Imperial College London
Department of Computing

# Artificial Intelligence Driven Anomaly Detection for Big Data Systems

Ahmad Alnafessah

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London, 11 November 2021

# Declaration of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own, except where otherwise stated.

# Copyright Statement

## Abstract

The main goal of this thesis is to contribute to the research on automated performance anomaly detection and interference prediction by implementing Artificial Intelligence (AI) solutions for complex distributed systems, especially for Big Data platforms within cloud computing environments. The late detection and manual resolutions of performance anomalies and system interference in Big Data systems may lead to performance violations and financial penalties. Motivated by this issue, we propose AI-based methodologies for anomaly detection and interference prediction tailored to Big Data and containerized batch platforms to better analyze system performance and effectively utilize computing resources within cloud environments. Therefore, new precise and efficient performance management methods are the key to handling performance anomalies and interference impacts to improve the efficiency of data center resources.

The first part of this thesis contributes to performance anomaly detection for in-memory Big Data platforms. We examine the performance of Big Data platforms and justify our choice of selecting the in-memory Apache Spark platform. An artificial neural network-driven methodology is proposed to detect and classify performance anomalies for batch workloads based on the RDD characteristics and operating system monitoring metrics. Our method is evaluated against other popular machine learning algorithms (ML), as well as against four different monitoring datasets. The results prove that our proposed method outperforms other ML methods, typically achieving 98–99% F-scores. Moreover, we prove that a random start instant, a random duration, and overlapped anomalies do not significantly impact the performance of our proposed methodology.

The second contribution addresses the challenge of anomaly identification within an in-memory streaming Big Data platform by investigating agile hybrid learning techniques. We develop TRACK (neural neTwoRk Anomaly deteCtion in sparK) and TRACK-Plus, two methods to efficiently train a class of machine learning models for performance anomaly detection using a fixed number of experiments. Our model revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve high accuracy. The objective is to accelerate the search process for finding the size of the training dataset, optimizing neural network configurations, and improving the performance

of anomaly classification. A validation based on several datasets from a real Apache Spark Streaming system is performed, demonstrating that the proposed methodology can efficiently identify performance anomalies, near-optimal configuration parameters, and a near-optimal training dataset size while reducing the number of experiments up to 75% compared with naïve anomaly detection training.

The last contribution overcomes the challenges of predicting completion time of containerized batch jobs and proactively avoiding performance interference by introducing an automated prediction solution to estimate interference among colocated batch jobs within the same computing environment. An AI-driven model is implemented to predict the interference among batch jobs before it occurs within system. Our interference detection model can alleviate and estimate the task slowdown affected by the interference. This model assists the system operators in making an accurate decision to optimize job placement. Our model is agnostic to the business logic internal to each job. Instead, it is learned from system performance data by applying artificial neural networks to establish the completion time prediction of batch jobs within the cloud environments. We compare our model with three other baseline models (queueing-theoretic model, operational analysis, and an empirical method) on historical measurements of job completion time and CPU run-queue size (i.e., the number of active threads in the system). The proposed model captures multithreading, operating system scheduling, sleeping time, and job priorities. A validation based on 4500 experiments based on the DaCapo benchmarking suite was carried out, confirming the predictive efficiency and capabilities of the proposed model by achieving up to 10% MAPE compared with the other models.

## Acknowledgements

The work described in this thesis is the product of years of collaboration, mentorship, friendship, and support from a number of people.

First and foremost, I would like to express my sincere gratitude to my academic supervisor, Giuliano Casale, for being so generous with his time, experience, and wisdom in guiding me through my PhD journey. His ability to derive sharp insights from my mess of thoughts was an invaluable resource that continues to inspire me to be a better and more thoughtful researcher. He inspires me to think broadly and come up with brilliant research ideas while giving me the freedom under his guidance to find the research direction that interests me the most. Giuliano has taught me how to be an effective communicator and how to lead. His advice on research and life will stay in my heart forever.

I have been so lucky to have an exemplary and visionary mentor such as H.E. Abdullah Alswaha, the chairman of the board of directors of KACST and the Saudi Minister of Communications and Information Technology. It would be impossible to count all the ways that he inspired me to push forward toward my future career. I appreciate and treasure everything he has taught me. I only hope I can return the favor sometime in the future.

I would like to sincerely thank H.E. Dr. Munir M. Eldesouki for his treasured support and advice. I would like to extend my sincere thanks to KACST for selecting me as the recipient of the generous KACST Scholarship. The work in this thesis is supported by KACST. Later work in this thesis is partially supported by the RADON project, funded by the European Union's Horizon Research and Innovation Program.

I am also grateful to the many QORE lab mates and collaborators who I worked with them. This includes the members from the University of Rome Tor Vergata Italy, Xlab, Athens Technology Center (ATC), the University of Stuttgart, and everyone else I have collaborated with over the years.

I would like to express my deepest gratitude to my family. My dedicated parents, Saleh Alnafessah and Latifah Alkhataf, have given me their unconditional love and unwavering support from thousands of miles away, motivating me to be persistent and to keep trying.

My beloved wife, Nihal, who has gone through all ups and downs of life with me in the last few years, thank you for all your support, understanding, and encouragements. I have shared so many moments of successes, failures, smiles, and tears along this journey with you. I am so grateful for having you in my life. My two beautiful daughters, Latifa and Ladon, have always inspired me to try to make a better version of myself as a human and allowed me to grow with them. Thank you all for your unconditional love. I promise I will always be there to support you and guide you every step of the way.

## Dedication

This thesis is dedicated to my loving father, Saleh Alnafessah, and my mother, Latifah Alkhataf. Saleh's words of encouragement and push for tenacity still ring in my ears. Latifah has always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve. God bless them.

‘You can never cross the ocean until you have the courage to lose sight of the shore.’

*Christopher Columbus*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Problem Overview

Artificial Intelligence (AI), cloud computing, and Big Data technologies have recently become the most impactful forms of technology innovation. According to Gartner, the usage of cloud-based Artificial Intelligence will increase by five times from 2019, making AI one of the top cloud services by 2023. In 2022, more than $362 billion in IT spending will be directly or indirectly allocated toward the shift to cloud computing services during the next years [1]. This transition will make cloud computing technology one of the most significant forms of IT spending since the early days of the digital age [2]. In this context, the term "cloud computing" refers to the applications delivered as services over the Internet and to the hardware and software in the data centers that provide those services [3]. This type of computing is ultimately attractive because it enables an organization to have a flexible operational management model that can be characterized by an on-demand computing paradigm, one that is based on a pay-per-use pricing model. In addition, cloud computing provides scalability [4], low start-up costs, and a limitless IT infrastructure in a short period of time [3]. These benefits of available computing resources and advancements in data storage have led to the significant increase in Big Data creation over the Internet, such as data from the Internet of Things (IoT), e-commerce, social networks, and multimedia.

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

In this thesis, we focus on automated anomaly detection and interference prediction for Big Data platforms and containerized batch technologies within cloud environments. The research on automated anomaly detection and interference prediction 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 [5, 6]. This leads to a demand for performance anomaly detection and interference prediction in cloud computing and Big Data systems that are both dynamic and proactive in nature [7]. The need to adapt these methods to a production environment with very different characteristics means that black-box machine learning techniques are ideally positioned as solutions that can automatically identify performance anomalies and systems interference. These techniques offer the ability to quickly learn baseline performance through a large amount of monitoring performance metrics, hence identifying normal and anomalous patterns [8]. Based on the nature of input data and the expected output, machine learning algorithms are classified into two main categories: supervised learning and unsupervised learning [7]; some machine learning techniques include classification based, regression based, neighbor based, and clustering based.

Classification techniques are a special case of supervised learning, in which the aim is to determine whether the instances in a given feature space belong to a specific class or to multiple classes [7]. There are popular classification techniques for anomaly identification, such as neural networks, support vector machines, and decision trees [9]. The classification technique is significantly affected by the accuracy of the labeled data and algorithms used.

For example, the training and testing phases for decision trees are usually faster than for support vector machines, which involve quadratic optimization.

We also focus on Apache Spark and containerized batch job technologies. There are various popular open source distributed data-processing frameworks related to Big Data technologies, such as Hadoop MapReduce, Apache Storm, and Apache Spark. Among these, in-memory processing technology like Apache Spark has become widely adopted by industries because of its speed, generality, ease of use, and compatibility with other Big Data systems [10]. Although Spark is developing gradually, there are currently still shortages in comprehensive performance analysis methods specifically developed for Spark and that can used to precisely detect performance anomalies [11]. The performance of in-memory processing frameworks can vary considerably depending on many factors, such as the type of input data, parallelism, application design, system configuration, and available computing resources [11, 12]. The heterogeneity of these factors make anomaly detection and prediction challenging, especially for critical applications. Therefore, there is a need to deeply investigate an in-memory processing technology like Spark and its performance bottlenecks to pinpoint the cause of a performance anomaly.

## 1.2 Contributions

This thesis provides solutions to assist engineers and system administrators in choosing the appropriate anomaly detection mechanisms for their in-memory Spark Streaming Big Data system. The proposed solution also can be used to help system operators to perform interference-free scheduling of jobs within the system. Moreover, it can be modified to help users in taking decisions for task placement within systems to reduce substantial computing network resource consumption.

The following subsection provides an overview of our three main research contributions.

### 1.2.1 AI Driven Anomaly Detection Methodology for In-Memory Big Data Systems

As a first contribution, we develop 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 the precision and recall of the classifiers. We also compare methods that are agnostic of the workflow of Spark jobs by using a novel method that leverages the specific characteristics of Spark fundamental data structure -the resilient distributed dataset (RDD)- to improve anomaly detection accuracy.

Our proposed method is evaluated against three popular machine learning algorithms, decision trees, nearest neighbor, and support vector machine. Our model is evaluated against four variants of the performance metrics that consider different monitoring datasets. The conducted experiments and results demonstrate that our proposed method works effectively and efficiently with complex scenarios and anomalies, such as CPU contention, memory contention, cache thrashing, and context switching anomalies. In addition, the proposed methodology is examined with different types of overlapped anomalies. Compared with other popular methods, the random instant and random duration of anomalies was not found to have an impact on the performance of our proposed methodology. Our results prove that our proposed method outperforms other methods, typically achieving 98–99% F-scores, while offering much greater accuracy than alternative techniques in detecting both the period in which anomalies occurred and their type.

### 1.2.2 Hybrid AI Anomaly Detection Model for Big Data Streaming Systems

The second contribution of this thesis is motivated by the difficulty of carrying out anomaly detection within Big Data streaming systems, especially for time-varying workloads and critical applications. Apache Spark has more than 200 configurable parameters, and some parameters may depend on each other and affect the overall performance of the platform. This large and complex configurable parameter space makes it difficult even for ex-

pert administrators to detect and classify anomalous performance within Spark Streaming clusters, because a certain performance level may simply depend on the chosen configuration. In response to this challenge, we develop agile hybrid learning techniques -*TRACK* and *TRACK-Plus*- for anomaly detection, which are black-box training methodologies for performance anomaly detection within Apache Spark Streaming workloads.

*TRACK* and *TRACK-Plus* efficiently train a class of machine learning models for performance anomaly detection using a fixed number of experiments. *TRACK* revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve a high F-score in a short period of time. *TRACK-Plus* is an automated fine-grained anomaly detection solution that adds to *TRACK* a second Bayesian Optimization cycle for fine-tuning the hyperparameters of artificial neural networks configuration. The objective is to accelerate the search process for optimizing neural network configurations and improve the performance of anomaly classification.

Our results indicate that our *TRACK* and *TRACK-Plus* solutions achieve a high accuracy (95% F-score) in significantly less time (80% less than normal). A validation based on a real dataset for the Apache Spark Streaming system has been provided to demonstrate that the proposed methodology identifies the performance anomalies, the ideal configuration parameters, and the training dataset size with up to 75% fewer experiments needing to be run. Finally, our proposed solutions not only identify anomalous performance with a high F-score but also classify anomalies, thereby saving considerable time in training the model. In addition, the proposed model can be easily generalized to cover unforeseen workload configurations. To the best of our knowledge, this contribution is among the very first works to provide a comprehensive methodology for both performance anomaly classification and the efficient optimization of artificial neural networks to detect anomalies within Apache Spark streaming system.

### 1.2.3 Interference Prediction for Containerized Batch Jobs

The third contribution of this thesis mainly focuses on examining the behavior of colocated batch jobs to proactively predict interference behaviors which can be used to enhance job placement within a Big Data production environment. If such behavior is not predicted and timely handled, it may cause savior consequences and reliability issues for critical systems within cloud environments. Many existing approaches for detecting system interference revolve around monitoring and collecting all the possible performance metrics of each running batch job and testing each job behavior under different resource configurations to predict the interference within system. These approaches are time-consuming for dynamic complex workloads as they try to collect all the performance metrics and test all the combinations of jobs behaviors, of which all jobs combinations needed to be profiled in advance under various system configurations.

To overcome this issue in batch job workloads and proactively avoid performance interference, this thesis introduces an automated prediction solution to estimate the interference between colocated containerized batch jobs within the same computing environment. An AI-driven model is implemented to predict the interference among workloads. This interference prediction model alleviates and estimates the tasks slowdown affected by the interference among running jobs. Our model assists the system operators in making an accurate decision to optimize batch job placement.

Our interference prediction solution is agnostic to the business logic internal to each job. Instead, it is learned from system performance data by applying artificial neural networks. The target is to establish the completion time prediction of the containerized batch job within cloud environments. It uses the profiling data for individual job $i$ and two $i * j$ jobs before attempting to predict the completion times when more than two jobs run simultaneously in the system. The prediction method works without any need for measurements from executions with more than two jobs; everything is predicted using only the available profiling data and the AI model.

Our method learns from data by applying artificial neural networks and comparing them

with the other three baseline models (queueing-theoretic model, operational analysis and no model) on historical measurements of job completion time and CPU run-queue size (i.e., the number of active threads in the system). The model is capable of capturing multithreading, operating system scheduling, sleeping time, and job priorities. A validation based on 4500 experiments using the DaCapo benchmarking suite [13] has been carried out, confirming the predictive efficiency and capabilities of the proposed model. The experimental results prove that our solution is powerful in predicting the potential interference among containerized batch jobs and can achieve up to 10% MAPE compared with other models. Our model is promising for microservices systems that can be extended to cover more advanced and complex production environments.

## 1.3 Thesis Outline

In summary, the purpose of this thesis is to analyze and optimize the performance of distributed Big Data and containerized batch job technologies, here with a focus on the performance anomalies detection and interference prediction. Table 1.1 shows the structure of the thesis. The outline of our main contributions for this thesis are as follow:

1. **Chapter 1:** Introduction of thesis to present the problem overview, objectives, and contributions
2. **Chapter 2:** Provides background information about Big Data platforms and gives a literature review, methods, and techniques.
3. **Chapter 3:** Implements a neural network-driven methodology for anomaly detection within in-memory systems. This chapter mainly focuses on Apache Spark batch workloads and anomaly detection to provides an introduction, motivating examples, methodology, and results.
4. **Chapter 4:** Presents the hybrid ML-based model for anomaly detection within in-memory Big Data streaming systems. This chapter provides an introduction, literature review, motivation of our contribution, our methodology, and the final results.
5. **Chapter 5:** Implements a batch completion time prediction and interference pre-

diction solution. This chapter gives an introduction, literature review, motivation, methodology, and the final results.

6. **Chapter 6:** This final chapter summarizes the thesis, provides a conclusion, and gives future research directions.

Table 1.1: Structure of the thesis

<table>
<tr><td colspan="3">Chapter 1<br>General introduction, research gaps, and thesis structure</td></tr>
<tr><td colspan="3">Chapter 2<br>Background information and literature review</td></tr>
<tr><td>Chapter 3<br><br>Short introduction<br>Motivating Example<br>Methodology<br>Evaluation<br>Results<br>Conclusion</td><td>Chapter 4<br><br>Short introduction<br>Motivating Example<br>Methodology<br>Evaluation<br>Results<br>Conclusion</td><td>Chapter 5<br><br>Short introduction<br>Motivating Example<br>Methodology<br>Evaluation<br>Results<br>Conclusion</td></tr>
<tr><td colspan="3">Chapter 6<br>General discussion, conclusion and future work</td></tr>
</table>

## 1.4 Publications

During my PhD program, I have published peer-reviewed publications, for all of which I am the first author, except [14, 15], for which I am a joint first author. These publications include the following:

### 1.4.1 Book Chapter

**[15]** A. Alnafessah, G. Russo Russo, V. Cardellini, G. Casale, F. Lo Presti. **AI-Driven Performance Management in Data-Intensive Applications**. Book chapter within *Communication Networks and Service Management in the Era of Artificial Intelligence and*

*Machine Learning book* 2021. The goal of this book chapter is to overview some recurring performance management activities for data-intensive applications, examining the role that artificial intelligence (AI) and machine learning are playing in enhancing practices related, among others, to configuration optimization, performance anomaly detection, load forecasting, and auto-scaling for these software systems.

### 1.4.2 Journals

**[16]** A. Alnafessah, G. Casale. **Artificial neural networks based techniques for anomaly detection in Apache Spark**. This a full paper has been accepted in Cluster Computing Journal 2019. This paper presents 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 metrics 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.

**[17]** A. Alnafessah, G. Casale. **TRACK-Plus: Optimizing Artificial Neural Networks for Hybrid Anomaly Detection in Data Streaming Systems**. Accepted as a full paper in IEEE ACCESS Journal 2020. This paper introduces TRACK-Plus, a black-box training methodology for performance anomaly detection. The method uses an artificial neural networks-driven methodology and Bayesian Optimization to identify anomalous performance and are validated on Apache Spark Streaming. TRACK-Plus has been extensively validated using a real Apache Spark Streaming system and achieve a high F-score while simultaneously reducing training time by 80% compared to efficiently detect anomalies.

**[14]** A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, A. Filieri. **Quality-Aware DevOps Research: Where Do We Stand?**. Accepted as full paper in IEEE ACCESS

Journal 2021. This paper addresses the gap by comprehensively surveying existing efforts in the area of quality-aware DevOps and categorizing them according to the stage of the DevOps lifecycle to which they primarily contribute. The survey holistically spans across all the DevOps stages, identify research efforts to improve architectural design, modeling and infrastructure-as-code, continuous-integration/continuous-delivery (CI/CD), testing and verification, and runtime management. The conducted analysis also outlines possible directions for future work in quality-aware DevOps, looking in particular at AI for DevOps and DevOps for AI software.

### 1.4.3 International conference papers

**[18]** A. Alnafessah, G. Casale. **A Neural-Network Driven Methodology for Anomaly Detection in Apache Spark**. Published in proceeding of 11th International Conference on the Quality of Information and Communications Technology (QUATIC), 2018. In this paper we consider in particular Spark-based workloads, in which the analytic operations are applied to a resilient distributed dataset (RDD). We develop a neural network based methodology for anomaly detection based on knowledge of the RDD characteristics. Using experiments against multiple workloads and anomaly types, we show that our method improves over other types of classifiers as well as against black box performance anomaly detection.

**[19]** A. Alnafessah, G. Casale. **TRACK: Optimizing Artificial Neural Networks for Anomaly Detection in Spark Streaming Systems**. Accepted as full paper in the proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'20). In this paper we introduce, TRACK, a new black-box training workload configuration optimization with a neural network driven methodology to identify anomalous performance in an in-memory Big Data Spark streaming platform. The proposed methodology revolves around using Bayesian Optimization to find the optimal training dataset size and configuration parameters to train the model efficiently. TRACK is validated on a real Apache Spark streaming system and the results show that the TRACK achieves the highest performance (95% for F-score) and reduces the training time by 80% to efficiently train the proposed anomaly detection model in the

in-memory streaming platform.

### 1.4.4 Posters

**[20]** A. Alnafessah, G. Casale. **Anomaly Detection for Big Data Technologies**. Published in Imperial College Computing Student Workshop (ICCSW 2018). The main goal of this research is to contribute to automated performance anomaly detection for large-scale and complex distributed systems, especially for Big Data applications within cloud computing. Investigating automated detection of anomalous performance behaviors by finding the relevant performance metrics with which to characterize behavior of systems. Another contribution focusing on pinpointing the cause of a performance anomaly due to internal or external faults.

**[21]** A. Alnafessah, G. Casale. **AI Driven Methodology for Anomaly Detection in Apache Spark Streaming Systems**. Published in the 3rd International Conference on Computer Applications and Information Security (ICCAIS 2020). This research introduces a new black-box training workload configuration optimization with a neural network driven methodology to identify anomalous performance in an in-memory Spark streaming Big Data platform. The proposed methodology effectively uses Bayesian Optimization to find the ideal training dataset size and Spark streaming workload configuration parameters to train the anomaly detection model. The proposed model is validated on the Apache Spark streaming system. The results demonstrate that the proposed solution succeeds and accurately detects many types of performance anomalies. In addition, the training time for the machine learning model is reduced by more than 50%, which offers a fast anomaly detection deployment for system developers to utilize more efficient monitoring solutions.

# Chapter 2

# Background

## 2.1 Introduction

In recent years, the prominence of Big Data has led to a growing interest in developing intelligent data-intensive software systems in several application domains. Data-driven systems that can extract knowledge, plan, and adapt to events through the processing, transformation, and analysis of datasets are increasingly widespread in both industry and society. From a technical standpoint, data-driven software systems are often built by leveraging features such as batch or streaming analytics, which are now easily programmable through platforms such as Apache Spark, Hadoop/MapReduce, Storm, Flink, among others. We outline popular Big Data processing platforms in Section 2.2 and Section 2.3.

There are various options when it comes to open source Big Data technologies for enterprises to work on data intensive applications and analyses. Although each of these technologies has advantages for specific purposes (e.g., batch processing), they may not be an ideal choice for other types of applications (e.g., streaming applications). Based on data processing methods, these technologies can be categorized into three main approaches: batch, stream, and micro-batch processing [22]. This chapter gives an overview of Apache Spark and other popular open source Big Data technologies while also discussing

the essential features of common platforms used to define Big Data applications.

Although the combination of batch and streaming workloads enables richer functionality, workload heterogeneity also means that achieving service-level objectives presents additional complexity in pinpointing the causes of performance degradation and identifying ways to address them. For example, performance metrics in data-driven software are difficult to predict because they often depend on data properties, such as volume or velocity, and frequently even on data type and content, making it difficult to reason about and tune system performance at design time. Furthermore, the combination of batch and streaming features in software means that different system components will strive to achieve different performance goals, that is high-throughput and high-utilization for analytics features and low latency for stream processing operators, making the process of runtime performance tuning a fairly heterogeneous and complex exercise.

To support these challenges, the goal of this chapter is to overview Big Data platforms and AI management techniques for monitoring, managing, and tuning the performance of data-intensive applications. AI methods offer considerable simplicity and flexibility in choosing the features that drive the management process, despite some opaqueness in presenting the way the models reach decisions.

Compared with traditional management methods, which either leverage low-level system characteristics or has a sensitivity about the distribution of data, AI management methods leverage learning on experimental datasets, hence reducing the dependence on assumptions and shifting the attention from conceptual modeling to data collection and model training. This offers considerable potential to increase the effectiveness of management methods in situations where the system behaves according to complex and unpredictable logic, as is often the case for systems driven by external data.

In summary, in this chapter, we examine the applicability of AI and other methods in the context of anomaly detection and interference prediction within data intensive applications. We survey in particular studies that illustrate the versatility of AI models when applied to popular data streaming and batch analytics platforms. Our aim in particular is to cover a broad spectrum of AI methods, to show the range of learning techniques that

may be applicable to the recurring management problems involved in data-driven systems. We look at common management tasks such as platform configuration, workload forecasting, monitoring, detection of performance anomalies, and interference within systems. We also give selected examples of big data platforms and AI techniques to give an intuition on their behavior, benefits, and limitations.

The structure of this chapter is as follows: Section 2.2, and Section 2.3 give an overview of the essential features of common execution platforms in use to define data-intensive applications and highlight the core performance management challenges associated with each of these platforms. Section 2.4 provides a literature review of exciting research in this area. Section 2.5 presents the methods and techniques that are used in this thesis.

## 2.2 Apache Spark

Spark is a large-scale in-memory processing technology that can support both batch and stream data processing, which can make it easy to use because of its low cost in supporting different types of workloads on the same engine in a production environment [10]. This type of system is an example of a micro-batch system. The main goal of Apache Spark is to speed up the batch processing of data by utilizing in-memory computation. According to Apache Spark, Spark is 100 times faster than Hadoop MapReduce for in-memory analytics [10]. Also, Spark is considered more efficient than MapReduce for complex applications.

As an alternative to Hadoop, Spark can be deployed over Hadoop Distributed File System (HDFS). It can also be deployed on Amazon EC2, Apache Mesos, or as a standalone cluster. In addition, it can access numerous data sources, including HDFS, Cassandra, HBase, Hive, and any Hadoop data source [10]. Apache Spark provides a general purposes engine for different kinds of computations, including iterative algorithms, job batches, streaming, and interactive queries. These different types of computation were previously difficult to find in the same distributed system [23]. Beyond its ability to perform batch and stream processing, Apache Spark also has a rich library that is built on top of Spark core engine [24].

### 2.2.1 Apache Spark Core and Upper Stack

#### Apache Spark Core

Apache Spark core engine has a general purpose and the ability to run over different types of cluster managers while accessing data from different resources [25]. Spark core is implemented in Scala, and it supports many APIs in Scala, Java, Python, and R. The core engine offers basic functionalities for in-memory cluster computing, such as task scheduling, memory management, fault recovery, and communicating with database systems [23]. In addition, Spark engine provides the API for the main programming data abstraction -the Resilient Distributed Dataset (RDD)- which allows for the scalability of data algorithms with high performance. The RDD has some operations, including data *transformation* and *actions*. Other Spark libraries and tools need these RDD operations for data analysis algorithms.

#### Spark Upper Stack

As a result of the speed and general purpose of Spark core engine, the engine offers a suitable environment on top of Apache Spark core to support different types of workloads and computations. These workloads include Spark SQL, Spark Streaming, Spark MLlib, and Spark GraphX applications. This upper stack offers benefits to combine all components in libraries in user applications [10]. Upper components can obtain some valuable benefits from this kind of tight integration with Spark core engine. First, the upper components can easily gain benefits from the continuous improvement and optimization of the lower stack components (Spark core). Second, cost and time saving are dome of the significant benefits for upper component users. These benefits can be achieved by offering different types of services and run them within the same system instead of having multiple systems and running each service on an independent system. Therefore, operation and maintenance are reduced for every system that needs configuration for deployment, testing, and support.

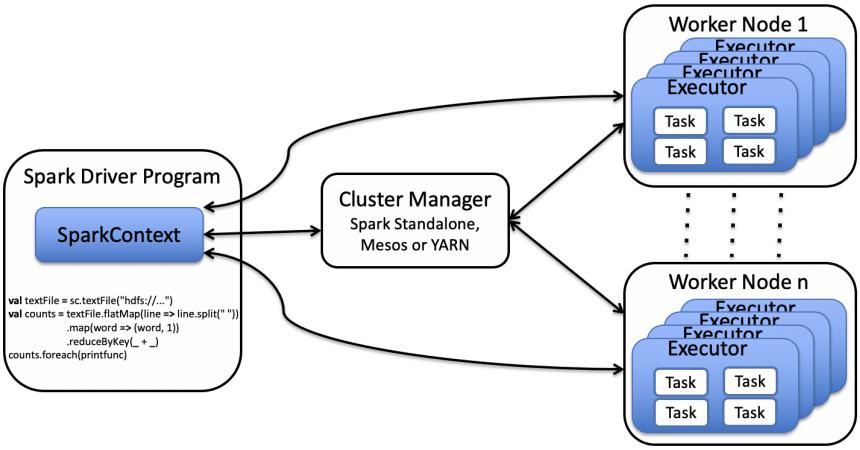



Figure 2.1: Spark Application Components

### 2.2.2 Apache Spark Application Architecture

Running the Spark application involves five main components: driver programs, cluster managers, worker nodes, executor processes, and tasks as shown in Figure 2.1. The Spark application runs as an independent set of processes on a cluster and is 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 recourses for the application. The cluster manager can be Mesos, YARN, or a Spark stand-alone cluster [10].

After resources are allocated, Spark acquires "*executors*" for an application on the "*worker node*" that provides CPU, memory, and storage resources to run the application codes in the cluster. Each application has its own "executor" processes that run tasks (the smallest unit of work for Spark executor) and keeps application data in the memory or disk. After acquiring "*executors*", the application code, which is defined as a JAR or Python file is sent to the *executors* on worker nodes. During the final step, *SparkContext* sends tasks to the *executors* to run [10].

This type of architecture is valuable for isolating the application from other applications on the schedulers and *executors* side. Therefore, each driver program can schedule its own

tasks, and each executor can only execute its tasks from the different applications run in different Java Virtual Machines. As a consequence, data and message exchange cannot be shared across Spark applications (instance of *SparkContext*) without using external storage, which may negatively impact the latency of the operations being run [22].

### Data Abstraction and Operation

The Spark engine provides the API for the main programming data abstraction -RDD- which enables the scalability of data algorithms with high performance. An RDD offers operations, including data *transformation* and *actions*, that can be used by other Spark libraries and tools for data analysis. This thesis (Chapter 3 and Chapter 4) proposes an anomaly detection method that performs an effective instantiation anomaly detection at the level of the RDDs. Thus, we provide a brief overview of the main features of these data structures and their relationship to the job execution flow within Spark.

Spark RDD is the core data abstraction of Apache Spark. 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 also 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. An 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 a string (using *sc.textFile()*) [10].

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 a *transformation* operation is filtering data that returns a new RDD that meets the filter conditions [26]. 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

```
val textFile1 = sc.textFile("hdfs://...")
val wordCounts = textFile1.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
wordCounts.saveAsTextFile("hdfs://...")
```

Figure 2.2: Simple WordCount Example

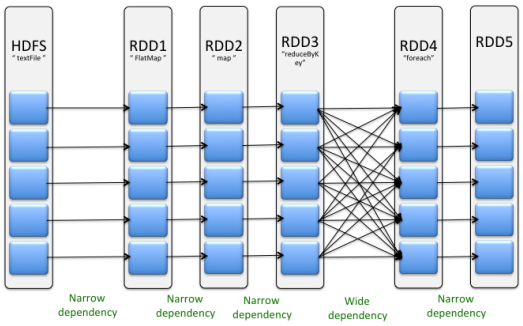
external storage, such as the Hadoop Distributed File System (HDFS). A basic example of an *action* operation is *First()*, which returns the first element in an RDD. Other *action* operations are *collect, count, first, takesample,* and *foreach* [10].

Some researchers describe *transformations* as lazy operations because their result is not instantly computed [10]. They are only computed when an action (e.g., *collect* or *count*) requires a result to be returned to the driver program [10]. This *lazy* operation enables Spark to run operations more efficiently. For example, this appears in cases where datasets are created by (*map()*) and will be used in a (*reduce()*) and where they return just the result of the (*reduce*) to the driver program instead of the larger mapped dataset [10]. This lazy evaluation optimizes memory usage in Spark.

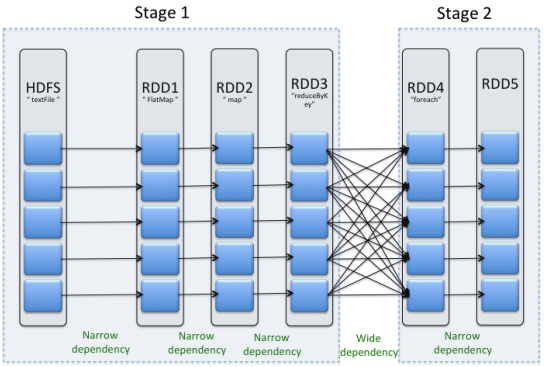
RDDs are reliable and use a fault-tolerant distributed memory abstraction. Spark has the ability to reliably log the *transformation* operation used to build its lineage graph rather than the actual data [27]. The lineage graph keeps track of all *transformations* that need to be applied to RDDs and the 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 which helps in facilitating description of lineage graphs [27].

### 2.2.3 Run-Time Spark Architecture

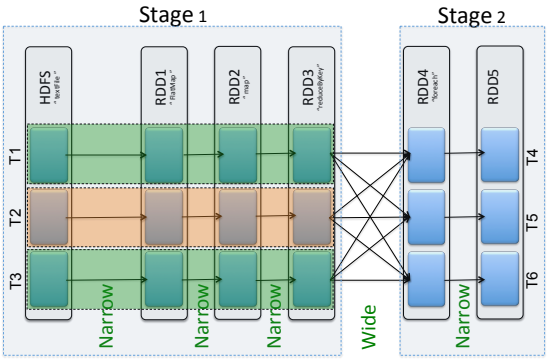
Apache Spark uses the representation of RDDs and takes into account the partitions of cached RDDs that are available in memory. Every Spark application (e.g., Figure 2.2) consists of jobs (Figure 2.3(a)), and each job is further divided into stages (Figure 2.3(b))




(a) Type of dependencies: Narrow and Wide




(b) A job is divided into two stages, which second stage has wide dependency on the first stage




(c) Stages are divided into tasks that are equal to the number of partitions in the same stage.

Figure 2.3: Simple DAG for WordCount example.

that depend on each other. Each stage is then composed of a collection of tasks, as shown in Figure 2.3(c) [28].

*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 an 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 [27].

*Spark Stage:* Breaking the RDD DAG at the shuffle boundaries will create stages. Each stage contains many pipelined RDD *transformation* operations that do not require any shuffling between operations, which is called a *narrow dependency* (e.g., *map, filter, etc.*). Otherwise, if the stages depend on each other through RDD *transformation* operations that require shuffling, then these are called *wide dependencies* (e.g., *group-by, join, etc.*) [27]. Therefore, every stage will contain only shuffle dependencies on other stages, not inside the same stage. The last stage inside the job generates results, and the stage is executed only when its parent stages have been executed. Figure 2.3(b) shows how the job is divided into two stages as a result of shuffle boundaries.

*Spark Task:* 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 Figure 2.3(c).

*Scheduling:* The task in Apache Spark is the smallest unit of work sent to the executor, and there is one task per RDD partition. The dependencies among the stages are unknown to the task scheduler. Each TaskSet contains fully independent tasks, which can run based on the location of the data and the currently cached RDD. Each task is sent to one machine [28]. Inside a single stage, the number of tasks is determined by the number of final RDD partitions in the same stage.

### 2.2.4 Monitoring Performance

Running data-intensive applications in distributing computing requires continued monitoring, especially in production environments. Any late detection of anomalous behavior

in the systems may negatively impact those systems, and they may cause prolonged performance violations with huge financial penalties [5, 6]. Therefore, there is a need to analyze application behavior to improve performance. Apache Spark offers many monitoring options, including web UI, metrics, and external instrumentation [10, 29].

Many external tools can be used for profiling to monitor the performance of Apache Spark applications, such as *Ganglia*, *dstat*, and *JVM* utilities [10]. *Ganglia* is a scalable distributed system for monitoring applications that uses a dashboard to show an overview of the overall cluster utilization and resource bottlenecks. Another tool that is used with Spark for OS profiling is *dstat*, which can give fine-grained profiling on a specific node.

## 2.3 Other Big Data Technologies

### 2.3.1 Hadoop MapReduce

Hadoop[1] implements the MapReduce paradigm and it is a well-known example of batch processing platform. It is used for intensive Big Data applications starting from a single server and can scale up to thousands of machines. Usually, MapReduce uses an existing dataset that is stored in Hadoop Distributed File System (HDFS) before beginning to process batch data. Processing data with native Hadoop can be paused or interrupted, but the dataset cannot be modified. This means that if current data is changed for any reason, the job needs to be run again.

Despite distinctive challenges arise in the area of optimal configuration of Hadoop platforms, performance management has some challenges in detecting and handling straggler tasks, which falls into the general problem area of performance anomaly detection and mitigation. This is a result of the synchronizations between dataflow tasks that can block progress until straggler tasks complete their activities.

[1] https://hadoop.apache.org/

### 2.3.2 Apache Flink

Apache Flink[2] is another open source distributed processing engine designed for low-latency streaming computation. Flink relies on in-memory computation and provides a unified API for processing both bounded and unbounded datasets [30]. However, differently from Spark, where batching has a primary role, Flink has been designed with streaming in mind. Indeed, Flink applications are built upon the concepts of *streams* and *transformations*. *Streams* represent (possibly unbounded) data flows, while *transformations* are operations that given one or more streams as input, one output or more streams as the result (e.g., filtering). A few higher-level libraries are built on top of these abstraction, easing the definition of common processing use cases (e.g., complex event processing and graph analytics).

At runtime, Flink applications are mapped to *streaming dataflows*, DAGs composed of processing nodes (often called *operators*), which implement *transformations*, connected by streams. For execution, Flink leverages a distributed architecture, designed according to the *master-worker* pattern. The master component is the *JobManager*, which coordinates distributed execution and is responsible for application scheduling, checkpointing, and recovery in case of failure. The *TaskManagers* (i.e., the workers) execute the application *tasks* (i.e., instances of operators) and manage the data transfers between them.

Performance management of Flink applications, which are often long-running, mainly involves runtime deployment and resource adaptation. First of all, varying infrastructure conditions may require migrating operator tasks between computing nodes during execution. Flink supports migrating both stateless and stateful tasks through the *savepoint* mechanism, which ensures no loss of information. Moreover, workload variability requires dynamically scaling the parallelism of Flink applications and balancing the load across the cluster to keep consistent performance levels over time.

[2]https://flink.apache.org

Table 2.1: Summary of the key characteristics of data-processing platform.

| | **Workloads** | | **Processing style** | |
|---|---|---|---|---|
| **Platform** | **Batch** | **Streaming** | **In-Memory** | **Disk-heavy** |
| Storm | | ✓ | ✓ | |
| Hadoop/MR | ✓ | | | ✓ |
| Spark | ✓ | ✓ | ✓ | |
| Flink | ✓ | ✓ | ✓ | |

### 2.3.3 Apache Storm

Apache Storm[3] is a popular open source platform used for distributed real-time stream processing. The platform offers very low latency for dataflow processing, making it an ideal option for real time processing [31]. Storm dataflow topologies involve two main node types: *spouts* and *bolts*. A spout is the source of the data stream at the input queue and may generate data by itself [32]. A bolt instead consumes the stream, operates transformations or computations, and ultimately produces an output stream as a result. Every task corresponds to one operating system thread. Storm topologies execute one or many worker processes, where each worker process maps to a separate Java virtual machine and can execute subsets of the tasks for topology.

Performance management of Storm applications frequently involves difficult decisions concerning optimal configuration options for *spouts* and *bolts*, ranging from decisions concerning buffer, message, and batch sizes, number of *bolts*, and selection of optimal waiting strategies. There is a limited understanding of the interplay between these parameters, posing intrinsic challenges for optimal system configuration. Moreover, the Storm system does not automatically manage load balancing and resource scaling, thus requiring ad-hoc performance management techniques. A summary of the key characteristics of each platform is shown in Table 2.1.

[3]https://storm.apache.org

## 2.4 Literature Review

System performance is often described in terms of the time taken to process some tasks or the set rate of tasks performed with a given amount of computing resources that are consumed within a given observation period [7]. With the growing complexity and dynamicity of Big Data and cloud systems, failure management requires significantly higher levels of automation and attention [33]. Performance anomaly detection techniques are greatly needed by Big Data and large scale systems. Performance interference and anomalies have become a major concern for academic researchers of Big Data technologies over cloud computing services. The ability to analyze data is vital to the process of detecting anomalous and interference behavior to resilience in production systems in spite of noise or risks arising in the production testbed (e.g., a cloud computing environment). Anomalous performance can occur as a result of service operator faults [34], software failures and user errors [35], environmental issues, and security violations [7], among others. Several studies illustrate that most of the root causes of bottlenecks and anomalous performance are machine resources such as computer processing units (CPUs) [36, 7, 37].

Many anomaly detection and interference prediction studies have been specifically conducted for certain application domains, while others are more generic. Hodge et al.[38] review techniques that have been developed in statistical analysis and machine learning for anomaly detection. The study conducted in [39] provided a structured and comprehensive overview of the research on anomaly detection that grouped existing techniques into different categories based on the underlying approach that they each adopted. We point to [38] and [39] for general discussions on machine learning, statistical analysis, and anomaly detection. Table 2.2 further shows a summary of detection techniques used in the context of cloud computing systems. The following subsections provide an overview about the exciting statistical and ML techniques for anomaly detection and interference prediction within systems.

### 2.4.1 Statistical Techniques

Some studies use statistical methods to detect anomalous behavior, such as Gaussian-based detection [40, 41], regression analysis [42, 43], and correlation analysis [44, 45, 46]. 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 multiple dimensional real-time datasets [39]. To avoid this assumption, we focus more on utilizing other approach, which is machine learning.

Gow et al. [47] claim that their method is an accurate way to characterize system performance signatures and that it is not customized to the system being analyzed. 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. [47] 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 [47] 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.

### 2.4.2 Machine Learning Techniques

Olumuyiwa et al. [7] argue that data mining and machine learning technologies have received growing attention for performance anomaly detection and diagnosis by the research community. Based on the nature of the input and the expected output, supervised learning or unsupervised learning may be used. A basic anomaly detection system observes the performance behaviors of the targeted system to collect measurements to generate essential profiles about normal system performance [7]. This observation will continue to detect any undesirable deviation or anomaly in performance and apply a root cause analysis to pinpoint causes of a performance anomaly due to internal or external faults. In general,

performance violation and anomalies are a single point or a group of data falling far away from the expected normal region. Anomalies may be classified into four types, including point, collective, contextual, and pattern anomalies [7]. Some of the main root causes of anomalies are system misconfiguration, workload burstiness, and buggy application code within Big Data and cloud computing environments.

Machine learning classification techniques are used to classify an input features into predefined classes of items in order to construct a classifier that can predict the class of each item in the dataset according to the class labels of this dataset. There are well-known classification techniques for anomaly identification, such as neural networks, decision tree, nearest neighbor, support vector machines, and Bayesian networks [9]. The classification technique is significantly affected by the accuracy of the labeled data and algorithms that have been used. For example, training and testing phases of decision trees algorithms are usually faster than in support vector machines, which involve quadratic optimization.

This thesis benefits from comparing the methods that have been shown in other works such as [7] to choose a suitable hybrid machine-learning solutions to advance the traditional approaches. The statistical detection techniques may require an assumption about data distribution to know prior, whereas a machine-learning technique does not need this assumption. The following subsection shows types of machine learning in the literature, which include supervised, unsupervised, semi-supervised learning techniques.

#### Supervised Learning Techniques

In a supervised learning technique, the training dataset is assumed to be available and contains well-labeled instances to distinguish between anomalies and normal classes. An example of a supervised technique is used in [48]. According to Gu and Wang [48], a stream-based anomaly behavior-detection technique for online application has been used to detect anomaly indications that relate to performance anomaly root localization. They apply Bayesian classification methods to detect an anomaly and its root causes within the data center. In addition, the authors [48] apply Markov models to detect the change in the patterns of different measurement metrics for system performance. Comparing

Markov modeling with Bayesian classification methods allows the prediction of anomalous behaviors that will likely occur in the future within the system. The authors implemented their experiment within the IBM system S-distributed stream processing computing cluster and used a real-system workload. Based on their experiments [48], they claim that their approach has high accuracy with low overhead while assessing anomaly prediction performance in a clustering system.

Fulp et al. [49] use a machine learning approach to detect and predict the likelihood of system failures using an SVM based on performance metrics from Linux system log files. They examine a dataset that contains actual file logs of a cluster with 1024 computing nodes. Their proposed solution achieves an acceptable level of classification performance which is 73%. Fulp et al. [49] however, consider only one type of system failure -hard disk failure- without examining the other common sources of systems failure, such as CPU, cache, etc. Although SVM models are effective at making a decisions from well-behaved feature vectors, they can be more expensive for modeling variations in large datasets and high-dimensional input features [50, 38, 39].

Stragglers are bottlenecks task-level, which are usually not a straightforward to be discovered and have high impacts on big data platforms. The causes of stragglers can be occurred as a result of data skew, resource contention and hardware failure [36, 51, 52]. Qi et al. [53] propose a white-box model that use classification and regression trees to analyze straggler root causes. The authors use raw metrics from Apache Spark logs and hardware sampling tools to train their model. To avoid overfitting issue, the authors use CART tree (Classification and Regression Tree), which offers some mitigation solutions. The solution includes a pruning technique (named CCP) when the tree growth completes. The pruning process continues for several iterations and the classification performance metrics are checked for each node and its leaves [53]. Such a process is time consuming, especially with intensive data streaming systems, so this study does not consider it.

While anomaly detection studies have been conducted in the literature for many purposes, there are not enough research studies that address anomaly detection and prediction issues, especially for Big Data technologies, such as Apache Spark. From the study in [48],

one of the main issues in the data streaming processing cluster is the bottleneck, which is worth investigating more in our research. This bottleneck can occur among distributed applications when the input queue of applications is saturated. Some reasons for a bottleneck include the shortage of computing resources given to an application, the violation of data rates of processing capacity, or the application may has misconfiguration.

**Unsupervised Learning Techniques**

The local outlier factor (LOF) algorithm is a type of neighbor-based technique of an unsupervised learning algorithm. The LOF algorithm is employed to detect anomalous behavior in cloud computing in [54]. The main idea is to identify anomalous performance by comparing the density of each instance. Any instance in low density is considered an anomaly. To improve performance anomaly detection, the LOF requires a significant effort in collecting a complete training dataset of normal behavior for applications before the starting the detection phase. This dataset is sometimes unavailable over online applications. Therefore, the authors [54] use an adaptive anomaly detection scheme for a cloud system based on the LOF. They argue that their scheme can learn application behavior in both training time and detecting time. In addition, the scheme is adaptive to the new changes during the detection phase, which offers a significant reduction of effort to collect the training dataset before the detection phase. The experimental results in [54] show that their scheme can effectively detect a performance anomaly with a low level of computational overhead.

Lin et al. [55] propose an anomaly detection technique for infrastructure as a service (IaaS) cloud computing environment using local outlier factor (LOF) algorithm to detect anomalies by analyzing the reduced performance feature dataset. LOF is used to assign a score for each group of performance metrics to assess the system behavior, where the behavior is considered anomalous if the score exceeds the predefined threshold. The authors validate their technique within a private cloud computing system that is built using OpenStack and Xen open-source software. Their result shows that the proposed technique outperforms principal components analysis (PCA).

Using the basic LOF requires considerable effort to collect enough datasets of normal behavior and requires intensive computations to calculate the distance scores of each instance during the test phase. According to [39], it is difficult to compute distance measurements for streaming data, and it cannot identify contextual performance anomalies. Therefore, it is important to keep in mind that the LOF needs to be adaptive to be used with Spark for any contextual anomaly detection.

**Semi-supervised Learning Techniques**

A semi-supervised learning technique is halfway between supervised and unsupervised learning techniques in which some training data are not labeled. It is often assumed that labeled data constitute a normal class while the remaining unlabeled data instances are anomalous. The author of [33] investigates autonomic anomaly detection across cloud computing systems. Concerning feature selection, the author applied metric selection and extraction methods to select the most relevant ones. The author applied principle component analysis (PCA) to reduce metric dimensions and maintain the variance in health-related data as much as possible. Moreover, semi-supervised model using decision tree classifiers is used in [33] to reduce metric dimensionality and to identify anomalies in the cloud systems. His experimental result shows that the method has the ability to successfully reduce metric dimensions and identify performance-anomalous behaviors.

Table 2.2, Table 2.3 and Table 2.4 further show a summary of detection techniques used in the context of cloud computing and distributed computing systems. Further advancement in hybrid solutions holds great potential for anomaly identification systems [56, 57]. Some performance anomaly identification studies and surveys have been conducted in the literature for different purposes [38, 7, 58, 59]; however, there is still a shortage of studies that propose efficient automated anomaly detection, especially for in-memory Big Data stream processing technologies as we study in the next sections. In this chapter, we utilize Bayesian optimization hyperparameter tuning and the efficiency of neural networks to accurately detect anomalous behavior in Big Data systems.

Table 2.2: Summary of the state-of-the-art anomaly detection techniques

| **Reference** | **Approach** | **Detection Technique** | **System/Environment** |
|---|---|---|---|
| Gow et al.(2013) [47] | Statistical | Regression curve fitting the service time-adapted cumulative distributed function | Online platform and configuration agnostic |
| Wang et al. (2011) [60] | Statistical | Gaussian-based detection | Online anomaly detection for conventional data centers |
| Markou and Singh (2003) [41] | Statistical | Gaussian-based detection | General |
| Kelly (2005) [42] | Statistical | Regression analysis | Globally-distributed commercial Web-based, Application & System metrics |
| Cherkasova et al. (2009) [43] | Statistical | Regression analysis | Enterprise web applications and conventional data center |
| Agarwala et al. (2007) [44] | Statistical | Correlation | Complex enterprise online applications and distributed System |
| Peiris et al. (2016) [45] | Statistical | Correlation | Orleans system, distributed system and distributed cloud computing services |
| Sharma et al. (2013) [46] | Statistical | Virtualized cloud computing and distributed systems | Hadoop, Olio and RUBiS. |
| Gu and Wang (2009) [48] | Machine learning | Supervised Bayesian classification | Online application for IBM S-distributed stream processing system |
| Huang et al.(2013) [54] | Machine learning | Unsupervised Neighbor-based technique (Local Outlier Factor algorithm) | General cloud computing system |
| Fu (2011) [33] | Machine learning | Semi-supervised Principle component analysis and Semi-supervised Decision-tree | Institute-wide cloud computing system |
| Ren et al(2018) [61] | Machine learning | Anomaly detection approach based on stage-task behaviors and Logistic Regression Model | Online framework for Apache Spark Streaming systems |
| Lu et al (2018) [62] | Machine Learning | Anomaly detection using convolutional neural networks based Model | Big Data system logs using Hadoop Distributed File System |

Table 2.3: Summary of the state-of-the-art anomaly detection techniques

| **Reference** | **Approach** | **Detection Technique** | **System/Environment** |
|---|---|---|---|
| Magalhaes et al. (2010)[63] | Statistical | Correlation techniques and time-series alignment algorithms to spot the relationship between the transactions response time and the workload to pinpoint the occurrence of anomalies for dynamic workloads | Web-based and component-based applications applications |
| Zhang et al. (2007) [64] | Statistical | Regression-based model for estimating CPU demands by different client transactions. The result of the regression method is used to parameterize an analytic model of queues | Enterprise e-commerce system and TPC-W benchmark [65] |
| Kelly (2005) [66] | Statistical | Simple queueing-theoretic observations with standard optimization methods for performance anomaly detection | Distributed commercial systems that serve real customers |
| Yang et al. (2007) [67] | Statistical and Signal Processing | Extending the traditional window-based strategy by using signal-processing techniques to filter out recurring, background variations to determine which resource is the probable cause of an anomalous performance in a system | Three Grid environment applications: Cactus, GridFTP, and a Sweep3d |
| Lu et al. (2017) [68] | Statistical | Off-line approach to detect abnormal Spark tasks and analyze the root causes based on statistical spatial-temporal analysis. The mean and standard deviation of all tasks in each stage are used to get information about macro-awareness on the task's execution time | Private Apache Spark cluster and SparkBench [11] |
| Garraghan et al. (2016)[37] | Statistical | Empirical analysis for straggler detection and root-cause for batch processes using a combination of offline execution patterns modeling and online analytic agents for monitoring | Virtualized Cloud data centers |
| Bodik et al. (2010) [69] | Hybrid | Logistic regression to select a set of relevant metric to minimize both the prediction errors and quantile to summarize the values of each performance metric across all the application servers | Enterprise cloud computing and web applications |
| Jallad et al. (2020)[70] | ML | Deep learning model to detect performance anomaly (LSTM neural networks) to detect unseen anomalies with a low less false-positive rate | NSL-KDD open source dataset that has six basic categories and 41 input features [71] |

Table 2.4: Summary of the state-of-the-art anomaly detection techniques

| **Reference** | **Appr** | **Detection Technique** | **System/Environment** |
|---|---|---|---|
| Chen et al. (2002)[72] | Hybrid | Using data mining and statistical techniques to correlate the normal and failure requests to pinpoint faults | PetStore, which is e-commerce environment based on the Java 2 Platform Enterprise Edition (J2EE) for Web and distributed systems. |
| Cohen et al. (2005)[73] | ML | Pattern recognition, clustering, information retrieval, and Tree-Augmented Naive Bayes models (TAN) are used to characterize each metric and its contribution | Two distributed applications, one runs synthetic workloads in a private system and the other one run real customer services in a globally-distributed production environment |
| Fulp et al. (2008) [49] | ML | Supervised detection and prediction of the hard disk failure using an SVM | Cluster with 1024 computing nodes |
| Pannu et al. (2012) [74] | ML | Supervised one-class SVM and SVM for self-evolving anomaly identification and prediction | Utility cloud computing systems |
| Baek et al. (2017) [75] | ML | Unsupervised labeling by clustering the training data set to create referential labels and supervised anomaly detection model that includes Naive Bayes, Adaboosting, SVM, and Random Forest | Enterprise and cloud computing systems |
| Ren et al (2018) [61] | ML | Anomaly detection approach based on stage-task behaviors that related to the task execution status to classify normal and abnormal workloads according to the offline logistic regression model for each batch | Homogeneous Apache Spark Yarn Cluster for streaming workload with BigdataBench benchmark [76] |
| Lu et al (2018) [62] | ML | Anomaly detection using convolutional neural networks based model, which is implemented with different filters to automatically train model on the relationships among events | Big Data system logs using Hadoop distributed file system |
| Qi et al. (2017) [53] | ML | White-box model for root-cause analysis of performance bottleneck and straggler based on CART decision tree | Private Apache Spark Cluster with HiBench benchmark to generate workloads |
| Yadwadkar et al. (2012) [77] | ML | Regression decision tree model periodically learns correlations between node level status and task execution time | Trace from Facebook Hadoop system and Berkeley EECS department's local Hadoop cluster (icluster) |
| Tan et al. (2011) [78] | ML | Semi-supervised one-class anomaly detection for streaming data using random half space trees | Open source datasets from KDD Cup 99 dataset [79] |
| Pu et al. (2020) [80] | ML | Unsupervised anomaly detection that combines clustering and SVM to identify anomalies within networks | Evaluation is based on public NSL-KDD datasets [71] |

### 2.4.3 Tools and Benchmarks for Big Data systems

Regarding Big Data technologies, Min et al. [11] have developed "SparkBench", which is an open source benchmark, particularly for Apache Spark. The authors cover four main application categories of Spark: streaming, machine learning, graph processing, and SQL applications. For each of these categories applications, distinct features have been identified by resource consumption (CPU, memory, disk, and network), data flow, and communication pattern, which may negatively affect the execution time of jobs. SparkBench covers many workloads that are used by Spark. SparkBench has the ability to measure job execution times and data processing rates (MB/second). The results illustrate that increasing task parallelism to fully leverage CPU resources can reduce the time required for job executions, however, over committing CPU resources may negatively impact execution time as a result of CPU bottleneck [11].

Although there are many other specific benchmarks for Big Data processing such as HiBench[4], GridMix[5], PigMix[6], and LinkBench[7] for Hadoop, there are no many open source benchmarks that are specifically designed for Apache Spark. This kind of Spark open source benchmark helps researchers to better understand the performance metric that may affect performance of the Spark application.

Lu et al. [81] presented a benchmark for distributed computing frameworks that is called *StreamBench*. The authors implemented it to enable adopting it with any stream processing systems. This benchmark utilizes a messaging system that has the ability to mediate between data stream providers and consumers. They utilized a messaging system in which generated input streaming is placed to gain insight into metrics during computation. They used Apache *Kafka* and *ZooKeeper*. In addition, they proposed four types of workload suites, including a performance workload suite, a multi-recipient performance suite, a fault tolerance suite and a durability suite [81]. These workload suites are used to measure and evaluate performance, fault tolerance and durability.

[4]https://github.com/intel-hadoop/HiBench
[5]https://hadoop.apache.org/docs/r1.2.1/gridmix.html
[6]https://cwiki.apache.org/ confluence/display/PIG/PigMix
[7]https://libraries.io/github/intel-hadoop/linkbench

Lu et al. [81] validate their benchmark by applying it to Apache Spark Streaming and Apache Storm to compare the two systems. According to their performance suites, under default configuration, Apache Spark Streaming throughput was five times greater than the throughput of Apache Storm. Regarding the latency, Apache Storm was considerably less than Apache Spark with a normal workload, but latency in Storm will exceed Spark if the workload becomes more complex and more scalable. With regard to fault tolerance ability, the authors found that Apache Spark outperforms Apache Storm through their durability test [81].

Although there are some benchmarks for distributing data computing systems, there are still fewer specific benchmarks that target open source Big Data distributed streaming frameworks in the literature. An interesting point mentioned by the authors [81] was the change in latency when the data became more complex. Spark was not affected in the same way as Storm. Investigating the impact of changing the workload to be more complex and scalable is crucial to understand the relation between workload and latency.

Cloud incident management has been investigated by Munteanu et al. [82]. They cover the incident life cycle and proposed a general architecture for incident management, particularly for cloud computing. Their incident life cycle is divided into different stages, including prevention, detection, analysis, containment, and recovery stages. In addition, they showed a number of current solutions such as PagerDuty, Oracle Enterprise Manager, IBM SmartCloud Control, and VMWare vCloud Suite. Regarding the standards, they mentioned two approaches including *ISO 20000* and *ITSM* that have come from *ITIL*. Although Munteanu et al. [82] presented some cloud incident management tools, most of them are commercial tools and not available as open source for academic research.

Few works exist for anomaly detection in Spark. Kay et al. [83] 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 involves analysis of blocking time, using white-box logging to measure time execution for

each task in order to pinpoint bottleneck root-causes.

The authors [83] find that network optimization can enhance job execution time by at most 2%. Therefore the network is not the main source of bottlenecks because there are more data transmissions to and from disks than over the network. Optimizing straggler tasks could enhance job execution time by at most 10% [83]. Blocked time analysis revealed two main causes of Spark stragglers: Java garbage collection and time to transfer to and from the disk. The authors conclude that early finding the main cause of stragglers may enhance not only stragglers jobs, but also non-straggler job at runtime by finding and fixing misconfiguration, which may cut execution time in half in some cases [83]. Finally, the authors claimed that jobs are usually bottlenecked by the CPU. Therefore, this thesis focuses more in CPU anomalies. In addition, some block time analysis lacks the generality of the general purpose of distributed systems performance analysis tools because it needs to use additional instrumentation within the system, unlike black-box analysis.

## 2.5 Methods and Techniques

In machine learning, classification techniques are needed to classify a dataset into predefined classes of items in order to construct a classifier that can predict the class of each item in the dataset according to the class labels of this dataset [84]. There are various problems in different fields (e.g., business, medicine, education etc.) that can be solved using machine learning classification techniques. Compared with the statistical classification, machine learning classification techniques outperform conventional statistical classification because many statistical techniques depend on the assumption that the data are generated from a particular distribution. Therefore, the statistical model will be accurate when that assumption satisfies the model before it can be used [39]. This constraint limits the statistical classification model, especially when assumptions about the distribution of the data do not hold, as in cases that involve highly dimensional real-time datasets.

In machine learning, there are two well-known challenges: 1) making the training error as small as possible in the training phase to solve underfitting issues, 2) reducing the gap between training error and test error to avoid overfitting the model [85]. The underfitting

issue happens when the model obtains a high error rate on the training dataset. On the other hand, the overfitting issue happens when there is a large gap between the error rate in the training phase and test phase.

One of the well-known critical issue for machine learning algorithms (ex: neural networks and other classifiers) is the feature selection of appropriate input features. The objective of feature selection is to discover the smallest set of input features and at the same time can achieve a desirable predictive performance. Zhang [86] points out that it is crucial to reduce the number of input features for the classifier to achieve satisfactory accuracy with less amount of computation in the model. One of the well-known technique for feature selection is Principle Component Analysis (PCA), which is a statistical way to reduce dimension without losing the important information that may negatively affect the quality performance of the classifier. PCA is applied on the input dataset as a step before training the neural network. The technique is considered a linear dimension reduction technique that has some limitations and it is not suitable for some complex problems with nonlinear correlation structures. This limitation can be avoided by directly applying the neural network to conduct dimension reduction [86].

In this section, different machine classification techniques are presented in order to understand the most suitable techniques that can be utilized in examining the performance anomalies that may occur in our testbed cluster. These techniques include neural networks, decision trees and the nearest neighbor methods.

### 2.5.1 Neural Network

Neural networks are a popular technique for many classification problems. This is because the neural network model is a data-driven self-adaptive method that can adjust itself to the datasets without requiring knowledge about the distribution or function of the used model [86]. Another reason for the popularity of neural networks are that it has the ability to approximate any function with arbitrary accuracy. This has caused the neural networks to be considered a universal functional approximation [86]. In addition, the neural network model is a nonlinear model that offers suitability and flexibility for many

real-world complex modules, which they are considered to be a nonlinear model.

In the real world, the neural network classification models have been successfully utilized in many different fields. For example, it has been applied in speech recognition [87, 88], material inspection [89, 90], motor monitoring [91], error recognition [92, 93], nuclear power plants [94, 95], clinical decision-making [96, 97, 98], cancer detection [99, 100, 101], and intrusion detection [102, 103, 104, 105].

According to [86], learning and generalization are considered to be the most prominent topic in the neural networks. Learning refers to the ability to approximate the underlying adaptive behavior from the training dataset. The generalization is one of the main advantages of using the neural network, which offers the ability to generalize the network by classifying (predicting) the class of the input dataset from the same classes of the training dataset, even if that input item has never been seen before [86]. This feature allows the classifier to achieve a desired accuracy level when classifying new or unknown objects. The *generalization error*, also called *test error*, is defined as the expected error on the new input dataset, which is different from the training dataset that is used for the training phase. The training and generalization error can vary depending on the size of the training dataset [85].

Usually, the neural network effectively fits training data with a very low level of bias, but there are some possibilities of risk that may cause overfitting issues that instigate variances in generalization [86]. The overfitting issue happens when there is a large gap between the error rate in the training phase and test phase. A study in [106] illustrates that the variance in machine learning is more critical than the learning bias in classification performance. Therefore, many studies have been conducted to solve and mitigate the negative impact of overfitting. Some of these solutions include cross validation [107], training with penalty terms and weight decay and node pruning [86].

**Neural Networks Components**

The neural network model has a network of processing elements that are called neurons. Each of these neurons has an input and output in which the input value affects the internal

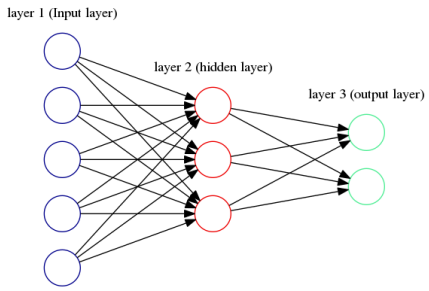



Figure 2.4: Feedforward Neural Networks.

state of the neurons in generating the output result. The neural network model is created by connecting a number of these neurons that are partitioned into layers to form a directed graph. The following subsection provides an overview about the main architecture of a neural network, which are neurons and connections.

**Neurons:** A neural network model has processing units called neurons that act as neurons in the human brain. The neurons are interconnected with each other and generate a sequence of real value for activations [108]. These neurons are distributed among the input, middle (also called hidden) and output layers. Therefore, there are three types of neurons or nodes: input, hidden and output neurons. The input neurons are in the input layer and they receive data from outside the neural networks. The hidden neurons are in the middle layers and are activated by weighted connections from the active neurons in the previous layer. The output neurons are in the output layer and they use the activation function in order to map the desired output.

The behavior of a neural network model is determined by a set of real-value parameters that are called *weights* and *biases*. Another task for these neurons is to adjust these *weights* of connections between neurons. Every hidden layer consists of many units (neurons), which operate in parallel and each one is represented as a vector-to-scalar function [85]. This means that each neuron receives many inputs from many neurons in the previous layer to compute its activation value, as shown in Figure 2.4. The activation value is calculated by the activation function that activates neurons by receiving an input from many other neurons in the previous layer. A well-know activation function is the *Sigmoid* and it is defined in Equation (2.1), where $x$ is input [85, 109]. This function is used to

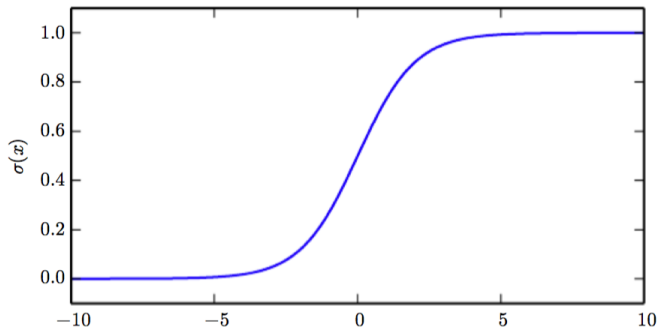



Figure 2.5: The Sigmoid function

produce output of Bernoulli distribution of range between 0 and 1 distribution as shown in Figure 2.5. The *Sigmoid* function is saturated when its argument is a large positive value or a very small negative value, which make function flat and insensitive to the small changed in its input arguments. This function is used in neural networks to activate neurons because it squashes the output to be always between 0 and 1.

$$\boldsymbol{\sigma(x)} = \frac{1}{1 + e^{-\boldsymbol{x}}} \tag{2.1}$$

**Connections and Weights:** The neural networks model contains many connections between layers, which transfer the outputs of neurons (predecessors) in one layer to the neurons (successors) in the next layer as an input. Therefore, every connection between hidden layers has a predecessor neuron and successor neuron. In addition, every connection is randomly assigned a weight $w$. Each time a neural network model is trained can result in a different output due to different initial weights and bias values. Consequently, different neural networks trained on the same problem can give different outputs for the same input features. Sometimes, retraining neural networks several times may enhance the performance of the F-score. This is because the weights of these connections significantly affect the prediction accuracy of the neural networks [85]. If the input feature $x$ has a connection with a positive weight, then increasing the value of this feature will increase the value of the prediction for $y$. Also, if the input feature has a negative weight, then increasing the value of this feature will decrease the value of prediction for $y$ [85]. If the feature has a zero weight, then it does not have any effect on the prediction. A simple

example of using weight is a linear regression, which is shown in equation (2.2).

$$\boldsymbol{y} = \boldsymbol{wx} + \boldsymbol{b} \tag{2.2}$$

The parameter $b$ is called a bias parameter, in which the output of the function is based on $b$ in the absence of input features. Linear regression is one of the simplest learning algorithms that can be used in neural networks and it provides a simple overview on how choosing the $w$ can significantly affect the general performance of the neural network algorithm [85].

**Feedforward Network**

Feedforward networks are the most straightforward neural networks. An example is shown in Figure 2.4. The main function of this type of network is to approximate function $f^*$. For example, the classifier in equation (2.3) aims to map an input $x$ to a specific class $y$:

$$y = f^*(x) \tag{2.3}$$

In the feedforwad neural networks, the mapping is defined as in equation (2.4), where parameter $\theta$ results in a better function approximation [85].

$$y = f(x; \theta) \tag{2.4}$$

This network has three types of layers, which are input layer, hidden layer and output layer. Each layer may have one or multiple processing elements (neurons). These neurons are connected with another neurons in different layer and each connection has a weight that can be adjusted in the training phase. Information is directed from the input to the output layer, which creates a directed acyclic graph [110]. There are no feedback connections in this type of network. The feedforward network is the basis of many other networks such as *convolutional* and *recurrent networks* [85].

Using this type of networks with a linear model has some limitations that can be overcome.

Linear regression and logistic regression are well-known linear models, which are considered to effectively fit the model. However, the capacity of a linear model is restricted to the linear function, as this model does not have the ability to understand the interaction between any two input variables [85]. In order to overcome this limitation and to make the linear model able to represent the nonlinear function of $x$,$\phi$ is used as a nonlinear transformation to provide new representation of the input function $x$ [85].

**Network Architecture:** In the neural networks, choosing the number of layers and neurons and how they are connected to each other plays an important role in obtaining a high level of accuracy for any classifier that uses a neural networks. In general, the network is divided into layers, and each layer has a set of neurons that operate in parallel in the same layer. Inside every layer, there are no connections among neurons. The layers are organized in a way that one layer is a function of the previous layer [85].

Equation (2.5) defines the first layer and Equation (2.6) defines the second layer.

$$h_1 = g_1(W_1 x_1 + b_1) \tag{2.5}$$

$$h_2 = g_2(W_2 h_2 + b_2) \tag{2.6}$$

This chain-based architecture needs to have the suitable depth of the network (number of layers) and the width of each layer (number of neurons in layer). Sometime choosing a network with only one hidden layer can satisfy the fitting process for the training dataset. Moreover, a neural network model that has more layers can often use less neurons in each layer and can generalize a model to the test dataset, but it may require more effort to optimize the model [85]. The ideal neural networks architecture for a specific task can be defined by performing many experiments to get the most desired accuracy.

In the real world, neural networks are considered to be more diverse, but they have the same basic concepts of depth and width regarding the feedforward network. Depending on the defined task of the neural networks, the architecture of the network may change, such as in the convolutional network and recurrent network. The convolutional network is

a specialized architecture that is used for computer vision applications, whereas the recurrent network is generalized from the feedforward neural networks for sequence processing purposes [85].

### Back Propagation Neural Networks

In the previous type of neural networks (feedforward), the information inside the network flows from the input $x$ toward hidden layers and then finally to produce output $y$. This process is called *forward propagation* algorithm. On the other hand, *back propagation* algorithm is about allowing the information about error to propagate back through network to compute the gradient descent [111, 85]. Nielsen describes the back propagation algorithm as the workhorse of the learning process in neural networks [112]. This is because it shows the significant effect of changing the weights and biases on the behavior of network and cost function.

The term *back propagation* comes from computing the error vector backward, starting from the last layer in the network [112]. Before the back propagation is initiated, there are other processes that must be done first. These processes include calculating the activation values of units and propagating them to the output units. Then the cost function will be applied to compare the actual output error results $y_P^o$ with the desired output values $d_o$. Usually, there will be a signal error $\delta_o^p$ from each unit in the output layer. The goal of back propagation is to reduce the amount of differences between the actual output and desired output to as much as possible [109]. This can be achieved by backward passes through every hidden layer in order to carry the error signal to all units in the networks and to recalculate the weights of connections in the hidden layers. Equation (2.7) provides the recursive procedures that compute all the error signals for all the units in the hidden layers [109]. The error measure can be written as a function of the network inputs from hidden layer $o = 1$ to output layer $N_o$. $w_{ho}$ is the weight of the connection from unit in hidden layer to output layer. The $F'$ is the squashing function for the $k^{th}$ unit and is evaluated at the network input ($s_h^P$) for that unit.

$$\delta_h^p = F'(s_h^P) \sum_{o=1}^{N_o} \delta_o^p w_{ho} \tag{2.7}$$

**Neural Network Regression model**

Beale et al. [113] claim that the neural networks can fit any practical function. Feed-forward neural network model is used with activation function (e.g., sigmoid) in the hidden layer and linear output neurons to fit multi-dimensional mapping problems, if provided with consistent input dataset size and enough number of units in each layer. More neurons and layers require more calculation; however, they allow the network to solve complex problems effectively.

For fitting problems, the neural network model is used to map between numeric input datasets to a set of numeric output targets. Depending on the nature of the problem and input datasets that are needed to be fitted, we may use different neural networks architectures. In this thesis (Chapter 5), our target is to fit the input data (system metrics) to predict the system throughput. The steps that are followed to develop the neural network models include data collection, network creation, network configuration, initialize the weights and biases, network training, validation, then the network can be used for testing new datasets.

### 2.5.2 Decision Tree

In this section, a brief overview about *Decision Tree* algorithm is provided because it is used in this thesis for comparison with our proposed models. *Decision Tree* is another type of predictive machine learning algorithm that is used for both classification and regression models [114]. It is called *Decision Tree* classification when it used for classification purposes, whereas it is called regression tree if it used for regression purposes. In this section, we focus more on the classification tree to classify the performance if there is any performance anomaly or not.

*Decision trees* are widely used due to their simplicity [114]. The classification tree algo-

rithm is utilized in many different fields. For example, it is used in finance [115], agriculture [116], astronomy [117], control systems [118], manufacturing [119], medicine [120] remote sensing [121], and anomaly detection [122, 123].

### Decision Tree Characteristics

Usually, classification trees are represented as hierarchical graphical structure, which facilitates the interpretation when compared to other machine learning techniques [114]. A *decision tree* model has a set of nodes, which form a *directed tree*. The main node in the tree is called *root node* that has no incoming edges, whereas all the other nodes in the tree have only one incoming edge. The node with outgoing edges is called *internal node* or *test node*. The others nodes with incoming edges, but without outgoing is called *leave node* or a *terminal node*. Every internal node in the tree divides the input space into two or more spaces depending on the function of input values [114]. In a simple *decision tree*, each *test node* receives an input instance to split the tree to subtrees according to this input.

Instances are classified by routing them from the main node *"root node"* of the tree down to the last specific node *leaf* in the tree [114]. This routing process starts from the top to bottom of the tree and it is achieved by applying classification test for every node along with the path to route input instance to the final *leaf node* in that tree. Each node in the tree has a label of its attributes which allows this nodes to classify according to these attributes. Each branch in the tree has a label of its corresponding values [114].

The complexity of *decision tree* is measured by some metrics include the total number of nodes, the total number of leaves nodes, the depth of tree, and the number of attributes in all the tree. There are some methods that are used in the literature to control the complexity of tree, which include stopping criteria and pruning method [114]. Sometime, a complicated *decision tree* may have some limitations in the ability to generalize model. In some cases, a simple tree may outperform the complicated tree even if the complicated tree shows better accuracy in the training phase [114].

### Decision Tree Induction Algorithms

Inductive machine learning is about the process of learning a set of rules from the training dataset to implement a classifier that has ability to generalize new test data. This section provides a general overview about some popular *decision tree* induction algorithms. These *decision tree* induction algorithms include *ID3* and *C4.5*. These types of algorithms use splitting criterion or pruning methods to control the complexity of tree [114]. The advantages and disadvantages are presented for each algorithm.

*ID3* algorithm is considered to be a simple *decision tree* algorithm [124]. The *ID3* does not apply pruning method and it does not handle numeric attributes or missing values. *ID3* uses information gain as a splitting criterion to make algorithm stop growing if all instances are belonging to a single value of target feature or if the best information gain is not greater than zero [114]. The main advantage of *ID3* is its simplicity to be used for learning purposes. Although, *ID3* is simple, it has many notable disadvantages include overfitting the training data and accepting only nominal attribute. Therefore, any continuous data needs to be converted to nominal data before using *ID3* [114].

Many drawbacks in the *ID3* have been tackled in *C4.5* algorithm [125]. *C4.5* algorithm is an evolution of the *ID3*. *C4.5* uses gain ratio as splitting criteria. The splitting process ceases if the number of instances that needs to split is below a specific threshold. After the growing phase, the error based pruning method is applied. This method is used to remove all branches that do not add any contribution to the performance accuracy and replaces these branches with leaf nodes [114]. Not like *ID3*, the *C4.5* has the ability to handle numeric attributes and also it handled missing value from the training dataset by using corrected gain ration criteria [125]. Another advantage of *C4.5* is the ability to handle any continuous attributes by splitting the range of value attributed into two branches (subsets). Therefore, all the attribute values above a specific threshold are located in the first branch, whereas the all the other attribute value under the threshold are located in the second branch.

There is an updated version of *C4.5* which is *CC5.0*. This updated version is a commercial

version and it provides some improvements which it has been claimed that it offers more efficient features than *C4.5* algorithm. These improvements include memory, predictive performance, and computation time. The comparison in [114] between *C4.5* and *CC5.0* shows that running experiment with *C4.5* takes 90 minutes, whereas *CC5.0* takes only 3.5 seconds .

### 2.5.3 Nearest Neighbor

Nearest Neighbor machine learning technique can be applied for both classification and regression models. Goodfellow [85] points out that $k$-Nearest Neighbor does not have an explicit training or learning phase. Thus, it is called lazy algorithm. The main concept of nearest neighbor technique has an assumption that the normal data instances usually occur in dense neighborhoods, whereas the abnormal data instance occurs far from its closest neighbors [39]. This type of machine learning technique requires a distance measure to be defined between two input data instances. This distance measure can be calculated in many different ways. Euclidean distance is often used for continuance attributes and it is shown in Equation (2.8) [38].

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2.8}$$

On the other hand, a simple matching coefficient is used for categorical attributes [39]. There are two main nearest neighbor based techniques that are used for anomaly detection purposes. The first technique uses the distance between the data instance and its $k^{th}$ nearest neighbor as a score of anomaly. The second technique it about computing the relative density of each data instance in order to compute the score of anomaly [39]. The $k^{th}$ nearest neighbor is considered to be a non-parametric technique which does not generally make any assumption on the underlying distribution and the module structure is determined from input dataset [39]. This non-parametric feature makes $k^{th}$ nearest neighbor an ideal choice if the distribution of dataset is unknown. The score of the anomaly for the data instance is known as the distance to its $k^{th}$ nearest neighbor in the input data set.

The disadvantage of the $k$ nearest neighbor is the lack of ability to learn about the discrimination between the input features [85]. For example, if there are a group of features $x_1$ to $x_{50}$ and the only feature $x_5$ is noticeably influencing the result of classification (i.e., $y = x_5$), then the $k^{th}$ Nearest Neighbor will not be able to detect the significant effect of $x_5$ feature. Therefore, the classification output $y$ will be calculated based on most of the features $x_1$ through $x_{50}$, but not only on feature $x_5$. This feature makes $k^{th}$ nearest neighbor affected by noise and irrelevant input features.

## 2.6 Summary

In the previous sections, the existing techniques that are developed in statistical analysis and machine learning domains are investigated, especially those studies on anomaly detection and interference prediction techniques. Table 2.2 shows a summary of these state-of-the-art techniques. Some concerns are related to the use of only statistical methods, such as assumptions about the distribution of data, and many statistical methods demonstrate sensitivity to variations in this assumption, especially for real-time datasets. However, machine-learning techniques do not require this assumption and facilitate the identification of performance anomalies, especially for distributed in-memory processing technology, such as Apache Spark.

During my PhD, there have been some challenges that occurred during my research. According to Chandola et al. [39], the sole use of a supervised anomaly detection method may cause some challenges that can limit the scope of its application, especially within dynamic environments such as Big Data and cloud systems. The first challenge is the size of the dataset of anomalous instances, which is less than a normal instance in the training data. However, this challenge of imbalanced class distribution has been overcome in the literature by using some machine learning techniques [126, 127, 128, 129]. The second challenge is that obtaining accurately labeled data, particularly for the performance of anomaly classes, is often arduous. However, some approaches provided in the literature address this issue. One approach is to inject artificial anomalies in the system to obtain enough anomaly labeled datasets for the training phase [130, 131].

The third challenge is obtaining accurately labeled datasets for new and unknown anomaly classes in the supervised machine learning technique. To overcome this challenge, unsupervised and semi-supervised learning techniques can be used (if there is a need) for the detection of performance anomalies within in-memory processing platforms. The fourth challenge is a lack of comprehensive open source datasets for Apache Spark that can offer all the system metrics with different types of workloads. Therefore, the needed datasets will be generated from the current benchmarks and from real Big Data applications to collect enough datasets for our proposed research; we will make them available as an open source for public.

Although some challenges have arisen during my research, there may also some opportunities to investigate new research areas. Although Apache Spark and containerized batch systems are developing gradually, there are still shortages in comprehensive performance analyses that are specifically built for Spark and containerized batch services within cloud that are used to detect performance anomalies and predict system interference. The performance of such systems can vary considerably depending on many factors, such as the type of input data, data size, application design, system configuration, used algorithms, and available computing resources. These factors make anomaly detection and interference prediction more challenging, especially for critical the applications in these distributed systems. Therefore, there is a need to deeply investigate performance bottlenecks and pinpoint the cause of a performance anomaly and its interference to improve the overall system performance.

Identifying the main performance metric that may affect the performance of the Big Data application is a first step in understanding the detection and prediction of performance anomalies and any potential interference with performance. Addressing these challenge offers valuable opportunities to investigate new hybrid-learning techniques for Big Data anomaly detection and interference prediction.

# Chapter 3

# AI Driven Anomaly Detection Methodology for In-Memory Systems

## 3.1 Introduction

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* arise from resource contention, failures, workload unpredictability, software bugs, and several other root causes. For instance, even though application workloads can feature intrinsic variability in the execution time because of the 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 run times that are perceived by end-users as being anomalous.

Research on automated anomaly detection methods is important in practice since the late detection and slow manual resolutions of anomalies in a production environment may cause prolonged service-level agreement violations, possibly incurring significant financial penalties [5, 6]. This leads to a demand for effective performance anomaly detection

methods in cloud computing and Big Data systems, which are both dynamic and proactive in nature [7]. The need to adapt these methods to the production environment with very 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 baseline performance through a large space of monitoring metrics, to identify normal and anomalous patterns later on [8].

In this chapter, we develop a neural networks 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, recall, and F-score of the classifiers. We also compared methods that are agnostic of the workflow of Spark jobs by using a novel method that leverages the specific characteristics of Spark's fundamental data structure -RDD- to improve anomaly detection accuracy.

Our experiments demonstrate that neural networks are both effective and efficient in detecting anomalies in the presence of 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 chapter provides an evaluation against three popular machine learning algorithms, such as decision trees, nearest neighbor, and SVM, as well as against four variants that consider different monitoring metrics in the training dataset. In addition, the proposed methodology is examined for different types of overlapped anomalies.

## 3.2 Motivating Example

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 the percentiles of the cumulative distribution function (CDF) of task execution times.

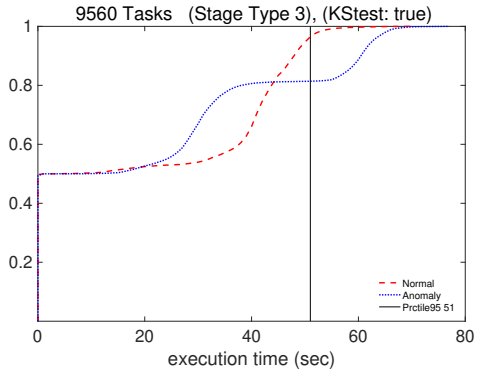
Our goal is to use CDF percentiles to discriminate whether a given task has experienced a performance anomaly or not.

We run a machine learning workload (K-means) on Spark system with nine different types of tasks. K-means workload is a popular machine learning algorithm, and it has high CPU utilization when it is run on the Apache Spark system. As discussed in Section 3.4.1, which details the experimental environment and process, we inject CPU contention using the *stress* tool for a continuous period of 17 hours, 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 with running the same workload without *stress*.

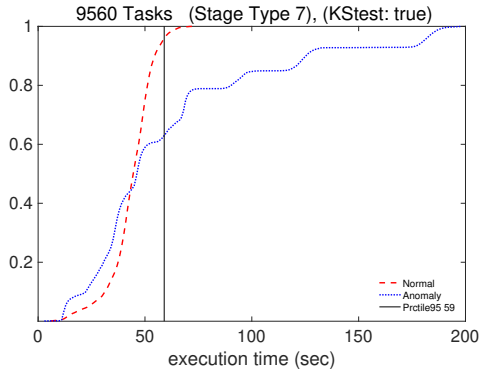
We then use the obtained task execution times to estimate the empirical Cumulative distribution function (CDF) for the execution time of tasks conditional on their stage; that is the population of samples that defines the CDF corresponds to the execution time of all the tasks that are executed in that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

We then determine the $95^{th}$, $75^{th}$, $50^{th}$, $25^{th}$, and $10^{th}$ 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, achieving only 0.2% for the F-score.

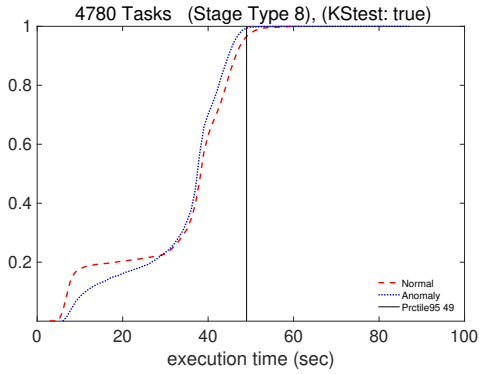
We used a two-sample Kolmogorov-Smirnov test to compare the two Apache Spark stages CDFs with and without anomalies [132]. The test result is true if the test rejects the null hypothesis at the 5% level and false otherwise, as shown in Figure 3.1. The three types of Apache Spark stages in Figure 3.1 illustrate that the three stages of CDFs obtained in an experiments with and without the injection of CPU contention. The three CDFs for the three different types of Apache Spark tasks make it difficult to determine whether there is an anomaly or not. For example, Figure 3.1(a) has a noticeable difference in the CDFs results for normal and abnormal Spark performance. On the other hand, Figure 3.1(b) also




(a) CDFs for stage type 3




(b) CDFs for stage type 7




(c) CDFs for stage type 8

Figure 3.1: CDF for the three types of Spark tasks when there is a short 50% CPU stress that affected tasks from stage type 3.

has a noticeable difference between the two experiments, but there were no performance anomalies that occurred during all the Spark tasks in stage 7. In addition, the CPU anomaly causes a delay while processing the tasks. This delay propagates through the Apache Spark DAG workflow and therefore also affects tasks that did not incur anomalies period.

In conclusion, this motivating example illustrates that CDF-based anomaly detection in Spark only at the level of execution times is significantly more 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.

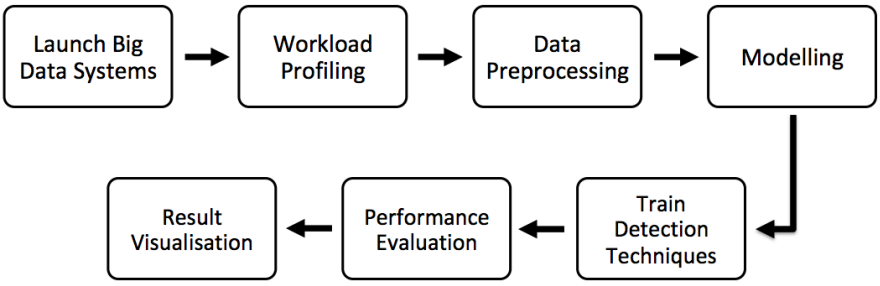



Figure 3.2: Methodology for anomaly detection

## 3.3 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 Figure 3.2. The following subsections discuss the proposed methodology covering the neural network model, feature selection, training, and testing.

### 3.3.1 Neural network model

Our methodology revolves around using a neural networks to detect anomalies in in-memory 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 networks. The scaled conjugate gradient training is normally faster than standard gradient descent algorithms [133].

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* ($\frac{\exp(x_i)}{\sum_j \exp(x_j)}$) transfer function is used in the output layer to handle classification problems with multiple classes. Then *cross-entropy*

$(-(y\log(p) + (1-y)\log(1-p)))$ is used as a cost function to assess the neural networks 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 [134].

The input layer contains a number of neurons equal to the number of input features of Spark metrics. The size of the hidden layer is determined by using a "trial and error" method, trying all the possible numbers between the sizes of input neurons and output neurons [135]. A hidden layer with ten neurons has achieved the most accurate results for our situation. The output layer of the neural network contains a number of neurons equal to the number of target classes (normal + types of anomalies), where each neuron generates 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 is four neurons.

Backpropagation algorithm is used to allow the information about classification error to propagate back through the network to compute the gradient. This provides a significant effect of changing the weights and biases based on the behavior of network and cost function. Scaled conjugate gradient backpropagation is used for the training process to updates weight and bias values according to the scaled conjugate gradient method.

### 3.3.2 Model training and testing

In the training process, as commonly conducted, the input dataset to the model is divided into three smaller datasets for training, validation, and testing. The training dataset is used for calculating the gradient and updating the network weights and biases. During the training process, the weights and biases are constantly updated until the magnitude of the scaled conjugate gradient reaches the minimum performance gradient or the number of validation checks. The training process stops if the magnitude of the gradient is less than a predefined threshold (e.g., $10^{-5}$).

The validation dataset is used for validation purposes where the error rate is decreased

before overfitting the dataset by checking the number of validation checks. The number of validation checks is about the number of successive iterations that the validation performance fails to decrease (e.g., six successive iterations). After convergence, we save the weights and biases at the minimum error for the validation subset. This method is called *early stopping* [136]. This helps to avoid overfitting issues.

We test our proposed model with new input dataset that was collected from the Apache Spark system. This new dataset was manually labeled in order to classify normal and anomalous behavior of target system. With this kind of classification techniques, accuracy is not a sufficient evaluation for a model with an imbalanced class distribution of data [114]. Therefore, sometimes the accuracy estimation may not correctly reflect the quality of the classifier. To avoid this issue, *sensitivity* and *precision* measures are used to evaluate the anomaly detection classifiers, which are standard metrics for quantifying the accuracy of the classifiers [137]. The following are the anomaly classification classes and their notations:

- True Positive ($tp$): The detection method correctly detected anomaly
- True Negative ($tn$): The detection method correctly did not detect anomaly when it did not exist.
- False Positive ($fp$): The detection method detected anomaly when it does not exist
- False Negative ($fn$): The detection methods missed detection of an anomaly when it actually exists

$$R = \frac{tp}{tp + fn} \tag{3.1}$$

$$P = \frac{tp}{tp + fp} \tag{3.2}$$

$$F_1 = 2\frac{PR}{P + R} \tag{3.3}$$

*Sensitivity* is also called *Recall*, which assesses the quality of a classifier in recognizing positive samples; it is defined in (3.1). *Recall* will become high when the anomaly-detection method can detect all anomalies. The second classification performance metric is *Precision*, which quantifies how many samples are classified as anomalies are indeed anomalies. This

is defined in (3.2). The *Precision* assesses the reliability of the detection method when it reports anomalies [137]. The trade-off between the *Recall* and *Precision* is *F-Score*, which is a summary score, and it is computed as a harmonic means of *Recall* and *Precision*. The *F-Score* metric is defined in (3.3).

### 3.3.3 Feature selection

The objective of features selection is to discover the smallest set of features and at the same time can achieve a desirable predictive performance. Metrics performance collection is a crucial step to automated detection of anomalous performance behaviors by finding the relevant performance metrics with which to characterize the behavior of systems. Many metrics can be collected, but it is challenging to decide which metrics are more valuable to assess system performance and pinpoint the anomalous behaviors. Our methodology focuses on combining many performance metrics related to the general aspect of systems, which cover memory and CPU metrics. Other internal metrics of Big Data system can be utilized, such as metrics about the shuffle read and jobs information. This internal and runtime metrics may positively contribute to increase the accuracy of anomaly detection methods.

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-second period out of a total runtime execution. On top of these four combinations, a baseline experiment is run without any contention, in order to also train the anomaly detection method with non-anomalous traces.

More details about experimental testbed are discussed later in Section 3.4.1. Only one node (S02) is injected by 50% CPU and memory contentions, but there was no contention in the other servers in the Spark cluster (node S01 and S03). To evaluate the DSM1 (only CPU metrics), the following five scenarios have been examined:

1. Running the benchmark without any contention on CPU and memory.
2. Running the benchmark with continuous contention on CPU at 50%.

3. Running the benchmark with continuous contention on memory at 50% memory.
4. Running the benchmark with a short time (90 sec) of contention on CPU 50%.
5. Running the benchmark with a short time (90 sec) of contention on memory by 50% of free memory.

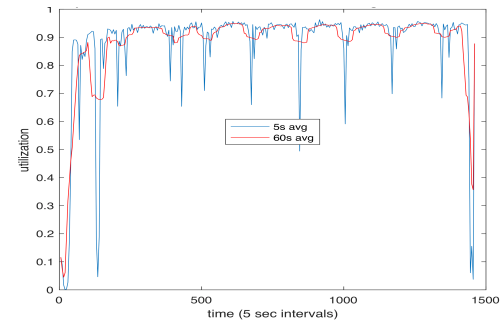
CPU utilization of S02 for the five scenarios are shown in Figure 3.3. The mean CPU utilization of S02 and S03 are shown in Figure 3.4.

Table 3.1: Comparison of the results among different contention scenarios on S02

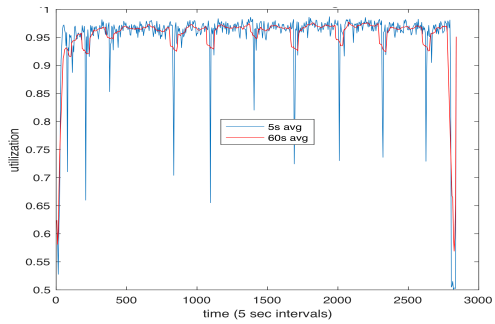
| **Server** | **Stress** | **Mean CPU** | **SD** | **Pr95** | **Pr99** | **Iqr** | **Memory** | **ExeTime Sec** |
|---|---|---|---|---|---|---|---|---|
| S01:Non | NO | 0.0203 | 0.0389 | 0.0950 | 0.2147 | 0.0177 | 89.3239 | 295 |
| S01:CPU50% | NO | 0.0174 | 0.0308 | 0.0663 | 0.1646 | 0.0176 | 89.5402 | 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 |

We compare the performance of a basic anomaly detection method, called DSM1, which relies solely on a neural networks 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 3.1 shows a comparison of the results among five different contention scenarios on S02 by running Spark K-means workload without contention, with continuous 50% CPU stress, with 90-sec 50% CPU stress, with continuous 50% memory stress, and with 90 sec 50% memory stress on only S02. Table 3.1 shows the mean CPU utilization, standard deviation, 95 percentile, 99 percentile, interquartile, memory usage, and total execution time.

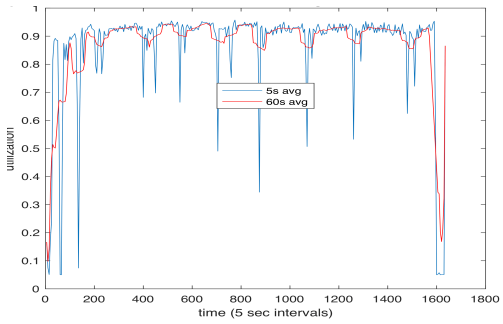
Table 3.1 shows that the different type and amounts of anomalies affect mean CPU and memory utilization in server S02 to detect the performance anomalies using DSM1 (only



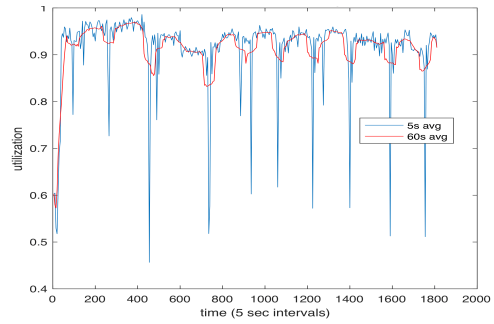

(a) S02 Without stress



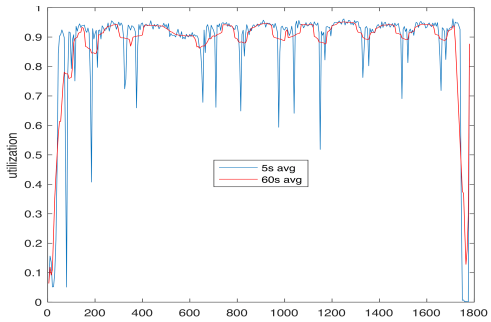

(b) S02 continuous stress by 50% CPU




(c) S02 with continuous stress by 50% Memory




(d) S02 with short time of stress on S02 CPU 50% for 90sec




(e) S02 with short time of stress on S02 Memory 50% for 90sec

Figure 3.3: S02 CPU utilization when single K-means workloads run on S02 With different scenarios of stress on S02 .

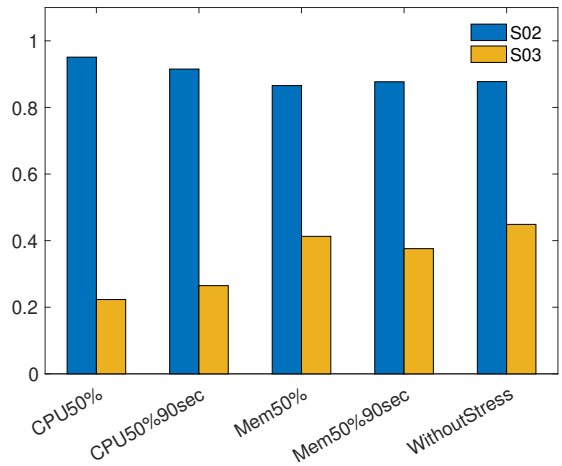



Figure 3.4: Mean CPU utilization of S02 and S03

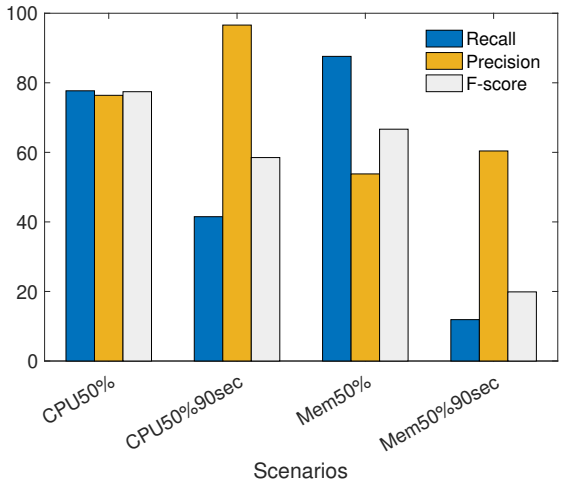



Figure 3.5: Neural network performance with DSM1 feature set in experiments with basic CPU and memory contention (continuous or 90-sec periods)

CPU metrics collected during the runtime of experiments). Machine learning has been used to detect these anomalies using neural networks driven methodology (discussed in Section 3.3.1) with DSM1. The classification performance metrics for a neural network trained on DSM1 are summarized in Figure 3.5.

The K-means workload does not heavily use memory (see Table 3.1). 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 Figure 3.5). Generally, short contention periods are harder to detect, as visible from the fact that a 90-second 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 model needs to train the algorithm with an enough dataset to detect memory contention. If we repeat the same experiment after adding memory monitoring metrics, referred to as the DSM2 dataset in Table 3.2, 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.

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.

### 3.3.4 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 networks trained using CPU utilization samples. In the beginning, the utilization of system resources are measured at one-second intervals. The CPU measurements are collected for DSM1, which are summarized later in Table 3.2. The CPU measurements are collected, which include the percentage of CPU utilization, the percentage of time that the CPUs are idle during an outstanding disk I/O request, the percentage of time spent in unintended waiting by the virtual CPU, and the percentage of time that the CPUs are idle.

DSM2 adds operating system memory usage metrics to the metrics employed by DSM1. We have experienced a low accuracy with DSM1 combination of CPU metrics. After observing the Spark system, we notice that the memory performance was noticeably impacted during the occurrence of different types of anomalies. Therefore, some memory metrics are also collected for DSM2, which include the metrics in Table 3.2. The measurements include free memory available, the amount of memory used, the percentage of memory used, the amount of memory used as buffers by the kernel, the amount of memory used to cache data by the kernel, and the amount of memory needed for the current workload.

The third method is DSM3 is build upon the list of metrics selected in [138], 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 3.2.

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 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.). These labels are used to classify performance to be either normal or anomalous behaviors in the training phase of the neural network.

Table 3.2: 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 |

## 3.4 Evaluation

In this section, we introduce an evaluation for the performance anomaly detection methodology proposed in Section 3.3. 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.

### 3.4.1 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 x Intel(R) Xeon(R) CPU 2.40GHz, 32 GB RAM, Ubuntu 16.04.3, and 130 GB Storage.
3. Node S03: 16 vcores x 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.

Table 3.3: *SparkBench* Workloads

| Application Type | Workloads |
|---|---|
| Graph Computation | Data Generator<br>Graph Generator |
| SQL Queries | SQL query over dataset |
| Machine Learning | Data Generator - K-means<br>Data Generator - Linear Regression<br>K-means<br>Logistic Regression |

### 3.4.2 Workload Generation

The effective use of benchmarks offers opportunities to examine and understand the system performance and identify potential areas for improvement and optimization. In general, there are two categories of benchmarks that are framework specific and multiple framework benchmarks. The first type is designed for a particular framework, such as *SparkBench* for Apache Spark [11]. The second type of benchmark is designed to include workload for various frameworks, such as BigBench [139], Gray sort [140], BigDataBench [76].

*SparkBench* covers the four main categories of Spark applications, including graph computation, streaming, SQL query, and the machine learning application [11]. *SparkBench* provides workload suites that include a collection of workloads that can be run either serially or in parallel [11]. Workloads include machine learning, graph computation, and SQL queries, as shown in Table 3.3. 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). 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 experimental runs 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 Figure 3.6, which has a single stage that contains a sequence of RDD and some of them are cached (green box for map RDD).

*SparkBench* provides a reliable feature, which is called workload suite [11]. This feature allows user to effectively control the level of workload parallelism to stress the Spark system. For example, the user can run $n$ K-means workloads and $m$ linear regressions in

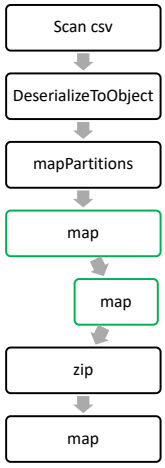



Figure 3.6: DAG diagram illustrates dependencies among operations on Spark RDDs for a single Spark stage within the K-means workload.

Table 3.4: 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. |

parallel, then launch serially $k$ SQL workloads. Therefore, the user has the ability to chain together different workloads with different parameter configurations and use varying levels of parallelism.

### 3.4.3 Anomaly Injection

Node S02 is used to inject anomalies into the Apache Spark computing environment using *stress* and *stress-ng* tools. Table 3.4 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 Section (Section 3.5).

### 3.4.4 Performance Data Collection

A summary of the performance data collected is shown in Table 3.2. In addition, Spark offers a configurable metrics system that allows Spark users to report metrics to a variety of sinks, such as HTTP, JMX, and CSV files. Corresponding to the Spark components (*workers, executor*, etc.), Spark metrics are separated in different instances. Each of these instances can configure a set of sinks to which metrics are reported [10]. We have used this feature to store all the required performance metrics that will be used as an input to the proposed anomaly detection technique. In addition, Spark also offers the ability to use different sets from third-party tools to monitor applications using the metric of the system [29], which can be used in the future.

## 3.5 Results

The experiments are conducted on a cluster (described in Section 3.4.1), which consisted of a master server (called S01) and two slave servers (S02 and S03). This cluster is isolated from other users during the experiments. A physical cluster is 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 methods.

### 3.5.1 Baseline Experiment

Three experiments with different types of anomalies are injected into the Spark cluster with random instant and random duration chosen uniformly between 0 and 240 seconds. Each experiment encompasses a single type of anomaly: CPU contention, cache thrashing, or context switching. We focus on evaluating neural networks trained on the DSM2 and DSM4 feature sets. Figure 3.7 shows the F-score obtained with the proposed neural networks classifier versus the nearest neighbor method used as a baseline. It is clear that the neural networks model outperforms the nearest neighbor 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.

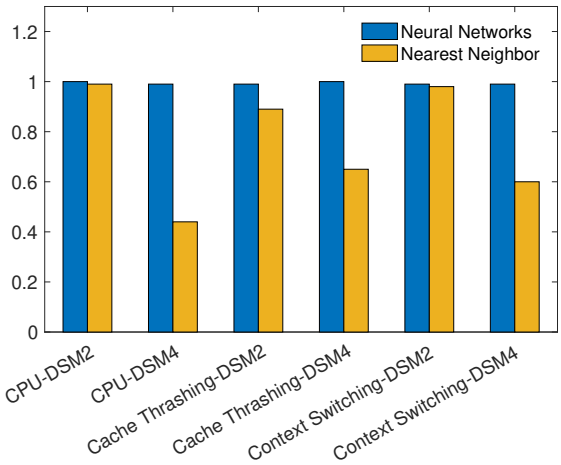



Figure 3.7: F-score performance metrics of neural networks and nearest neighbor for anomaly detection techniques

Table 3.5: Quality metrics for Neural Networks classification to detect the existence of anomaly that may be CPU, cache thrashing or context switching.

| Recall | precision | F-Score |
|---|---|---|
| 1 | .99 | 0.99 |

Another experiment has been conducted, which aims to generalize our model to detect the existence of anomalies that may be CPU stress, cache thrashing stress, or context switching stress. Three types of anomalies have been injected without overlapping to spark cluster to train and test the neural network model. The total number of tasks is 200K tasks which the are used for training, validation and testing. The output from the neural networks are two classes, which are 0 for normal and 1 for anomalous. Table 3.5 reflects the capability of the neural network model in detecting all the three types of performance anomalies.

### 3.5.2 Sensitivity to Training Dataset Size

Figure 3.8 depicts the impact of Spark workload size on the F-score for anomaly detection using DSM4 and three 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

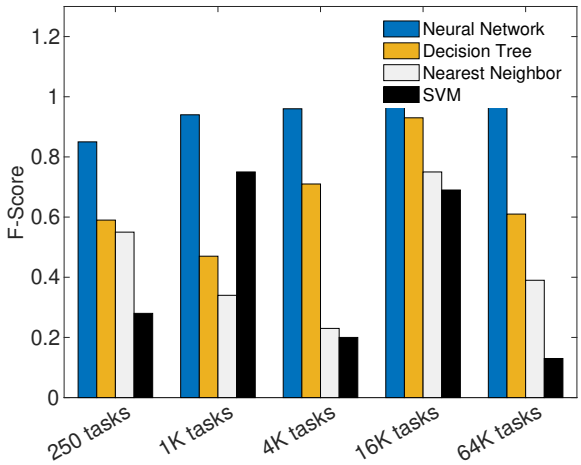


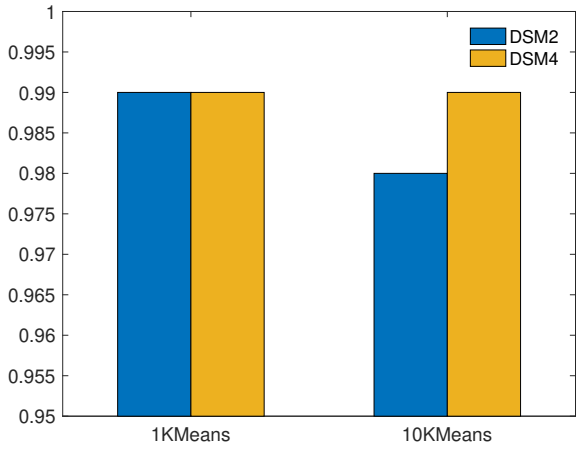

Figure 3.8: Impact of workload size on F-score for Neural Networks, Decision Tree, Nearest Neighbor, and SVM using ~~DSM2 and~~ DSM4 feature sets.

configuration. Figure 3.8 shows that the proposed technique achieved 85% F-score with a *micro* Spark workload (250 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.

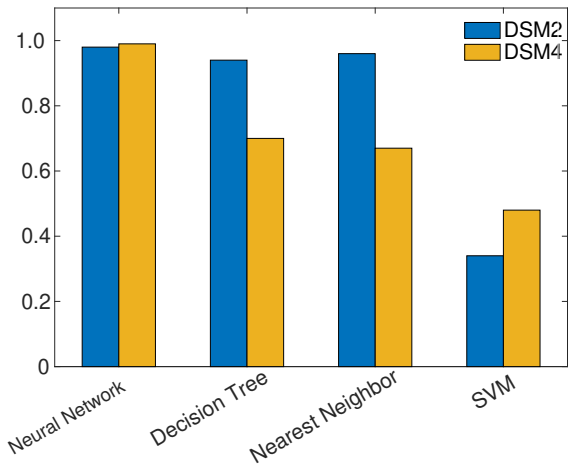
### 3.5.3 Sensitivity to Parallelism and Input Data Sizes

To evaluate the impact of parallelism, 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 and interference 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 a simple 50% CPU contention injection into the Spark cluster. Figure 3.9(a) 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 hours for execution. In order to evaluate the proposed anomaly detection methods, three machine learning algorithms have been applied to detect performance anomalies with DSM2 and DSM4 as inputs to the anomaly detection




(a) Comparison between DSM2 and DSM4 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

Figure 3.9: Impact of parallelism and input data size of workload on anomaly detection methods.

methods. These algorithms include neural networks, decision tree, nearest neighbor, and SVM. Figure 3.9(b) shows that the neural network model 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 minute, 3 minutes, 9 minutes, and 19 minutes respectively.

The neural network model is more effective than other algorithms. Standard classification methods, such as SVM, do not perform well with imbalanced and skewed datasets. It is challenging to get the optimal separation hyperplane for an SVM model trained with imbalanced datasets. The imbalance in datasets negatively influences the performance of most classifiers [141]. The results in Figure 3.9(b) prove that the neural network model is more robust than the other algorithms, which are affected by the size of the input data to workloads when the input data was increased to 64 GB.

### 3.5.4 Classifying Anomaly Types

In this section, 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 4 classes: normal, CPU anomalies, cache thrashing anomalies, and context switching anomalies. The classification is at the level of individual Spark tasks.

The total number of Spark tasks collected during the execution amount to a total of 400K tasks. Table 3.6 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.

### 3.5.5 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. Our model is trained 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 with cache stress, CPU with context switching stress, and cache with context switching stress. The proposed solution is validated with two types of Spark workload: K-means

Table 3.6: Classification of anomaly types using DSM3 and DSM4. (Recall=R, Precision=P, and F-score=F)

| *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 |

and SQL workloads, as shown in Tables 3.7 and 3.8. 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.

Table 3.7: Classification of 7 overlapped anomalies using DSM3 and DSM4: K-means workload. (Recall=R, Precision=P, and F-score=F)

| *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 | 0.27 | 0.28 |

Table 3.8: Classification of 7 overlapped anomalies using DSM3 and DSM4: SQL workload. (Recall=R, Precision=P, and F-score=F)

| *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 |

## 3.6 Conclusion

This chapter introduces a challenging case that utilizes the anomaly detection techniques in a complex Big Data and cloud computing environment. Among the various Big Data technologies, in-memory processing technology like Apache Spark has become widely adopted by various industries. Although Spark is gradually developing, currently there is still a shortage of anomaly detection methods for performance anomalies in Spark systems. This chapter addresses this challenge by developing a neural networks driven methodology for anomaly detection based on knowledge of the RDD characteristics.

Compared with CPU contention, memory contention does not have visible effects on the mean CPU usage in the Spark cluster. The anomaly detection method for Apache Spark is significantly enhanced by using the performance metrics for both the CPU and memory of the Spark cluster. In addition, there are no significant effects of parallelism when detecting the anomalies in the Apache Spark cluster using neural networks algorithm.

Our results demonstrate that the proposed method works effectively for complex scenarios where there are 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.

In terms of future work, the present method is sufficiently efficient to be considered for online anomaly detection. Deep Learning solutions may also be explored to learn more about complex features from the performance metrics of the Spark system, possibly leading to even the more accurate detection and prediction of critical anomalies.

# Chapter 4

# Hybrid AI Anomaly Detection Model for Big Data Streaming Systems

## 4.1 Introduction

Various open source Big data platforms exist for Big Data and data-intensive application development. Although each of these technologies has advantages for specific purposes (e.g., batch processing), they may not be ideal choices for other types of applications (e.g., real-time monitoring and streaming workloads). Big Data workloads can be analyzed using three main approaches: batch processing, stream processing, and micro-batch processing [22]. The analysis of a large amount of static data over a certain time period can be used for the batch processing approach. For real-time data analysis, a stream processing platform is ideal, especially when there is a need for microsecond responses. The micro-batch processing technique deals with streaming workloads as a sequence of smaller data blocks that have the ability to perform near real-time processing [22].

Among the various Big Data streaming technologies, in-memory processing technology, such as Apache Spark Streaming, has become widely adopted by industry because of its

speed, generality, ease of use, and compatibility with other Big Data systems. Here, We consider Spark Streaming workloads, in which analytic operations are applied to a resilient distributed dataset (RDD).

Machine learning algorithms have received growing attention from the research community because of their performance anomaly identification and diagnosis capabilities. Moreover, machine learning classification techniques are widely used to classify inputs based on their features into predefined classes to build a classifier that can predict the class of each item according to class labels. There are some popular classification techniques used for performance anomaly detection, such as neural networks, support vector machines (SVM) [49], and Bayesian networks [9].

With the growing complexity of Big Data and cloud systems, failure management system requires significantly higher levels of automation and attention [33]. Here, an anomaly is defined as an abnormal behavior during the execution of an applications. It could be the result of resource contention, hardware failures, misconfiguration, or several other root causes. Although some studies address the challenges of performance anomaly detection for batch processing [62, 53, 142], there is a lack of effective automated performance anomaly detection solutions specifically built for Apache Spark Streaming systems. There is a need for a technique that can be used to efficiently train a machine learning model to detect and predict performance anomalies within streaming workloads in production environments.

Anomaly detection within a Big Data streaming system is considered to be more challenging, especially for time-varying workloads and critical applications in distributed systems. Therefore, there is a need to deeply investigate in-memory processing technology performance, such as Spark Streaming performance, to pinpoint the causes of performance anomalies. This chapter addresses the challenge of anomaly identification by investigating new hybrid learning techniques for anomaly detection within in-memory Big Data streaming systems within cloud computing. We developed TRACK (neural neTwoRks Anomaly deteCtion for sparK) and TRACK-Plus which are two methods to efficiently train machine learning models for performance anomaly detection using a fixed number of experiments.

TRACK offers a tuning method capable of training a machine learning model with a limited

budget and limited number of experiments. TRACK revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the model to achieve the highest accuracy (95% F-score) within short amount of time (saves 80% less than normal time). TRACK-Plus is an automated fine-grained anomaly detection solution that adds to TRACK a second Bayesian Optimization cycle for fine-tuning the hyperparameters of artificial neural network configuration. The objective is to accelerate the search process for optimizing neural network configurations and improve the performance of anomaly classification.

In this chapter, validation based on several datasets from a real Apache Spark Streaming system is performed to demonstrate that the proposed methodology can efficiently identify performance anomalies, near-optimal configuration parameters, and a near-optimal training dataset size while reducing the number of experiments. Our results indicate that the reduction in experimental need can be up to 75% compared with naïve anomaly detection training. To the best of our knowledge, this solution is among the very first studies to provide a comprehensive methodology for both performance anomaly classification and the efficient optimization of artificial neural networks to detect anomalies within Spark streaming systems.

Our core contributions in this chapter are the following:

- Providing an updated discussion of existing anomaly detection techniques and algorithms that should be further researched by the community invested in this challenging problem space.
- Conducting a comparative analysis of four well-known anomaly detection techniques and algorithms to help system administrators in choosing the appropriate anomaly detection mechanisms for their in-memory Spark Streaming Big Data system.
- Addressing the challenge of anomaly identification and classification for streaming systems by investigating effective hybrid learning techniques for anomaly detection in Spark Streaming Big Data systems.
- Presenting a comprehensive methodology to automate the search for the ideal dataset size with which to train the detection model and automate the tuning of neural

networks hyperparameters, hence allowing for the identification of the most efficient network architecture and configuration.

The rest of the chapter is organized as follows: The prerequisite background on Apache Spark Streaming are given in Sections 4.2. This is followed by a motivating example in Section 4.3. The proposed methodology of this work is presented in Section 4.4, followed by a systematic evaluation in Section 4.5 and 4.6. Finally, Section 4.7 provides a discussion and the conclusions.

## 4.2 Background Information

The following subsections briefly describe the required background for Apache Spark Streaming, Bayesian Optimization, and neural networks.

### 4.2.1 Apache Spark Streaming

This subsection gives an overview of a popular current open source Big Data technology which is Apache Spark streaming. As a result of the speed and general purpose of Spark core engine, the engine offers an effective environment on top of the Spark core to support different types of libraries and computations. These libraries include Spark SQL, Spark Streaming, Spark MLib, and Spark GraphX applications. This upper stack offers the benefits of combining all components in the libraries in user applications [10]. The upper components library can obtain valuable benefits from the tight integration with the Spark core engine. First, upper components can gain benefits from the continuous improvement and optimization of the lower stack components (Spark core). Second, the cost and time saving are significant benefits for upper component users because they offer different types of services and run them within the same system instead of having multiple systems and running each service on an independent system. Therefore, operation and maintenance are reduced for system that needs configuration for deployment, testing, and support.

Apache Spark Stream processing has gained attention because of its wide range of data processing applications in Big Data systems. Some reasons for the increasing level of

interesting are the ease of use, fault tolerance of live data, and suitability of integration with other batch processing systems. Stream data can be ingested from many streaming sources to be processed and used by other systems [143]. Spark Streaming operates in a way that the entire received data stream is divided into batches to be ready for processing by the main Spark engine. The final data -the processed results- will then be in batches. The input data stream can be fed from many different sources (e.g., Kafka, Flume, Twitter, etc.). The stream data can be processed using some advance Spark libraries for machine learning and graph processing algorithms. The final output data from Spark Streaming can be pushed to databases or other systems [143].

Inside the Spark system, Spark streaming receives live stream data as an input to the system. Then, Spark Streaming divides streaming workloads into numeors batches workloads, which are passed as inputs to the Spark core engine for data processing purposes. In Spark Streaming, the high-level basic abstraction is called discretized stream (*DStreams*), and it is a continuous stream of data. Each DStream is either an input data stream that is received from other streaming sources, or it is a result of a processed data stream created from the input streams [143].

Internally, each *DStream* contains a sequence of Spark Resilient Distributed Datasets (RDDs), which are the main Spark core data abstractions. RDDs cannot be changed and can be executed in parallel. In addition, RDDs offer operations, including data transformation and actions, that can be used for Spark Streaming for data analysis. Each RDD in the *DStream* represents data for a specific time interval. Therefore, all operations that are applied to *DStream* will be applied to the RDDs within the same *DStream* [143].

*WordCount* benchmark is a conventional CPU-intensive benchmark and is widely accepted as a standard micro-benchmark for Big Data platforms [144, 145, 146, 147]. *WordCount* benchmark splits each line into multiple words, then aggregates the total count of each word before updating an in-memory map with the word as the key and the frequency of words as the value. The *WordCount* application in Spark Streaming receives a streaming workload from local network to count number of words per messages. The Main Dstream data are divided into many RDDs for certain time intervals. Then, some Spark operation,

such as wide and narrow operations, are perform to count the number of words in each Spark RDD. More details about Spark operations are discussed in Chapter 3.

### 4.2.2 Neural Network Model

The proposed neural network model in Section 3.3.1 is used , which contains three layers, which input, hidden, and output layer. There are more complex neural network architectures that require additional execution time and computing resources such as convolutional neural networks, which is a type of deep neural networks. These neural networks are usually used for image processing, which has high number of input features and output classes. The neural networks model used here has fewer input features (less than 30 features) and output classes than what is used in image processing classification. Therefore, in our case, neural networks with three layers achieve accurate performance classifications with less competition process.

### 4.2.3 Bayesian Optimization

The proposed methodology revolves around using Bayesian Optimization (BO) to find the optimal dataset size and configuration parameters for training the neural networks to generalize the model so it will detect anomalous behaviors in the Spark Streaming system. When utilizing BO, there are two main choices to make: using prior over functions and type of acquisition function [148]. It is essential to choose prior over functions to express assumptions about the optimized function. Due to its simplicity and tractability, we chose the Gaussian process prior for our proposed model. There are different types of acquisition functions, such as *Expected Improvement*[148], *Probability of Improvement*[149], *Lower Confidence Bound*[150], and *Per Second and Plus*. Each type of acquisition function is further discussed in [151].

The acquisition function is used to evaluate a point $x$ based on the posterior distribution function to guide exploration and evaluate the next point [148]. The *Expected Improvement* acquisition function in [152] is used to evaluate the expected performance improvement in the neural networks detection model $f(x)$ and ignore any values that increase the error rate of the model. In other words, $x_{best}$ is the location of the smallest posterior mean (optimal

workload configuration) and $\mu_Q(x_{best})$ is the smallest value of the posterior mean. The expected improvement can be described as follows:

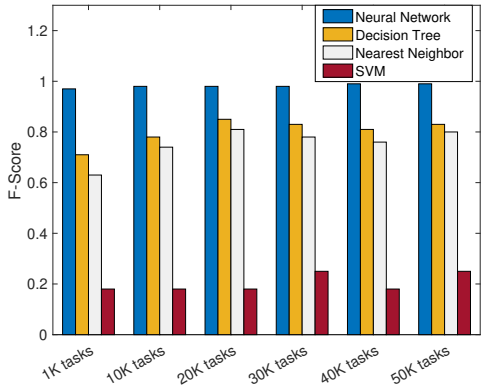
$$EI(x) = E_Q[max(0, \mu_Q(x_{best}) - f(x))] \tag{4.1}$$

$E_Q$ indicates the expectation assumed under the posterior distribution given the evaluations of $f$ at $x_1, x_2, ..., x_n$. The time to assess the objective function may vary depending on the region [152].

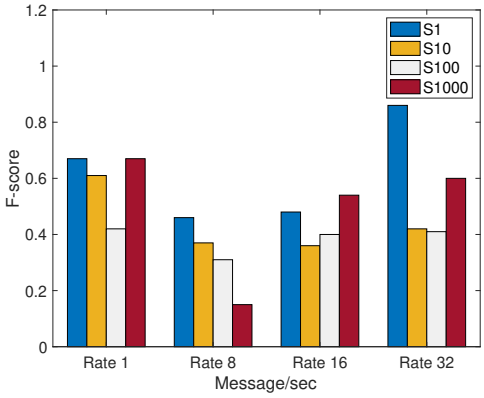
## 4.3 Motivating Example

In this section, we briefly illustrate the problem area and the benefits of Bayesian Optimization for anomaly detection. We developed *Network WordCountExp* benchmark, which is a customized benchmark for stream processing Big Data systems to generate our dataset for training purposes (more detail in Section 4.5.2). The workloads are exponentially generated as messages to be sent to the data stream processing system with some predefined characteristics, such as the rate of (*message/sec*) and the size of messages in lines. The Spark system is monitored at all times and we consider different levels of logging, ranging from Spark streaming jobs measurements to the complete availability of Spark tasks execution logs, which we extensively evaluated in Chapter 3.

A detailed comparison is shown in Figure 4.1(a) to examine the impact of Spark streaming workload size (number of tasks within workload) on the neural networks model and compare it with other three popular algorithms in the domain of anomaly detection, which are nearest neighbor, decision tree and SVM. Six Spark streaming workload sizes with the same configurations are examined for sensitivity analysis, which are 1k, 10k, 20k, 30k, 40k, and 50k tasks. From Figure 4.1(a), it is evident that the neural networks model outperforms all the other algorithms and by achieving 98% for F-score on average for the six workloads. Regarding the execution time, the neural networks, decision tree, nearest neighbor, and SVM took approximately 1 min, 2 min, 5 min, and 21 min respectively. Standard classification methods, such as SVM, do not perform well with imbalanced and




(a) Comparison and sensitivity analysis of Spark workloads size (number of Spark tasks) on neural networks, decision tree, nearest neighbor, and SVM for CPU anomalies detection in Spark streaming workloads.




(b) F-score for anomaly detection using neural network with Rate 1, 8, 16, and 32. Size of message 1, 10 ,100, and 1000

Figure 4.1: Motivation examples for TRACK.

skewed datasets. It is challenging to get the optimal separation hyperplane for an SVM model trained with imbalanced datasets, similar to our case. The imbalance in datasets negatively influences the performance of most classifiers. It is clear that Neural network is the ideal choice for anomaly detection within Spark streaming system.

We examine an anomaly detection that is trained using a new neural networks model with a single Spark streaming workload configuration (Rate 2 and Size 1000 lines) and test it against two unseen streams workload configurations without injecting any anomaly. The first workload has rate 11 and size 1000 ; then the model achieved a 98% for F-score. The second workload has rate 2 and size 5000, which the neural network model achieved a 98%

for F-score. These two experiments show that the neural networks's performance is still robust even though the streaming workload configurations are changed without affecting the performance of the neural network model.

The same network that is trained on a single Spark streaming workload configuration (Rate 2 and Size 1000 lines) is used for some selected possible parameters of streaming workloads configurations for Size (1, 10, 100, and 1000 lines) and Rate (1, 2, 4, 8, 16, and 32), with artificially injected CPU anomalies. The F-score performance of the anomaly detection model dramatically decreased to be between 0.1% and 3%. It is clear that the neural network, in this case, failed to detect the CPU anomalies when the streaming workload configuration was changed. Therefore, there is a need to further train the model on more possible configuration parameters to detect anomalies. This baseline experiment shows that there is an important need for a solution to find the optimal dataset size and configuration parameters of streaming workloads to train the anomaly detection model to generalize the model to detect anomalous behaviors in in-memory Big Data systems.

Figure 4.1(b) shows some design factors and response variables (F-score) for different streaming workload configurations where the proposed neural network is trained with a single combination of configurations parameters (e.g., rate $r$ and size $s$) and test it against other workloads stream configurations, which include rates (1, 8, 16, and 32) and sizes (1, 10, 100, and 1000). As can be seen from Figure 4.1(b), it is not apparent which set of workload configurations that can be used to efficiently train the machine learning model to achieve the highest accuracy whit less time consuming to train the model and detect the anomalous performance in the Spark streaming system. With Network WordCount Spark streaming application (only two parameters), it is also difficult to find the the ideal dataset size to efficiently train the anomaly detection model to comprehensively cover all the seen types of anomalies.

## 4.4 Methodology

In this section, we introduce TRACK and TRACK-Plus, a methodology driven by Bayesian Optimization (BO) and neural networks to train, detect, and classify performance anoma-

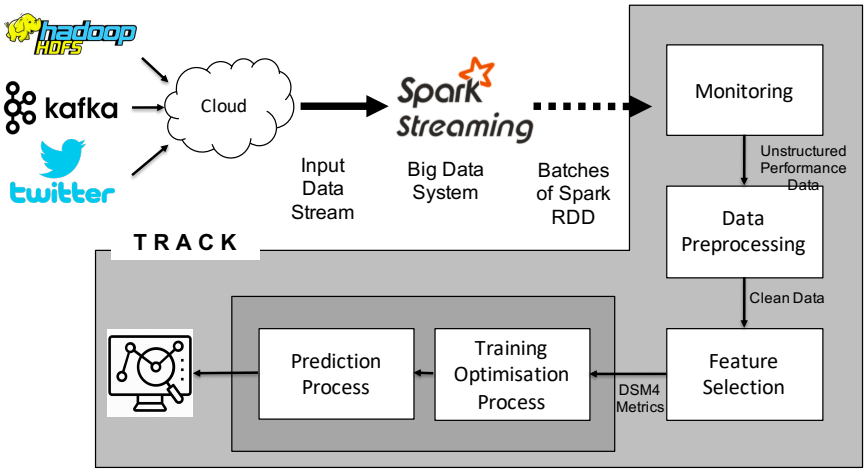



Figure 4.2: TRACK Methodology for anomaly detection)

lies in Apache Spark Streaming systems. Figure 4.2 shows the TRACK processes of anomaly detection for the proposed method. The following subsections give a brief overview of BO and neural networks, discuss the proposed methodology in detail, cover training, testing, and feature selection phases.

### 4.4.1 Machine Learning Model

The neural networks model is used to accurately detect anomalous performance in in-memory Big Data systems such as Apache Spark. The proposed neural network model in Chapter 3 with backpropagation and conjugate gradient are used to train the neural networks to update values of weights and biases in networks. The scaled conjugate is used because it is usually faster than other gradient algorithms [133], especially for time-dependent applications such as real time stream processing.

The neural networks model uses a *Sigmoid* transfer function Equation 4.2 as an activation function, and *Softmax* transfer function is used in the output layer to handle classification problems with multiple classes. For cost function, *cross-entropy* is used to evaluate the performance of neural networks. *Cross-entropy* is used because it has practical advantages over squared-error cost function. It can maintain good classification performance even for

problems with limited data [134].

$$\sigma(x) = \frac{1}{1 + e^{-\boldsymbol{x}}} \tag{4.2}$$

The proposed neural networks contain three types of layers. The performance metrics (*DSM4*) that is described in Chapter 3 is used as input features to input layer. The hidden layer has a number of layers (1, 2, or 3) and number of neurons determined using a *trial and error* method, choosing a number between the sizes of input features $n_i$ and output classes $n_o$ [135]. A hidden layer size between $n_i$ and $n_o$ satisfies our goal in achieving accurate results. In our case, the hidden layer size of 5, 10, 15, and 20 achieve 98%, 99%, 96%, and 96% F-scores, respectively. The Hidden layer with ten neurons achieves the highest F-score with the Spark Streaming workload. The output layer contains a number of neurons equal to the number of target classes (types of anomalies), where each neuron generates boolean values: either 0 for normal behavior or 1 for anomalous behavior.

### 4.4.2 Bayesian Optimization

TRACK and TRACK-Plus use Bayesian Optimization to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve high accuracy in a short period of time. Due to its simplicity and tractability, we chose the Gaussian process prior for our proposed model.

To improve the performance of the proposed methodology, our TRACK method uses a *customized* acquisition function that utilizes time weighting and the *Expected Improvement* for the acquisition function. The *Expected Improvement* acquisition function assesses the current improvement in the objective function and avoids all outputs that may undermine the performance of objective function output. In addition, the acquisition function operates such that during the evaluation of the objective function by the BO model, another Bayesian model (time-weighting) evaluates the time of the objective function [152]. The final acquisition function is as follows:

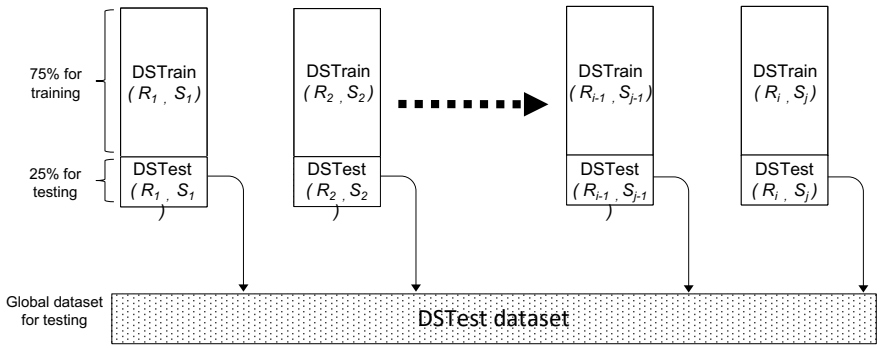



Figure 4.3: Dividing dataset into *DSTrain* and *DSTest* sets for training and testing, respectively.

$$EI_PS(x) = \frac{EI(x)}{\mu_t(x)} \tag{4.3}$$

$$EI_PS(x) = \frac{E_Q[max(0, \mu_Q(x_{best}) - f(x))]}{\mu_t(x)} \tag{4.4}$$

where $\mu_t(x)$ describes the posterior mean of the Gaussian process model timing [152]. A coupled constraint is evaluated only by evaluating the objective function. In our case, the objective function is the performance evaluation of the neural networks model by calculating the F-score. The coupled constraint is that the F-score of the model is not less than a predetermined value (e.g., 90%). The model has several points that are equal to the number of all possible combination parameters of Spark Streaming workload configurations.

### 4.4.3 Model Training and Testing

The Spark Streaming system is randomly injected with anomalies to test the proposed anomaly detection model. For the training process (covers local training, local validation, and local testing), the dataset for every combination of workload configuration parameters (e.g., size $s$ and rate $r$) is divided into two sets: 75% for model training (DSTrain) and 25% for a global testing dataset (DSTest), as shown in Figure 4.3. The local DSTrain set for the model is divided into three subsets: local training (70%), local validation (15%), and local testing (15%). The training subset is used to train the model, whereas the validation subset is used to validate the model and to avoid overfitting and underfitting issues. The

local testing subset is used to test the model against a single combination of configuration parameters for Spark Streaming workloads. The DSTest set is used to globally test the model, which includes 25% from each possible combination of Spark Streaming workload configuration parameters. This subset is used to independently assess the trained model and to generalize the model.

The streaming workload configurations consist of all the possible combinations of configuration parameters of $Rate_n$ and $Size_m$, for a total of $n \times m$ combinations ($n \times m$ *DSTrain*). The training part of the dataset (*DSTrain*) is divided into 10 equal subsets to find the ideal size of the training dataset. For example, the dataset *DSTrain* workload configuration with rate $r_i$ and size $s_j$ is divided into 10 subsets according to the following equation:

$$DSTrain_{r_i,s_j} = DSTrain_{r_i s_j,1} + ... + DSTrain_{r_i s_j,10} \tag{4.5}$$

The total number of all the possible data subsets is $n \times m \times 10$, so it would be challenging and time-consuming to find the optimal combination of configuration parameters and dataset sizes to train the model. More detailed information about TRACK and TRACK-Plus is presented in Algorithm 1 and Algorithm 2. To assess the proposed model, we use a well-known standard classification performance metric, which is F-score (F), defined in Section 3.3.2 alongside the standard metrics of Precision (P) and Recall (R).

**Algorithm 1:** Training and testing methodology for TRACK.

**Input:** Predefined anomaly detection performance $\mathcal{F}$, Workload configuration space $\mathcal{X}$, and system metrics dataset $\mathcal{D}$

**Output:** Optimal trained neural network model $\mathcal{M}$, which is able to identify anomalies within Spark Streaming with the high predefined F-score in the least amount of time.

1 Configuring streaming workload benchmark
2 Workload generation with configuration space $\mathcal{X}$
3 Streaming workload from network $\mathcal{W} \rightarrow$ Spark system
4 System profiling to collect performance dataset
5 Data cleaning and preprocessing $\rightarrow \mathcal{D}$
6 $DSTrain = 75\%$ of $\mathcal{D} \leftarrow$ total training dataset
7 $DSTest = 25\%$ of $\mathcal{D} \leftarrow$ total testing dataset
8 $DSTrain_c$ is empty
9 $F = 0 \leftarrow$ current f-score
10 $Default\_Net\_Config$ : 3 *layers*, 10 units in hidden layer, and *cross-entropy*
11 $i = 0$
12 **while** *( ( $F \leqslant \mathcal{F}$) AND ($i \leqslant size(\mathcal{X})$) )* **do**
13 $\quad \mathcal{X}_i = EI_PS(\mathcal{X}) \leftarrow$ acquisition function
14 $\quad DSTrain_c = DSTrain_c + DSTrain_{X_i}$
15 $\quad \mathcal{M} =$ TrainNN($DSTrain_c$ , $NetConfig$ ) $\leftarrow$ train model on the new dataset configuration
16 $\quad F =$ Max( $Fscore(\mathcal{M}$(DSTest)) , $F$ )
17 $\quad i = i + 1$

**Algorithm 2:** Training and testing methodology for TRACK-Plus.

**Input:** Predefined anomaly detection performance $\mathcal{F}$, Workload configuration space $\mathcal{X}$, and system metrics dataset $\mathcal{D}$

**Output:** Optimal hypertuned trained neural network model $\mathcal{M}$, which is able to generate an agile model to classify anomalies within Spark Streaming with the high predefined F-score in the least amount of time.

1 Configuring streaming workload benchmark
2 Workload generation with configuration space $\mathcal{X}$
3 Neural Networks with configuration space $\mathcal{NN}$
4 Streaming workload from network $\mathcal{W} \rightarrow$ Spark system
5 System profiling to collect performance dataset
6 Data cleaning and preprocessing $\rightarrow \mathcal{D}$
7 $DSTrain = 75\%$ of $\mathcal{D} \leftarrow$ total training dataset
8 $DSTest = 25\%$ of $\mathcal{D} \leftarrow$ total testing dataset
9 $DSTrain_c$ is empty and $F = 0 \leftarrow$ current f-score
10 $Default\_Network\_Configurations$ : $\mathcal{L}$ *layers*, $\mathcal{U}$ units in hidden layer, and $\mathcal{P}$ *Performance function*
11 $i = 0$
12 **while** $( ( F \leqslant \mathcal{F}) \ AND \ (i \leqslant size(\mathcal{X})) )$ **do**
13 $\quad \mathcal{X}_i = EI_PS(\mathcal{X}) \leftarrow$ acquisition function finds next workload configurations
14 $\quad DSTrain_c = DSTrain_c + DSTrain_{X_i} \leftarrow$
15 $\quad j = 0$ and $i = i + 1$
16 $\quad$ **while** $( ( F \leqslant \mathcal{F}) \ AND \ (j \leqslant size(\mathcal{NN})) )$ **do**
17 $\quad\quad NetConfig_j = EI_PS(\mathcal{NN}) \leftarrow$ acquisition function finds next neural networks configuration
18 $\quad\quad \mathcal{M} = \text{TrainNN}(DSTrain_c \ , NetConfig_j \ ) \leftarrow$ train neural network model on the new dataset configuration
19 $\quad\quad F = \text{Max}( \ Fscore(\mathcal{M}(\text{DSTest})) \ , F \ )$
20 $\quad\quad j = j + 1$
21 $\quad \mathcal{M} = \text{TrainNN}(DSTrain_c \ , NetConfig \ ) \leftarrow$ train neural network model on the new dataset and hyperparameters configuration
22 $\quad F = \text{Max}( \ Fscore(\mathcal{M}(\text{DSTest})) \ , F \ )$

### 4.4.4 Feature Selection

The Spark system is monitored at all times and we consider different levels of logging, ranging from Spark jobs measurements to the complete availability of Spark task execution logs, which are used in Chapter 3. These logs provide a reflection of the full details of a Spark system performance. The performance monitoring happens in the background without generating any noticeable overhead in the Spark system.

In this work, we extend the method proposed in Chapter 3, called DSM4, which has been built upon the list of Spark performance metrics presented in Chapter 3. DSM4 examines the internal Apache Spark architecture and the Directed Acyclic Graph (DAG) of the Spark application by relying on information from Apache Spark systems. This information includes Spark executors, shuffle read, shuffle write, memory spill, java garbage collection, tasks, stages, jobs, applications, identifications, and execution timestamps for Spark resilient distributed datasets (RDDs). The collected Spark performance metrics are in time series and manually labeled as either normal or anomalous before they are passed as inputs to the proposed model. The proposed methodology assumes that the collected data is pre-processed to ensure the exclusion of any mislabeled training instances and to validate the datasets before passing them to the BO and neural networks model to improve their quality. For example, we avoid duplicated task measurements and exclude samples if features are missing as a result of the monitoring service level anomalies.

## 4.5 Evaluation

This section evaluates the proposed methodology against a random search (RS) algorithm as a baseline for the same datasets, which are generated from the Apache Spark Streaming system.

### 4.5.1 Experimental Testbed

The experiments are conducted on a Spark Streaming system with 16 core Intel(R) Xeon(R) CPU 2.30 GHz, 32 Gb RAM, Ubuntu 16.04.3, and 2 TB of storage. The Apache

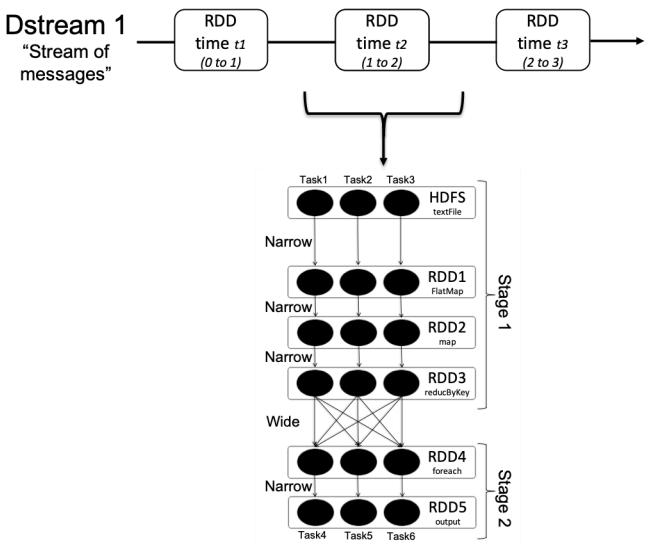



Figure 4.4: Spark Streaming with WordCount example of DStream

Spark is deployed with the Spark Standalone Cluster Manager, 16 executors, and a first-in-first-out scheduler option for deployment. Performance monitoring and data collection are done in the background without causing any noticeable overhead on the system. *Spark History* is used to actively record the performance metrics of internal Spark architecture, such as Spark DAG jobs, stages, and tasks.

### 4.5.2 Workload Generation

To evaluate the accuracy of the proposed anomaly detection methodology, we developed the customized *Network WordCountExp* benchmark for Big Data stream processing systems to generate datasets for training and testing purposes.

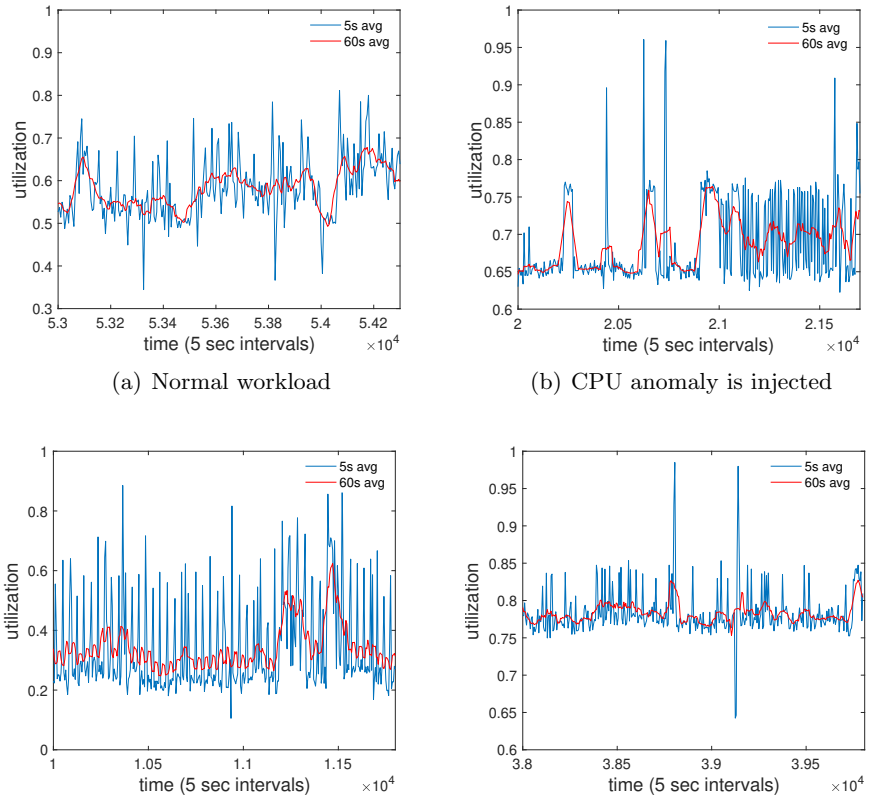
*WordCount* is a conventional CPU-intensive benchmark and is widely accepted as a standard micro-benchmark for Big Data platforms [144, 145, 146, 147, 77]. The *WordCount* benchmark splits each line of text into multiple words, then aggregates the total number of times each word appears throughout and updates an in-memory map with the words as the key and the frequency of the words as the value. Figure 4.4 shows a *WordCount* example of Spark Streaming that receives a streaming workload from a local network to count the number of words per message. The Main *DStream* data is divided into many RDDs for a certain time interval, then some Spark operations, such as wide and narrow

operations, are done to count the number of words in each Spark RDD. More details about Spark operations are discussed in Chapter 3.

The workloads are exponentially generated (with exponential distribution) as messages sent through the system network to the data stream processing system with some predefined characteristics such as the rate of sending messages per second and the size of messages. *WordCount* is used extensively with many different configurations to evaluate and compare the results of the proposed methodology within in-memory Spark Streaming systems. More than 960 experiments are conducted and 230 Gb of data are collected from the Spark Streaming system, which we use to evaluate the proposed work. The dataset covers four types of injected anomalies within Spark Streaming workloads: normal, CPU anomaly, cache thrashing, and context switching. CPU utilization of the Spark system is shown with different types of anomalous performance in Figure 4.5.

### 4.5.3 Anomaly Injection

To inject different types of anomalies, the open-source tool (*stress-ng*) is used to evaluate the proposed methodology with the Spark Streaming system (discussed in Chapter 3). A list of performance anomalies is used to generate CPU stress, cache thrashing stress, and context switching stress as shown in Table 3.4. The CPU stress spawns $n$ workers to run the *sqrt()* function; the cache thrashing stress causes $n$ processes to perform random widespread memory read-and-writes to thrash the CPU cache; and the context switching stress has $n$ processes that forces context switching. The injected anomaly and the used benchmark are configured depending on the objective of the experiment, which will be discussed in Section 4.6.




(a) Normal workload

(b) CPU anomaly is injected

(c) Cache thrashing anomaly is injected

(d) Context switching anomaly is injected

Figure 4.5: CPU utilization for Spark Streaming workload with normal and anomalous performance.

## 4.6 Results

The proposed methodology is evaluated on an isolated Spark Streaming system, discussed in Section 4.5.1. We avoid using a virtual Spark System, which ensures that all performance metrics are accurately measured. The following subsection show results and advantages of our proposed solution.

### 4.6.1 Finding The Ideal Dataset Size To Train the Neural Network Model

Figure 4.6 shows a sensitivity analysis for the size of collected datasets to train the neural network algorithms to learn complex nonlinear relationships among performance metrics

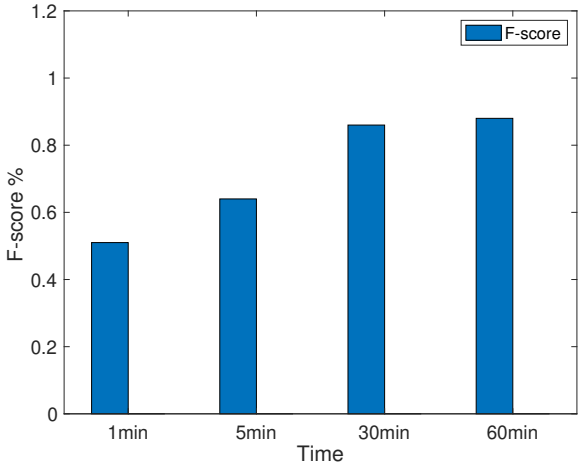



Figure 4.6: The performance of ANNs models that is trained with training dataset that have been collected for 1, 5, 30, and 60 min.

and detect the anomalous performance within Spark systems. It is clear that the size of collected training data significantly impacts the ANN model, while it is challenging to find the optimal size of the training data. The small size of training dataset causes unacceptable F-score, whereas a large dataset may lead to a waste of computing resources.

### 4.6.2 Finding The Ideal Workload Configuration For Model Training

The previous discussion regarding the motivating example (Section 4.3) describes the need to find the ideal single workload configurations set (e.g., rate $r_i$ and size $s_j$) that could be used to train the proposed anomaly detection model to pinpoint the abnormal behavior with the highest possible F-score. This facilitates the use of a single workload configuration to be generalized and used to detect anomalies with the other workload configurations. The Spark Streaming workload has all possible combinations of *rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*, for a total of 16 combinations.

A Bayesian Optimization (BO) and neural networks model (described in Section 4.4.2 and in 4.2.2) are used to address the need for determining the ideal single workload configuration (rate $r_i$ and size $s_j$) with the minimum number of running experiments $n$. To ensure accurate results, the experiments are conducted 50 times, then the average of $n$ is calculated. The results show that the ideal F-score is reached with the minimum num-

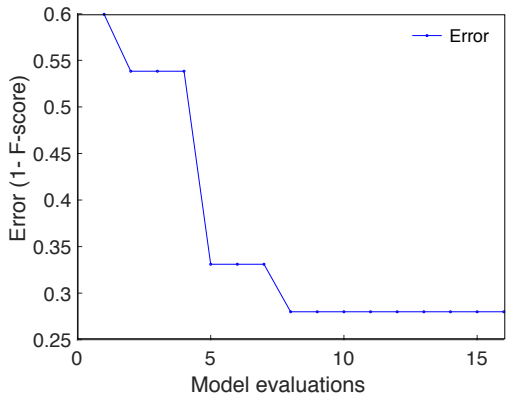



Figure 4.7: Testing results on all possible combinations using Bayesian Optimization and neural networks.

ber of running experiments (*n=8*), which is 50% less than the total number of possible configurations (*n=16*).

Figure 4.7 shows the training process results for each workload configuration, tested on all possible combinations of streaming workload configurations using Bayesian Optimization and neural networks. The performance results of the proposed model when it is individually trained on each workload configuration (rate $r_i$ and size $s_j$) and tested against all possible combinations of streaming workload configurations using BO and neural networks. The estimated objective value is the deviation from the ideal F-score (*error = 1- F-score*). Figure 4.7 illustrates that with the given dataset, the workload configuration ($r = 32$, $s = 1$) can be used to train the anomaly detection model to detect abnormal behavior with all other streaming workload configurations with the highest F-scores equaling 72% after running only 8 of 16 experiments. The next section explores a new approach to optimize the model and obtain a higher F-score using less time in training processes.

### 4.6.3 Bayesian Optimization Model to Train Anomaly Detection Technique

A BO model (discussed in Section 4.4.2) is used to find the optimal size of the training dataset and the streaming workload configurations set to achieve the highest accuracy with the least time spent training the proposed anomaly detection model. The model training and datasets of anomaly detection are comprehensively discussed in Section 4.4.3 and 4.4.4.

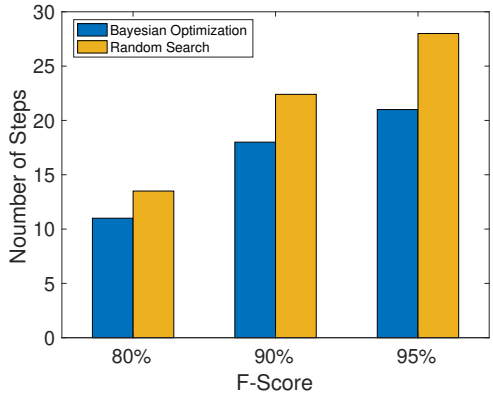



Figure 4.8: Comparison of Bayesian Optimization (BO) and RS to detect CPU anomalies.

Table 4.1: Performance of different types of acquisition functions to reach 95% F-score.

| **Acquisition Functions** | **Number of steps** |
|---|---|
| BO expected-improvement | 64.3 |
| BO expected-improvement-plus | 21 |
| BO expected-improvement-per-second | 30.2 |
| BO lower-confidence-bound | 54.9 |
| BO probability-of-improvement | 43.4 |

Figure 4.8 depicts a comparison of BO and RS to reach a predefined F-score with the fewest training steps from the total 160 steps. The conducted experiments have workloads containing both normal and anomalous CPU behaviors with all possible combinations of workload configurations.

Figure 4.8 shows the average of the 50 experiments where the neural networks model is trained using BO to achieve the predefined F-score. With BO, the trained model reaches a 95% F-score in 21 steps, whereas an RS uses 28 steps (enhanced by 25%). This proves that the proposed model can reduce the computation by 25%. Table 4.1 shows the performance of five types of acquisition functions that are used with BOs. The right column shows the average number of steps for 50 experiments to find the ideal dataset size to train the model.

Two other types of anomalies may disrupt the performance of the Big Data stream processing system. These two types are cache thrashing and context switching. The proposed model can detect both cache thrashing and context switching anomalies with F-scores of 80% and 95%, respectively. Figure 4.9 shows a comparison of BO and RS to reach predefined F-scores (80% for cache stress and 90% for context switching stress) with the

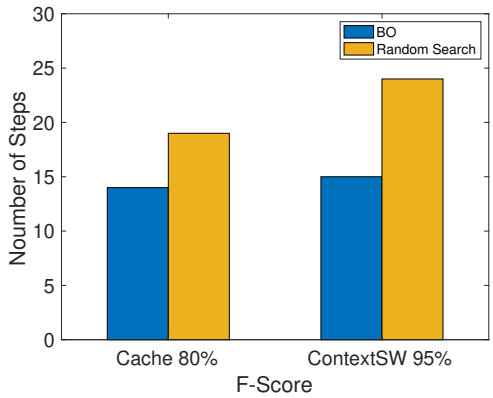



Figure 4.9: Comparison of Bayesian Optimization (BO) and RS to detect cash and context switching anomalies.

minimum number of steps. Workloads have all possible combinations of parameters. The proposed model outperforms RS by more than 25% and can reduce the amount of computations from 160 experiments to 14, as can be seen in Figure 4.9 with cache thrashing anomalies.

### 4.6.4 Sensitivity Analyses of Training Dataset Size

In this subsection, the impact of the training dataset size is examined to prove the robustness of the proposed model. The amount of anomalous Spark tasks decrease by 50% to 75% of the anomalous workload in Section 4.6.3. Table 4.2 shows sensitivity analysis demonstrating the impact of reducing the overall anomalous training dataset size by 50% to 75%. BO is compared against RS to assess when each would reach ideal performance (95% F-score) with the fewest possible steps and the least training data. Workloads contains all possible combinations of rates 1, 8, 16, and 32 *message/sec* and sizes 1, 10, 100, and 1000 *line/message*. It depicts the impact of the Spark workload training set size on the proposed stream anomaly detection model. The BO with neural networks model achieves the highest performance in detecting all three types of performance anomalies in Spark Streaming systems. This proves that the proposed model is robust against changes in the size of the overall input training datasets.

Table 4.2: Sensitivity analysis of reducing the training dataset size

| Algorithm | CPU | Cache | Context Sw |
|---|---|---|---|
| BO | 38 | 19.9 | 16 |
| Random Search | 44 | 24 | 25 |

Table 4.3: Testing the proposed model against new unseen workload configurations with three types of performance anomalies.

| Types of Anomalies | F-score ± Standard deviation |
|---|---|
| CPU | 0.93±0.01 |
| Cache Thrashing | 0.77±0.02 |
| Context Switching | 0.72±0.04 |

### 4.6.5 New Unseen Workload Configurations

This section presents the training of the proposed model with predefined workload configurations (*rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*) and generalizes the model to perform just as accurately with new unseen workload configurations (e.g., $r_i = 20$ and $s_j = 150$). In this case, the workload is more realistic and reflects the workload characteristics of the real stream processing system in the production environment.

For the training phase, the same BO and neural network configurations proposed in Section 4.6.3 are used to train the model on predefined workload configurations (*rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*) to reach a 95% F-score for detecting CPU performance anomalies. For the testing phase, the final model of the training phase is used to detect anomalous behavior but with new unseen workload configurations. The rate could be between 1 to 32 and the size could have ranged from 1 to 1000. The total number of possible configuration combinations is 32000.

Table 4.3 shows the F-score of the proposed model when it is tested against three types of anomalies (i.e., CPU, cache thrashing, and context switching) with new unseen workload configurations (rate 1 to 32 and the size 1 to 1000). Our model can be generalized to cover unseen workload configurations. The proposed anomaly detection model can be trained on 16 workload configurations to be generalized to detect anomalies against 32000 different workload configurations.

Table 4.4: Performance of TRACK for detecting and classifying anomalies based on their root causes.

| Type of workload | Recall | Pres | F-score |
|---|---|---|---|
| Normal | 0.89 | 0.91 | 0.90 |
| CPU | 0.56 | 0.55 | 0.55 |
| Cache Thrashing | 0.97 | 0.95 | 0.96 |
| Context Switching | 0.54 | 0.56 | 0.55 |
| Average | 0.74 | 0.74 | 0.74 |

### 4.6.6 Detecting and Classifying Performance Anomalies

In this section, we show that TRACK not only can detect anomalous performance but also classifies workloads into four types: normal, CPU anomaly, cache anomaly, and context switching anomaly. The anomaly detection using TRACK achieves 74% for detecting and classifying Spark Streaming performance, as seen in Table 4.4. The results depict the ability to classify the root causes. The low F-score in classifying the CPU and context switching is that there are similar performance behaviors between these anomalies, making it challenging to differentiate between them. This issue can be overcome by increasing the number of CPU and context switching datasets to train the ML model comprehensively. However, the proposed model is capable to detect the new unseen CPU and context switching anomalies with 93% and 72% as can be seen in Section 4.6.5. The next section introduces a new optimized version of TRACK called TRACK-Plus to find the ideal neural network configuration to accelerate the search process and improve anomaly classification.

### 4.6.7 TRACK-Plus for Optimizing the Choice of Neural Networks Architecture

The performance of TRACK-Plus is evaluated using the two BO models discussed in Section 4.4.2. The first model BO1 is used to find the ideal dataset training size as described in Section 4.4.3. BO1 optimizes the choices for three Spark Streaming workload configurations, which are the rate of messages per second (1, 8, 16, and 32), message size (1, 10, 100, and 1000), and the size of the training dataset (1 to 10). The total number of possible configurations is $4 \times 4 \times 10$, which comes close to 160 different possible combinations.

Table 4.5: All possible optimized configurations for TRACK-Plus including two BO models.

| BO# | Parameters | Possible Configurations |
|---|---|---|
| BO1 | Message Rate (*message/sec*) | 1, 8, 16, or 32 |
| | Message Size (*line/message*) | 1, 10, 100, or 1000 |
| | Training Dataset Size | 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10 |
| BO2 | Number of Neurons | 5, 10, 15, or 20 |
| | Number of Hidden Layers | 1, 2, or 3 |
| | Performance Function | mae, mse, sae, sse, or crossentropy |

The objective of the second model BO2 is to automate the search to achieve the most efficient architecture of neural networks (with a predefined list of configurations) by optimizing the tuning process of the hyperparameters of the neural networks. In practice, different configurations of hyperparameters can significantly impact the performance of the neural networks. In this chapter, we focus more on hyperparameters related to neural network training and structure, including the number of hidden layers, number of neurons in each layer, and performance functions. Five well-known performance functions have been examined in TRACK-Plus, which are:

- Mean absolute error performance function. $mae = (\frac{1}{n}) \sum_{i=1}^{n} |y_i - \hat{y}_i|$
- Mean squared error performance function. $mse = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
- Sum absolute error performance function. $sae = \sum_{i=1}^{n} |y_i - \hat{y}_i|$
- Sum squared error performance function. $mse = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
- Cross-entropy performance. $crossentropy = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$

The total number of possible configuration combinations for BO2 is $5 \times 3 \times 4 = 60$ different possible configurations. Details of the configuration parameters of the two BO models can be found in Table 4.5.

Even with the limited number of configurations to train the anomaly detection technique, TRACK-Plus offers an efficient solution in finding the ideal training dataset size and the most efficient neural network configurations to accurately detect the anomalous performance within the Spark Streaming system. For example, Table 4.5, with the list of the total number of possible configuration combinations, shows that there are $160 \times 60 = 9600$ possible configurations. It is clear that finding the ideal configurations with which to train

Table 4.6: The ideal configuration for the neural networks.

| Parameters | Configurations | Number of Selection |
|---|---|---|
| Performance Func | mae | 16 |
| | mse | 3 |
| | sae | 21 |
| | sse | 8 |
| | crossentropy | 2 |
| #Neurons/Layer | 5 | 18 |
| | 10 | 14 |
| | 15 | 8 |
| | 20 | 10 |
| #Layers | 1 | 25 |
| | 2 | 9 |
| | 3 | 16 |

the anomaly detection model is more time-consuming and resource intensive when using either the traditional search or the manual configurations.

Table 4.6 shows the ideal configuration for the neural networks in terms of performance function, number of layers, and number of neurons in each layer. The number of selection refers to how many times the specific configuration is found to have the highest F-score against other configurations. The average results of 50 experiments where the TRACK-Plus optimizes the training process of anomaly detection to achieve the predefined F-score, which is 70% (the highest possible F-score for classifying the anomalies that can be reached using the proposed solution). With the given conditions of Spark Streaming workloads, we find that the ideal neural networks configurations are *sae* performance function, five neurons/layer, and one hidden layer.

## 4.7 Conclusion

To develop effective fault-tolerant system performance, it is vital to detect anomalous performance and service-level disruption events within data intensive systems. The growing complexity of Big Data systems makes performance anomaly detection more challenging, especially for critical streaming workload applications in distributed systems environments. Therefore, the performance of an in-memory processing technology like Apache Spark Streaming must be thoroughly investigated to pinpoint the causes of performance

anomalies.

Collecting all the possible performance measurements from Big Data systems to train an anomaly detection model is computationally expensive, especially for critical systems such as online banking, stock trading, and air traffic control systems. Even with the *WordCount* Spark Streaming application (which only has two parameters: $r$ and $s$), it is considered time-consuming and costly to find the ideal dataset size to efficiently train the anomaly detection model so that it will comprehensively cover all seen and unseen anomalies.

This chapter contributes to addressing the challenge of anomalous identification by proposing a new hybrid learning solutions -TRACK and TRACK-Plus- for anomaly detection within in-memory Big Data systems. The anomaly detection and tuning methods are developed using Bayesian Optimization and neural networks to train the model with a limited budget and limited computing resources. As can be seen from the experimental results, the proposed model efficiently finds the optimal training dataset size and configuration parameters to accurately identify different types of performance anomalies in Big Data systems. The proposed model achieves the highest accuracy (95% F-score) in significantly less time (80% less than normal). A validation based on a real dataset for the Apache Spark Streaming system has been provided to demonstrate that the proposed methodology can identify the performance anomalies, the ideal configuration parameters, and the training dataset size with up to 75% fewer experiments. Finally, the proposed solution not only identifies anomalous performance with a high F-score but also classifies anomalies, thereby saving considerable time in training the model. In addition, the proposed model can be easily generalized to cover unforeseen workload configurations.

In terms of future work, it is crucial to comprehensively investigate an anomaly detection and prediction for systems that run both batch and stream processing workloads at the same time. Such systems will have increased complexity and performance fluctuations, which may need more effective anomaly detection solutions. Exploring Deep Learning algorithms may hold opportunities to accurately detect and predict performance anomalies in distributed complex systems.

# Chapter 5

# Interference Prediction for Containerized Batch Jobs

## 5.1 Introduction

The rapid developments of Big Data technologies and data science have increased the demands for productive solutions to obtain desirable performance within data centers. The need for the effective integration between data science technologies and Big Data analysis platforms can be achieved by utilizing containerized batch architectures. Numerous Big Data technologies started to use more effective architectures, such as containerized batch applications, to gain the benefits of modularity, service consolidation, fast prototyping of cloud-based applications, autoscaling, flexibility, and reusability. Therefore, implementing containerized batches as the building blocks of Big Data and cloud architecture can offer certain advantages [153, 154]. Some recent studies have introduced the benefits of combining Big Data technologies (such as Spark and Hadoop) with data science tools (such as Jupyter) by using container architecture [155, 156, 157]. However, interference among colocated containerized batch jobs causes performance unpredictability, which is a major challenge when integrating containerized batch applications, Big Data platforms, and cloud computing adoption, which can come with performance, cost, and revenue implications [154]. Containerized batch workloads have unique patterns of container char-

acteristics, completion time, performance variation, and a relative standard deviation of response time [158]. Predicting the performance behavior and interference among colocated batch jobs are even more critical for effectively predicting interference within a cloud infrastructure for Big Data analytics.

A typical data center leverages over provisioned computing resources for applications to manage a large number of colocated jobs, fluctuating workloads, and peak demands. This results in low utilization, hence impling high maintenance costs for the servers. Applications and systems interference can be challenging to understand and control within production environments for two main reasons, which are inter-parameter dependencies within cloud systems and the dependence of system performance on the nature of correlations [159]. There are several factors that can illustrate the difficulty in detecting and predicting interference within a production environment. These factors include mixes of heterogeneous applications, unpredictable system inputs, unknown optimal system performance, and corporate policy and structure [160]. All these factors cause challenges in detecting system interference and have to be carefully and precisely addressed.

Several types of interference and performance anomalies may arise within data centers and these can negatively impact response times, packet loss, bandwidth, and CPU utilization [161]. In addition, interference among multiple running jobs in the same computing node may result in performance degradation because of resource contention [162]. Interference detection and prediction among colocated batch jobs within a container-based environment are still not comprehensively examined in the literature [163]. Therefore, this chapter focuses on examining the behavior of colocated jobs within cloud and Big Data systems.

Numerous existing interference detection approaches (e.g., SVM at level of containers [164]) within batch workloads and container environments need to be deeply investigated because they may not always be efficiently applied within data centers. Indeed, they suffer from computational complexity or a high rate of false-positive alarms [165, 166]. Lately, AI algorithms (e.g., neural networks) have attracted researchers in the domain of interference prediction and have a noticeable role in detecting precisely the most relevant features from massive datasets using backpropagation [166].

Many existing approaches for detecting system interference revolve around collecting all the possible performance metrics of each running job and testing each job behavior under different resource configurations to predict the interference within system. These approaches are time-consuming, making it difficult for dynamic complex workloads to collect all the performance metrics and test all the combinations of jobs behaviors. In addition, many existing solutions in the literature consider the correction of the impacts of interference after they occur. However, there is an urgent need for a proactive solution that can predict interference in advance to avoid any degradation of system performance, especially within critical systems.

To overcome the issue of interference among colocated batch jobs and proactively avoiding performance degradation within system, this chapter introduces an automated prediction solution that can estimate interference between colocated batch jobs within the same computing environment. An AI-driven model is implemented to predict the interference among colocated batches and containerized jobs. This interference prediction model can alleviate and estimate tasks slowdown arising from the interference among the running workloads. This model assists the system operators in making an accurate decision to optimize jobs placement.

Our interference prediction solution is agnostic of the business logic internal to each job. Instead, it is learned from system performance data by applying artificial neural network method. The target is to establish the completion time prediction of the batch job within cloud and Big Data environments. We assume to have profiling data for individual job $i$ and two $i \times j$ jobs; then we can attempt using neural network model to predict the completion times when more than two jobs run simultaneously in the system (e.g., two, four, six, etc.).

The AI model is initially constructed through offline training datasets that are collected from the batch Big Data platform within the cloud system. The proposed model estimates the job completion time (JCT), which can be used to enhance the job placement within the system and predict any possible risk, allowing administrators to take proactive action before performance degradation arises.

Table 5.1: List of Dacapo jobs

| **Benchmarks** | **description** |
|---|---|
| Batik | Image generation based on Apache Batik. Single threaded. |
| Luindex | Document indexing with Apache Lucene. Mostly single threaded |
| Sunflow | ray tracing. Multithreaded |
| Xalan | Java XML/XSLT processing. Multithreaded. |
| Jython | Java-based scripting (Python-like). Mostly single threaded. |
| Lusearch | Document search with Apache Lucene. Multi-threaded. |

The proposed model is validated against batch workloads that are available in the DaCapo suite [13]. Dacapo is chosen because it is one of most popular Java benchmarks and is ideal for scientific purposes and evaluation. This Dacapo suite is a Java benchmarks cited in over 1100 scientific papers and coauthored by Intel, IBM Research, and leading academic institutions. The following subsection provides details about the Dacapo benchmark and experimental testbed. Dacapo benchmarks are used which have a set of Java open source, diverse, and real-world applications with nontrivial memory loads. The DaCapo benchmarks issuing HTTP or SOAP calls are excluded because we expected these to be more typical of transactional workloads. The remaining benchmarks are shown in Table 5.1.

In summary, the core contributions in this chapter are as follows:

- An AI interference prediction model that is agnostic of the business logic internal to each job. Our approach learns from data by applying artificial neural networks and compare our model with the other three baseline models (queueing-theoretic model, operational analysis, and an empirical method) on historical measurements of job completion time and CPU run-queue size (i.e., the number of active threads in the system).
- The model captures multi-threading, operating system scheduling, sleeping time, and job priorities.

- A validation on 4500 experiments based on the DaCapo benchmarking suite [13] has been carried out, confirming the predictive efficiency and capabilities of the proposed model.

The rest of the chapter is organized as follows: the related work and motivating example are given in Sections 5.2 and 5.3, respectively. The proposed methodology of this work is presented in Section 5.4, followed by a comprehensive evaluation in Section 5.5. The results of the proposed solutions are discussed in Section 5.6 Finally, Section 5.7 provides a discussion and the conclusions.

## 5.2 Literature Review

In this section, we focus on the literature of systems interference within containerized and batch jobs systems. Using a variety of performance metrics and techniques, several works study the interference within systems . These works can be classified into two areas. The first area focuses on learning-based techniques [167, 168], which use machine learning approaches to develop a model that can learn the behavior of normal and abnormal system performance to detect or predict interference. For example, these studies investigate SVMs [161], convolutional neural networks (CNNs) [165], regularized linear regression [169], collaborative filtering classification [170], and decision tree [171]. The second direction utilizes analytical and formal methods to identify interference, such as completion time modeling [172], statistical regression [173], and queueing theory [174].

Figure 5.1 presents a taxonomy for interference detection and prediction methods in the literature. The following subsections review these two directions in the context of performance interference detection and prediction approaches for batch workloads.

### 5.2.1 Learning Based Techniques

Using a decision tree algorithm, Dwyer et al. [171] propose a methodology for modeling performance slow down on multicore systems. Although their model can classify the system interference as high or low, it cannot quantify interference or provide a numerical

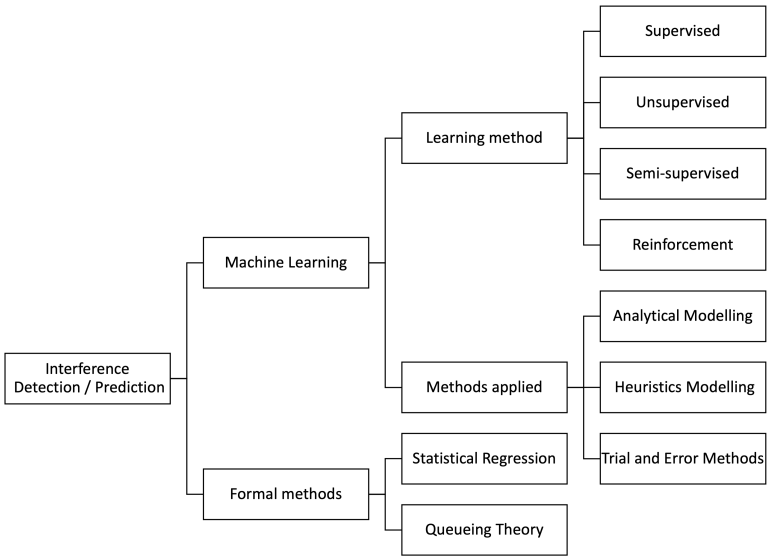



Figure 5.1: Taxonomy of interference detection and prediction

estimation of interference. Bu et al. [175] present a task placement strategy to alleviate and estimate the task slowdown affected by the interference among virtual machines for MapReduce tasks that are scheduling optimization at the application level. The authors [175] use the Gauss-Newton algorithm to find an optimal solution.

Classification techniques are machine learning approaches that have the ability to assign samples to target classes. In performance prediction, the classification approaches are based on the assumption that tasks can be grouped into classes with similar performance behavior. Delimitrou et al. [170] present Quasar, which uses collaborative filtering techniques to quickly classify workloads for an incoming job based on a short test run of the application; the goal here is to determine suitable computing resources to pack workloads with the available resources. The authors assume that the workload can be partitioned into jobs that have similar behaviors. We note that for complex workloads, it is time-consuming to test every possible combination of job behaviors that have to be profiled under different resource configurations to assign the behavior into a particular class. This limitation is not discussed in [170].

Meyer et al. [168] present a supervised interference classifier model based on SVM for classification and K-Means for clustering to improve scheduling for dynamic applications in clouds systems; they show that their model can automatically define interference level

ranges from applications by classifying the instances into segments and dividing the interference monitoring metrics into interference classes. Although this classification approach predicts the class or level of interference, there is an urgent need to specify the amount of interference among colocated services to precisely manage these services within complex systems. This can be done using a regression model, which is not examined in [168].

Kousiouris et al. [176] present a black-box method based on genetically optimized neural networks to investigate the degradation of system performance, comparing their model with a linear regression method. They use a neural network model to correlate the input training dataset with the predicted output data to find the overall dependency of the output from the input. They provide the rationale that they use neural networks with genetic algorithm to dynamically configure the architecture of neural networks.

Ye et al. [177] propose three machine learning regression methods -neural networks, SVM, and linear regression- to predict container performance for container allocation based on resource metrics. They focus on CPU, memory, and I/O to characterize Apache Spark performance while running well-known benchmarks, including KMeans, PageRank, Sort, and logistic regression. Their results show that the neural networks and SVM outperform linear regression. The limitation of the proposed solution is that they consider a specific application (Apache Spark).

Tang et al. [178] implement Fisher, a container performance prediction based on a deep neural network model within a cloud environment; they use long short-term memory (LSTM) to predict container performance and enhance container placement decisions in advance. They train their neural network model on web service and database datasets. The training datasets include 10 input features covering CPU, memory, disk, and network performance metrics. Their results show a promising usage in the time series prediction for container and web applications domain.

### 5.2.2 Formal Methods

Batch workloads, such as the workloads used in MapReduce, are sometime deployed on physical servers to avoid performance overhead in virtualized environments [179, 180,

181]. Sharma et al. [179] utilize spare computing resources by consolidating the batch jobs to reduce the interference (CPU, memory, and I/O) across collocated MapReduce applications. The scheduler they propose has two phases —job classification and dynamic resource management— to improve system utilization. They use statistical regression predictive models for understanding the runtime resource interference. The issue with the proposed model is that the correction only works after the interference occurs within system.

Kambadur et al. [160] introduce a generic measurement solution to analyze system interference for large-scale applications. They use statistical estimators and performance indicators that aggregate the normal system performance measurements to compare them with future observed samples. The instructions per cycle (IPC) is used as an estimator to capture cache and memory contention impacts. Their approach of using IPC counters is not always appropriate because architectural enhancements may cause the IPC to be improved; even the application performance is decreased for multithreaded applications. Therefore, the proposed solution is specific for servers that are identical in every respect (CPU type, clock speed, memory, OS, etc.).

## 5.3 Motivating Example

This section aims at illustrating the motivation and need for an accurate interference prediction that can be used within a complex production environment. We demonstrate the importance of predicting the impact of batch job interference when having colocated batch jobs to efficiently utilize system resources. We use testbed (shown later in Section 5.5.2) to run experiments for five minutes. Figure 5.2 shows the CPU utilization of two types ($Sunflow$ and $Xalan$) of batch jobs that run DaCapo benchmarks when they run alone, and in the case where two, four, or six replicas of the same service run colocated.

In these experiments, the benchmarks are cyclically run. It is clear that predicting resource utilization does not add to the actual measured utilization of job $i$. Adding the amount of CPU utilization of the same multiple jobs of $i$ for $n$ times is not equal to the real utilization of $i$ batch jobs that are run simultaneously. For example, running a single $Sunflow$ has

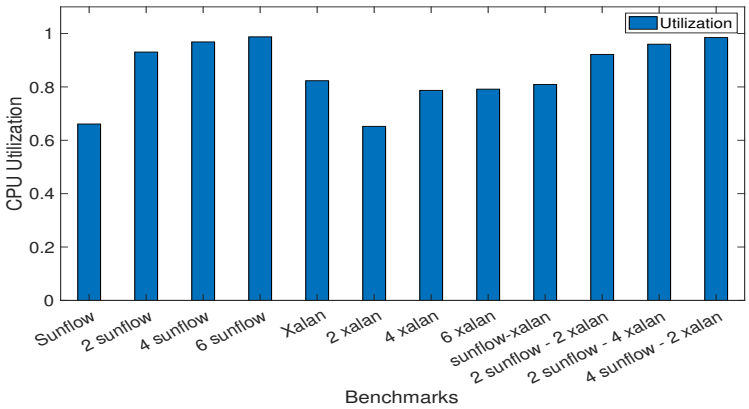



Figure 5.2: Running 1, 2, 4, 6 Dacapo jobs CPU utilization

CPU utilization of 65%, while two jobs $Sunflow$ reach 90%, four remain at 95%, and six reach 100% CPU utilization. A similar nonadditive trend is also observed for $Xalan$ job. Various factors can be the reason for this, from underutilization of the CPU because of access to other resources (e.g., bottlenecks) to dynamic voltage scaling, which changes the CPU speed as more jobs are colocated. Overall, this experiment indicates that additive utilization formulas, such as those common in queueing theory and operational analysis [182] are not necessarily representative for complex workload types.

From this example, we can conclude that predicting system performance when colocated batch jobs is challenging and needs an efficient predicting model that can proactively avoid performance degradation. To overcome this challenge, a neural network algorithm can be used for the performance prediction of colocated batch jobs; however, it is not a straightforward task and needs to be precisely configured and trained. A neural network algorithm with backpropagation and a conjugate gradient are used for training, here by using datasets that are collected from workloads, which run $i$ and $i \times j$ Dacapo batch jobs (discussed later in Section 5.5.1) are used to detect interference when more than two unseen batch jobs at the same time (e.g., $i \times j \times k \times l$). Figure 5.3 shows the R value is a measure of how well the regression predictions approximate the real data points. The R value for the prediction result of the testing phase for neural network model, which is 26%. This low performance prediction of the throughput illustrates that applying the neural network algorithm needs to be carefully tuned with precise preprocessing of the

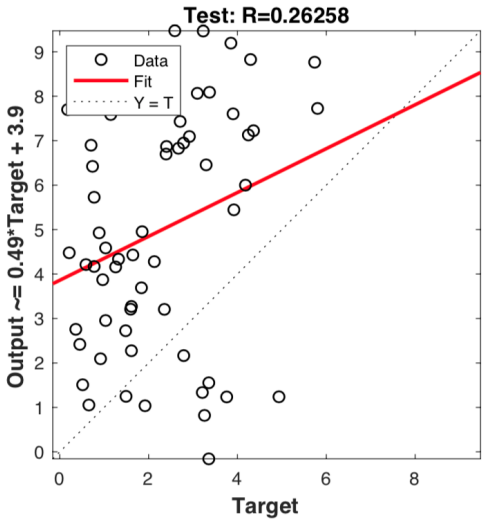



Figure 5.3: Regression for more than two batch jobs ( for example $i \times j \times k \times l$ jobs).

input features to cover unseen performance behavior within system.

## 5.4 Methodology

The goal of this chapter is to establish a low completion time prediction error with a small computation overhead by using a small set of profiling datasets to generalize it to detect the unseen interference of colocated running batch jobs and containers. Profiling data for a single job $i$ and two jobs $i \times j$ are monitored and collected in an attempt to predict the completion times when more than two batch jobs $i \times j \times \ldots \times k$ run at the same time within the same system. The proposed prediction method works without any need for measurements from the executions with more than two jobs (e.g., $i \times j \times \cdots \times k$); everything is predicted using only the profiling data of $i$ and $i \times j$ jobs and the neural networks prediction model.

A machine learning algorithm using a neural network-driven model with backpropagation and a conjugate gradient are used to train the prediction model. In this work, the feature selection process covers many performance metrics (such as hyperthreading, CPU nice, RunQ, throughput, type of job, etc.) as the input features to the model for training purposes. The other models, such as *Queueing* model, are an approximation of the real system because there are many system features other than those things not captured (e.g., *hyper-*

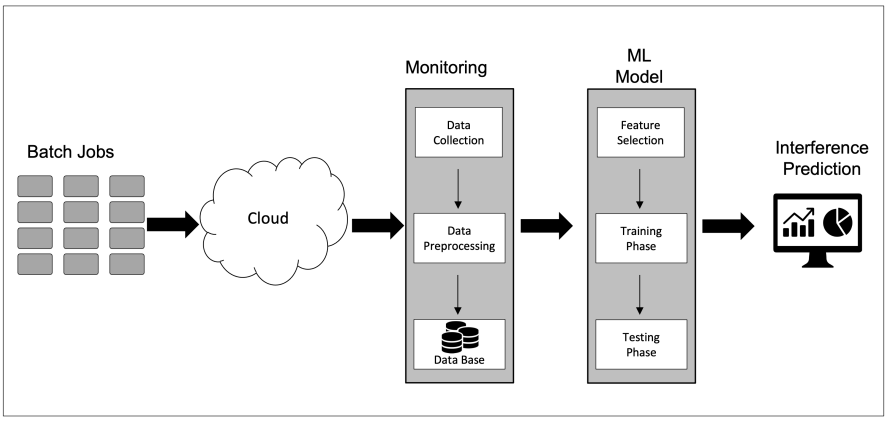



Figure 5.4: The proposed methodology for interference prediction.

*threading, memory bandwidth, etc.*). Our hypothesis is that our proposed neural network model can handle such complexity. Figure 5.4 shows the high-level methodological diagram of the proposed interference prediction. The following subsections give more details about our proposed model and other models, which are used for comparison purposes.

### 5.4.1 AI Regression model

Our proposed neural network models are trained using the Levenberg-Marquardt back-propagation algorithm (if there is no memory limitation) or scaled conjugate. The training process of our model continues until the value of the validation error does not decrease for $n$ iterations (e.g., six). One to three hidden layers are used with 10 neurons, here depending on the size of input dataset. The number of neurons can be increased if the network accuracy is not as expected. The total input training dataset is divided into three subsets, which are 70% for the training phase, 15% for the validation phase to generalize the model and to stop training before overfitting, and 15% for the testing phase to make sure that the model is generalized correctly or not.

### 5.4.2 Other Models

The proposed model is compared against three other models for evaluation purposes. The other models include Queueing Network (QN) model, Operational Analysis (OA) model, and No model. The following subsections provide more details for each of these models:

### Queueing Network (QN) model

The model aims at representing the state and CPU usage of individual threads of the batch jobs running in the system. We first introduce our queueing-based completion time model. The model aims at representing the state and CPU usage of individual threads of the batch jobs running in the system. We call a thread *active* if it is either *running* or *ready* for execution at a CPU core, and *blocked* otherwise. In order to deliver predictions in the absence of historical data, we use state-based modelling to capture the essential features of *weighted fair queueing*, which is a common form of CPU scheduling used in modern operating systems (*e.g.,* in the default Linux scheduler CFS). Our queueing network model abstracts CPU scheduling in an operating system similar to Linux.

Taking Linux as a reference example, a fair queueing scheduler is available at each individual CPU core. Here the scheduler manages a set of active threads, each with an associated weight $w_j$. The weights $w_j$ are assigned by the kernel to the threads based on their priority, in such a way that high priority jobs receive a larger share of the CPU core, but without starving low priority jobs. The scheduler ensures that time slices are allocated *proportionally* to the weights $w_j$. The challenge is therefore to devise a general method to represent this scheduling policy in the completion time prediction.

### Resource contention model

In the proposed model, we internally track the mix of threads present at each CPU core and their state. The goal is to continuously capture the time slicing behavior of the fair queueing scheduler at each core. This can be done using the queueing network model shown in Figure 5.5, which may be seen as a high-level abstraction of a continuous-time Markov chain. In this Markov chain, the state is defined in terms of tuples $(n_{11}, ..., n_{MJ}, b_1, ..., b_J)$ where $n_{mj}$ is the number of threads spawned by job $j = 1, ..., J$ that are active at CPU core $m = 1, ..., M$, and $b_j$ is the number of blocked threads of job $j$.

The key features of the model are as follows. The model describes moving units that correspond to threads. Specifically, a batch job $j$ is represented by a set of $N_J^{max}$ threads that cyclically visit $M$ queues, where $N_J^{max}$ represent the maximum threading level of the

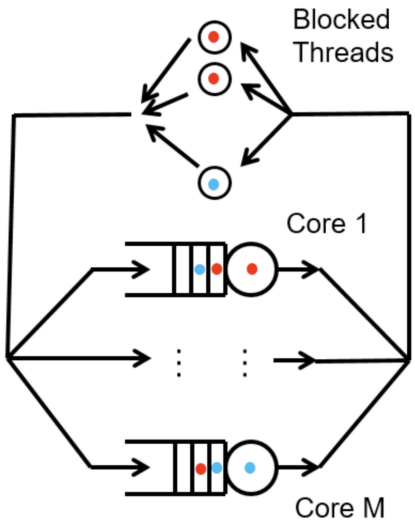



Figure 5.5: Completion time model. Threads of the same color belong to the same batch job.

batch job, and each queue represents a logical core within the system. The model also includes a delay node, shown at the top of the figure, in which threads reside while in *blocked* state. This can happen when the CPU saturated with batch jobs.

Under context switching, the blocked threads can restart execution at any core, which is here modelled by the fact that threads are dispatched by the scheduler randomly, and with identical probabilities, to the cores. A job completes once all of its threads visit $V_j$ times the cores[1]. Once at the cores, threads are scheduled according to the weighted fair queueing policy. The processing time at each core is a random variable for which we assume to know only the mean $T_j$. The mean time spent in the blocking state by a thread is instead denoted by $Z_j$.

### Input/output parameters

In order to instantiate the model, the following profiling data was collected for each job by running it alone on the server:

- $N_j^{max}/N_j^0$: maximum/average number of active threads spawned by job $j$. During profiling the ~~jj~~ number of active threads was collected every 5 seconds using sar

[1]Due to insensitivity properties of the considered queues, if $V_j$ is unknown it is possible to assume $V_j = 1$, provided that $T_j$ is replaced by the *total* amount of CPU time used by the threads across the $V_j$ visits.

-q (field: runq-sz; in our tests Linux included in this number the threads that are currently executing on the cores).

- ${C_j}^0$: the average completion time of job $j$ (in seconds) when running alone in the system.
- $X_j^0$: the average completion rate of the job (= number of times the job completes / total duration of the experiment in seconds), when running alone in the system.

Applying Little's law [183], the above data provides the mean blocking time of a thread:

$$Z_j = \frac{N_j^{max} - N_j^0}{N_j^{max} N_j^0} \tag{5.1}$$

This relation may be justified as follows: $N_j^{max} \times X_j^0$ approximates the average rate at which threads become blocked, $N_j^{max} - X_j^0$ is the average number of threads in blocked state, thus their ratio by Little's law provides the average time a thread spends in blocked state. Furthermore, we estimate $T_j$ from profiling data using a *maximum-likelihood estimation* approach that yields the following formula in [184]:

$$T_j = \frac{N_j^0}{(N_j^{max} - N_j^0)} \left(\frac{Z_i}{M + N_j^0 (N_j^{max} - 1)/N_j^{max}}\right) \tag{5.2}$$

In addition to the above parameters, for each job j, the model will also require:

- $M$ : the number of logical cores available on the server.
- $w_j$: the fair queueing weight of the threads of job $j$. In Linux the kernel assigns a weight $w_j = 1024 \times (1.25)^{-nice(j)}$, where *nice(j)* is the job's priority class (also called nice value).

Using the input parameters above, the model will output the predicted completion times $C_j$ for each job $j$, taking into account the CPU contention from all the other running batch jobs. Note that our model does not ask to supply the statistical distributions of processing times and blocking times. This is theoretically justified by insensitivity properties of the queues that we are considering, which depend weakly (and in some cases do not depend

at all) on moments other than the mean.

For the queueing model to remain valid, two main modelling assumptions should hold:

1. **A1:** The maximum number of threads spawned by job $j$ during profiling, *i.e.* $N_j^{max}$, will not significantly change in the presence of CPU contention by other jobs.
2. **A2**: The average time $Z_j$ a thread is in *blocked* state, as determined during profiling, does not significantly change when other jobs run on the system.

We expect A1 to be valid most times, since $N_j^{max}$ typically depends on the inner logic of the job, not on the system load. We validated this assumption in our experiments based on the DaCapo batch jobs. For CPU-bound batch jobs, we also expect A2 to hold. However A2 could be violated if jobs place a high utilization on I/O channels, storage, or network, in such a way that queueing at these resources becomes the dominant factor for completion time. Indeed, our model describes CPU contention, thus it is not designed to predict contention at non-CPU resources. This is considered as a limitation of queueing based model.

**Operational Analysis (OA) model**

While our model is rather simple to instantiate, requiring mainly measurements of the CPU runqueue size, it may be possible to have the situation where only CPU utilization and completion time data are available. We therefore consider another simplified predictor model for this case, based on operational analysis laws [182], which describe the mutual relationships between basic system measurements of completion times and rates. We illustrate the method in the case of two batch jobs, but it is applicable to an arbitrary number of batch jobs.

We consider two batch jobs $i$ and $j$ having completion times $C_i^0$ and $C_j^0$ when they run alone in the system. Assume that they require to complete an equivalent of $m_i$ and $m_j$ CPU cores at full utilization, out of the $M$ available cores. We wish to predict their completion time $C_i$ and $C_j$ when two jobs run together. We have two cases:

- If $m_i + m_j < M$, there will be no CPU oversubscription and the method will simply

return the completion time values collected during profiling, i.e., $C_i = C_i^0$ and $C_j = C_j^0$

- Otherwise, we assign to each job a fraction of the CPU proportional to the average number of cores it requires, i.e., job $i$ receives a share $s_i$ equal to:

$$s_i = M\left(\frac{m_i}{m_i + m_j}\right) \tag{5.3}$$

where $S$ is the sum of any known sleep time that the job will incur (*e.g.,* due to the precedence constraints in the job stream). Since the throughput $X_i^0$ was collected using an equivalent of $m_i$ cores, this means that running together the two benchmarks we expect throughput:

$$X_i = X_i^\circ \frac{s_i}{m_i} \tag{5.4}$$

We linearly interpolate $X_i^0$ based on the number of available cores that is now available to job $i$. We can readily determine the job completion time as $C_i = 1/X_i$.

To obtain similar predictions in the case where we know that a job will sleep for $S$ seconds before becoming active, for example due to blocking waiting for data from another job. In this case, the above predictions can be refined by adding $S$ to the value of $C_i$ obtained with the formulas above. The above predictor is effective with simple workloads, but falls short for example when jobs have different priorities. In such cases, it is preferable to adopt the queueing theory model, since it is designed explicitly to represent weighted fair queueing.

**No model**

No model means that we assume the same completion time value obtained during profiling, when the job runs alone in the system. In other words, we neither make a prediction nor assume a resource model. This is a useful baseline as it shows what we gain from defining a resource model as in neural networks, OA and QN.

Table 5.2: The representation of Job names as a part of input features.

| Exp | Job 1 | Job 2 | Job 3 | Job 4 | Job 5 | Job 6 |
|---|---|---|---|---|---|---|
| Exp 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Exp 2 | 0 | 0 | 2 | 0 | 0 | 0 |
| Exp 3 | 0 | 0 | 1 | 0 | 1 | 0 |
| Exp 4 | 1 | 0 | 0 | 1 | 0 | 0 |

### 5.4.3 Feature Selection and Model Training Options

To instantiate the model, the below profiling data were collected for each job $i$ and two jobs $i \times j$ by running them on our cluster. Some of input features may include the following:

- types of jobs, which are represented in Table 5.2. The table shows the number and types of running jobs.
- noht: hyperthreading
- nice: CPU nice. CPU scheduling priority, in which higher vales mean a lower priority and lower values means higher priority. The default priority has a nice value of 0.
- sleep time: the time in between successive runs of a benchmark within the same experiment.
- runQ: the number of active threads in the system. During profiling, the number of active threads is collected every five seconds using $sar - q$ (field: $runq - sz$; in our tests Linux included in this number the threads that are currently executing on the cores).
- tput: throughput

Using the input features above, the model predicts the average completion times $Cj$ (the average completion time of job $j$ (in seconds) when running alone in the system) for all running jobs, taking into account the CPU contention from all the other running batch jobs. More input features can be collected and used to cover the performance pattern of the benchmark behaviors. In our case, the above performance metrics are more than enough to accurately predict the interference among colocated jobs.

The model is fed with $SAR$ performance metrics collected for Docker containers that run DaCapo batch jobs. The user may vary the system configurations on which the job is

run, which may be used for the interference-aware job placement. Currently, the following configuration factors can be varied with the tool: containerized batch job type (name of job), whether hyperthreading is activated or not, CPU nice, and sleep time between the runs. The training datasets $D_{Train}$ of single and two jobs are used in the training phase. The testing datasets $D_{Test}$ are sets of more than two running jobs that are used for generalization and prediction purposes.

More detailed information about our methodology is presented in Algorithm 3. To assess the proposed model, we use a well-known standard regression performance metric— the mean absolute percentage error (MAPE)— to measure the prediction accuracy.The proposed methodology aims to train the neural networks mode, which is able to predict the throughput of colocated containerized batch jobs with a low MAPE. The methodology covers different phases, including performance metrics collection, datasets cleaning, building the neural networks model, training phase, and testing phase.

**Algorithm 3:** Training and testing methodology for predicting interference of colocated containerized batch jobs.

**Input:** Workload configuration space $\mathcal{X}$, target MAPE, and system metrics dataset $\mathcal{D}$

**Output:** Optimal trained neural network model $\mathcal{M}$, which is able to predict throughput of colocated containerized batch jobs with the low $MAPE$.

1 Configuring benchmark
2 Workload generation of single and two jobs with SAR metrics and configuration space $X_{train}$
3 Workload generation of more than two jobs with SAR metrics and configuration space $X_{test}$
4 System profiling to collect performance dataset $D$
5 Data cleansing and prepossessing of $D \rightarrow D_{Train}$ and $D_{Test}$
6 $DSTrain = 75\%$ of $D_{Train} \leftarrow$ total training dataset
7 $DSValidation = 15\%$ of $D_{Train} \leftarrow$ total validation dataset
8 $DSTest = 15\%$ of $D_{Train} \leftarrow$ total testing dataset
9 $F = 0 \leftarrow$ MAPE
10 $Default\_Net\_Config$ : 3 *layers*, 10 units in hidden layer, and *cross-entropy*
11 $Array\_F$ : Array that saves all the prediction results of models. $i = 0$
12 $\mathcal{N} = 50$
13 **while** $( ( F \leqslant MAPE)$ *AND* $(i \leqslant \mathcal{N}) )$ **do**
14 $\quad \mathcal{M} = TrainNeuralNetworks(DSTrain, DSValidation, DSTest, Default\_Net\_Config)$
15 $\quad \mathcal{F} = Test\_NeuralNetworks\ (\mathcal{M}, D_{Test}, Default\_Net\_Config) \leftarrow$ test model on the new dataset $D_{Test}$
16 $\quad Array\_F(i, 1) = (\mathcal{F}, \mathcal{M})$
17 $\quad i = i + 1$
18 $Best\_MAPE\_NN = min(Array\_F) \leftarrow$ Most accurate network
19 Predicting interference among containerized batch jobs = $Best\_MAPE\_NN(D_{Test})$

Table 5.3: Throughput of all DaCapo jobs combinations

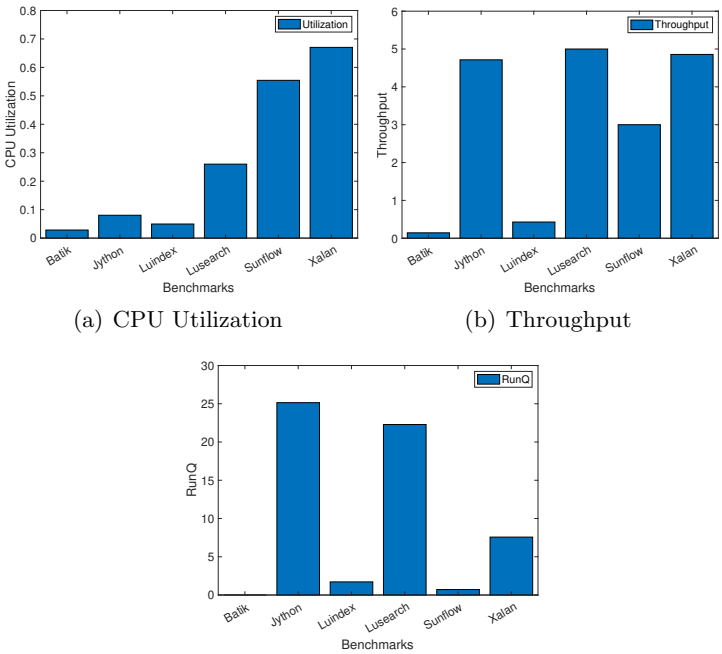
| **Benchmarks** | **Average CPU Utilization** |
|---|---|
| Batik | 2.83% |
| Luindex | 4.95% |
| Sunflow | 55.45% |
| Xalan | 67.03% |
| Jython | 8.03% |
| Lusearch | 25.99% |

## 5.5 Evaluation

### 5.5.1 DaCapo benchmark

Over 4500 experiments are carried out using five different multicore servers running Ubuntu Linux. A multithreaded benchmark typically spawns two to eight threads. Table 5.3 depicts the CPU utilization when a single DaCapo job is run alone in the system. The Batik benchmark has the lowest CPU utilization *2.83%*, whereas the Xalan benchmark has the highest CPU utilization (*67.03%*). The bar chart in Figure 5.6 presents the CPU utilization, throughput, and RunQ performance metrics used to train the neural network models. *Jython, Lusearch,* and *Xalan* benchmarks have the highest throughput, as shown in Figure 5.6(b). Here, ~~*RunQ,*~~ *Jython, and Lusearch* have more active threads compered with the other benchmarks. Figure 5.7 and Figure 5.8 show CPU utilization when running each containerized DaCapo batch job, which reflect the different performance behaviors that make the interference detection more challenging.

When running $i \times j$ benchmarks together, CPU utilization is typically around 70% to 90% on machines with 8 to 16 logical cores. When $i \times j \times \ldots \times k$ benchmarks are run, the machines are always saturated near 100% utilization. Table 5.4 and Figure 5.8 show the throughput and RunQ metrics when colocated jobs are run at the same time, which reflect the variation of system behavior. As discussed in the section of motivating example (Section 5.3), adding the amount of CPU utilization of the same multiple job of $i$ for $n$ times is not equal to the real utilization of $i$ jobs run simultaneously. We have also varied the benchmark mix, the hyperthreading setup, and job priorities and randomized the sleep time in between successive runs of a benchmark within the same experiment. The default




(a) CPU Utilization (b) Throughput

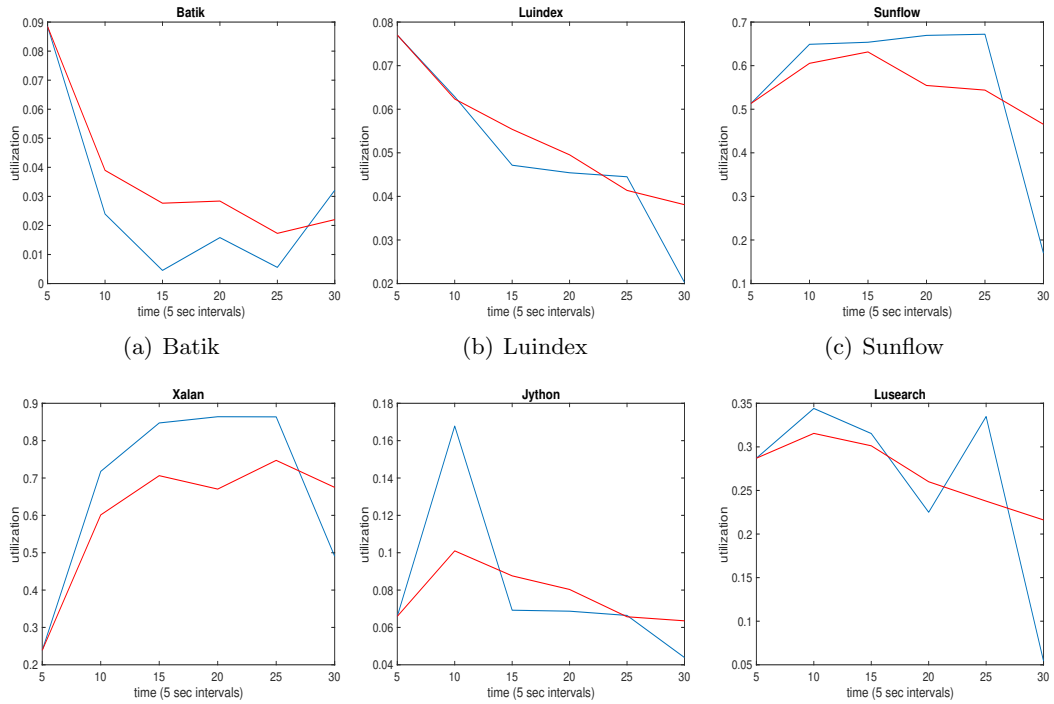
(c) RunQ

Figure 5.6: DaCapo benchmarks performance.

Table 5.4: Throughput and RunQ of all DaCapo jobs combinations.

| **Benchmarks** | **Throughput** | **RunQ** | **Benchmarks** | **Throughput** | **RunQ** |
|---|---|---|---|---|---|
| batik | 0.1429 | 0 | luindex-lusearch | 7.5714 | 8.1429 |
| jython | 0.4286 | 1.7143 | luindex-sunflow | 6.5714 | 23.1429 |
| luindex | 3.0000 | 0.7143 | luindex-xalan | 6.2857 | 17.8571 |
| sunflow | 4.7143 | 25.1429 | lusearch-jython | 4.4286 | 6.1429 |
| xalan | 5.0000 | 22.2857 | lusearch-luindex | 7.5714 | 8.8571 |
| batik-batik | 0.2857 | 0 | lusearch-lusearch | 7.1429 | 15.7143 |
| batik-jython | 0.5714 | 3.8571 | lusearch-sunflow | 6.5714 | 19.1429 |
| batik-luindex | 3.5714 | 0.8571 | lusearch-xalan | 6.2857 | 13.8571 |
| batik-lusearch | 4.7143 | 4.2857 | sunflow-batik | 4.8571 | 22.2857 |
| batik-sunflow | 4.8571 | 14.1429 | sunflow-jython | 4.5714 | 15.2857 |
| jython-batik | 0.5714 | 1.1429 | sunflow-lusearch | 6.2857 | 25.8571 |
| jython-jython | 0.5714 | 5.2857 | sunflow-sunflow | 6.1429 | 34.2857 |
| jython-luindex | 3.0000 | 4.1429 | sunflow-xalan | 6.0000 | 27.5714 |
| jython-lusearch | 4.5714 | 5.2857 | xalan-batik | 5.2857 | 18.2857 |
| jython-sunflow | 4.5714 | 22.1429 | xalan-jython | 4.8571 | 20.4286 |
| jython-xalan | 5.0000 | 14.4286 | xalan-luindex | 7.7143 | 20.7143 |

setup was to run experiments without hyperthreading and no sleep time. Each experiment was stopped after five minutes because the performance behavior stabilized fairly quickly.




(a) Batik (b) Luindex (c) Sunflow

(d) Xalan (e) Jython (f) Lusearch

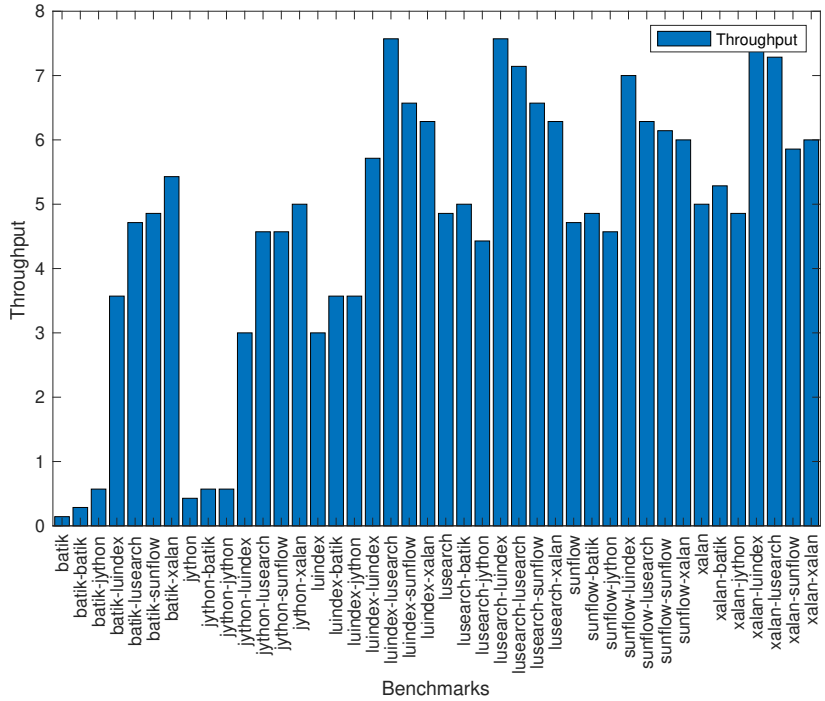
Figure 5.7: DaCapo jobs types and CPUs performance. The blue line is the average CPU utilization of 5 sec, and red line is the Average CPU utilization for 60 sec.

### 5.5.2 Experimental Testbed

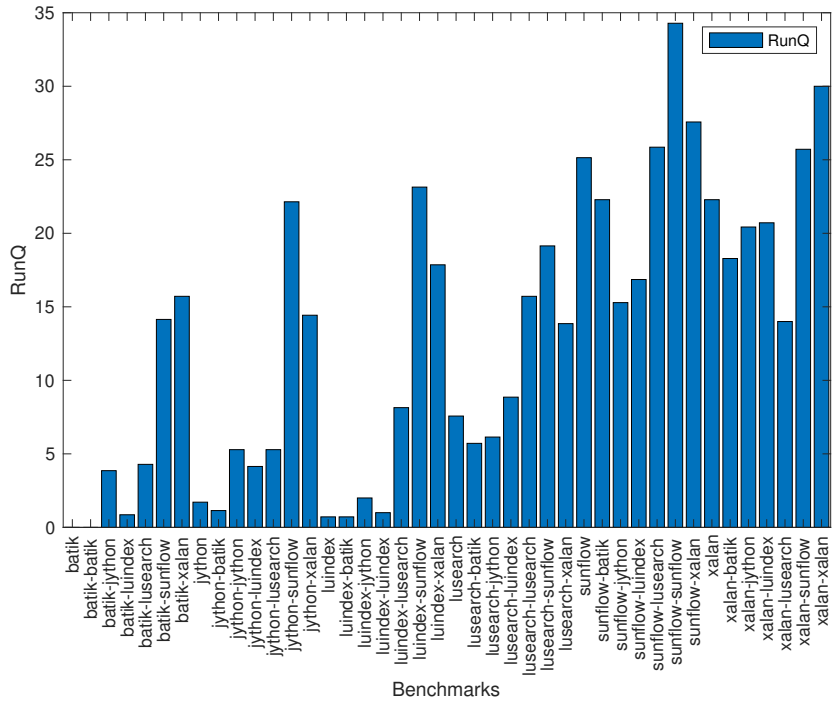
The proposed methodology is evaluated with Docker environment on an isolated Linux cluster that contains three physical servers: S01, S02, and S03. We avoid using a virtual environment to make sure that all the performance metrics are accurately measured. The specifications for these servers are as follows:

1. Node S01: 8 vcores Intel(R) Xeon(R) CPU 3.70GHz, 64 GB RAM, Ubuntu 16.04.3, and 246 GB Storage.
2. Node S02: 8 vcores x Intel(R) Xeon(R) CPU 3.70GHz, 64 GB RAM, Ubuntu 16.04.3, and 1.1 TB Storage (includes 881 GB SSD).
3. Node S03: 32 vcores x Intel(R) Xeon(R) CPU 2.10GHz, 16 GB RAM, Ubuntu 16.04.3, and 1 TB Storage.

Monitoring data collection took place in the background, with no significant overhead on

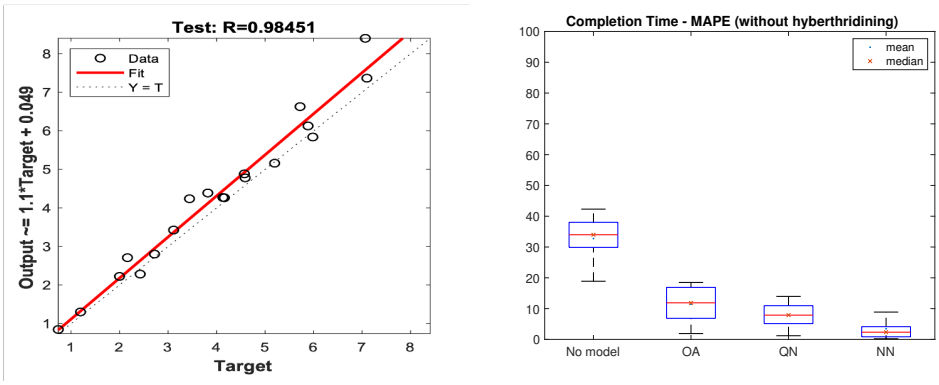



(a) Throughput




(b) RunQ

Figure 5.8: DaCapo jobs and performance metrics.




(a) Correlation coefficient (R-value) between the output and actual target classes.

(b) Mean absolute percentage error (MAPE).

Figure 5.9: Test of the proposed model against $i \times j$ jobs

the system. All machines use *sar* (System Activity Reporter) and *Sysstat* to collect CPU, memory, I/O, and network metrics.

## 5.6 Results

The following subsections provide a discussion, sensitivity analysis, and question that are answered to validate the proposed solution.

### 5.6.1 Can the models capture CPU contention for two jobs?

The testing results of the proposed model against two jobs are shown in Figure 5.9. The boxplot diagram in Figure 5.9(b) indicates the mean (blue) and median (red) of the prediction error for data collected across all possible of job combinations of the benchmarks (e.g., xalan-xalan, xalan-batik, jython-luindex, ..., etc). The blue box is the inter-quantile range (25th-75th percentile). The R-value (Figure 5.9(a)) and MAPE (Figure 5.9(b)) for predicting the completion time of $i \times j$ jobs are 98% and 5%, respectively. As it can be seen in Figure 5.9(b), our model noticeably outperforms the other three models. The goal of next subsection is to generalize the solution to cover more than $i \times j$ jobs.

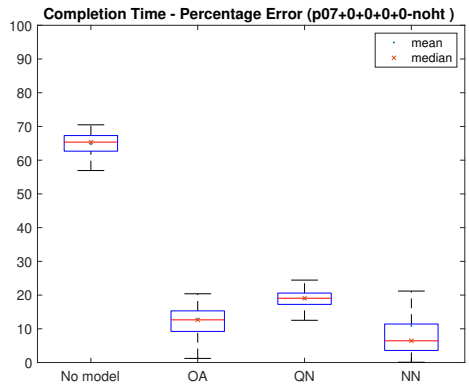
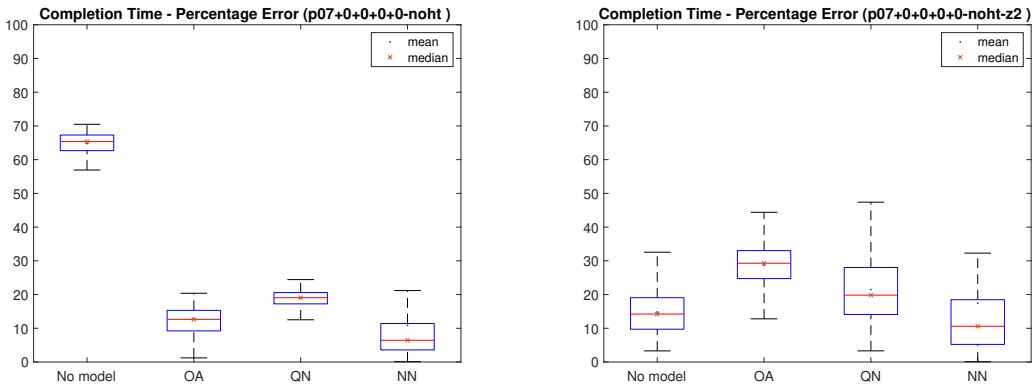



Figure 5.10: Test of the proposed model for $i \times j \times k \times l$ jobs against three other methods: queueing theory, operational analysis and no model

### 5.6.2 Can the models capture CPU contention for more than two batch jobs ($i \times j \times \cdots \times k$) ?

Sometime, it is hard and time consuming to collect all the possible combination of workloads, especially when there are many types of jobs. For example, when there are 50 types of batch jobs, the total possible combination to run only four colocated jobs is more than 292000. Therefore, there is a need to predict profiling of over all possible combination of group of running batch job by training the machine learning model on a small set of running colocated jobs (ex single and two jobs). Here, we generalize our model to predict more than two jobs by using only the profiling of individual and two jobs. Figure 5.10 shows a comparison against the other three well-known methods: Queueing theory, operational analysis, and no model. The boxplot diagram in Figure 5.10 indicates the mean and median of the prediction error for data collected across all possible of $i \times j \times k \times l$ job combinations of the DaCapo benchmarks (e.g., xalan-xalan-xalan-batik, jython-luindex-xalan-batik, ...).

The proposed solution significantly outperforms other methods by achieving 9% for mean percentage error compared with queueing theory, operational analysis, and no model, which achieve 21%, 15%, and 68%, respectively. The proposed model not only predicts completion time for $i \times j$ jobs with high accuracy, but also saves considerable time in training the model to predict completion time for more than $i \times j$ jobs and can easily be




(a) No sleep time

(b) Random sleep time (mean 2s)

Figure 5.11: Model comparison as sleep time increases.

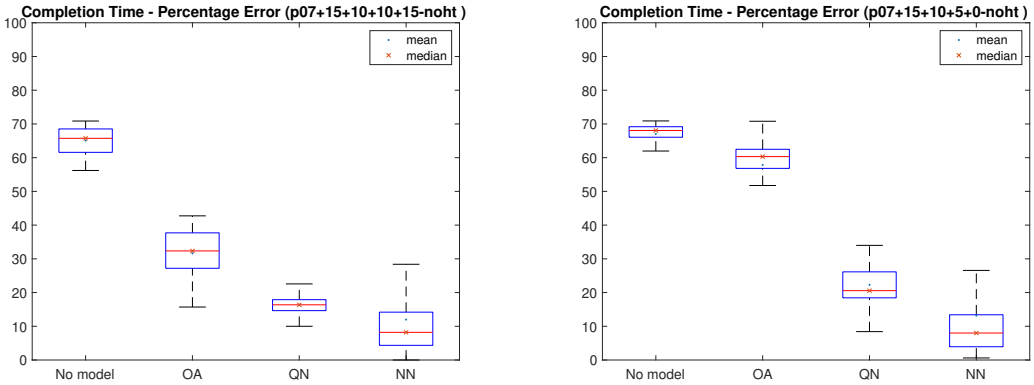
generalized to cover unforeseen workload combinations.

### 5.6.3 How sensitive the models are to the effect of sleep times?

Figure 5.11(a) illustrates the result of experiments where $i \times j \times k \times l$ jobs are simultaneous run without priorities and changed the sleep time between the runs. The intention is to mimic the effect of job dependencies, where sleep happens because a job waits for data to start. The results indicate that NN, OA and QN models are rather insensitive in terms of error after the addition of sleep times. The prediction without model would instead face an error around 65%, which can be easily fixed by adding the amount of the sleeping time (for example 2s) to the measured completion time values, as shown in the Figure 5.11(b). Adding n seconds of sleep time reduces two things: (1) dynamism or volatility in the workloads making detection easier, (2) reduces the CPU utilization and makes completion times mostly static reducing variation in its distribution making the problem trivial. Overall, this indicates that sleep times do not represent a significant challenge for prediction.

### 5.6.4 How sensitive the models are to job priorities?

Figure 5.12(a) shows results with $i \times j \times \cdots \times k$ running jobs (for example four jobs), but with different priorities assigned to the jobs using the nice command. The nice values are




(a) *nice:* +15+10+10+15

(b) *nice:* +15+10+5+0-noht

Figure 5.12: Model sensitivity to job priorities: p07+15+10+5+0-noht.

indicated in the figure caption, larger nice values mean lower priority. The default priority has a nice value of 0. The experiment demonstrates that our neural network model can predict the effect of job priorities with the lowest MAPE compared with the other three models.

### 5.6.5 Can the model capture interference among containerized batch jobs

In this section, the proposed neural network model is evaluated using containerized batch system to examine the capability in predicting the interference among containerized batch jobs that are run simultaneously using docker environment. Single and two jobs profiling datasets are collected to train the neural network model, then we test the model against four containerized batch jobs. Figure 5.13 proves that the neural networks outperform other models by achieving 24% of MAPE. Finally, our model can be generalized to predict interference of colocated jobs within containerized batch system.

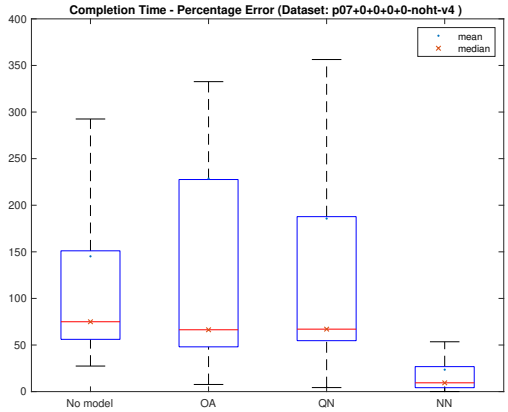



Figure 5.13: Predicting batch job interference within docker environment

## 5.7 Conclusion and Future Direction

This chapter provides an overview of the existing interference detection and prediction studies in the literature and proposes a taxonomy of the current approaches. In addition, a completion time prediction method is developed, featuring increasing accuracy in return for more profiling data. The proposed neural network-driven model does not rely on micro-architectural knowledge of systems. The model proves to be effective when the system uses or does not use job priorities or sleep time. The proposed solution predicts the completion time for $i \times j \times \ldots \times k$ containerized batch jobs with a high accuracy and saves a considerable amount of time in training the model to predict throughput for more than $i \times j$ jobs and can easily be generalized to cover unforeseen workload combinations.

The experimental results prove that our solution is effective and capable of detecting the potential interference among containerized batch jobs and can achieve up to 10% MAPE compared with other models. Table 5.5 summarize the comparison with the other three interference prediction models, which prove that our solution outperforms other models. The ✓ means that the model can predict the interference of colocated containerized batch jobs, whereas × means that the model dose not perform well in predicting the interference. The proposed solution also can be used to help system operators to perform interference-free scheduling of jobs within the system. Moreover, it can be modified to benefit users in making decision for task placement within systems to reduce substantial computing network resource consumption.

Table 5.5: Comparison of NN model against the other three proposed model.

| | Neural Networks | Baseline QT | Baseline OA | No Model |
|---|---|---|---|---|
| jobs $\leq 2$ | ✓ | ✓ | ✓ | × |
| jobs $> 2$ | ✓ | ✓ | ✓ | × |
| Sleep time | ✓ | × | × | ✓ |
| Job priorities | ✓ | × | × | × |

Our AI-driven interference prediction model has been used in an international joint project called RADON [2], which aims to develop a model-driven DevOps framework for creating and managing applications based on serverless computing systems. Regarding the future direction, the introduced AI-driven solution in this chapter is promising for containerized batch systems and can be extended to cover more advanced and complex production environments. One possible direction is to utilize the autotuning technique developed previously in Chapter 4 to precisely customize the architecture of the neural networks to detect more complex interference scenarios in distributed systems. In addition, we plan to generalize our model to cover the streaming workloads using deep learning model to predict the interference among colocated jobs. Such adoption will intend to reach power efficiency and enhance computing resource utilization.

[2]RADON project `http://radon-h2020.eu`

# Chapter 6

# Conclusion

## 6.1 Summary of Thesis Achievements

This thesis explores new approaches for performance evaluation, anomaly detection, and interference prediction of Big Data systems by implementing AI solutions. Late detection and manual resolutions of performance anomalies and system interference in Big Data systems within a cloud environment may lead to performance violations and financial penalties. Motivated by this issue, we propose an AI-based methodology for anomaly detection and interference prediction tailored to Big Data and containerized batch job platforms to better analyze system performance and effectively utilize computing resources within cloud environments.

To achieve this, we first start in Chapter 3 by examining the performance of Big Data platforms and justifying our choice of selecting the in-memory Apache Spark platform. Then, we propose an artificial neural network-driven methodology for anomaly detection for batch workloads based on knowledge of RDD characteristics to quickly sift through Spark log 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 SVM- as well as against four variants that consider

different monitoring datasets. The results show that our proposed method outperforms other methods, typically achieving 98–99% F-scores, and offering much greater accuracy than alternative techniques in detecting both the period in which the anomalies occurred and their type. In Chapter 3, 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.

In Chapter 4, we address the challenge of anomaly identification within in-memory streaming Big Data platforms by investigating agile hybrid learning techniques for anomaly detection. We describe TRACK (neural neTwoRk Anomaly deteCtion in sparK) and TRACK-Plus, two methods that can efficiently train a classes of machine learning models for performance anomaly detection using a fixed number of experiments. TRACK revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve high accuracy in a short period of time. TRACK-Plus is an automated fine-grained anomaly detection solution that adds to TRACK a second Bayesian Optimization cycle for fine-tuning the hyperparameters for artificial neural network configurations. The objective is to accelerate the search process for optimizing neural network configurations and improving the performance of anomaly classification within the data center.

A validation based on several datasets from a real Apache Spark Streaming system is performed to demonstrate that the proposed methodology can efficiently identify performance anomalies, near-optimal configuration parameters, and a near-optimal training dataset size while reducing the number of experiments. Our results indicate that the reduction in experiments that need to be run can be up to 75% compared with naïve anomaly detection training. This chapter provides a comprehensive methodology for both performance anomaly classification and the efficient optimization of artificial neural networks to detect anomalies within streaming systems.

Chapter 5 overcomes the issue of colocated batch jobs and proactively avoiding performance interference by introducing an automated prediction solution to estimate interference between co-located batch jobs within the same computing environment. An AI-driven model is implemented to predict the interference among batch services. This interference detection model can alleviate and estimate the task slowdown affected by the interference among running batch jobs. This model assists the system operators in making an accurate decision to optimize job placement. Our interference prediction solution (Chapter 5) is agnostic of the business logic internal to each job. Instead, it is learned from system performance data by applying artificial neural networks to establish the completion time prediction of batch jobs and containers within the cloud environments. We assume to have profiling data for the individual job ($i$) and two jobs ($i \times j$) , before attempting to use neural networks to predict the completion times when more than two jobs simultaneously run in the system.

We compare our model with the other three baseline models (queueing-theoretic model, operational analysis model, and an empirical model) on historical measurements of job completion time and CPU run-queue size (i.e., the number of active threads in the system) for containerized batch job. The proposed model captures multithreading, operating system scheduling, sleeping time, and job priorities with in the systems. A validation based on 4500 experiments based on the DaCapo benchmarking suite is carried out, confirming the predictive efficiency and capabilities of the proposed model with different system configurations. The experimental results prove that our solution is effective and capable of detecting the potential interference among batch jobs, achieving up to 10% MAPE compared with the other three models.

## 6.2 Future work

A wide range of challenges related to our topic hold the potential for further exploration in future work. Therefore, we present several directions that we believe are the most promising and relevant for the performance evaluation and management of Big Data and cloud computing platforms.

The current methodology in Chapter 3 requires a centralized node to run the AI model, 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 of the hardware resources and validating the proposed methodology, the current artificial neural network algorithm has been trained on offline data, which can easily generalize the algorithm to work with the online Spark systems. In terms of future work, it would be interesting to explore online anomaly detection for real-time applications.

It is worth examining the applicability of the proposed models in this thesis to other Big Data platforms, such as Hadoop, Storm, and Flink. The behavior of the Apache Spark computing framework differs from the Hadoop framework. Apache Spark uses an in-memory computing approach to store intermediate results in memory (discussed in Section 3), whereas Hadoop uses a disk-based computing approach. Therefore, we encourage further investigation of the use of the anomaly detection techniques proposed in this thesis on other Big Data technologies.

The upcoming new types of applications and complex data-intensive technologies raise new challenges for detecting and predicting anomalies in Big Data system environments. In the future, the existing techniques will become less efficient for identifying new types of anomalies that have unknown and complex features. In some cases, irrelevant performance features can conceal the presence of anomalies. Therefore, it is important to explore new promising learning techniques to identify new and unseen types of anomalies. In the literature, Deep Learning algorithms have received attention as effective solutions in a range of complex problem domains involving supervised and unsupervised machine learning. These techniques have been used efficiently to detect anomalies in other systems, such as 5G network systems and power plants. 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.

In terms of open challenges, it is crucial to deeply investigate anomaly detection and prediction for complex systems that simultaneously contain both batch and stream processing

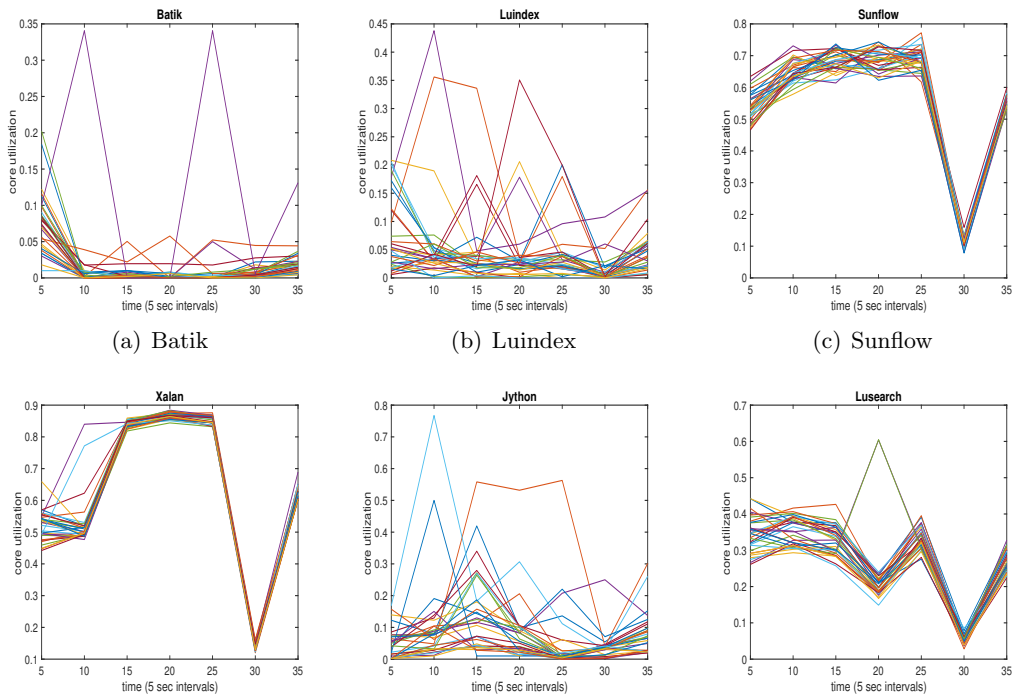
workloads. These systems will have increased complexity and performance fluctuations, which may need more effective anomaly detection solutions. Exploring more advanced machine learning algorithms may hold opportunities to accurately detect and predict the performance anomaly in distributed heterogeneous complex systems.

The introduced AI-driven solution in chapter 5 is promising for microservices systems and can be extended to cover more advanced streaming workloads within complex production environments. One possible direction is to utilize the autotuning technique developed in Chapter 4 to precisely customize the architecture of the neural networks to detect more complex interference scenarios in distributed systems. Such adoption aims to reach power efficiency, enhance computing resource utilization, and optimize job scheduling.

# Appendix A

# Appendix

Figure A.1 shows CPUs utilization when running each containerised Dacpo batch job, which reflect different performance behaviors that makes the interference detection in our case more challenging.




(a) Batik (b) Luindex (c) Sunflow

(d) Xalan (e) Jython (f) Lusearch

Figure A.1: DaCapo jobs types and CPUs performance.

## A.1 Academic contributions to conferences and workshops

Table A.1: List of academic contributions to conferences and workshops

| Activities | Overview | Date / Organizer |
|---|---|---|
| Publication | Paper Title: QT A Quality Testing Tool for Data-Intensive Applications | 21/8/2018 - Annual International Conference on ICT: Big Data, Cloud and Security - ICT-BDCS 2017 |
| Judge | Graduate School PhD Summer Showcase (Poster Competition) | 14/07/2017 - Graduate School |
| Review | Paper Title : ADaCS A Tool for Analyzing Data Collection | 06/06/2017 - 14th European Performance Engineering Workshop, Berlin (Germany) |
| | Paper about Stochastic model of extraction time evolution components | 2/10/2017 - VALUETOOLS 2017 - 11th EAI International Conference on Performance Evaluation Methodologies and Tools (Valuetools 2017) |
| | Paper about operational analysis for virtual Server sprawl including efficiency, consolidation and slowdown. | 3/11/2017 - the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018) |
| | Paper about a scalable hybrid variability for distributed systems | 3/11/2017 - 6th European Conference on Service-Oriented and Cloud Computing (ESOCC2017) |
| | Paper about a potential of Fog Computing to improve Big Data applications | 3/11/2017 - 6th European Conference on Service-Oriented and Cloud Computing (ESOCC2017) |
| | Paper about mining for unstructured system logs using event sequences to detect anomalies | 12/1/2017 - The IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) |
| | Paper about anomaly detection for docker container using neural network technique | 12/1/2018 - The IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) |
| | Paper about examining replication factor to improve performance Hadoop Distributed file system. | 7/2/2018 - 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018) |

# Bibliography


[1] Gartner, Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 18 percent in 2021.
URL https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-perce

[2] Gartner, Gartner Says by 2020 "Cloud Shift" Will Affect More Than $1 Trillion in IT Spending.
URL http://www.gartner.com/newsroom/id/3384720

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, A view of cloud computing, Communications of the ACM 53 (4) (2010) 50–58.

[4] Y. Chen, R. Sion, To cloud or not to cloud?: musings on costs and viability, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, 2011, p. 29.

[5] D. J. Dean, H. Nguyen, X. Gu, Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems, in: Proceedings of the 9th international conference on Autonomic computing, ACM, 2012, pp. 191–200.

[6] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, D. Rajan, Prepare: Predictive performance anomaly prevention for virtualized cloud systems, in: 2012 IEEE 32nd International Conference on Distributed Computing Systems, 2012, pp. 285–294. doi:10.1109/ICDCS.2012.65.

[7] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth, Performance Anomaly Detection and Bottleneck Identification, ACM Computing Surveys 48 (1) (2015) 1–35.

doi:10.1145/2791120.
URL http://dl.acm.org/citation.cfm?id=2808687.2791120

[8] S. Rogers, M. Girolami, A first course in machine learning, CRC Press, 2015.

[9] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, Supervised machine learning: A review of classification techniques (2007).

[10] Apache Spark™, Lightning-fast unified analytics engine.
URL https://spark.apache.org

[11] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.

[12] K. Wang, M. M. H. Khan, Performance prediction for apache spark platform, Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H (2015) 166–173doi:10.1109/HPCC-CSS-ICESS.2015.246.

[13] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al., The dacapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, pp. 169–190.

[14] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, A. Filieri, Quality-aware devops research: Where do we stand?, IEEE Access 9 (2021) 44476–44489.

[15] A. Alnafessah, G. Russo Russo, V. Cardellini, G. Casale, F. Lo Presti, Ai-driven performance management in data-intensive applications, Communication Networks and Service Management in the Era of Artificial Intelligence and Machine Learning (2021) 199–222.

[16] A. Alnafessah, G. Casale, Artificial neural networks based techniques for anomaly detection in apache spark, Cluster Computing (2019) 1–16.

[17] A. Alnafessah, G. Casale, Track-plus: Optimizing artificial neural networks for hybrid anomaly detection in data streaming systems, IEEE Access 8 (2020) 146613–146626.

[18] A. Alnafessah, G. Casale, 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), IEEE, 2018, pp. 201–209.

[19] A. S. Alnafessah, G. Casale, Track: Optimizing artificial neural networks for anomaly detection in spark streaming systems, in: Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools, 2020, pp. 188–191.

[20] A. Alnafessah, G. Casale, Anomaly detection for big data technologies, in: 2018 Imperial College Computing Student Workshop (ICCSW 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[21] A. Alnafessah, G. Casale, Ai driven methodology for anomaly detection in apache spark streaming systems, in: 2020 3rd International Conference on Computer Applications & Information Security (ICCAIS), IEEE, 2020, pp. 1–2.

[22] M. A. Lopez, A. G. P. Lobato, O. C. M. B. Duarte, A Performance Comparison of Open-Source Stream Processing Platforms, 2016 IEEE Global Communications Conference (GLOBECOM) (2016) 1–6doi:10.1109/GLOCOM.2016.7841533.
URL http://ieeexplore.ieee.org/document/7841533/

[23] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning spark: lightning-fast big data analysis, " O'Reilly Media, Inc.", 2015.

[24] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, Mllib: Machine learning in apache spark, The Journal of Machine Learning Research 17 (1) (2016) 1235–1241.

[25] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang, Big data analytics on Apache Spark, International Journal of Data Science and Analytics 1 (3) (2016) 145–164. doi:10.1007/s41060-016-0027-9.

[26] S. Sakr, Big Data 2.0 Processing Systems: A Survey, Springer, 2016.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, 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, USENIX Association, 2012, p. 2.

[28] Apache Spark™, DAGScheduler.
URL `https://github.com/apache/spark/`

[29] I. Ganelin, E. Orhian, K. Sasaki, B. York, Spark: Big Data Cluster Computing in Production, John Wiley & Sons, 2016.

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.

[31] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, Storm@ twitter, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, 2014, pp. 147–156.

[32] S. Shahrivari, Beyond Batch Processing: Towards Real-Time and Streaming Big Data, Computers 3 (4) (2014) 117–129. `arXiv:1403.3375`, `doi:10.3390/computers3040117`.
URL `http://www.mdpi.com/2073-431X/3/4/117/htm`

[33] S. Fu, Performance metric selection for autonomic anomaly detection on cloud computing systems, in: Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE, IEEE, 2011, pp. 1–5.

[34] D. Oppenheimer, A. Ganapathi, D. A. Patterson, Why do Internet services fail, and what can be done about it?, in: USENIX symposium on internet technologies and systems, Vol. 67, Seattle, WA, 2003.

[35] S. Pertet, P. Narasimhan, Causes of failure in web applications (cmu-pdl-05-109), Parallel Data Laboratory (2005) 48.

[36] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: Osdi, Vol. 10, 2010, p. 24.

[37] P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters, IEEE Transactions on Services Computing 12 (1) (2016) 91–104.

[38] V. J. Hodge, J. Austin, A survey of outlier detection methodologies, Artificial intelligence review 22 (2) (2004) 85–126.

[39] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM computing surveys (CSUR) 41 (3) (2009) 15.

[40] T. Wang, W. Zhang, J. Wei, H. Zhong, Workload-aware online anomaly detection in enterprise applications with local outlier factor, in: 2012 IEEE 36th Annual Computer Software and Applications Conference, IEEE, 2012, pp. 25–34.

[41] M. Markou, S. Singh, Novelty detection: a review—part 1: statistical approaches, Signal processing 83 (12) (2003) 2481–2497.

[42] T. Kelly, Detecting Performance Anomalies in Global Applications., in: WORLDS, Vol. 5, 2005, pp. 42–47.

[43] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, E. Smirni, Automated anomaly detection and performance modeling of enterprise applications, ACM Transactions on Computer Systems (TOCS) 27 (3) (2009) 6.

[44] S. Agarwala, F. Alegre, K. Schwan, J. Mehalingham, E2eprof: Automated end-to-end performance management for enterprise systems, in: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), IEEE, 2007, pp. 749–758.

[45] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, C. Konig, Pad: Performance anomaly detection in multi-server distributed systems, in: 2014 IEEE 7th International Conference on Cloud Computing, IEEE, 2014, pp. 769–776.

[46] B. Sharma, P. Jayachandran, A. Verma, C. R. Das, CloudPD: Problem determination and diagnosis in shared dynamic clouds, in: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2013, pp. 1–12.

[47] R. Gow, S. Venugopal, P. K. Ray, 'The tail wags the dog': A study of anomaly detection in commercial application performance, Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS (2013) 355–359`arXiv:1307.1743`, `doi:10.1109/MASCOTS.2013.51`.

[48] X. Gu, H. Wang, Online anomaly prediction for robust cluster systems, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 1000–1011.

[49] E. W. Fulp, G. A. Fink, J. N. Haack, Predicting computer system failures using support vector machines., WASL 8 (2008) 5–5.

[50] S. M. Erfani, S. Rajasegarar, S. Karunasekera, C. Leckie, High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning, Pattern Recognition 58 (2016) 121–134.

[51] Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, IEEE Transactions on Computers 63 (4) (2013) 954–967.

[52] J. Tan, S. Kavulya, R. Gandhi, P. Narasimhan, Visual, log-based causal tracing for performance debugging of mapreduce systems, in: 2010 IEEE 30th International Conference on Distributed Computing Systems, IEEE, 2010, pp. 795–806.

[53] W. Qi, Y. Li, H. Zhou, W. Li, H. Yang, Data mining based root-cause analysis of performance bottleneck for big data workload, in: 2017 IEEE HPCC Conferencece, IEEE, 2017, pp. 254–261.

[54] T. Huang, Y. Zhu, Q. Zhang, Y. Zhu, D. Wang, M. Qiu, L. Liu, An lof-based adaptive anomaly detection scheme for cloud computing, in: Computer Software

and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual, IEEE, 2013, pp. 206–211.

[55] M. Lin, Z. Yao, F. Gao, Y. Li, Toward anomaly detection in iaas cloud computing platforms, International Journal of Security and Its Applications 9 (12) (2015) 175–188.

[56] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, D. Xu, CLUE: System trace analytics for cloud service performance diagnosis, IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World (2014). doi:10.1109/NOMS.2014.6838348.

[57] T. Shon, J. Moon, A hybrid machine learning approach to network anomaly detection, Information Sciences 177 (18) (2007) 3799–3821.

[58] Y. Si, Y. Wang, D. Zhou, Key-performance-indicator-related process monitoring based on improved kernel partial least squares, IEEE Transactions on Industrial Electronics (2020).

[59] Y. Wang, Y. Si, B. Huang, Z. Lou, Survey on the theoretical research and engineering applications of multivariate statistics process monitoring algorithms: 2008–2017, The Canadian Journal of Chemical Engineering 96 (10) (2018) 2073–2085.

[60] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, K. Schwan, Statistical techniques for online anomaly detection in data centers, in: 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, IEEE, 2011, pp. 385–392.

[61] R. Ren, S. Tian, L. Wang, Online anomaly detection framework for spark systems via stage-task behavior modeling, in: Proceedings of the 15th ACM International Conference on Computing Frontiers, ACM, 2018, pp. 256–259.

[62] S. Lu, X. Wei, Y. Li, L. Wang, 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), IEEE, 2018, pp. 151–158.

[63] J. P. Magalhaes, L. M. Silva, Detection of performance anomalies in web-based applications, in: 2010 Ninth IEEE International Symposium on Network Computing and Applications, IEEE, 2010, pp. 60–67.

[64] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, E. Smirni, R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads, in: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2007, pp. 244–265.

[65] D. A. Menascé, Tpc-w: A benchmark for e-commerce, IEEE Internet Computing 6 (3) (2002) 83–87.

[66] T. Kelly, Detecting performance anomalies in global applications., in: WORLDS, Vol. 5, 2005, pp. 42–47.

[67] L. Yang, C. Liu, J. M. Schopf, I. Foster, Anomaly detection and diagnosis in grid environments, in: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007, pp. 1–9.

[68] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, L. Wang, Log-based abnormal task detection and root cause analysis for spark, in: 2017 IEEE International Conference on Web Services (ICWS), IEEE, 2017, pp. 389–396.

[69] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, H. Andersen, Fingerprinting the datacenter: automated classification of performance crises, in: Proceedings of the 5th European conference on Computer systems, 2010, pp. 111–124.

[70] K. Al Jallad, M. Aljnidi, M. S. Desouki, Anomaly detection optimization using big data and deep learning to reduce false-positive, Journal of Big Data 7 (1) (2020) 1–12.

[71] C. I. for Cybersecurity, NSL-KDD dataset , visited on 2021-10-5 (1999).
URL `https://www.unb.ca/cic/datasets/nsl.html`

[72] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, in: Proceedings International Conference on Dependable Systems and Networks, IEEE, 2002, pp. 595–604.

[73] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox, Capturing, indexing, clustering, and retrieving system history, ACM SIGOPS Operating Systems Review 39 (5) (2005) 105–118.

[74] H. S. Pannu, J. Liu, S. Fu, A self-evolving anomaly detection framework for developing highly dependable utility clouds, in: 2012 IEEE Global Communications Conference (GLOBECOM), IEEE, 2012, pp. 1605–1610.

[75] S. Baek, D. Kwon, J. Kim, S. C. Suh, H. Kim, I. Kim, Unsupervised labeling for supervised anomaly detection in enterprise and cloud networks, in: 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), IEEE, 2017, pp. 205–210.

[76] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al., Bigdatabench: A big data benchmark suite from internet services, in: High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, IEEE, 2014, pp. 488–499.

[77] N. J. Yadwadkar, W. Choi, Proactive straggler avoidance using machine learning, White paper, University of Berkeley (2012).

[78] S. C. Tan, K. M. Ting, T. F. Liu, Fast anomaly detection for streaming data, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[79] KDD-99, KDD Cup 1999 Data , visited on 2020-5-5 (1999).
URL `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

[80] G. Pu, L. Wang, J. Shen, F. Dong, A hybrid unsupervised clustering-based anomaly detection method, Tsinghua Science and Technology 26 (2) (2020) 146–153.

[81] R. Lu, G. Wu, B. Xie, J. Hu, Stream bench: Towards benchmarking modern distributed stream computing frameworks, in: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on, IEEE, 2014, pp. 69–78.

[82] V. I. Munteanu, A. Edmonds, T. M. Bohnert, T.-F. Fortis, Cloud Incident Management, Challenges, Research Directions, and Architectural Approach, Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (2014) 786–791doi:10.1109/UCC.2014.128.
URL http://dx.doi.org/10.1109/UCC.2014.128

[83] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, V. ICSI, Making Sense of Performance in Data Analytics Frameworks., in: NSDI, Vol. 15, 2015, pp. 293–307.

[84] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[85] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[86] G. P. Zhang, Neural networks for classification: a survey, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 30 (4) (2000) 451–462.

[87] A. Graves, A.-r. Mohamed, G. Hinton, Speech recognition with deep recurrent neural networks, in: Acoustics, speech and signal processing (icassp), 2013 ieee international conference on, IEEE, 2013, pp. 6645–6649.

[88] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal Processing Magazine 29 (6) (2012) 82–97.

[89] S. Mukhopadhyay, Quality inspection of electroplated materials using planar type micro-magnetic sensors with post-processing from neural network model, IEE Proceedings-Science, Measurement and Technology 149 (4) (2002) 165–171.

[90] J. Lampinen, S. Smolander, M. Korhonen, Wood surface inspection system based on generic visual features, Singapore: World Scientific, 1998.

[91] T. Petsche, A. Marcantonio, C. Darken, S. J. Hanson, G. M. Kuhn, I. Santoso, An autoassociator for on-line motor monitoring, in: Industrial Applications of Neural Networks, World Scientific, 1998, pp. 91–97.

[92] J.-D. Wu, C.-H. Liu, An expert system for fault diagnosis in internal combustion engines using wavelet packet transform and neural network, Expert systems with applications 36 (3) (2009) 4278–4286.

[93] J. Hoskins, K. Kaliyur, D. Himmelblau, Incipient fault detection and diagnosis using artificial neural networks, in: Neural Networks, 1990., 1990 IJCNN International Joint Conference on, IEEE, 1990, pp. 81–86.

[94] M. N. Khajavi, M. B. Menhaj, A. A. Suratgar, A neural network controller for load following operation of nuclear reactors, Annals of Nuclear Energy 29 (6) (2002) 751–760.

[95] E. B. Bartlett, R. E. Uhrig, Nuclear power plant status diagnostics using an artificial neural network, Nuclear Technology 97 (3) (1992) 272–281.

[96] J. Yang, G. Xu, H. Kong, Y. Zheng, T. Pang, Q. Yang, Artificial neural network classification based on high-performance liquid chromatography of urinary and serum nucleosides for the clinical diagnosis of cancer, Journal of Chromatography B 780 (1) (2002) 27–33.

[97] W. G. Baxt, Use of an artificial neural network for data analysis in clinical decision-making: the diagnosis of acute coronary occlusion, Neural computation 2 (4) (1990) 480–489.

[98] W. G. Baxt, Use of an artificial neural network for the diagnosis of myocardial infarction, Annals of internal medicine 115 (11) (1991) 843–848.

[99] A. Bhardwaj, A. Tiwari, Breast cancer diagnosis using genetically optimized neural network model, Expert Systems with Applications 42 (10) (2015) 4611–4620.

[100] H. B. Burke, Artificial neural networks for cancer research: outcome prediction, in: Seminars in Surgical Oncology, Vol. 10, Wiley Online Library, 1994, pp. 73–79.

[101] H. B. Burke, P. H. Goodman, D. B. Rosen, D. E. Henson, J. N. Weinstein, F. E. Harrell, J. R. Marks, D. P. Winchester, D. G. Bostwick, Artificial neural networks improve the accuracy of cancer survival prediction, Cancer 79 (4) (1997) 857–862.

[102] J. Ryan, M.-J. Lin, R. Miikkulainen, Intrusion detection with neural networks, in: Advances in neural information processing systems, 1998, pp. 943–949.

[103] C. Zhang, J. Jiang, M. Kamel, Intrusion detection using hierarchical neural networks, Pattern Recognition Letters 26 (6) (2005) 779–791.

[104] S. Jo, H. Sung, B. Ahn, A comparative study on the performance of intrusion detection using decision tree and artificial neural network models, Journal of the Korea Society of Digital Industry and Information Management 11 (4) (2015) 33–45.

[105] Z. Jadidi, V. Muthukkumarasamy, E. Sithirasenan, M. Sheikhan, Flow-based anomaly detection using neural network optimized with gsa algorithm, in: Distributed computing systems workshops (ICDCSW), 2013 IEEE 33rd international conference on, IEEE, 2013, pp. 76–81.

[106] T. G. Dietterich, E. B. Kong, Machine learning bias, statistical bias, and statistical variance of decision tree algorithms, Tech. rep., Technical report, Department of Computer Science, Oregon State University (1995).

[107] N. Morgan, H. Bourlard, Generalization and parameter estimation in feedforward nets: Some experiments, in: Advances in neural information processing systems, 1990, pp. 630–637.

[108] J. Schmidhuber, Deep learning in neural networks: An overview, Neural networks 61 (2015) 85–117.

[109] B. Kröse, B. Krose, P. van der Smagt, P. Smagt, An introduction to neural networks (1993).

[110] G. Kesavaraj, S. Sukumaran, A study on classification techniques in data mining, in: Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on, IEEE, 2013, pp. 1–7.

[111] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, nature 323 (6088) (1986) 533.

[112] M. A. Nielsen, Neural networks and deep learning, Determination Press, 2015.

[113] M. H. Beale, M. T. Hagan, H. B. Demuth, Neural network toolbox, User's Guide, MathWorks 2 (2010) 77–81.

[114] R. Lior, et al., Data mining with decision trees: theory and applications, Vol. 81, World scientific, 2014.

[115] S. K. Murthy, Automatic construction of decision trees from data: A multi-disciplinary survey, Data mining and knowledge discovery 2 (4) (1998) 345–389.

[116] R. J. McQueen, S. R. Garner, C. G. Nevill-Manning, I. H. Witten, Applying machine learning to agricultural data, Computers and electronics in agriculture 12 (4) (1995) 275–293.

[117] S. Salzberg, R. Chandar, H. Ford, S. K. Murthy, R. White, Decision trees for automated identification of cosmic-ray hits in hubble space telescope images, Publications of the Astronomical Society of the Pacific 107 (709) (1995) 279.

[118] K. Hunt, Classification by induction: application to modelling and control of non-linear dynamical systems, Intelligent Systems Engineering 2 (4) (1993) 231–245.

[119] K. B. Irani, J. Cheng, U. M. Fayyad, Z. Qian, Applying machine learning to semiconductor manufacturing, iEEE Expert 8 (1) (1993) 41–47.

[120] I. Kononenko, Inductive and bayesian learning in medical diagnosis, Applied Artificial Intelligence an International Journal 7 (4) (1993) 317–337.

[121] B. Kim, D. A. Landgrebe, Hierarchical classifier design in high-dimensional numerous class cases, IEEE Transactions on Geoscience and Remote Sensing 29 (4) (1991) 518–528.

[122] A. P. Muniyandi, R. Rajeswari, R. Rajaram, Network anomaly detection by cascading k-means clustering and c4. 5 decision tree algorithm, Procedia Engineering 30 (2012) 174–182.

[123] O. Depren, M. Topallar, E. Anarim, M. K. Ciliz, An intelligent intrusion detection system (ids) for anomaly and misuse detection in computer networks, Expert systems with Applications 29 (4) (2005) 713–722.

[124] J. R. Quinlan, Induction of decision trees, Machine learning 1 (1) (1986) 81–106.

[125] J. R. Quinlan, C4. 5: programs for machine learning, Elsevier, 2014.

[126] C. Phua, D. Alahakoon, V. Lee, Minority report in fraud detection: classification of skewed data, Acm sigkdd explorations newsletter 6 (1) (2004) 50–59.

[127] M. V. Joshi, R. C. Agarwal, V. Kumar, Mining needle in a haystack: classifying rare classes via two-phase rule induction, ACM SIGMOD Record 30 (2) (2001) 91–102.

[128] N. V. Chawla, N. Japkowicz, A. Kotcz, Special issue on learning from imbalanced data sets, ACM Sigkdd Explorations Newsletter 6 (1) (2004) 1–6.

[129] M. V. Joshi, R. C. Agarwal, V. Kumar, Predicting rare classes: Can boosting make any weak learner strong?, in: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2002, pp. 297–306.

[130] I. Steinwart, D. Hush, C. Scovel, A classification framework for anomaly detection, Journal of Machine Learning Research 6 (Feb) (2005) 211–232.

[131] N. Abe, B. Zadrozny, J. Langford, Outlier detection by active learning, in: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2006, pp. 504–509.

[132] H. W. Lilliefors, On the kolmogorov-smirnov test for normality with mean and variance unknown, Journal of the American statistical Association 62 (318) (1967) 399–402.

[133] M. F. Møller, A scaled conjugate gradient algorithm for fast supervised learning, Neural networks 6 (4) (1993) 525–533.

[134] D. M. Kline, V. L. Berardi, Revisiting squared-error and cross-entropy functions for training neural network classifiers, Neural Computing & Applications 14 (4) (2005) 310–318.

[135] K. G. Sheela, S. N. Deepa, Review on methods to fix number of hidden neurons in neural networks, Mathematical Problems in Engineering 2013 (2013).

[136] R. Caruana, S. Lawrence, C. L. Giles, Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping, in: Advances in neural information processing systems, 2001, pp. 402–408.

[137] G. Casale, C. Ragusa, P. Parpas, A feasibility study of host-level contention detection by guest virtual machines, in: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, Vol. 2, IEEE, 2013, pp. 152–157.

[138] P. Zheng, B. C. Lee, Hound: Causal learning for datacenter-scale straggler diagnosis, Proceedings of the ACM on Measurement and Analysis of Computing Systems 2 (1) (2018) 17.

[139] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, H.-A. Jacobsen, Bigbench: towards an industry standard benchmark for big data analytics, in: Proceedings of the 2013 ACM SIGMOD international conference on Management of data, ACM, 2013, pp. 1197–1208.

[140] C. Nyberg, M. Shah, N. Govindaraju, Sort benchmark home page (2009).

[141] J. Cervantes, F. Garcia-Lamont, L. Rodríguez-Mazahua, A. Lopez, A comprehensive survey on support vector machine classification: Applications, challenges and trends, Neurocomputing 408 (2020) 189–215.

[142] S. Lu, X. Wei, B. Rao, B. Tak, L. Wang, L. Wang, Ladra: Log-based abnormal task detection and root-cause analysis in big data processing with spark, Future Generation Computer Systems 95 (2019) 392–403.

[143] Apache Spark, ApacheSpark Streaming, visited on 1-10-2019 (2019).
URL `https://spark.apache.org/docs/latest/streaming-programming-guide.html`

[144] S. Qian, G. Wu, J. Huang, T. Das, Benchmarking modern distributed streaming platforms, in: 2016 IEEE International Conference on Industrial Technology (ICIT), IEEE, 2016, pp. 592–598.

[145] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The hibench benchmark suite: Characterization of the mapreduce-based data analysis, in: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), IEEE, 2010, pp. 41–51.

[146] R. Lu, G. Wu, B. Xie, J. Hu, Stream bench: Towards benchmarking modern distributed stream computing frameworks, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, IEEE, 2014, pp. 69–78.

[147] R. Han, L. K. John, J. Zhan, Benchmarking big data systems: A review, IEEE Transactions on Services Computing 11 (3) (2017) 580–597.

[148] J. Snoek, H. Larochelle, R. P. Adams, Practical bayesian optimization of machine learning algorithms, in: Advances in neural information processing systems, 2012, pp. 2951–2959.

[149] H. J. Kushner, A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise, Journal of Basic Engineering 86 (1) (1964) 97–106.

[150] N. Srinivas, A. Krause, S. M. Kakade, M. Seeger, Gaussian process optimization in the bandit setting: No regret and experimental design, arXiv preprint arXiv:0912.3995 (2009).

[151] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, N. De Freitas, Taking the human out of the loop: A review of bayesian optimization, Proceedings of the IEEE 104 (1) (2015) 148–175.

[152] Mathworks, Bayesian Optimization Algorithm, visited on 1-10-2019 (2019). URL `https://uk.mathworks.com/help/stats/bayesian-optimization-algorithm.html#bva8tie-1`

[153] S. Ceesay, A. Barker, B. Varghese, Plug and play bench: Simplifying big data benchmarking using containers, in: 2017 IEEE International Conference on Big Data (Big Data), IEEE, 2017, pp. 2821–2828.

[154] E. Casalicchio, S. Iannucci, The state-of-the-art in container technologies: Application, orchestration and security, Concurrency and Computation: Practice and Experience 32 (17) (2020) e5668.

[155] T. Šimko, L. A. Heinrich, C. Lange, A. E. Lintuluoto, D. M. MacDonell, A. Mečionis, D. Rodríguez Rodríguez, P. Shandilya, M. Vidal García, Scalable declarative hep analysis workflows for containerised compute clouds, Frontiers in big Data 4 (2021) 13.

[156] K. Miao, J. Li, W. Hong, M. Chen, A microservice-based big data analysis platform for online educational applications, Scientific Programming 2020 (2020).

[157] A. Shakir, D. Staegemann, M. Volk, N. Jamous, K. Turowski, Towards a concept for building a big data architecture with microservices, in: Business Information Systems, 2021, pp. 83–94.

[158] M. Genkin, F. Dehne, P. Navarro, S. Zhou, Machine-learning based spark and hadoop workload classification using container performance patterns, in: International Symposium on Benchmarking, Measuring and Optimization, Springer, 2018, pp. 118–130.

[159] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, A. Verma, Mitigating interference in cloud services by middleware reconfiguration, in: Proceedings of the 15th International Middleware Conference, 2014, pp. 277–288.

[160] M. Kambadur, T. Moseley, R. Hank, M. A. Kim, Measuring interference between live datacenter applications, in: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–12.

[161] A. M. El-Shamy, N. A. El-Fishawy, G. Attiya, M. A. Mohamed, Anomaly detection and bottleneck identification of the distributed application in cloud data center using software–defined networking, Egyptian Informatics Journal (2021).

[162] R. Reis, F. Rossi, T. Ferreto, Irene: Interference and high availability aware microservice-based applications placement for edge computing (2020).

[163] K. Ye, Y. Liu, G. Xu, C.-Z. Xu, Fault injection and detection for artificial intelligence applications in container-based clouds, in: International Conference on Cloud Computing, Springer, 2018, pp. 112–127.

[164] K. Ye, Y. Ji, Performance tuning and modeling for big data applications in docker containers, in: 2017 International Conference on Networking, Architecture, and Storage (NAS), IEEE, 2017, pp. 1–6.

[165] S. Garg, K. Kaur, N. Kumar, G. Kaddoum, A. Y. Zomaya, R. Ranjan, A hybrid deep learning-based model for anomaly detection in cloud datacenter networks, IEEE Transactions on Network and Service Management 16 (3) (2019) 924–935.

[166] S. Garg, A. Singh, S. Batra, N. Kumar, L. T. Yang, Uav-empowered edge computing environment for cyber-threat detection in smart vehicles, IEEE network 32 (3) (2018) 42–51.

[167] S. Amri, H. Hamdi, Z. Brahmi, Inter-vm interference in cloud environments: A survey, in: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), IEEE, 2017, pp. 154–159.

[168] V. Meyer, D. F. Kirchoff, M. L. da Silva, C. A. De Rose, An interference-aware application classifier based on machine learning to improve scheduling in clouds, in: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2020, pp. 80–87.

[169] N. Mishra, J. D. Lafferty, H. Hoffmann, Esp: A machine learning approach to predicting application interference, in: 2017 IEEE International Conference on Autonomic Computing (ICAC), IEEE, 2017, pp. 125–134.

[170] C. Delimitrou, C. Kozyrakis, Quasar: resource-efficient and qos-aware cluster management, ACM SIGPLAN Notices 49 (4) (2014) 127–144.

[171] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, J. Pei, A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads, in: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–11.

[172] Y. Yuan, H. Wang, D. Wang, J. Liu, On interference-aware provisioning for cloud-based big data processing, in: 2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS), IEEE, 2013, pp. 1–6.

[173] Q. Noorshams, D. Bruhn, S. Kounev, R. Reussner, Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, 2013, pp. 283–294.

[174] G. Casale, S. Kraft, D. Krishnamurthy, A model of storage i/o performance interference in virtualized systems, in: 2011 31st International Conference on Distributed Computing Systems Workshops, IEEE, 2011, pp. 34–39.

[175] X. Bu, J. Rao, C.-z. Xu, Interference and locality-aware task scheduling for mapreduce applications in virtual clusters, in: Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, 2013, pp. 227–238.

[176] G. Kousiouris, T. Cucinotta, T. Varvarigou, The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks, Journal of Systems and Software 84 (8) (2011) 1270–1291.

[177] K. Ye, Y. Kou, C. Lu, Y. Wang, C.-Z. Xu, Modeling application performance in docker containers using machine learning techniques, in: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2018, pp. 1–6.

[178] X. Tang, Q. Liu, Y. Dong, J. Han, Z. Zhang, Fisher: An efficient container load prediction model with deep neural network in clouds, in: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), IEEE, 2018, pp. 199–206.

[179] B. Sharma, T. Wood, C. R. Das, Hybridmr: A hierarchical mapreduce scheduler for

hybrid data centers, in: 2013 IEEE 33rd International Conference on Distributed Computing Systems, IEEE, 2013, pp. 102–111.

[180] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, X. Shi, Evaluating mapreduce on virtual machines: The hadoop case, in: IEEE International Conference on Cloud Computing, Springer, 2009, pp. 519–528.

[181] H. Kang, Y. Chen, J. L. Wong, R. Sion, J. Wu, Enhancement of xen's scheduler for mapreduce workloads, in: Proceedings of the 20th international symposium on High performance distributed computing, 2011, pp. 251–262.

[182] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, Quantitative system performance: computer system analysis using queueing network models, Prentice-Hall, Inc., 1984.

[183] J. D. Little, A proof for the queuing formula: L= $\lambda$ w, Operations research 9 (3) (1961) 383–387.

[184] W. Wang, G. Casale, A. Kattepur, M. Nambiar, Maximum likelihood estimation of closed queueing network demands from queue length data, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, 2016, pp. 3–14.

## 6.2 List of Publications

IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received February 6, 2021, accepted February 26, 2021, date of publication March 9, 2021, date of current version March 26, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3064867*

# Quality-Aware DevOps Research: Where Do We Stand?

**AHMAD ALNAFESSAH, ALIM UL GIAS, RUNAN WANG, LULAI ZHU, GIULIANO CASALE, AND ANTONIO FILIERI**
Department of Computing, Imperial College London, London SW7 2AZ, U.K.

Corresponding author: Giuliano Casale (g.casale@imperial.ac.uk)

The work of Ahmad Alnafessah was supported in part by the National Center for Artificial Intelligence and Big Data Technologies, King Abdulaziz City for Science and Technology (KACST), Saudi Arabia. The work of Alim Ul Gias was supported by the Commonwealth Scholarship Commission in the U.K. The work of Ahmad Alnafessah, Lulai Zhu and Giuliano Casale was supported in part by the RADON project, funded by the European Union's Horizon 2020 Research and Innovation Program under Grant 825040.

**ABSTRACT** DevOps is an emerging paradigm that reduces the barriers between developers and operations teams to offer continuous fast delivery and enable quick responses to changing requirements within the software life cycle. A significant volume of activity has been carried out in recent years with the aim of coupling DevOps stages with tools and methods to improve the quality of the produced software and the underpinning delivery methodology. While the research community has produced a sustained effort by conducting numerous studies and innovative development tools to support quality analyses within DevOps, there is still a limited cohesion between the research themes in this domain and a shortage of surveys that holistically examine quality engineering work within DevOps. In this paper, we address the gap by comprehensively surveying existing efforts in this area, categorizing them according to the stage of the DevOps lifecycle to which they primarily contribute. The survey holistically spans across all the DevOps stages, identify research efforts to improve architectural design, modeling and infrastructure-as-code, continuous-integration/continuous-delivery (CI/CD), testing and verification, and runtime management. Our analysis also outlines possible directions for future work in quality-aware DevOps, looking in particular at *AI for DevOps* and *DevOps for AI software*.

**INDEX TERMS** DevOps, CI/CD, infrastructure as code, testing, artificial intelligence, verification.

## I. INTRODUCTION

The rapid evolution of cloud and virtualization technologies over the last 15 years has brought to software vendors the ability to easily and programmatically control a broad set of computing resources in the execution environment of a software system. This development has then paved the way to increased levels of automation in the way software applications are delivered to production, for example by enabling continuous integration of new version of the application code. The resulting delivery paradigm, which places more attention towards continuous re-release, unified tooling and organizational processes, is often referred to as *DevOps*. Common DevOps advances include for example continuous-integration/continuous-delivery (CI/CD) pipelines, and highly-automated orchestration and configuration solutions for the runtime environment [1], [2].

DevOps tools and methods have also reduced the cultural and methodological divide between developers and operators [3], leading to the formation of many new organizational structures within software vendors, such as virtual teams composed of both developers and operators, and the establishment of new professional figures often referred to as DevOps engineers, who center their activity on tooling and automation across the whole application lifecycle.

A methodology that releases application versions at a faster pace than traditional methods is effective only if coupled with testing tools that can reduce the likelihood of failures in production. For this reason, quality assurance in DevOps is often a synonym of continuous *functional* testing methods to check the correctness of application prior to deployment. However, to accelerate the pace of delivery, quite often DevOps

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

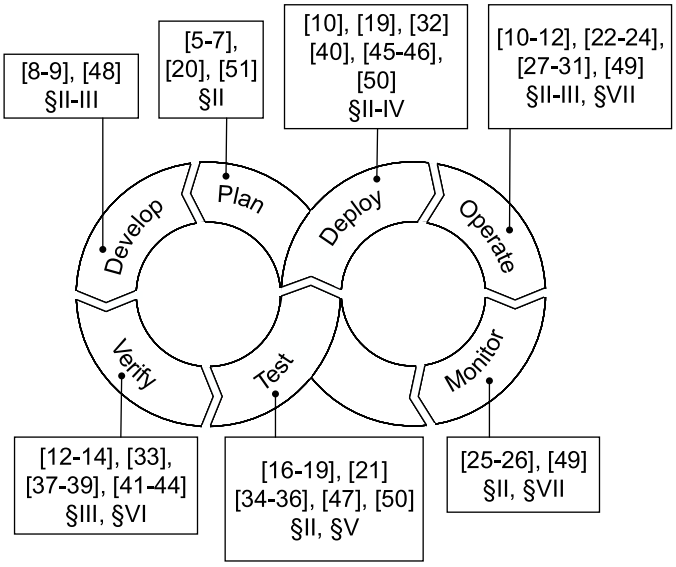



**FIGURE 1.** Stages of the DevOps Cycle. The citations indicate works that mainly perform research in the context of that stage. Relevant sections of this paper are also indicated.

testing methods lead to restrict the depth of scrutiny of the software system, leaving rooms for significant defects and bugs to still emerge in production. Several defects are related to properties other than correctness, such as performance, reliability, or cost. This problem has raised the attention of many research groups, leading to several research works that attempt to couple DevOps methods with novel forms of rapid and automated quality assurance, centered on a broader range of quality characteristics (*e.g.*, performance, reliability, availability, scalability, ...).

While the research community has sustained this effort by publishing numerous studies and innovative tools and methods to support quality analyses within DevOps, there is still a limited cohesion between the research themes in this domain and a shortage of surveys that holistically examine quality engineering work within DevOps. In this work, we address this gap by offering a survey of recent efforts in the area of *quality-aware* DevOps. Our work focuses on research efforts that aim at coupling the rapid delivery of DevOps with techniques to ensure that software artifacts also meet quality expectations on non-functional properties. Our analysis covers several tens of papers, categorizing the different contributions according to the software engineering area they mainly contribute to including architectural design, modeling, continuous integration and delivery, infrastructure, testing, verification, CI/CD and infrastructure-as-code, runtime management. We look also at the positioning of these works within the DevOps lifecycle stages. The paper, in particular, reveals that a highly-diverse body of work has been published on the subject, which yet leaves ample margins to carry out further investigations in areas that are systematically under-investigated from a quality angle, e.g., CI/CD & IaC and architectural design. We further look at the state-of-the-art on two emerging trends: *AI for DevOps* and *DevOps for AI software*, as these are expected to dominate the DevOps landscape in the years to come, and survey early works in these areas.

### A. THE DevOps CYCLE

The reference stages of the DevOps lifecycle we consider are illustrated in Figure 1. The figure lists the bibliographic references that we classify as doing research in the context of that stage. Note that a paper may touch upon multiple stages, in which case it is listed on all relevant stages. The stage definitions we adopt are as follows:

- *Plan*: This stage aims at defining the objectives and requirements of the software production, along with the initial plan for updates and release across iterations.
- *Develop*: Based on the plan, developers focus on development and reviewing of software code and/or IaC. Typically, in this phase the code undergoes frequent commits on code repositories as well as integration and unit tests based on build automation tools.
- *Verify*: Verification is the process to evaluate the correctness of software artifacts in terms of the requirements.
- *Test*: In this phase, automation testing will be performed continuously to ensure the quality of the software artifact. Contrary to traditional tests, this phase can include the release of trial versions to part of the end user base, by means of canary testing.
- *Deploy*: This stage focuses on continuously re-deployment of the software in the production environment. This phase entails the problem of configuration management of the target platforms and resources.
- *Operate*: Operation in DevOps cycles deals with configuration and management of the software application

after deployment, e.g., resource provisioning and auto-scaling. Orchestrators and other runtime methods can be used to automatically instantiate and adapt at run-time the application topology and components.
- *Monitor*: Monitor the performance of deployed applications by collecting and analyzing usage data, which can help to detect and identify exceptions and provide feedback to iteratively improve the software. Continuous tracing and diagnostics of production problems is important to guide the evolution of the application across release cycles.

The above phases are qualitatively similar to those carried out in traditional software engineering methodologies, with the main difference being that the process of releasing the software artifact is continuous and highly-automated. We point the reader to books such as [4] for additional details on the above methodological phases.

### B. METHODOLOGY

Our review methodology is as follows. We examine computer science peer-reviewed journals and conference papers written in English between 2015 and 2020. Papers are obtained with systematic searches using Google scholar for search strings always including "DevOps", one quality term between "performance", "scalability", "quality", "quality-aware", "reliability", "availability", or "survivability"; and a third term matching the title of the sections of this paper (e.g., "verification", "CI/CD", etc). Books, presentations, thesis, technical reports, white papers, and patents are excluded from this study. After collecting the pool of paper, due to space limitations we have narrowed down the list to around 10 in each section. This has been done with manual screening of each paper, trying to identify a subset of papers that was representative of the whole category, as our goal is to illustrate different research challenges and approaches, rather than exhaustively list every individual contribution.

We have aligned these collected pool of papers with different stages of the DevOps lifecycle and presented in Figure 1. We also present a mapping of these papers with the considered quality attributes in Table 1. The table also classifies the papers based on their methodology. We considered the following methodologies:"Model-based" - if the authors emphasize modeling abstractions, "Empirical" - if the authors designed a model-free approach and their decision process is based on the collected static or runtime data, and "Hybrid/Other" - if the authors used a combination of model-based and empirical approach or other methods.

### C. CONTRIBUTIONS AND ORGANIZATION

Summarizing the core contributions of this paper are as follow:

- We survey recent works in the area of quality-aware DevOps, outlining the main contributions and comparatively position them to other DevOps works within the same field.
- We organize the surveyed papers into different categories, both globally across the survey and locally within each research area, offering a better qualitative understanding of the areas of main interest and current research gaps.
- For each category, we identify open research challenges, offering several ideas for further exploration by researchers in upcoming years.
- We outline open research directions and ongoing work in emerging DevOps trends related to the use of AI technology, which we expect to foster novel solutions in the quality engineering space in the near future.

The rest of the papers is organized as follows. Sections 2-8 survey recent research work across the considered areas, namely: architecture design (§2), model-based DevOps (§3), CI/CD (§4), testing (§5), verification (§6), and runtime management (§7). Each section outlines context, summarizing research papers, and giving guidance for future work. Section 8 discusses ongoing work in *DevOps for AI* and in *AI for DevOps*. Section 9 draws conclusions.

## II. ARCHITECTURE DESIGN

*Context:* The architecture design of today's software systems, particularly cloud applications, allows for a rapid extension of features and functions with minor modifications to existing implementations. This requires increased communication and collaboration between development and operation teams to achieve a strong integration of coding, building, testing, packaging, releasing, configuring and monitoring activities. Therefore, designing the software architecture has profound implications not just on the software, but also on the overall DevOps delivery process. Recent research on architecture design in the context of DevOps centers around the following challenges:

**AD1** Refactoring monolithic applications into microservices;
**AD2** Modeling the architectures of cloud-native applications;
**AD3** Deciding architecture design variants through testing or experimentation;
**AD4** Adopting new architectural styles with best practices and tactics.

*Quality-Aware DevOps Research:* By migrating a mobile-app backend to microservices, Balalaie *et al.* [5] show how the microservices architecture could be beneficial, especially in shipping new features and providing built-in scalability. They also report on architectural patterns observed in migration projects, which can help practitioners and consultants to address **AD1** with a DevOps methodology.

Two papers are found to target **AD2**. Di Nitto *et al.* [48] introduce SQUID, a framework that provides DevOps-ready software architecture descriptions through model-based documentation of software architectures and their quality properties in DevOps scenarios. Meanwhile, Heinrich *et al.* [49] propose iObserve, an approach to enriching and updating

**TABLE 1.** Mapping of the reviewed paper with the quality attributes. The corresponding section of the papers are also provided.

| Quality Attr. \ Paradigm | Model-based | Empirical | Hybrid/Other |
|---|---|---|---|
| Maintainability / Transparency / Evolution | | §II: [5]–[7] | §III: [8], [9] |
| Performance / Scalability | §III: [10]–[14]<br>§IV: [15]<br>§V: [16]–[18]<br>§VII: [12], [27]–[31] | §II: [5]–[7], [19], [20]<br>§V: [21]<br>§VII: [22]–[24] | §VII: [25], [26] |
| Reliability / Fault-tolerance | §III: [11], [32], [33]<br>§V: [34]–[36] | §II: [6] | |
| Safety / Correctness | §VI: [37], [38] | §VI: [39] | §IV: [40]<br>§VI: [41]–[44] |
| Survivability / Security | §IV: [45]<br>§V: [47] | §IV: [46] | |
| Cost of ownership | §III: [10], [11] | §II: [20] | |
| Any quality attribute | §II: [48]–[50] | §II: [51] | |

the architectural development models of cloud-based software applications with operational observations so that the resulting architectural runtime models are usable during the operation phase.

In order to tackle **AD3**, Avritzer *et al.* [19] introduce an approach to automatically assessing the scalability of configuration alternatives for the microservices architecture through load testing. This approach provides a domain-based metric that can be used to make informed decisions about which configuration alternative to select. By contrast, Jiménez *et al.* [50] proposes a framework for quality-driven adaptive continuous experimentation. This framework dedicates three feedback loops to control the satisfaction of high-level quality goals through experiment design and conduct experimental trials for infrastructure configuration and architecture design variants.

Reference [6], [20], [51] and [7] provide solutions to **AD4** for the microservices architectural style by practicing DevOps in industrial use cases. Using OpenStack as case study, the authors of [20] compares the efficiency of DevOps in container-based and VM-based deployments and explores the scalability of stateless and stateful containerized components. Reference [51] discusses DevOps practices and architecting tactics for developing large-scale systems, like a Neo-Metropolis BDaaS platform. The authors of [6] show how the properties of the microservices architecture facilitate the scalability, agility and reliability of e-commerce applications. Differently, [7] advocates the use of microservices for software development in connected car business and proposes a suitable team setup for establishing a DevOps culture.

*Analysis of Open Challenges:* DevOps architectural work has focused on microservices, but novel research opportunities arise to extend this research, for example by including in architecture design also Function as a service (FaaS) elements. FaaS refers to a novel serverless computing paradigm that may radically evolve the landscape of software architectures. It enables software engineers to virtualize the business logic of an application as individual functions registered in the cloud. Because of advantages brought by the serverless FaaS paradigm, software vendors tend to migrate their existing products onto FaaS platforms, e.g., AWS Lambda and OpenFaaS. There is however a lack of approaches available in the literature to automatically decomposing monolithic applications into architectures containing serverless functions. Moreover, the choice of a suitable architectural granularity is an open problem, e.g., when to prefer a serverless function to a microservice.

## III. MODEL-BASED DevOps

*Context:* Modeling provides a flexible and efficient means to study the qualitative and quantitative properties of a given system in an abstract language, thus being widely applied in support of various development and operation activities. As aforementioned, it can help with the architecture design of modern software systems [48], [49]. Models can either take the form of mathematical models or code in a domain-specific language to declare properties. Within this trend, models may refer to the system or its environment. In the former case, they provide abstractions for the software inter-dependencies or dynamic behavior. In the latter case, they provide abstractions to specify and automate the configuration of a target deployment environment.

This section focuses on model-based DevOps frameworks and methods that cope with the following challenges:

- **MD1** Assessing the quality of systems under development;
- **MD2** Optimizing system configurations in a cloud environment;
- **MD3** Specifying the required underlying infrastructure as code.

*Quality-Aware DevOps Research:* As a requisite for quality assurance, **MD1** receives continuous attention from the research community. Gorbenko *et al.* [33] provide a time-probabilistic failure model for distributed systems that follows the service-oriented paradigm to define interaction with clients over the Internet and clouds. A three-layer queueing network is proposed by Barna *et al.* [12] for developing autonomic management systems. This model is proven to be robust and accurate in predicting system performance under a variety of workloads and topologies. Take a tax fraud detection system as an example, Perez-Palacin *et al.* [13]

**TABLE 2.** Overview of publications on architecture design.

| Ref. \ Aim | Refactoring | Modeling | Decision | Adoption |
|---|---|---|---|---|
| Balalaie et al. [5] | ✓ | | | |
| Di Nitto et al. [48] | | ✓ | | |
| Heinrich et al. [49] | | ✓ | | |
| Avritzer et al. [19] | | | ✓ | |
| Jiménez et al. [50] | | | ✓ | |
| Kang et al. [20] | | | | ✓ |
| Chen et al. [51] | | | | ✓ |
| Hasselbring et al. [6] | | | | ✓ |
| Schneider [7] | | | | ✓ |

**TABLE 3.** Summary of publications on model-based DevOps.

| Ref. \ Aim | Assessment | Optimization | Specification |
|---|---|---|---|
| Anatoliy et al. [33] | ✓ | | |
| Barna et al. [12] | ✓ | | |
| Perez-Palacin et al. [13] | ✓ | | |
| Peuster and Karl [14] | ✓ | | |
| Guerriero et al. [10] | | ✓ | |
| Suk et al. [32] | | ✓ | |
| Sun et al. [11] | | ✓ | |
| Rahman et al. [8] | | | ✓ |
| Tamburri et al. [9] | | | ✓ |

show the use of Petri nets for the performance analysis of data-intensive applications. Peuster and Karl [14] present an automatable and platform-agnostic modeling approach that can profile the performance of an entire service function chain at once.

**MD2** is a common issue that needs to be addressed in the deployment and operation phases so as to adapt DevOps for a cloud environment. Guerriero *et al.* [10] propose SPACE4Cloud, an integrated framework for the deployment optimization and resource allocation of cloud applications represented as PCM models. A proactive application placement algorithm is introduced by Suk *et al.* [32]. This algorithm uses failure indexes evaluated by modeling application turnover and infrastructure failure as stochastic processes. Sun *et al.* [11] present a stochastic model and an optimization method to minimize the completion time, availability degradation, and monetary cost of the rolling upgrade procedure through appropriate parameterization.

The emergence of Infrastructure-as-Code (IaC) is a response to **MD3**. IaC often relies on textual resource models to configure the application environment. It is especially useful to increase the repeatability of configuration tasks in distributed architectures, where many dependencies exist between software components and virtualized resources. Comparatively to other areas surveyed in this paper, IaC quality research is in its infancy and relatively few works exist. Two representative examples are [8] and [9]. In [8], the authors discuss a qualitative analysis of over 1700 IaC scripts to identify code smells in IaC code, i.e., code snippets that are indicative of some deeper violation of design principles or best practices. The paper considers in particular security smells related to cryptography, authentication and hard-coded secrets, among others. The authors of [9] explore the use of intent modeling as a way to ensure the correctness of IaC. This is based on the idea of specifying IaC in terms of the high-level final state/goal that needs to be reached, operating at a higher level of abstraction than detailed sub-activities. The paper focuses on the standardized TOSCA language, which offers an implementation of this approach. TOSCA models are also compatible with the execution of IaC scripts in languages such as Ansible, effectively offering polyglot IaC. Rahman *et al.* [52] provides a systematic survey of quality in IaC, noting its current underdevelopment in the research literature. They carry out an analysis of 32 publications. Within this paper, IaC work insists primarily on quality dimensions such as the reliability, repair, testing, idempotency of IaC scripts.

*Analysis of Open Challenges:* Although the serverless FaaS paradigm simplifies user involvement in resource allocation at runtime and saves operating costs by billing executions at the function level. It is often difficult to decide the optimal configuration of serverless functions comprising an application, which minimizes the operating costs while satisfying the performance requirements. This raises the needs for models that can accurately predict the performance of FaaS-based applications as well as approaches that can effectively optimize their deployment. In the literature, no work seems to have been carried out for either.

Our survey also reveals that IaC research is still at an early stage, and thus many outlets for research exist in developing tools to increase the quality of IaC artifacts. Because IaC scripts can be specified using model-based declarative languages (e.g., TOSCA) or be written with specialized languages (e.g., Ansible), a research question is how to develop holistic and polyglot defect prediction and debugging environments for IaC. Another potential aspect to consider in IaC involves the quantification of costs associated to maintenance and configuration operations. At present such costs can only be indirectly estimated from execution logs, but there is practical business value in estimating these figures from code or software artifacts. Lastly, the relative merits of different IaC technical approaches used in industry are currently not well understood in the literature. More systematic investigations on these matters may be relevant to researchers.

## IV. CONTINUOUS INTEGRATION AND DELIVERY (CI/CD)

*Context:* CI/CD pipelines provide the technical means to automate recurring tasks related to deployment, testing, and orchestration of cloud native applications. Market solutions such as Jenkins, CircleCI, Trevis, and several others can be used to coordinate delivery and quality checks on the application source code and associated software artifacts, prior to their production use. Several books and papers overview the general properties of CI/CD, e.g., [1] discusses the broader applicability and benefits of CI/CD in industrial context. More recently, [2] provides a large-scale empirical study on the impact of continuous integration on software development practice. Strategies to concretely adopt CI/CD in organizations are exemplified in [53].

**TABLE 4. Comparing publications on CI/CD.**

| Ref. \ Attrb. | SLA-aware | Security-aware | Data-aware |
|---|---|---|---|
| R. Meissner et al. [40] | | | ✓ |
| M. Mazkatli et al. [46] | | ✓ | |
| F. Boyer et al. [15] | ✓ | | |
| N. Ferry et al. [45] | | ✓ | |

*Quality-Aware DevOps Research:* It is well known that CI/CD finds immediate application to quality assurance via unit testing of functional properties. For example, the practitioner interviews in [54] reveal that deployability via CI/CD and architectural design to improve test quality are relevant dimensions in DevOps. We point the reader to a broader overview of related testing research in Section V. We instead here focus on innovative uses of CI/CD in the context of quality assurance, which offer novel outlets for research. Research in quality-aware CI/CD has centered on the following challenges:

**CC1** Performance-aware CI/CD
**CC2** Data-aware CI/CD
**CC3** Secure CI/CD

An example of work that addresses **CC1** is [15], which uses CI/CI to ease updates while releasing new versions of microservices. The authors propose an architecture-based CI/CD approach, rather than using scripts, to update the microservices while in production. They define templates for different architectural models based on which an application can be updated to a target architecture using simple commands. In addition, they incorporate common update strategies, such CleanRedeploy, BlueGreen, Canary, etc., from which an appropriate strategy can be selected to satisfy specific SLA requirements. They demonstrate the effectiveness of the approach updating an application in production.

Another example of work in the context of **CC1** is [46], where the authors propose a roadmap to apply CI/CD to incrementally maintain and parameterize application performance models. The approach entails to react to changes in the application source code and then apply targeted monitoring and statistical estimation methods to update resource demands, probabilities of selecting particular code branches, loop execution numbers, and other relevant parameters. Such updates are essential to continuously evolve the quality-aware toolchain analysis synchronously with code commits.

An instance of work that tackles **CC2** is as follows. Data stores based on query languages such as RDF can be directly stored on systems such as Github, allowing to coordinate the publishing of data with CI/CD pipelines. This triggers the question on what is the inter-play between CI/CD and data quality engineering. [40] provides an overview of tools that can be integrated in CI/CD pipelines to continuously meet quality requirements on data. The include utilities for RDF serialization quality checks, ontology validation tools, data anti-patterns, linked open quality data assessment. An example of CI pipeline to holistically coordinate the surveyed data quality assurance tools is described in the paper.

The recent work in [45] illustrates another approach to quality-aware DevOps, where the goal of the study is to continuously integrate and orchestrate a system so to ensure security and privacy. This aligns to challenge **CC3**. Model-driven engineering methods are coupled with secure DevOps practices to allow continuous changes in the deployment environment. This is based on a so-called "models@runtime" approach, where the application model is evolved directly in the production environment in response to dynamic events that occur therein.

*Analysis of Open Challenges:* The above papers exemplify novel trends in CI/CD towards integrating in the CI/CD pipeline the specification of data services. It is possible to envision that similar needs will arise in connection with AI/ML services, which require a continuous evolution of data pipelines, learning and training services alongside the application. CI/CD support specific to such kind of services offers outlet for novel research.

## V. DevOps TESTING

*Context:* DevOps has been widely adopted in enterprises, which leads to shortened development cycles and involvement of automation. With speedy iterations, the risk and cost of quality assurance increase at the same time. Testing is of great importance to ensure the quality of software in DevOps practice. In particular, automating the testing process enables continuous testing of the frequent code changes occurring throughout the development cycle. The following challenges are highlighted in the literature:

**TS1** Automatic workload selection and specification of target services.
**TS2** Test automation frameworks to enable automatic execution within DevOps cycles.
**TS3** Employing testing strategies to adapt to frequent changes in DevOps.

*Quality-Aware DevOps Research:* In the phase of test specification, a central goal is to maximize the coverage of new changes and specify unit tests aiming at specific and identifiable target services or functions, to make test results quickly actionable for developers. Besides common functional testing, which is not specific of DevOps, automated load testing allows spotting possible performance issues during the integration phase, preventing them from manifesting in production. In [16], Schulz *et al*. propose an approach of load testing selection based on contextual information that focuses on **TS1**. Workloads can be automatically selected according to monitoring data, target services, along with testing requirements in the proposed load testing process.

The authors in [17] focus on representative workload models for load testing of individual microservices in session-based systems. Two algorithms are proposed to enable extracting specific workload for the microservices under testing and consequent adjustments of workload models. Such an approach aims at only target microservices and their

**TABLE 5. Comparing publications on DevOps testing.**

| Ref. \ Phase | Specification | Modeling | Execution |
|---|---|---|---|
| Schulz et al. [16] | ✓ | ✓ | ✓ |
| Schulz et al. [17] | ✓ | ✓ | |
| Schulz et al. [18] | ✓ | ✓ | |
| Ferme et al. [34] | ✓ | ✓ | |
| Pietrantuono et al. [35] | | | ✓ |
| Schermann et al. [21] | | | ✓ |
| Schermann et al. [36] | ✓ | | ✓ |
| Dullmann et al. [47] | | ✓ | ✓ |

dependencies so that they can reduce the testing cost, addressing the issue of **TS1**.

In [18], the authors also propose to solve **TS1** by introducing a behavior-driven load testing language (BDLT), which is designed to describe performance concerns in natural language that can be easily adopted by users. Based on BDLT, testing workloads can be automatically generated with the method in [16].

To enable the automated execution of tests for the purpose of faster integration and delivery, several test automation frameworks and tools have been proposed to address the problem of **TS2**. For example, in [34], the authors address automated testing workflows in the process of continuous integration, focusing on unit tests and integration tests. Pietrantuono *et al.* in [35] present a continuous software reliability testing approach called DevOpRET. This approach mainly involves usage monitoring and operational profile estimation and updating. By monitoring the endpoint users, estimated operational profile is able to be updated with the actual user profile. At each DevOps cycle, the reliability testing can be executed based on the continuously updated operational profile.

In addition, testing techniques such as canary releases and shadow/dark launches are being increasingly adopted as strategies to automate testing execution in **TS3**. In an empirical study on continuous experimentation [21], the authors provide an overview of continuous experimentation practices that contain canary releases, dark launches and A/B testing in both research and practice of DevOps. In [36], Schermann *et al.* proposed a live testing model and implemented a middleware, Bifrost, to specify testing strategies and execute tests through traffic routing. Bifrost is able to describe release techniques with multiple phases including canary releasing, dark launching, A/B testing and gradual rollout in YAML-based language. The authors of [47] address the issues of dependability and security of CD pipelines, proposing involving testing strategies, such as canary releases and A/B testing, into building and integration pipelines execution.

*Analysis of Open Challenges:* Rapid changes bring new challenges to test specification and execution in practice. Learning and analyzing the internal and external dependencies between components at specification stages could also inform test specifications, enabling a more effective identification of tests that need to be re-executed after a change, trading off test complexity for coverage. In addition, test automation needs to meet the dynamics of iterations in DevOps cycles. For example, the objective of each iteration may change, which will lead to involve different test strategies into the respective iteration to meet the QA requirements.

## VI. VERIFICATION IN DevOps

*Context:* Verification complements testing in software quality assurance processes. Compared to testing, it leverages mathematical abstractions and code/model semantics to prove the (partial) correctness of artifacts with respect to a variety of properties. While still in their infancy, verification methods tailored to DevOps are gaining traction in both industry and academia for their potential to deliver stronger quality guarantees than testing. This section reviews a selection of paper relevant to verification in DevOps (Table 6). The main open challenges discussed through this section are summarized in the following points:

**VE1** Develop diff-time verification methods for prompt and localized feedback to keep developers engaged
**VE2** Increase compositionality and incrementality to support the analysis of large, rapidly changing code bases
**VE3** Feed information from design time to runtime and viceversa to improve runtime tasks and verification

*Quality-Aware DevOps Research:* Despite not as widely adopted as testing, verification is applied at several stages of a DevOps cycle, including, in order: design, build, diff, land, and production times [55].

*Design-time* methods analyze pre-implementation software artifacts, including goal or architectural models from the DevOps plan and create phases. User-provided abstractions, e.g., statecharts or unambiguous dialects of UML, are automatically translated into formal models to verify artifacts' properties. For example, analytical models of performance and reliability obtained from higher-level modeling languages like the Palladio Component Model [56] or OASIS TOSCA can be analyzed with numerical routines or probabilistic model checking [57].

*Build-time* methods are usually embedded within compilers and IDEs, providing quick feedback to the developers about the module they are implementing. These are usually light-weight static analyses performed with tools like Valgrind [41] and ASan to detect buffer overflows or dangling pointers and profile C/C++ artifacts, or FindBugs to localize several classes of bugs in Java artifacts [42]. While most of these methods are not specific to it, DevOps needs started to push for adapting them into staged analyses, where static code information computed during build time are carried on to later development stages and runtime to enable subsequent analyses [58]. For example, in [39], Beigi-Mohammadi *et al.* exploit control flow analysis to extract security-related predicates to be checked during operation, enabling automatic adaptation actions for early countering potential attacks.

*Diff-time* is the gatekeeping at the end of code creation, when submitted code waits for review and approval. Verification methods in this phase usually completes within a few

**TABLE 6. Verification and static analysis through DevOps phases.**

| Phase / Ref. | Design/ build-time | Diff time | Land time | Runtime |
|---|---|---|---|---|
| Koziolek et al. [56] | ✓ | | | |
| Brunnert et al. [57] | ✓ | | | |
| Nethercote et al. [41] | ✓ | | | |
| Sadowski et al. [42] | ✓ | ✓ | | |
| O'Hearn [55] | | ✓ | | |
| Backes et al. [37] | | ✓ | | |
| Larus et al. [44] | | ✓ | ✓ | |
| Harman et al. [43] | | ✓ | ✓ | |
| Beigi-Mohammadi et al. [39] | ✓ | | | ✓ |
| Filieri et al. [38] | | | | ✓ |

tens of minutes [55] to allow their reports to complement human code reviews. The peculiarity of this phase is its intrinsic incrementality: only portions of an artifact change since the last run of the analysis, and they can be identified by the diff. While few academic tools specialize for diff-time analysis, notable industrial contributions include Facebook's Infer [43], Amazon's s2n [37], and Microsoft' Prefast [44].

*Land-time* occurs after a diff is approved and before release to production. This phase is allowed longer execution time (typically from hours to overnight) and can operate from built and executable modules, which can be analyzed both statically and dynamically [43]. Microsoft Prefix [44] is an example of tools used in this stage.

Finally, in production, *runtime* verification methods can be used to detect requirements violations as they happen. These methods require the instrumentation of the application with monitors and probes to measure specific quantities or detect the violation of safety/security predicates [39], [57]. Methods based on partial evaluation compute surrogate model of the system that enable efficient verification at runtime, after current monitoring information are gathered (e.g., [38] verifies probabilistic properties).

*Analysis of Open Challenges:* Broadly speaking, verification requires formal models and analysis algorithms. Model-driven processes exploit human ingenuity to produce semantically richer models of an application and its environment. While these models allow the use of established model checking algorithms, keeping the models consistent with the application code may be challenging in all but the few domains where fully automated code generation is possible. Nonetheless, where available, even partial design models should be used in the future to improve the effectiveness of later-stage methods. This includes, for example, the contextualization of build-time verification within realistic usage profiles specified by the designers, as well as using design models to narrow down the relevant scenarios for diff-time and runtime analysis, reducing the relevant search space to cut verification time (V3). To improve diff-time verification, research has to focus on compositional methods which enable incremental re-analysis of only the changed parts of a codebase [55]. This overall addresses challenge **VE1**. Academic research largely underestimated so far the importance of prompt and localized developer feedback, preferring detailed verification reports produced overnight at land-time. However, empirical evidence from industry suggests that diff-time verification is more effective for bug fixing and keeps developers more engaged [42]. Developing compositional verification algorithms often requires to reduce the expressiveness of verifiable properties, which may nonetheless allow to intercept problems before they reach production, which falls under challenge **VE2**. Adequate design-time models can also help to narrow down the state space to be verified at later process stages, bringing a global view hard to infer from lower level artifacts like code or binaries. Finally, runtime verification methods, whether measurement/probing based or model-based, have the potential of observing the application within its actual execution environment, which may differ from design-time assumptions or land-time simulations. This addresses challenge **VE3**. The ability to promptly detect issue while the application is running can reduce the exposition time to a bug, but also enable automatic adaptation actions to self-protect an application or its infrastructure.

## VII. RUNTIME SERVICE MANAGEMENT

*Context:* Runtime service management particularly concerns dynamic resource scheduling of microservices. Microservices is one of the core DevOps practices. It is a design principle to build applications with fine-grained services. One of the benefits arising from this is the ability to manage each service individually. However, regardless of this benefit, managing microservices at runtime is not trivial. This involves multiple research challenges, regarding monitoring, configuration options, decision making, etc. In a nutshell, the following challenges were highlighted by the researchers:

**RM1** Monitoring microservices
**RM2** Container placement strategy
**RM3** Autoscaling microservices

*Quality-Aware DevOps Research:* **RM1** can be considered as an ensemble of complex sub-problems. Researchers often focus on these sub-problems rather than the overall issue. For example, Noor *et al.* [25] focused on the issue of collecting data from heterogeneous virtualization architecture. They have developed a framework M3 using the SIGAR library and RESTful API that collects both the system and process level metrics through separate agents. On the other hand, Miglierina and Tamburri [26] have focused on reducing the complexity of monitoring configuration management. They proposed Omnia that addresses this issue through Monitoring Configuration as Code. This is realized by defining a set of vocabulary and protocols, which are used to setup and update the monitoring configurations for popular tools like InfluxDB, Prometheus, Grafana, etc.

**RM2** is often addressed based on the scale of the computing environment. For example, in [22], Boza *et al.* proposed kube-scheduler to address RM2 in a typical multi-server setup. kube-scheduler is a performance-aware orchestrator, based on Kubernetes, that take container placement decisions by considering the number of available CPUs in the host

machine and how they effect the runtime and initialization time performance. However, considering a geo-distributed environment like edge or fog, the placement approach needs to be adapted to incorporate information on heterogeneous computing environment. Rossi *et al.* [27] focused on this issue and proposed ge-kube. Along with the placement issue, ge-kube focuses on the elasticity problem as well, thus addressing **RM3**. They resolved the placement issue by formulating it as an optimization problem and the elasticity issue is resolved using model-based Reinforcement Learning (RL).

Recently, compared to RM1 and RM2, **RM3** has gained more attention from the researchers. Multiple autoscalers have been proposed to address this issue, each differing on how the problem is perceived. In [23], Kwan *et al.* propose an autoscaler, HyScale, that focuses on the performance trade-offs between horizontal and vertical scaling. HyScale's principle is to scale vertically if the resources are available. If not, it performs horizontal scaling. Rossi *et al.* [28] also emphasize the use of both horizontal and vertical scaling. However, in contrast to [23], they adopted a model-based approach. It is based on a novel Reinforcement Learning model that relies on approximations (state transition probabilities and the associated costs) from monitoring data. They realized the so-called Elastic Docker Swarm (EDS) by integrating their method with Docker Swarm.

Another context in **RM3** is coordinated scaling to solve the bottleneck shift problem. Bauer *et al.* [29] present Chamulteon that focuses on that issue. It is based on queueing models which are used to forecast system performance and taking a coordinated scaling actions. Chamulteon also includes a workload forecasting component, which makes it proactive. Barna *et al.* [12] propose an Autonomic Management System (AMS) based on Layered Queueing Network (LQN) that inherently offers coordinated autoscaling. However, Chamulteon and AMS both do not consider vertical scaling. In [30], Gias *et al.* present an autoscaler, ATOM, that supports both horizontal and vertical scaling along with coordinated autoscaling. Similar to AMS, ATOM is based on LQN models but it considers both a microservice CPU share (vertical scaling) and the number of its replicas (horizontal scaling) during performance forecasting.

In [24], Qiu et. al present FIRM that focuses on fine grain resource management of microservices considering resources like cache, network bandwidth, CPU, memory, etc. However, unlike most of the approaches, it opted for a model-free method. Their approach relies on a combination of support vector machine and Reinforcement Learning to identify and allocate resources to bottleneck microservices. Rossi *et al.* highlight another important issue - decentralizing the autoscaler components and propose a hierarchical autoscaler me-Kube [31]. Such decentralization makes the autoscaler more scalable when deployed in a large cluster. Although me-Kube uses queueing models, it does not support coordinated scaling as they model each microservice separately rather than the overall application.

**TABLE 7. Comparing different autoscalers for microservices.**

| Ref. \ Attrb. | Model | Vertical | Proactive | Coordinated |
|---|---|---|---|---|
| Kwan et al. [23] | | ✓ | | |
| Rossi et al. [28] | ✓ | ✓ | | |
| Bauer et al. [29] | ✓ | | ✓ | ✓ |
| Barna et al. [12] | ✓ | | | ✓ |
| Gias et al. [30] | ✓ | ✓ | | ✓ |
| Rossi et al. [27] | ✓ | ✓ | | |
| Qiu et. al. [24] | | ✓ | | ✓ |
| Rossi et al. [31] | ✓ | | ✓ | |

A comparison of these autoscalers, based on different attributes (model-based, supporting vertical scaling, proactive, coordinated scaling), are presented in Table 7.

*Analysis of Open Challenges:* Regardless of the progress made, there are still multiple research challenges concerning runtime service management. A major challenge concerning **RM1** is providing support for the model-based approaches. Thus, a monitoring framework for microservices should focus on providing metrics related to queueing or machine learning models, like queue length, arrival rates, transition probabilities, etc., to improve the estimates of different model-based runtime controllers. In addition, they can also leverage machine learning techniques to provide insight of a system architecture such that a model can be automatically generated.

For a runtime controller, focusing on **RM2**, a major challenge is to forecast the performance of container groups rather than a single container. A container group can represent a chain of microservices. Considering a single container alone will only provide a partial view of performance in that particular cluster node. On the other hand, the controllers focusing on **RM3**, particularly the model-based ones, should emphasize faster decision making. This issue can be solved by being proactive but that requires a huge volume of data for accurate forecasting. Thus, researchers should investigate the effectiveness of hybrid autoscalers that combines proactive, simple reactive and model-based reactive approaches.

## VIII. EMERGING TRENDS: BRIDGING AI/ML, BIG DATA, AND DevOps

Artificial Intelligence (AI) and machine learning (ML) algorithms are being increasingly used by industry for monitoring and development to boost performance. These techniques offer the ability to quickly learn the pattern of baseline performance from a large space of performance metrics to diagnose system issues. AI/ML can play a crucial role in accelerating DevOps efficiency for today's dynamic and distributed data-intensive environment. The future of DevOps will be AI, ML, data-intensive driven, which offer potential benefits to enhance functionality and transform how system developers and administrators can design, test, deploy, and maintain systems. Monitoring the modern DevOps environment involves a high level of complexity that AI/ML techniques can alleviate. Dealing with Exabytes of data to investigate the root causes analysis using conventional DevOps solutions

may lead to unexpectedly long time to identify the reason of failures within complex distributed systems.

AI/ML solutions for DevOps are utilized to play a significant role in automating and enhancing DevOps processes. Sun *et al*. [59] propose a non-intrusive automated fault diagnosis for public cloud (such as Amazon Web Service) and rolling upgrade DevOps operation using system logs and machine learning algorithm. They use performance metrics that are collected during monitoring time to train the ML classifier for detecting issues and to expect behavior over every time interval within the system. Their proposed approach achieves on average 90% for recall and precision. The study in [59] demonstrates that using ML for fault diagnoses within DevOps operations (such as rolling upgrade) is promising.

Real challenges for building AIOps solutions are presented by Dang *et al*. [60] based on practices within Microsoft. The term AIOps comes from Gartner to address the challenges of DevOps using AI. Dang *et al*. [60] mention that AIOps is about enabling software and system engineers to operate services efficiently using ML and AI solutions. The added values of AIOps includes: ensuring high service quality, offering high service intelligence, increasing engineering productivity, and decreasing operational cost. Around 60% of firms will adopt AI and ML analytics for DevOps by 2024 to accelerate service delivery, improve performance, and secure systems [60].

Nogueira *et al*. [61] review existent research that applies ML to optimize the quality of process within DevOps pipeline. ML techniques have the ability to provide insight into specific IT processes to effectively assist stakeholders in recognizing improvements that are needed within the software development life cycle. Kumar *et al*. [62] present Sankie, which is an AI Platform for Azure DevOps which is a scalable and general service that is developed to assist and impact all stages of the modern software development life cycle. The proposed AI platform can provide smart and actionable recommendations to system developers and administrators, which include training, recommending, explaining, and evaluating. The proposed platform is used at Microsoft and is enabled for over 50 repositories internally.

There are some DevOps solutions for AI/ML. While AL/ML offers valuable benefits to DevOps, there are some existing DevOps solutions for AL/ML stakeholders that help in developing continuous efficient AL/ML services. Ciucu *et al*. [63] develop a software architecture solution that can ensure the continuous development of computer vision applications. They examine high-performance computing and GPU resource management for model implementation within data centers to enhance the integration process and performance optimization. The integration covers software services and microservices to orchestrate the containers within systems to high availability services.

Palacin *et al*. [13] present a DevOps industrial application that focuses on software quality evaluation tools for tax fraud detection in the context of improving the quality and reliability of Big Data. During development iterations, the impact of quality assessment is reported with a particular focus on the accomplishment of performance requirements during the continuous adding of new functionalities to systems. The authors in [13] target applications that manage billions of invoice records. The evaluation is conducted using simulation (SimTool), which is developed by DICE European project developers for quality analysis. The goal is to reduce the number of DevOps iterations.

**TABLE 8. Taxonomy for AI/ML, Big Data, and DevOps.**

| Ref. \ Target | AI for DevOps | DevOps for AI | Big Data |
|---|---|---|---|
| Sun *et al*. [59] | ✓ | | ✓ |
| Dang *et al*. [60] | ✓ | | ✓ |
| Nogueira *et al*. [61] | ✓ | | |
| Kumar *et al*. [62] | ✓ | | |
| Ciucu *et al*. [63] | | ✓ | |
| Palacin *et al*. [13] | | | ✓ |
| Fox *et al*. [65] | | | ✓ |
| Di Nitto *et al*. [48] | | | ✓ |
| Chen *et al*. [64] | | | ✓ |

Regarding software architecture, Di Nitto *et al*. [48] investigate concerns and obstacles, which are needed to be tackled in DevOps scenarios. The authors in [48] present Specification Quality In DevOps (SQUID), which is a framework for software architecture. The proposed framework is evaluated in the Big Data domain. SQUID is evaluated on a real industrial DevOps scenario, to find SQUID's pros and limitations.

Chen *et al*. [64] contribute to the field of Big Data and DevOps by presenting a methodology revolve around architecture-centric Agile Big data Analytics (AABA), which is evaluated on many Big Data analytics projects in security, cloud-based mobile, healthcare, etc. The authors [64] conclude that architecture agility has a significant impact on the rapid continuous delivery within Big data intensive applications. Finally, it is obvious that AI/ML will play a crucial role in improving DevOps productivity for future dynamic and distributed data-intensive systems. The future of DevOps will be AI, ML, data-intensive driven that offer potential advantages to improve functionality and transform how system developers and administrators can design, test, deploy, and maintain systems.

*Analysis of Open Challenges:* While AI/ML clearly provides valuable benefits to DevOps, there are some potential challenges that may arise in the future. This is because they are fundamentally different from conventional applications, and it is crucial to take into account that they have a different development lifecycle. Another well-known challenge of AI/ML is the availability of sufficient real-world datasets to build, train, and test the model before deploying it into the real production environment. In addition, the characteristic of the system may continuously change, which make the AI/ML model fail to be generalized from datasets that are used for training purposes. Therefore, AI/ML requires continuous model evaluating, tuning, retraining, and retesting.

A team from Microsoft illustrates that the data used in AI systems are large, specific for each context, and complicated for explaining and becoming a burden. These factors

**TABLE 9. Summary of challenges and current contributions.**

| Challenges | Reference | Domain | Scope / Results |
|---|---|---|---|
| AD1 | Balalaie et al. [5] | methodology | Benefits of the microservice architecture and architectural patterns for migration. |
| AD2 | Di Nitto et al. [48] | specification | `SQUID` - A framework for modeling software architectures and their quality properties. |
| | Heinrich et al. [49] | | `iObserve` - An approach to reusing architectural development models during the operation phase. |
| AD3 | Avritzer et al. [19] | configuration | An approach to the scalability assessment of the microservices architecture. |
| | Heinrich et al. [49] | | A framework that enables quality-driven adaptive continuous experimentation. |
| AD4 | Kang et al. [20] | exercise | Solutions to design cloud infrastructure based on containerized microservices. |
| | Chen et al. [51] | | DevOps practices and architecting tactics for developing large-scale systems. |
| | Hasselbring et al. [6] | | A use case of the microservices architecture in e-commerce platforms. |
| | Schneider [7] | | A use case of the microservices architecture in connected cars . |
| MD1 | Gorbenko et al. [33] | analysis | A time-probabilistic failure model for service-oriented distributed systems. |
| | Barna et al. [12] | | A three-layer queueing network for autonomic management systems. |
| | Perez-Palacin et al. [13] | | Petri nets to analyze the performance of data-intensive applications. |
| | Peuster and Karl [14] | | An approach to profiling the performance of an entire service function chain. |
| MD2 | Guerriero et al. [10] | configuration | `SPACE4Cloud` - An framework that enables deployment optimization and resource allocation on clouds. |
| | Suk et al. [32] | | An algorithm for application placement in a high turnover environment. |
| | Sun et al. [11] | | A method for optimizing the parameters of the rolling upgrade procedure. |
| MD3 | Rahman et al. [8] | automation | An analysis of security-related code smells over massive IaC scripts. |
| | Tamburri et al. [9] | | An approach to ensuring the correctness of IaC code through intent modeling. |
| CC1 | Mazkatli and Koziolek [46] | integration | An approach in an agile development process for integrating application models with CI/CD. |
| CC2 | Meissner and Junghanns [40] | | An overview of tools that can be integrated in CI/CD pipelines to continuously meet quality requirements on data. |
| CC3 | Ferry et al. [45] | | Model-driven approach to continuously integrate and orchestrate a system so to ensure security and privacy. |
| TS1 | Schulz et al. [16] | specification | Representative workload models and automatic workload selection based on contextual information. |
| | Schulz et al. [17] | | Specific workload selection for target microservices in load testing. |
| | Schulz et al. [18] | | `BDLT` - Behavior-driven load testing language to describe performance concerns. |
| TS2 | Ferme et al. [34] | automation | Automated testing workflows for CI/CD. |
| | Pietrantuono et al. [35] | | `DevOpRET`- A continuous software reliability testing approach. |
| TS3 | Schermann et al. [21] | execution | Continuous experimentation practices with testing strategies. |
| | Schermann et al. [36] | | `Bifrost` - A living test model with testing strategies and automatic execution. |
| | Dullmann et al. [47] | | Canary releases and A/B testing strategies in CD pipelines. |
| VE1 | Sadowski et al. [42] | development / verification | Diff-time verification methods to include formal methods within code-review processes, providing developers with local and contextualized feedback. |
| VE2 | O'Hearn [55] | verification | Compositional verification to reuse evidence and lemmas computed from previous versions of the application to achieve efficiency via incrementality on new versions |
| VE3 | Filieri et al. [38] | verification / monitoring | Improved verification efficiency at runtime via design-time partial evaluation and pre-processing |
| RM1 | Noor et al. [25] | monitoring | `M3` - A decentralized monitoring framework for heterogeneous cloud. |
| | Miglierina et al. [26] | | `Omnia` - A monitoring configuration approach based on the concepts of IaC. |
| RM2 | Boza et al. [22] | placement | `kube-scheduler` - A performance-aware container orchestrator based on Kubernetes. |
| | Rossi et al. [27] | | `ge-kube` - A container orchestrator for geographically distributed clusters |
| RM3 | Kwan et al. [23] | autoscaling | `HyScale` - A threshold-based autoscaler that considers both horizontal and vertical scaling. |
| | Rossi et al. [28] | | `EDS` - An autoscaler based on model-based RL for Docker Swarm. |
| | Gias et al. [30] | | `ATOM` - An LQN-based autoscaler that aims to optimize the system throughput. |
| | Qiu et. al. [24] | | `FIRM` - An autosaler supporting fine-grained resource management using model-free RL. |
| | Bauer et al. [29] | | `Chamulteon` - A queueing model based horizontal autoscaler that includes workload forecasting. |
| | Barna et al. [12] | | `AMS` - A horizontal autoscaler for containerized applications based on LQN. |
| | Rossi et al. [31] | | `me-kube` - A hierarchical autoscaler based on model-free RL. |

make it challenging to integrate AI model on a large scale and distributed system. Therefore, system engineers need to carefully collect and preprocess datasets before training and tuning AI algorithms to gain high accuracy performance. In addition, the collected data has to be efficiently stored and updated continuously with a predefined schema. Another challenge is that datasets' schema may frequently change in a real-time, which need to be resilient with continuous

developments when new data is ingested from large scale systems with changeable performance characteristics [66].

When developing a large scale DevOps project, it is challenging to maintain modularity. This is because the AI models are developed in a separated and isolated environment to ensure the prevention of interference among systems components. These separated AI subsystems are developed by different engineering teams, where AI services communicate with other systems in non-obvious ways using a controlled API that has to be precisely controlled [67], [68]. This kind of challenge may cause error to be propagated among system and impact the overall performance of services. There is a need for a more advanced researches and effective solutions to continuous update of AI models and discover the unseen misconceptions among system components while taring data characteristics are changing.

## IX. CONCLUSION

DevOps methods have reduced the cultural and methodological gap between developers and operators, which lead to the formation of many new organizational structures, such as virtual teams working on both development and operation tasks. This motivates the establishment of new professional figures, often referred to as DevOps engineers, who center their activity on tooling and automation across the whole application lifecycle. DevOps paradigm allocates more attention towards continuous re-release, unified tooling and organizational processes. Common DevOps advances include, for example, continuous-integration/continuous-delivery (CI/CD) pipelines and highly-automated orchestration solutions for the run-time environment.

This survey reviews recent research to support DevOps with quality-aware software engineering tools. The paper reviews the context in which research was carried out and reveals some gaps in areas such as continuous-integration/continuous-delivery (CI/CD), incremental verification, and infrastructure-as-code (IaC). Table 9 provides a summary of challenges and current contributions with the domain of quality-aware DevOps. Initial activity on the upcoming *AI for DevOps* and *DevOps for AI software* as also been surveyed, outlining possible directions for further work.

## ACKNOWLEDGMENT

*(Ahmad Alnafessah, Alim Ul Gias, Runan Wang, and Lulai Zhu contributed equally to this work.)*

## REFERENCES

[1] M. Virmani, "Understanding DevOps & bridging the gap from continuous integration to continuous delivery," in *Proc. INTECH*, May 2015, pp. 78–82. [Online]. Available: http://ieeexplore.ieee.org/document/7173368/

[2] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in *Proc. ASE*. Urbana, IL, USA: IEEE, Oct. 2017, pp. 60–71. [Online]. Available: http://ieeexplore.ieee.org/document/8115619/

[3] J. Wettinger, U. Breitenbücher, and F. Leymann, "Devopslang-bridging the gap between development and operations," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.* Berlin, Germany: Springer, Sep. 2014, pp. 108–122.

[4] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps—A Software Architect's Perspective* (SEI Series in Software Engineering). Reading, MA, USA: Addison-Wesley, 2015. [Online]. Available: http://bookshop.pearson.de/devops.html?productid=208463

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May 2016.

[6] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *Proc. ICSAW*, Apr. 2017, pp. 243–246.

[7] T. Schneider, "Achieving cloud scalability with microservices and DevOps in the connected car domain," in *Proc. Soft. Eng. CEUR*, 2016, pp. 138–141.

[8] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proc. ICSE*, 2019, pp. 164–175.

[9] D. A. Tamburri, W.-J. Van den Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski, "TOSCA-based intent modelling: Goal-modelling for infrastructure-as-code," *Softw.-Intensive Cyber-Phys. Syst.*, vol. 34, nos. 2–3, pp. 163–172, Jun. 2019.

[10] M. Guerriero, M. Ciavotta, G. P. Gibilisco, and D. Ardagna, "A model-driven DevOps framework for QoS-aware cloud applications," in *Proc. SYNASC*, Sep. 2015, pp. 345–351.

[11] D. Sun, S. Chen, G. Li, Y. Zhang, and M. Atif, "Multi-objective optimisation of online distributed software update for DevOps in clouds," *ACM Trans. Internet Technol.*, vol. 19, no. 3, pp. 1–20, Nov. 2019.

[12] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable DevOps," in *Proc. SEAMS*, May 2017, pp. 65–75.

[13] D. Perez-Palacin, Y. Ridene, and J. Merseguer, "Quality assessment in DevOps: Automated analysis of a tax fraud detection system," in *Proc. 8th ACM/SPEC ICPE*, 2017, pp. 133–138.

[14] M. Peuster and H. Karl, "Profile your chains, not functions: Automated network service profiling in DevOps environments," in *Proc. NFV-SDN*, 2017, pp. 1–6.

[15] F. Boyer, X. Etchevers, N. De Palma, and X. Tao, "Architecture-based automated updates of distributed microservices," in *Proc. Int. Conf. Service-Oriented Comput.* Cham, Switzerland: Springer, Nov. 2018, pp. 21–36.

[16] H. Schulz, T. Angerstein, and A. van Hoorn, "Towards automating representative load testing in continuous software engineering," in *Proc. ICPE (Companion)*, 2018, pp. 123–126.

[17] H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn, "Microservice-tailored generation of session-based workload models for representative load testing," in *Proc. MASCOTS*, 2019, pp. 323–335.

[18] H. Schulz, D. Okanović, A. van Hoorn, V. Ferme, and C. Pautasso, "Behavior-driven load testing using contextual knowledge-approach and experiences," in *Proc. ICPE*, 2019, pp. 265–272.

[19] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. V. Hoorn, H. Schulz, D. Menasché, and V. Rufino, "Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests," *J. Syst. Softw.*, vol. 165, Jul. 2020, Art. no. 110564.

[20] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in *Proc. IC2E*, 2016, pp. 202–211.

[21] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall, "We're doing it live: A multi-method empirical study on continuous experimentation," *Inf. Softw. Technol.*, vol. 99, pp. 41–57, Jul. 2018.

[22] E. F. Boza, C. L. Abad, S. P. Narayanan, B. Balasubramanian, and M. Jang, "A case for performance-aware deployment of containers," in *Proc. WOC*, 2019, pp. 25–30.

[23] A. Kwan, J. Wong, H. Jacobsen, and V. Muthusamy, "HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. ICDCS*, 2019, pp. 80–90.

[24] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. OSDI*. Berkeley, CA, USA: USENIX, 2020, pp. 805–825.

[25] A. Noor, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar, "A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments," in *Proc. CLOUD*, 2019, pp. 156–163.

[26] M. Miglierina and D. A. Tamburri, "Towards omnia: A monitoring factory for quality-aware DevOps," in *Proc. ICPE ACM/SPEC*, 2017, pp. 145–150.

[27] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Comput. Commun.*, vol. 159, pp. 161–174, Jun. 2020.

[28] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. CLOUD*, 2019, pp. 329–338.
[29] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. ICDCS*, 2019, pp. 2015–2025.
[30] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. ICDCS*, Jul. 2019, pp. 1994–2004.
[31] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *Proc. ACSOS*, 2020, pp. 28–37.
[32] T. Suk, J. Hwang, M. F. Bulut, and Z. Zeng, "Failure-aware application placement modeling and optimization in high turnover DevOps environment," in *Proc. CLOUD*, 2019, pp. 115–123.
[33] A. Gorbenko, A. Romanovsky, and O. Tarasyuk, "Fault tolerant Internet computing: Benchmarking and modelling trade-offs between availability, latency and consistency," *J. Netw. Comput. Appl.*, vol. 146, Nov. 2019, Art. no. 102412.
[34] V. Ferme and C. Pautasso, "Towards holistic continuous software performance assessment," in *Proc. ICPE (Companion)*, 2017, pp. 159–164.
[35] R. Pietrantuono, A. Bertolino, G. De Angelis, B. Miranda, and S. Russo, "Towards continuous software reliability testing in DevOps," in *Proc. AST*, 2019, pp. 21–27.
[36] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proc. ACM Middleware*, 2016, pp. 1–14.
[37] J. Backes, C. Varming, M. Whalen, P. Bolignano, B. Cook, A. Gacek, K. S. Luckow, N. Rungta, M. Schaef, C. Schlesinger, and R. Tanash, "One-click formal methods," *IEEE Softw.*, vol. 36, no. 6, pp. 61–65, Nov. 2019.
[38] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting self-adaptation via quantitative verification and sensitivity analysis at run time," *IEEE Trans. Softw. Eng.*, vol. 42, no. 1, pp. 75–99, Jan. 2016.
[39] N. Beigi-Mohammadi, M. Litoiu, M. Emami-Taba, L. Tahvildari, M. Fokaefs, E. Merlo, and I. V. Onut, "A DevOps framework for quality-driven self-protection in Web software systems," in *Proc. CASCON*, 2018, pp. 270–274.
[40] R. Meissner and K. Junghanns, "Using DevOps principles to continuously monitor RDF data quality," in *Proc. SEMANTiCS*. Leipzig, Germany: ACM, 2016, pp. 189–192. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2993318.2993351
[41] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. PLDI*, 2007, pp. 89–100.
[42] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018, doi: 10.1145/3188720.
[43] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proc. SCAM*, 2018, pp. 1–23.
[44] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Softw.*, vol. 21, no. 3, pp. 92–100, May 2004.
[45] N. Ferry, P. H. Nguyen, H. Song, E. Rios, E. Iturbe, S. Martinez, and A. Rego, "Continuous deployment of trustworthy smart IoT systems," *J. Object Technol*, vol. 19, no. 2, pp. 16:1–23:1, 2020.
[46] M. Mazkatli and A. Koziolek, "Continuous integration of performance model," in *Proc. ICPE (Companion)*. Berlin Germany: ACM, Apr. 2018, pp. 153–158. [Online]. Available: https://dl.acm.org/doi/10.1145/3185768.3186285
[47] T. F. Düllmann, C. Paule, and A. van Hoorn, "Exploiting DevOps practices for dependable and secure continuous delivery pipelines," in *Proc. RCoSE*, 2018, pp. 27–30.
[48] E. Di Nitto, P. Jamshidi, M. Guerriero, I. Spais, and D. A. Tamburri, "A software architecture framework for quality-aware DevOps," in *Proc. QUDOS*, 2016, pp. 12–17.
[49] R. Heinrich, R. Jung, C. Zirkelbach, W. Hasselbring, and R. Reussner, "An architectural model-based approach to quality-aware DevOps in cloud applications," in *Software Architecture for Big Data and the Cloud*. Amsterdam, The Netherlands: Elsevier, 2017, pp. 69–89.
[50] M. Jiménez, L. F. Rivera, N. M. Villegas, G. Tamura, H. A. Müller, and N. Bencomo, "An architectural framework for quality-driven adaptive continuous experimentation," in *Proc. RCoSE/DDrEE*, 2019, pp. 20–23.
[51] H.-M. Chen, R. Kazman, S. Haziyev, V. Kropov, and D. Chtchourov, "Architectural support for DevOps in a neo-metropolis BDaaS platform," in *Proc. SRDSW*, 2015, pp. 25–30.
[52] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, Apr. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584918302507
[53] L. Chen, "Continuous delivery: Overcoming adoption challenges," *J. Syst. Softw.*, vol. 128, pp. 72–86, Jun. 2017.
[54] M. Shahin, M. A. Babar, and L. Zhu, "The intersection of continuous deployment and architecting process: Practitioners' perspectives," in *Proc. ESEM*. Ciudad Real, Spain: ACM, 2016, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2961111.2962587
[55] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proc. LICS*, 2018, pp. 13–25, doi: 10.1145/3209108.3209109.
[56] H. Koziolek and R. Reussner, "A model transformation from the palladio component model to layered queueing networks," in *Proc. SPEC Int. Perform. Eval. Workshop*. Berlin, Germany: Springer, Jun. 2008, pp. 58–78.
[57] A. Brunnert *et al.*, "Performance-oriented DevOps: A research agenda," 2015, *arXiv:1508.04752*. [Online]. Available: https://arxiv.org/abs/1508.04752
[58] E. Di Nitto, P. Jamshidi, M. Guerriero, I. Spais, and D. A. Tamburri, "A software architecture framework for quality-aware DevOps," in *Proc. QUDOS*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 12–17, doi: 10.1145/2945408.2945411.
[59] D. Sun, M. Fu, L. Zhu, G. Li, and Q. Lu, "Non-intrusive anomaly detection with streaming performance metrics and logs for DevOps in public clouds: A case study in AWS," *IEEE Trans. Emerg. Topics Comput.*, vol. 4, no. 2, pp. 278–289, Apr. 2016.
[60] Y. Dang, Q. Lin, and P. Huang, "AIOPS: Real-world challenges and research innovations," in *Proc. IEEE/ACM ICSE (Companion)*, 2019, pp. 4–5.
[61] A. F. Nogueira, J. C. Ribeiro, M. Zenha-Rela, and A. Craske, "Improving La Redoute's CI/CD pipeline and DevOps processes by applying machine learning techniques," in *Proc. QUATIC*, 2018, pp. 282–286.
[62] R. Kumar, C. Bansal, C. Maddila, N. Sharma, S. Martelock, and R. Bhargava, "Building Sankie: An AI platform for DevOps," in *Proc. IEEE/ACM BotSE*, 2019, pp. 48–53.
[63] R. Ciucu, F. C. Adochiei, I.-R. Adochiei, F. Argatu, G. C. Seriţan, B. Enache, S. Grigorescu, and V. V. Argatu, "Innovative DevOps for artificial intelligence," *Sci. Bull. Electr. Eng. Fac.*, vol. 19, no. 1, pp. 58–63, Apr. 2019.
[64] H.-M. Chen, R. Kazman, and S. Haziyev, "Agile big data analytics development: An architecture-centric approach," in *Proc. HICSS*, 2016, pp. 5378–5387.
[65] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow, "HPC-ABDS high performance computing enhanced apache big data stack," in *Proc. IEEE/ACM CCGRID*, Jun. 2015, pp. 1057–1066.
[66] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, Jun. 2019, pp. 291–300.
[67] C. R. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "Sometimes you need to see through walls: A field study of application programming interfaces," in *Proc. ACM Conf. Comput. Supported Cooperat. Work*, 2004, pp. 63–71.
[68] C. R. De Souza, D. Redmiles, and P. Dourish, "'Breaking the code' moving between private and public work in collaborative software development," in *Proc. Int. ACM SIGGROUP Conf. Supporting Group Work*, 2003, pp. 105–114.

**AHMAD ALNAFESSAH** is currently pursuing the Ph.D. degree with the Department of Computing, Imperial College London. He was a Senior Academic Researcher with the National Centre for AI and Big Data Technologies, KACST, from 2012 to 2017. His research focuses on performance engineering for big data systems and AI, with a specific focus on in-memory platforms. He is interested in big data systems, AI, IoT, HPC, complex distributed systems, and cloud computing.

**ALIM UL GIAS** received the bachelor's degree in information technology (major in software engineering) from the Institute of Information Technology (IIT), University of Dhaka (DU), and the M.Sc. degree in software engineering from IIT, DU, with thesis on adaptive software performance testing. He is currently pursuing the Ph.D. degree with Imperial College London, U.K. His current research interest includes software performance engineering using stochastic process modeling and machine learning.

**RUNAN WANG** received the B.Eng. and M.Sc. degrees in software engineering from the Beijing Institute of Technology (BIT), China, in 2017 and 2019, respectively. She is currently pursuing the Ph.D. degree with the Department of Computing, Imperial College London. Her current research focuses on performance models, program engineering, and machine learning.

**LULAI ZHU** received the B.Eng. and M.Eng. degrees in measurement technology and instruments from Sichuan University, China, in 2008 and 2011, respectively, and the M.Sc. degree in computing science from Imperial College London, U.K., in 2016. He is currently pursuing the Ph.D. degree with the Department of Computing. He is currently an RA with the Department of Computing. His current research focuses on performance models, such as queuing networks and Petri nets, and their applications to distributed systems and cloud computing.

**GIULIANO CASALE** joined the Department of Computing, 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, U.K., 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 articles. 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. He was a recipient of multiple awards, recently the Best Paper Award at ACM SIGMETRICS in 2017. He serves on the Editorial Board of IEEE Transactions on Network and Service Management (TNSM) and ACM TOMPECS and as the Chair of ACM SIGMETRICS.

**ANTONIO FILIERI** is an Assistant Professor with Imperial College London, U.K. His main interests are in the application of mathematical methods for software engineering, in particular, probability, statistics, logic, and control theory. His recent work includes exact and approximate methods for probabilistic symbolic execution, incremental verification, quantitative software modeling and verification at runtime, and control-theoretical software adaptation. www.antonio.filieri.name.

IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received July 13, 2020, accepted July 27, 2020, date of publication August 10, 2020, date of current version August 20, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3015346*

# TRACK-Plus: Optimizing Artificial Neural Networks for Hybrid Anomaly Detection in Data Streaming Systems

**AHMAD ALNAFESSAH[1,2], (Member, IEEE), AND GIULIANO CASALE[1], (Member, IEEE)**
[1]Department of Computing, Imperial College London, London SW7 2AZ, U.K.
[2]National Center for Artificial Intelligence and Big Data Technologies, King Abdulaziz City for Science and Technology (KACST), Riyadh 12354, Saudi Arabia

Corresponding author: Ahmad Alnafessah (a.alnafessah17@imperial.ac.uk)

This work was supported in part by the National Center for Artificial Intelligence and Big Data Technologies, King Abdulaziz City for Science and Technology (KACST), Saudi Arabia, and in part by the European Union's Horizon 2020 Research and Innovation Program, RADON, under Grant 825040.

**ABSTRACT** Software applications can feature intrinsic variability in their execution time due to interference from other applications or software contention from other users, which may lead to unexpectedly long running times and anomalous performance. There is thus a need for effective automated performance anomaly detection methods that can be used within production environments to avoid any late detection of unexpected degradations of service level. To address this challenge, we introduce TRACK-Plus a black-box training methodology for performance anomaly detection. The method uses an artificial neural networks-driven methodology and Bayesian Optimization to identify anomalous performance and are validated on Apache Spark Streaming. TRACK-Plus has been extensively validated using a real Apache Spark Streaming system and achieve a high F-score while simultaneously reducing training time by 80% compared to efficiently detect anomalies.

**INDEX TERMS** Apache Spark, artificial intelligence, big data, machine learning, neural networks, performance anomalies.

## I. INTRODUCTION

In-memory processing technologies used for Big Data have been widely adopted in industry, in particular, Apache Spark has drawn particular attention because of its speed, generality, and ease of use. Here, we consider Apache Spark-based streaming workloads in which analytic operations are applied by means of resilient distributed datasets (RDDs). Our goal is to develop automated techniques to support performance anomaly detection. Although our focus is on a particular platform, elements of this approach may be exploited in the context of other stream processing systems.

Artificial Intelligence and machine learning algorithms are being increasingly used by researchers for performance anomaly identification and diagnosis [1]–[4]. Moreover, machine learning classification techniques are widely used to classify inputs based on their features into predefined classes to build a classifier that can predict the class of each item according to class labels. Popular classification techniques for performance anomaly detection include neural networks, support vector machines (SVMs) [5], and Bayesian networks [6].

The attention for anomaly detection is motivated by the fact that, with the growing complexity of Big Data and cloud systems, service-level management requires significantly higher levels of automation and attention [7]. An *anomaly* is defined as an abnormal behavior during the execution of a program. It could arise due to resource contention or service level disruptions, among several other factors. While some studies address the challenges of performance anomaly detection for batch processing [4], [8], [9], there is a demand for effective automated performance anomaly detection solutions specifically built for industrial-strength streaming systems, such as Apache Spark. This is because the platform does not natively report in log files either root causes of abnormal Spark tasks or information about when anomalous scenarios happen within the cluster [10]. Therefore, a practical solution is needed that can efficiently train a machine learning model

The associate editor coordinating the review of this manuscript and approving it for publication was Youqing Wang.

to identify performance anomalies within streaming workloads in production environments to produce such reports automatically.

This work is also motivated by the difficulty of carrying out anomaly detection within Big Data streaming systems, especially for time-varying workloads and critical applications. Apache Spark has more than 200 configurable parameters, and some parameters may depend on each other and affect the overall platform performance [11]. This large and complex configurable parameter space makes it difficult even for expert administrators to detect and classify anomalous performance within Spark Streaming clusters, as some performance level may simply depend on the chosen configuration. Therefore, the interaction between the performance of in-memory processing technologies and their configuration needs to be characterized in order to pinpoint and diagnose the root causes of anomalies, a classification task for which artificial intelligence methods are naturally well-suited.

This paper addresses the challenge of anomaly identification by investigating agile hybrid learning techniques for anomaly detection. We describe TRACK (neural neTwoRk Anomaly deteCtion in sparK) and TRACK-Plus, two methods to efficiently train a class of machine learning models for performance anomaly detection using a fixed number of experiments. TRACK revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve high accuracy in a short period of time. TRACK-Plus is an automated fine-grained anomaly detection solution that adds to track a second Bayesian Optimization cycle for fine-tuning the hyperparameters of artificial neural network configuration. The objective is to accelerate the search process for optimizing neural network configurations and improving the performance of anomaly classification.

A validation based on several datasets from a real Apache Spark Streaming system is performed to demonstrate that the proposed methodology can efficiently identify performance anomalies, near-optimal configuration parameters, and a near-optimal training dataset size while reducing the number of experiments. Our results indicate that the reduction in experimental need can be up to 75% compared to naïve anomaly detection training. To the best of our knowledge, this paper is among the very first works that provide a comprehensive methodology for both performance anomaly classification and the efficient optimization of artificial neural networks to detect anomalies within streaming systems.

This paper extends a preliminary abstract in [12] by providing a comprehensive evaluation and classification model for three anomalous Spark Streaming workloads. In addition, the proposed methodology has also been enhanced by developing TRACK-Plus to simultaneously configure the artificial neural network used for anomaly detection. Our core contributions in this paper are as follows:

- Providing an updated discussion of existing anomaly detection techniques and algorithms that should be further researched by the community invested in this challenging problem space.
- Conducting a comparative analysis of four well-known anomaly detection techniques and algorithms to help system administrators in choosing the appropriate anomaly detection mechanisms for their in-memory Spark Streaming Big Data system.
- Addressing the challenge of anomaly identification and classification by investigating new hybrid learning techniques for anomaly detection in Spark Streaming Big Data systems.
- Presenting a comprehensive methodology to automate the search for the ideal dataset size with which to train the detection model and automate the tuning of neural networks hyperparameters to identify the most efficient network architecture and configuration.

The rest of the paper is organized as follows: A motivating example is comprehensively discussed in Section III, then the prerequisite background information about Apache Spark is presented in Sections IV. The proposed methodology of this work is presented in Section V, followed by a systematic evaluation in Section VI, and the results are discussed in Section VII. Related work is overviewed in Sections II. Finally, the VIII section presents a discussion and conclusions.

## II. RELATED WORK

Performance anomaly detection techniques are important to optimize service levels in Big Data applications and large-scale distributed systems. Although the root cause of bottleneck and anomalous performance is often CPU congestion [13]–[15], Big data workloads are often also cache, memory, and network intensive, requiring advance techniques for their identification and mitigation.

Fulp *et al.* [5] use a machine learning approach to detect and predict the likelihood of service level disruptions using an SVM based on information from Linux system log files. They examine a dataset that contains over 24 months of actual file logs from a cluster with 1024 computing nodes. Their proposed solution achieves an acceptable level of classification performance of 73%. Fulp *et al.* [5], however, consider only one type of system failure —hard disk failure— without examining other common sources of systems failure, including CPU, cache, etc. Although SVM models are effective at making decisions from well-behaved feature vectors, they can be more expensive for modeling variations in large datasets and high-dimensional input features [16]–[18].

Qi *et al.* [8] propose a white-box model that uses classification and regression trees to analyze straggler root causes. The authors use raw metrics from Apache Spark logs and hardware sampling tools to train their model. The conventional decision tree algorithm has a drawback, however, which is the issue of overfitting. To avoid this issue, the authors use a special type of tree called a CART tree (*classification and*

*regression tree*) which offered some mitigation solutions. The solution includes a pruning technique (called CCP) when the tree growth is completed. The pruning process continues for several iterations and the classification performance metrics are checked for each node and its leaves [8]. Such a process is time-consuming, especially with intensive data streaming systems, so this study does not consider it. In addition, the presented work in [8] does not cover streaming processing workloads.

Lin *et al.* [19] propose an anomaly detection technique for infrastructure as a service (IaaS) cloud computing environment using local outlier factor (LoF) algorithm to detect anomalies by analyzing the reduced performance feature dataset. LOF is used to assign a score for each group of performance metrics to assess the system behavior, where the behavior is considered an anomalous if the score exceeds the predefined threshold. The authors validate their technique within a private cloud computing system that is built using OpenStack and Xen open-source software. Their result shows that the proposed technique outperforms principal components analysis (PCA).

Huang *et al.* [20] use an adaptive local outlier factor (LOF), a type of neighbor-based technique, for an anomaly detection scheme in cloud systems. They argue that their scheme could learn application behaviors in both training and detecting time. In addition, the scheme is adaptive to changes during the detection phase, which potentially significantly reduces the effort to collect the training dataset before the detection phase. The experimental results in [20] show that their scheme could detect performance anomalies with a low level of computational overhead. However, using the basic LOF requires considerable effort to collect enough datasets of normal behavior and requires intensive computations to calculate the distance scores of each instance during the test phase. According to [18], it is challenging to compute distance measurements for complex data, and such computations cannot identify some performance anomalies. Therefore, it is essential to keep in mind that the LOF needs to be adapted to be used with Spark Streaming for anomaly detection.

Table 1 further shows a summary of anomaly detection techniques used in the context of cloud and distributed computing systems. Further advancement in hybrid solutions holds great potential for anomaly identification systems [21], [22]. Some performance anomaly identification studies and surveys have been conducted in the literature for different purposes [14], [17], [23], [24]; however, there is still a shortage of studies that propose efficient automated anomaly detection, especially for in-memory Big Data stream processing technologies as we study in the next sections.

## III. MOTIVATING EXAMPLE

In this section, we briefly illustrate the problem area and the benefits of Bayesian Optimization for anomaly detection. We have developed the customized benchmark *Network WordCountExp* for stream processing systems to generate our dataset for training purposes, more details are given in Section VI-B. Messages are sent to the data stream processing system with a fixed rate per second and number of lines per message. The inter-arrival time of messages is exponentially distributed. The Spark system is monitored at all times and we consider different levels of logging, ranging from measurements of Spark Streaming jobs to full recording of tasks execution logs, more details about Spark logging may be found in [41].

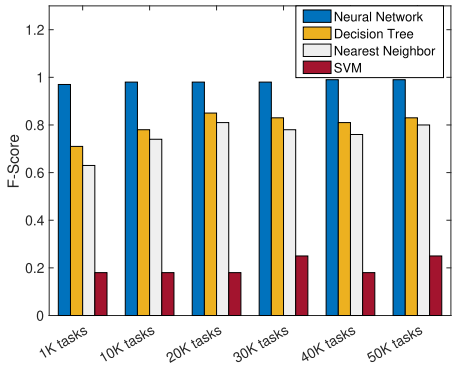
A detailed comparison is shown in Figure 1(a). The figure depicts the impact of Spark workload size, in terms of the number of tasks within the workload, on the neural networks model and comparison with other three well-known algorithms, namely nearest neighbor, decision tree, and support vector machine (SVM). The F-score metric is used to evaluate the accuracy of the neural network. Six training workload sizes with the same configurations are examined for sensitivity analysis, namely: 1000, 10000, 20000, 30000, 40000, and 50000 Spark Streaming tasks. From Figure 1(a), we see that the neural networks model outperforms all the other algorithms, achieving 98% F-score on average for the six different workload sizes. In comparison, the other methods feature a F-score on average 0.8% for decision tree, 0.75% for nearest neighbor, 0.2% for SVM. The computational complexity of neural networks depends on architecture of network, e.g. number of input features, number of layers, size of layer, etc. The complexity of the proposed neural networks is $O(m * N^{\frac{3}{2}})$, where $m$ is number of iterations [42]. In terms of execution time, the neural networks, decision tree, nearest neighbor, and SVM took approximately 1 min, 2 min, 5 min, and 21 min, respectively. This example suggests that neural networks tend to be more effective than other AI/ML methods for anomaly detection within the Spark Streaming system, which motivates our interest to examine and train these models in streaming context.

We now examine an anomaly detection model that is trained using another neural network model. This model is trained with a single Spark Streaming workload configuration (with a *rate* of 2 *message/sec* and a *size* of 1000 *line/message*) by testing it against two unseen streaming workload configurations without injecting any anomalies. The first workload has a *rate* of 11 *message/sec* and a *size* of 1000 *line/message*, and the model achieves a 98% F-score. The second workload has a *rate* of 2 *message/sec* with a *size* of 5000 *line/message*, and the neural network model achieves a 98% F-score. These experiments demonstrate that the performance of the neural networks model is robust and unaffected by changes to streaming workload configurations when there are no anomalies.

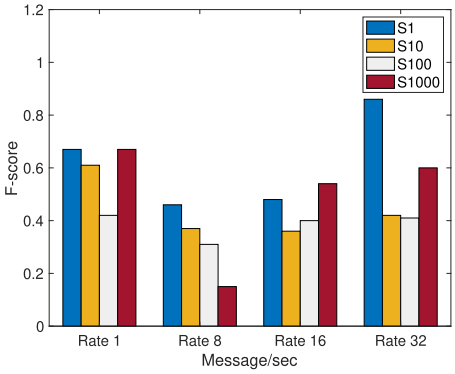
The same neural network is now trained on a single Spark Streaming workload configuration (with a *rate* of 2 *message/sec* and *size* of 1000 *line/message*) is typically used for some selected possible parameters of streaming workload configurations for *Sizes* 1, 10, 100, and 1000 *line/message* and *rates* of 1, 2, 4, 8, 16, and 32 *message/sec*, with artificial CPU anomalies injected. The F-score performance of the

**TABLE 1. Summary of the state-of-the-art anomaly detection techniques.**

| Reference | Approach | Detection Technique | System/Environment |
|---|---|---|---|
| Magalhaes et al. (2010) [25] | Statistical | Correlation techniques and time-series alignment algorithms to spot the relationship between the transactions response time and the workload to pinpoint the occurrence of anomalies for dynamic workloads | Web-based and component-based applications applications |
| Zhang et al. (2007) [26] | Statistical | Regression-based model for estimating CPU demands by different client transactions. The result of the regression method is used to parameterize an analytic model of queues | Enterprise e-commerce system and TPC-W benchmark [27] |
| Kelly (2005) [28] | Statistical | Simple queueing-theoretic observations with standard optimization methods for performance anomaly detection | Distributed commercial systems that serve real customers |
| Yang et al. (2007) [29] | Statistical and Signal Processing | Extending the traditional window-based strategy by using signal-processing techniques to filter out recurring, background variations to determine which resource is the probable cause of an anomalous performance in a system | Three Grid environment applications: Cactus, GridFTP, and a Sweep3d |
| Lu et al. (2017) [10] | Statistical | Off-line approach to detect abnormal Spark tasks and analyze the root causes based on statistical spatial-temporal analysis. The mean and standard deviation of all tasks in each stage are used to get information about macro-awareness on the task's execution time | Private Apache Spark cluster and SparkBench [30] |
| Garraghan et al. (2016) [15] | Statistical | Empirical analysis for straggler detection and root-cause for batch processes using a combination of offline execution patterns modeling and online analytic agents for monitoring | Virtualized Cloud data centers |
| Bodik et al. (2010) [31] | Hybrid | Logistic regression to select a set of relevant metric to minimize both the prediction errors and quantile to summarize the values of each performance metric across all the application servers | Enterprise cloud computing and web applications |
| Chen et al. (2002) [32] | Hybrid | Using data mining and statistical techniques to correlate the normal and failure requests to pinpoint faults | PetStore, which is e-commerce environment based on the Java 2 Platform Enterprise Edition (J2EE) for Web and distributed systems. |
| Cohen et al. (2005) [33] | Machine learning | Pattern recognition, clustering, information retrieval, and Tree-Augmented Naive Bayes models (TAN) are used to characterize each metric and its contribution | Two distributed applications, one runs synthetic workloads in a private system and the other one run real customer services in a globally-distributed production environment |
| Fulp et al. (2008) [5] | Machine learning | Supervised detection and prediction of the hard disk failure using an SVM | Cluster with 1024 computing nodes |
| Pannu et al. (2012) [34] | Machine learning | Supervised one-class SVM and SVM for self-evolving anomaly identification and prediction | Utility cloud computing systems |
| Baek et al. (2017) [3] | Machine learning | Unsupervised labeling by clustering the training data set to create referential labels and supervised anomaly detection model that includes Naive Bayes, Adaboosting, SVM, and Random Forest | Enterprise and cloud computing systems |
| Ren et al (2018) [35] | Machine learning | Anomaly detection approach based on stage-task behaviors that related to the task execution status to classify normal and abnormal workloads according to the offline logistic regression model for each batch | Homogeneous Apache Spark Yarn Cluster for streaming workload with BigdataBench benchmark [36] |
| Lu et al (2018) [4] | Machine Learning | Anomaly detection using convolutional neural networks based model, which is implemented with different filters to automatically train model on the relationships among events | Big Data system logs using Hadoop distributed file system |
| Qi et al. (2017) [8] | Machine learning | White-box model for root-cause analysis of performance bottleneck and straggler based on CART decision tree | Private Apache Spark Cluster with HiBench benchmark to generate workloads |
| Yadwadkar et al. (2012) [37] | Machine Learning | Regression decision tree model periodically learns correlations between node level status and task execution time | Trace from Facebook Hadoop system and Berkeley EECS department's local Hadoop cluster (icluster) |
| Tan et al. (2011) [38] | Machine Learning | Semi-supervised one-class anomaly detection for streaming data using random half space trees | Open source datasets from KDD Cup 99 dataset [39] |
| Lin et al. (2011) [40] | Machine Learning | principal components analysis (PCA) for anomaly detection | Linux OS and high-speed network |




(a) Comparison and sensitivity analysis of Spark workload sizes (the number of Spark tasks) on neural networks, decision tree, nearest neighbor, and SVM for CPU anomaly detection in Spark Streaming workloads.




(b) F-score for anomaly detection using neural network with *rates* 1, 8, 16, and 32 and *sizes* 1, 10 ,100, and 1000.

**FIGURE 1. Motivation examples for TRACK. Figure 1(b) depicts a sensitivity analysis of four algorithms. Figure 1(a) shows the performance variance of neural networks with adjusted workload configuration parameters.**

anomaly detection model dramatically decreases to a small figure between 0.1% and 3%. It is clear that the neural networks model fails to detect the CPU anomalies when the streaming workload configuration is changed. Therefore, the model requires additional training with more possible configuration parameters to detect anomalies. This baseline experiment demonstrates the critical need for a solution that would find the optimal dataset size and configuration parameters of a streaming workload for training the anomaly detection model within an in-memory Big Data system for generalization purposes.

Figure 1(b) shows some design factors and response variables (F-scores) for different streaming workload configurations where the proposed neural network model is trained using a single combination of configurations parameters (e.g., rate $r$ and size $s$) and tested against other workloads stream configurations, which includes *rates* 1, 8, 16, and 32 *message/sec* and *sizes* 1, 10, 100, and 1000 *line/message*. As can be seen from Figure 1(b), it is not apparent which set of workload configurations would efficiently train the machine learning model to achieve the highest accuracy in a given time. The goal of this paper is to address the problem of joint optimization of neural network and experimental training.

## IV. BACKGROUND INFORMATION

The following subsections briefly describe required background on Apache Spark Streaming, Bayesian Optimization, and neural networks.

### A. APACHE SPARK STREAMING

Apache Spark stream processing has gained traction for a wide range of data processing applications in Big Data systems because of its ease of use, fault tolerance of stream data processing, and suitability of integration with other batch processing systems. Stream data can be ingested from many streaming sources to be processed and used by other systems [43]. Spark Streaming operates in a way that divides the entire received data stream into batches to be processed by the main Spark engine. The final data, i.e., the processed results, will consequently be segmented in batches. The input data stream can be fed from many different sources (Kafka, Flume, Twitter, etc.), and the stream data can be processed by advanced Spark libraries for machine learning and graph processing algorithms. The final output data from Spark Streaming can then be pushed out to databases or other systems [43].

Inside the Spark system, live stream data is fed to the Spark Streaming system. Spark Streaming divides the streaming workload into many batch workloads, which are then passed as inputs into the Spark core engine for data processing. In Spark Streaming, high-level basic abstractions are called discretized streams (*DStreams*) and are continuous streams of data. Each *DStream* is either an input data stream received from other streaming sources or it is the result of a processed data stream created from the input streams [43].

Internally, each *DStream* contains a sequence of Spark Resilient Distributed Datasets (RDDs), which are the main Spark Core data abstractions. RDDs cannot be changed and can be executed in parallel. In addition, RDD offers operations, including data transformation and actions, that can be used for Spark Streaming for data analysis. Each RDD in the *DStream* represents data for a specific time interval. Therefore, all operations are applied on *DStream* will be applied to the RDDs within the same *DStream* [43].

### B. NEURAL NETWORK MODEL

The term *backpropagation* in neural networks comes from computing the error vector backward, starting from the last layer in the network [44]. Before backpropagation is initiated, other processes are done first. These processes include calculating the activation values of units and propagating them to the output units. Then the cost function is applied to compare the actual output error results $y_P^o$ with the desired output values $d_o$. There is usually the signal error $\delta_o^P$ from each unit in the output layer. The goal of backpropagation is to reduce the amount of difference between the actual output and the desired output as much as possible [45]. This can be achieved by backward passes through every hidden layer in order to carry the error signal to all units in the neural networks and to recalculate the weights of connections in the hidden layers.

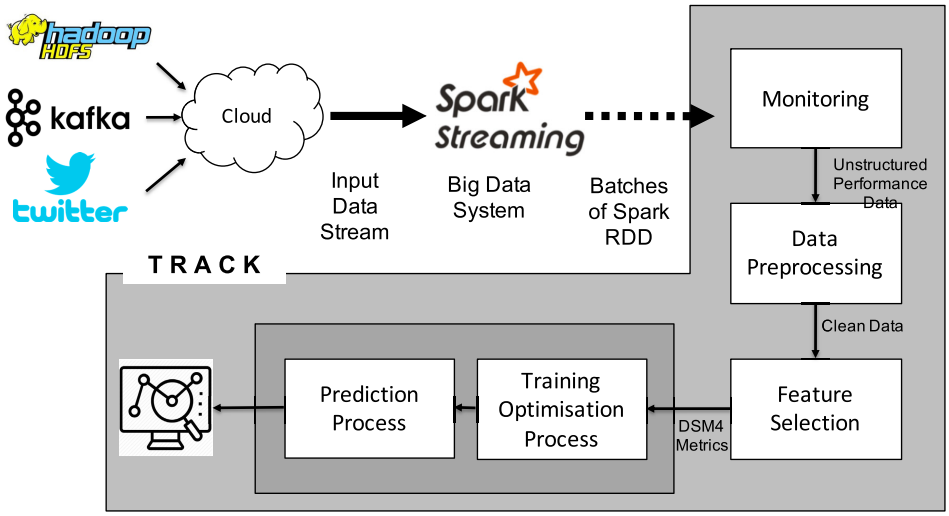



**FIGURE 2.** TRACK-Plus methodology for anomaly detection.

Equation (1) provides the recursive procedures that compute all error signals $\delta_h^p$ for all units in the hidden layers [45]. The $F'$ is the derivative of squashing function for the $k^{th}$ unit in the neural network and is evaluated at the network input ($s_h^P$) for that unit, where $P$ is the input features vector, $N_o$ number of units in output layer, $h$ is a hidden unit, $o$ is an output unit, and $w_{ho}$ is the weight of the connection between unit in hidden and output layer.

$$\delta_h^p = F'(s_h^P) \sum_{o=1}^{N_o} \delta_o^p w_{ho} \tag{1}$$

The traditional neural networks contain three layers, which input, hidden, and output layer. There are more complex structures of neural networks that require additional execution time and computing resources such as convolutional neural networks, a type of deep neural networks. These neural networks are usually used for image processing. The neural networks model used here has fewer input features (less than 30 features) and output classes than what is used in image processing classification. Therefore, in our case, neural networks with three layers achieve accurate performance classifications.

### C. BAYESIAN OPTIMIZATION

The proposed methodology revolves around using Bayesian Optimization (BO) to find the optimal dataset size and configuration parameters for training the neural network to generalize the model so it will detect anomalous behaviors in the Spark Streaming system.

When utilizing BO, there are two main choices to make: using prior over functions and type of acquisition function [46]. It is essential to choose prior over functions to express assumptions about the optimized function. There are different types of acquisition functions, such as *Expected Improvement* [46], *Probability of Improvement* [47], *Lower Confidence Bound* [48], and *Per Second and Plus*. Each type of acquisition function is further discussed in [49].

## V. METHODOLOGY

In this section, we introduce TRACK and TRACK-Plus, methodology driven by Bayesian Optimization (BO) and neural networks to train and detect, classify performance anomalies in Apache Spark Streaming systems. Figure 2 shows the TRACK processes of anomaly detection.

### A. MACHINE LEARNING MODEL

The neural networks model is used to accurately detect anomalous performance within in-memory Big Data systems such as Apache Spark. The proposed neural networks model in [41] with backpropagation and conjugate gradients is used to train the neural networks to update values of weights and biases in networks. The scaled conjugate is used because it is often faster than other gradient algorithms [50], especially for time-dependent applications such as real-time stream processing.

The neural networks model uses a *Sigmoid* transfer function equation(2) as an activation function, where $x$ includes values of input values to neuron, wights, and bias. *Softmax* transfer function is used in the output layer to handle classification problems with multiple classes of anomalies. For a cost function, *cross-entropy* is used to evaluate the performance of neural networks model. *Cross-entropy* is used because it has significant practical advantages over squared-error cost functions [51].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

The proposed neural networks contain three types of layers. The first type of layer is the input layer, which includes a number of neurons equal to the number of input features. The second type of layer is the hidden layer, which has a number of layers (1, 2, or 3) and number of neurons determined using a *trial and error* method, choosing a number between the sizes of input features $n_i$ and output classes $n_o$ [52]. A hidden layer size between $n_i$ and $n_o$ satisfies our goal in achieving accurate results. In our case, the hidden layer size of 5,

10, 15, and 20 achieve 98%, 99%, 96%, and 96% F-scores, respectively. The Hidden layer with ten neurons achieves the highest F-score with the Spark Streaming workload. The output layer contains a number of neurons equal to the number of target classes (types of anomalies), where each neuron generates boolean values: either 0 for normal behavior or 1 for anomalous behavior.

TRACK and TRACK-Plus use Bayesian Optimization to find the optimal training dataset size and configuration parameters to efficiently train the anomaly detection model to achieve high accuracy in a short period of time. Due to its simplicity and tractability, we chose the Gaussian process prior for our proposed model. The acquisition function is used to evaluate a point $x$ based on the posterior distribution function to guide exploration and evaluate the next point [46].

The *Expected Improvement* acquisition function in [53] is used to evaluate the expected performance improvement in the neural networks detection model $f(x)$ and ignore any values that increase the error rate of the model. In other words, $x_{best}$ is the location of the smallest posterior mean (optimal workload configuration) and $\mu_Q(x_{best})$ is the smallest value of the posterior mean. The expected improvement can be described as follows:

$$EI(x) = E_Q[max(0, \mu_Q(x_{best}) - f(x))] \tag{3}$$

$E_Q$ indicates the expectation assumed under the posterior distribution given the evaluations of $f$ at $x_1, x_2, \ldots, x_n$. The time to assess the objective function may vary depending on the region [53].

To improve the performance of the proposed methodology, our TRACK method uses a *customized* acquisition function that utilizes time weighting and the *Expected Improvement* for the acquisition function. The *Expected Improvement* acquisition function assesses the current improvement in the objective function and avoids all outputs that may undermine the performance of objective function output. In addition, the acquisition function operates such that during the evaluation of the objective function by the BO model, another Bayesian model (time-weighting) evaluates the time of the objective function [53]. The final acquisition function is as follows:

$$EI_PS(x) = \frac{EI(x)}{\mu_t(x)} \tag{4}$$

$$EI_PS(x) = \frac{E_Q[max(0, \mu_Q(x_{best}) - f(x))]}{\mu_t(x)} \tag{5}$$

where $\mu_t(x)$ describes the posterior mean of the Gaussian process model timing [53]. A coupled constraint is evaluated only by evaluating the objective function. In our case, the objective function is the performance evaluation of the neural networks model by calculating the F-score. The coupled constraint is that the F-score of the model is not less than a predetermined value (e.g., 90%). The model has several points that are equal to the number of all possible combination parameters of Spark Streaming workload configurations.

**Algorithm 1:** Training and Testing Methodology for TRACK

**Input:** Predefined anomaly detection performance $\mathcal{F}$, Workload configuration space $\mathcal{X}$, and system metrics dataset $\mathcal{D}$

**Output:** Optimal trained neural network model $\mathcal{M}$, which is able to identify anomalies within Spark Streaming with the high predefined F-score in the least amount of time.

Configuring streaming workload benchmark
Workload generation with configuration space $\mathcal{X}$
Streaming workload from network $\mathcal{W} \rightarrow$ Spark system
System profiling to collect performance dataset
Data cleansing and preprocessing $\rightarrow \mathcal{D}$
$DSTrain = 75\%$ of $\mathcal{D} \leftarrow$ total training dataset
$DSTest = 25\%$ of $\mathcal{D} \leftarrow$ total testing dataset
$DSTrain_c$ is empty
$F = 0 \leftarrow$ current f-score
*Default_Net_Config*: 3 *layers*, 10 units in hidden layer, and *cross-entropy*
$i =$ zero
**while** $( ( F \leqslant \mathcal{F}) \; AND \; (i \leqslant size(\mathcal{X})) )$ **do**
  $\mathcal{X}_i = EI_PS(\mathcal{X}) \leftarrow$ acquisition function
  $DSTrain_c = DSTrain_c + DSTrain_{X_i} \leftarrow$
  $\mathcal{M} =$ TrainNN($DSTrain_c$, $NetConfig$ ) $\leftarrow$ train model on the new dataset configuration
  $\mathcal{F} =$ Max( $Fscore(\mathcal{M}(\text{DSTest}))$, $\mathcal{F}$ )
  $i = i + 1$

### B. MODEL TRAINING, VALIDATION AND TESTING

The Spark Streaming system is randomly injected with anomalies to test the proposed anomaly detection model. For the training process (covers local training, local validation, and local testing), the dataset for every combination of workload configuration parameters (e.g., size $s$ and rate $r$) is divided into two sets: 75% for model training (DSTrain) and 25% for a global testing dataset (DSTest), as shown in Figure 3. The local DSTrain set for the model is divided into three subsets: local training (70%), local validation (15%), and local testing (15%). The training subset is used to train the model, whereas the validation subset is used to validate the model and to avoid overfitting and underfitting issues. The local testing subset is used to test the model against a single combination of configuration parameters for Spark Streaming workloads. The DSTest set is used to globally test the

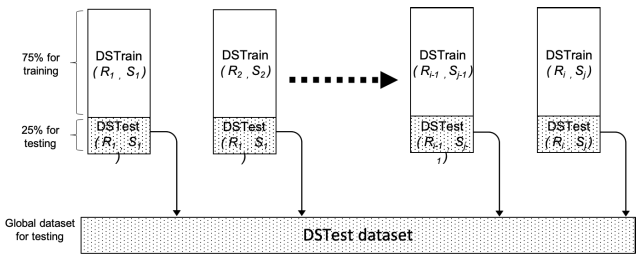



**FIGURE 3.** Dividing dataset into *DSTrain* and *DSTest* sets for training and testing, respectively.

model, which includes 25% from each possible combination of Spark Streaming workload configuration parameters. This subset is used to independently assess the trained model and to generalize the model.

The streaming workload configurations consist of all the possible combinations of configuration parameters of $Rate_n$ and $Size_m$, for a total of $n \times m$ combinations ($n \times m$ *DSTrain*). The training part of the dataset (*DSTrain*) is divided into 10 equal subsets to find the ideal size of the training dataset. For example, the dataset *DSTrain* workload configuration with rate $r_i$ and size $s_j$ is divided into 10 subsets according to the following equation:

$$DSTrain_{r_i,s_j} = DSTrain_{r_i s_j,1} + \ldots + DSTrain_{r_i s_j,10} \quad (6)$$

The total number of all the possible data subsets is $n \times m \times 10$, so it would be challenging and time-consuming to find the optimal combination of configuration parameters and dataset sizes to train the model. More detailed information about TRACK and TRACK-Plus is presented in Algorithm 1 and Algorithm 2. To assess the proposed model, we use a well-known standard classification performance metric, which is F-score (F), defined in the Appendix alongside the standard metrics of Precision (P) and Recall (R). Further information is provided in the Appendix about Precision, Recall, and F-score.

### C. FEATURE SELECTION

The Spark system is monitored at all times and we consider different levels of logging, ranging from Spark jobs measurements to the complete availability of Spark task execution logs, which are used in [41]. These logs provide a reflection of the full details of a Spark system performance. The performance monitoring happens in the background without generating any noticeable overhead in the Spark system.

In this work, we extend the method proposed in [41], called DSM4, which has been built upon the list of Spark performance metrics presented in [41]. DSM4 examines the internal Apache Spark architecture and the Directed Acyclic Graph (DAG) of the Spark application by relying on information from Apache Spark systems. This information includes Spark executors, shuffle read, shuffle write, memory spill, java garbage collection, tasks, stages, jobs, applications, identifications, and execution timestamps for Spark resilient distributed datasets (RDDs). The collected Spark performance metrics are in time series and manually labeled as either normal or anomalous before they are passed as inputs to the proposed model. The proposed methodology assumes that the collected data is pre-processed to ensure the exclusion of any mislabeled training instances and to validate the datasets before passing them to the BO and neural networks model to improve their quality. For example, we avoid duplicated task measurements and exclude samples if features are missing as a result of the monitoring service level anomalies.

## VI. EVALUATION

This section evaluates the proposed methodology using a random search (RS) algorithm as a baseline for the same datasets, which are generated from the Apache Spark Streaming system.

**Algorithm 2:** Training and Testing Methodology for TRACK-Plus

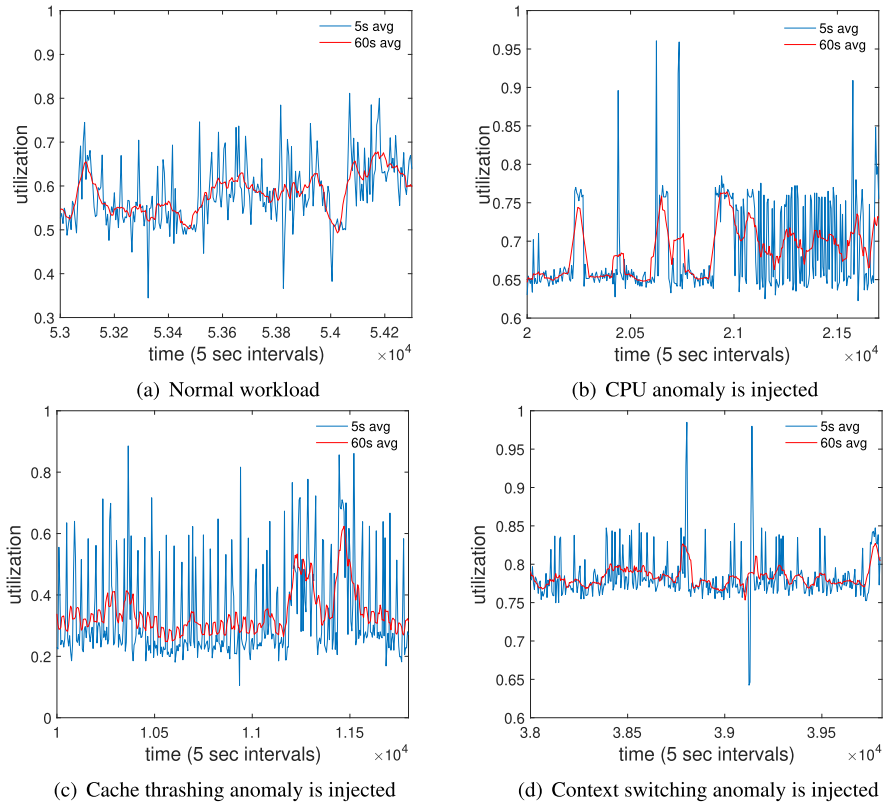
**Input:** Predefined anomaly detection performance $\mathcal{F}$, Workload configuration space $\mathcal{X}$, and system metrics dataset $\mathcal{D}$

**Output:** Optimal hypertuned trained neural network model $\mathcal{M}$, which is able to generate an agile model to classify anomalies within Spark Streaming with the high predefined F-score in the least amount of time.

Configuring streaming workload benchmark
Workload generation with configuration space $\mathcal{X}$
Neural Networks with configuration space $\mathcal{NN}$
Streaming workload from network $\mathcal{W} \rightarrow$ Spark system
System profiling to collect performance dataset
Data cleansing and preprocessing $\rightarrow \mathcal{D}$
$DSTrain = 75\%$ of $\mathcal{D} \leftarrow$ total training dataset
$DSTest = 25\%$ of $\mathcal{D} \leftarrow$ total testing dataset
$DSTrain_c$ is empty and $F = 0 \leftarrow$ current f-score
*Default_Network_Configurations*: $\mathcal{L}$ *layers*, $\mathcal{U}$ units in hidden layer, and $\mathcal{P}$ *Performance function*
$i =$ zero
**while** $( ( F \leqslant \mathcal{F}) \; AND \; (i \leqslant size(\mathcal{X})) )$ **do**
  $\mathcal{X}_i = EI_PS(\mathcal{X}) \leftarrow$ acquisition function finds next workload configurations
  $DSTrain_c = DSTrain_c + DSTrain_{X_i} \leftarrow$
  $j =$ zero and $i = i + 1$
  **while** $( ( F \leqslant \mathcal{F}) \; AND \; (j \leqslant size(\mathcal{NN})) )$ **do**
    $NetConfig_j = EI_PS(\mathcal{NN}) \leftarrow$ acquisition function finds next neural networks configuration
    $\mathcal{M} =$ TrainNN($DSTrain_c$, $NetConfig_j$ ) $\leftarrow$ train neural network model on the new dataset configuration
    $\mathcal{F} =$ Max( $Fscore(\mathcal{M}$(DSTest)), $\mathcal{F}$ )
    $j = j + 1$
  $\mathcal{M} =$ TrainNN($DSTrain_c$, $NetConfig$ ) $\leftarrow$ train neural network model on the new dataset and hyperparameters configuration
  $\mathcal{F} =$ Max( $Fscore(\mathcal{M}$(DSTest)), $\mathcal{F}$ )

### A. EXPERIMENTAL TESTBED

The experiments are conducted on a Spark Streaming system with 16 core Intel(R) Xeon(R) CPU 2.30 GHz, 32 Gb RAM, Ubuntu 16.04.3, and 2 TB of storage. The Apache Spark is deployed with the Spark Standalone Cluster Manager, 16 executors, and a first-in-first-out scheduler option for deployment. Performance monitoring and data collection are done in the background without causing any noticeable overhead on the system. *Spark History* is used to actively record the performance metrics of internal Spark architecture, such as Spark DAG jobs, stages, and tasks.




(a) Normal workload

(b) CPU anomaly is injected

(c) Cache thrashing anomaly is injected

(d) Context switching anomaly is injected

**FIGURE 4.** CPU utilization for Spark Streaming workload with normal and anomalous performance.

### B. WORKLOAD GENERATION

To evaluate the accuracy of the proposed anomaly detection methodology, we developed the customized *WordCountExp* benchmark for Big Data stream processing systems to generate datasets for training and testing purposes. The workloads are exponentially generated (with exponential distribution) as messages sent through the system network to the data stream processing system with some predefined characteristics such as the rate of sending messages per second and the size of messages. *WordCountExp* is used extensively with many different configurations to evaluate and compare the results of the proposed methodology within in-memory Spark Streaming systems. More than 960 experiments are conducted and 230 Gb of data are collected from the Spark Streaming system, which we use to evaluate the proposed work. The dataset covers four types of injected anomalies within Spark Streaming workloads: normal, CPU anomaly, cache thrashing, and context switching. CPU utilization of the Spark system is shown with different types of anomalous performance in Figure 4.

WordCount is a conventional CPU-intensive benchmark and is widely accepted as a standard micro-benchmark for Big Data platforms [37], [54]–[57]. The WordCount benchmark splits each line of text into multiple words, then aggregates the total number of times each word appears throughout and updates an in-memory map with the words as the key and the frequency of the words as the value. Figure 5 shows a wordcount example of Spark Streaming that receives a streaming workload from a local network to count the number of words per message. The Main *DStream* data is divided into many RDDs for a certain time interval, then some Spark operations, such as wide and narrow operations, are done to count the number of words in each Spark RDD. More details about Spark operations are discussed in [41].

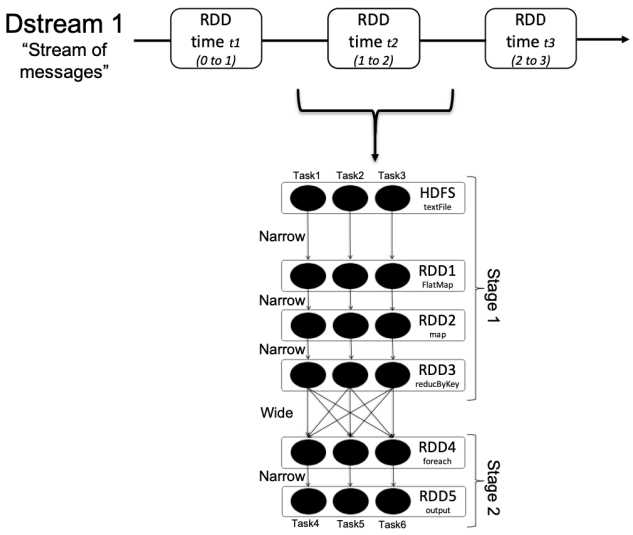



**FIGURE 5.** Spark Streaming with WordCount example of DStream.

### C. ANOMALY INJECTION

To inject different types of anomalies, the open-source tool (*stress-ng*) is used to evaluate the proposed methodology with the Spark Streaming system [41]. A list of performance

**TABLE 2. 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. |

anomalies is used to generate CPU stress, cache thrashing stress, and context switching stress as shown in Table 2. The CPU stress spawns $n$ workers to run the sqrt() function; the cache thrashing stress causes $n$ processes to perform random widespread memory read-and-writes to thrash the CPU cache; and the context switching stress has $n$ processes that forces context switching. The injected anomaly and the used benchmark are configured depending on the objective of the experiment, which will be discussed in Section VII.

## VII. RESULTS

The proposed methodology is evaluated on an isolated Spark Streaming system, discussed in Section VI-A. We avoid using a virtual Spark System, which ensures that all performance metrics are accurately measured.

### A. FINDING THE IDEAL WORKLOAD CONFIGURATION FOR MODEL TRAINING

The previous discussion regarding the motivating example (Section III) describes the need to find the ideal single workload configurations set (e.g., rate $r_i$ and size $s_j$) that could be used to train the proposed anomaly detection model to pinpoint the abnormal behavior with the highest possible F-score. This facilitates the use of a single workload configuration to be generalized and used to detect anomalies with the other workload configurations. The Spark Streaming workload has all possible combinations of *rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*, for a total of 16 combinations.

A Bayesian Optimization (BO) and neural networks model (described in Section IV-C and in IV-B) are used to address the need for determining the ideal single workload configuration (rate $r_i$ and size $s_j$) with the minimum number of running experiments $n$. To ensure accurate results, the experiments are conducted 50 times, then the average of $n$ is calculated. The results show that the ideal F-score is reached with the minimum number of running experiments ($n$=8), which is 50% less than the total number of possible configurations ($n$=16).

Figure 6 shows the performance results of the proposed model when it is individually trained on each workload configuration (rate $r_i$ and size $s_j$) and tested against all possible combinations of streaming workload configurations using BO and neural networks. The estimated objective value is the deviation from the ideal F-score ($error = 1\text{-}\ F\text{-}score$).

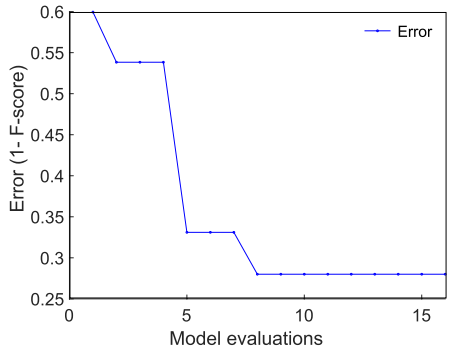



**FIGURE 6. Training process results for each workload configuration, tested on all possible combinations of streaming workload configurations using Bayesian Optimization and neural networks.**

Figure 6 illustrates that with the given dataset, the workload configuration ($r = 32$, $s = 1$) can be used to train the anomaly detection model to detect abnormal behavior with all other streaming workload configurations with the highest F-scores equaling 72% after running only 8 of 16 experiments. The next section explores a new approach to optimize the model and obtain a higher F-score using less time in training processes.

### B. BAYESIAN OPTIMIZATION MODEL TO TRAIN ANOMALY DETECTION TECHNIQUE

A BO model (discussed in Section IV-C) is used to find the optimal size of the training dataset and the streaming workload configurations set to achieve the highest accuracy with the least time spent training the proposed anomaly detection model. The model training and datasets of anomaly detection are comprehensively discussed in Section V-B and V-C. Figure 7 depicts a comparison of BO and RS to reach a predefined F-score with the fewest training steps from the total 160 steps. The conducted experiments have workloads containing both normal and anomalous CPU behaviors with all possible combinations of workload configurations. Figure 7 shows the average of the 50 experiments where the neural networks model is trained using BO to achieve the predefined F-score. With BO, the trained model reaches a 95% F-score in 21 steps, whereas an RS uses 28 steps (enhanced by 25%). This proves that the proposed model can reduce the time and computation process by 25%. Table 3 shows the performance of five different types of acquisition functions that are used with BOs.

Two other types of anomalies may disrupt the performance of the Big Data stream processing system. These two types

**TABLE 3. Performance of different types of acquisition functions to reach 95% F-score. The right column shows the average number of steps for 50 experiments to find the ideal dataset size to train the model.**

| Acquisition Functions | Number of steps |
|---|---|
| BO expected-improvement | 64.3 |
| BO expected-improvement-plus | 21 |
| BO expected-improvement-per-second | 30.2 |
| BO lower-confidence-bound | 54.9 |
| BO probability-of-improvement | 43.4 |

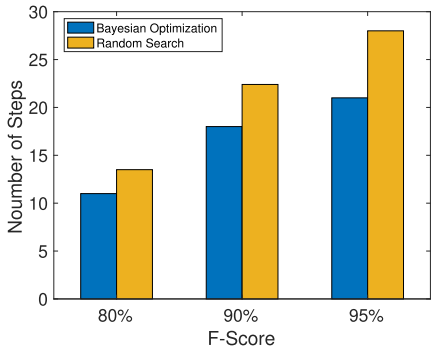



**FIGURE 7.** Comparison of Bayesian Optimization (BO) and RS for reaching a predefined F-score with the fewest training steps. Workloads have all possible combinations of parameters and CPU anomalies.

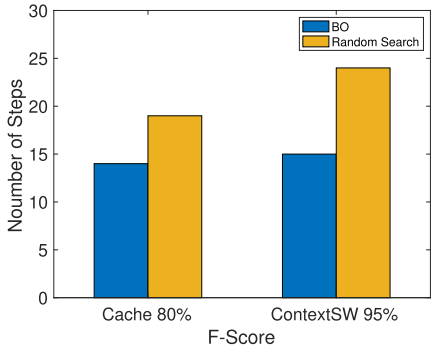



**FIGURE 8.** Comparison of BO and RS to reach predefined F-scores (80% for cache stress and 90% for context switching stress) with the minimum number of steps. Workloads have all possible combinations of parameters.

are cache thrashing and context switching. The proposed model can detect both cache thrashing and context switching anomalies with F-scores of 80% and 95%, respectively. Figure 8 shows that the proposed model outperforms RS by more than 25% and can reduce the amount of computations from 160 experiments to 14, as can be seen in Figure 8 with cache thrashing anomalies.

### C. SENSITIVITY ANALYSES OF TRAINING DATASET SIZE

In this subsection, the impact of the training dataset size is examined to prove the robustness of the proposed model. The amount of anomalous Spark tasks decrease by 50% to 75% of the anomalous workload in Section VII-B. Table 4 depicts the impact of the Spark workload training set size on the proposed stream anomaly detection model. The BO with neural networks model achieves the highest performance in detecting all three types of performance anomalies in Spark

**TABLE 4.** Sensitivity analysis demonstrating the impact of reducing the overall anomalous training dataset size by 50% to 75%. BO is compared against RS to assess when each would reach ideal performance (95% F-score) with the fewest possible steps and the least training data. Workloads contains all possible combinations of rates 1, 8, 16, and 32 *message/sec* and sizes 1, 10, 100, and 1000 *line/message*.

| Algorithm | CPU | Cache | Context Sw |
|---|---|---|---|
| BO | 38 | 19.9 | 16 |
| Random Search | 44 | 24 | 25 |

**TABLE 5.** Testing the proposed model against new unseen workload configurations with three types of performance anomalies.

| Types of Anomalies | F-score ± Standard deviation |
|---|---|
| CPU | 0.93±0.01 |
| Cache Thrashing | 0.77±0.02 |
| Context Switching | 0.72±0.04 |

Streaming systems. This proves that the proposed model is robust against changes in the size of the overall input training datasets.

### D. NEW UNSEEN WORKLOAD CONFIGURATIONS

This section presents the training of the proposed model with predefined workload configurations (*rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*) and generalizes the model to perform just as accurately with new unseen workload configurations (e.g., $r_i = 20$ and $s_j = 150$). In this case, the workload is more realistic and reflects the workload characteristics of the real stream processing system in the production environment.

For the training phase, the same BO and neural network configurations in Section VII-B are used to train the model on predefined workload configurations (*rates 1, 8, 16, and 32 message/sec* and *sizes 1, 10, 100, and 1000 line/message*) to reach a 95% F-score for detecting CPU performance anomalies. For the testing phase, the final model of the training phase is used to detect anomalous behavior but with new unseen workload configurations. The rate could be between 1 to 32 and the size could have ranged from 1 to 1000. The total number of possible configuration combinations is 32k.

Table 5 shows the performance of the proposed model when it is tested against three types of anomalies (i.e., CPU, cache thrashing, and context switching). As seen in Table 5, the proposed anomaly detection model can be trained on 16 workload configurations to be generalized to detect anomalies against 32k different workload configurations.

### E. DETECTING AND CLASSIFYING PERFORMANCE ANOMALIES

In this section, we show that TRACK will not only detect anomalous performance but also classify workloads into four types: normal, CPU anomaly, cache anomaly, and context switching anomaly. The anomaly detection using TRACK achieves 74% for detecting and classifying Spark Streaming performance, as seen in Table 6. The next section introduces a new optimized version of TRACK called TRACK-Plus to find the ideal neural network configuration to accelerate the search process and improve anomaly classification.

### F. TRACK-PLUS FOR OPTIMIZING THE CHOICE OF NEURAL NETWORKS ARCHITECTURE

The performance of TRACK-Plus is evaluated using the two BO models discussed in Section IV-C. The first BO1 is used to find the ideal dataset training size as described

**TABLE 6. Performance of TRACK for detecting and classifying anomalies based on their root causes.**

| Type of workload | Recall | Pres | F-score |
|---|---|---|---|
| Normal | 0.89 | 0.91 | 0.90 |
| CPU | 0.56 | 0.55 | 0.55 |
| Cache Thrashing | 0.97 | 0.95 | 0.96 |
| Context Switching | 0.54 | 0.56 | 0.55 |
| Average | 0.74 | 0.74 | 0.74 |

**TABLE 7. All possible optimized configurations for TRACK-Plus including two BO models.**

| BO# | Parameters | Possible Configurations |
|---|---|---|
| BO1 | Message Rate (*message/sec*) | 1, 8, 16, or 32 |
| | Message Size (*line/message*) | 1, 10, 100, or 1000 |
| | Training Dataset Size | 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10 |
| BO2 | Number of Neurons | 5, 10, 15, or 20 |
| | Number of Hidden Layers | 1, 2, or 3 |
| | Performance Function | mae, mse, sae, sse, or crossentropy |

in Section V-B. BO1 optimizes the choices for three Spark Streaming workload configurations, which are the rate of messages per second (1, 8, 16, and 32), message size (1, 10, 100, and 1000), and the size of the training dataset (1 to 10). The total number of possible configurations is $4 \times 4 \times 10$, which comes close to 160 different possible combinations.

The objective of the second BO2 is to automate the search to achieve the most efficient architecture of neural networks (with a predefined list of configurations) by optimizing the tuning process of the hyperparameters of the neural networks. In practice, different configurations of hyperparameters can significantly impact the performance of the neural networks. In this study, we focus more on hyperparameters related to neural network training and structure, including the number of hidden layers, number of neurons in each layer, and performance functions. Five well-known performance functions have been examined in TRACK-Plus, which are mean absolute error, mean squared error, sum absolute error, sum squared error, and cross-entropy. The total number of possible configuration combinations for BO2 is $5 \times 3 \times 4 = 60$ different possible configurations. Details of the configuration parameters of the two BO models can be found in Table 7.

Even with the limited number of configurations to train the anomaly detection technique, TRACK-Plus offers an efficient solution in finding the ideal training dataset size and the most efficient neural network configurations to accurately detect the anomalous performance within the Spark Streaming system. For example, Table 7, with the list of the total number of possible configuration combinations, shows that there are $160 \times 60 = 9600$ possible configurations. It is clear that finding the ideal configurations with which to train the anomaly detection model is more time-consuming and resource intensive when using either the traditional search or the manual configurations.

**TABLE 8. The ideal configuration for the neural networks in terms of performance function, number of layers, and number of neurons in each layer.**

| Parameters | Configurations | Number of Selection |
|---|---|---|
| Performance Func | mae | 16 |
| | mse | 3 |
| | sae | 21 |
| | sse | 8 |
| | crossentropy | 2 |
| #Neurons/Layer | 5 | 18 |
| | 10 | 14 |
| | 15 | 8 |
| | 20 | 10 |
| #Layers | 1 | 25 |
| | 2 | 9 |
| | 3 | 16 |

Table 8 shows the average results of 50 experiments where the TRACK-Plus optimizes the training process of anomaly detection to achieve the predefined F-score, which is 70% (the highest possible F-score for classifying the anomalies). With the given conditions of Spark Streaming workloads, we find that the ideal neural networks configurations are *sae* performance function, five neurons/layer, and one hidden layer.

## VIII. CONCLUSION

To develop effective fault-tolerant system performance, it is vital to detect anomalous performance and service level disruption events within data intensive systems. The growing complexity of Big Data systems makes performance anomaly detection more challenging, especially for critical streaming workload applications in distributed systems environments. Therefore, the performance of in-memory processing technology like Apache Spark Streaming must be thoroughly investigated to pinpoint the causes of performance anomalies.

Collecting all possible performance measurements from Big Data systems to train the anomaly detection system is computationally expensive, especially for critical systems such as online banking, stock trading, and air traffic control systems. Even with the WordCount Spark Streaming application (only has two parameters $r$ and $s$), it is considered a time-consuming and costly intensive computing to find the ideal dataset size to efficiently train the anomaly detection model so it will comprehensively cover all seen and unseen anomalies.

This paper contributes by addressing the challenge of anomalous identification by proposing a new hybrid learning solutions, TRACK and TRACK-Plus, for anomaly detection within in-memory Big Data systems. The anomaly detection and tuning method are developed using Bayesian Optimization and neural networks to train the model with a limited budget and limited computing resources. As can be seen from the experimental results, the proposed model efficiently finds the optimal training dataset size and configuration parameters to accurately identify different types of performance anomalies in Big Data systems. The proposed model achieves the

highest accuracy (95% F-score) in significantly less time (80% less than normal). A validation based on a real dataset for the Apache Spark Streaming system has been provided to demonstrate that the proposed methodology identifies the performance anomalies, the ideal configuration parameters, and the training dataset size with up to 75% fewer experiments. Finally, the proposed solutions not only identifies anomalous performance with a high F-score but also classifies anomalies, thereby saving considerable time in training the model. In addition, the proposed model can be easily generalized to cover unforeseen workload configurations.

In terms of future work, it is crucial to deeply investigate an anomaly detection and prediction for systems that contain both batch and stream processing workloads at the same time. Such systems will have more increasing complexity and performance fluctuation, which may need more effective anomaly detection solutions. Exploring deep Learning algorithms may hold opportunities to accurately detect and predict the performance anomaly in distributed complex systems.

## APPENDIXES

*Recall* (also called *Sensitivity*) and *Precision* performance measures are used in this paper to evaluate Track and Track-Plus for anomaly detection classifiers. These performance metrics are commonly used and standard metrics for quantifying the accuracy of the classifiers [58]. The following are the anomaly classification classes and their notations:

- True positive (*tp*): a correct detection of anomalies
- False positive (*fp*): a detection of anomalies when it does not exist
- False negative (*fn*): a missed detection of anomalies when it exists.

$$R = \frac{tp}{tp + fn} \tag{7}$$

$R$ is the Recall and it answers this question: of all the samples which are anomalies, how many of those are correctly detected?. *Recall* assesses the effectiveness of a classifier in identifying positive samples. *Recall* will become higher when the anomaly detection classifier can detect all anomalies.

$$P = \frac{tp}{tp + fp} \tag{8}$$

$P$ is *Precision* and it answers this question: how many of those samples which are labeled as anomaly are actually anomaly?. *Precision* quantifies class agreement on how many samples classified as positive (*anomaly*) are, indeed, positive. It assesses the reliability of the detection method when it detects anomalies.

$$F = 2\frac{PR}{P + R} \tag{9}$$

*F-score* reflects the relations between data's anomalous labels and those given by a classifier. *F-score* captures the trade-off between *Recall* and *Precision*. It shows a summary score computed for harmonic mean of *Recall* and *Precision*. Throughout this paper, we use the F-score as the main performance metric. The formulas of *Recall*, *Precision*, and *F-score* reflect the quality of classifier in detecting the positive samples (anomalies in our case), without paying significance attention on the correct classification [59].

## REFERENCES

[1] S. Fu, J. Liu, and H. Pannu, "A hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines," in *Proc. Int. Conf. Adv. Data Mining Appl.* Nanjing, China: Springer, 2012, pp. 726–738.

[2] X. Gu and H. Wang, "Online anomaly prediction for robust cluster systems," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 1000–1011.

[3] S. Baek, D. Kwon, J. Kim, S. C. Suh, H. Kim, and I. Kim, "Unsupervised labeling for supervised anomaly detection in enterprise and cloud networks," in *Proc. IEEE 4th Int. Conf. Cyber Secur. Cloud Comput. (CSCloud)*, Jun. 2017, pp. 205–210.

[4] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *Proc. IEEE 16th Intl Conf Dependable, Autonomic Secure Comput.*, Aug. 2018, pp. 151–158.

[5] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines," in *Proc. WASL*, vol. 8, Dec. 2008, p. 5.

[6] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerg. Artif. Intell. Appl. Comput. Eng.*, vol. 160, no. 1, pp. 3–24, 2007.

[7] S. Fu, "Performance metric selection for autonomic anomaly detection on cloud computing systems," in *Proc. IEEE Global Telecommun. Conf.*, Dec. 2011, pp. 1–5.

[8] W. Qi, Y. Li, H. Zhou, W. Li, and H. Yang, "Data mining based root-cause analysis of performance bottleneck for big data workload," in *Proc. IEEE 19th Int. Conf. High Perform. Comput. Commun.*, Dec. 2017, pp. 254–261.

[9] S. Lu, X. Wei, B. Rao, B. Tak, L. Wang, and L. Wang, "LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with spark," *Future Gener. Comput. Syst.*, vol. 95, pp. 392–403, Jun. 2019.

[10] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jun. 2017, pp. 389–396.

[11] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Comput. Surveys*, vol. 53, no. 2, pp. 1–37, Jul. 2020.

[12] A. S. Alnafessah and G. Casale, "TRACK: Optimizing artificial neural networks for anomaly detection in spark streaming systems," in *Proc. 13th EAI Int. Conf. Perform. Eval. Methodol. Tools*, May 2020, pp. 188–191.

[13] G. Ananthanarayanan, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. OSDI*, Oct. 2010, vol. 10, no. 1, p. 24.

[14] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1–35, Sep. 2015.

[15] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Trans. Services Comput.*, vol. 12, no. 1, pp. 91–104, Jan. 2019.

[16] S. M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie, "High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning," *Pattern Recognit.*, vol. 58, pp. 121–134, Oct. 2016.

[17] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artif. Intell. Rev.*, vol. 22, no. 2, pp. 85–126, Oct. 2004.

[18] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, p. 15, 2009.

[19] M. Lin, Z. Yao, F. Gao, and Y. Li, "Toward anomaly detection in iaas cloud computing platforms," *Int. J. Secur. Its Appl.*, vol. 9, no. 12, pp. 175–188, 2015.

[20] T. Huang, Y. Zhu, Q. Zhang, Y. Zhu, D. Wang, M. Qiu, and L. Liu, "An LOF-based adaptive anomaly detection scheme for cloud computing," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf. Workshops*, Jul. 2013, pp. 206–211.

[21] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, and D. Xu, "CLUE: System trace analytics for cloud service performance diagnosis," in *Proc. IEEE Netw. Operations Manage. Symp. (NOMS)*, May 2014, pp. 1–9.

[22] T. Shon and J. Moon, "A hybrid machine learning approach to network anomaly detection," *Inf. Sci.*, vol. 177, no. 18, pp. 3799–3821, Sep. 2007.
[23] Y. Si, Y. Wang, and D. Zhou, "Key-Performance-Indicator-Related process monitoring based on improved kernel partial least squares," *IEEE Trans. Ind. Electron.*, early access, Feb. 13, 2020, doi: 10.1109/TIE.2020.2972472.
[24] Y. Wang, Y. Si, B. Huang, and Z. Lou, "Survey on the theoretical research and engineering applications of multivariate statistics process monitoring algorithms: 2008-2017," *Can. J. Chem. Eng.*, vol. 96, no. 10, pp. 2073–2085, Oct. 2018.
[25] J. P. Magalhaes and L. M. Silva, "Detection of performance anomalies in Web-based applications," in *Proc. 9th IEEE Int. Symp. Netw. Comput. Appl.*, Jul. 2010, pp. 60–67.
[26] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni, "R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads," in *Proc. Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2007, pp. 244–265.
[27] D. A. Menasce, "TPC-W: A benchmark for e-commerce," *IEEE Internet Comput.*, vol. 6, no. 3, pp. 83–87, May 2002.
[28] T. Kelly, "Detecting performance anomalies in global applications," in *Proc. WORLDS*, vol. 5, 2005, pp. 42–47.
[29] L. Yang, C. Liu, J. M. Schopf, and I. Foster, "Anomaly detection and diagnosis in grid environments," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, pp. 1–9.
[30] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, p. 53.
[31] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 111–124.
[32] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic Internet services," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 595–604.
[33] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," *ACM SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 105–118, Oct. 2005.
[34] H. S. Pannu, J. Liu, and S. Fu, "A self-evolving anomaly detection framework for developing highly dependable utility clouds," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2012, pp. 1605–1610.
[35] R. Ren, S. Tian, and L. Wang, "Online anomaly detection framework for spark systems via stage-task behavior modeling," in *Proc. 15th ACM Int. Conf. Comput. Frontiers*, May 2018, pp. 256–259.
[36] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from Internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 488–499.
[37] N. J. Yadwadkar and W. Choi, "Proactive straggler avoidance using machine learning," Univ. Berkeley, Berkeley, CA, USA, White Paper, 2012.
[38] S. C. Tan, K. M. Ting, and T. F. Liu, "Fast anomaly detection for streaming data," in *Proc. 38nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 1511–1516.
[39] KDD-99. (1999). *Spark Streaming Programming Guide*. Accessed: May 5, 2020. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
[40] M. Lin, S. Chen, G. Chang, L. Liu, and X. Gong, "Research on pca-based anomaly detection system in high-speed network," *J. Comput. Inf. Syst.*, vol. 7, no. 7, pp. 2315–2321, 2011.
[41] A. Alnafessah and G. Casale, "Artificial neural networks based techniques for anomaly detection in apache spark," *Cluster Comput.*, vol. 23, pp. 1–16, Oct. 2019.
[42] Y. Owechko and S. Shams, "Comparison of neural network and genetic algorithms for a resource allocation problem," in *Proc. IEEE Int. Conf. Neural Netw. (ICNN)*, 1994, pp. 4655–4660.
[43] ApacheSpark. (2019). *Spark Streaming Programming Guide*. Accessed: Oct. 1, 2019. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html
[44] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/
[45] B. Kröse, B. Krose, P. van der Smagt, and P. Smagt, *An Introduction to Neural Networks*. Citeseer, 1993.
[46] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 2951–2959.
[47] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *J. Basic Eng.*, vol. 86, no. 1, pp. 97–106, Mar. 1964.
[48] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," 2009, *arXiv:0912.3995*. [Online]. Available: http://arxiv.org/abs/0912.3995
[49] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.
[50] M. F. Møller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Netw.*, vol. 6, no. 4, pp. 525–533, Jan. 1993.
[51] D. M. Kline and V. L. Berardi, "Revisiting squared-error and cross-entropy functions for training neural network classifiers," *Neural Comput. Appl.*, vol. 14, no. 4, pp. 310–318, Dec. 2005.
[52] K. G. Sheela and S. N. Deepa, "Review on methods to fix number of hidden neurons in neural networks," *Math. Problems Eng.*, vol. 2013, Jun. 2013, Art. no. 425740.
[53] Mathworks. (2019). *Bayesian Optimization Algorithm*. Accessed: Oct. 1, 2019. [Online]. Available: https://uk.mathworks.com/help/stats/bayesian-optimization-algorithm.html#bva8tie-1
[54] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Mar. 2016, pp. 592–598.
[55] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Mar. 2010, pp. 41–51.
[56] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, Dec. 2014, pp. 69–78.
[57] R. Han, L. Kurian John, and J. Zhan, "Benchmarking big data systems: A review," *IEEE Trans. Services Comput.*, vol. 11, no. 3, pp. 580–597, Jun. 2018.
[58] G. Casale, C. Ragusa, and P. Parpas, "A feasibility study of host-level contention detection by guest virtual machines," in *Proc. IEEE 5th Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2013, pp. 152–157.
[59] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Inf. Process. Manage.*, vol. 45, no. 4, pp. 427–437, Jul. 2009.

**AHMAD ALNAFESSAH** (Member, IEEE) is currently pursuing the Ph.D. degree with the Department of Computing, Imperial College London. He was a Senior Academic Researcher with the National Centre for AI and Big Data Technologies, KACST, from 2012 to 2017. His research interests include performance engineering for big data systems and AI, with a specific focus on in-memory platforms, the IoT, HPC, complex distributed systems, and cloud computing.

**GIULIANO CASALE** (Member, IEEE) was a Scientist with SAP Research, U.K., and a Consultant with the Capacity Planning Industry. He joined the Department of Computing, Imperial College London, in 2010, where he is currently a Senior Lecturer in modeling and simulation. He also teaches and does research. He has published more than 100 refereed articles. His research interests include performance engineering, cloud computing, and big data. He has served on the technical program committee of over 80 conferences and workshops. His research was a recipient of multiple awards, such as the Best Paper Award from ACM SIGMETRICS, in 2017. He served as the Co-Chair for several conferences in performance engineering, such as ACM SIGMETRICS/performance. He also serves as the Chair for ACM SIGMETRICS. He serves on the editorial boards of the IEEE Transactions on Network and Service Management and *TOMPECS* (ACM).

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


**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].

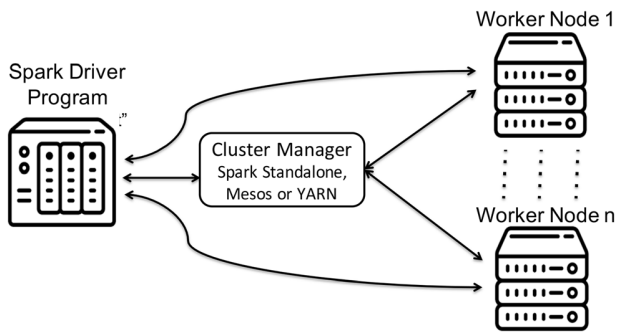



**Fig. 1** Spark application components

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

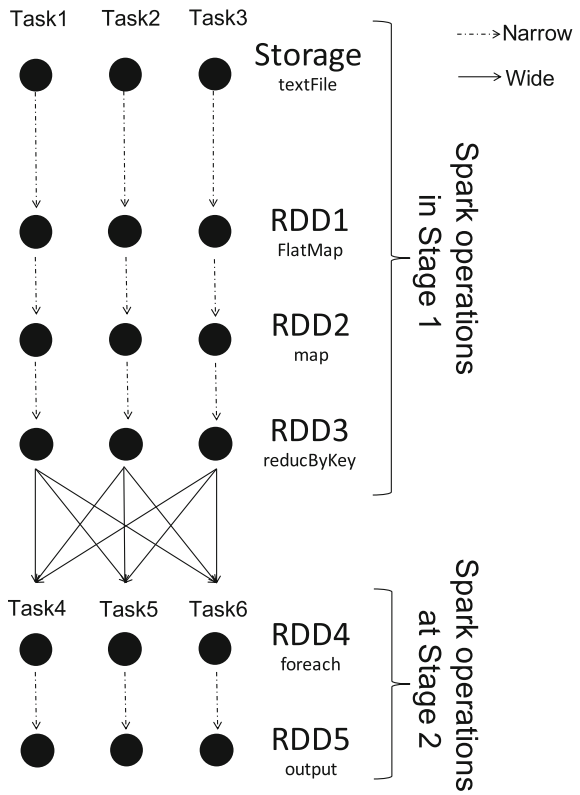



**Fig. 2** Spark DAG for a WordCount application with two stages each consisting of three tasks

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

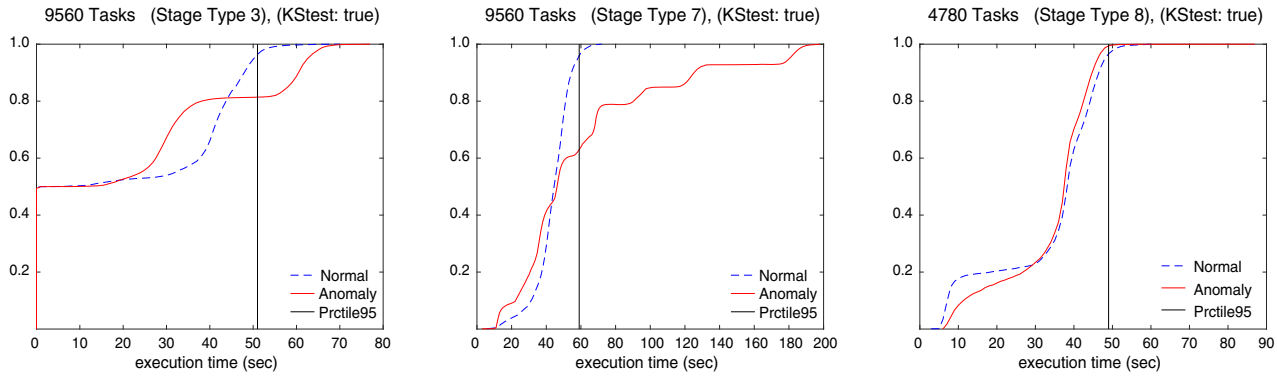
In order to motivate the use of machine learning approaches in anomaly detection methods for Spark, we consider the performance of a simple statistical detection technique based on percentiles of the cumulative distribution function (CDF) of task execution times. Our goal is to use CDF percentiles to 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].

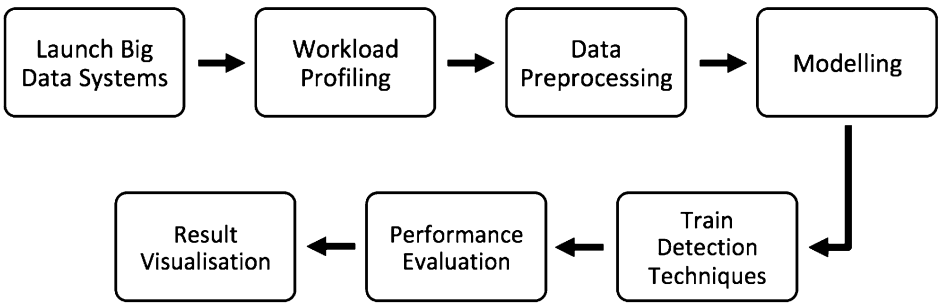



**Fig. 4** Methodology for anomaly detection

#### 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

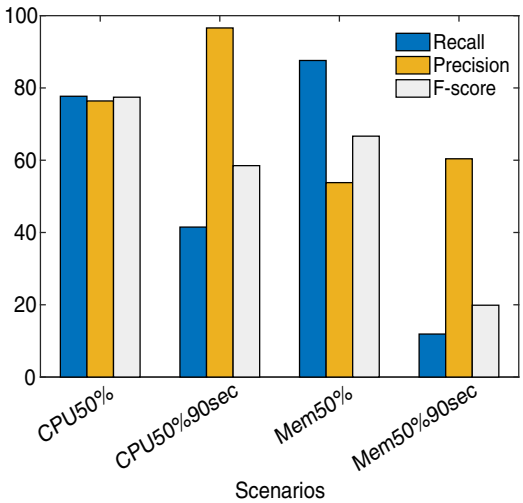



**Fig. 5** Neural network performance with DSM1 feature set for experiments with basic CPU and memory contention (continuous or 90-s periods)

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

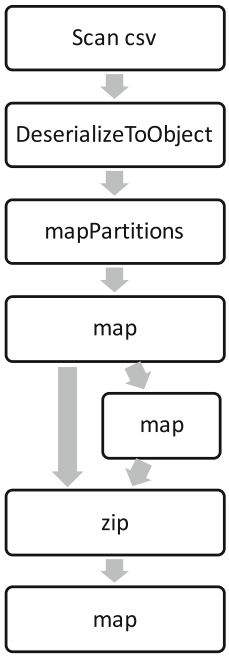



**Fig. 6** DAG diagram illustrates dependencies among operations on Spark RDDs for a single Spark stage within the K-means workload

**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.

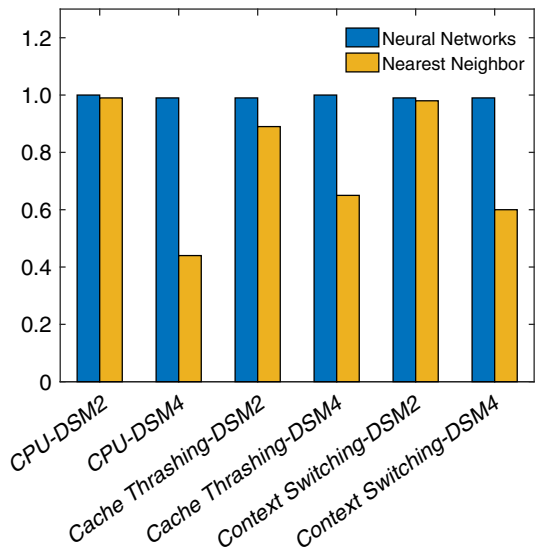



**Fig. 7** F-score performance metrics of neural networks and nearest neighbor under various scenarios

#### 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

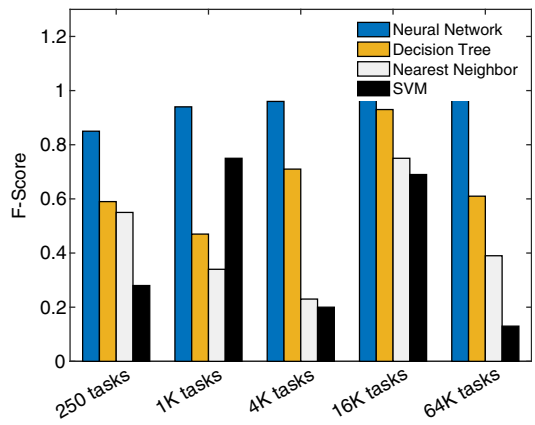
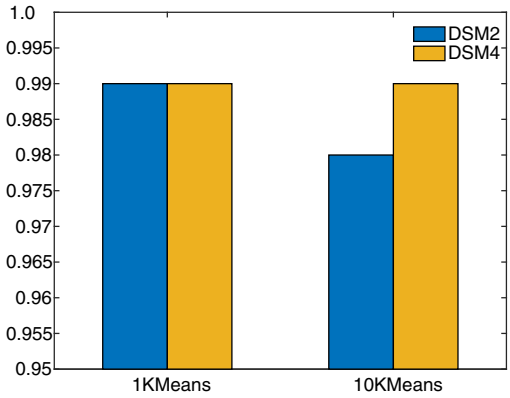
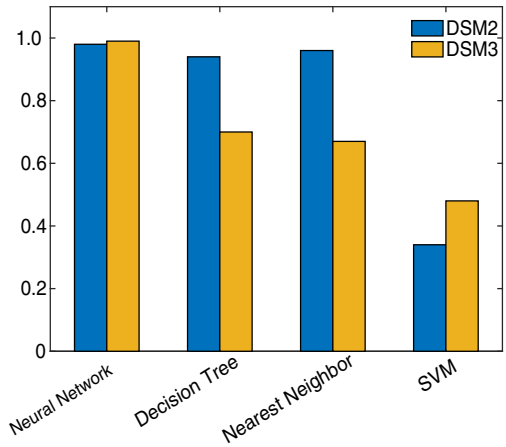



**Fig. 8** Impact of workload size on F-score for Neural Networks, Decision Tree, Nearest Neighbor, and SVM using DSM4 feature sets




**(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).



## 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$^{TM}$: 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)



**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.

# TRACK: Optimizing Artificial Neural Networks for Anomaly Detection in Spark Streaming Systems

Ahmad S Alnafessah
a.alnafessah17@imperial.ac.uk
Imperial College London
London, UK

Giuliano Casale
g.casale@imperial.ac.uk
Imperial College London
London, UK

## ABSTRACT

Due to the growth of Big Data processing technologies and cloud computing services, it is common to have multiple tenants share the same computing resources, which may cause performance anomalies. There is an urgent need for an effective performance anomaly detection method that can be used within the production environment to avoid any late detection of unexpected system failures. To address this challenge, we introduce, TRACK, a new black-box *training workload configuration* optimization with a neural network driven methodology to identify anomalous performance in an in-memory Big Data Spark streaming platform. The proposed methodology revolves around using Bayesian optimization to find the optimal training dataset size and configuration parameters to train the model efficiently. TRACK is validated on a real Apache Spark streaming system and the results show that the TRACK achieves the highest performance (95% for F-score) and reduces the training time by 80% to efficiently train the proposed anomaly detection model in the in-memory streaming platform.

## CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Machine learning**; • **Security and privacy** → *Intrusion/anomaly detection and malware mitigation.*

## KEYWORDS

Performance Anomalies, Apache Spark, Artificial Intelligence, Neural Network, Big Data, Machine Learning

**ACM Reference Format:**
Ahmad S Alnafessah and Giuliano Casale. 2018. TRACK: Optimizing Artificial Neural Networks for Anomaly Detection in Spark Streaming Systems. In *VALUETOOLS '20: ACM VALUETOOLS, May 2020Tsukuba, Japan.* ACM, New York, NY, USA, 4 pages.

## 1 INTRODUCTION

There are various options for open-source Big Data technologies for data-intensive applications. Among various Big Data technologies, in-memory processing technology, such as Apache Spark, has become widely adopted by industries because of its speed, generality, ease of use, and compatibility with other Big Data systems. We here consider Spark based streaming workloads.

With the growing complexity of Big Data and cloud systems, failure management requires significantly higher levels of automation and attention. An anomaly is defined as an abnormal behavior during the execution of a program. While some studies address the challenges of performance anomaly detection for batch processing [8], there is a lack of effective automated performance anomaly detection solutions specifically built for Apache Spark streaming systems. There is a need for a technique that can be used to efficiently train a machine learning model to detect performance anomalies within streaming workloads in production environments.

This paper contributes to address the challenge of anomaly identification by investigating new hybrid learning techniques for in-memory Big Data systems. We developed and optimized artificial neural neTwoRks based methodology for Anomaly deteCtion in sparK streaming systems (TRACK), which has a tuning method capable of training a machine learning model with a limited budget and number of experiments. TRACK revolves around using neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the model to achieve the highest accuracy (95% F-score) and reduces the training time by 80%. A validation based on real datasets from Apache Spark streaming system is provided to demonstrate that the proposed methodology identifies performance anomalies, the ideal configuration parameter, and the optimal training dataset size while reducing the number of experiments by 75%.

## 2 RELATED WORK

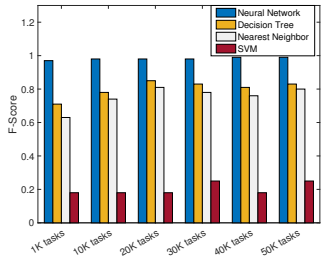
Performance anomaly detection techniques are highly needed by Big Data and large scale systems. Several studies illustrate that most of the root causes of bottleneck and anomalous performance are machine resources such as computer processing units (CPUs) [2].

Fulp et al. [4] use a machine learning approach to detect and predict the likelihood of system failures using an SVM based on information of Linux system log files. Although SVM models are effective at making decisions from well-behaved feature vectors, they can be more expensive for modeling the variation in large datasets and high-dimensional input features [3]. Qi et al. [8] propose a white-box model that uses classification and regression trees to analyze straggler root causes.

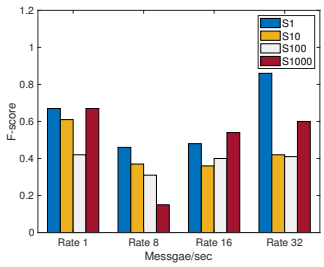
In this paper, we utilize the performance of Bayesian optimization hyperparameter tuning and the efficiency of neural networks to accurately detect anomalous behavior in Big Data systems. Some performance anomaly identification studies have been conducted in the literature for different purposes. However, currently there is still a shortage in studies that offer efficient automated anomaly


*VALUETOOLS'20, May 2020, Tsukuba, Japan*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
DOI: 10.1145/3388831.3388860




(a) Comparison and sensitivity analysis of the impact of Spark workloads size (number of Spark tasks) on neural networks, decision tree, nearest neighbor, and SVM for CPU anomalies detection in Spark streaming workloads.




(b) F-score for anomaly detection using neural network with Rate 1, 8, 16, and 32. Size (1, 10 ,100, and 1000)

**Figure 1: Motivation examples for TRACK.**

detection, especially for in-memory Big Data stream processing technologies, such as Apache Spark streaming.

## 3 MOTIVATING EXAMPLE

Here we demonstrate that the neural networks is more accurate than other well-known algorithms. For our experiments, the workloads are exponentially generated as messages to be sent to the data stream processing system with some predefined characteristics, such as the rate of (*message/sec*) and the size of messages in lines. A detailed comparison is shown in Figure 1(a) to examine the impact of Spark workload size (number of tasks) on the neural networks model and compares it with the nearest neighbor, decision tree, and SVM. Six Spark streaming workload sizes with the same configurations are examined for sensitivity analysis, which are 1k, 10k, 20k, 30k, 40k, and 50k tasks. From Figure 1(a), it is evident that the neural networks model outperforms all the other algorithms.

We examine a baseline experiment to prove that there is an important need for a solution to find the optimal dataset size and configuration parameters of stream workload to train the anomaly detection model to generalize the model to detect anomalous behaviors in in-memory Big Data systems. Figure 1(b) shows some design factors and response variables (F-score) for different streaming workload configurations where the proposed neural network is trained with a single combination of configurations parameters (e.g., rate $r$ and size $s$) and test it against all the other workloads

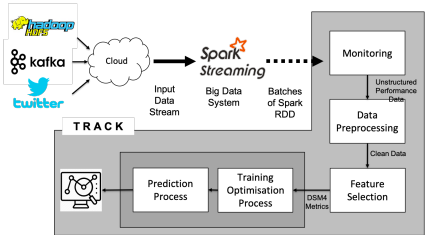



**Figure 2: TRACK Methodology for anomaly detection)**

stream configurations, which include rates (1,8,16, and 32) and sizes (1,10,100, and 1000). As can be seen from Figure 1(b), it is not apparent which set of workload configurations that can be used to efficiently train the machine learning model to achieve the highest accuracy with less time consuming to train the model and detect the anomalous performance in the Spark streaming system. With WordCount Spark streaming application (only two parameters ), it is also challenging to find the ideal dataset size to efficiently train the anomaly detection model to comprehensively cover all the seen types of anomalies.

## 4 METHODOLOGY

In this section, we introduce TRACK, which is Bayesian Optimization (BO) and neural network driven methodology to train and detect performance anomalies in Apache Spark streaming systems. Figure 2 shows the TRACK processes of anomaly detection for the proposed method. The following subsections give a brief information about BO, neural network, training, testing, and feature selection.

### 4.1 Neural Network Model

The proposed neural networks model in [1] with backpropagation and conjugate gradient are used to train the neural networks to update values of weights and biases in networks. The scaled conjugate is used because it is usually faster than other gradient algorithms [7], especially for time-dependent applications such as real time stream processing. The neural networks model uses a *Sigmoid* transfer function ($\sigma(x) = 1/(1 + e^{-\boldsymbol{x}})$) as an activation function, and *Softmax* transfer function is used in the output layer to handle classification problems with multiple classes. For cost function, *cross-entropy* is used to evaluate the performance of neural networks. *Cross-entropy* is used because it has significant, practical advantages over squared-error cost function [5].

The proposed neural networks contain three layers. The first layer is the input layer that includes a number of neurons equal to the number of input features. The second layer is the hidden layer, which has a number of neurons that is determined by using a *trial and error* method, choosing a number between the sizes of input features $n_i$ and output classes $n_o$ [9]. The hidden layer size between $n_i$ and $n_o$ satisfies our goal in achieving accurate results. The output layer 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.

### 4.2 Bayesian Optimization

The proposed methodology revolves around using BO to find the optimal dataset size and configuration parameters to train the neural networks to generalize the model to detect the anomalous behaviors in the Spark streaming system.

When doing Bayesian optimization, there are two main choices to make, which are prior over functions and acquisition function.

*Expected-Improvement* function in [6] is used as an acquisition function to evaluate the expected amount of performance improvement in the neural network detection model $f(x)$ and ignore any values that cause an increase in the error rate of the model. In other words, $x_{best}$ is the location of the smallest posterior mean (optimal workload configuration), and $\mu_Q(x_{best})$ is the smallest value of the posterior mean. The expected improvement can be described as follows $EI(x) = E_Q[max(0, \mu_Q(x_{best}) - f(x))]$. $E_Q$ indicates the expectation taken under the posterior distribution given evaluations of $f$ at $x_1, x_2, ..., x_n$. The time to assess the objective function may vary over some range of points depending on the region [6].

### 4.3 Model Training and Testing

The Streaming workload configurations consist of all possible combinations of configuration parameters of $Rate_n$ and $Size_m$, which in total will be $n * m$ combinations ($n * m$ *DSTrain*). The training part of the dataset (*DSTrain*) is divided into 10 equal subsets to find the ideal size dataset. For example, the dataset *DSTrain* workload configuration with rate $r_i$ and size $s_j$ is divided into 10 subsets according to $DSTrain_{r_i,s_j} = DSTrain_{r_i s_j,1} + .... + DSTrain_{r_i s_j,10}$.

The total number of all possible data subsets is $n*m*10$, which is hard and time consuming to find the optimal configuration combinations parameters and dataset size to train the model. More details information is presented in Algorithm 1. To assess the proposed model, we use a well-known standard classification performance metric, which is F-score (F).

**Algorithm 1:** Training and testing methodology for TRACK

**Input:** Predefined anomaly detection performance $\mathcal{F}$, Configuration space $\mathcal{X}$, and system metrics dataset $\mathcal{D}$

**Output:** Optimal trained neural network model $\mathcal{M}$, which is able to identify anomalous performance in Spark streaming platform with highest predefined accuracy with less amount of time

Configuring streaming workload benchmark
Workload generation with configuration space $\mathcal{X}$
Streaming workload from local network $\mathcal{N} \rightarrow$ Spark system
System profiling to collect dataset of performance metrics
Data cleansing and preprocessing $\rightarrow \mathcal{D}$
$DSTrain$ = 75% of $\mathcal{D} \leftarrow$ total training dataset
$DSTest$ = 25% of $\mathcal{D} \leftarrow$ total testing dataset
$F = 0$ and $DSTrain_c$ is empty $\leftarrow$ current f-score and training data
**while** $(F \leqslant \mathcal{F})$ **do**
  $\mathcal{X}_i = EI_PS(\mathcal{X}) \leftarrow$ acquisition function finds next configuration
  $DSTrain_c = DSTrain_c + DSTrain_{X_i} \leftarrow$ adding current dataset to the previous dataset
  $\mathcal{M}$ = TrainNN($DSTrain_c$ ) $\leftarrow$ train neural network model on the new dataset configuration
  $\mathcal{F} = Fscore(\mathcal{M}(\text{DSTest}))$
  Algorithm
**end**

### 4.4 Feature Selection

In this work, we extend our previous proposed method called dataset method four (DSM4), which is built upon a list of Spark performance metrics that are presented in [1]. DSM4 examines the comprehensive internal Spark architecture and Directed Acyclic Graph Spark application by relying on information from the Apache Spark systems, such as Spark executors, shuffle read, shuffle write, memory spill, java garbage collection, tasks, stages, jobs, applications, and execution timestamps for Spark resilient distributed datasets (RDDs). The collected Spark performance metrics are in time series and manually labeled either as normal or anomalous, before passing them as inputs to the proposed model.

## 5 EVALUATION

This section provides an evaluation of the proposed methodology. We use a random search (RD) algorithm as a baseline on the same datasets, which are generated from Apache Spark Streaming system.

To evaluate the accuracy of the proposed anomaly detection methodology, we developed *Network WordCountExp* benchmark, which is a customized benchmark for stream processing Big Data systems to generate our dataset for datasets generation and training purposes. *Wordcount* is a conventional CPU-intensive benchmark and is widely accepted as a standard micro-benchmark for big data plat forms. The workloads are exponentially generated (with exponential distribution). More than 570 experiments and 135 GB of data have been collected from the Spark streaming system, which we used them to evaluate the proposed work. To inject different types of anomalies, an open source tools (*stress-ng*) have been used to evaluate the proposed methodology with Spark streaming system. A list of performance anomalies is used to generate CPU stress, cache thrashing stress, and context switching stress.

## 6 RESULTS

### 6.1 Finding The Ideal Single Workload Configuration For Model Training

From the previous discussion about motivation example (Section 3), there is a need for a solution to find the ideal single workload configuration (e,g,. Rate $r_i$ and Size $s_j$) that can be used to train the proposed anomaly detection model to pinpoint the abnormal behavior with highest possible F-score. This will facilitate the using of single workload configuration to be generalized and used to detect anomalies with the other workload configurations. The Spark Streaming workload is used with all possible combinations of *Rate (1, 8, 16, ,and 32)* and *Size (1, 10, 100, and 1000)*, which are 16 combinations in total.

A Bayesian optimization and Neural Networks models (described in Section 4.2 and 4.1) are used to address the need of determining

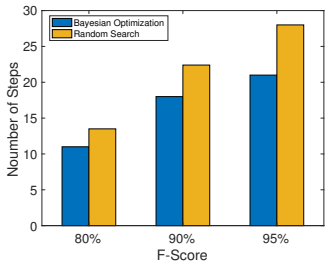



**Figure 3: Comparison between Bayesian Optimization and Random Search to reach a predefined F-Score with the most minimum number of steps. Workload has all the possible combination of parameters and CPU anomalies.**

the ideal single workload configuration (Rate $r_i$ and Size $s_j$) with the minimum number of running experiments *n*. To get an accurate result, we repeated the experiments for 50 times then calculate the average of *n*. The result shows that the ideal F-score is reached with a minimum number of running experiments (*n=8*), which is less by 50% of the total number of possible configuration (*n=16*).

## 6.2 Bayesian Optimization Model to Train Anomaly Detection Technique

A Bayesian Optimization model (discussed in Section 4.2) is used to find the optimal size of the training dataset and the stream workload configurations set to achieve the highest accuracy with less amount of time in training the proposed anomaly detection model. Figure 3 depicts a comparison between Bayesian Optimization and Random Search to reach a predefined F-Score with the most possible minimum number of training steps from the total number of steps, which are 160 steps. The conducted experiments have workloads that contain normal and CPU anomalous behaviors with all the possible combinations of workload configurations. Figure 3 shows the average of 50 experiments that the neural networks train with Bayesian Optimization to achieve the predefined F-score. With Bayesian Optimization, the trained model reaches 95% F-score in 21 steps, whereas 28 steps using a random search (enhanced by 25%). This proves that the proposed model can reduce the time and computation process by 25%.

There are other two types of anomalies, which may disrupt the performance of the stream processing system. These two types are cache thrashing and context switching. The proposed model can detect the cache thrashing and context switching anomalies with F-score equal 80% and 95% respectively. The proposed model outperforms the Random Search by more than 25% and can reduce the amount of computations from 160 experiments to 14.

## 6.3 New Unseen Workload Configurations

The goal of this section is to train the proposed model on predefined workload configurations (*Rate (1, 8, 16, ,and 32)* and *Size (1, 10, 100, and 1000)*) and generalize the model to perform accurately as well with unseen new workload configurations (e,g,. $r_i = 20$ and $s_j = 150$). In this case, the workload is more realistic and reflects the workload characteristics of the stream processing system.

For the training phase, the same Bayesian Optimization and Neural Networks configurations in Section 6.2 are used to train the model on predefined workload configurations (*Rate (1, 8, 16, and 32)* and *Size (1, 10, 100, and 1000)*) to reach 95% F-score for detecting the CPU performance anomalies. For the testing phase, the final model of the training phase is used to detect anomalous behavior but with new unseen workload configurations. Rate can be between 1 to 32 and size can be between 1 to 1000. With the three anomalous workloads (CPU, cache thrashing, and contexts switching), the F-score and standard deviations are 0.93±0.01 for CPU, 0.77±0.02 for cache thrashing, and 0.72±0.04 for context switching. The proposed anomaly detection model has the ability to be trained on 16 workload configurations to be generalized to detect anomalies against 32000 different workload configurations.

# 7 CONCLUSION

This paper contributes to addresses the challenge of anomalous identification by proposing a new hybrid learning solution, TRACK, for anomaly detection in in-memory Big Data systems. The anomaly detection and tuning method are developed using Bayesian Optimization and neural networks to train the model with a limited budget and computing resources. As can be seen from the experimental results, the proposed model can find the optimal training dataset size and configuration parameters to accurately identify different types of performance anomalies in Big Data systems. The proposed model achieves the highest accuracy (95% F-score) and reduces the execution time by 80%. A validation based on a real dataset for the Apache Spark streaming system is provided to demonstrate that the proposed methodology identifies the performance anomalies, the ideal configuration parameter, and training dataset size with up to 75% fewer experiments. In addition, the proposed model can be easily generalized to cover unforeseen workload configurations.

# ACKNOWLEDGMENTS

This research is funded by KACST in Saudi Arabia and the European Union's Horizon 2020 research and innovation program under grant agreement No. 825040 (RADON).

# REFERENCES

[1] A. Alnafessah and G. Casale. 2019. Artificial neural networks based techniques for anomaly detection in Apache Spark. *Cluster Computing* (2019), 1–16.
[2] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Osdi*, Vol. 10. 24.
[3] S. M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie. 2016. High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning. *Pattern Recognition* 58 (2016), 121–134.
[4] E. W. Fulp, G. A. Fink, and J. N. Haack. 2008. Predicting Computer System Failures Using Support Vector Machines. *WASL* 8 (2008), 5–5.
[5] D. M. Kline and V. L. Berardi. 2005. Revisiting squared-error and cross-entropy functions for training neural network classifiers. *Neural Computing & Applications* 14, 4 (2005), 310–318.
[6] Mathworks. 2019. Bayesian Optimization Algorithm. https://uk.mathworks.com/help/stats/bayesian-optimization-algorithm.html#bva8tie-1 visited on 1-10-2019.
[7] M. F. Møller. 1993. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks* 6, 4 (1993), 525–533.
[8] W. Qi, Y. Li, H. Zhou, W. Li, and H. Yang. 2017. Data Mining Based Root-Cause Analysis of Performance Bottleneck for Big Data Workload. In *2017 IEEE HPCC Conferencece*. IEEE, 254–261.
[9] K. G. Sheela and S. N. Deepa. 2013. Review on methods to fix number of hidden neurons in neural networks. *Mathematical Problems in Engineering* 2013 (2013).

# AI Driven Methodology for Anomaly Detection in Apache Spark Streaming Systems

Ahmad Alnafessah, and Giuliano Casale
*Imperial College London London, UK*
*a.alnafessah17@imperial.ac.uk , g.casale@imperial.ac.uk*

**ABSTARCT**
Cloud computing, Artificial Intelligence, and Big Data technologies have recently become one of the most impactful forms of technology innovation. It is common to have multiple users share the same computing resources. This practice noticeably leads to performance anomalies. For instance, some applications can feature variability in processing time due to interference from other applications, or software contention from the other users, which may lead to unexpectedly long execution time and be considered anomalous. There is an urgent need for an automated effective performance anomaly detection method that can be used within the production environment for the streaming system to avoid any late detection of unexpected system failures. To address this challenge, we introduce a new black-box training workload configuration optimization with a neural network driven methodology to identify anomalous performance in an in-memory Spark streaming Big Data platform. The proposed methodology effectively uses Bayesian Optimization to find the ideal training dataset size and Spark streaming workload configuration parameters to train the anomaly detection model. The proposed model is validated on the Apache Spark streaming system. The results demonstrate that the proposed solution succeeds and accurately detects many types of performance anomalies. In addition, the training time for the machine learning model is reduced by more than 50%, which offers a fast anomaly detection deployment for system developers to utilize more efficient monitoring solutions.

***Keywords*:** Performance anomalies, Apache Spark, Big data, Machine learning, and Artificial intelligence.

P1.01

# AI Driven Methodology for Anomaly Detection in Spark Streaming System


**Ahmad Alnafessah[1], and Giuliano Casale[1]**

[1] Imperial College London London, UK

a.alnafessah17@imperial.ac.uk , g.casale@imperial.ac.uk



3rd ICCAIS 2020 International Conference on Computer Applications & Information Security

ICCAIS 2020

http://www.iccais.tech/


## 1. Introduction

- There is an urgent need for an effective performance anomaly detection method that can be used within the production environment to avoid any late detection of unexpected system failures.
- Among various Big Data technologies, in-memory processing technology, such as Apache Spark, has become widely adopted by industries because of its speed, generality, ease of use, and compatibility with other Big Data systems. We here consider Spark based streaming workloads.

## 2. Motivation and Background

- With the growing complexity of Big Data systems, failure management requires significantly higher levels of automation and attention. While some studies address the challenges of performance anomaly detection for batch processing [1], there is a lack of effective automated performance anomaly detection solutions specifically built for Spark streaming systems. Figure 1 shows some design factors and response variables (F- score) for different streaming workload configurations where the proposed neural network is trained with a single combination of configurations parameters and test it against other workloads stream configurations.

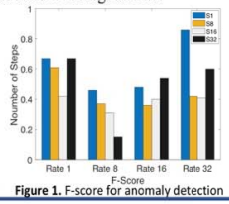



**Figure 1.** F-score for anomaly detection

## 3. Major Contributions

- This study contributes to address the challenge of anomaly identification by investigating new hybrid learning techniques for anomaly detection in in-memory Big Data systems within cloud computing. We developed a solution, which has a tuning method capable of training a machine learning model with a limited budget and limited number of experiments. Our methodology revolves around using artificial neural networks with Bayesian Optimization (BO) to find the optimal training dataset size and configuration parameters to efficiently train the model to achieve the highest accuracy (95% F-score) and reduce the training time by 80%. A validation based on real datasets from Apache Spark streaming system is provided to demonstrate that the proposed methodology identifies performance anomalies, the ideal configuration parameter, and the optimal training dataset size while reducing the number of experiments by 75%.

## 4. Methodology

- Neural Network Model: Neural network model with backpropagation and conjugate gradient are used to train the neural networks. The scaled conjugate is used because it is usually faster than other gradient algorithms [2], especially for time-dependent applications such as real time stream processing.
- Bayesian Optimization: The proposed methodology revolves around using BO to find the optimal dataset size and configuration parameters to train the neural networks to generalize the model to detect the anomalous behaviors in the Spark streaming system.
- Feature Selection: In this work, we extend the proposed method called dataset method four (DSM4), which is built upon the list of Spark performance metrics that are presented in [1].

## 5. Evaluation

- A schematic view of anomaly detection detailed processes is shown below:

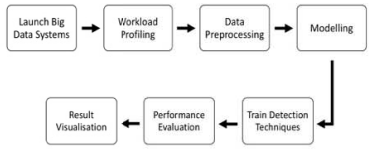
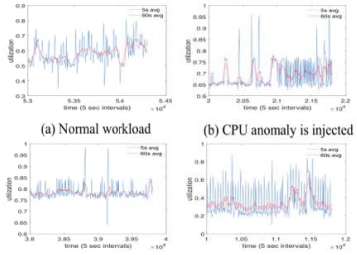



**Figure 2.** Anomaly detection processes

- Workload Generation : More than 570 experiments have been conducted and 135 GB of data have been collected from the Spark streaming system, which we used them to evaluate the proposed work.
- Anomaly Injection: A list of performance anomalies is used to generate CPU stress, cache thrashing stress, and context switching stress.



(a) Normal workload (b) CPU anomaly is injected

(c) Context switching anomaly is injected (d) Cache thrashing anomaly is injected

**Figure 3.** CPU utilization for Spark streaming workload with normal and anomalous performance.

## 6. Results

- The proposed methodology is evaluated on an isolated Spark streaming system. We avoid using a virtual Spark system to make sure that all the performance metrics are accurately measured.
- Testing the proposed model against the three types of anomalies:

| Types of Anomalies | F-score |
|---|---|
| CPU | 99% |
| Cache Thrashing | 95% |
| Context Switching | 80% |

**Table 1.** Performance evaluation of proposed model

- Finding the ideal workload configuration for training:

| Algorithms | 80% F | 90% F | 95% F |
|---|---|---|---|
| BO | 11 | 18 | 21 |
| RS | 13.5 | 22.4 | 28 |

**Table 2.** Comparison between Bayesian Optimization and Random Search to reach a predefined F-Score with the minimum number of steps.

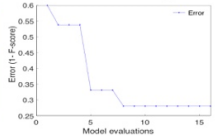



**Figure 4.** Training processes result of each workload configuration and test it on all possible combinations workload configuration.

## 8. Conclusions

- The proposed solution not only identifies anomalous with high F-score, but also saves considerable time in training the model. In addition, the proposed model can be easily generalized to cover unforeseen workload configurations.

## 9. References


1. Alnafessah, A., & Casale, G. (2019). Artificial neural networks based techniques for anomaly detection in Apache Spark. *Cluster Computing*, 1-16.
2. Møller, M. F. (1990). *A scaled conjugate gradient algorithm for fast supervised learning*. Aarhus University, Computer Science Department.

# Anomaly Detection for Big Data Technologies

**Ahmad Alnafessah**
Department of Computing, Imperial College London, United Kingdom
a.alnafessah17@imperial.ac.uk

**Giuliano Casale**
Department of Computing, Imperial College London, United Kingdom
g.casale@imperial.ac.uk

## Abstract

The main goal of this research is to contribute to automated performance anomaly detection for large-scale and complex distributed systems, especially for Big Data applications within cloud computing. The main points that we will investigate are:

- Automated detection of anomalous performance behaviors by finding the relevant performance metrics with which to characterize behavior of systems.
- Performance anomaly localization: To pinpoint the cause of a performance anomaly due to internal or external faults.
- Investigation of the possibility of anomaly prediction. Failure prediction aims to determine the possible occurrences of catastrophic events in the near future and will enable system developers to utilize effective monitoring solutions to guarantee system availability.
- Assessment for the potential of hybrid methods that combine machine learning with traditional methods used in performance for anomaly detection.

The topic of this research proposal will offer me the opportunity to more deeply apply my interest in the field of performance anomaly detection and prediction by investigating and using novel optimization strategies. In addition, this research provides a very interesting case of utilizing the anomaly detection techniques in a large-scale Big Data and cloud computing environment. Among the various Big Data technologies, in-memory processing technology like Apache Spark has become widely adopted by industries as result of its speed, generality, ease of use, and compatibility with other Big Data systems. Although Spark is developing gradually, currently there are still shortages in comprehensive performance analyses that specifically build for Spark and are used to detect performance anomalies. Therefore, this raises my interest in addressing this challenge by investigating new hybrid learning techniques for anomaly detection in large-scale and complex systems, especially for in-memory processing Big Data platforms within cloud computing.

**2012 ACM Subject Classification** Computing methodologies → Anomaly detection

**Keywords and phrases** Performance anomalies, Apache Spark, Neural Network, Resilient Distributed Dataset (RDD)

**Digital Object Identifier** 10.4230/OASIcs.ICCSW.2018.8

**Category** Poster Track


2018 Imperial College Computing Student Workshop (ICCSW 2018).
Editors: Edoardo Pirovano and Eva Graversen; Article No. 8; pp. 8:1–8:1
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2018 International Conference on the Quality of Information and Communications Technology

# A Neural-Network Driven Methodology for Anomaly Detection in Apache Spark

Ahmad Alnafessah
*Department of Computing*
*Imperial College London, UK*
a.alnafessah17@imperial.ac.uk

Giuliano Casale
*Department of Computing*
*Imperial College London, UK*
g.casale@imperial.ac.uk

***Abstract*—In cloud computing services, multiple tenants sharing the same computing resources can cause performance anomalies, either due to normal or malicious user behaviors. When a Big Data application is deployed over a private or public cloud and does not perform as well as expected, it is however challenging to reliably detect a performance anomaly and minimize its consequences. We here consider in particular Spark-based workloads, in which the analytic operations are applied to a resilient distributed dataset (RDD). We develop a neural network based methodology for anomaly detection based on knowledge of the RDD characteristics. Using experiments against multiple workloads and anomaly types, we show that our method improves over other types of classifiers as well as against black box performance anomaly detection.**

***Index Terms*—Performance anomalies, Apache Spark, Neural Network, Resilient Distributed Dataset (RDD)**

## I. Introduction

Anomalies can arise for various reasons, such as interference from other applications and software bugs. Any late detection and manual resolutions of anomalous behavior in systems may negatively impact cloud and Big Data systems because they may cause prolonged performance violations with a significant financial penalty [1] [2]. Therefore, performance anomaly detection in cloud computing and Big Data systems must be dynamic and proactive in nature [3].

Many effective solutions leverage the power of machine learning techniques to identify security and performance anomalies. These techniques offer the ability to quickly sift through massive metric space to identify normal and anomalous patterns [4]. Based on the nature of input data and expected output, machine-learning algorithms are classified into two main categories: supervised learning and unsupervised learning [3]. Some machine learning techniques include classification-based, neighbor-based, and clustering-based.

Classification techniques are a special case of supervised learning. The aim of these techniques is to determine whether the instances in a given feature space belong to a specific class or to multiple classes [3]. There are well-known classification techniques for anomaly identification, such as neural networks, support vector machines, and Bayesian networks [5]. The classification technique is significantly affected by the accuracy of the labeled data and algorithms that have been used. For example, training and testing phases for decision trees are usually faster than for support vector machines, which involve quadratic optimization.

There are various popular open source distributed data-processing frameworks related to Big Data technologies, such as Hadoop MapReduce, Apache Storm, and Apache Spark. Among these various Big Data technologies, in-memory processing technology like Apache Spark has become widely adopted by industries because of its speed, generality, ease of use, and compatibility with other Big Data systems [6]. Although Spark is developing gradually, there are currently still shortages in comprehensive performance analyses that are specifically developed for Spark and are used to detect performance anomalies [7]. The performance of in-memory processing frameworks can vary considerably depending on many factors, such as the type of input data, data size, application design, system configuration, algorithms used, and available computing resources [7] [8]. These factors make anomaly detection more challenging, especially for critical applications in such distributed systems. Therefore, there is a challenging need to deeply investigate in-memory processing technology like Spark performance bottlenecks and to pinpoint the cause of a performance anomaly to improve it.

The main contribution of this paper is developing a neural network based methodology for anomaly detection that is able to improve the accuracy of anomaly detection based on knowledge of the RDD characteristics. The conducted experiments and results demonstrate that the proposed method works effectively and efficiently with complex scenarios and anomalies, such as CPU contention, memory contention, cache thrashing and context switching anomalies. Compared with other popular methods, the random instant and random duration of anomalies did not impact the performance of our proposed methodology.

The rest of the paper is organized as follows: some existence works are discussed in Section II. Basic information about in-memory Big Data technology is provided in Section III. The methodology of this work is presented in Section IV, and the evaluation of methodology is discussed in Section V. All the experimental results and findings are presented in Section VI, and Section VII gives the summary and future work.

978-1-5386-5841-3/18/$31.00 ©2018 IEEE
DOI 10.1109/QUATIC.2018.00038

## II. Related Work

System performance is often described in terms of the time taken to process some tasks or the set rate of tasks performed with a given amount of computing resources that are consumed within a given observation period [3]. With the growing complexity and dynamicity of Big Data and cloud systems, failure management requires significantly higher levels of automation and attention [9]. Performance anomalies have become a major concern for academic researchers of Big Data technologies over cloud computing services. The ability to analyze data is vital to the process of detecting anomalous behavior to resilience in production systems in spite of noise or risks arising in the production testbed (e.g., a cloud computing environment). Anomalous performance can occur as a result of service operator faults [10], software failures and user errors [11], environmental issues, and security violations [3], among others.

Many anomaly detection studies have been specifically conducted for certain application domains, while others are more generic. The survey in [12] extensively investigates many techniques that have been developed in statistical analysis and machine learning domains, especially for anomaly detection techniques. The study conducted in [13] provided a structured and comprehensive overview of the research on anomaly detection that grouped existing techniques into different categories based on the underlying approach that they each adopted.

Data mining and machine learning technologies have received growing attention for performance anomaly detection and diagnosis by the research community. Based on the nature of the input and the expected output, supervised learning or unsupervised learning may be used. A basic anomaly detection system observes the performance behaviors of the targeted system to collect measurements to generate essential profiles about normal system performance [3]. This observation will continue to detect any undesirable deviation or anomaly in performance and apply a root cause analysis to pinpoint causes of a performance anomaly due to internal or external faults.

Machine learning classification techniques are used to classify an input features into predefined classes of items in order to construct a classifier that can predict the class of each item in the dataset according to the class labels of this dataset. There are well-known classification techniques for anomaly identification, such as neural networks, decision tree, nearest neighbor, support vector machines, and Bayesian networks [5]. The classification technique is significantly affected by the accuracy of the labeled data and algorithms that have been used.

In a supervised learning technique, the training dataset is assumed to be available for use and contains correctly-labeled instances to distinguish between anomalies and normal classes. An example of a supervised technique is used in [14]. According to Gu and Wang [14], a stream-based anomaly behavior-detection technique for online application has been used to detect anomaly indications that relate to performance anomaly root localization. They apply Bayesian classification methods to detect an anomaly and its root. In addition, the authors [14] apply Markov models to detect the change in the patterns of different measurement metrics.

The local outlier factor (LOF) algorithm is a type of neighbor-based technique of an unsupervised learning algorithm. The LOF algorithm was employed to detect anomalous behavior in cloud computing systems in [15]. The main idea is to identify anomaly by comparing the density of each instance. Any instance in low density is considered an anomaly. To improve performance anomaly detection, the LOF requires a significant effort in collecting a complete training dataset of normal behavior for applications over cloud computing before the detection phase. This dataset is sometimes unavailable over online applications. Therefore, the authors [15] use an adaptive anomaly detection scheme for a cloud system based on the LOF.

Backpropagation neural network is a popular technique for many classification problems. This is because the neural network is a data-driven self-adaptive method that can adjust itself to the datasets without requiring knowledge about the distribution or function of the used model [16]. Another reason for the using neural network is that it has the ability to approximate any function with arbitrary accuracy. This has caused the neural network to be considered a universal functional approximation [16].

Learning and generalization are considered to be the most prominent topic in the neural network. Learning refers to the ability to approximate the underlying adaptive behavior from the training dataset. The generalization is one of the main advantages of using the neural network, which offers the ability to generalize the network by classifying (predicting) the class of the input dataset from the same classes of the training dataset, even if that input item has never been seen before [16]. This feature allows the classifier to achieve a desired accuracy level when classifying new or unknown objects.

While anomaly detection studies have been conducted in the literature for many purposes, there are not enough research studies that address anomaly detection and prediction issues, especially for in-memory Big Data technologies, such as Apache Spark.

## III. Background on Apache Spark

Apache Spark is a large-scale in-memory processing technology that can support both batch and stream processing data, which can make it easy with a low cost to support different types of workloads on the same engine in a production environment [6]. This type of system is an example of a micro-batch system. Spark offers general solutions for different data processing types and it provides built-in tools to support other services, such as graph analysis and machine learning. The main goal of Apache Spark is to speed up the batch processing of data by utilizing in-memory computation. According to Apache Spark, Spark can be up to 100 times faster than Hadoop MapReduce for in-memory analytics [6].

Apache Spark provides a general purposes engine for different kinds of computation, including iterative algorithms, job batches, streaming, and interactive queries. These different types of computation were previously difficult to find supported altogether in the same distributed system [17]. Beyond the ability to perform batch and stream processing, Apache Spark also has a rich library that is built on top of Spark core engine [18].

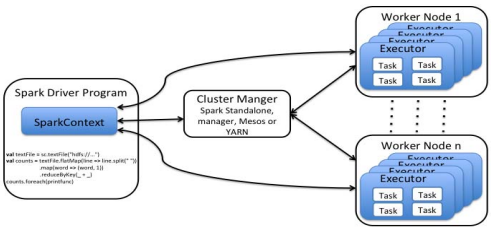



Fig. 1. Spark Application Components

*1) Apache Spark Application Architecture:* 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 and coordinates 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 recourses for the application. The cluster manager can be Mesos, YARN, or a Spark stand-alone cluster [6].

*2) Resilient Distributed Datasets:* The anomaly detection method we propose performs in its most effective instantiation anomaly detection at the level of Spark's Resilient Distributed Datasets (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 parallelized 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()) [6].

Every Spark application (e.g., Fig.2) consists of jobs (Fig.3(a)), and each job is further divided into stages (Fig.3(b)) that depend on each other. Each stage is then composed of a collection of tasks (Fig.3(c)) [19]. A Spark job is created when the 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 [20].

Breaking the RDD DAG at shuffle boundaries will create the 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.). On the other hand, the stages inside the single job are divided into many stages, which depend on each other when there are RDD transformation operations that require shuffling which is called wide dependency (e.g., group-by, join, etc.) [20]. 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. Fig.3(b) shows how the job is divided into two stages as a result of shuffle boundaries.

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. 3(c).

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 [19]. Inside a single stage, the number of tasks is determined by the number of the final RDD partitions in the same stage.

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
counts.foreach(printfunc)
```

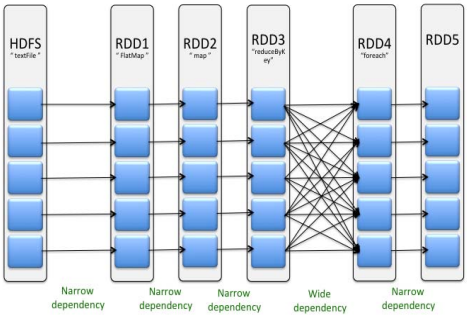
Fig. 2. Simple WordCount Example

## IV. Methodology

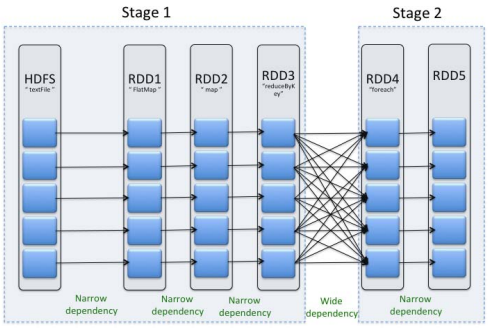
The main goal of this research is to contribute to performance anomaly detection for Big Data applications within cloud computing. In this section, we present our neural network driven methodology for anomaly detection in Apache Spark environment. This methodology covers monitoring the environment, model building, training, and testing.

### A. Monitoring

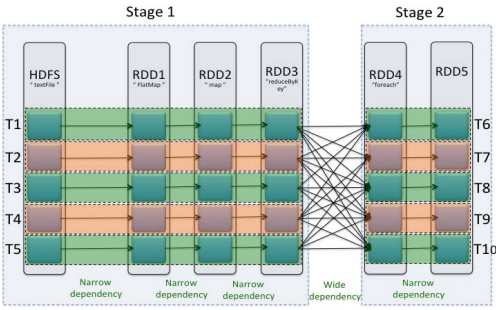
The monitoring process is needed to observe the system to identify the normal and anomalous behaviors. This includes




(a) Type of dependencies: Narrow and Wide




(b) A job is divided into two stages, which second stage has wide dependency on the first stage




(c) Stages are divided into tasks that are equal to the number of partitions in the same stage.

Fig. 3. Simple DAG for WordCount example.

monitoring all machines and activities of that system. These activities may be related to tasks, stages, jobs, CPU, memory, network, I/O, etc. Many metrics can be collected, but it is challenging to decide which metrics are more valuable to assess system performance and pinpoint the anomalous behaviors.

In the beginning, the utilizations of system resources were measured at one-second intervals. The CPU measurements were collected, which are summarized in Table I. After observing the Spark system, we notice that the memory performance was noticeably impacted during the occurrence of different types of anomalies. Therefore, some memory metrics were also collected, which include the metrics in Table I.

Other performance data were collected from Apache Spark log files to provide an understanding of how Spark works to launch jobs and tasks to executors. This data contains comprehensive statistics about the Spark metrics, building upon the list of metrics selected in [21], and includes information about tasks, RDD, stages, jobs, and applications. The measurements of the internal Spark metrics are summarized in Table I. All data collection took place in the background without causing any noticeable overhead on the Spark Cluster.

It is crucial to reduce the number of input features for the classifier to achieve satisfactory accuracy with less amount of computation in the model [16]. Therefore, the collected data is pre-processed to eliminate any mislabeled training instances and validate the datasets before passing them to the neural networks to improve the quality of datasets. For example, any metric is messing or any utilization measurements are larger than 100% or less than 0% will be removed.

All the collected metrics are labeled either as normal or anomaly before passing them as input to our anomaly detection method for training, validation, and testing. During the experiment, Spark cluster is artificially injected by anomaly and at the same time the comprehensive information about anomaly is recorded for labeling purposes for training phase of machine learning algorithms. This information about anomalies includes start time, end time, and type of anomalies. These labels are used to classify performance to be either a normal or anomalous behaviors in the training phase of the neural network. In cases where production data has known anomalies, then such labeling step is not needed.

### B. Build Model

We built our model using backpropagation neural networks with scaled conjugate gradient for training process to update weight and bias values according to the scaled conjugate gradient method. To define a pattern recognition of anomalies, the collected performance metrics (discussed in Section IV-A) are arranged as input columns in a matrix. Then, the set of labels is arranged in a matrix that has one column which contains either 0 for normal or 1 for an anomaly.

Before we initiate the backpropagation process, we calculate the activation values of units in the hidden layer and propagating them to the output layer. *Sigmoid* transfer function is used in the hidden layer to introduce nonlinearity in the model of neural networks. Then the *Cross Entropy* is used as a cost function to assess the network performance and compare the actual output error results with the desired output values (labeled data). The *Cross Entropy* is used because it has practical advantages over other cost functions, such as mean squared error. It can maintain the performance even for problems with limited data [22].

Backpropagation algorithm is used to allow the information about classification error to propagate back through the network to compute the gradient. This provides a significant effect of changing the weights and biases based on the behavior of

TABLE I
LIST OF PERFORMANCE METRICS THAT ARE COLLECTED

| Resources | Metrics |
|---|---|
| CPU | • CPU utilization<br>• percentage of time that the CPUs were idle during outstanding disk I/O request<br>• percentage of time spent in involuntary wait by the virtual CPU<br>• percentage of time that the CPUs were idle |
| Memory | • kbmemfree: free memory in KB on hostname<br>• kbmemused: used memory in KB on hostname<br>• X.memused: used memory in % on hostname<br>• kbbuffers: buffer memory in KB on hostname<br>• kbcached: cached memory in KB on hostname<br>• kbcommit: commited memory in KB on hostname<br>• X.commit: commited memory in % on hostname<br>• kbactive: active memory in KB on hostname<br>• kbinact: inactive memory in KB on hostname<br>• kbdirty: dirty memory in KB on hostname |
| Spark Logs | • Stage info: Stage ID, Executor Deserialize CPU Time, Executor Deserialize Time,<br>• Executor Run Time, Executor CPU Time, Finish Time of stage, Launch Time of stage<br>• Task spill: Disk Bytes Spilled, Memory Bytes Spilled<br>• Task shuffle read: Local Blocks Fetched, Local Bytes Read, Remote Blocks Fetched, Remote Bytes Read, and Fetch Wait Time<br>• Task Shuffle Write: Shuffle Write Time, Shuffle Bytes Written<br>• Garbage Collection: JVM GC Time |

network and cost function. Scaled conjugate gradient back-propagation is used for training process to updates weight and bias values according to the scaled conjugate gradient method. The Scaled conjugate gradient training method is used because it is much faster than standard gradient descent algorithms [23].

### C. Training

In training process, all the collected metrics of Spark tasks and RDD have labels that specify if the task has been affected by the injected artificial anomalies or not. These labels are used to train the neural networks in detecting pattern of anomalies based on knowledge of the tasks and RDD characteristics. All the performance metrics about Spark RDD and tasks that are discussed in Section IV-A will be passed as inputs to the input layer of the neural network. The output layer on neural network contains a single neuron because there is a single target value that generates either 0 for normal behavior or 1 for anomalous behavior.

The input dataset to the model is divided into three subsets for training, validation, and testing. The first subset is used for calculating the gradient and updating the network weights and biases. During the training process, the weights and biases are constantly updated until the magnitude of scaled conjugate gradient reaches the minimum performance gradient or number of validation checks. Therefore, the training process will stop if the magnitude of the gradient is less than *1e-5*. This is the default limit and it can be adjusted.

The second subset is used for validation purpose where the error rate is decreased before overfit dataset by checking the number of validation checks. The number of validation checks is about the number of successive iterations that the validation performance fails to decrease. By default, this number is set to be 6. If the error rate is not decreasing for 6 successive iterations, the training will enforce to stop. At this point, we saved the weight and biases at the minimum error for the validation subset. This method called *early stopping* [24]. This will avoid issues of overfitting dataset. The test subset is independently used to assess generalization of the method.

### D. Testing

We test our proposed model with new input dataset that was collected from the system. This new dataset was manually labeled in order to classify normal and anomalous behavior of target system. With this kind of classification techniques, accuracy is not a sufficient evaluation for a model with an imbalanced class distribution of data [25]. Therefore, sometimes the accuracy estimation may not correctly reflect the quality of the classifier. To avoid this issue, *sensitivity* and *precision* measures are used to evaluate the anomaly detection classifiers, which are standard metrics for quantifying the accuracy of the classifiers [26]. The following are the anomaly classification classes and their notations:

- True Positive (*tp*): The detection method correctly detected anomaly
- True Negative (*tn*): The detection method correctly did not detect anomaly when it did not exist.
- False Positive (*fp*): The detection method detected anomaly when it does not exist
- False Negative (*fn*): The detection methods missed detection of an anomaly when it actually exists

$$R = \frac{tp}{tp + fn} \tag{1}$$

$$P = \frac{tp}{tp + fp} \tag{2}$$

$$F_1 = 2\frac{PR}{P + R} \tag{3}$$

*Sensitivity* is also called *Recall*, which assesses the quality of a classifier in recognizing positive samples; it is defined in (1). *Recall* will become high when the anomaly-detection method can detect all anomalies. The second classification performance metric is *Precision*, which quantifies how many samples are classified as anomalies are indeed anomalies. This is defined in (2). The *Precision* assesses the reliability of the detection method when it reports anomalies [26]. The trade-off between the *Recall* and *Precision* is *F-Score*, which is a summary score, and it is computed as a harmonic means of *Recall* and *Precision*. The *F-Score* metric is defined in (3).

## V. Evaluation

In this section, we introduce an evaluation for our methodology of performance anomaly detection for in-memory processing big data technologies. This can be achieved by generating a real workload with anomalies, collecting performance metrics, applying anomaly detection technique, and evaluating the classification performance metrics to examine the efficiency of the proposed solution. The rest of this section discusses in details the experimental testbed, workload generation, performance metrics collection, and evaluation of our anomaly detection framework.

### A. Experimental Testbed

The experiments were conducted on a cluster that contains three physical servers, which are S01, S02, and S03. The specifications for these servers are as follow:

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

The Apache Spark is deployed on the cluster where S01 is a master and the other two servers are slaves (workers). Spark was configured to use the Spark Standalone Cluster Manager, 36 executors, FIFO scheduler, and a client mode for deployment. Node S01 hosts the SparkBench benchmark to generate the Spark workload and launch Spark jobs. Node S02 is used to inject some anomalies and contentions into the Apache Spark computing environment. To inject these anomalies, some open source tools have been used in our experiments. These tools include *stress* and *stress-ng*. In addition, many performance metrics have been collected from the Spark cluster. We used the *sar* (System Activity Reporter) monitoring tool to evaluate application performance. This tool collects valuable performance data for CPU, memory, I/O, and network. More information about the used benchmark and data collection process is provided in Section (V-B), and Section (V-C) respectively.

### B. Benchmarks and Workload Generation

The effective use of benchmarks offers opportunities to examine and understand the system performance and identify potential areas for improvement and optimization. In general, there are two categories of benchmarks that are framework specific and multiple framework benchmarks. In our research, SparkBench is used because it is designed specifically for Spark to provide a comprehensive set of workloads and to cover the four main categories of Spark application, including graph computation, streaming, SQL query, and the machine learning application.

SparkBench is being developed at the Spark Technology Center at IBM. It is an open source benchmark, and it offers many types of workloads, such as machine learning, graph computation, and SQL queries. In our research, the KMeans data generator is used to generate various KMeans datasets with different sizes (e.g., 2 GB, 8 GB, 32 GB, and 64 GB). The KMeans workload has been intensively used in our experiments with many different configurations of Spark and SparkBench parameters to compare the performance results of different scenarios.

SparkBench provides a reliable feature, which is called workload suite [7]. This suite may include a collection of one or many workloads that can be run either serially or in parallel. Therefore, the user can effectively control the level of workload parallelism to stress the Spark system. For example, the user can run five KMeans workloads and two linear regressions in parallel, then launch serially three SQL workloads. Therefore, the user has the ability to chain together different workloads with different parameter configurations and use many levels of parallelism. This feature of parallelism facilitates the job of developing simulations of a real-world production environment.

### C. Performance Data Collection

The performance metrics are collected from all the machines in the Spark cluster using the *sar* open source monitoring tool in Linux *Sysstat* package to measure the utilizations of system resources at one second intervals. This tool collects valuable performance data from CPU, memory, I/O, and network. The CPU and memory measurements are discussed in Section IV-A. All data collection took place in the background without having any overhead on the Spark system.

In addition, Spark offers a configurable metrics system that allows Spark users to report metrics to a variety of sinks, such as HTTP, JMX, and CSV files. Corresponding to the Spark components (workers, executor, etc.), Spark metrics are separated in different instances. Each of these instances can configure a set of sinks to which metrics are reported [6]. In addition, Spark also offers the ability to use different sets from third-party tools to monitor applications using the metric of the system [27], which can be used in the future.

## VI. EXPERIMENTAL RESULTS

The experiments were conducted on a cluster (described in Section V-A), 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. This guarantees that the contention on CPU, memory, and disk are caused from our side, not from the contention that may occur among VMs in the visualized environment.

The objective is to examine the performance metrics and Spark RDD to detect performance anomalies in in-memory big data framework. A series of experiments were conducted on the Spark cluster to evaluate and address the following research questions:

### A. ***RQ1: How well does the method perform with different time durations of anomalies?***

To address this question, five experiments have been conducted to examine the impact of disruption to CPU and mem-

ory with a different time (continuous and 90sec stress). Only S02 has been injected by 50% CPU and memory contentions. There was no contention in the other servers in the cluster. These experiments cover the following five scenarios that have been examined:

1) Running the benchmark without any contention on CPU and memory
2) Running the benchmark with continuous contention on CPU by 50%
3) Running the benchmark with continuous contention on memory by 50% of free memory
4) Running the benchmark with a short time (90 sec) of contention on CPU 50%
5) Running the benchmark with a short time (90 sec) of contention on memory by 50% of free memory

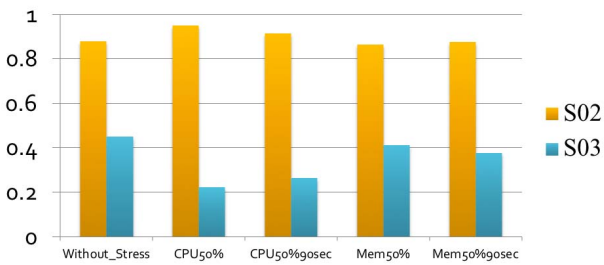



Fig. 4. Mean CPU utilization of S02 and S03

Fig.4 shows the mean CPU utilization of S02 and S03. Table II shows the comparison of results among different contention scenarios on S02. To detect the performance anomalies, CPU metrics have been collected during the runtime of experiments. Machine learning has been used to detect these anomalies. The neural network has been applied with only CPU metrics, and the performance metrics that quantify detection techniques are summarized in Fig.5. CPU utilization of S02 with short 50% CPU contention (90 sec) is shown in Fig.6.

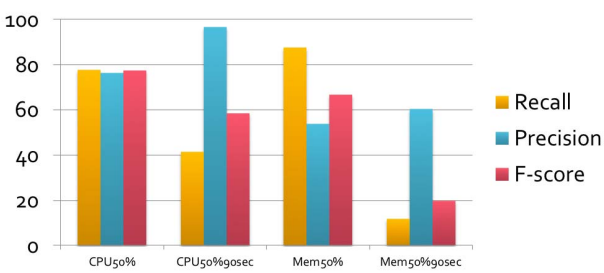



Fig. 5. Performance metrics that quantify detection techniques in section VI-A.

The memory contention for both scenarios (continuous and 90 sec) does not have visible effects on the mean CPU usage as shown in Table II. The KMeens workload does not heavily use memory. Therefore, short time memory contention (90 sec) has not affected mean CPU utilization and has not been effectively detected by the classifier, as it achieved 19.88% F-score. The short time contention on CPU (90 sec) has been detected with an F-score of 58.05%, whereas short time (90sec) contention on memory has been misclassified with an F-Score of 19.88%. This is because: 1) the neural network needs to train the algorithm with a bigger dataset to detect memory contention. 2) short CPU contention "90 sec" has a greater effect on mean CPU utilization, which facilitates the detection of CPU contention.

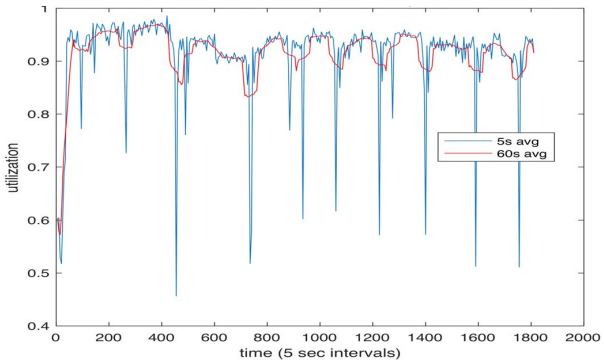



Fig. 6. S02 CPU utilization when single KMeans workloads run on S02 with a short time of CPU 50% stress on S02

### *B. RQ2: How well does the method detect anomalies with different types of performance metrics?*

Two experiments have been conducted to examine anomaly detection techniques using the same configuration of neural networks in RQ1 (Section VI-A). In addition to the CPU metrics, more performance metrics related to memory (discussed in V-C) were collected in order to optimize the performance of anomaly detection methods. In the first experiment, a single KMeans workload was run without any contention. In the second experiment, a single KMeans workload was launched, and the Spark cluster was injected with a continuous CPU anomaly for the duration of the experiment.

The CPU and memory metrics have been collected to apply the machine learning algorithms. After considering both the collected memory and CPU metrics, the performance metrics of F-Score increased to be (%99) for Neural networks. Before adding the memory metrics to the anomaly detection technique, the performance metrics was %77.44 for F-Score. Therefore, the anomaly detection method for Apache Spark is significantly enhanced by using performance metrics for both the CPU and memory.

### *C. RQ3: How well does the method detect anomalies with different parallelism and input data sizes of workloads?*

In this section, two experiments have been conducted. Each experiment runs ten parallel KMeans workloads at the same time. Each workload has an input data size of 64 GB. The first experiment was launched without injecting any contention to the CPU, whereas the second experiment, a continuance 50%

TABLE II
RUNNING SPARK KMEANS WORKLOAD WITHOUT CONTENTION, WITH CONTINUOUS 50% CPU STRESS, WITH 90 SEC 50% CPU STRESS, WITH CONTINUOUS 50% MEMORY STRESS, AND WITH 90 SEC 50% MEMORY STRESS ON ONLY S02.

| Server | Stress | MeanCPU | SD | Pr95 | Pr99 | Iqr | UsedMem | ExeTimeSec |
|---|---|---|---|---|---|---|---|---|
| S01:Non | NO | 0.0203 | 0.0389 | 0.0950 | 0.2147 | 0.0177 | 89.3239 | 295.0000 |
| S01:CPU50% | NO | 0.0174 | 0.0308 | 0.0663 | 0.1646 | 0.0176 | 89.5402 | 567.0000 |
| S01:CPU50%90s | NO | 0.0210 | 0.0359 | 0.0874 | 0.2166 | 0.0218 | 89.8094 | 376.0000 |
| S01:Mem50% | NO | 0.0205 | 0.0376 | 0.0768 | 0.2346 | 0.0211 | 90.0187 | 326.0000 |
| S01:Mem%90s | NO | 0.0193 | 0.0356 | 0.0715 | 0.2094 | 0.0190 | 90.2926 | 355.0000 |
| S02: Non | No | 0.8776 | 0.1849 | 0.9519 | 0.9561 | 0.0304 | 81.2464 | 295.0000 |
| S02:CPU50% | Yes | 0.9510 | 0.0701 | 0.9799 | 0.9833 | 0.0158 | 81.7595 | 567.0000 |
| S02:CPU50%90s | Yes | 0.9152 | 0.0806 | 0.9693 | 0.9748 | 0.0315 | 81.9844 | 376.0000 |
| S02:Mem50% | Yes | 0.8656 | 0.1880 | 0.9479 | 0.9527 | 0.0318 | 93.2561 | 326.0000 |
| S02:Mem50%90s | Yes | 0.8770 | 0.1825 | 0.9513 | 0.9574 | 0.0337 | 85.0864 | 355.0000 |
| S03: Non | NO | 0.4488 | 0.4443 | 0.9489 | 0.9550 | 0.9271 | 90.0702 | 295.0000 |
| S03:CPU50% | NO | 0.2231 | 0.3719 | 0.9361 | 0.9504 | 0.3580 | 90.4513 | 567.0000 |
| S03:CPU50%90s | NO | 0.2649 | 0.3572 | 0.8831 | 0.9356 | 0.6816 | 91.1414 | 376.0000 |
| S03:Mem50% | NO | 0.4129 | 0.4357 | 0.9422 | 0.9507 | 0.9115 | 91.2038 | 326.0000 |
| S03:Mem50%90s | No | 0.3760 | 0.4310 | 0.9402 | 0.9506 | 0.8914 | 91.3892 | 355.0000 |

CPU contention, was injected in S02. The parameters of two experiments were configured, as shown in TableIII.

TABLE III
CONFIGURATION OF SPARK WORKLOAD FOR EXPERIMENT FOR RQ3.

| Exp # | Contention | Data size | K | Seed | Parallelism |
|---|---|---|---|---|---|
| 1 | No Contention | 64GB | 2K | 127L | Yes |
| 2 | 50% CPU Contention | 64GB | 2K | 127L | Yes |

Each experiment took approximately 17 hours for execution. In order to compare the proposed anomaly detection method, three machine learning algorithms have been applied to detect performance anomaly. These algorithms include neural networks, decision tree, and nearest neighbor. Fig.7 depicts that the neural network has the highest F-score, and it selectively detects the anomalies in the Apache Spark cluster. The nearest neighbor is the second highest F-score, then the decision tree. Regarding the time it takes to apply each algorithm, the neural network, decision tree, and nearest neighbor took approximately 1 min, 3 min, and 9 min, respectively. The neural network is more effective than the other algorithms. The results in Fig.7 prove that the three algorithms were affected by the size of the input data to workloads when the input data were increased to 64 GB. The neural network is affected by 1%, decision tree by 6%, and nearest neighbor by 4%. Moreover, there were no significant effects of parallelism on neural networks for detecting the anomalies in Apache Spark cluster.

### D. *RQ4: To what extent can the detection method identify anomalies with random occurrence and duration?*

To have more complex and realistic experiments, three experiments with three different type of anomalies were injected into the Spark cluster with random instant and random duration from 0sec to 240sec. These anomalies include CPU, cache thrashing and context switching. For the cache thrashing scenario, many processes were launched to perform random wide spread memory read and writes to thrash the CPU cache. In order to inject context switching anomalies, many processes

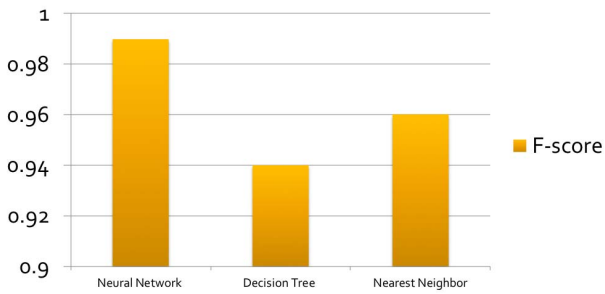



Fig. 7. Performance metrics for machine learning algorithms with 50%CPU contention on only S02

were launched to force context switching. The three experiments have been conducted to examine anomaly detection techniques using the same configuration Spark workload in Table III.

This time, more performance data about the internal metrics of Spark were collected from Apache Spark log files. This includes information about RDD, tasks, stages, jobs, and applications (discussed in Section IV-A). Two machine learning algorithms were used to compare the performance of detection methods. These algorithms include neural networks and nearest neighbor. Fig.8 shows performance metrics of F-score for neural networks and nearest neighbor for anomaly detection. From Fig.8, it clear that the neural networks outperform the nearest neighbor algorithm in detecting all the three types of anomalies. Therefore, the random instant and random duration of the three types of anomalies did not impact the performance of neural networks compared with the nearest neighbor.

## VII. CONCLUSION

This paper provides a challenging case of utilizing the anomaly detection techniques in a complex Big Data and cloud computing environment. Among the various Big Data technologies, in-memory processing technology like Apache Spark has become widely adopted by industries. Although

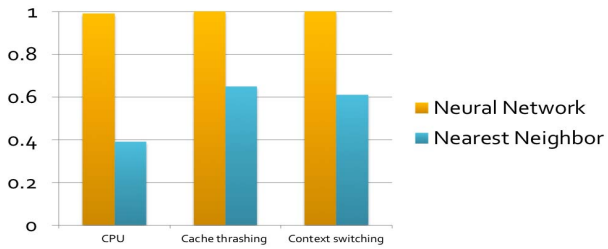



Fig. 8. F-score performance metrics of neural networks and nearest neighbor for anomaly detection techniques

Spark is developing gradually, currently there are still shortages in comprehensive performance analyses that specifically build for Spark and are used to detect performance anomalies. This paper focuses on addressing the challenge of detecting the anomalies by developing a neural network based methodology for anomaly detection that is able to improve the accuracy of anomaly detection based on knowledge of the RDD characteristics. Compares with CPU contention, the memory contention does not have visible effects on the mean CPU usage in the Spark cluster. The anomaly detection method for Apache Spark is significantly enhanced by using performance metrics for both the CPU and memory of Spark cluster. In addition, there were no significant effects of parallelism on detecting the anomalies in Apache Spark cluster using the neural networks. The conducted experiments and results demonstrate that the proposed method works effectively for complex scenarios and anomalies, such as CPU contention, cache thrashing and context switching anomalies. Moreover, the random instant and random duration anomalies did not impact the performance of our proposed solution.

Future work will focus on extended evaluation of an accurate anomaly detection technique that can continuously learn from the Spark environment about the new types of anomalies that may affect Spark performance. Therefore, Deep Learning techniques will be explored to obtain efficient solutions using supervised and unsupervised machine learning for in-memory frameworks. Deep Learning will be used to learn more complex features from the performance metrics of the Spark system, which can lead to more accurate detection and prediction of critical anomalies.

## Acknowledgment

This research is partly supported by King Abdulaziz City for Science and Technology (KACST) in Kingdom of Saudi Arabia.

## References

[1] D. J. Dean, H. Nguyen, X. Gu, Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems, in: Proceedings of the 9th international conference on Autonomic computing, ACM, 2012, pp. 191–200.

[2] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, D. Rajan, Prepare: Predictive performance anomaly prevention for virtualized cloud systems, in: 2012 IEEE 32nd International Conference on Distributed Computing Systems, 2012, pp. 285–294. doi:10.1109/ICDCS.2012.65.

[3] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth, Performance Anomaly Detection and Bottleneck Identification, ACM Computing Surveys 48 (1) (2015) 1–35. doi:10.1145/2791120.
URL http://dl.acm.org/citation.cfm?id=2808687.2791120

[4] S. Rogers, M. Girolami, A first course in machine learning, CRC Press, 2015.

[5] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, Supervised machine learning: A review of classification techniques (2007).

[6] Apache Spark™, Lightning-fast unified analytics engine.
URL https://spark.apache.org

[7] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.

[8] K. Wang, M. M. H. Khan, Performance prediction for apache spark platform, Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H (2015) 166–173doi:10.1109/HPCC-CSS-ICESS.2015.246.

[9] S. Fu, Performance metric selection for autonomic anomaly detection on cloud computing systems, in: Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE, IEEE, 2011, pp. 1–5.

[10] D. Oppenheimer, A. Ganapathi, D. A. Patterson, Why do Internet services fail, and what can be done about it?, in: USENIX symposium on internet technologies and systems, Vol. 67, Seattle, WA, 2003.

[11] S. Pertet, P. Narasimhan, Causes of failure in web applications (cmu-pdl-05-109), Parallel Data Laboratory (2005) 48.

[12] V. J. Hodge, J. Austin, A survey of outlier detection methodologies, Artificial intelligence review 22 (2) (2004) 85–126.

[13] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM computing surveys (CSUR) 41 (3) (2009) 15.

[14] X. Gu, H. Wang, Online anomaly prediction for robust cluster systems, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 1000–1011.

[15] T. Huang, Y. Zhu, Q. Zhang, Y. Zhu, D. Wang, M. Qiu, L. Liu, An lof-based adaptive anomaly detection scheme for cloud computing, in: Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual, IEEE, 2013, pp. 206–211.

[16] G. P. Zhang, Neural networks for classification: a survey, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 30 (4) (2000) 451–462.

[17] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning spark: lightning-fast big data analysis, " O'Reilly Media, Inc.", 2015.

[18] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, Mllib: Machine learning in apache spark, The Journal of Machine Learning Research 17 (1) (2016) 1235–1241.

[19] Apache Spark™, DAGScheduler.
URL https://github.com/apache/spark/

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, 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, USENIX Association, 2012, p. 2.

[21] P. Zheng, B. C. Lee, Hound: Causal learning for datacenter-scale straggler diagnosis, Proceedings of the ACM on Measurement and Analysis of Computing Systems 2 (1) (2018) 17.

[22] D. M. Kline, V. L. Berardi, Revisiting squared-error and cross-entropy functions for training neural network classifiers, Neural Computing & Applications 14 (4) (2005) 310–318.

[23] M. F. Møller, A scaled conjugate gradient algorithm for fast supervised learning, Neural networks 6 (4) (1993) 525–533.

[24] R. Caruana, S. Lawrence, C. L. Giles, Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping, in: Advances in neural information processing systems, 2001, pp. 402–408.

[25] R. Lior, et al., Data mining with decision trees: theory and applications, Vol. 81, World scientific, 2014.

[26] G. Casale, C. Ragusa, P. Parpas, A feasibility study of host-level contention detection by guest virtual machines, in: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, Vol. 2, IEEE, 2013, pp. 152–157.

[27] I. Ganelin, E. Orhian, K. Sasaki, B. York, Spark: Big Data Cluster Computing in Production, John Wiley & Sons, 2016.

# Chapter 1

# AI-driven performance management in data-intensive applications

**Ahmad Alnafessah,[1] Gabriele Russo Russo,[2] Valeria Cardellini,[2] Giuliano Casale,[1*] and Francesco Lo Presti[2]**

[1] *Department of Computing, Imperial College London, London, UK*
[2] *Department of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, Rome, Italy*

*Corresponding Author: Giuliano Casale; g.casale@imperial.ac.uk

Data-intensive applications have attracted considerable attention in recent years. Business organizations are increasingly becoming data-driven and therefore look for novel ways to collect, analyze, and leverage the data at their disposal. The goal of this chapter is to overview some recurring performance management activities for data-intensive applications, examining the role that AI and machine learning are playing in enhancing practices related, among others, to configuration optimization, performance anomaly detection, load forecasting, and auto-scaling for this class of software systems.

**Keywords:** Performance management, AI, Data-intensive applications, batch analytics, streaming

## 1.1. Introduction

In recent years, the prominence of Big data has led to a growth in interest for developing intelligent data-intensive software systems in several application domains. Data-driven systems that can extract knowledge, plan, and adapt to events through processing, transformation, and analysis of datasets are thus increasingly widespread in both industry and society.

From a technical standpoint, data-driven software systems are often built by leveraging features such as batch analytics or streaming, now easily programmable through in-memory platforms such as Apache Spark, Hadoop/MapReduce, Storm, Flink, among others. We outline popular data processing platforms in Section 1.2. Although the combination of batch and streaming workloads enables richer functionalities, workload heterogeneity also means that achieving service levels objectives presents additional complexity in pinpointing causes of performance degradation and identifying tunings to address them. For example, performance metrics in data-driven software are difficult to predict as they often depend on data properties, such as volume or velocity, and frequently even on data type and content, making it difficult to reason about and tune system performance at design time. Furthermore, the combination of batch and streaming features in a software means that different system components will strive to achieve different performance goals, i.e., high-throughput and high-utilization for analytics features and low-latency for stream processing operators, making the process of runtime performance tuning a fairly heterogeneous and complex exercise.

To support these challenges, the goal of this chapter is to overview AI management techniques that are available in the literature to manage and tune the performance of data-intensive applications. AI methods offer considerable

simplicity and flexibility in choosing the features that drive the management process, in spite of some opaqueness in presenting the way the models reach decisions.

Compared to traditional management methods, which either leverage low-level system characteristics or use mathematical modeling abstractions, AI management methods leverage learning on experimental datasets that reduce dependence on assumptions and shift the attention from conceptual modeling to data-collection and model training. This offers considerable potential to increase the effectiveness of management methods in situations where the system behaves according to complex and unpredictable logic, as it is often the case for systems driven by external data.

Summarizing, in this chapter we examine the applicability of AI methods in the context of data-intensive applications. We survey in particular studies that illustrate the versatility of AI models when applied to popular data streaming and batch analytics platforms. Our aim in particular is to cover a broad spectrum of AI methods, in order to inform the reader on the range of learning techniques that may be applicable to recurring management problems involved in data-driven systems. We look at common management tasks such as platform configuration, workload forecasting, resource scaling, monitoring and detection of performance anomalies. We also give selected examples to build an intuition on their behavior, benefits and limitations.

## 1.2. Data-processing frameworks

In this section, we overview the essential features of common execution platforms in use to define data-intensive applications. A summary of the key char-

**Table 1.1:** Summary of the key characteristics of data-processing platform.

| | Workloads | | Processing style | |
|---|---|---|---|---|
| **Platform** | **Batch** | **Streaming** | **In-Memory** | **Disk-heavy** |
| Storm | | ✓ | ✓ | |
| Hadoop/MR | ✓ | | | ✓ |
| Spark | ✓ | ✓ | ✓ | |
| Flink | ✓ | ✓ | ✓ | |

acteristics of each platform is shown in Table 1.1. The section also highlights core performance management challenges associated to each of these platforms.

### 1.2.1. Apache Storm

Apache Storm[1] is a popular open source platform used for distributed real-time stream processing. The platform offers very low latency for dataflow processing, making it an ideal option for real time processing Toshniwal et al. [2014]. Storm dataflow topologies involves two main node types: spouts and bolts. A spout is the source of the data stream at the input queue and may generate data by itself Shahrivari [2014]. A bolt instead consumes the stream, operates transformations or computations, and ultimately produces an output stream as a result. Every task corresponds to one operating system thread.

Performance management of Storm applications frequently involves difficult decisions concerning optimal configuration options for spouts and bolts, ranging from decisions concerning buffer, message, and batch sizes, number of bolts, and selection of optimal waiting strategies. There is a limited understanding of the interplay between these parameters, posing intrinsic challenges

[1] http://storm.apache.org/

for optimal system configuration. Moreover, the Storm system does not automatically manage load balancing and resource scaling, thus requiring ad-hoc performance management techniques.

### 1.2.2. Hadoop MapReduce

Hadoop implements the MapReduce paradigm and it is a well-known example of batch processing platform. It is used for intensive Big Data applications starting from a single server and can scale up to thousands of machines [2].Usually, MapReduce uses an existing dataset that is stored in Hadoop Distributed File System (HDFS) before beginning to process batch data. Processing with native Hadoop can be paused or interrupted, but the dataset cannot be modified. This means that if current data is changed for any reason, the job needs to be run again.

Despite distinctive challenges arise in the area of optimal configuration of Hadoop platforms, over the years performance management has insisted in particular on the problem of detecting and handling straggler tasks, which falls into the general problem area of performance anomaly detection and mitigation. This is a result of the synchronizations between dataflow tasks that can block progress until straggler tasks complete their activities.

[2] https://hadoop.apache.org/

### 1.2.3. Apache Spark

Apache Spark[3] is a large-scale in-memory processing framework that can support both batch and stream data processing, which can make it easy with a low cost to support different types of workloads on the same engine in a production environment, such as those arising from graph analysis and machine learning applications. Compared to older solutions such as Hadoop, the main goal of Apache Spark is to speed up batch processing by utilizing in-memory computation. Thanks to a reduced use of intermediate storage of processing results, Spark is orders of magnitude faster than Hadoop for in-memory analytics.

Spark can be deployed over Hadoop as an alternative to MapReduce, as well as on Amazon EC2, Apache Mesos, or as a standalone cluster. In addition, it can access many data sources, including HDFS, Cassandra, HBase, Hive, Tachyon, and any Hadoop data source. Spark provides a general purpose engine for different kinds of computation, including iterative algorithms, job batches, streaming, and interactive queries. These different types of computation were previously difficult to find in the same distributed system Karau et al. [2015]. Beyond the ability to perform batch and stream processing, Spark also provides a rich library that is built on top of its core engine Meng et al. [2016].

In terms of performance management, Apache Spark has around 200 complex parameter configurations (e.g., executors, CPU cores, memory, shuffle behavior, compression), which may significantly impact the overall Spark system performance Herodotou et al. [2020]. The microarchitectural behaviors of Spark are different from those of other Big Data technologies. The Spark core data abstraction is the Resilient Distributed Dataset (RDD), which cannot be

[3] https://spark.apache.org/

modified and RDD can be executed in parallel on different nodes. In addition, Spark needs more advanced auto tuning solutions to boost its performance within production environment. Therefore, precisely performance management to optimally manage and auto-tune Spark is needed to increase the performance efficiency of such a complex system and immediately gain advantages of cost and time saving.

### 1.2.4. Apache Flink

Apache Flink Carbone et al. [2015] is another open source distributed processing engine designed for low-latency streaming computation. Analogously to Spark, Flink relies on in-memory computation and provides a unified API for processing both bounded and unbounded datasets. However, differently from Spark, where batching has a primary role, Flink has been designed with streaming in mind. Indeed, Flink applications are built upon the concepts of *streams* and *transformations*. Streams represent (possibly unbounded) data flows, while transformations are operations that, given one or more streams as input, output one or more streams as the result (e.g., filtering). A few higher-level libraries are built on top of these abstraction, easing the definition of common processing use cases (e.g., complex event processing, graph analytics).

At runtime, Flink applications are mapped to *streaming dataflows*, DAGs composed of processing nodes (often called *operators*), which implement transformations, connected by streams. For execution, Flink leverages a distributed architecture, designed according to the *master-worker* pattern. The master component is the *JobManager*, which coordinates distributed execution and

is responsible for application scheduling, checkpointing, and recovery in case of failure. The TaskManagers (i.e., the workers) execute the application *tasks* (i.e., instances of operators) and manage the data transfers between them.

Performance management of Flink applications, which are often long-running, mainly involves runtime deployment and resource adaptation. First of all, varying infrastructure conditions may require migrating operator tasks between computing nodes during execution. Flink supports migrating both stateless and stateful tasks through the *savepoint* mechanism, which ensures no loss of information. Moreover, workload variability requires dynamically scaling the parallelism of Flink applications and balancing the load across the cluster to keep consistent performance levels over time. To this end, load prediction techniques can be helpful to proactively adapt application configuration.

## 1.3. State of the art

We review in this section the existing techniques for dealing with the most relevant performance management issues in the context of data-intensive applications, with particular emphasis on AI-based approaches.

### 1.3.1. Optimal configuration

As a consequence of the availability of numerous configuration parameters, their optimization is a critical task in the domain of data-intensive systems. The goal of configuration optimization is to find the ideal configuration with respect to the system performance. Various automated parameter tuning methods have been proposed in the literature, which are discussed in the following sections .

#### 1.3.1.1. Traditional approaches

Gunawan and Lau [2011] propose a parameter tuning framework based on Design of Experiments (DOE) approach. Their goal is to find an initial range of parameter values for automated tuning using a factorial experiment design to screen and rank all the parameters, so as to focus the search on the most influencing parameters. In addition, Gunawan and Lau [2011] examine Response Surface methodology, which is a model-based approach within DOE that can be used to quantify the effect of each parameter to find the most promising initial range for the vital parameter values. Their approach can be integrated with existing automated parameter tuning configuration, called ParamILS and Randomized Convex Search (RCS). Their method seams promising for both discrete and continuous parameter configuration settings.

Shi et al. [2014] introduce MRTuner from IBM, which is a tool to enable holistic optimization for MapReduce jobs. Their design uses an efficient search algorithm (Grid-based Search) to find the optimal execution plan. Around twenty configuration parameters are investigated to understand the relationships that have a noticeable impact on MapReduce performance. The tool is evaluated using HiBench on two Hadoop clusters. Their results show MRTuner has low latency and can find accurate execution plans.

Bilal and Canini [2017] examine an automatic parameter tuning framework for stream processing platforms. Gray-Box, Black-box analysis, and a rule-based optimization method are combined, and configuration parameters are initialized using Latin Hypercube Sampling. Hill Climbing Algorithm is used to explore the configuration space. Bilal and Canini [2017] evaluate using three benchmark applications within the Apache Storm streaming system. They find that rule-based can converge up to five times faster than other approaches,

making it suitable for parameter tuning within stream processing platforms.

#### 1.3.1.2. AI approaches

A machine learning approach is used by Chen et al. [2015] to appropriately tune configuration parameters of Hadoop. Their approach has two stages, which are the prediction stage to estimate the performance of a MapReduce job and the optimization stage to repeatedly search for the optimal configuration parameters. The authors claim that their method can improve Hadoop performance up to eight times compared with traditional methods.

Wang et al. [2016] introduce a parameter tuning method based on binary classification and multi-classification for Apache Spark systems. Decision trees are used for auto-tuning of configurations with four different types of workloads, which are Sort, Wordcount, Grep and NavieBayes workloads from *BigDataBench* benchmark. Their experimental results show that the proposed method can improve Spark performance on average by 36% compared to default Spark configuration.

Hernández et al. [2018] optimize parallelism for data-intensive platforms using machine learning. They use Boosted regression trees are used as the authors claim that they have the lowest variance compared with other algorithms. In addition, they argue that decision trees are interpretable, which means that it is possible to quantify the impact of collected features on the overall performance. They evaluate proposed solution using a benchmark of 15 different Spark applications running on YARN. The results show that their task parallelization method is capable of improving the performance of Spark by 51%.

Bayesian Optimization (BO) is an effective and efficient method for auto-

tuning systems and machine learning algorithms. Joy et al. [2016] propose a framework that uses BO to tune hyperparameters of data-intensive applications. Their idea is dividing the data into small chunks with the same size to boost the search by applying BO tuning in parallel. To validate the performance of their framework, they use the proposed method to tune two machine learning algorithms, Deep Neural Networks (DNN) and Support Vector Machines (SVM). BO offers effective hyperparameters tuning with less computational overhead.

Jamshidi and Casale [2016] tackles the challenging issue of finding optimal configurations for a data-intensive streaming system by proposing auto-tuning methods that can help systems administrators to determine the near-optimal configurations with a limited budget of experiments. Their solution revolves around BO for configuration optimization, which utilizes Gaussian Processes (GP) to continuously capture posterior distributions of configuration space for the application. Their method works in a way that the optimal configurations will eventually be discovered. The authors validate the proposed method using a Storm clsuter in the cloud.

Yigitbasi et al. [2013] examine and explore a machine learning model to tune the configuration parameters of Hadoop and MapReduce. They use Support Vector Regression (SVR) to a smart search algorithm in terms of the effectiveness of parameter space exploration. Their results show that SVR obtains higher accuracy than Starfish auto-tuner, which uses a cost-based search model.

Liao et al. [2013] illustrate that the Hadoop platform has hundreds of configuration parameters that have very complicated interactions. This wide configuration space makes it time-consuming for system administrators to optimally

tune the Hadoop parameters. They provide an evaluation to automate the tuning process of Hadoop based on a cost-based and machine learning approach (neural network, SVR, multiple linear regression, and decision trees).

Di Sanzo et al. [2012] provide a study about auto-tuning cloud-based in-memory transactional data grids configuration by using a machine learning black-box approach. They use artificial neural networks (ANN) to optimize the dynamic selection of the amount of cache servers and the replication level of data objects to reduce the cost of cloud system operations. They conduct preliminary experiments based on a synthetic benchmark and a real data grid system that is run on Amazon EC2 virtual servers. The authors conclude that the ANN-based approach is effective for tuning transactional data grids. There are some additional works related to optimal configuration using CART tree Nguyen et al. [2018], long short-term memory (LSTM) Fang et al. [2019], regression trees, nearest neighbor Berral et al. [2015], and reinforcement learning Peng et al. [2017].

#### 1.3.1.3. Example: AI-based optimal configuration

In this section, we illustrate the effectiveness of BO for finding optimal configurations by searching through the configuration space. The goal of BO is to utilize the prior knowledge and evidence to optimize the posterior at each evaluation step, to reduce the gap between the actual global optimization and expected optimization for the model Brochu et al. [2010]. Compared with traditional search algorithms (grid search, random search and manual tuning), Alnafessah and Casale [2020] show that how BO can facilitate parameter tuning with more parameters and fewer number of experiments to find optimal configurations.

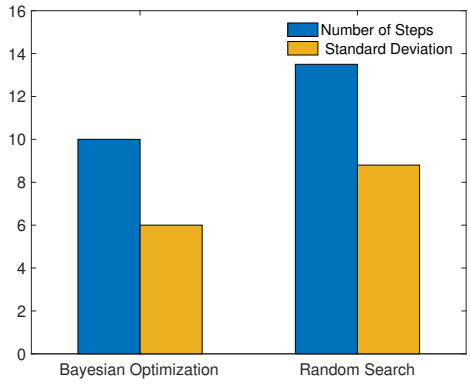



**Figure 1.1:** Comparison between Bayesian Optimization and Random Search to reach the highest F-score

BO is an ideal choice to find the optimal training dataset size and configuration parameters to efficiently and effectively train the anomaly detection model to achieve high F-score in a short period of time. Before applying BO, there are two main choices that need to be carefully make, which is the prior over functions and type of acquisition function Snoek et al. [2012]. We use Gaussian Processes (GP), which are stochastic processes defined by the property that any finite set of $N$ points induces a multivariate Gaussian distribution Snoek et al. [2012], Shahriari et al. [2015]. They are are efficient for uncertainty estimation. We use the *Expected Improvement* acquisition function that Snoek et al. [2012] provide for configuration space with high uncertainty and high estimated value to evaluate a point $x$ to sample based on the posterior distribution function to guide exploration. It can trade-off between exploration of the configuration search space and exploitation of current promising subspace.

Figure 1.1 shows a comparison between BO and random search in achieving the high performance training of the machine learning algorithm (ANNs

in our case) to efficiently detect anomalies (CPU, Cache Thrashing, and Context Switching) within Apache Spark Streaming datasets that Alnafessah and Casale [2020] provides with predefined F-score. For CPU anomaly detection, the BO can optimally train the anomaly detection model using 10 combinations of configurations, whereas the random search needs 14 combinations of configurations on average. In addition, BO outperforms the random search in training the AI model for detecting CPU, cache and context switching anomalies.

### 1.3.2. Performance anomaly detection

System performance is often described in terms of the time taken to process a set of tasks with a given amount of computing resources that are consumed within a given observation period Ibidunmoye et al. [2015]. The growing complexity and dynamicity of cloud systems and data-intensive technologies requires significantly higher levels of automation and significant attention Fu [2011]. Performance anomalies have become a major concern for developers and academic researchers particularly for Big Data and Artificial Intelligence technologies over cloud computing systems. Anomalous performance can occur as a result of service operator faults Oppenheimer et al. [2003], system failures, user errors Pertet and Narasimhan [2005], environmental issues, and security violations Ibidunmoye et al. [2015], among others.

Many anomaly detection studies are generic, while others are specifically conducted for certain application domains (e.g., data-intensive applications, networking, web based application, etc). There are studies that provide an overview about techniques that have been developed in traditional statistical

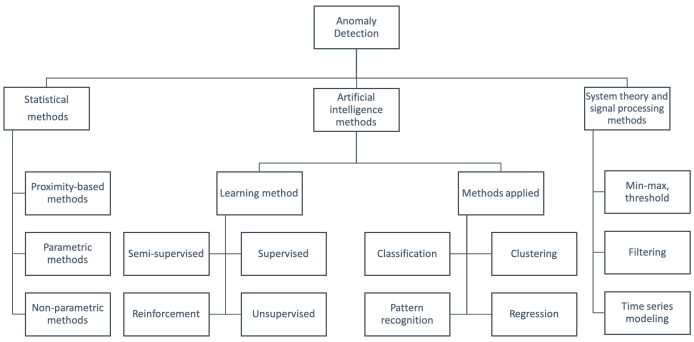



**Figure 1.2:** A taxonomy of anomaly detection techniques generalizing the ones in Sebestyen et al. [2018], Hodge and Austin [2004], Chandola et al. [2009].

approaches and machine learning techniques for anomaly identification Hodge and Austin [2004], Chandola et al. [2009], Ibidunmoye et al. [2015]. Figure 1.2 shows a taxonomy of existing anomaly detection techniques that is build on common taxonomy based on Sebestyen et al. [2018], Hodge and Austin [2004], Chandola et al. [2009].

#### 1.3.2.1. Traditional approaches

Statistical techniques are the earliest approaches used for performance anomaly detection. Lu et al. [2017] use a statistical offline approach to detect abnormal Apache Spark tasks and analyze the root causes based on statistical spatial-temporal analysis. They use some features related to execution time, memory usage, garbage collection, and data locality of each Spark task to determine the degree of abnormal tasks. They use mean and standard deviation of all tasks in each stage to decide the threshold and to get information about macro-awareness on the task execution time. They analyze performance issues using

factor combination criteria for every performance anomaly based on weighted factors. They validate their method on a private Spark cluster and SparkBench Li et al..

Kelly [2005] examines how to get insights about performance issues of globally distributed systems using simple queueing theoretic observations together with standard optimization methods. The author obtain extensive empirical results from three distributed commercial production systems that serve real customers.

Yang et al. [2007] propose an anomaly detection and diagnosis solution within grid environments using statistical and signal processing approaches. Their work extends the traditional window-based strategy by using signal processing to filter out recurring background variations and determine which resource is the probable cause of an anomalous performance in a system. They use window averaging, which is a widely used statistical anomaly identification technique because it is simple and efficient. The anomalies are injected into three grid systems (Cactus, GridFTP, and Sweep3d) at random time intervals.

#### 1.3.2.2. AI approaches

Research on automated anomaly detection is essential in practice because any late detection or slow manual resolution of performance anomalies in a real production environment may cause prolonged service-level agreement violations and significant financial penalties Dean et al. [2012], Tan et al. [2012]. This leads to a demand for performance anomaly detection solutions in cloud computing and data intensive systems that are both dynamic and proactive in nature Ibidunmoye et al. [2015]. The need to develop these methods for production environment with very different characteristics means that AI is

ideally positioned for system diagnosis to automatically identify performance anomalies. These techniques provide the capability to quickly learn baseline performance characteristics through a large monitoring metrics space in order to distinguish normal and anomalous patterns Rogers and Girolami [2016].

Classification techniques aim to determine whether the instances in a given feature space belong to a particular class or multiple classes Ibidunmoye et al. [2015]. There are popular classification techniques for anomaly identification, such as ANNs, SVM, and nearest neighbor. The classification technique is significantly affected by the accuracy of the labeled data and algorithms that have been used. For example, the training and testing processes for decision trees algorithms are usually faster than SVM, which involve quadratic optimization.

Alnafessah and Casale [2018] propose an ANN-driven methodology for anomaly identification, particularly for Apache Spark. The authors use a machine learning approach to quickly sift through Spark logs and system monitoring metrics to precisely detect and classify anomalous behaviors. The authors evaluate the proposed method against three popular machine learning algorithms, decision trees, nearest neighbor, and SVM, as well as against four different monitoring datasets. Their results show that the recommended method has ability to classify overlapped anomalies and outperforms other methods by obtaining 98%-99% F-scores, and offering much higher performance than alternative techniques to detect both the period in which anomalies occurred and their type.

Lu et al. [2018] utilize convolutional neural networks (CNN) for performance anomaly diagnosis for Big Data system logs, and specifically for the Hadoop Distributed File System (HDFS) logs. They implement the proposed model with different filters to automatically train model on the relationships

among events. The CNN is configured to have *logkey2vec* embeddings, three 1D convolutional layers, dropout layer, fully connected softmax layer, and max-pooling. The authors provide a comparison between CNNs and other well-known networks such as Long Short Term Memory networks (LSTM) and Multilayer Perceptron (MLP). The experimental results show that the CNN model is more accurate and faster in detecting anomalies than LSTM and MLP for HDFS logs.

Fulp et al. [2008] predicting system failures using Support Vector Machines algorithm (SVM) for binary classification based on system log files. The proposed approach utilizes advantages of the sequential nature of logs and uses use a sliding window of messages to predict the likelihood of system failure within that has 1024 computing nodes. The SVM associates the messages to a class of normal or abnormal event. Their results show that the proposed solution can predict hard disk failure with 73% accuracy. Fu et al. [2012] propose a hybrid anomaly identification Framework using one-class and two-class SVM algorithms. They claim that their approach does require prior knowledge about system failure history and offers self adapt learning from observing system failure within cloud environment.

There are several other studies in the literature that deal with stragglers. Yadwadkar and Choi [2012] introduce a proactive straggler avoidance regression decision tree model that periodically learns correlations between node level status and task execution time for MapReduce logs. The authors justify the choice of regression trees by showing the fast prediction of stragglers. They apply their method on a trace from Facebook Hadoop system and Berkeley EECS department's local Hadoop cluster (icluster). Qi et al. [2017] use a white-box model that utilizes classification and regression trees for root causes

analyses for Spark logs and hardware sampling tools to train their model. A special type of tree called a CART tree (*classification and regression tree*) is used to mitigate overfitting issues. They use a customized prune method for several iterations to improve analysis accuracy and the classification performance metrics are checked for each node and its leaves. The authors applied their method on Spark with HiBench benchmark.

Based on the local neighborhoods of event, the neighbor-based technique uses unsupervised learning to analyze data instances. This technique can distinguish an anomalous instance among normal instances because normal instances usually occur in dense neighborhoods, whereas anomalous instances occur far from their closest neighbors (Chandola et al. [2009]). Huang et al. [2013] propose a special type of neighbor-based technique, called local outlier factor (LOF), for an anomaly identification that can learn system behaviors during training and detecting time within cloud computing environment. They argue that their method is adaptive to changes, detects contextual anomalies, and requires less effort for collecting performance metrics for training process.

#### 1.3.2.3. Example: ANNs-based anomaly detection

Classification techniques aim to determine whether the instances in a given feature space belong to a particular class or multiple classes Ibidunmoye et al. [2015]. ANNs algorithms are the most popular classification technique for anomaly identification. This is because ANNs represent a data-driven, non-linear and self-adaptive method that can adjust itself to the given datasets without requiring prior knowledge about the distribution or function of the used model, and generalize the models even to input data that has never been seen before Zhang [2000]. These advantages have caused ANNs to be considered

a universal functional approximation.

One of the well-known critical issues for ANNs and other classifiers is feature selection. The objective of feature selection is to discover the smallest set of appropriate input features and at the same time achieve the desirable predictive performance. It is crucial to reduce the number of input features for the classifier to achieve satisfactory accuracy with reduced computation in the model Zhang [2000].

We use Backpropagation and conjugate gradients to train ANNs, that is to update values of weights and biases in the network. We use scaled conjugate because it is often fast Møller [1993], especially for time-dependent applications. *Sigmoid* transfer function is often used as an activation function in the hidden layer because it exists between (0 to 1), where zero means absence of the feature and one means its presence. In addtion, we use *Softmax* transfer function in the output layer to handle classification problems with multiple classes (e.g., normal, CPU anomaly, cache thrashing anomaly, context switching anomaly). For a cost function, we use *cross-entropy* to evaluate the performance and compare the actual output error results with the desired output values (labeled data). We use *Cross-entropy* because it has significant practical advantages over squared-error cost functions Kline and Berardi [2005].

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, by trying all the possible numbers between the sizes of input neurons and output neurons Sheela and Deepa [2013]. The output layer contains a number of neurons equal to the number of target classes (normal + types of anomalies).

Figure1.3 shows a sensitivity analysis for the size of collected datasets to

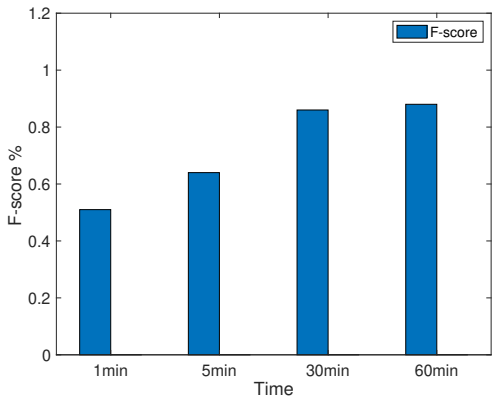



**Figure 1.3:** The performance of ANNs models that is trained with training dataset that have been collected for 1, 5, 30, and 60 min.

train the neural network algorithms to learn complex nonlinear relationships among performance metrics and detect the anomalous performance within Spark systems. It is clear that the size of collected training data significantly impact the ANN model, while it is challenging to find the optimal size of the training data. The small size of training dataset causes unacceptable F-score, whereas a large dataset may lead to a waste of computing resources.

### 1.3.3. Load prediction

Data streaming applications usually deal with unbounded data flows, meaning that they are kept in execution indefinitely. As a consequence, these applications likely face different working conditions over time (e.g., varying workloads), hence requiring dynamic resource management solutions to keep acceptable performance. Indeed, researchers have spent a lot of effort aiming to enhance streaming systems with online adaptation capabilities and, in particu-

lar, *elasticity*, that is the ability to dynamically acquire and release computing resources Hummer et al. [2013] as needed. To this end, there are two main research directions so far, revolving around (i) application load prediction and (ii) auto-scaling policies. In this section, we focus initially on load prediction.

#### 1.3.3.1. Traditional approaches

Given the variability that often characterizes streaming workloads, predicting the application processing load in the future (e.g., application input data rate) is a difficult yet important task for driving resource management with foresight. To this end, traditional time series forecasting methods can be useful in the context of streaming applications. For instance, Imai et al. [2018] use the well-known ARIMA model for workload forecasting. They additionally consider an online regression approach for predicting the maximum sustainable throughput of streaming applications, and scale the number of virtual machines (VMs) allocated to the system accordingly. Kombi et al. [2019] instead exploit regression techniques to predict the input rate of Storm operators, and drive the application auto-scaling. They consider three prediction models, respectively based on linear, logarithmic, and exponential regression, and select the best model to use at runtime based on fitting accuracy observed in the previous iteration.

#### 1.3.3.2. AI approaches

AI techniques often allow to outperform traditional forecasting approaches for workload and resource utilization prediction. For instance, Zacheilas et al. [2015] use GPs for predicting the future input rate and processing latency of operators, and hence drive horizontal elasticity of *complex event processing*

applications running on top of Storm. Their elasticity algorithm exploits the uncertainty estimation provided by GPs to avoid making auto-scaling decisions whenever the uncertainty level is considered too high. Hu et al. [2019] instead use SVR for predicting resource usage of Spark Streaming applications, and allocate virtual machines for the cluster so as to meet SLA requirements. Runsewe and Samaan [2017] also target Spark Streaming and leverage Layered Hidden Markov Models to predict the resource usage of multiple applications running on a Spark cluster. Based on the obtained predictions, they scale the Spark cluster as needed.

A few works have investigated the use of NNs to predict the future load of data streaming applications. Lombardi et al. [2018] propose ELYSIUM, a multi-level elasticity solution for Storm, which controls both the operator parallelism and the number of worker nodes in the Storm cluster. ELYSIUM relies on NNs for predicting both (i) the application input rate in the near future, and (ii) the CPU utilization of each application operator based on the input rate.

Mu et al. use DNNs for multi-step operator performance prediction. They define two prediction strategies based on DNNs, to be used, respectively, on offline and online collected metrics. They use ensemble learning techniques to merge the offline and online predictions, and obtain the final prediction, which can be used to drive auto-scaling. Xu et al. [2019] rely on DNNs as well, to predict operator performance online. Specifically, they exploit Recurrent DNNs to make accurate performance predictions, which also account for interference due to co-located operators (i.e., operators deployed in the same worker node). They integrate this solution in Storm, and their experiments show that it outperforms ARIMA- and SVR-based approaches for prediction. Khoshkbarforoushha et al. [2017] instead exploit Mixture Density Networks to estimate

resource usage of streaming applications as probability density functions. They show how the resulting distribution based workload prediction can be applied to drive both auto-scaling and application admission control in presence of SLAs.

### 1.3.4. Scaling techniques

Data-intensive systems largely exploit parallelism to efficiently process high-volume datasets. For streaming applications, whose datasets are collected in real-time and hence are not known at deployment time, scaling the amount of computing resources at runtime is fundamental to avoid the risk of under- or over-provisioning resources.

#### 1.3.4.1. Traditional approaches

As extensively surveyed in Röger and Mayer [2019], researchers so far have investigated a large number of approaches to devise auto-scaling policies for streaming systems, including, e.g., queueing theory, control theory, state-space based methods, and, recently, AI. Existing solutions can be classified as either reactive and proactive. *Reactive* solutions make auto-scaling decisions in response to observed changes (e.g., increase in the application input data rate), whilst *proactive* approaches try to adapt the application deployment before observing changes, based on predictions.

Among the reactive approaches, threshold-based policies are widely adopted. According to these policies, auto-scaling actions are triggered whenever one or more observed metrics (e.g., resource utilization, throughput) violate predefined threshold values Castro Fernandez et al. [2013], Gedik et al. [2014].

Other works instead rely on models to periodically evaluate the expected application performance or resource utilization, and trigger scaling actions accordingly. For instance, Lohrmann et al. [2015] rely on queueing theory to model application performance, and make horizontal scaling decisions so as to meet response time requirements.

Among the proactive auto-scaling solutions, several works present policies that exploit load prediction to make scaling decisions. Indeed, most the prediction solutions mentioned in the previous section, are complemented with auto-scaling mechanisms. A different approach is considered in De Matteis and Mencagli [2016], where *model predictive control* is used to proactively scale streaming operators, also combining horizontal and vertical elasticity.

#### 1.3.4.2. AI approaches

The behavior of traditional auto-scaling solutions often depends on manually configured parameters, and AI-based approaches aim at overcoming this limitation. For instance, Reinforcement Learning (RL) is a class of methods allowing agents (e.g., resource managers) to learn policies by direct interaction with their environment (e.g., managed applications). RL has been adopted by several works to derive auto-scaling policies for streaming applications at runtime. For instance, Heinze et al. [2014] use the SARSA algorithm, relying on a reward function that captures the difference between current operator CPU utilization and target utilization values. Similarly, Cheng et al. [2018] rely on the well-known Q-learning algorithm to adapt the amount of resources allocated to jobs running in Spark Streaming. They consider a performance-oriented reward function that accounts for throughput and latency. Lombardi et al. [2018] also use Q-learning to automatically tune the parameters for a

threshold-based auto-scaling algorithm.

Russo Russo et al. [2019] consider the auto-scaling problem in presence of heterogeneous computing resources to host parallel operator instances. They use linear function approximation to deal with the large model state space, and investigate model-based initialization to speedup the learning process. Their reward function accounts for the amount of allocated resources, the adaptation cost, and a SLO violation penalty.

### 1.3.5. Example: RL-based auto-scaling policies

RL agents learn by experience the *actions* to perform in order to maximize a cumulative *reward* over time (Sutton and Barto [2018]). We define the task faced by RL agents as an infinite-horizon, discrete-time Markov Decision Process (MDP), where agents perform an action at every time step, selected according to their *policy* and the observed current *state*. Following action execution, agents get a reward and possibly enter a new state. Their goal is maximizing the (discounted) cumulative reward over the infinite time horizon.

The auto-scaling problem for a streaming operator could be modeled as follows. Considering a slotted time model, we define the state at time step i, $s_i$, as the pair $(k_i, \lambda_i)$, where $1 \leq k_i \leq K_{max}$ denotes the operator parallelism, and $\lambda_i$ the monitored input rate (discretized using a suitable quantum). Actions in this model represent scaling operations that alter the parallelism level, hence they are selected from the set $\mathcal{A} = \{-1, 0, +1\}$, except for the states where no further scale-out (or, scale-in) is allowed.

In the context of resource management, it is often convenient to reason in terms of *cost* instead of reward. Therefore, we define the cost $c(s, a, s')$

paid for operating the system in state $s'$ after taking action $a$ in $s$, and let the reward $r(s, a, s') = -c(s, a, s')$. Depending on the specific scenario, the cost function may capture different aspects. We define it as a weighted sum of three cost components: resources cost (proportional to the operator parallelism level), adaptation cost (capturing the overhead due to scaling), and performance violation cost (paid whenever the chosen configuration does not satisfy performance requirements).

Most RL algorithms rely on the so-called *Q-function* $Q(s, a)$, which estimates the cumulative discounted reward obtained in the long-term when choosing action $a$ in $s$. The most popular RL algorithm is Q-learning, which performs a single $Q$ update at every time step:

$$Q^{new}(s_i, a_i) \longleftarrow (1 - \alpha)Q^{old}(s_i, a_i) + \alpha \left( r_i + \gamma \max_{a'} Q^{old}(s_{i+1}, a') \right) \tag{1.1}$$

where $r_i$ is the reward obtained at time $i$, and $\alpha \in (0, 1)$ is the learning rate. Q-learning is easy to implement, and guarantees convergence to the optimal policy as infinite exploration is provided. However, to achieve faster convergence, in practice it is worth including any available knowledge about the model in the learning algorithm (*model-based* RL). For instance, the concept of *post-decision state* (PDS) (Mastronarde and van der Schaar [2011]) can be used to separate the known system dynamics (e.g., impact of a scaling action on the parallelism) from the unknown ones (e.g., input rate variations), and let the agent only learn the latter. To demonstrate the benefits provided by PDS, we simulated the execution of a data streaming operator under varying input data rate, using the RL-based auto-scaling policy. Figure 1.4 shows a sample of the workload we used, and the average reward accumulated over time by the RL agent, in the case of plain Q-learning and Q-learning with PDS. We can note

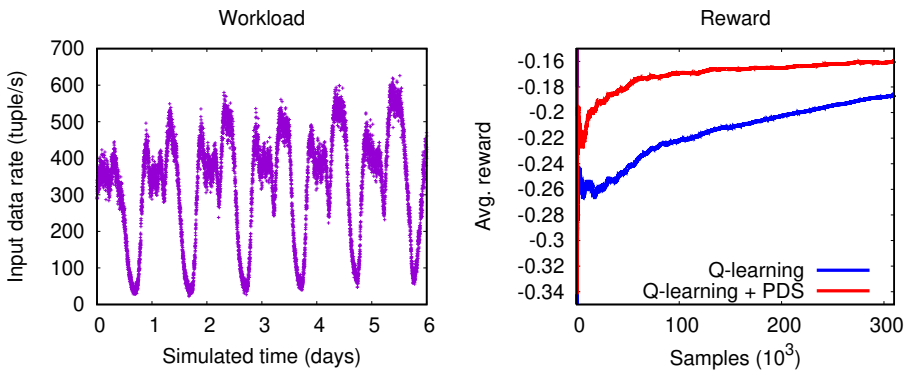



**Figure 1.4:** Streaming workload in a simulated experiment (left), and average reward obtained by RL-based auto-scaling agents (right).

that the PDS-based agent clearly outperforms plain Q-learning, as it exploits the available knowledge about the system and needs to learn fewer parameters.

This model can be extended to account for other adaptation mechanisms (e.g., operator migration), infrastructure characteristics, or to include more general workload characterizations. As more complex models are used, however, the state space often grows significantly, requiring, e.g., function approximation techniques for manipulating the Q-function. To this end, Deep RL is receiving growing interest, where DNNs are used to approximate $Q$.

## 1.4. Summary and conclusion

Table 1.2 summarizes the mapping of AI models to management and resource allocation activities we have referenced through this chapter. Our analysis reveals that AI-driven optimal configuration and anomaly detection in data-intensive applications have received considerable attention already, covered a

**Table 1.2:** Summary of the AI techniques considered for performance management of data-intensive systems: ✓: in this chapter, ∗: in extended bibliography.

| **AI method** | **Opt. Config.** | **Anom. Det.** | **Load Pred.** | **Scaling** |
|---|---|---|---|---|
| Neural networks | ✓ | ✓ | ✓ | |
| Boosted regr. trees | ✓ | | | |
| CART | ✓ | ✓ | | |
| Decision trees | ✓ | ✓ | | |
| Gaussian processes | ✓ | | ✓ | |
| LSTM | ✓ | ✓ | | |
| Nearest neighbor | ✓ | ✓ | | |
| RL | ✓ | | | ✓ |
| SVMs | ✓ | ✓ | ✓ | |

broad range of methods. In spite of this, certain techniques, such as reinforcement learning, still present open room for further investigation.

As mentioned, performance management for data streaming systems benefits from runtime load prediction, which is often coupled with auto-scaling or admission control mechanisms. Neural networks, and in particular DNNs, have received the largest share of attention so far, as they have been shown to outperform other techniques, like SVMs and GPs. The adoption of other approaches, already used for forecasting in other domains, including, e.g., long-short term memory (LSTM) models and Regression Trees, could be the subject of future investigation.

As regards the definition of auto-scaling control policies for data-intensive applications, only RL techniques have been exploited so far. Nevertheless, as the class of RL methods is quite large and complex, this research direction is far from being completely explored. Among the main issues to be tackled when adopting RL algorithms, state space explosion is critical in the context of performance management, as it limits the granularity and completeness of

the application and performance models used by the agents. To overcome this issue, *Deep RL* (DRL) algorithms exploit DNNs for approximate learning in otherwise intractable tasks. For instance, Li et al. [2018] have recently applied DRL for controlling the placement of streaming operators over a cluster of machines. Further investigations are needed in the literature to understand the potential of Deep RL in the management of data-intensive systems.

In conclusion, the chapter has shown that AI methods are already subject to intense research work across various management areas, with the areas of optimal configuration and anomaly detection being the most mature. The richness and the rapid evolution of the AI research landscape offer considerable opportunities to further raise the maturity of AI methods. Our analysis reveals that, although significant work already exists in this area, load prediction and scaling techniques would particularly benefit from a broader research investigation.

# Bibliography


A. S. Alnafessah and G. Casale. A neural-network driven methodology for anomaly detection in Apache Spark. In *2018 11th QUATIC*. IEEE, 2018.

A. S. Alnafessah and G. Casale. TRACK: Optimizing artificial neural networks for anomaly detection in Spark Streaming systems. In *13th VALUETOOLS*, 2020.

J. L. Berral, N. Poggi, D. Carrera, A. Call, R. Reinauer, and D. Green. ALOJA-ML: A framework for automating characterization and knowledge discovery in Hadoop deployments. In *21th ACM SIGKDD*, 2015.

M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *2017 SoCC*. ACM, 2017.

E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *2013 ACM SIGMOD*. ACM, 2013.

V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15, 2009.

C.-O. Chen, Y.-Q. Zhuo, C.-C. Yeh, C.-M. Lin, and S.-W. Liao. Machine learning-based configuration parameter tuning on Hadoop system. In *2015 IEEE Int'l Congress on Big Data*. IEEE, 2015.

D. Cheng, X. Zhou, Y. Wang, and C. Jiang. Adaptive scheduling parallel jobs with dynamic batching in Spark Streaming. *IEEE TPDS*, 29(12):2672–2685, 2018.

T. De Matteis and G. Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. *SIGPLAN Not.*, 51(8), 2016.

D. J. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proc. of ICAC '12*. ACM, 2012.

P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia. Auto-tuning of cloud-based in-memory transactional data grids via machine learning. In *2012 2nd NCCA*. IEEE, 2012.

H. Fang, W. Lu, Q. Li, J. Kong, L. Liang, B. Kong, and Z. Zhu. Predictive analytics based knowledge-defined orchestration in a hybrid optical/electrical datacenter network testbed. *J. of Lightwave Technology*, 37(19):4921–4934, 2019.

S. Fu. Performance metric selection for autonomic anomaly detection on cloud computing systems. In *2011 IEEE GLOBECOM*. IEEE, 2011.

S. Fu, J. Liu, and H. Pannu. A hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines. In *ADMA 2012: Advanced Data Mining and Applications*. Springer, 2012.

E. W. Fulp, G. A. Fink, and J. N. Haack. Predicting computer system failures using support vector machines. In *WASL '08*. USENIX Association, 2008.

B. Gedik, S. Schneider, M. Hirzel, and K. Wu. Elastic scaling for data stream processing. *IEEE TPDS*, 25(6):1447–1463, 2014.

A. Gunawan and Lindawati Lau, H. C. Fine-tuning algorithm parameters using the design of experiments approach. In *LION 2011: Learning and Intelligent Optimization*. Springer, 2011.

T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *2014 ICDEW*, 2014.

Á. B. Hernández, M. S. Perez, S. Gupta, and V. Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*, 86: 1076–1092, 2018.

H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv.*, 53(2):1–37, 2020.

V. J. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

Z. Hu, H. Kang, and M. Zheng. Stream data load prediction for resource scaling using online support vector regression. *Algorithms*, 12(2):37, 2019.

T. Huang, Y. Zhu, Q. Zhang, Y. Zhu, D. Wang, M. Qiu, and L. Liu. An LOF-based adaptive anomaly detection scheme for cloud computing. In *2013 IEEE COMPSACW*, pages 206–211. IEEE, 2013.

W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. *WIREs Data Mining and Knowledge Discovery*, 3(5):333–345, 2013.

O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1):1–35, 2015.

S. Imai, S. Patterson, and C. A. Varela. Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In *2018 18th IEEE/ACM CCGRID*, pages 62–71, 2018.

P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *2016 24th IEEE MASCOTS*. IEEE, 2016.

T. T. Joy, S. Rana, S. Gupta, and S. Venkatesh. Hyperparameter tuning for big data using Bayesian optimisation. In *2016 23rd ICPR*. IEEE, 2016.

H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-fast Big Data Analysis*. O'Reilly Media, Inc., 2015. ISBN 144935906X.

Terence Kelly. Detecting performance anomalies in global applications. In *WORLDS*, volume 5, pages 42–47, 2005.

A. Khoshkbarforoushha, R. Ranjan, R. Gaire, E. Abbasnejad, L. Wang, and A. Y. Zomaya. Distribution based workload modelling of continuous queries in clouds. *IEEE Transactions on Emerging Topics in Computing*, 5(1):120–133, 2017.

D. M. Kline and V. L. Berardi. Revisiting squared-error and cross-entropy functions for training NN classifiers. *Neural Computing Applications*, 14(4):310–318, 2005.

R. K. Kombi, N. Lumineau, P. Lamarre, N. Rivetti, and Y. Busnel. DABS-Storm: A data-aware approach for elastic stream processing. In *Transactions on Large-Scale Data and Knowledge-Centered Systems XL*, pages 58–93. Springer, 2019.

M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. In *2015 12th ACM CF*, number 53, pages 1–8. ACM.

T. Li, Z. Xu, J. Tang, and Y. Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment*, 11(6):705–718, 2018.

G. Liao, K. Datta, and T. L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *Euro-Par 2013*. Springer, 2013.

B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *2015 35th IEEE ICDCS*, 2015.

F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE TPDS*, 29(3):572–585, 2018.

S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang. Log-based abnormal task detection and root cause analysis for Spark. In *2017 IEEE ICWS*. IEEE, 2017.

S. Lu, X. Wei, Y. Li, and L. Wang. Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th DASC/PiCom/DataCom/CyberSciTech*. IEEE, 2018.

N. Mastronarde and M. van der Schaar. Fast reinforcement learning for energy-efficient wireless communication. *IEEE Trans Signal Process.*, 59(12):6262–6266, 2011.

X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, and S. Owen. MLlib: Machine learning in Apache Spark. *JMLR*, 17(1): 1235–1241, 2016.

M. F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525–533, 1993.

W. Mu, Z. Jin, F. Liu, W. Zhu, and W. Wang. OMOPredictor: An online multi-step operator performance prediction framework in distributed streaming processing. In *2019 IEEE ISPA/BDCloud/SocialCom/SustainCom*.

N. Nguyen, M. M. H. Khan, and K. Wang. Towards automatic tuning of Apache Spark configuration. In *2018 IEEE 11th CLOUD*. IEEE, 2018.

D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *2003 4th USITS*. USENIX Association, 2003.

C. Peng, C. Zhang, C. Peng, and J. Man. A reinforcement learning approach to map reduce auto-configuration under networked environment. *Int'l J. of Security and Networks*, 12 (3):135–140, 2017.

S. Pertet and P. Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

W. Qi, Y. Li, H. Zhou, W. Li, and H. Yang. Data mining based root-cause analysis of performance bottleneck for big data workload. In *2017 IEEE HPCC/SmartCity/DSS*. IEEE, 2017.

H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), 2019.

S. Rogers and M Girolami. *A First Course in Machine Learning*. Chapman and Hall/CRC, 2nd edition, 2016.

O. Runsewe and N. Samaan. Cloud resource scaling for big data streaming applications using a Layered Multi-dimensional Hidden Markov Model. In *2017 17th IEEE/ACM CCGRID*, 2017.

G. Russo Russo, V. Cardellini, and F. Lo Presti. Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In *13th ACM DEBS*, 2019.

G. Sebestyen, A. Hangan, Z. Czako, and G. Kovacs. A taxonomy and platform for anomaly detection. In *2018 IEEE AQTR*. IEEE, 2018.

B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

S. Shahrivari. Beyond batch processing: Towards real-time and streaming big data. *IEEE Computers*, (4):117–129, 2014.

K. G. Sheela and S. N. Deepa. Review on methods to fix number of hidden neurons in neural networks. *Mathematical Problems in Engineering*, 2013.

J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. MRTuner: A toolkit to enable holistic optimization for MapReduce jobs. *Proceedings of the VLDB Endowment*, 7(13):1319–1330, 2014.

J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *2012 NISP*, pages 2951—2959, 2012.

R. S. Sutton and A. G. Barto. *Reinforcement Learning - An Introduction*. MIT Press, 2nd edition, 2018.

Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd ICDCS*, 2012.

A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, and J. Donham. Storm @Twitter. In *2014 ACM SIGMOD*. ACM, 2014.

G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of Spark based on machine learning. In *2016 IEEE 18th HPCC/SmartCity/DSS*. IEEE, 2016.

J. Xu, J. Tang, Z. Xu, C. Yin, K. Kwiat, and C. Kamhoua. A deep recurrent neural network based predictive control framework for reliable distributed stream data processing. In *2019 IEEE IPDPS*, 2019.

N. J Yadwadkar and W. Choi. Proactive straggler avoidance using machine learning. *White paper, University of Berkeley*, 2012.

L. Yang, C. Liu, J. M. Schopf, and I. Foster. Anomaly detection and diagnosis in grid environments. In *2007 ACM/IEEE Supercomputing*, 2007.

N. Yigitbasi, T. L Willke, G. Liao, and D. Epema. Towards machine learning-based auto-tuning of MapReduce. In *2013 IEEE 21st MASCOTS*. IEEE, 2013.

N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. Elastic complex event processing exploiting prediction. In *2015 IEEE Big Data*, 2015.

G. P. Zhang. Neural networks for classification: A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(4):451–462, 2000.