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 the following 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 [30, 12]. Hidden hardware problems, shared resource contention [12, 18], fluctuating CPU frequency [39], orphan processes [16], and memory-related problems (e.g., memory leak) [2] are some common *anomalies* that cause performance variations. Some of the anomalies even force executing programs to terminate prematurely [16]. 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 [1], 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., [27, 45, 14, 13]), 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 [43]. 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 [15, 14]. 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 [44, 37], HPC telemetry data analysis [14, 13, 43], and sensor networks anomaly detection [32].

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 [33, 40]. 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 [25]. 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

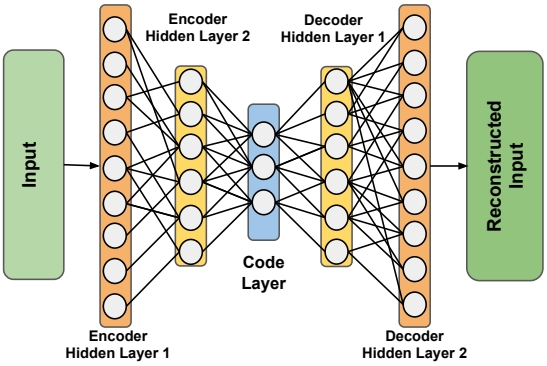

Fig. 1. A generic representation of an autoencoder with multiple hidden layers. The autoencoder learns to reconstruct the input data by learning the weights in the hidden layers.

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 [17, 24, 11]. To perform classification with autoencoders, researchers use encoded features as inputs to supervised machine learning models such as support vector machines (SVM), logistic regression (LR), or neural networks [28, 31].

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 [26, 39]. Ates et al. design a random forest (RF) based framework for application classification on compute nodes [6]. 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 [27]. Baseman et al. apply a technique named *classifier-adjusted density estimation* to HPC sensor data [9]. 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

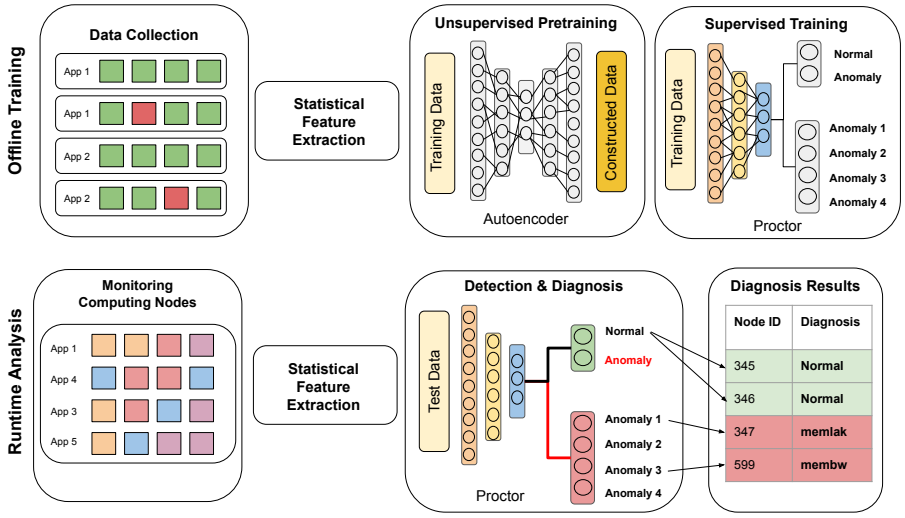

Fig. 2. The high-level architecture of *Proctor*. We collect telemetry data from normal and anomalous application runs and apply statistical feature extraction to convert raw time series into a suitable format for our autoencoder-based framework. We train an autoencoder with unlabeled normal and anomalous samples during the unsupervised pretraining stage to learn high-level characteristics. Then, we 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 [15]. 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 [43].

Existing methods either detect anomalies in a fully supervised way [43] or they use semi-/unsupervised methods but only detect anomalies [14, 9] 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, followed by 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 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 samples in an unsupervised manner. In *Proctor*, a sample refers to the entire set of telemetry data collected during an application run on a compute node. Based on the autoencoder's encoder mapping output and using some labeled normal and anomalous samples, we train supervised classifiers that are able to diagnose anomalies. At runtime, Proctor then applies the trained model on collected telemetry samples to detect and diagnose performance anomalies. We next explain these steps in detail.

3.1 Feature Extraction

We implement Tuncer et al.'s easy-to-compute statistical features [42] to convert multivariate time series data into a suitable format for *Proctor*. Some features are simple order statistics (e.g., 25^{th},75^{th}, and 90^{th} percentiles, and standard deviation), and some of them are useful for time series clustering such as skewness and kurtosis. This step reduces the overhead that would be 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 Lightweight Distributed Metric Service (LDMS) [1], Ganglia [20] or Examon [10].

3.2 Unsupervised Pretraining

We implement two different autoencoder topologies, deep autoencoders and stacked autoencoders [31], and compare their efficacy to make a selection. Deep autoencoders and stacked autoencoders serve as effective pretraining methods due to their unsupervised nature for classification tasks when many unlabeled samples are available [3, 21].

In the autoencoder, our training objective is to learn the weights for the encoder and decoder layers so that the reconstructed input is as close to the original input as possible. In other words, the goal is to minimize the difference between X and $\overline{X}$ by performing the following optimization [46]:

$$\min_{D,E} \; ||X - D(E(X))||. \tag{2}$$

We train the autoencoder via backpropagation, which is a way of updating the weights and biases of the layers to perform the optimization in Eq. (2).

We use deep autoencoders in the rest of this paper as they provide higher prediction accuracy in our results compared to stacked autoencoders.

3.3 Supervised Training

For anomaly diagnosis, we implement two different supervised training methods that differentiate anomaly types and choose the best performing one in the evaluation. The first one is *fine-tuning*. 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 method uses the encoded features directly as input to traditional supervised machine-learning models such as LR, RF, and SVM. In our experiments, the second method provides higher accuracy so we only train the supervised models with the encoded data in the rest of the paper.

3.4 Detection and Diagnosis at Runtime

At runtime, Proctor collects telemetry data from compute nodes using a monitoring framework and applies statistical feature extraction. Then, we use the model trained on these features for diagnosis. As described earlier, *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 two baseline methods for anomaly detection and diagnosis, and compare Proctor against these baselines. This section describes the monitoring framework that collects system telemetry data, data sets for anomaly diagnosis, HPC applications, and performance anomalies we use to evaluate our proposed *Proctor* framework.

4.1 HPC Systems and Applications

We conduct experiments on a testbed system, Volta, and on a production HPC system, Eclipse. We run both 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) [8] and Mantevo Benchmark Suite [23]. The Mantevo Suite was developed by Sandia National Laboratories for performance and scaling experiments. In addition, we use the Kripke application, which is a proxy

application that simulates particle transportation [29]. We list all applications used in our experiments in Table 1. We run each application across 4 or 32 compute nodes 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. In the experiments on 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 [36]; HACC, an extreme-scale cosmological simulation [22]; sw4, a popular 3D seismic model [35]. The other three, ExaMiniMD, SWFFT, and sw4lite, are proxy applications from the ECP Proxy Apps Suite [19]. We list all applications used in our experiments in Table 2. We run each application on 4 nodes for 20-45 minutes.

Table 1. Applications we run on 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 LDMS to collect telemetry data from different subsystems. 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, etc.) across the whole system [38]. At every second, LDMS collects hundreds of 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 [1, 38]. We collect 806 metrics and 721 metrics from Eclipse and Volta, respectively. We fill out any missing metric values using linear interpolation and calculate the difference of cumulative counter values since we are interested in the change. We also exclude the first and last 60 seconds of the collected time series for each application to avoid any fluctuations during the initialization and termination phases.

Table 2. Applications we run on 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 exhibit anomalous characteristics. To systematically train and test our framework, we use synthetic anomalies from the HPC Performance Anomaly Suite (HPAS) [7] to mimic anomalous behavior during an application run. HPAS is an open-source performance anomaly suite to reproduce performance variations. Synthetic anomalies in HPAS, 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 the HPAS anomalies used in our 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*. The first one is the framework proposed by Tuncer et al. [43] (referred to as RF-Tuncer), which

uses statistical feature extraction and a fully supervised RF classifier. The second one is the autoencoder-based anomaly detection approach proposed by Borghesi et al. [14] (referred to as AE-Borghesi).

RF-Tuncer [43] uses statistical feature extraction and feature selection strategies and combines them with an RF classifier to diagnose anomaly types [43]. They use LDMS to collect different metrics (e.g., memory metrics, CPU metrics) while applications run 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 [15] 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 the *Examon* [4] data collection infrastructure to monitor the D.A.V.I.D.E [10] HPC system which has 45 compute nodes. Examon collects up to 170 metrics with 5s and 10s granularity for Intelligent Platform Management Interface (IPMI) and OpenPOWER POWER8 on-chip controller (OCC) metrics, respectively. They use coarse-grained aggregated telemetry data with a 5-minute aggregation time window. To mimic their data collection schema, we apply the same aggregation 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 a threshold to detect anomalies. To select this threshold, they vary the percentiles of the reconstruction error observed in the training data and select the value that gives the best F1-score in the validation data. At runtime, if a sample has a higher reconstruction error than the threshold, it is labeled as anomalous.

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

After finding the best parameters for the autoencoder, we stack them to experiment with stacked autoencoders. For the supervised training stage, we experiment with a neural network, an SVM, and an 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 *Proctor* includes a deep autoencoder with 2000 neurons in the code layer and uses SVM and LR for the supervised training part. Stacked autoencoders perform similarly to deep autoencoders, but we choose deep autoencoders because of their lower false alarm rate. We use the *Adadelta* optimizer, which enforces a monotonically decreasing learning rate and minimizes *Mean Squared Error* 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. [15]:

1. An input layer,
2. A dense *code* layer with a number of neurons ten times larger than input neurons with *Rectified Linear Units [34]* activation and an *L1 norm* [5] regularizer,
3. An output layer with a number of neurons equal to input features with *Linear* activations.

We train the AE-Borghesi model with the *Adam* optimizer, which finds individual learning rates for each parameter by minimizing the *Mean Absolute Error* 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* [41] layers as the authors suggested [14]. However, our implementation with dropout layers gives slightly worse results than the original 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 data sets we use in our evaluation. Then, we compare the anomaly detection and diagnosis results of our framework against the baselines. We also evaluate the performance in cases when a previously unseen anomaly type exists in the test data.

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 percentages of labeled data. 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 data sets where the number of normal data points is in the overwhelming majority compared to anomalous data points. To assess anomaly detection performance (i.e., distinguishing between normal versus anomalous) of the models, we use the false alarm rate which indicates the percentage of normal runs identified as one of the anomaly types, and the anomaly miss rate, which indicates the percentage of anomalous runs (any anomaly) identified as normal. To improve confidence in our 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 Data Set Preparation

We devise three experimental scenarios to evaluate the performance of *Proctor*, AE-Borghesi, and RF-Tuncer. While preparing data sets for the proposed experimental scenarios, we use 5-fold stratified cross-validation, and we ensure that any training or testing data set contains every application and anomaly type. The Eclipse data set has 1526 normal samples and 2304 anomalous samples, where each anomaly type is equally represented among the anomalous samples. We use 611 normal samples and 68-70 anomalous samples in training, representing an anomaly ratio of 10% (i.e., anomaly ratio is the number of anomalous runs divided by all runs). This anomaly ratio mimics a production system scenario where anomalous runs are rare compared to normal runs. The Volta data set has 18980 normal samples and 1932 anomalous samples. We use 5694 normal samples and 618-620 anomalous samples in training, representing an anomaly ratio of 10%. In both data sets, samples that are not used during training are placed in the testing data set. We fit a *MinMax* scaler to the training data set, where each feature value is scaled between 0 and 1, and then use the same scaler in the testing data set.

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 data sets. Chosen labeled data percentages are different

due to the size of the data sets. In the Eclipse data set, when the labeled data percentage is 10%, it corresponds to approximately 65 labeled samples in total; in the Volta data set, 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 in anomaly detection is to compare *Proctor*s performance with AE-Borghesi and RF-Tuncer for anomaly detection across different labeled data percentages. For the anomaly detection task, all anomalies are labeled with the same label (i.e., without diagnosing the type of anomaly) regardless of their types. In the unsupervised pretraining part, *Proctor* uses the whole training data set 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 testing data set. Then, we repeat the same procedure for each predetermined labeled percentage value.

We train AE-Borghesi by using normal data in the training data set. It is important to note that AE-Borghesi 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 in our experiments. This threshold 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 the false alarm rate. *Proctor* outperforms RF-Tuncer by 50% on average in the anomaly miss rate.

Due to the simple thresholding used in AE-Borghesi, as well as the existence of multiple anomaly types in our data sets, 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. After 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 in anomaly diagnosis analysis is to compare *Proctor*'s classification F1-score with RF-Tuncer for anomaly diagnosis across different percentages of available labeled data. In the unsupervised pretraining part, *Proctor* uses the whole training data without any supervision. In the supervised training part, we train RF-Tuncer and *Proctor* using a percentage of the labeled data and evaluate their performance in a constant testing data set. We repeat the process for each labeled data percentage value.

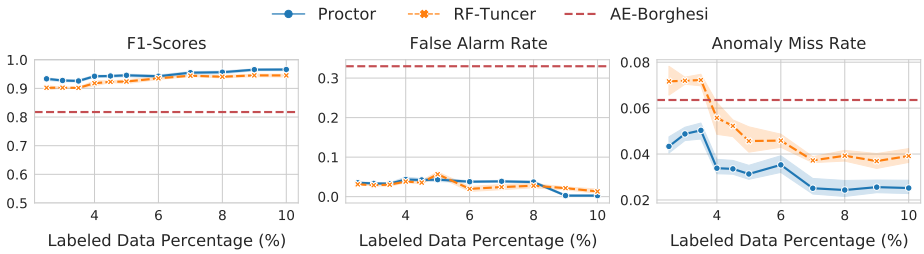

Fig. 3. Comparison of the anomaly detection performance of *Proctor* with AE-Borghesi and RF-Tuncer using the Eclipse data set. *Proctor* performs better than the baselines in F1-score and anomaly miss rate, while maintaining a similar false alarm rate with RF-Tuncer.

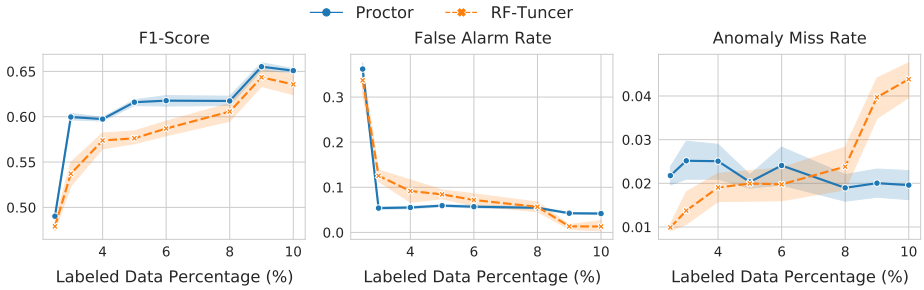

Fig. 4. Comparison of the anomaly diagnosis performance of *Proctor* with RF-Tuncer using the Eclipse data set. Proctor performs better in F1-score and false alarm rate while maintaining a stable anomaly miss rate.

Figure 4 shows the macro average F1-scores for our method and RF-Tuncer for the Eclipse data set. *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 the macro average F1-scores for *Proctor* and RF-Tuncer for the Volta data set. In terms of the F1-score, *Proctor* outperforms RF-Tuncer by 25% on average (and up to 50%) and maintains similar alarm and miss rates to RF-Tuncer. *Proctor* outperforms RF-Tuncer for most of the cases in terms of all categories until we have approximately 20 labeled data samples in total. After this point, the fully supervised RF-Tuncer method has sufficient labeled anomalous data for training to achieve accurate predictions. RF-Tuncer achieves a similar F1-score to *Proctor* faster in the Volta data set compared to the Eclipse data set. The main reason behind this is less complex application characteristics in the Volta data set.

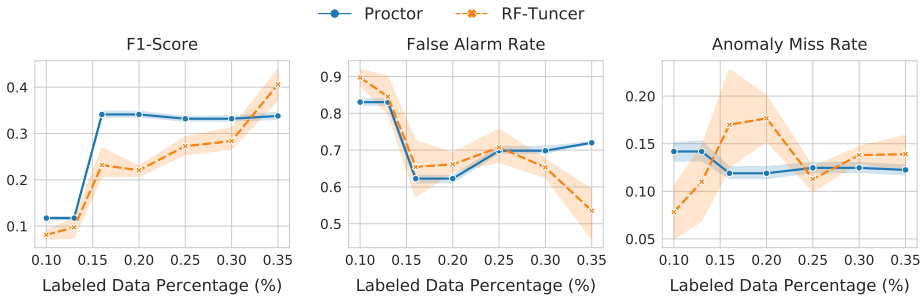

Fig. 5. Comparison of the anomaly diagnosis performance of *Proctor* with RF-Tuncer using the Volta data set. *Proctor* outperforms RF-Tuncer for most of the cases across all categories.

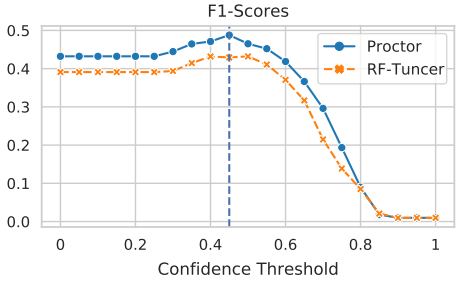

Fig. 6. Choosing a threshold that gives the highest F1-score by sweeping confidence thresholds.

5.5 Impact of Previously Unseen Anomalies

Our primary goal in this scenario is to evaluate the performance of *Proctor* and RF-Tuncer when there are unknown (i.e., previously unseen) anomalies in the testing data set. Since a variety of performance anomalies exists in the production environment, it is common to observe anomalies other than those used during training. We follow the same unsupervised pretraining and supervised training approaches described above, except for one difference: we remove a selected unknown anomaly type from the training set during the supervised training stage and keep the other anomalies. After training, we first test the model on the same training data, this time including the removed anomaly, to determine a confidence threshold. We vary the threshold and choose a threshold value that provides the highest F1-score, and then, evaluate the trained model on a testing data set that consists of all anomalies. We label the sample as *unknown* if the model's highest confidence score for normal and anomalous classes is lower than the selected threshold. RF-Tuncer uses a multiclass RF, and it requires all classes to exist in the training data set; thus, not to put RF-Tuncer at a disadvantage, we apply a *one-versus-rest* strategy to their RF classifier as well.

We experiment on Eclipse data with all labeled data percentages in Sec. 5.2 and report F1-scores, anomaly miss rates, and false alarm rates for selected la-

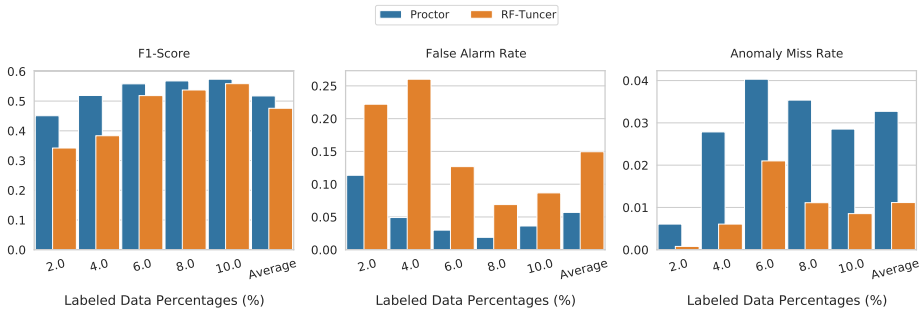

Fig. 7. When there are unknown anomaly types in the testing data set, *Proctor* performs better than RF-Tuncer in terms of F1-score and false alarm rate.

beled data percentages. Figure 6 shows the F1-score across different confidence thresholds. We choose 0.45 as a threshold and compare both methods' anomaly diagnosis performance in Fig.7. Here, *Proctor* outperforms the baseline by 10% on average in terms of the F1-score while maintaining a 66% lower false alarm rate on average. RF-Tuncer's anomaly miss rate is better than Proctor's, however, both rates are very close to zero.

6 Conclusion

Performance variation in HPC systems degrades user satisfaction, 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. However, existing automated methods rely on large labeled data sets for training. This paper proposed *Proctor*, a semi-supervised performance anomaly detection and diagnosis framework for limited labeled data scenarios in production systems. We evaluated our framework using data collected from two different HPC systems, including a production HPC system. We demonstrated that our approach is superior to state-of-the-art approaches in terms of F1-score, anomaly miss rate, and false alarm rate when only a limited set of labeled data is available. We also showed that *Proctor* is robust in presence of previously unseen anomalies and it successfully labeled them as "unknown" in our experiments.

As a next step, 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 performance 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. Agelastos, A., Allan, B., Brandt, J., et al.: 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)
2. Agelastos, A., Allan, B., Brandt, J., et al.: Toward rapid understanding of production HPC applications and systems. In: IEEE International Conference on Cluster Computing. pp. 464–473 (2015)
3. 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)
4. Ahmad, W.A., Bartolini, A., Beneventi, F., et al.: Design of an energy aware petaflops class high performance cluster based on power architecture. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 964–973 (2017)
5. 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)
6. 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)
7. Ates, E., Zhang, Y., Aksar, B., et al.: HPAS: An HPC performance anomaly suite for reproducing performance variations. In: ACM Proceedings of the 48th Intl. Conference on Parallel Processing. p. 110 (Aug 2019)
8. Bailey, D.H., Barszcz, E., Barton, J.T., et al.: The NAS parallel benchmarks summary and preliminary results. In: Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 158–165 (1991)
9. 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)
10. 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). pp. 1038–1043 (2017)
11. Bengio, Y.: Learning deep architectures for AI. Now Publishers Inc (2009)

12. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: SC'13: IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2013)
13. 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)
14. 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**, 634644 (Oct 2019)
15. 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**, 94289433 (Jul 2019), arXiv: 1811.05269
16. Brandt, J., Chen, F., et al.: Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example. In: IEEE Intl. Conf. on Dependable Systems and Networks Workshops (DSN-W). pp. 2–7 (2010)
17. Ciregan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: IEEE conference on computer vision and pattern recognition. pp. 3642–3649 (2012)
18. Dorier, M., Antoniu, G., Ross, R., et al.: Calciom: Mitigating i/o interference in HPC systems through cross-application coordination. In: IEEE 28th International Parallel and Distributed Processing Symposium. pp. 155–164 (2014)
19. Exascale proxy applications, https://proxyapps.exascaleproject.org/
20. Ganglia monitoring system, http://ganglia.info/
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. Habib, S., Morozov, V., Frontiere, N., Finkel, H., Pope, A., Heitmann, K.: Hacc: Extreme scaling and performance across diverse architectures. In: SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–10. IEEE (2013)
23. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories, Tech. Rep. SAND2009-5574 **3** (2009)
24. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural computation **18**(7), 1527–1554 (2006)
25. 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. 310. NIPS'93, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
26. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Computing Surveys (CSUR) **48**(1), 1–35 (2015)
27. Klinkenberg, J., Terboven, C., Lankes, S., Müller, M.S.: Data mining-based analysis of HPC center operations. In: IEEE International Conference on Cluster Computing. pp. 766–773 (2017)
28. Kunang, Y.N., Nurmaini, S., Stiawan, D., Zarkasi, A., Jasmir, F.: Automatic features extraction using autoencoder in intrusion detection system. In: IEEE International Conference on Electrical Engineering and Computer Science (ICECOS). pp. 219–224 (2018)

29. Kunen, A.J., Bailey, T.S., Brown, P.N.: Kripke-a massively parallel transport mini-app. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2015)
30. Leung, V.J., Bender, M.A., Bunde, D.P., Phillips, C.A.: Algorithmic support for commodity-based parallel computing systems. Tech. rep., Sandia National Laboratories (2003)
31. Liu, G., Bao, H., Han, B.: A stacked autoencoder-based deep neural network for achieving gearbox fault diagnosis. Mathematical Problems in Engineering (2018)
32. Luo, T., Nagarajan, S.G.: Distributed anomaly detection using autoencoder neural networks in wsn for iot. In: IEEE Intl. Conference on Communications (ICC). pp. 1–6 (2018)
33. 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
34. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML (2010)
35. Petersson, N., Sjögreen, B.: Sw4 v1.1 [software] (2014). https://doi.org/http://doi.org/10.5281/zenodo.571844
36. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of computational physics **117**(1), 1–19 (1995)
37. 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: Computer-Aided Diagnosis. vol. 10575, p. 105751P. International Society for Optics and Photonics (2018)
38. Schwaller, B., Tucker, N., Tucker, T., Allan, B., Brandt, J.: HPC system data pipeline to enable meaningful insights through analysis-driven visualizations. In: IEEE International Conference on Cluster Computing. p. 433441 (Sep 2020)
39. 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)
40. Song, H., Jiang, Z., et al.: A hybrid semi-supervised anomaly detection model for high-dimensional data. Computational intelligence and neuroscience (2017)
41. 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)
42. 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)
43. 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)
44. 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)
45. 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)
46. 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)