Stream-Based Event Prediction Using Bayesian and Bloom Filters

Miao Wang
Performance Engineering Lab
School of Computer Science and Informatics
University College Dublin, Ireland
miao.wang@ucdconnect.ie

Viliam Holub
Performance Engineering Lab
School of Computer Science and Informatics
University College Dublin, Ireland
viliam.holub@ucd.ie

John Murphy
Performance Engineering Lab
School of Computer Science and Informatics
University College Dublin, Ireland
j.murphy@ucd.ie

Patrick O'Sullivan
Collaboration Solution System
IBM Software Group
Dublin, Ireland
patosullivan@ie.ibm.com

ABSTRACT

Nowadays, enterprise software systems store a large amount of operational information in logs. Manually analysing these data can be time-consuming and error-prone. Although a static knowledge database eases the task to capture recurring problems, maintaining such a knowledge repository requires periodic knowledge updates by domain experts. Moreover, as the repository grows, the problem of memory efficiency will also arise.

Our goal is to enable administrators to efficiently capture interesting data in a high volume stream of events in real-time. We are proposing a statistical approach for software applications to be automatically trained with a smaller dataset to efficiently predict interesting data under such conditions. The proposed solution maintains a stable memory usage by migrating keywords from a dynamic data structure to fixed sized data structures (Bloom Filter). In particular, the solution has achieved better prediction results by enhancing the Bayesian theory with belief modifiers.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information filtering; H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Algorithms, Design, Performance

Keywords

Stream Event Prediction, Statistical, High Performance

1. INTRODUCTION

Over the last decade, computers have been fast developed with many advances in hardware technologies from powerful servers to lightweight handsets. The highly developed technologies have also caused the amount of computer data to rapidly grow such that gigabytes of data are now very common. In particular, data streams have gained a great popularity for many real-life use cases such as video/voice streaming, distributed file sharing, and sensor networks. However, as the information contained in a data stream would continuously flow through the system, the data processing algorithms need to be designed as efficiently as possible to avoid data to be queued in the network. As a result, when dealing with a large volume of stream data, processing the same piece of information multiple times is not feasible, instead, single pass techniques are often preferred. Furthermore, data streams can also lead to other problems in relation to limited CPU and memory resources [1].

A common approach to recognise recurring problems and interesting data is to use a set of patterns and rules maintained in a static knowledge database, which must be defined prior to the data matching process [12, 14, 31]. However, most of pattern-based approaches require domain expert knowledge and extra maintenance costs for periodically updates [4], of which tasks are often manually carried out [26]. In summary, when dealing with a high volume of stream events, which are single raw text messages such as log records, a pattern-based approach would suffer following problems: a) Can not easily be extended to recognise new patterns; b) If the size of patterns keep growing, there can be too much information saved in the memory repository; c) A disk-based pattern repository can cause significant performance overhead in the pattern detection process due to much slower data access speed caused by disk I/O operations. As a result, an automatic way to detect problematic and interesting events has been strongly required by system administrators [10].

The statistical approach [10, 19, 26, 27] is dynamic and enables a software application to be automatically trained with a fraction of the dataset to capture interesting data, which may not even have been seen before. The focus of this paper is to discuss the statistical techniques to automatically predict interesting events through a raw event stream by assuming no dependency exists between each event and also no prior knowledge on incoming events either.

The bag-of-words is a traditional strategy to save training keywords and corresponding statistics in the memory [28]. In particular, the tree data structure allows keywords with the same prefixes to share a single branch, which is more memory efficient than saving each individual keyword separately. However, a tree has to be expanded for each distinct keyword, in which case the size of the tree can easily grow up depending on the number of distinct keywords and their length. As a result, using the tree data structure can potentially carry a risk of the memory-intensive problem, especially in a large-scale heterogenous environment such as Cloud [15], where a variety of software applications can work together and produce a high volume of dynamic data contents such as transaction IDs and session IDs.

Furthermore, we are proposing a statistical approach to predict interesting events through a dynamic stream events. Our approach has been built top of Bayesian and Bloom filter techniques with the ability to predict the interesting events through a large volume of stream data produced by enterprise software applications. In particular, this paper contains following contributions:

1. Using normalised probabilities to avoid the lowest and highest probabilities especially for the zero probability to cause inaccurate prediction results.

2. Allowing the memory to be effectively maintained by periodically migrating keywords from a dynamic data structure (Tree) to fixed sized data structures (Bloom Filter), which are created based on minimal and maximal probabilities. In particular, such a keyword migration does not require the quality of prediction results to be sacrificed.

3. Proposing the belief modifier to improve the accuracy of the Bayesian filtering technique.

4. The proposed solution enables both event training and prediction processes to deliver a high system throughput to cope with a large volume of stream events.

5. In addition, we are able to show that the proposed event prediction mechanism can also be used to discover interesting data through static documents.

2. BACKGROUND AND MOTIVATIONS

Log data analysis is an important task for administrators to understand the behaviour of software applications. However, due to the amount of information, manually performing such a task is time-consuming and thus ineffective. To ease the log analysis tasks, Run Time Correlation Engine (RTCE) [11] has been developed as a framework to automatically collect large amounts of log data from different application logs, and correlate them together to provide a single view to monitor all application instances. RTCE was designed to handle generic data types and developed in conjunction with our industrial partner (IBM Dublin Software Group), where RTCE has been successfully deployed by their testing teams. In particular, many teams have achieved significant time saving by using RTCE to automate their log analysis tasks [30] for variants of real industrial applications. In this paper, we are using the RTCE framework, which contains various log adapters to automate event conversion tasks for enterprise application logs.

The current success of RTCE has been mainly built based on the efficient run-time data analysis tasks such as the symptom matching algorithm [31], which enables the XPath-based evaluation engine to efficiently process a large set of XPath queries to match against stream events. However, maintaining a large symptom database, which contains hundreds of thousands of entries is not an easy task, which requires domain expert knowledge for each enterprise application and frequent data updates. To ensure our research work is able to solve more real-life problems for real industries, we have also carried our regular weekly meetings with IBM System Verification Test (SVT) team managers and experts to discuss further improvements to ease their testing experiences. As a result, we have decided to discover a new approach to enable their daily tasks to maintain the knowledge database to be automated as well as automating the problem detection process. The goal is to enhance the current RTCE framework with additional automated functionality for testing teams to gain more time saving on system monitoring related tasks.

3. EVENT PREDICTION SYSTEM

In order to support efficient event training and prediction processes for timely data analysis without sacrificing the quality of result, we have developed a new stream processing engine consisting of following components:

- **Event Trainer**: continuously breaks incoming events into a set of keywords and sends these keywords with statistics to appropriate data structures.

- **Tree**: saves keywords and statistics, which have not rarely or popularly occurred.

- **Bloom Filter (BlmF)** [6]: maintains two bit arrays separately such that one for rare keywords and the other one for popular keywords.

- **Event Predictor**: continuously examines the probability of the belief for each incoming event to find interesting ones with strong believes.

- **Bayesian Filter (BayeF)** [24]: calculate the overall probability of the belief for a set of keywords using the tree and BlmFs.

The purpose of distinguishing rare and popular keywords from the normal ones is to reduce the memory overhead caused by the tree. Typically a rare keyword is a set of distinct characters from other keywords. To save such a keyword, the tree data structure has to be expanded with a new branch and leaf, which will cause the size of tree grows up. On the other hand, the popular keywords are the ones that can be migrated to reduce the performance overhead for frequent calculations, which are mostly repeated. Furthermore in addition to the core system components, we have also used following thresholds to ease the event training and prediction process:

- **Minimal probability** (P_{min}): The minimal frequency of a keyword that could possibly occur in interesting events. This value has to be normalised to be greater than zero.

- **Maximal probability** (P_{max}): The maximal frequency of a keyword that could possibly occur in interesting events. This value has to be normalised to be less than one.

- **Strong threshold** (ST): A probability threshold to decide if an event is interesting such that the overall probability of the event has to be greater than or equal to this value.

- **Strong belief modifier** (T_s): The probability of how likely an event can be believed interesting by a given keyword, which has occurred in interesting events.

- **Weak belief modifier** (T_w): The probability of how likely an event can be believed interesting by a given keyword, which has not occurred in interesting events.

3.1 Event Trainer Overview

The event training engine is responsible for constructing a repository of keywords and corresponding statistics of their occurrences. The aim to allow a high volume of stream events to be efficiently trained without sacrificing the quality of the repository for event prediction during run-time.

As shown in Figure 1, when an event arrives in the system, the event training will use the *'Keyword Extractor'* to convert the message body into a list of keywords. Then the extracted keywords will be added to the *'Tree'* data structure and the statistics of each keyword will also be updated on associated leaf nodes. Once the tree is fully updated, all the updated and new leaf nodes will be sent to the *'Threshold Examiner'*, which periodically checks if any keyword has occurred with a rare or popular frequency, which has been calculated as:

$$p = Freq_{(interesting)} = \frac{occurrence\ in\ interesting\ events}{the\ total\ occurrence}$$

(In this example, we use P_{min} as the rare frequency and P_{max} as the popular frequency for rare and popular keywords, which have 1% and 99% probabilities to occur in interesting events respectively while both values can be re-adjusted depending on the actual use cases.)

If any leaf node with a rare or popular frequency is found, the *'TreeNode Extractor'* will find the corresponding keyword from the *'Tree'* using the parent reference contained in each tree node. During the keyword extraction, all the standalone tree nodes will be discarded to reduce the memory usage of the tree. Thereafter, the *'TreeNode Extractor'* will send rare and popular keywords to the rare and popular BlmF processors respectively.

During the training phase, a user has to define the pattern to select interesting events depending on the actual datasets. For example, the keyword *'spam'* would allow the *Event Trainer* to select all spam messages as interesting events. The purpose is to allow our system to be trained with a little user input (common keywords) to automatically predict future events.

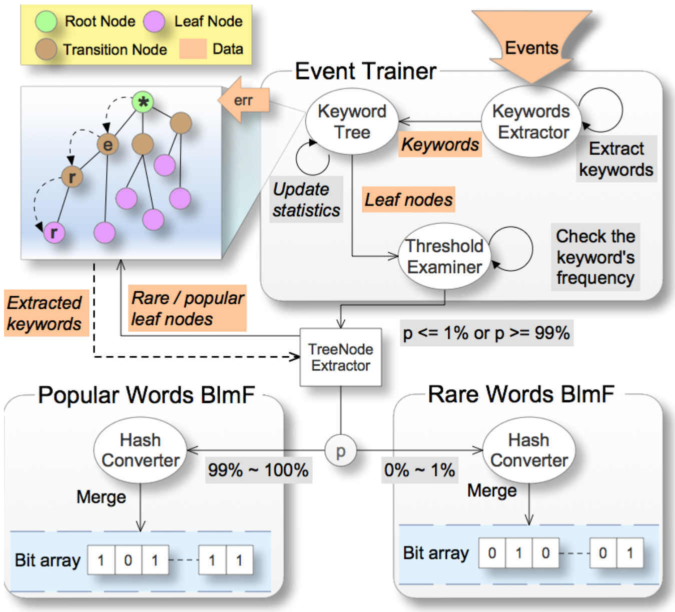

Figure 1: Stream-Based Event Training System.

3.2 Tree Structure

The tree structure (Figure 1) consists of a static root node and a list of leaf and transition nodes. New transition nodes are created when none of the existing nodes can form an incoming keyword. The last character of a keyword always settles on a leaf node. In the case of common strings being shared by multiple keywords, we allow all existing nodes to be reusable for them. As a result, a transition node can also become a leaf node if one keyword string is a subset of another keyword. As shown in the graph, the leaf node *'r'* will be reused as a transition node, if the keyword *'error'* comes. Furthermore, a path traversal between a leaf node and the root node can always precisely form a previously processed keyword. Consequently, each keyword's statistics such as the frequency occurring in interesting events and overall occurrence only need to be saved on their corresponding leaf nodes. For our current implementation, the statistics are collected as integers with a maximal limit of $2^{63} - 1$, which we expect to work for most situations.

During the tree node extraction phase, the engine will iterate all leaf nodes that have rare and popular occurrences in interesting events. From each leaf node the engine will traverse back to the root node in order to form the corresponding keyword for BlmFs. During the tree traversal the engine will also discard the current leaf node and all other transition nodes, unless any of them have been either linked by other descendant nodes or used as the leaf nodes by other keywords. By performing such a keyword migration process, the memory usage for the tree data structure can be greatly saved and the effectiveness is shown in the experiment section 4.2.

Furthermore, the rare and popular thresholds are defined as the lower and upper bounds of keywords' occurrence that core engine calculates. In this paper, we mainly use percentages for statistical calculations thus 1% and 99% have been used as the thresholds for determining rare and popular keywords. By defining the lower and upper bounds of keywords' occurrence, we are also able to assign fixed probabilities in the BlmFs during keyword lookups to avoid unnecessary calculations for a better performance.

From the memory consumption perspective, there are also other competitive techniques available. For example, keywords can be converted into hash values and saved in a hash table; A list of keywords can also be compressed using various data compression techniques. Although both approaches would allow keywords to be stored with a reduced memory consumption than the tree data structure, during the keyword lookup process the tree data structure would become more optimal. Because a hash table has to iterate through a list of records to search for a given keyword and results in longer response time if the number of records is large. By using data compression techniques, keywords need to do the data decompression, which will introduces extra performance overhead. Comparing other techniques, a key feature of using the tree data structure as the main knowledge repository to save keywords along with their occurrences is because of its efficient keyword lookups, which allows a given keyword to be searched through a large set of keywords via a single pass.

3.3 Bloom Filter

The BlmF technique has the advantage of space efficiency by encoding and merging each keyword into a fixed sized bit array. When a BlmF stores a keyword, it encodes the keyword into hash values (a bit array with 0 and 1 values) using a hash function and then merges the result into a predefined bit array. Then when a BlmF checks a keyword's existence, it has to convert the keyword into a hash value and check if all 1's in the keyword's hash value can be matched to the 1's in the BlmF's bit array. If all 1's are matched, then it means the BF has previously stored the same keyword in the bit array. Otherwise, the keyword has not been stored. As all keywords are saved together, it is impossible to assign or fetch statistics for individual keywords from the BlmF's bit array. Instead when a rare or popular keyword is found, the BlmF processor will simply return a fixed frequency of P_{min} or P_{max} accordingly.

We use the following function to calculate the size of the bit array:

$$m = -\frac{n \cdot \log f}{(\log 2)^2}$$

Where *'m'* is the number of bits required in the array, *'n'* is the total number of keywords going to be inserted into the bit array (the maximal capacity of the BlmF), and *'f'* is the false positive rate, which can be tolerated by this BlmF. (During the implementation, we have defined each BlmF, which consumes less than 1 MB memory space to allow maximal $100,000$ keywords to be saved by tolerating 0.01 false positive.) To efficiently convert each keyword into hash values, the MurmurHash2 [1] algorithm has been used.

Furthermore rare and popular keywords are migrated from the tree data structure based on their extreme occurrences, during the keyword lookup process the fixed occurrence values for BlmFs' keywords are immediately returned as the minimal and maximal likelihoods to believe an event to be interesting. By defining such normalised thresholds to represent the lower and upper bounds of likelihoods, we assume that the prediction process can tolerant little precision loss, in which case it might result in mismatching a small fraction of interesting events that have their overall likelihoods slightly below the predefined (ST). However, depending on the actual deployment environment, the optimal value of ST can be re-adjusted to either capture more mismatched interesting events by reducing ST or less non-interesting events by increasing ST.

[1]http://code.google.com/p/smhasher/wiki/MurmurHash2

The BlmF-based solution has been proposed as a space efficient technique for examining the keywords' existence. However, a BlmF has to risk certain chances of false positive, which causes a keyword to be falsely reported as being contained in the bit array. The false positive problem occurs because the 1's bits of a keyword are fully satisfied by the BlmF's bit array but in fact some 1's bits in the BlmF are set by storing other keywords' hash values rather than the current keyword. An example of such a problem is illustrated in Figure 2. As assuming the original bit array has not stored the hash value for neither *'Keyword A'*, *'Keyword B'*, nor *'Keyword C'*, when *'Keyword A'* arrives for the training purpose, all the corresponding 1's bits must be turned on within the original bit array. However, the new bit array produced after such a merging process might also contain all necessary 1's bits for other keywords, which have never been stored before. For example, during the querying phase, although both *'Keyword B'* and *'Keyword C'* have never been merged into the bit array, the hash value of *'Keyword C'* has just become fully satisfied after the new bit array has the 1's bits turned for both *'Keyword A'* and previous keywords. Consequently, the *'Keyword C'* would be falsely reported as a keyword, which has been theoretically stored in the bit array previously.

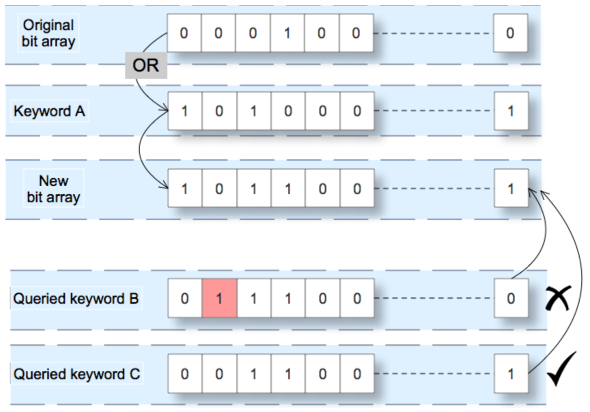

Figure 2: The False Positive Problem in Bloom Filter.

3.4 Event Predictor

The event prediction engine is responsible for evaluating the probability of each arrival event to be believed interesting based on previous keywords and statistics, which have been saved in the *'Tree'* and rare and popular BlmFs. The aim is to develop an efficient event prediction algorithm to capture interesting events through a high volume of stream events in real-time.

During the event prediction process, once a stream event arrives, the event prediction engine uses the Algorithm EP to calculate the overall probability of the event and determine how likely the event is believed to be interesting. Before an

Algorithm EP BayeF-based Event Prediction (*event*)

```
1: $T_s = 90\%$ {Strong belief modifier}
2: $T_w = 10\%$ {Weak belief modifier}
3: $ST = 90\%$ {Strong Threshold}
4: $P_{min} = 1\%$ {Minimal probability}
5: $P_{max} = 99\%$ {Maximal probability}
6: $k = 0$ {Keyword Counter}
7: for all keywords in event do
8:     if keyword exists on the tree then
9:         $p^t_k = \frac{occurrence\ in\ interesting\ events}{the\ total\ occurrence}$
10:    else if keyword exists in rare BlmF then
11:        $p^t_k = P_{min}$
12:    else if keyword exists in popular BlmF then
13:        $p^t_k = P_{max}$
14:    else {keyword is unknown}
15:        skip and continue with the next keyword
16:    end if
17:    $p_k = p^t_k \times T_s$
18:    $p'_k = (1 - p^t_k) \times T_w$
19:    $k = k + 1$
20: end for
21: if k is equal to 0 then
22:    return No keyword found
23: else
24:    $P = \frac{\prod_{i=1}^{k}(p_i)}{\prod_{i=1}^{k}(p_i) + \prod_{i=1}^{k}(p'_i)}$
25:    if P is greater than or equal to ST then
26:        return Event is interesting
27:    else
28:        return Event is not interesting
29:    end if
30: end if
```

event is processed, the engine will initialise a variable *'k'* as the counter for the number of processed keywords, which are recognisable within the training repository (See line: 6). Then for each arrival event, the engine will extract all the keywords from the message body (See line: 7) and collect the associated probability for each keyword (See line: $8-16$).

In order to reduce the chances of having false positive problems in both BlmFs, the execution order of the probability collection process needs to be properly arranged such that the engine always tries to find an extracted keyword from the tree first before using BlmFs. As a result, the keywords saved on the tree will not be falsely found from BlmFs. Thereafter, if a keyword is found on the tree, the corresponding probability for the current event to be believed interesting is going to be calculated based on the previous formula *'$Freq_{(interesting)}$'*. If the keyword can be found in the rare BlmF, a static probability will be assigned to the keyword using the value of P_{min}. Consequently, if a keyword can only be found in the popular BlmF, the P_{max} will be assigned to the keyword. However, if the keyword has not been found in neither *'Tree'* nor the two BlmFs, the keyword is considered neutral such that its probability will be skipped for the current event's probability calculation.

Furthermore for the best practice, the execution order between rare and popular BlmFs should be adjusted based on the number of merged keywords in their bit arrays such that the one with a smaller number is always executed before the other. As more keywords are merged into the bit array, more 1's bits will be turned on, in which case the chances of having false positive problems will be increased. Therefore when collecting the probability for each keyword, executing the BlmF with less merged keywords prior to the other can practically reduce the risk of false positive problems. Such an optimisation can be implemented before the *'for loop'* (line: 7) by assigning BlmF to two separate variables and then replacing line$10, 12$ with the BlmF variables containing less and more keywords respectively, while assigning the corresponding probability to p^t_k (line: $11, 13$).

Once a probability is successfully calculated for a keyword, the engine will use the Bayesian theory to calculate the likelihood statistics of the current event based on the processed keywords. The basic formula is defined as:

$$P = \frac{p_1 \cdot p_2 \cdot p_3 ... \cdot p_k}{p_1 \cdot p_2 \cdot p_3 ... \cdot p_k + p'_1 \cdot p'_2 \cdot p'_3 ... \cdot p'_k}$$

P is the probability of how likely the event is believed to be interesting, p_k is the probability of the keyword, which has occurred in interesting events to allow an event to be believed interesting, and p'_k is the probability of the keyword, which has not occurred in interesting events to allow an event to be believed interesting. If p^t_k is the probability of the keyword occurring in interesting events ($Freq_{(interesting)}$), then:

- $p_k = p^t_k \times T_s$: By applying the strong belief modifier to the frequency of the keywords occurring in interesting events, the probability of of an event believed to be interesting can be more realistically reflected, because a single interesting keyword should not make the event believed to be interesting with 100% certainty. Instead, an interesting keyword should only give a strong belief (90%) to the event for being interesting.

- $p'_k = (1 - p^t_k) \times T_w$: As opposite to p_k, p'_k is calculated based on a weak belief modifier to the frequency of the keyword occurring in non-interesting events. Although in this case the probability represents the chance of a keyword to occur in non-interesting events, the belief of such events to be interesting should not be absolutely impossible. Instead, a non-interesting keyword should give a weak belief (10%) to the event for being interesting.

As shown in the Algorithm EP, the engine updates the product over the overall probability from the $1st$ to current keywords with two realistic probabilities of p_k and p'_k (See line: $17-19$). When the engine finishes collecting all probabilities with the keyword counter *'k'*, the overall probability of the belief for the current event being interesting is calculated using the Bayesian theory as line 24. In this case, $\prod_{i=1}^{k}(p_i)$ is the probability of the event to be interesting based on keywords occurred in interesting events and $\prod_{i=1}^{k}(p'_i)$ represents how likely the event is interesting based on keywords occurred in non-interesting events. An the end of the algorithm, the engine will compare the overall probability *'P'* to the predefined threshold *'ST'* to determine whether the event is interesting (See line: $25-29$).

4. EXPERIMENT

The implementation of the proposed solution has been developed in Java and evaluated using a standard desktop machine[2]. During all experiment tests, the implemented application has been running with a default memory setting, which allows maximal 123MB to be consumed and using 20% $\sim$ 31% of the quad-core CPU on average. To clearly demonstrate the performance of our solution, the experiment has been broken down into 7 sub-sections.

In Section 4.1, we will explain the memory-intensive problem caused by the tree data structure. In Section 4.2, we will show the effectiveness of the keyword migration mechanism for stream-based event training. In Section 4.3, we will evaluate the quality of our solution from the following aspects:

- **False positive**: representing the percentage of the non-interesting events, which should not be captured out of the entire set of captured events. The value gives an impact on the amount of non-interesting events that an administrator needs to filter out from the predicted events.

- **False negative**: representing the percentage of the missing events out of the interesting events, all of which should be captured. The value gives an impact on the amount of interesting events that an administrator needs to filter out from the entire event stream.

As a high quality result, the value of both evaluation criteria should be kept low. In the case that only one of evaluation criteria can be maintained low, the false positive is always sacrificed to achieve a low false negative value. Because finding interesting data through a high volume of stream events is much more difficult and time-consuming than finding non-interesting data through a small fraction of the stream dataset.

In particular, Section 4.3 will also demonstrate the system throughput for stream-based event training and prediction processes in addition to the accuracy assessment. The aim is to show the applicability of applying our solution for real-time event prediction use cases.

In Section 4.4, we will show the effectiveness of the belief modifiers by comparing to the BayeF without belief modifiers. In Section 4.5, we will demonstrate the performance of RTCE before and after being integrated with our solution. The aim is to show the performance overhead introduced by our solution to the original RTCE framework. In Section 4.6, we will evaluate the quality of our solution by processing static data using two open source datasets. The aim is to show the applicability of our solution for various data types. In Section 4.7, we will compare three other classifiers against our approach.

4.1 Memory-Intensive Problem for Tree

During our experimental test, we use the B-Tree data structure to store indexing keywords. Each tree node consists of three possible fields: a word character in all tree nodes, a word frequency value on the leaf node, and a list of node transitions on the transition nodes. Such a tree data structure can achieve space efficiency for storing keywords with the same prefix on a single tree branch. In order to simulate unique keywords, our test data consists of a list of random UUIDs[3] (Universal Unique IDentifier [20]).

[2]Ubuntu 8.10, x86_64 GNU/Linux, J2RE 1.6.0 J9 2.4 Linux amd64-64, Intel Q9400 2.66GHz, 8GB RAM, Western Digital 7200rpm SATA-II

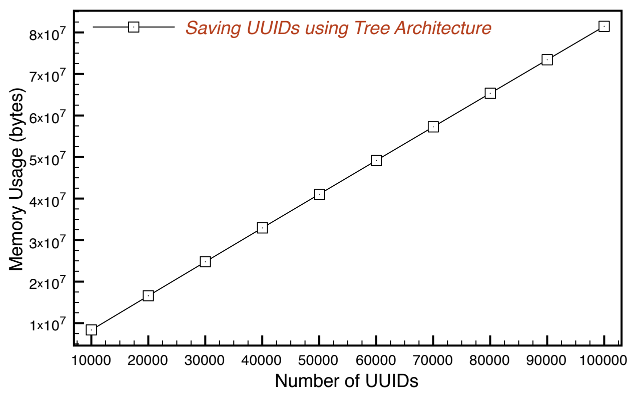

Figure 3: Memory consumptions for saving UUIDs in a tree architecture

Figure 3 illustrates the memory consumptions for storing different amounts of UUIDs in the tree data structure. As the number of stored UUIDs increases, the more memory are consumed by the tree. To be precise, the size of the tree is increased by 8 megabytes for every 10,000 UUIDs under our Java implementation. In the case that tens of thousands of UUIDs need to be processed, the memory exhaustion would become significant. In addition, the memory consumption shown in the graph is only for storing a series of UUIDs. If storing real log events, which may contain the UUID, timestamp, and message body for each log entry, then the size of the tree will consume even more memory space. Therefore, we would conclude that purely using the tree data structure to store the training keywords and statistics is not a feasible solution to deal with a large volume of stream events.

4.2 Keyword Migration for Event Training

As explained previously, processing a dynamic stream data can potentially cause the memory-intensive problem. Beyond the solution of upgrading machines with more physical memory chips, we have proposed a new approach to migrate rare and popular keywords into two fixed sized BlmFs. The effectiveness of our approach is shown in Figure 4.

During the experiment, the implemented application has been constantly trained with a set of log events collected from the IBM SVT lab environment. The overall dataset consists of $27m$ lines of events and $2.5g$ in size. The number of processed keywords is $62k$, which consequently require $563k$ tree nodes to be created in total. The number of tree nodes is periodically examined against the pre-defined threshold of $100k$ to trigger the keyword migration process, which has caused the number of keywords to be dramatically reduced from the tree as shown in Figure 4.

Moreover, the set of log files used in this experiment is only a snapshot of a typical enterprise application in software industries. In the case of handling a real production application running for weeks or even months, the training dataset can be much bigger. As a result, purely saving all keywords and statistical data on a tree data structure would require much more memory space than our approach.

[3]http://docs.oracle.com/javase/1.5.0/docs/api/java/util/UUID.html

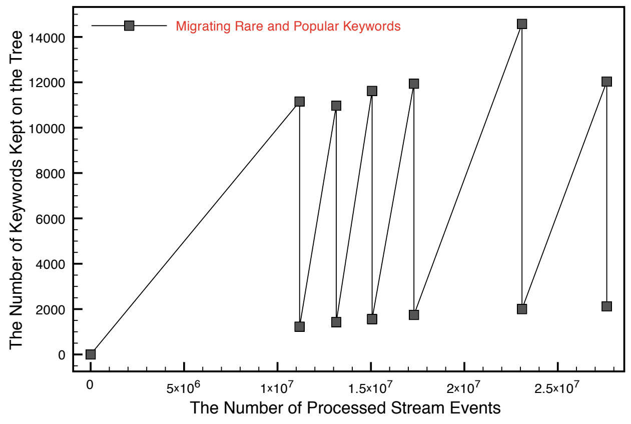

Figure 4: Keyword Migration

Table 1: Performance of Stream Event Prediction

Stream Source:	Application Server	UI Server
Training Throughput	$118k$ events/sec $10.2m$ bytes/sec	$115k$ events/sec $10.7m$ bytes/sec
Training Dataset	$1.4m$ events (First 25% of stream events)	$2.7m$ events
Interesting Events	20,826	42,184
Rare Words	8,937	2,170
Popular Words	792	40
Training Words	10,394	3,134
Prediction Throughput	$119k$ events/sec $10.3m$ bytes/sec	$112k$ events/sec $10.4m$ bytes/sec
Prediction Dataset	$5.9m$ events	$10.8m$ events
Neutral Words	137,667	3,936
Interesting Events (Predicted)	80,898	173,753
Total Events (Predicted)	93,092	192,725
False Positive	13%	9.8%
False Negative	0.3%	0%

4.3 Stream-Based Event Prediction

For the stream-based event prediction, we have used another desktop machine (Pentium4) as the load generator connected to the implemented application via a 100Mbps local network. The data source consists of two sets of log files produced by various system components of an enterprise social application [13]. During the experiment, the load generator has been constantly sending each line of log events to the implemented application until the last event is reached. (The load generator is capable of sending up to $130k$ events per second on average.) Based on the experiment result, we have seen that using 90% as the threshold value for ST and T_s and 10% as T_w has been able to predict interesting events with a reasonable accuracy, which is expected to tolerate 10% false positives and 1% false negatives.

As shown in Table 1, the two stream sources are collected from an application server and UI server. From the system throughput perspective, the UI Server events have been processed with a slightly lower event processing rate (events/sec) but a higher data processing rate (bytes/sec). The main reason is because each UI Server event is longer than the application server. More importantly, the implemented application is able to deliver a high system throughput for both applications during event training and prediction phases. As a result, the proposed solution should be efficient enough to handle real-time event prediction use cases.

During the event training phase, the implemented application has only been trained with a 25% of the overall stream data source produced by each enterprise application. For the best practise, the training dataset for the implemented application should be made as small as possible to reduce the performance overhead introduced by the training process. From the QoS perspective, the more data is trained, the more accurate results can be achieved because less neutral words will be experienced during the prediction phase. However, sometimes the increased training dataset can not significantly improve the quality of the result, instead a significant performance overhead can be introduced. As a result, after trying different sizes of training datasets, the percentage 25% has been found as the most optimal training threshold for all use cases presented in this paper.

In order to maintain a stable heap usage, the keyword migration mechanism has been used to move the rare and popular keywords from the tree to BlmF. As shown in the table, the majority of processed event keywords have rarely occurred, while a small set of keywords have popularly occurred in both application servers. Because the BlmFs are fixed sized data structures, the significant amount of migrated keywords removed from the tree can dramatically reduce the overall heap usage.

From the performance perspective, RTCE has been able to process over a hundred thousands of events per second for both training and prediction processes. The data processing throughput has almost reached the maximal theoretical limitation on the network bandwidth ($12.5m$ bytes/sec). However due to time constraints, we have been able to carry out additional tests in a more powerful hardware environment, which would be done in our future work. In this paper, the main purpose of demonstrating the result of system performance for the current experiment is to show the prospective efficiency of our event training and prediction processes to handle a high volume of stream data. In particular, all the event training and prediction results are almost processed in real-time with no sophisticated buffering techniques being enabled.

Furthermore, due to a small dataset being trained for the implemented application, there has been many neutral keywords found during the event prediction phase. For the application server, 57% of the neutral keywords belong to interesting events and for the UI server 19% of neutral keywords belong to interesting events. Although providing a complete training with the entire stream would make more realistic statistics available for the probability calculation, such an experiment has not been able to significantly improve the quality of the result. In both stream sources, the non-interesting keywords have got low occurrences, which can only give little impact on the probabilities (‘P’) of interesting events. As a result, the false negative values for both servers have remained the same for a complete training.

From the false positive perspective, by training the full dataset the application server has achieved a lower value of

12%, while the UI Server has got a slightly higher value of 10.3% comparing to the 25% of training data. In the application server, most of the interesting keywords have been seen in the interesting events but only a small number of interesting keywords exist in the non-interesting events. When the number of non-interesting keywords increases with a complete training, the probabilities (*'P'*) on many of the non-interesting events have been dramatically reduced such that thousands of such events have no longer believed to be interesting. As a result, the value of false positive has been reduced to 12% from 13%.

However, with a complete training the UI server has achieved a slightly different result, which has increased the value of false positive to 10.3% from 9.8%. For a small fraction of the stream events, the probability of the belief (*'P'*) has surpassed the threshold of interesting events because some of the newly recognised interesting keywords have gained higher occurrences than the non-interesting keywords. But as the probabilities of the majority of the non-interesting events have still been dominated by the non-interesting keywords, most of the non-interesting events were still classified in the result. Thus, a small variance of the false positive has been seen.

More importantly, we have seen that a complete training has only been giving small impact on the quality of result. As the training workload and duration have been significantly increased for training, the trade-off between accuracy and performance has not been equalised. Therefore, a 25% of training dataset is optimal for event training and sufficient for prediction. In real use cases, such a configuration can be predefined based on the requirement of the monitored application. For example, if a testing team is going to evaluate the stability of an enterprise application based on a 4-days run, our solution can be trained with stream data produced by the application during the 1*st* day.

4.4 Belief Modifier Effectiveness

In order to examine how effectively the strong and weak belief modifiers can impact on the event prediction results. Two experiment tests have been carried out for the same stream dataset, which consists of log events produced by the Application and UI servers. The experiment results have shown that applying the belief modifiers the average probability of each stream event has been increased and enabled the engine to make more accurate decisions.

As shown in Figure 5, without using the belief modifiers the event probabilities calculated by the BayeF has varied between 1.0^{-77} and 0.99. By using the belief modifiers, the probabilities varied between 1.0^{-33} and 0.99. In addition, without belief modifiers the most of the events have achieved the probabilities between 1.0^{-34} and 1.0^{-14} while using belief modifiers most events have had probabilities between 1.0^{-15} and 1.0^{-7}. Although most of the events have gained the probabilities below the strong threshold, the belief modifiers have enabled the engine to capture interesting events with a relatively high accuracy as shown in the previous Table 1.

Figure 6 illustrates a similar event distribution to what the application server has achieved. The average event probabilities with the belief modifiers are higher than the ones without using belief modifiers. While most of the stream events, which contained a small of interesting keywords have gained low probabilities (lower than the strong threshold), by using belief modifiers the proposed solution can still capture all the interesting events in the prediction result dataset, which also contains 9.8% of non-interesting events. (See Table 1)

Table 2 gives a detailed breakdown of the event prediction result without using the belief modifiers. During this test, we have trained the engine with the same dataset (25%) for each event stream. However, the experiment has shown a bad quality for the event prediction result, which have gained 62% and 3.4% of false negative rates for the application and UI servers respectively. As discussed previously, the false negative is a worse quality indicator than false positive. Because to deal with a false positive problem, administrators only need to go through a small dataset containing all the predicted events. In the case that a high false negative is achieved, the administrator has to go through the entire stream dataset to find interesting events, which were missing. However, finding the other 62% of interesting events for the application server through millions of events can be an extremely time-consuming task for administrators to carry out.

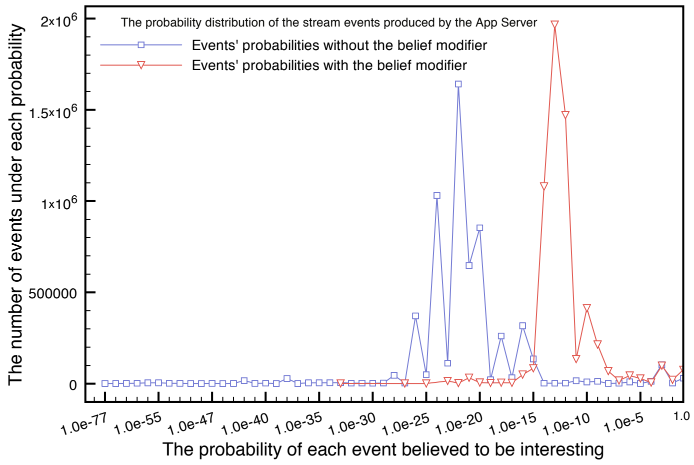

Figure 5: The Event Distribution of Bayesian Probabilities across Application Server's Event Stream.

Table 2: The Prediction Results Without Belief Modifiers

Stream Source:	Application Server	UI Server
Interesting Events (Predicted)	30,778	167,844
Total Events (Predicted)	30,778	167,867
False Positive	0%	0.014%
False Negative	62%	3.4%

4.5 RTCE Integration

In addition to the experiment tests for the standalone application, we have also integrated the solution into the RTCE framework. The goal is to extend the tool with more useful functionality to enable testing teams to easily perform system monitoring related tasks.

During the experiment, we have increased the maximal heap usage to 256 MB for running the RTCE framework

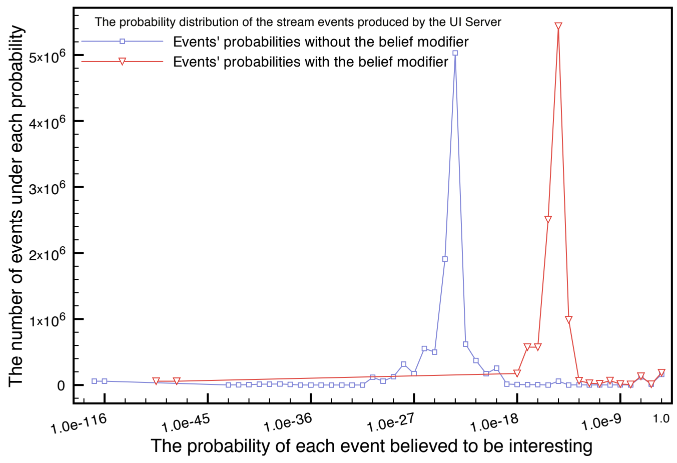

Figure 6: The Event Distribution of Bayesian Probabilities across UI Server's Event Stream.

due to more system components being loaded. Because the proposed solution has mainly focused to enhance the run-time data analysis tasks, the data persistence functionality of RTCE, which is the *'Run'* system component described in the previous work [11] has been disabled during each test. Also, the same stream dataset has been used for all the tests.

As shown in Table 3, the first experiment test has been carried out to show the baseline for the RTCE performance assessment. The result has shown that the original RTCE has achieved a high system throughput of 108.4k events per second, which equally gives the data processing rate of 10.06 megabytes per second. Thereafter, the result of the second test has shown that after integrating our solution to the RTCE framework, the system throughput has slightly decreased to 102.2k events per second for the event training process. Finally, when the integrated RTCE framework performs the event prediction tasks, the system throughput has slightly increased to 106.2k events per second but still lower than the original RTCE performance. From the CPU usage perspective, the integrated RTCE framework has shown a significant increase than the original RTCE due to extra data mining tasks being performed.

Table 3: The Performance of RTCE with Event Prediction Integrated

RTCE Performance	CPU Usage	System Throughput (events/sec)	(bytes/sec)
Original	27.5% $\sim$ 30%	108.4k	10.06m
Integrated (Training)	41% $\sim$ 48.5%	102.2k	9.48m
Integrated (Prediction)	40% $\sim$ 46%	106.2k	9.86m

4.6 SMS Spam and Farm Ad Prediction

Beyond the stream-based event prediction experiment, we have also assessed our solution against static documents: SMS messages [2] and farm ads [23], which have been used by other research works of [3] and [22] respectively.

As shown in Table 4, both documents have been trained to predict interesting records, which are spam SMS messages and accepted farm ads. Because the size of each document is small such that our solution can almost process both documents instantly, the training and prediction throughputs are excluded in this table. By training first 25% of each document, the SMS and farm-ad documents have been trained with 202 spams and 568 accepted-ads respectively.

Table 4: SMS Spams & Farm Ads Prediction

Data Source:	SMS Messages	Farm Ads
Training Dataset	1,393 Messages	1,035 Ads
Interesting Records	Spam: 202	Accepted: 568
Training Words	4,240	17,971
Prediction Dataset	5,574 Messages	4,143 Ads
Neutral Words	10,627	68,001
Interesting Events (Predicted)	741	1,708
Total Events (Predicted)	865	3,027
False Positive	14.9%	43.5%
False Negative	0.8%	22.7%

During the event prediction phase, a significant amount of neutral keywords have been seen for both SMS and farm-ad documents. As discussed previously, depending on the ratio of interesting to non-interesting keywords in the new messages, the values of false positive and false negative can vary a lot.

For the SMS messages, our solution has achieved a 14.9% of false positive and 0.8% of false negative. For the farm ads, 43.5% of false positive and 22.7% of false negative have been achieved. Comparing to the SMS document and other stream-based log events, the farm-ads document has been predicted with much higher false positives due to stronger believes built based on interesting keywords in the new events. Because in many of the new non-interesting events, the interesting keywords have occurred more frequently than non-interesting keywords. As the neutral keywords can not give any impact on the probabilities, such non-interesting events have been falsely included in the interesting event category.

Furthermore during the prediction phase, 99.2% of new spam messages in the SMS dataset have higher recurrence of interesting keywords than non-interesting keywords. As a result, there has been only 0.8% of spam messages missing. In the farm-ad dataset, 24% of accepted-ads have been missing due to many non-interesting keywords occurred in the new accepted-ads, which probabilities have been reduced below the strong threshold due to the lack of unrecognised interesting keywords contained in the neutral keyword set.

4.7 Comparing to Other Classifiers

In order to further assess the quality of our solution, we have compared our approach with three other popular classifiers: Naive-Bayes Tree (NBTree) [18], $C4.5$ [25], and Sequential Minimisation Optimisation (SMO) [17]. $C4.5$ is a decision tree based algorithm featuring the branch pruning ability. NBTree is a hybrid algorithm consisting of both Naive-Bayes and decision tree classifiers. SMO is the successor to the Support Vector Machine (SVM) and optimises the data processing speed. To compare the performance of each algorithm against our approach, we have obtained the implementation for each algorithm from Weka [4], which is an open source project with a wide range of machine learning algorithms implemented in Java.

During the first experiment, we have used the farm-ads dataset to assess the performance of each algorithm. As shown in Table 5, we have trained all four algorithms with 25% of farm-ads for prediction. As a result, both $C4.5$ and NBTree have achieved the highest number of correctly classified farm-ads, while RTCE and SMO being the second and third accurate approaches. From the memory efficiency perspective, the $C4.5$ and RTCE have both been able to successfully run with the default JVM setting, which is 123MB. Furthermore, the time taken to build and predict farm-ads has also been measured for each algorithm to assess their capabilities to fulfil the real-time event prediction requirement. As shown in the table, RTCE has been the least time consuming approach to process farm-ads, while NBTree being the slowest approach.

Table 5: Classifier Comparison Using Farm Ads

Classifiers:	C4.5	SMO	NBTree	RTCE
Training Data	$1,035$ Farm Ads			
Prediction Data	$4,143$ Farm Ads			
Correctly Classified	$2,361$ 56.98%	$2,210$ 53.34%	$2,361$ 56.98%	$2,322$ 56.04%
Incorrectly Classified	$1,782$ 43.01%	$1,933$ 46.65%	$1,782$ 43.01%	$1,821$ 43.95%
Memory Setting	default	512MB	512MB	default
Time to Build	$10.64s$	$2.11s$	$45.44s$	$0.077s$
Time to Predict	$2.58s$	$2.79s$	$5.56s$	$0.698s$

During the second experiment, we have used the SMS message dataset to assess the performance. The same as farm-ads experiment each algorithm has been trained with 25% of SMS messages. As shown in Table 6, RTCE has achieved the highest number of correctly classified SMS messages. Furthermore while $C4.5$ and RTCE are still the most memory efficient approaches, RTCE has been found to be the fastest approach to build and predict SMS messages.

Table 6: Classifier Comparison Using SMS Spams

Classifiers:	C4.5	SMO	NBTree	RTCE
Training Data	$1,393$ SMS Messages			
Prediction Data	$5,574$ SMS Messages			
Correctly Classified	$4,827$ 86.59%	$4,827$ 86.59%	4827 86.59%	$5,438$ 97.56%
Incorrectly Classified	747 13.40%	747 13.40%	747 13.40%	136 2.43%
Memory Setting	default	256MB	256MB	default
Time to Build	$3.64s$	$0.64s$	$7.6s$	$0.035s$
Time to Predict	$0.8s$	$1.09s$	$1.47s$	$0.09s$

Additionally we have also carried out a training test using the full dataset of farm-ads for $C4.5$, which has been the most memory efficient approach beyond RTCE. However, the experiment has shown that $C4.5$ requires a minimum $1.5GB$ memory setting to successfully process the full farm-ads dataset while taking $354.38s$ and $34.41s$ to build and predict data respectively. Comparing to the our approach, which has been able to process the full dataset almost instantly ($0.109s$ to build and $0.792s$ to predict) with the default JVM memory setting, the $C4.5$ is still much less memory efficient and slower.

[4]Weka: Machine Learning Software in Java, http://sourceforge.net/projects/weka/files/weka-3-6/3.6.8

5. RELATED WORK

Naive Bayes Classifier (NBC) [24] has been proposed as an efficient solution to classify raw text documents with multiple attributes. The probabilities of various terms for certain attributes are calculated based on their frequencies in the training dataset. Comparing to other learning methods, NBC is more efficient and suitable for processing a large set of data due to the avoidance of the searching process for possible hypothesis. In addition, NBC also classifies documents when the contents of data are independent. In the case that one of the terms has gained a 0 probability, an estimating probability has to be calculated by adding *'mp'* to the *'numerator'* and *'m'* to the *'dominator'*, where *'m'* is the equivalent sample size and *'p'* is the estimated prior probability. Such a 0 probability problem is avoided in our solution by assigning the minimal probability (1%). Furthermore, a following research work [5] has found that NBC scores can unrealistically move close to 0 or 1 when the number of input words increases.

Iterative Bayes Classifier (IBC) [7] and Adaptive Bayes Classifier (ABC) [8] are two of extended approaches to the core NBC technique. The IBC can further improve the prediction probability by iterating the training dataset. After a normal NBC training process, the IBC needs to cycle through the same dataset again to allow the desired classes to be predicted more confidently, in which case IBC would increase the probability of predicted classes and decrease the probabilities of other classes that are not selected. As a result, the confidence of selecting the predicted classes will become stronger and give more significant impact on the believes of further predictions. For the ABC approach, the confidence of predicted classes is improved by learning from new data contents instead of iterating the old training dataset. In the case of predicting stream events, the ABC approach is more suitable for handling dynamic data contents, which can not be accurately predicted by IBC if there is any changes occurring in the stream. However, neither NBC, IBC, nor ABC has considered the memory-intensive problem to fulfill the requirement of processing a large volume of stream data.

The Conceptual Clustering and Prediction (CCP) framework [16] has been proposed as a stream-based email filtering system, which assumes there is only a limited memory resource available. In their related work section, a number of previous works have been discussed to emphasis the memory constraint problem for stream processing. In order to solve such a memory issue, CCP uses a mapping function to transform every batch of stream data into conceptual vectors and groups multiple vectors into clusters. For each cluster, CCP needs to assign a corresponding classifier, which is also trained and updated by the batch of data. By only keeping the cluster centroid and corresponding classifiers in the main memory, the CCP framework can easily cope with a stream data while there is a constraint on the memory usage. However, in order to achieve an optimal performance, the proper batch size can only be selected after the preliminary experiment results are gained.

Context-aware Ensemble (CAE) [9] is one of recently proposed stream processing engines, which use the ensemble approach [29] to train and update classifiers. CAE maintains a pool of classifiers and based on their weights, an appropriate classifier is used and trained for recurring concepts processing. CAE can improve the adaptation to recurring concepts by exploiting the relationship between contexts and concepts. To deal with memory constraints, CAE can periodically prune away classifiers, which have the lowest utilisation when the maximal number of classifiers is reached. However, the experiment result shows that the accuracy of the CAE approach would be significantly affected when there is some noise data occurring in the underlying stream.

Classifier4j [21] is an open source tool supporting text classification based on the Bayesian theorem. Classifier4j uses the relational database (RDBMS) technology to save the processed keywords and corresponding statistics. Because RDBMS persists data directly onto the hard disk, the memory-intensive problem can be easily avoided by Classifier4j due to a much smaller set of data being kept in the memory. However, as each RDBMS data access requires one disk I/O operation, which is much slower than the RAM data access, the overall system throughput can be limited by a large amount of I/O waiting time while dealing with a high volume of stream data. Moreover, Classifier4j only uses the keywords' frequency as the probability of the belief by assuming a 100% belief for keywords existing in both interesting and non-interesting events. As discussed previously, this mechanism is likely to produce inaccurate probability results when dealing with a high volume of dynamic stream data, where many interesting keywords in predicting events have only been trained with low probabilities.

From the log analysis perspective, Xu.W and et al. [32] have proposed an interesting four-step approach to detect problems through system console logs. Their first step enables unstructured log messages to be parsed based on the logging points of source codes. The second step allows predefined functions to be executed to provide fine-grained information for the third step, which detects the anomalies through variants of variables and identifiers. The last visualisation step shows a decision tree graph to allow users to gain a detailed problem description. By making use of several open source libraries such as Apache Lucene and Hadoop, they have been able to process millions of messages per minute with dozens of nodes while achieving a high result accuracy. Comparing to their approach, which mainly focuses anomaly detection based on related numerical variables and identifiers, our solution is a more light-weighted approach to only predict the existence of an anomaly message rather than message correlation. However, during the experiment, their approach has failed to parse some of the "message type" attributes containing long strings, in which case a complete understanding of all message types, which are not required by our approach is required by their approach to properly process various logs. Furthermore to ease troubleshooting tasks, LogEnhancer [33] has been proposed to enhance logging points in the source code. Comparing to these two approaches, both of which would require the access to the source code and domain expert knowledge to understand it, our solution has been mainly focusing on anomaly detection with non-expert knowledge and assuming source code access is not always possible.

6. CONCLUSIONS

In this paper, we have explained the advantage of using a statistical approach to predict interesting data based on limited training resources. Thereafter, the memory-intensive problem has been detailed under the context of coping with a large volume of stream events. The proposed solution tackles such a problem by using two BlmFs, which are both fixed sized and efficiently save periodically migrated keywords from the tree data structures. To avoid inaccurate prediction results, the lowest and highest probabilities have been normalised using minimal and maximal probabilities respectively. By assigning the minimal and maximal probabilities to each BlmF, the migrated keywords have avoided the sacrifice to the quality of prediction results. The proposed solution has also enhanced the Bayesian theory with strong and weak belief modifiers to achieve better prediction results. Furthermore during the experiment, the implemented event prediction system has been able to predict interesting data through a high system throughput for log-based stream events produced by enterprise software applications, while only being trained with a fraction of the entire stream. As a result, we would conclude that the goal of the research, which develops an effective and efficient event prediction technique to allow stream processing engines to capture interesting data with a high throughput in real-time has been achieved. In the future work, we would like to find a way to automatically discover the optimal belief modifiers without relying on any experiment experience and also try our solution in a much larger environment with much higher workloads over a long period of time.

7. ACKNOWLEDGMENTS

This research was funded by Science Foundation Ireland (SFI) via grant 08/SRC/11403 FAME SRC (Federated, Autonomic Management of End-to-End Communications Services – Scientific Research Cluster).

8. REFERENCES

[1] C. C. Aggarwal. Chapter 1: Introduction to data streams. In *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 1–6. Springer, Heidelberg, 2007.

[2] T. A. Almeida and J. M. G. Hidalgo. Sms spam collection, Sept. 2012. Available [online]: http://www.dt.fee.unicamp.br/∼tiago/smsspamcollection/.

[3] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami. Contributions to the study of sms spam filtering: new collection and results. In *Proceedings of the 11th ACM symposium on Document engineering*, DocEng '11, pages 259–262, New York, NY, USA, 2011. ACM.

[4] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, and C. D. Spyropoulos. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, pages 160–167, New York, NY, USA, 2000. ACM.

[5] P. Bennet. *Assessing the Calibration of Naive Bayes' Posterior Estimates.* In Technical Report CMU-CS-00-155. School of Computer Science, Carnegie Mellon University, 2000.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[7] J. Gama. Iterative naive bayes. In *Proceedings of the Second International Conference on Discovery Science*, DS '99, pages 80–91, London, UK, UK, 1999. Springer-Verlag.

[8] J. Gama and G. Castillo. Adaptive bayes. In F. Garijo, J. Riquelme, and M. Toro, editors, *Advances in Artificial Intelligence, IBERAMIA 2002*, volume 2527 of *Lecture Notes in Computer Science*, pages 765–774. Springer Berlin / Heidelberg, 2002.

[9] J. a. B. Gomes, E. Menasalvas, and P. A. C. Sousa. Learning recurring concepts from data streams with a context-aware ensemble. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 994–999, New York, NY, USA, 2011. ACM.

[10] C. Green and P. Edwards. Using machine learning to enhance software tools for internet information management. In *Proceedings of the AAAI Workshop on Internet-based Information Systems*, pages 48–55, 1996.

[11] V. Holub, T. Parsons, P. O'Sullivan, and J. Murphy. Run-time correlation engine for system monitoring and testing. In *ICAC-INDST '09: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 9–18, New York, NY, USA, 2009. ACM.

[12] IBM. Websphere application server version 6.0.2, Sept. 2010. Available [online]: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/org.eclipse.hyades.log.ui.doc.user/concepts/cesdb.htm.

[13] IBM. Ibm collaboration software, Sept. 2012. Available [online]: http://www-01.ibm.com/software/lotus.

[14] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin. Ibm redbooks, 2004. Available [online]: www.redbooks.ibm.com/redbooks/pdfs/sg246635.pdf.

[15] S. J.Carolan. Introduction to cloud computing, June 2009. Available [online]: http://www.scribd.com/doc/17274860/Introduction-to-Cloud-Computing-Architecture.

[16] I. Katakis, G. Tsoumakas, and I. Vlahavas. Tracking recurring contexts using ensemble classifiers: an application to email filtering. *Knowl. Inf. Syst.*, 22(3):371–391, Mar. 2010.

[17] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Comput.*, 13(3):637–649, Mar. 2001.

[18] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *proceedings of the second international conference on knowledge discovery and data mining*, pages 202–207. AAAI Press, 1996.

[19] K. Lang. Newsweeder: Learning to filter netnews. In *in Proceedings of the 12th International Machine Learning Conference*, 1995.

[20] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace, July 2005.

[21] N. Lothian. Classifier4j, Sept. 2012. Available [online]: http://classifier4j.sourceforge.net.

[22] C. Mesterharm and M. J. Pazzani. Active learning using on-line algorithms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 850–858, New York, NY, USA, 2011. ACM.

[23] C. Mesterharm and M. J. Pazzani. UCI machine learning repository: Farm ads data set, Sept. 2012. Available [online]: http://archive.ics.uci.edu/ml/datasets/Farm+Ads.

[24] T. M. Mitchell. Chapter 6: Bayesian learning. In *Machine Learning*, pages 156–158, 177–179. McGraw-Hill, Inc., New York, USA, 1st edition, 1997.

[25] J. R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[26] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[27] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, Mar. 2002.

[28] B. Sriram, D. Fuhry, E. Demir, H. Ferhatosmanoglu, and M. Demirbas. Short text classification in twitter to improve information filtering. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 841–842, New York, NY, USA, 2010. ACM.

[29] W. N. Street and Y. Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 377–382, New York, NY, USA, 2001. ACM.

[30] M. Wang, V. Holub, T. Parsons, J. Murphy, and P. O'Sullivan. Scalable run-time correlation engine for monitoring in a cloud computing environment. In *Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 29–38, Washington, DC, USA, 2010. IEEE Computer Society.

[31] M. Wang, V. Holub, T. Parsons, P. O'Sullivan, and J. Murphy. Symptom matching for event streams. *IET Software*, 6(4):296–306, 2012.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[33] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 3–14, New York, NY, USA, 2011. ACM.