Towards MapReduce based Bayesian Deep Learning Network for Monitoring Big Data Applications

M. Omair Shafiq, Eric Torunski
School of Information Technology,
Carleton University,
Ottawa, Ontario, Canada.
Email: omair.shafiq@carleton.ca

***Abstract* — One of the most commonly used ways to monitor execution of software applications is by analyzing logs. Logs are execution foot-print of software applications that are produced and stored for real-time or post-execution analysis of execution. With the software applications becoming large, complex, distributed, web-scale, also called as big data applications, logs produced by such software applications are also large-scale. That means, such logs are large in volume, velocity and variety. That makes it crucial to have such logs analyzed in an automated, scalable and effective manner to ensure high veracity and have analytics with high value. In this paper, we present our proposed solution of a formal model for organizing and structuring logs. We then present a Bayesian deep learning network based analysis approach that utilizes the formal model for logs to detect and predict any possible faults and consequences of such faults. Moreover, we also present our MapReduce based distributed, parallel, single-pass and incremental approach to build, train and execute the proposed Bayesian deep learning framework. This helps in effective processing of logs on cloud platforms and therefore efficient handling of logs that are produced at the scale of big data by big data applications.**

Keywords – Formal model; Logs; Bayesian networks; Deep learning; Fault detection; Execution; Monitoring; Big Data; Applications; MapReduce

I. INTRODUCTION

As we know that software applications produce logs to keep record of execution. Such record of execution is analyzed at real-time or at a later stage to analyze, track events, patterns, errors, faults or exceptions. However, with rapid enhancements in the digital age, software applications are becoming large, complex, multi-component, loosely-coupled, web-scale and highly available. While such properties empower software applications, that also makes software application produce logs that are massive in size, high in speed and have a wide variety. In other words, log data that is produced during execution of large-scale software applications is big data. We refer to software applications that produce logs with properties of big data, as big data applications. Recent advancements in software applications have made such applications to be large, complex, distributed, web-scale and data-intensive in nature [1]. Such applications carry out extensive processing of data and are based on distributed and multiple components, executing concurrently, and producing significant execution logs [2].

Traditional techniques of modeling logs in software applications are to use logging libraries and use naïve techniques, along with description in natural language. Consequently, traditional techniques for monitoring of software applications execution are by having software engineers, developers or system administrators manually scan logs to find out information related to any important events, errors, faults, bugs, or exceptions. Clearly, this approach is very limited, hard and cumbersome. Now that the software applications have become large-scale, i.e., big data applications, and produce logs that are also large in scale, i.e., big data, such traditional techniques of producing and processing logs are insufficient and limited. Over the last decade, there have been efforts to build tools for automated analysis of log data. Such tools include, but not limited to, Such tools include, but not limited to, Adiscon LogAnalyzer, WebLog Expert, GitHub Log-analyzer, Sys-Log ManageEngine, and many more. The issues with these tools are that such tools can only help in analysis of smaller amounts of log data that is small-scale and discrete in nature. Most of such tools are unable to process natural language statements in logs. Key challenges are (1) lack of formalism of statements in log data, (2) lack of overall meta-model to structure log data, (3) lack of analytical techniques that take into account meta-model and formalism of logs to carry out analysis, (4) lack of ability of efficient and effective analytical techniques to process logs with big data properties. In other words, most of the solutions address the problem of what to log. However, many solutions lack in the ability to address the problem of how to log.

In this paper, we present our proposed solution of a formal model for organizing and structuring logs. We then present a Bayesian deep learning network based analysis approach that utilizes the formal model for logs to detect and predict any possible faults and consequences of such faults. Moreover, we also present our MapReduce based distributed, parallel, single-pass and incremental approach to build, train and execute the proposed Bayesian deep learning framework. This helps in effective processing of logs on cloud platforms and therefore efficient handling of logs that are produced at the scale of big data by big data applications. MapReduce is a well-known big data programming model and is an effective way of handling and processing big data. Using MapReduce, we tackle the big data in two steps called as map and reduce. The map step takes care of computing global solution by dividing logs into different subsets. The reduce step then performs processing on

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

the independent subsets. After processing of subsets of logs is completed, the local results are produced, which are then combined to generate global solution.

The automated detection mechanism for faults and effects on components includes real-time processing and analysis of execution logs. Therefore, it is important to keep it distributed, parallel, single-pass and incremental. The motivation to keep the solution distributed in nature is because cloud computing platforms can be utilized to process incoming execution logs in a distributed manner. The motivation to keep the solution parallel in nature is because all the distributed computation resources could be able to process execution logs, not only distributed, but also concurrently, and do not have to wait for output from any other computation nodes. It is important to note that big data applications, due to large-scale, multi-component and complex, carry out extensive processing and produce execution logs with high speed. Therefore, it is important to process data in a single iteration only and not requiring any re-iteration on data that has already been processed. It is also important to build our solution that is incremental, so that the fault-detection mechanism can update its model with newly discovered patterns, on top of the existing model, without having to re-iterate through the data that is already processed. Last but not least, is also important to get the best value out of analyzing execution logs with high accuracy in the process of fault-detection.

The rest of the paper is structured as follows. Section 2 presents and discusses related work. Section 3 presents important definitions, our proposed Bayesian deep learning network model, and a MapReduce based processing of such model for fault detection in big data applications. Section 4 presents extensive experimental and evaluation results of our proposed algorithms, complexity analysis, performance, and accuracy analysis. Section 5 presents conclusions followed by acknowledgements and references.

II. Related Work

In this section, we present a brief overview of related works that we found related to the distributed and parallel classification techniques and further perform fault or consequence detection to monitor software applications. Authors in [7] and [8] present a way of implementing several classification algorithms in parallel based on MapReduce. They showed how k-NN, Naïve Bayesian Model and Decision Tree can be modified to use the Map-Reduce paradigm. They tested the parallel speedups of their parallel implementations and found that they had a linear speedup with datasets ranging from 0.25 GB up to 8GB.

In [9], the author presents a survey of large-scale parallel and distributed data mining algorithms. The author discusses issues and challenges in implementing large-scale data-mining algorithms. They give a wide overview of design considerations in parallel computing, such as memory architecture, task vs. data parallelism, static vs. dynamic load balancing. They then present many contemporary papers and where they fall within these design decision categories.

In [10], the authors present SPRINT, a decision-tree based classification algorithm. The algorithm can be parallelized to handle large datasets stored on a cluster of machines. Their results of speedup factors were run using up to 16 processors, and they show that the algorithm scales reasonably with the number of processors used.

In [11], the authors present ScalParC, an algorithm for parallel decision tree classification. It is used for handling large datasets but does not have the scaling problems of other parallel implementations of the SPRINT classifier. They ran their algorithm on a computer using up to 128 processors and observed nearly ideal declines in the memory requirements as the number of processors doubles. The parallel runtimes of the algorithm saw less gains as more processors were used, which is typical as the communication overhead increases.

In [12], the authors describe possible approaches for parallelizing different data mining algorithms. They describe how to use both data and task parallelism in implementing the C4.5 algorithm. They tested the speedup of their algorithms on parallel machines but their results are mixed due to the load imbalance caused by the datasets used.

In [13], the authors present parallel algorithms for building decision tree classifiers on a shared-memory multiprocessor computer. One algorithm, MWK, uses data parallelism and task pipelining. The other algorithm, SUBTREE, uses task parallelism to achieve load balancing. Since their algorithms use a shared-memory architecture, they are limited by the number of processors that can be put into a single machine. At the time of writing this paper, this was 8 processors sharing 1GB of memory.

In [14], an open source statistics framework is presented which performs principal component analysis. Their work introduces general update formulas that allow for pairwise and incremental updates of arbitrary-order centered statistical moments. They test the speedup of their parallel implementation and find that it has near perfect linear speedup.

In [15], the authors present an algorithm, called DCMSVM, for distributed parallel training of single-machine multiclass SVMs. Their algorithm is based on Crammer & Singer's multiclass single-machine SVM formulation, but parallelizes the consensus augmented optimization problem. Their implementation and evaluation shows advantages in both accuracy and training time. Their work allows single machine methods to scale to larger problems.

In [16], the authors present a new distributed machine learning framework to implement various ML algorithms that are iterative-convergent. Their design principles focus on topics of error tolerance, dynamic structural dependence, and rapid convergence of parameters. Using their framework, programmers can implement data and model-parallel ML algorithms. Their framework provides facilities for accessing parameters representing the global model state, and for scheduling updates of the model. To test their framework, they implemented data-parallel distance metric learning, parallel lasso, LDA, matrix factorization, and deep learning (CNN). They compare their implementation with other frameworks, and find that in general, Petuum is 2 to 6 times faster.

In [17], the authors investigate the feasibility of implementing one-pass analytics algorithms using MapReduce. They present limitations of Hadoop-based MapReduce in implementing one-pass analytics. To overcome these barriers, they modified an implementation of Hadoop so that it uses

hash techniques for fast in-memory processing. Their performance evaluation shows that their hash techniques significantly improve the progress of map tasks, and enable fast in-memory processing of the reduce function, given sufficient memory.

In [18], the authors present methods for data mining when the data is dynamic and distributed. Their algorithm is used for mining frequent item sets, even when new data is arriving. They can update their results without rescanning the entire dataset. Their testing shows that their approach is an order of magnitude speedup over the naïve recomputation method.

Another paper [19] presents a method for quickly rejecting sub-optimal model parameter configurations when tuning the objective function corresponding to each configuration. This is the basic idea of an online approximate gradient descent which they implement in GLADE. Their paper does not give numeric results for the improvement over previous works, however they show graphs supporting the premise that their approach provides real benefits to analytics model training.

A very popular paper [20] from Chu et al. shows how the Map-Reduce paradigm can be used to implement various machine learning algorithms in parallel. They implement k-means, logistic regression, naïve bayes, SVM, ICA, PCA, and several others, with the goal of showing they can all be implemented using their parallelization framework. Their implementation is not meant to be faster to customized parallel algorithms, but rather it shows that the framework is generic and can be used as a basis for many different implementations. To test the speedup of their framework, they implemented many common machine learning algorithms first as a serial implementation, and then in parallel using map-reduce. The results show that their implementation gives good parallel speedup, and are able to scale to the number of processors, in this case a maximum of 64.

In [21], the authors propose improvements to the naïve Bayes algorithm for text classification. Automatic text classification is a problem for the traditional naïve Bayes algorithm due to problems in the parameter estimation process. The authors propose two empirical heuristics: per-document normalization and feature weighting, which improve the accuracy when using standard benchmark collections.

In [22], authors present a comparison of several predictive monitoring techniques. Previously, monitoring techniques have been used individually but the authors test a combination of approaches: machine learning, constraint satisfaction, and QoS aggregation. They found that constraint satisfaction combined with QoS aggregation improved precision by 14%. Combining machine learning with constraint satisfaction improved the recall by 23%.

In [23], the authors used Bayesian networks to predict problems of processes in software development. Software development processes historically have a low chance of success. The authors investigated scrum-based software development process, and modeled it as a Bayesian network. The goal is to determine process problems that increase the risk of failure. They model the probability functions of relationships by asking software developers through online surveys. They then validated their process by following two software development projects in Brazil.

In [24], the define the requirements for a real-time monitoring system for software running on a computer. They motivation they provide is to evaluate the performance of a program, and to detect programming errors. This paper was written in 1984, so many of the concepts they present are now commonplace. This includes variable watch expressions, software profiler. In this paper, the authors implemented the features using a separate processor so that they could measure accurate timing values of the software execution.

In [25], the authors present a new monitoring and measurement software architecture for software defined infrastructure, which they call MonArch. It is based on the concept of Software Defined Infrastructure. The goal is to offer monitoring as a service, as well as analytics data to customers. Their implementation is deployed using the SAVI Testbed in order to gather results about its performance and scalability. They found that the system is able to handle users' requests with short delay, while being able to scale close to linearly with the number of user requests. Their ability to also process large numbers of incoming monitoring messages also scales close to linearly. They comment that they will extend their work to improve the analytics part of their implementation to provide advanced anomaly detection, auto diagnosis, and root cause analysis functionality.

In [26], the authors present a cooling schedule for multi-core systems, that still maintain service level agreements (SLAs) within a 10% margin. They have implemented a scheduler in to the Linux kernel that achieves a 25.8%, with an 8.7% performance loss. This is done by reducing the computational complexity of the scheduler, grouping tasks to reduce Dynamic Voltage Frequency scaling (DVFS) operations, and using a feedback mechanism to allow performance loss within tolerance levels of the SLA.

In [27], the authors introduce a software visualization tool that provides a high–level view of what is going on in a Java system. Their program shows how many class methods are being called, the number of memory allocations, number of thread synchronizations, as well as the amount of time threads spend in each of their states (running, running synchronized, performing I/O, waiting, blocked, sleeping or dead). Each attribute mentioned is displayed in a grid of icons, with height, color, brightness all having a different meaning.

In [28], authors show how the Densification Power Law (DPL) can be used as a metric for software execution. The DPL says that as a networked system evolves, the number of edges and nodes grows with a super linear relation. The authors look for this relationship in a static call graph of software by investigating 15 open-source Java programs. The authors find that pattern repeats in the software, and they explain that it is due to the reuse of software methods.

In [29], the authors present manufacturing execution systems, and how they are used to gather data on industrial production to get insight into processes. They then present feasibility of using these same processes for software development. They prototype a system, called Modularity Debt Management Decision Support System (MDM-DSS). They list several problems they encountered in their implementation, such as a lack of common interfaces and protocols for gathering data. They highlight the importance and potential

gains of creating a standard interface for various tools to provide data (HR and costs) for analysis.

We found out that most of the techniques attempted to realize parallel or MapReduce based classification techniques, however, focused on volume of the data. Other techniques attempted to utilize classification and other similar techniques in development, monitoring and measurement of software systems. The issues and limitations with these approaches are that most of the approaches rely on unstructured and unorganized software development and execution data which limits their ability to perform fault or problem detection. Our proposed solution, first of all, presented a formal model for execution logs, that provides a solid underlying organization for data. Secondly, most of the approaches focus on building solutions for particular and small-scale software applications or projects. Our solution especially focuses on emerging software applications that are large, distributed, complex and data-intensive in-nature. Our solution is a combination of formal modeling of execution logs, building distributed, parallel, single-pass and incremental Bayesian deep learning network classification based on MapReduce. We further adapt it to utilize formal log models to perform scalable, effective and efficient detection of faults and effect on components in big data applications.

III. Proposed Solution

In this section, we present our proposed solution of monitoring execution of big data applications. Our proposed solution consists of formal model for logs as prescribed by definitions, a Bayesian deep learning network model, and a MapReduce based approach for training such Bayesian deep learning network to detect faults and consequences of such faults during execution of big data applications. As we know that big data applications produce logs with high volume, velocity, and variety.

A. Definitions

Given below are some definitions to formally describe our proposed solution.

Definition 1 (Big Data Application): It is a software application that is large, complex, multi-component and possibly distributed. It produces execution data that has high volume, velocity, and variety.

Definition 2 (Component - C): Let C be a component in a software application that may be responsible for certain task(s) in the application.

$$C = (UniqueID, ComponentName, ComponentStatus, Inputs, Outputs)$$

UniqueID is a unique identifier that is assigned to a component. *ComponentName* is a human readable name of a component. *ComponentStatus* refers to the current state of a given component that may range from start to end. *Inputs* refer to a set of key-value pairs that be provided as input to a given component. *Outputs* refer to a set of key-value pairs that be produced as output by a given component.

Definition 3 (Inputs): Let I be a set of key-value pairs that may be required by a component as input.

$$Inputs = \{ kv_1, kv_2, kv_3, \ldots kv_n \}$$

Definition 4 (*Outputs*): Let *O* be a set of key-value pairs that may be produced by a component as output.

$$Outputs = \{ kv_1, kv_2, kv_3, \ldots kv_n \}$$

Definition 5 (Event - E): Let E be an event that is a concrete step, in a software, during its execution, in a given instant of time.

$$E = \{ A_1, A_2, A_3, \ldots A_n \}$$

Where A_i can be any attribute including unique identification, event name, timestamp, event status, component, one or more attributes of application specific data as key value pairs. Therefore, E can also be defined as:

$$E = (UniqueID, EventName, TimeStamp, EventStatus, Component, KeyValuePairs(n))$$

UniqueID is a unique identifier that is assigned to an event. *EventName* is a human readable name of an event. *TimeStamp* contains exact date and time the event took place. *Component* is the name of the component in which the event took place. *EventStatus* refers to the current state of a given event that may range from start to end. *KeyValuePairs(n)* represent multiple key-value pairs that may contain application specific data and variables to be recorded in logs.

Definition 6 (ΔT): Let ΔT be an interval between two timestamps in which a set of events (*Es*) may take place, such that:

$$Es = \{ E_1, E_2, E_3, \ldots E_n \}$$

Definition 7 (*Fault*): Let *Fault* be a known error, exception, irregularity in an application. There can be one or more, and up to k, faults that may occur during execution of a software application:

$$Faults = \{ Fault_1, Fault_2, Fault_3, \ldots Fault_k \}$$

B. Bayesian Deep Learning Network Model for Fault Detection

In this section, we present our proposed Bayesian deep learning network model for detection faults in big data applications. Bayesian network is a graphical representation of a probability distribution over a set of different events that may occur during execution of big data applications. Bayesian networks are represented as a directed acyclic graph (DAG) in which vertices represent variables and edges represent conditional dependencies between the corresponding vertices. Our proposed Bayesian deep learning network has three key segments. First segment is composed of events that may include different features of events as mentioned in definition 5. Second segment is fault as mentioned in definition 7. Third segment is components as mentioned in definition 2. Our proposed Bayesian deep learning framework captures events that may case faults and eventually the faults that may further cause consequences as effects on functionality of different components of big data applications. Figure 1 depicts our proposed Bayesian deep learning framework with three different segments designed to capture potential dependencies, and therefore, correlations between different events, faults and components in big data applications.

Our proposed Bayesian deep learning network is based on Bayesian classification model [8] which is a probabilistic model and is based on Bayes' theorem. Our class variable of

the classifier may have different class labels that are different possible fault types as shown in (1) and (2).

Faults = { $Fault_1$, $Fault_2$, $Fault_3$, … $Fault_k$ } (1)

ClassVar = Faults = {NoFault, MemoryError, DatabaseError, RequestError, ResponseError, InputError, OutputError, ConnectivityError, OtherError} (2)

In this particular model, *NoFault* means that the event is not classified as a fault or error. *MemoryError* means that the event is classified as a fault related to error originating from main memory limitation of the machine execution the application. *DatabaseError* means that the event is classified as a fault related to error originating from database connected to the application. *RequestError* means that the event is classified as a fault related to error originating from input request received by the application being monitored. *ResponseError* means that the event is classified as a fault related to an error originating from output response that is produced by the application being monitored. *InputError* means that the event is classified as a fault related to an error originating from invalidity of the input data that is received by the application being monitored. *OutputError* means that the event is classified as a fault related to an error originating from invalidity of the output data that is produced by the application being monitored. *ConnectivityError* means that the event is classified as a fault related to an error originating from connectivity of components within the application being monitored. *OtherError* means that the event is neither classified as no fault nor classified as any other known error types. In other words, the event is unclassified.

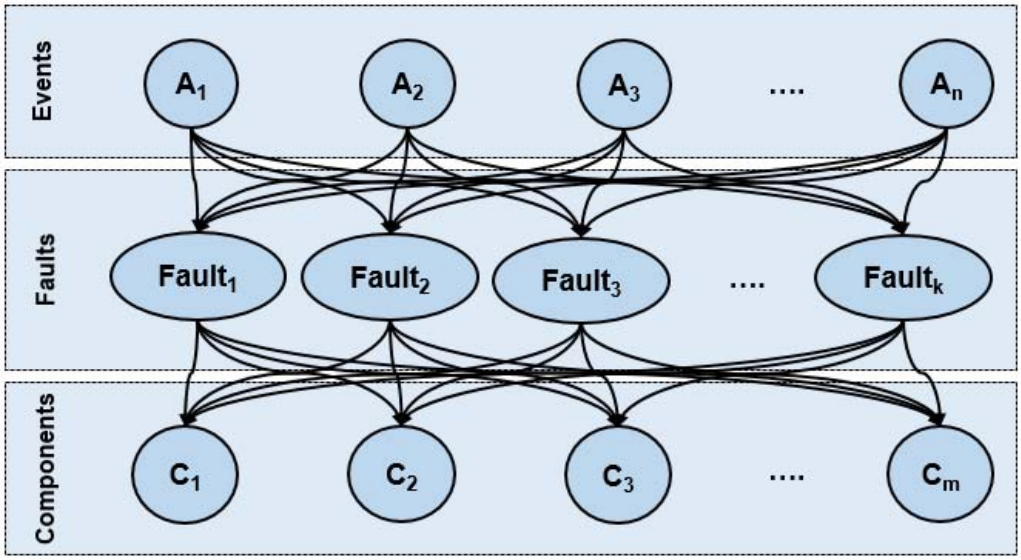

Figure 1: A first outlook of Bayesian deep learning network for Fault Detection

There can be multiple domain variables with multiple domain labels based on which our classifier can be trained and perform classification. Therefore, a classifier can be multiple domain variables as different possible events as shown in (3).

DomVars = Events = { E_1, E_2, E_3, … E_m } (3)

Each of the events, as domain variables, can have different and multiple labels, called as event labels, according to the definition presented for events as *definition 4*. For example, domain variable 1 (e.g., E_1) may have different values for unique identification, event name, timestamp, status of event, component in which the event took place, and the set of key-value pairs that contain application specific data.

E_i = { A_1 , A_2 , A_3 , … A_n } … (4)

Therefore, a domain variable can be defined as:

$DomVar_1$ = E_1 = { *UniqueID, EventName, TimeStamp, EventStatus, Component, BusinessProcess, KeyValuePairs(n)* } (5)

Similarly, there can be more and independent domain variable with multiple and independent labels respectively. A Bayesian classifier determines class label for its class variable based on multiple and independent domain variables with multiple domain labels respectively. The classification is carried out as shown in (5), (6), (7), and (8).

P ($Fault_1$) = $\sum_{i=1}^{n}$ P ($Fault_1$ | $E.A_1$, $E.A_2$, $E.A_3$, … $E.A_n$) … (5)

P ($Fault_2$) = $\sum_{i=1}^{n}$ P ($Fault_2$ | $E.A_1$, $E.A_2$, $E.A_3$, … $E.A_n$) … (6)

P ($Fault_3$) = $\sum_{i=1}^{n}$ P ($Fault_3$ | $E.A_1$, $E.A_2$, $E.A_3$, … $E.A_n$) … (7)

…

P ($Fault_k$) = $\sum_{i=1}^{n}$ P ($Fault_k$ | $E.A_1$, $E.A_2$, $E.A_3$, … $E.A_n$) … (8)

We assume that individual attributes within events are independent to each other. Therefore, conditional distribution over class variable *Fault* can be expressed as shown in (9).

$$\mathbf{P\ (Fault \mid E.A_1, E.A_2, E.A_3 \ldots E.A_n) = (1/Z) * P(Fault) * \prod_{i=1}^{n} P\ (E.A_i \mid Fault) \ldots (9)}$$

Whereas, *Z* is the scaling factor that is dependent only on the domain variables as events that are known through data that is processed for training the classifier or performing classifications. As per our derivation of naïve Bayesian classification model, it combines the model along with a decision rule which is as simple as selecting the most probable. Therefore, the classification of a Bayesian classifier can be expressed as shown in (10).

$$\mathbf{Classify\ (E.A_1, E.A_2, E.A_3 \ldots E.A_n) = argmax\ P\ (Fault = faultLabel) * \prod_{i=1}^{n} P\ (E.A_i = a_i \mid Fault = faultLabel) \ldots (10)}$$

Fault is the overall class variable, whereas *faultLabel* is any particular value of the fault type as class variable. The same applies to A_i *as domain variable and* a_i as any particular value of the domain variable.

We also attempt to detect correlations of faults that may occur in application with potential consequences or effects on functionality of components. This helps system administrators, programmers and other similar stakeholders in not only detecting faults but also components that are potentially affected as a result of such faults. We calculate probabilities of such correlations between different possible faults and components using (11), (12), (13), and (14).

$$\mathbf{P\ (C_1) = \sum_{i=1}^{k} P\ (\ C_1 \mid Fault_1, Fault_2, Fault_3, \ldots Fault_k) \ldots (11)}$$

$$\mathbf{P\ (C_2) = \sum_{i=1}^{k} P\ (\ C_2 \mid Fault_1, Fault_2, Fault_3, \ldots Fault_k) \ldots (12)}$$

$$\mathbf{P\ (C_3) = \sum_{i=1}^{k} P\ (\ C_3 \mid Fault_1, Fault_2, Fault_3, \ldots Fault_k) \ldots (13)}$$

…

$$\mathbf{P\ (C_m) = \sum_{i=1}^{k} P\ (\ C_m \mid Fault_1, Fault_2, Fault_3, \ldots Fault_k) \ldots (14)}$$

C. MapReduce based Bayesian classification for Bayesian deep learning network to carry out fault-detection

In this section, we present our solution of performing classification on log data that has high volume, velocity and variety, as produced by big data applications. Our approach is to divide complex Bayesian network into simpler classification models and then compute it using MapReduce based classification. Our proposed solution is based on MapReduce [3] programming model and implemented on Hadoop platform that stores data on Hadoop Distributed File System (HDFS) [4]. MapReduce is a well-known programming model that enables processing of large-scale data (i.e., data with large volume, velocity and variety) to be carried out in a distributed and parallel manner. The key benefit to use MapReduce is that it allows specifying data storage, distribution to different nodes for parallel processing as well as any required load balancing. Our methodology in building our MapReduce based distributed and parallel Bayesian classification is to adapt the original algorithm and make it a MapReduce task, which is similar by analogy to [5] [6]. Therefore, our proposed solution has two stages that are based on two different steps of the MapReduce. First stage is named as *map* that computes global classification solution. The second stage is called as *reduce* that performs processing on subsets of data and calculates local solution for classification. The iteration between map and reduce steps is then performed that generates global classification solution by combining different and multiple local classification solutions.

The map stage is responsible for initializing, calculating and maintaining global solution in the name node of Hadoop platform. It takes data that is in Hadoop Distributed File System (HDFS) [4] format based on key-value pairs, i.e., *Tuple <key, value>*. Key acts as an index and points to the position of data that could be at start, or somewhere in the middle of the file with a given offset from the start. The value is the actual data as content in the data file. The key-value pair format enables the map stage at name node to split data into different and multiple subsets and assign it to different data nodes. After that, local classification solutions are computed at each of the data nodes executing independently and in-parallel to each other. The outcomes of the data nodes are referred to as local classification solutions. The map stage then computes a global classification solution based on multiple local classification solutions.

The reduce stage is executed independently, and concurrently on each of the individual data nodes. Each of the data nodes are given reference to a given subset of logs to compute local classification solutions. Local classification solutions are then produced and returned to the map stage for generating global solution. The reduce algorithm is described as algorithm 3.

It is evident from the execution of mapper stage in algorithm 2 that it has two steps. The first step invokes reduce stage and provides reference to subsets of datasets to the data nodes. Each of the data nodes then processes assigned subset of dataset and generates local classification solution. However, the mapper stage requires to meaningfully combine results from all the data nodes to generate global classification solution. For the sake of clarify, we have described the intermediate stage as a combine stage.

Algorithm 1 computes rLocals as a set of references to different subsets of logs in the log repository with each of the subset of logs belonging to a particular fault type and component. In this way, we attempt to slice Bayesian network into simpler Bayesian classification problem.

Algorithm 1: initialization of mapper
Inputs: {***rGlobal*** as reference to log repository, ***kFaults***, as number of known faults, ***mComponents*** as number of components}
Output: {***n*** number of pointers as ***rLocals*** pointing to subsets of logs assigned to different available data nodes}
Algorithm: ***map*** (***rGlobal, kFaults, mComponents***)
1. Use ***rGlobal*** and sort logs in HDFS format based on the

unique keys of the key-value pairs.
2. ***nDiv*** = total number of available key value pairs in log repository / ***kFaults + mComponents***
3. Iterate ***i*** from 1 to ***kFaults + mComponents***
   3.1. ***rLocals[i]*** = ***rGlobal***
   3.2. ***rGlobal*** += ***nDiv***
   3.3. ***nNodes[i]*** = ***rLocals[i]***
4. Output ***rLocals*** as a set of references to subsets of data.

Algorithm 2 executes mapper and further initiates reduce stage by providing references of subsets of logs as rLocals to different available data nodes as nNodes. The reduce stage performs computation of local classification results and then generates a global solution. This algorithm also plays an important and crucial role in combining local classification solutions into global classification solution by combining local probabilities for all the data nodes into global probabilities for attributes, faults, and potential effect on components.

Algorithm 2: execution of mapper

Inputs: {***rLocals*** as a set of references to subsets of logs, ***nNodes*** as number of available data nodes}

Output: {***globalFaultProbabilities***, ***globalEventProbabilities***}

Algorithm: map (*rLocals, nNodes*)

1. Iterate ***i*** from 1 to ***nNodes***
   1.1. reduce (***nNodes[i], rLocals[i]***)
2. Combine ***attribute*** probabilities
   2.1. Initialize ***globalAttributeProbabilities***
   2.2. Iterate ***i*** from 1 to ***nNodes***
      2.2.1. Update ***globalAttributeProbabilities*** with ***localAttributeProbabilities*** for ***nNodes[i]***
3. Combine ***fault*** probabilities
   3.1. Initialize ***globalFaultProbabilities***
   3.2. Iterate ***i*** from 1 to ***nNodes***
      3.2.1. Update ***globalFaultProbabilities*** with ***localFaultProbabilities*** for ***nNodes[i]***
4. Combine ***componenteffect*** probabilities
   4.1. Initialize ***globalComponentEffectProbabilities***
   4.2. Iterate ***i*** from 1 to ***nNodes***
      4.2.1. Update ***globalComponentEffectProbabilities*** with ***localComponentEffectProbabilities*** for ***nNodes[i]***
5. Output ***globalAttributeProbabilites*** , ***globalFaultProbabilites*** and ***globalComponentEffectProbabilities***

Algorithm 3 outputs local class probabilities for each of the class labels as well as local event probabilities for each of the event labels.

Algorithm 3: reduce stage

Input: {***iLocal*** as reference to ***ith*** dataset, ***iNode*** as the ***ith*** data node}

Output:{ ***localAttributeProbabilities, localFaultProbabilities***, ***localComponentEffectProbabilities*** }

Algorithm: ***reduce (iLocal, iNode)***

1. For each of the key-value pair from the subset of logs
   1.1. Update ***localAttributeProbabilities***
      1.1.1. For each ***Attribute***
         1.1.1.1. If ***attributeLabel*** exists, increment by 1
         1.1.1.2. Else create new ***attributeLabel*** and set to 1
   1.2. Update ***localFaultProbabilities***
      1.2.1. For each ***Fault***
         1.2.1.1. If ***faultLabel*** exists, increment by 1
         1.2.1.2. Else create new ***faultLabel*** and set to 1
   1.3. Update ***localComponentEffectProbabilities***
      1.3.1. For each ***ComponentEffect***
         1.3.1.1. If ***ceLabel*** exists, increment by 1
         1.3.1.2. Else create new ***ceLabel*** and set to 1
   1.4. Output local probabilities
      1.4.1. ***localAttributeProbabilities*** as number of each ***attributeLabel*** / sum of all ***attributeLabels***
      1.4.2. ***localFaultProbabilities*** as number of each ***faultLabel*** / sum of all ***faultLabels***
      1.4.3. ***localComponentEffectProbabilities*** as number of each ***componenteffectLabel*** / sum of all ***componenteffectLabels***

Training of Classifier for Big Data on MapReduce:

Algorithm 4 invokes the algorithms 1, 2 and 3 to calculate global attribute, fault and component effect probabilities using MapReduce paradigm on Hadoop platform. The MapReduce paradigm enables efficient and effective processing of logs as large-scale data by different individual data nodes executing parallel to each other. These algorithms serve as basis for training our proposed solution of MapReduce based distributed, single pass, parallel and incremental Bayesian classification, based on our proposed Bayesian deep learning network model, for big data applications. Algorithms are executed in following steps to perform the necessary training:

Algorithm 4: Training

1. Initialization of mapper using ***Algorithm 1***
2. Execution of mapper using ***Algorithm 2***, and
   2.1. Execution of reduce using ***Algorithm 3*** within ***Algorithm 2***

Prediction using Classifier for Big Data on MapReduce:

After training Bayesian classifier on MapReduce, the next step is to perform prediction. Prediction is performed based on classification using the trained Bayesian classifier on MapReduce. The prediction relies on the global classification solution produced by the map stage as described in the training procedure and explicitly described in the algorithms 1, 2, 3 and 4. The prediction is carried out using algorithm 5.

Algorithm 5: Detection of Fault and Component effect using classification

Input: { ***globalAttributeProbabilites, globalFaultProbabilites***, ***globalComponentEffectProbabilites, attributeLabels*** in ***Attributes***}

Output: {Probabilities for all possible ***faultLabels*** in ***Faults***, and ***ceLabels*** in ***ComponentEffects*** }

Algorithm: ***classification***

// Calculate prior probability

1. For each ***attributeLabel*** in ***Attributes***
   1.1. Probability of $attributeLabel_i$ = number of occurrences of $attributeLabel_i$ / sum of all occurrences of all ***attributeLabels***
2. For each ***faultLabel*** in ***Faults***
   2.1. Probability of $faultLabel_i$ = number of occurrences of $faultLabel_i$ / sum of all occurrences of all ***faultLabels***
3. For each ***ceLabel*** in ***ComponentEffect***
   3.1. Probability of $ceLabel_i$ = number of occurrences of $ceLabel_i$ / sum of all occurrences of all ***ceLabels***

// Calculate evidence

4. For each ***faultLabel*** in ***Faults***
   4.1. For each ***Attribute*** in ***Attributes***
      4.1.1. For each ***attributeLabel*** in ***Attribute***
         4.1.1.1. Probability = number of occurrences / total number of occurrences
5. For each ***ceLabel*** in ***ComponentEffect***
   5.1. For each ***Attribute*** in ***Attributes***
      5.1.1. For each ***attributeLabel*** in ***Attribute***
         5.1.1.1. Probability = number of occurrences / total number of occurrences

// Calculate likelihood

6. For each ***faultLabel*** in ***Faults***
   6.1. For each ***Attribute*** in ***Attributes***
      6.1.1. For each ***attributeLabel*** in ***Attribute***
         6.1.1.1. Probability = number of occurrences / total number of occurrences
7. For each ***ceLabel*** in ***ComponentEffect***
   7.1. For each ***Attribute*** in ***Attributes***
      7.1.1. For each ***attributeLabel*** in ***Attribute***
         7.1.1.1. Probability = number of occurrences / total number of occurrences
8. Calculate posterior probability for each ***faultLabel,*** and ***ceLabels*** using Prior Probability * Likelihood / Evidence
9. Output posterior probabilities for ***faultLabels*** in ***Faults***, and ***ceLabels*** in ***ComponentEffects***

IV. Evaluation

This section presents evaluation of our proposed solution of fault detection in big data applications. We discuss complexity analysis the algorithms to find out the feasibility of our proposed solution. After that, we have carried out experimental evaluation to find out how our proposed solution can efficiently process large-scale log data from big data applications.

As we know that the logs produced by big data applications have high volumes. Therefore, we analyzed our proposed solution to find out if our proposed training algorithm can sustain with log data that is large in volume. We carried out experiments with increasing size (i.e., volume) of the log data used for training, and with increasing number of computation nodes (i.e., data nodes in Hadoop platform). In principle, time taken by our proposed solution for training of classifier should be same if we increase volume of training data in direct proportion to the number of data nodes in Hadoop platform. In the experiments, we used different sizes of log data as 10, 20, 30, 40 GB that were executed on Hadoop platform with 1, 2, 4, and 8 data nodes respectively. We noted that, once we started increasing the number of data nodes in Hadoop platform, the inter-node communication overhead also started to increase, but with relatively lesser significance. Therefore, it does not affect overall performance of our proposed solution. Figure 2 shows the performance of our proposed solution that how it scales with increasing volume of log data with higher numbers of data nodes.

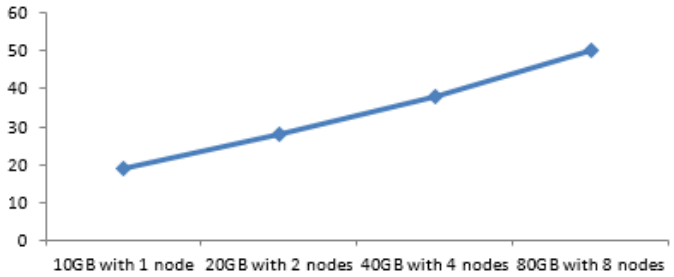

Figure 2: Time taken vs Size of Logs and Number of Nodes

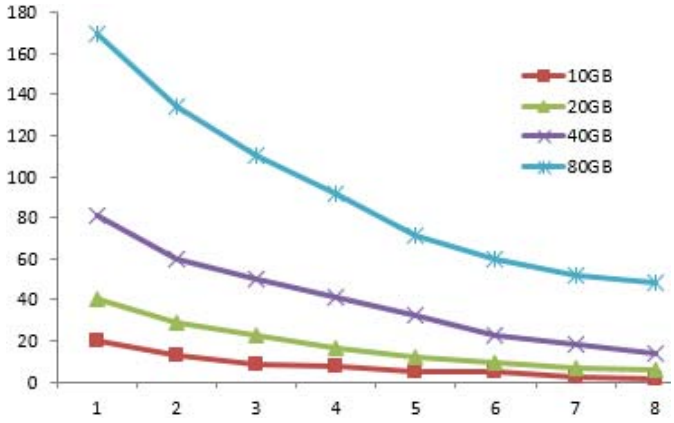

Figure 3: Time taken versus Number of nodes

We also know that big data applications not only produce logs with high volume but with also high velocity. Therefore, we further analyzed our proposed solution that how we can train our classification technique if the speed with which the log data is being generated is increased to simulate high velocity. We kept the volume to the log dataset to be the same and we changed the number of data nodes for computation in different runs to imitate variation in velocity of the data produced. Different experiment runs were conduced with increasing number of data nodes while the volume of log data

was kept being the same. In principle, there should have been a similar to linear decrement in the time taken by our proposed solution, however, we found out that it is possible if there are no communication overheads. Experimental results demonstrate that our training algorithm shows a linear trend but encounters inter-node communication overhead once we increased the number of nodes. We further noted that the communication overhead was found to be significantly less compared to the time taken to train classifier using our proposed solution. In the multiple experiment runs, we used fixed size log dataset with 1, 2, 4, and 8 nodes respectively. Similarly, we used more log datasets with different volumes and with different numbers of data nodes. Figure 3 shows the performance of our proposed solution with increasing velocity of logs with different numbers of data nodes.

V. CONCLUSIONS

Emerging software applications are becoming large-scale, complex, distribute and data-intensive, and produce enormous amounts of logs, i.e., big data applications. Such logs are produced with high volume, velocity and variety. Processing and analysis of logs at a large-scale becomes challenging. We presented our proposed solution of a formal model for organizing and structuring logs. We then presented a Bayesian deep learning network based analysis approach that utilizes the formal model for logs to carry out detection of any possible faults and consequences, as effect on components, of such faults. We further presented a MapReduce based distributed, parallel, single-pass and incremental approach to build, train and execute the proposed Bayesian deep learning framework. We found out that our proposed solution helps in effective processing of logs on cloud platforms and therefore efficient handling of logs that are produced at the scale of big data by big data applications. We further carried out extensive evaluation to justify the effectiveness and usability of our proposed solution for monitoring big data applications. We analyzed complexity and scalability of our proposed solution in which we demonstrated that how our proposed solution performs efficiently and effectively with the log data that is large scale. We also carried out experiments to measure accuracy and therefore demonstrating usefulness of our proposed solution for monitoring big data applications. This work opens doors for us to build even more intense analysis and prediction techniques and solution for monitoring big data applications.

REFERENCES

[1] V. Gorodetsky, Big Data: Opportunities, Challenges and Solutions. Information and Communication Technologies in Education, Research, and Industrial Applications, 3-22, 2014.

[2] C. L. Philip Chen, Chun-Yang Zhang, "Data-intensive applications, challenges, techniques & technologies: A survey on Big Data", Elsevier Info. Sci. Journal, pp 314–347, Vol 275, Aug 2014.

[3] Hadoop: Open source implementation of MapReduce, http://lucene.apache.org/hadoop/

[4] D. Borthakur, "HDFS architecture guide." Hadoop Apache Project, http://hadoop. apache. org/common/docs/current/hdfs design. pdf (2008).

[5] M. O. Shafiq, "Event Segmentation using MapReduce based Big Data Clustering", 2016 IEEE Intl Conference on Big Data (IEEE BigData 2016), 5-8 Dec 2016, Washington, DC, USA.

[6] M. O. Shafiq, E. Torunski, "A Parallel K-Medoids Algorithm for Clustering based on MapReduce", 15th IEEE International Conference on Machine Learning & Applications (IEEE ICMLA 2016), 18-20 Dec 2016, Anaheim, California, USA.

[7] L. Zhou, H. Wang, W. Wang, Parallel Implementation of Classification Algorithms Based on Cloud Computing Environment, TELKOMNIKA Journal of Electrical Engineering, Vol.10, No.5, September 2012.

[8] Q. He, F. Zhuang, J. Li, Z. Shi, Parallel Implementation of Classification Algorithms Based on MapReduce, International Conference on Rough Sets and Knowledge Technology (RSKT 2010), Oct 2010, Beijing, China.

[9] M. J. Zaki, Parallel and Distributed Data Mining: An Introduction, pp 1-23, In: Zaki M.J., Ho CT. (eds) Large-Scale Parallel Data Mining. Lecture Notes in Computer Science, Vol 1759. Springer, Berlin, Heidelberg.

[10] J. Shafer, R. Agrawal, M. Mehta, Sprint: A scalable parallel classifier for data mining, In: 22nd VLDB Conference. (1996).

[11] M. Joshi, G. Karypis, V. Kumar: ScalParC: A scalable and parallel classification algorithm for mining large datasets. In: Intl. Parallel Processing Symposium. (1998)

[12] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Huning, M. Kohler, J. Sutiwaraphun, H. W. To, Y. Dan, Y: Large scale data mining: Challenges and responses. In: 3rd Intl. Conf. on Knowledge Discovery and Data Mining. (1997)

[13] M.J. Zaki, C.T. Ho, R. Agrawal: Parallel classification for data mining on shared-memory multiprocessors. In: 15th IEEE Intl. Conf. on Data Engineering. (1999)

[14] J. Bennett, R. Grout, P. Pebay, D. Roe, D. Thompson, Numerically stable, single-pass, parallel statistics algorithms, IEEE International Conference on Cluster Computing and Workshops, 2009, Oct 2009, New Orleans, LA, USA.

[15] X. Han, A. C. Berg, DCMSVM: Distributed Parallel Training For Single-Machine Multiclass Classifiers, IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2012), Providence, RI, USA, 3554-3561, June 16-21, 2012.

[16] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data, SigKDD 2015.

[17] B. Li, E. Mazur, Y. Diao, A. McGregor, P. Shenoy, A Platform for Scalable One-Pass Analytics using MapReduce SIGMOD'11, June12–16, 2011, Athens, Greece.

[18] M. Eric Otey, S. Parthasarathy, C. Wang, A. Veloso, W. Meira, Jr, Parallel and Distributed Methods for Incremental Frequent Itemset MiningIEEE Transactions on Systems, Man, and Cybernetics, Vol. 34, No. 6, December 2004.

[19] F. Rusu, C. Qin, M. Torres, "Scalable Analytics Model Calibration with Online Aggregation", IEEE Data Eng. Bull. 38(3): 30-43, 2015.

[20] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, K. Olukotun. Map-reduce for machine learning on multicore. In NIPS 2007, 3-6 Dec 2007, Vancouver, BC, Canada.

[21] S. B. Kim, K. S. Han, H. C. Rim, S. H. Myaeng, "Some effective techniques for naive bayes text classification". Knowledge and Data Engineering, IEEE Transactions 18(11), pp. 1457-1466, 2006.

[22] A. Metzger, P. Leitner, D. Ivanovic, E. Schmieders, R. Franklin, M. Carro, S. Dustdar, K. Pohl, Comparing and Combining Predictive Business Process Monitoring Techniques, IEEE Trans. on Systems, Man, & Cybernetics: Systems, Vol 45, Issue 2, Feb 2015.

[23] M. Perkusich, G. Soares, H. Almeida, A. Perkusich, A procedure to detect problems of processes in software development projects using Bayesian networks, Expert Systems with Applications, Vol 42, Issue 1, Jan 2015, Pages 437–450.

[24] B. Plattner, "Real-Time Execution Monitoring", IEEE Transactions on Software Engineering, Vol SE-10, Issue: 6, Nov. 1984.

[25] J. Lin, R. Ravichandiran, H. Bannazadeh, A. L. Garcia, Monitoring and measurement in software-defined infrastructure, IFIP/IEEE Intl Symposium on Integrated Network Management (IM), 11-15 May 2015.

[26] Z. Zhang, J. M. Chang, A cool scheduler for multi-core systems exploiting program phases. IEEE Trans. Comput 63(5), 1061–1073 (2014).

[27] S.P. Reiss, Dynamic detection and visualization of software phases. ACM SIGSOFT Softw. Eng. Notes 30(4), 1–6 (2005)

[28] Y. Qu, Q. Zheng, T. Liu, J. Li, X. Guan, In-depth measurement and analysis on densification power law of software execution, WETSoM 2014 Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics.

[29] M. Naedele, H. M. Chen, R. Kazman, Y. Cai, L. Xiao, C. V.A. Silva, Manufacturing execution systems: A vision for managing software development, Journal of Systems and Software, Volume 101, March 2015, Pages 59–68.