, error type and methods) as the full text. In contrast to extracting all the text directly, our approach not only excludes the text of logging

Algorithm 5: Methods Extraction Algorithm

```
/* All the data structures are used as with Roslyn API
   and system API in C#.                                  */
```

Input: An abstract syntax tree: **SyntaxTree** *syntaxTree*, a syntax node of the focused snippet (exception snippet or return-value-check snippet): **SyntaxNode** *focusedSnippet*, the maximal levels to trace back: **int** *maxLevel*

Output: A list of methods: *methodList*

```
1  List<String> methodList ← null;
2  Identify the containing method of focusedSnippet;
3  Get fullMethodName of the containing method, and add it into methodList;
   /* fullMethodName is comprised of its namespace, class
   name, and method name */
4  Traverse syntaxTree to get allMethodDeclarations;
5  Queue<Tuple<SyntaxNode, int> > methodQueue;
6  Add (focusedSnippet, 0) into methodQueue;      /* trace back level=0 */
7  while methodQueue is not empty do
8      Take an element (methodNode, level) out of methodQueue;
9      List<SyntaxNode> invocationList ← null;
10     Add all the methods invoked by methodNode to invocationList;    /* Skip
       the methods enclosed in any other focused snippet */
11     foreach invokedMethod in invocationList do
12         Get fullMethodName of invokedMethod;
13         if methodList does not contain fullMethodName then    /* Visit the
           method that has not been recorded in methodList */
14             Add fullMethodName into methodList;
15             if level > maxLevel then        /* Guarantee the maximal
               trace back level to maxLevel */
16                 continue;
17             if fullMethodName.StartWith("System") then          /* Skip
               system methods */
18                 continue;
19             if allMethodDeclarations contains fullMethodName then    /* Add
               invokedMethod into the queue if it is
               user-defined */
20                 Add (invokedMethod, level + 1) into methodQueue;
21 return methodList;
```

methods, but also includes the names of the callee methods, the

containing method, as well as their namespaces and classes. With such text, we can extract the textual features using the bag-of-words model through a set of widely-used text processing operations: **1) Tokenization**. In our case, we exploit the common use of camel case and special characters (*e.g.*, "_") in naming to split all the text into terms. This tokenization approach is simple yet effective, which works well in our studied projects as well as some other related work [29, 107]. Then we convert all the uppercase characters into lowercase ones. **2) Stemming**. This step is to identify the ground form of each term, where the affixes and other lexical components are removed. For example, "methods" will be stemmed into "method". **3) Stop words removal**. To get rid of some useless and noisy terms, we use a list of stop words to filter them, where terms like "the", "that" are discarded. Besides, we remove all the terms with a length smaller than 3, such as "i", "on", "is", etc. **4) Term weighting**. We use the standard TF-IDF [131] weighting scheme to assign values to each term. Since the use of these techniques in code processing has been carefully reported in [29, 32, 155], we omit the details here. In our study, these processing steps are performed using Weka [62].

Syntactic features

As indicated in Section 6.1.2, there are many situations of not logging, even for typical error sites such as exceptions and function-return errors. Some potential errors have no critical impact on the normal operation of the whole system, some are resolved by recovery actions such as retry or walk-around, and some others are explicitly reported (*e.g.*, by setting flags, re-throwing, or returning special values) to the subsequent or upper-level operations (*e.g.*, caller method) to handle.

To capture these contextual factors, we also extract some key syntactic features from each focused code snippet: 1) *SettingFlag*. We identify whether there is an assignment statement with an

assigned value like -1/null/false/empty. 2) *Throw*. We identify whether there is a *throw* statement. 3) *Return*. We identify whether any special value (*e.g.*, -1/null/false/empty) is returned. 4) *RecoverFlag*. We check whether there is a new *try* statement inside. 5) *OtherOperation*. We check whether there is any other operations included except the above five ones. 6) *EmptyBlock*. We find that the developers sometimes catch and then do nothing. We thus identify whether the catch block is empty. Note that all these identification processes have excluded logging statements at the first place, and all these features have *Boolean* values. In addition, we employ the feature *LOC* to measure the lines of code in the code snippet, and the feature *NumOfMethods* to measure the number of the extracted methods. An example is shown in Figure 6.1(c).

6.2.3 Feature Selection

The above feature extraction process, however, can generate tens of thousands of features, due to the large vocabulary of methods and (textual) terms extracted from the data instances. These features further lead to high-dimensional (*e.g.*, 72K features in System-B) yet highly-sparse feature vectors, because most of the features are actually infrequent across all data instances. Furthermore, some of these features (*e.g.*, textual features parsed from some specific variable names) may be irrelevant and have negative impact on the performance of the predictive model.

In such a setting, we make use of a two-step feature selection process to remove irrelevant features and reduce the dimensionality of feature vectors. First, we institute a threshold that constraints the minimum frequency of a feature that occurs across all data instances. We set the threshold to 5 in our experiments and thus eliminate a significant number (*e.g.*, 68% in System-B) of infrequent features. Second, we employ a well-known approach, *information gain* [23], to perform further feature selection. Information gain is widely-

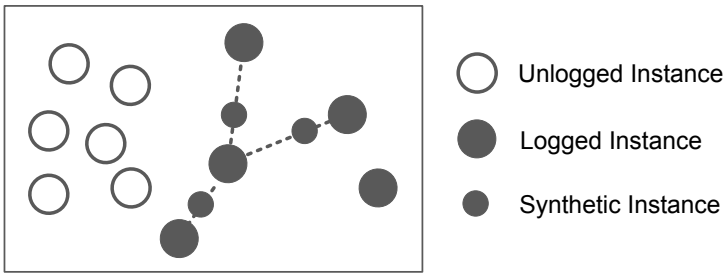


Figure 6.6: Illustration of Imbalance Handling

used and effective in text categorization [23]. We carefully set the minimum information gain to filter out many irrelevant features and reduce the feature dimensionality to around 1000. This two-step feature selection process works well in our experiments, so we do not further evaluate the impact of thresholds on the results.

6.2.4 Imbalance Handling

One critical challenge faced by our model is the high imbalance of data. For our studied systems, only 25.3% of exception snippets and 9.3% of return-value-check snippets are logged. This reveals an imbalance ratio up to $48.8:1$ between unlogged (majority) instances and logged (minority) instances. Data imbalance is a common issue in real-world machine learning applications, and as we will show in Section 6.3.4, it can heavily influence the prediction performance. In our study, we employ a state-of-the-art approach, SMO-TE [42], to balance the data by creating synthetic instances from the minority class. SMOTE first identifies the k-nearest minority neighbours (measured by the cosine similarity between their feature vectors) for each examined minority instance, and then randomly generates synthetic instances between the instance and its neighbour. As illustrated in Figure 6.6, three synthetic instances are generated between the examined instance and its 3-nearest neighbours. The value k and the number of synthetic instances to generate are set as user input. In our experiments, we employ the Weka [62]

implementation of SMOTE.

6.2.5 Noise Handling

Another challenge lies in the data noises. In the framework of "learning to log", we implicitly assume good logging quality in the training data, which therefore facilitates the automatic learning of good logging "rules" for new development. However, there is no guarantee about the quality of logging in reality, due to the lack of "ground truth" on what is optimal logging. Considering the active maintenance and the long history of evolution of our studied software systems, it is still reasonable to assume that "most" of the data instances are enclosed with good logging decisions, while only a small portion of them may reveal incorrect decisions, which we refer to as *data noises*. For example, some instances that deserve logging are actually not logged, while some others without the need of logging are logged. These data noises thus have flipped logging labels.

We attempt to detect and eliminate such data noises, and help the model learn the common knowledge of logging more effectively. In many real-world applications, perfect data labels are impossible (or difficult) to obtain [57]. Kim et al. have proposed a simple and effective noise detection approach (namely CLNI) for defect prediction [71]. We adapt this approach to deal with our specific case, and find that it works well (as demonstrated in Section 6.3.5).

Traditionally, CLNI identifies the k-nearest neighbours for each instance and examines the labels of its neighbours. If a certain number of neighbours have an opposite label, the examined instance will be flagged as a noise. However, we observe a high imbalance ratio, for example up to $48.8:1$ in MonoDevelop, between unlogged (majority) instances and logged (minority) instances. Therefore, the majority instances tend to dominate the neighbourhood of an examined instance, which makes the identification of k-nearest neigh-

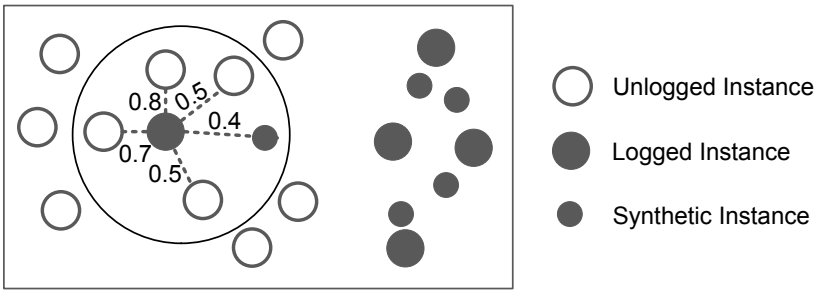


Figure 6.7: Illustration of Noise Handling

bours in CLNI biased to the majority class. To handle this issue, we apply a state-of-the-art imbalance handling approach, SMOTE [42]. SMOTE balances the data instances by creating synthetic logged instances as shown in Figure 6.7. Consequently, both classes have an equal number of data instances, which eliminates the inherent bias to the majority class when we identify the k-nearest neighbours of an instance. Next, we quantify each examined instance i with a noise degree value: $\varphi_i = \sum_{j \in S_i} w_{ij}$, where S_i denotes the set of neighbours with opposite label with i, and w_{ij} is the weight to characterize the different impacts of different neighbours in S_i. In contrast to CLNI that uses $w_{ij} = 1$, we take w_{ij} as the cosine similarity between features of i and j. This is based on the intuition that instances with higher similarity between each other are more likely to share the same label. Therefore, the greater the value φ_i is, the higher probability the examined instance i is a noise. For example in Figure 6.7, $\varphi_i = 2.5$. We flag the instances with top ranked φ_i values and remove them as noises, while leveraging the remaining data for model training.

6.3 Evaluation

In this section, we conduct comprehensive experiments to evaluate the effectiveness of *LogAdvisor*. In particular, we intend to answer the following research questions.

***RQ1*:** What is the accuracy of *LogAdvisor*?

***RQ2*:** What is the effect of different learning models?

***RQ3*:** What is the effect of imbalance handling?

***RQ4*:** What is the effect of noise handling?

***RQ5*:** How does *LogAdvisor* perform on the "golden set"?

***RQ6*:** How does *LogAdvisor* perform in the cross-project learning scenario?

For ease of reproducing and applying our approach to future research, we release our source code and detailed study materials (*e.g.*, data, questionnaire) on our project page[2].

6.3.1 Experimental Setup

After obtaining the feature vectors and their corresponding logging labels, we employ Weka [62] to perform model training and evaluation. Due to the imbalanced nature of our data, we apply the Weka implementation of SMOTE [42] to balance the training data for model construction. By default, we use decision tree (J48) as the learning model, because of its good performance (Section 6.3.3) and ease of interpretation. Except for the cross-project evaluation (Section 6.3.7), all of the experiments are evaluated on all of the extracted data instances by using the 10-fold cross evaluation mechanism [131].

As recommended in other related work [46, 149], we evaluate *LogAdvisor* using *balanced accuracy* (BA) [35], which is the average of the proportion of logged instances and the proportion of unlogged instances that are correctly classified. BA is calculated as follows:

$$BA = \frac{1}{2} \times \frac{TP}{TP + FN} + \frac{1}{2} \times \frac{TN}{TN + FP} \ , \qquad (6.1)$$

[2] http://cuhk-cse.github.io/LogAdvisor

Table 6.4: Balanced Accuracy of Different Approaches

Approaches	Exception Snippets			
	System-A	System-B	SharpDev	MonoDev
Random	0.499	0.500	0.496	0.503
ErrLog	0.500	0.500	0.500	0.500
Error Type	0.719	0.637	0.724	0.797
Methods	0.672	0.690	0.603	0.678
Textual Features	0.768	0.712	0.719	0.797
Syntactic Features	0.884	0.858	0.779	0.829
LogAdvisor	**0.934**	**0.927**	**0.846**	**0.932**

Approaches	Return-value-check Snippets			
	System-A	System-B	SharpDev	MonoDev
Random	0.500	0.494	0.505	0.503
ErrLog	0.500	0.500	0.500	0.500
Error Type	0.743	0.748	0.829	0.813
Methods	0.689	0.699	0.772	0.769
Textual Features	0.814	0.768	0.781	0.808
Syntactic Features	0.762	0.764	0.794	0.758
LogAdvisor	**0.903**	**0.927**	**0.865**	**0.918**

where TP, FP, TN, and FN denote true positives, false positives, true negatives, and false negatives, respectively. BA weights the performance on each of the two classes equally, thus avoiding inflated performance evaluation on imbalanced data. For example, with an imbalance ratio of $48.8 : 1$ in MonoDevelop, a trivial classifier that always predict "not logging (unlogged)" can achieve 98% accuracy, but would result in a low balanced accuracy of 49%. It is also worth noting that although precision, recall, and f-measure are widely-used metrics for evluating classification performance, we mainly use these results as references. This is because it is not easy to distinguish the positive class and the negative class while both logged and unlogged instances are equally important. This would lead to ambiguous semantic meanings of these metrics in our senarios.

6.3.2 Prediction Accuracy

We compare *LogAdvisor* with two baseline approaches: random and ErrLog [139]. By random, we mimic the situation where a developer has no knowledge about logging and perform the logging decision with a random probability of 0.5. ErrLog is proposed in [139] that makes conservative logging (*i.e.*, log all the generic exceptions such as exceptions and function-return errors) for failure diagnosis. The results are provided in Table 6.4.

As we can observe, both random and ErrLog have balanced accuracy of approximately 50%. Random logging has equal accuracy of 50% on either class. ErrLog logs every instance, achieving 100% accuracy on logged class, and 0% on unlogged class. Overall, the balanced accuracy of *LogAdvisor* is high, ranging from 84.6% to 93.4%, indicating high similarity to the logging decisions manually made by developers. Thus, *LogAdvisor* can learn a good representation of the common logging knowledge, and serve as a good baseline for guiding developers' logging behaviours towards better logging practice.

We also evaluate the effect of different contextual features (error type, methods, textual features, and syntactic features) on the prediction accuracy, as presented in Table 6.4. We can see that every type of contextual feature is useful, which leads to much higher balanced accuracy than random and ErrLog. *LogAdvisor*, by combining all these useful features, makes further improvement and achieves the highest balanced accuracy. These results also reveal that the contextual features extracted from the focused code snippets provide good indication of logging practices of developers.

For reference purpose, the results on other metrics such as precision, recall and F-score are provided in Table 6.5. Under a random setting, the experiments are run for 100 times and the average values are reported. As we can observe, for balanced dataset like System-A (43.8% positives), the F-Score is approximately 0.5.

Table 6.5: Prediction Accuracy w.r.t. Precision, Recall, and F-Score

Approaches	System-A			System-B		
	Prec.	Recall	F-Score	Prec.	Recall	F-Score
Random	0.437	0.499	0.466	0.209	0.500	0.295
ErrLog	0.438	1	0.609	0.209	1	0.346
Exception Type	0.639	0.784	0.705	0.307	0.796	0.443
Methods	0.501	0.907	0.646	0.604	0.459	0.522
Textual Features	0.738	0.698	0.718	0.656	0.713	0.683
Syntactic Features	0.816	0.933	0.871	0.695	0.783	0.736
LogAdvisor	0.871	0.921	0.895	0.810	0.869	0.838

Approaches	SharpDevelop			MonoDevelop		
	Prec.	Recall	F-Score	Prec.	Recall	F-Score
Random	0.188	0.503	0.274	0.191	0.499	0.276
ErrLog	0.187	1	0.315	0.191	1	0.321
Exception Type	0.328	0.853	0.474	0.413	0.903	0.567
Methods	0.519	0.278	0.362	0.529	0.450	0.486
Textual Features	0.588	0.532	0.558	0.606	0.634	0.620
Syntactic Features	0.908	0.587	0.713	0.853	0.671	0.751
LogAdvisor	0.793	0.714	0.752	0.856	0.824	0.839

For other datasets whose data are imbalanced, the F-Score is much lower. For ErrLog, because of its conservative logging, it will logs each exception instance, and thus get a recall of 100%. However, its precision is heavily influenced by the data imbalance, leading to low F-Score close to the random approach. Our LogAdvisor, by combining all useful features, achieves a high F-Score with 0.752$\sim$0.895.

6.3.3 The Effect of Different Learning Models

By default, we use decision tree (J48) to train our model, due to its simplicity as well as its effectiveness shown in our previous study [59]. We also examine the impact of different learning models on the prediction accuracy. We have tried a number of popular learning models, including Naive Bayes, Bayes Net, Logistic Regression, Support Vector Machine (SVM), and Decision Tree,

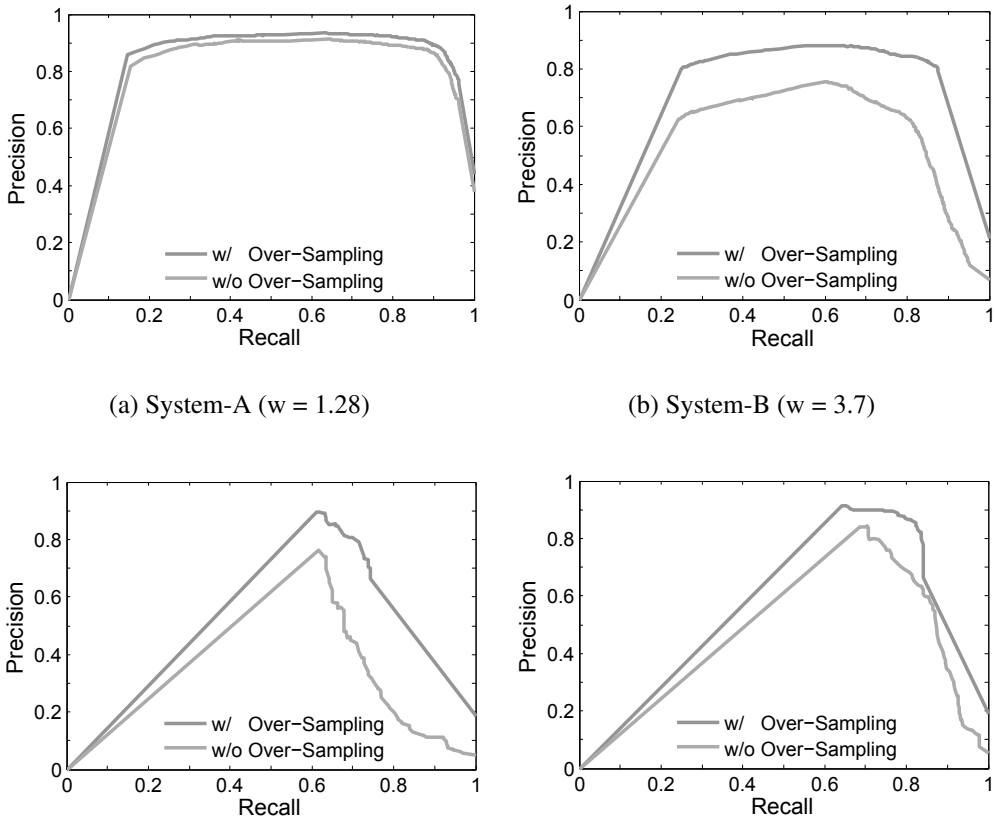
Table 6.6: Balanced Accuracy of Different Learning Models

Models	Exception Snippets			
	System-A	System-B	SharpDev	MonoDev
Naive Bayes	0.701	0.623	0.686	0.714
Bayes Net	0.729	0.751	0.688	0.862
Logistic Regression	0.881	0.834	0.772	0.858
SVM	0.898	0.886	0.878	0.903
Decision Tree	0.934	0.927	0.846	0.932
Models	Return-value-check Snippets			
	System-A	System-B	SharpDev	MonoDev
Naive Bayes	0.746	0.766	0.788	0.762
Bayes Net	0.802	0.814	0.845	0.859
Logistic Regression	0.806	0.834	0.856	0.848
SVM	0.815	0.885	0.873	0.877
Decision Tree	0.903	0.927	0.865	0.918

by using their Weka implementations. The evaluation results in Table 6.6 show that all the learning models lead to overall good prediction accuracy. In particular, Bayes-based learning models are based on probability theory. Unlike natural language text, the features extracted from source code are short and linguistically sparse, so Bayes-based learning models work slightly worse in our settings. Logistic Regression is a linear classifier, thus it may not fit well with our data. Decision Tree achieves the best overall accuracy, because this algorithm can solve non-linear classification problem. Furthermore, this algorithm can implicitly perform feature selection, which removes the redundant or irrelevant features and runs much faster than SVM for our data.

6.3.4 The Effect of Imbalance Handling

The data we collected from our studied software systems are imbalanced between logged instances and unlogged instances. To study the impact of data imbalance on prediction accuracy, we apply the state-of-the-art imbalance handling approach (SMOTE) described in Section 6.2.4 to balance the training data and evaluate the perfor-



(a) System-A (w = 1.28)

(b) System-B (w = 3.7)

(c) SharpDevelop (w = 4.34)

(d) MonoDevelop (w = 4.24)

Figure 6.8: Impact of Data Imbalance Handling on Prediction Accuracy

mance improvement. Figure 6.8 provides the precision-recall plots of the prediction results on exception snippets, which present the trade-off between precision and recall. The weights used in our experiments are also shown in the figure. For example, we create synthetic logged instances of System-B by a weight of 3.7. We can see that System-B, SharpDevelop and MonoDevelop have large improvement on prediction accuracy, while the improvement on System-A is small, because the data of System-A is more balanced compared to the others. The results indicate that our imbalance handling approach is helpful at improving the prediction accuracy.

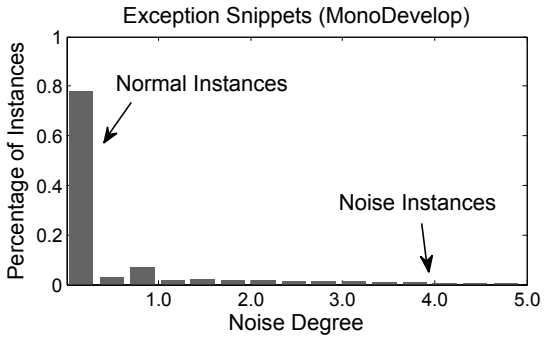
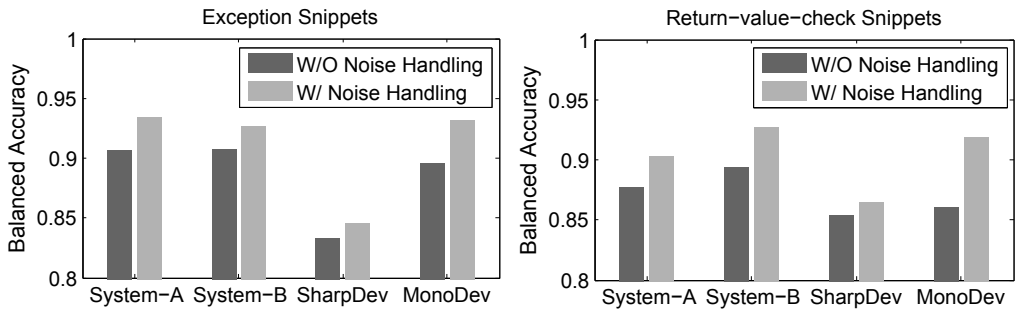


Figure 6.9: Instance Distribution over Noise Degree

6.3.5 The Effect of Noise Handling

To evaluate the effect of noise handling approach, we first study the instance distribution across the noise degree (φ_i) values, and then compare the prediction results with noise handling and those without noise handling. For ease of presentation, we only plot the instance distribution regarding exception snippets of MonoDevelop in Fig. 6.9, while the results of other systems are also similar. In particular, we set the number of nearest neighbours, k, to 5. So φ_i has a value range of $0 \sim 5$. It shows that the majority (about 88%) of instances have a noise degree value close to 0, indicating that each examined instance has the same logging label with almost all of its nearest neighbours. Only a small proportion of instances are likely noise data (*e.g.*, those with noise degree $\varphi_i > 3$). To some extent, this reveals the quality of data. In our study, we tune the threshold and flag about 5% of instances with top ranked φ_i values as noises, which are removed them in the training phase. As the evaluation results shown in Fig. 6.10(a)(b), the noise handling approach makes further improvement on the prediction accuracy. It indicates that properly removing potential noise data can make our model learn the common logging knowledge more effectively.



(a) The Effect on Exception Snippets

(b) The Effect on Return-value-check Snippets

Figure 6.10: Noise Handling Evaluation Results

Table 6.7: Accuracy on Golden Set

Focused Code Snippets	SharpDevelop	MonoDevelop
Exception Snippets	0.667	0.902
Return-value-check Snippets	0.789	0.751
Overall	0.750	0.854

6.3.6 Evaluation on Golden Set

Due to the lack of "ground truth" on optimal logging, there is no guarantee about the logging quality of the collected data, even though we employ several mature software systems as our subjects. The logging behaviours of developers can be ad-hoc. In some cases, developers perform logging as afterthoughts, for example, after a failure happens and logs are needed. Logging statements may be added, modified, and even deleted with the evolution of systems, as reported in [140].

To extract the "golden practice" of logging, we examine the revision histories of the two open-source projects, and find out the logging statements that are added or modified along with patches. We use these logging statements as the "golden set" for evaluation, because they are likely good representatives of useful logging instances. Specifically, we collect 75 such logging instances in

SharpDevelop and 135 such logging instances in MonoDevelop, including a total of 116 exception snippets and 94 return-value-check snippets. We utilize this golden set as the testing data to evaluate the performance of *LogAdvisor*. Table 6.7 show the accuracy results, with an overall accuracy ranging from 75%∼85%, which denotes the percentage of logging statements that are covered by *LogAdvisor*.

6.3.7 Cross-Project Evaluation

In within-project learning, *LogAdvisor* leverages the existing logging instances within the same project as training data to construct the predictive model. The above experiments provide promising results on the prediction accuracy of within-project evaluation, strongly indicating that *LogAdvisor* likely work well in the scenario of developing some new components in the same project. However, many real-world projects are small or new, which have limited training data for model construction. In such cases, it is valuable to explore whether cross-project learning can help.

In cross-project learning, we enrich the training data by incorporating the data instances extracted from a similar project (*source project*), and then apply the trained model to the *target project* for logging prediction. However, in contrast to within-project learning, cross-project learning is significantly more challenging [177], such as handling project-specific features. To address these challenges, we extract the common features that are shared between projects. We find that many system APIs and error types are actually common among different projects. We further leverage these common features to evaluate the performance of cross-project learning between different pairs of our studied systems (one source project for training and one target project for testing).

Figure 6.11 presents the cross-project evaluation results with comparison to the within-project evaluation results. The corre-

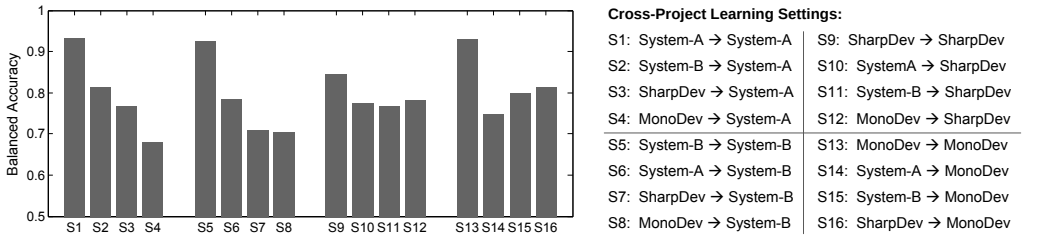


Figure 6.11: Cross-Project Evaluation Results (Training Project $\rightarrow$ Testing Project)

sponding experimental settings are shown in the right panel of the figure, where we use one project for training and one project for testing. Especially for within-project evaluation, we use 10-fold cross evaluation within the same project to evaluate the performance. Taking System-A as an example, we obtain a balanced accuracy of 93.4% for within-project evaluation (S1), while achieving 81.5%, 76.7% , and 68.0% balanced accuracy by using System-B (S2), SharpDevelop (S3), and MonoDevelop (S4) as training project respectively. Overall, the results indicate that the performance of cross-project learning is largely degraded compared with within-project learning. The reason is that different projects may follow different logging practices, and some project-specific knowledge (e.g., domain exceptions and methods) are challenging to adapt to other projects. Similar to our intuition, we find that the cross-project evaluation results among similar projects (*e.g.*, System-A and System-B, SharpDevelop and MonoDevelop) are slightly better than the results among dissimilar projects (*e.g.*, System-A and SharpDevelop, System-B and MonoDevelop). Furthermore, these results can serve as a baseline for further improvement by exploring other sophisticated techniques, such as transfer learning across projects [93].

6.4 User Study

To further measure the effectiveness of *LogAdvisor*, we conduct a controlled user study among engineers from Microsoft and a local IT company in China. We invited 37 participants in total, including 23 staff developers and 14 interns, who have an average of 4.9 years of programming experience. In addition, 22 (59%) of them use logging frequently while 12 (32%) of them use logging occasionally. The user study is conducted through an online questionnaire, which consists of 11 questions: 5 questions for the background of participants and their understanding on logging practices, 4 questions for case studies on logging, and 2 questions for assessment of our logging suggestion results. For reproducibility, a copy of the questionnaire is provided on our project page[3].

To perform logging case studies, we randomly select 20 exception snippets and 20 return-value-check snippets from MonoDevelop. Half of them are logged, while the other half are not. We remove the logging statements in code snippets and ask participants to make logging decisions on whether to log. The original logging labels made by code owners are taken as the "ground truth". However, sometimes, it is hard for participants (not code owners themselves) to understand the code logic well by reading only a small code snippet. To mitigate this issue, we group two code snippets with different logging labels (*e.g.*, one logged exception and one unlogged exception) into a pair. Then we ask the participants to choose which one is more likely to be logged from the pair, because it is easier for an participant to make choice through comparison. To evaluate the effectiveness of *LogAdvisor*, two groups of pairs are provided: one group with our logging suggestions, and the other group without logging suggestions. The suggestion results are provided from our trained model, with an accuracy of approximately 80% on these case-study snippets. To make a fair comparison, each participant

[3] `http://cuhk-cse.github.io/LogAdvisor`

marks an equal number of pairs in each group, and each pair is marked by at least three participants. In particular, we leverage the online survey system, Qualtrics[4], to build 10 questionnaires, each using 4 different pairs of code snippets. We distribute the survey links evenly to the participants. Furthermore, we record the time they spend on making each logging choice using the timing functionality of Qualtrics.

Results: We evaluate the accuracy that the participants correctly recover the logging decisions of the code owners. For the group without logging suggestions, the accuracy is 60%, while the group with logging suggestions achieves an accuracy of 75%, with a relative improvement of 25%. As for time consumption, the participants took 33% less time on average to make a logging choice with our logging suggestions (28 seconds v.s. 42 seconds). In addition, we query the feedback from the participants by the question "Do you think the suggestion result is useful for your logging choice?", and 70% of participants think it is useful. These results provide a strong evidence in the effectiveness of our logging suggestion.

6.5 Limitations and Discussion

Logging quality: The approach of "learning to log" works under the premise that the training data have high logging quality. In such a setting, the constructed model can represent the common (and good) logging knowledge and generalize well to predictions of new instances. However, there is no "ground truth" on what is high-quality (or optimal) logging. In our study, we assume that our studied software systems have reasonably good logging implementations due to their high code quality, active maintenance and long history of evolution. To a certain degree, it has been endorsed by our evaluation results (*e.g.*, high prediction accuracy, positive user feedback). Besides, our noise handling approach can

[4] `http://qtrial.qualtrics.com`

further mitigate the data quality issue by detecting and omitting the noisy logging instances from the training data, thus improving the performance of *LogAdvisor*.

Diversity of subject software systems: Our study was conducted on four software systems written in C#, thus its validity may be threatened by the limited diversity of our studied systems. To mitigate this threat, we choose the subjects including both commercial software systems from a leading software company like Microsoft and popular open-source software systems on GitHub. These systems are actively maintained and have a long history of evolution, which can serve as a representative of real practice. Besides, two of them are online services while the other two are IDEs, thus yielding both similar projects and dissimilar projects for our study. We believe that our approach and the results derived from these systems are easily reproducible and can be generalizable to many other software systems. Future studies on more types of software systems may further reduce this threat.

Where to log v.s. what to log: To achieve good logging quality, developers need to make informed decisions on both where to log and what to log. The ideal of "learning to log" is to help developers resolve both decisions. However, as an initial step towards this goal, we focus primarily on where to log in this work, because it is the first logging decision to make and sometimes can determine (or narrows down) what to log. For example, when developers decide to log an exception, the contents to be recorded become much more specific, including the exception message, stack trace, etc. Besides, a recent study [141] has built an *LogEnhancer* tool that can enrich the recorded contents by automatically identifying and inserting critical variable values into the existing logging statements. As part of our future work, this tool can be further integrated into our "learning to log" framework to facilitate log automation, where *LogAdvisor* determines where to log and *LogEnhancer* determines what to log.

Potential Improvements: Towards "learning to log", we still have a number of potential directions that deserve further exploration for improvements: *1) Other factors on logging decision*. The logging behaviours of developers can be quite complex and vary among developers. Also, the logging statements can be dynamically updated, such as deletion and modification. Thus, additional consideration of factors such as code owner, check-in time and execution frequency of code may further enhance the performance of logging prediction. *2) Interdependence of logging statements*. Our approach identifies each logging point sequentially and in isolation. In some cases, logging at one point may impact another. For example, a *try-catch block* may be enclosed in another *catch* block, and the exception may be thrown to the upper one to log. Or sometimes, logging statements at critical points are used together to record the execution path. Further exploration of a joint inference model (*e.g.*, graphical models, Markov chains) may help in this case. *3) Runtime logging*. Current logging statements are mostly statically inserted into the code. There is a new proposal for runtime logging, in which whether to log or not can be determined at runtime. For example, logs may be recorded by adaptive sampling [139] or only be recorded when encountering some problems (*e.g.*, a failed request or a long response) [14]. Although such sophisticated runtime logging mechanism is not supported by our studied systems, it is a promising direction for exploration to balance utility and overhead of logging.

6.6 Summary

Strategic logging is important yet difficult for online service system development. However, current logging practices are not well documented and cannot provide strong guidance on developers' logging decisions. To fill this gap, we propose a "learning to log" framework, which aims to automatically learn the common logging

practices from existing code repositories. As a proof of concept, we implement an automatic logging suggestion tool, *LogAdvisor*, which can help developers make informed logging decisions on where to log and potentially reduce their effort on logging. Evaluation results on both industrial and open-source software systems, as well as a controlled user study, demonstrate the feasibility and effectiveness of *LogAdvisor*. To the best of our knowledge, this study makes the first attempt to provide automatic logging tool support for developers. We believe it is an important step towards automatic logging.

On the other hand, although our experiments have shown strong evidence on the usefulness of our logging suggestion tool LogAdvisor, it is still a prototype now. In our future work, we will implement it as a IDE-pluggable tool and demonstrate its real use of logging suggestion to developers.

□ **End of chapter.**

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the main contributions of this thesis and provide several interesting future directions.

7.1 Conclusion

Quality management is critical to online service systems. However, in many cases, traditional engineering approaches are inapplicable to deal with service quality problems in production. Online service systems are generating a variety of service data, including quality of service (QoS) information of Web services, service logs, service dependency graphs, etc. In this thesis, we have developed data-driven approaches to gaining actionable insights from such service data to aid in service quality management of online service systems.

In particular, in Chapter 3, we propose a Web service positioning (WSP) framework for joint response time monitoring and prediction. The WSP framework is built by combining both advantages of network coordinate based approaches and collaborative filtering based approaches. Our WSP framework solves the data sparsity problem of existing approaches and significantly enhances the prediction accuracy. We also conduct a large-scale experiment to collect real-world response time data of Web services and further leverage the data to verify the effectiveness of our WSP approach. The dataset

includes 359,400 response time values from 200 users on 1,597 real-world Web services, which is essential to future research in this field.

In Chapter 4, we study the problem online QoS prediction of Web services. This is the first work to address the problem of QoS prediction on candidate services to guide candidate service selection for runtime service adaptation. A novel QoS prediction approach, adaptive matrix factorization (AMF), has been proposed to achieve this goal. AMF extends the traditional matrix factorization model with techniques of data transformation, online learning, and adaptive weights to achieve accurate, online, and scalable QoS predictions. Comprehensive experiments are conducted based on a real-world large-scale QoS dataset of Web services to evaluate our AMF approach in terms of accuracy, efficiency, and scalability.

In Chapter 5, we conduct the first work to cope with the privacy issue in QoS-based Web service recommendation. We propose a simple yet effective privacy-preserving framework, and further develop two representative privacy-preserving QoS prediction approaches, P-UIPCC and P-PMF, under this framework. By using our approaches, QoS-based Web service recommendation can be made without revealing private QoS data of users. Comprehensive experiments are conducted on a real-world large-scale QoS dataset of Web services to evaluate the effectiveness of privacy-preserving QoS prediction approaches.

In Chapter 6, we propose a "learning to log" framework to help developers make informed logging decisions during development. Logging is currently an important yet tough decision which mostly depends on the domain knowledge of developers. To reduce the effort on making logging decisions, we provide the design and implementation of a logging suggestion tool, LogAdvisor, which automatically learns the common logging practices on where to log from existing logging instances and further leverages them for actionable suggestions to developers. Specifically, we identify the important factors for determining where to log and extract them as

structural features, textual features, and syntactic features. Then, by applying machine learning techniques (e.g., feature selection and classifier learning) and noise handling techniques, we achieve high accuracy of logging suggestions. We evaluate LogAdvisor on two industrial online service systems from Microsoft and two open-source software systems from Github.

In summary, by use of data-driven approaches, we have addressed some important problems in quality management of online service systems, including response time prediction of Web services, online QoS prediction of Web services, privacy-preserving QoS prediction of Web services, and learning to log for runtime service adaptation. Moverover, for ease of reproducing our research results and to promote future research on related topics, all of the data and source code used in this thesis have been made publicly available.

7.2 Future Work

Data-driven quality management of online service systems is a promising field of research and practice. Although we have made a number of significant achievements in this thesis, there are still many interesting research directions that deserve for future investigations.

Context-Aware Reliability Prediction

Reliability, as one of the most important QoS attributes, measures the probability of failure-free software operation for a specified period of time in a specified environment. Reliability prediction is an important task in software reliability engineering, which can aid in evaluating software design decisions for building reliable software systems. Reliability prediction is an important task in software reliability engineering, which has been widely studied in the last decades. However, most of these existing models target at analyzing traditional white-box software systems, where

the reliabilities of system components are all known or can be estimated through behaviour models from internal information of the components. Modelling and predicting user-perceived reliability of black-box services remain an open research problem.

Nowadays, various Web services are in widespread use in building online service systems, where each service provides a black-box functionality via some standard interfaces. To evaluate the reliability of a (third-party) black-box service, traditional white-box reliability prediction approaches become inapplicable due to a lack of its internal behaviour information. In addition, different from stand-alone software systems, Web services operate over the Internet and likely serve different users spanning worldwide. Therefore, the user-perceived service reliability depends not only on the service itself, but also heavily on the invocation context (*e.g.*, user locations, service workloads, network conditions). For instance, the user-perceived reliability may differ from user to user due to different user locations, and vary from time to time due to dynamic service workloads and network conditions. In such a setting, context-aware reliability prediction is an interesting direction for future research.

Automatic Logging

In our thesis, we propose a "learning to log" framework which can automatically learn common logging practices from existing code repositories to help developers make informed logging decisions. This is the first step towards automatic logging, but it is far from perfect. Current logging of software systems is typically made manually through developers by inserting static logging statements into source code, and prints log messages at some fixed program locations. The quality of logging, therefore, heavily depends on the expertise of developers who make logging decisions. In many cases, such logging strategy may be sub-optimal because the logging behaviours of developers are not collectively optimized, which in turn leads to redundant or useless logs. In addition, all the logging

statements, once being inserted by developers, will always print the same logs for every execution. This is a static logging strategy which cannot evolve adaptively with runtime operational environment.

However, a software system is rapidly evolving and its operation context is also constantly changing. On one hand, for modern software systems, developers are quickly fixing defects and developing new features, especially under the current agile development environment. Such rapid development brings in a lot of fresh code, which deserves more inspection via logging because the code is less frequently executed. It is desired that the logging strategy can be evolved too, with old code being less logged while new code producing more log information. On the other hand, the logging overhead is closely related to operational environment. For example, in online service systems, the logging overhead can be rapidly exposing, when a spike occurs in the number of requests. It will be more cost-effective to log these user requests adaptively in an on-demand manner.

It is highly desired that logging points can be placed wisely at different positions along critical execution paths, but each logging point can dynamically determine whether to log during each execution according to the operational context. For example, for online service systems, we can merely log the runtime information when a request is failed, or else eliminating the log information in memory directly to reduce the overhead of writing logs into a file. More research in this direction is necessary to facilitate increased intelligence and automation in logging.

Massive Log Analysis

For large-scale online service systems, there are typically millions of users who generate tens of thousands of requests to the system every second. With the continuous 24x7 operating of these online systems, we are able to collect massive log data. Logs are extremely critical in system troubleshooting. When online service systems

fail in the field, in many cases, developers can only rely on the logs to pinpoint the root causes, because it is difficult to stop the system for debugging, or to reproduce the system problems in the lab due to distributed operational environment and dynamic service workloads. The log data bring us more complete information of the system behaviours together with the challenges to store and analyze. However, in current practice, logs are typically retrieved by searching for some keyword such as "fail", "error", or other customized regular expressions. There may be a lot of matches where substantial manual efforts are needed to identify the culprit. Therefore, manual analysis of system data is lab-intensive, error-prone, and inefficient. Many organizations are still struggling to process large amounts of log data and are spending significant amounts of time analyzing logs for actionable insights.

On the other hand, techniques of data analytic such as machine learning and data mining have been widely studied over last decades, yielding a wide variety of applications in bioinformatics, robotics, image processing, and natural language processing. However, the use of data analytics in troubleshooting online service systems is less explored, which deserves for future study. For example, PCA can be used to reduce the redundant information of logs and extract effective features, and sparse coding can be used to model different features that may act as potential root causes for a system problem.

However, to achieve effective and efficient log analysis, there are several unique challenges to be addressed: (1) **Data volume**: With a system scaling up, dealing with a large volume of log data poses a significant challenge to the efficiency of traditional log analysis techniques. (2) **Unstructured data**: To enable the convenience and flexibility of logging, logs are usually written in natural language by developers to indicate the content of each event. The unstructured nature makes the analysis of such logs very difficult. (3) **Temporal-spatial property**: Logs are typically generated as a type of temporal-spatial data. Each log has a timestamp, which is

a temporal property. Also, each stamp is generated by a specific machine, which shows its spatial property. How best it is to leverage such temporal-spatial properties of logs for troubleshooting has continued to be a vexing problem.

These challenges may not be even encountered in traditional research in data analytics, but are unique in massive log analysis. Log analysis is a widely-studied but open problem. More research efforts are desired to address these unique challenges and further advance the current log analysis techniques for system troubleshooting, such as anomaly detection, and system problem diagnosis.

□ **End of chapter.**

Appendix A

List of Publications

1. **Jieming Zhu**, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 415-425, 2015.

2. **Jieming Zhu**, Pinjia He, Zibin Zheng, and Michael R. Lyu. A Privacy-Preserving QoS Prediction Framework for Web Service Recommendation. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 241-248, 2015.

3. Cuiyun Gao, Baoxiang Wang, Pinjia He, **Jieming Zhu**, Yangfan Zhou, and Michael R. Lyu. PAID: Prioritizing App Issues for Developers by Tracking User Reviews Over Versions. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

4. Qiang Fu, **Jieming Zhu**, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 24-33, 2015.

5. **Jieming Zhu**, Pinjia He, Zibin Zheng, and Michael R. Lyu. Towards Online, Accurate, and Scalable QoS Prediction for

Runtime Service Adaptation. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 318-327, 2014.

6. Pinjia He, **Jieming Zhu**, Zibin Zheng, Jianlong Xu, and Michael R. Lyu. Location-based Hierarchical Matrix Factorization for Web Service Recommendation. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 297-304, 2014.

7. Pinjia He, **Jieming Zhu**, Jianlong Xu, and Michael R. Lyu. A Hierarchical Matrix Factorization Approach for Location-based Web Service QoS Prediction. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages 290-295, 2014.

8. **Jieming Zhu**, Zibin Zheng, and Michael R. Lyu. DR^2: Dynamic Request Routing for Tolerating Latency Variability in Online Cloud Applications. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 589-596, 2013.

9. Zibin Zheng, **Jieming Zhu**, and Michael R. Lyu. Service-generated Big Data and Big Data-as-a-Service: An Overview. In *Proc. of the IEEE International Congress on Big Data*, pages 403-410, 2013.

10. **Jieming Zhu**, Zibin Zheng, Yangfan Zhou, and Michael R. Lyu. Scaling Service-oriented Applications into Geo-distributed Clouds. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages 335-340, 2013.

11. **Jieming Zhu**, Yu Kang, Zibin Zheng, and Michael R. Lyu. WSP: A Network Coordinate based Web Service Positioning Framework for Response Time Prediction. In *Proc. of the*

IEEE International Conference on Web Services (ICWS), pages 90-97, 2012.

12. **Jieming Zhu**, Yu Kang, Zibin Zheng and Michael R. Lyu. A Clustering-based QoS Prediction Approach for Web Services Recommendation. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages 93-98, 2012.

□ **End of chapter.**

Bibliography

[1] 10 best practices with exceptions. http://www.wikijava.org/wiki/10_best_practices_with_Exceptions.

[2] 7 good rules to log exceptions. http://codemonkeyism.com/7-good-rules-to-log-exceptions.

[3] 7 more good tips on logging. http://codemonkeyism.com/7-more-good-tips-on-logging.

[4] Achieving rapid response times in large online services. http://www.slideshare.net/yarapavan/achieving-rapid-response-times-in-large-online-services.

[5] Amazon just lost $4.8M after going down for 40 minutes. http://www.geekwire.com/2013/amazon-lost-5m-40-minutes.

[6] The art of logging. http://www.codeproject.com/Articles/42354/The-Art-of-Logging.

[7] Code to logging ratio? http://stackoverflow.com/questions/153524/code-to-logging-ratio#153547.

[8] Exception logging in javascript. https://developer.mozilla.org/en-US/docs/Exception_logging_in_JavaScript.

[9] Exception logging: Why not log all exceptions? http://stackoverflow.com/questions/25560953/exception-logging-why-not-log-all-exceptions.

[10] Exponential moving average. http://en.wikipedia.org/wiki/Moving_average.

[11] Google's 5-minute outage means $545,000 revenue loss, 40% drop in global website traffic. http://www.neowin.net/news/googles-5-minute-outage-means-545000-revenue-loss-40-drop-in-global-website-traffic.

[12] Logging best practices. https://idea.popcount.org/2013-12-31-logging-best-practises.

[13] Microsoft Roslyn CTP. http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx.

[14] Optimal logging testing blog (Google). http://googletesting.blogspot.com/2013/06/optimal-logging.html.

[15] Overview of unified logging system (ULS). http://msdn.microsoft.com/en-us/library/office/ff512738(v=office.14).aspx.

[16] The problem with logging. http://blog.codinghorror.com/the-problem-with-logging.

[17] Singular value decomposition (SVD). http://en.wikipedia.org/wiki/Singular_value_decomposition.

[18] The SOA source book. https://www.opengroup.org/soa/source-book/soa/soa_ea.htm.

[19] SOAP, XML-RPC, and REST. `http://jimmyzimmerman.com/blog/2007/01/soap-xml-rpc-and-rest.html`.

[20] Why not log all exceptions in MonoDevelop? `http://lists.ximian.com/pipermail/monodevelop-list/2014-August/016201.html`.

[21] Why not log all exceptions in SharpDevelop? `https://github.com/icsharpcode/SharpDevelop/issues/554`.

[22] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–32, 2010.

[23] C. C. Aggarwal and C. Zhai. A survey of text classification algorithms. *Mining Text Data*, pages 163–222, 2012.

[24] M. Alicherry and T. V. Lakshman. Network aware resource allocation in distributed clouds. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 963–971, 2012.

[25] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient QoS-aware service composition. In *Proc. of the ACM International Conference on World Wide Web (WWW)*, pages 881–890, 2009.

[26] M. Alrifai, D. Skoutas, and T. Risse. Selecting skyline services for qos-based web service composition. In *Proc. of the ACM International Conference on World Wide Web (WWW)*, pages 11–20, 2010.

[27] A. Amin, L. Grunske, and A. Colman. An automated approach to forecasting qos attributes based on linear and non-linear time series modeling. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 130–139, 2012.

[28] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering (TSE)*, 33(6), 2007.

[29] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 157–166, 2010.

[30] N. Ball and P. Pietzuch. Distributed content delivery using load-aware network coordinates. In *Proc. of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 77:1–77:6, 2008.

[31] L. Baresi and S. Guinea. Self-supervising BPEL processes. *IEEE Transactions on Software Engineering (TSE)*, 37(2):247–263, 2011.

[32] B. Bassett and N. A. Kraft. Structural information based term weighting in text retrieval for feature location. In *Proc. of the International Conference on Program Comprehension (ICPC)*, pages 133–141, 2013.

[33] A. Bilge, C. Kaleli, I. Yakut, I. Gunes, and H. Polat. A survey of privacy-preserving collaborative filtering schemes. *International Journal of Software Engineering and Knowledge Engineering*, pages 1085–1108, 2013.

[34] M. Björkqvist, L. Y. Chen, and W. Binder. Dynamic replication in service-oriented systems. In *Proc. of the IEEE/ACM*

International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 531–538, 2012.

[35] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann. The balanced accuracy and its posterior distribution. In *Proc. of the International Conference on Pattern Recognition (ICPR)*, pages 3121–3124, 2010.

[36] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online web systems auto-configuration. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 2–11, 2009.

[37] J. Canny. Collaborative filtering with privacy. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 45–57, 2002.

[38] J. F. Canny. Collaborative filtering with privacy. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 45–57, 2002.

[39] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola. MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering (TSE)*, 38(5):1138–1159, 2012.

[40] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, and R. Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 131–140, 2009.

[41] F. Chang, R. Viswanathan, and T. L. Wood. Placement in clouds for application-level latency requirements. In *Proc.*

of the IEEE International Conference on Cloud Computing (CLOUD), pages 327–335, 2012.

[42] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.

[43] X. Chen, X. Liu, Z. Huang, and H. Sun. RegionKNN: A scalable hybrid collaborative filtering algorithm for personalized web service recommendation. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 9–16, 2010.

[44] X. Chen, Z. Zheng, X. Liu, Z. Huang, and H. Sun. Personalized QoS-aware web service recommendation and visualization. *IEEE Transactions on Services Computing (TSC)*, 2011.

[45] X. Chen, Z. Zheng, Q. Yu, and M. R. Lyu. Web service recommendation via exploiting location and QoS information. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(7):1913–1924, 2014.

[46] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 231–244, 2004.

[47] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 105–118, 2005.

[48] E. Costante, F. Paci, and N. Zannone. Privacy-aware web service composition and ranking. In *Proc. of the IEEE In-*

ternational Conference on Web Services (ICWS), pages 131–138, 2013.

[49] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proc. of the European Conference on Computer Systems (EuroSys)*, pages 199–214, 2011.

[50] F. Dabek, R. Cox, M. F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *Proc. of the ACM SIGCOMM Conference (SIGCOMM)*, pages 15–26, 2004.

[51] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[52] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[53] B. Donnet, B. Gueye, and M. A. Kâafar. A survey on network coordinates systems, design, and security. *IEEE Communications Surveys and Tutorials*, 12(4):488–503, 2010.

[54] J. Edmondson, A. Gokhale, and D. Schmidt. Approximation techniques for maintaining real-time deployments informed by user-provided dataflows within a cloud. In *Proc. of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2012.

[55] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu. Topology-aware deployment of scientific applications in cloud comput-

ing. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 319–326, 2012.

[56] U. Feige. A threshold of ln$n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[57] B. Frénay and M. Verleysen. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 25(5):845–869, 2014.

[58] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of the IEEE Conference on Data Mining (ICDM)*, pages 149–158, 2009.

[59] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2014.

[60] I. Goiri, J. Guitart, and J. Torres. Characterizing cloud federation for enhancing providers' profit. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 123–130, 2010.

[61] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[62] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Exploration Newsletter*, 11(1):10–18, 2009.

[63] A. E. Hassan and T. Xie. Software intelligence: the future of mining software engineering data. In *Proc. of the Workshop on Future of Software Engineering Research, with the*

ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pages 161–166, 2010.

[64] Q. He, J. Yan, H. Jin, and Y. Yang. Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. *IEEE Transactions on Software Engineering (TSE)*, 40(2):192–215, 2014.

[65] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proc. of the Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 230–237, 1999.

[66] T. R. Hoens, M. Blanton, A. Steele, and N. V. Chawla. Reliable medical recommendation systems with patient privacy. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 4(4):67, 2013.

[67] B. Jiang, W. K. Chan, Z. Zhang, and T. H. Tse. Where to adapt dynamic service compositions. In *Proc. of the ACM International Conference on World Wide Web (WWW)*, 2009.

[68] Y. Jiang, J. Liu, M. Tang, and X. F. Liu. An effective web service recommendation method based on personalized collaborative filtering. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, 2011.

[69] Y. Kang, Z. Zheng, and M. R. Lyu. A latency-aware co-deployment mechanism for cloud-based services. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 630–637, 2012.

[70] Y. Kang, Y. Zhou, Z. Zheng, and M. R. Lyu. A user experience-based cloud service redeployment mechanism. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 227–234, 2011.

[71] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 481–490, 2011.

[72] A. Klein, F. Ishikawa, and S. Honiden. Towards network-aware service composition in the cloud. In *Proc. of the ACM International Conference on World Wide Web (WWW)*, pages 959–968, 2012.

[73] W. Klosgen. Anonymization techniques for knowledge discovery in databases. In *Proc. of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 186–191, 1995.

[74] G. Lee, J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy. The unified logging infrastructure for data analytics at twitter. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pages 1771–1780, 2012.

[75] S. Lee and S. Sahu. Network distance based coordinate systems for p2p multimedia streaming. In *Proc. of the IEEE Network Operations and Management Symposium (NOMS)*, pages 793–796, 2010.

[76] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of SLA violations in composite services. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, 2010.

[77] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering (TSE)*, 34(4):485–496, 2008.

[78] X. Liu and I. Fulia. Incorporating user, topic, and service related latent factors into web service recommendation. In

Proc. of the IEEE International Conference on Web Services (ICWS), pages 185–192, 2015.

[79] W. Lo, J. Yin, S. Deng, Y. Li, and Z. Wu. An extended matrix factorization approach for QoS prediction in service selection. In *Proc. of the IEEE International Conference on Services Computing (SCC)*, pages 162–169, 2012.

[80] X. Lu, H. Wang, J. Wang, J. Xu, and D. Li. Internet-based virtual computing environment: Beyond the data center as a computer. *Future Generation Computer Systems*, 29(1):309–322, 2013.

[81] M. R. Lyu. *Handbook of software reliability engineering*. IEEE Computer Society Press, 1996.

[82] M. R. Lyu and L. Zhang. Guest editorial: Recommendation techniques for services computing and cloud computing. *IEEE Transactions on Services Computing (TSC)*, 8(3):422–424, 2015.

[83] H. Ma, I. King, and M. R. Lyu. Effective missing data prediction for collaborative filtering. In *Proc. of the ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 39–46, 2007.

[84] Y. Mao, L. Saul, and J. Smith. IDES: An internet distance estimation service for large networks. *IEEE Journal on Selected Areas in Communications*, 24(12):2273 –2284, 2006.

[85] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 627–636, 2009.

[86] M. T. Melo, S. Nickel, and F. S. da Gama. Facility location and supply chain management - a review. *European Journal of Operational Research*, 196(2):401–412, 2009.

[87] A. Metzger, C.-H. Chi, Y. Engel, and A. Marconi. Research challenges on online service quality prediction for proactive adaptation. In *Proc. of the Workshop on European Software Services and Systems Research - Results and Challenges (S-Cube)*, pages 51–57, 2012.

[88] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(6):1245–1255, 2013.

[89] M. Montanari, J. H. Huh, D. Dagit, R. Bobba, and R. H. Campbell. Evidence of log integrity in policy-based security monitoring. In *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pages 1–6, 2012.

[90] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proc. of the ACM International Conference on World Wide Web (WWW)*, 2008.

[91] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 26–26, 2012.

[92] V. Nallur and R. Bahsoon. A decentralized self-adaptation mechanism for service-based applications in the cloud. *IEEE Transactions on Software Engineering (TSE)*, 39(5):591–612, 2013.

[93] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 382–391, 2013.

[94] J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[95] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2002.

[96] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 801–812, 2013.

[97] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 378–388, 2013.

[98] A. J. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM (CACM)*, 55(2):55–61, 2012.

[99] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 815–825, 2012.

[100] H. Polat and W. Du. Privacy-preserving collaborative filtering using randomized perturbation techniques. In *Proc. of the IEEE Conference on Data Mining (ICDM)*, pages 625–628, 2003.

[101] Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a distance oracle for billion-node graphs. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pages 61–72, 2013.

[102] H. Qian. Proximity-aware cloud selection and virtual machine allocation in iaas cloud platforms. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, 2013.

[103] M. Qiao, H. Cheng, and J. X. Yu. Querying shortest path distance with bounded errors in large graphs. In *Proc. of the Conference on Scientific and Statistical Database Management (SSDBM)*, pages 255–273, 2011.

[104] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 1587–1596, 2001.

[105] W. Qiu, Z. Zheng, X. Wang, X. Yang, and M. R. Lyu. Reputation-aware qos value prediction of web services. In *Proc. of the IEEE International Conference on Services Computing (SCC)*, pages 41–48, 2013.

[106] N. Ramakrishnan, B. J. Keller, B. J. Mirza, A. Y. Grama, and G. Karypis. Privacy risks in recommender systems. *IEEE Internet Computing*, 5(6):54–62, 2001.

[107] R. Saha, M. Lease, S. Khurshid, and D. Perry.

[108] R. M. Sakia. The box-cox transformation technique: A review. *Journal of the Royal Statistical Societys*, 41(2):169–178, 1992.

[109] R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. In *Proc. of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2007.

[110] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, 2009.

[111] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proc. of the International World Wide Web Conference (WWW)*, pages 285–295, 2001.

[112] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 402–411, 2013.

[113] W. Shang, M. Nagappan, and A. E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 2013.

[114] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang. Understanding log lines using development knowledge. In *Proc. of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[115] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei. Personalized QoS prediction for web services via collaborative filtering. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 439–446, 2007.

[116] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei. Personalized QoS prediction for web services via collaborative filtering. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 439 –446, 2007.

[117] A. Shapiro and Y. Wardi. Convergence analysis of gradient descent stochastic algorithms. *Journal of Optimization Theory and Applications*, 91:439–454, 1996.

[118] A. Squicciarini, B. Carminati, and S. Karumanchi. A privacy-preserving approach for web service selection and provisioning. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2011.

[119] N. Srebro and T. Jaakkola. Weighted low-rank approximations. In *Proc. of the International Conference on Machine Learning (ICML)*, pages 720–727, 2003.

[120] M. Steiner and E. W. Biersack. Where is my peer? evaluation of the vivaldi network coordinate system in azureus. In *Proc. of the International IFIP TC 6 Networking Conference (NETWORKING)*, pages 145–156, 2009.

[121] M. Steiner, B. G. Gaglianello, V. K. Gurbani, V. Hilt, W. D. Roome, M. Scharf, and T. Voith. Network-aware service placement in a distributed cloud environment. In *Proc. of the ACM SIGCOMM Conference (SIGCOMM)*, pages 73–74, 2012.

[122] D. Strom and J. F. van der Zwet. Truth and lies about latency in the cloud. Interxion^{TM} white paper, 2012.

[123] X. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. Artificial Intellegence*, 2009.

[124] R. L. R. T. H. Cormen, C. E. Leiserson and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[125] X. Tang and J. Xu. Qos-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(10):921–932, 2005.

[126] S. Tbahriti, C. Ghedira, B. Medjahed, and M. Mrissa. Privacy-enhanced web service composition. *IEEE Transactions on Services Computing (TSC)*, 7(2):210–222, 2014.

[127] G. Tian, J. Wang, K. He, P. C. K. Hung, and C. Sun. Time-aware web service recommendations using implicit feedback. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 273–280, 2014.

[128] H. Wada, J. Suzuki, Y. Yamano, and K. Oba. Evolutionary deployment optimization for service-oriented clouds. *Software: Practice and Experience*, 41(5):469–493, 2011.

[129] C. Wang and J.-L. Pazat. A two-phase online prediction approach for accurate and timely adaptation decision. In *Proc. of the IEEE International Conference on Services Computing (SCC)*, pages 218–225, 2012.

[130] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 27–27, 2012.

[131] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann Publishers Inc., 2005.

[132] B. Xia, Y. Fan, C. Wu, K. Huang, W. Tan, J. Zhang, and B. Bai. Domain-aware service recommendation for service composition. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 439–446, 2014.

[133] T. Xie, J. Pei, and A. E. Hassan. Mining software engineering data. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 172–173, 2007.

[134] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *IEEE Computer*, 42(8):55–62, 2009.

[135] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 117–132, 2009.

[136] W. Xu, V. Venkatakrishnan, R. Sekar, and I. Ramakrishnan. A framework for building privacy-conscious composite web services. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 655–662. IEEE, 2006.

[137] C. Yu and L. Huang. Time-aware collaborative filtering for QoS-based service recommendation. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 265–272, 2014.

[138] D. Yu, Y. Liu, Y. Xu, and Y. Yin. Personalized QoS prediction for web services using latent factor models. In *Proc. of the IEEE International Conference on Services Computing (SCC)*, pages 107–114, 2014.

[139] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 293–306, 2012.

[140] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 102–112, 2012.

[141] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM*

Transactions on Computer Systems (TOCS), 30(1):4:1–4:28, 2012.

[142] Z. I. M. Yusoh and M. Tang. Composite saas placement and resource optimization in cloud computing using evolutionary algorithms. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 590–597, 2012.

[143] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering (TSE)*, 30(5):311–327, 2004.

[144] J. Zhan, C. Hsieh, I. Wang, T. Hsu, C. Liau, and D. Wang. Privacy-preserving collaborative recommender systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 40(4):472–476, 2010.

[145] D. Zhang, S. Han, Y. Dang, J. Lou, H. Zhang, and T. Xie. Software analytics in practice. *IEEE Software*, 30(5):30–37, 2013.

[146] D. Zhang and T. Xie. Software analytics: achievements and challenges. In *Proc. of the International Conference on Software Engineering (ICSE)*, page 1487, 2013.

[147] L. Zhang, J. Zhang, and H. Cai. *Services Computing: Core Enabling Technology of the Modern Services Industry*. Tsinghua University Press, 2007.

[148] Q. Zhang, Q. Zhu, M. F. Zhani, and R. Boutaba. Dynamic service placement in geographically distributed clouds. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 526–535, 2012.

[149] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system

performance problems. In *Proc. of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 644–653, 2005.

[150] W. Zhang, H. Sun, X. Liu, and X. Guo. Temporal qos-aware web service recommendation via non-negative tensor factorization. In *Proc. of the International World Wide Web Conference (WWW)*, pages 585–596, 2014.

[151] Y. Zhang, Z. Zheng, and M. R. Lyu. Wsexpress: A QoS-aware search engine for web services. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 91–98, 2010.

[152] Y. Zhang, Z. Zheng, and M. R. Lyu. Exploring latent features for memory-based QoS prediction in cloud computing. In *Proc. of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 1–10, 2011.

[153] Y. Zhang, Z. Zheng, and M. R. Lyu. WSPred: A time-aware personalized QoS prediction framework for web services. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 210–219, 2011.

[154] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao. Orion: shortest path estimation for large social graphs. In *Proc. of the 3rd Workshop on Online Social Networks (WOSN)*, 2010.

[155] W. Zheng, Q. Zhang, and M. Lyu. Cross-library api recommendation using web search engines. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 480–483, 2011.

[156] Z. Zheng and M. R. Lyu. An adaptive qos-aware fault tolerance strategy for web services. *Empirical Software Engineering*, 15(4):323–345, 2010.

[157] Z. Zheng and M. R. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 35–44, 2010.

[158] Z. Zheng and M. R. Lyu. Personalized reliability prediction of web services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):12, 2013.

[159] Z. Zheng and M. R. Lyu. Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. *IEEE Transactions on Computers (TC)*, 64(1):219–232, 2015.

[160] Z. Zheng, M. R. Lyu, and H. Wang. Service fault tolerance for highly reliable service-oriented systems: an overview. *SCIENCE CHINA Information Sciences*, 58(5):1–12, 2015.

[161] Z. Zheng, H. Ma, M. R. Lyu, and I. King. WSRec: A collaborative filtering based web service recommender system. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 437–444, 2009.

[162] Z. Zheng, H. Ma, M. R. Lyu, and I. King. QoS-aware web service recommendation by collaborative filtering. *IEEE Transactions on Services Computing (TSC)*, 4(2):140–152, 2011.

[163] Z. Zheng, H. Ma, M. R. Lyu, and I. King. QoS-aware web service recommendation by collaborative filtering. *IEEE Transactions on Services Computing (TSC)*, 4(2):140–152, 2011.

[164] Z. Zheng, H. Ma, M. R. Lyu, and I. King. Collaborative web service qos prediction via neighborhood integrated matrix factorization. *IEEE Transactions on Services Computing (TSC)*, 6(3):289–299, 2013.

[165] Z. Zheng, X. Wu, Y. Zhang, M. R. Lyu, and J. Wang. QoS ranking prediction for cloud services. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(6):1213–1222, 2013.

[166] Z. Zheng, Y. Zhang, and M. R. Lyu. Cloudrank: A QoS-driven component ranking framework for cloud computing. In *Proc. of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 184–193, 2010.

[167] Z. Zheng, Y. Zhang, and M. R. Lyu. Investigating QoS of real-world web services. *IEEE Transactions on Services Computing (TSC)*, 7(1):32–39, 2014.

[168] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King. Component ranking for fault-tolerant cloud applications. *IEEE Transactions on Services Computing (TSC)*, 5(4):540–550, 2012.

[169] Z. Zheng, J. Zhu, and M. R. Lyu. Service-generated big data and big data-as-a-service: An overview. In *Proc. of the IEEE International Congress on Big Data*, pages 403–410, 2013.

[170] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

[171] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 415–425, 2015.

[172] J. Zhu, P. He, Z. Zheng, and M. R. Lyu. Towards online, accurate, and scalable qos prediction for runtime service adaptation. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 318–327, 2014.

[173] J. Zhu, P. He, Z. Zheng, and M. R. Lyu. A privacy-preserving qos prediction framework for web service recommendation. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 241–248, 2015.

[174] J. Zhu, Y. Kang, Z. Zheng, and M. R. Lyu. WSP: A network coordinate based web service positioning framework for response time prediction. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 90–97, 2012.

[175] J. Zhu, Z. Zheng, and M. R. Lyu. DR2: dynamic request routing for tolerating latency variability in online cloud applications. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 589–596, 2013.

[176] J. Zhu, Z. Zheng, Y. Zhou, and M. R. Lyu. Scaling service-oriented applications into geo-distributed clouds. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages 335–340, 2013.

[177] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 91–100, 2009.