A Lightweight Algorithm for Message Type Extraction in Event Logs *

Adetokunbo Makanju, A. Nur Zincir-Heywood, Evangelos E. Milios
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia
B3H 1W5
Canada
{makanju, zincir, eem}@cs.dal.ca

June 26, 2009

Abstract

Message type or message cluster extraction is an important task in automatic application log analysis. When the message types that exist in a log file are defined, they form the basis for carrying out other automatic application log analysis tasks. In this paper we introduce a novel algorithm for carrying out this task. IPLoM, which stands for *Iterative Partitioning Log Mining*, works through a 4-step process. The first 3 steps hierarchically partition the event log into groups of event log messages or event clusters. In its 4th and final stage IPLoM produces a message type description or line format for each of the message clusters. IPLoM is able to find clusters in data irrespective of the frequency of its instances in the data, it scales gracefully in face of long message type patterns and produces message type descriptions at a level of abstraction which is preferred by a human observer. Evaluations show that IPLoM outperforms similar algorithms statistically significantly.

1 Introduction

The goal of autonomic computing as espoused by IBM's senior vice president of research, Paul Horn in March 2001 can be defined as the goal of building self-managing computing systems [1]. The four key concepts of self-management in autonomic computing are self-configuration, self-optimization, self-healing and

*Preliminary results and a summary description of the algorithms in this paper were presented in *A. Makanju et. al.,* **Clustering Event Logs Using Iterative Partitioning**. *Published in the Proceedings of the 15th ACM Conference on Knowledge Discovery and Data Mining. June 28th - July 1st 2009. Paris, France.*

self-protection. Given the increasing complexity of computing infrastructure which is stretching the human capability to manage it to its limits, the goal of autonomic computing is a desirable one. However, it is a long term goal which must first start with the building of computing systems which can automatically gather and analyze information about their states to support decisions made by human administrators [1].

Event logs generated by applications that run on a system consist of several independent lines of text data, which contain information that pertains to events that occur within a system. This makes them an important source of information to system administrators in fault management which is a cornerstone for self-healing, and intrusion detection and prevention, which is an important cornerstone for self-protection. Therefore as we move toward the goal of building systems that are capable of self-healing and self-protection, an important step would be to build systems that are capable of automatically analyzing the contents of their log files to provide useful information to the system administrators.

A basic task in automatic analysis of log files is message type or event cluster extraction [2, 3, 4, 5, 6]. Extraction of message types makes it possible to abstract the contents of event logs and facilitates further analysis and the building of computational models. Message type descriptions are the templates on which the individual messages in any event log are built. For example, this line of code:
sprintf(message, Connection from %s port %d, ipaddress, portnumber);
in a *C* program could produce the following log entries:
"Connection from 192.168.10.6 port 25"
"Connection from 192.168.10.6 port 80"
"Connection from 192.168.10.7 port 25"
"Connection from 192.168.10.8 port 21".

These four log entries would form a cluster or event type in the event log and can be represented by the message type description (or line format):
*"Connection from * port *".*
The wild-cards "*" represent message variables. We will adopt this cluster representation in the rest of our work. The goal of message type mining is to find the message type representations of the message clusters that exist in a log file. So far techniques for automatically mining these line patterns from event logs have been based on the Apriori algorithm [7] for frequent itemsets from data, e.g. SLCT (Simple Log File Clustering Tool) [8] and Loghound [9], or other line pattern discovery techniques like Teiresias [10] designed for other domains [5].

In this paper we introduce IPLoM (*Iterative Partitioning Log Mining*), a novel algorithm for the mining of event type patterns from event logs. IPLoM works through a 3-Step partitioning process, which partitions a log file into its respective clusters. In a fourth and final stage the algorithm produces a cluster description for each leaf partition of the log file. These cluster descriptions then become event type patterns discovered by the algorithm. IPLoM is able to find clusters in data irrespective of the frequency of its instances in the data, it scales

gracefully in face of long message type patterns and it produces message type descriptions at a level of abstraction which is preferred by a human observer. In our experiments we compared the outputs of IPLoM, SLCT, Loghound and Teiresias on 7 different event log files, against message types produced manually on the event log files by our faculty's tech support group. Results demonstrate that IPLoM consistently outperforms the other algorithms on similar tasks.

The rest of this paper is organized as follows: section 2 discusses previous work in event type pattern mining and categorization. Section 3 outlines the proposed algorithm and the methodology to evaluate its performance. Section 4 describes the results whereas section 5 presents the conclusion and the future work.

2 Background and Previous Work

2.1 Definitions

We begin this section by first defining some of the terminology used in this paper.

- **Event Log:** A text based audit trail of events that occur within the system or application processes on a computer system. The lines in Fig. 1 shows examples of the contents of an event log.
- **Event:** An independent line of text within an event log which details a single occurrence on the system. An event typically contains not only a *message* but other other fields of information like a *Date*, *Source* and *Tag* e.g as defined in the `syslog` RFC (Request for Comment) [11]. For message type extraction we are only interested in the *message* field of the event. This is why events are sometimes referred to in literature as messages or transactions. In Fig. 1 each individual line represents an event. The first five fields (delimited by whitespace) in each line represent the *Date*, *Source*, *Message Type*, *Facility* and *Severity* of each event. The remainder parts of the event represent the free-form *message* that we are interested in.
- **Token:** A single word delimited by white space within the *message* field of an event. For example in the first line of Fig. 1, the words *instruction*, *cache*, *parity*, *error* and *corrected* are tokens in that message.
- **Event Size:** The number of individual tokens in the message field of an event. The first line of Fig. 1 has an event size of 5, while the third line has an event size of 2.
- **Event Cluster/Message Type:** These are *message* fields of entries within an event log produced by the same print statement. Consecutive pairs of lines in Fig. 1 belong to the same event cluster.

```
2005-06-03-15.42.50.823719 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-03-15.42.50.982731 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-06-22.41.37.357738 R20-M0-NA-C:J15-U11 RAS KERNEL INFO generating core.3740
2005-06-06-22.41.37.392258 R20-M0-NA-C:J17-U11 RAS KERNEL INFO generating core.3612
2005-06-11-19.20.25.104537 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-06-11-19.20.25.393590 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-07-01-17.52.23.557949 R22-M0-NA-C:J05-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
2005-07-01-17.52.23.584839 R22-M0-NA-C:J03-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
```

Figure 1: An example application event log file.

- **Cluster Description/Message Type Description/Line Format:** A textual template containing wild-cards which represents all members of an event cluster. The message in the 3rd and 4th lines of Fig. 1 have a cluster description of *"generating *"*.
- **Constant Token:** A token within the *message* field of an event which is not represented by a wild-card value in its associated message type description. The token *generating* in the 3rd line of Fig. 1 is a constant token.
- **Variable Token:** A token within the *message* field of an event which is represented by a wild-card value in its associated message type description. The token *core.3740* in the 3rd line of Fig. 1 is a variable token.

2.2 Previous Work

Data clustering as a technique in data mining or machine learning is a process whereby entities are sorted into groups called clusters, where members of each cluster are similar to each other and dissimilar from members of other groups. Clustering can be useful in the interpretation and classification of large datasets, which may be overwhelming to analyze manually. Clustering therefore can be a useful first step in the automatic analysis of event logs.

If each textual line in an event log is considered a data point and its individual words considered attributes, then the clustering task reduces to one in which similar log messages are grouped together. For example the log entry *Command has completed successfully* can be considered a 4-dimensional data point with the following attributes *"Command", "has", "completed", "successfully"*. However, as stated in [8], traditional clustering algorithms are not suitable for event logs for the following reasons:

1. The event lines do not have a fixed number of attributes.
2. The data point attributes i.e. the individual words or tokens on each line, are categorical. Most conventional clustering algorithms are designed for numerical attributes.
3. Traditional clustering algorithms also tend to ignore the order of attributes. In event logs the attribute order is important.

While several algorithms like CLIQUE [12], CURE [13] and MAFIA [14] have been designed for clustering high dimensional data, these algorithms are

still not quite suitable for log files because an algorithm suitable for clustering event logs needs to not just be able to deal with high dimensional data, it also needs to be able to deal with data with different attribute types[8, 15].

For these reasons several algorithms and techniques for automatic clustering and/or categorization of log files have been developed. Moreover, some researchers have also attempted to use techniques designed for pattern discovery in other types of textual data to the task of clustering event logs.

In [16] the authors attempt to classify raw event logs into a set of categories based on the IBM CBE (Common Base Event) format [17] using Hidden Markov Models (HMM) and a modified Naive Bayesian Model. They report 85% and 82% classification accuracy respectively. While similar, the automatic categorization done in [16] is not the same as discovering event log clusters or formats. This is because the work done in [16] is a supervised classification problem, with predefined categories, while the problem we tackle is unsupervised, with the final categories not known apriori.

On the other hand SLCT [8] and Loghound [9] are two algorithms, which were designed specifically for automatically clustering log files, and discovering event formats. This is similar to our objective in this paper. Because both SLCT and Loghound are similar to the Apriori algorithm [7] they require the user to provide a support threshold value as input.

SLCT works through a three step process. The steps are described below

1. It firsts identifies the frequent words (words that occur more frequently than the support threshold value) or 1-itemsets from the data

2. It then extracts the combinations of these 1-itemsets that occur in each line in the data-set. These 1-itemset combinations are cluster candidates.

3. Finally, those cluster candidates that occur more frequently than the support value are then selected as the clusters in the data-set.

Loghound on the other hand discovers frequent patterns from event logs by utilizing a frequent itemset mining algorithm, which mirrors the Apriori algorithm more closely than SLCT because it works by finding itemsets which may contain more than *1* word up to a maximum value provided by the user. With both SLCT and Loghound, lines that do not match any of the frequent patterns discovered are classified as outliers.

SLCT and Loghound have received considerable attention and have been used in the implementation of the Sisyphus Log Data Mining toolkit [18], as part of the LogView log visualization tool [19] and in online failure prediction [20].

A comparison of SLCT against a bio-informatics pattern discovery algorithm developed by IBM called Teiresias [10] is carried out in [5]. Teiresias was designed to discover all patterns of at least a given specificity and support in categorical data. Teiresias can be described as an algorithm that takes a set of strings X and breaks them up into a set of unique characters C, which are the building blocks of the strings. It then proceeds to find all motifs (patterns)

having specificity at least L/W, where L is the number of non-wild-card characters from C and W is the width of the motif with wild-cards included. A support value K can also be provided i.e. Teiresias only finds motifs that occur at least K times in the set of strings X. While Teiresias was judged to work just as effectively as SLCT by the author, it was found not to scale efficiently to large data-sets.

In our work we introduce IPLoM, a novel log-clustering algorithm. IPLoM works differently from the other clustering algorithms described above as it is not based on the Apriori algorithm and does not explicitly try to find line formats. The algorithm works by creating a hierarchical partitioning of the log data. The leaf nodes of this hierarchical partitioning of the data are considered clusters of the log data and they are used to find the cluster descriptions or line formats that define each cluster. Our experiments demonstrate that IPLoM outperforms SLCT, Loghound and Teiresias when they are evaluated on the same data-sets.

3 Methodology

In this section we first give a detailed description of our proposed algorithm and our methodology for testing its performance against those of previous algorithms.

3.1 The IPLoM Algorithm

The IPLoM algorithm is designed as a log data clustering algorithm. It works by iteratively partitioning a set of log messages used as training exemplars. At each step of the partitioning process the resultant partitions come closer to containing only log messages which are produced by the same line format. At the end of the partitioning process the algorithm attempts to discover the line formats that produced the lines in each partition, these discovered partitions and line formats are the output of the algorithm.

An outline of the four steps of the algorithm is given in Fig. 2. The algorithm is designed to discover all possible line formats in the initial set of log messages and does require a support threshold like SLCT or Loghound. As it may be sometimes required to find only line formats that have a support that exceeds a certain threshold, a file prune function (Algorithm 1) is incorporated into the algorithm. By removing the partitions that fall below the threshold value at the end of each partitioning step, we are able to produce only line formats that meet the desired support threshold at the end of the algorithm. The use of the file prune function is however optional.

The following sub-sections describe each step of the algorithm in more detail.

3.2 Step 1: Partition by event size.

The first step of the partitioning process works on the assumption that log messages that have the same message type description are likely to have the

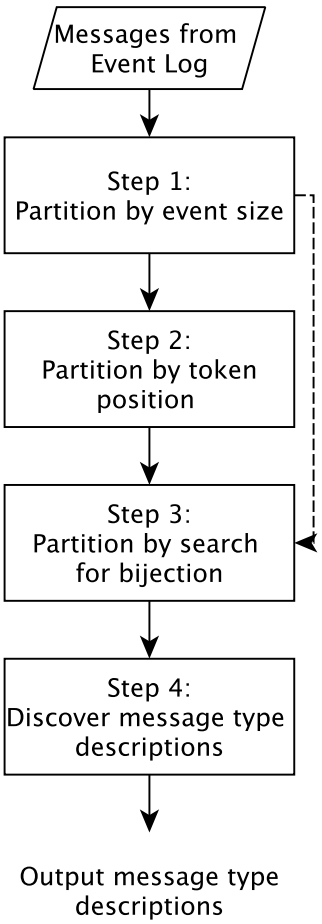


Figure 2: Overview of IPLoM processing steps.

same event size. For this reason IPLoM's first step (Fig. 3) uses the event size heuristic to partition the log messages. By partition we mean non-overlapping groupings of the messages. Additional heuristic criteria are used in the remaining steps to further divide the initial partitions. The partitioning process induces a hierarchy of maximum depth 4 on the messages and the number of nodes on each level is data dependent. Consider the cluster description *"Connection from *"*, which contains 3 tokens. It can be intuitively concluded that all the instances of this cluster e.g. *"Connection from 255.255.255.255"* and *"Connection from 0.0.0.0"* would also contain the same number of tokens. By partitioning our data first by event size we are taking advantage of the property of most cluster instances of having the same event size, therefore the resultant partitions of this heuristic are likely to contain the instances of the different clusters which have the same event size. A detailed description of this step of

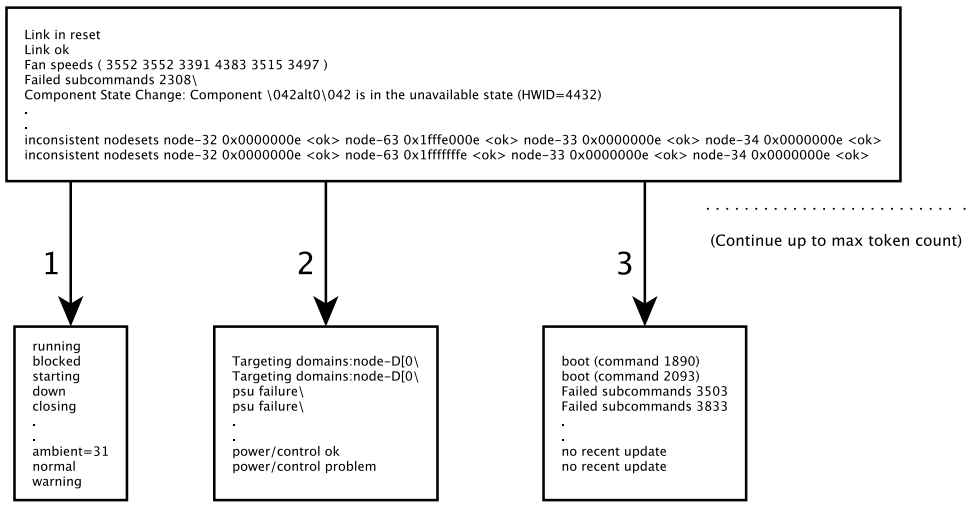


Figure 3: IPLoM Step-1: Partition by event size.

the algorithm is given in section A of the Appendix.

3.3 Step 2: Partition by token position.

At this point each partition of the log data contains log messages which are of the same size and can therefore be viewed as n-tuples, with n being the event size of the log messages in the partition. This step of the algorithm works on the assumption that the column with the least number of variables (unique words) is likely to contain words which are constant in that position of the message type descriptions that produced them. Our heuristic is therefore to find the token position with the least number of unique values and further split each partition using the unique values in this token position i.e. each resultant partition will contain only one of those unique values in the token position discovered, as can be seen in the example outlined in Fig. 4. A detailed description of this step of the partitioning process is outlined in Algorithm 2.

Despite the fact that we use the token position with the least number of unique tokens, it is still possible that some of the values in the token position might actually be variables in the original message type descriptions. While an error of this type may have little effect on Recall, it could adversely affect Precision. To mitigate the effects of this error a partition support threshold could be introduced. We group any partition which falls below the provided threshold into one partition (Algorithm 3). The intuition here is that a partition that is produced using an actual variable value may not have enough lines to exceed a certain percentage (the partition support threshold) of the log messages in the partition. It should be noted that this partition threshold is not necessary for

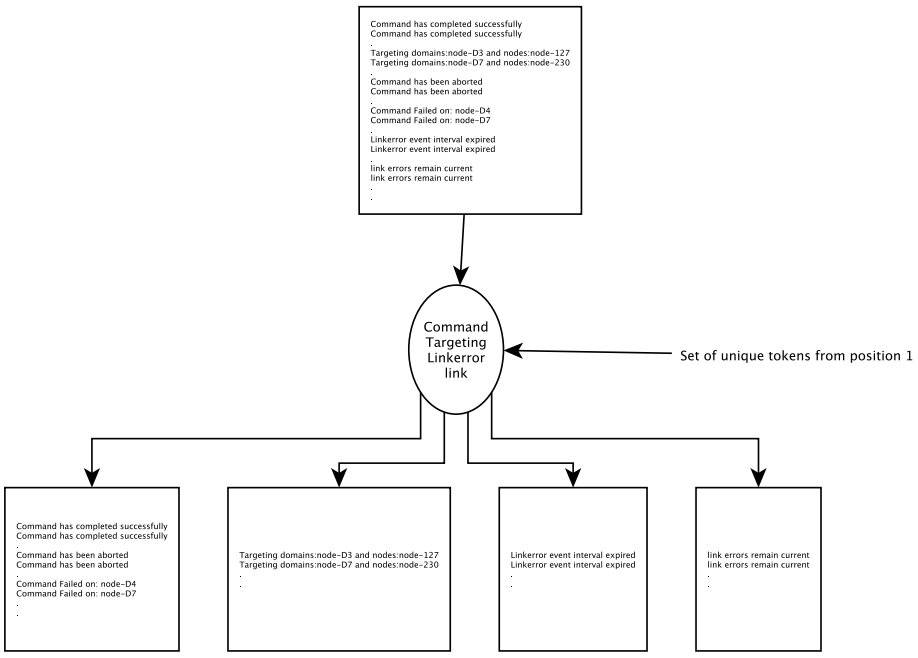


Figure 4: IPLoM Step-2: Partition by token position.

the algorithm to function and is only introduced to give the system administrators the flexibility to influence the partitioning based on expert knowledge they may have. This support threshold was not used in the experimental results presented here.

3.4 Step 3: Partition by search for bijection

In the third and final partitioning step we partition by searching for bijective relationships between the set of unique tokens in two token positions selected using a criterion as described in detail in Algorithm 4. A summary of the heuristic would be to select the first two token positions with the most frequently occurring event size value greater than 1. A bijective function is a *1-1* relation that is both injective and surjective. When a bijection exists between two elements in the sets of tokens, this usually implies that a strong relationship exists between them and log messages that have these token values in the corresponding token positions are separated into a new partition.

Sometimes the relations found are not *1-1* but *1-M*, *M-1* and *M-M*. In the example given in Fig. 5 the tokens *Failed* and *on:* have a 1-1 relationship because all lines that contain the token *Failed* in position 2 also contain the

Algorithm 1 File_Prune Function

Input: Collection $C[]$ of log file partitions.
Real number FS as file support threshold.
Output: Collection $C[]$ of log file partitions with support greater than FS.
for every *partition in* C **do**
  $Supp = \frac{\#LinesInPartition}{\#LinesInCollection}$
  if $Supp < FS$ **then**
    Delete partition from $C[]$
  end if
end for
Return(C)

Algorithm 2 IPLoM Step 2

Input: Collection C_In of log file partitions from Step 1.
Output: Collection C_Out of log file partitions derived from C_In.
for every *partition in* C_In **do** {Lines in each partition have same event size}
  Create temporary collection $Temp\_C$
  Determine token position P with lowest cardinality with respect to set of unique tokens.
  Create a partition for each token value in the set of unique tokens that appear in position P.
  for each line in partition **do**
    Add line to partition for the value that appears in position P of the line.
  end for
  Add newly created partitions to collection $Temp\_C$
  $Temp\_C = Partition\_Prune(Temp\_C)$ {See Algorithm 3}
  Add all partitions from $Temp\_C$ to C_Out
end for
$C\_Out = File\_Prune(C\_Out)$ {See Algorithm 1}
Return(C_Out)

Algorithm 3 Partition_Prune Function

Input: Collection $C[]$ of log file partitions.
Real number PS as partition support threshold.
Output: Collection $C[]$ of log file partitions with all partitions with support less than PS grouped into one partition.
Create temporary partition $Temp\_P$
for every *partition in* C **do**
  $Supp = \frac{\#LinesInPartition}{\#LinesInCollection}$
  if $Supp < PS$ **then**
    Add lines from partition to $Temp\_P$
    Delete partition from $C[]$
  end if
end for
Add partition $Temp\_P$ to collection $C[]$
Return(C)

token *on:* in position 3 and vice versa. On the other hand token *has* has a 1-M relationship with tokens *completed* and *been* as all lines that contain the token *has* in position 2 contains either tokens *completed* or *been* in position 3, a M-1 relationship will be the reverse of this scenario. To illustrate a M-M relationship, consider the event messages given below with positions 3 and 4 chosen using our heuristic.

```
Fan speeds 3552 3552 3391 4245 3515 3497
Fan speeds 3552 3534 3375 4787 3515 3479
Fan speeds 3552 3534 3375 6250 3515 3479
Fan speeds 3552 3534 3375 **** 3515 3479
Fan speeds 3311 3534 3375 4017 3515 3479
```

It is obvious that no discernible relationship can be found with the tokens in the chosen positions. Token *3552 (in position 3)* maps to tokens *3552 (in position 4)* and *3534*. On the other hand token *3311* also maps to token *3534*, this makes it impossible to split these messages using their token relationships. It is a scenario like this that we refer to as a M-M relationship.

In the case of *1-M* and *M-1* relations, the *M* side of the relation could represent variable values (so we are dealing with only one message type description) or constant values (so each value actually represents a different message type description). The diagram in Fig. 6 describes the simple heuristic that we developed to deal with this problem. Using the ratio between the number of unique values in the set and the number of lines that have these values in the corresponding token position in the partition, and two threshold values, a decision is made on whether to treat the *M* side as consisting of constant values or variable values. *M-M* relationships are iteratively split into separate *1-M* relationships or ignored depending on if the partition is coming from Step-1 or Step-2 of the partitioning process respectively.

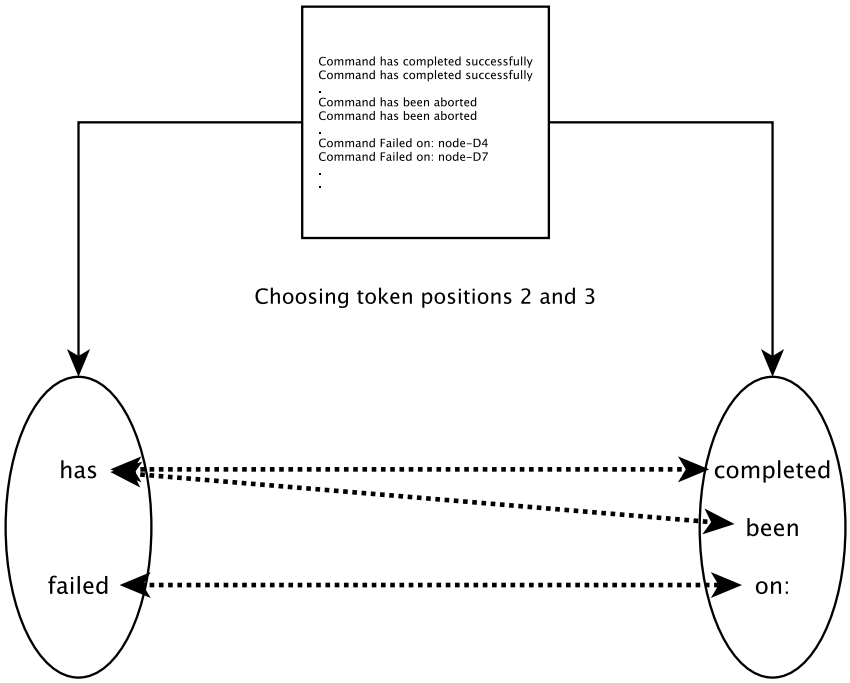


Figure 5: IPLoM Step-3: Partition by search for bijection.

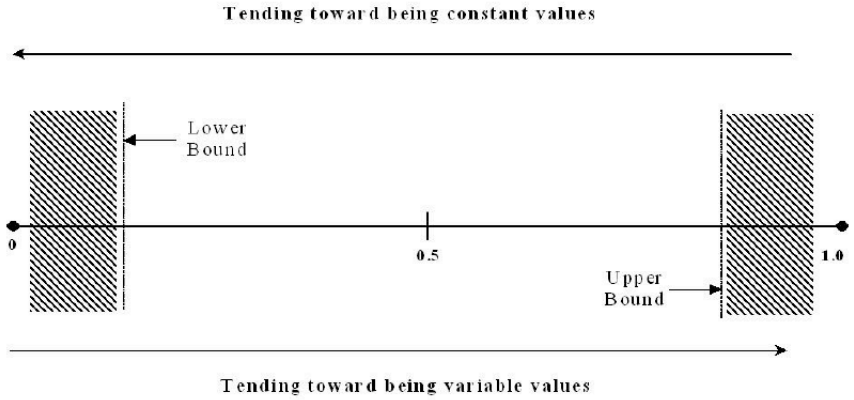


Figure 6: Deciding on how to treat 1-M and M-1 relationships.

Algorithm 4 IPLoM Step 3

Input: Collection C_In of partitions from Step 1 or Step 2.
Output: Collection C_Out of partitions derived from C_In.

for every $partition\ in\ C\_In as P\_In$ **do**
  Create temporary collection $Temp\_C$
  DeterminePlandP2(P_In){See Algorithm 5}
  Create sets $S1$ and $S2$ of unique tokens from $P1$ and $P2$ respectively.
  for each element in $S1$ **do**
    Determine mapping type of element in relation to $S2$.
    if mapping is $1-1$ **then**
      $split\_pos\ =\ P1$
    else if mapping is $1-M$ **then**
      Create set S_Temp with token values on the many side of the relationship.
      $split\_rank := Get\_Rank\_Position(S\_Temp)$. {See Algorithm 6.}
      if $split\_rank = 1$ **then**
        $split\_pos\ =\ P1$
      else
        $split\_pos\ =\ P2$
      end if
    else if mapping is $M-1$ **then**
      Create set S_Temp with token values on the many side of the relationship.
      $split\_rank := Get\_Rank\_Position(S\_Temp)$.
      if $split\_rank = 2$ **then**
        $split\_pos\ =\ P2$
      else
        $split\_pos\ =\ P1$
      end if
    else {mapping is $M-M$}
      if partition has gone through step 2 **then**
        Move to next token.
      else {partition is from step 1}
        Create sets S_Temp1 and S_Temp2 with token values on both sides of the relationship.
        if S_Temp1 has lower cardinality **then**
          $split\_pos\ =\ P1$
        else {S_Temp2 has lower cardinality}
          $split\_pos\ =\ P2$
        end if
      end if
    end if
    Split partition into new partitions based on token values in $split\_pos$.
    if partition is empty **then**
      Move to next partition.
    end if
  end for
  if partition is not empty **then**
    Create new partition with remainder lines.
  end if
  Add new partitions to $Temp\_C$
  $Temp\_C\ =\ Partition\_Prune(Temp\_C)$ {See Algorithm 3}
  Add all partitions from $Temp\_C$ to C_Out
end for
$C\_Out\ =\ File\_Prune(C\_Out)$ {See Algorithm 1}
Return(C_Out)

Algorithm 5 Procedure DetermineP1andP2

Input: Partition P.
Real number CT as cluster goodness threshold.
Determine event size of P as $token\_count$.
if $token\_count > 2$ **then**
  Determine the number of token positions with only one unique value as $count\_1$.
  $GC = \frac{count\_1}{token\_count}$
  if $GC < CT$ **then**
    $(P1, P2) = Get\_Mapping\_Positions(P)$ {See Algorithm 7}
  else
    Return to calling procedure, add P to C_Out and move to next partition.
  end if
else if $token\_count = 2$ **then**
  $(P1, P2) = Get\_Mapping\_Positions(P)$
else
  Return to calling procedure, add P to C_Out and move to next partition.
end if
Return()

Algorithm 6 Get_Rank_Position Function

Input: Set S of token values from the M side of a $1 - M$ or $M - 1$ mapping of a log file partition.
Real number $lower\_bound$.
Real number $upper\_bound$.
Output: Integer $split\_rank$. $split\_rank$ can have values of either 1 or 2.
$Distance = \frac{Cardinality\ of\ S}{\#Lines\_that\_match\_S}$
if $Distance \leq lower\_bound$ **then**
  if *Mapping is 1-M* ***then***
    $split\_rank = 2$
  else
    $split\_rank = 1$ {Mapping *is* M-1}
  end if
else if $Distance \geq upper\_bound$ **then**
  if *Mapping is 1-M* ***then***
    $split\_rank = 1$
  else
    $split\_rank = 2$ {Mapping *is* M-1}
  end if
else
  if *Mapping is 1-M* ***then***
    $split\_rank = 1$
  else
    $split\_rank = 2$ {Mapping *is* M-1}
  end if
end if
Return(split_rank)

Before partitions are passed through the partitioning process of Step 3 of the algorithm they are evaluated to see if they already form good clusters. To do this a *cluster goodness threshold* is introduced into the algorithm. The *cluster goodness threshold* is the ratio of the number of token positions that have only one unique value to the event size of the lines in the partition. Partitions that have a value higher than the cluster goodness threshold are considered good clusters and are not partitioned any further in this step.

3.5 Step 4: Discover message type descriptions (line formats) from each partition.

In this step of the algorithm partitioning is complete and we assume that each partition represents a cluster i.e. every log message in the partition was produced using the same line format. A message type description or line format consists of a line of text where constant values are represented literally and variable values are represented using wild-card values. This is done by counting the number of unique tokens in each token position of a partition. If a token position has only one value then it is considered a constant value in the line format, if it is more than one then it is considered a variable. This process is illustrated in Fig. 7.

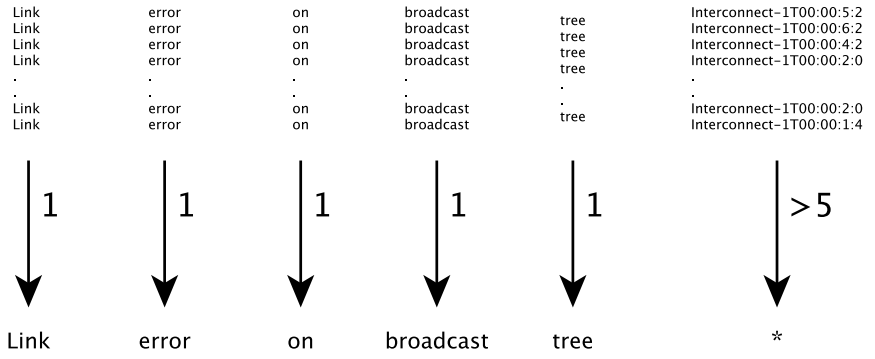


Figure 7: IPLoM Step-4: Discover message type descriptions. Numbers by arrows represent the number of unique tokens.

3.6 Algorithm Parameters

In this section we give a brief overview of the parameters/thresholds used by IPLoM. The fact that IPLoM has several parameters, which can be used to tune its performance, it provides flexibility for the system administrators since this gives them the option of using their expert knowledge when they see it necessary.

100pt[t]

Table 1: Log Data Statistics

Name	Description	No. of Messages	No. of Formats (Manual)
HPC	High Performance Cluster Log (Los Alamos)	433490	106
Syslog	OpenBSD Syslog	3261	60
Windows	Windows Oracle Application Log	9664	161
Access	Apache Access Log	69902	14
Error	Apache Error Log	626411	166
System	OS X Syslog	24524	9
Rewrite	Apache mod_rewrite Log	22176	10

- **File Support Threshold:** Ranges between *0-1*. It reduces the number of clusters produced by IPLoM. Any cluster whose instances have a support value less than this threshold is discarded. The higher this value is set to, the fewer the number of clusters that will be produced. This parameter is similar to the *"support threshold* defined for SLCT and Loghound.

- **Partition Support Threshold:** Ranges between *0-1*. It is essentially a threshold that controls backtracking. Based on our experiments, the guideline is to set this parameter to very low values i.e. < 0.05 for optimum performance.

- **Upper_Bound and Lower_Bound:** Ranges between *0-1*. They control the decision on how to treat M side of relationships in Step-2. Lower_Bound should usually take values < 0.5 while Upper_Bound takes values > 0.5.

- **Cluster Goodness Threshold:** Ranges between *0-1*. It is used to avoid further partitioning. Its optimal should lie in the range of $0.3 - 0.6$.

3.7 Experiments

In order to evaluate the performance of IPLoM, we selected open source implementations of algorithms previously used in system/application log data mining. For this reason SLCT, Loghound and Teiresias were selected. We therefore tested the four algorithms against seven log data-sets which we compiled from different sources, Table 1 gives an overview of the data-sets used. The HPC log file is an open source data-set collected on high performance clusters at the Los Alamos National Laboratory NM, USA [21]. The Access, Error, System and Rewrite data-sets were collected on our faculty network at Dalhousie, while the Syslog and Windows files were collected on servers owned by a large ISP working with our research group. Due to privacy issues we are not able to make this data available to the public.

The message type descriptions of these 7 data-sets were produced manually by Tech-Support members of the Dalhousie Faculty of Computer Science. Table 1 gives the number of clusters identified in each file manually. Again due to privacy issues we are able to provide manually produced cluster descriptions

only for the HPC data [1]. These cluster descriptions then became our gold standard, against which to measure the performance of the algorithms as an information retrieval (IR) task. As in classic IR, our performance metrics were Recall, Precision and F-Measure which are described by Eqs. (1), (2) and (3) respectively. The terms TP, FP and FN in the equations are the number of True Positives, False Positives and False Negatives respectively. Their values are derived by comparing the set of manually produced message type descriptions to the set of retrieved formats produced by each algorithm. In our evaluation a message type description is still considered an FP even if matches a manually produced message type description to some degree, the match has to be exact for it to be considered a TP.

For completeness we evaluated our Precision, Recall and F-Measure values using three different methods. In the two methods we evaluated the results of the algorithms as a classification problem. Using the manually produced event types as classes we evaluated how effectively the automatically produced classification matched the manually produced labels. This classification evaluation produced Micro-average and Macro-average results. These results are referred to as *"Micro"* and *"Macro"* in the results section. In the third method the manually produced message type descriptions are compared against the automatically produced ones. This evaluation method is called "IR" in the results section. We however believe that what we called the "IR" method evaluation satisfies our goals better as it tests the goodness of the clusters produced. The next section gives more details about the results of our experiments.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

4 Results

In our first set of experiments we tested SLCT, Loghound, Teiresias and IPLoM on the data-sets outlined in Table 1. The parameter values used in running the algorithms in all cases are provided in Tables 2, 3 and 4. The rationale for choosing the support values used for SLCT, Loghound and IPLoM is explained later in this section and this leads to two classes of experiments. The seed value for SLCT and Loghound is a seed for a random number generator used by the algorithms, all other parameter values for SLCT and Loghound are left at their default values. The parameters for Teiresias were also chosen to achieve the lowest support value allowed by the algorithm. The IPLoM parameters were

[1] Descriptions are available for download from http://torch.cs.dal.ca/~makanju/iplom

all set empirically except in the case of the cluster goodness threshold and the partition support threshold .

In setting the cluster goodness threshold we ran IPLoM on the HPC file while varying this value. The parameter was then set to the value (0.34) that gave the best result and was kept constant for the other files used in our experiments. On the other hand, the partition support threshold was set to 0 to provide a baseline performance. A 0 setting for the performance threshold implies that no backtracking was done during partitioning.

It is pertinent to note that we were unable to test the Teiresias algorithm against all our data-sets. This was due to its inability to scale to the size of our data-sets. This is a problem that is attested to in [5]. Thus in this work, Teiresias could only be tested against the Syslog data-set, as it crashed against all the other data sets.

Table 2: SLCT and Loghound Parameters

Parameter	**Value**
Support Threshold (-s)	0.01 - 0.1
Seed (-i)	5

Table 3: Teiresias Parameters

Parameter	**Value**
Sequence Version	On
L (min. no. of non wild card literals in pattern)	1
W (max. extent spanned by L consecutive non wild card literals)	15
K (Min. no. of lines for pattern to appear in)	2

Table 4: IPLoM Parameters

Parameter	**Value**
File Support Threshold (Percentage)	0 - 0.1
File Support Threshold (Absolute)	1 - 20
Partition Support Threshold	0
Lower Bound	0.1
Upper Bound	0.9
Cluster Goodness Threshold	0.34

SLCT, Loghound and Teiresias need a line support threshold to produce clusters. For SLCT and Loghound this support value can be specified either as a percentage of the number of events in the event log or as an absolute value. For this reason we run two set of experiments using percentage specified support values and absolute value specified support values. In either case we set these support values low because intuitively this allows for finding most of the clusters in the data, which is one of our goals. The next two sections present the results of our first set of experiments.

4.1 Absolute Support Values

In this set of experiments we compare the result of the algorithms using absolute support values in the range of 1 - 20. SLCT and Loghound cannot be run with an absolute support value of 1, so we run them with 2 instead. An absolute support value of 1 means every line/word will be considered frequent and the result of the algorithms will be reduced to the trivial case of finding unique lines. An absolute support value of 1 however is IPLoM's default setting, which highlights an advantage of IPLoM, i.e. a line support value is optional. Since Teiresias worked only on the Syslog data-set, its results are not included in our analysis, which used the parameter values listed in Table 3, Teiresias produced a Recall performance of 0.1, a Precision performance of 0.04 which led to an F-Measure performance of 0.06 using the IR evaluation method.

The average F-Measure results for the other 3 algorithms are highlighted in Table 5. The results show that IPLoM performs better than the other algorithms on all data sets in the IR evaluation, which measures the goodness of the clusters produced. In the Micro and Macro evaluations, IPLoM still does better than the other algorithms in general. However, we see performance improvement from SLCT and Loghound and in one case (with the Syslog data-set) SLCT actually performs better than IPLoM.

The rest of the results are evaluated using the IR method. A detailed summary of the IR Recall, Precision and F-Measure results can be found in Section C of the Appendix.

Table 5: Average F-Measure performance of algorithms using absolute support values

F-MEASURE PERFORMANCE						
	HPC			Syslog		
	SLCT	**Loghound**	**IPLoM**	**SLCT**	**Loghound**	**IPLoM**
Micro	0.64	0.55	0.66	0.10	0.06	0.07
Macro	0.25	0.45	0.45	0.13	0.07	0.11
IR	0.02	0.01	0.59	0.14	0.08	0.14
	Windows			**Access**		
	SLCT	**Loghound**	**IPLoM**	**SLCT**	**Loghound**	**IPLoM**
Micro	0.22	0.25	0.28	0.00	0.00	0.00
Macro	0.17	0.18	0.22	0.11	0.15	0.20
IR	0.18	0.11	0.34	0.00	0.00	0.26
	Error			**System**		
	SLCT	**Loghound**	**IPLoM**	**SLCT**	**Loghound**	**IPLoM**
Micro	0.68	0.28	0.82	0.20	0.16	0.83
Macro	0.22	0.17	0.31	0.18	0.09	0.56
IR	0.01	0.01	0.43	0.15	0.07	0.75
	Rewrite					
	SLCT	**Loghound**	**IPLoM**			
Micro	0.08	0.05	0.83			
Macro	0.10	0.13	0.30			
IR	0.01	0.01	0.49			

As discussed earlier, IPLoM was designed to be able to run without the provision of a line support value, however the concept of a line support value can be introduced. Its performance at this support value could therefore be

Table 6: Log Data Event Size Statistics

Name	Min	Max	Avg.
HPC	1	95	30.7
Syslog	1	25	4.57
Windows	2	82	22.38
Access	3	13	5.0
Error	1	41	9.12
System	1	11	2.97
Rewrite	3	14	10.1

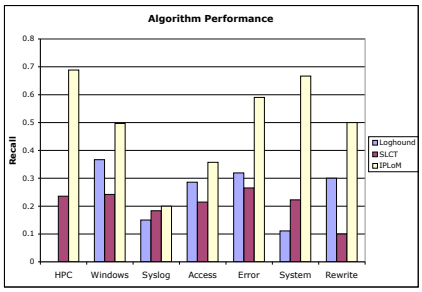
Table 7: Algorithm performance based on cluster event size

Event Size Range	No. of Clusters	Percentage Retrieved(%)		
		SLCT	Loghound	IPLoM
1 - 10	316	12.97	13.29	53.80
11 - 20	142	7.04	9.15	49.30
>21	68	15.15	16.67	51.52

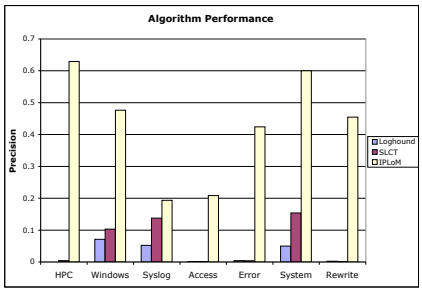
considered its default performance. As SLCT and Loghound cannot be run without providing a support value, the closest thing to running them without a support value will be an absolute support value of 2. When we look at the IR performance of the algorithms at this support level we see that the average F-Measure performance across the data-sets is 0.07, 0.04 and 0.46 for SLCT, Loghound and IPLoM respectively. The graphs in Figure 8 visualize these results for Recall, Precision and F-Measure metrics for all the algorithms.

However, as stated in [22], in cases where data sets have relatively long patterns or low minimum support thresholds are been used, apriori based algorithms incur non-trivial computational cost during candidate generation. The event size statistics for our data sets are outlined in Table 6, which shows the *HPC* file as having the largest maximum and average event size. Loghound was unable to produce results on this data set with an absolute support value of 2, because the algorithm crashed due to the large number of item-sets that had to be generated as can be seen in Figure 8. This was however not a problem for SLCT (as it generates only 1-item-sets). This results show that Loghound is vulnerable to the computational cost problems outlined in [22], which is however not a problem for IPLoM as its computational complexity is not adversely affected by long patterns or low minimum support thresholds. In terms of performance based on event size, Table 7 shows consistent performance from IPLoM irrespective of event size, while SLCT and Loghound seem to suffer for mid-size clusters.

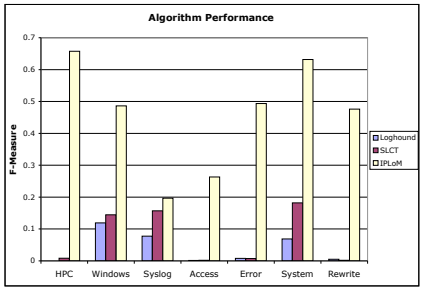
One of the cardinal goals in the design of IPLoM is the ability to discover clusters in event logs irrespective of how frequently its instances appear in the data. The performance of the algorithms using this evaluation criterion is outlined in Table 8. The results show a reduction in performance for all the algorithms for clusters with a few instances, however IPLoM's performance was more resilient.



(a) Recall



(b) Precision



(c) F-Measure

Figure 8: Comparing algorithm IR performance at lowest support values.

Table 8: Algorithm performance based on cluster instance frequency

Instance Frequency Range	No. of Clusters	Percentage Retrieved(%)		
		SLCT	Loghound	IPLoM
1 - 10	263	2.66	1.90	44.87
11 - 100	144	16.67	18.75	47.92
101 - 1000	68	20.59	23.53	72.06
>1000	51	34.00	38.00	82.00

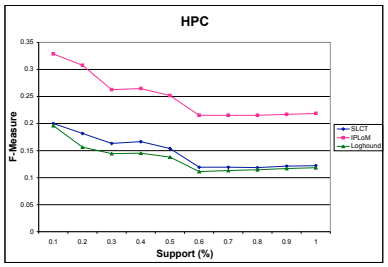
4.2 Percentage based Support Values

A system administrator can specify a support value using an absolute value (as in the section above) or a value that is dependent on the number of lines in the event log i.e. a percentage. To determine IPLoMs performance according to this specication of support value, we ran another set of experiments using percentage based support values. For the same reasons as described above the range of values are low i.e. 0.1% - 1.0%. The F-Measure results of this scenario are shown in Fig 9, where we note IPLoM performing better than the other algorithms on all the tasks. A single factor ANOVA test done at 5% significance on the results shows a statistically significant difference in all the results except in the case of the Syslog file. Table 9 provides a summary of these results. Similar results for Recall and Precision are given in sections B, D and E of the Appendix.

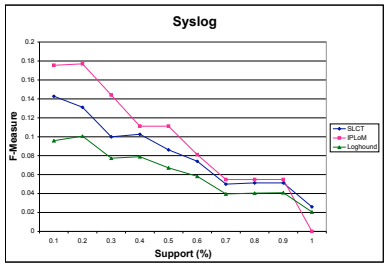
4.3 Parameter Sensitivity Analysis

IPLoM has 5 parameter values which can affect its results. These parameters are the File Support Threshold (FST), Partition Support Threshold (PST), Cluster Goodness Threshold (CGT) and the Upper Bound (UB) and Lower Bound (LB) thresholds used to decide if the "many" end of a 1-M relationship represents constant values or variable values. It is important that we assess the sensitivity of IPLoM's performance to the value settings of these parameters. In this section we present such an analysis. We ran IPLoM against the data-sets using a wide range of values as outlined in Table 10, because the FST used in IPLoM is similar to the support threshold used in SLCT and Loghound we also ran tests on them using the range of values for FST in Table 10.

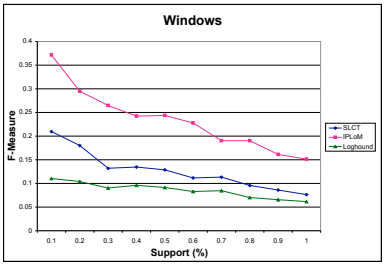
The results show that IPLoM is most sensitive to varying values of FST as can be seen in Fig. 10. This can be explained by the observation that increasing the support value decreases the number of event types that can be found, since any event type with instances that fall below the support value cannot be found. The graphs however show that generally for support values greater than 20% there is not much difference in the performance of the algorithms. Using the standard deviation of the results over the range of results for each parameter, as seen in Table 11, we can evaluate the sensitivity of the algorithms to changing parameter values. The results show that IPLoM is stable in face of changing parameter values. The largest standard deviation values are found with IPLoM



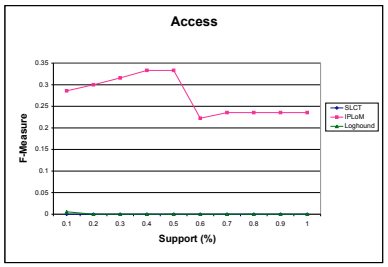
(a) HPC



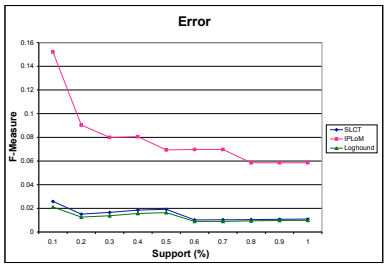
(b) Syslog



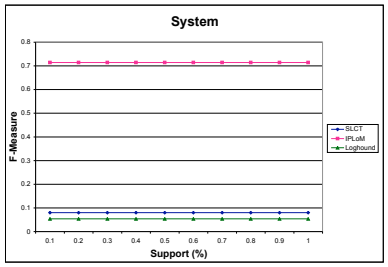
(c) Windows



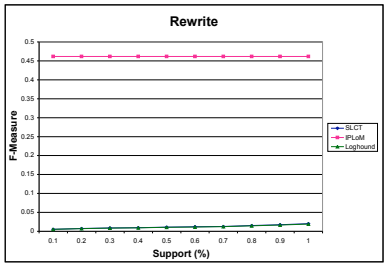
(d) Access



(e) Error



(f) System



(g) Rewrite

Figure 9: Comparing F-Measure performance of IPLoM, Loghound and SLCT using percentage based support values.

Table 9: Anova Results for F-Measure Performance

HPC						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.08	2	0.04	35.35	2.89E-08	3.35
Within Groups	0.03	27	0.00			
Total	0.03	29				
SYSLOG						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.01	2	0.00	1.63	0.21	3.35
Within Groups	0.04	27	0.00			
Total	0.05	29				
WINDOWS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.08	2	0.04	26.54	4.22E-07	3.35
Within Groups	0.04	27	0.00			
Total	0.12	29				
ACCESS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.50	2	0.25	366.31	2.73E-20	3.35
Within Groups	0.02	27	0.00			
Total	0.51	29				
ERROR						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.03	2	0.01	51.34	6.291E-10	3.35
Within Groups	0.01	27	0.00			
Total	0.03	29				
SYSTEM						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	2.80	2	1.40	1.96E+34	0	3.35
Within Groups	1.93E-33	27	7.13E-35			
Total	2.80	29				
REWRITE						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	1.35	2	0.67	51076.72	4.98E-49	3.35
Within Groups	4.0E-04	27	1.32E-05			
Total	1.34	29				

under the FST parameter, which is due to IPLoM's superior performance for FST values less than 20%.

4.4 Analysis of Incorrect Results

The IPLoM algorithm, as with all algorithms that utilize heuristics, is capable of making errors and does in fact make errors during its partitioning phase. Our analysis of these errors is described in this section. In some cases these problems can be mitigated by preprocessing of the event data or post-processing of results, and we also describe the possible remedy.

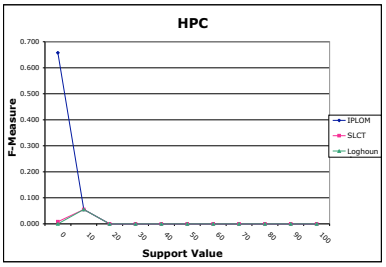
4.4.1 Insufficient Information in Data

Apart from the cluster descriptions produced by all the algorithms as output, IPLoM has the added advantage of producing the partitions of the log data which represent the actual clusters. This gives us two sets of results we can evaluate for IPLoM, the clusters and their descriptions. In our evaluation of the partition results of IPLoM, we discovered that in certain cases that it was impossible for IPLoM to produce the right cluster descriptions for a partition due to the fact that the partition contained only one event line or all the event lines were identical. This situation would not pose a problem for a human subject as they are able to use semantic and domain knowledge to determine the right cluster description. This problem is illustrated in Fig. 11. This indicates that the IR comparison of the cluster descriptions produced by IPLoM does not give a complete picture of IPLoM's performance. To get a complete picture of IPLoM's capabilities, we evaluated IPLoM's performance based on partitioning. These results are called *Partitions* in Fig. 12, while the cluster description results are called *Before* . The partition comparison differs from the cluster description by including as correct, cases where IPLoM came up with the right partition but was unable to come up with the right cluster description. The results show an average F-Measure of 0.48 and 0.78 for IPLoM when evaluating the results of IPLoM's cluster description output and partition output respectively. Similar results are also noticed for Precision and Recall.

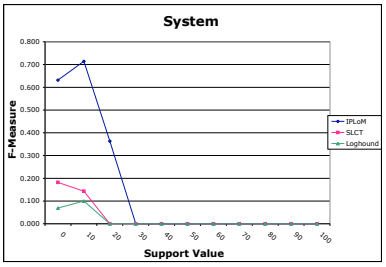
Due to the fact that SLCT and Loghound do not generate partitions to evaluate against (these partitions can however be found through post-processing if desired) and it can be argued that the insufficient information in data scenario could also apply to them, we constructed another experiment. In this case, we

Table 10: Parameter Value Ranges Used for Sensitivity Analysis

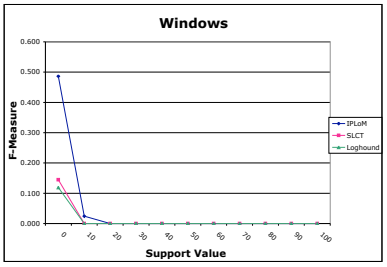
Parameter	**Range**
File Support Threshold(%)	0 - 100
Partition Support Threshold(%)	0 - 5
Lower Bound	0.1 - 0.5
Upper Bound	0.5 - 0.9
Cluster Goodness Threshold	0 - 1



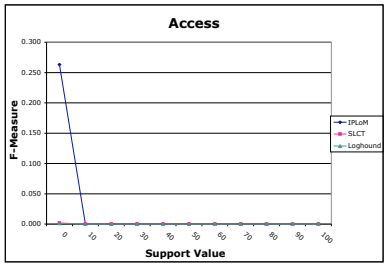
(a) HPC



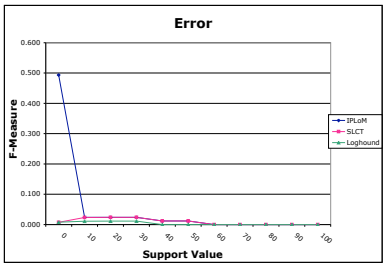
(b) Syslog



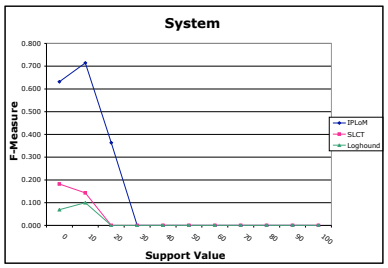
(c) Windows



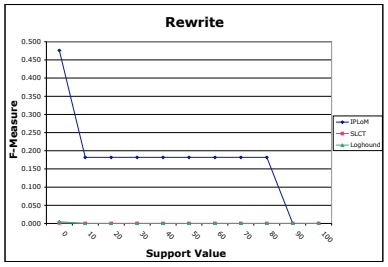
(d) Access



(e) Error



(f) System



(g) Rewrite

Figure 10: F-Measure performance of IPLoM, Loghound and SLCT against FST values in the range 0% - 100%. 0% support values for SLCT and Loghound are equivalent to using an absolute support value of 2.

inserted counter-examples for all the cases where there was insufficient information in the event data for the algorithms to come up with the cluster descriptions and ran SLCT, Loghound and IPLoM against the new datasets with counter-examples inserted. SLCT and Loghound were run in this case with the absolute support values which gave their best results in the experiments described in Section 4.1 above while IPLoM was run in its default state. This results are called *After* in Table 12. The results show that unlike SLCT and Loghound, IPLoM was able to make use of the new information to improve its results in all cases. IPLoM's *After* results however did not measure up to its *Partitions* due to the fact in some cases the third step of IPLoM's partitioning process was able to discern some of the counter-examples and place them in separate groups at the end of the partitioning process. This however did not occur in all cases hence we see improved performance.

We consider the *Partitions* results to be the most accurate illustration of IPLoM's capabilities. These results show that IPLoM can achieve an average Recall of 0.83, Precision of 0.74 and F-Measure of 0.78.

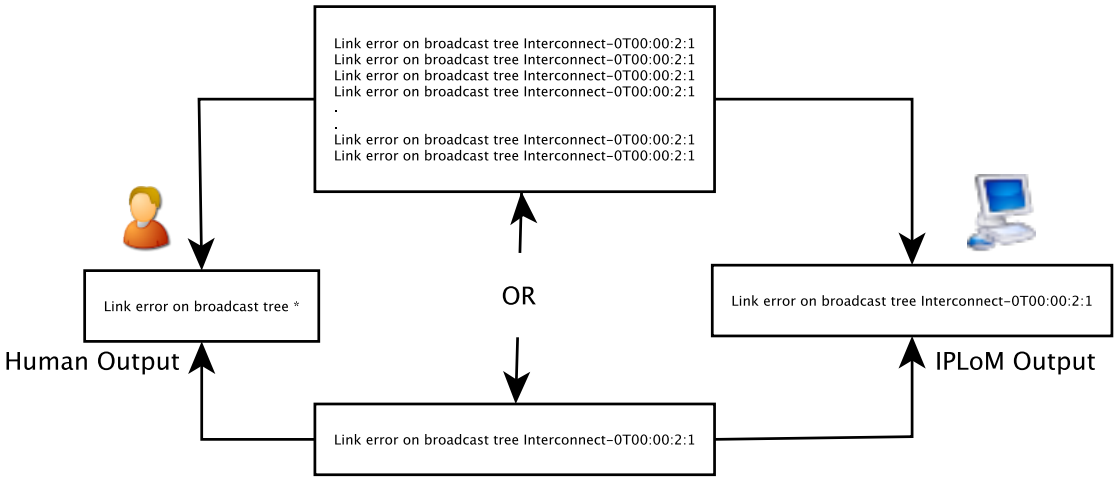


Figure 11: Example: Insufficient Information in Data.

Table 11: Standard Deviation over F-Measure results for parameter values

	FST			PST	CGT	LB:UB
	Loghound	**SLCT**	**IPLoM**	**IPLoM**	**IPLoM**	**IPLoM**
HPC	0.017	0.017	0.197	0.009	0.107	0.048
Syslog	0.025	0.000	0.058	0.000	0.013	0.025
Windows	0.036	0.044	0.146	0.002	0.088	0.018
Access	0.000	0.001	0.079	0.036	0.085	0.006
Error	0.005	0.010	0.146	0.034	0.034	0.028
System	0.035	0.066	0.279	0.000	0.000	0.197
Rewrite	0.001	0.000	0.123	0.000	0.000	0.000

4.4.2 Ambiguous Token Delimiters

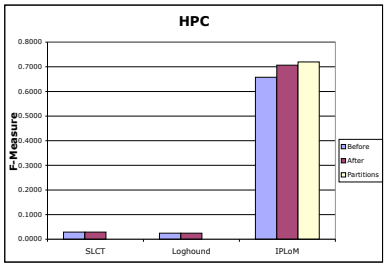
In the problem of message type extraction, the assumption is that the space character acts as the token delimiter. On close inspection of certain messages in the log files we find this not to be true in all cases. The most common example occurs when part or all of a message contains a *"variable = value"* phrase. In some cases there's no space character between the *variable* token and the = sign and also between the *value* token and the = sign. This scenario becomes a problem for IPLoM when for instance these log messages *"Temperature reading: ambient=30"*, *"Temperature reading: ambient=25"* and *"Temperature reading: ambient=28* are evaluated to the type *"Temperature reading: ambient=*"* by a human observer. When and if IPLoM correctly produces a partition containing these log messages the type produced will be *"Temperature reading: *"*, due to IPLoM's inability to separate the tokens in the *"variable = value"* phrase. This scenario can also occur in other cases when token delimiters are ambiguous.

An approach to mitigating this problem is outlined in [5]. By scanning for words containing an = sign before mining for clusters and splitting such words into three parts at the = sign. The word triple can then be concatenated at the end of the clustering process, this will ensure that we can still match future instances of the message type to the message type description produced.

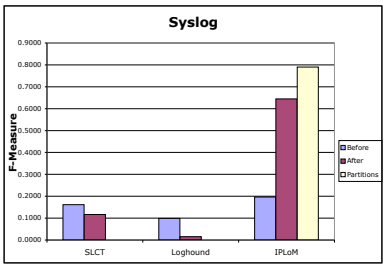
4.4.3 Clusters with events of variable size

Another scenario that occurs is with clusters with events of variable size. We assume that messages belonging to the same cluster should have the same number of tokens or event size. Again on close inspection we find this not to be true in some cases. Clusters with events of variable sizes usually occur when a variable position in the line format can contain strings instead just single words. For example consider these messages *"The LightScribe service has started"* and *"The Message Queuing service has started"* which should belong to the same message type and have message type description *"The * service has started"*, having a differing number of tokens. These messages would be separated by the Step-1 of IPLoM's partitioning process and IPLoM would likely produce two message type descriptions for this cluster *"The * service has started"* and *"The * * service has started"*, the latter line format being redundant.

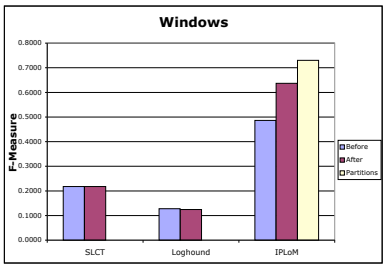
This problem can be mitigated by performing message type description refinement after the message types are produced. By searching through the message type descriptions and finding message type descriptions with consecutive occurrences of the wild-card character *"*"* (that is more than one wild-card character appearing directly after each other). When this message type descriptions are found, we can replace the consecutive multiple wild-card characters with a single wild-card character, doing this will make all the message type descriptions produced for the clusters with events of variable size identical. We can then search through the complete list of message type descriptions for repetitions. If repetitions are found the message type description with consecutive occurrences of the wild-card character can then be discarded. The message type



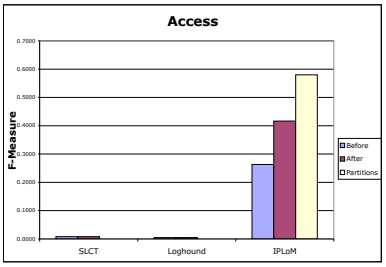
(a) HPC



(b) Syslog

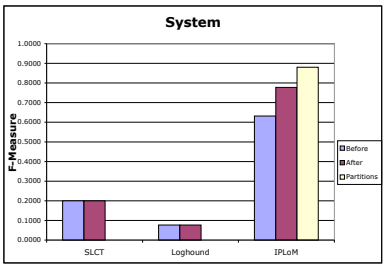


(c) Windows

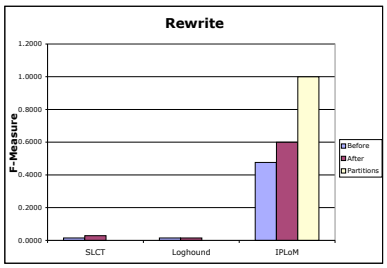


(d) Access

(e) Error



(f) System



(g) Rewrite

Figure 12: Comparing F-Measure performance of IPLoM, Loghound and SLCT before insertion of counter-examples, after insertion of counter examples and evaluating the accuracy of cluster partitioning before the insertion of counter examples.

description can be returned unchanged to the list of message type descriptions if no repetition is found.

5 Conclusion and Future Work

Due to the size and complexity of sources of information used by system administrators in fault management, it has become imperative to find ways to manage these sources of information automatically. Application logs are one such source.

In this work, we present our work on designing a novel message type extraction and partitioning algorithm, IPLoM. Through a 3-Step hierarchical partitioning process IPLoM partitions log data into its respective clusters. In its 4th and final stage IPLoM produces message type descriptions or line formats for each of the clusters produced. IPLoM is able to find message clusters whether or not its instances are frequent. We also ascertained that IPLoM produces cluster descriptions which match human judgement more closely when compared to SLCT, Loghound and Teiresias. It is also shown that IPLoM demonstrated statistically significantly better performance than either SLCT or Loghound on six of the seven different datasets tested.

Message types are fundamental units in any application log file. Determining what message types can be produced by an application accurately and efficiently is therefore a fundamental step in the automatic analysis of log files. Message types, once determined, not only provide groupings for categorizing and summarizing log data, which simplifies further processing steps like visualization or mathematical modeling, but also provides a way of labeling the individual terms (distinct word and position pairs) in the data.

Future work on IPLoM will involve using the information derived from the results of IPLoM in other automatic log analysis tasks which help with fault management.

With the issue of message types sorted out, our overall goal with this work is to develop methods for automatically extracting information about previous failure events from application logs and feeding this information into a knowledge-base which can be searched by system administrators when a new failure event occurs. When a network fault occurs system administrators attempt to relate symptoms on the network with possible root causes and solutions. Over time experienced administrators can draw on past experience to solve fault cases quickly. The value of experience gained by system administrators is therefore high. A system that cannot only store this sort of *experience* but can also retrieve this *experience information* without manual input would not only be advantageous for experienced system administrators by aiding recall but will shorten the learning curve for new administrators. This would lead to an optimal use of resources during downtime events.

According to [24], the process of troubleshooting a network fault follows a seven-step process. The 4th and 5th stages of this process i.e. root cause identification and resolution planning are said to take up to 80% of the time

spent troubleshooting network failure. Reducing the amount of time spent in troubleshooting is an open research question [25]. Previous work in this area has proposed knowledge-base systems where administrators can store and retrieve information about previous down time events [25], [26], [27], [28],[24]. These knowledge-base systems however do not always meet expectations because they require the manual entry of information. System administrators usually lack the incentive to input their knowledge into the knowledge-base.

Automating the population of such knowledge-bases is therefore desirable. As log files form a major source of information used by system administrators in troubleshooting, the importance of automating the extraction of fault related information from them is self-evident. Using machine learning and information retrieval techniques, coupled with message type clusters, this goal can be achieved. Our future work with automatic fault management will proceed along these lines.

Acknowledgements

This research is supported by a Natural Science and Engineering Research Council of Canada (NSERC) Strategic Project Grant. The authors would also like to acknowledge the staff of Palomino System Innovations Inc., TARA Inc. and Dal-CS Tech-Support for their support in completing this work. This work is conducted as part of the Dalhousie NIMS Lab at http://www.cs.dal.ca/projectx/.

References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer, Monthly publication of the IEEE Computer Society*, vol. 36, pp. 41– 50, June 2003.

[2] M. Klemettinen, "A Knowledge Discovery Methodology for Telecommunications Network Alarm Databases," Ph.D. dissertation, University of Helsinki, 1999.

[3] S. Ma, , and J. Hellerstein, "Mining Partially Periodic Event Patterns with Unknown Periods," in *Proceedings of the 16th International Conference on Data Engineering*, 2000, pp. 205–214.

[4] Q. Zheng, K. Xu, W. Lv, and S. Ma, "Intelligent Search for Correlated Alarm from Database Containing Noise Data," in *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2002, pp. 405–419.

[5] J. Stearley, "Towards Informatic Analysis of Syslogs," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004, pp. 309–318.

[6] R. Vaarandi, "Mining Event Logs with Slct and Loghound," in *Proceedings of the 2008 IEEE/IFIP Network Operations and Management Symposium*, April 2008, pp. 1071–1074.

[7] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 12–15 1994, pp. 487–499.

[8] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns from Event Logs," in *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, 2003, pp. 119–126.

[9] ——, "A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs," in *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems (LNCS)*, vol. 3283, 2004, pp. 293–308.

[10] I. Rigoutsos and A. Floratos, "Combinatorial Pattern Discovery in Biological Sequences: The Teiresias Algorithm," in *BioInformatics*, vol. 14. Oxford University Press, 1998, pp. 55–67.

[11] C. Lonvick, "The BSD Syslog Protocol," RFC3164, August 2001.

[12] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic Subspace Clustering of High Dimensional for Data Mining Applications," in *Proceedings of the ACM SIGMOID International Conference on Management of Data*, 1998.

[13] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," in *Proceedings of ACM SIGMOD International Conference on Management of Dat*, 1998, pp. 73–84.

[14] S. Goil, H. Nagesh, and A. Choudhary, "MAFIA: Efficient and Scalable Subspace Clustering for Very Large Data Sets," Northwestern University, Tech. Rep., 1999.

[15] J. H. Bellec and M. T. Kechadi, "CUFRES: Clustering using Fuzzy Representative Events Selection for the Fault Recognition Problem in Telecommunications Networks," in *PIKM '07: Proceedings of the ACM first Ph.D. workshop in CIKM*. New York, NY, USA: ACM, 2007, pp. 55 – 62.

[16] T. Li, F. Liang, S. Ma, and W. Peng, "An Integrated Framework on Mining Log Files for Computing System Management," in *Proceedings of of ACM KDD 2005*, 2005, pp. 776–781.

[17] B. T. et. al., "Automating Problem Determination: A First Step Toward Self Healing Computing Systems," IBM White Paper, October 2003. [Online]. Available: http://www-106.ibm.com/developerworks/autonomic/library/ac-summary/ac-prob.html

[18] J. Stearley, "Sisyphus Log Data Mining Toolkit," Accessed from the Web, January 2009. [Online]. Available: http://www.cs.sandia.gov/sisyphus

[19] A. Makanju, S. Brooks, N. Zincir-Heywood, and E. E. Milios, "Logview: Visualizing Event Log Clusters," in *Proceedings of Sixth Annual Conference on Privacy, Security and Trust. PST 2008*, October 2008, pp. 99 – 108.

[20] F. Salfener and M. Malek, "Using Hidden Semi-Markov Models for Effective Online Failure Prediction," in *26th IEEE International Symposium on Reliable Distributed Systems.*, 2007, pp. 161–174.

[21] L. Los Alamos National Security, "Operational Data to Support and Enable Computer Science Research," Published to the web, January 2009. [Online]. Available: http://www.pdl.cmu.edu/FailureData/andhttp://institutes.lanl.gov/data/fdata/

[22] J. Han, J. Pei, and Y.Yin, "Mining Frequent Patterns without Candidate Generation," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 1–12.

[23] A. Oliner, A. Aiken, and J. Stearley, "Alert Detection in System Logs," in *Proceedings of the International Conference on Data Mining (ICDM). Pisa, Italy.* Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 959–964.

[24] G. Penido and J. Nogueira, "An Automatic Fault Diagnosis and Correction System for Telecommunications Management," in *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, 1999, pp. 777–791.

[25] A. George, A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Information Retrieval in Network Administration," in *Proceedings of the 6th Annual Communication Networks and Services Research Conference.*, May 2008, pp. 561 – 568.

[26] R. Cronk, P. Callahan, and L. Bernstein, "Rule-based Expert Systems for Network Management and Operations: An Introduction," in *IEEE Network*, 1988, pp. 7–21.

[27] H. Inamura, O. Takahashi, T. Ishikawa, H. Shigeno, and K. Okada, "Automating Detection of Faults in TCP Implementations," in *18th International Conference on Advanced Information Networking and Applications*, vol. 1, 2004, pp. 315–320.

[28] L. Lewis, "A Case-based Reasoning Approach to the Management of Faults in Communication Networks," in *Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1993, pp. 1422–1429.

Appendix

A Step-1: Partition by event size

Algorithm 8 provides a detailed description of the first step of the IPLoM algorithm.

Algorithm 8 IPLoM Step 1

Input: Log file containing log messages.
Output: Collection C of log file partitions.
1: **for** each line in the log file **do**
2: Determine the count of tokens in line as *token_count*. {Token delimiter is assumed to be space character.}
3: **if** partition for lines with *token_count* tokens exists **then**
4: Add line to the appropriate partition.
5: **else**
6: Create partition for lines with *token_count* tokens.
7: Add line to the appropriate partition.
8: **end if**
9: **end for**
10: Add all partitions to C.
11: $C = File\_Prune(C)$
12: Return(C)

B Precision and Recall Performance

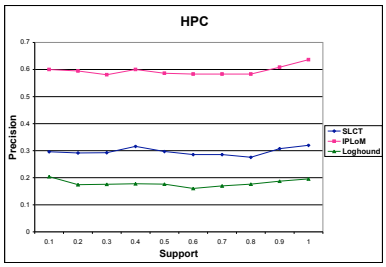
Figs. 13 and 14 show the comparison of the Precision and Recall performances of IPLoM against those of SLCT and Loghound. The results show as IPLoM outperforming both algorithms in all cases.

C IR Results Summary for Absolute Support Value Experiments

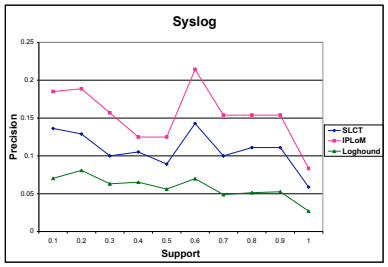
This section shows the IR Recall, Precision, and F-Measure results summary for the absolute support value experiments. The results are presented in Tables 12, 13 and 14 respectively.

Table 12: IR Recall Result Summary for Absolute Support Value Experiments

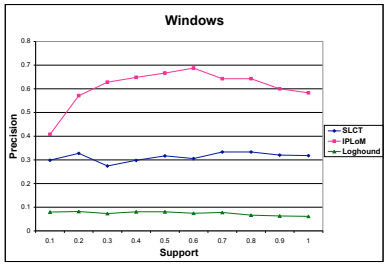
	SLCT	**Loghound**	**IPLoM (CD)**
HPC	0.25	0.45	0.55
Syslog	0.13	0.10	0.13
Windows	0.17	0.22	0.26
Access	0.11	0.19	0.24
Error	0.23	0.21	0.38
System	0.18	0.11	0.67
Rewrite	0.10	0.16	0.36



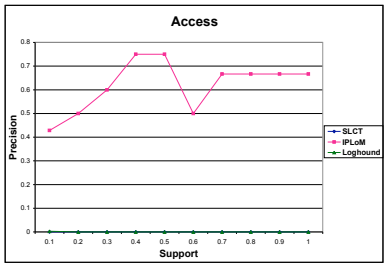
(a) HPC



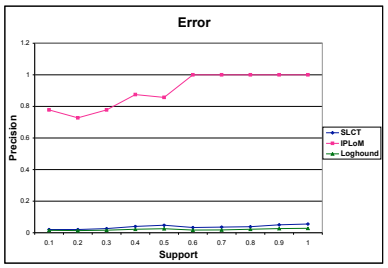
(b) Syslog



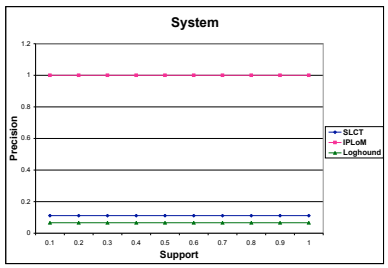
(c) Windows



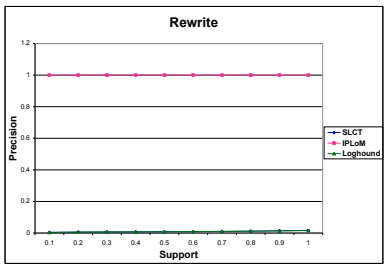
(d) Access



(e) Error

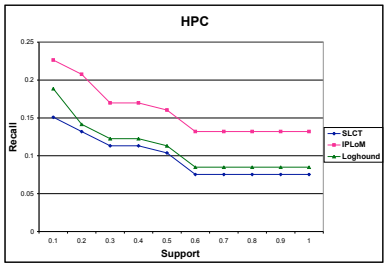


(f) System

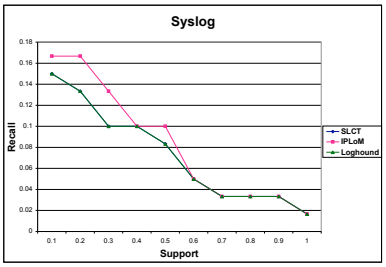


(g) Rewrite

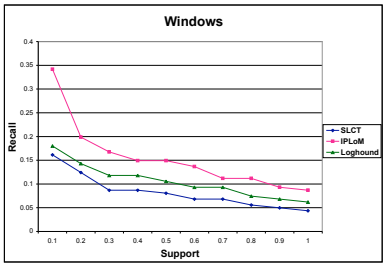
Figure 13: Comparing Precision performance of IPLoM, Loghound and SLCT using percentage support values.



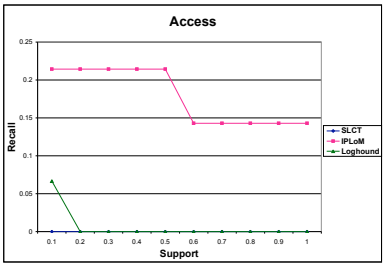
(a) HPC



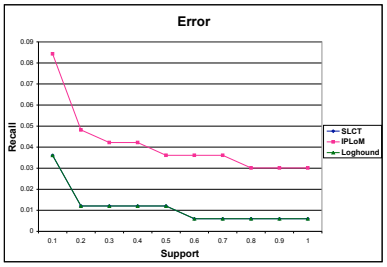
(b) Syslog



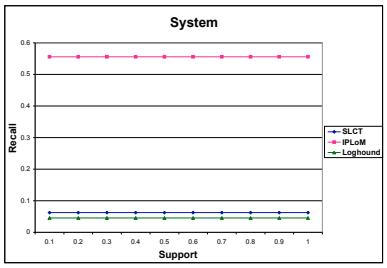
(c) Windows



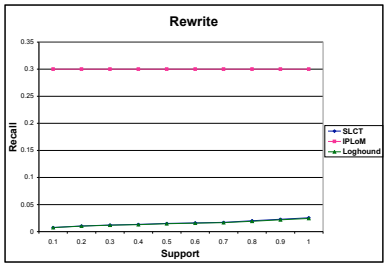
(d) Access



(e) Error



(f) System



(g) Rewrite

Figure 14: Comparing Recall performance of IPLoM, Loghound and SLCT using percentage support values.

D Anova Results on Precision Performance

This section shows the results of the Anova test on the precision performance of IPLoM, SLCT and Loghound in Table 15. The results show a statistical significant difference in all cases.

E Anova Results on Recall Performance

This section shows the results of the Anova test on the Recall performance of IPLoM, SLCT and Loghound in Table 15. The results do not show a statistical significant difference in all cases, the HPC, Syslog and Windows files did not show a statistically significant difference.

Table 13: IR Precision Result Summary for Absolute Support Value Experiments

	SLCT	**Loghound**	**IPLoM (CD)**
HPC	0.01	0.01	0.64
Syslog	0.16	0.06	0.18
Windows	0.25	0.08	0.58
Access	0.00	0.00	0.30
Error	0.01	0.01	0.58
System	0.14	0.05	0.87
Rewrite	0.01	0.01	0.89

Table 14: IR F-Measure Result Summary for Absolute Support Value Experiments

	SLCT	**Loghound**	**IPLoM (CD)**
HPC	0.02	0.01	0.59
Syslog	0.14	0.08	0.14
Windows	0.18	0.11	0.34
Access	0.00	0.00	0.26
Error	0.01	0.01	0.43
System	0.15	0.07	0.75
Rewrite	0.01	0.01	0.49

Algorithm 7 Get_Mapping_Positions Function

Input: Log file partition P.
Output: Integer token positions $P1$ and $P2$ as ($P1$,$P2$).

Determine event size of P as *count_token*
if *count_token* = *2* **then**
  Set $P1$ to first token position.
  Set $P2$ to second token position.
else {count_token is > 2}
  if *P went through step 2* ***then***
    Determine cardinality of each token position.
    Determine the event size value with the highest frequency other than value 1 as $freq\_card$.
    if there is a tie for highest frequency value **then**
      Select lower token value as $freq\_card$
    end if
    if the frequency of $freq\_card > 1$ **then**
      Set $P1$ to first token position with cardinality $freq\_card$.
      Set $P2$ to second token position with cardinality $freq\_card$.
    else {the frequency of $freq\_card = 1$}
      Set $P1$ to first token position with cardinality $freq\_card$.
      Set $P2$ to first token position with the next most frequent cardinality other than value 1.
    end if
  else {*P is from Step 1*}
    Set P1 to first token position with lowest cardinality.
    Set P2 to second token position with lowest cardinality or first token position with the second lowest cardinality.
  end if
end if
{Cardinality of P1 can be equal to cardinality of P2}
Return((P1,P2))

Table 15: Anova Results on Precision Performance

HPC						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.92	2	0.50	2133.74	1.9E-30	3.3541
Within Groups	0.00	27	0.00			
Total	0.1822	29				
SYSLOG						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.05	2	0.02	30.96	1.03E-07	3.3541
Within Groups	0.02	27	0.00			
Total	0.07	29				
WINDOWS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	1.55	2	0.78	1343.19	19.35E-28	3.3541
Within Groups	0.02	27	0.00			
Total	1.57	29				
ACCESS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	2.56	2	1.28	318.62	1.67E-19	3.3541
Within Groups	0.11	27	0.00			
Total	2.67	29				
ERROR						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	5.08	2	2.54	603.31	3.91E-23	3.3541
Within Groups	0.11	27	0.00			
Total	5.20	29				
SYSTEM						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	5.54	2	2.77	9.72E+33	0	3.3541
Within Groups	7.7E-33	27	2.85E-34			
Total	5.54	29				
REWRITE						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	6.54	2	3.27	348431.5	2.76E-60	3.3541
Within Groups	0.00	27	9.39E-06			
Total	6.54	29				

Table 16: Anova Results on Recall Performance

HPC						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.08	2	0.04	35.35	2.89E-08	3.35
Within Groups	0.03	27	0.00			
Total	0.11	29				
SYSLOG						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.00	2	0.00	0.13	0.88	3.35
Within Groups	0.07	27	0.00			
Total	0.07	29				
WINDOWS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.01	2	0.01	3.88	0.03	3.35
Within Groups	0.037	27	0.00			
Total	0.05	29				
ACCESS						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.20	2	0.10	165.13	7.21E-16	3.3541
Within Groups	0.02	27	0.00			
Total	0.22	29				
ERROR						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.01	2	0.00	21.09	3.05E-06	3.3541
Within Groups	0.00	27	0.00			
Total	0.01	29				
SYSTEM						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	1.68	2	0.84	1.83E+32	0	3.35
Within Groups	1.24E-31	27	4.58E-33			
Total	1.68	29				
REWRITE						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	0.54	2	0.27	13812.74	2.29E-41	3.35
Within Groups	0.00	27	1.95E-05			
Total	0.54	29				