Clustering Event Logs Using Iterative Partitioning

Adetokunbo Makanju
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia
B3H 1W5, Canada
makanju@cs.dal.ca

A. Nur Zincir-Heywood
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia
B3H 1W5, Canada
zincir@cs.dal.ca

Evangelos E. Milios
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia
B3H 1W5, Canada
eem@cs.dal.ca

ABSTRACT

The importance of event logs, as a source of information in systems and network management cannot be overemphasized. With the ever increasing size and complexity of today's event logs, the task of analyzing event logs has become cumbersome to carry out manually. For this reason recent research has focused on the automatic analysis of these log files. In this paper we present IPLoM (Iterative Partitioning Log Mining), a novel algorithm for the mining of clusters from event logs. Through a 3-Step hierarchical partitioning process IPLoM partitions log data into its respective clusters. In its 4th and final stage IPLoM produces cluster descriptions or line formats for each of the clusters produced. Unlike other similar algorithms IPLoM is not based on the Apriori algorithm and it is able to find clusters in data whether or not its instances appear frequently. Evaluations show that IPLoM outperforms the other algorithms statistically significantly, and it is also able to achieve an average F-Measure performance 78% when the closest other algorithm achieves an F-Measure performance of 10%.

Categories and Subject Descriptors

I.5.3 [**Pattern Recognition**]: Clustering—*algorithms*

General Terms

Algorithms, Experimentation

Keywords

Event Log Mining, Fault Management, Telecommunications

1. INTRODUCTION

In today's ever changing digital world, virtually all computing systems are designed to log information about their operational status, environment changes, configuration modifications and errors into an event log of some sort (e.g. syslog or an application log). For this reason event logs have become an important source of information about the health or operational status of a computing infrastructure and is relied on by system, network and security analysts as a major source of information during downtime or security incidents.

However the size of event logs have continued to grow with the ever-increasing size of today's computing and communication infrastructure. This has made the task of reviewing event logs both cumbersome and error prone to be handled manually. Therefore automatic analysis of log files has become an important research problem that has received considerable attention [12, 16].

Due to the fundamental nature of event logs, the problem of finding *frequent event type patterns* has become an important topic in the field of automatic log file analysis [4, 8, 18]. Specifically, in system log files, records are usually expected to contain a timestamp, a source and a message as defined by the syslog RFC (Request for Comments)[6]. Moreover, a similar pattern is also applicable to different application log files where this time the message will be defined in the corresponding RFCs. However, in most cases, the description in the RFC will be without an explicit reference to an event type. Fortunately events of the same type are produced using the same line pattern in their *message* portion and these line patterns correspond to event types. So far techniques for automatically mining these line patterns from event logs have mostly been based on the Apriori algorithm for frequent itemsets from data, e.g. SLCT (Simple Logfile Clustering Tool) [14] and Loghound [15], while others have adopted other line pattern discovery techniques like Teiresias to the domain [12].

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 hierarchical clustering 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 differs from other event type mining algorithms for the following reasons: It is not based on the Apriori algorithm, which is computationally inefficient for mining longer patterns as shown in previous literature [3]. It is able to discover clusters irrespective of how frequently pattern instances appear in the data. The use of a pattern support threshold, which is mandatory for other similar algorithms, is optional for IPLoM, running IPLoM without a pattern support threshold provides the possibility that all potential clusters will be found. In our experiments we compared IPLoM, SLCT,

KDD'09, June 28–July 1, 2009, Paris, France.

Loghound and Teiresias on 7 different event log files, which were manually labeled by our faculty's tech support group. Results show that IPLoM consistently outperforms the other algorithms and achieves an average (across the datasets) F-Measure of 78% whereas the second best algorithm (SLCT) achieves an average F-Measure of 10%.

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

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 data sets, 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 job of clustering an event log 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 [14], 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. Event log lines are high dimensional. Traditional clustering algorithms are not well suited for high dimensional data.

4. Traditional clustering algorithms also tend to ignore the order of attributes. In event logs the attribute order is important.

While several algorithms like CLIQUE, CURE and MAFIA have been designed for clustering high dimensional data[1, 14], 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, ignore the order of the input records and discover clusters that exist in subspaces of the high dimensional data [1, 14].

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 [5] the authors attempt to classify raw event logs into a set of categories based on the IBM CBE (Common Base Event) format [2] using Hidden Markov Models (HMM) and a modified Naive Bayesian Model. They were able to achieve 85% and 82% classification accuracy respectively. While similar, the automatic categorization done in [5] is not the same as discovering event log clusters or formats. This is because the work done in [5] 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 [14] and Loghound [15] 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, they require the user to provide a support threshold value as input. This support threshold is not only used to control the output of these algorithms but is fundamental to their internal mechanism.

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 [13], as part of the LogView log visualization tool [9] and in online failure prediction [11]. Fig. 1 shows four examples of the type of clusters that SLCT and Loghound are able to find, the asterisks in each line indicate placeholders that can match any word. We will adopt this cluster representation in the rest of our work.

A comparison of SLCT against a bio-informatics pattern discovery algorithm developed by IBM called Teiresias is carried out in [12]. 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 at least a specificity determined by L/W, where L is the number of non-wildcard characters from C and W is the width of the motif with wildcards 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 adjudged to work just as effectively as SLCT by the author it was found not to scale efficiently to large data sets.

```
Feb * * big sshd[*]: Invalid user * from *
* - - [* -0500] "GET *
Oct * * big pop3d: LOGOUT, ip=[*]
* * * big pop3d: Connection, ip=[*]
```

Figure 1: Sample clusters generated by SLCT and Loghound

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 show 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 how our proposed algorithm works after which we describe in detail our methodology for testing its performance against those of pre-existing 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.

The four steps of which IPLoM goes through are:

1. Partition by token count.
2. Partition by token position.
3. Partition by search for bijection.
4. Discover cluster descriptions/line formats.

The steps are described in more detail below. The algorithm is designed to discover all possible line formats in the initial set of log messages. As it may be sometimes required to find only line formats that have a support that exceeds a certain threshold, the file prune function is incorporated into the algorithm. The file prune function works by getting rid of all partitions that fall below the file support threshold value at the end of each partitioning step. This way, we are able to produce only line formats that meet the desired file support threshold at the end of the algorithm. Running IPLoM without a file support threshold is its default state.

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

3.2 Step 1: Partition by token count.

The first step of the partitioning process works on the assumption that log messages that have the same line format are likely to have the same token length. For this reason

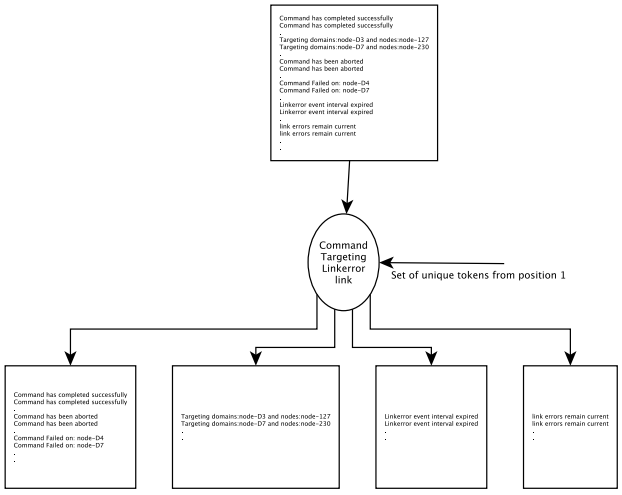


Figure 2: IPLoM Step-2: Partition by token position.

IPLoM's first step uses the token count heuristic to partition the log messages. Additional heuristic criteria are used in the remaining steps to further partition the initial partitions. 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 token count we are taking advantage of the property of most cluster instances of having the same token length, therefore the resultant partitions of this heuristic are likely to contain the instances of the different clusters which have the same token count. A detailed description of this step of the algorithm can be found in [10].

3.3 Step 2: Partition by token position.

At this point each partition of the log data contains log messages with the same length and can therefore be viewed as n-tuples, with n being the token length 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 that are constant in that position of the line formats 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. 2. A detailed description of this step of the partitioning process is outlined in [10].

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 line formats. 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 1). 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.

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 described in detail in Algorithm 2. A summary of the heuristic would be to select the first two token positions with the most frequently occuring token count 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. 3 the tokens *Failed* and *on:* have a 1-1 relationship because all lines that contain the token *Failed* in position 2 also contain the 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 line format) or constant values (so each value actually represents a different line format). The diagram in Fig. 4 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.

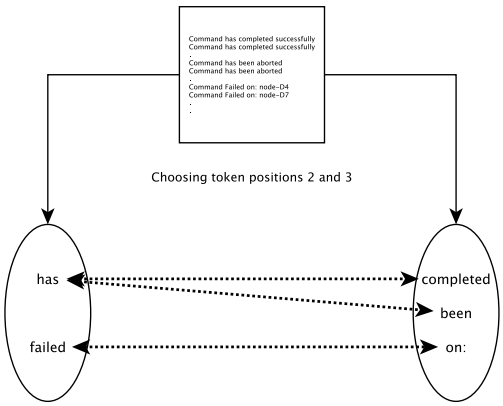


Figure 3: IPLoM Step-3: Partition by search for bijection.

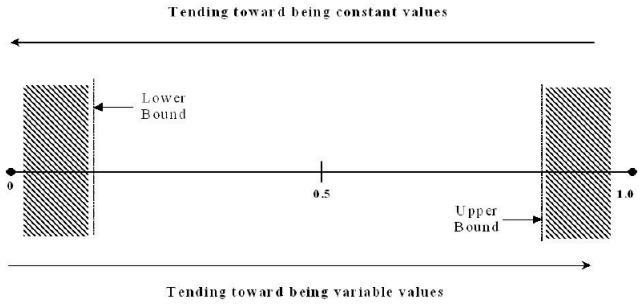


Figure 4: Deciding on how to treat 1-M and M-1 relationships.

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 total token length of the lines in the partition. Partitions that have a value higher than the cluster goodness threshold are considered good partitions and are not partitioned any further in this step.

3.5 Step 4: Discover cluster 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 cluster description or line format consists of a line of text where constant values are represented literally and variable values are represented using wildcard 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. 5.

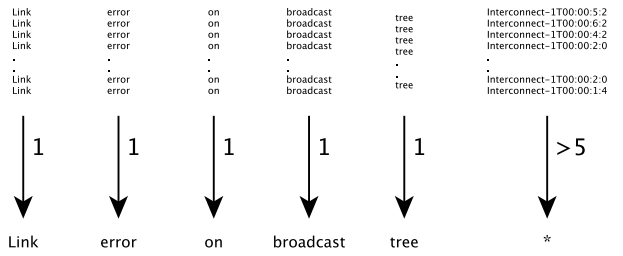


Figure 5: IPLoM Step-4: Discover cluster descriptions. The unique token counts are beside arrows

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.

- **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.
- **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 $0.3 - 0.6$.

Sensitivity analysis performed to evaluate the stability of IPLoM using different values for the parameters shows little deviation in performance.

3.7 Experiments

In order to evaluate the performance of IPLoM, we selected open source implementations of algorithms, which had previously been 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 datasets used. The HPC log file is an open source data set collected on high performance clusters at the Los Alamos National Laboratory NM, USA [7]. The Access, Error, System and Rewrite datasets 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.

Tech-Support members of the Dalhousie Faculty of Computer Science produced the line format cluster descriptions of these 7 datasets manually. Table 1 gives the number of clusters identified in each file manually. Again due to privacy issues we are unable to provide manually produced cluster descriptions of the non-opensource log files but the manually produced cluster descriptions for the HPC data are available for download [1]. These cluster descriptions then became our gold standard, against which to measure the performance of the other algorithms as an information retrieval (IR) task. As in classic IR, our performance metrics are Recall, Precision and F-Measure, which are described in [17]. The True Positive(TP), False Positive(FP) and False Negative(FN) values were derived by comparing the set of manually produced line formats to the set of retrieved formats produced by each algorithm. In our evaluation a line format is still considered a FP even if matches a manually produced line format to some degree, the match has to be exact for it to be considered a TP. The next section gives more details about the results of our experiments.

4. RESULTS

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 Table 2. The rationale for choosing the support values used for SLCT, Loghound and IPLoM is explained later in this section. 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 all set intuitively except in 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. The partition support threshold was set to 0 to provide a baseline performance. 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 [12]. Thus in this work, it was only tested against the Syslog data set.

Table 2: Algorithm Parameters

SLCT and Loghound Parameters	**Value**
Support Threshold (-s)	0.01 - 0.1
Seed (-i)	5
Teiresias Parameters	**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
IPLoM Parameters	**Value**
File Support Threshold	0 - 0.1
Partition Support Threshold	0
Lower Bound	0.1
Upper Bound	0.9
Cluster Goodness Threshold	0.34

[1]http://torch.cs.dal.ca/~makanju/iplom

Table 1: Log Data Statistics

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

Table 3: Anova Results for F-Measure Performance

HPC	F	P-value	F crit
	8.06	1.8E-02	3.35
SYSLOG	**F**	**P-value**	**F crit**
	2.00	0.15	3.35
WINDOWS	**F**	**P-value**	**F crit**
	23.39	1.28E-06	3.35
ACCESS	**F**	**P-value**	**F crit**
	455.90	1.56E-21	3.35
ERROR	**F**	**P-value**	**F crit**
	50.57	7.41E-10	3.35
SYSTEM	**F**	**P-value**	**F crit**
	1.96E+34	0	3.35
REWRITE	**F**	**P-value**	**F crit**
	51076.72	4.98E-49	3.35

SLCT, Loghound and Teiresias cannot produce clusters if a line support threshold is not provided. This makes it difficult to compare with IPLoM's default output. Also, the concept of support threshold as used in Teiresias was not implementable in IPLoM due to its use of a specificity value i.e. L/W. For this reason we compare IPLoM against the other algorithms using 2 different scenarios.

In the first scenario we use a range of low support values (intuitively the algorithms should produce more clusters with lower support values) against IPLoM, SLCT and Loghound. The F-Measure results of this scenario are shown in Fig 6, the results clearly show IPLoM performing better than the other algorithms in all the tasks. A single factor ANOVA test done at 5% significance on the results show a statistically significant difference in all the results except in the case of the Syslog file, Table 3 provides a summary of these results. Similar results for Recall and Precision can be found in [10].

In the second scenario we compare the default performance of IPLoM against the best performance of SLCT, Loghound and Teiresias (we used the lowest support value possible for Teiresias). 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. 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 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. 7.

So in scenario 2 we provide the comparison of the results of IPLoM's cluster description output and its partition output, as shown in Fig. 8. The partition comparison differs from the cluster description by including as correct cases where

Table 4: Log Data Token Length 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

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.

However, as stated in [3], in cases where data sets have relatively long patterns or low minimum support thresholds are used, apriori based algorithms suffer non-trivial costs during candidate generation. The token length statistics for our datasets are outlined in Table 4, this shows the *HPC* file as having the largest maximum and average token length. Loghound was unable to produce results on this dataset with a line count support value of 2, the algorithm crashed due to the large number of item-sets that had to be generated. This was however not a problem for SLCT (as it generates only 1-item-sets). This results show that Loghound is still vulnerable to problems outlined in [3], this is however not a problem for IPLoM as it is computational complexity is not adversely affected by long patterns or low minimum support thresholds. In terms of performance based on cluster token length, Table 5 shows consistent performance from IPLoM irrespective of token length, while SLCT and Loghound seem to suffer for mid-length 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 criteria is outlined in Table 6. The results show a reduction in performance for all the algorithms for clusters with a few instances, however

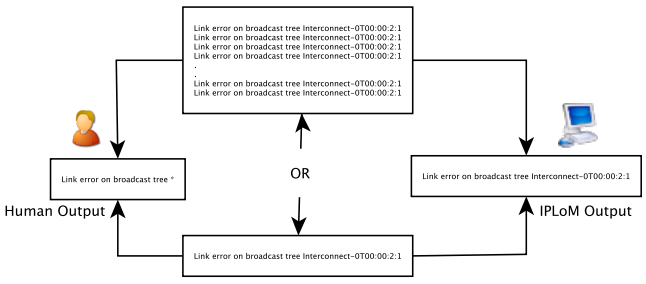
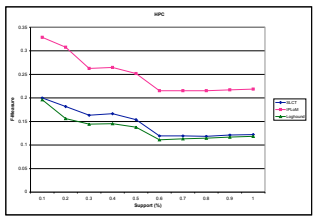
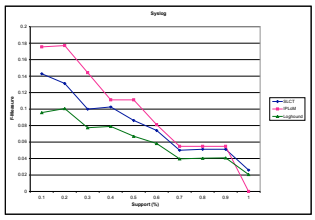


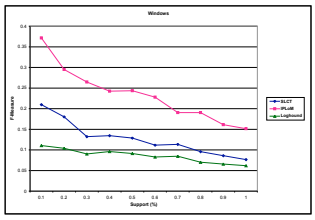
Figure 7: Example: Right Partition, Wrong Cluster Description.



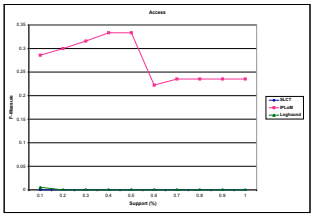
(a) HPC



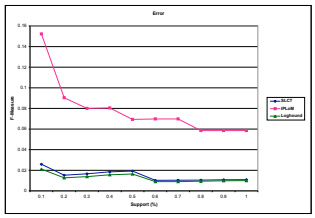
(b) Syslog



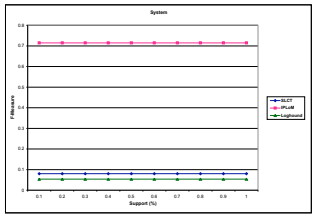
(c) Windows



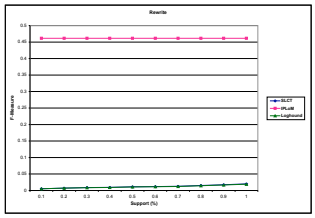
(d) Access



(e) Error

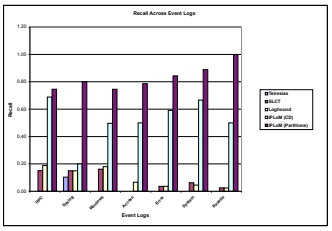


(f) System

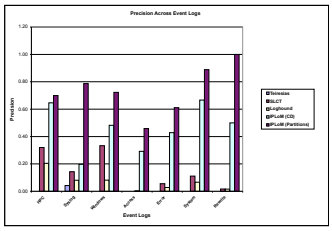


(g) Rewrite

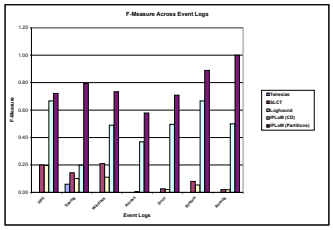
Figure 6: Comparing F-Measure performance of IPLoM, Loghound and SLCT using support values.



(a) Recall



(b) Precision



(c) F-Measure

Figure 8: Comparing performance of default IPLoM output with best case performances of Teiresias, Loghound and SLCT.

Table 5: Algorithm performance based on cluster token length

Token Length 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

Table 6: 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

IPLoM's performance was more resilient. The results used in our token length and cluster instance frequency evaluations are based on cluster description formats only.

5. CONCLUSION AND FUTURE WORK

In this paper we introduce IPLoM, a novel algorithm for the mining of event log clusters. Through a 3-Step hierarchical partitioning process IPLoM partitions log data into its respective clusters. In its 4th and final stage IPLoM produces cluster descriptions or line formats for each of the clusters produced.

We implemented IPLoM and tested its performance against the performance of algorithms previously used in the same task i.e. SLCT, Loghound and Teiresias. In our experiments we compared the results of the algorithms against results produced manually by human subjects on seven different data sets. The results show that IPLoM has an average (across the data sets) Recall of 0.81, Precision of 0.73 and F-Measure of 0.76. It is also shown that IPLoM demonstrated statistically significantly better performance than either SLCT or Loghound on six of the seven different data sets. Future work will focus on using the results of IPLoM i.e. the extracted cluster formats, in other automatic log analysis tasks.

Acknowledgements

This research is supported by the NSERC Strategic Grant and network, ISSNet. 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/.

6. REFERENCES

[1] 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*, pages 55 – 62, New York, NY, USA, 2007. ACM.

[2] B. T. et. al. Automating problem determination: A first step toward self healing computing systems. IBM White Paper, October 2003.

[3] 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*, pages 1–12, 2000.

[4] M. Klemettinen. *A Knowledge Discovery Methodology for Telecommunications Network Alarm Databases*. PhD thesis, University of Helsinki, 1999.

[5] 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*, pages 776–781, 2005.

[6] C. Lonvick. The BSD syslog protocol. RFC3164, August 2001.

[7] L. Los Alamos National Security. Operational data to support and enable computer science research. Published to the web, January 2009.

[8] S. Ma, , and J. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of the 16th International Conference on Data Engineering*, pages 205–214, 2000.

[9] 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*, pages 99 – 108, October 2008.

[10] A. Makanju, N. Zincir-Heywood, and E. E. Milios. Iplom: Iterative partitioning log mining. Technical report, Dalhousie University, February 2009.

[11] F. Salfener and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *26th IEEE International Symposium on Reliable Distributed Systems.*, pages 161–174, 2007.

[12] J. Stearley. Towards informatic analysis of syslogs. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 309–318, 2004.

[13] J. Stearley. Sisyphus log data mining toolkit. Accessed from the Web, January 2009.

[14] 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)*, pages 119–126, 2003.

[15] R. Vaarandi. 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)*, volume 3283, pages 293–308, 2004.

[16] R. Vaarandi. Mining event logs with slct and loghound. In *Proceedings of the 2008 IEEE/IFIP Network Operations and Management Symposium*, pages 1071–1074, April 2008.

[17] Wikipedia.org. Precision and Recall - Wikipedia, the free encyclopedia. Published to the web, http://en.wikipedia.org/wiki/Precision_and_Recall. Last checked April 23, 2009.

[18] Q. Zheng, K. Xu, W. Lv, and S. Ma. Intelligent search for correlated alaram from database containing noise data. In *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 405–419, 2002.

Algorithm 1 Partition_Prune Function

Input: Collection $C[]$ of log ↓le partitions.
Real number PS as partition support threshold.
Output: Collection $C[]$ of log ↓le 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)

Algorithm 2 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 $Temp\_C$
  DetermineP1andP2(P_In){See Algorithm 3}
  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 4.}
      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 1}
  Add all partitions from $Temp\_C$ to C_Out
end for
$C\_Out = File\_Prune(C\_Out)$ {See Algorithm ??}
Return(C_Out)

Algorithm 3 Procedure DetermineP1andP2

Input: Partition P.
Real number CT as cluster goodness threshold.

Determine token count 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 5}
  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 4 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 ↓le 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)

Algorithm 5 Get_Mapping_Positions Function

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

Determine token count of P as $count\_token$
if $count\_token = 2$ **then**
  Set $P1$ to ↓rst 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 token count 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 ↓rst 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 ↓rst token position with cardinality $freq\_card$.
      Set $P2$ to ↓rst 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))