Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

4-18-2014

Event Mining for System and Service Management

Liang Tang
Florida International University, tangl99@gmail.com

DOI: 10.25148/etd.FI14071115
Follow this and additional works at: https://digitalcommons.fiu.edu/etd

Recommended Citation
Tang, Liang, "Event Mining for System and Service Management" (2014). *FIU Electronic Theses and Dissertations*. 1442.
https://digitalcommons.fiu.edu/etd/1442

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

EVENT MINING FOR SYSTEM AND SERVICE MANAGEMENT

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Liang Tang

2014

To: Dean Giri Narasimhan
College of Engineering and Computing

This dissertation, written by Liang Tang, and entitled Event Mining for System and Service Management, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

S.S. Iyengar

Shu-Ching Chen

Jinpeng Wei

Zhenmin Chen

Tao Li, Major Professor

Date of Defense: April 18, 2014

The dissertation of Liang Tang is approved.

Dean Giri Narasimhan
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

© Copyright 2014 by Liang Tang
All rights reserved.

DEDICATION

I dedicate my dissertation work to my family. A special feeling of gratitude to my loving parents, whose words of encouragement and push for tenacity ring in my ears.

I also dedicate this dissertation to my many friends who have supported me throughout the process. I will always appreciate all they have done, especially for helping me develop my technology skills and the many hours of proofreading.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Tao Li, for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. He has the attitude and the substance of a genius: he continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. Without his guidance and persistent help this dissertation would not have been possible.

I would like to thank Dr. Shu-Ching Chen, who lets me join the BCIN research team and provides many valuable experience in the research that are beyond the textbooks, patiently corrected my writing and financially supported my research.

I would also like to thank other committee members Dr. S.S. Iyengar, Dr. Jinpeng Wei and Dr. Zhenmin Chen for their encouraging words, thoughtful criticism, and time and attention during busy semesters.

In addition, I thank Larisa Shwartz, who is my mentor during my 2 summer internships in IBM Watson Research Center, and other colleagues in the team for sharing their knowledge, working experience and comments on my research work.

I thank our department staff for assisting me with the administrative tasks necessary for completing my doctoral program: Olga Carbonell, Steven Luis, Luis Rivera, Ivana Rodriguez and Maureen Braham.

Finally I would also like to thank my parents. They are always supporting me and encouraging me with their best wishes.

ABSTRACT OF THE DISSERTATION

EVENT MINING FOR SYSTEM AND SERVICE MANAGEMENT

by

Liang Tang

Florida International University, 2014

Miami, Florida

Professor Tao Li, Major Professor

Modern IT infrastructures are constructed by large scale computing systems and administered by IT service providers. Manually maintaining such large computing systems is costly and inefficient. Service providers often seek automatic or semi-automatic methodologies of detecting and resolving system issues to improve their service quality and efficiency. This dissertation investigates several data-driven approaches for assisting service providers in achieving this goal. The detailed problems studied by these approaches can be categorized into the three aspects in the service workflow: 1) preprocessing raw textual system logs to structural events; 2) refining monitoring configurations for eliminating false positives and false negatives; 3) improving the efficiency of system diagnosis on detected alerts. Solving these problems usually requires a huge amount of domain knowledge about the particular computing systems. The approaches investigated by this dissertation are developed based on event mining algorithms, which are able to automatically derive part of that knowledge from the historical system logs, events and tickets.

In particular, two textual clustering algorithms are developed for converting raw textual logs into system events. For refining the monitoring configuration, a rule based alert prediction algorithm is proposed for eliminating false alerts (false positives) without losing any real alert and a textual classification method is applied to identify the missing alerts (false negatives) from manual incident tickets. For system diagnosis, this dissertation presents an efficient algorithm for discovering the temporal dependencies between system events with corresponding time lags, which can help the administrators to determine the redundancies

of deployed monitoring situations and dependencies of system components. To improve the efficiency of incident ticket resolving, several KNN-based algorithms that recommend relevant historical tickets with resolutions for incoming tickets are investigated. Finally, this dissertation offers a novel algorithm for searching similar textual event segments over large system logs that assists administrators to locate similar system behaviors in the logs. Extensive empirical evaluation on system logs, events and tickets from real IT infrastructures demonstrates the effectiveness and efficiency of the proposed approaches.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION . . . 1
1.1 Motivation . . . 1
1.2 Problem Statement . . . 4
1.3 Contributions . . . 6
1.3.1 System Logs Preprocessing . . . 6
1.3.2 Monitoring Configuration Optimization . . . 7
1.3.3 System Diagnosis . . . 9
1.4 Roadmap . . . 11

2. PRELIMINARY WORK . . . 12
2.1 System Monitoring and Alert Detection . . . 12
2.2 Event Generation From Textual Logs . . . 14
2.3 Temporal Pattern Discovery . . . 15
2.4 Recommending Relevant Tickets and Resolutions . . . 17
2.5 Similarity Search over Textual and Sequential Data . . . 18

3. TEXTUAL LOG PREPROCESSING . . . 21
3.1 Tree Structure Based Clustering . . . 23
3.1.1 Evaluation . . . 27
3.2 Message Signature Based Clustering . . . 37
3.2.1 Comparing with k-means clustering problem . . . 41
3.2.2 An approximated version of problem . . . 44
3.2.3 Local search . . . 45
3.2.4 Connection between Φ and F: . . . 47
3.2.5 Why choose this potential function? . . . 49
3.2.6 Evaluation . . . 50
3.3 Summary . . . 59

4. MONITORING OPTIMIZATION . . . 61
4.1 False Positive and False Negative in IT Service . . . 61
4.2 Eliminating False Positive . . . 63
4.3 Eliminating False Negative . . . 68
4.3.1 Selective Ticket Labeling . . . 69
4.4 Evaluation . . . 71
4.4.1 Evaluation on Historical Data . . . 72
4.4.2 Evaluation on Production Servers . . . 77
4.5 Summary . . . 79

5. SYSTEM DIAGNOSIS 80
5.1 Discovering Temporal Dependencies with Time Lags 80
5.1.1 Algorithms 82
5.1.2 Evaluation 91
5.2 Recommending Incident Resolutions 100
5.2.1 A Basic KNN-based Recommendation 102
5.2.2 Evaluation 110
5.3 Searching Similar Textual Event Segments 120
5.3.1 Suffix Matrix with Random Mask 123
5.3.2 Evaluation 137
5.4 Summary 148

6. CONCLUSION AND FUTURE WORK 149
6.1 Conclusion 149
6.2 Limitation of Proposed Methods and Future work 151
6.2.1 System Event Generation 151
6.2.2 Monitoring Optimization and Resolution Recommendation 152
6.2.3 Temporal Dependency and Lag Discovery 152
6.2.4 Similarity Search over Textual Event Sequence 153

BIBLIOGRAPHY 154

VITA 166

LIST OF FIGURES

FIGURE PAGE

1.1 Overview of Research Problems 5

3.1 Event timeline for the FileZilla log example. 22

3.2 Two status messages in PVFS2. 25

3.3 The Efficiency of K-Medoids on FileZilla logs 32

3.4 The Efficiency of K-Medoids on PVFS2 logs 32

3.5 The Efficiency of K-Medoids on Apache logs 33

3.6 The Scalability of K-Medoids on FileZilla logs 33

3.7 The Scalability of K-Medoids on PVFS2 logs 34

3.8 The Scalability of K-Medoids on Apache logs 34

3.9 Space Cost of LogTree. 36

3.10 A case study of the Apache HTTP server log. 37

3.11 Function $g(r)$, $|C| = 100$ 49

3.12 Vocabulary size 51

3.13 Average Running Time for FileZilla logs 55

3.14 Average Running Time for ThunderBird logs 56

3.15 Average Running Time for Apache logs 56

3.16 Varying parameter λ' 57

3.17 Effectiveness of Potential Function 57

3.18 Scalability of `LogSig` 58

4.1 False Positive Alert Duration 63

4.2 Flowchart for Ticket Creation 64

4.3 Number of Situation Tickets 69

4.4 Flow Chart of Classification Model 71

4.5 Eliminated False Positive Tickets 73

4.6 Postponed Real Tickets 73

4.7 Comparison with Revalidate Method 74
4.8 Accuracy of Situation Discovery for File System Space Alert 75
4.9 Accuracy of Situation Discovery for Disk Space Alert 75
4.10 Accuracy of Situation Discovery for Service not available 76
4.11 Accuracy of Situation Discovery for Router/switch down 76
4.12 Ticket Volume Changes on Account1 77
4.13 Event Volume Changes on Account2 78
5.1 Lag Interval for Temporal Dependency 81
5.2 Sorted Table 85
5.3 Incremental Sorted Table 89
5.4 Runtime on Synthetic Data 94
5.5 Plotting for Account2 Data 95
5.6 Running Time on Account1 Data 96
5.7 Running Time on Account2 Data 97
5.8 Number of Results by Varying χ_c^2 98
5.9 Num. of Results by Varying $minsup$ 99
5.10 Running time by Varying χ_c^2 99
5.11 Running time by Varying $minsup$ 100
5.12 Numbers of Tickets and Distinct Resolutions 101
5.13 Top Repeated Resolutions of Account1 102
5.14 Top Repeated Resolutions of Account2 102
5.15 Top Repeated Resolutions of Account3 103
5.16 Accuracy for $K = 10$, $k = 3$ 111
5.17 Accuracy for Real Tickets and $K = 10$, $k = 3$ 112
5.18 Weighted Accuracy for $K = 10$, $k = 3$ 112
5.19 Accuracy for $K = 20$, $k = 5$ 113

5.20 Accuracy for Real Tickets and $K = 20$, $k = 5$ 113
5.21 Weighted Accuracy for $K = 20$, $k = 5$ 114
5.22 Average Penalty for $K = 10$, $k = 3$ 114
5.23 Average Penalty for $K = 20$, $k = 5$ 115
5.24 Overall Score for $K = 10$, $k = 3$ 115
5.25 Overall Score for $K = 20$, $k = 5$ 116
5.26 Weighted accuracy for account1 by varying k, $K = 10$ 116
5.27 Weighted accuracy for account2 by varying k, $K = 10$ 116
5.28 Weighted accuracy for account3 by varying k, $K = 10$ 117
5.29 Average penalty for account1 by varying k, $K = 10$ 117
5.30 Average penalty for account2 by varying k, $K = 10$ 117
5.31 Average penalty for account3 by varying k, $K = 10$ 118
5.32 Average penalty for account1 by varying k, $K = 10$ 118
5.33 Average penalty for account2 by varying k, $K = 10$ 118
5.34 Average penalty for account3 by varying k, $K = 10$ 119
5.35 An Example of LSH-DOC 124
5.36 An Example of LSH-SEP 125
5.37 An example of $l < |Q|$ 126
5.38 Dissimilar Events in Segments 132
5.39 Random Sequence Mask 133
5.40 Average Search Cost Curve ($n = 100K, |\mathcal{Z}_{H,S}| = 16, \theta = 0.5, |Q| = 10, \delta = 0.8, k = 2$) 137
5.41 RecallRatio comparison for ThunderBird Logs 140
5.42 RecallRatio comparison for Apache Logs 141
5.43 Number of Probed Candidates for ThunderBird Logs 142
5.44 Number of Probed Candidates for Apache Logs 143
5.45 RecallRatio for TG1 143

5.46 Varying m . . . 144

5.47 Varying θ . . . 144

5.48 Varying r . . . 145

5.49 Peak Memory Cost for ThunderBird Logs . . . 145

5.50 Peak Memory Cost for Apache Logs . . . 146

5.51 Indexing Time for ThunderBird Logs . . . 146

5.52 Indexing Time for Apache Logs . . . 147

CHAPTER 1

INTRODUCTION

1.1 Motivation

Large computing systems are often constructed in distributed IT environments and maintained by IT service providers. IT service providers are facing an increasingly intense competitive landscape and growing industry requirements. In their quest to maximize customer satisfaction, service providers seek to employ intelligent solutions, which provide deep analysis, orchestration of business processes and capabilities for optimizing the level of service and cost. Today's competitive business climate, and the complexity of service environments, dictate efficient and cost-effective service delivery and support. This is largely achieved through service-providing facilities to collaborate with system management tools, combined with automation of routine maintenance procedures including problem detection, determination and resolution for the service infrastructure [MSGL09] [TLP+12] [ABD+07] [WE11] [YPZ10]. IT Infrastructure Library (ITIL) addresses monitoring as a continual cycle of monitoring, reporting and subsequent action that provides measurement and control of services [urlg].

Modern forms of distributed computing (say, cloud) provide some standardization of the initial configuration of the hardware and software. However, in order to enable most enterprise level applications, an individual infrastructure for the given application must be created and maintained on behalf of each outsourcing customer. This requirement creates great variability in the services provided by IT support teams. The aforementioned issues contribute largely to the fact that routine maintenance of the information systems remains semi-automated, and manually performed. Significant initiatives like autonomic computing led to awareness of the problem in the scientific and industrial communities and helped to introduce more sophisticated and automated procedures, which increase the productivity

and guarantee the overall quality of the delivered service. Automatic problem detection is typically realized by system monitoring software, such as IBM Tivoli Monitoring [urlf] and HP OpenView [urld]. System monitoring is an automated reactive system that provides an effective and reliable means of ensuring that degradation of the vital signs, defined by acceptable thresholds or monitoring conditions (situations), is flagged as a problem candidate (monitoring event) and sent to the service delivery teams as an incident ticket.

There has been a great deal of effort spent on developing monitoring conditions (situations) that can identify potentially unsafe functioning of the system [HSF06] [RBV03]. However, it is understandably difficult to recognize and quantify influential factors in malfunctioning of a complex system. Therefore classical monitoring tends to rely on periodical probing of a system for conditions which could potentially contribute to the system's misbehavior. Upon detection of the predefined conditions, the monitoring systems trigger events that automatically generate incident tickets. Defining monitoring situations requires the knowledge of a particular system and its relationships with other hardware and software systems. It is a known practice to define conservative conditions in nature thus tending to err on the side of caution. This practice leads to a large number of tickets that require no action (non-actionable or false positive). Continuous updating of IT infrastructures also leads to a number of system alerts that are not captured by system monitoring (false negative). The false negatives eventually cause system faults, such as system crashes and data loss, which are extremely harmful to enterprise users. In system and networking management, many previous studies focus on developing new detection methods for minimizing the number of false negatives [XHF+09b] [OAS08] [LV02] [SOR+03] [BJR12]. In reality, it is not easy to change the internal components of existing monitoring software products, such as IBM Tivoli Monitoring [urlf], which are already deployed in hundreds of thousands of servers. The performance of problem detection also depends on the configurations for those methods. To improve the performance of monitoring systems and the problem analy-

sis, a straightforward solution is to acquire more domain knowledge and expertise to define more precise monitoring configurations and problem scope to inspect. However, there are two limitations of this solution in reality. First, the domain knowledge is usually regarded as the experiences of experts. Different system administrator has different domain knowledge. For instance, an Oracle DBA may not have the knowledge to identify an issue from NAS (Network Attached Storage) devices. The task of gathering domain knowledge from many administrators is time-consuming as well. Second, the domain knowledge about a particular system is likely to change over time. An appropriate monitoring situation may not be appropriate after installing new hardware or software. Re-collecting the domain knowledge takes a long time, so it is difficult to keep the gathered information up-to-date.

When a system alert is detected, performing a detailed analysis for this alert requires a lot of domain knowledge and experience about the particular system. The system administrators usually have to analyze a huge amount of historical system logs and events. The logs and events describe the status of each component and record system internal operations, such as the starting and stopping of services, detection of network connections, software configuration modifications, and execution errors. System administrators utilize the these data to understand the past system behaviors and diagnose the root cause of the alert. Most system logs are raw textual and unstructured. Usually, there are two challenges in analyzing system log data. The first challenge is transforming raw textual logs into system events. The second challenge is to develop efficient algorithms to analyze the hidden relations or patterns among these system events. A lot of studies investigate the second challenge and develop many algorithms to mine system events [PPLW07] [XHF+08] [HMP02] [LLMP05] [GJCH09] [OAS08] [WWLW10] [KT08]. The traditional solution to the first challenge is to develop a specialized log parser for a particular system. However, it requires users to fully understand all kinds of log messages from the system. In prac-

tice, this task is time-consuming, or impossible given the complexity of current computing systems. In addition, specialized log parsers may not work well for other systems.

Data mining is a series of techniques for automatically and efficiently extracting valuable knowledge from historical data. In system and service management, the historical data includes the historical system events, monitoring events and incident tickets. The service providers usually keep track of the historical system events (generated by the production systems), monitoring events (generated by the monitoring system) and incident tickets (edited by humans) to diagnose incoming system issues. The system events and monitoring events describe system internal operations, alerts and faults. The incident tickets reveal the human judgements on these events in terms of system incidents. Automatic or semi-automatic mining the knowledge from those historical events and tickets can efficiently improve the performance of monitoring systems and problem diagnosis.

1.2 Problem Statement

The research problems of this dissertation can be summarized into the following three aspects:

- Data Preprocessing: *How to convert raw textual logs into system events?*

  Most system logs are raw textual and unstructured [ABCM09], but existing data mining techniques for system events focus on structured and discrete events [PPLW07] [XHF+08] [HMP02] [LLMP05] [GJCH09] [OAS08] [WWLW10] [KT08]. To make use of these existing techniques, a data preprocessing is needed for converting them to structured events. However, different system generates various formats of logs, building a log parser for every type of logs is impractical and costly.

- Monitoring Optimization: *How to define better monitoring configurations?*

  The objective is to eliminate the false negatives and false positives of monitoring

without changing existing deployed monitoring systems. This task requires domain knowledge for particular computing systems. Since acquiring the domain knowledge from experts is difficult, it is necessary to come up with an automatic or semi-automatic approach to extract these knowledge from historical events and tickets to achieve this goal. Moreover, this methodologies should be able to be applied to various IT environments.

- System Diagnosis: *How to help administrators perform a detailed diagnosis for detected system issues?*

  Performing a detailed diagnosis for a system issue mainly includes finding the root cause and resolutions. It requires a deep understanding about the target system. In real-world IT infrastructures, many system issues are repeated and the associated resolutions can be found in the relevant events and tickets resolved in the past. Hence, this knowledge can be learnt from the historical data. The approaches utilizing the historical data can help the administrators to narrow down the scope of the potential issues and find the root cause with resolutions more efficiently.

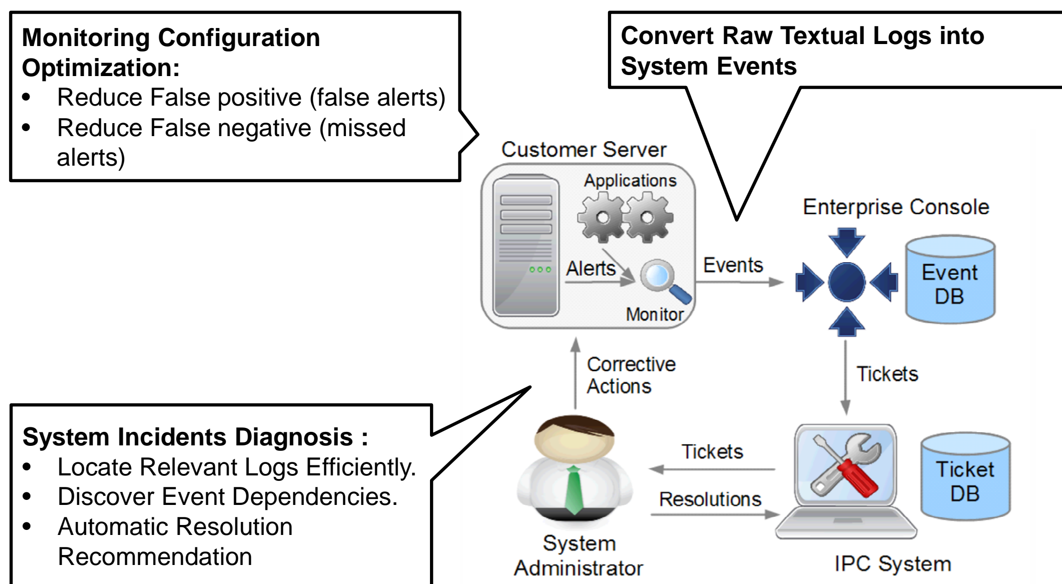

Figure 1.1: Overview of Research Problems

Figure 1.1 summarizes the three problems in the workflow of the system management and IT services. This typical workflow of problem detection, determination and resolution for the IT service provider is prescribed by the ITIL specification [urlg]. Detection is usually provided by monitoring software running on the servers of an enterprise customer, which computes metrics for the hardware and software performance at regular intervals. The metrics are then compared to acceptable thresholds, known as monitoring situations, and any violation results in an alert. If the alert persists beyond a certain delay specified in the situation, the monitor emits an event. Events coming from a customer's entire IT environment are consolidated in an enterprise console. The console uses rule-, case- or knowledge-based engines to analyze the monitoring events and decide whether to open a service ticket in the Incident, Problem, Change (IPC) system. Additional tickets are created upon customer request. The information accumulated in the ticket is used by the System Administrators (SAs) for problem determination and resolution. As part of the service contracts between the customer and the service provider, the SLA (Service Level Agreement) specifies the maximum resolution times for various categories of tickets.

1.3 Contributions

This dissertation investigates the three aforementioned problems and proposes data-driven solutions to improve the quality and efficiency of the current IT service and system management. The contribution of this dissertation can be summarized into following aspects.

1.3.1 System Logs Preprocessing

This dissertation first illustrates the drawbacks of existing techniques for event generation from system logs and then presents two novel textual clustering algorithms, `LogTree` and `LogSig`, which automatically preprocess the raw textual system logs into discrete

system events. Extensive experiments on real system logs show that the two proposed algorithms outperform other alternative clustering algorithms in terms of the accuracy of event generation.

`LogTree` Algorithm

The `LogTree` algorithm is a novel and algorithm-independent framework for event generation from raw textual log messages. `LogTree` first utilizes the format and structural information of logs to create a tree representation of each log message. Then, it computes the similarity using this tree representation in the clustering process, which enhances the clustering accuracy. In addition, an indexing data structure, Message Segment Table is developed in the `LogTree` algorithm to significantly improve the efficiency of the clustering algorithm. This work has been published in the IEEE international conference on Data Mining (ICDM) 2010 [TL10].

`LogSig` Algorithm

The `LogSig` algorithm is a message signature based clustering algorithm. By searching the most representative message signatures, `LogSig` categorizes the textual log messages into several event types. `LogSig` can handle various types of log data and is able to incorporate the domain knowledge provided by experts to achieve a high clustering accuracy. This work has been published in the ACM Conference on Information and Knowledge Management (CIKM) 2011 [TLP11].

1.3.2 Monitoring Configuration Optimization

For system monitoring, this dissertation focuses on the problem of eliminating false alerts (false positives) and missing alerts (false negatives) by refining the configurations of monitoring systems. According to the analysis on large sets of historical monitoring events and

tickets, we reveal several main reasons of triggering false positives and false negatives and then propose our solutions. The proposed solutions avoid changing the existing deployed monitoring systems and are practical for service providers.

Eliminating False Positives

This dissertation describes a novel methodology for minimizing the number of false positives while preserving all alerts which require corrective action. The proposed method defines monitoring conditions and the optimal corresponding delay times based on an off-line analysis of historical monitoring events and corresponding incident tickets. Potential monitoring situations are built on a set of predictive rules that are automatically generated by a rule-based learning algorithm with coverage, confidence and rule complexity criteria. These situations and delay times are propagated as configurations into run-time monitoring systems. The proposed methodology has been assessed by both off-line evaluation with historical data and on-line evaluation with production servers. The evaluation results depict the effectiveness of this method in reducing the number of false positives while retaining all real alerts with the minimal delay. This work has been published in the IEEE/IFIP Network Operations and Management Symposium (NOMS) 2012 [TLP+12] and implemented in the event and ticket analysis portal of the IBM IT service management platform. This work is also filed by IBM Watson Research Center as US patent YOR820110662US1 "Methods and Apparatus for System Monitoring" published in May 9, 2013.

Eliminating False Negatives

This dissertation presents an automatic approach for discovering the false negatives (missing alerts) from incidents tickets that are created by humans. The discovered results help the system administrators correct the misconfigurations and minimize the number of false negatives in future. This approach applies a text classification model for analyzing the

descriptions of incident tickets and identifying the corresponding system issues. The domain knowledge for describing those issues can be incorporated to assist with this model. Experiments are conducted on real system incident tickets from a large enterprise IT infrastructure. The experimental results demonstrate the effectiveness of the proposed approach. This work has been published in the International Conference on Network and Service Management (CNSM) 2013 [TLSG13a].

1.3.3 System Diagnosis

For system diagnosis, this dissertation develops several semi-automatic methods that provide administrators with assists in analyzing large scale system events, logs and tickets. The developed methods aim to solve the following practical problems.

Discovering Temporal Dependencies with Time Lag

This dissertation studies the problem of finding temporal dependency of events with the associated time lags from an event sequence. The temporal dependency among system events (or monitoring events) reveals the dependency of the system components (or correlations of monitoring situations). The time lag is a key feature of the hidden temporal dependencies, which plays an essential role in interpreting the cause of these dependencies. Traditional temporal mining algorithms either use a predefined time window to analyze the event sequence, or employ statistical techniques to simply derive the time dependencies among events. Such paradigms cannot effectively handle varied data with special properties, e.g., the interleaved temporal dependencies. This dissertation first investigates the correlations between the temporal dependency with other temporal patterns, and then proposes a generalized framework to resolve the problem. By utilizing the sorted table in representing time lags among events, the proposed algorithm achieves an elegant balance between the time cost and the space cost. Extensive empirical evaluation on both synthetic and real data

sets demonstrates the efficiency and effectiveness of the proposed algorithm in finding the temporal dependencies with time lags in sequential data. This work has been published and included in the proceedings of the 18th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD) 2012 [TLS12].

Recommending Relevant Incident Ticket with Resolutions

This dissertation introduces a recommendation approach to assist system administrators in resolving the incoming incident tickets that generated by monitoring systems. Those tickets usually are triggered by repeated system issues, therefore, it is practical to employ a recommendation algorithm to recommend relevant tickets with resolutions from historical data. We first present an analysis of the historical incident tickets that are collected from a large service provider, and then propose two recommendation algorithms for this kind of tickets utilizing historical tickets. The proposed algorithms take into account the potential misleading results caused by the tickets of false positives. An additional penalty is incorporated into the algorithms to control the number of misleading resolutions in the recommended results. An extensive empirical evaluation on three ticket data sets demonstrates that our proposed algorithms achieve a high accuracy with a small percentage of misleading results. This work has been published in the IFIP/IEEE International Symposium on Integrated Network Management (IM) 2013 [TLSG13b].

Searching Similar Textual Event Segments

System administrators usually review similar system behaviors to identify the root case of the incoming alert by investigating system logs. Most system logs are textual event sequences, where each event is represented by a log message. Locating similar system behaviors in such logs is equivalent to finding similar segments over the textual event sequence. Similarity search has been widely studied for symbolic and time series data in which each

data object is a symbolic or numeric value. However, efficiently searching similar segments over textual sequences is a novel problem and not fully studied. Existing search indexing for textual data only focuses on unordered data. Substring matching methods are able to efficiently find matched segments over a sequence, but their sequences are single-valued rather than text. This dissertation presents a novel indexing method, suffix matrix, for efficiently searching similar segments over textual event sequences. It provides an integration of two disparate techniques: locality-sensitive hashing and suffix arrays. This method also supports the k-dissimilar segment search. A k-dissimilar segment is a segment that has at most k dissimilar events to the query sequence. By using *random sequence mask* proposed in this work, this method can have a high probability to reach all k-dissimilar segments without increasing much search cost. We conduct experiments on real system log data and the experimental results show that the proposed method outperforms alternative methods using existing techniques. This work has been published in the ACM Conference on Information and Knowledge Management (CIKM) 2013 [TLCZ13].

1.4 Roadmap

The rest of the dissertation is organized as follows: Chapter 2 provides a brief introduction of the preliminary work for event mining algorithms and the system and IT service management. Chapter 3 presents the problem statement and proposed algorithms for the textual log preprocessing problem. Chapter 4 first briefly introduces the background of the IT service management with the false negative and false positive issues, and then discusses the proposed data-driven approaches for solving the two issues. Chapter 5 describes three practical problems in system diagnosis with the proposed solutions: 1) discovering the lags of temporal dependencies, 2) recommending relevant tickets with solutions for incoming tickets, 3) efficient similarity searching over textual event sequences. Chapter 6 concludes this dissertation and discusses the future work.

CHAPTER 2

PRELIMINARY WORK

This chapter summarizes the preliminary work of the techniques presented in this dissertation. Generally, the preliminary work involves several research areas in computer science, which includes: system monitoring with alert detection, temporal pattern discovery, recommendation system and similarity search.

2.1 System Monitoring and Alert Detection

System monitoring, as part of the automated service management, has become a significant research area of the IT industry in the past few years. Commercial products such as IBM Tivoli [urle], HP OpenView [urld] and Splunk [urlk] provide system monitoring. Numerous studies [KRRS08] [ADNR07] [MJ93] [XZB05] [ESV03] [RLS+97] focus on monitoring that is critical for a distributed network. The monitoring targets include the components or subsystems of IT infrastructures, such as the hardware of the system (CPU, hard disk) or the software (a database engine, a web server). Once certain system alarms are captured, the system monitoring software will generate the event tickets into the ticketing system. Automated ticket resolution is much harder than automated system monitoring because it requires vast domain knowledge about the target infrastructure. Some prior studies apply approaches in text mining to explore the related ticket resolutions from the ticketing database [SCT+08, WLZG11]. Other works propose methods for refining the work order of resolving tickets [SCT+08, MMY+10]. A number of studies focused on the analysis of historical events with the goal of improving the understanding of system behaviors. A significant amount of work was done on analysis of system log files and monitoring events. Another area of interest is the identification of actionable patterns of events and misses, or false negatives, by the monitoring system. False negatives are indications of a problem in

the monitoring software configuration, wherein a faulty state of the system does not cause monitoring alerts.

Network monitoring is used to check the "health" of communications by inspecting data transmission flow, sniffing data packets, analyzing bandwidth, etc. [KRRS08] [ADNR07] [MJ93] [XZB05] [ESV03] [RLS$^+$97]. It is able to detect node failures, network intrusions, or other abnormal situations in the distributed system. The main difference between the network monitoring and framework we consider is the monitored target, which can be any component or subsystem of the system, hardware (such as CPU, hard disk) or software (such as a database engine, or web server). Only the system administrators, who are working the monitored server, can determine whether an alert is real or false. This is why we incorporate ticket resolutions, which record how system administrators resolve those alerts using our solution.

A significant amount of work in data mining has been done to identify actionable patterns of events. See example, [HMP02], [PPLW07], [KT08], [TLS12]. Different types of patterns, such as (partially) periodic patterns, event bursts, and mutually dependent patterns were introduced to describe system management events. Efficient algorithms were developed to find and interpret such patterns. Our work is based on the part of an event processing workflow that takes into account the human processing of the tickets. This allowed us to identify non-actionable patterns and misses of the monitoring system configuration with significant precision. In the event processing workflow, false positive events are transformed into false positive tickets. Identification of false positive events makes it possible to significantly reduce the number of false positive tickets. The translation of the actionable patterns to enterprise software rules is considered in [GJCH09] and [PTG$^+$03].

Dealing with false negatives, or misses of the system alerts, usually includes the consideration of additional source of data. In our case, this additional source is ticketing data. As a source of information, it is difficult data to process, because there are no supporting

standards or structure, and ticketing records are usually byproducts of the system administrator's work, which are mainly incomplete and unfinished. An additional difficulty is that false negatives are rare and unbalanced due to the fact that historically tested and tuned configurations of the monitoring systems are used. Methods of dealing with unbalanced data was considered for example in [CBHK02].

2.2 Event Generation From Textual Logs

One challenge of performing automated analysis of system logs is transforming the logs into a collection of system events. The number of distinct events observed can be very large and also grow rapidly due to the large vocabulary size as well as various parameters in log generation [ABCM09]. In addition, variability in log languages creates difficulty in deciphering events and errors reported by multiple products and components [Ste04]. Once the log data has been transformed into the canonical form, the second challenge is the design of efficient algorithms for analyzing log patterns from the events. Recently, there has been lots of research on using data mining and machine learning techniques for analyzing system logs and most of them address the second challenge [PPLW07] [XHF+08] [HMP02] [LLMP05] [GJCH09]. They focus on analyzing log patterns from events for problem determination such as discovering temporal patterns of system events, predicting and characterizing system behaviors, and performing system performance debugging. Most of these works generally assume the log data has been converted into events and ignore the complexities and difficulties in transforming the raw logs into a collection of events.

It has been shown in [Ste04] that log messages are relatively short text messages but could have a large vocabulary size. This characteristic often leads to a poor performance when using the bag-of-words model in text mining on log data. The reason is that, each single log message has only a few terms, but the vocabulary size is very large. Hence, the vector space established on sets of terms would be very sparse.

Recent studies [ABCM09] [MZHM09] apply data clustering techniques to automatically partition log messages into different groups. Each message group represents a particular type of events. Due to the short length and large vocabulary size of log messages [Ste04], traditional data clustering methods based on the *bag-of-word* model cannot perform well when applied to the log message data. Therefore, new clustering methods have been introduced to utilize both the format and the structure information of log data [ABCM09] [MZHM09]. However, these methods only work well for strictly formatted/structured logs and their performances heavily rely on the format/structure features of the log messages.

2.3 Temporal Pattern Discovery

System and monitoring events are stored as temporal sequences. Understanding the temporal dependencies of these events helps to discover the relationships among the system components and find out the root cause of the system alerts. In temporal data mining, the input data is typically a sequence of discrete items associated with time stamps [Mör06] [Mit10]. Let A and B be two types of items, a temporal dependency for A and B, written as $A \to B$, denotes that the occurrence of B depends on the occurrence of A. The dependency indicates that an item A is often followed by an item B. Let $[t_1, t_2]$ be the range of the lag for the dependent A and B. This temporal dependency with $[t_1, t_2]$ is written as $A \to_{[t_1,t_2]} B$ [GKK+09].

Previous work of temporal dependency discovery can be categorized by the data set type. The first category is for market basket data, which is a collection of transactions [TSK05] where each transaction is a sequence of items. The purpose of this type of temporal dependency discovery is to find frequent subsequences which are contained by a certain amount of transactions. Typical algorithms are GSP [SA96b], FreeSpan[HPMA+00], PrefixSpan[PHMA+01], and SPAM[AFGY02]. The second category is for the time series data. A temporal dependency of this category is seen as a correlation on multiple time

series variables [ZS06] [Dhu10], which determines whether one time series is useful in forecasting another. Our work belongs to the third category, which is for temporal symbolic sequences. The input data is an item sequence and each item is associated with a time stamp. An item may represent an event or a behavior in history [LDH+10] [Mör06] [Mit10] [MF10][LLMP05]. The purpose is to find various temporal relationships among these events or behaviors. Many temporal patterns proposed in previous work can be considered as special cases of temporal dependencies with different lag intervals.

Table 2.1: Relation with Other Temporal Patterns

Temporal Pattern	**An Example**	**Equivalent Temporal Dependency with Lag Interval**
Mutually dependent pattern [MH01a]	$\{A, B\}$	$A \rightarrow_{[0,\delta]} B$, $B \rightarrow_{[0,\delta]} A$
Partially periodic pattern [MH01b]	A with periodic p and a given time tolerance δ	$A \rightarrow_{[p-\delta,p+\delta]} A$
Frequent episode pattern [MTV97]	$A \rightarrow B \rightarrow C$ with a given time window p	$A \rightarrow_{[0,p]} B$, $B \rightarrow_{[0,p]} C$
Loose temporal pattern [LM04]	B follows by A before time t	$A \rightarrow_{[0,t]} B$
Stringent temporal pattern [LM04]	B follows by A about time t with a given time tolerance δ	$A \rightarrow_{[t-\delta,t+\delta]} B$

Table 2.1 lists several types of temporal patterns proposed in the literature and their corresponding temporal dependencies with lag intervals. A mutually dependent pattern (**m-pattern**) $\{A, B\}$, can be described as two temporal dependencies $A \rightarrow_{[0,\delta]} B$ and $B \rightarrow_{[0,\delta]} A$. Items of A and B in an **m-pattern** appear almost together so that $t_1 = 0, t_2 \leq \delta$, where δ is the time tolerance. A partially periodic pattern (**p-pattern**) [MH01b] for a single item A, can be expressed as a temporal dependency $A \rightarrow_{[p-\delta,p+\delta]} A$, where p is the period. Frequent episodes $A \rightarrow B \rightarrow C$ can be separated to $A \rightarrow_{[0,p]} B$ and $B \rightarrow_{[0,p]} C$ where p is the parameter of the time window length [MTV97]. [LM04] proposes *loose temporal pattern* and *stringent temporal pattern*. As shown in Table 2.1, the two types of temporal patterns can be explained by two temporal dependencies with particular constraints on the

lag intervals. One common problem of these algorithms is how to set the precise parameter about the time window [MTV97] [MH01b] [BO07]. For example, for discovering partially periodic patterns, if δ is too small, the identification of partially periodic patterns would be too strict and no result can be found; if the δ is too large, many false results would be found. [LSU07] [LSU05] [MR04] directly find frequent episodes according to the occurrences of episodes in the data sequence. The discovered frequent episode may not have fixed lag intervals for the represented temporal dependency. The method proposed in this dissertation does not require users to specify the parameters about the time window and is able to discover interleaved temporal dependencies.

2.4 Recommending Relevant Tickets and Resolutions

One major cost of modern IT service is the manpower. In large service providers, the service centers are constituted by hundred or thousands of IT experts, who take charge of various incident tickets every day. Therefore, service providers heavily rely on the human efficiency for such task as root cause analysis and incident ticket resolving. Automatic techniques of recommending relevant historical tickets with resolutions can significantly improve the efficiency of humans in this task. Based on the relevant tickets, the human can correlate related system problems happening before and perform a deeper system diagnosis. The solutions described in relevant historical tickets also provide best practices for solving similar issues.

Recommendation technique has also been widely studied in e-commerce and online advertising areas. With the development of e-commerce and online advertising, a substantial amount of research has been devoted to the recommendation system. The existing recommendation algorithms can be categorized into two types. The first type is learning based recommendation, in which the algorithm aims to maximize the rate of user response, such as the user click or conversation. The recommendation problem is then naturally formulat-

ed as a prediction problem. It utilizes a prediction algorithm to compute the probability of the user response on each item. Then, it recommends the one having the largest probability. Most prediction algorithms can be utilized in the recommendation, such as naive bayes classification, linear regression, logistic regression and matrix factorization [MS99, Bis06].

The second type of recommendation algorithm focuses on the relevance of items or users, rather than the user response. Lots of algorithms proposed for promoting products to online users [BK07, DL05, Kor09, LMX11] belong to this type. They can be categorized as item-based [SKKR00, Kar01, NK11] and user-based algorithms [TH01, Kor09, BK07, DL05]. Our work in this dissertation is the item-based. Every ticket is regarded as an item in our scenario. The difference between our work and traditional item-based algorithms is that, in e-commerce, products are maintained by reliable sellers, or there is another procedure to assure the quality of selling products. The recommendation algorithms usually do not need to consider the problem of fake or low-quality products. But in service management, false tickets are unavoidable. The tickets with ticket resolutions are recorded in the database of the ticketing system. In some real-world ticketing systems, the false ticket is the majority of all tickets. Moreover, when a ticket arrives, the recommendation algorithm does not know this alert is real or false in advance. The traditional recommendation algorithms do not take into account the types of tickets and as a result would recommend misleading resolutions.

2.5 Similarity Search over Textual and Sequential Data

The similarity search problem in low-dimensional data spaces has been studied extensively. A number of tree structure based algorithms are devised to support the similarity queries and nearest neighbors queries, such as R-Tree [Gut84], KD-Tree [Ben90] and SR-Tree [KS97]. These previous algorithms are known to work well in low-dimensional data spaces. But for high-dimensional data spaces, their search time cost or indexing space cost

grows to an exponential number of the dimensionality. In textual information retrieval and image processing domains, the descriptor of a data object is usually a high-dimensional vector. Hence, those tree structured based algorithms are not appropriate in these domains. Locality-Sensitive Hashing (LSH) is a randomized approximate algorithm for the similar search in high-dimensional data space [GIM99, AI06]. It is applicable for high dimensional data and has been successfully used in image data or textual data. Min-Hash is a widely used hash function for textual data [BCFM98], which can quickly estimate the $sim(x, y)$ of x and y. In natural language processing, a w-shingling is a set of unique contiguous subsequences of words/terms in a document. The similarity function $sim(x, y)$ is usually chosen as the Jaccard similarity over the w-shinglings of x and y.

Substring search in sequential data has been studied for years. Suffix tree and suffix array are two typical methods for on-line searching matched substrings over a sequence [Wei73, MM93]. By using a binary search over the suffix array, the method can find matched substring in $O(\log n)$, where n is the length of the string. Compressed suffix arrays and BWT-based compressed full-text indices make further efforts to reduce the search time and space cost based on suffix arrays [GV05, BW94]. Time series data is real-valued sequence data. A lot of efficient similarity search methods are proposed and studied for time series data [Pop02, LC08]. But their target is a set of data points, rather than a set of segments of the sequence. Moreover, each data point in time series is a real-valued vector, not a textual message or document.

In system management, log and system event analysis is a fundamental method to maintain, diagnose and optimize large production systems [XHF+08, XHF+09a, TLS12, TLP+12, TLSG13b, TLP+13]. Log event search as a basic functionality is embedded in many system management, log analysis and system monitoring platforms [urle, urlk, urlh]. Users can input relational query conditions or a set of keywords to query related system event logs in history. This kind of log search has no difference with a traditional database

query or a keywords search. Their search target is a single event, not continuous subsequence or segements of events.

CHAPTER 3

TEXTUAL LOG PREPROCESSING

A lot of studies investigate the system event mining and develop many algorithms for discovering the abnormal system behaviors and relationships of events/system components [PPLW07] [XHF^{+}08] [HMP02] [LLMP05] [GJCH09] [OAS08] [WWLW10] [KT08]. In those studies, the event data is a collection of discrete items or structured events, rather than textual log messages. However, most of the computing system only generate the textual logs for human to view. Since the large volume of the logs in production systems, it is difficult for human to inspect those large amount of log data. The research objective of the problem is to develop a method for converting the raw textual system logs into discrete events, such that the existing event mining algorithms can be applied to do automatic analysis on the log data.

A straightforward solution is to develop a specialized log parser for a particular system. However, it requires users fully understand all kinds of log messages from the system. In practice, this is time-consuming or not impossible given the complexity of current computing systems. In addition, a specialized log parser is not universal and does not work well for other types of systems.

Table 3.1 shows an example of the SFTP [1] log collected from FileZilla [urlb]. In order to analyze the behaviors, the raw log messages need to be translated to several types of events. Figure 3.1 shows the corresponding event timeline created by the log messages. The event timeline provides a convenient platform for people to understand log behaviors and to discover log patterns.

Recent studies for converting the raw textual logs into system events are discussed in Section 2.2. These studies apply data clustering techniques to automatically partition log messages into different groups, where each message group represents a particular type of

[1] SFTP: Simple File Transfer Procotol

Table 3.1: An Example of FileZilla's log.

No.	Message
s_1	2010-05-02 00:21:39 Command: put "E:/Tomcat/apps/index.html" "/disk/...
s_2	2010-05-02 00:21:40 Status: File transfer successful, transferred 823 bytes...
s_3	2010-05-02 00:21:41 Command: cd "/disk/storage006/users/lt...
s_4	2010-05-02 00:21:42 Command: cd "/disk/storage006/users/lt...
s_5	2010-05-02 00:21:42 Command: cd "/disk/storage006/users/lt...
s_6	2010-05-02 00:21:42 Command: put "E:/Tomcat/apps/record1.html" "/disk/...
s_7	2010-05-02 00:21:42 Status: Listing directory /disk/storage006/users/lt...
s_8	2010-05-02 00:21:42 Status: File transfer successful, transferred 1,232 bytes...
s_9	2010-05-02 00:21:42 Command: put "E:/Tomcat/apps/record2.html" "/disk/...
s_{10}	2010-05-02 00:21:42 Response: New directory is: "/disk/storage006/users/lt...
s_{11}	2010-05-02 00:21:42 Command: mkdir "libraries"
s_{12}	2010-05-02 00:21:42 Error: Directory /disk/storage006/users/lt...
s_{13}	2010-05-02 00:21:44 Status: Retrieving directory listing...
s_{14}	2010-05-02 00:21:44 Command: ls
s_{15}	2010-05-02 00:21:45 Command: cd "/disk/storage006/users/lt...
...	... ...

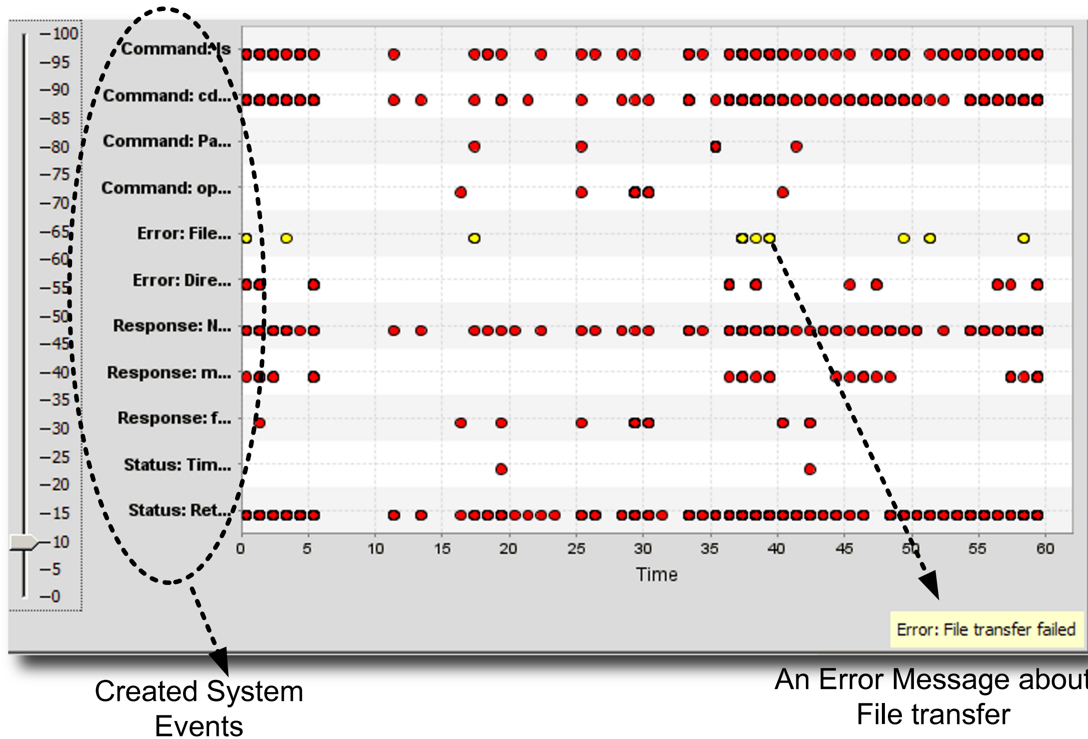

Figure 3.1: Event timeline for the FileZilla log example.

events. They only work well for strictly formatted/structured logs and their performances heavily rely on the format/structure features of the log messages. In this chapter, we present two novel clustering approaches for system event generation.

3.1 Tree Structure Based Clustering

The first proposed approach is a tree structure based clustering algorithm, `LogTree`, which computes the similarity of log messages based on the established tree representation in the clustering process.

Formally, a series of system log is a set of messages $S = \{s_1, s_2, \cdots, s_n\}$, where s_i is a log message, $i = 1, 2, \cdots, n$, and n is the number of log messages. The length of S is denoted by $|S|$, i.e., $n = |S|$. The objective of the event creation is to find a representative set of message S^*, to express the information of S as much as possible, where $|S^*| = k \leq |S|$, each message of S^* represents one type of event, and k is a user-defined parameter. The intuition is illustrated in the following Example.

Example 1. *Table 3.1 shows a set of 15 log messages generated by the FileZilla client. It mainly consists of 6 types of messages, which include 4 different commands (e.g., "put", "cd", "mkdir", and "ls"), responses, and errors. Therefore, the representative set S^* could be created to be $\{s_1, s_2, s_3, s_7, s_{11}, s_{14}\}$, where every type of the command, response and error is covered by S^*, and $k = 6$.*

We hope the created events to cover the original log as much as possible. The quality of S^* can be measured by the *event coverage*.

Definition 3.1.1. *Given two sets of log messages S^* and S, $|S^*| \leq |S|$, the* event coverage *of S^* with respect to S is $J_C(S^*, S)$, which can be computed as follows:*

$$J_C(S^*, S) = \sum_{x \in S} \max_{x^* \in S^*} F_C(x^*, x),$$

where $F_C(x^*, x)$ is the similarity function of the log message x^* and the log message x.

Given a series of system log S with a user-defined parameter $0 \leq k \leq |S|$, the goal is to find a representative set $S^* \subseteq S$, which satisfies:

$$\begin{aligned} & \max J_C(S^*, S), \\ \text{subject to} \quad & |S^*| = k. \end{aligned}$$

Clearly, the system event generation can be regarded as a text clustering problem [SM84] where an event is the centroid or medoid of one cluster. However, those traditional text clustering methods are not appropriate for system logs. We show that those methods, which only extract the information at the word level, cannot produce an acceptable accuracy of the clustering of system logs.

It has been shown in [Ste04] that log messages are relatively short text messages but have large vocabulary size. As a result, two messages of the same event type share very few common words. It is possible two messages of the same type have two totally different sets of words. The following is an example of two messages from the PVFS2 log file [urlj]. The two messages are status messages. Both of them belong to the same event type ***status*** which prints out the current status of the PVFS2 internal engine.

$$\text{bytes read}: \ 0 \ \ 0 \ \ 0 \ \ 0 \ \ 0 \ \ 0$$
$$\text{metadata keyval ops}: \ 1 \ \ 1 \ \ 1 \ \ 1 \ \ 1 \ \ 1$$

Note that the two messages have no words in common and clustering analysis purely based on the word level information would not reveal any similarity between the two messages. The similarity scores between the two messages (the cosine similarity [SM84], the Jaccard similarity [TSK05] or the words matching similarity [ABCM09]) are 0. Although there is no common words between the two messages, the structure and format information implicitly suggest that the two messages could belong to a same category as shown in Figure 3.2. The intuition is straightforward: two messages are both split by the ':'; the left parts are

both English words, and the right parts are 6 numbers separated by a tab. Actually, people often guess the types of messages from the structure and format information as well. In

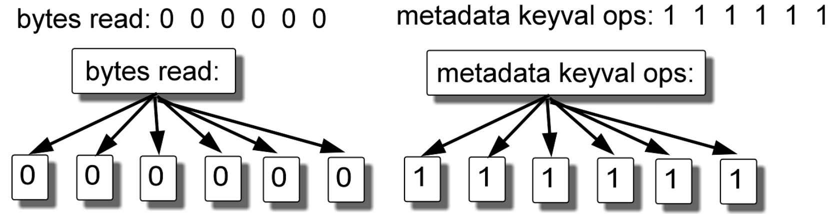

Figure 3.2: Two status messages in PVFS2.

real system applications, the structure of log messages often implies critical information. The same type of messages are usually assembled by the same template, so the structure of log messages indicates which internal component generates this log message. Therefore, we should consider the structure information of each log message instead of just treating it as a sentence. Furthermore, two additional information should be considered as well:

- *symbols* The symbols, such as ':', '[', are important to identify the templates of the log message. They should be utilized in computing the similarity of two log message segments.
- *word/term categories* If two log messages are generated by the same template, even if they have different sets of words/terms, the categories of words should be similar. In our system, there are six categories $\boldsymbol{T} = \{$ *word, number, symbol, date, IP, comment* $\}$. Given a term w in a message segment m_1, $t(w)$ denotes the category of the w. $t(w) \in \boldsymbol{T}$.

Based on this intuition, the similarity function of the log messages F_C can be defined as follows:

Definition 3.1.2. *Given two log messages s_1 and s_2, let $T_1 = \{V_1, E_1, L, r_1, P\}$ and $T_2 = \{V_2, E_2, L, r_2, P\}$ be the corresponding semi-structural log messages of s_1 and s_2*

respectively, the coverage function $F_C(s_1, s_2)$ is computed as follows:

$$F_C(s_1, s_2) = \frac{F'_C(r_1, r_2, \lambda) + F'_C(r_2, r_1, \lambda)}{2},$$

where

$$F'_C(v_1, v_2, w) = w \cdot d(L(v_1), L(v_2)) + \sum_{(v,u) \in M^*_C(v_1, v_2)} F'_C(v, u, w \cdot \lambda),$$

$M^*_C(v_1, v_2)$ is the best matching between v_1's children and v_2's children, and λ is a parameter, $0 \leq \lambda \leq 1$.

Note that the function F_C is obtained by another recursive function F'_C. F'_C computes the similarity of two subtrees rooted at two given nodes v_1 and v_2 respectively. To compare the two subtrees, besides the root nodes v_1 and v_2, F'_C needs to consider the similarity of their children as well. Then, there is a problem that which child of v_1 should be compared with which child of v_2. In other words, we have to find the best matching $M^*_C(v_1, v_2)$ in computing F'_C. Finding the best matching is actually a maximal weighted bipartite matching problem. In the implementation, we can use a simple greedy strategy to find the matching. For each child of v_1, we assign it to the maximal matched node in unassigned children of v_2. This time complexity of the greedy approach is $O(n_1 n_2)$ where n_1 and n_2 are the numbers of children of v_1 and v_2, respectively. F'_C requires another parameter w, which is a decay factor. In order to improve the importance of higher level nodes, this decay factor is used to decrease the contribution of similarities at a lower level. Since $\lambda \leq 1$, the decay factor w decreases along with the recursion depth.

3.1.1 Evaluation

This section presents two evaluations for the tree structure based clustering method on several real data sets.

Experimental Platforms

Our system is developed in Java 1.5 Platform. Table 3.2 shows the summary of two machines where we run our experiments. All experiments except for scalability test are conducted in Machine1, which is a 32-bits machine. As for the scalability experiment, the program needs over 2G main memory, so the scalability experiment is conducted in Machine2, which is a 64-bits machine. All the experimental programs are single-threaded.

Table 3.2: Experimental Machines

Machine	OS	CPU	Memory	JVM Heap Size
Machine1	Windows 7	Intel Core i5 @2.53GHz	4G	1.5G
Machine2	Linux 2.6.18	Intel Xeon(R) X5460@3.16GHz	32G	3G

Data Collection

In order to evaluate our work, we collect the log data from 4 different and popular real systems. Table 3.3 shows the summary of our collected log data. The log data is collected from the server machines/systems in the computer lab of a research center. Those systems are very common system services installed in the many data centers.

- FileZilla client 3.3[urlb] log, which records the client's operations and responses from the FTP/SFTP server.
- MySQL 5.1.31[urli] error log. The MySQL database is hosted in a developer machine, which consists of the error messages from the MySQL database engine.

- PVFS2 server 2.8.2[urlj] log. It contains errors, internal operations, status information of one virtual file sever.
- Apache HTTP Server 2.x[urla] error log. It is obtained from the hosts for the center website. The error log mainly records various bad HTTP requests with corresponding client information.

Table 3.3: Log data summary.

System	System Type	#Messages	#Words per message	#Types
FileZilla	SFTP/FTP Client	22,421	7 to 15	4
MySQL	Database Engine	270	8 to 35	4
PVFS2	Parallel File System	95,496	2 to 20	4
Apache	Web Server	236,055	10 to 20	5

Comparative Methods

In order to evaluate the effectiveness and efficiency of our work, we use 4 other related and traditional methods in the experiments. Table 3.4 shows all the comparative methods used in the experiments. As for "Tree Kernel", the tree structure is the same as that used in the our method *LogTree*. Since the tree node of the log message is not labeled, we can only choose sparse tree kernel for "Tree Kernel" [CS04]. The experiments of the event generation are conducted using two clustering algorithms, K-Medoids [HKP05] and Single-Linkage [TSK05]. The reason that we choose the two algorithms is that K-Medoids is the basic and classical algorithm for data clustering, and Single-Linkage is a typical hierarchical clustering which is actually used in our system. It should be pointed out that our comparisons are focus on similarity measurements which are independent from a specific clustering algorithm. We expect that the insights gained from our experiment comparisons can be generalized to other clustering algorithms as well.

Table 3.4: Summary of comparative methods.

Method	Description
“TF-IDF”	the classical text clustering method using the vector space model with tf-idf transformation.
“Tree Kernel”	the tree kernel similarity introduced in [CS04].
‘Matching”	the method using words matching similarity in [ABCM09].
“LogTree”	our method using semi-structural log and Message Segment Table.
“Jaccard”	Jaccard Index similarity of two log messages.

The Quality of Events Generation

The entire log is split into different time frames. Each time frame is composed of 2000 log messages and labeled with the frame number. For example, Apache2 denotes the 2th frame of the Apache log. The quality of the results is evaluated by the F-measure (F1-score) [SM84]. First, the log messages are manually classified into several types. Then, the cluster label for each log message is obtained by the clustering algorithm. The F-measure score is then computed from message types and clustered labels. Table 3.5 and Table 3.6 show the F-measure scores of K-Medoids and Single-Linkage clusterings with different similarity approaches respectively. Since the result of K-Medoids algorithm varies by initial choice of seeds, we run 5 times for each K-Medoids clustering and the entries in Table 3.5 are computed by averaging the 5 runs.

Only “Tree Kernel” and “LogTree” need to set parameters. “Tree Kernel” has only one parameter, λ_s, to penalize matching subsequences of nodes [CS04]. We run it under different parameter settings, and select the best result for comparison. Another parameter k is the number of clusters for clustering algorithm, which is equal to the number of the types of log messages. Table 3.7 shows the parameters used for “Tree Kernel” and “LogTree”.

FileZilla log consists of 4 types of log messages. One observation is that, the root node of the semi-structural log is sufficient to discriminate the type of a message. Mean-

Table 3.5: F-Measures of K-Medoids

Logs	TF-IDF	Tree Kernel	Matching	LogTree	Jaccard
FileZilla1	0.8461	**1.0**	0.6065	**1.0**	0.6550
FileZilla2	0.8068	**1.0**	0.5831	**1.0**	0.5936
FileZilla3	0.6180	**1.0**	0.8994	**1.0**	0.5289
FileZilla4	0.6838	0.9327	**0.9545**	0.9353	0.7580
PVFS1	0.6304	0.7346	0.7473	**0.8628**	0.6434
PVFS2	0.5909	0.6753	**0.7495**	0.6753	0.6667
PVFS3	0.5927	0.5255	0.5938	**0.7973**	0.5145
PVFS4	0.4527	0.5272	0.5680	**0.8508**	0.5386
MySQL	0.4927	0.8197	**0.8222**	**0.8222**	0.5138
Apache1	0.7305	0.7393	0.9706	**0.9956**	0.7478
Apache2	0.6435	0.7735	0.9401	**0.9743**	0.7529
Apache3	0.9042	0.7652	0.7006	**0.9980**	0.8490
Apache4	0.4564	0.8348	0.7292	**0.9950**	0.6460
Apache5	0.4451	0.7051	0.5757	**0.9828**	0.6997

Table 3.6: F-Measures of Single-Linkage

Logs	TF-IDF	Tree Kernel	Matching	LogTree	Jaccard
FileZilla1	0.6842	**0.9994**	0.8848	0.9271	0.6707
FileZilla2	0.5059	0.8423	0.7911	**0.9951**	0.5173
FileZilla3	0.5613	**0.9972**	0.4720	0.9832	0.5514
FileZilla4	0.8670	**0.9966**	0.9913	0.9943	0.6996
PVFS1	0.7336	0.9652	0.6764	**0.9867**	0.4883
PVFS2	0.8180	**0.8190**	0.7644	0.8184	0.6667
PVFS3	0.7149	0.7891	0.7140	**0.9188**	0.5157
PVFS4	0.7198	0.7522	0.6827	**0.8136**	0.6345
MySQL	0.4859	0.6189	**0.8705**	0.8450	0.5138
Apache1	0.7501	0.9148	0.7628	**0.9248**	0.7473
Apache2	0.7515	**0.9503**	0.8178	0.9414	0.7529
Apache3	0.8475	0.8644	0.9294	**0.9594**	0.8485
Apache4	0.9552	0.9152	0.9501	**0.9613**	0.6460
Apache5	0.7882	0.9419	0.8534	**0.9568**	0.6997

Table 3.7: Parameter settings

Log Type	k	λ_s	λ	α
FileZilla	4	0.8	0.7	0.1
MySQL	4	0.8	0.3	0.1
PVFS2	4	0.8	0.7	0.1
Apache	5	0.8	0.01	0.1

while, the root node produces the largest contribution in the similarity in “Tree Kernel” and “LogTree”. So the two methods benefit from the structural information to achieve a high clustering performance.

PVFS2 log records various kinds of status messages, errors and internal operations. None of the methods can perform perfectly. The reason is that, in some cases, two log

messages composed of distinct sets of words could belong to one type. Thus, it is difficult to cluster this kind of messages into one cluster.

MySQL error log is small, but some messages are very long. Those messages are all assembled by fixed templates. The parameter part is very short comparing with the total length of the template, so the similarity of [ABCM09] based on the templates wouldn't be interfered by the parameter parts very much. Therefore, "Matching" always achieves the highest performance.

Apache error log is very similar to FillZilla log. But it contains more useless components to identify the types of the error message, such as the client information. In our semi-structural log, those useless components are located at low level nodes. Therefore, when the parameter λ becomes small, their contributions to the similarity are reduced, then the overall performance becomes better.

To sum up, the "Tree Kernel" and "LogTree" methods outperform other methods. The main reason is that, the two methods capture both the word level information as well as the structural and format information of the log messages. In the next subsection, we show that our "LogTree" is more efficient than "Tree Kernel".

The Efficiency of Event Generation

We records the running time of each clustering algorithm on the log data. Due to the space limitation, we only show the running time of K-Medoids algorithm on FileZilla log, PVFS2 log, and Apache error log in Figure 3.3, 3.4 and 3.5. The running time is the average number of 5 runs. In the implementation, we build the similarity matrix of each pair of log messages at the beginning, whose time complexity is $O(N^2)$ where N is the number of samples. Thus, the majority of the running time is used for building the similarity matrix. As for "LogTree", the threshold of Message Segment Table is $f_{min} = 0.00001$.

The parameter choice depends on the size of the main memory. **Note that the running time of LogTree includes the time for building MST.**

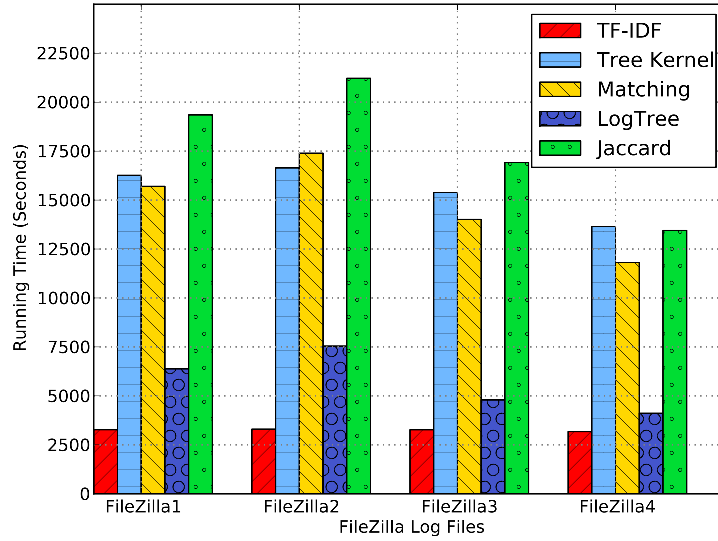

Figure 3.3: The Efficiency of K-Medoids on FileZilla logs

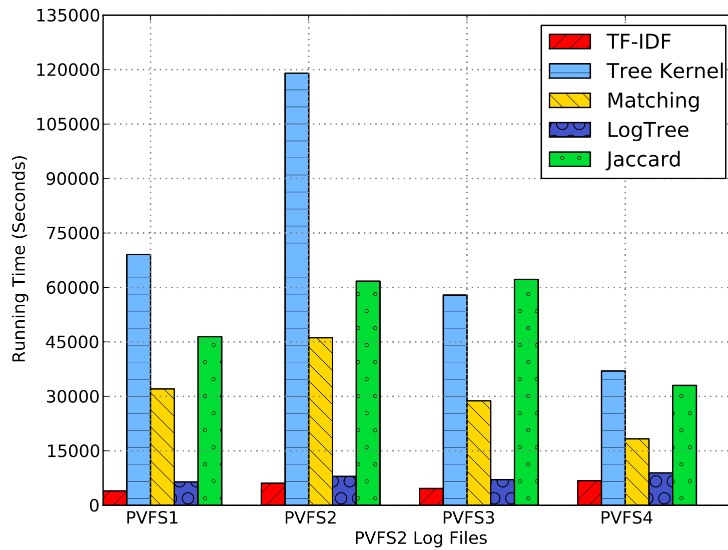

Figure 3.4: The Efficiency of K-Medoids on PVFS2 logs

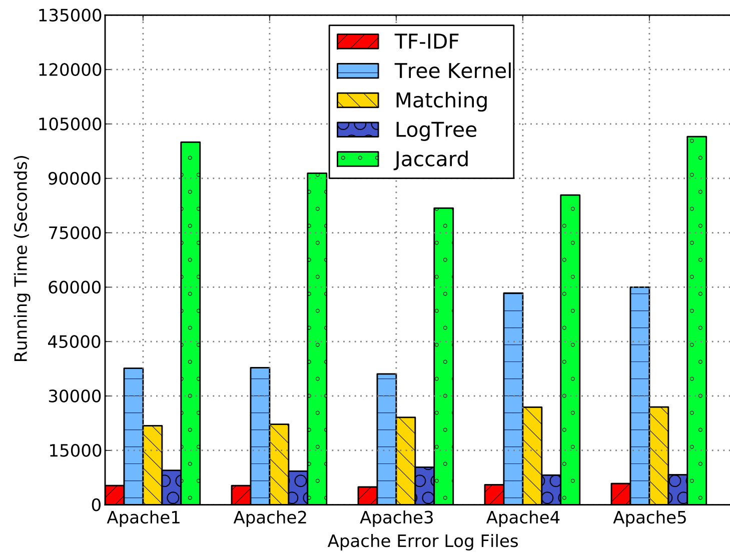

Figure 3.5: The Efficiency of K-Medoids on Apache logs

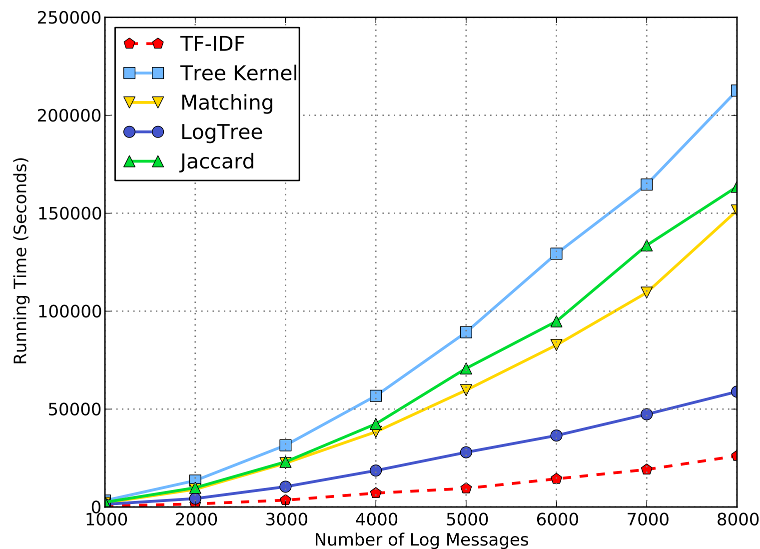

Figure 3.6: The Scalability of K-Medoids on FileZilla logs

In Figures 3.3, 3.4 and 3.5, "TF-IDF" is the most efficient approach in the vector space model based text clustering. The reason is that, the sparse vector is a compact representation of the log message. The cosine similarity of two sparse vectors can be obtained in one pass. The vector transformation can be achieved in a linear time complexity by using

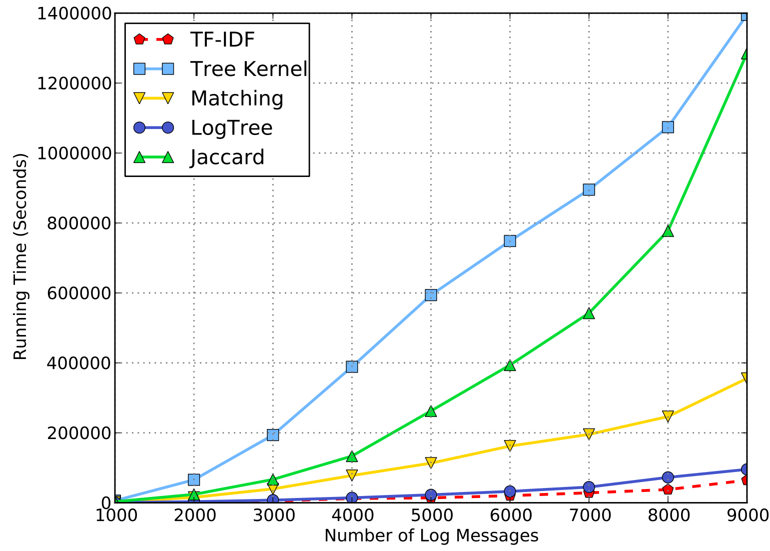

Figure 3.7: The Scalability of K-Medoids on PVFS2 logs

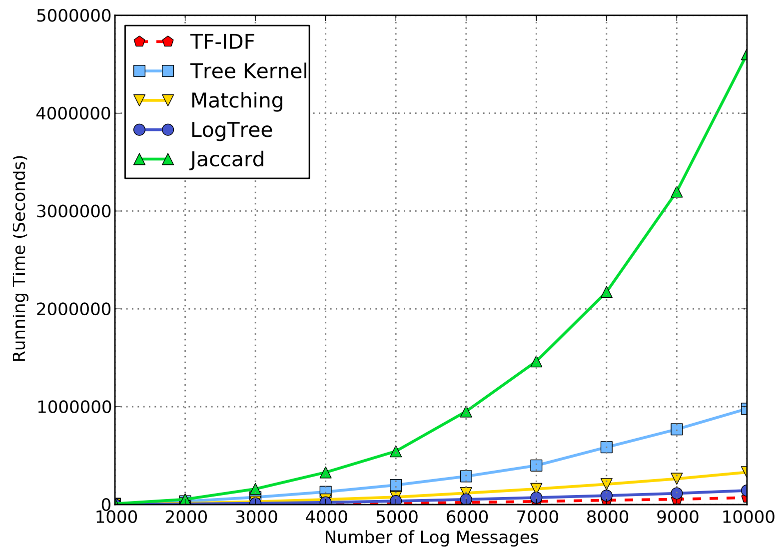

Figure 3.8: The Scalability of K-Medoids on Apache logs

a hash table. Furthermore, the cosine similarity of vectors do not consider the structural information of two log messages.

Our proposed approach, "LogTree", is in the second place in Figures 3.3, 3.4 and 3.5. With the help of the Message Segment Table, it can save a lot of computation to obtain the

similarity of two tree nodes. However, in order to consider the structural information of the log message, the similarity function F_C still has to find the most matched node in each level of the tree. So it cannot be completed in one pass as the cosine similarity.

"Tree Kernel", "Matching" and "Jaccard" are slower than the previous two methods. One reason is that, those three methods do not provide a compact representation of the log message in the main memory. For the similarity of every two messages, they all have to access the original messages, requiring more CPU and I/O costs. As for "Tree Kernel", it compares every pair of nodes in the same level and its time complexity $O(mn^3)$ is very large, where m and n are the number of nodes in the two trees respectively [CS04].

The Scalability of Event Generation

We run all methods on the logs with different sizes to evaluate their time scalability. Figure 3.6, 3.7 and 3.8 show the scalability results of K-Medoids algorithm with different similarity measurements. The running time is obtained by averaging 5 different runs as mentioned before. This experiment needs more than 2G main memory, so it is conducted in a powerful machine. The results shown in Figure 3.6, 3.7 and 3.8 are consistent with the efficiency tests in previous subsection. "TF-IDF" is the most efficient approach, and our proposed method,"LogTree", is in the second place, where the threshold for MST $f_{min} = 0.00001$.

The space costs for all methods are identical except for our method "LogTree". For "LogTree", there is an additional message segment table. The message segment table is always maintained in the main memory. Figure 3.9 shows the space cost of message segment tables, which is the sum of the entries of each level's MST, where $f_{min} = 0.00001$. In this figure, FileZilla log has the largest space cost in MSTs. The reason is that, the diversity of FileZilla log is very low, so MST almost covers all message segments. On the other hand, the diversity of PVFS2 log is high, which covers various kinds of status messages, error,

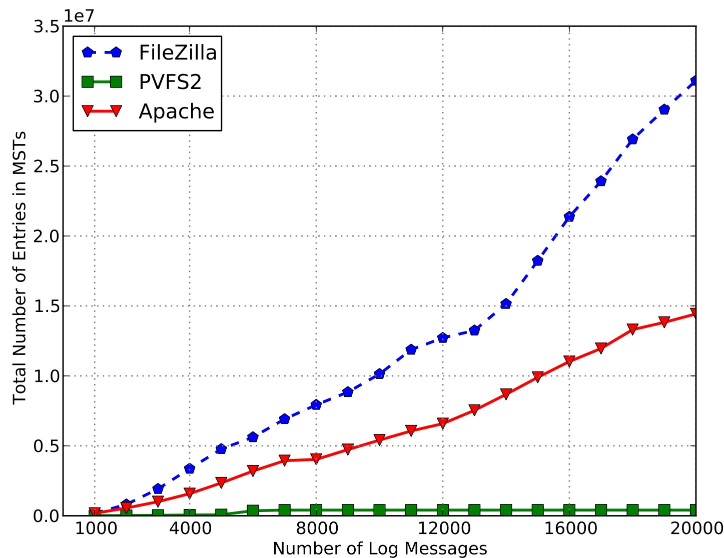

Figure 3.9: Space Cost of LogTree.

internal operations. Thus, only a few message segments' frequencies are greater than f_{min} and are maintained in the MST.

Every entry of MST is a float number, which occupies 4 bytes. The largest actual memory cost of those MSTs in Figure 3.9 is $3.2 \times 10^7 \times 4$ = 128M bytes. Comparing to the similarity matrix of log messages built by the clustering algorithm, $20000 \times 20000/2 \times 4$ = 1.6G bytes, the MST's cost can be ignored.

A Case Study

We have developed a log analysis toolkit using *Logtree* for events generation from system log data. Figure 3.10 shows a case study of using our developed toolkit for detecting configuration errors in Apache Web Server. The configuration error is usually cased by human, which is quite different from random TCP transmission failures, or disk read errors. As a result, configuration errors typically lead to certain patterns. However, the Apache error log file has over 200K log messages. It is difficult to discover those patterns directly from the raw log messages. Figure 3.10 shows the event timeline window of our toolkit,

where the user can easily identify the configuration error in the time frame. This error is related to the permission setting of the HTML file. It causes continuous permission denied errors in a short time. In addition, by using the hierarchical clustering method, *LogTree* provides multi-level views of the events. The user could use the slider to choose a deeper view of events to check detail information about this error.

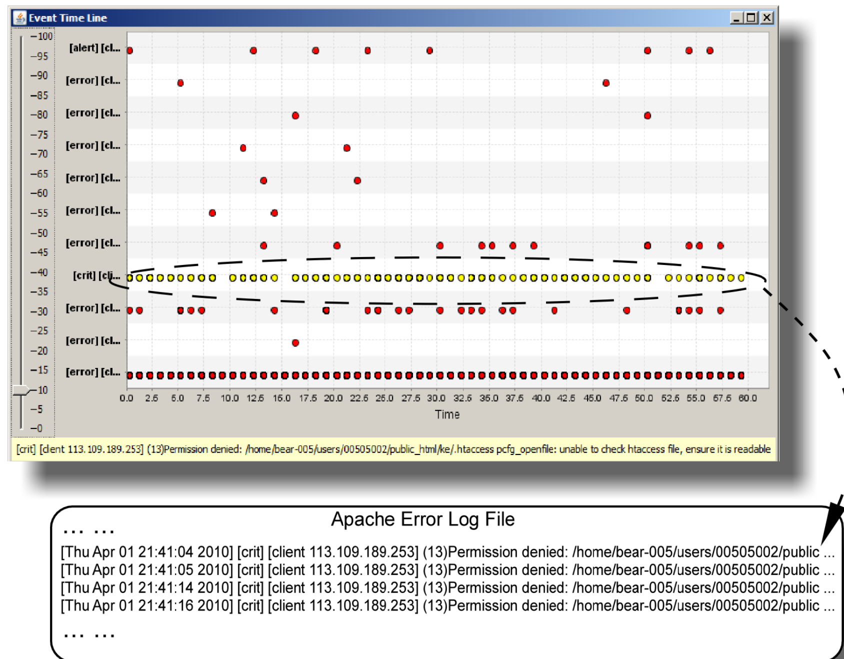

Figure 3.10: A case study of the Apache HTTP server log.

3.2 Message Signature Based Clustering

Message signature based clustering is the second proposed algorithm for converting the textual logs into system events in this dissertation. Since this algorithm is based the captured message signature of log messages, it is called `LogSig`.

Each log message consists of a sequence of terms. Some of the terms are variables or parameters for a system event, such as the host name, the user name, IP address and so

on. Other terms are plain text words describing semantic information of the event. For example, three sample log messages of the Hadoop system [urlc] describing one type of events about the IPC (Inter-Process Communication) subsystem are listed below:

```
1.  2011-01-26 13:02:28,335 INFO org.apache.hadoop.ipc.
Server:  IPC Server Responder:  starting;

2.  2011-01-27 09:24:17,057 INFO org.apache.hadoop.ipc.
Server:  IPC Server listener on 9000:  starting;

3.  2011-01-27 23:46:21,883 INFO org.apache.hadoop.ipc.
Server:  IPC Server handler 1 on 9000:  starting.
```

The three messages contain many different words(or terms), such as the date, the hours, the handler name, and the port number. People can identify them as the same event type because they share a common subsequence: "*INFO: org.apache.hadoop.ipc .Server: IPC Server:starting*". Let's consider how the three log messages are generated by the system. The Java source code for generating them is described below:

```
logger = Logger.getLogger("org.apache.hadoop.ipc.Server");
logger.info("IPC Server "+handlerName+":  starting");
```

where `logger` is the log producer for the IPC subsystem. Using different parameters, such as `handlerName`, the code can output different log messages. But the subsequence "*INFO: org.apache.hadoop.ipc.Server: IPC Server : starting*" is fixed in the source code. It will never change unless the source code has been modified.

Therefore, the fixed subsequence can be viewed as a **signature** for an event type. In other words, we can check the signatures to identify the event type of a log message. Other parameter terms in the log message should be ignored, since messages of the same event type can have different parameter terms. Note that some parameters, such as the `handlerName` in this example, consist of different numbers of terms. Consequently, the position of a message signature may vary in different log messages. Hence, the string matching similarity proposed in [ABCM09] would mismatch some terms. Another method `IPLoM` proposed in [MZHM09] also fails to partition log messages using the term count since the length of `handlerName` is not fixed and three log messages have different numbers of terms. Given an arbitrary log message, we do not know in advance which item is of its signature, or which term is its parameter. That is the key challenge to address.

The goal is to identify the event type of each log message according to a set of message signatures. Given a log message and a set of signatures, we need a metric to determine which signature best matches this log message. Therefore, we propose the *Match Score* metric first.

Let $\mathcal{D}$ be a set of log messages, $\mathcal{D} = \{X_1, ..., X_N\}$, where X_i is the ith log message, $i = 1, 2, ..., N$. Each X_i is a sequence of terms, i.e., $X_i = w_{i_1} w_{i_2} .... w_{i_{n_i}}$. A message signature S is also a sequence of terms $S = w_{j_1} w_{j_2} .... w_{j_n}$.

Given a sequence $X = w_1 w_2 ... w_n$ and a term w_i, $w_i \in X$ indicates w_i is a term in X. $X - \{w_i\}$ denotes a subsequence $w_1 ... w_{i-1} w_{i+1} ... w_n$. $|X|$ denotes the length of the sequence X. $LCS(X, S)$ denotes the *Longest Common Subsequence* between two sequences X and S.

Definition 3.2.1. *(**Match Score**) Given a log message X_i and a message signature S, the* match score *is computed by the function below:*

$$\begin{aligned} match(X_i, S) &= |LCS(X_i, S)| - (|S| - |LCS(X_i, S)|) \\ &= 2|LCS(X_i, S)| - |S|. \end{aligned}$$

Intuitively, $|LCS(X_i, S)|$ is the number of terms in X_i matched with S. $|S| - |LCS(X_i, S)|$ is the number of terms in X_i not matched with S. $match(X_i, S)$ is the number of matched terms minus the number of not-matched terms. We illustrate this by a simple example below:

Example 2. *A log messages $X = abcdef$ and a message signature $S = axcey$. The longest common subsequence $LCS(X, S) = ace$. The matched terms are "a","c","e", shown by*

Table 3.8: Example of Match Score

X	a	b	c	d	e	f
S	a	***x***	c		e	***y***

underline words in Table 3.8. "x" and "y" in S are not matched with any term in X. Hence, $match(X, S) = |ace| - |xy| = 1$.

Note that this score can be negative. $match(X_i, S)$ is used to measure the degree of the log message X_i owning the signature S. If two log messages X_i and X_j have the same signature S, then we regard X_i and X_j as of the same event type. The longest common subsequence matching is a widely used similarity metric in biological data analysis [BKWZ07] [NNL06], such as RNA sequences.

If all message signatures S_1, S_2,...,S_k are known, identifying the event type of each log message in $\mathcal{D}$ is straightforward. But we don't know any message signature at the beginning. Therefore, we should partition log messages and find their message signatures

simultaneously. The optimal result is that, within each partition, every log message matches its signature as much as possible. This problem is formulated below.

Problem 1. *Given a set of log messages $\mathcal{D}$ and an integer k, find k message signatures $\mathcal{S} = \{S_1, ..., S_k\}$ and a k-partition C_1,...,C_k of $\mathcal{D}$ to maximize*

$$J(\mathcal{S}, \mathcal{D}) = \sum_{i=1}^{k} \sum_{X_j \in C_i} match(X_j, S_i).$$

The objective function $J(\mathcal{S}, \mathcal{D})$ is the summation of all match scores. It is similar to the k-means clustering problem. The choice of k depends on the user's domain knowledge to the system logs. If there is no domain knowledge, we can borrow the idea from the method finding k for k-means [HE03], which plots clustering results with k. We can also display generated message signatures for $k = 2, 3, ..$ until the results can be approved by experts.

3.2.1 Comparing with k-means clustering problem

Problem 1 is similar to the classic k-means clustering problem, since a message signature can be regarded as the representative of a cluster. People may ask the following questions:

- Why we propose the $match$ function to find the optimal partition?
- Why not use the LCS as the similarity function to do k-means clustering?

The answer for the two questions is that, our goal is not to find good clusters of log messages, but to find the message signatures of all types of log messages. K-means can ensure every two messages in one cluster share a subsequence. However, it cannot guarantee that there exists a common subsequence shared by all (or most) messages in one cluster. We illustrate this by the following example.

Example 3. *There are three log messages X_1: "abcdef, X_2: "abghij" and X_3: "xyghef". Clearly, $LCS(X_1, X_2)$=2, $LCS(X_2, X_3)$=2, and $LCS(X_1, X_3)$=2. However, there is no*

common subsequence that exists among all X_1, X_2 and X_3. In our case, it means there is no message signature to describe all three log messages. Hence, it is hard to believe that they are generated by the same log message template.

Problem 1 is an NP-hard problem, even if $k = 1$. When $k = 1$, we can reduce the *Multiple Longest Common Subsequence* problem to the Problem 1. The *Multiple Longest Common Subsequence* problem is a known NP-hard [Mai78].

Lemma 3.2.2. *Problem 1 is an NP-hard problem when $k = 1$.*

Proof: Let $\mathcal{D} = \{X_1, ..., X_N\}$. When $k = 1$, $\mathcal{S} = \{S_1\}$. Construct another set of N sequences $\mathcal{Y} = \{Y_1, ..., Y_N\}$, in which each term is unique in both $\mathcal{D}$ and $\mathcal{Y}$. Let $\mathcal{D}' = \mathcal{D} \cup \mathcal{Y}$,

$$J(\mathcal{S}, \mathcal{D}') \quad = \textstyle\sum_{X_j \in \mathcal{D}} match(X_j, S_1) + \sum_{Y_l \in \mathcal{Y}} match(Y_l, S_1)$$

Let S_1^* be the optimal message signature for $\mathcal{D}'$, i.e.,

$$S_1^* = \arg\max_{S_1} J(\{S_1\}, \mathcal{D}').$$

Then, the longest common subsequence of X_1,...,X_N must be an optimal solution S_1^*. This can be proved by contradiction as follows. Let S_{lcs} be the longest common subsequence of X_1,...,X_N. Note that S_{lcs} may be an empty sequence if there is no common subsequence among all messages.

Case 1: If there exists a term $w_i \in S_1^*$, but $w_i \notin S_{lcs}$. Since $w_i \notin S_{lcs}$, w_i is not matched with at least one message in X_1,...,X_N. Moreover, Y_1,...,Y_N are composed by unique terms, so w_i cannot be matched with any of them. In $\mathcal{D}'$, the number of messages not matching

w_i is at least $N+1$, which is greater than the number of messages matching w_i. Therefore,

$$J(\{S_1^* - \{w_i\}\}, \mathcal{D}') > J(\{S_1^*\}, \mathcal{D}'),$$

which contradicts with $S_1^* = \arg\max_{S_1} J(\{S_1\}, \mathcal{D}')$.

Case 2: If there exists a term $w_i \in S_{lcs}$, but $w_i \notin S_1^*$. Since $w_i \in S_{lcs}$, X_1,...,X_N all match w_i. The total number of messages that match w_i in $\mathcal{D}'$ is N. Then, there are N remaining messages not matching w_i: Y_1,...,Y_N. Therefore,

$$J(\{S_{lcs}\}, \mathcal{D}') = J(\{S_1^*\}, \mathcal{D}'),$$

which indicates S_{lcs} is also an optimal solution to maximize objective function J on $\mathcal{D}'$.

To sum up the two cases above, if there is a polynomial time-complexity solution to find the optimal solution S_1^* in $\mathcal{D}'$, the *Multiple Longest Common Subsequence problem* for X_1,...,X_N can be solved in polynomial time as well. However, *Multiple Longest Common Subsequence problem* is an NP-hard problem [Mai78].

Lemma 3.2.3. *If when $k = n$ Problem 1 is NP-hard, then when $k = n + 1$ Problem 1 is NP-hard, where n is a positive integer.*

Proof-Sketch: This can be proved by contradiction. We can construct a message Y whose term set has no overlap to the term set of messages in $\mathcal{D}$ in a linear time. Suppose the optimal solution for $k = n$ and $\mathcal{D}$ is $\mathcal{C} = \{C_1, ..., C_k\}$, then the optimal solution for $k = n + 1$ and $\mathcal{D} \cup \{Y\}$ should be $\mathcal{C}' = \{C_1, ..., C_k, \{Y\}\}$. If there is a polynomial time solution for Problem 1 when $k = n + 1$, we could solve Problem 1 when $k = n$ in polynomial time.

Since the original problem is NP-hard, we can solve an approximated version of the Problem 1 that is easier to come up with an efficient algorithm. The first step is to separate

every log message into several pairs of terms. The second step is to find k groups of log messages using *local search* strategy such that each group share common pairs as many as possible. The last step is to construct message signatures based on identified common pairs for each message group.

3.2.2 An approximated version of problem

Notations: Let X be a log message, $R(X)$ denotes the set of term pairs converted from X, and $|R(X)|$ denotes the number of term pairs in $R(X)$.

Problem 2. *Given a set of log messages $\mathcal{D}$ and an integer k, find a k-partition $\mathcal{C} = \{C_1, ..., C_k\}$ of $\mathcal{D}$ to maximize objective function $F(\mathcal{C}, \mathcal{D})$:*

$$F(\mathcal{C}, \mathcal{D}) = \sum_{i=1}^{k} | \bigcap_{X_j \in C_i} R(X_j)|.$$

Object function $F(\mathcal{C}, \mathcal{D})$ is the total number of common pairs over all groups. Intuitively, if a group has more common pairs, it is more likely to have a longer common subsequence. Then, the match score of that group would be higher. Therefore, maximizing function F is approximately maximizing J in Problem 1. Lemma 3.2.5 shows the average lower bound for this approximation.

Lemma 3.2.4. *Given a message group C, it has n common term pairs, then the length of the longest common subsequence of messages in C is at least $\lceil \sqrt{2n} \rceil$.*

Proof-sketch: Let l be the length of a longest common subsequence of messages in C. Let $T(l)$ be the number of term pairs that generated by that longest common subsequence. Since each term pair has two terms, this sequence can generate at most $\binom{l}{2}$ pairs. Hence, $T(l) \leq \binom{l}{2} = l(l-1)/2$. Note that each term pair of the longest common subsequence is

a common term pair in C. Now, we already know $T(l) = n$, so $T(l) = n \leq l(l-1)/2$. Then, we have $l \geq \lceil\sqrt{2n}\rceil$.

Lemma 3.2.5. *Given a set of log messages $\mathcal{D}$ and a k-partition $\mathcal{C} = \{C_1, ..., C_k\}$ of $\mathcal{D}$, if $F(\mathcal{C}, \mathcal{D}) \geq y$, y is a constant, we can find a set of message signatures $\mathcal{S}$ such that on average:*

$$J(\mathcal{S}, \mathcal{D}) \geq |\mathcal{D}| \cdot \lceil\sqrt{\frac{2y}{k}}\rceil$$

Proof-sketch: Since $F(\mathcal{C}, \mathcal{D}) \geq y$, on average, each group has at least y/k common pairs. Then for each group, by Lemma 3.2.4, the length of the longest common subsequence must be at least $\lceil\sqrt{\frac{2y}{k}}\rceil$. If we choose this longest common subsequence as the message signature, each log message can match at least $\lceil\sqrt{\frac{2y}{k}}\rceil$ terms of the signature. As a result, the match score of each log message is at least $\lceil\sqrt{\frac{2y}{k}}\rceil$. $\mathcal{D}$ has $|\mathcal{D}|$ messages. Then, we have the total match score $J(\mathcal{S}, \mathcal{D}) \geq |\mathcal{D}| \cdot \lceil\sqrt{\frac{2y}{k}}\rceil$ on average.

Lemma 3.2.5 shows that, maximizing the $F(\mathcal{C}, \mathcal{D})$ is approximately maximizing the original objective function $J(\mathcal{S}, \mathcal{D})$. But $F(\mathcal{C}, \mathcal{D})$ is easier to optimize because it deals with discrete pairs.

3.2.3 Local search

The `LogSig` algorithm applies the *local search* strategy to solve Problem 2. It iteratively moves one message to another message group to increase the objective function as large as possible. However, unlike the classic *local search* optimization method, the movement is not explicitly determined by objective function $F(\cdot)$. The reason is that, the value of $F(\cdot)$ may only be updated after a bunch of movements, not just after every single movement. We illustrate this by the following example.

Example 4. *Message set $\mathcal{D}$ is composed of 100 "`ab`" and 100 "`cd`". Now we have 2-partition $\mathcal{C} = \{C_1, C_2\}$. Each message group has 50% of each message type as shown in Table 3.9. The optimal 2-partition is C_1 has 100 "`ab`" and C_2 has 100 "`cd`", or in*

Table 3.9: Example of two message groups

group / term pair	C_1	C_2
"`ab`"	50	50
"`cd`"	50	50

the reverse way. However, beginning with current C_1 and C_2, $F(\mathcal{C}, \mathcal{D})$ is always 0 until we move 50 "`ab`" from C_2 to C_1, or move 50 "`cd`" from C_1 to C_2. Hence, for first 50 movements, $F(\mathcal{C}, \mathcal{D})$ cannot guide the local search because no matter what movement you choose, it is always 0.

Therefore, $F(\cdot)$ is not proper to guide the movement in the local search. The decision of every movement should consider the *potential* value of the objective function, rather than the immediate value. So we develop the *potential function* to guide the local search instead.

Notations: Given a message group C, $R(C)$ denotes the union set of term pairs from messages of C. For a term pair $r \in R(C)$, $N(r, C)$ denotes the number of messages in C which contains r. $p(r, C) = N(r, C)/|C|$ is the portion of messages in C having r .

Definition 3.2.6. *Given a message group C, the potential of C is defined as $\phi(C)$,*

$$\phi(C) = \sum_{r \in R(C)} N(r, C)[p(r, C)]^2.$$

The potential value indicates the overall "purity" of term pairs in C. $\phi(C)$ is maximized when every term pair is contained by every message in the group. In that case, for each r, $N(r, C) = |C|$, $\phi(C) = |C| \cdot |R(C)|$. It also means all term pairs are common pairs shared

by every log message. $\phi(C)$ is minimized when each term pair in $R(C)$ is only contained by one message in C. In that case, for each r, $N(r, C) = 1$, $|R(C)| = |C|$, $\phi(C) = 1/|C|$.

Definition 3.2.7. *Given a k-partition $\mathcal{C} = \{C_1, ..., C_k\}$ of a message set $\mathcal{D}$, the overall potential of $\mathcal{D}$ is defined as $\Phi(\mathcal{D})$,*

$$\Phi(\mathcal{D}) = \sum_{i=1}^{k} \phi(C_i),$$

where $\phi(C_i)$ is the potential of C_i, $i = 1, ..., k$.

3.2.4 Connection between Φ and F:

Objective function F computes the total number of common term pairs in each group. Both Φ and F are maximized when each term pair is a common term in its corresponding message group. Let's consider the average case.

Lemma 3.2.8. *Given a set of log messages $\mathcal{D}$ and a k-partition $\mathcal{C} = \{C_1, ..., C_k\}$ of $\mathcal{D}$, if $F(\mathcal{C}, \mathcal{D}) \geq y$, y is a constant, then in the average case, $\Phi(\mathcal{D}) \geq y \cdot |\mathcal{D}|/k$.*

Proof-sketch: Since $F(\mathcal{C}, \mathcal{D}) \geq y$, there are at least y common term pairs distributed in message groups. For each common term pair r_i, let C_i be its corresponding group. On average, $|C_i| = |\mathcal{D}|/k$. Note that the common pair r_i appears in every message of C_i, so $N(r_i, C_i) = |C_i| = |\mathcal{D}|/k$ and $p(r_i, C_i) = 1$. There are at least y common term pairs, by Definition 3.2.6, we have $\Phi(\mathcal{D}) \geq y \cdot |\mathcal{D}|/k$.

Lemma 3.2.8 implies, in the average case, if we try to increase the value of F to be at least y, we have to increase the overall potential Φ to be at least $y \cdot |\mathcal{D}|/k$. As for the local search algorithm, we mentioned that Φ is easier to optimize than F.

Let $\Delta_{i\underset{\rightarrow}{X}j}\Phi(\mathcal{D})$ denote the increase of $\Phi(\mathcal{D})$ by moving $X \in \mathcal{D}$ from group C_i into group C_j, $i, j = 1, ..., k$, $i \neq j$. Then, by Definition 3.2.7,

$$\begin{aligned}\Delta_{i\underset{\rightarrow}{X}j}\Phi(\mathcal{D}) &= [\phi(C_j \cup \{X\}) - \phi(C_j)] \\ &\quad -[\phi(C_i) - \phi(C_i - \{X\})],\end{aligned}$$

where $\phi(C_j \cup \{X\}) - \phi(C_j)$ is the potential increase brought by inserting X to C_j, $\phi(C_i) - \phi(C_i - \{X\})$ is the potential loss brought by removing X from C_i. Algorithm 1 is the pseudocode of the local search algorithm in `LogSig`. Basically, it iteratively updates every log message's group according to $\Delta_{i\underset{\rightarrow}{X}j}\Phi(\mathcal{D})$ to increase $\Phi(\mathcal{D})$ until no more update operation can be done.

Algorithm 1 `LogSig_localsearch` $(\mathcal{D}, k)$

Parameter: $\mathcal{D}$: log messages set; k: the number of groups to partition;
Result: $\mathcal{C}$: log message partition;

1: $\mathcal{C} \leftarrow RandomSeeds(k)$
2: $\mathcal{C}' \leftarrow \emptyset$ // Last iteration's partition
3: Create a map G to store message's group index
4: **for** $C_i \in \mathcal{C}$ **do**
5:     **for** $X_j \in C_i$ **do**
6:         $G[X_j] \leftarrow i$
7:     **end for**
8: **end for**
9: **while** $\mathcal{C} \neq \mathcal{C}'$ **do**
10:     $\mathcal{C}' \leftarrow \mathcal{C}$
11:     **for** $X_j \in D$ **do**
12:         $i \leftarrow G[X_j]$
13:         $j^* = \arg\max_{j=1,..,k} \Delta_{i\underset{\rightarrow}{X}j}\Phi(\mathcal{D})$
14:         **if** $i \neq j^*$ **then**
15:             $C_i \leftarrow C_i - \{X_j\}$
16:             $C_{j^*} \leftarrow C_{j^*} \cup \{X_j\}$
17:             $G[X_j] \leftarrow j^*$
18:         **end if**
19:     **end for**
20: **end while**
21: **return** $\mathcal{C}$

3.2.5 Why choose this potential function?

Given a message group C, let $g(r) = N(r, C)[p(r, C)]^2$, then $\phi(C) = \sum_{r \in R(C)} g(r)$. Since we have to consider all term pairs in C, we define $\phi(C)$ as the sum of all $g(r)$. As for $g(r)$, it should be a convex function. Figure 3.11 shows a curve of $g(r)$ by varying the number of messages having r, i.e., $N(r, C)$. The reason for why $g(r)$ is convex is that, we hope to

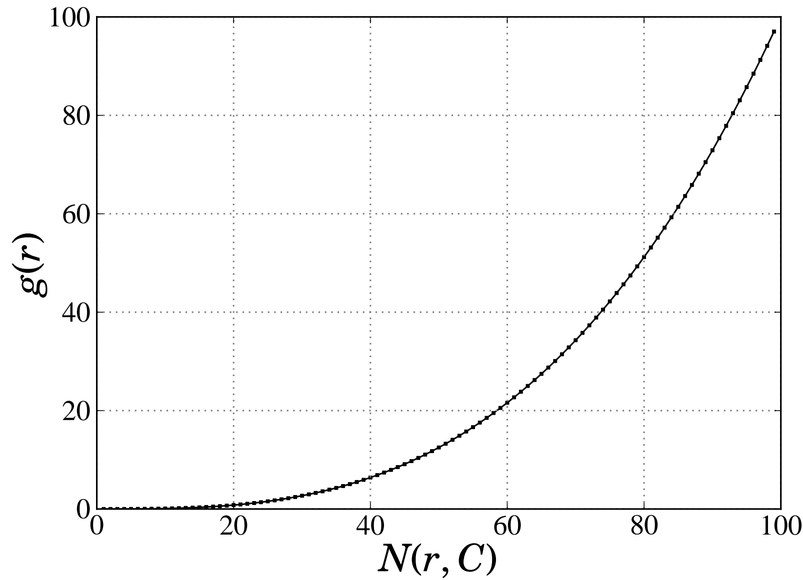

Figure 3.11: Function $g(r)$, $|C| = 100$

give larger awards to r when r is about to being a common term pair. That is because, if $N(r, C)$ is large, then r is more likely to be a common term pair. Only when r becomes a common term pair, it can increase $F(\cdot)$. In other words, r has more potential to increase the value of objective function $F(\cdot)$, so the algorithm should pay more attention to r first.

3.2.6 Evaluation

Experimental Platforms

We implement our algorithm and other comparative algorithms in Java 1.6 platform. Table 3.10 summarizes our experimental environment.

Table 3.10: Experimental Machine

OS	**CPU**	**bits**	**Memory**	**JVM Heap Size**
Linux 2.6.18	Intel Xeon(R) @ 2.5GHz, 8 core	64	16G	12G

Data Collection

We collect log data from 5 different real systems, which are summarized in Table 3.11. Logs of FileZilla [urlb], PVFS2 [urlj] Apache [urla] and Hadoop [urlc] are collected from the server machines/systems in the computer lab of a research center. Log data of ThunderBird [urll] is collected from a supercomputer in Sandia National Lab. The true categories of log messages are obtained by specialized log parsers. For instance, FillZilla's log messages are categorized into 4 types: "*Command*", "*Status*", "*Response*", "*Error*". Apache error log messages are categorized by the error type: "*Permission denied*", "*File not exist*" and so on.

Table 3.11: Summary of Collected System Logs

System	**Description**	**#Messages**	**#Terms Per Message**	**#Category**
FileZilla	SFTP/FTP Client	22,421	7 to 15	4
ThunderBird	Supercomputer	3,248,239	15 to 30	12
PVFS2	Parallel File System	95,496	2 to 20	11
Apache Error	Web Server	236,055	10 to 20	6
Hadoop	Parallel Computing Platform	2,479	15 to 30	11

The vocabulary size is an important characteristic of log data. Figure 3.12 exhibits the vocabulary sizes of the 5 different logs along with the data size. It can be seen that some vocabulary size could become very large if the data size is large.

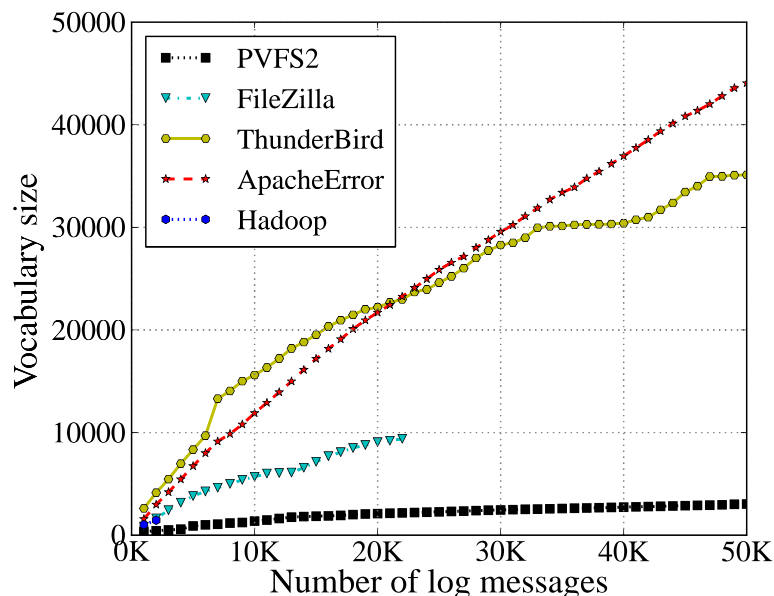

Figure 3.12: Vocabulary size

Comparative Algorithms

We compare our algorithm with 7 alternative algorithms in this experiment. Those algorithms are described in Table 3.12. 6 of them are unsupervised algorithms which only look at the terms of log messages. 3 of them are semi-supervised algorithms which are able to incorporate the domain knowledge. `IPLoM` [MZHM09] and `StringMatch` [ABCM09] are two methods proposed in recent related literatures . `VectorModel` [SM84], `Jaccard` [TSK05], `StringKernel` [LSST$^+$02] are traditional methods for text clustering. `VectorModel` and `semi-StringKernel` are implemented by k-means clustering algorithm [TSK05]. `Jaccard` and `StringMatch` are implemented by k-medoid algorithm [HKP05], since they cannot compute the centroid point of a cluster. As for `Jaccard`, the Jaccard similarity is obtained by a hash table to accelerate the computation. `VectorModel` and `StringKernel` use *Sparse Vector* [SM84] to reduce the computation and space costs.

`semi-LogSig`, `semi-StringKernel` and `semi-Jaccard` are semi-supervised versions of `LogSig`, `StringKernel` and `Jaccard` respectively. To make a fair com-

Table 3.12: Summary of comparative algorithms

Algorithm	Description
`VectorModel`	Vector space model proposed in information retreival
`Jaccard`	Jaccard similarity based k-medoid algorithm
`StringKernel`	String kernel based k-means algorithm
`IPLoM`	Iterative partition method proposed in [MZHM09]
`StringMatch`	String matching method proposed in [ABCM09]
`LogSig`	Message signature based method proposed in this paper
`semi-LogSig`	`LogSig` incorporating domain knowledge
`semi-StringKernel`	Weighted string kernel based k-means
`semi-Jaccard`	Weighted Jaccard similarity based k-medoid

parison, all those semi-supervised algorithms incorporate the **same** domain knowledge offered by users. Specifically, the 3 algorithms run on the same transformed feature layer, and the same sensitive phrases $\mathcal{P}_S$ and trivial phrases $\mathcal{P}_T$. Obviously, the choices of features, $\mathcal{P}_S$ and $\mathcal{P}_T$ have a huge impact to the performances of semi-supervised algorithms. But we only compare a semi-supervised algorithm with other semi-supervised algorithms. Hence, they are compared under the same choice of features, $\mathcal{P}_S$ and $\mathcal{P}_T$. The approaches for those 3 algorithms to incorporate with features, $\mathcal{P}_S$ and $\mathcal{P}_T$ are described as follows:

Feature Layer: Replacing every log message by the transformed sequence of terms with features.

$\mathcal{P}_S$ **and** $\mathcal{P}_T$**:** As for `semi-StringKernel`, replacing Euclidean distance by Mahalanobis distance [BBM04]:

$$D_M(x, y) = \sqrt{(x - y)^T M (x - y)}.$$

where matrix M is constructed according to term pairs $\mathcal{P}_S$, $\mathcal{P}_T$ and λ'. As for `semi-Jaccard`, for each term, multiply a weight λ' (or $1/\lambda'$) if this term appears in $\mathcal{P}_S$ (or $\mathcal{P}_T$).

Table 3.13: Summary of small log data

Measure	**#Message**	**#Feature**	$\|R(\mathcal{P})_S\|$	$\|R(\mathcal{P})_T\|$
FileZilla	8555	10	4	4
ThunderBird	5000	10	11	9
PVFS2	12570	10	10	1
Apache Error	5000	2	4	2
Hadoop	2479	2	7	3

Table 3.14: Average F-Measure Comparison

Algorithm \ **Log Data**	**FileZilla**	**PVFS2**	**ThunderBird**	**Apache Error**	**Hadoop**
`Jaccard`	0.3794	0.4072	0.6503	0.7866	0.5088
`VectorModel`	0.4443	0.5243	0.4963	0.7575	0.3506
`IPLoM`	0.2415	0.2993	**0.8881**	0.7409	0.2015
`StringMatch`	0.5639	0.4774	0.6663	0.7932	0.4840
`StringKernel`$_{0.8}$	0.4462	0.3894	0.6416	0.8810	0.3103
`StringKernel`$_{0.5}$	0.4716	0.4345	0.7361	**0.9616**	0.3963
`StringKernel`$_{0.3}$	0.4139	0.6189	0.8321	0.9291	0.4256
`LogSig`	**0.6949**	**0.7179**	0.7882	0.9521	**0.7658**
`semi-Jaccard`	0.8283	0.4017	0.7222	0.7415	0.4997
`semi-StringKernel`$_{0.8}$	0.8951	0.6471	0.7657	0.8645	0.7162
`semi-StringKernel`$_{0.5}$	0.7920	0.4245	0.7466	**0.8991**	0.7461
`semi-StringKernel`$_{0.3}$	0.8325	0.7925	0.7113	0.8537	0.6259
`semi-LogSig`	**1.0000**	**0.8009**	**0.8547**	0.7707	**0.9531**

`Jaccard`, `StringMatch` and `semi-Jaccard` algorithms apply classic k-medoid algorithm for message clustering. The time complexity of k-medoid algorithm is very high: $O(tn^2)$ [TKK06], where t is the number of iterations, n is the number of log messages. As a result, those 3 algorithms are not capable of handling large log data. Therefore, for the accuracy comparison, we split our log files into smaller files by time frame, and conduct the experiments on the small log data. The amounts of log messages, features, term pairs in $\mathcal{P}_S$ and $\mathcal{P}_T$ are summarized in Table 3.13.

Quality of Generated Events

Table 3.14 shows the accuracy comparison of generated system events by different algorithms. The accuracy is evaluated by F-measure (F1 score) [SM84], which is a traditional metric combining *precision* and *recall*. Since the results of k-medoid, k-means and `LogSig` depend on the initial random seeds, we run each algorithm for **10 times**,

and put the average F-measures into Table 3.14. From this table, it can be seen that StringKernel and LogSig outperform other algorithms in terms of the overall performance.

Jaccard and VectorModel apply the *bag-of-word* model, which ignores the order information about terms. Log messages are usually short, so the information from the bag-of-word model is very limited. In addition, different log messages have many identical terms, such as *date*, *username*. That's the reason why the two methods cannot achieve high F-measures. IPLoM performs well in ThunderBird log data, but poorly in other log data. The reason is that, the first step of IPLoM is to partition log message by the term count. One type of log message may have different numbers of terms. For instance, in FileZilla logs, the length of *Command* messages depends on the type of SFTP/FTP command in the message. But for ThunderBird, most event types are strictly associated with one message format. Therefore, IPLoM could easily achieve the highest score.

Due to the *Curse of dimensionality* [TSK05], k-means based StringKernel is not easy to converge in a high dimensional space. Figure 3.12 shows that, 50K ThunderBird log messages contain over 30K distinct terms. As a result, the transformed space has over $(30K)^2 = 900M$ dimensions. It is quite sparse for 50K data points.

It is worthy to note that in Thunderbird and Apache Error logs the vocabulary size increases almost infinitely (see Figure 3.12), then LogSig does not achieve the best performance. The main reason is that, when the vocabulary size is large, the number of possible choices of the signature terms is also large. Then the performance of LogSig may suffer from the large solution space for the local search algorithm.

Generated message signatures are used as descriptors for system events, so that users can understand the meanings of those events. Due to the space limit, we cannot list all message signatures. Table 3.15 shows generated signatures of FileZilla and Apache Error by semi-LogSig, in which features are indicated by *italic words*.

Table 3.15: Message Signatures

System Log	Message Signature	Associated Category
FileZilla	*Date Hours Number Number* Status: ...	Status
	Date Hours Number Number Response: *Number*	Response
	Date Hours Number Number Command:	Command
	Date Hours Number Number Error: File transfer failed	Error
Apache Error	*Timestamp* (13) Permission denied: /home/bear-005/users/xxx/public_html/ke/.htaccess pcfg_openfile: unable to check htaccess file ensure it is readable	Permission denied
	Timestamp Error [client] File does not exist: /opt/website/sites/users.cs.fiu.edu/ data/favicon.ico	File does not exist
	Timestamp Error [client 66.249.65.4] suexec *policy* violation: see suexec log for more details	Policy violation
	Timestamp /home/hpdrc-demo/sdbtools/public_html/ hpdrc2/.htaccess: AuthName takes one argument The *Authentication* realm (e.g. "Members Only")	Authentication
	Timestamp Error [client] 2010-04-01 using	N/A
	Timestamp Error [client]	N/A

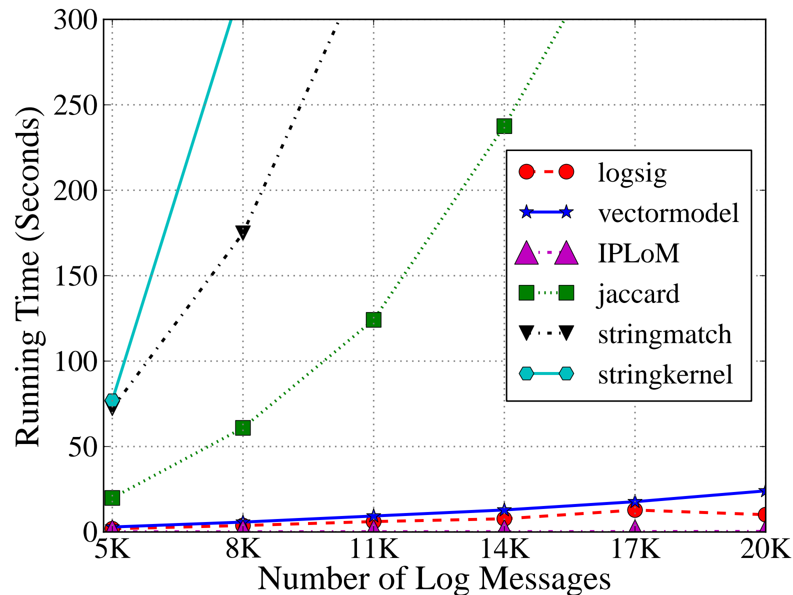

Figure 3.13: Average Running Time for FileZilla logs

As for FileZilla log, each message signature corresponds to a message category, so that the F-measure of FileZilla could achieve 1.0. But for Apache Error log, Only 4 message signatures are associated with corresponding categories. The other 2 signatures are generated by two ill-partitioned message groups. They cannot be associates with any category

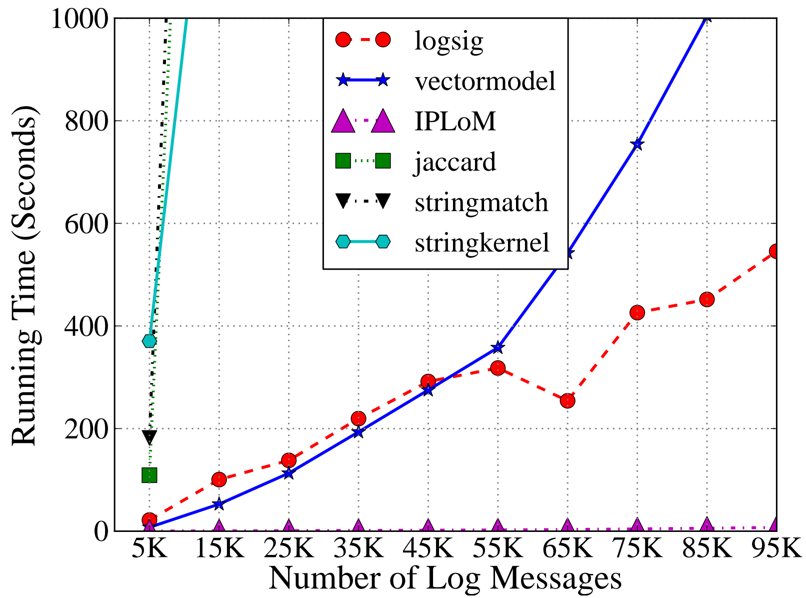

Figure 3.14: Average Running Time for ThunderBird logs

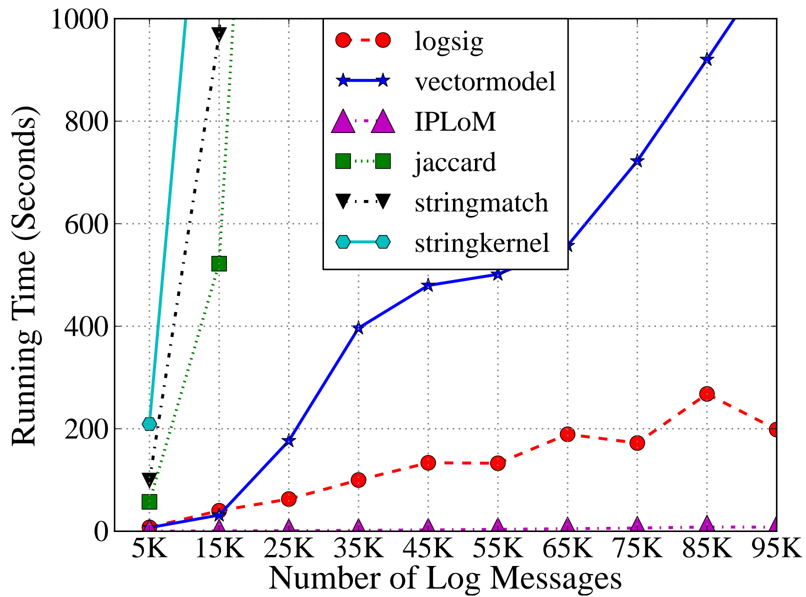

Figure 3.15: Average Running Time for Apache logs

of Apache Error logs. As a result, their "Associated Category" in Table 3.15 are "N/A". Therefore, the overall F-measure on Apache error log in Table 3.14 is only 0.7707.

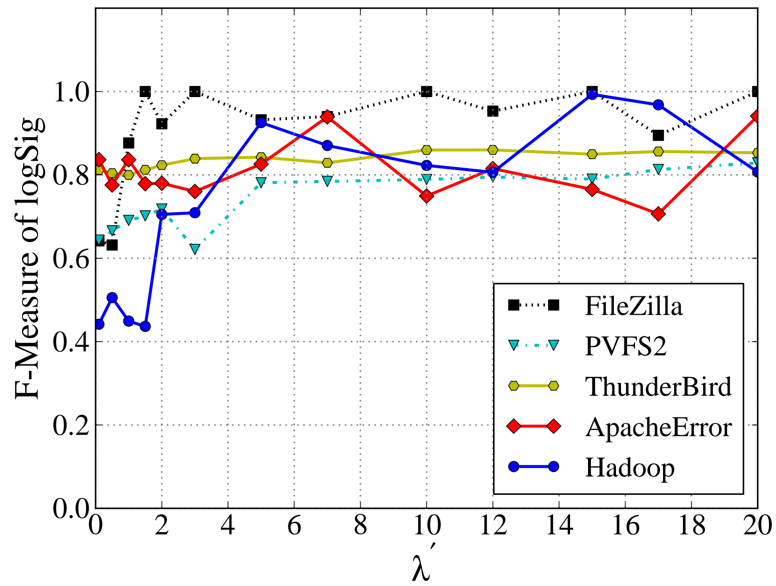

Figure 3.16: Varying parameter λ'

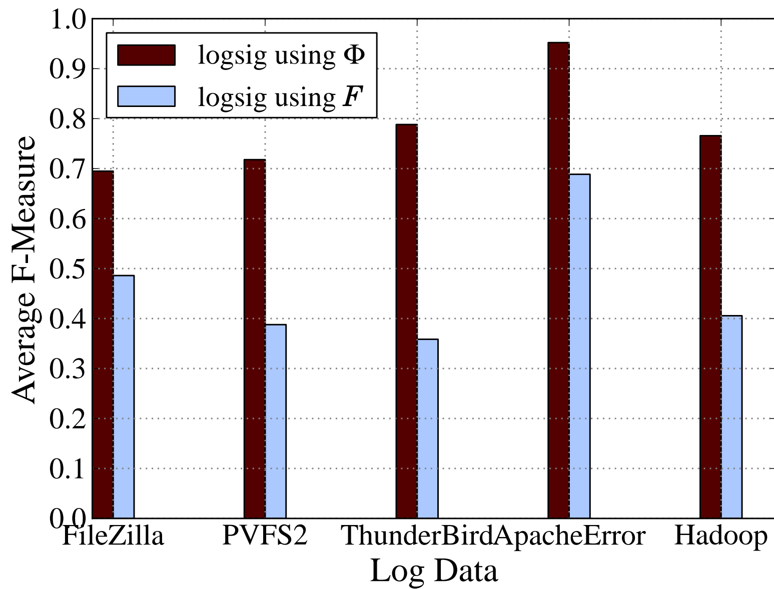

Figure 3.17: Effectiveness of Potential Function

All those algorithms have the parameter k, which is the number of events to create. We let k be the actual number of message categories. String kernel method has an additional parameter λ, which is the decay factor of a pair of terms. We use $\texttt{StringKernel}_\lambda$ to denote

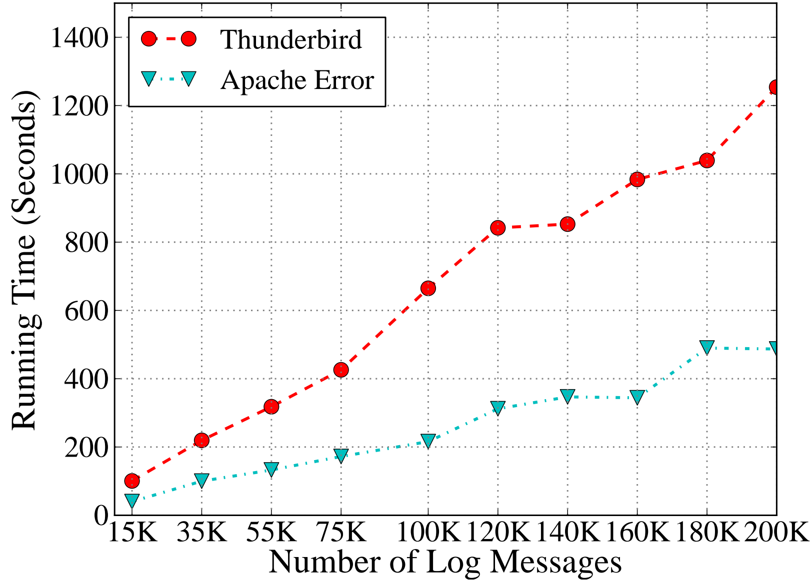

Figure 3.18: Scalability of `LogSig`

the string kernel method using decay factor λ. In our experiments, we set up string kernel algorithms using three different decay factors: $\texttt{StringKernel}_{0.8}$, $\texttt{StringKernel}_{0.5}$ and $\texttt{StringKernel}_{0.3}$.

As for the parameter λ' of our algorithm `LogSig`, we set $\lambda' = 10$ based on the experimental result shown by Figure 3.16. For each value of λ', we run the algorithm for **10 times**, and plot the average F-measure in this figure. It can be seen that, the performance becomes stable when λ' is greater than 4.

Effectiveness of Potential Function

To evaluate the effectiveness of the potential function Φ, we compare our proposed `LogSig` algorithm with another `LogSig` algorithm which uses the objective function F to guide its local search.

Figure 3.17 shows the average F-measures of the two algorithms on each data set. Clearly, our proposed potential function Φ is more effective than F in all data sets. In addition,

we find LogSig algorithm using F always converges within 2 or 3 iterations. In other words, F is more likely to stop at a local optima in the local search.

Scalability

Scalability is an important factor for log analysis algorithms. Many high performance computing systems generate more than 1Mbytes log messages per second [OS07]. Figure 3.13, Figure 3.14 and Figure 3.15 show the average running time comparison for all algorithms on the data sets with different sizes. We run each algorithm 3 times and plot the average running times. IPLoM is the fastest algorithm. The running times of other algorithms depend on the number of iterations. Clearly, k-medoid based algorithms are not capable of handling large log data. Moreover, StringKernel is not efficient even though we use *Sparse Vector* to implement the computation of its kernel functions. We keep track of its running process, and find that the low speed convergence is mainly due to the high dimensionality of the converted vectors.

Figure 3.18 shows the scalability of LogSig algorithm on ThunderBird logs and Apache Error logs. Its actual running time is approximated linear with the log data size.

3.3 Summary

This chapter studies the problem of preprocessing raw textual system logs into discrete system events. The discrete events are more convenient for human to plot and explore. The existing solution is to implement a full log parser, which is time-consuming and difficult since many softwares are not open-source and do not have complete documents. Recent studies apply clustering algorithms on the log messages to generate the events, however, the accuracy of their work heavily relies on the format/structure of the targeting logs. This chapter presents two novel clustering based approaches : LogTree and LogSig. The LogTree algorithm is a novel and algorithm-independent framework for event generation

from raw textual log messages. `LogTree` utilizes the format and structural information of log messages in the clustering process and increases the clustering accuracy. The `LogSig` algorithm is a message signature based clustering algorithm. By searching the most representative message signatures, `LogSig` categorizes the textual log messages into several event types. `LogSig` can handle various types of log data and is able to incorporate the domain knowledge provided by experts to achieve a high clustering accuracy. We conduct experiments on real system logs. The experimental results show that the two algorithms outperform alternative clustering algorithms in terms of the accuracy of event generation.

CHAPTER 4

MONITORING OPTIMIZATION

Defining appropriate monitoring situations requires the knowledge of a particular system and its relationships with other hardware and software systems. It is a known practice to define conservative conditions in nature, thus erring on the side of caution. This practice leads to a large number of tickets that require no action (false positives). Continuous updating of modern IT infrastructures also leads to a number of system faults that are not captured by system monitoring (false negatives). Our research work for this aspect is to utilize the data mining techniques to minimize the number of false positives and false negatives in automatic monitoring systems in large and dynamic IT infrastructures. This approach utilizes the historical monitoring events and incident tickets and is able to help system administrators improve monitoring configurations.

This chapter first introduces the problem of false positive and false negative in IT service management, and then presents the developed methods for eliminating false positives and false negatives by optimizing configurations of existing monitoring systems. The preliminary work for system monitoring and alert detection has been discussed in Section 2.1.

4.1 False Positive and False Negative in IT Service

Performing a detailed analysis of IT system usage is time-consuming, so SAs often rely on default monitoring situations. Furthermore, IT system usage is likely to change over time. This often results in a large number of alerts and tickets (see Table 4.1).

Whether a ticket is real or false is determined by the resolution message entered in the ticket tracking database by the system administrator it was assigned to. It is not rare to observe entire categories of alerts, such as CPU or paging utilization alerts, that are almost exclusively false positives. When reading the resolution messages one by one, it can be

Table 4.1: Definitions for Alert, Event and Ticket

False Positive Alert	An alert for which the system administrator does not need to take any action.
False Negative Alert	A missed alert that is not captured due to inappropriate monitoring configuration.
False Alert	False positive alert
Real Alert	An alert that requires the system administrator to fix the corresponding problem on the server.
Alert Duration	The length of time from an alert creation to its clearing.
Transient Alert	An alert that is automatically cleared before the technician opens its corresponding ticket.
Event	The notification of an alert to the Enterprise Console.
False Positive Ticket	A ticket created from a false positive alert.
False Negative Ticket	A ticket created manually identifying a condition that should have been capture by automatic monitoring.
False Ticket	A ticket created from a false alert.
Real Ticket	A ticket created from a real alert.

simple to find an explanation: Anti-virus processes cause prolonged CPU spikes at regular intervals; databases may reserve large amount of disk space in advance, making the monitors believe the system is running out of storage. With only slightly more effort, one can also fine-tune the thresholds of certain numerical monitored metrics, such as the metrics involved in paging utilization measurement. There are rarely enough human resources, however, to correct the monitoring situations one system at a time, and we need an algorithm capable of discovering these usage-specific rules. There has been a great deal of effort spent on developing the monitoring conditions (situations) that can identify potentially unsafe functioning of the system [HSF06] [RBV03]. It is understandably difficult, however, to recognize and quantify influential factors in the malfunctioning of a complex system. Therefore classical monitoring tends to rely on periodical probing of a system for conditions that could potentially contribute to the system's misbehavior. Upon detection of the predefined conditions, the monitoring systems trigger events that automatically generate incident tickets. In this dissertation, we study the problem of improving the quality of monitoring based on the analysis for historical monitoring events and incident tickets.

4.2 Eliminating False Positive

The objective is to eliminate as many false alerts as possible while retaining all real alerts. A naive solution is to build a predictive classifier and adjust the monitoring situations according to the classifier. Unfortunately, no prediction approach can guarantee 100% success for real alerts, but a single missed one may cause serious problems, such as system crashes or data loss.

The vast majority of the false positive alerts are transient, such as temporary spikes in CPU and paging utilization, service restarts, and server reboots. These transient alerts automatically disappear after a while, but their tickets are created in the ticketing system. When system administrators open the tickets and log on the server, they cannot find the problem described by those tickets. Figure 4.1 shows the duration histogram of false positive alerts raised by one monitoring situation. This particular situation checks the status of a service and generates an alert without delay if the service is stopped or shutdown. These false positive alerts are collected from one server of a customer account for 3 months. As shown by this figure, more than 75% of the alerts can be cleared automatically by waiting 20 minutes. It is possible for a transient alert to be caused by a real system problem. From

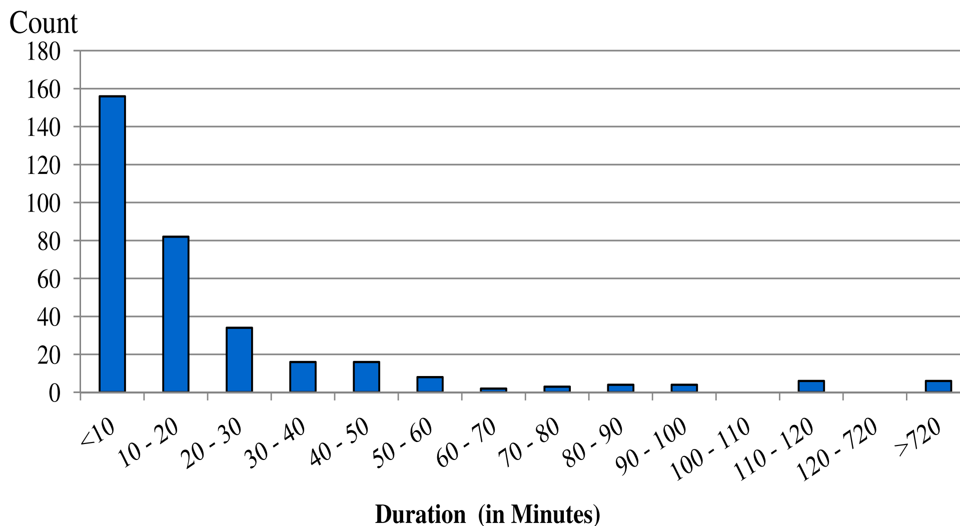

Figure 4.1: False Positive Alert Duration

the perspective of the system administrators, however, if the problem cannot be found when logging on the server, there is nothing they can do with the alert, no matter what happened before. Some transient alerts may be indications of future real alerts and may be useful. But if those real alerts arise later on, the monitoring system will detect them even if the transient alerts were ignored. Therefore, all transient alerts are considered false negative.

Eliminating False Positive Alerts Safely

Our solution first predicts whether an alert is real or false. If it is predicted as real, a ticket will be created. Otherwise, the ticket creation will be postponed. Our solution also determines how long is it to be postponed. Even if a real alert is incorrectly classified as false, its ticket will eventually be created before violating the SLA. Figure 4.2 shows a flowchart of an incoming event. It reveals two key problems for this approach: (1) How to

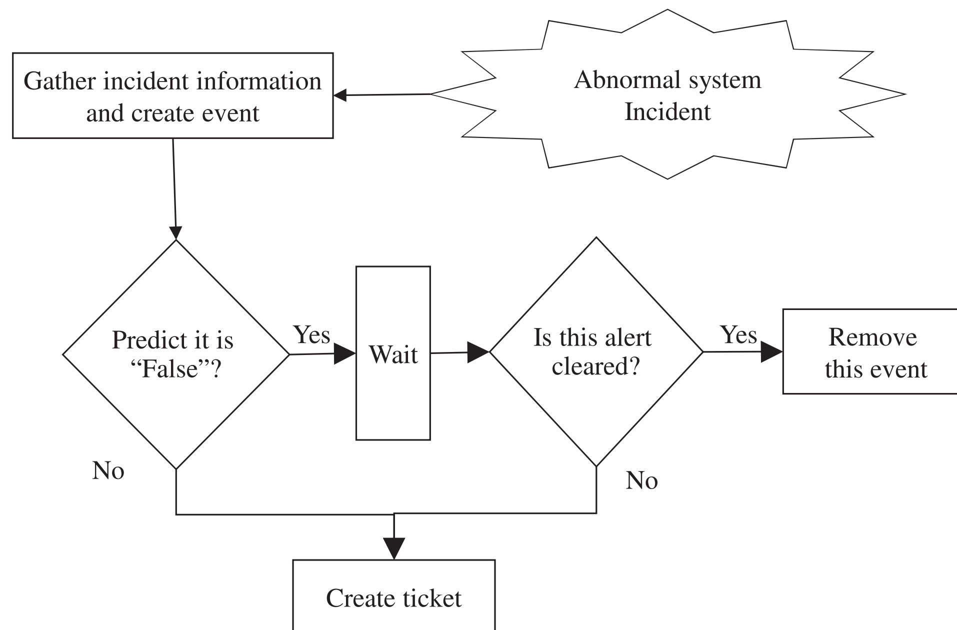

Figure 4.2: Flowchart for Ticket Creation

predict whether an alert is false or real? (2) If an alert is identified as false, what waiting time should be applied before ticket creation?

In our approach, the predictor is implemented by a rule-based classifier based on the historical tickets and events. The ground truth of the events is obtained from the associated tickets. Each historical ticket has one column that suggests this ticket is real or false. This column is manually filled by the system administrators and stored in the ticketing system. There are two reasons for choosing a rule-based predictor. First, each monitoring situation is equivalent to a quantitative rule. The predictor can be directly implemented in the existing monitoring system. Other sophisticated classification algorithms, such as *support vector machine* and *neural network*, may have a higher precision in predicting. Their classifiers, however, are very difficult to implement as monitoring situations in real systems. Second, a rule-based predictor is easily verifiable by the end users. Other complicated classification models represented by linear/non-linear equations or neural networks are very hard for end users to verify. If the analyzed results could not be verified by the system administrators, they would not be utilized in real production servers.

Predictive Rule

The alert predictor roughly assigns a label to each alert, "false" or "real." It is built on a set of predictive rules that are automatically generated by a rule-based learning algorithm [SA96a] based on historical events and alert tickets. Example 5 shows a predictive rule, where "*PROC_CPU_TIME*" is the CPU usage of a process. Here "*PROC_NAME*" is the name of the process.

Example 5. **if** PROC_CPU_TIME $>$ 50% **and** PROC_NAME = 'Rtvscan', **then** this alert is false.

A predictive rule consists of a rule condition and an alert label. A rule condition is a conjunction of *literals*, where each *literal* is composed of an event attribute, a relational operator and a constant value. In Example 5, "*PROC_CPU_TIME* $>$ *50%*" and "*PROC_NAME = 'Rtvscan'*" are two *literals*, where "*PROC_CPU_TIME*" and "*PROC_NAME*" are event

attributes, ">" and "=" are relational operators, and "50%" and "*Rtvscan*" are constant values. If an alert event satisfies a rule condition, we call this alert covered by this rule.

Predictive Rule Generation

The rule-based learning algorithm [SA96a] first creates all *literals* by scanning historical events. Then, it applies a breadth-first search for enumerating all *literals* in finding predictive rules, i.e., those rules having predictive power. This algorithm has two criteria to quantify the minimum predictive power: the minimum confidence $minconf$ and the minimum support $minsup$ [SA96a]. In our case, $minconf$ is the minimum ratio of the numbers of the covered false alerts and all alerts covered by the rule, and $minsup$ is the minimum ratio of the number of alerts covered by the rule and the total number of alerts. The two criteria govern the performance of our method, defined as the total number of removed false alerts. To achieve the best performance, we loop through the values of $minconf$ and $minsup$ and compute their performances.

Predictive Rule Selection

Although the predictive rule learning algorithm can learn many rules from the historical events with tickets, we only select those with strong predictive power. In our solution, Laplace accuracy [YH03] [PMM+94] [Li06] is used for estimating the predictive power of a rule. According to the SLA, real tickets must be acknowledged and resolved within a certain time. The maximum allowed delay time is specified by a user-oriented parameter $delay_{max}$ for each rule. In the calculation of Laplace accuracy, those false alerts are treated as real alerts if their durations are greater than $delay_{max}$. $delay_{max}$ is given by the system administrators according to the severity of system incidents and the SLA.

Another issue is rule redundancy. For example, let us consider the two predictive rules:

X. *PROC_CPU_TIME* $>$ 50% ***and*** *PROC_NAME* = 'Rtvscan'

Y. *PROC_CPU_TIME* $>$ 60% ***and*** *PROC_NAME* = 'Rtvscan'

Clearly, if an alert satisfies Rule Y, then it must satisfy Rule X as well. In other words, Rule Y is more specific than Rule X. If Rule Y has a lower accuracy than Rule X, then Rule Y is redundant given Rule X (but Rule X is not redundant given Rule Y). In our solution, we perform redundant rule pruning to discard the more specific rules with lower accuracies. The detailed algorithm is described in [TLP$^+$12].

Calculating Waiting Time for Each Rule

Waiting time is the duration by which tickets should be postponed if their corresponding alerts are classified as false. It is not unique for all monitoring situations. Since an alert can be covered by multiple predictive rules, we set up different waiting times for each of them. The waiting time can be transformed into two parameters in monitoring systems, the length of the polling interval with the minimum polling count [urlf]. For example, the situation described in Example 5 predicts false alerts about CPU utilization of 'Rtvscan.' We can also find another predictive rule as follows:

if *PROC_CPU_TIME* $>$ *50%* ***and*** *PROC_NAME = 'perl logqueue.pl',* ***then*** *this alert is false.*

The job of 'perl', however, is different from that of 'Rtvscan.' Their durations are not the same, and the waiting time will differ accordingly. In order to remove as many false alerts as possible , we set the waiting time of a selected rule as the longest duration of the transient alerts covered by it. For a selected predictive rule p, its waiting time is

$$wait_p = \max_{e \in \mathcal{F}_p} e.duration,$$

where $\mathcal{F}_p = \{e|e \in \mathcal{F}, isCovered(p,e) =' true'\}$, and $\mathcal{F}$ is the set of transient events. Clearly, for any rule $p \in \mathcal{P}$, $wait_p$ has a upper bound, $wait_p \leq delay_{max}$. Therefore, no ticket can be postponed for more than $delay_{max}$.

4.3 Eliminating False Negative

False negative alerts are the missing alerts that are not captured by the monitoring system due to some misconfiguration. Real-world IT infrastructures are often over-monitored. False negative alerts are much fewer than false positive alerts. Since the number of false negative alerts is quite small, we only focus on the methodologies for discovering them with their corresponding monitoring situations. The system administrators can easily correct the misconfiguration by referring the results. The false negative tickets are recorded by the system administrators in the manual tickets. Each manual ticket consists of several textual messages that describe the detailed problem. In addition to system fault issues, manual tickets also track many other customer requests such as asking for resetting database passwords, installing a new web server and so on. The customer request is the majority of the manual tickets. In our system the work for false negative alerts is to find out those monitoring related tickets among all manual tickets. This problem is formed as a binary text classification problem. Given an incident ticket, our method classifies it into "1" or "0", where "1" indicates this ticket is a false negative ticket, otherwise it is not. For each monitoring situation, we build a binary text classifier.

There are two challenges for building the classification model. First, the manual ticket data is highly imbalanced since most of the manual tickets are customer requests and only very few are false negative tickets. Figure 4.3 shows various system situation issues in two manual ticket sets. This manual ticket set is collected from a large customer account in IBM IT service centers. The first month has 9854 manual tickets and the second month has 10109 manual tickets overall. As shown in this figure, only about 1% manual tickets are

false negatives. Second, labeled data is very limited. Most system administrators are only

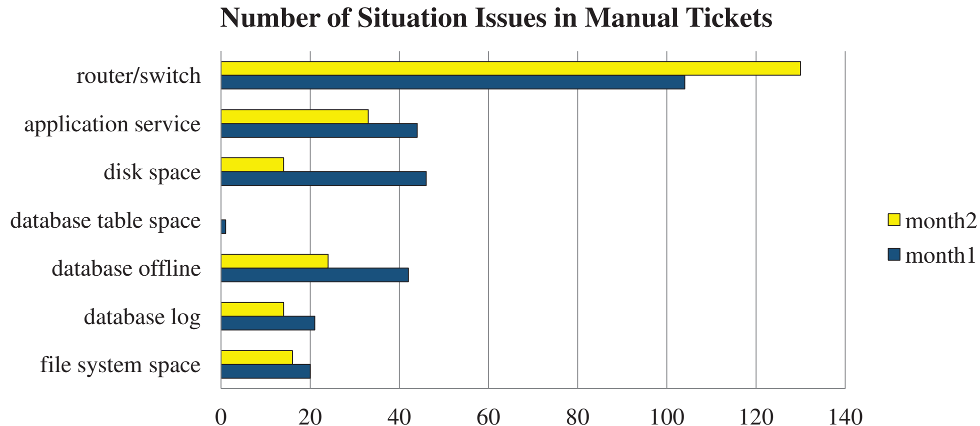

Figure 4.3: Number of Situation Tickets

working on some parts of incident tickets. Only a few experts can label all tickets.

4.3.1 Selective Ticket Labeling

It is time-consuming for human experts to scan all manual tickets and label their classes for training. In our approach, we only select a small proportion of tickets for labeling. A naive method is randomly selecting a subset of the manual tickets as the training data. However, the selection is crucial to the highly imbalanced data. Since the monitoring related tickets are very rare, the randomly selected training data would probably not contain any monitoring related ticket. As a result, the classification model cannot be trained well. On the other hand, we do not know which ticket is related to monitoring or not before we obtain the tickets' class labels. To solve this problem, we utilize domain words in system management for the training ticket selection. The domain words are some proper nouns or verbs that indicate the scope of the system issues. For example, everyone uses "DB2" to indicate the concept of IBM DB2 database. If a ticket is about the DB2 issue, it must contain the word "DB2". "DB2" is a domain word. There are not many variabilities for the concepts described by the domain words. Therefore, those domain words is helpful

to reduce the ticket candidates for labeling. Table 4.2 lists examples of the domain words with their corresponding situations. The domain words can be obtained from the experts or related documents.

Table 4.2: Domain Word Examples

Situation Issue	Words
DB2 tablespace Utilization	DB2, tablespace
File System Space Utilization	space,file
Disk Space Capacity	space,drive
Service Not Available	service,down
Router/Switch Down	router

In the training ticket selection, we first compute the relevance score of each manual ticket and ranks all the tickets based on the score, and then select the top k tickets in the ranked list, where k is a predefined parameter. Given a ticket T, the relevance score is computed as follows:

$$score(T) = \max\{|w(T) \cap M_1|, ..., |w(T) \cap M_l|\},$$

where $w(T)$ is the word set of ticket T, l is the number of predefined situations, M_i is the given domain word set for the i-th situation, $i = 1, ..., l$. Intuitively, the score is the largest number of the common words between the ticket and the domain words.

In dual supervision learning[SM08], the domain words are seen as the labeled features , which can also be used in active learning for selecting unlabeled data instances. But in our application, we have only the positive features but no negative features, and the data is highly imbalanced. Therefore, the uncertainty-based approach and the density-based approach in active learning are not appropriate for our system.

Classification Model Building

The situation ticket is identified by applying a SVM classification model [TSK05] on the ticket texts. For training this model, we have two types of input data: 1) the selectively

labeled tickets, and 2) the domain words. To utilize the domain words, we treat each domain word as a *pseudo-ticket* and put all *pseudo-tickets* into the training ticket set. To deal with the imbalanced data, the minority class tickets are over-sampled until the number of positive tickets is equal to the number of the negative tickets [CBHK02]. Figure 4.4 shows the flow chart for building the SVM classification model.

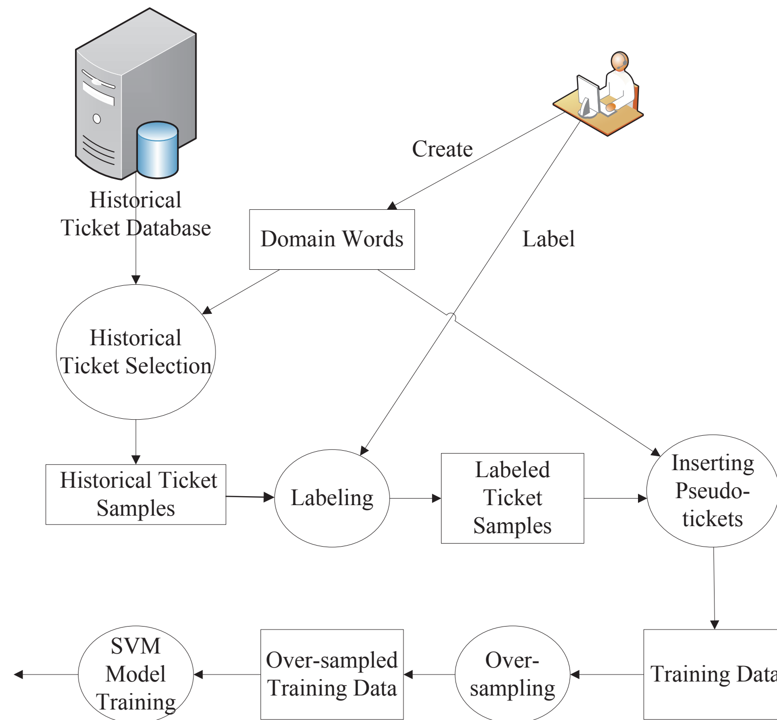

Figure 4.4: Flow Chart of Classification Model

4.4 Evaluation

This section presents empirical studies for our system. The system and the analysis results have been deployed for several customer accounts of IBM IT service. The empirical studies have two types of evaluation. The first type of evaluation is on the collected historical data to validate the performance of the algorithms. The second one is on the production severs of IBM customers to validate the effectiveness on real IT infrastructures.

4.4.1 Evaluation on Historical Data

Our system is developed by Java 1.6. This testing machine is Windows XP with Intel Core 2 Duo CPU 2.4GHz and 3GB of RAM. Experimental monitoring events and tickets are

Table 4.3: Data Summary

Data Set	$\|\mathcal{D}\|$	N_{non}	# Attributes	# Situations	# Nodes
Account1	50,377	39,971	1082	320	1212

collected from production servers of the IBM Tivoli Monitoring system [urle], summarized in Table 4.3. The data set of each account covers a period of 3 months. $|\mathcal{D}|$ is the number of events that generated tickets in the ticketing systems. N_{non} is the number of false events in all ticketed events. # Attributes is the total number of attributes of all events. # Situations is the number of monitoring situations. # Nodes is the number of monitored servers. In addition to the auto-generated tickets, we also collect manual tickets from two months. The first month has 9584 manual tickets. The second month has 10109 manual tickets.

Evaluation for False Positives

There are two performance measures:

- FP: The number of false tickets eliminated.
- FD: The number of real tickets postponed.

To achieve a better performance, a system should have a larger FP with a smaller FD. We split each data set into the training part and the testing part. "Testing Data Ratio" is the fraction of the testing part in the data set, and the rest is the training part. For example, "Testing Data Ratio=0.9" means that 90% of the data is used for testing and 10% is used for training. All FP and FD are only evaluated for the testing part.

Based on the experience of the system administrators, we set $delay_{max} = 360$ minutes for all monitoring situations. Figures 4.5, 4.6 and 4.7 present the experimental results. Our

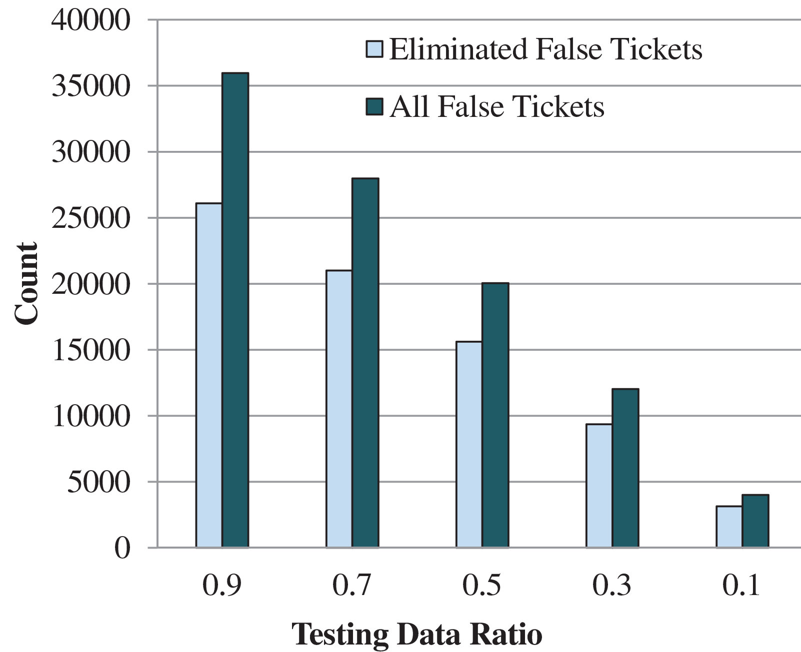

Figure 4.5: Eliminated False Positive Tickets

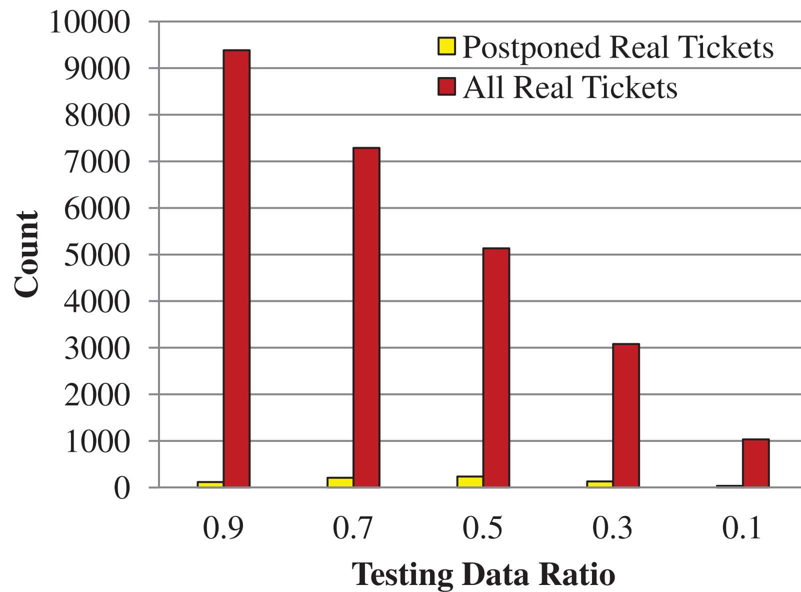

Figure 4.6: Postponed Real Tickets

method eliminates more than 75% of the false alerts and only postpones less than 3% of the real tickets.

Since most alert detection methods cannot guarantee no false negatives, we only compare our method with the idea mentioned in [CMB08], $Revalidate$, which revalidates the

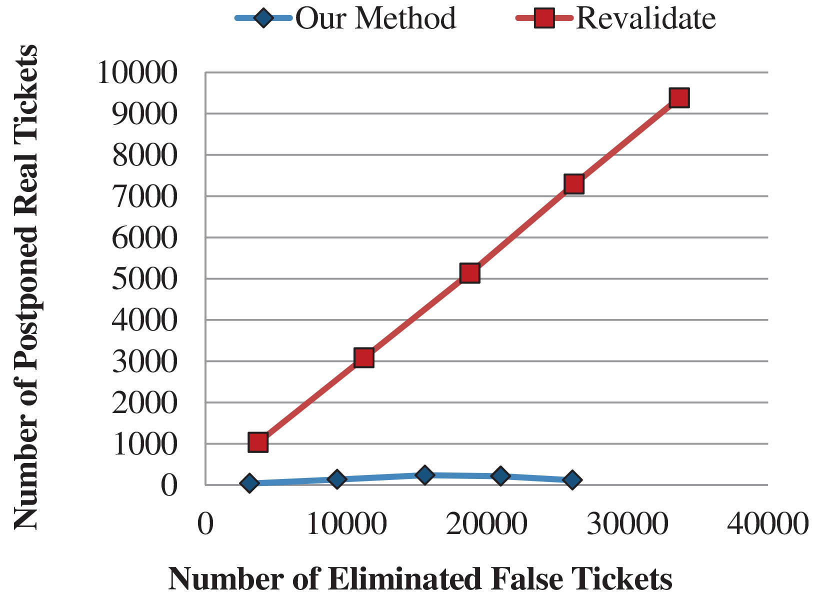

Figure 4.7: Comparison with Revalidate Method

status of events and postpones all tickets. $Revalidate$ has only one parameter, the postponement time, which is the maximum allowed delay time $delay_{max}$. Figure 4.7 compares the respective performance of our method and $Revalidate$, where each point corresponds to a different test data ratio. While $Revalidate$ is clearly better in terms of elimination of false alerts, it postpones all real tickets, the postponement volume being 1000 to 10000 times larger than our method.

Tables 4.4 lists several discovered predictive rules for false alerts, where $wait_p$ is the delay time for a rule, FP_p is the number of false alerts eliminated by a rule in the testing data, and FD_p is the number of real tickets postponed by a rule in the testing data.

Table 4.4: Sampled Rules for Account2 with Testing Data Ratio = 0.3

Situation	Rule Condition	$wait_p$	FP_p	FD_p
cpu_xuxw_std	N/A	355 min	7093	5
monlog_3ntw_std	current_size_64 $>=$ 0 **and** record_count $>=$ 737161	80 min	23	0
svc_3ntw_vsa_std	binary_path = R:\IBMTEMP\VSA\VSASvc_Cli.exs	30 min	27	0
fss_xuxw_std	inodes_used $<=$ 1616 **and** mount_point_u = /logs	285 min	12	2
fss_xuxw_std	inodes_used $<=$ 1616 **and** sub_origin = /logs	285 min	12	2

Evaluation for False Negatives

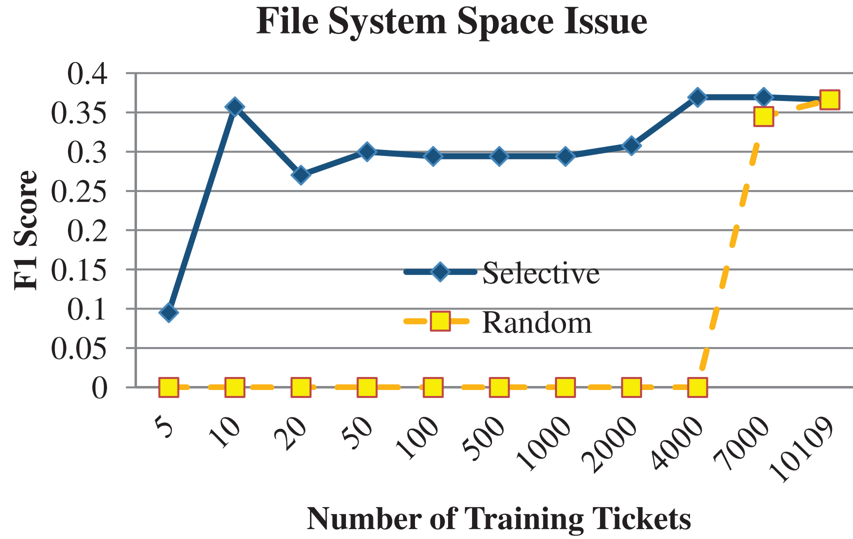

Figure 4.8: Accuracy of Situation Discovery for File System Space Alert

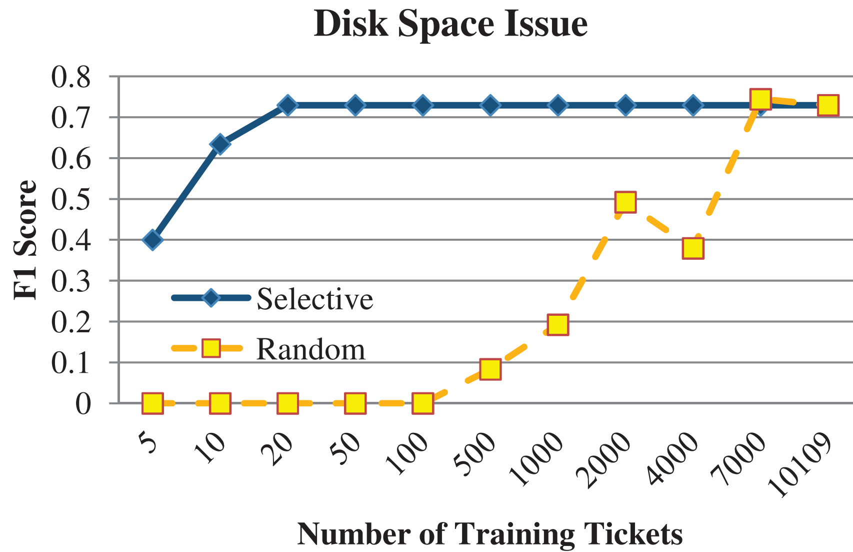

Figure 4.9: Accuracy of Situation Discovery for Disk Space Alert

The effectiveness is evaluated by the accuracy of the situation discovery. The accuracy is measured by precision, recall and F1score, which are the standard accuracy metrics in classification problems [SM84]. We use one month's tickets as the training data, and the

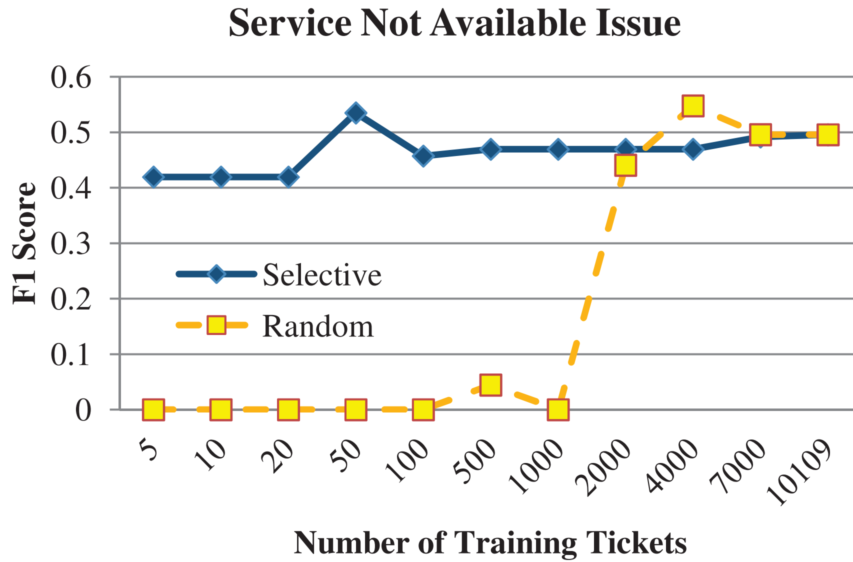

Figure 4.10: Accuracy of Situation Discovery for Service not available

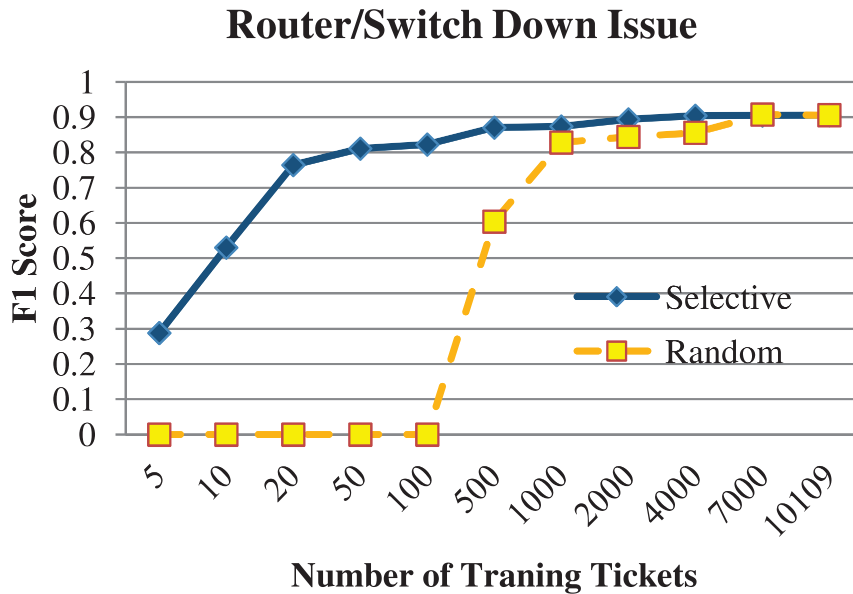

Figure 4.11: Accuracy of Situation Discovery for Router/switch down

other month's tickets as the testing data. We first test the accuracy of the word-match method. The words are predefined in Table 4.5.

Figures 4.8 to 4.11 show the tested F1 scores [TSK05] of four monitoring situations about file system space issue, disk space issue, service availability and router/swith issues. Our method is denoted as "Selective", the second baseline method is denoted as "Random".

Table 4.5: Accuracy of the word-match method

Situation	Words	Precision	Recall	F1Score
File System Space	space,file	0.0341	0.8	0.0654
Disk Space	space,drive	0.1477	0.9565	0.2558
Service Not Available	service,down	0.1941	0.75	0.3084
Router/Switch Down	router	0.6581	0.7404	0.6968

The "Random" method randomly selects a subset of manual tickets as the training data for building the SVM model. The domain words for our "Selective" method are shown in Table 4.5. As shown by those figures, the "Random" method can only achieve the same accuracy of our method when the number of training tickets is large (above 5000). This is because the real situation tickets are in the minority of the training data set. The training tickets in "Random" cannot capture real situation tickets unless the training data is large. If the training data is large, however, labeling would be time-consuming for humans.

4.4.2 Evaluation on Production Servers

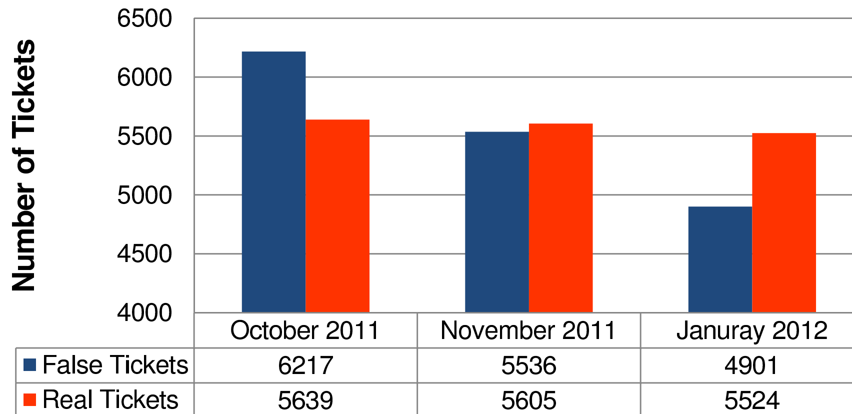

Figure 4.12: Ticket Volume Changes on Account1

The analytic results have been deployed into several customer accounts of IBM IT services. The service teams track the changes on those customer accounts after the deployment. Figure 4.12 shows the deployed results of one account in three months. Account1 is the account that provides the historical data in the previous evaluation of this paper. This

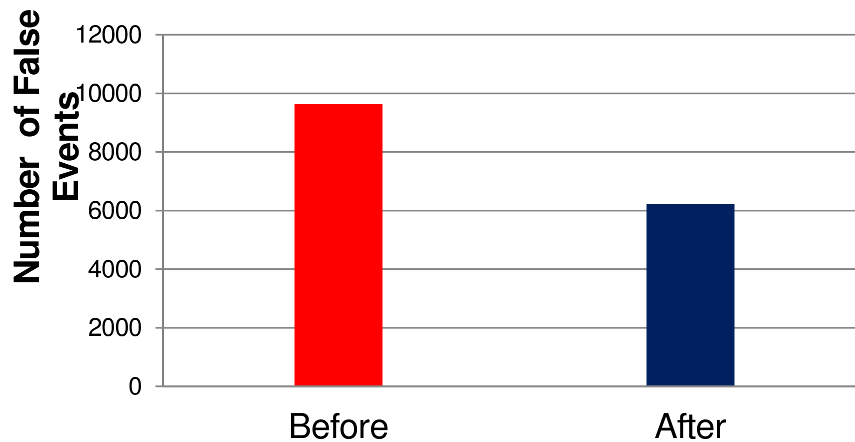

Figure 4.13: Event Volume Changes on Account2

customer account is a large financial company in the United States. Its production servers are used mainly to support financial investments. The deployment of our work is a step-by-step approach. In the first month, the deployment is only on a small group of testing and development machines. Then it spreads to a wide area of its IT environment. Hence, in Figure 4.12, the effect of our work gradually appears in three months. Although this company's IT infrastructure changes every day, compared to the changes of real tickets, the reduction for the false tickets is still obvious. The total number of false positive tickets has been reduced by 21%.

Figure 4.13 shows the evaluation results for another customer account of IBM IT services. They compare the number of false alerts before deployment and after deployment. Before the deployment, this account has many inappropriate CPU and networking monitoring situations, which produce a large number of false alerts every day. By adjusting those monitoring situations according to our analysis reports, more than 30% of the false alerts are eliminated.

Table 4.6 shows a sample list of discovered false negative tickets with their monitoring situations on Account1. For privacy issues, the administrators' names and the server names are replaced by "xxx". Most of the false negative tickets are caused by some new servers or new databases that are not added into the configuration of monitoring systems. When

Table 4.6: False Negative Tickets

Situation	Ticket
dsp_3ntc_std	Please clear space from E drive xxxx-fa-ntfwwfdb Please clear space from E drive xxxx-fa-ntfwwfdb.it is having 2 MB free...
fss_rlzc_std	/opt file system is is almost full on us97udb010ampsb Hi Team@/opt file system is almost full. Please clear some space /home/dbasso¿df -h /optFilesystem...
svc_3ntc_std	RFS101681 E2 Frontier all RecAdmin services are down Frontier RecAdmin services are not running on the batch server Kindly logon to the server : xxx.xxx.155.183/xxx ...
dboffln_3oqc_std	DB2 is not connectable from xxxxx Hi Team@Can you please look into why we are unable to connect to Porfolio XRef DB.Server : xxxx12DB Instance : sec_mastId : ipxrbtchWhile...
dboffln_3oqc_std	Unable to login to DB server Hi Team@We had raised a request 131443 for access on the E1 and E2 serversE1 - Full access@ to read/write/execute programs Hostname Server xxxxx xxx.xxx.147.194

the new servers and new databases incur system faults or issues, only the database administrators or storage administrators discover them and create the manual tickets. The false negative tickets are quite few in real production servers, so there is no obvious impact on the volume change after the deployment.

4.5 Summary

This chapter first describes the problem of false positive and false negative in most real-world monitoring systems. Based on a large collection of historical monitoring events and tickets from several service providers, we investigate the main reasons of this problem, and propose two data-driven approaches to optimize the automatic system monitoring in large IT infrastructures. By combing the system event data and ticket data collected from IT service centers, the proposed approaches reduce the number of false positive (non-actionable) alerts and the number of false negative (missing) alerts for the automatic monitoring system. It minimizes the cost of providing effective and reliable means for problem detection. This work has been implemented as a system in the IBM IT service management platform and deployed in several IBM service centers. This system is used periodically to refine and adjust monitoring situations after a system has gone through a change, thus helping to enhance the overall reliability in IT service management.

CHAPTER 5

SYSTEM DIAGNOSIS

System diagnosis is performed by humans. The objective of our work in this chapter is the provide semi-automatic approaches for helping humans accomplish this task. The effectiveness and efficiency of the system diagnosis can be improved by various aspects of methodologies, including the system design, system optimization and so on. This dissertation only focuses on data-driven techniques for assisting system administrators. This chapter is divided by three subproblems that we may encounter in traditional system diagnosis. We first introduce the problem of discovering temporal dependencies with time lags, which is often used to find the dependency among system components or the correlations of monitoring situations. Then, we present several proposed recommendation algorithms for automatically recommending relevant incident tickets with their resolutions for incoming tickets. The system administrators can correlate the similar system issues happening before and find the best practices for resolving same type of issues. Meanwhile, the proposed algorithms take into account the falsity of the tickets to avoid recommending misleading information for humans. Finally, a novel indexing technique is developed for facilitating the similarity search over textual event sequences.

5.1 Discovering Temporal Dependencies with Time Lags

Temporal dependencies are often used for prediction. Let A and B be two types of items, a temporal dependency for A and B, written as $A \rightarrow B$, denotes that the occurrence of B depends on the occurrence of A. The dependency indicates that an item A is often followed by an item B. Let $[t_1, t_2]$ be the lag interval of the lag for two dependent A and B. For example, in system management, disk capacity alert and database alert are two item types. When the disk capacity is full, the database engine often raises a database alert in

the next 5 to 6 minutes as shown in Figure 5.1. Hence, the disk capacity has a temporal dependency with the database. $[5\mathtt{min}, 6\mathtt{min}]$ is the lag interval between the two dependent system alerts. $disk_capacity_alert \rightarrow_{[5\mathtt{min},6\mathtt{min}]} database_alert$ describes the temporal dependency with the associated lag interval. In this dissertation, we study the problem of finding appropriate lag intervals for two dependent item types.

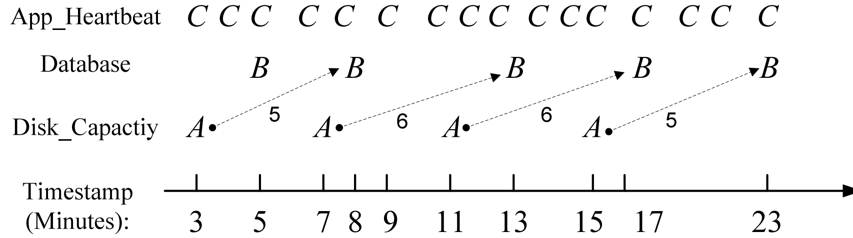

Figure 5.1: Lag Interval for Temporal Dependency

In Figure 5.1, $[5\mathtt{min}, 6\mathtt{min}]$ is the predicted time range, indicating when a database alert occurs after a disk capacity alert is received. Furthermore, the associated lag interval characterizes the cause of a temporal dependency. For example, if the database is writing a huge temporal log file which is larger than the disk free space, the database alert is immediately raised in $[0\mathtt{min}, 1\mathtt{min}]$. But if the disk free capacity is consumed by other applications, the database engine can only detect this alert when it runs queries. The associate time lags in such a case would be larger than 1 minute.

Previous work for discovering temporal dependencies does not consider interleaved dependencies [LM04] [BO07] [MTV97]. For $A \rightarrow B$, they assume that an item A can only have a dependency with its first following B. However, it is possible that an item A has a dependency with any following B. For example, in Figure 5.1, the time lag for two dependent A and B is 5 to 6 minutes, but the time lag for two adjacent A's is only 4 minutes. All A's have a dependency with the second following B, not the first following B. Hence, the dependencies among these dependent A and B are interleaved. For two item types, the numbers of time stamps are both $O(n)$, The number of possible time lags is $O(n^2)$. Thus, the number of lag intervals is $O(n^4)$. The challenge of our work is how

to efficiently find appropriate lag intervals over the $O(n^4)$ candidates. Other preliminary work has been discussed in Section 2.3.

5.1.1 Algorithms

Given an item sequence $S = x_1x_2...x_N$, x_i denotes the type of the i-th item, and $t(x_i)$ denotes the time stamp of x_i, $i = 1, 2, ..., N$. Intuitively, if there is a temporal dependency $A \rightarrow_{[t_1,t_2]} B$ in S, there must be a lot of A's that are followed by some B with a time lag in $[t_1, t_2]$. Let $n_{[t_1,t_2]}$ denote the observed number of A's in this situation. For instance, in Figure 5.1, every A is followed by a B with a time lag of 5 or 6 minutes, so $n_{[5,6]} = 4$. Only the second A is followed by a B with a time lag of 0 or 1 minute, so $n_{[0,1]} = 1$. Let $r = [t_1, t_2]$ be a lag interval. One question is that, what is the minimum required n_r that we can utilize to identify the dependency of A and B with r. In this example, the minimum required n_r cannot be greater than 4 since the sequence has at most 4 A's. However, if let $r = [0, +\infty]$, we can easily have $n_r = 4$. [MH01b] proposes a chi-square test approach to determine the minimum required n_r, where the chi-square statistic measures the degree of the independence by comparing the observed n_r with the expected n_r under the independent assumption. The null distribution of the statistic is approximated by the chi-squared distribution with 1 degree of freedom. Let χ^2_r denote the chi-square statistic for n_r. A high χ^2_r indicates the observed n_r in the given sequence cannot be explained by randomness. The chi-square statistic is defined as follows:

$$\chi^2_r = \frac{(n_r - n_A P_r)^2}{n_A P_r (1 - P_r)}, \tag{5.1}$$

where n_A is the number of A's in the data sequence, P_r is the probability of a B appearing in r from a random sequence. Hence, $n_A P_r$ is the expected number of A's that are followed by some B with a time lag in r. $n_A P_r(1 - P_r)$ is the variance. Note that the random sequence

should have the same sampling rate for B as the given sequence S. The randomness is only for the positions of B items. It is known that a random sequence usually follows the Poisson process, which assumes the probability of an item appearing in an interval is proportional to the length of the interval [Ros95]. Therefore,

$$P_r = |r| \cdot \frac{n_B}{T}, \tag{5.2}$$

where $|r|$ is the length of r, $|r| = t_2 - t_1 + w_B$, w_B is the minimum time lag of two adjacent B's, $w_B > 0$, and n_B is the number of B's in S. For lag interval r, the absolute length is $t_2 - t_1$. w_B is added to $|r|$ because without w_B when $t_1 = t_2$, $|r| = 0$, P_r is always 0 no matter how large the n_B is. As a result, χ^2_r would be overestimated. In reality, the time stamps of items are discrete samples and w_B is the observed sampling period for B items. Hence, the probability of a B appearing in $t_2 - t_1$ time units is equal to the probability of a B appearing in $t_2 - t_1 + w_B$ time units.

The value of χ^2_r is defined in terms of a confidence level. For example, 95% confidence level corresponds to $\chi^2_r = 3.84$. Based on Eq.(5.1), the observed n_r should be greater than $\sqrt{3.84 n_A P_r (1 - P_r)} + n_A P_r$. Note that we only care positive dependencies, so

$$n_r - n_A P_r > 0. \tag{5.3}$$

To ensure a discovered temporal dependency fits the entire data sequence, ***support*** [AS94] [SA96b] [MH01b] is used in our work. For $A \to_r B$, the support $supp_A(r)$ (or $supp_B(r)$) is the number of A's (or B's) that satisfy $A \to_r B$ divided by the total number of items N. ***minsup*** is the minimum threshold for both $supp_A(r)$ and $supp_B(r)$ specified by the user [SA96b] [MH01b]. Based on the two minimum thresholds χ^2_c and $minsup$, Definition 5.1.1 defines the qualified lag interval that we try to find.

Definition 5.1.1. *Given an item sequence S with two item types A and B, a lag interval $r = [t_1, t_2]$ is qualified if and only if $\chi^2_r > \chi^2_c$, $supp_A(r) > minsup$ and $supp_B(r) > minsup$, where χ^2_c and $minsup$ are two minimum thresholds specified by the user.*

We first develop a straightforward algorithm, a *brute-force* algorithm. Then, we propose two new algorithms, *STScan* and *STScan**, which are much more efficient than the *brute-force* algorithm. A lower bound of the time complexity for finding qualified lag intervals is also studied in this work. Finally, we discuss how to incorporate the domain knowledge to speed up the algorithms.

The Brute-Force Algorithm

To find all qualified lag intervals, a straightforward algorithm is to enumerate all possible lag intervals, compute their χ^2_r and supports, and then check whether they are qualified or not. This algorithm is called *brute-force*. Clearly, its time cost is very large. Let n be the number of distinct time stamps of S, $r = [t_1, t_2]$. The numbers of possible t_1 and t_2 are $O(n^2)$, and then the number of possible r is $O(n^4)$. For each lag interval, there is at least $O(n)$ cost to scan the entire sequence S to compute the χ^2_r and the supports. Therefore, the overall time cost of the *brute-force* algorithm is $O(n^5)$, which is not affordable for large data sequences.

The STScan Algorithm

To avoid re-scanning the data sequence, we develop a sorted table based algorithm. A sorted table is a sorted linked list with a collection of sorted integer arrays. Each entry of the linked list is attached to two sorted integer arrays. Figure 5.2 shows an example of the sorted array. In our algorithm, we store every time lag $t(x_j) - t(x_i)$ into each entry of linked list, where $x_i = A$, $x_j = B$, i, j are integers from 1 to N. Two arrays attached to the entry $t(x_j) - t(x_i)$ are the collections of i and j. In other words, the two arrays are the

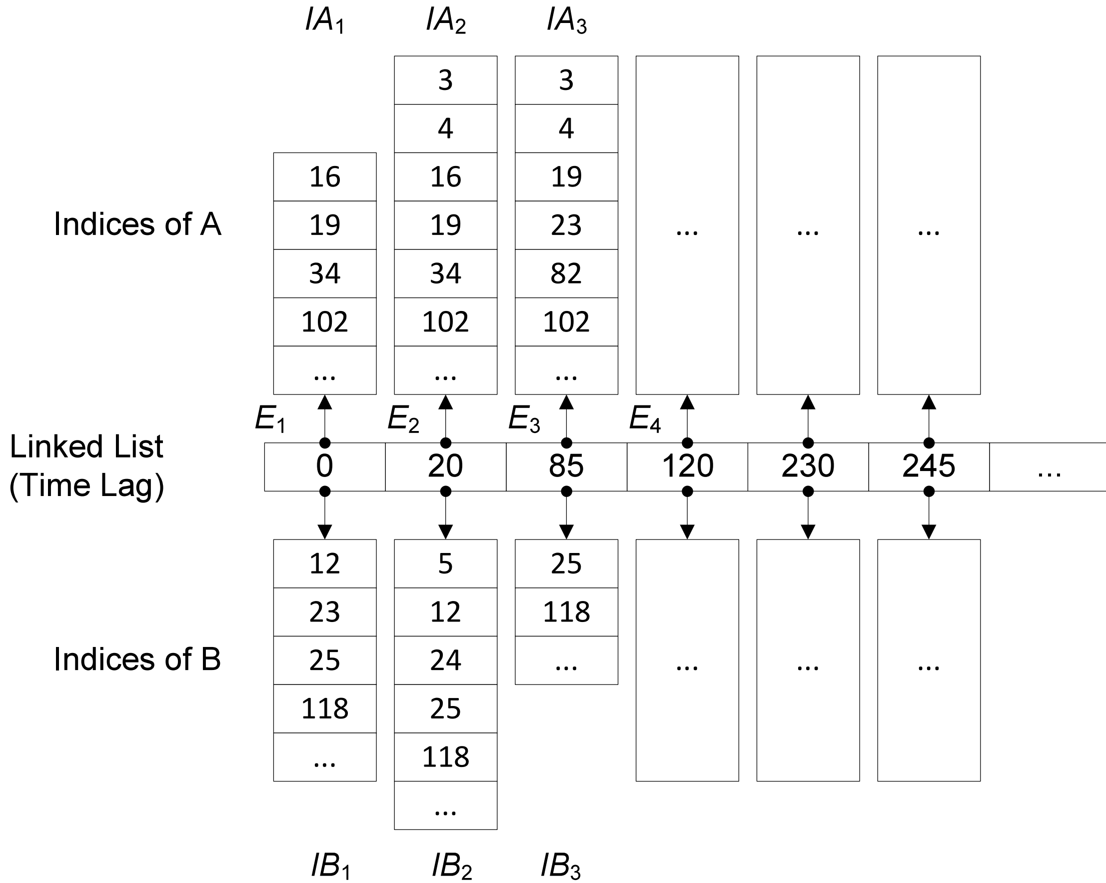

Figure 5.2: Sorted Table

indices of A's and B's. Let E_i denote the i-th entry of the linked list and $v(E_i)$ denote the time lag stored at E_i. IA_i and IB_i denote the indices of A's and B's that are attached to E_i. For example in Figure 5.2, $x_3 = A$, $x_5 = B$, $t(x_5) - t(x_3) = 20$. Since $v(E_2) = 20$, IA_2 contains 3 and IB_2 contains 5. Any feasible lag interval can be represented as a subsegment of the linked list. For example in Figure 5.2, $E_2E_3E_4$ represents the lag interval $[20, 120]$.

To create the sorted table for a sequence S, each time lag between an A and a B is first inserted into a red-black tree. The key of the red-black tree node is the time lag, the value is the pair of indices of A and B. Once the tree is built, we traverse the tree in ascending order to create the linked list of the sorted table. In the sequence S, the number A and B are both $O(N)$, so the number of $t(x_j) - t(x_i)$ is $O(N^2)$. The time cost of creating the red-black tree is $O(N^2 \log N^2) = O(N^2 \log N)$. Traversing the tree costs $O(N^2)$. Hence,

the overall time cost of creating a sorted table is $O(N^2 \log N)$, which is the known lower bound of sorting $X + Y$ where X and Y are two variables [HB96]. The linked list has $O(N^2)$ entries, and each attached integer array has $O(N)$ elements, so it seems that the space cost of a sorted table is $O(N^2 \cdot N) = O(N^3)$. However, Lemma 5.1.2 shows that the actual space cost of a sorted table is $O(N^2)$, which is same as the red-black tree.

Lemma 5.1.2. *Given an item sequence S having N items, the space cost of its sorted table is $O(N^2)$.*

Proof. Since the numbers of A's and B's are both $O(N)$, the number of pairs (x_i, x_j) is $O(N^2)$, where $x_i = A$, $x_j = B$, $x_i, x_j \in S$. Every pair associated with three entries in the sorted table: the time stamp distance, the index of an A and the index of a B. Therefore, each pair (x_i, x_j) introduces 3 space cost. The total space cost of the sorted table is $O(3N^2) = O(N^2)$. □

Once the sorted table is created, finding all qualified lag intervals is scanning the subsegments of the linked list. However, the number of entries in the linked list is $O(N^2)$, so there are $O(N^4)$ distinct subsegments. Scanning all subsegments is still time-consuming. Fortunately, based on the minimum thresholds on the chi-square statistic and the support, the length of a qualified lag interval cannot be large.

Lemma 5.1.3. *Given two minimum thresholds χ^2_c and $minsup$, the length of any qualified lag interval is less than $\frac{T}{N} \cdot \frac{1}{minsup}$.*

Proof. Let r be a qualified lag interval. Based Eq.(5.1) and Inequality.(5.3), χ^2_r increases along with n_r. Since $n_r \leq n_A$,

$$\frac{(n_A - n_A P_r)^2}{n_A P_r (1 - P_r)} \geq \chi^2_r > \chi^2_c \implies P_r < \frac{n_A}{\chi^2_c + n_A}.$$

By substituting Eq. 5.2 to the previous inequality,

$$|r| < \frac{n_A}{\chi_c^2 + n_A} \cdot \frac{T}{n_B}.$$

Since $n_B > N \cdot minsup$, $\chi_c^2 > 0$, we have

$$|r| < \frac{T}{N} \cdot \frac{1}{minsup} = |r|_{max}.$$

□

$\frac{T}{N}$ is exactly the average period of items, which is determined by the sampling rate of this sequence. For example, in system event sequences, the monitoring system checks the system status for every 30 seconds and records system events into the sequence. The average period of items is 30 seconds. Therefore, we consider $\frac{T}{N}$ as a constant. $minsup$ is also a constant, so $|r|_{max}$ is a constant.

Algorithm *STScan* states the pseudocode for finding all qualified lag intervals. $len(ST)$ denotes the number of entries of the linked list in sorted table ST. This algorithm sequentially scans all subsegments starting with E_1, E_2, ..., $E_{len(ST)}$. Based on Lemma 5.1.3, it only scans the subsegment with $|r| < |r|_{max}$. To calculate the χ_r^2 and the supports, for each subsegment, it cumulatively stores the aggregate indices of A's and B's and the corresponding lag interval r. For each subsegment, $n_r = |IA_r|$, $supp_A(r) = |IA_r|/N$, $supp_B(r) = |IB_r|/N$.

Lemma 5.1.4. *The time cost of* STScan *is $O(N^2)$, where N is the number of items in the data sequence.*

Proof. For each entry E_{i+j} in the linked list, the time cost of merging IA_{i+j} and IB_{i+j} to IA_r and IB_r is $|IA_{i+j}| + |IB_{i+j}|$ by using a hash table. Let l_i be the largest length of the

Algorithm 2 *STScan* $(S, A, B, ST, \chi_c^2, minsup)$

Input: S : input sequence; A, B: two item types; ST : sorted table; χ_c^2: minimum chi-square statistic threshold; $minsup$: minimum support.
Output: a set of qualified lag intervals;

```
1:  R ← ∅
2:  Scan S to find w_B
3:  for i = 1 to len(ST) do
4:      IA_r ← ∅, IB_r ← ∅
5:      t_1 ← v(E_i)
6:      j ← 0
7:      while i + j ≤ len(ST) do
8:          t_2 ← v(E_{i+j})
9:          r ← [t_1, t_2]
10:         |r| ← t_2 − t_1 + w_B
11:         if |r| ≥ |r|_max then
12:             break
13:         end if
14:         IA_r ← IA_r ∪ IA_{i+j}
15:         IB_r ← IB_r ∪ IB_{i+j}
16:         j ← j + 1
17:         if |IA_r|/N ≤ minsup or |IB_r|/N ≤ minsup then
18:             continue
19:         end if
20:         Calculate χ_r^2 from |IA_r| and |r|
21:         if χ_r^2 > χ_c^2 then
22:             R ← R ∪ r
23:         end if
24:     end while
25: end for
26: return R
```

scanned subsegments starting at E_i. Let l_{max} be the maximum l_i, $i = 1, ..., len(ST)$. The total time cost is:

$$
\begin{aligned}
T(N) &= \sum_{i=1}^{len(ST)} \sum_{j=0}^{l_i-1} (|IA_{i+j}| + |IB_{i+j}|) \\
&\leq \sum_{i=1}^{len(ST)} \sum_{j=0}^{l_{max}-1} (|IA_{i+j}| + |IB_{i+j}|) \\
&\leq l_{max} \cdot \sum_{i=1}^{len(ST)} (|IA_i| + |IB_i|)
\end{aligned}
$$

$\sum_{i=1}^{len(ST)}(|IA_i| + |IB_i|)$ is exactly the total number of integers in all integer arrays. Based on Lemma 5.1.2, $\sum_{i=1}^{len(ST)}(|IA_i| + |IB_i|) = O(N^2)$. Then $T(N) = O(l_{max} \cdot N^2)$. Let $E_k...E_{k+l}$ be the subsegment for a qualified lag interval, $v(E_{k+i}) \geq 0$, $i = 0, ..., l$. The length of this lag interval is $|r| = v(E_{k+l_{max}}) - v(E_k) < |r|_{max}$, then $l_{max} < |r|_{max}$ and l_{max} is not depending on N. Assume Δ_E is the average $v(E_{k+1}) - v(E_k)$, $k = 1, ..., len(ST) - 1$, we obtain a tighter bound of l_{max}, i.e., $l_{max} \leq |r|_{max}/\Delta_E \leq \frac{T}{N \cdot \Delta_E} \cdot \frac{1}{minsup}$. Therefore, the overall time cost is $T(N) = O(N^2)$. $\square$

STScan* Algorithm

To reduce the space cost of *STScan* algorithm, we develop an improved algorithm *STScan** which utilizes the increment sorted table and sequence compression.

Lemma 5.1.2 shows the space cost of a complete sorted table is $O(N^2)$. Algorithm *STScan* sequentially scans the subsegments starting from E_1 to $E_{len(ST)}$, so it does not need to access every entry at every time. Based on this observation, we develop an incremental sorted table based algorithm with an $O(N)$ space cost. This algorithm incrementally creates the entries of the sorted table along with the subsegment scanning process.

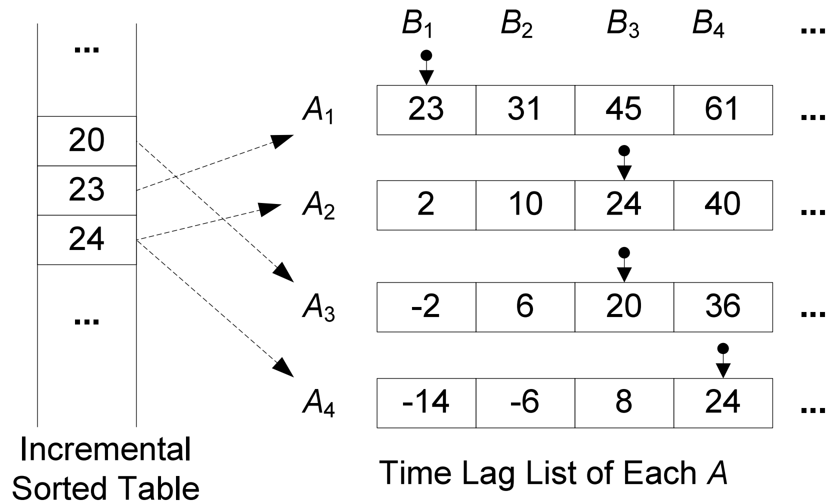

Figure 5.3: Incremental Sorted Table

The linked list of a sorted table can be created by merging all time lag lists of A's (Figure 5.3), where A_i and B_j denote the i-th A and the j-th B, $i, j = 1, 2, \ldots$. The j-th entry in the list of A_i stores $t(B_j) - t(A_i)$. The time lag lists of all A's are not necessary to be created in the memory because we only need to know $t(B_j)$ and $t(A_j)$. This can be done just with an indices arrays of all A's and all B's respectively. By using N-way merging algorithm, each entry of the linked list would be created sequentially. The indices of A's and B's attached to each entry are also recorded during the merging process. Base on Lemma 5.1.3, the length of a qualified lag interval is at most $|r|_{max}$, therefore, we only keep track of the recent l_{max} entries. The space cost for storing l_{max} entries is at most $O(l_{max} \cdot N) = O(N)$. A heap used by the merging process costs $O(N)$ space. Then, the overall space cost of the incremental sorted table is $O(N)$. The time cost of merging $O(N)$ lists with total $O(N^2)$ elements is still $O(N^2 \log N)$.

In many real-world applications, some items may share the same time stamp since they are sampled within the same sampling cycle. To save the time cost, we compress the original S to another compact sequence S'. At each time stamp t in S, if there are k items of type I, we create a triple (I, t, k) into S', where k is the cardinality of this triple. To handle S', the only needed change of our algorithm is that the $|IA_r|$ and $|IB_r|$ become the total cardinalities of triples in IA_r and IB_r respectively. Clearly, S' is more compact than S. S' has $O(n)$ triples, where n is the number of distinct time stamps of S, $n \leq N$. Creating S' costs an $O(N)$ time complexity. By using S', the time cost of ***STScan**** becomes $O(N + n^2 \log n)$ and the space cost of the incremental sorted table becomes $O(n)$.

For analyzing large sequences, an $O(n)$ or $O(n \log n)$ algorithm is needed. However, we find that the time complexity of any algorithm for our problem is at least $O(n^2)$ (Lemma 5.1.5). The proof is to reduce the 3SUM$'$ problem to our problem, and the 3SUM$'$ has no $o(n^2)$ solution [GO95]. To answer whether $O(n^2)$ is the tightest lower bound or not, a further study is needed.

Lemma 5.1.5. *Finding a qualified lag interval cannot be solved in $o(n^2)$ in the worst case, where n is the number of distinct time stamps of the given sequence.*

Proof. Assume that an algorithm $\mathbf{P}$ can find a qualified lag interval in $o(n^2)$ in any case, we can construct an algorithm to solve the 3SUM′ problem in $o(n^2)$ as follows. Given three sets of integers X, Y, and Z such that $|X| + |Y| + |Z| = n$, we construct a compressed sequence S' of items which only has two item types A and B as follows:

1. For each x_i in X, create an A at time stamp x_i.
2. For each y_i in Y, create a B at time stamp y_i.
3. For each z_i in Z, create $n + 1$ A's at time stamp $\beta(i + 1) + z_i$ and $n + 1$ B's at time stamp $\beta(i + 1)$, where β is the diameter of set $X \cup Y$, which is the largest integer minus the smallest integer in $X \cup Y$.

Only the lag intervals created from z_i have $n_r \geq n + 1$. If there are three integers $y_j \in Y$, $x_k \in X$, $z_i \in Z$ such that $y_j - x_k = z_i$, the lag interval of z_i must have $n_r \geq n+2$. Then, we substitute $n_r = n + 2$ into Eq. 5.1 to find the appropriate threshold χ^2_c, and call algorithm $\mathbf{P}$ to find all z_i that have $n_r \geq n + 2$. By filtering out the situations of $y_j - y_k = z_i$ and $x_j - x_k = z_i$, we can obtain the desired three integers such that $y_j - x_k = z_i$ if they exist. S' has at most $2n$ distinct time stamps. The time cost of creating S' is $O(2n) = O(n)$. $\mathbf{P}$ is an $o(n^2)$ algorithm. Filtering the result of $\mathbf{P}$ is $O(n)$ since $|Z| \leq n$. Therefore, the overall solution for the 3SUM′ problem is $O(n) + o(n^2) + O(n) = o(n^2)$. However, it is believed that the 3SUM′ problem has no $o(n^2)$ solution [GO95]. Therefore, $\mathbf{P}$ does not exist. □

5.1.2 Evaluation

This section presents our empirical study of discovering lag intervals on both synthetic data sets and real data sets in terms of the effectiveness and efficiency.

Experimental Platform and Algorithms

All comparative algorithms are implemented in Java 1.6 platform. Table 5.1 summarizes our experimental environment. At present, the most dedicated algorithm for finding lag

Table 5.1: Experimental Machine

OS	CPU	bits	Memory	JVM Heap Size
Linux 2.6.18	Intel Xeon(R) @ 2.5GHz, 8 core	64	16G	12G

intervals is the inter-arrival clustering method [LM04] [MH01b], denoted by *inter-arrival*. For $A \rightarrow B$, an inter-arrival is the time lag of an A to its first following B. A dense cluster created from all inter-arrivals indicates its time lag frequently appears in the sequence. Thus, a qualified lag interval is probably around this time lag. This algorithm is very efficient and only has a linear time cost, however, it does not consider the interleaved dependencies. We also implement the four algorithms, *brute-force*, *brute-force**, *STScan* and *STScan**, to compare with in this experiment. *brute-force** is the improved version of *brute-force* which utilizes the pruning strategy about $|r|_{max}$ mentioned in Lemma 5.1.3. For each test, we enumerate all pairwise temporal dependencies for discovering the qualified lag intervals.

Synthetic Data

The synthetic data consists of 7 data sequences. Each sequence is first generated from a random item sequence with 8 item types, denoted by I_1,...,I_8. The average sample period of items is 100. Three predefined temporal dependencies are randomly embedded into each random sequence and shown in Table 5.2. For each temporal dependency $I_i \rightarrow_{[t_1,t_2]} I_j$, we first randomly choose an item x_i and an integer $t \in [t_1, t_2]$, and then let $x_i = I_i$ and the item at $t(x_i) + t$ be I_j. We repeat this process until $\chi^2_{[t_1,t_2]}$ and the support are greater than the specified thresholds. Note that the time lags in these lag intervals are larger than the

Table 5.2: Embedded Temporal Dependencies

Embedded Temporal Dependencies	Support
$I_1 \rightarrow_{[400,500]} I_2$	0.1
$I_2 \rightarrow_{[1000,1100]} I_3$	0.12
$I_4 \rightarrow_{[5500,5800]} I_5$	0.15

average sample period of items, so all three temporal dependencies are very likely to be interleaved dependencies.

The effectiveness of the algorithm result is validated by comparing the discovered results with the embedded lag intervals and measured by the recall [TSK05]. We do not care the precision because every algorithm can achieve the 100% precision if this algorithm is correct. We let $\chi^2_c = 10.83$ which represents a 99.9% confidence level, $minsup = 0.1$. There is no surprise that all the algorithms proposed in this paper, *brute-force*, *brute-force**, *STScan* and *STScan**, find all the embedded lag intervals since they scan the entire space of the lag interval. Thus, the recalls of these methods are 1.0. The parameter δ of *inter-arrival* is varied from 1 to 2000. However, *inter-arrival* does not find any qualified lag interval in the synthetic data and its recall is 0. The reason is that, the qualified lag intervals are [400,500], [1000,1100] and [5500,5800], but most inter-arrival times in the sequence are close to 100. Thus, *inter-arrival* can only probe the lag intervals around 100.

The empirical efficiency is evaluated by the CPU running time (Figure 5.4). *inter-arrival* is a linear algorithm, so it runs much faster than other algorithms. The running time of the *brute-force* algorithm increases extremely fast so that it can only handle very tiny data sets. By adding the pruning strategy about $|r|_{max}$ to *brute-force*, the *brute-force** algorithm runs a little bit faster than the *brute-force* algorithm, but it still can only handle small data sets. *STScan** compresses the sequence before the lag interval discovering, therefore, *STScan** is a little bit more efficient than *STScan*.

STScan has not finish the tests for larger data sets because it runs out of memory. Table 5.4 lists the approximate peak numbers of allocated objects in Java heap memory (not including the data sequence). It confirms Lemma 5.1.2 that the sorted table takes an $O(N^2)$

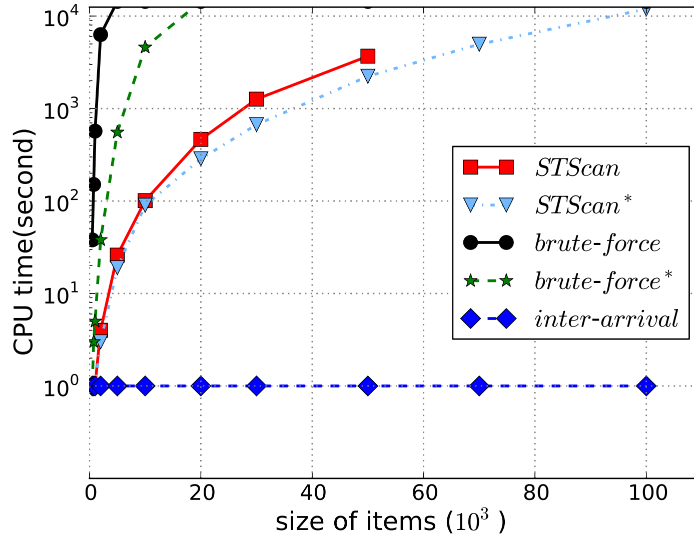

Figure 5.4: Runtime on Synthetic Data

Table 5.3: Discovered Temporal Dependencies with Lag Intervals

Data set	Dependency	χ^2_r	$support$
Account1	$MSG_Plat_APP \rightarrow_{[3600,3600]} MSG_Plat_APP$	$\geq$ 1000.0	0.07
	$Linux_Process \rightarrow_{[0,96]} Process$	134.56	0.05
	$SMP_CPU \rightarrow_{[0,27]} Linux_Process$	978.87	0.06
	$AS_MSG \rightarrow_{[102,102]} AS_MSG$	$\geq$ 1000.0	0.08
Account2	$TEC_Error \rightarrow_{[0,1]} Ticket_Retry$	$\geq$ 1000.0	0.12
	$Ticket_Retry \rightarrow_{[0,1]} TEC_Error$	$\geq$ 1000.0	0.12
	$AIX_HW_ERROR \rightarrow_{[25,25]} AIX_HW_ERROR$	282.53	0.15
	$AIX_HW_ERROR \rightarrow_{[8,9]} AIX_HW_ERROR$	144.62	0.24

Table 5.4: Space Cost on Synthetic Data

Data size / **Algorithm**	10^3	10×10^3	50×10^3	100×10^3
STScan	3×10^4	3×10^6	8×10^7	OutOfMemory
*STScan**	10^3	10^4	5×10^4	10^5
brute-force	9×10^2	10^4	5×10^4	9×10^4
*brute-force**	9×10^2	10^4	5×10^4	9×10^4
inter-arrival	$< 10^2$	$< 10^2$	$< 10^2$	$< 10^2$

space cost. It also shows that, the space costs of *STScan**, *brute-force* and *brute-force** are all $O(N)$ as mentioned before. Assuming each Java object only occupies an integer(8

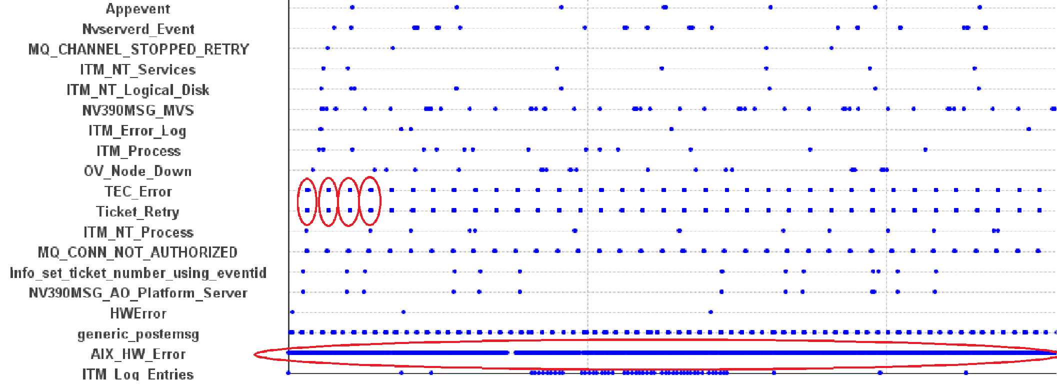

Figure 5.5: Plotting for Account2 Data

bytes), *STScan* would cost over 10G bytes memory for 50×10^3 items. Hence, it runs out of memory when the data size becomes larger. However, by using the incremental sorted table, for the same data set, *STScan** only costs 10M memory. *inter-arrival* only stores the clusters of all inter-arrivals, so its space cost is small.

Real Data

Table 5.5: Real System Events

Data set	Time Frame	#Events	#Event Types
Account1	54 days	1,124,834	95
Account2	32 days	2,076,408	104

Two real data sets are collected from IT outsourcing centers by IBM Tivoli monitoring system [urlf] [TLP$^+$12], denoted as Account1 and Account2. Each data set is a collection of system events from hundreds of application servers and data server. These system events are mostly system alerts triggered by some monitoring situations (e.g. the CPU utilization is above a threshold). Table 5.5 shows the time frames and the sizes of the two real data sets. To discover the temporal dependencies with qualified lag intervals, we let $\chi^2_c = 6.64$ which corresponds to the confidence level 99%, and $minsup = 0.05$. A constraint that $t_2 \leq \texttt{1hour}$ is applied to this testing from the domain experts. δ of *inter-arrival* is varied from 1 to 2000.

Figures 5.6 and 5.7 show the running times of all algorithms on the two real data sets. As for *STScan* and *STScan** , the running times grow slower than in Figure 5.4 because the constraint $t_2 \leq 1\texttt{hour}$ reduces their time complexities. Table 5.6 lists the peak numbers of allocated memory objects in JVM on Account2 data. The results on Account1 data is similar to this table.

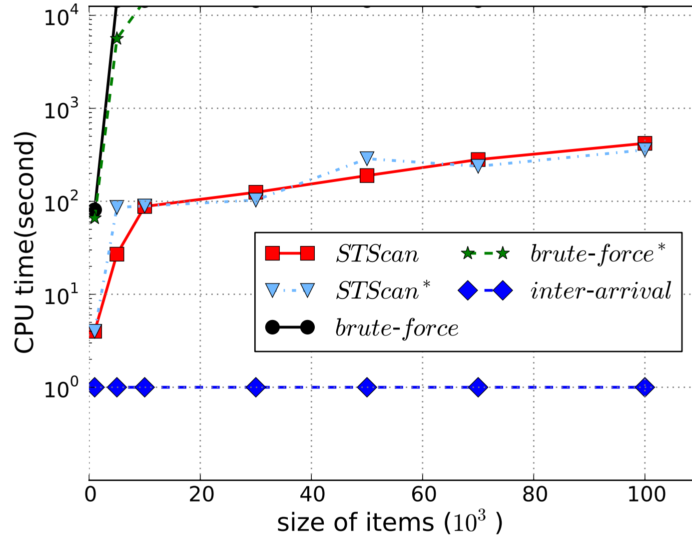

Figure 5.6: Running Time on Account1 Data

Table 5.6: Space Cost on Account2 Data

Algorithm \ Data size	10^3	10×10^3	50×10^3	100×10^3
STScan	4×10^4	3×10^6	1×10^7	3×10^7
*STScan**	10^3	6×10^3	5×10^4	10^5
brute-force	9×10^2	3×10^3	3×10^3	3×10^3
*brute-force**	9×10^2	3×10^3	3×10^3	3×10^3
inter-arrival	$< 10^2$	$< 10^2$	$< 10^2$	$< 10^2$

Table 5.3 lists several discovered temporal dependencies with qualified lag intervals. *inter-arrival* only finds the first two temporal dependencies on Account2 data. The reason is that, only the two temporal dependencies have very small lag intervals which are just the

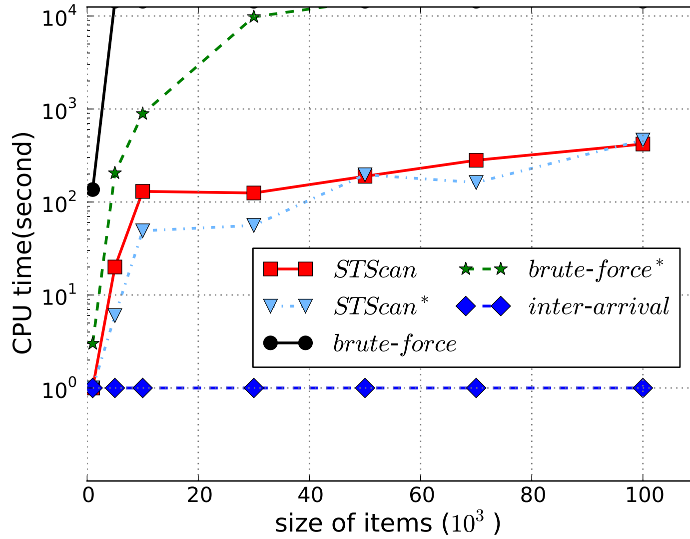

Figure 5.7: Running Time on Account2 Data

inter-arrivals of the events. However, the lag intervals for other temporal dependencies are larger than most inter-arrivals, so *inter-arrival* fails.

In Table 5.3, the first discovered temporal dependency for Account1 shows that MSG_Plat_APP is a periodic pattern with a period of 1 hour. This pattern indicates this event MSG_Plat_APP is a heartbeat signal from an application. The second and third discovered temporal dependencies can be viewed as a case study for event correlation [KYY+95]. Since most servers are Linux servers, so the alerts from processes must be also from Linux processes. Therefore, for Account1, process events and Linux process events can be automatically correlated. High CPU utilization alerts (*SMP_CPU*) can only be triggered by abnormal processes, so *SMP_CPU* events can also be correlated with *Linux_Process* events. In Account2, the first two temporal dependencies compose a mutual dependency pattern between *TEC_Error* and *Ticket_Retry*. It can be explained by a programming logic in IBM Tivoli monitoring system. When the monitoring system fails to generate the incident ticket to the ticketing system, it will report a *TEC* error and retry the ticket generation. Therefore, *TEC_Error* and *Ticket_Retry* events are often raised together. The third and

fourth discovered temporal dependencies for Account2 are related to a hardware error of an AIX server but with different lag intervals. This is caused by a polling monitoring situation. When an AIX server is down, the monitoring system continuously receive *AIX_HW_Error* events when polling that AIX server. Thus, this *AIX_HW_Error* event exhibits a periodic pattern. To validate the discovered results, we plot the temporal events into a graphical chart. Figure 5.5 is a screen shot of the plotting for Account2 data. The x-axis is the time stamp, the y-axis is the event type. As shown by this figure, *TEC_Error* and *Ticket_Retry* exhibit a mutually dependency since they are always generated at the almost same time. *AIX_HW_Error* is a polling event.

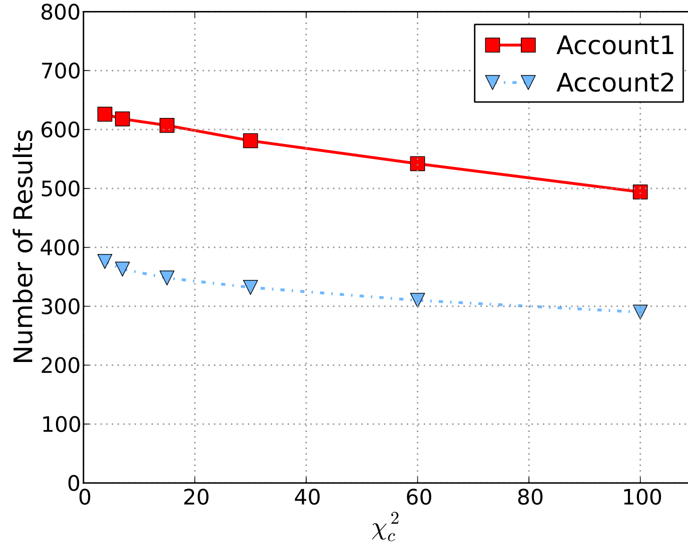

Figure 5.8: Number of Results by Varying χ_c^2

To test the sensitivity of parameters, we vary χ_c^2 and $minsup$ and test the numbers of discovered temporal dependencies (Figures 5.8 and Figure 5.9) and the running time (Figure 5.10 and Figure 5.11). When varying χ_c^2, $minsup = 0.05$; When varying $minsup$, $\chi_c^2 = 6.64$ (with 99% confidence level). χ_c^2 is not sensitive to the algorithm result because the associated confidence level is only from 95% to 99.99% although χ_c^2 is varied from 3.84 to 100. By varying $minsup$, the number of discovered temporal dependencies exponential-

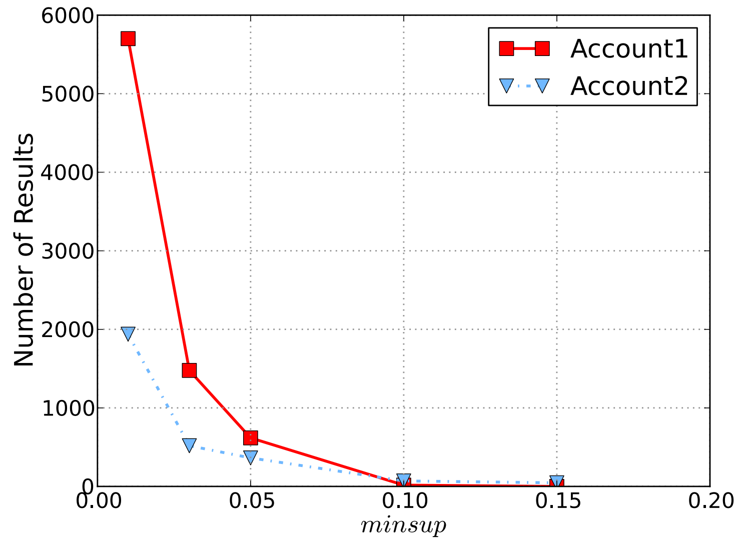

Figure 5.9: Num. of Results by Varying $minsup$

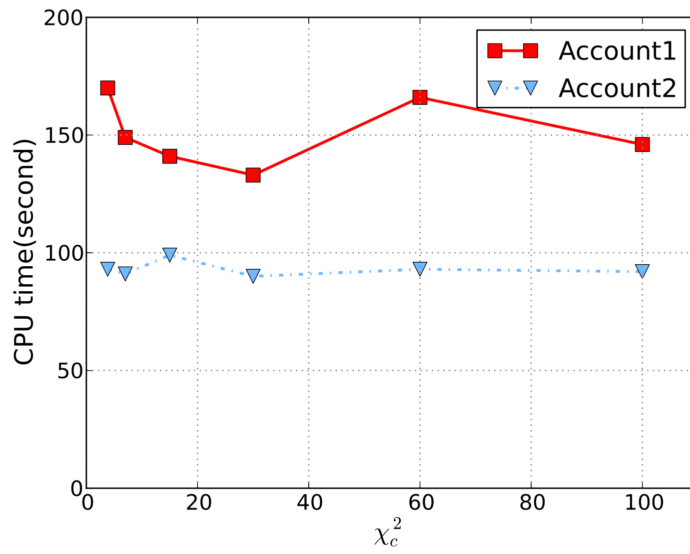

Figure 5.10: Running time by Varying χ_c^2

ly decreases as shown in Figure 5.9. As mentioned in [MH01b], the effective choice of $minsup$ is 0.001 to 0.1.

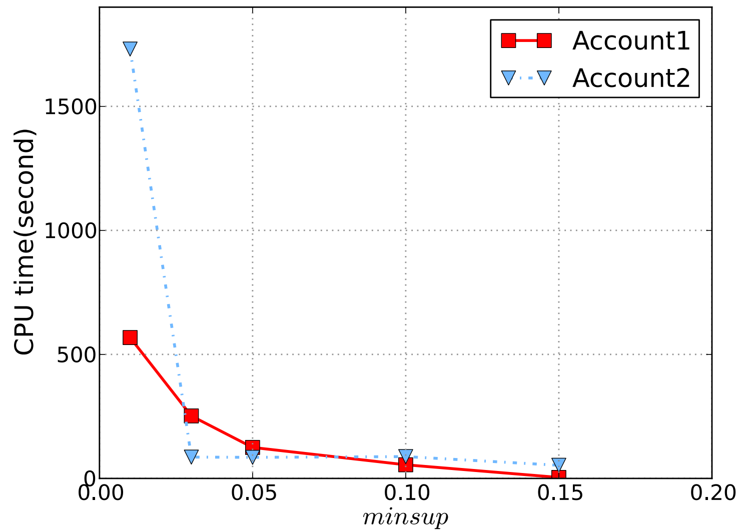

Figure 5.11: Running time by Varying $minsup$

5.2 Recommending Incident Resolutions

With the development of e-commerce, a substantial amount of research has been devoted to the recommendation systems. These recommendation systems determine items or products to be recommended based on prior behavior of the user or similar users and on the item itself. An increasing amount of user interactions has provided these applications with a large amount of information that can be converted into knowledge. In this dissertation we apply this approach to the resolution of incident tickets for maintaining service infrastructures. In addition, we extend the recommendation methodology to take into account possible falsity of the tickets. We focus on the event tickets (or automatic tickets), which are incident tickets generated by monitoring systems. We believe our work can help service providers to efficiently find appropriate problem resolutions and correlate related tickets resolved in the past. Most service providers keep track of a large amount of historical tickets with resolutions. The resolution is usually stored as a plain text which describes how this ticketed incident has been resolved. We analyzed historical event tickets collected

from three different accounts managed IBM Global Services. We consider an account as an aggregate of services using a common infrastructure. One observation is that many event tickets share the same resolutions. If two events are similar, then their triggered tickets probably have the same resolution. Therefore, we consider recommending a resolution for an incoming ticket based on the event information and historical tickets. The preliminary work for various recommendation algorithms has been discussed in Section 2.4.

We analyzed ticket data from three different accounts managed by IBM Global Services. One observation is that many ticket resolutions repeatedly appear in the ticket database. For example, for a low disk capacity ticket, usual resolutions are deletion of temporal files, backup data, or addion of a new disk. Unusual resolutions are very rare. The collected ticket sets from the three accounts are denoted by "account1", "accoun-

Table 5.7: Data Summary

Data set	Num. of Servers	Num. of Tickets	Time Frame
account1	1,145	50,377	55 days
account2	614	6,121	29 days
account3	391	4,066	48 days

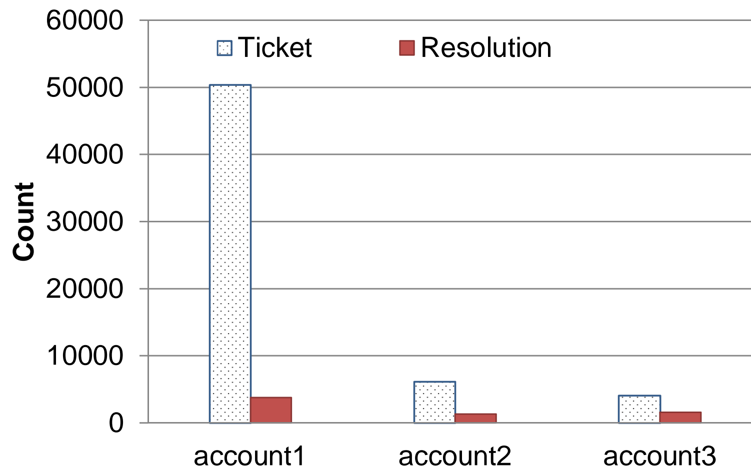

Figure 5.12: Numbers of Tickets and Distinct Resolutions

t2" and "account3" respectively. Table 5.7 summarizes the three data sets. Figure 5.12 shows the numbers of tickets and distinct resolutions and Figures 5.13 to 5.15 show the top

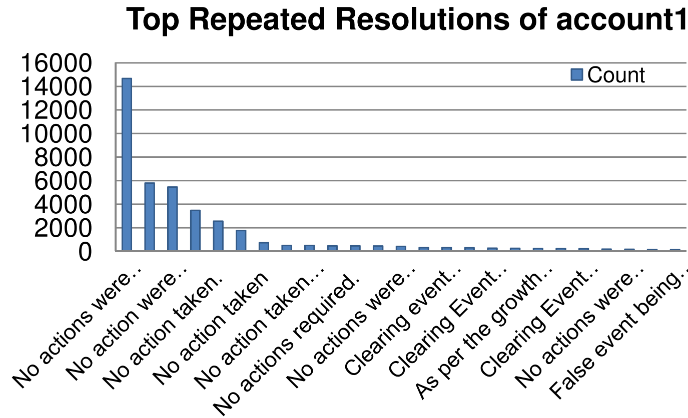

Figure 5.13: Top Repeated Resolutions of Account1

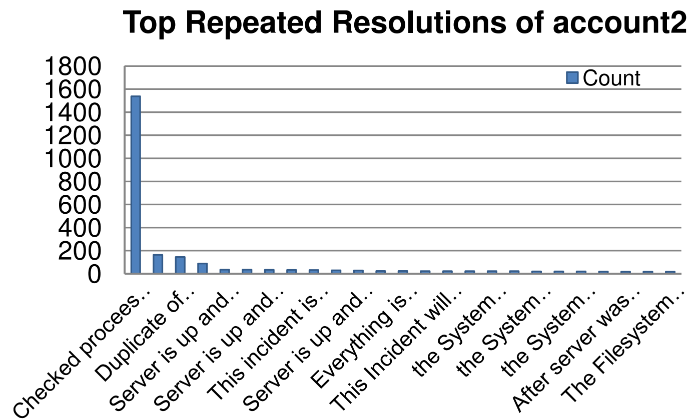

Figure 5.14: Top Repeated Resolutions of Account2

repeated resolutions in each data set. It is seen that the number of distinct resolutions is much smaller than the number of tickets - in other words, multiple tickets share the same resolutions. For example (Figure 5.13) the first resolution, “No actions were...”, appears more than 14000 times in “account1”.

5.2.1 A Basic KNN-based Recommendation

Given an incoming event ticket, the objective of the resolution recommendation is to find k resolutions as close as possible to the the true one for some user-specified parameter k. The

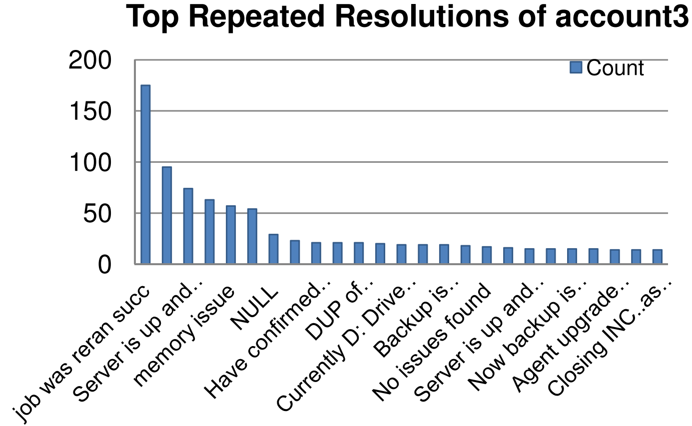

Figure 5.15: Top Repeated Resolutions of Account3

recommendation problem is often related to that of predicting the top k possible resolutions. A straightforward approach is to apply the KNN algorithm which searches the K nearest neighbors of the given ticket (K is a predefined parameter), and recommends the top $k \leq K$ representative resolutions among them [SKKR00, TSK05]. The nearest neighbors are indicated by similarities of the associated events of the tickets. In this dissertation, the representativeness is measured by the number of occurrences in the K neighbors.

Table 5.8: Notations for KNN based Recommendation Algorithms

Notation	Description
D	Set of historical tickets
$\|\cdot\|$	Size of set
t_i	i-th event ticket
$r(t_i)$	Resolution description of t_i
$e(t_i)$	Associate event of t_i
$c(t_i)$	Type of ticket t_i, $c(t_i) = 1$ indicates t_i is a real ticket,$c(t_i) = 0$ indicates t_i is a false ticket.
$A(e)$	Set of attributes of event e
$sim(e_1, e_2)$	Similarity of events e_1 and e_2
$sim_a(e_1, e_2)$	Similarity of a values of event e_1 and e_2
K	Number of nearest neighbors in the KNN algorithm
k	Number of recommended resolutions for a ticket, $k \leq K$

Table 5.8 lists the notations used in this dissertation. Let $D = \{t_1, ..., t_n\}$ be the set of historical event tickets and t_i be the i-th ticket in D, $i = 1, ..., n$. Let $r(t_i)$ denote the resolution description of t_i, $e(t_i)$ is the associated event of t_i. Given an event ticket

t, the nearest neighbor of t is the ticket t_i which maximizes $sim(e(t), e(t_i))$, $t_i \in D$, where $sim(\cdot, \cdot)$ is a similarity function for events. Each event consists of event attributes with values. Let $A(e)$ denote the set of attributes of event e. The similarity for events is computed as the summation of the similarities for all attributes. There are three types of event attributes: categorical, numeric and textual (shown by Table 5.9). Given an attribute

Table 5.9: Event Attribute Types

Type	Example
Categorical	host name, process name, ...
Numeric	CPU utilization, disk free space percentage, ...
Textual	event message,...

a and two events e_1 and e_2, $a \in A(e_1)$ and $a \in A(e_2)$, the values of a in e_1 and e_2 are denoted by $a(e_1)$ and $a(e_2)$. The similarity of e_1 and e_2 with respect to a is

$$sim_a(e_1, e_2) = \begin{cases} I[a(e_1) = a(e_2)], & \text{if } a \text{ is categorical,} \\ \frac{|a(e_1) - a(e_2)|}{max|a(e_i) - a(e_j)|}, & \text{if } a \text{ is numeric,} \\ Jaccard(a(e_1), a(e_2)), & \text{if } a \text{ is textual,} \end{cases}$$

where $I(\cdot)$ is the indicator function returning 1 if the input condition holds, and 0 otherwise. Let $max|a(e_i) - a(e_j)|$ be the size of the value range of a. $Jaccard(\cdot, \cdot)$ is the Jaccard index for *bag of words model* [SM84], frequently used to compute the similarity of two texts. Its value is the proportion of common words in the two texts. Note that for any type of attribute, inequality $0 \leq sim_a(e_1, e_2) \leq 1$ holds. Then, the similarity for two events e_1 and e_2 is computed as

$$sim(e_1, e_2) = \frac{\sum_{a \in A(e_1) \cap A(e_2)} sim_a(e_1, e_2)}{|A(e_1) \cup A(e_2)|}. \tag{5.4}$$

Clearly, $0 \leq sim(e_1, e_2) \leq 1$. To identify the type of attribute a, we only need to scan all appearing values of a. If all values are composed of digits and a dot, a is numeric. If

some value of a contains a sentence or phrase, then a is textual. Otherwise, a is categorical.

A Division Method

Traditional recommendation algorithms focus on the accuracy of the recommended results. However, in automated service management, false alarms are unavoidable in both the historical and incoming tickets. The resolutions of false tickets are short comments such as "this is a false alarm", "everything is fine" and "no problem found". If we recommend a false ticket's resolution for a real ticket, it would cause the system administrator to overlook the real system problem. and besides, none of the information in this resolution is helpful. Note that in a large enterprise IT environment, overlooking a real system problem may have serious consequences such as system crashes. Therefore, we consider incorporation of penalties in the recommendation results. There are two cases meriting a penalty: recommendation of a false ticket's resolution for a real ticket, and recommendation of a real ticket's resolution for a false ticket. The penalty in the first case should be larger since the real ticket is more important. The two cases are analogous to the *false negative* and *false positive* in prediction problems [TSK05], but note that our recommendation target is the ticket resolution, not its type. A false ticket's event may also have a high similarity with that of a real one. The objective of the recommendation algorithm is now maximized accuracy under minimized penalty.

A straightforward solution consists in dividing all historical tickets into two sets comprising the real and false tickets respectively. Then, it builds a KNN-based recommender for each set respectively. Another ticket type predictor is created, establishing whether an incoming ticket is real or false, with the appropriate recommender used accordingly. The divide method works as follows: it first uses a type predictor to predict whether the incoming ticket is real or false. If it is real, then it recommends the tickets from the real historic tickets; if it is false, it recommends the tickets from the false historic tickets. The historic

tickets are already processed by the system admin, so their types are known and we do not have to predict them.

The division method is simple, but relies heavily on the precision of the ticket type predictor which cannot be perfect. If the ticket type prediction is correct, there will be no penalty for any recommendation result. If the ticket type prediction is wrong, every recommended resolution will incur a penalty. For example, if the incoming ticket is real, but the predictor says it is a false ticket, so this method only recommends false tickets. As a result, all the recommendations would incur penalties.

A Probabilistic Fusion Method

To overcome the limitation of the division method, we develop a probabilistic fusion method. The framework of the basic KNN-based recommendation is retained, with difference that, the penalty and probability distribution of the ticket type are incorporated in the similarity function.

Let $loss_{real}$ be the loss for recommending a false ticket's resolution for a real ticket, and $loss_{false}$ be the loss for recommending a real ticket's resolution for a false one. For example, $loss_{real}$ would be the penalty for missing a real alert specified in the SLA (Service Level Agreement), e.g., 2700 dollars. $loss_{false}$ would be the human resource waste for handing a false alert, e.g, 300 dollars. In IT service management, $loss_{real}$ is always a fixed value in the contract with a particular consumer. $loss_{false}$ can be calculated from the human resource cost with the numbers of false tickets and real tickets in recent months. Therefore, for one application, $loss_{real}$ and $loss_{false}$ are both constant values. The total loss $loss_{real} + loss_{false}$ is also a constant value. Then, we let $\lambda = \frac{loss_{real}}{loss_{real}+loss_{false}}$. In the previous example, $loss_{real} + loss_{false} = 2700 + 300 = 3000$ and $\lambda = 2700/3000 = 0.9$. Clearly, $0 \leq \lambda \leq 1$. In other words, λ is the proportional loss of recommending a false ticket's resolution to a real ticket. $1 - \lambda$ is the proportional loss of recommending a real

ticket's resolution to a false ticket. The penalty function is

$$\lambda_t(t_i) = \begin{cases} \lambda, & t \text{ is a real ticket, } t_i \text{ is a false ticket} \\ 1-\lambda, & t \text{ is a false ticket, } t_i \text{ is a real ticket} \\ 0, & \text{otherwise,} \end{cases}$$

where t is the incoming ticket and t_i is the historical one whose resolution is recommended for t. Conversely, an award function can be defined as $f_t(t_i) = 1 - \lambda_t(t_i)$. Since $0 \leq \lambda_t(t_i) \leq 1$, $0 \leq f_t(t_i) \leq 1$.

Let $c(\cdot)$ denote the ticket type. $c(t_i) = 1$ indicates t_i is a real ticket; $c(t_i) = 0$ indicates t_i is a false ticket. Since t is an incoming ticket, the value of $c(t)$ is not known. Using a ticket type predictor, we can estimate the distribution of the binary random variable $c(t)$. The idea of this method is to incorporate the expected award in the similarity function. The new similarity function $sim'(\cdot,\cdot)$ is defined as:

$$sim'(e(t), e(t_i)) = E[f_t(t_i)] \cdot sim(e(t), e(t_i)), \tag{5.5}$$

where $sim(\cdot,\cdot)$ is the original similarity function defined by Eq. (5.4), and $E[f_t(t_i)]$ is the expected award, $E[f_t(t_i)] = 1 - E[\lambda_t(t_i)]$. If t_i and t have the same ticket type then $E[f_t(t_i)] = 1$ and $sim'(e(t), e(t_i)) = sim(e(t), e(t_i))$, otherwise $sim'(e(t), e(t_i)) < sim(e(t), e(t_i))$. Generally, the expected award is computed as

$$E[f_t(t_i)] \;=\; E[1 - \lambda_t(t_i)] = 1 - E[\lambda_t(t_i)]$$

Based on the definition of $\lambda_t(t_i)$, the expected penalty is

$$\begin{aligned} E[\lambda_t(t_i)] &= P[c(t)=1, c(t_i)=0] \cdot \lambda \\ &+ P[c(t)=0, c(t_i)=1] \cdot (1-\lambda) \\ &+ P[c(t)=0, c(t_i)=0] \cdot 0 \\ &+ P[c(t)=1, c(t_i)=1] \cdot 0 \end{aligned}$$

Since t_i is the historical ticket and $c(t_i)$ is observed, if the given t_i is a real ticket, then

$$E[\lambda_t(t_i)] = P[c(t)=0] \cdot (1-\lambda) + P[c(t)=1] \cdot 0 = P[c(t)=0] \cdot (1-\lambda).$$

If the given t_i is a false ticket, then

$$E[\lambda_t(t_i)] = P[c(t)=1] \cdot \lambda + P[c(t)=0] \cdot 0 = P[c(t)=1] \cdot \lambda.$$

Note that all factors in the new similarity function are of the same scale, i.e., $[0,1]$, thus $0 \leq sim'(\cdot,\cdot) \leq 1$.

Example 6 illustrates how the new similarity function combines awards and similarities to affect the recommendation results.

Example 6. *Let $D = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, where t_1,t_2 and t_3 are false tickets and others are real tickets. Let $\lambda = 0.6$ since a real ticket is more important than a false ticket. Given an incoming ticket is t, by using a ticket type predictor, we estimate $P[c(t)=1] = 0.6$, $P[c(t)=0] = 1-0.6 = 0.4$. Thus, t is more likely to be a real ticket. Table 5.10 lists all related information about all tickets in D. Let $K=3$, Table 5.11 shows the K nearest tickets selected by different methods. The basic KNN-based algorithm selects the 3 nearest tickets based on $sim(e(t), e(t_i))$. Since the incoming ticket t is more likely*

Table 5.10: Summary of Tickets in D

	t_1	t_2	t_3	t_4	t_5	t_6
$c(t_i)$	0	0	0	1	1	1
$sim(e(t), e(t_i))$	0.1	0.45	0.9	0.5	0.4	0.1
$E[f_t(t_i)]$	0.64	0.64	0.64	0.84	0.84	0.84
$sim'(e(t), e(t_i))$	0.064	0.288	0.576	0.42	0.336	0.084

Table 5.11: Selected K Nearest Tickets

Method	K **Nearest Tickets**
Basic KNN-based	t_2, t_3 and t_4
Dividing Method	t_4, t_5 and t_6
Probabilistic Fusion Method	t_3, t_4 and t_5

to be a real ticket, the dividing method selects all real tickets. However, t_6 only has a very small similarity with t, $sim(e(t), e(t_6)) = 0.1$, but t_3 has the highest similarity 0.9 with t even though t_3 is a false ticket. To balance all related factors for recommendation, the probabilistic fusion method first computes the expected award of each ticket $E[f_t(t_i)]$, $t_i \in D$. If t_i is a false ticket, If t_i is a real ticket,

$$E[f_t(t_i)] = 1 - P[c(t) = 0] \cdot (1 - \lambda) = 1 - 0.4 \cdot 0.4 = 0.84.$$

Based on the values of $sim'(e(t), e(t_i)) = E[f_t(t_i)] \cdot sim(e(t), e(t_i))$, t_3, t_4 and t_5 are selected finally.

Prediction of Ticket Type

Given an incoming ticket t, the probabilistic fusion method needs to estimate the distribution of $P[c(t)]$. The dividing method also has to predict whether t is a real ticket or a false ticket. There are many binary classification algorithms for estimating $P[c(t)]$. In our implementation, we utilize another KNN classifier. The features are the event attributes and the classification label is the ticket type. The KNN classifier first finds the K nearest tickets in D, denoted as $D_K = \{t_{j_1}, ..., t_{j_k}\}$. Then, $P[c(t) = 1]$ is the proportion of real

tickets in D_K and $P[c(t) = 0]$ is the proportion of false tickets in D_K. Formally,

$$\begin{aligned} P[c(t) = 1] &= |\{t_j | t_j \in D_K, c(t_j) = 1\}|/K \\ P[c(t) = 0] &= 1 - P[c(t) = 1]. \end{aligned}$$

5.2.2 Evaluation

Implementation and Testing Environment

We implemented four algorithms: KNN, weighted KNN [Dud76], the division method and the probabilistic fusion method, which are denoted by "KNN", "WeightedKNN", "Divide" and "Fusion" respectively. Our proposed two algorithms, "Divide" and "Fusion", are based on the weighted KNN algorithm framework. We choose the KNN-based algorithm as the baseline because it is the most widely used Top-N item based recommendation algorithm. Certainly, we can use SVM to predict the ticket type to be false or real. But our core idea is not about classification, but to combine the penalty for the misleading resolution into the recommendation algorithm.

All algorithms are implemented by Java 1.6. This testing machine is Windows XP with Intel Core 2 Duo CPU 2.4GHz and 3GB of RAM.

Experimental Data

Experimental event tickets are collected from three accounts managed by IBM Global Services denoted later "account1", "account2" and "account3". The monitoring events are captured by IBM Tivoli Monitoring [urle]. The ticket sets are summarized in Table 5.7.

Accuracy

For each ticket set, the first 90% tickets are used as the historic tickets and the remaining 10% tickets are used for testing. Hit rate is a widely used metric for evaluating the accuracy in item-based recommendation algorithms [DK04, Kar01, NK11].

$$\text{Accuracy} = \text{Hit-Rate} = |Hit(C)|/|C|,$$

where C is the testing set, and $Hit(C)$ is the set for which one of the recommended resolutions is *hit* by the true resolution. If the recommendation resolution is truly relevant to the ticket, we say that recommended resolution is *hit* by the true resolution.

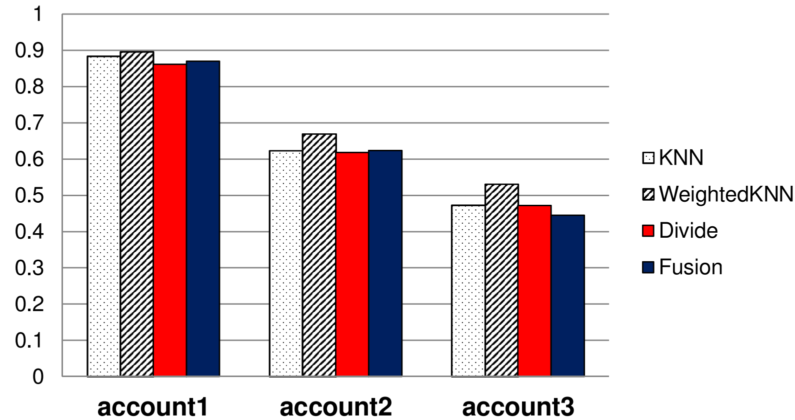

Figure 5.16: Accuracy for $K = 10$, $k = 3$

Since real tickets are more important than false ones, we define another accuracy measure, the weighted accuracy, which assigns weights to real and false tickets. The weighted accuracy is computed as follows:

$$\text{Weighted Accuracy} = \frac{\lambda \cdot |Hit(C_{real})| + (1-\lambda) \cdot |Hit(C_{false})|}{\lambda \cdot |C_{real}| + (1-\lambda) \cdot |C_{false}|},$$

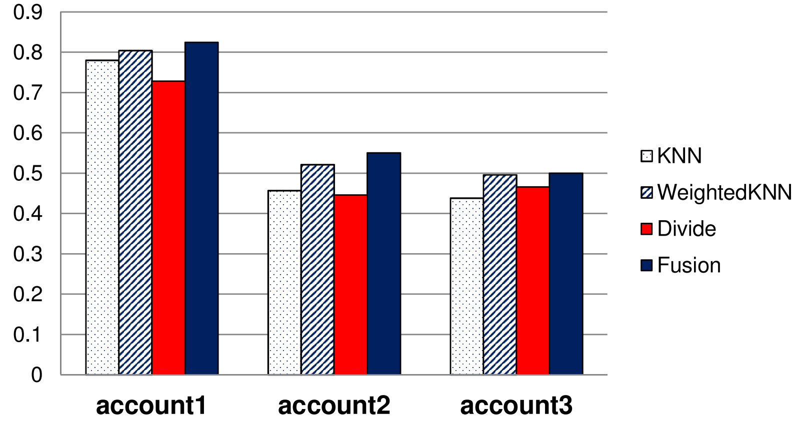

Figure 5.17: Accuracy for Real Tickets and $K = 10$, $k = 3$

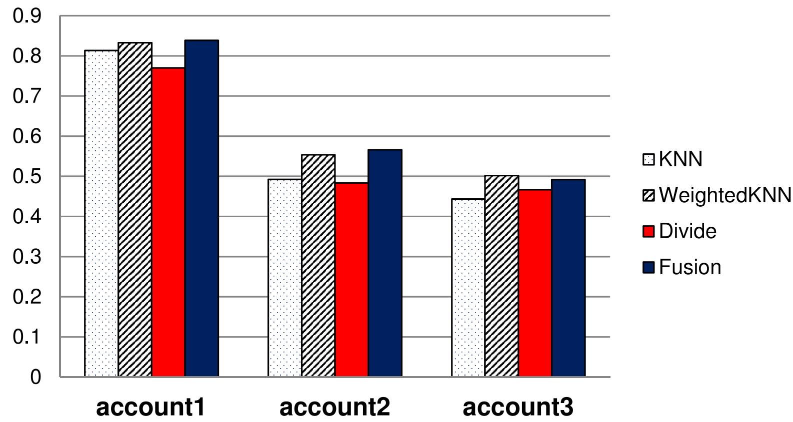

Figure 5.18: Weighted Accuracy for $K = 10$, $k = 3$

where C_{real} is the set of real testing tickets, C_{false} is the set of false testing tickets, $C_{real} \cup C_{false} = C$, λ is the importance weight of the real tickets, $0 \leq \lambda \leq 1$, it is also the penalty mentioned before. In this evaluation, $\lambda = 0.9$ since the real tickets are much more important than the false tickets in reality. We also test other large λ values, such as 0.8 and 0.99. The accuracy comparison results have no significant change.

We vary K and k from 1 to 20 to obtain different parameter settings. Figures 5.16 to 5.21 are the testing results for $K = 10, k = 3$ and $K = 20, k = 5$. The comparison results for other parameter settings are similar to the two figures. It is seen that, the weighted KNN algorithm always achieves the highest accuracy in the three data sets. But for real tickets,

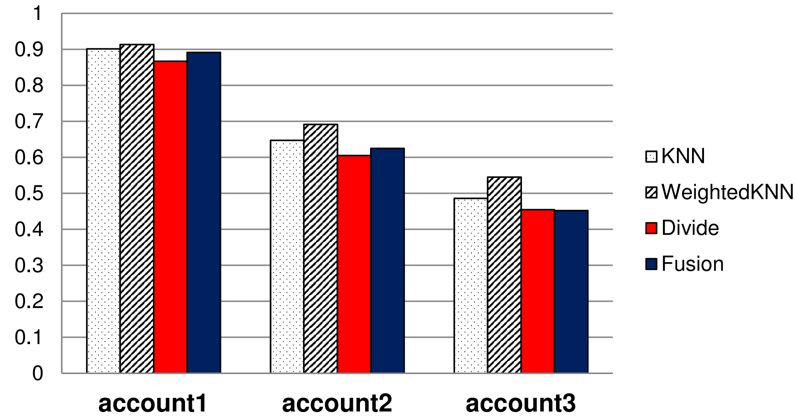

Figure 5.19: Accuracy for $K = 20$, $k = 5$

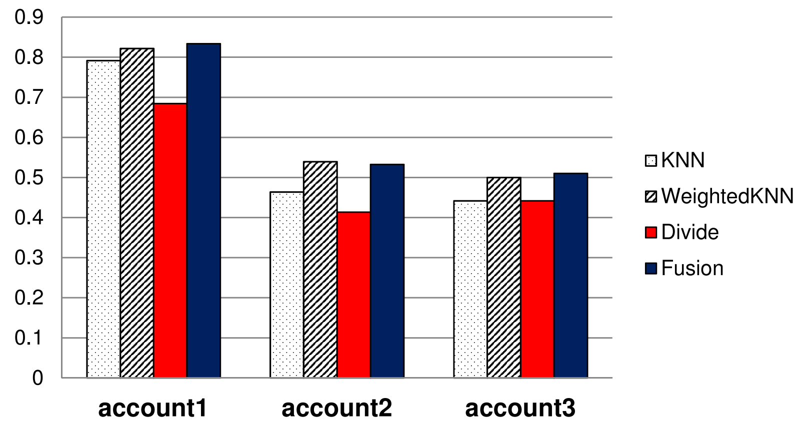

Figure 5.20: Accuracy for Real Tickets and $K = 20$, $k = 5$

our proposed probabilistic fusion method outperforms other algorithms (Figures 5.17 and 5.20). As for the weighted accuracy in Figures 5.18 and 5.21, the weighted KNN and the probabilistic fusion are still the two best algorithms, and neither of them outperforms the other in all data sets. Overall, the performances of all four algorithms are close. For each comparison, the difference between the highest one and the lowest one is about 10%.

Penalty

Figures 5.22 and 5.23 show the average penalty for each testing ticket. We assigned a higher importance to the real tickets, $\lambda = 0.9$. As shown by these figures, our proposed

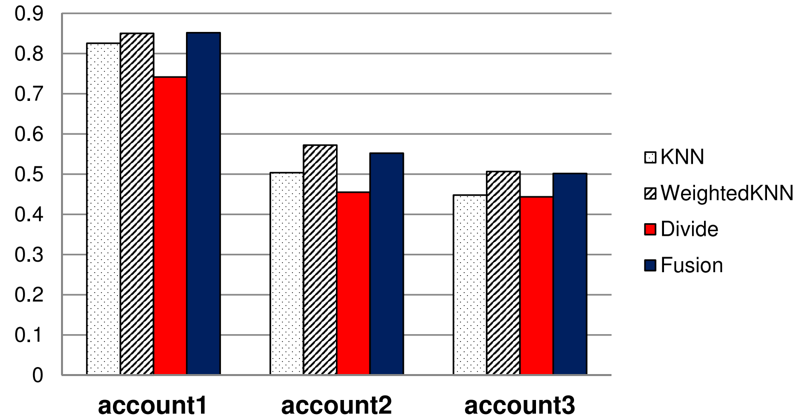

Figure 5.21: Weighted Accuracy for $K = 20$, $k = 5$

two algorithms have smaller penalties than the traditional KNN-based recommendation algorithms. The probabilistic fusion method outperforms the division method, which relies heavily on the ticket type predictor. Overall, our probabilistic fusion method only has about 1/3 of the penalties of the traditional KNN-based algorithms.

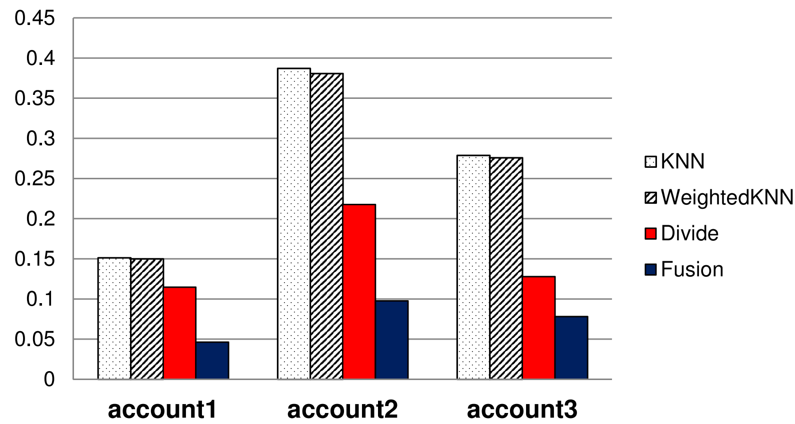

Figure 5.22: Average Penalty for $K = 10$, $k = 3$

Overall Performance

An overall quantity metric is used for evaluating the recommendation algorithms, covering both the accuracy and the average penalty. It is defined as overall score = weighted accuracy / average penalty. If the weighted accuracy is higher or the average penalty is lower, then

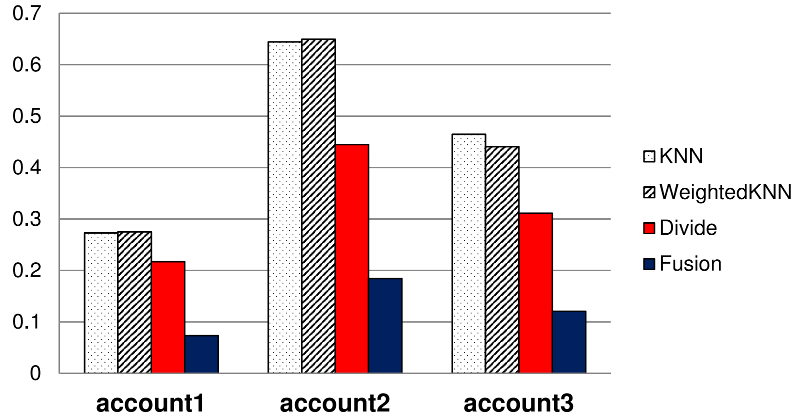

Figure 5.23: Average Penalty for $K = 20$, $k = 5$

the overall score becomes higher and the overall performance is better. Figures 5.24 and 5.25 show the overall scores of all algorithms for two parameter settings. It is seen that, our proposed algorithms are always better than the KNN-based algorithms in each data set.

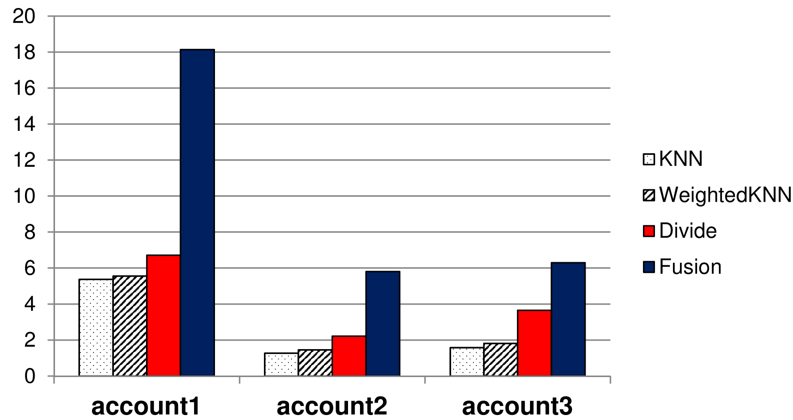

Figure 5.24: Overall Score for $K = 10$, $k = 3$

Variation of Parameters

To compare the results of each algorithm, we vary the number of each recommendation resolutions, k. Figures 5.26 to 5.34 show the weighted accuracies, average penalties and overall scores by varying k from 1 to 8, with $K = 10$. For other values of K, the comparison results are similar to the three figures. As shown by Figures 5.26 to 5.28, when we

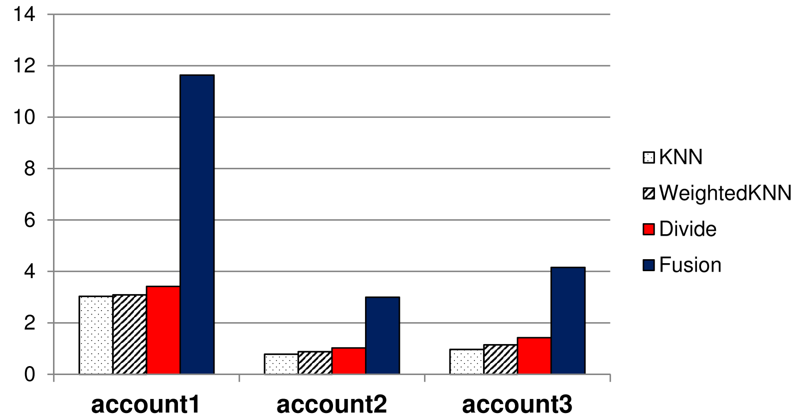

Figure 5.25: Overall Score for $K = 20$, $k = 5$

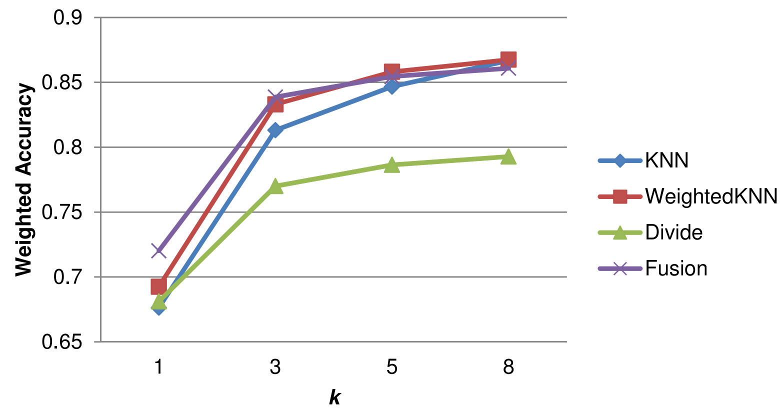

Figure 5.26: Weighted accuracy for account1 by varying k, $K = 10$

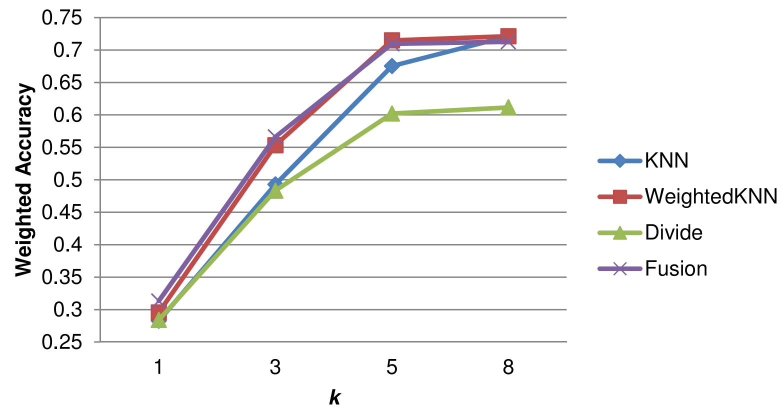

Figure 5.27: Weighted accuracy for account2 by varying k, $K = 10$

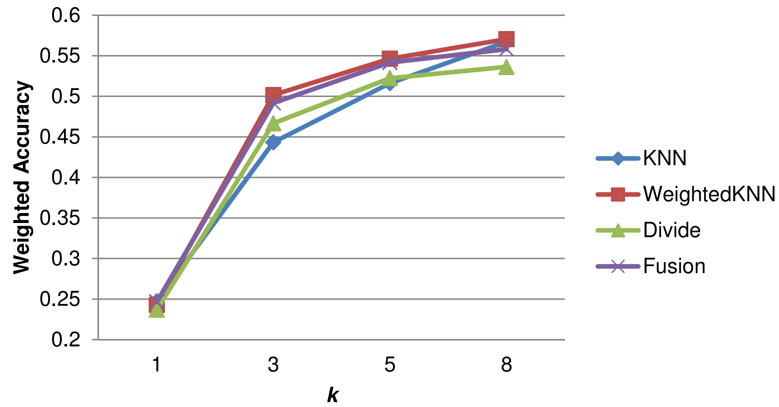

Figure 5.28: Weighted accuracy for account3 by varying k, $K = 10$

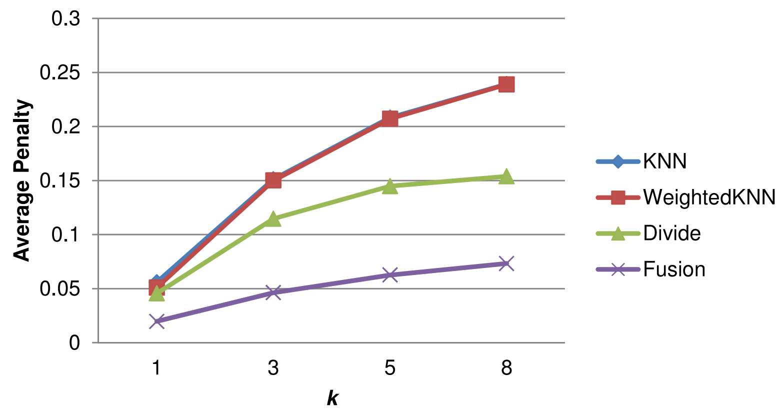

Figure 5.29: Average penalty for account1 by varying k, $K = 10$

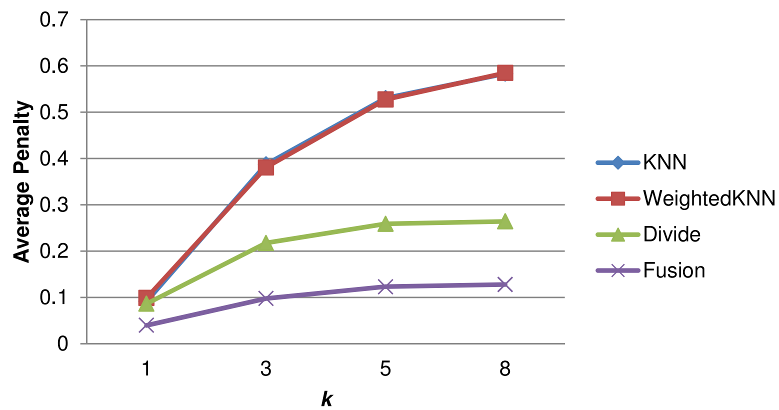

Figure 5.30: Average penalty for account2 by varying k, $K = 10$

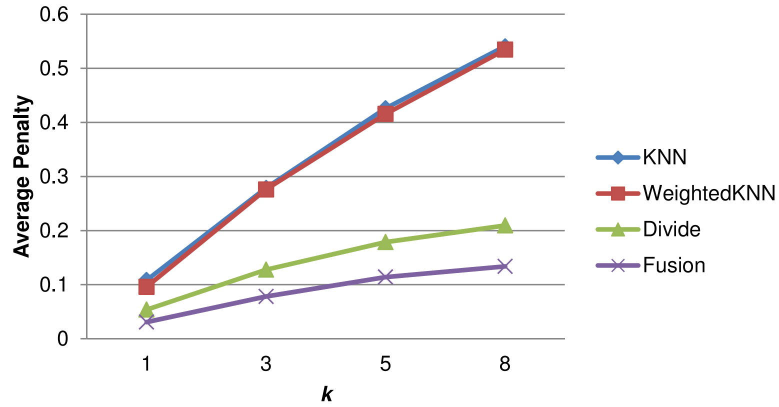

Figure 5.31: Average penalty for account3 by varying k, $K = 10$

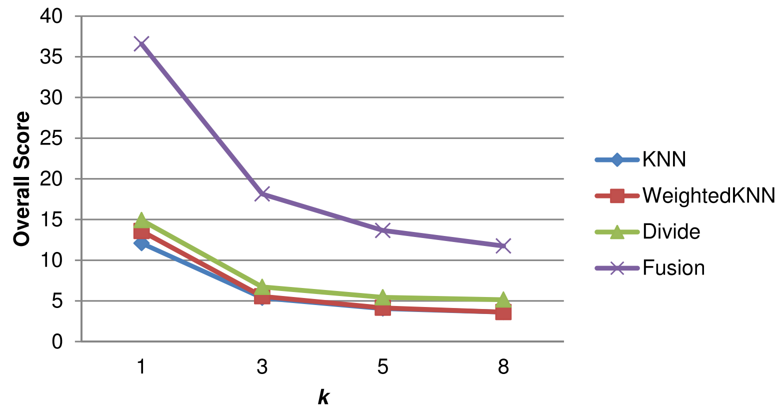

Figure 5.32: Average penalty for account1 by varying k, $K = 10$

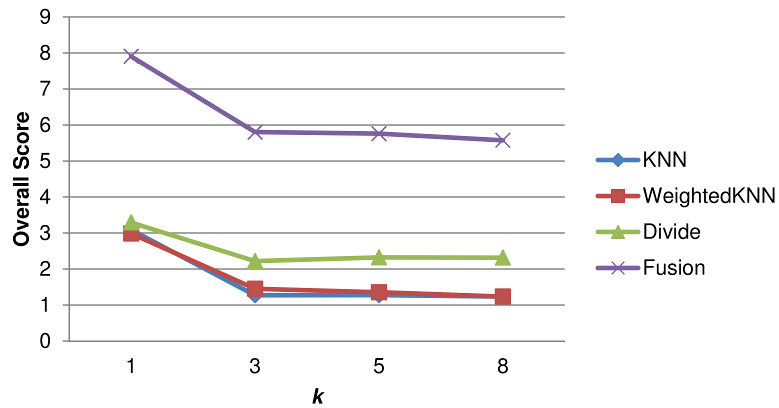

Figure 5.33: Average penalty for account2 by varying k, $K = 10$

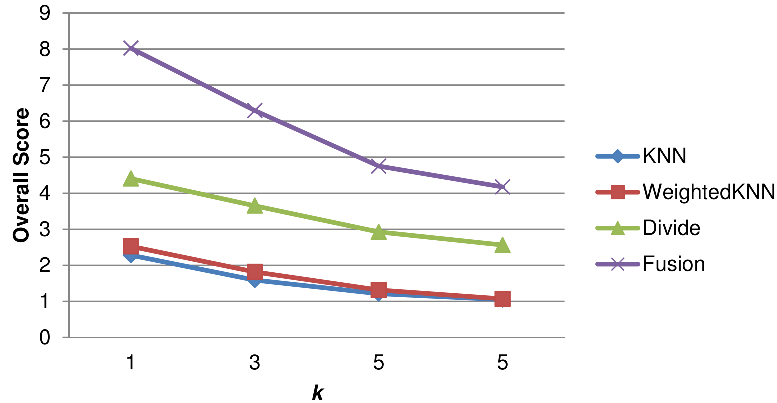

Figure 5.34: Average penalty for account3 by varying k, $K = 10$

increase the value of k, the size of the recommendation results becomes larger. Then the probability of one recommended resolution being *hit* by the true resolution also increases. Therefore, the weighted accuracy becomes higher. Except for the division method, all algorithms have similar weighted accuracies for each k. However, as k is increased and there are more recommended resolutions, there are more potential penalties in the recommended resolutions. Hence, the average penalty also becomes higher (Figures 5.29 to 5.31). Finally, Figures 5.32 to 5.34 compare the overall performance by varying k. Clearly, the probabilistic fusion method outperforms other algorithms for every k.

A Case Study

We select an event ticket in "account1" to illustrate why our proposed algorithms are better than the traditional KNN-based algorithms. Table 5.12 shows a list of recommended resolutions given by each algorithm. The testing ticket is a real event ticket triggered by a low capacity alert for the file system. Its true resolution of this ticket is: "cleaned up the FS using RMAN retention policies..." RMAN is a data backup and recovery tool in Oracle database. The general idea of this resolution is to use this tool to clean up the old data.

As shown by Table 5.12, the first resolution recommended by KNN and WeightedKNN is a false ticket's resolution: "No actions were taken by GLDO for this Clearing Event..."

Table 5.12: A Case Study for $K = 10, k = 3$

Algorithm	Recommended Resolution	Is Hit	Is A Real Ticket's Resolution	Penalty
KNN	No actions were taken by GLDO for this Clearing Event...	no	false	0.9
	Clean up the backup filesystem. Filesystem k-bytes used avail capacity...	no	true	0
	Duplicated 28106883...	no	true	0
WeightedKNN	No actions were taken by GLDO for this Clearing Event...	no	false	0.9
	I cleaned up the FS using RMAN retention policies...	yes	true	0
	Duplicated 28106883...	no	true	0
Divide	Duplicated 28106883...	no	true	0
	Another device failure has been reported for this node...	no	true	0
	I cleaned up the FS using RMAN retention policies...	yes	true	0
Fusion	Duplicated 28106883...	no	true	0
	Another device failure has been reported for this node...	no	true	0
	I cleaned up the FS using RMAN retention policies...	yes	true	0

It might be caused by a temporal file generated by some application, which would clean up the temporal file automatically after its job was done. When the system administrator opened that ticket, the problem was gone, and that ticket is seen as false. However, the testing ticket is real and would not disappear unless the problem was actually fixed. This resolution from the false ticket would have misled the system administrator to overlook this problem. Consequently, a penalty of $\lambda = 0.9$ is given to KNN and WeightedKNN.

WeightedKNN, Divide and Fusion all successfully find the true resolution of this testing ticket, but WeightedKNN has one false resolution, so its penalty is 0.9. Our proposed methods, Divide and Fusion, have no penalty for this ticket. Therefore, the two methods are better than WeightedKNN.

5.3 Searching Similar Textual Event Segments

Sequential data is prevalent in many real-world applications such as bioinformatics, system security and networking. Similarity search is one of the most fundamental techniques in sequential data management. A lot of efficient approaches are designed for searching over

symbolic sequences or time series data, such as DNA sequences, stock prices, network packets and video streams. A textual event sequence is a sequence of events, where each event is a plain text or message. For example, in system management, most system logs are textual event sequences which describe the corresponding system behaviors, such as the starting and stopping of services, detection of network connections, software configuration modifications, and execution errors [TLP11] [OAS08, MZHM09, TL10, XHF+08]. System administrators utilize the event logs to understand system behaviors. Similar system events reveal potential similar system behaviors in history which help administrators to diagnose system problems. For example, four log messages collected from a supercomputer [urll] in Sandia National Laboratories are listed below:

```
- 1131564688 2005.11.09 en257 Nov 9 11:31:28 en257/en257
ntpd[1978]:  ntpd exiting on signal 15
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  failed
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  ntpd shutdown failed
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  ntpd startup failed
```

The four log messages describe a failure in restarting of the `ntpd` (Network Time Protocol daemon). The system administrators need to first know the reason why the `ntpd` could not restart and then come up with a solution to resolve this problem. A typical approach is to compare the current four log messages with the historical `ntpd` restarting logs and see what is the difference with them. Then the administrators can find out which steps or parameters might cause this failure. To retrieve the relevant historical log messages, the

four log messages can be used as a query to search over the historical event logs. However, the size of the entire historical logs is usually very large, so it is not efficient to go through all event messages. For example, IBM Tivoli Monitoring 6.x [urle] usually generates over 100G bytes system events for just one month from 600 windows servers. Searching over such a large scale event sequence is challenging and the searching index is necessary for speeding up this process. Current system management tools and software can only search a single event by keywords or relational query conditions [urle, urlk, urlh]. However, a system behavior is usually described by several continuous event messages not just one single event, as shown in the above `ntpd` example. In addition, the number of event messages for a system behavior is not a fixed number, so it is hard to decide what is the appropriate segment length for building the index.

Existing search indexing methods for textual data and sequential data can be summarized into two different categories. In our problem, however, each of them has its own limitation. For the textual data, the locality-sensitive hashing (LSH) [GIM99] with the Min-Hash [BCFM98] function is a common scheme. But these LSH based methods only focus on unordered data [GIM99, BCG05, Ste07]. In a textual event sequence, the order information cannot be ignored since different orders indicate different execution flows of the system. For sequential data, the segment search problem is a substring matching problem. Most existing methods are hash index based, suffix tree based, suffix arrays based or BOWTIE based [GV05, MM93, KA05, AGM$^+$90, LTPS09, BRCR94]. These methods can keep the order information of elements, but their sequence elements are single values rather than texts. Their search targets are the matched substrings. In our problem, the similar segments are not necessary to be matched substrings. The detailed discussion of the similarity search over textual data and sequential data is provided in Section 2.5.

5.3.1 Suffix Matrix with Random Mask

Problem Formulation

Let $S = e_1e_2...e_n$ be a sequence of n event messages, where e_i denotes the i-th event, $i = 1, 2, ..., n$. $|S|$ denotes the length of sequence S, which is the number of events in S. $\mathcal{E}$ denotes the universe of events. $sim(e_i, e_j)$ is a similarity function which measures the similarity between event e_i and event e_j, where $e_i \in \mathcal{E}$, $e_j \in \mathcal{E}$. Jaccard coefficient [TSK05] with 2-shingling [BGMZ97] is utilized as the similarity function $sim(\cdot, \cdot)$ because each event is a textual message.

Definition 5.3.1. (Segment) *Given a sequence of events $S = e_1...e_n$, a segment of S is a sequence $L = e_{m+1}e_{m+2}...e_{m+l}$, where l is the length of L, $l \leq n$, and $0 \leq m \leq n - l$.*

The problem is formally stated as follows.

Problem 3. (Problem Statement) *Given an event sequence S and a query event sequence Q, find all segments with length $|Q|$ in S which are similar to Q.*

Similar segments are defined based on the event similarity. Given two segments $L_1 = e_{11}e_{12}...e_{1l}$, $L_2 = e_{21}e_{22}...e_{2l}$, we consider the number of dissimilar events in L_1 and L_2. If the number of dissimilar event pairs is at most k, then L_1 and L_2 are similar. This definition is also called k-dissimilar:

$$N_{dissim}(L_1, L_2, \delta) = \sum_{i=1}^{l} z_i \leq k,$$

where

$$z_i = \begin{cases} 1, & sim(e_{1i}, e_{2i}) < \delta \\ 0, & \text{otherwise} \end{cases},$$

and δ is a user-defined threshold for the event similarity. The k-dissimilar corresponds to the well-known k-mismatch or k-error in the subsequence matching problem [LTP11].

Potential Solutions by LSH

The locality-sensitive hashing (LSH) [GIM99] with the Min-Hash [BCFM98] function is a common scheme for the similarity search over texts and documents. LSH is a s-traightforward solution for our problem. We can consider each segment as a small "document" by concatenating its event messages. Figure 5.35 shows a textual event sequence $S = e_1e_2...e_{i+1}e_{i+2}...$, where e_i is a textual event. In this sequence, every 4 adjacent event messages are seen as a "document", such as L_{i+1}, L_{i+2} and so on. The traditional LSH with the Min-Hash function can be utilized on these small "documents" to speed up the similar search. This solution is called LSH-DOC as a baseline method. However, this solution ignores the order information of events, because the similarity score obtained by the Min-Hash does not consider the order of elements in each "document".

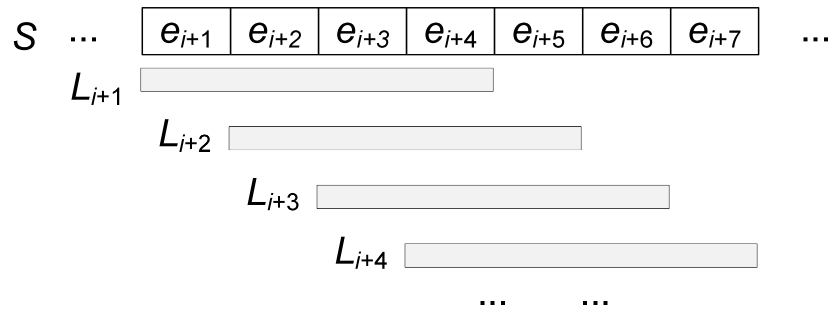

Figure 5.35: An Example of LSH-DOC

To preserve the order information, we can distribute the hash functions to individual regions of segments. For example, the length of the indexed segment is 4, and we have 40 hash functions. We assign every 10 hash functions to every event in the segment. Then, each hash function can only be used to index the events from one region of the segment. Figure 5.36 shows a sequence S with several segments L_{i+1},..., L_{i+4}, where p_1,..., p_4 are 4 regions of each segment and each region contains one event. Every p_j has 10 hash functions

to compute the hash values of the contained event, $j = 1, ..., 4$. If the hash signatures of two segments are identical, it is probably that every region's events are similar. Thus, the order information is preserved. This solution is called LSH-SEP as another baseline method.

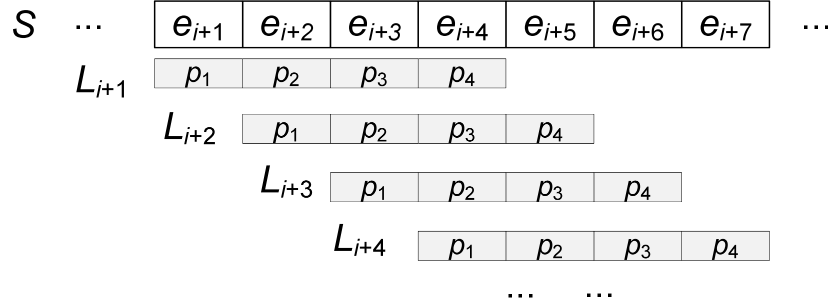

Figure 5.36: An Example of LSH-SEP

k-dissimilar segments are two segments which contain at most k dissimilar events inside. To search the *k*-dissimilar segments, a common approach is to split the query sequence Q into $k+1$ non-overlapping segments. If a segment L has at most *k* dissimilar events to Q, then there must be one segment of Q which has no dissimilar event with its corresponding region of L. Then, we can use any search method for exact similar segments to search the *k*-dissimilar segments. This idea is applied in many biological sequence matching algorithms [AGM$^+$90]. But there is a drawback for the two previous potential solutions: *they all assume that the length of indexed segments* l *is equal to the length of query sequence* $|Q|$. The query sequence Q is given by the user at runtime, so $|Q|$ is not fixed. However, if we do not know the length of the query sequence Q in advance, we cannot determine the appropriate segment length l for building the index. If $l > |Q|$, none of the similar segments could be retrieved correctly. If $l < |Q|$, we have to split Q into shorter subsegments of length l, and then query those shorter subsegments instead of Q. Although all correct similar segments can be retrieved, the search cost would be large, because the subsegments of Q are shorter than Q and the number of retrieve candidates is thus larger [LTP11]. Figure 5.37 shows an example for the case $l < |Q|$. Since the length of indexed segments is l and less than $|Q|$, LSH-DOC and LSH-SEP have to split Q into subsegments L_1, L_2 and L_3, $|L_i| = l$,

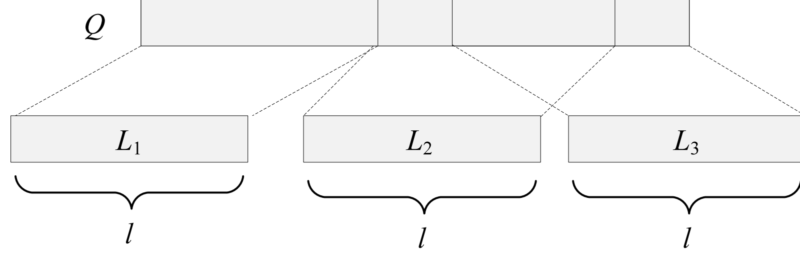

Figure 5.37: An example of $l < |Q|$

$i = 1, .., 3$. Then, LSH-DOC and LSH-SEP use the three subsegments to query the segment candidates. If a segment candidate is similar to Q, its corresponding region must be similar to a subsegment L_i, but not vice versa. Therefore, the acquired candidates for L_i must be more than those for Q. Scanning a large number of candidates is time-consuming. Therefore, the optimal case is $l = |Q|$. But $|Q|$ is not fixed at runtime.

Suffix Matrix Indexing

Let h be a hash function from LSH family. h maps an event to an integer, $h : \mathcal{E} \rightarrow \mathcal{Z}_h$, where $\mathcal{E}$ is the universe of textual events, and $\mathcal{Z}_h$ is the universe of hash values. In suffix matrix, Min-Hash [BCFM98] is the hash function. By taking a Min-Hash function h, a textual event sequence $S = e_1...e_n$ is mapped into a sequence of hash values $h(S) = h(e_1)...h(e_n)$. Suppose we have m independent hash functions, we can have m distinct hash value sequences. Then, we create m suffix arrays from the m hash value sequences respectively. The suffix matrix of S is constructed by the m suffix arrays, where each row is a suffix array.

Definition 5.3.2. (Suffix Matrix) *Given a sequence of events $S = e_1...e_n$ and a set of independent hash functions $H = \{h_1, ..., h_m\}$, let $h_i(S)$ be the sequence of hash values, i.e., $h_i(S) = h_i(e_1)...h_i(e_n)$. The suffix matrix of S is $\mathbf{M}_{S,m} = [A_1^T, ..., A_m^T]^T$, where A_i^T is the suffix array of $h_i(S)$ and $i = 1, ..., m$.*

We illustrate the suffix matrix by an example as follows:

Example 7. *Let S be a sequence of events, $S = e_1e_2e_3e_4$. H is a set of independent hash functions for events, $H = \{h_1, h_2, h_3\}$. For each event and hash function, the computed the hash value is shown in Table 5.13.*

Table 5.13: An Example of Hash Value Table

Event	e_1	e_2	e_3	e_4
h_1	0	2	1	0
h_2	3	0	3	1
h_3	1	2	2	0

Let $h_i(S)$ denote the i-th row of Table 5.13. By sorting the suffixes in each row of Table 5.13, we could get the suffix matrix $\mathbf{M}_{S,m}$ below.

$$\mathbf{M}_{S,m} = \begin{bmatrix} 3 & 0 & 2 & 1 \\ 1 & 3 & 0 & 2 \\ 3 & 0 & 2 & 1 \end{bmatrix}.$$

For instance, the first row of $\mathbf{M}_{S,m}$: 3021, is the suffix array of $h_1(S) = 0210$.

There are a lot of efficient algorithms for constructing the suffix arrays [GV05, MM93, KA05]. The simplest algorithm is sorting all suffixes of the sequence with a time complexity $O(n \log n)$. Thus, the time complexity of constructing the suffix matrix $\mathbf{M}_{S,m}$ is $O(mn \log n)$, where n is the length of the historical sequence and m is the number of hash functions.

Searching over Suffix Matrix

Similar to the traditional LSH, the search algorithm based on a suffix matrix consists of two steps. The first step is to acquire the candidate segments. Those candidates are potential similar segments to the query sequence. The second step is to filter the candidates by computing their exact similarity scores. Since the second step is straightforward and is the same as the traditional LSH, we only present the first step of the search algorithm.

Given a set of independent hash functions $H = \{h_1, ..., h_m\}$ and a query sequence $Q = e_{q1}e_{q2}...e_{qn}$, let $\mathbf{Q}_H = [h_i(e_{qj})]_{m \times n}$, $\mathbf{M}_{S,m}(i)$ and $\mathbf{Q}_H(i)$ denote the i-th rows of $\mathbf{M}_{S,m}$ and $\mathbf{Q}_H$ respectively, $i = 1, ..., m$, $j = 1, ..., n$. Since $\mathbf{M}_{S,m}(i)$ is a suffix array, we obtain these entries that matched with $\mathbf{Q}_H(i)$ by a binary search. $\mathbf{M}_{S,m}$ has m rows, we apply m binary searches to retrieve m entry sets. If one segment appears at least r times in the m sets, then this segment is considered to be a candidate. Parameters r and m will be discussed at a later stage of this section.

Algorithm 3 states the candidates search algorithm. $h(i)$ is the i-th hash function in H. Q_{h_i} is the hash-value sequence of Q mapped by h_i. SA_i is the i-th row of the suffix matrix $\mathbf{M}_{S,m}$, and $SA_i[l]$ is the suffix at position l in SA_i. $CompareAt(Q_{h_i}, SA_i[l])$ is a subroutine to compare the order of two suffixes Q_{h_i} and $SA_i[l]$ for the binary search. If Q_{h_i} is greater than $SA_i[l]$, it returns 1; if Q_{h_i} is smaller than $SA_i[l]$, it returns -1; otherwise, it returns 0. $Extract(Q_{h_i}, SA_i, pos)$ is a subroutine to extract the segments candidates from the position pos. Since H has m hash functions, $C[L]$ records the number of times that the segment L is extracted in the m iterations. The final candidates are only those segments which are extracted for at least r times. The time cost issue will be discussed later.

If a segment L of S is returned by the Algorithm 3, we call L is ***reached*** by this algorithm. We illustrate how the binary search works for one hash function $h_i \in H$ by the following example.

Example 8. *Given an event sequence S with a hash function $h_i \in H$, we compute the hash value sequence $h_i(S)$ shown in Table 5.14. Let the query sequence be Q, and $h_i(Q) = 31$, where each digit represents a hash value. The sorted suffixes of $h_i(S)$ are shown in Table*

Table 5.14: Hash Value Sequence $h_i(S)$

$h_i(S)$	5	3	1	4	3	1	0
Position	0	1	2	3	4	5	6

5.15. We use $h_i(Q) = 31$ to search all matched suffixes in Table 5.15. In Algorithm 3,

Algorithm 3 SearchCandidates (Q, δ)

Parameter: Q : query sequence, δ: threshold of event similarity;
Result: $\mathcal{C}$: segment candidates.

1: Create a counting map $\mathcal{C}$
2: **for** $i = 1$ **to** $|H|$ **do**
3: $Q_{h_i} \leftarrow h_i(Q)$
4: $SA_i \leftarrow \mathbf{M}_{S,m}(i)$
5: $left \leftarrow 0, right \leftarrow |SA_i| - 1$
6: **if** $CompareAt(Q_{h_i}, SA_i[left]) < 0$ **then**
7: **continue**
8: **end if**
9: **if** $CompareAt(Q_{h_i}, SA_i[right]) > 0$ **then**
10: **continue**
11: **end if**
12: $pos \leftarrow -1$
13: *// Binary search*
14: **while** $right - left > 1$ **do**
15: $mid \leftarrow \lfloor (left + right)/2 \rfloor$
16: $ret \leftarrow CompareAt(Q_{h_i}, SA_i[mid])$
17: **if** $ret < 0$ **then**
18: $right \leftarrow mid$
19: **else if** $ret > 0$ **then**
20: $left \leftarrow mid$
21: **else**
22: $pos \leftarrow mid$
23: **break**
24: **end if**
25: **end while**
26: **if** $pos = -1$ **then**
27: $pos \leftarrow right$
28: **end if**
29: *// Extract segment candidates*
30: **for** $L \in Extract(Q_{h_i}, SA_i, pos)$ **do**
31: $\mathcal{C}[L] \leftarrow \mathcal{C}[L] + 1$
32: **end for**
33: **end for**
34: **for** $L \in \mathcal{C}$ **do**
35: **if** $\mathcal{C}[L] < r$ **then**
36: del $\mathcal{C}[L]$
37: **end if**
38: **end for**

by using the binary search, we could find the matched suffix : 310*. Then, the* $Extract$ *subroutine probes the neighborhood of suffix* 310*, to find all matched suffixes with* $h_i(Q)$*. Finally, the two segments at position 4 and 1 are extracted. If the two segments are extracted*

Table 5.15: Sorted Suffixes of $h_i(S)$

Index	Position	Hashed Suffix
0	6	0
1	5	10
2	2	14310
3	4	310
4	1	314310
5	3	4319
6	0	5314310

for at least r independent hash functions, then the two segments are the final candidates returned by the Algorithm 3.

Lemma 5.3.3. *Given an event sequence S and a query event sequence Q, L is a segment of S, $|L| = |Q|$, δ_1 and δ_2 are two thresholds for similar events, $0 \leq \delta_2 < \delta_1 \leq 1$, then:*

- *if $N_{dissim}(L, Q, \delta_1) = 0$, then the probability that L is reached by Algorithm 3 is at least $F(m - r; m, 1 - \delta_1^{|Q|})$;*
- *if $N_{dissim}(L, Q, \delta_2) \geq k$, $1 \leq k \leq |Q|$, then the probability that L is reached by Algorithm 3 is at most $F(m - r; m, 1 - \delta_2^k)$,*

where $F(\cdot; n, p)$ is the cumulative distribution function of Binomial distribution $B(n, p)$, and r is a parameter for Algorithm 3.

Proof. Let's first consider the case $N_{dissim}(L, Q, \delta_1) = 0$, which indicates every corresponding events in L and Q are similar and the similarity is at least δ_1. The hash function h_i belongs to the LSH family, so we have $Pr(h_i(e_1) = h_i(e_2)) = sim(e_1, e_2) \geq \delta_1$. L and Q have $|Q|$ events, so for one hash function, the probability that hash values of all those events are identical is at least $\delta_1^{|Q|}$. Once those hash values are identical, L must be found by a binary search over one suffix array in $\mathbf{M}_{S,m}$. Hence, for one suffix array, the probability of L being found is $\delta_1^{|Q|}$. $\mathbf{M}_{S,m}$ has m suffix arrays. The number of those suffix arrays that L is found follows the Binomial distribution $B(m, \delta_1^{|Q|})$. Then, the probability that there are at least r suffix arrays that L is reached is $1 - F(r; m, \delta_1^{|Q|}) = F(m - r; m, 1 - \delta_1^{|Q|})$.

The second case that $N_{dissim}(L, Q, \delta_2) \geq k$ indicates there are at least dissimilar k events and their similarities are less than δ_2. The probability that hash values of all those events in L and Q are identical is at most δ_2^k. The proof is analogous to that of the first case. □

Lemma 5.3.3 is to ensure that if a segment L is similar to the query sequence Q, then it is very likely to be reached by our algorithm; if L is dissimilar to the query sequence Q, then it is very unlikely to be reached. The probabilities shown in this lemma are the false negative probability and the false positive probability. The choice of r controls the tradeoff between the probabilities. F-measure is a combined measurement for the two factors [SM84]. The optimal r is the one that maximizes the F-measure score. Since r can only be an integer, we can enumerate all possible values of r from 1 to m to find the optimal r.

However, this algorithm cannot handle the case that if there are two dissimilar events inside L and Q. The algorithm narrows down the search space step by step according to each element of Q. A dissimilar event between Q and Q's similar segments in L would lead the algorithm to incorrect following steps.

Randomly Masked Suffix Matrix

Figure 5.38 shows an example of a query sequence Q and a segment L. There is only one dissimilar event pair between Q "1133" and L "1933", which is the second one, '9' in L with '1' in Q. Clearly, the traditional binary search cannot find "1933" by using "1133" as the query. To overcome this problem, a straightforward idea is to skip the dissimilar event between Q and L. However, the dissimilar event can be any event inside L. We do not know which event is the dissimilar event to skip before knowing Q. If two similar segments are allowed to have at most k dissimilar events, the search problem is called the k-dissimilar search. Our proposed method is summarized as follows:

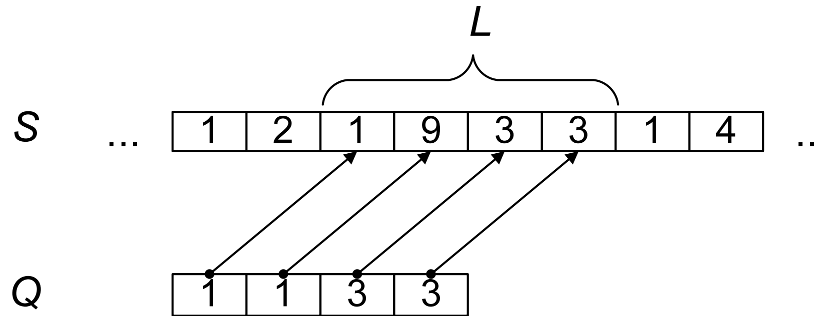

Figure 5.38: Dissimilar Events in Segments

Offline Step:

1. Apply f min-hash functions on the given textual sequence to convert it into f hash-valued sequences.
2. Generate f random sequence masks and apply them to the f hash-valued sequences (one to one).
3. Sort the f masked sequences to f suffix arrays and store them with the random sequence masks to disk files.

Online Step:

1. Apply the f min-hash functions on the given query sequence to convert it into f hash-valued sequences.
2. Load the f random sequence masks and apply them to the f hash-valued query sequences.
3. Invoke f binary searches by using the f masked query sequences over the f suffix arrays and find segment candidates that has been extracted at least r times.

Random Sequence Mask

A sequence mask is a sequence of bits. If these bits are randomly and independently generated, this sequence mask is a random sequence mask.

Definition 5.3.4. *A random sequence mask is a sequence of random bits in which each bit follows Bernoulli distribution with parameter θ: $P(bit = 1) = \theta$, $P(bit = 0) = 1 - \theta$, where $0.5 \leq \theta < 1$.*

Figure 5.39 shows a hash-value sequence $h(S)$ and two random sequence masks: M_1 and M_2. $M_i(h(S))$ is the masked sequence by *AND* operator: $h(S)$ *AND* M_i, where $i = 1, 2$. White cells indicate the events that are kept in $M_i(h(S))$, and dark cells indicate those events to skip. The optimal mask is the one such that all dissimilar events are located in the

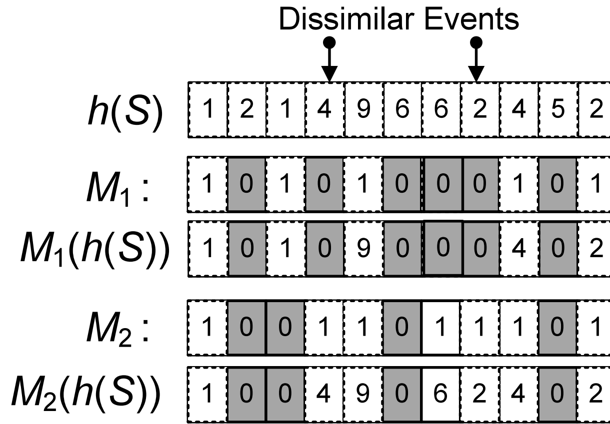

Figure 5.39: Random Sequence Mask

dark cells. In other words, the optimal mask is able to skip all dissimilar events. We call this kind of random sequence masks as the *perfect* sequence masks. In Figure 5.39, there are 2 dissimilar events in S: the 4th event and the 8th event. M_1 skips the 4th event and the 8th event in their masked sequences, so M_1 is a *perfect* sequence mask. Once we have a *perfect* sequence mask, previous search algorithms can be applied on those masked hash value sequences without considering dissimilar events.

Lemma 5.3.5. *Given an event sequence S, a query sequence Q, and f independent random sequence masks with parameter θ, let L be a segment of S, $|Q| = |L|$. If the number of dissimilar event pairs of L and Q is k, then the probability that there are at least m perfect sequence masks is at least $F(f - m; f, 1 - (1 - \theta)^k)$, where F is the cumulative probability function of Binomial distribution.*

Proof. Since each bit in each mask follows the Bernoulli distribution with parameter θ, the probability that the corresponding bit of one dissimilar even is 0 is $1-\theta$ in one mask. Then, the probability that all corresponding bits of k dissimilar events are 0 is $(1-\theta)^k$ in one mask. Hence, the probability that one random sequence mask is a *perfect* sequence mask is $(1-\theta)^k$. Then, $F(f-m; f, 1-(1-\theta)^k)$ is the probability for this case happens m times in f independent random sequence masks. □

Randomly Masked Suffix Matrix

A randomly masked suffix matrix is a suffix matrix, where each suffix array is masked by a random sequence mask. We use $\mathrm{M}_{S,f,\theta}$ to denote a randomly masked suffix matrix, where S is the event sequence to index, f is the number of independent LSH hash functions, and θ is the parameter for each random sequence mask. Note that, $\mathrm{M}_{S,f,\theta}$ still consists of f rows by $n = |S|$ columns.

Lemma 5.3.6. *Given an event sequence S, a randomly masked suffix matrix $\mathrm{M}_{S,f,\theta}$ of S and a query sequence Q, L is a segment of S, $|L| = |Q|$. If the number of dissimilar events between L and Q is at most k, then the probability that L is reached by Algorithm 3 is at least*

$$Pr_{reach} \geq \sum_{m=r}^{f} F(f-m; f, 1-(1-\theta)^k) \cdot F(m-r; m, 1-\delta^{|Q|\cdot\theta}),$$

where δ and r are parameters of Algorithm 3.

This probability combines the two previous probabilities in Lemma 5.3.3 and Lemma 5.3.5. m becomes a hidden variable, which is the number of *perfect* sequence masks. By considering all possible m, this lemma is proved. Here the expected number of kept events in every $|Q|$ events by one random sequence mask is $|Q| \cdot \theta$.

Analytical Search Cost

Given an event sequence S and its randomly masked suffix matrix $\mathbf{M}_{S,f,\theta}$, $n = |S|$, the cost of acquiring candidates mainly depends on the number of binary search on suffixes. Recall that $\mathbf{M}_{S,f,\theta}$ is f by n. Each row of it is a suffix array. f binary searches must be executed. Each binary search cost is $\log n$. The total cost of acquiring candidates is $f \log n$.

The cost of filtering candidates mainly depends on the number of candidates acquired. Let $\mathcal{Z}_h$ denote the universe of hash values. Given an event sequence S and a set of hash functions H, $\mathcal{Z}_{H,S}$ denotes the set of hash values output by each hash function in H with each event in S. $\mathcal{Z}_{H,S} \subseteq \mathcal{Z}_h$, because some hash value may not appear in the sequence S. In average, each event in S has $Z = |\mathcal{Z}_{H,S}|$ distinct hash values. Let Q be the query sequence. For each suffix array in $\mathbf{M}_{S,f,\theta}$, the average number of acquired candidates is:

$$N_{Candidates} = \frac{n}{Z^{|Q| \cdot \theta}}.$$

The total number of acquired candidates is at most $f \cdot N_{Candidate}$. A hash table is used to merge the f sets of candidates into one set. Its cost is $f \cdot N_{Candidate}$. To sum up the two parts, given an interleaved suffix matrix $\mathbf{M}_{S,f,\theta}$ and a query sequence Q, the total search cost is

$$Cost_{search} = f \cdot (\log n + \frac{n}{Z^{|Q| \cdot \theta}}).$$

Why the potential solutions are not efficient?

For potential solutions (i.e., LSH-DOC and LSH-SEP) and suffix matrix, the second part of cost is the major cost of the search. Here we only consider the number of acquired candidates to compare the analytical search cost. The average number of acquired candidates

by LSH-DOC and LSH-SEP is at least:

$$N'_{Candidates} = \frac{n}{Z^{|Q|/(k+1)}}.$$

When $|Q| \cdot \theta \geq \log_Z f + |Q|/(k+1)$, $f \cdot N_{Candidate} \leq N'_{Candidates}$. Z depends on the number of 2-shinglings, which is approximated to the square of the vocabulary size of log messages. Hence, Z is a huge number, $\log_Z f$ can be ignored. Since $\theta \geq 0.5$, $k \geq 1$, we always have $|Q| \cdot \theta \geq |Q|/(k+1)$. Therefore, the acquired candidates of suffix matrix are less than or equal to those of LSH-DOC and LSH-SEP.

Offline Parameter Choice

The parameters f and θ balances the search costs and search result accuracy. These two parameters are decided in the offline step before building the suffix matrix. Let $Cost_{max}$ be the search cost budget, the parameter choosing problem is to maximize Pr_{reach} subject to $Cost_{search} \leq Cost_{max}$. A practical issue is that the suffix matrix is constructed in the offline phase, but $|Q|$ and δ can only be known in the online phase. A simple approach to find out the optimal f and θ is looking at the historical queries to estimate $|Q|$ and δ. This procedure can be seen as a *training* procedure. Once the two offline parameters are obtain, other parameters are found by solving the maximization problem. The objective function Pr_{reach} is not convex, but it can be solved by the enumeration method since all tuning parameters are small integers.

The next question is how to determine $Cost_{max}$. We can choose $Cost_{max}$ according to the average search cost curve. Figure 5.40 shows a curve about the analytical search cost and the probability Pr_{reach}, where $m = \lfloor Cost_{search}/(\log n + \frac{n}{|\mathcal{Z}_{H,S}|^{|Q|\cdot\theta}})\rfloor$. According to this curve, we suggest users to choose $Cost_{max}$ between 100 and 200, because larger search costs would not significantly improve the accuracy any more.

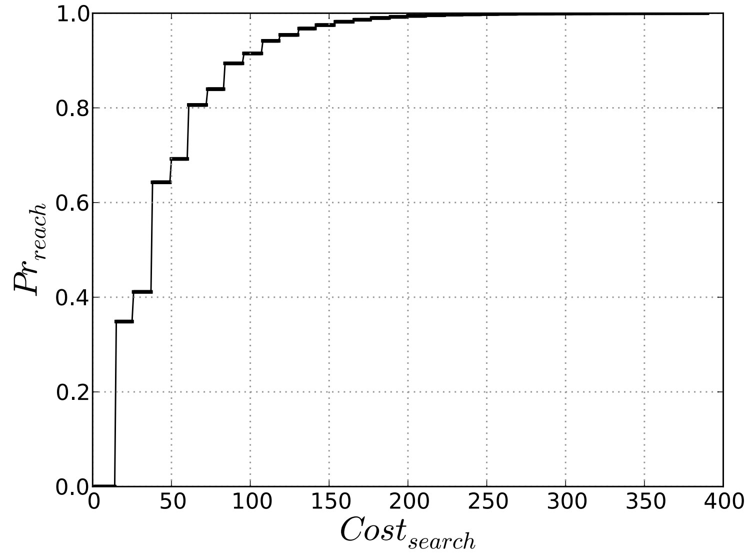

Figure 5.40: Average Search Cost Curve ($n = 100K, |\mathcal{Z}_{H,S}| = 16, \theta = 0.5, |Q| = 10, \delta = 0.8, k = 2$)

Scalability

The time complexity of the offline suffix matrix construction is $O(n \log n)$. The online search is $O(\log n)$. The only problem for scaling suffix matrix when the memory cost exceeds the limitation. In this case, the suffix matrix can be stored in the external memory or a distributed system.

5.3.2 Evaluation

In this section, I conduct experiments on real system event logs to evaluate our proposed method.

Experimental Platform

We implement LSH-DOC, LSH-SEP and our method in Java 1.6. Table 5.16 summarizes our experimental machine.

Table 5.16: Experimental Machine

OS	**CPU**	**JRE**	**JVM Heap Size**
Linux 2.6.18	Intel Xeon(R) @ 2.5GHz, 8 core, 64bits	J2SE 1.6	2G

Data Collection

Our experimental system logs are collected from two different real systems. Apache HTTP error logs are collected from the server machines in the computer lab of a research center and have about 236,055 log messages. Logs of ThunderBird [urll] are collected from a supercomputer in Sandia National Lab. The first 350,000 log messages from the ThunderBird system logs are used for this evaluation.

Testing Queries

Each query sequence is a segment randomly picked from the event sequence. Table 5.17 lists detailed information about the 6 groups, where $|Q|$ indicates the length of the query sequences. The true results for each query are obtained by the *brute-force* method, which scans through every segment of the sequence one by one to find all true results.

Table 5.17: Testing Query Groups

Group	Num. of Queries	$\|Q\|$	k	δ
TG1	100	6	1	0.8
TG2	100	12	3	0.65
TG3	100	18	5	0.6
TG4	100	24	7	0.5
TG5	100	30	9	0.5
TG6	100	36	11	0.5

Baseline Methods

We compare our method with baseline methods LSH-DOC, LSH-SEP stated before. The two methods are both LSH based methods applying to the sequential data. In order to handle the k-dissimilar approximation queries, the indexed segment length l for LSH-DOC and LSH-SEP can be at most $|Q|/(k+1) = 3$, so we set $l = 3$.

Online Searching

Suffix matrix and LSH based methods all consist of two steps. The first step is to search segment candidates from its index. The second step is filtering acquired candidates by computing their exact similarities. Because of the second step, the precision of the search results is always 1.0. Thus, the quality of results only depends on the recall. By appropriate parameter settings, all the methods can achieve high recalls, but we also consider the associated time cost. For a certain recall, if the search time is smaller, the performance is better. An extreme case is the *brute-force* method that always has the 1.0 recall, but it has to visit all segments of the sequence, so the time cost is huge. We define the recall ratio as a normalized metric for evaluating the goodness of the search results:

$$RecallRatio = \begin{cases} \frac{Recall}{SearchTime}, & Recall \geq recall_{min} \\ 0, & \text{otherwise} \end{cases},$$

where $recall_{min}$ is a user-specified threshold for the minimum acceptable recall. If the recall is less than $recall_{min}$, the search result is then not acceptable by the user. In our evaluation, $recall_{min} = 0.5$, which means any method should capture at least half of the true results. The unit of the search time is millisecond. $RecallRatio$ is expressed as the portion of true results obtained per millisecond. Clearly, $RecallRatio$ is higher, the performance is better.

LSH-DOC, LSH-SEP and suffix matrix have different parameters. We vary the value of each parameter in each method, and then select the best performance of each method to compare. LSH-DOC and LSH-SEP have two parameters to set, which are the length of hash vectors b and the number of hash tables t. b varies from 5 to 35. t varies from 2 to 25. We also consider the different number of buckets for LSH-DOC and LSH-SEP. Due to the Java heap size limitation, the number of hash buckets is fixed to be 8000. For suffix matrix,

r is chosen according to the method mentioned before. f and m vary from 2 to 30. θ varies from 0.5 to 1.

Figures 5.41 and 5.42 show the $RecallRatios$ for each testing group. Overall, suffix matrix achieves the best performance on the two data sets. However, LSH based methods outperform suffix matrix on short queries (TG1). Moreover, in Apache Logs with TG4, LSH-SEP is als better than suffix matrix.

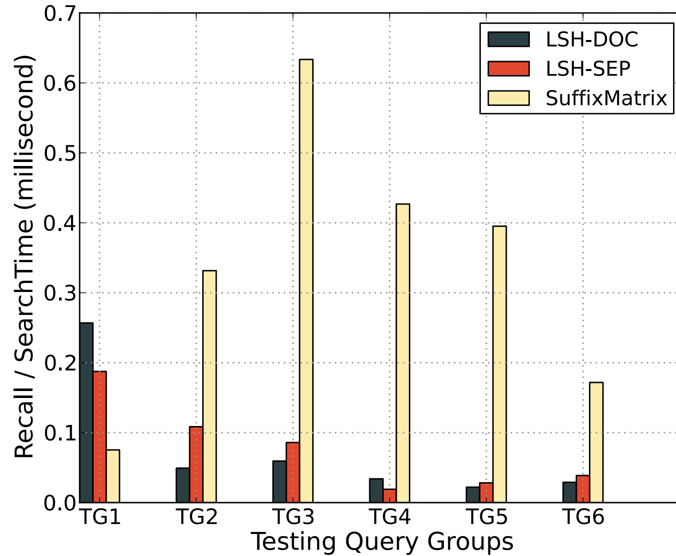

Figure 5.41: RecallRatio comparison for ThunderBird Logs

To find out the reason why in TG1 suffix matrix performs worse than LSH-DOC or LSH-SEP, we record the number of acquired candidates for each method and the number of true results. Figures 5.43 and 5.44 show the actual acquired candidates for each testing group with each method. Table 5.18 shows the numbers of true results for each testing group. From the two figures, we can see that suffix matrix acquired much more candidates than other methods in TG1. In other words, suffix matrix has a higher collision probability of dissimilar segments in its hashing scheme.

To overcome this problem, a common trick in LSH is to make the hash functions be "stricter". For example, there are $d+1$ independent hash functions in LSH family, h_1,...,h_d

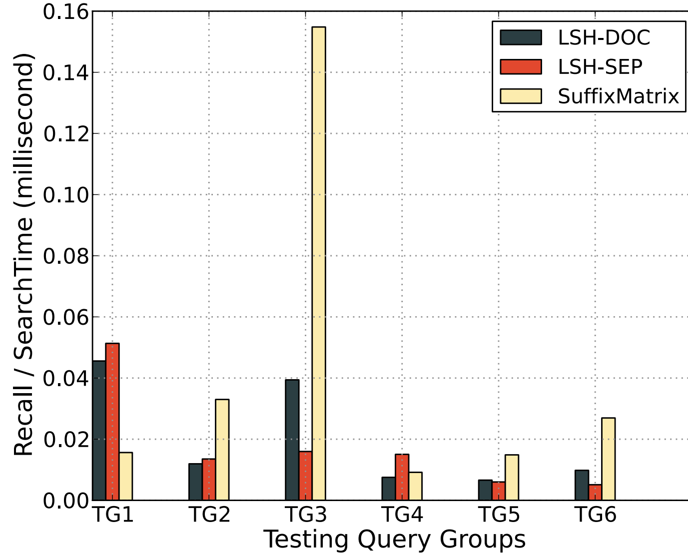

Figure 5.42: RecallRatio comparison for Apache Logs

Table 5.18: Number of True Results

Dataset	TG1	TG2	TG3	TG4	TG5	TG6
ThunderBird Logs	4.12	2.81	27.46	53.24	57.35	7.21
Apache Logs	378.82	669.58	435.94	1139.15	1337.23	990.63

and h. We can construct a "stricter" hash function $h' = h(h_1(x), h_2(x), ..., h_d(x))$. If two events e_1 and e_2 are not similar, i.e., $sim(e_1, e_2) < \delta$, the collision probability of h_i is $Pr[h_i(e_1) = h_i(e_2)] = sim(e_1, e_2) < \delta$, which can be large if δ is large, $i = 1, ..., d$. But the collision probability of h' is

$$\begin{aligned} Pr[h'(e_1) = h'(e_2)] &= \prod_{i=1}^{n} Pr[h_i(e_1) = h_i(e_2))] \\ &= [sim(e_1, e_2)]^d < sim(e_1, e_2). \end{aligned}$$

Figure 5.45 shows the performance of the suffix matrix by using "stricter" hash functions (denoted as "SuffixMatrix(Strict)") in TG1. Each "stricter" hash function is constructed by 20 independent Min-Hash functions. The testing result shows, "SuffixMatrix(Strict)"

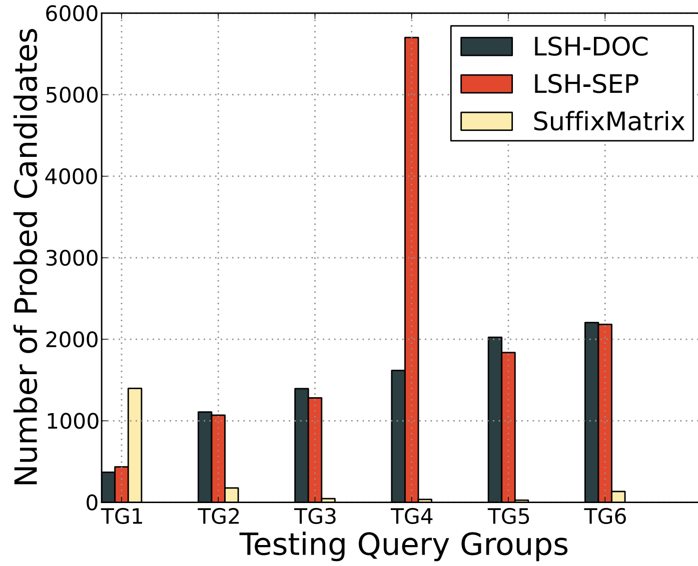

Figure 5.43: Number of Probed Candidates for ThunderBird Logs

outperforms all other methods for both Thunderbird logs and Apache logs in TG1. Table 5.19 are the parameters and other performance measures of "SuffixMatrix(Strict)". By using "stricter" hash functions, the suffix matrix reduces 90% to 95% of previous candidates. As a result, the search time becomes much smaller than before. The choice of the number of hash functions for a "stricter" hash function, d, is a tuning parameter and determined by the data distribution. Note that the parameters of LSH-DOC and LSH-SEP in this test are already tuned by varying the values of b and t.

Table 5.19: "SuffixMatrix(Strict)" for TG1

Dataset	**Parameters**	$Recall$	$SearchTime$	Num. of Probed
ThunderBird Logs	$m = 2, \theta = 0.9$	0.9776	1.23 ms	5.04
Apache Logs	$m = 2, \theta = 0.8$	0.7279	2.24ms	152.75

To verify Lemma 5.3.6, we vary each parameter of suffix matrix and test the recall of search results. We randomly sample 100,000 log messages from the ThunderBird logs and randomly pick 100 event segments as the query sequences. The length of each query sequence is 16. Other querying criteria are $k = 5$ and $\delta = 0.5$. Figure 5.46 shows that

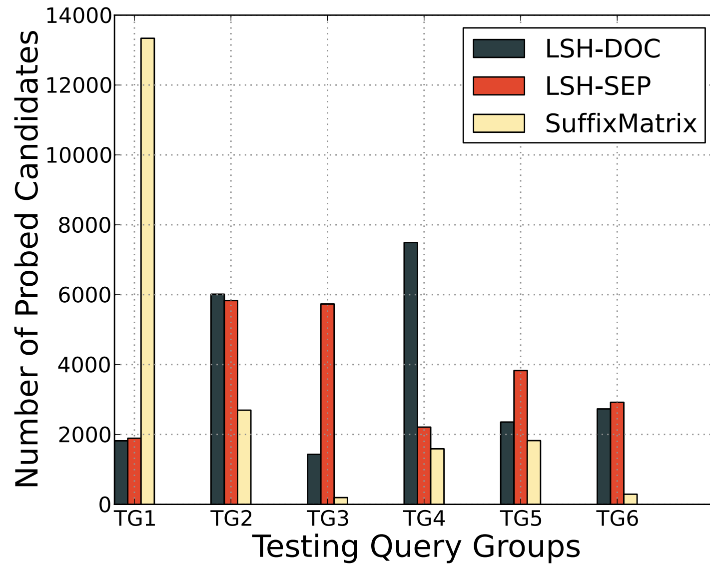

Figure 5.44: Number of Probed Candidates for Apache Logs

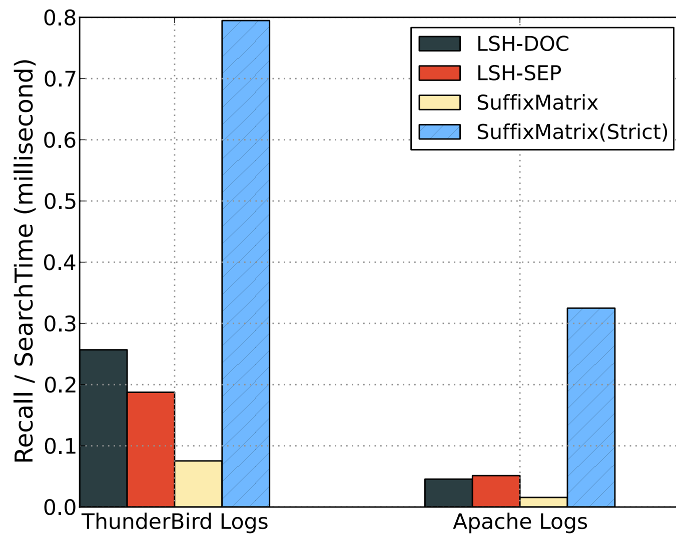

Figure 5.45: RecallRatio for TG1

the increase of m will improve the recall. Figure 5.48 verifies that if r becomes larger, the recall will decrease. Since the random sequence masks are randomly generated, the trends of the recall are not stable and a few jumps are in the curves. But generally, the recall

curves drop down when we enlarge the θ for the random sequence mask. To sum up, the results shown these Figures can partially verify Lemma 5.3.6.

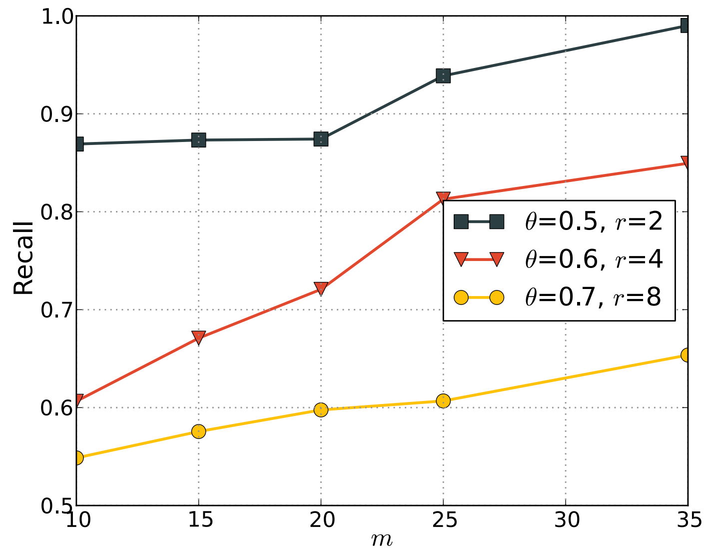

Figure 5.46: Varying m

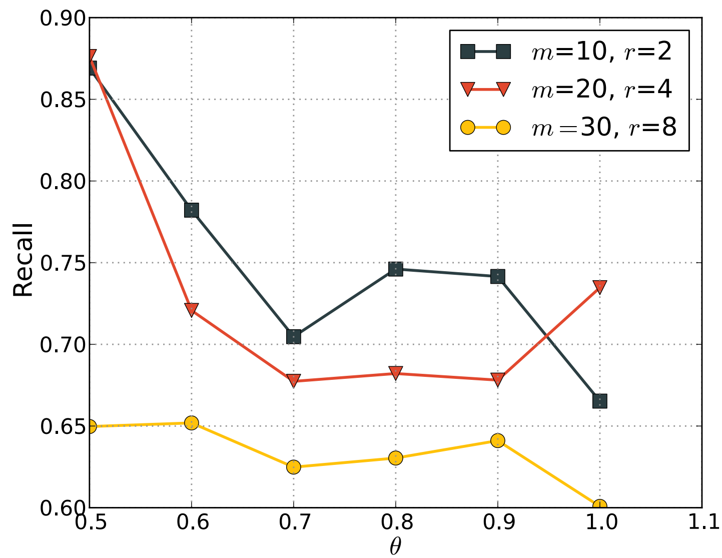

Figure 5.47: Varying θ

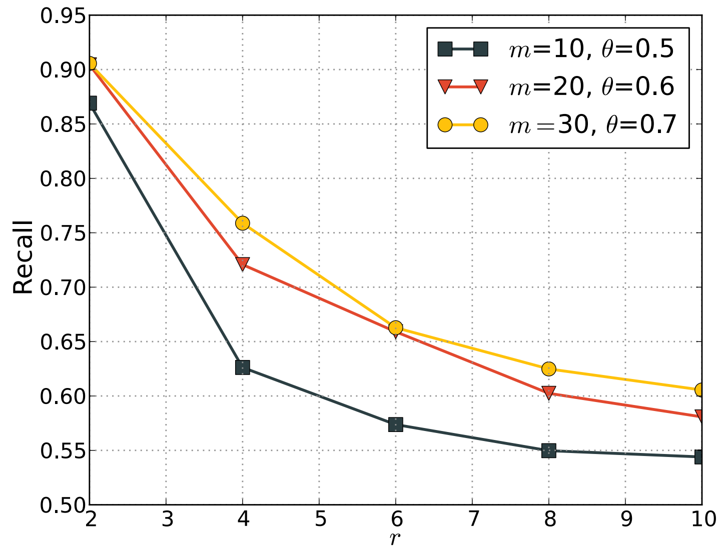

Figure 5.48: Varying r

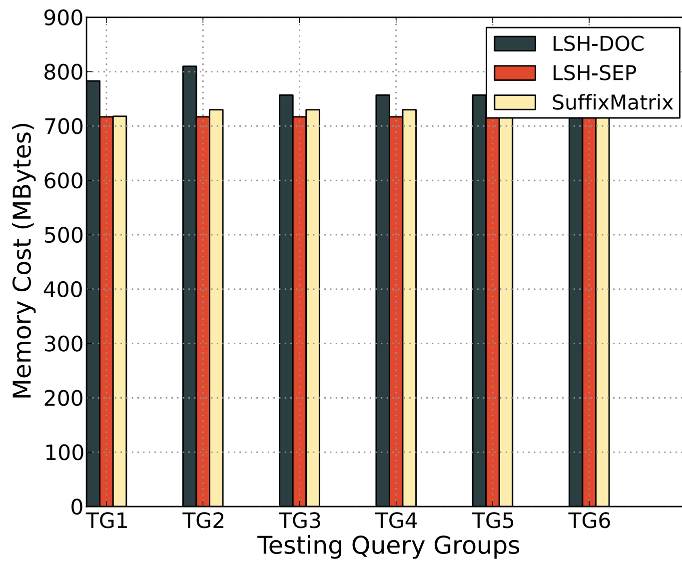

Figure 5.49: Peak Memory Cost for ThunderBird Logs

Offline Indexing

Space cost is an important factor for evaluating these methods [BRCR94] [GV05] [GP09] [LTP11]. If the space cost is too large, the index cannot be loaded into the main memory.

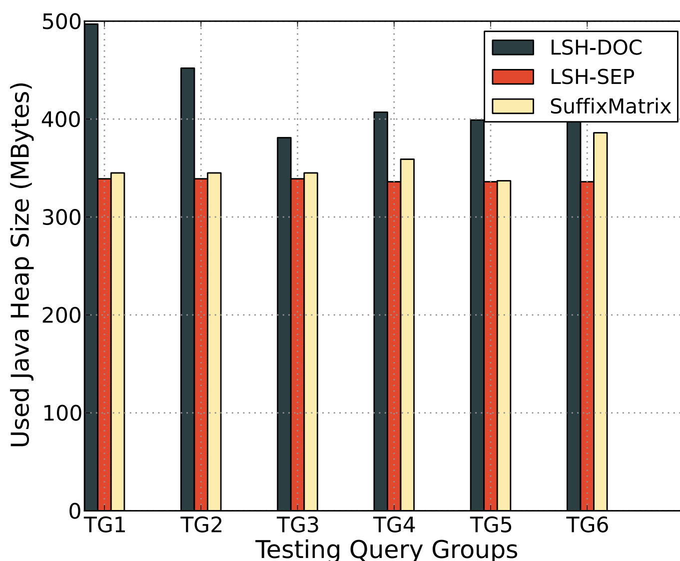

Figure 5.50: Peak Memory Cost for Apache Logs

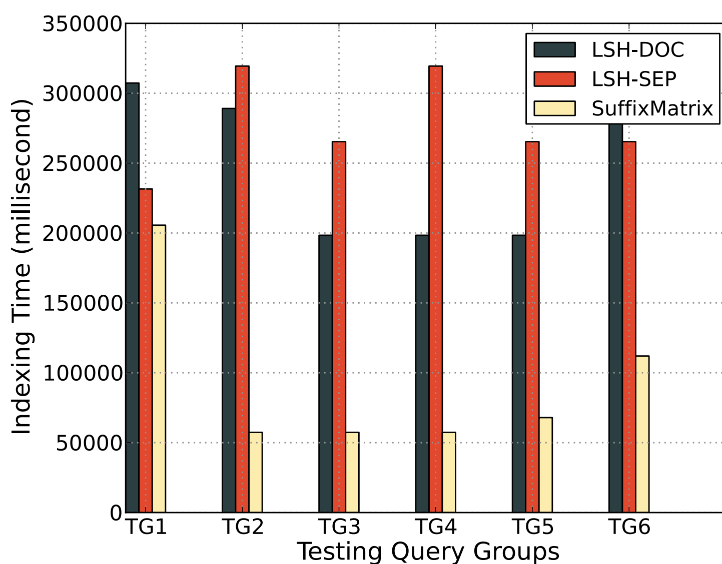

Figure 5.51: Indexing Time for ThunderBird Logs

To exclude the disk I/O cost for the online searching, we load all event messages and index data into the main memory. The total space cost can be directly measured by the allocated heap memory size in JVM. Note that the allocated memory does not only contain the index, it also includes the original log event messages, 2-shinglings of each event message

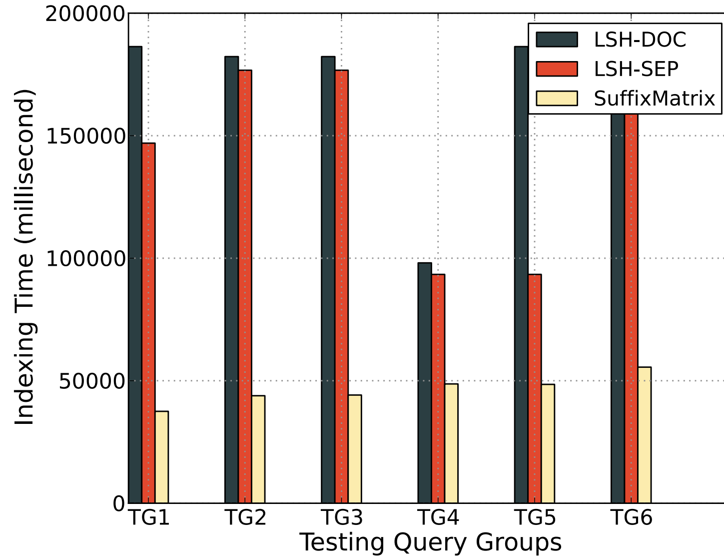

Figure 5.52: Indexing Time for Apache Logs

and the corresponding Java objects information maintained by JVM. We use Java object serialization to compute the exact size of the allocated memory. Figures 5.49 and 5.50 show the total used memory size for each testing group. The parameters of each method are the same as in Figures 5.41 and 5.42. The total space costs for LSH-SEP and suffix matrix are almost the same because they both build the hash index for each event message only once. But LSH-DOC builds the hash indices for each event l times since each event is contained by l continuous segments, where l is the length of the indexed segment and $l = 3$.

Indexing time is the time cost for building the index. Figures 5.51 and 5.52 show the indexing time for each method. The time complexities of LSH-DOC and LSH-SEP are $O(nlbt \cdot c_h)$ and $O(nbt \cdot c_h)$, where n is the number of event messages, l is the indexed segment length, b is the length of the hash vector, t is the number of hash tables, and c_h is the cost of Min-Hash function for one event message. Although for each testing group, the selected LSH-DOC and LSH-SEP may have different b and t, in general LSH-SEP is more efficient than LSH-DOC. The time complexity of suffix matrix for building the index is $O(mn \log n + mn \cdot c_h)$, where m is the number of rows of the suffix matrix. It seems that

the time complexity of suffix matrix is bigger than LSH based methods if we only consider n as a variable. However, as shown in Figures 5.51 and 5.52, suffix matrix is actually the most efficient method in building index. The main reason is $m \ll b \cdot t$. In addition, the time cost of Min-Hash function, c_h, is not small since it has to randomly permute the 2-shinglings of an event message.

5.4 Summary

System diagnosis requires a huge amount of domain knowledge and intensive data analysis. The manpower cost of the ticket resolving is one major cost of all IT service providers. This chapter studies several data-driven approaches for helping domain experts accomplish this task. We first present a novel algorithm for discovering temporal dependencies with time lags, in which the discovered results reveal the dependency among system components and the correlations of monitoring situations. Then, we present several KNN-based recommendation algorithms for automatically recommending incident tickets with their resolutions from a large historical ticket set. The recommendation is based on the relevance of the system problems described by tickets. It also takes into account the falsity of tickets to avoid misleading information of the results. Based on the recommended tickets and resolutions, the system administrators can easily correlate similar system issues happening before and find best practices for handling those issues without manually looking up historical tickets. Finally, we target on the efficient search problem of locating similar system behaviours over large scale textual log sequences. A novel indexing technique is described for facilitating the similarity search. Extensive experiments on real system events, logs and tickets demonstrate the effectiveness and efficiency of the proposed data-driven approaches.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Modern IT infrastructures are constituted by large scale computing systems including various hardware and softwares and often administered by IT service providers. Supporting such complex systems requires a huge amount of domain knowledge and experiences. The manpower cost is one of the major cost for all IT service providers. Service providers often seek automatic or semi-automatic methodologies of detecting and resolving system issues to improve their service quality and efficiency. This dissertation investigates several data-driven approaches for improving the quality and efficiency of IT service and system management. The improvements focus on three components of the service workflow: data preprocess, system monitoring and system diagnosis. Data preprocess involves extracting various raw system logs and converting them into a well formatted data warehouse of the service provider. System monitoring is usually provided by monitoring software running on the customer servers, which computes metrics for the hardware and software performance at regular intervals. The metrics are then compared to acceptable thresholds (known as monitoring situations), and any violation results in an alert. If the alert persists beyond a certain delay specified in the situation, the monitor emits an event. Events coming from a customer's entire IT environment are consolidated in an enterprise console. The console uses rule-, case- or knowledge-based engines to analyze the monitoring events and decide whether to open a service ticket in the Incident, Problem, Change system. Additional tickets are created upon customer request. System diagnosis is performed on the created tickets by system administrators. Each ticket is assigned to one or several administrators. The assigned administrator then checks the reported system, inspects the root cause of the

described issues, and executes corrective actions to resolve the tickets. The information accumulated in the ticket records the problem determination and resolution.

In particular, in the aspect of data preprocess, this dissertation presents two novel textual clustering algorithms for preprocessing textual system logs to structured system events. The structured system events are easier to analyze and explore by system administrators. For system monitoring, this dissertation focuses on the problem of eliminating false alarms (false positives) and missing alarms (false negatives) by refining the configurations of monitoring systems. Several reasons of triggering false positives and false negatives are analyzed based on a large amount of historical monitoring events and tickets collected from several IT service providers. Based on the revealed seasons, a rule based alert prediction algorithm is proposed for eliminating false alarms (false positives) without losing any real alarm and a textual classification method is applied to automatically discover the missing alerts (false negatives) from manual incident tickets. For system diagnosis assistance, this dissertation presents an efficient algorithm for discovering the temporal dependencies between system events with time lags, which can help the administrators to determine the redundancies of deployed monitoring situations and dependencies of system components. To improve the efficiency of incident ticket resolving, KNN-based recommendation algorithms are investigated to recommend relevant historical tickets with resolutions for the administrators. Finally, this dissertation offers a novel algorithm for searching similar textual event segments over large system logs and assisting experts to locate similar system behaviours in the logs. Extensive empirical evaluation on system logs/events/tickets from real large IT infrastructures demonstrates the effectiveness and efficiency of the proposed data-driven approaches by this dissertation.

6.2 Limitation of Proposed Methods and Future work

6.2.1 System Event Generation

The proposed methods for preprocessing raw textual logs to system events only generate the discrete events. It would be more helpful if the algorithm extracts the detailed attribute values from the log messages into the events, such as the IP address, machine name, and available disk space. This work is related to the information extraction technique, which is a widely studied area in natural language processing. However, as mentioned previously, different system logs have different formats and structures. Building an extractor for various system logs is challenging. Meanwhile, many information extraction approaches are learning based algorithms. They require the user to provide a set of annotated data to train the model. Annotating various log messages is time-consuming for humans. As for the future work, we consider some semi-supervised learning algorithm that only needs a small amount of annotated data or partially annotated data. The algorithm can infer the appropriate format and structure from the small amount of training data and automatically utilize other unannotated data to build the model.

As for the message signature based clustering algorithm `LogSig`, we consider the three aspects to investigate in the future. First, we consider using the partial match rather than the longest common subsequence to compute the match score. Second, the match score can be also normalized as the match ratio, which is percentage of the matched terms between two log messages. Finally, in some application cases, not all the matched words have the same importance to indicate the event type. Therefore, it is natural to add different weights for different matched terms in the computation of the match score.

6.2.2 Monitoring Optimization and Resolution Recommendation

In the proposed methods for improving monitoring system and recommending relevant ticket resolution, the ticket data is seen as the ground truth for solving the described system issues. In real scenario, service providers have over thousands of system administrators. Some of them are experienced, but some of them are lack of experience or have different expertise to determine the real causes of incident tickets. Therefore, the information in the historical tickets may not be always precise and correct. It is possible that noisy and inconsistent resolutions are contained by the given ticket data set. Therefore, the results generated by the proposed methods can be conflict or hazard. In the future work, we consider the uncertainty of each historical ticket. We hope to build an additional assessment model to determine the quality of tickets and add the quality score into our methods.

6.2.3 Temporal Dependency and Lag Discovery

The time complexities of the proposed *STScan* and *STScan** algorithms are $O(N^2)$ and $O(N^2 \log N)$ respectively, where N is the number of items. Although we prove that there is no algorithm that can find all qualified lag intervals in $o(N^2)$, this time cost is still too high for large data sets in practice. In the future work, we will work on deriving an approximate algorithm for solving this problem. We hope to find a randomized approach that can find all qualified lag intervals with a high probability but the time complexity can be reduced to $O(N)$. Moreover, since most event sequences are collected as streaming data, it is also useful to come up with a streaming algorithm that can incrementally discover the qualified lag intervals without storing the entire sequence of data.

6.2.4 Similarity Search over Textual Event Sequence

In many real applications, the textual event sequence is collected in an incremental manner. New events are appended into the historical data set periodically. The current indexing method of suffix matrix has to rebuild the entire index for each append. As the data set becomes large, rebuilding the entire index would become impractical. Based on the proposed suffix matrix method, in the next step we will consider to develop a dynamic indexing algorithm that can incrementally append new data objects into an existing index without rebuilding all index.

BIBLIOGRAPHY

[ABCM09] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Proceedings of ECML/PKDD*, pages 227–243, Bled, Slovenia, September 2009.

[ABD$^+$07] Naga Ayachitula, Melissa J. Buco, Yixin Diao, Maheswaran Surendra, Raju Pavuluri, Larisa Shwartz, and Christopher Ward. IT service management automation - a hybrid methodology to integrate and orchestrate collaborative human centric and automation centric workflows. In *IEEE SCC*, pages 574–581, 2007.

[ADNR07] Shipra Agrawal, Supratim Deb, K. V. M. Naidu, and Rajeev Rastogi. Efficient detection of distributed constraint violations. In *Proceedings of ICDE*, pages 1320–1324, Istanbul, Turkey, 2007.

[AFGY02] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of KDD*, pages 429–435, 2002.

[AGM$^+$90] Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[AI06] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of FOCS*, pages 459–468, Berkeley, CA, USA, September 2006.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proccedings of VLDB*, pages 487–499, 1994.

[BBM04] Mikhail Bilenko, Sugato Basu, and Raymond J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of ICML*, Alberta, Canada, July 2004.

[BCFM98] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of STOC*, pages 327–336, Dallas, Texas, USA, May 1998.

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[Ben90] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proceedings of the Sixth Annual Symposium on Computational Geometry (SoCG)*, pages 187–197, Berkeley, California, USA, June 1990.

[BGMZ97] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks (CN)*, 29(8-13):1157–1166, March 1997.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 2006.

[BJR12] Anne Bouillard, Aurore Junier, and Benoit Ronot. Hidden anomaly detection in telecommunication networks. In *Proceedings of CNSM*, pages 82–90, 2012.

[BK07] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *ICDM*, pages 43–52, 2007.

[BKWZ07] Sergey Bereg, Marcin Kubica, Tomasz Walen, and Binhai Zhu. RNA multiple structural alignment with longest common subsequences. *Journal of Combinatorial Optimization*, 13(2):179–188, 2007.

[BO07] Khellaf Bouandas and Aomar Osmani. Mining association rules in temporal sequences. In *Proceedings of CIDM*, pages 610–615, 2007.

[BRCR94] Paul Bieganski, John Riedl, John V. Carlis, and Ernest F. Retzel. Generalized suffix trees for biological sequence data: Applications and implementation. In *Proceedings of HICSS*, pages 35–44, Dallas, Texas, USA, May 1994.

[BW94] Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[CMB08] Lius A. Castillo, Paul D. Mahaffey, and Jeff P. Bascle. Apparatus and method for monitoring objects in a network and automatically validating events relating to the objects. U.S. Patent, December 2008. US 7,469,287 B1.

[CS04] Aron Culotta and Jeffrey S. Sorensen. Dependency tree kernels for relation extraction. In *Proceedings of ACL*, pages 423–429, Barcelona, Spain, July 2004.

[Dhu10] Amit Dhurandhar. Learning maximum lag for grouped graphical granger models. In *ICDM Workshops*, pages 217–224, 2010.

[DK04] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems*, 22(1):143–177, January 2004.

[DL05] Yi Ding and Xue Li. Time weight collaborative filtering. In *ACM CIKM*, pages 485–492, 2005.

[Dud76] Sahibsingh A. Dudani. The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems Man and Cybernetics*, SMC-6(4):325–327, april 1976.

[ESV03] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM Conference*, pages 137–148, 2003.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of VLDB*, pages 518–529, Edinburgh, Scotland, UK, September 1999.

[GJCH09] Jing Gao, Guofei Jiang, Haifeng Chen, and Jiawei Han. Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems. In *Proceedings of ICDCS*, pages 623–630, 2009.

[GKK$^+$09] Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. Sequential dependencies. *PVLDB*, 2(1):574–585, 2009.

[GO95] Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5:165–185, 1995.

[GP09] Mohammadreza Ghodsi and Mihai Pop. Inexact local alignment search over suffix arrays. In *Proceedings of BIBM*, pages 83–87, Washington, DC, USA, September 2009.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD conference*, pages 47–57, Boston, Massachusetts, USA, June 1984.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[HB96] Antonio Hernandez-Barrera. Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 289–294, August 1996.

[HE03] Greg Hamerly and Charles Elkan. Learning the k in k-means. In *Proceedings of NIPS*, Vancouver, British Columbia, Canada, December 2003.

[HKP05] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques, 2ed.* Morgan Kaufmann, 2005.

[HMP02] Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 43(3):475–493, 2002.

[HPMA$^+$00] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proccedings of KDD*, pages 355–359, 2000.

[HSF06] Evan Hoke, Jimeng Sun, and Christos Faloutsos. InteMon: Intelligent system monitoring on large clusters. In *Proceedings of VLDB*, pages 1239–1242, 2006.

[KA05] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

[Kar01] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *CIKM*, pages 247–254, 2001.

[Kor09] Yehuda Koren. Collaborative filtering with temporal dynamics. In *KDD*, pages 447–456, 2009.

[KRRS08] Srinivas R. Kashyap, Jeyashankher Ramamirtham, Rajeev Rastogi, and Pushpraj Shukla. Efficient constraint monitoring using adaptive thresholds. In *Proceedings of ICDE*, pages 526–535, Cancun, Mexico, 2008.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of ACM SIGMOD conference*, pages 369–380, Tucson, Arizona, USA, May 1997.

[KT08] Jerry Kiernan and Evimaria Terzi. Constructing comprehensive summaries of large event sequences. In *Proceedings of ACM KDD*, pages 417–425, Las Vegas, Nevada, USA, August 2008.

[KYY+95] S. Kliger, Shaula Yemini, Yechiam Yemini, David Ohsie, and Salvatore J. Stolfo. A coding approach to event correlation. In *Integrated Network Management*, pages 266–277, 1995.

[LC08] Xiang Lian and Lei Chen. Efficient similarity search over future stream time series. *TKDE*, 20(1):40–54, 2008.

[LDH+10] Zhenhui Li, Bolin Ding, Jiawei Han, Roland Kays, and Peter Nye. Mining periodic behaviors for moving objects. In *Proccedings of KDD*, pages 1099–1108, 2010.

[Li06] Jiuyong Li. Robust rule-based prediction. *IEEE Trans. Knowl. Data Eng. (TKDE)*, 18(8):1043–1054, August 2006.

[LLMP05] Tao Li, Feng Liang, Sheng Ma, and Wei Peng. An integrated framework on mining logs files for computing system management. In *Proceedings of ACM KDD*, pages 776–781, August 2005.

[LM04] Tao Li and Sheng Ma. Mining temporal patterns without predefined time windows. In *Proceedings of ICDM*, pages 451–454, November 2004.

[LMX11] Liwei Liu, Nikolay Mehandjiev, and Dong-Ling Xu. Multi-criteria service recommendation based on user criteria preferences. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 77–84, 2011.

[LSST+02] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444, March 2002.

[LSU05] Srivatsan Laxman, P. S. Sastry, and K. P. Unnikrishnan. Discovering frequent episodes and learning hidden markov models: A formal connection. *IEEE Trans. Knowl. Data Eng.*, 17(11):1505–1517, 2005.

[LSU07] Srivatsan Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *Proceedings of ACM KDD*, pages 410–419, August 2007.

[LTP11] Yinan Li, Allison Terrell, and Jignesh M. Patel. Wham: A high-throughput sequence alignment method. In *SIGMOD*, 2011.

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10, 2009.

[LV02] Yihua Liao and V. Rao Vemuri. Using text categorization techniques for intrusion detection. In *USENIX Security Symposium*, pages 51–59, 2002.

[Mai78] David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, April 1978.

[MF10] Fabian Mörchen and Dmitriy Fradkin. Robust mining of time intervals with semi-interval partial order patterns. In *Proccedings of SDM*, pages 315–326, 2010.

[MH01a] Sheng Ma and Joseph L. Hellerstein. Mining mutually dependent patterns. In *Proceedings of ICDE*, pages 409–416, 2001.

[MH01b] Sheng Ma and Joseph L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of ICDE*, pages 205–214, 2001.

[Mit10] Theophano Mitsa. *Temporal Data Mining*. Chapman and Hall/CRC, 2010.

[MJ93] Steven Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference*, pages 259–270, 1993.

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[MMY+10] Gengxin Miao, Louise E. Moser, Xifeng Yan, Shu Tao, Yi Chen, and Nikos Anerousis. Generative models for ticket resolution in expert networks. In *KDD*, pages 733–742, 2010.

[Mör06] Fabian Mörchen. Algorithms for time series knowledge mining. In *Proccedings of KDD*, pages 668–673, 2006.

[MR04] Nicolas Méger and Christophe Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *Proccedings of PKDD*, pages 313–324, 2004.

[MS99] Christopher D. Manning and Hinrich Schuetze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[MSGL09] Patricia Marcu, Larisa Shwartz, Genady Grabarnik, and David Loewenstern. Managing faults in the service delivery process of service provider coalitions. In *IEEE SCC*, pages 65–72, 2009.

[MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[MZHM09] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of ACM KDD*, pages 1255–1264, Paris, France, June 2009.

[NK11] Xia Ning and George Karypis. SLIM: Sparse linear methods for top-n recommender systems. In *ICDM*, pages 497–506, 2011.

[NNL06] Kang Ning, Hoong Kee Ng, and Hon Wai Leong. Finding patterns in biological sequences by longest common subsequencesand shortest common supersequences. In *Proceedings of BIBE*, pages 53–60, Arlington, Virginia, USA, 2006.

[OAS08] Adam J. Oliner, Alex Aiken, and Jon Stearley. Alert detection in system logs. In *Proceedings of IEEE ICDM*, pages 959–964, 2008.

[OS07] Adam J. Oliner and Jon Stearley:. What supercomputers say: A study of five system logs. In *Proceedings of DSN 2007*, pages 575–584, Edinburgh, UK, June 2007.

[PHMA+01] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of EDBT*, pages 215–224, 2001.

[PMM+94] Michael J. Pazzani, Christopher J. Merz, Patrick M. Murphy, Kamal Ali, Timothy Hume, and Clifford Brunk. Reducing misclassification costs. In *Proceedings of ICML*, pages 217–225, New Brunswick, NJ, USA, July 1994.

[Pop02] Ivan Popivanov. Similarity search over time series data using wavelets. In *ICDE*, pages 212–221, 2002.

[PPLW07] Wei Peng, Charles Perng, Tao Li, and Haixun Wang. Event summarization for system management. In *Proceedings of ACM KDD*, pages 1028–1032, 2007.

[PTG+03] C.S. Perng, D. Thoenen, G. Grabarnik, S. Ma, and J. Hellerstein. Data-driven validation, completion and construction of event relationship networks. In *Proceedings of ACM SIGKDD*, pages 729–734, 2003.

[RBV03] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:164–206, 2003.

[RLS+97] Marcus J. Ranum, Kent Landfield, Michael T. Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a generalized tool for network monitoring. In *USENIX Systems Administration Conference*, pages 1–8, 1997.

[Ros95] Sheldon M. Ross. *Stochastic Processes*. Wiley, 1995.

[SA96a] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM SIGMOD*, pages 1–12, 1996.

[SA96b] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of EDBT*, pages 3–17, 1996.

[SCT+08] Qihong Shao, Yi Chen, Shu Tao, Xifeng Yan, and Nikos Anerousis. EasyTicket: a ticket routing recommendation engine for enterprise problem resolution. *PVLDB*, 1(2):1436–1439, 2008.

[SKKR00] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John T. Riedl. Application of dimensionality reduction in recommender system – a case study. In *ACM WebKDD Workshop*, 2000.

[SM84] Gerard Salton and Michael McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1984.

[SM08] Vikas Sindhwani and Prem Melville. Document-word co-regularization for semi-supervised sentiment analysis. In *Proceedings of ICDM*, pages 1025–1030, 2008.

[SOR+03] Ramendra K. Sahoo, Adam J. Oliner, Irina Rish, Manish Gupta, José E. Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of ACM KDD*, pages 426–435, 2003.

[Ste04] John Stearley. Towards informatic analysis of syslogs. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 309–318, San Diego, California, USA, September 2004.

[Ste07] Benno Stein. Principles of hash-based text retrieval. In *SIGIR*, pages 527–534, 2007.

[TH01] Loren Terveen and Will Hill. Beyond recommender systems: Helping people help each other. In *HCI in the New Millennium*, pages 487–509, 2001.

[TKK06] Sergios Theodoridis and 3rd edition Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, 2006.

[TL10] Liang Tang and Tao Li. LogTree: A framework for generating system events from raw textual logs. In *Proceedings of ICDM*, pages 491–500, December 2010.

[TLCZ13] Liang Tang, Tao Li, Shu-Ching Chen, and Shunzhi Zhu. Searching similar segments over textual event sequences. In *Proceedings of ACM CIKM*, pages 329–338, 2013.

[TLP11] Liang Tang, Tao Li, and Chang-Shing Perng. LogSig: Generating system events from raw textual logs. In *Proceedings of ACM CIKM*, pages 785–794, 2011.

[TLP+12] Liang Tang, Tao Li, Florian Pinel, Larisa Shwartz, and Genady Grabarnik. Optimizing system monitoring configurations for non-actionable alerts. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, pages 34–42, 2012.

[TLP+13] Liang Tang, Tao Li, Florian Pinel, Larisa Shwartz, and Genady Grabarnik. An integrated framework for optimizing automatic monitoring systems in large it infrastructures. In *Proceedings of ACM KDD*, 2013.

[TLS12] Liang Tang, Tao Li, and Larisa Shwartz. Discovering lag intervals for temporal dependencies. In *Proceedings of ACM SIGKDD*, pages 633–641, 2012.

[TLSG13a] Liang Tang, Tao Li, Larisa Shwartz, and Genady Grabarnik. Identifying missed monitoring alerts based on unstructured incident tickets. In *Proceedings of CNSM*, pages 143–146, 2013.

[TLSG13b] Liang Tang, Tao Li, Larisa Shwartz, and Genady Grabarnik. Recommending resolutions for problems identified by monitoring. In *Proceedings of IEEE/IFIP International Symposium on Integrated Network Management*, pages 134–142, 2013.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[urla] Apache HTTP Server : An Open-Source HTTP Web Server. `http://httpd.apache.org/`.

[urlb] FileZilla: An open-source and free FTP/SFTP solution. `http://filezilla-project.org`.

[urlc] Hadoop : An Open-Source MapReduce computing platform. `http://hadoop.apache.org/`.

[urld] HP OpenView : Network and Systems Management Products. `http://www8.hp.com/us/en/software/enterprise-software.html`.

[urle] IBM Tivoli : Integrated Service Management software. `http://www-01.ibm.com/software/tivoli/`.

[urlf] IBM Tivoli Monitoring. `http://www-01.ibm.com/software/tivoli/products/monitor/`.

[urlg] ITIL. `http://www.itil-officialsite.com`.

[urlh] LogLogic: A real-time log analysis and report generation system. `http://www.splunk.com/`.

[urli] MySQL: The world's most popular open source database. `http://www.mysql.com`.

[urlj] PVFS2 : The state-of-the-art parallel I/O and high performance virtual file system. http://pvfs.org.

[urlk] Splunk: A commercial machine data managment engine. http://www.splunk.com/.

[urll] ThunderBird: A supercomputer in Sandia National Laboratories. http://www.cs.sandia.gov/~jrstear/logs/.

[WE11] Bruno Wassermann and Wolfgang Emmerich. Monere: Monitoring of service compositions for failure diagnosis. In *ICSOC*, pages 344–358, 2011.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of FOCS*, pages 1–11, Iowa City, Iowa, USA, September 1973.

[WLZG11] Dingding Wang, Tao Li, Shenghuo Zhu, and Yihong Gong. iHelp: An intelligent online helpdesk system. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 41(1):173–182, 2011.

[WWLW10] Peng Wang, Haixun Wang, Majin Liu, and Wei Wang. An algorithmic approach to event summarization. In *Proceedings of ACM SIGMOD*, pages 183–194, Indianapolis, Indiana, USA, June 2010.

[XHF+08] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Mining console logs for large-scale system problem detection. In *SysML*, December 2008.

[XHF+09a] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Large-scale system problem detection by mining console logs. In *Proceedings of ACM SOSP*, Big Sky, Montana, USA, October 2009.

[XHF+09b] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Online system problem detection by mining patterns of console logs. In *Proceedings of IEEE ICDM*, pages 588–597, 2009.

[XZB05] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *ACM SIGCOMM Conference*, pages 169–180, 2005.

[YH03] Xiaoxin Yin and Jiawei Han. CPAR: Classification based on predictive association rules. In *Proceedings of SDM*, 2003.

[YPZ10] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repair vs. recomposition for broken service compositions. In *ICSOC*, pages 152–166, 2010.

[ZS06] Wei-Xing Zhou and Didier Sornette. Non-parametric determination of real-time lag structure between two time series: The 'optimal thermal causal path' method with applications to economic data. *Journal of Macroeconomics*, 28(1):195 – 224, 2006.

VITA

LIANG TANG

Jan 2, 1983	Born, Chongqing, P.R. China
2006	B.A., Computer Science Sichuan University Chengdu, P.R. China
2009	M.S., Computer Science Sichuan University Chengdu, P.R. China
2009–Present	Ph.D., Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Liang Tang, Romer Rosales, Ajit Singh, Deepak Agarwal, (2013). *Automatic Ad Format Selection via Contextual Bandits*. In Proceedings of the 22th ACM Conference on Information and Knowledge Management, San Francisco, USA, Dec, Pages 1587-1594.

Liang Tang, Tao Li, Shu-Ching Chen, Shuzhi Zhu, (2013). *Searching Similar Segments over Textual Event Sequences*. In Proceedings of the 22th ACM Conference on Information and Knowledge Management, San Francisco, USA, Dec.

Li Zheng, Chao Shen, Liang Tang, Chunqiu Zeng, Tao Li, Steve Luis, Shu-Ching Chen, (2013). *Data Mining Meets the Needs of Disaster Information Management*, IEEE Transactions on Human-Matching Systems, 2013, Volume 43, Issue: 5, September 2013, Pages 451 - 464.

Shunzhi Zhu, Liang Tang, Tao Li, (2013). *Finding multiple global linear correlations in sparse and noisy data sets*. Knowledge-Based Systems, 2013, Volume 53, November 2013, Pages 40-50.

Liang Tang, Tao Li, Larisa Shwartz, Genady Ya. Grabarnik, (2013). *Identifying Missed Monitoring Alerts based on Unstructured Incident Tickets*. In Proceedings of the 9th International Conference on Network and Service Management, Zurich, Swizerland, Oct. 2013, Pages 143-146.

Liang Tang, Tao Li, Larisa Shwartz, Florian Pinel, Genady Ya. Grabarnik, (2013). *An Integrated Framework for Optimizing Automatic Monitoring Systems in Large IT Infras-*

tructures. In Proceedings of the 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Chicago, USA, Aug., Pages 1249-1257.

Liang Tang, Tao Li, Yexi Jiang, Zhiyuan Chen, (2013). *Dynamic Query Forms for Database Queries*. IEEE Transactions on Knowledge and Data Engineering(TKDE), 19 April 2013.

Liang Tang, Tao Li, Larisa Shwartz, Genady Grabarnik, (2013). *Recommending Resolutions for Problems Identified by Monitoring*. In Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, Pages 134-142.

Li Zheng, Chao Shen, Liang Tang, Chunqiu Zeng, Tao Li, Steve Luis, Shu-Ching Chen and Jainendra K. Navlakha, (2012). *Disaster SitRep - A Vertical Search Engine and Information Analysis Tool in Disaster Management Domain*. In Proceedings of the 13th IEEE International Conference on Information Integration and Reuse, Pages 457-465.

Liang Tang, Tao Li, Larisa Shwartz, (2012).*Discovering Lag Intervals for Temporal Dependencies*. In Proceedings of the 18th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Beijing, China, Aug, Pages 633-641.

Liang Tang, Tao Li, Florian Pinel, Larisa Shwartz, Genady Grabarnik, (2012). *Optimizing System Monitoring Configurations for Non-Actionable Alerts*. In Proceedings of IEEE/IFIP Network Operations and Management Symposium, Pages 34-42.

Liang Tang, Tao Li, Chang-Shing Perng, (2011). *LogSig: Generating System Events from Raw Textual Logs*. In Proceedings of the 20th ACM Conference on Information and Knowledge Management, Pages 785-794.

Li Zheng, Chao Shen, Liang Tang, Tao Li, Steve Luis, Shu-Ching Chen, (2011). *Applying Data Mining Techniques to Address Disaster Information Management Challenges on Mobile Devices*. In Proceedings of the 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Pages 283-291.

Liang Tang, Tao Li, (2010). *LogTree: A Framework for Generating System Events from Raw Textual Logs*. In Processdings of the 10th IEEE International Conference on Data Mining, Pages 491-500.

Li Zheng, Chao Shen, Liang Tang, Tao Li, Steve Luis, Shu-Ching Chen, Vagelis Hristidis, (2010). *Using Data Mining Techniques to Address Critical Information Exchange Needs in Disaster Affected Public-Private Networks*. In Proceedings of the 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Pages 125-134.