Cluster Computing (2020) 23:1345–1360
https://doi.org/10.1007/s10586-019-02998-y

Artificial neural networks based techniques for anomaly detection in Apache Spark

Ahmad Alnafessah[1] · Giuliano Casale[1]

Received: 11 February 2019 / Revised: 5 September 2019 / Accepted: 3 October 2019 / Published online: 23 October 2019
© The Author(s) 2019, corrected publication 2020

Abstract
Late detection and manual resolutions of performance anomalies in Cloud Computing and Big Data systems may lead to performance violations and financial penalties. Motivated by this issue, we propose an artificial neural network based methodology for anomaly detection tailored to the Apache Spark in-memory processing platform. Apache Spark is widely adopted by industry because of its speed and generality, however there is still a shortage of comprehensive performance anomaly detection methods applicable to this platform. We propose an artificial neural networks driven methodology to quickly sift through Spark logs data and operating system monitoring metrics to accurately detect and classify anomalous behaviors based on the Spark resilient distributed dataset characteristics. The proposed method is evaluated against three popular machine learning algorithms, decision trees, nearest neighbor, and support vector machine, as well as against four variants that consider different monitoring datasets. The results prove that our proposed method outperforms other methods, typically achieving 98–99% F-scores, and offering much greater accuracy than alternative techniques to detect both the period in which anomalies occurred and their type.

Keywords Performance anomalies · Apache Spark · Neural network · Big data · Machine learning · Artificial intelligence · Resilient distributed dataset (RDD)

1 Introduction

Cloud computing and Big Data technologies have become one of the most impactful forms of technology innovation [16]. Cloud Computing provides scalability [10], low start-up costs [6], and a virtually limitless IT infrastructure that can be provisioned in a short period of time [5]. The combined benefits of available computing resources and advancements in data storage encourage a significant increase in Big Data creation over the Internet, such as data from the Internet of Things (IoT), e-commerce, social networks, and multimedia, increasing the popularity of in-memory data processing technologies, such as Apache Spark [4].

Due to the widespread growth of data processing services, it is not uncommon for a data processing system to have multiple tenants sharing the same computing resources, leading to *performance anomalies* due to resource contention, failures, workload unpredictability, software bugs, and several other root causes. For instance, even though application workloads can feature intrinsic variability in execution time due to variability in the dataset sizes, uncertainty scheduling decisions of the platform, interference from other applications, and software contention from the other users can lead to unexpectedly long running times that are perceived by end-users as being anomalous.

Research on automated anomaly detection methods is important in practice since late detection and slow manual resolutions of anomalies in a production environment may cause prolonged service-level agreement violations, possibly incurring significant financial penalties [12, 40]. This leads to a demand for performance anomaly detection in cloud computing and Big Data systems that are both dynamic and proactive in nature [21]. The need to adapt these methods to production environment with very

✉ Ahmad Alnafessah
a.alnafessah17@imperial.ac.uk

Giuliano Casale
g.casale@imperial.ac.uk

[1] Imperial College London, London, UK

different characteristics means that black-box machine learning techniques are ideally positioned to automatically identify performance anomalies. These techniques offer the ability to quickly learn the baseline performance from a large space of monitoring metrics to identify normal and anomalous patterns [36].

In this paper, we develop a neural network based methodology for anomaly detection tailored to the characteristics of Apache Spark. In particular, we explore the consequences of using an increasing number and variety of monitoring metrics for anomaly detection, showing the consequent trade-offs on precision and recall of the classifiers. We also compared methods that are agnostic of the workflow of Spark jobs with a novel method that leverages the specific characteristics of Spark's fundamental data structure, the resilient distributed dataset (RDD) to improve anomaly detection accuracy.

Our experiments demonstrate that neural networks are both effective and efficient in detecting anomalies in the presence of a heterogeneous workloads and anomalies, the latter including CPU contention, memory contention, cache thrashing and context switching anomalies. We further explore the sensitivity of the proposed method against other machine learning classifiers and with multiple variations on the duration and temporal occurrence of the anomalies.

This paper extends an earlier work [2] by providing an evaluation against three popular machine learning algorithms, decision trees, nearest neighbor, and support vector machine (SVM), as well as against four variants that consider different monitoring metrics in the training dataset. In addition, the proposed methodology is examined with different types of overlapped anomalies. The rest of the paper is organized as follows: prior art and prerequisite background on in-memory technologies are given in Sect. 2, followed by a motivating example in Sect. 3. The proposed methodology of this work is presented in Sect. 4, followed by systematic evaluation in Sect. 5. Finally, Sect. 6 gives conclusions and outlines future work.

2 Background

2.1 Related work

We point to [9] and [19] for general discussions on machine learning, statistical analysis, and anomaly detection. Table 1 further shows a summary of detection techniques used in the context of cloud computing systems. Some studies have used statistical methods to detect anomalous behavior, such as Gaussian-based detection [31, 43], regression analysis [11, 23], and correlation analysis [1, 34, 38]. Many statistical techniques depend on the assumption that the data are generated from a particular distribution and can be brittle when assumptions about the distribution of the data do not hold. For example, distribution assumptions often do not hold true in cases that involve highly dimensional real-time datasets [9].

Gow et al. [17] propose a method to characterize system performance signatures. The authors explored the service measurement paradigm by utilizing a black box M/M/1 queueing model and regression curve fitting the service time-adapted cumulative distributed function. They examined how anomaly performance can be detected by tracing any changes in the regression parameters. Gow et al. [17] use probabilistic distribution of performance deviation between current and old production conditions. The authors argued that this method could be utilized to identify slow events of an application. The method that has been used by authors [17] is worth examining in our research, specifically the anomaly detection part because applying such a method is not specific to any certain n-tier architecture, which makes its methods a platform agnostic. We focus here on methods that address these limitations based on machine learning techniques such as classification, neighbor-based methods, and clustering, either with supervised or unsupervised learning approaches [21].

Gu and Wang propose a supervised Bayesian classification technique in [18] to detect anomaly indications that relate to performance anomaly root localization. They apply Bayesian classification methods to detect an anomaly and its root, alongside Markov models to detect the change in the patterns of different measurement metrics. Combining Markov modeling with Bayesian classification methods allows the prediction of anomalous behaviors that will likely occur in the future.

The local outlier factor (LOF) algorithm is a type of neighbor-based technique for unsupervised anomaly detection, as shown for cloud computing systems in [20]. The main idea is to identify anomalies by comparing the local density deviation of a data point (instance) with its neighbors. Each instance with a lower density than its neighbors is considered an anomaly.

The work in [14] considers the cloud computing system and applies principal component analysis (PCA) to reduce metric dimensions and maintain the data variance. Semi-supervised decision tree classifiers are used to reduce metric dimensionality and to identify anomalies.

Few works exist for anomaly detection in Spark. Ousterhout et al. [33] develop a method to quantify end-to-end performance bottlenecks in large-scale distributed computing systems to analyze Apache Spark performance. The authors explore the importance of disk I/O, network I/O as causes of bottlenecks. They apply their method to examine the system performance of two industry SQL benchmarks and one production workload. The approach

Table 1 Summary of the state-of-the-art techniques

References	Approach	Detection technique	System/environment
Gow et al. [17]	Statistical	Regression curve fitting the service time-adapted cumulative distributed function	Online platform and configuration agnostic
Wang et al. [42]	Statistical	Gaussian-based detection	Online anomaly detection for conventional data centers
Markou and Singh [31]	Statistical	Gaussian-based detection	General
Kelly [23]	Statistical	Regression analysis	Globally-distributed commercial web-based, application and system metrics
Cherkasova et al. [11]	Statistical	Regression analysis	Enterprise web applications and conventional data center
Agarwala et al. [1]	Statistical	Correlation	Complex enterprise online applications and distributed system
Peiris et al. [34]	Statistical	Correlation	Orleans system, distributed system and distributed cloud computing services
Sharma et al. [38]	Statistical	Virtualized cloud computing and distributed systems	Hadoop, Olio and RUBiS
Gu and Wang [18]	Machine learning	Supervised Bayesian classification	Online application for IBM S-distributed stream processing system
Huang et al. [20]	Machine learning	Unsupervised neighbor-based technique (local outlier factor algorithm)	General cloud computing system
Fu [14]	Machine learning	Semi-supervised principle component analysis and Semi-supervised Decision-tree	Institute-wide cloud computing system
Fu et al. [15]	Machine Learning	One class and two class support vector machines	Cloud computing environments
Ren et al. [35]	Machine learning	Anomaly detection approach based on stage-task behaviors and logistic regression model	Online framework for Apache Spark streaming systems
Lu et al. [29]	Machine Learning	Anomaly detection using convolutional neural networks based model	Big Data system logs using Hadoop distributed file fystem

involves analysis of blocking time, using white-box logging to measure time execution for each task in order to pinpoint bottleneck root-causes.

Support vector machines [41] algorithm is used for anomaly detection in the form of one class SVM. This algorithm uses one class to learn the regions, which contain boundary of training data instance [9]. Kernels can be used to learn complex areas. Each test instance is used to determine that instance is located inside the learned region (normal instance) or outside the learned region (anomalous). The anomaly detection techniques using SVM are used for intrusion detection [25], documents classification [30], and cloud systems [15]. Although one-class SVM is effective at making a decision from well-behaved feature vectors, it can be more expensive for modeling the variation in large datasets and high-dimensional input features [9, 13, 19].

Convolution neural networks are widely used for a variety of learning tasks. They are commonly more effective for image classification issues than fully connected feedforward neural networks. In large images, where thousands or millions of weights are needed to train the network, issues such as slow training time, overfitting, and underfitting issues can be alleviated using convolutional neural networks, which have the ability to reduce the size of input features (e.g., a matrix of image size) to lower dimensions using convolutions operations [28]. In our case, the proposed neural networks based techniques for anomaly detection in Apache Spark cluster has less number of input features and output classes than what is used in image processing classification, making less relevant the use of techniques such as convolutional neural networks.

2.2 Apache Spark

Apache Spark is a large-scale in-memory processing technology that can support both batch and stream data processing [4]. The main goal of Apache Spark is to speed up the batch processing of data through in-memory computation. Spark can be up to 100 times faster than Hadoop MapReduce for in-memory analytics [4]. The core engine of Apache Spark offers basic functionalities for in-memory

cluster computing, such as task scheduling, memory management, fault recovery, and communicating with database systems [22].

Running Spark application involves five main components, including driver programs, cluster managers, worker nodes, executor processes, and tasks as shown in Fig. 1. The Spark application runs as an independent set of processes on a cluster, which are coordinated by an object called SparkContext. This object is the entry point to Spark, and it is created in a *driver program*, which is the main function in Spark. In cluster mode, SparkContext has the ability to communicate with many cluster managers to allocate sufficient resources for the application. The cluster manager can be Mesos, YARN, or a Spark stand-alone cluster [4].

2.2.1 Resilient distributed datasets

Spark engine provides the API for the main programming data abstraction, which is the Resilient Distributed Dataset (RDD) to enable the scalability of data algorithms with high performance. RDD offers operations, including data transformation and actions, that can be used by other Spark libraries and tools for data analysis. This paper proposes an anomaly detection method that performs in its most effective instantiation anomaly detection at the level of the RDDs. We thus briefly overview the main features of these data structures and their relationship to the job execution flow within Spark.

The RDD is Spark's core data abstraction. It is an immutable distributed collection of objects that can be executed in parallel. It is resilient because an RDD is immutable and cannot be changed after its creation. An RDD is distributed because it is sent across multiple nodes in a cluster. Every RDD is further split into multiple partitions that can be computed on different nodes. This means that the higher the number of partitions, the larger parallelism will be. RDD can be created by either loading an external dataset or by paralleling an existing collection of objects in their driver programs. One simple example of creating an RDD is by loading a text file as an RDD of string (using sc.textFile()) [4].

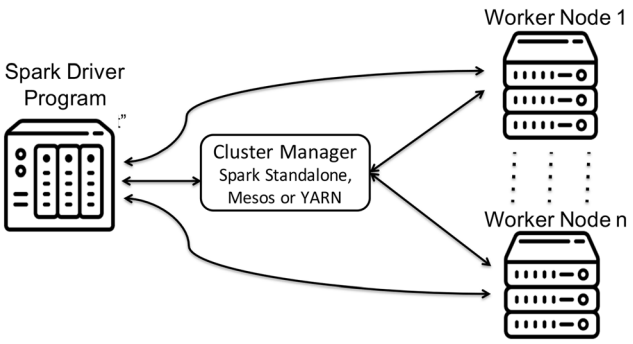


Fig. 1 Spark application components

After creation, two types of operations can be applied to RDDs: *transformations* and *actions*. A transformation creates a new RDD from an existing RDD. In addition, when applying a transformation, it does not modify the original RDD. An example of transformation operation is filtering data that returns a new RDD that meets filter conditions [37]. Some other transformation operations are map, distinct, union, sample, groupByKey, and join. The second type of RDD operation is an action, which returns a resulting value after running a computation and either returns it to the driver program or saves it to external storage, such as Hadoop Distributed File System (HDFS). A basic example of an action operation is *First()*, which returns the first element in an RDD. Other action operations are *collect, count, first, takesample,* and *foreach* [4].

RDDs are reliable and use a fault-tolerant distributed memory abstraction. Spark has the ability to reliably log the transformation operation used to build its lineage graph rather than the actual data [44]. The lineage graph keeps track of all transformations that need to be applied to RDDs and information about data location. Therefore, if some partition of an RDD is missing or damaged due to node failure, there is enough information about how it was derived from other RDDs to efficiently recompute this missing partition in a reliable way. Hence, missing RDDs can be quickly recomputed without needing costly data replication. An RDD is designed to be immutable to facilitate describing lineage graphs [44].

2.2.2 Jobs, stages, and tasks

Every Spark application consists of jobs , each job is further divided into stages that depend on each other. Each stage is then composed of a collection of tasks as shown in Fig. 2 [3].

Spark Job. A Spark job is created when an action operation (e.g., count, reduce, collect, save, etc.) is called to run on the RDD in the user's driver program. Therefore, each action operation on RDD in the Spark application will correspond to a new job. There will be as many jobs as the number of action operations occurring in the user's driver program. Thus, the user's driver program is called an application rather than a job. The job scheduler examines the RDD and its lineage graph to build a directed acyclic graph (DAG) of the stages to be executed [44].

Spark Stage. Breaking the RDD DAG at shuffle boundaries will create stages. Each stage contains many pipelined RDD transformation operations that do not require any shuffling between operations, which is called narrow dependency (e.g., map, filter, etc.). Otherwise, if

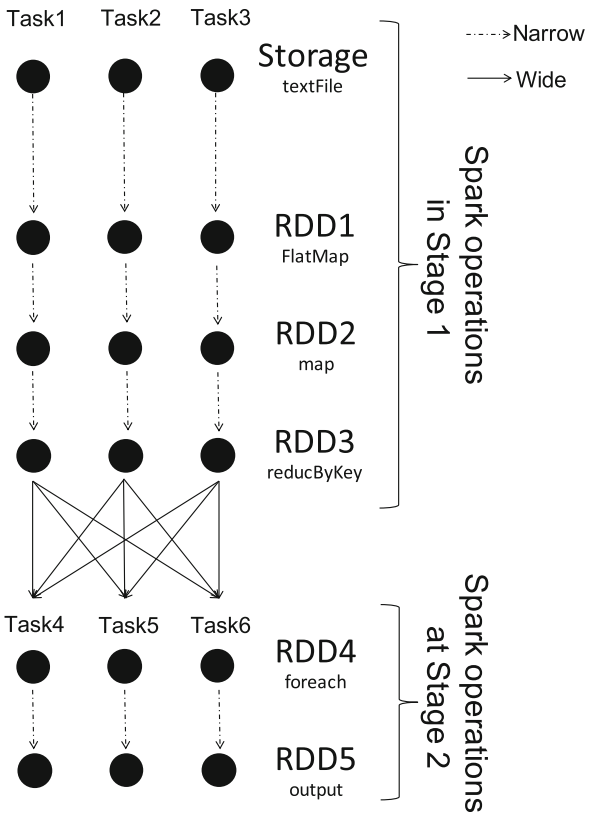


Fig. 2 Spark DAG for a WordCount application with two stages each consisting of three tasks

stages depend on each other through RDD transformation operations that require shuffling, these are called wide dependencies (e.g., group-by, join, etc.) [44]. Therefore, every stage will contain only shuffle dependencies on other stages, but not inside the same stage. The last stage inside the job generates results and the stage is executed only when its parent stages are executed. Figure 2 shows how the job is divided into two stages as a result of shuffle boundaries.

Spark Task. The stage scheduling is implemented in DAGScheduler, which computes a DAG of stages for each job and finds a minimal schedule to run that job. The DAGScheduler submits stages as a group of tasks (TaskSets) to the task scheduler to run them on the cluster via the cluster manager (e.g., Spark Standalone, Mesos or YARN) as shown in Fig. 2.

Scheduling. The task in Apache Spark is the smallest unit of work that is sent to the executor, and there is one task per RDD partition. The dependencies among stages are unknown to the task scheduler. Each TaskSet contains fully independent tasks, which can run based on the location of data and the current cached RDD. Each task is sent to one machine [3]. Inside a single stage, the number of tasks is determined by the number of the final RDD partitions in the same stage.

3 Motivating example

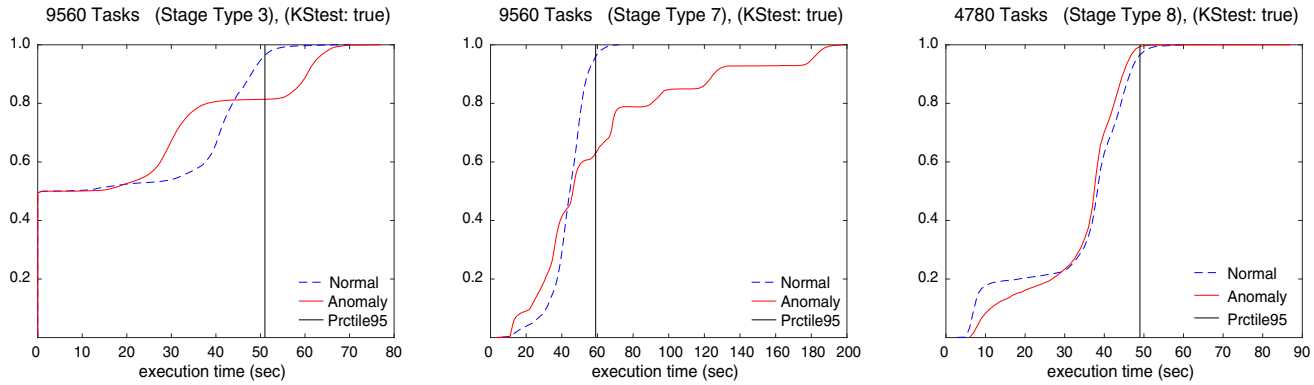
In order to motivate the use of machine learning approaches in anomaly detection methods for Spark, we consider the performance of a simple statistical detection technique based on percentiles of the cumulative distribution function (CDF) of task execution times. Our goal is to use CDF percentiles to discriminate whether a given task has experienced a performance anomaly or not.

We run a KMeans Spark workload with nine different types of tasks. More details about Spark experimental testbed and process are provided in Sect. 5.1. We inject CPU contention using the *stress* tool for a continuous period of 17 h, which corresponds to 100% of the total execution time of a job. The intensity of the CPU load injected in the system amounts to an extra 50% average utilization compared to running the same workload without *stress*.

We then use the obtained task execution times to estimate the empirical CDF for the execution time of tasks conditional on their stage; i.e., the population of samples that defines the CDF corresponds to the execution time of all tasks that executed in that specific stage. Note that since we run 10 parallel K-means workloads, each stage and its inner tasks are executed multiple times. We shall refer to this CDF as a *stage CDF*.

We then determine the 95th, 75th, 50th, 25th, and 10th percentiles of all the stage CDFs and assess whether they can be used as a threshold to declare whether a job suffered an execution time anomaly. When there is a continuous stress CPU anomaly, the F-score is 93%, which is acceptable. However, this technique failed to detect a short random time CPU anomaly by achieving only 0.2% for the F-score.

We used a two-sample Kolmogorov–Smirnov test to compare the two stages CDFs with and without anomalies [27]. The test result is true if the test rejects the null hypothesis at the 5% level, and false otherwise, as shown in Fig. 3. The three types of Spark stages in Fig. 3 illustrate three stages CDFs obtained in an experiment with and without injection of CPU contention. The three CDFs for the three different types of tasks make it difficult to determine whether there is an anomaly or not. For example, Fig. 3 has a noticeable difference in the CDFs for normal and abnormal performance. On the other hand, Fig. 3 also has a noticeable difference between the two experiments, but there were no anomalies occurred during all tasks in stage 7. In addition, the CPU anomaly causes a delay while processing the tasks. This delay propagates through the Spark DAG workflow and therefore also affects tasks that did not incur anomalies period.



(a) CDFs for stage type 3 **(b)** CDFs for stage type 7 **(c)** CDFs for stage type 8

Fig. 3 CDF for the three types of Spark tasks under a short 50% CPU stress affecting tasks in stage type 3

In conclusion, this motivating example illustrates that CDF-based anomaly detection in Spark only at the level of execution times is significantly prone to errors. In the next sections, we explore more advanced and general methodology based on a machine learning technique that is capable of considering multiple monitoring metrics and pinpointing anomalous tasks with high F-score performance metrics.

4 Methodology

In this section, we present our neural network driven methodology for anomaly detection in Apache Spark systems. A schematic view of anomaly detection detailed processes is shown in Fig. 4. The following subsections discuss the proposed methodology which covers the neural network model, feature selection, training, and testing.

4.1 Neural network model

Our methodology revolves around using a neural network to detect anomalies in Apache Spark environment. The standard backpropagation with a scaled conjugate gradient is used for the training process to update weight and bias values of the neural network. The scaled conjugate gradient training is normally faster than standard gradient descent algorithms [32].

Before we initiate the backpropagation process, we calculate the activation values of units in the hidden layer and propagate them to the output layer. A *sigmoid* transfer function (non-linear activation function) is used in the hidden layer because it exists between (0 to 1), where zero means absence of the feature and one means its presence. In neural networks, non-linearity is needed in the activation functions because it produces a nonlinear decision boundary via non-linear combinations of the weights and inputs to the neural networks. *Sigmoid* introduces non-linearity in the model of neural networks, as most of the real classification problems are non-linear. *Softmax* transfer function is used in the output layer to handle classification problems with multiple classes. Then *cross-entropy* is used as a cost function to assess the neural network performance and compare the actual output error results with the desired output values (labeled data). *Cross-entropy* is used because it has practical advantages over other cost functions; e.g., it can maintain good classification performance even for problems with limited data [24].

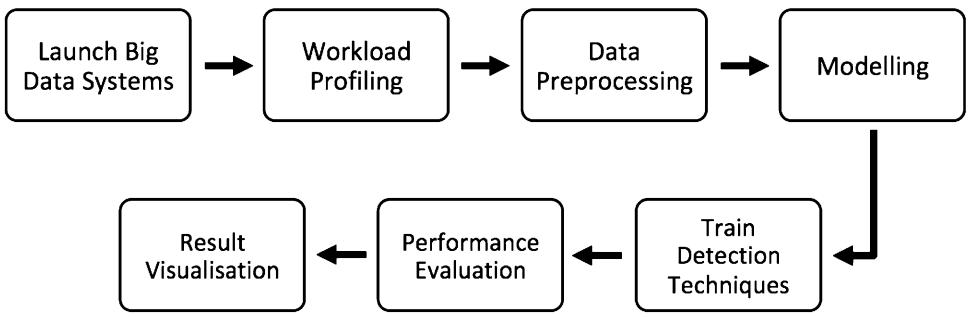


Fig. 4 Methodology for anomaly detection

4.1.1 Structure of model

The proposed neural networks contain three layers, which are input, hidden, and output layer. The input layer contains a number of neurons equal to the number of input features. The size of the hidden layer is determined by using a "trial and error" method, choosing a number between the sizes of input neurons and output neurons [39]. A hidden layer with ten neurons has achieved the most accurate results for our situation as shown in Table 2. The output layer of the neural network contains a number of neurons equal to the number of target classes (types of anomalies), where each neuron generates boolean values, which are 0 for normal behavior or 1 for anomalous behavior. For example, if there are three types of anomalies (CPU, cache thrashing, and context switching), then the size of the output layer will be three neurons and each of them outputs a boolean value.

4.2 Model training and testing

In the training process, the input data to the model is divided into three smaller subsets, called training (70%), validation (15%), and testing (15%) sets. The training set is used for calculating the gradient and updating the network weights and biases. During the training process, the weights and biases are updated continuously until the magnitude of the scaled conjugate gradient reaches the minimum gradient.

The validation set is used to avoid overfitting. The error rate during the validation phase is decreased until the magnitude of the gradient is less than a predefined threshold (e.g., 10^{-5}) or hits the maximum number of validation checks. The number of validation checks is the number of successive iterations in which the validation performance fails to decrease (we use a maximum of six successive iterations). After convergence, we save the weights and biases at the minimum error for the validation subset. The early stopping method we have described above is known to avoid overfitting issues [7].

A third subset is used for testing purposes. It is independently used to assess the ability of the trained model to be generalized. Throughout the paper, we use as the main test metric the standard *F-score* (*F*), which is defined in the Appendix alongside the standard notions of *Precision* (*P*) and *Recall* (*R*).

Table 2 Impact of hidden layer size on F-score for Neural Networks using DSM4 feature sets

Hidden layer size (neurons)	F-score
5	0.98
10	0.99
15	0.96
20	0.96

4.3 Feature selection

To evaluate the impact of the choice of input monitoring features, we consider a simple workload execution in which a K-means workload is injected with 50% CPU and memory contention overheads using the *stress* tool, either continuously for the duration of the experiment or in a 90-s period out of a total runtime execution. This includes five different scenarios, which are *Non, CPU50%, CPU50%90s, Mem50%, and Mem50%90*. First scenario *Non* is for running the benchmark without any contention on CPU and memory, second scenario CPU50 is for running the benchmark with continuous contention on CPU at 50%, third scenario *CPU50%90s* is for running the benchmark with a short time (90 s) of contention on CPU 50%, fourth scenario *Mem50%* is for running the benchmark with continuous contention on memory at 50% of free memory, and fifth scenario *Mem50%90s* is for running the benchmark with a short time (90 s) of contention on memory by 50% of free memory.

We compare the performance of a basic anomaly detection method, called DSM1, which relies solely on a neural network trained using samples collected at the operating system level of CPU utilization, time spent by the processor waiting for I/O, and CPU steal percentage. Table 3 shows a comparison of the system performance metrics among different contention scenarios on S02. The classification performance metrics for a neural network trained on this basic set of measures are summarized in Fig. 5.

The K-means workload does not heavily use memory (see Table 3). Therefore, memory contention does not have a noticeable effect on the DSM1 dataset, and the F-score is as low as 19.88% when the memory contention is temporary (see Fig. 5). This is because DSM1 does not consider the memory metrics for Spark cluster. Generally, short contention periods are harder to detect, as visible from the fact that a 90-s CPU anomaly has an F-score of 58.05%, compared to a 77.44% F-score when there is a continuous CPU stress injection. We interpret this as due to the fact that the neural network needs to train the algorithm with a bigger dataset to detect memory contention. If we repeat the same experiment after adding memory monitoring metrics, referred to as the DSM2 dataset in Table 4, the F-score immediately increases from 77.44 to 99% for continuous CPU anomaly injection, highlighting the importance of carefully selecting monitoring metrics even if they do not immediately relate to the metrics that are mostly affected by the anomaly injection.

Table 3 Running Spark K-means workload without contention(Non), with continuous 50% CPU stress (CPU50%), with 90-s 50% CPU stress (CPU50%90s), with continuous 50% memory stress (Mem50%), and with 90 s 50% memory stress on only S02 (Mem%90s)

Server	Stress	MeanCPU	SD	Pr95	Pr99	Iqr	UsedMem	ExeTimeSec
S01:Non	No	0.0203	0.0389	0.0950	0.2147	0.0177	89.3239	295
S01:CPU50%	No	0.0174	0.0308	0.0663	0.1646	0.0176	89.5402	567
S01:CPU50%90s	No	0.0210	0.0359	0.0874	0.2166	0.0218	89.8094	376
S01:Mem50%	No	0.0205	0.0376	0.0768	0.2346	0.0211	90.0187	326
S01:Mem%90s	No	0.0193	0.0356	0.0715	0.2094	0.0190	90.2926	355
S02: Non	No	0.8776	0.1849	0.9519	0.9561	0.0304	81.2464	295
S02:CPU50%	Yes	0.9510	0.0701	0.9799	0.9833	0.0158	81.7595	567
S02:CPU50%90s	Yes	0.9152	0.0806	0.9693	0.9748	0.0315	81.9844	376
S02:Mem50%	Yes	0.8656	0.1880	0.9479	0.9527	0.0318	93.2561	326
S02:Mem50%90s	Yes	0.8770	0.1825	0.9513	0.9574	0.0337	85.0864	355
S03: Non	No	0.4488	0.4443	0.9489	0.9550	0.9271	90.0702	295
S03:CPU50%	No	0.2231	0.3719	0.9361	0.9504	0.3580	90.4513	567
S03:CPU50%90s	No	0.2649	0.3572	0.8831	0.9356	0.6816	91.1414	376
S03:Mem50%	No	0.4129	0.4357	0.9422	0.9507	0.9115	91.2038	326
S03:Mem50%90s	No	0.3760	0.4310	0.9402	0.9506	0.8914	91.3892	355

It is clear that the different type and amounts of anomalies affect mean CPU and memory utilization in server S02

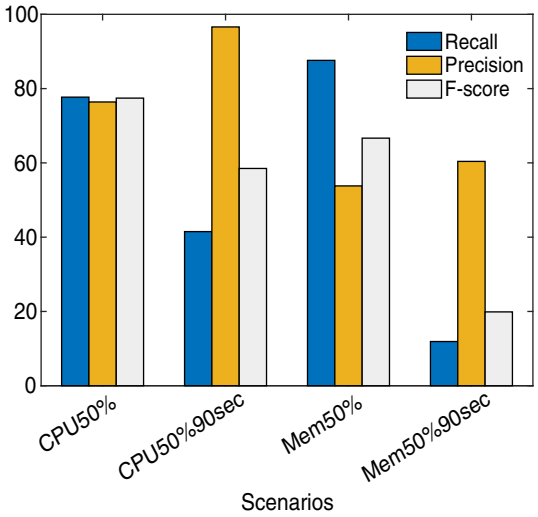


Fig. 5 Neural network performance with DSM1 feature set for experiments with basic CPU and memory contention (continuous or 90-s periods)

The above results suggest that while a reduced set of core metrics can substantially decrease the training time of the model, an important consideration for example in online applications, it can be counterproductive to perform feature selection by reasoning on the root causes that generate the anomaly.

4.4 Training data

We assume the Spark testbed to be monitored at all machines. We considered different levels of logging, ranging from basic CPU utilization readings to complete availability of Spark execution logs. The logs provide details on activities related to tasks, stages, jobs, CPU, memory, network, I/O, etc. Many metrics can be collected, but it is challenging to decide which ones are more valuable to assess system performance and pinpoint the anomalies, as this may depend on the workload. All data collection in our experiments took place in the background without causing any noticeable overhead on the Spark cluster.

In this work, we propose four methods, called dataset method 1 (DSM1), dataset method 2 (DSM2), dataset method 3 (DSM3) and dataset method 4 (DSM4). DSM1, introduced earlier, relies solely on a neural network trained using CPU utilization samples. DSM2 adds operating system memory usage metrics to the metrics employed by DSM1. The third method is DSM3 is build upon the list of metrics selected in [45], which examines the internal Spark architecture by relying on information available in the Apache Spark log, such as Spark executors, shuffle read, shuffle write, memory spill, and java garbage collection. DSM3 does not reflect the RDD DAG of Spark application. The fourth method is DSM4 which includes comprehensive internal metrics about Spark tasks that enable the proposed technique to track the Spark RDD DAG to detect the performance anomalies. These metrics include comprehensive statistics about identificators and execution timestamps for Spark RDDs, tasks, stages, jobs, and applications. The detailed monitoring features used to train these four methods are listed in Table 4.

In the proposed methodology, we assume that the collected data is pre-processed by the end to ensure elimination of any mislabeled training instances and to validate the

Table 4 List of performance metrics for the DSM1, DSM2, DSM3, and DSM4 methods

Methods		Metrics
DSM2	DSM1	CPU utilization
		Percentage of time that the CPUs were idle during outstanding disk I/O request
		Percentage of time spent in involuntary wait by the virtual CPU
		Percentage of time that the CPUs were idle
		kbmemfree: free memory in KB on hostname
		kbmemused: used memory in KB on hostname
		X.memused: used memory in % on hostname
		kbbuffers: buffer memory in KB on hostname
		kbcached: cached memory in KB on hostname
		kbcommit: committed memory in KB on hostname
		X.commit: committed memory in % on hostname
		kbactive: active memory in KB on hostname
		kbinact: inactive memory in KB on hostname
		kbdirty: dirty memory in KB on hostname
DSM4	DSM3	Task spill: Disk Bytes Spilled
		Executor Deserialize Time
		Executor Run Time
		Bytes Read: Total input size
		Bytes Written: total output size
		Garbage Collection: JVM GC Time
		Memory Bytes Spilled: Number of bytes spilled to disk
		Task Result Size
		Task Shuffle Read Metrics: Fetch Wait Time, Local Blocks Fetched, Local Bytes Read, Remote Blocks Fetched, and Remote Bytes Read
		Task Shuffle write Metrics: Shuffle Bytes Written and Shuffle Write Time
		Stage ID
		Task info: Launch Time, Finish Time, Executor CPU Time, Executor Deserialize CPU Time, Input Records Read, Output Records Written, Result Serialization Time, Total Records Read for Shuffle, and Total Shuffle Records Written

datasets before passing them to the neural networks to improve their quality. For example, we sanitize utilization measurements larger than 100% or less than 0% by removing the corresponding entries; similarly, we exclude from the datasets samples when some of the features are missing, so that the input dataset is uniform.

All the collected metrics are time series, which are additionally labeled either as normal or anomalous in a supervised fashion, before passing them as input to our anomaly detection method for training, validation, and testing. In an application scenario, labeling could either be applied using known anomalies observed in the past in production datasets or carrying out an offline training based on the forced injection of some baseline anomalies. Features we have used to qualify the characteristics of the anomalies include information on their start time, end time, and type (e.g., CPU, memory, etc.).

5 Evaluation

In this section, we introduce an evaluation for the performance anomaly detection methodology proposed in Sect. 4. In particular, having shown before the benefits of using an increasingly large dataset, we focus on evaluating neural networks trained on the DSM2 and DSM4 feature sets. We use as a baseline a nearest neighbor classifier trained on the same data.

5.1 Experimental testbed

Experiments are conducted on a cluster that contains three physical servers: S01, S02, and S03. The specifications for these servers are as follows:

1. Node S01: 16 vcores Intel(R) Xeon(R) CPU 2.30GHz, 32 GB RAM, Ubuntu 16.04.3, and 2TB Storage.
2. Node S02: 20 vcores × Intel(R) Xeon(R) CPU 2.40GHz, 32 GB RAM, Ubuntu 16.04.3, and 130 GB Storage.
3. Node S03: 16 vcores × Intel(R) Xeon(R) CPU 1.90GHz, 32 GB RAM, Ubuntu 16.04.3, and 130 GB Storage.

The hyperthreading option is enabled on S01, S02, and S03 to make a single physical processor resources appear as two logical processors. Apache Spark is deployed such that S01 is a master and the other two servers are slaves (workers). Spark is configured to use the Spark Standalone Cluster Manager, 36 executors, FIFO scheduler, and a client mode for deployment. Node S01 hosts the benchmark to generate the Spark workload and launch Spark jobs. The other nodes run the 36 executors. Monitoring data collection took place in the background, with no significant overhead on the Spark system. All machines use *sar* (System Activity Reporter) and *Sysstat* to collect CPU, memory, I/O, and network metrics. Log files from Spark are also collected to later extract the metrics for DSM4.

5.2 Workload generation

SparkBench provides workload suites that include a collection of workloads that can be run either serially or in parallel [26]. Workloads include machine learning, graph computation, and SQL queries, as shown in Table 5. In this section, the K-means data generator is used to generate various K-means datasets of different sizes (e.g., 2 GB, 8 GB, 32 GB, and 64 GB), a default number of clusters ($K = 2$), and a seed value 127L. The K-means workload is intensively used in our experiments with many alternative configurations for Spark and SparkBench parameters to compare the performance results under different scenarios. More than 1450 experiments have been conducted and more than 3.7TB of data have been collected to examine our proposed solution. An example of RDD DAG for K-means Spark job is shown in Fig. 6, which has a single stage that contains a sequence of RDD operations (e.g., Scan csv, DeserializeToObject, mapPartitions, etc.). These RDDs operations depend on each other and some may be cached.

5.3 Anomaly injection

Node S02 is used to inject anomalies into the Apache Spark computing environment using *stress* and *stress-ng* tools. Table 6 shows a list of the four types of anomalies that have been used throughout the experiments. *Stress* is used to generate memory anomalies, whereas *stress-ng* is used to generate CPU, cache thrashing, and context switching. Each experiment has different configurations, depending on the objective of the conducted experiment, which will be discussed in detail in the following (Sect. 5.4).

Table 5 SparkBench workloads

Application type	Workloads
Graph computation	Data generator
	Graph generator
SQL queries	SQL query over dataset
Machine learning	Data generator—K-means
	Data generator—linear regression
	K-means
	Logistic regression

5.4 Results

The experiments are conducted on a cluster (described in Sect. 5.1), which consisted of one master server (called S1) and two slave servers (called S02 and S03). This cluster was isolated from other users during the experiments. A physical cluster was used instead of a virtual cluster to avoid any possibility of deviations in measurements. A series of experiments are conducted on the Spark cluster to evaluate the proposed anomaly detection technique.

5.4.1 Baseline experiment

Three experiments with different types of anomalies are injected into the Spark cluster with random instant and

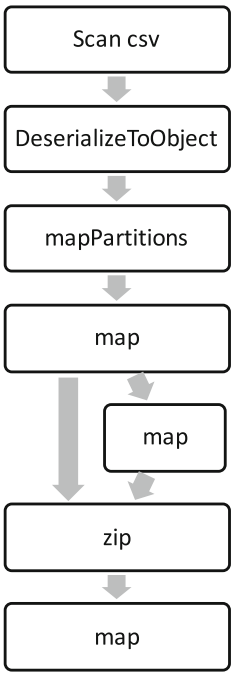


Fig. 6 DAG diagram illustrates dependencies among operations on Spark RDDs for a single Spark stage within the K-means workload

Table 6 Types of anomalies

Type #	Description
CPU	Spawn n workers running the *sqrt()* function
Memory	Continuously writing to allocated memory in order to cause memory stress
Cache thrashing	n processes perform random widespread memory read and writes to thrash the CPU cache
Context switching	n processes force context switching

random duration chosen uniformly between 0 and 240 s. Each experiment encompasses a single type of anomaly: CPU contention, cache thrashing, or context switching. The average number of samples that are used to train and test model for every experiment is 64K samples. We focus on evaluating neural networks trained on the DSM2 and DSM4 feature sets. Figure 7 shows the F-score obtained with the proposed neural network classifier versus the nearest neighbor method used as a baseline. It is clear that the neural network outperforms the nearest neighbor algorithm in detecting all the three types of anomalies. Moreover, the random instant and random duration of the three types of anomalies have little impact on the performance of the neural networks compared with the nearest neighbor.

5.4.2 Sensitivity to training set size

Figure 8 depicts the impact of Spark workload size on the F-score for anomaly detection using DSM4 and four different types of algorithms, which include Neural Networks, Decision Tree, Nearest Neighbor, and SVM. The first workload has 250 Spark tasks (micro), the second workload has 1K Spark tasks (small), the third workload has 4K Spark tasks (medium), the fourth workload has 16K Spark tasks (large), and the fifth workload has 64K Spark tasks (x-large). All these workloads have the same benchmark and spark configuration. Figure 8 shows that the proposed technique achieved 85% F-score with a micro Spark workload (200 tasks), whereas the F-score increased when the size of workload increased to reach 99% F-score for the x-large Spark workload. This proves that the neural networks achieve higher F-score than Decision Tree, Nearest Neighbor, and SVM even with more heavy Spark workload.

5.4.3 Sensitivity to parallelism and input data sizes

In this section, we consider the execution of ten parallel K-means workloads at the same time. This represents a more complex scenario than the ones considered before since the anomalies are overlapped to resource contention effects, making it difficult for classifiers to discern whether a heightened resource usage is due to the workload itself or an exogenous anomaly. As before, the workload input data size is 64 GB and we consider 50% CPU contention injection into the Spark cluster. Figure 9 shows the minor impact on DSM2 and DSM4 when there are a single K-means workload and 10 parallel K-means workloads with continuous CPU contention.

Each experiment took approximately 17 h for execution. In order to evaluate the proposed anomaly detection methods, four machine learning algorithms have been applied to detect performance anomalies with DSM2 and DSM4 as inputs to the anomaly detection methods. These algorithms include neural networks, decision tree, nearest neighbor, and SVM. Figure 9 shows that the neural network has the highest F-score, and it selectively detects the anomalies in the Apache Spark cluster. The nearest neighbor has the second highest F-score, then the decision tree and SVM respectively. Regarding the execution time of each algorithm, the neural network, decision tree, nearest neighbor, and SVM took approximately 1 min, 3 min, 9 min, and 19 min respectively. The neural network is more effective than other algorithms. The results in Fig. 9 prove that the neural network is more robust than the other three algorithms, which are affected by the size of the input

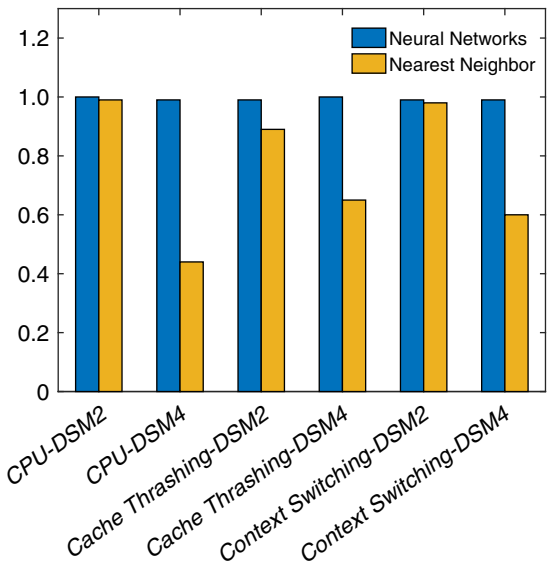


Fig. 7 F-score performance metrics of neural networks and nearest neighbor under various scenarios

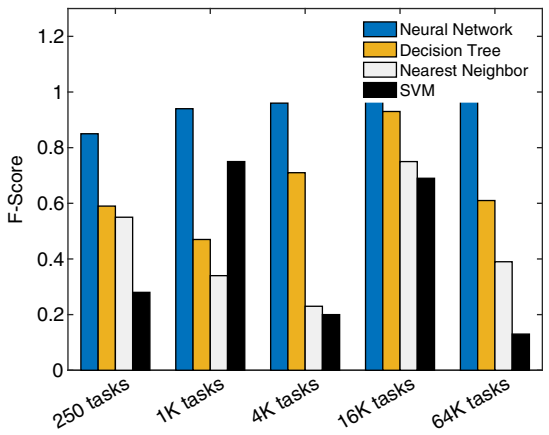
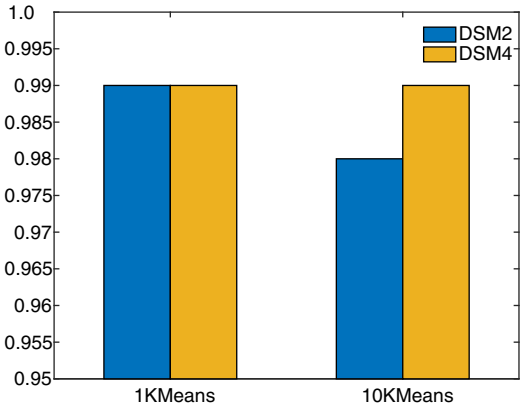
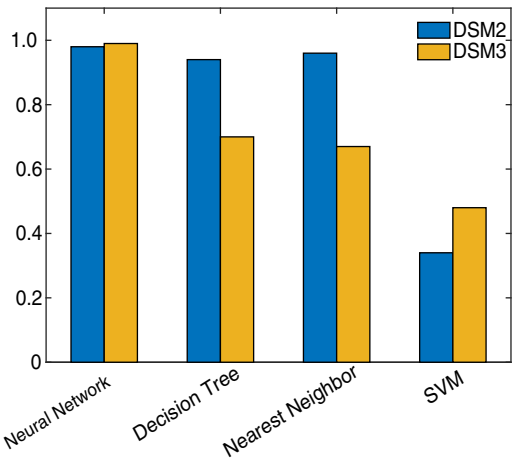


Fig. 8 Impact of workload size on F-score for Neural Networks, Decision Tree, Nearest Neighbor, and SVM using DSM4 feature sets



(a) Comparison between DSM2 and DSM4 using neural networks when there are a single K-means workload and 10 parallel K-means workloads with continuous CPU contention



(b) Performance metrics for machine learning algorithms with 50%CPU contention on only S02 with comparison between DSM2 and DSM4

Fig. 9 Impact of parallelism and input data size of workload on anomaly detection methods

data to workloads when the input data was increased to 64 GB.

5.4.4 Classifying anomaly types

In this experiment we assess the ability of the proposed technique not only to detect that an experiment has suffered an anomaly, but also to qualify the type of anomaly. In this experiment we consider simultaneous injection of CPU, cache thrashing, and context switching anomalies. The classification therefore has four classes: normal, CPU anomalies, cache thrashing anomalies, and context switching anomalies. The classification is at the level of individual Spark tasks.

Table 7 Classification of anomaly types using DSM3 and DSM4

DSM3: neural network	R	P	F
Normal	0.99	0.81	0.89
CPU	0.21	0.97	0.34
Cache thrashing	0.34	0.81	0.47
Context switching	0.38	0.96	0.54
Average F-score	0.48	0.88	0.56
DSM3: nearest neighbor	**R**	**P**	**F**
Normal	0.87	0.83	0.85
CPU	0.36	0.45	0.40
Cache thrashing	0.29	0.30	0.29
Context switching	0.16	0.15	0.16
Average F-score	0.42	0.43	0.42
DSM4: neural network	**R**	**P**	**F**
Normal	1	1	1
CPU	1	1	1
Cache thrashing	0.97	1	0.98
Context switching	0.98	0.99	0.98
Average F-score	0.98	0.99	0.99
DSM4: nearest neighbor	**R**	**P**	**F**
Normal	0.98	0.98	0.98
CPU	1.00	1.00	1.00
Cache thrashing	0.76	0.73	0.75
Context switching	0.09	0.09	0.09
Average F-score	0.71	0.70	0.70

R recall, *P* precision, and *F* F-score

The total number of Spark tasks collected during the execution amount to a total of 400K tasks. Table 7 illustrates that DSM4 with the neural network algorithm outperform DSM3 and nearest neighbor technique, retaining a 99% F-score, whereas the nearest neighbor algorithm achieves only a 70% F-score.

5.4.5 Classifying overlapped anomalies

Because many types of anomalies may occur at the same random time from different sources and for various reasons in complex systems, there is a vital need to go beyond detection of a single type of anomaly. To offer a solution for such need, the proposed technique is validated with DSM4 to prove its capability to detect overlapped anomalies when they occur at the same time. We trained our model over many Spark workloads with a total number of 950K Spark tasks. The proposed technique classifies the

Spark performance into seven types: normal, CPU stress, cache stress, context switching stress, CPU and cache stress, CPU and context switching stress, and cache and context switching stress. The proposed solution is validated with two types of Spark workload: K-means and SQL workload, as shown in Tables 8 and 9. The overall F-score for classifying the Spark performance using Neural Networks and DSM4 is 98%. Finally, it is evident that the proposed technique is capable of detecting and classifying the three types of anomalies with more complex scenarios such as parallel workload, random occurrence and overlapped anomalies. DSM4 is more agile and has the ability not only to detect anomalies, but also to classify them and find the affected Spark task, which is hard to do with DSM2 and DSM3 without having comprehensive access to the Spark logs.

The conducted experiments and the obtained results show interesting implications that prove the importance of utilizing memory performance metrics and the internal metrics of Apache Spark architecture. After adding memory monitoring metrics, referred to as the DSM2 dataset in Table 3, the F-score of anomaly detection readily increases from 77.44% to 99% (as discussed in Sect. 4.3) for CPU anomaly injection, highlighting the importance of carefully selecting monitoring metrics, even if they are not intuitive to relate to the anomaly. Another implication includes the importance of optimizing the use of the internal features of Spark architecture and dependencies between RDDs, as done in the DSM4 dataset in Table 4, and its components to accurately detect and classify anomalous behaviors based on the Spark resilient distributed dataset (RDD) characteristics.

6 Conclusion

Although Apache Spark is developing gradually, currently there is still a shortage of anomaly detection methods for performance anomalies for such in-memory Big Data technologies. This paper addresses this challenge by developing a neural network driven methodology for anomaly detection based on knowledge of the RDD characteristics.

Our experiments demonstrate that the proposed method works effectively for complex scenarios with multiple types of anomalies, such as CPU contention, cache thrashing, and context switching anomalies. Moreover, we have shown that a random start instant, a random duration, and overlapped anomalies do not have a significant impact on the performance of the proposed methodology.

The current methodology requires a centralized node that runs the neural network, which may not be effective for large scale data centers. Distributed online detection techniques that rely on a collection of neural networks may be considered for large scale systems. Due to the limitation on the hardware resources and to validate the proposed methodology, the current artificial neural networks

Table 8 Classification of 7 overlapped anomalies using DSM3 and DSM4: K-means workload

DSM3: neural networks	R	P	F
Normal	0.99	0.80	0.88
CPU	0.26	0.84	0.40
Cache thrashing	0.23	0.67	0.34
Context switching	0.36	0.95	0.52
CPU + cache	0.28	0.94	0.43
CPU + context switching	0.25	0.78	0.38
Cache + context switching	0.24	0.83	0.37
Average F-score	0.37	0.83	0.48
DSM3: nearest neighbor	R	P	F
Normal	0.80	0.77	0.78
CPU	0.20	0.25	0.22
Cache thrashing	0.11	0.11	0.11
Context switching	0.16	0.16	0.16
CPU + cache	0.18	0.19	0.19
CPU + context switching	0.15	0.15	0.15
Cache + context switching	0.15	0.15	0.15
Average F-score	0.25	0.25	0.25
DSM4: neural network	R	P	F
Normal	1	1	1
CPU	1	1	1
Cache thrashing	0.98	0.98	0.98
Context switching	0.94	0.99	0.96
CPU + cache	0.95	1	0.97
CPU + context switching	0.91	0.96	0.93
Cache + context switching	0.99	0.99	0.99
Average F-score	0.97	0.99	0.98
DSM4: nearest neighbor	R	P	F
Normal	0.84	0.84	0.84
CPU	0.50	0.50	0.50
Cache thrashing	0.06	0.06	0.06
Context switching	0.12	0.12	0.12
CPU + cache	0.13	0.13	0.13
CPU + context switching	0.10	0.10	0.10
Cache + context switching	0.12	0.12	0.12
Average F-score	0.27	0.28	0.27

R recall, *P* precision, and *F* F-score

Table 9 Classification of 7 overlapped anomalies using DSM3 and DSM4: SQL workload

DSM3: neural networks	R	P	F
Normal	0.67	0.57	0.62
CPU	0.45	0.65	0.53
Cache thrashing	0.42	0.51	0.46
Context switching	0.66	0.28	0.39
CPU + cache	0.04	0.29	0.07
CPU + context switching	0.21	0.24	0.22
Cache + context switching	0.26	0.27	0.26
Average F-score	0.39	0.40	0.37
DSM3: nearest neighbor	R	P	F
Normal	0.33	0.33	0.33
CPU	0.16	0.16	0.16
Cache thrashing	0.16	0.15	0.16
Context switching	0.17	0.17	0.17
CPU + cache	0.16	0.16	0.16
CPU + context switching	0.07	0.07	0.07
Cache + context switching	0.08	0.08	0.08
Average F-score	0.16	0.16	0.16
DSM4: neural network	R	P	F
Normal	1	1	1
CPU	1	0.99	0.99
Cache thrashing	0.98	1	0.99
Context switching	1	0.98	0.99
CPU + cache	1	1	1
CPU + context switching	0.97	1	0.98
Cache + context switching	1	0.97	0.98
Average F-score	0.99	0.99	0.99
DSM4: nearest neighbor	R	P	F
Normal	0.50	0.50	0.50
CPU	0.30	0.30	0.30
Cache thrashing	0.60	0.55	0.57
Context switching	0.47	0.47	0.47
CPU + cache	0.30	0.30	0.30
CPU + context switching	0.12	0.12	0.12
Cache + context switching	0.15	0.15	0.15
Average F-score	0.35	0.34	0.34

R recall, *P* precision, and *F* F-score

algorithm has been trained on offline data, which can easily generalize it to work with the online Spark systems.

In terms of future work, it would be interesting to explore online anomaly detection. Deep Learning techniques may also be explored to learn more about complex features from the performance metrics of the Spark system, possibly leading to even more accurate detection and prediction of critical anomalies.

Acknowledgements This research is funded by King Abdulaziz City for Science and Technology (KACST) in Saudi Arabia and partly by the European Union's Horizon 2020 research and innovation program under grant agreement No. 825040 (RADON).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Precision, Recall, and F-score

Sensitivity and *Precision* measures are used to evaluate the anomaly detection classifiers, which are standard metrics for quantifying the accuracy of the classifiers [8]. The following are the anomaly classification classes and their notations: true positives (*tp*), true negative (*tn*), false positives (*fp*), and false negatives (*fn*).

Throughout the paper, we use as main test metric the F-score (*F*), which is defined as follows:

$$R = \frac{tp}{tp + fn} \qquad P = \frac{tp}{tp + fp} \qquad F = 2\frac{PR}{P + R} \tag{1}$$

where *R* is the *Recall*, which assesses the quality of a classifier in recognizing positive samples, and *P* is *Precision*, which quantifies how many samples classified as anomalies are indeed anomalies. *Recall* will become high when the anomaly-detection method can detect all anomalies. The *Precision* assesses the reliability of the detection method when it reports anomalies. The trade-off between the *Recall* and *Precision* is captured by the *F-score*, which is a summary score, and it is computed as the harmonic mean of *Recall* and *Precision*.

References

1. Agarwala, S., Alegre, F., Schwan, K., Mehalingham, J.: E2eprof: Automated end-to-end performance management for enterprise systems. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pp. 749–758 IEEE. (2007)
2. Alnafessah, A., Casale, G.: A neural-network driven methodology for anomaly detection in apache spark. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 201–209 (2018). https://doi.org/10.1109/QUATIC.2018.00038
3. Apache Spark™: DAGScheduler. https://github.com/apache/spark/ (2018). Accessed 25 Nov 2018

4. Apache Spark ™: Lightning-fast unified analytics engine. https://spark.apache.org (2018). Accessed 1 Nov 2018
5. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
6. Buyya, R., Srirama, S.N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., Gelenbe, E., Javadi, B., Vaquero, L.M., Netto, M.A., et al.: A manifesto for future generation cloud computing: Research directions for the next decade. (2017) arXiv preprint arXiv:1711.09123
7. Caruana, R., Lawrence, S., Giles, C.L.: Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. In: Advances in Neural Information Processing Systems, pp. 402–408 (2001)
8. Casale, G., Ragusa, C., Parpas, P.: A feasibility study of host-level contention detection by guest virtual machines. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), vol. 2, pp. 152–157. IEEE (2013)
9. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: a survey. ACM Comput. Surv. **41**(3), 15 (2009)
10. Chen, Y., Sion, R.: To cloud or not to cloud?: musings on costs and viability. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 29. ACM (2011)
11. Cherkasova, L., Ozonat, K., Mi, N., Symons, J., Smirni, E.: Automated anomaly detection and performance modeling of enterprise applications. ACM Trans. Comput. Syst. **27**(3), 6 (2009)
12. Dean, D.J., Nguyen, H., Gu, X.: Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In: Proceedings of the 9th International Conference on Autonomic Computing, pp. 191–200. ACM (2012)
13. Erfani, S.M., Rajasegarar, S., Karunasekera, S., Leckie, C.: High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. Pattern Recognit. **58**, 121–134 (2016)
14. Fu, S.: Performance metric selection for autonomic anomaly detection on cloud computing systems. In: 2011 IEEE on Global Telecommunications Conference (GLOBECOM 2011), pp. 1–5. IEEE (2011)
15. Fu, S., Liu, J., Pannu, H.: A hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines. In: International Conference on Advanced Data Mining and Applications, pp. 726–738. Springer, New York (2012)
16. Gartner: Gartner Says by 2020 "Cloud Shift" Will Affect More Than \$1 Trillion in IT Spending. https://www.gartner.com/en/newsroom/press-releases/2016-07-20-gartner-says-by-2020-cloud-shift-will-affect-more-than-1-trillion-in-it-spending. Accessed 10 Jan 2018
17. Gow, R., Venugopal, S., Ray, P.K.: 'The tail wags the dog': a study of anomaly detection in commercial application performance. In: Proceedings—IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS pp. 355–359 (2013). https://doi.org/10.1109/MASCOTS.2013.51
18. Gu, X., Wang, H.: Online anomaly prediction for robust cluster systems. In: 2009 IEEE 25th International Conference on Data Engineering, pp. 1000–1011. IEEE (2009)
19. Hodge, V.J., Austin, J.: A survey of outlier detection methodologies. Artif. Intell. Rev. **22**(2), 85–126 (2004)
20. Huang, T., Zhu, Y., Zhang, Q., Zhu, Y., Wang, D., Qiu, M., Liu, L.: An lof-based adaptive anomaly detection scheme for cloud computing. In: 2013 IEEE 37th Annual on Computer Software and Applications Conference Workshops (COMPSACW), pp. 206–211. IEEE (2013)
21. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Comput. Surv. **48**(1), 1–35 (2015). https://doi.org/10.1145/2791120
22. Karau, H., Konwinski, A., Wendell, P., Zaharia, M.: Learning Spark: Lightning-Fast Big Data Analysis. O'Reilly Media Inc, Sebastopol (2015)
23. Kelly, T.: Detecting performance anomalies in global applications. WORLDS **5**, 42–47 (2005)
24. Kline, D.M., Berardi, V.L.: Revisiting squared-error and cross-entropy functions for training neural network classifiers. Neural Comput. Appl. **14**(4), 310–318 (2005)
25. Li, K.L., Huang, H.K., Tian, S.F., Xu, W.: Improving one-class svm for anomaly detection. In: Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693), vol. 5, pp. 3077–3081. IEEE (2003)
26. Li, M., Tan, J., Wang, Y., Zhang, L., Salapura, V.: Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, p. 53. ACM (2015)
27. Lilliefors, H.W.: On the Kolmogorov-Smirnov test for normality with mean and variance unknown. J. Am. Stat. Assoc. **62**(318), 399–402 (1967)
28. Liu, N., Wan, L., Zhang, Y., Zhou, T., Huo, H., Fang, T.: Exploiting convolutional neural networks with deeply local description for remote sensing image classification. IEEE Access **6**, 11215–11228 (2018). https://doi.org/10.1109/ACCESS.2018.2798799
29. Lu, S., Wei, X., Li, Y., Wang, L.: Detecting anomaly in big data system logs using convolutional neural network. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pp. 151–158. IEEE (2018)
30. Manevitz, L.M., Yousef, M.: One-class SVMS for document classification. J. Mach. Learn. Res. **2**, 139–154 (2001)
31. Markou, M., Singh, S.: Novelty detection: a review-part 1: statistical approaches. Signal Process. **83**(12), 2481–2497 (2003)
32. Møller, M.F.: A scaled conjugate gradient algorithm for fast supervised learning. Neural Netw. **6**(4), 525–533 (1993)
33. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.G., ICSI, V.: Making sense of performance in data analytics frameworks. In: NSDI, vol. 15, pp. 293–307 (2015)
34. Peiris, M., Hill, J.H., Thelin, J., Bykov, S., Kliot, G., Konig, C.: Pad: Performance anomaly detection in multi-server distributed systems. In: 2014 IEEE 7th International Conference on Cloud Computing, pp. 769–776. IEEE (2014)
35. Ren, R., Tian, S., Wang, L.: Online anomaly detection framework for spark systems via stage-task behavior modeling. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 256–259. ACM (2018)
36. Rogers, S., Girolami, M.: A First Course in Machine Learning, 2nd edn. Chapman and Hall/CRC, New York (2016)
37. Sakr, S.: Big Data 2.0 Processing Systems: A Survey, 1st edn. Springer, New York (2016)
38. Sharma, B., Jayachandran, P., Verma, A., Das, C.R.: CloudPD: problem determination and diagnosis in shared dynamic clouds. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–12. IEEE (2013)
39. Sheela, K.G., Deepa, S.N.: Review on methods to fix number of hidden neurons in neural networks. Math. Probl. Eng. **2013** (2013)
40. Tan, Y., Nguyen, H., Shen, Z., Gu, X., Venkatramani, C., Rajan, D.: Prepare: Predictive performance anomaly prevention for

virtualized cloud systems. In: 2012 IEEE 32nd International Conference on Distributed Computing Systems, pp. 285–294 (2012). https://doi.org/10.1109/ICDCS.2012.65
41. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, New York (2013)
42. Wang, C., Viswanathan, K., Choudur, L., Talwar, V., Satterfield, W., Schwan, K.: Statistical techniques for online anomaly detection in data centers. In: 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, pp. 385–392. IEEE (2011)
43. Wang, T., Zhang, W., Wei, J., Zhong, H.: Workload-aware online anomaly detection in enterprise applications with local outlier factor. In: 2012 IEEE 36th Annual Computer Software and Applications Conference, pp. 25–34. IEEE (2012)
44. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)
45. Zheng, P., Lee, B.C.: Hound: causal learning for datacenter-scale straggler diagnosis. Proc. ACM Meas. Anal. Comput. Syst. **2**(1), 17 (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ahmad Alnafessah Ahmad Alnafessah is currently a Ph.D. student at the Department of Computing, Imperial College London. Previously, he was a senior academic researcher at the National Centre for AI and Big Data Technologies KACST from 2012 to 2017. His research focuses on performance engineering for big data systems, with a specific focus on in-memory platforms. I am interested in big data systems, AI, IoT, HPC, complex distributed systems and cloud computing.

Giuliano Casale Giuliano Casale joined the Department of Computing at Imperial College London in 2010, where he is currently a Senior Lecturer in modeling and simulation. Previously, he worked as a scientist at SAP Research UK and as a consultant in the capacity planning industry. He teaches and does research in performance engineering, cloud computing, and Big data, topics on which he has published more than 100 refereed papers. He has served on the technical program committee of over 80 conferences and workshops and as co-chair for several conferences in the area of performance engineering such as ACM SIGMETRICS/Performance. His research is recipient of multiple awards, recently the best paper award at ACM SIGMETRICS 2017. He serves on the editorial boards of IEEE TNSM and ACM TOMPECS and as chair of ACM SIGMETRICS.