SAND2021-5056C

Proctor: A Semi-Supervised Performance Anomaly Diagnosis Framework for Production HPC Systems

Burak Aksar[1][0000−0003−3627−7311], Yijia Zhang[1], Emre Ates[1], Benjamin Schwaller[2], Omar Aaziz[2], Vitus J. Leung[2], Jim Brandt[2], Manuel Egele[1], and Ayse K. Coskun[1]

[1] Boston University, Boston MA 02215, USA
{baksar,zhangyj,ates,megele,acoskun}@bu.edu
[2] Sandia National Laboratories, Albuquerque NM 87123, USA
{bschwal,oaaziz,vjleung,brandt}@sandia.gov

Abstract. Performance variation diagnosis in High-Performance Computing (HPC) systems is a challenging problem due to the size and complexity of the systems. Application performance variation leads to premature termination of jobs, decreased energy efficiency, or wasted computing resources. Manual root-cause analysis of performance variation based on system telemetry has become an increasingly time-intensive process as it relies on human experts and the size of telemetry data has grown. Recent methods use supervised machine learning models to automatically diagnose previously encountered performance anomalies in compute nodes. However, supervised machine learning models require large labeled data sets for training. This labeled data requirement is restrictive for many real-world application domains, including HPC systems, because collecting labeled data is challenging and time-consuming, especially considering anomalies that sparsely occur.

This paper proposes a novel *semi-supervised framework* that diagnoses previously encountered performance anomalies in HPC systems using a limited number of labeled data points, which is more suitable for production system deployment. Our framework first learns performance anomalies' characteristics by using historical telemetry data in an unsupervised fashion. In a following fine-tuning process, we leverage supervised classifiers to identify anomaly types. While most semi-supervised approaches do not typically use anomalous samples, our framework takes advantage of a few labeled anomalous samples to *classify* anomaly types. We evaluate our framework on a production HPC system and on a testbed HPC cluster. We show that our proposed framework achieves 60% F1-score on average, outperforming state-of-the-art supervised methods by 11%, and maintains an average 0.06% anomaly miss rate.

Keywords: anomaly diagnosis · semi-supervised learning · high performance computing.

1 Introduction

Modern High-Performance Computing (HPC) systems are massive systems that perform many complex operations concurrently and they are critical for many science and engineering applications. Considering these systems' user demands and complexity, applications even with the same input deck are subject to substantial performance variations, such as running time changes of 100% or higher [27, 34]. Hidden hardware problems, shared resource contention [14, 20], fluctuating CPU frequency [35], orphan processes [18], and memory-related problems (e.g., memory leak) [5] are some common *anomalies* that cause performance variations. Some of the anomalies even force executing programs to terminate prematurely [18]. These performance variations may trigger sub-optimal scheduling and waste computing power, resulting in degraded overall computing efficiency and user dissatisfaction.

System administrators typically assess system health and identify the root causes of performance variations by gathering and inspecting telemetry data. Considering billions of telemetry data points are generated daily [4], manual analysis of system logs or resource usage data is not feasible due to being highly error-prone and time-consuming. Automated analytics, especially in the diagnosis of *anomalies*, are promising because they can reduce the mitigation time of problems, leading to the prevention of wasted computing power. Although various statistical and machine learning-based techniques have been proposed to detect anomalies in HPC systems (e.g., [25, 41, 16, 15]), one main drawback is that they require a human operator to understand the root causes (i.e., diagnose anomalies) and label anomalous data. Tuncer et al.'s recent method performs automated anomaly diagnosis using supervised machine learning successfully when labeled healthy and anomalous data is available [39]. A common disadvantage of such fully supervised approaches is that they require a large set of *labeled* data that corresponds to the normal/anomalous state of a compute node.

Borghesi et al.'s recent method is semi-supervised and focuses on detecting anomalous runs, but without the ability to diagnose root causes for performance anomalies since they only use normal data samples in training [17, 16]. Especially in production HPC systems, a large amount of telemetry data is available, but data labels are scarce. Thus, frameworks that are able to work with a limited amount of labeled data while identifying the root cause of performance anomalies would significantly improve the performance of production HPC systems.

In this paper, we propose *Proctor*, a semi-supervised performance anomaly diagnosis framework, which detects and identifies performance anomalies in compute nodes using a significantly smaller amount of labeled data compared to supervised baselines; hence Proctor is more suitable for HPC production deployment. Proctor utilizes resource usage characteristics of applications collected by monitoring frameworks to train machine learning models. We evaluate the effectiveness of *Proctor* on a production HPC system and on an HPC testbed using multiple real applications and benchmark suites with synthetic anomalies. Our specific contributions are as follows:

– A novel semi-supervised framework that, once trained, automatically detects and diagnoses known anomalies that contribute to performance variations. We argue that our proposed framework is more suitable for deployment into production HPC systems than previous works as it requires substantially less labeled data.[3]
– Demonstration of the efficacy of our framework on a production HPC system and a testbed HPC cluster. We show that *Proctor* achieves 60% F1-score on average and outperforms supervised baselines by 11% in F1-score while maintaining an average 0.06% anomaly miss rate.

The rest of the paper starts with an overview of the related work. Sec. 3 describes the technical details of the proposed framework, Sec. 4 explains our experimental methodology, Sec. 5 presents our results, and we conclude in Sec. 6.

2 Related Work and Background

Detection of anomalies in high-dimensional data is a fundamental research topic with numerous applications in the real world. Some example application fields include, but are not limited to, medical anomaly detection [40, 32], HPC telemetry data analysis [16, 15, 39], and sensor networks anomaly detection [29].

2.1 Anomaly Detection and Autoencoders

Machine learning is widely used in anomaly detection, with a variety of supervised, semi-supervised, or unsupervised approaches. Supervised models require normal and anomalous samples to classify anomaly types. In contrast to supervised methods, semi-supervised anomaly detection (SSAD) methods use labeled *normal* samples to identify anomalies. A common SSAD technique is to use autoencoders trained with normal data [30, 36]. An autoencoder is an artificial neural network (ANN) composed of three main sequential layers: the input layer, the *code* layer, and the output (or reconstruction) layer. Autoencoders do not require class/label information since all layers are operating in an unsupervised paradigm [23]. An autoencoder with more than one hidden layer is known as a deep autoencoder and is shown in Figure 1. A deep autoencoder learns to reconstruct the input data through a pair of encoder and decoder mappings, which are composed of hidden layers, as follows:

$$\overline{X} = D(E(X)), \tag{1}$$

where X is the input data, E is an encoder mapping from the input data to the code layer, D is a decoder mapping from the code layer to the output layer, and $\overline{X}$ is the reconstructed version of the input data. During the training stage, the model learns to reconstruct input data by minimizing the *reconstruction error*, which is one way of measuring how well an autoencoder learned. During

[3] Our implementation is available at: https://github.com/peaclab/Proctor

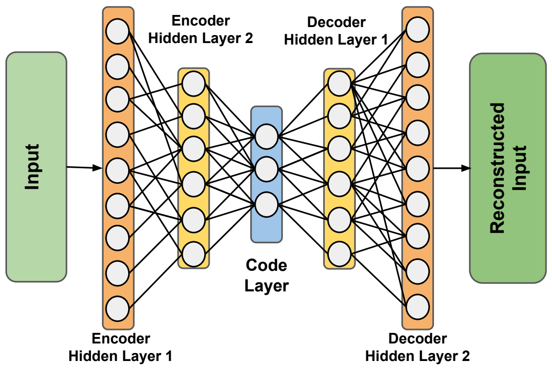

Fig. 1. A generic representation of an autoencoder with multiple hidden layers. The autoencoder learns to reconstruct the input data by learning the weights in the hidden layers.

the testing stage, an autoencoder classifies a sample as anomalous if the sample's reconstruction error is higher than the predetermined threshold. Stacked autoencoders integrate multiple autoencoders together, where the *code* layer of one autoencoder serves as the input of the other autoencoder. Deep architectures and stacked autoencoders have been shown to produce more abstract representations, improving the classification accuracy [19, 22, 13]. To perform classification with autoencoders, researchers use encoded features as inputs to supervised machine learning models (e.g., support vector machines, logistic regression, or neural networks) [26, 28].

In this work, we use autoencoders as unsupervised feature extractors, along with supervised classifiers to diagnose performance variations in HPC systems.

2.2 Machine Learning for HPC Monitoring Analytics

Due to the complexity of HPC systems and the size of the telemetry data (e.g., billions of data points per day), HPC centers have been investing in research on machine-learning-based approaches to automate performance anomaly analysis [24, 35]. Ates et al. design a random forest (RF) based framework for application classification on compute nodes [9]. Klinkenberg et al. define a supervised learning system that extracts statistical features and uses an RF classifier to detect important node failures before they occur [25]. Baseman et al. apply a technique named *classifier-adjusted density estimation* to HPC sensor data [11]. Using density estimation, they learn to generate synthetic samples. Then, both real and synthetically generated data is used to train an RF classifier and assign an "anomalousness" score to each data point to detect performance anomalies. Borghesi et al. use a simple autoencoder structure trained on only normal data instances and perform reconstruction-error-based anomaly detection in compute

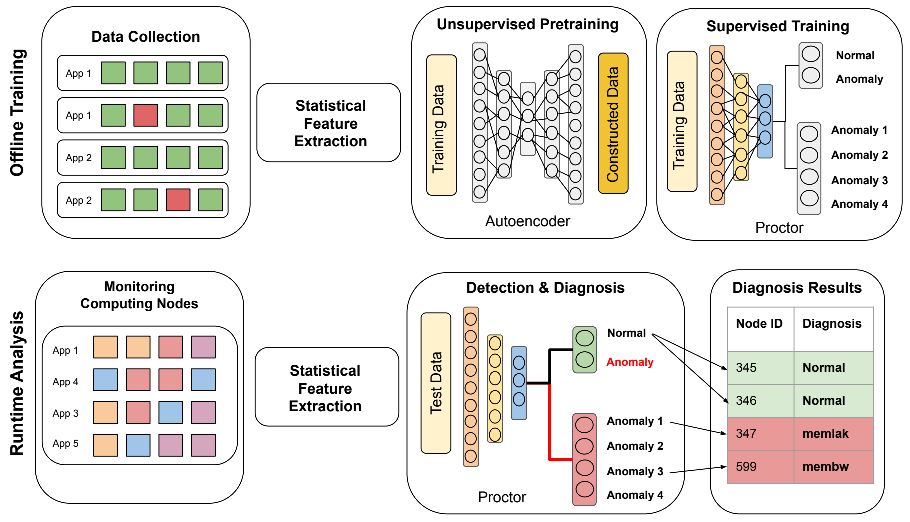

Fig. 2. The high-level architecture of *Proctor*. We collect telemetry data from normal and anomalous application runs and apply statistical feature extraction to convert raw time series into a suitable format for our autoencoder-based framework. We train an autoencoder with unlabeled normal and anomalous samples during the unsupervised pretraining stage to learn high-level characteristics. Then, we use the encoder of an autoencoder to train classifiers with a few labeled samples to diagnose anomalies. At runtime, we feed the trained model with telemetry data and classify anomalies on compute nodes.

nodes [17]. For anomaly diagnosis, which is classifying different types of performance anomalies as opposed to solely detecting anomalies, the most relevant work is Tuncer et al.'s method, where they apply statistical feature extraction along with a feature selection process to diagnose different anomaly types such as memory leak, CPU contention, and others [39].

Existing methods either detect anomalies in a fully supervised way [39] or they use semi-/unsupervised methods but only detect anomalies [16, 11] without diagnosing/classifying their root cause. Our work is distinct from related work because our proposed framework is the first to *detect* and *diagnose* performance anomalies in a semi-supervised way using substantially less labeled data compared to supervised approaches.

3 Our Proposed Framework: PROCTOR

Our main objective is to detect whether a compute node in a system exhibits anomalous behavior (i.e., causing performance variability), and if it does, we aim to classify the *type* of anomaly (e.g., memory leak or contention in a specific subsystem) in an application-agnostic fashion. We focus on anomalies that cause performance variability, where applications execute without terminating/crashing.

Such anomalies are often more challenging to detect and diagnose compared to faults that lead to errors in programs or premature termination.

We propose a semi-supervised anomaly diagnosis framework called *Proctor* based on an autoencoder and a classification layer that diagnoses performance variations on compute nodes. Figure 2 shows an overview of our framework. We collect telemetry data from compute nodes while running applications with and without synthetic anomalies. Note that our framework is independent of the underlying monitoring framework. After that, we extract the raw time series' statistical features and train an autoencoder to learn a representation (encoding) of normal and anomalous runs in an unsupervised manner. Using the trained autoencoder's *code* layer where the compressed knowledge representation exists, we perform *two-level* supervised training using different supervised classifiers. In the rest of this section, we explain these steps in detail.

3.1 Feature Extraction

We implement Tuncer et al.'s easy-to-compute statistical features [38] to convert multivariate time series data into a suitable format for *Proctor*. Some features are simple order statistics (e.g., 25th,75th, 90th percentile, and standard deviation), and some of them are useful for time series clustering such as skewness and kurtosis. This step is important because it reduces the unacceptable overhead caused by using raw time series metrics generated from thousands of compute nodes. The statistical feature extraction methodology is independent of the monitoring framework and can be used across different HPC monitoring tools such as Ganglia [3] and Examon [12].

3.2 Unsupervised Pretraining

We implement two different autoencoder topologies and compare their efficacy to make a selection. Deep autoencoders and stacked autoencoders serve as a more effective initialization due to their unsupervised nature for classification tasks when sparsely labeled data is available [6, 21]. In autoencoder, our training objective is to learn the weights for the encoding and decoding layers so that the reconstructed input is as close to the original input as possible. In another word, the goal is to minimize the difference between X and $\overline{X}$ by performing the following optimization [42]:

$$\underset{D,E}{min}\ ||X - D(E(X))||. \tag{2}$$

We train autoencoder via backpropagation, which is a way of updating the weights and biases of the layers to perform the optimization in Eq. (2). Stacked autoencoders are an extended version of autoencoders which is composed of multiple AE blocks, and the training process of the SAE is as follows:

1. Train the first AE block.

2. Fix the weights of the first AE block, and use its code layer to encode input data.
3. Train the second AE block by feeding it with the encoding generated by the first AE block.
4. Fix the weights of the second AE block.
5. Stack the first and the second AEs' code layers.

3.3 Supervised Training

We implement two different supervised training methodologies that discriminate anomaly types for anomaly diagnosis and choose the best performing one in the evaluation. The first one is *finetuning.* We freeze the pre-trained autoencoder's weights and add another fully-connected neural network layer after the encoder part. After that, we retrain the new network to classify the anomaly types as shown in the supervised training part of Fig. 2. The second methodology is using the encoded features directly as input to traditional supervised machine-learning models such as logistic regression, RF, and support vector machines. In this case, we only train the new model with the encoded data.

3.4 Detection & Diagnosis

Proctor has a *two-level classification* process. In the first level, *Proctor* decides whether a sample is normal or anomalous. If it is anomalous, we feed the sample to the diagnosis layer to identify the anomaly type.

4 Experimental Methodology

We run controlled experiments on two different HPC systems by running synthetic anomalies with a set of HPC applications. We also describe the implementation details of the two baselines we choose for anomaly diagnosis and detection. This section describes the monitoring framework, datasets, HPC applications, and performance anomalies we use to evaluate our proposed *Proctor* framework against the baselines.

4.1 HPC Systems and Applications

We conduct experiments on a testbed system, Volta, and on a production HPC system, Eclipse. Also, we run benchmarks and real applications to evaluate the performance of *Proctor* against baselines.

Volta is a Cray XC30m testbed supercomputer located at Sandia National Laboratories. Volta consists of 52 compute nodes, organized in 13 fully connected switches with four nodes per switch. Each node has 64GB of memory and two sockets, each with an Intel Xeon E5-2695 v2 CPU with 12 2-way hyper-threaded cores. To cover a representative set of HPC applications in Volta, we use NAS Parallel Benchmarks (NPB) and Mantevo Benchmark Suite. The Mantevo Benchmark Suite is for performance and scaling experiments and developed by Sandia National Laboratories. Additionally, we use the Kripke application, which is another proxy application that simulates particle transportation. We list all applications used for the experiments in Table 1. We run each application across different numbers of nodes (4 or 32) for 10-15 minutes using different application input decks.

Eclipse is a production HPC system located at Sandia National Laboratories. Eclipse consists of 1488 compute nodes, and it is capable of 1.8 petaflops. Each node has 128GB memory and two sockets, each with 18 E5-2695 v4 CPU cores [1]. In Eclipse, we use six applications, LAMMPS, HACC, sw4, ExaMiniMD, SWFFT, and sw4lite. Among them, there are three real applications: LAMMPS, a molecular dynamics simulation with a focus on materials modeling; HACC, an extreme-scale cosmological simulation; sw4, a popular 3D seismic model. The other three, ExaMiniMD, SWFFT, and sw4lite, are proxy applications from the ECP Proxy Apps Suite [2]. We list all applications used for the experiments in Table 2. We run each application on 4 nodes for 20-45 minutes.

Table 1. Applications used in Volta for data collection.

Benchmark	Application	Description
NAS	BT	Block tri-diagonal solver
	CG	Conjugate gradient
	FT	3D Fast Fourier Transform
	LU	Gauss-Seidel solver
	MG	Multi-grid on meshes
	SP	Scalar penta-diagonal solver
Mantevo	miniMD	Molecular dynamics
	CoMD	Molecular dynamics
	miniGhost	Partial differential equations
	miniAMR	Stencil calculation
Other	Kripke	Particle transport

4.2 Monitoring Framework

We use the Lightweight Distributed Metric Service (LDMS) to collect telemetry data from different subsystems [4]. LDMS is a low overhead monitoring framework for HPC systems with a high sampling rate. LDMS collects data simultaneously for each subsystem component (e.g., memory-related metrics, network counters) across the whole system [33]. At every second, LDMS can collect more than a thousand metrics per node in the categories as described below:

- Memory (e.g., currently free, active, inactive memory)
- CPU (e.g., per-core and overall idle time, I/O wait time)

– Network (e.g., received/transmitted packets, average packet size, link status)
– Shared File System (e.g., open, read, write counts)
– Cray performance counters (e.g., power consumption, write-back counters)
– Virtual Memory (e.g., free, active and inactive pages)

LDMS is deployed on both systems and it constantly monitors the health of the systems [4, 33]. We collect 806 metrics and 721 metrics from Eclipse and Volta, respectively. We fill out missing metric values with linear interpolation and take the difference of cumulative counters since we are interested in the increase. We also exclude the first and last 60 seconds of the collected time series to avoid fluctuations during the initialization and termination phases of applications.

Table 2. Applications used in Eclipse for data collection.

Benchmark	Application	Description
Real Applications	LAMMPS	Molecular dynamics
	HACC	Cosmological simulation
	SW4	Seismic modeling
ECP Proxy Suite	ExaMiniMD	Molecular dynamics
	SWFFT	3D Fast Fourier Transform
	SW4LITE	Numerical kernel optimizations

4.3 Synthetic Anomalies

To learn individual anomaly signatures and detect them at runtime, *Proctor* needs a few labeled samples that show anomalous characteristics. To systematically train and test our framework, we use synthetic anomalies from HPC Performance Anomaly Suite (HPAS) [10] to mimic anomalous behavior during an application run. HPAS is an open-source performance anomaly suite to reproduce performance variations. These synthetic anomalies target five major subsystems: CPU, cache, memory, network, and shared storage. We inject anomalies with multiple configurations to mimic different performance variation levels, as listed in Table 3. While running a multi-node application, we run a synthetic anomaly on a single node in Volta, and we run a synthetic anomaly on every node that the application uses in Eclipse. Each compute node data is labeled with an anomaly type if an anomaly is injected, otherwise labeled as normal.

Table 3. A list of anomalies used in experiments.

Anomaly type	Anomaly behavior	Configuration
CPU intensive process	Arithmetic operations	-u 100%, 80%
Cache contention	Cache read & write	-c L1, L2 / -m 1, 2
Memory bandwidth contention	Uncached memory write	-s 4K, 8K, 32K
Memory leakage	Increasingly allocate & fill memory	-s 1,3,10 M / -p 0.2,0.4,1

4.4 Baselines

We implement two baseline methods to compare against *Proctor* under different experimental scenarios. The first one is the framework proposed by Tuncer et al. [39] (referred to as RF-Tuncer) that uses statistical feature extraction and an RF classifier. The second one is the autoencoder-based anomaly detection approach proposed by Borghesi et al. [16] (referred to as AE-Borghesi).

RF-Tuncer [39] uses statistical feature extraction and feature selection strategies and combines them with an RF classifier to diagnose anomaly types [39]. They use LDMS to collect different metrics (e.g., memory metrics, CPU metrics) while applications operate with and without anomalies at every second. They label each node with the injected anomaly type during the application run. Application runs without injected anomalies are labeled "normal". During an offline training phase, they train supervised models and test the saved models at runtime after statistical feature extraction and feature selection are applied.

AE-Borghesi [17] trains an autoencoder with only *normal* samples and detects anomalies according to a statistically determined threshold. It is important to note that their method is limited to anomaly detection instead of classifying anomaly types. *Proctor* can also detect anomalies by slightly modifying the network in the supervised training stage. Borghesi et al. use *Examon* [7] data collection infrastructure to monitor the D.A.V.I.D.E [12] HPC system which has 45 compute nodes. Examon can collect up to 170 metrics with 5s and 10s granularity, respectively, for Intelligent Platform Management Interface (IPMI) and Open-POWER POWER8 on-chip controller (OCC) metrics. They use coarse-grained aggregated telemetry data with a 5 min aggregation time window. To mimic their data collection schema, we apply the same technique. The authors inject three anomalous policies that change CPU frequency, clock speed, and power consumption to mimic anomalous behavior (e.g., *powersave* sets the CPU frequency to the lowest available). They train an autoencoder with only normal data (i.e., intervals without anomaly injection) and select the threshold to detect anomalies. To select a threshold, they try different percentiles of the reconstruction error of training data and select the value that gives the best F1-score in the validation data. During testing, if a sample has a higher reconstruction error than the threshold, it is labeled as having an anomaly.

4.5 Implementation Details

Proctor: We implement our framework in Tensorflow. We create a hyperparameter space using the following values and search the space to find the best values for the autoencoder:

1. Batch size: 32, 64, 128, and 256
2. Number of neurons in hidden a layer: 200, 500, 1000, 2000

3. Number of hidden layers: 1, 2, 3, 5
4. Number of epochs: 50, 100, 300, 500, 1000, 5000
5. Optimizer: Adam, Adadelta, SGD
6. Dropout: 0, 0.1, 0.2, 0.3

When we find the best parameters for the autoencoder, we stack them to experiment with stacked autoencoders. For the supervised training stage, we use a neural network, Support Vector Machine (SVM), and Logistic Regression (LR). All classifiers are trained using the *one-versus-rest* strategy, which creates an individual classifier for each class. For the neural networks, we use *Adam* optimizer and minimize *Categorical Cross-Entropy* loss.

The final structure of the *Proctor* is a deep autoencoder with 2000 neurons in the code layer and uses SVM and LR for the supervised training part. Stacked autoencoders also perform similarly, but we choose deep autoencoders because of a lower false alarm rate. We use the *Adadelta* optimizer, which enforces a monotonically decreasing learning rate and minimizes *Mean Squared Error* (MSE) during the training with a 20% validation split. We also set *EarlyStopping* callback, which stops when the chosen performance measure stops improving.

AE-Borghesi: We adopted the following network topology according to the descriptions of Borghesi et al. [17]:

1. An input layer
2. A dense *code* layer with a number of neurons ten times larger than input neurons with *Rectified Linear Units [31]* (ReLU) activation and an *L1 norm* [8] regularizer
3. An output layer with a number of neurons equal to input features with *Linear* activations

The model is trained with *Adam* optimizer, which finds individual learning rates for each parameter, by minimizing the *Mean Absolute Error* (MAE) for 100 epochs with a batch size of 32. We conduct a hyparameter search for the number of neurons in the code layer so as not to put AE-Borghesi at a disadvantage. We also implement their approach with *Dropout* [37] layers as they suggested [16]. However, it gives slightly worse results than the previous topology, so we only present the best results.

RF-Tuncer: We implement feature extraction and feature selection using *scipy-stats* module. We choose the best performing classifier, RF, and set the number of decision trees to 100 after hyperparameter search. For RF, we use *scikit-learn* implementation.

5 Evaluation

In this section, we first explain the metrics and datasets we use in our evaluation. Next, we compare the anomaly detection and diagnosis results of our framework against baselines. Then, we evaluate the performance in cases when an unseen anomaly exists.

5.1 Performance Metrics

We report our evaluation results with 5-fold stratified cross-validation for each experimental scenario and observe the F1-score, anomaly miss rate (i.e., false negative rate), and false alarm rate (i.e., false positive rate) for different labeled data percentages. F1-score is the harmonic mean of precision and recall and it is widely used in multiclass classification problems. We calculate the macro average F1-score which does not take label imbalance into account, hence treating all classes equally. Note that this is important in imbalanced datasets where the number of normal data points are in majority compared to anomalous data points. To assess anomaly detection performance (i.e., distinguishing between normal versus anomalous) of the models, we use false alarm rate which indicates the percentage of normal runs identified as one of the anomaly types, and anomaly miss rate, which indicates the percentage of anomalous runs (any anomaly) identified as normal. To mitigate noise in the results, we run each classifier ten times and average the results.

$$False\ Alarm\ Rate = \frac{False\ Positives}{False\ Positives + True\ Negatives} \tag{3}$$

$$Anomaly\ Miss\ Rate = \frac{False\ Negatives}{False\ Negatives + True\ Positives} \tag{4}$$

5.2 Dataset Preparation

We devise three experimental scenarios to evaluate the performance of *Proctor*, AE-Borghesi, and RF-Tuncer. While preparing datasets for the proposed experimental scenarios, we use 5-fold stratified cross-validation, and we ensure that any training or testing set contains every application and anomaly type. The Eclipse dataset has 1526 normal samples and 2304 anomalous samples where each anomaly type has an equal number of samples. A sample refers to the telemetry data collected during an application run on a compute node. We use 611 normal samples in training and we set the anomaly ratio to 10%, i.e., the number of anomalous runs divided by all runs, to mimic a production system scenario where anomalous runs are rare compared to normal runs. The Volta dataset has 18980 normal samples and 1932 anomalous samples. We use 5694 normal samples in the training and set the anomaly ratio to 10%. We use the remaining samples to create test data. We fit a *MinMax* scaler to the training data, where each feature value is scaled between 0 and 1, and then use the same scaler in the test data.

For the supervised training part (only for *Proctor* and RF-Tuncer), we mimic a case where labeled data are accumulating over time, i.e., we start from having only a few labeled data (e.g., 1-2 labeled example per class) and increase the number of labeled data gradually. Chosen labeled data percentages are the following: 2%, 3%, 4%, 5%, 6%, 8%, 10% for Eclipse, and 0.1%, 0.15%, 0.2%, 0.25%, 0.30%, 0.35% for Volta datasets. Chosen labeled data percentages are different due to the size of the datasets. In the Eclipse dataset, when the labeled data percentage is 10%, it corresponds to approximately 65 labeled samples in total; in the Volta dataset, when the labeled data percentage is 0.35%, it corresponds to approximately 25 labeled samples in total.

5.3 Anomaly Detection Results

The main goal of this scenario is to compare *Proctor*'s performance with AE-Borghesi for anomaly detection across different labeled data percentages. For the anomaly detection task, all anomalies are labeled with the same label regardless of their types. In the unsupervised pre-training part, *Proctor* uses the whole training data without any supervision (i.e., data are unlabeled). In the supervised training part, we train RF-Tuncer and *Proctor* with the selected labeled data and evaluate their performance in the same test set. Then, we repeat the same procedure for each predetermined labeled percentage value.

We train AE-Borghesi by using normal data in the training set. It is important to note that this method does not have a supervised training part like *Proctor* and RF-Tuncer. We choose the 63^{th} percentile of the mean absolute reconstruction error as a threshold since it achieves the best F1-score in the validation data and it is used to classify whether a run is anomalous or not.

As shown in Fig. 3, *Proctor* outperforms the baselines in F1-score and anomaly miss rate for most cases even with very few labeled data points. Both *Proctor* and RF-Tuncer perform similarly in terms of false alarm rate. *Proctor* outperforms RF-Tuncer by 50% on average in anomaly miss rate.

Due to the simple thresholding used in AE-Borghesi as well as the existence of multiple anomaly types in our datasets, AE-Borghesi performs poorly compared to others. In addition, AE-Borghesi needs to be trained with only normal data points, so a system administrator or subject matter expert needs to ensure that system health status is normal to train AE-Borghesi. On the other hand, *Proctor* can be directly deployed and continuously trained with available telemetry data regardless of the system's health status. Besides training *Proctor* with unlabeled telemetry data, when a subject matter expert labels some anomalous events, these labeled data can be used in the supervised training part of *Proctor*.

5.4 Anomaly Diagnosis Results

The main goal of this scenario is to compare *Proctor*'s classification F1-score with RF-Tuncer for anomaly diagnosis across different labeled data percentages. In the unsupervised pre-training part, *Proctor* uses the whole training data without any supervision. In the supervised training part, we train RF-Tuncer and *Proctor* with a selected labeled data percentage and evaluate their performance in a constant test set. We do the same for other labeled data percentage values.

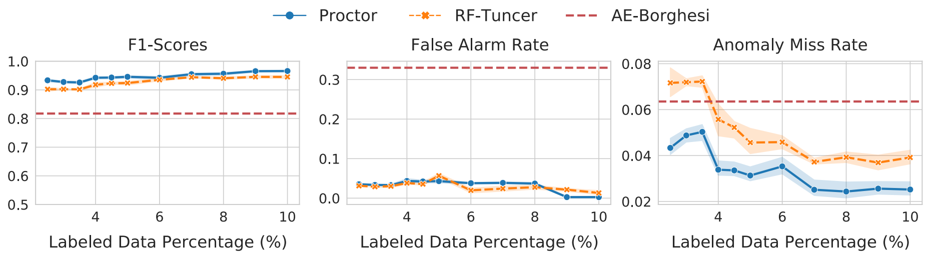

Fig. 3. Comparing the anomaly detection performance of *Proctor* with AE-Borghesi in Eclipse dataset. *Proctor* outperforms the baselines in F1-score and anomaly miss rate while maintaining a similar false alarm rate with RF-Tuncer.

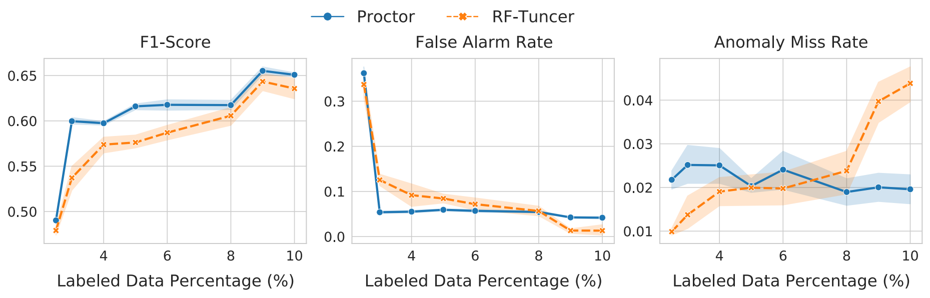

Fig. 4. Comparing the anomaly diagnosis performance of *Proctor* with RF-Tuncer in Eclipse dataset. Proctor performs better in F1-score and false alarm rate while maintaining a stable anomaly miss rate.

Figure 4 shows macro average F1-scores for our method and RF-Tuncer in the Eclipse dataset. *Proctor* outperforms RF-Tuncer by 4.5% on average (and up to 11%) while maintaining a low false alarm rate and anomaly miss rate. RF-Tuncer performs slightly better in terms of anomaly miss rate when the labeled data percentage is less than 5%. However, the anomaly miss rate of RF-Tuncer increases when the labeled data percentage increases, whereas the anomaly miss rate of *Proctor* is stable and keeps below 2.5%.

Figure 5 shows macro average F1-scores for *Proctor* and RF-Tuncer in the Volta dataset. *Proctor* outperforms RF-Tuncer for most of the cases in terms of all categories until we have approximately 20 labeled data samples in total. In terms of F1-score, *Proctor* outperforms RF-Tuncer by 25% on average (and up to 50%) and maintains similar alarm and miss rates to RF-Tuncer. RF-Tuncer achieves a similar F1-score to *Proctor* faster in the Volta dataset compared to the Eclipse dataset. The main reason behind this is less complex application characteristics in the Volta dataset.

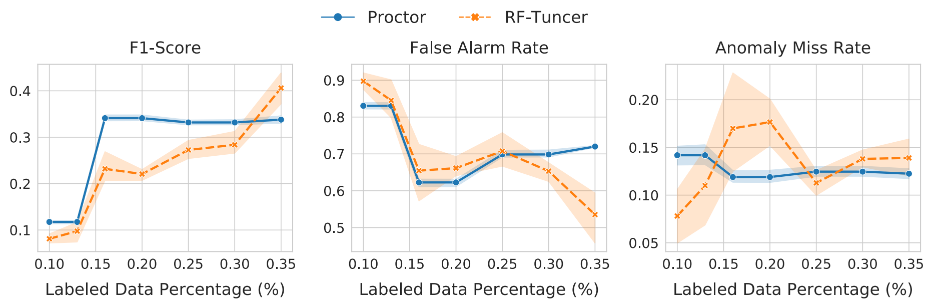

Fig. 5. Comparing the anomaly diagnosis performance of *Proctor* with RF-Tuncer in Volta dataset. *Proctor* outperforms RF-Tuncer for most of the cases across all categories.

5.5 Unseen Anomaly Detection Results

This scenario's primary goal is to evaluate the performance of *Proctor* and RF-Tuncer for unknown anomalies in the test set. Since different performance anomalies exist, it is common to observe anomalies other than those used during training, especially in the production environment. We follow the same unsupervised pretraining and supervised training approaches described above except for one difference: we remove the selected unseen anomaly type from the training set during the supervised training and keep the other anomalies. After the training, we test the model on the same training data, including the removed anomaly, to determine a threshold. While changing the confidence threshold, we choose the threshold that provides the highest F1-score and then perform testing on a dataset that consists of all anomalies. We label the sample as *unknown* if the model's highest confidence is lower than the selected threshold. RF-Tuncer uses a multiclass RF, and it requires all classes to exist in the training data; however, not to put RF-Tuncer at a disadvantage, we apply the *one-versus-rest* strategy to an RF classifier as well. We experiment with all labeled data percentages and report selected percentage results individually in all evaluation metrics.

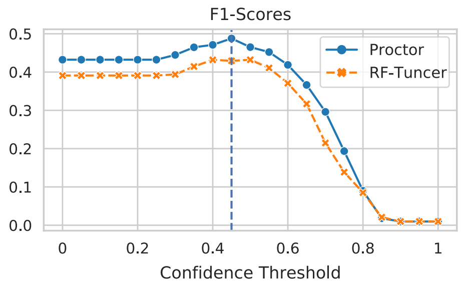

Fig. 6. Choosing a threshold that gives the highest F1-score by sweeping confidence thresholds.

Figure 6 shows the F1-score across different confidence thresholds. We choose 0.45 as a threshold and compare both method's anomaly diagnosis performance in Fig.7. Here, *Proctor* outperforms the baseline by 10% on average in terms of

F1-score while maintaining a 3x lower false alarm rate on average. RF-Tuncer's anomaly miss rate is 3x better than Proctor's, however, both rates are very close to zero.

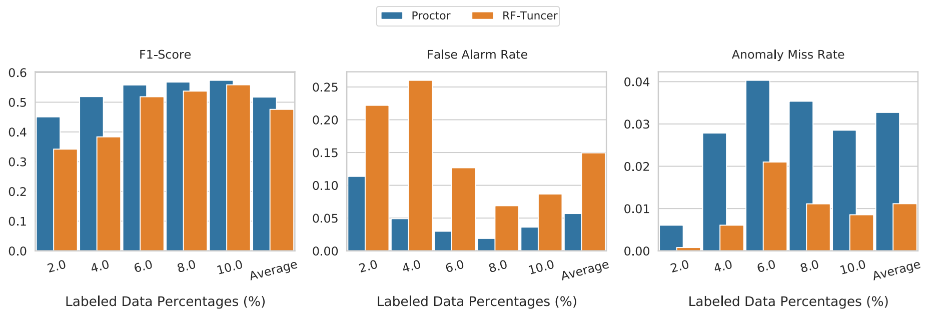

Fig. 7. When there are unknown anomaly types, *Proctor* performs better than RF-Tuncer in terms of F1-score and false alarm rate.

6 Conclusion

Performance variation in HPC systems reduces the efficiency of resource utilization and wastes computing power. Considering the growing size and complexity of HPC systems, automated performance anomaly diagnosis has become increasingly crucial for robust and efficient service. We proposed *Proctor*, a semi-supervised anomaly detection and diagnosis framework for limited labeled data scenarios in production systems. We evaluated our platform in two different HPC systems, including a production level HPC system. We demonstrated that our approach is superior to other state-of-the-art approaches in terms of F1-score, anomaly miss rate, and false alarm rate when only limited labeled data are available. We also showed that our framework is robust against unseen anomalies during training and successfully labeled them as "unknown".

As a future work, we will focus on deploying our framework into a production HPC machine and integrating a user / system administrator feedback system that allows us to label suspicious application runs for continuous model improvement. Furthermore, we will focus on generative machine learning models to synthetically generate anomalous application runs to achieve a higher diagnosis rate with our proposed framework.

Acknowledgment

This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell

International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

1. https://hpc.sandia.gov/
2. https://proxyapps.exascaleproject.org/
3. Ganglia monitoring system, http://ganglia.info/
4. Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., Tucker, T.: The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 154–165 (2014)
5. Agelastos, A., Allan, B., Brandt, J., et al.: Toward rapid understanding of production hpc applications and systems. In: International Conference on Cluster Computing. pp. 464–473. IEEE (2015)
6. Agrawal, P., Girshick, R., Malik, J.: Analyzing the performance of multilayer neural networks for object recognition. In: European conference on computer vision. pp. 329–344. Springer (2014)
7. Ahmad, W.A., Bartolini, A., Beneventi, F., et al.: Design of an energy aware petaflops class high performance cluster based on power architecture. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 964–973. IEEE (2017)
8. Alain, G., Bengio, Y.: What regularized auto-encoders learn from the data-generating distribution. The Journal of Machine Learning Research **15**(1), 3563–3593 (2014)
9. Ates, E., Tuncer, O., Turk, A., Leung, V.J., Brandt, J., Egele, M., Coskun, A.K.: Taxonomist: Application detection through rich monitoring data. In: European Conference on Parallel Processing. pp. 92–105. Springer (2018)
10. Ates, E., Zhang, Y., Aksar, B., et al.: Hpas: An hpc performance anomaly suite for reproducing performance variations. In: Proceedings of the 48th Intl. Conference on Parallel Processing. p. 1–10. ACM (Aug 2019)
11. Baseman, E., Blanchard, S., DeBardeleben, N., Bonnie, A., Morrow, A.: Interpretable anomaly detection for monitoring of high performance computing systems. In: Outlier Definition, Detection, and Description on Demand Workshop at ACM SIGKDD. San Francisco (Aug 2016) (2016)
12. Beneventi, F., Bartolini, A., Cavazzoni, C., Benini, L.: Continuous learning of hpc infrastructure models using big data analytics and in-memory processing tools. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017. pp. 1038–1043 (2017)
13. Bengio, Y.: Learning deep architectures for AI. Now Publishers Inc (2009)
14. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2013)

15. Bodik, P., Goldszmidt, M., Fox, A., Woodard, D.B., Andersen, H.: Fingerprinting the datacenter: automated classification of performance crises. In: Proceedings of the 5th European conference on Computer systems. pp. 111–124 (2010)
16. Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., Benini, L.: A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems. Engineering Applications of Artificial Intelligence **85**, 634–644 (Oct 2019)
17. Borghesi, A., Bartolini, A., Lombardi, M., et al.: Anomaly detection using autoencoders in high performance computing systems. Proceedings of the AAAI Conference on Artificial Intelligence **33**, 9428–9433 (Jul 2019), arXiv: 1811.05269
18. Brandt, J., Chen, F., et al.: Quantifying effectiveness of failure prediction and response in hpc systems: Methodology and example. In: 2010 Intl. Conf. on Dependable Systems and Networks Workshops (DSN-W). pp. 2–7. IEEE (2010)
19. Ciregan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: 2012 IEEE conference on computer vision and pattern recognition. pp. 3642–3649. IEEE (2012)
20. Dorier, M., Antoniu, G., Ross, R., et al.: Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 155–164. IEEE (2014)
21. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 580–587 (2014)
22. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural computation **18**(7), 1527–1554 (2006)
23. Hinton, G.E., Zemel, R.S.: Autoencoders, minimum description length and helmholtz free energy. In: Proceedings of the 6th Intl. Conference on Neural Information Processing Systems. p. 3–10. NIPS'93, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
24. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Computing Surveys (CSUR) **48**(1), 1–35 (2015)
25. Klinkenberg, J., Terboven, C., Lankes, S., Müller, M.S.: Data mining-based analysis of hpc center operations. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 766–773. IEEE (2017)
26. Kunang, Y.N., Nurmaini, S., Stiawan, D., Zarkasi, A., Jasmir, F.: Automatic features extraction using autoencoder in intrusion detection system. In: 2018 International Conference on Electrical Engineering and Computer Science (ICECOS). pp. 219–224. IEEE (2018)
27. Leung, V.J., et al.: Algorithmic support for commodity-based parallel computing systems. Tech. rep., Sandia National Laboratories (2003)
28. Liu, G., et al.: A stacked autoencoder-based deep neural network for achieving gearbox fault diagnosis. Mathematical Problems in Engineering **2018** (2018)
29. Luo, T., Nagarajan, S.G.: Distributed anomaly detection using autoencoder neural networks in wsn for iot. In: 2018 IEEE Intl. Conference on Communications (ICC). pp. 1–6. IEEE (2018)
30. Minhas, M.S., Zelek, J.: Semi-supervised anomaly detection using autoencoders. arXiv:2001.03674 [cs, eess, stat] (Jan 2020), http://arxiv.org/abs/2001.03674, arXiv: 2001.03674
31. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML (2010)

32. Sato, D., Hanaoka, S., Nomura, Y., et al.: A primitive study on unsupervised anomaly detection with an autoencoder in emergency head ct volumes. In: Medical Imaging 2018: Computer-Aided Diagnosis. vol. 10575, p. 105751P. International Society for Optics and Photonics (2018)
33. Schwaller, B., Tucker, N., Tucker, T., Allan, B., Brandt, J.: Hpc system data pipeline to enable meaningful insights through analysis-driven visualizations. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER). p. 433–441. IEEE (Sep 2020), https://ieeexplore.ieee.org/document/9229587/
34. Skinner, D., Kramer, W.: Understanding the causes of performance variability in hpc workloads. In: IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005. pp. 137–149. IEEE (2005)
35. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al.: Addressing failures in exascale computing. The International Journal of High Performance Computing Applications **28**(2), 129–173 (2014)
36. Song, H., Jiang, Z., et al.: A hybrid semi-supervised anomaly detection model for high-dimensional data. Computational intelligence and neuroscience **2017** (2017)
37. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research **15**(1), 1929–1958 (2014)
38. Tuncer, O., Ates, E., Zhang, Y., et al.: Diagnosing performance variations in hpc applications using machine learning. In: Intl. Supercomputing Conference. pp. 355–373. Springer (2017)
39. Tuncer, O., Ates, E., Zhang, Y., et al.: Online diagnosis of performance variation in hpc systems using machine learning. IEEE Transactions on Parallel and Distributed Systems **30**(4), 883–896 (2018)
40. Wang, K., Zhao, Y., Xiong, Q., Fan, M., Sun, G., Ma, L., Liu, T.: Research on healthy anomaly detection model based on deep learning from multiple time-series physiological signals. Scientific Programming **2016** (2016)
41. Yu, L., Lan, Z.: A scalable, non-parametric method for detecting performance anomaly in large scale computing. IEEE Transactions on Parallel and Distributed Systems **27**(7), 1902–1914 (2015)
42. Zhou, C., Paffenroth, R.C.: Anomaly detection with robust deep autoencoders. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 665–674 (2017)