

# LogSig: Generating System Events from Raw Textual Logs

Liang Tang  
School of Computer Science  
Florida International University  
11200 S.W. 8th Street  
Miami, Florida, 33199  
U.S.A  
ltang002@cs.fiu.edu

Tao Li  
School of Computer Science  
Florida International University  
11200 S.W. 8th Street  
Miami, Florida, 33199  
U.S.A  
taoli@cs.fiu.edu

Chang-Shing Perng  
IBM T.J Watson Research  
Center  
19 Skyline Drive  
Hawthorne, NY, 10532  
U.S.A  
perng@us.ibm.com

## ABSTRACT

Modern computing systems generate large amounts of log data. System administrators or domain experts utilize the log data to understand and optimize system behaviors. Most system logs are raw textual and unstructured. One main fundamental challenge in automated log analysis is the generation of system events from raw textual logs. Log messages are relatively short text messages but may have a large vocabulary, which often result in poor performance when applying traditional text clustering techniques to the log data. Other related methods have various limitations and only work well for some particular system logs. In this paper, we propose a message signature based algorithm **logSig** to generate system events from textual log messages. By searching the most representative message signatures, **logSig** categorizes log messages into a set of event types. **logSig** can handle various types of log data, and is able to incorporate human's domain knowledge to achieve a high performance. We conduct experiments on five real system log data. Experiments show that **logSig** outperforms other alternative algorithms in terms of the overall performance.

## Categories and Subject Descriptors

I.5.4 [Pattern Recognition]: Applications; H.4.m [Information Systems Applications]: Miscellaneous; G.2.3 [Discrete Mathematics]: Applications

## General Terms

Algorithms, Experimentation

## Keywords

Event generation, Message signature, System logs

## 1. INTRODUCTION

Modern computing systems generate large amounts of log data. The log data describes the status of each component

and records system internal operations, such as the starting and stopping of services, detection of network connections, software configuration modifications, and execution errors. System administrators or domain experts utilize the log data to understand and optimize system behaviors.

Most system logs are raw textual and unstructured. There are two challenges in analyzing system log data. The first challenge is transforming raw textual logs into system events. The number of distinct events observed can be very large and also grows rapidly due to the large vocabulary size as well as various parameters in log generation [6]. Once raw textual logs are transformed into events, the second challenge is to develop efficient algorithms to analyze or summarize the patterns from events. A lot of studies investigate the second challenge and develop many algorithms to mine system events [22] [30] [13] [15] [10] [20] [29] [14]. In this paper, we focus on the first challenge. The traditional solution to the first challenge 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, if 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.

Recent studies [6] [18] [26] 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 [24], 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 [6] [18] [26]. 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. To address these limitations, this paper proposes a message signature based algorithm **logSig** to generate system events from raw textual logs. It can handle various types of log data, and is able to incorporate human's domain knowledge to achieve a high performance.

### 1.1 Motivation

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.  
Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...\$10.00.

messages of the Hadoop system [3] 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 [6] would mismatch some terms. Another method IPLoM proposed in [18] 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 we aim to address in this paper.

## 1.2 Contributions

In this paper, we first describe the drawbacks of traditional text clustering techniques for event generation from log messages. We show that, it is difficult for the *bag-of-word* model to accurately partition log messages. We also analyze that, the string kernel based approach would be inefficient when the log vocabulary size is large. In addition, we discuss some limitations of related approaches proposed in previous literatures.

Then, we propose `logSig` algorithm to generate system events from textual log messages. `logSig` algorithm tries to find  $k$  message signatures to match all given messages as much as possible, where  $k$  is specified by the user. We con-

duct experiments on five real system log data. Experiments show that `logSig` outperforms other alternative algorithms in terms of the overall performance.

The rest of the paper is organized as follows: Section 2 describes the related work about system event generation from textual logs. Then, we formulate the problem of the system events generation in Section 3. In Section 4, we present an overview of the `logSig` algorithm. Section 5 discusses some detailed implementation issues. Section 6 proposes two approaches for incorporating the domain knowledge to improve the accuracy of the `logSig` algorithm. In Section 7, we present the experimental studies on five real system log data. Finally, Section 8 concludes our paper and discusses the future work.

## 2. RELATED WORK

It has been shown in [24] 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. The string kernel can be used to extract deep semantic information (e.g., the order of terms) to improve the performance of the clustering algorithm. It maps a string to a high dimensional vector to represent all possible term orders. However, because of the large vocabulary size, the dimensionality of the transformed space would be very high. Although the kernel trick does not have to explicitly create those high dimensional vectors, clustering algorithms would still be influenced by the high dimensionality due to the *Curse of Dimensionality* [25].

The related work about the log data analysis can be broadly summarized into two categories. One category is on system event generation from raw log data [6] [9] [18] [26] and the other category is on analyzing patterns from system events [22] [30] [13] [15] [10] [29] [14]. Our work in this paper belongs to the first category. A word matching similarity measurement is introduced in [6] for clustering the log messages. One problem is that, if most terms of a log message are parameter terms, then this type of log messages may not have many matched common words. This method is denoted as `StringMatch` in this paper. [18] develops a 4-steps partitioning method IPLoM for clustering the log messages based on inherent characteristics of the log format. However, the method can only be useful for strictly formatted log data. The `logSig` algorithm proposed in this paper can handle various types of log data without much prior knowledge about the log format.

## 3. PROBLEM FORMULATION

The goal of this paper 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.

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

Given a sequence  $X = w_1w_2\dots w_n$  and a term  $w_i$ ,  $w_i \in X$

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

**DEFINITION 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 1.** 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

**Table 1: Example of Match Score**

$X$	a	b	c	d	e	f
$S$	<u>a</u>	<u>x</u>	<u>c</u>		<u>e</u>	<u>y</u>

by underline words in Table 1. “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 [7] [19], such as RNA sequences.

### 3.1 Problem Statement

If all message signatures  $S_1, S_2, \dots, S_k$  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  $\mathcal{S} = \{S_1, \dots, S_k\}$  and a  $k$ -partition  $C_1, \dots, C_k$  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 [11], which plots clustering results with  $k$ . We can also display generated message signatures for  $k = 2, 3, \dots$  until the results can be approved by experts.

#### 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 question: *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 2.** 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.

### 3.2 Problem Analysis

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 [17].

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

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

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

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  $X_1, \dots, X_N$  must be an optimal solution  $S_1^*$ . This can be proved by contradiction as follows. Let  $S_{lcs}$  be the longest common subsequence of  $X_1, \dots, X_N$ . 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  $X_1, \dots, X_N$ . Moreover,  $Y_1, \dots, Y_N$  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  $w_i \in S_{lcs}$ ,  $X_1, \dots, X_N$  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  $w_i$ :  $Y_1, \dots, Y_N$ . 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  $X_1, \dots, X_N$  can be solved in polynomial time as well. However, *Multiple Longest Common Subsequence* problem is an NP-hard problem [17].

LEMMA 2. 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  $\mathcal{C} = \{C_1, \dots, C_k\}$ , then the optimal solution for  $k = n + 1$  and  $\mathcal{D} \cup \{Y\}$  should be  $\mathcal{C}' = \{C_1, \dots, C_k, \{Y\}\}$ . 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.

## 4. ALGORITHM OVERVIEW

In this section, we first present an approximated version of Problem 1 and then present our **logSig** algorithm. **logSig** algorithm consists of three steps. 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.

### 4.1 Term Pair Generation

The first step of **logSig** algorithm is converting each log message into a set of term pairs. For example, there is a log message collected from FileZilla [2] client:

```
2010-05-02 11:34:06 Command: mkdir ".indexes"
```

We extract each pairwise of terms and preserve the order of two terms. Then, the converted pairs are as follows:

```
{2010-05-02,11:34:06}, {2010-05-02,Command:},
{2010-05-02, mkdir}, {2010-05-02, ".indexes" }
{11:34:06,Command:}, {11:34:06,mkdir},
{11:34:06, ".indexes"}, {Command:,mkdir},
{Command:,".indexes"}, {mkdir,".indexes"}
```

The converted term pairs preserve the order information of message terms. On the other hand, the computation on the discrete term pairs is easier than a sequence. A similar idea was proposed in string kernel [16] for text classification. Their output is a high dimensional vector, and our output is a set of pairs.

### 4.2 Log Messages Partition

The second step is to partition log messages into  $k$  groups based on converted term pairs.

#### 4.2.1 The Approximated Version of Problem 1

**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  $\mathcal{C} = \{C_1, \dots, C_k\}$  of  $\mathcal{D}$  to maximize objective function  $F(\mathcal{C}, \mathcal{D})$ :

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

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 4 shows the average lower bound for this approximation.

LEMMA 3. 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 4. Given a set of log messages  $\mathcal{D}$  and a  $k$ -partition  $\mathcal{C} = \{C_1, \dots, C_k\}$  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, 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 4 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.

#### 4.2.2 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 3. 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 2. The optimal 2-partition is  $C_1$  has 100 “ab” and  $C_2$

Table 2: Example of two message groups

	group	
term pair	$C_1$	$C_2$
“ab”	50	50
“cd”	50	50

has 100 “cd”, or in 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 2.** 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.** Given a  $k$ -partition  $\mathcal{C} = \{C_1, \dots, C_k\}$  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  $C_i$ ,  $i = 1, \dots, k$ .

#### Connection between $\Phi$ and $F$ :

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

**LEMMA 5.** Given a set of log messages  $\mathcal{D}$  and a  $k$ -partition  $\mathcal{C} = \{C_1, \dots, C_k\}$  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 2, we have  $\Phi(\mathcal{D}) \geq y \cdot |\mathcal{D}|/k$ .

Lemma 5 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  $\Phi$  to be at least  $y \cdot |\mathcal{D}|/k$ . As for the local search algorithm, we mentioned that  $\Phi$  is easier to optimize than  $F$ .

Let  $\Delta_{iX_j}\Phi(\mathcal{D})$  denote the increase of  $\Phi(\mathcal{D})$  by moving  $X \in \mathcal{D}$  from group  $C_i$  into group  $C_j$ ,  $i, j = 1, \dots, k$ ,  $i \neq j$ . Then, by Definition 3,

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

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  $\Delta_{iX_j}\Phi(\mathcal{D})$  to increase  $\Phi(\mathcal{D})$  until no more update operation can be done.

---

#### Algorithm 1 **logSig\_localsearch** ( $\mathcal{D}, k$ )

---

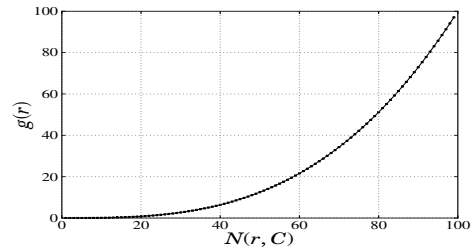
**Parameter:**  $\mathcal{D}$  : log messages set;  $k$ : the number of groups to partition;  
**Result:**  $\mathcal{C}$  : log message partition;

- 1:  $\mathcal{C} \leftarrow \text{RandomSeeds}(k)$
- 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**  $X_j \in \mathcal{D}$  **do**
- 12:      $i \leftarrow G[X_j]$
- 13:      $j^* = \arg \max_{j=1, \dots, k} \Delta_{iX_j}\Phi(\mathcal{D})$
- 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**  $\mathcal{C}$

---

#### 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 1 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



**Figure 1: Function  $g(r)$ ,  $|C| = 100$**

to 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.

In the experimental section, we will empirically compare the effectiveness of our proposed potential function  $\Phi$  with the objective function function  $F$  in the local search.

### 4.3 Message Signature Construction

The final step of **logSig** algorithm is to construct the message signature for each message group. Recall that a message signature is a sequence of terms that has a high match score to every message in the corresponding group. So it could be constructed by highly frequent term pairs identified in the second step.

**LEMMA 6.** Let  $S$  be an optimal message signature for a message group  $C$ , the occurrence number of every term  $w_j \in S$  must be equal or greater than  $\lfloor |C|/2 \rfloor$ .

The proof of Lemma 6 is similar to the proof of Lemma 1. If there exists a term  $w'_j \in S$  only appearing in less than  $\lfloor |C|/2 \rfloor$  messages, we can have:  $J(\{S - \{w'_j\}\}, C) > J(\{S\}, C)$ , then  $S$  is worse than  $S - \{w'_j\}$ . Thus,  $S$  is not optimal.

Lemma 6 indicates that we only need to care about those terms which appear in at least one half of the messages in a group. By scanning every message in a group, we could obtain the optimal sequence of those terms. Since log messages are usually very short, there are only a few term whose occurrence number is equal or greater than  $\lfloor |C|/2 \rfloor$ . Therefore, there are not many candidate sequences generated by those terms. We enumerate each one of them and select the best one to be the message signature.

## 5. ALGORITHM IMPLEMENTATION

In this section, we discuss some detailed issues about the implementation of the `logSig` algorithm.

### Term Pair Histogram

To efficiently compute the potential value of each message group  $C_i \in \mathcal{C}$ , a histogram is maintained for each group. The histogram is implemented by a hash table, whose key is a term pair and the value is the number of messages containing that term pair in the group. Then, for a term pair  $r$ ,  $c(r, C_i)$  and  $p(r, C_i)$  can be obtained in a constant time complexity.

### Approximately Computing $\Delta\Phi(\mathcal{D})$

The straightforward way to compute  $\Delta_{iX_j}\Phi(\mathcal{D})$  is enumerating all term pairs in  $C_i$  and  $C_j$ . The computation cost of finding the maximum  $\Delta_{iX_j}\Phi(\mathcal{D})$  is  $O(\sum_{l=1}^k |R(C_l)|)$ , which is the total number of term pairs in the entire data set. In log data, this number is even tens of times greater than  $|\mathcal{D}|$ . Hence, this computation cost is not affordable for each single log message.

Our approximated solution is to consider the change of  $N(r, C)$ , for each  $r \in R(X)$ . The reason is that,  $|C|$  is large. The impact to  $|C|$  by inserting or removing one message could be ignored comparing to the impact to  $N(r, C)$ .  $|C| \approx |C| + 1 \approx |C| - 1$ . So  $|C|$  can be treated as a constant in one exchange operation in *local search*. Then,

$$\frac{\partial\phi(C)}{\partial N(r, C)} = \frac{\partial[|N(r, C)|^3 / |C|^2]}{\partial N(r, C)} = \frac{3|N(r, C)|^2}{|C|^2} = 3[p(r, C)]^2.$$

$$\Delta_{iX_j}\Phi(\mathcal{D}) \approx 3 \cdot \sum_{r \in R(X)} [p(r, C_j)]^2 - [p(r, C_i)]^2.$$

By utilizing the histograms,  $p(r, C_j)$  and  $p(r, C_i)$  can be obtained in a constant time. Hence, for each log message  $X \in \mathcal{D}$ , the time complexity of finding the largest  $\Delta_{iX_j}\Phi(\mathcal{D})$  is at most  $O(k \cdot |R(X)|)$ . Although  $\Delta_{iX_j}\Phi(\mathcal{D})$  is would underestimate the actual change of  $\Phi(\mathcal{D})$ , it can save a lot of computation cost.

## 5.1 Algorithm Analysis

### Convergence and Local Optima

In the local search algorithm, exchanging messages' groups only increases the overall potential  $\Phi(\mathcal{D})$ . There is no operation to decrease the overall potential.  $\Phi(\mathcal{D})$  is not infinite, which is less than or equal to  $|R(\mathcal{D})| \cdot |\mathcal{D}|$ . Therefore, the local search algorithm would converge.

Similar to other local search based optimization algorithms, `logSig` may converge into a local optima as well. However,

`logSig`'s potential function provides a more reliable heuristic to guide the optimization process. It is much less likely to stop at a local optima.

### Time Complexity and Space Complexity

The time complexity of converting log messages into term pairs is  $O(|\mathcal{D}| \cdot L^2)$ , where  $L$  is the maximum length of log messages. For every  $X_j \in \mathcal{D}$ ,  $L^2 \geq |R(X_j)|$ . For the local search, each iteration goes through every message. So the time complexity of an iteration is  $O(k \cdot L^2 \cdot |\mathcal{D}|)$ . Let  $t$  be the number of iterations, so the time complexity of the local search is  $O(t \cdot k \cdot L^2 \cdot |\mathcal{D}|)$ . Our experiments shows that  $t$  is usually 3 to 12 for most system log data.

The space cost of the algorithm is mainly determined by the term pair histograms created for every message group. The total space complexity is the total number of term pairs,  $O(|R(\mathcal{D})|)$ .

## 6. INCORPORATING DOMAIN KNOWLEDGE

In real world applications, analyzing system behaviors mainly relies on the generated system events, so the event generation algorithm should be reliable. Current approaches for grouping log messages into different event types are totally unsupervised. In practice, for most sorts of logs (e.g., Hadoop system logs), there are some domain knowledge about a fixed catalog of Java exceptions. In addition, the log generation mechanisms implicitly create some associations between the terminologies and the situations. Incorporating domain knowledge into the clustering process could improve the performance of event generation. In this section, we present two approaches for incorporating domain knowledge to improve the accuracy of `logSig` algorithm.

### 6.1 Term Feature Layer

Some terms or words in log messages share some common features which can help our algorithm identify the log message type. The features are domain knowledge from human experts. For example, "2011-02-01" and "2011-01-02" are different terms, but they are both *dates*; "192.168.0.12" and "202.119.23.10" are different terms, but they are both *IP addresses*.

`logSig` algorithm allows users to provide a list of features. Each feature is described by a regular expression. An additional feature layer is built to incorporate those features for representing log messages. The feature layer is created by regular expression matching. For example, we have following regular expressions<sup>1</sup>:

```
Timestamp: \d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}
FilePath: "(\\w|\\)"
```

Then, we could scan the log message  $X_1$  to create the feature layer  $Y_1$  as shown in Table 3. The feature layer  $Y_1$  contains

Table 3: Example of Feature Layer

$X_1$	2010-05-02	11:34:06	Command:	mkdir	".indexes"
$Y_1$	Timestamp		Command:	mkdir	FilePath

more semantic information. It can gather similar terms and reduce noisy terms. In addition, regular expression is easy for domain experts to write. Using a sophisticated regular language toolkit, a feature layer can be built efficiently.

<sup>1</sup>The grammar of regular expressions is defined by Java Regular Library

## 6.2 Constraints on Message Signature

In reality, domain experts usually identify the event type of a log message by scanning keywords or phrases. For example, as for SFTP/FTP logs, they would be sensitive to those phrases: “Command”, “No such file”, “Error”, “Transfer failed” and so on. Those sensitive phrases should be included in the generated message signature to express system events. On the other hand, domain experts also know that some trivial phrases should be ignored, such as the message IDs and the timestamps. Those trivial phrases should not be included in any message signature. To sum up, those knowledge can be transferred as constraints on message signatures. Those constraints can help the event generation algorithm to improve its accuracy. Unlike traditional constraints in semi-supervised clustering, such as *Must-Link* or *Cannot-Link* [28], our constraints are placed in the subsequences (or phrases) of messages, not on the messages themselves.

### Constraint-based logSig Algorithm

To incorporate constraints on message phrases, Problem 1 can be revised as follows:

**PROBLEM 3.** *Given a set of log messages  $\mathcal{D}$ , a set of sensitive phrases  $\mathcal{P}_S$  and trivial phrases  $\mathcal{P}_T$ , find  $k$  message signatures  $\mathcal{S} = \{S_1, \dots, S_k\}$  with a  $k$ -partition  $C_1, \dots, C_k$  of  $\mathcal{D}$  to maximize:*

$$J'(\mathcal{S}, \mathcal{D}, \mathcal{P}_S, \mathcal{P}_T) = \lambda \sum_{i=1}^k \sum_{X_j \in C_i} \text{match}(X_j, S_i) + (1 - \lambda) \sum_{i=1}^k \left[ \sum_{P_s \in \mathcal{P}_S} \text{match}(P_s, S_i) - \sum_{P_t \in \mathcal{P}_T} \text{match}(P_t, S_i) \right],$$

where  $\lambda$  is a user-defined parameter between 0 and 1.

The revised optimization problem can be solved by constraint-based logSig Algorithm. The basic idea of constraint-based logSig is to increase the weight of term pairs in  $R(\mathcal{P}_S)$  and decrease the weight of term pairs in  $R(\mathcal{P}_T)$ . Let  $w_r$  denote the weight of pair  $r$ . The only revised part in logSig algorithm is  $\Delta_{iX_j} \Phi(\mathcal{D})$ . By multiplying the weight  $w_r$ , it becomes:

$$\Delta_{iX_j} \Phi(\mathcal{D}) \approx 3 \sum_{r \in R(X)} \left[ [p(r, C_j)]^2 - [p(r, C_i)]^2 \right] \cdot w_r,$$

and

$$w_r = \begin{cases} 1.0, & r \notin R(\mathcal{P}_S), r \notin R(\mathcal{P}_T) \\ \lambda', & r \in R(\mathcal{P}_S), r \notin R(\mathcal{P}_T) \\ 1/\lambda', & r \notin R(\mathcal{P}_S), r \in R(\mathcal{P}_T) \end{cases},$$

where  $\lambda' \geq 1$  is a user-defined parameter that can be approximately derived from the original parameter  $\lambda$ .  $\lambda'$  is utilized to increase (and decrease) the importance of term pairs in sensitive phrases (and trivial phrases). The choice of  $\lambda'$  depends on users' confidence in those sensitive phrases and trivial phrases. In experiments, we let  $\lambda' = 10.0$ . We show the performances of the algorithm when varying  $\lambda'$  from 0.0 to 20.0 and explain why we choose 10 in the experimental section.

## 7. EVALUATION

### 7.1 Experimental Platforms

We implement our algorithm and other comparative algorithms in Java 1.6 platform. Table 5 summarizes our experimental environment.

Table 5: Experimental Machine

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

### 7.2 Data Collection

We collect log data from 5 different real systems, which are summarized in Table 6. Logs of FileZilla [2], PVFS2 [4] Apache [1] and Hadoop [3] are collected from the server machines/systems in the computer lab of a research center. Log data of ThunderBird [5] is collected from a supercomputer in Sandia National Lab. The true categories of log messages are obtained by specialized log parsers. For instance, FileZilla'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 6: 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 2 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.

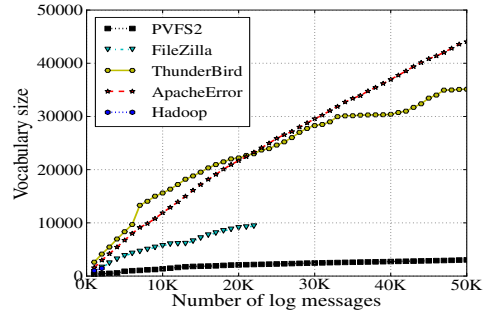


Figure 2: Vocabulary size

### 7.3 Comparative Algorithms

We compare our algorithm with 7 alternative algorithms in this experiment. Those algorithms are described in Table 7. 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 [18] and StringMatch [6] are two methods proposed in recent related literatures. VectorModel [23], Jaccard [25], StringKernel [16] are traditional methods for text clustering. VectorModel and semi-StringKernel are implemented by  $k$ -means clustering algorithm [25]. Jaccard and StringMatch are implemented by  $k$ -medoid algorithm [12], 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 [23] to reduce the computation and space costs.

Table 4: Average F-Measure Comparison

Algorithm	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	<b>0.8881</b>	0.7409	0.2015
StringMatch	0.5639	0.4774	0.6663	0.7932	0.4840
StringKernel <sub>0.8</sub>	0.4462	0.3894	0.6416	0.8810	0.3103
StringKernel <sub>0.5</sub>	0.4716	0.4345	0.7361	<b>0.9616</b>	0.3963
StringKernel <sub>0.3</sub>	0.4139	0.6189	0.8321	0.9291	0.4256
logSig	<b>0.6949</b>	<b>0.7179</b>	0.7882	0.9521	<b>0.7658</b>
semi-Jaccard	0.8283	0.4017	0.7222	0.7415	0.4997
semi-StringKernel <sub>0.8</sub>	0.8951	0.6471	0.7657	0.8645	0.7162
semi-StringKernel <sub>0.5</sub>	0.7920	0.4245	0.7466	<b>0.8991</b>	0.7461
semi-StringKernel <sub>0.3</sub>	0.8325	0.7925	0.7113	0.8537	0.6259
semi-logSig	<b>1.0000</b>	<b>0.8009</b>	<b>0.8547</b>	0.7707	<b>0.9531</b>

Table 7: Summary of comparative algorithms

Algorithm	Description
VectorModel	Vector space model proposed in information retrieval
Jaccard	Jaccard similarity based $k$ -medoid algorithm
StringKernel	String kernel based $k$ -means algorithm
IPLoM	Iterative partition method proposed in [18]
StringMatch	String matching method proposed in [6]
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

semi-logSig, semi-StringKernel and semi-Jaccard are semi-supervised versions of logSig, StringKernel and Jaccard respectively. To make a fair comparison, 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 [8]:

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

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

Jaccard, StringMatch and semi-Jaccard algorithms apply classic  $k$ -medoid algorithm for message clustering. The time complexity of  $k$ -medoid algorithm is very high:  $O(tn^2)$  [27], 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

Table 8: 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

$\mathcal{P}_T$  are summarized in Table 8. In Section 7.5, larger logs are used to test the scalability.

## 7.4 Quality of Generated Events

### 7.4.1 Average F-Measure

Table 4 shows the accuracy comparison of generated system events by different algorithms. The accuracy is evaluated by F-measure (F1 score) [23], 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 4. 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* [25],  $k$ -means based StringKernel is not easy to converge in a high dimensional space. Figure 2 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.

### 7.4.2 Message Signatures

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

Table 9: Message Signatures

System Log	Message Signature	Associated Category
FileZilla	<i>Date Hours Number Number</i> Status: ...	Status
	<i>Date Hours Number Number</i> Response: <i>Number</i>	Response
	<i>Date Hours Number Number</i> Command:	Command
	<i>Date Hours Number Number</i> Error: File transfer failed	Error
Apache Error	<i>Timestamp</i> ( 13 ) Permission denied: /home/bear-005/users/xxx/public_html/ke/.htaccess	Permission denied
	<i>pcfg_openfile</i> : unable to check htaccess file ensure it is readable	
	<i>Timestamp Error</i> [ client ] File does not exist: /opt/website/sites/users.cs.fiu.edu/data/favicon.ico	File does not exist
	<i>Timestamp Error</i> [ client 66.249.65.4 ] suexec <i>policy</i> violation: see suexec log for more details	Policy violation
	<i>Timestamp</i> /home/hpdrcc-demo/sdbtools/public_html/hpdrcc2/.htaccess: AuthName takes one argument The <i>Authentication</i> realm ( e.g. "Members Only" )	Authentication
	<i>Timestamp Error</i> [ client ] 2010-04-01 using	N/A
	<i>Timestamp Error</i> [ client ]	N/A

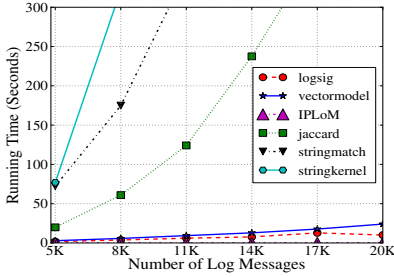


Figure 3: Average Running Time for FileZilla logs

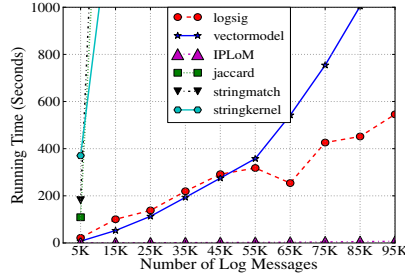


Figure 4: Average Running Time for ThunderBird logs

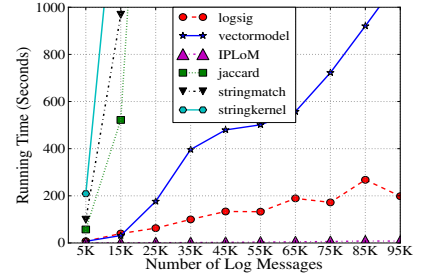


Figure 5: Average Running Time for Apache logs

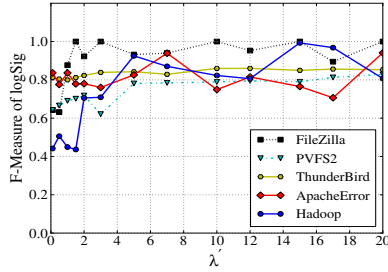


Figure 6: Varying parameter  $\lambda'$

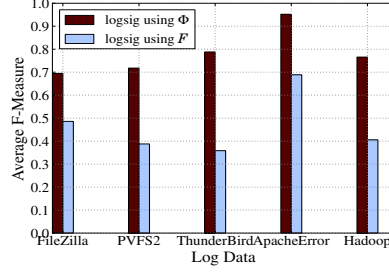


Figure 7: Effectiveness of Potential Function

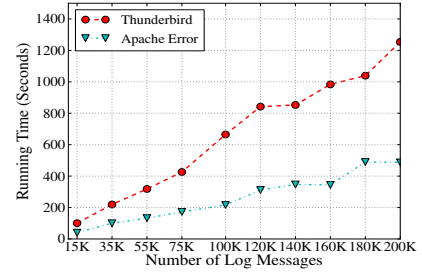


Figure 8: Scalability of logSig

message signatures. Table 9 shows generated signatures of FileZilla and Apache Error by semi-logSig, in which features are indicated by *italic words*.

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 of Apache Error logs. As a result, their ‘‘Associated Category’’ in Table 9 are ‘‘N/A’’. Therefore, the overall F-measure on Apache error log in Table 4 is only 0.7707.

### 7.4.3 Parameter Setting

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  $\lambda$ , which is the decay factor of a pair of terms. We use `StringKernel $\lambda$`  to denote the string kernel method using decay factor  $\lambda$ . In our experiments, we set up

string kernel algorithms using three different decay factors: `StringKernel $_{0.8}$` , `StringKernel $_{0.5}$`  and `StringKernel $_{0.3}$` .

As for the parameter  $\lambda'$  of our algorithm logSig, we set  $\lambda' = 10$  based on the experimental result shown by Figure 6. For each value of  $\lambda'$ , 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  $\lambda'$  is greater than 4.

### 7.4.4 Effectiveness of Potential Function

To evaluate the effectiveness of the potential function  $\Phi$ , we compare our proposed logSig algorithm with another logSig algorithm which uses the objective function  $F$  to guide its local search. Figure 7 shows the average F-measures of the two algorithms on each data set. Clearly, our proposed potential function  $\Phi$  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. The reason for this has been discussed in Section 4.2.2.

## 7.5 Scalability

Scalability is an important factor for log analysis algorithms. Many high performance computing systems generate more than 1Mbytes log messages per second [21]. Figure 3, Figure 4 and Figure 5 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 out the low speed convergence is mainly due to the high dimensionality.

Figure 8 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.

## 8. CONCLUSION AND FUTURE WORK

In this paper, we show the drawbacks of traditional methods and previous log message analyzing methods. To address the limitations of existing methods, we propose **logSig**, a message signature based method for events creation from system log messages. **logSig** utilizes the common subsequence information of messages to partition and describe the events generated from log messages.

As for the future work, we will integrate the structural learning techniques into our framework to capture the structural information of log messages. We hope those structures could improve the performance of the **logSig** algorithm.

## Acknowledgement

The work is supported in part by NSF grants IIS-0546280 and HRD-0833093.

## 9. REFERENCES

- [1] Apache HTTP Server : An Open-Source HTTP Web Server. <http://httpd.apache.org/>.
- [2] FileZilla: An open-source and free FTP/SFTP solution. <http://filezilla-project.org>.
- [3] Hadoop : An Open-Source MapReduce computing platform. <http://hadoop.apache.org/>.
- [4] PVFS2 : The state-of-the-art parallel I/O and high performance virtual file system. <http://pvfs.org>.
- [5] ThunderBird: A supercomputer in Sandia National Laboratories. <http://www.cs.sandia.gov/~jrstear/logs/>.
- [6] M. Aharon, G. Barash, I. Cohen, and E. 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.
- [7] S. Bereg, M. Kubica, T. Walen, and B. Zhu. RNA multiple structural alignment with longest common subsequences. *Journal of Combinatorial Optimization*, 13(2):179–188, 2007.
- [8] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of ICML*, Alberta, Canada, July 2004.
- [9] K. Fisher, D. Walker, and K. Q. Zhu. Incremental learning of system log formats. *SIGOPS Oper. Syst. Rev.*, 44(1):85–90, 2010.
- [10] J. Gao, G. Jiang, H. Chen, and J. Han. Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems. In *Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS'09)*, pages 623–630, 2009.
- [11] G. Hamerly and C. Elkan. Learning the  $k$  in  $k$ -means. In *Proceedings of NIPS*, Vancouver, British Columbia, Canada, December 2003.
- [12] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*, 2ed. Morgan Kaufmann, 2005.
- [13] J. L. Hellerstein, S. Ma, and C.-S. Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 43(3):475–493, 2002.
- [14] J. Kiernan and E. Terzi. Constructing comprehensive summaries of large event sequences. In *Proceedings of ACM KDD*, pages 417–425, Las Vegas, Nevada, USA, August 2008.
- [15] T. Li, F. Liang, S. Ma, and W. Peng. An integrated framework on mining logs files for computing system management. In *Proceedings of ACM KDD*, pages 776–781, Chicago, Illinois, USA, August 2005.
- [16] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444, March 2002.
- [17] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, April 1978.
- [18] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of ACM KDD*, pages 1255–1264, Paris, France, June 2009.
- [19] K. Ning, H. K. Ng, and H. W. Leong. Finding patterns in biological sequences by longest common subsequences and shortest common supersequences. In *Proceedings of BIBE*, pages 53–60, Arlington, Virginia, USA, 2006.
- [20] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *Proceedings of ICDM*, pages 959–964, Pisa, Italy, December 2008.
- [21] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proceedings of DSN 2007*, pages 575–584, Edinburgh, UK, June 2007.
- [22] W. Peng, C. Perng, T. Li, and H. Wang. Event summarization for system management. In *Proceedings of ACM KDD*, pages 1028–1032, San Jose, California, USA, August 2007.
- [23] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1984.
- [24] J. Stearley. Towards informatic analysis of syslogs. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 309–318, San Diego, California, USA, September 2004.
- [25] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.
- [26] L. Tang and T. Li. LogTree: A framework for generating system events from raw textual logs. In *Proceedings of ICDM*, pages 491–500, Sydney, Australia, December 2010.
- [27] S. Theodoridis and r. e. Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, 2006.
- [28] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained  $k$ -means clustering with background knowledge. In *Proceedings of ICML*, pages 577–584, June 2001.
- [29] P. Wang, H. Wang, M. Liu, and W. Wang. An algorithmic approach to event summarization. In *Proceedings of ACM SIGMOD*, pages 183–194, Indianapolis, Indiana, USA, June 2010.
- [30] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Mining console logs for large-scale system problem detection. In *SysML*, San Diego, CA, USA, December 2008.