EAI Endorsed Transactions
on Self-Adaptive Systems
Research Article ICST.ORG

Evaluating Unsupervised Fault Detection in Self-Healing Systems Using Stochastic Primitives

Chris Schneider[1,*], Adam Barker[1,†], Simon Dobson[1,‡]

[1]School of Computer Science, University of St Andrews, Fife, Scotland, KY16 9SX, UK

Abstract

Autonomous fault detection represents one approach for reducing operational costs in large-scale computing environments. However, little empirical evidence exists regarding the implementation or comparison of such methodologies, or offers proof that such approaches reduce costs. This paper compares the effectiveness of several types of stochastic primitives using unsupervised learning to heuristically determine the root causes of faults. The results suggest that self-healing systems frameworks leveraging these techniques can reliably and autonomously determine the source of an anomaly within as little as five minutes. This finding lays the foundation for determining the potential these approaches have for reducing operational costs and ultimately concludes with new avenues for exploring anomaly prediction.

Keywords: Self-healing; Systems; Fault; Anomaly; Detection; Machine Learning; Computational Intelligence; Autonomic Computing; Artificial Neural Networks; Self-Organising Maps; Hidden Markov Models; Restricted Boltzmann Machines; Recurrent Neural Networks.

Received on 19 November 2014, accepted on 09 January 2015, published on 28 January 2015

Copyright © 2015 C. Schneider *et al*., licensed to ICST. This is an open access article distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/3.0/), which permits unlimited use, distribution and reproduction in any medium so long as the original work is properly cited.

doi: 10.4108/sas.1.1.e3

1. Introduction

The operational costs of large-scale computing environments are continuing to increase. In order to address this problem, self-managing systems are being developed that reduce the need for human oversight. Self-healing systems are a type of self-managing system that operates by autonomously detecting then recovering from faults. Although there have been numerous advances in this area of study, most self-healing systems continue to require the use of labelled data [8, 9, 18, 22, 28]. This constraint poses challenges for the continued reduction of costs, and inherently emphasises reactive over pro-active approaches [7].

This paper discusses the use of Artificial Neural Network (ANN), Hidden Markov Model (HMM), and Restricted Boltzmann Machine (RBM) approaches that use unsupervised learning algorithms for autonomously generating a *fault hypothesis* [16] indicating the source of a fault within a system – the first step in autonomous fault mitigation. It then summarises the results of each of these experiments before presenting a comparison using extrapolated results. It concludes with lessons learned and a future research section where new avenues of exploration are briefly discussed including further testing related to the results presented here and an untested approach for determining feature locality using shared forecasting.

*chris.schneider@st-andrews.ac.uk, †adam.barker@st-andrews.ac.uk
‡simon.dobson@st-andrews.ac.uk

The importance of reducing human oversight in managing computing environments is multi-faceted. Although numerous direct benefits exist – such as the reduction of staff involvement and their associated operating costs – further achievements can also be realised. Notably, self-healing systems have properties that are showing inherent benefits to both change-control schemas and preserving baseline configurations [20]. The lack of change control or a baseline configuration can both introduce faults and present problems in determining their respective sources.

Self-healing systems methodologies show the capability to heuristically detect and resolve faults without human supervision [7, 23, 33]. This is important when considering costs and time requirements associated with training technical members of staff. If a system can find an appropriate recovery solution without the need for a subject matter expert the associated costs can be immediately recovered.

How well these approaches operate is not clearly understood, nor is a solution agreed upon. The accurate identification of faults, their root causes, and their solutions have posed notable challenges in both Machine Learning, and Computational Intelligence. There is no assurance, for example, that self-healing systems leveraging heuristic, evolutionary, or search-space based algorithms will find an appropriate solution for a given fault – or that the solution

found will be optimal. Furthermore, computational costs of approaches that leverage these methodologies are typically higher than others, and inherently carry a certain amount of risk of failing to identify or incorrectly identifying a potential root cause of an anomaly. There are also considerations that further complicate these matters such as error accumulation when forecasting multiple values in a series.

Anecdotal evidence suggests that in professional computing environments the failure to recognise or mitigate a service impacting fault is never an acceptable state – and that false identification of root causes is to be avoided whenever possible. It is clear, however, that such circumstances do happen under human supervision. Moving to a software based approach poses serious questions regarding accountability in these situations – currently associated with human administrators – and liability. Both of these topics are outside of the scope of this paper, but the preference in supervised management approaches lends evidence to the importance of these criteria [4, 17, 19, 25, 31, 35]. It is possible that greater accuracy in heuristic approaches can alleviate these issues, however this is an area that is still to be explored. The question remains: How can we further the autonomous behaviours of self-healing systems whilst reducing the operating costs of large-scale computing environments?

Previous research has shown that it is possible to autonomously identify anomalies by using stochastic primitives such as HMMs [29], RBMs [30], and ANNs [7, 29]. When utilising unsupervised learning, these approaches can accurately forecast feature behaviour without human intervention. The combination of these two factors allows for a shift from reactive to proactive fault identification – a major advancement in fault detection.

Supervised learning is limited in that it is constrained to reactive approaches. By shifting to unsupervised learning, proactive, prediction-based fault models can be used to manage faults that have not been seen before. Accurately performing this task effectively nullifies the primary advantage of supervised learning techniques.

However, there are challenges in unsupervised approaches. Understanding how far behaviours can be predicted and to what level of accuracy remains an open area of study. In particular, the accumulation of error in multi-step ahead predictions is not easily removed [6]. This is a problem that is also evident when examining relationships between multiple features where predictions and models are almost guaranteed to be different. There is also little practical data available to the public. A small number of performance evaluations have been conducted in controlled (*i.e.* non-production) computing environments but little empirical evidence exists for autonomously identifying faults using unsupervised learning in live systems or under the specific context of self-contained applications, such as front-end services or web-applications.

The rest of this paper is organised as follows: Section 2 contains a summary of the latest unsupervised self-healing methodologies leveraging stochastic primitives. Section 3 describes their implementation details, key components, and variables. Section 4 presents experimental results, whilst Section 5 discusses these results and open problems. Section 6 concludes and highlights topics for future research.

2. Methodologies

Unsupervised anomaly detection within self-healing systems is in a relatively early stage of development. The use of evolutionary programming techniques have been present as far back as 2009 [24]. However, recent studies have emphasised the use of stochastic primitives [7, 28–30]. In order to quickly understand where common effort is being placed, a brief overview of these studies is provided. This includes their implemented primitive types, learning algorithm(s), and software suites – which in turn detail how information is gathered, and what faults are injected. Dividing these areas into distinct units for evaluation establishes the groundwork to describe commonalities in their respective implementations (Section 3) and results (Section 4).

There are two primary approaches described in this paper. The first is generically referred to as A*nomaly Detection Frameworks* (ADFs). They operate by sampling information from a system, storing changes in that information into vectors, and then using those vectors to train various stochastic primitives to predict future feature behaviours: ANNs, HMMs, and RBMs. The second approach is called *Unsupervised Behavioural Learning* (UBL) [7]. UBL operates similarly, but uses a special type of ANN called a self-organising map (SOM) that reduces relationships between feature sets into a two-dimensional lattice. These relationships are used to forecast feature behaviours in a similar fashion to the ADFs. In both cases the expectation of actual behaviours are compared to those that have been forecasted. In both approaches the forecasted behaviours and a separate set of operational criteria are used as a *fault model* to determine when a fault is present [16].

The UBL and ADF implementations both use unlabelled data to forecast anomalies by predicting unexpected changes in feature attributes. Predictions are made by observing a period of known or assumed-good states to train primitives in order to recognise an expected set of behaviours. Once this training is complete, observed data is then classified heuristically into one of several states.

How each of these approaches operates is slightly different from the other. In UBL relationships are represented by neural weights that in turn model systems behaviours. These weights are adjusted and leveraged via learning and mapping phases, respectively. The learning phase executes a series of weight adjustment steps based on Euclidean distance. Each neuron is measured against an input vector independently and the lowest distance value returned is selected as the primary weight to be adjusted. These adjustments are made through a learning coefficient that manages how fast (*i.e.* elastic) the primitive updates its neural weights.

The mapping phase is what one might expect – rather than updating the SOM, an evaluation is run through the already trained primitive to determine differences between current and expected behaviours. In this instance an area value is evaluated between two neurons; larger results indicate a potential anomaly. This is a shared approach between both the ADF and UBL implementations – the exigencies of which are discussed later in Section 5.

Information gathering in UBL primarily occurs via interfaces to Xenstore. This includes information that is pulled directly from associated libraries, system files, and a custom daemon. Once collected the information is used to train the SOM or to determine the root cause of a potential anomaly. Training happens either locally or via a networked virtual machine (VM) instance.

Fault injection in UBL occurs through the use of existing performance-testing software suites including: *CPUHog*, *NetHog*, *CpuLeak*, *Bottleneck*, and *HTTPerf*. Injections happen externally against *RUBiS*, *IBM System S*, and *Hadoop* instances. Notably, the authors of UBL explain the decision to use these suites as a measure to ensure a commonality in accepted platform and software usage.

The second approach has no acronym, but uses the term Anomaly Detection Framework (ADF) to denote the primary service in two separate but related studies for detecting faults [29, 30]. The ADFs collect information and evaluate potential root causes of a fault in a method that is similar to UBL, but with fewer software overlays and with different primitives.

The learning mechanism in these studies varies based on the expected output. For single point predictions the HMM utilises Baum-Welch [1, 2], whilst the ANN leverages Naïve Bayes [21]. Multi-point predictions produced by the RBM use Contrastive Divergence Learning (CDL) [5]. The differences between these approaches are multi-faceted and are superficially discussed in Section 3.3. Details on these algorithms are outside of the scope of this paper, however, they can be summarised as the differences between when learning updates occur within each primitive: immediately after an event has occurred, after all events have occurred, and after a fault has been detected, respectively.

Similar to UBL, all of the ADF approaches compare expected feature behaviours with their actual performance. Specifically, differences between the expected behaviours and their actual behaviours are used to determine the presence of anomalies by forecasting behaviours using previously observed states. As expected, each learning mechanism within the ADF implementation has a different way of achieving this goal.

Single point prediction algorithms use previously observed feature behaviours to determine whether or not the latest set of feature behaviours within a faulty configuration state are expected. This works by autonomously classifying the system's configuration as either good or bad using a series of high-level fitness tests. Known *Good* configurations – those that pass all of their fitness tests – have their features' behavioural data stored in a vector. This vector is then used to train the primitives implemented by the ADFs. If a fitness test fails, the resulting faulty configuration undergoes analysis for unexpected feature behaviours.

Multi-point forecasting in the ADFs operates a little differently. The system's configuration data is still classified in the same manner, but there are two key differences: The evaluation of the root cause of the fault is done lazily, and matrices are used rather than vectors. The use of matrices is a requirement of the primitive and its associated learning algorithm – but the overall comparison strategy remains the same. The RBM categorises vectors holistically within the matrix as either good or bad, and then uses regression analysis to forecast $\kappa + 1$ values, where κ is the dimension of the matrix.

The ADF gathers information via the Windows Management Instrumentation (WMI). Attribute and feature data is extracted and then examined on a per feature basis once per minute. Changes in the features' values are denoted in a binary fashion and then stored as a vector then input into a matrix. This is an important distinction from UBL which samples once per second.

Faults were injected using two different methods: *adverse configuration changes* (ACCs), and *direct fault injections* (DFIs). ACCs included shutting off critical services and were used to emulate human errors within systems – such as forgetting to enable a service after a control change. DFIs included forcibly terminating critical services and denying access to fundamental resources – such as disk partitions and volatile memory.

3. Implementations

The methodologies described in section 2 have varying but similar implementations. Both approaches adhere loosely to the Monitor, Analyse, Predict, and

Execute closed control loop using feature 'Knowledge' (MAPE+K) [15]. However, because there is an emphasis on the use of unsupervised learning, the classification of data before it can be used is paramount. This classification before analysis represents an extra step compared to existing approaches.

The technical differences in how these approaches collect, classify, and analyse information are described in this section. Specifically, the number and types of features being monitored, and how the collected data is used, stored, and evaluated are examined. These properties are discussed before a traditional analysis of the self-healing frameworks' prediction (*i.e.* planning) and recovery strategies – including what learning algorithms have been implemented, and under what assumptions. Visual representations of these processes for the ADFs can be found in previous publications [29, 30].

3.1. Collection

Information collection for all approaches happens either through the use of local daemons or via API interfaces. These interfaces act as controlled, authenticated gateways into a host, and as a mechanism for formatting the returned data. Their implementations differ, however, in both their fidelity and their use.

UBL leverages interfaces to Xenstore 3.0.3 (`libxenstat,libvirt`), data pulled directly from a CentOS /proc file-system, and a custom memory monitoring daemon. In addition to the expected memory performance metrics, results include I/O data for both disk and network devices, CPU usage, and other properties from the role management interface – *i.e.*, Domain 0. This information is collected once per second and is stored ephemerally.

The hardware of the UBL VMs is described as 5 physical servers with 3.0GHz Xeon CPUs with 4GB RAM using CentOS 5.2 with Xen 3.0.3. The guest VMs also run 64-bit CentOS 5.2. Each physical host manages 5 VM instances.

Once the sampled information is collected, it is used to greedily train a SOM either locally or via a separate VM. Information is continuously fed and thus is not limited to a specific time-span or window. This information is either used until an anomaly is predicted by the SOM, or for training until a convergence is encountered – a topic that is discussed further in Section 3.3.

Each ADF is implemented on an identically configured Windows 7 VM running IIS 7.5 that is administered via VMWare workstation 10.0.1. VMs consisted of 2GB of RAM, and access to one Intel i7-4770 CPU (2 cores). Feature data was sampled through WMI using the .NET framework (C#, v.4.5).

The ADFs interface with WMI using the following WMI classes provided at compile time: *BIOS*, *ComputerSystem*, *DiskDrive*, *LogicalDisk*, *NetworkAdapter*, *NetworkAdapterConfiguration*, *OperatingSystem*, *PhysicalMemory*, *Processor*, *QuickFixEngineering*, *Service*, and *SystemAccount*. Each class consists of approximately 7,500 features, each with their own associated attributes. These classes were selected from the more than 300 existing options in order to limit the volume of data collected and to avoid overlaps.

In both ADF instances, the features and attributes within the aforementioned classes are catalogued at a rate of once per minute then stored locally both in volatile and non-volatile memory. Each collection consists of datasets, tables, and tuples parsed into binary vectors or raw configuration data stored as XML, respectively. Storing information in XML files is used for resuming the service when running identical fault tests under variable conditions, such as using fewer configuration samples.

All collected data via the ADFs is subject to an expiration criterion. The maximum number of samples collected by the ADFs is 30. Once this limit is reached the oldest datasets are expired. This is a critical factor in how the ADFs infer behaviours differently from UBL and is discussed further in sections 3.3 and 4.

3.2. Classification

The information gathered by these approaches consists of unlabelled performance metrics and configuration data. However, using this information to decide on the source of an anomaly first requires that this information be accurately classified – a non-trivial problem.

The state of the art for autonomously and accurately classifying unlabelled data is outside of the scope of this paper. However, exigencies in these areas are occasionally addressed in their respective solutions. This includes the strategies implemented by each of the aforementioned approaches, and their respective limitations.

UBL has three classifications for state: Normal, Pre-Failure, and Failure. Each state is determined by calculating the Manhattan distance of a neighbourhood area size using individual neurons. This calculation measures the total geometric distance between two points–typically on a grid. By analysing the differences in neighbourhood area sizes, UBL is able to classify the behaviours of individual features as being in one of three aforementioned categories.

Smaller changes in area size represent a higher likelihood of predicted feature activity. Likewise, progressively larger area thresholds indicate anomalies in performance – the greater the distance the more likely the respective feature is in either a pre-failure or failure state. The distances between these values also

help sort potential problems by order of significance. This is an important step when analysing the potential root cause issues (Sections 3.3, 4).

Expected feature behaviours are determined by using a *bootstrap phase*. Once UBL is instantiated a pre-determined training phrase is executed based on direct observation of the system's features. This stage occurs for as long as it takes to update every neuron in the SOM a total of ten times. The significance of using the value ten is not described in the original work, however the average time to train the SOM is mentioned as ranging between 42 seconds to 7 minutes. The time to fully train a SOM is dependant in these instances on what percentage of the CPU is exclusively dedicated to this purpose.

The ADFs classify information into only two states: *Good* or *Faulty*. However, rather than labelling the behaviours of individual features, the entirety of a system's configuration is given a classification before looking for feature changes. Once a classification is made, the data is then parsed in either a greedy [29] or lazy fashion [30], respectively.

Classification of a system's overall state is provided through the use of fitness tests. These tests supply high-level insight into the health of the system by performing basic tasks associated with a system's role – *e.g.*, the status of associated web services if the role is a web server. Fitness tests include network connectivity, service availability (*IISAdmin/w3svc*), and volatile and non-volatile memory availability tests including free memory and hard-disk accessibility. If the system passes all of its fitness tests its entire configuration is assumed to be good.

If any of the fitness tests fail then information from previously validated (*Good*) configuration samples are converted to binary vectors based on changes in their attributes. Afterwards the vectors are used to train primitives – one per feature – before being used to analyse the the root cause of a fault – further details of this are discussed in Section 3.3.

Like UBL a training interval is required once a fault is suspected to be present. The ADFs are fully trained after initial anomaly detection between 1,000 and 8,000 *ElapsedTicks*[1] in lazy instances. This measurement translates roughly between two and 16 seconds, respectively. In greedy instances training is completed before anomaly detection occurs and results are displayed between 1,000 and 2,000 *ElapsedTicks* – about two to four seconds (Figure 8).

There are a number of differences in how the UBL and ADF approaches classify data – from how much data is utilised, at what point the information is classified, and both how and when data is processed. These differences are associated with the relative uses of each framework although some properties are based on assumptions.

For example, the training phase for UBL was tested *in situ* before being applied. Using a training phase provides an advantage in that it does not require a specific set of fitness tests or roles to be provided before classifying data. However, using fitness tests follows a common tenet in self-managing systems research – the ability to provide high-level policies to systems as a primary form of administration. It also allows for the specification of specific areas of interest – an approach that can reduce false positives.

How long the data is stored and how it is ingested also play roles in classification. Using a windowed approach for information parsing allows for the avoidance of convergence in training data, and greater adaptivity to changing environmental variables. Both of these properties represent advantages in implementation but they come with a cost. Windowing necessitates more memory and post-processing requirements as purely additive measures are no longer sufficient. As such, the expectation is for windowed information to take longer to classify and process.

The way data is ingested impacts when and how classification occurs. WMI provides attributes associated with features in a semi-structured, non-uniquely identified tuple. In order to address removals of devices and multiple features that share a similar name-space this exigency must first be addressed. The ADFs use a dictionary indicating a shared column between all features sampled within a specific WMI class as a unique identifier, such as serial number, ID number, or other immutable property. This information is then stored along with an autonomously synthesised schema. Xen samples metric data from a number of different features. As the values within these samples have multiple ranges, their relative performance and consequent classifications can become difficult. UBL's solution is to normalise this information to unilaterally use the same evaluation techniques across all features. This reduces the fidelity of the content, but lowers the programmatic overhead needed to classify the sampled data.

The classification of data happens at the feature and system levels for UBL and the ADFs, respectively. This distinction impacts other aspects such as frequency of data collection and the number of features and attributes sampled. It also effects how the data is analysed: There is an implied relationship between the number of observations and what predictions, if any, can be made. However, a larger number of observations does not always provide for more accurate results.

[1] *ElapsedTicks: This measurement is used to ensure greater reproducibility. Using measurements based strictly upon timing produces issues including minute differences in machine configurations and operating systems. Further reading on this topic can be found here: http://bit.ly/1n6VOpQ.*

Figure 1. Summary of UBL & ADF Framework Attributes

	UBL	ADFs	
Primitives	ANN (SOM)	ANN, HMM	RBM
Classes	3	2	2
Learning Algorithms	Euclidean, Manhattan	Naïve Bayes, Baum-Welch	CDL
Training Cycles	Conditional: All neurons updated x10	5–30 system samples	
Windowed	No	Yes	Yes
Polling Interval (Sec)	1	60	60
Forecasting Capabilities	Multi-point	Single-point	Both
Ingest Type	Greedy	Greedy	Lazy

3.3. Analysis

In order for a self-healing system to correctly identify the potential cause of an anomaly, it must first determine that a fault is present. As such, anomaly detection in self-healing systems occurs after data has been collected, but typically during either categorisation or, in more reactive implementations, when collected data is being parsed. In either case the categorisation behaviours must occur before an analysis can be made. It is the accuracy of the collective categorisation, parsing, and post-processing behaviours that determine their effectiveness.

To measure the accuracy of these predictions each of the respective experiments used a series of tests where the general source of the fault was known to the administrator but not known to the framework. Each result was then validated to ensure that the correct number of true and false positives or negatives, respectively, had been attributed. Although their goals are similar, the UBL and ADF experiments categorise information differently at a number of levels. This makes a direct comparison of the analysis methodologies challenging but not impossible.

UBL primarily focuses on understanding changes in feature attribute data. These changes are analysed for behavioural deviations from expected norms and then investigated based on severity. If a feature is determined to be operating sufficiently outside of the norm, it is labelled as in either a pre-failure or failure state – what constitutes as sufficient depends on the relative neighbourhood area size.

Using the aforementioned *bootstrap phase*, an expected set of behaviours are established for a collection of features. Each individual feature that is sampled is mapped using a combination of κ-fold cross-validated, randomised initial weights and input vectors which are updated into the SOM. As previously mentioned, this process repeats until each input vector has been updated ten times at which point the SOM is considered trained and ready for use.

Once the SOM is trained it begins periodically sampling the system for further data via Xen's 'Domain 0' interface. The data is then either updated into the SOM or a differential analysis is performed using the Euclidean distance of an input measurement vector against each neuron's weight vectors, respectively.

The weights in the SOM can be updated incrementally or every time a sample is provided. A sample contains information about the system's performance and behaviours as discussed in Section 3.1 every 60 seconds. If an update occurs, each neuron in the SOM has its weights validated via a neighbourhood area size calculation using the summed Manhattan distance of each of its neighbours.

The total Manhattan distance metric is the primary indicator of both fault presence and source. Neurons that have a small area mapping are assumed to be operating normally. Those neurons with larger distance spreads are indicators of either precursors to potential or existing anomalies depending on severity. The threshold for making these determinations are not explicitly given in the original work, but a 50% increase over the example value is given as an indicator of a pre-failure neuron.

The source of an anomaly is determined using the preserved geometric positions of each of the neurons within the SOM. Mapping neurons' expected behaviours with those near to the neurons suspected to be in an anomalous state provide an updated distance measurement from which to evaluate the cause. The inference between the feature and its associated neurons provides an avenue for identification.

UBL's low level analysis provides an agnostic approach to determining the source of a fault. It has advantages in that it does not require roles or additional pre-requisites to be supplied outside of the *bootstrap phase* before becoming operational. However, there are some constraints and implementation details that differ from the ADF instances that are worth mentioning explicitly.

The SOM within UBL can update its weight incrementally but not indefinitely. In the current implementation there is no way to expire old data as it is immediately incorporated within the SOM and then the sample is expired. Since the information cannot be expired the SOM's weights will eventually converge. This eventually creates an inability to perform the differential methods needed for classification and analysis.

The feature data examined by UBL is normalised into a range from 0 to 100. By examining both the minimum and maximum possible values of a feature

before executing the number of neurons required drops substantially. Specifically, the SOM is implemented as a 32x32 matrix containing 1024 neurons. Normalisation of values may produce a drop in the fidelity of information. For example, if the change in the neuron's actual value represents less than 1% of its total possible value, then the change may not occur within the SOM. In some instances the original work states that normalisation values have also been reported to be over 100. These incidents are claimed to be non-impacting, but are not completely understood in terms of implementation or how they might influence the analysis of anomalies.

Using randomised weights ensures more stable results by avoiding bias in knowing the initial weight vectors. In this case, UBL instantiates them individually per neuron to avoid only partial map training when a bad seed value is encountered. Notably, the ADF approaches uses a similar technique by randomising learning weights in hidden neurons.

The ADF approaches operate under similar conditions and principles to UBL. However, instead of focusing exclusively on feature behaviours, a combination of systems validation (*i.e.* fitness) tests are used to provide context to changes in attribute data. Contextual information offers clues as to what shifts in attribute behaviours are expected or unexpected by attributing their values with service-level objectives (SLOs). If all SLOs are successfully being met, then the configuration is given a context of being in the aforementioned *Good* state. In this case, SLOs are incorporated into and represented by fitness tests.

Whilst the system is in a *Good* state information is sampled via WMI then saved to both volatile and non-volatile memory. In the first ADF experiment, this information was greedily learned by the respective primitives for each feature set at the time of ingest – either an ANN or an HMM. In the second experiment this approach was adjusted to a lazy operation in order to optimise memory usage and minimise the impact to the system whilst under operating conditions.

Initially all primitives were retrained upon every sample. Since the samples were stored in memory, a complete vector could be regenerated at each polling interval relatively quickly, if not somewhat expensively. This allowed for windowed training for each detected feature and associated attribute by destroying, re-instantiating, and re-training the primitive. The shift to a lazy approach removes the high computational costs of the previous approach, but increases the delay in returning the ordered list of potential root causes (*i.e.* 'leads'). This is because each primitive (*i.e.* RBM) must be fully trained before the list can be generated – a process that can only be started once one or more fitness tests have failed.

Samples are additively saved until a maximum number of specimens is reached, or a fault is encountered – whichever occurs first. If a maximum number of samples is reached, the oldest sample is expired before the current system's configuration is added to the ADF's collection of data-sets. If a fault is encountered the existing samples are used to generate vectors which then instantiate and train their respective primitives. As previously mentioned, the ADF's maximum sample count is 30 in all instances.

Vectors of length $\kappa - 1$ are produced – where κ is the number of samples that are stored after passing their respective fitness tests. As a comparison must be done to provide the values in each point in the vector, there is always one value missing. Training for either ANN, HMM, or RBM primitives occurs via either Naïve Bayes, Baum-Welch, or CDL, respectively.

Once the primitives are trained they can forecast either single (ANN, HMM) or multiple points of behaviour (RBM). This ability is gained from the predictive reasoning capabilities of the algorithm along with the physical structure of the primitive(s). In the case of RBMs, for example, an undirected graphical model uses an approximated gradient for the log-likelihood of a specific behaviour. This is sampled using a Markov chain which is weighted towards the last observed state.

For reasons of scope and complexity, the functional and operational aspects of the respective learning algorithms are not addressed here in detail. The details of these algorithms are generally agreed to be well documented and readily available. However, CDL remains an evolving and not entirely understood methodology [5, 10, 32].

Each vector that displays a change in its last *Good* sample is marked for forecasting. If the forecasted behaviour from the primitive does not match the actual behaviour of the feature then it is added to the list of leads and given a confidence value. The confidence value represents the inverse likelihood that an event should have been seen using the last κ-1 samples based on the respective learning algorithm. Those behaviours determined least likely to have occurred are placed at the top of the list before being returned.

The ADFs emphasise a high-level approach to determining both the presence and source of an anomaly. They are designed to automate alerting procedures and operate as independent services running in large-scale, centrally managed computing environments. However, requiring a set of role-based fitness tests means that they operate less agnostically than UBL.

Like UBL, the ADFs have some unique constraints, advantages, and disadvantages. In particular, training time, polling intervals, and a higher degree of initial

human oversight are major points of difference in implementation.

A minimum of two *Good* samples must be provided before a failure can be analysed by the ADFs. This is similar to UBL's 10-update pre-requisite, but with the added difference that the polling intervals are done at fixed time differences. This allows for some measure of prediction as to when training will be complete and allows for a greater volume of data collection, but it also presumably requires a much longer training period. This coincides with the aforementioned polling attributes being once per minute for the ADFs, and 60 times per minute with UBL.

Forecasting behaviours is not an exact process. In each instance the abilities of the primitives are constrained by the usual problems but also the additional restrictions of their learning algorithms. This includes accepted and expected error in probabilistic learning, and numerous other factors.

Using fitness tests may have some advantages in specificity, but it requires greater initial human oversight. By looking for general areas where problems may exist it stands to reason that correctly detecting a fault that is associated within such an area is more likely – so long as the tests and analysis logic use the same points of reference. For example, a network connectivity fitness test may help indicate which features are more likely to be the source of the anomaly assuming the context of the test is incorporated into the analysis logic.

Training RBMs is relatively expensive. Compared to the iterative updates of UBL and the first ADF framework, the second ADF approaches are particularly intensive. Each RBM requires 5,000 training cycles (*i.e.* epochs) before being utilised. If there are a large number of differences in attribute states between the last known good and fault configuration samples, it could take several minutes for a potential root cause to be proffered by the application.

There is a certain amount of elasticity in the ADFs ability to forecast feature behaviours. The size of the training sets – both with respect to the number of features being monitored and the total number of observed configurations – influences the processing time, adaptivity, and accuracy metrics. By increasing the frequency of the polling interval, the ADFs maximum window is constrained to a shorter time period. Once the maximum number of samples is reached, old data is discarded. Consequently, the ability to forecast data becomes restricted to a small subset of information. Increasing the maximum number of observed samples used for parsing comes with higher resource constraints, but more stable predictions and less sensitivity to outliers. Expiring old information helps to retain the correct scope from which to draw conclusions and avoids problems in over-training and convergence.

Lastly, different learning algorithms provide different results – which is both why they are interesting and why they are difficult to compare. It is not always clear as to how factors impact each other, or if there are relationships between attributes either in the learning algorithm or the data when a fault is detected. However, some of the results suggest that it should be possible to help determine the existence of such relationships.

4. Results

The UBL and ADF experiments successfully demonstrate the ability to accurately predict and detect anomalies whilst identifying their respective sources. This result comes with varying degrees of success but overall represents a milestone in the autonomous mitigation of errors. By analysing the source of a fault, self-healing systems continue to develop more advanced recovery strategies and techniques and provide one avenue for the reduction of operating costs.

This section presents a summarised version of the original results from each of the aforementioned studies. The original findings are first presented and then compared using synthesised data from each approach. A discussion of these findings is presented afterwards in the following section.

4.1. Summary of Findings

UBL's results can be summarised as being able to accurately predict the onset of performance anomalies within a system using a SOM. The success of this claim is contrasted against two additional unsupervised learning schemes which are outperformed: *Principle Component Analysis* (PCA) and *κ-Nearest Neighbour*; (κ-NN). Findings are reported primarily using graphical representations with the most successful results being further explained textually. Graphics consist primarily of *receiver operating characteristic* (ROC) curves and bar charts that contain averages from "30 to 40" iterations of a specific experiment. Experiments are run using a multitude of stress testing suites (Section 2) and well known infrastructure applications including RUBiS, IBM System S, and Hadoop.

The graphs demonstrate supporting evidence for several advancements in unsupervised anomaly detection. This evidence is divided by suite and application in the original paper [7] but can be broadly summarised in terms of accuracy metrics, achieved lead time, and the effects smoothing has on anomaly identification.

Lead time generated by the forecasting capabilities of the SOM varied between 3 and 50 seconds. How much lead time is generated depends heavily upon what test was being examined, how fast the anomaly manifested, and the level of noise in the respective dataset. Average

lead times were reported along with maximum values in the original works and are summarised as follows:

	Hadoop		System S		RUBiS	
Testing Suite	Avg	Max	Avg	Max	Avg	Max
Bottleneck			5	6		
CpuHog	3	4	3	4		
CpuLeak						40
MemLeak	24	25	47	50	7	50
NetHog					7	7

Figure 2. Lead Times: UBL. This chart represents the number of seconds UBL identified a failure before it reached a terminal threshold; higher values are better. Blank = No Data.

Use of smoothing was shown to be most effective for generating higher true positives in noisy datasets and where anomalies appear quickly. Results showed an increase in true positive rates under these circumstances although they were accompanied by the expected caveat of potentially removing valuable fidelity within the data. Highlights are provided in the following section to illustrate some characteristics of the reported results.

Notably, not every experiment in UBL used the same testing suites or the same types of evaluation criteria. In the case of Hadoop, for example, a 50-point moving average smoothing test is uniquely executed to illustrate improved true positive rates on noisy datasets. However, in some instances using smoothing caused UBL to exhibit low specificity. This was particularly evident in the *MemLeak* and *CPUHog* tests for Hadoop and IBM System S, respectively. Although the former was attributed to noise, the IBM System S false positives are explained due to the rapidity in which CPU spikes appear.

A summary of tests and their results are provided to the reader for ease of reference (Figure 3):

	Hadoop		System S		RUBiS	
Testing Suite	TP	FP	TP	FP	TP	FP
Bottleneck			*G*	*G*		
CpuHog	*G*	*G*	93%	0.5%		
CpuLeak					*G*	*G*
MemLeak	*G*	*G*	98%	1.7%	97%	2.0%
NetHog					87%	4.7%

Figure 3. Summary of Results: UBL. Some results are presented textually, graphically, or not at all. *G* = Data via Graph Only, Blank = No Data, TP = True Positives, FP = False Positives.

The ADF findings came from two separate studies – one using an ANN and an HMM, and another using an RBM. As previously mentioned each primitive has an associated learning algorithm – either -Naïve Bayes, Baum-Welch, and CDL, respectively. The ADF collected information over a period of time between 5 and 30 minutes via WMI about the system before it was then injected with either an ACC or DFI. The faults were injected in the ADFs exactly 30 times.

Results from these studies were contrasted using performance metrics such as *Total Leads*, *Confidence*, *Fault Position*, *Time-Taken*, and *Precision*. These aspects represented the number of paths for investigation, their unsupervised predicted likelihood for being the correct source of the anomaly, where the lead appeared in the descending ordered list of leads, how long it took to complete list generation, and a manual account for the number of correctly and incorrectly identified sources of anomalies, respectively.

Preliminary results positively demonstrated the ability to autonomously and accurately identify the source of a fault based on feature changes. This was achieved by training a stochastic primitive using the frequency in which a specific feature attributed changed within a time-window. It was also possible to predict the likelihood of behavioural changes in properties. The subsequent study expanded upon this approach by improving the accuracy and precision metrics, lowering resource requirements, and setting the groundwork for establishing further anomaly detection studies via stacked RBMs (*i.e.* 'deep-learning') approaches. It also came with the added cost of longer wait times for results, higher variability within those results, and – although the experiment was run using the same amount of data – the acknowledgement that under ideal circumstances the RBM primitive requires larger training periods than the ANN or HMM approaches. Lastly, the ability to leverage multi-step ahead forecasting of feature behaviour – something UBL was able to demonstrate a year prior – is incorporated.

Results in the ADF approaches were presented via line charts that referenced the aforementioned evaluation criteria either by total time the ADF had been running or the number of configuration samples (Figure 4). Additionally, textual information – including source code and results gathered during the experiment – are provided publicly at the end of Section 4. A VM containing the ADFs is available upon request.

Overall UBL's performance is shown to be faster and more precise than the ADF's baseline study with ANNs and HMMs – but this does not appear to be the case when it used RBMs. Similarly, the fault position metric was most consistent with the ANN. The reasons for these results seem likely to do with how the learning algorithm is implemented, and the degree of randomisation in the ANN primitive.

A direct comparison in these respects is somewhat challenging to implement. Whilst the two approaches

are useful for front-end systems and IaaS infrastructures, their methods centre on different types of information analysis and some information is not publicly available. UBL focuses on performance metrics produced and analysed once a second, where as the ADFs examine changes in feature behaviours over a specified period of time – up to 30 minutes. The differences in analysing feature performance versus changes and the frequency in which this data is examined impacts several other assumptions within the respective frameworks.

4.2. Synthesis

Differences in the approaches of both the UBL and the ADFs make direct comparison difficult. To mitigate this issue a common baseline between each approach is synthesised using similar performance metrics and criteria whilst also acknowledging their fundamental distinctions.

There are a number of differences between UBL and the ADF approaches. In addition to sampling frequency, major distinctions include: The type and volume of data being sampled, forecasting capabilities, pro-active versus reactive behaviours, and classification criteria. The resultant data for each of these studies is presented differently.

The majority of results associated with the UBL approach are presented using the aforementioned ROC curves. These graphs are described as displaying the "anomaly prediction accuracy" of UBL via the *true positive rate* (1) and *false positive rate* (2) metrics.

UBL provided two consistent sets of results for each ROC curve – a non-smoothed series and 5 point moving average smoothed series. These datasets were labelled as UBL-NS and UBL-5PtS, respectively. Although other data was provided it was not included in the synthesised results due the period nature in which it was provided. Using this information, and the publicly provided ADF results for fault position, total time taken, precision, confidence, and total leads, the following metrics were generated: *Precision*, *Prediction Time*, and *Fault Position*.

	ANN		HMM		RBM	
	Avg	Max	Avg	Max	Avg	Max
Leads	15	16	18	27	225	767
Confidnc	81.2	91.8	99.9	99.9	97.0	99.8
Position	3.58	9	2.41	5	1.87	19
Time(s)	2.12	2.36	1.19	1.75	12.1	30.9
Precision	24.06	34	21.6	36	86.1	100

Figure 4. Summary of Results: ADFs. A switch between greedy and lazy algorithms manifested as both a large time increase between when a fault was found and an ordered list of *fault hypothesises* (*i.e.* leads) was returned and improved accuracy.

Figure 5. Anomaly Prediction Accuracy Formulas: UBL.

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}} \tag{1}$$

$$A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{2}$$

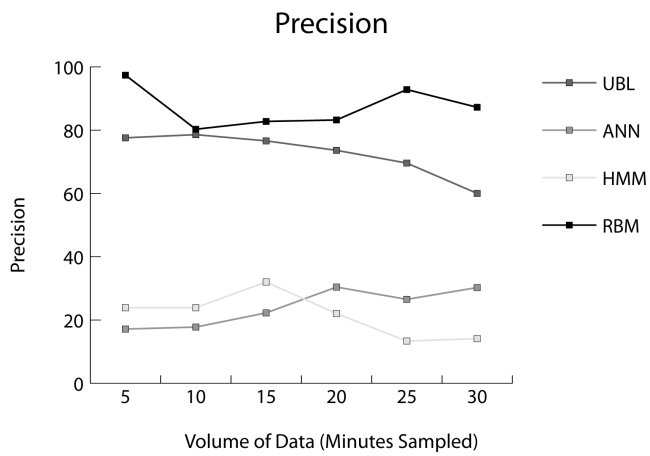

Figure 6. Precision Measurements for UBL & the ADFs. The precision of both approaches drops as more data is added. This may be due to a confirmed convergence problem (UBL) or over-training the primitives (ADFs/UBL).

The precision data was generated by taking *true positive rates* and averaging the values of all relevant experiments at key time intervals (Figure 6). In the case of the ADFs this meant using values for tests that leveraged the same primitives in 5 minute intervals. Each interval represented a sixth of the total results. Similarly, the UBL data used both the NS and 5PtS sets to generate the precision data points at intervals that matched a sixth of the volume of data.

After comparing the results a drop in precision was noted for both UBL and the ADFs over time. The one exception to this was an experiment that leveraged HMMs which continued to improve – a situation that could be explained by possible over-training. As more data is received by the primitives and learned, their sensitivity to new information is reduced. This effectively causes a bias towards more extreme outliers.

A lack of sensitivity appears to be more evident in situations where learned information is not expired. UBL lists a known convergence problem after too many learning updates to the SOM – this effectively limits the

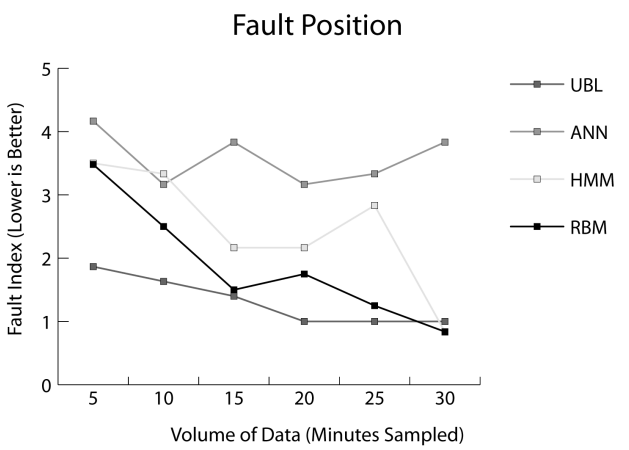

Figure 7. Average Position of Faults Based on Approach. UBL and the ADFs prioritise potential sources of faults. Correct recommendations are represented as an average of all tests based on primitive type. Lower values signify better recommendations.

maximum operating time of this approach. Conversely, the ADF experiments keep a window of information created by user specified values of polling frequency (in milliseconds), and total number of samples. Although both incur degradation, UBL's appears to be much more rapid – however the variability in the RBM data makes it difficult to be certain in all cases.

Using true positive rates demonstrated if the correct solution was discovered, however in many instances multiple potential faults were provided. In UBL neurons 'voted' between possible root causes, whilst the ADFs used confidence values to build an ordered list of leads. Both systems can be seen as basic recommendation engines with solutions being ordered or weighted in some fashion. By understanding where the correct fault was within those engines (either by weight or by position) a demonstrable type of effectiveness is provided for each approach (Figure 7). In this instance, the fault position of the correct root cause is evaluated against the total number of recommendations. The lower the value, the sooner the engine selects the correct root cause. ANN is the most consistent technique, but is equalled by both RBM and HMM given sufficient data volume.

However, accuracy in predictions are often directly related to resource availability. Therefore, the balance between how fast an application returns a result and its level of accuracy is paramount. Fault position was contrasted with resource utilisation by examining the total amount of time a framework took to indicate the source of a fault – collectively referred to as *Prediction Time*. *Prediction Time* was based on the total number of milliseconds from when a fault was first suspected and when the results were fully produced by the primitive.

In instances where greedy algorithms were used the amount of time it took to predict a fault was fairly static. This was an expected result as the systems in question processed the same amount of data in the same fashion at regular intervals. However, the ADF's lazy implementation of RBMs showed a varying amount of time to process information (Figure 8). UBL reported a static 490ms requirement per minute of data gathered to update the SOM before generating a prediction. Using this value the total amount of time per sample was plotted out in minutes. The ADF's initially followed the same pattern as UBL – although due to the complexity differences of their respective learning algorithms they executed much faster. CDL required nearly the same amount of time as the SOM required to update its neurons.

Implementation played a role in the evaluation of time-based metrics. The two primitives used in the first ADF experiment leveraged a greedy implementation using a windowed collection of datasets. This meant that once a minute all primitives would be discarded and retrained – an action requiring 50% of the total CPU activity on the VM for about 15 seconds. By processing this data upfront the system was able to return results relatively quickly – between 500ms and 4,500ms. However, the impact to the system's performance was clearly a disadvantage. UBL's linear training and fast prediction times were impressive. They illustrated an effective approach for determining errant feature behaviours within 2,450ms and 14,700ms (Figures 2,8) ± 2.5ms. As UBL's primary goal was to pro-actively predict anomalies this time was particularly important. Faults that were identified quickly enough could ideally be addressed before fully manifesting. This was a fundamental difference from the ADF approaches which emphasised reactive behaviour by updating future iterations of VMs.

RBMs in the ADF approach compared similarly to UBL time-wise when predicting the root cause of a fault. A substantial increase in time was noted between the two ADF experiments – the latter was based on a lazy implementation for data ingest. This accounted for both the increase and difference between the processing times for the same volume of data in the other ADF experiments. Times ranged from 1,217ms to 14,198ms based on the number of samples provided to the ADF. However, as the data is windowed, prediction times were expected to stop increasing after the maximum number (*i.e.* 30) samples was reached.

Prediction Time shares a relationship with training time. If the majority of training comes before the prediction takes place, then the prediction time is reduced. Notably, training time for the primitives varies based on a number of characteristics including

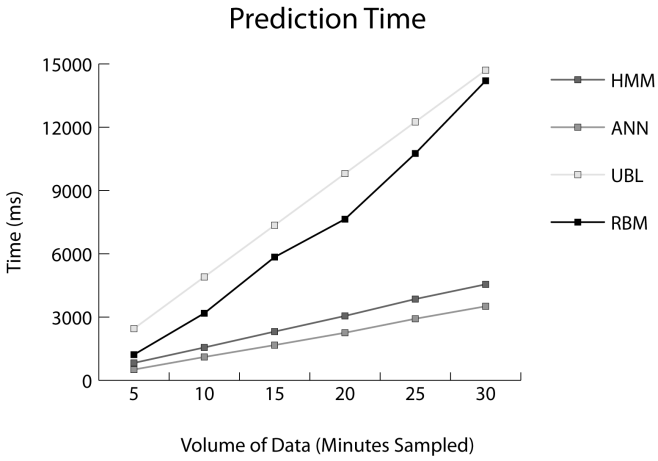

Figure 8. Total Training Time Based on Volume. The training time for each approach is impacted by how much data is present and when data is processed. Faster training times are particularly critical in ensuring lead times for proactive anomaly mitigation.

required epochs, neurons per primitive, volume of data, type of data, and of course which learning algorithm was being utilised. Of the two instances where lazy algorithms were implemented training times appeared to be similar.

Neuron count totaled 1,024 and ~7,200 in the SOM and various ADF approaches, respectively. Primitives incorporating these neurons required up to 6,000 training epochs for UBL, and 5,000 per primitive when RBMs were in place. In other ADF instances training periods used as few as 5 epochs per primitive. Depending upon which learning algorithm was in place, the time for completing training in a primitive varied. Clearly different learning algorithms had different rates of success, but their overall effectiveness was also bounded by resource constraints. Given a greater number of resources – such as time, memory, and clock cycles – the accuracy of the predictions increased, but only to a point.

The data sampled from UBL is approximated. The ROC curves were sampled at high fidelity to approximate their original metrics (*e.g.* precision). This allowed for the comparison of UBL's performance values with those from the ADF experiments. In some instances values overlapped or are otherwise illegible – on those occasions an educated guess is made. Lastly, the number of samples in the ROC curves is not explicitly listed. However, based on the number of fault injections reported in the original work, each graph is assumed to span at least 30 samples. All associated data is made public[2].

5. Discussion

By analysing changes in performance metrics and feature attributes it is possible in many cases to accurately determine the presence and cause of a fault using stochastic primitives. The results from these studies show a shift from reactive to predictive measures, and that specific attributes can be correctly associated with a fault using abnormal variations in either performance metrics or frequency in feature changes. However, there is room for improvement in these approaches – particularly in noisy datasets, feature locality, and distributed learning.

5.1. Observations

Some of the discovered results are surprising. Due to the volume of data, the ADF approaches were expected to take longer to find a solution than UBL. Instead, the time values are similar but the accuracy values are not. Two things are gained from this: Firstly, it is clear that resource utilisation does impact a framework's ability to accurately generate a *fault hypothesis*. Preliminary data shows that by increasing the number of training epochs it is possible to achieve better results. However, some algorithms appear to be more efficient than others. Secondly, by comparing the precision (Figure 6) and fault position (Figure 7) metrics, it is evident that the RBM approach demonstrates far fewer false positives than UBL. This is beneficial in environments seeking to reduce type I errors, but it might be optimised by using fewer neurons. However, other types of primitives may yield stronger results – a topic discussed later in this section.

Training periods are necessary for both approaches before operating. Whilst an improvement over prior research in their ability to use unlabelled data, the existence of these requirements represent a fundamental problem: How to balance instantiating a framework that can accurately detect faults and reducing the initial training period. Several problems have emerged in trying to balance these two factors. However, there may be a solution to this problem using an evolutionary approach.

SLOs and fitness tests can provide measures of a system's health, but more importantly they offer a way to administer a system from a higher administrative level. This is one of the primary goals of self-managing and self-adaptive systems research [12, 14]. In each of the aforementioned studies, progress in this area was

[2] *Results sampled from UBL, along with the latest ADF source code and a subset of results, can be acquired here: http://bit.ly/1oGBX67.*

apparent. However, not all SLOs are created equal and during normal use variance needs to be accounted for.

In each case the use of presumed or verified 'known-good' datasets were used to help classify sampled information. The ADFs' use of fitness tests to determine the general health for the system allowed for faster training, but did not take into account individual feature changes until after a fault was detected. This could make feature locality more difficult to determine in its current form. Similarly, UBL used a vetted series of inputs to resolve a number of factors associated with SOM instantiation and training. The use of a static training mechanism is, arguably, a potential source of problems for dynamic fault detection, *a priori*.

The use of a layered approach towards classification is one of the primary differences between the aforementioned studies. The UBL study ignored so-called 'constant' metrics – those values that changed very infrequently – in favour of minimising resource usage whilst focusing on metric changes simultaneously. However, the ADF approaches did exactly the opposite; they reinforced non-changing attributes as nominal behaviours and used this information to update confidence values when changes occurred and only when necessary.

Not having to write independent policies is one advantage that UBL has over other approaches: No policies need to be written explicitly outlining the purpose of the machine. Normalising all values and providing a static minimum and maximum allows for autonomous evaluation of the system regardless of role. This means faster provisioning but perhaps less targeted behavioural adjustments after instantiation.

UBL tested centralised versus localised training benefits. UBL focused on identifying and resolving an anomaly within a specific window or 'lead time'. The time to transfer data to and from a training VM and implement a solution often exceeded the lead time that was generated by the SOM.

5.2. Lessons

Ultimately the UBL study deemed the centralised training approach too slow for practical use, however it may be the way forward depending on environmental conditions. Rather than requiring greater lead time, the solution may be to offer a temporary fix whilst augmenting future VMs against similar faults. Evolutionary methodologies have already been shown capable of synthesising new, valid systems configurations under similar circumstances [23]. The next step may be to use directed training techniques to augment the accuracy of these approaches and to reduce their respective resource costs.

Fidelity is an issue that appeared several times in each study. Greater fidelity often came with longer collection times, higher noise ratios, and longer training periods. However, it also provided more opportunities for inference. Finding a balance between how much information is necessary is an open problem – both contextually and otherwise.

It is clear that depth of fidelity is a property that changes based on what problem is being assessed – although the layered approach of general classification of systems' health metrics then analysis provided positive initial results. UBLs approach to handling noise is to use smoothing and normalisation techniques. In several instances smoothing produces a notable increase in false positives, but it also correctly identifies faults in some of the noisier datasets.

Using stored data provides a way to resume analysis in the ADFs, but this feature is notably absent in UBL. *Drift* – the gradual change in a system's configuration and state from one time interval to the next – makes direct comparisons difficult when resuming from a previously saved state. However, saving information rather than using it directly provides some key advantages including the live cloning of VMs to do multi-point analysis, and the use of multiple-concurrent training sessions under very similar (if not identical) assumptions.

UBL's predictive approach provides unique advantages via the potentially immediate mitigation of anomalies. The use of an additional pre-failure state provides a window for mitigation that is absent in the ADF experiments. It would be interesting to see if the aforementioned lead times generated by UBL are faster than current VM replacement techniques in IaaS, PaaS, and SaaS infrastructures. If not, then replacement may be a better option. Likewise, the use of simple success or failure states is too binary to match the gradual emergence of some faults. The ADF experiments could leverage gradients in SLO or fitness violations to determine stronger confidence values when generating leads.

It is suspected that UBL's results could be improved by using a windowed data-set. Despite its otherwise positive results, the over-training of the SOM appeared to critically inhibit anomaly detection at large. Adding this feature may not only improve their results, but would presumably allow for greater resiliency to changing operational circumstances such as increases and decreases in service usage and associated resource availability.

6. Conclusion

The UBL and ADF experiments demonstrate that it is possible to accurately identify both the presence and potential cause of a fault in a short amount of time using unlabelled data. This comes directly from a system's observed behaviours under normal load within virtualised computing environments, using stochastic primitives such as ANNs, HMMs, and RBMs. However,

accuracy and balancing resource utilisation in these approaches remain core challenges, as well as the lack of human-subject studies demonstrating the potential for reducing for costs.

7. Future Work

There are a number of different areas in which autonomous fault prediction could benefit from future research. Continuing to further understand the differences behind the aforementioned approaches and why results have varied remains an obvious first avenue. Although a baseline has been established and some expectations on performance have been observed, the variables involved and their relationships could be further investigated. Doing so would ideally allow for more concrete conclusions as to the exact reasons why *Precision* metrics differ between UBL and the ADFs, and why ANNs show more consistent *Fault Positions*.

Additionally, new avenues for fault detection may prove useful to the field – including the combination of fault localisation and evolutionary techniques, understanding multi-step ahead prediction using multiple points of inference, fidelity studies – how much is enough data – , public studies involving human subjects and error detection, and efficiencies in learning algorithms used in stochastic primitives.

With regard to the latter, the advancement of autonomous fault prediction would benefit from more efficient learning algorithms, a comparison of a greater number of stochastic primitives, and integration with evolutionary programming techniques. By combining these areas of study local minima in search-space based approaches may be reduced, and optimisation in implementation could be further achieved.

How anomalies are inferred in these approaches does not currently take into account feature locality. Each methodology has the ability to forecast feature or performance metrics using observations from known good states. However, linking relationships between multiple features remains unexplored. By understanding where relationships exist between features, the solutions provided by self-healing frameworks could avoid problems with deterministic solutions. Additionally, combining multi-step ahead prediction and feature locality studies could yield stronger recovery strategies.

Although no experiments have yet been performed, the ability to retroactively forecast attribute behaviours via the RBM already exists. It operates by synthesising vectors given an input for a specific feature once learning has been completed. These vectors are of κ length, with each point representing a specific time-interval. Instances where predicted changes diverge may be indicators of problems, particularly if paired regularly with other synthesised vectors at the same time-interval.

In addition to evolutionary techniques, layering RBMs could improve the accuracy of fault cause identification. Layered (*i.e.* stacked) RBMs provide a vetted system for using probabilistic models to infer relationships between features in a variety of fields [11, 13, 26, 27]. RBMs also have an impressive ability to provide contextual inference in noisy datasets, however an alternative is to use *Generative Stochastic Networks* (GSNs). This is a new type of stochastic primitive that uses advances in the backward propagation of errors [3].

Efficiency in learning algorithms, particularly for fully recurrent neural networks, remains a critical stumbling block and is one aspect that could directly and positively impact future studies.

Lastly, there are no public studies comparing self-healing software and human subjects. Without this information, determining if a proposed solution is able to reduce costs and if the solution meets the minimum criteria for accuracy and speed does not seem attainable. A study in this area would be immensely useful to the field.

Acknowledgements. This research was partially supported by the Scottish Informatics and Computer Science Alliance (SICSA). The authors would like to thank Saleem Bhatti, Ildikó Pete, and César Souza [34] for their insights and technical suggestions.

References

[1] Baum, L. and Petrie, T. (1966) Statistical inference for probabilistic functions of finite state markov chains. *The Annals of Mathematical Statistics* **37**(6): 1554–63.

[2] Baum, L. and Petrie, T. (1967) An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology. *Bulletin of the American Mathematical Society* **73**(3): 360–3.

[3] Bengio, Y., Thibodeau-Laufer, E. and Yosinski, J. (2014) Deep generative stochastic networks trainable by backprop. In *Proceedings of the Thirty-one International Conference on Machine Learning (ICML'14)* (Springer).

[4] Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F. and Mirandola, R. (2011) Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering* **PP**(99): 1–23.

[5] Carreira-Perpinan, M. and Hinton, G. (2002) On contrastive divergence learning. Department of Computer Science, University of Toronto.

[6] Cheng, H., Tan, P.N., Gao, J. and Scripps, J. (2006) Multistep-ahead time series prediction. In *Advances in Knowledge Discovery and Data Mining* (New York, NY, USA: Springer), 765–774.

[7] Dean, D.J., Nguyen, H. and Gu, X. (2012) Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings*

of the 9th international conference on Autonomic computing, ICAC '12 (New York, NY, USA: ACM): 181–190. doi:10.1145 2371536.2371571.

[8] Dobson, S., Sterritt, R., Nixon, P. and Hinchey, M. (2010) Fulfilling the vision of autonomic computing. *IEEE Computer* **43**(1): 35–41.

[9] Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P. *et al.* (2006) A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems* **1**: 223–259. doi:10.1145/1186778.1186782.

[10] Fellows, I.E. (2014) Why (and when and how) contrastive divergence, ArXiv.org. Http://arxiv.org/pdf/1405.0602v1.pdf.

[11] Ghoshal, A., Swietojanski, P. and Renals, S. (2013) Multilingual training of deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (IEEE): 7319–7323.

[12] Horn, P. (2001) Autonomic computing: IBM's perspective on the state of information technology. .

[13] Humphrey, E.J., Bello, J.P. and LeCun, Y. (2012) Moving beyond feature design: Deep architectures and automatic feature learning in music informatics. In Gouyon, F., Herrera, P., Martins, L.G. and Müller, M. [eds.] *ISMIR* (FEUP Edições): 403–408. ISBN: 978-972-752-144-9.

[14] Kephart, J.O. (2011) Autonomic computing: The first decade. In *International Conference on Autonomic Computing* (Karlsruhe, Germany: ACM SIGARCH/USENIX): 1–56. New York, NY.

[15] Kephart, J.O. and Chess, D.M. (2003) The vision of autonomic computing. *Computer* **36, Issue: 1**: 41–50.

[16] Koopman, P. (2003) Elements of the self-healing system problem space.

[17] Li, G., Liao, L., Song, D., Wang, J., Sun, F. and Liang, G. (2013) A self-healing framework for qos-aware web service composition via case-based reasoning. In *Web Technologies and Applications* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **7808**, 654–661.

[18] McCann, J. and Huebscher, M. (2004) Evaluation issues in autonomic computing. In *Grid and Cooperatve Computing - GCC 2004 Workshops* (Springer Berlin), **3252**, 597–608.

[19] Menasce, D., Gomaa, H., Malek, S. and Sousa, J. (2011) Sassy: A framework for self-architecting service-oriented systems. *Software, IEEE* **28**(6): 78–85. doi:10.1109/MS.2011.22.

[20] Miorandi, D., Lowe, D. and Yamamoto, L. (2010) Embryonic models for self–healing distributed services. In *Bioinspired Models of Network, Information, and Computing Systems* (Springer Berlin Heidelberg), *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* **39**, 152–166.

[21] Pearl, J. (1988) *Probabilistic reasoning in intelligent systems: networks of plausible inference* (Morgan Kaufmann).

[22] Psaier, H. and Dustdar, S. (2010) A survey on self-healing systems: approaches and systems. *Computing* **91, Issue: 1**: 43–73.

[23] Ramirez, A.J., Knoester, D.B., Cheng, B.H. and Mckinley, P.K. () Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems. *Cluster Computing* **14**(3): 229–244.

[24] Ramirez, A.J., Knoester, D.B., Cheng, B.H. and McKinley, P.K. (2009) Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09 (New York, NY, USA: ACM): 97–106.

[25] Rilling, L. (2006) Vigne: Towards a self-healing grid operating system. In *Euro-Par 2006 Parallel Processing* (Springer Berlin / Heidelberg), *Lecture Notes in Computer Science* **4128**, 437–447.

[26] Sánchez-Gutiérrez, M.E., Albornoz, E.M., Martinez-Licona, F., Rufiner, H.L. and Goddard, J. (2014) Deep learning for emotional speech recognition. In *Pattern Recognition* (Springer), 311–320.

[27] Schmidhuber, J. (2014) Deep learning in neural networks: An overview. *CoRR* **abs/1404.7828**. Http://arxiv.org/abs/1404.7828.

[28] Schneider, C., Barker, A. and Dobson, S. (2013) A survey of self-healing systems frameworks. In *Software Practice and Experience* (Wiley).

[29] Schneider, C., Barker, A. and Dobson, S. (2014) Autonomous fault detection in self-healing systems: Comparing hidden markov models and artificial neural networks. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14 (New York, NY, USA: ACM): 24:24–24:31.

[30] Schneider, C., Barker, A. and Dobson, S. (2014) Autonomous fault detection in self-healing systems using restricted boltzmann machines. In *11th IEEE International Conference and Workshops on the Engineering of Autonomic Autonomous Systems*, IEEE Computer Society (Laurel, Maryland: IEEE). Submitted 15 May 2014, Accepted 12 August 2014.

[31] Schuler, C., Weber, R., Schuldt, H. and j. Schek, H. (2004) Scalable peer-to-peer process management - the osiris approach. In *In: Proceedings of the 2 nd International Conference on Web Services (ICWS'2004)* (San Diego, CA: IEEE Computer Society): 26–34. Washington DC, USA.

[32] Schulz, H., Müller, A. and Behnke, S. (2010) Investigating convergence of restricted boltzmann machine learning. In *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning*.

[33] Shehory, O. (2007) *A Self-healing Approach to Designing and Deploying Complex, Distributed and Concurrent Software Systems* (Springer-Verlag), *Lecture Notes in Computer Science* **4411**, 3–13.

[34] Souza, C.R. (2013) Accord.net framework. Http://accord-framework.net/.

[35] Stojnic, N. and Schuldt, H. (2012) Osiris-sr: A safety ring for self-healing distributed composite service execution. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on* (Zürich, Switzerland: ACM): 21–26. doi:10.1109/SEAMS.2012.6224387. New York, NY.