

Machine Learning for the System Administrator

Marc Chiarini and Alva Couch, Tufts University
{marc.chiarini,alva.couch}@tufts.edu

Abstract

In this paper, we explore how some of the principal ideas in machine learning can be applied to the specific task of monitoring computer systems. We show that machine learning can be a valuable tool for accomplishing everyday administrative tasks, and a worthy addition to the arsenal of tools utilized by the system administrator. Difficulties in utilizing machine learning do not arise from the science or mathematics of the discipline, but rather, from the need to adopt a new world-view and practice in order to use the ideas effectively. We show some examples of the kind of thought and practice necessary to apply machine learning, and discuss its strengths, its weaknesses, and its future potential in the context of system administration.

[Note to the Program Committee: This work was inspired by a final paper written by John Orthoefer (Akamai) for a class in machine learning, about the potential for use of machine learning in monitoring. John was planning to be a co-author of this paper, but a recent change in job has left him too busy and his time commitment too uncertain to commit to working on the paper at this time. He says he will be in a better position to perhaps contribute in a month or two. Because involving him in the draft-writing process at this time is impractical, we are submitting this paper ourselves, but if the paper is accepted and John gets more time to work upon it in the summer, we will gladly add him as co-author of this paper.]

1 Monitoring

The term “machine learning” suggests that we are making machines “think for us” but – in fact – the field is concerned with helping us *use* machines better by automatically distinguishing between “good” and “bad” machine behavior. An example of this is to use machine learning for monitoring. When monitoring, we usually have a reasonable idea of what defines “good behavior”, while “bad behavior” is anything else. This corresponds to a “one-class machine learning problem” in which the learning algorithm has to tell us whether a specific condition is “good” or “not good”, i.e. bad.

There are many reasons to use machine learning for the purpose of monitoring. In what follows, we will explain some of these, including:

1. There is potential for improving the accuracy of monitoring software, providing fewer false and missed alarms due to higher specificity and sensitivity in the decision-making process.
2. One can achieve this higher accuracy by studying data in ways that are difficult for humans but easy for machines, e.g., by considering couplings between measurements that would be difficult to specify by hand, either because they are too complex or because they are hidden from the naked eye.
3. The resulting practices can become easier than manually configuring alarm rules for changing conditions.

2 Defining normalcy

The purpose of the classification algorithms we will consider in this paper is to determine whether a process, system, network, or other entity is behaving “normally”. We begin our discussion of machine learning with a survey of the ways in which “normal” behavior is defined.

By far, the most common definition of normal behavior for an entity is that measurements of its behavior fall within a range of acceptable values. In many monitoring tools, one configures the system to raise alarms for abnormal events, based upon measurements that exceed preconfigured absolute values or percentages. Many sophisticated mechanisms exist for reducing the frequency of alarms and avoiding false alarms, including measuring the frequency with which set-points are exceeded, etc. In more recent work, normalcy is defined via the use of statistics[7], principal component analysis[1], and time series analysis[2, 6, 13]. In CEnvD[8], the experimental monitoring environment of CFengine[3, 4, 5, 9], normalcy is defined via percentage thresholds relative to average observed behavior over time. These are the first steps toward the idea formed in this paper, which defines normalcy even more generally, based upon the shape of multivariate data when plotted in space.

[To the Program Committee: clearly, there is a lot more to say here about other monitoring strategies, but for the purpose of the draft, we wanted to concentrate on the machine learning.]

3 Data shape

Looking over data to point out what is “good” or “bad” gives us an intuitive idea of what machine learning can do for us. When looking at statistics for any group of entities, e.g. computer servers, we usually define good and bad in terms of thresholds, above or below which parameters go out of range. But groups of parameters also exhibit *shapes* that indicate “good” or “bad” behavior.

There are three kinds of shapes that indicate system health or illness: covariance, contravariance, and independence. For example, “network packets in”

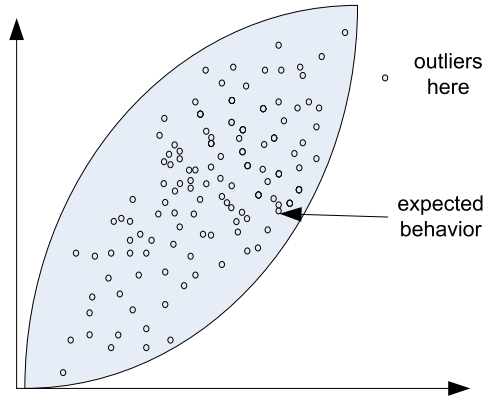


Figure 1: Covariance of two measurements frequently looks like an increasing diagonal of measurements in the “good” set.

and “network packets out” are usually *covariant* parameters, in the sense that input and output sizes are related and increase and decrease together. On the other hand, for a process, system time and user time are *contravariant* parameters, in the sense that when one is large during a given time interval, the other should be small.

Covariance occurs when two or more parameters increase and decrease together. When covariance is expected, a lack of covariance may be considered “bad”. For example, if, suddenly, “network packets in” and “network packets out” start exhibiting big differences, rather than similarities, this indicates that a service has gone awry, even though we do not know *which* service. Covariance usually looks like an increasing diagonal line between parameters (Figure 1).

Contravariance instead occurs when two parameters vary inversely. It is usual, e.g., for the total I/O that a process can do to be constrained so that the sum of all I/O is less than some limit. If disk I/O goes up, network I/O must go down, for some interval, so those are contravariant. Contravariant relationships between two parameters frequently appear as “saddles” or “hyperbolas” (Figure 2). If one parameter is large, the other is small.

A third parameter relationship is “independence”. If two parameter values are independent, then their values form a roughly circular “cloud” in two dimensions. Independent parameters can be slightly covariant or contravariant, depending upon the shape of their clouds. A cloud elongated along the increasing diagonal indicates a slight tendency toward covariance, while a cloud elongated along the decreasing diagonal indicates a slight tendency toward contravariance.

The key to each of these situations is that the values that two parameters can take are not the whole plane, but some fraction thereof. Covariant pairs occupy the increasing diagonal, and there is something wrong if a pair of measurements is far enough off of that diagonal. Likewise, contravariant pairs occupy the space

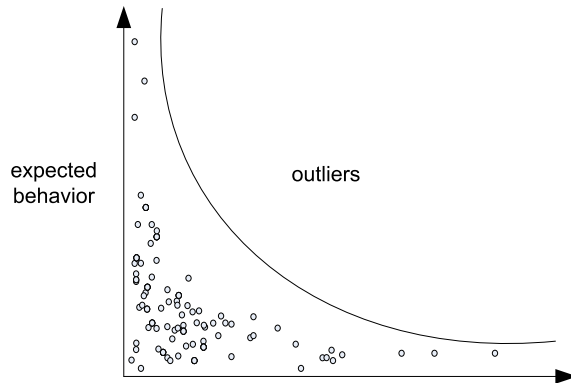


Figure 2: Contravariance frequently looks like a “saddle” of measurements in the “good” set, with all above the saddle disallowed.

near the axes, and are not normally found near the diagonal. If we can detect when covariant pairs don’t covary, or when contravariant parameters seem to be linked or independent, we will be able to detect many more anomalous conditions than we can detect with thresholds.

4 Soft dependencies

Much work has already been done on dependency analysis in complex systems. Static analysis[14, 12] allows one to detect conditions in configuration that affect the results of changes, while dynamic analysis[11] describes what happens at runtime. Both of these are “hard dependencies”, in the sense that some feature of a system determines exactly what the behavior of another feature will be. But there are other kinds of dependencies in a complex system.

Many dependencies in a complex system are “soft”, in the sense that the dependency is statistical rather than causal. A covariance or contravariance does not indicate that parameters are linked, but rather that there is a *tendency* for the parameters to vary either in concert or in opposition, and that this is not always an exact relationship. For example, load averages over one and five minutes are covariant, even though they are almost never equal.

Covariance and contravariance can be statistically tested, for populations of things. If we have a population of pairs $\langle X, Y \rangle$, then the covariance of the pairs is $\text{Cov}(X, Y) = E((X - E(X))(Y - E(Y)))$ where $E(A)$ is the expected value of A . Usually $E(A)$ is the mean \bar{A} of values in A , so that the covariance can be rewritten as $\text{Cov}(X, Y) = E((X - \bar{X})(Y - \bar{Y}))$. $\text{Cov}(X, Y)$ is 1 if $\langle X, Y \rangle$ are covariant, -1 if contravariant, and 0 if they are neither covariant or contravariant. Independence is stronger than lack of information; while two independent

parameters do indeed have a 0 covariance, this is not sufficient to *guarantee* independence.

The problem with statistical definitions of covariance is that, while one can check a *population* of points for covariance or contravariance, one cannot check a *single point* for whether it exhibits the appropriate character *of a population*. So, while we might know that parameters exhibit coupling, this does not help us determine whether a particular pair $\langle X, Y \rangle$ is reasonable.

5 Exploiting topology

One key to utilizing covariance and contravariance as a monitoring tool is to exploit the “shape” of existing measurements. We could simply draw in the curves that delimit proper values and check for points outside those curves (as in Figures 1 and 2). This would have the benefit that we could check for more erroneous conditions than when using thresholds, but has the problem that the work of drawing in the curve must be redone every time the system changes. What is “normal behavior” for a system *depends upon its configuration*. Redrawing the curves every time the system changes is much too much work.

However, using shapes to determine outliers is desirable, even if it is too much work for a human. One advantage of utilizing shape rather than threshold is improved *specificity*. Specificity is an attribute of classification algorithms for determining whether an entity belongs to a specific set or not. In this case, if a pair is in the set of “normal measurements”, then we do not raise an alarm; else we do. There are four possibilities for each decision: the algorithm can decide that the item is in the set or not in the set, while the item can *actually* be in the set or not. Thus there are four properties we can measure:

1. The number of true positives P_T of the items judged to be inside the set that are actually in the set.
2. The number of false positives P_F of the items judged to be inside the set that are not in the set.
3. The number of true negatives N_T of the items judged to be outside the set that are actually not in the set.
4. The number of false negatives N_F of the items judged to be outside the set that are actually in the set.

For the person doing monitoring, these translate as follows:

1. P_T is a count of circumstances in which no alarm was raised and nothing was wrong.
2. P_F is a count of circumstances in which no alarm was raised when something went wrong.
3. N_T is a count of circumstances in which an alarm was raised and a real problem existed.

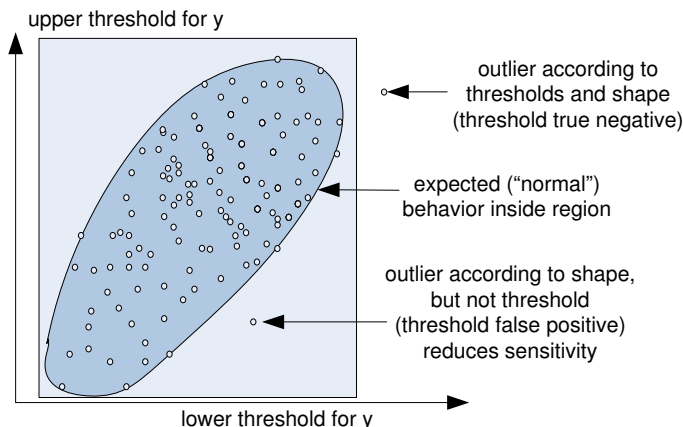


Figure 3: Thresholds that completely contain the desirable data set allow measurements that violate covariance.

4. N_F is a count of circumstances in which an alarm was raised when nothing was wrong.

N_F is the count of times one was bothered without cause, while P_F is the count of times that something serious went unnoticed.

One can define the *specificity* of a monitoring approach as $N_T/(N_T + P_F)$, a percentage of the time that an alarm is raised, given that it is required. 100% specificity means that every time we needed an alarm, we got one. 50% specificity means that roughly half of the time, a needed alarm was not raised when needed.

A related parameter is the *sensitivity* of a monitoring approach, which is instead the ratio of $P_T/(N_F + P_T)$. A sensitivity of 100% means that we detect all the positives correctly, while a sensitivity of 50% means that roughly half of the time, we misidentify a negative as a positive.

If s is a hypothetical sample from a set S :

	actually $s \in S$	actually $s \notin S$
predict $s \in S$	P_T	P_F
predict $s \notin S$	N_F	N_T
statistics	sensitivity	specificity
formulas	$P_T/(P_T + N_F)$	$N_T/(P_F + N_T)$

Clearly, we want both 100% sensitivity and 100% specificity.

6 Dependencies and thresholds

The reason that sensitivity and specificity are important is that the traditional method of monitoring via threshold values cannot accurately detect problems

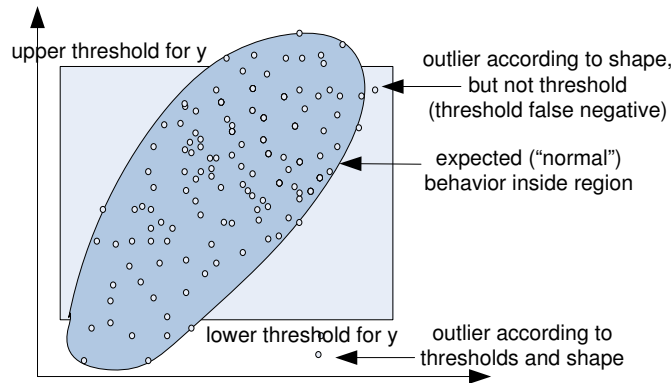


Figure 4: Thresholds that are set lower to increase sensitivity cause loss of specificity.

with a soft dependency. Consider, e.g., the situation in Figure 3. There is a covariant dependency between the X and Y axes, but thresholds cannot express that dependency. On both sides of the covariant cloud, there are $\langle X, Y \rangle$ values that should raise alarms but do not. So, if the threshold is set to contain the whole square, it has 100% sensitivity (all positives are treated as positives) but it exhibits lower specificity (many possible negatives are treated as positive).

One might think that one could repair the problem by lowering thresholds, but *this trades one type of accuracy for another*. The situation in Figure 4 shows thresholds lower than the actual acceptable bounds. There is higher specificity (because some false negatives are included), but at the expense of lower sensitivity (not all positives are treated as positive) and more false alarms.

Thus there is some motivation for being able to determine whether a given measurement conforms to the previously measured shape of the data.

7 Exploiting machine learning

What machine learning does for us in this situation is to allow us, simply and automatically, to detect situations in which groups of parameters do not conform to established norms. In the covariant case, we can detect when points are not on the diagonal, while in the contravariant case, we can detect when there are points in the middle. We can do this without specifying anything other than a sample of normal behavior, and we do not even need to know beforehand whether pairs are covariant or contravariant. The machine learning algorithm figures that out for us.

The basic idea of “one-class learning” is to use a set of “good examples” of system behavior or performance to define “shapes” that tell us whether a future state of the system is good or bad. For example, given a set of samples exhibiting

covariance, machine learning can draw a boundary around these points and flag any situation outside this norm (Figure 5). Given a set of samples exhibiting contravariance, we can likewise draw a boundary around the samples and detect if any situation violates the behavior we consider “normal”. The wondrous thing about these examples is that the *same* machine learning algorithm can handle either case.

Let us consider the special case of “one-class learning with radial kernel”. This algorithm starts at a given origin and tries to form a surface around good parameter values by drawing vectors from that origin to the points on the boundary of the “good” set. We presume that every point between the origin and the boundary is included, and everything else is outside. If we consider the origin of the surface to be (0,0) in two dimensions, then this learning algorithm can learn both covariant (Figure 5) and contravariant (Figure 6) relationships.

One important thing to note, even in two dimensions, is that erroneous behavior that is *not* detected by thresholding *is* detected by considering covariance and contravariance. If one’s network service suddenly stops responding, then there is no “threshold” that can detect this. In contrast, the covariance between input and output will be violated immediately. The points “to the side” are violations of covariance but not of threshold. Further, there are many more of these than there are “normal” points, so the potential to miss important events is high.

Even more importantly, machine learning can detect such covariance and contravariance in any number of dimensions of data, and can handle situations in which both covariant and contravariant relationships define “normalcy”. Thus it is possible to think of machine learning as *encapsulating the soft dependencies in monitoring data* and allowing one to check for those dependencies.

8 Tools

Several tools currently exist for applying machine learning to various fields. For this paper’s research, we use `libsvm`[10], a publicly available C++ and Java library for implementing support-vector machines (SVMs), the form of machine learning we employed to create the majority of our figures. `Libsvm` comes with a *NIX command-line interface that provides easy access to various kinds of support-vector algorithm. The library also exists in many other forms, including the Perl module `Algorithm::SVM`, which John Orthofer used to create a prototype monitoring framework that inspired this paper.

We think our research shows that any tools for applying SVM algorithms must come with some way to visualize the effects of one’s actions. When utilized naïvely, the SVM algorithms can do unpredictable things that may or may not be beneficial. It helps to know the shapes for which one is searching *before* training an SVM to look for them. While this is not always an easy task, it is usually easier than other alternatives that offer a lower payout.

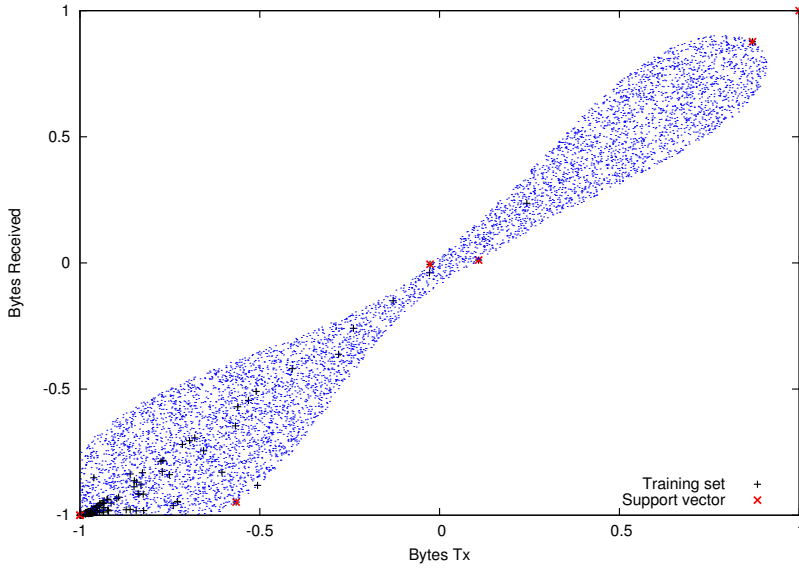


Figure 5: A machine-learned classifier for covariant data includes points around the data but also a few extra points. Red x marks are the “support vectors” whose positions define the shape. Black plus marks are members of the training set used to build the classifier. Blue dots represent data points that are considered to be within the “good” data set.

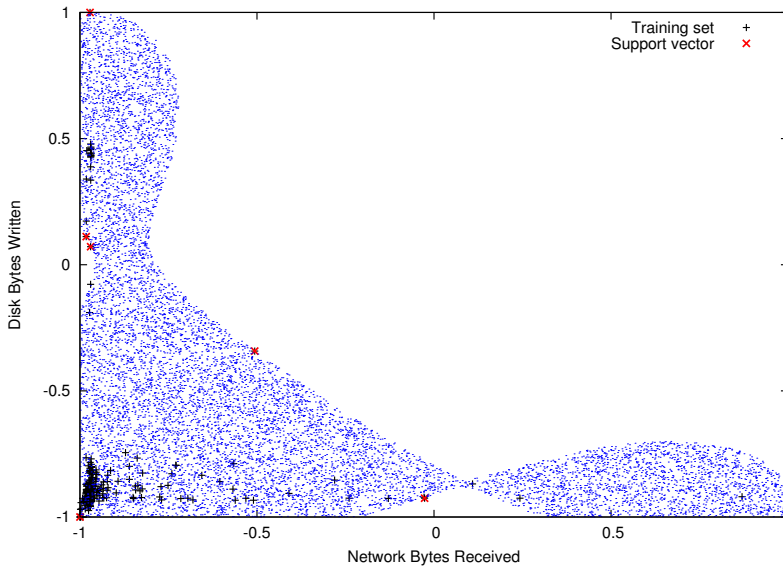


Figure 6: A machine-learned classifier for contravariant data includes some points around the data. The classifier may need further tuning.

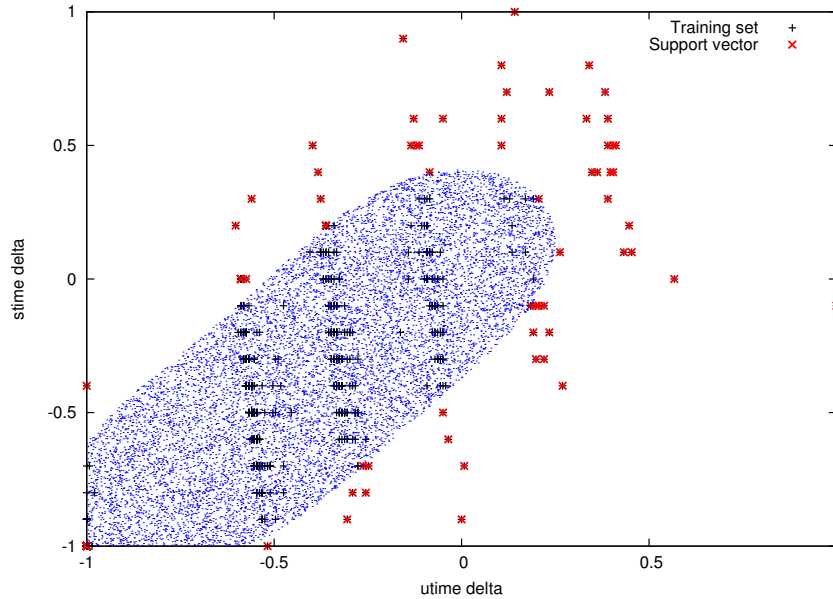


Figure 7: Vertical sets of training points come from two different classes of process (user and system daemon), but the classifier can't distinguish them.

9 Limitations

Applying machine learning to monitoring has limitations that are related to its strengths. Each limitation lowers specificity and/or sensitivity.

The first limitation is that one's accuracy is completely dependent upon the reasonable choice of a training set from which to derive bounds. If data in the training set for normal behavior represent alarm conditions, then specificity is lower, because these situations will be misinterpreted as "good". If the data in the training set do not represent all "good" conditions, sensitivity is lower, because "good" conditions can be interpreted as "bad".

The second limitation arises in handling "multi-class" input. Often, one believes that a set of parameters represents the behavior of one population when it actually describes two different populations with distinct behaviors. Consider, e.g., the situation in Figure 7. Here a pair of parameters (utime and stime) exhibit different covariances for two distinct populations, e.g., system and user processes. The problem with this picture is that any point *between* the population lines will be considered "good", even though it does not correspond to any observed concept of good behavior, which reduces specificity. Thus it is necessary to filter input data into different "classes" to improve specificity, and treat the two kinds of data as distinct, even though they arise from the same source.

The third limitation is sensitivity to the choice of training data. It is pos-

sible to get a “bad sample” and have to augment it by hand in order to make the learning algorithm work properly. E.g., it is not possible to make decisions about high load without actually observing high-load relationships between parameters. In turn, there is a “training delay” while machine learning algorithms gather enough data to describe something as “normal” or not. During this training time, one must live with the standard mechanisms of defining normalcy via thresholds.

A related limitation is sensitivity to noise in data. If the input data is “noisy”, in the sense that the training set contains some percentage of “bad” examples, then the classifier will make mistakes based upon that noise. There are many mechanisms for dealing with noise.

A last limitation is chaos. The classifiers we have studied behave chaotically at the boundaries of learned classes, in the sense that two points that are very close and adjacent may be classified in different classes. This is a property of the classifier algorithm and the way that it copes with noise and other distractions.

It is thus important, in using this data, to understand the “character” of the data being classified, as well as the character of the algorithms being used, and not just to classify data blindly.

[To the program committee: the final paper will also include a brief technical description of how support-vector machines work, as well as more examples of high-dimensional machine learning.]

10 Future work

The dream of this work is that eventually, there will be something like a Nagios plugin that does this kind of filtering for every sysadmin, and that the language of covariance and contravariance and shape and topology will augment and empower the traditional monitoring language of threshold, bound, and memory. But this language must be learned. It is a language of instances rather than boundaries between instances, and relies upon actual behavior to define normality and exception. It does not replace thresholding, but rather augments it as another mechanism for defining target states.

In this paper, we have mostly covered so-called “one-class machine learning”, in which the objective is to create a filter that distinguishes one class of things from all others. This is the only kind of filter that does not require extensive human input to train. The other kind of filter, “two-class learning”, takes as input examples of both good and bad behavior. This means that someone has to identify what is “bad”. But, as a result of incorporating this data, “two-class learning” can be much more accurate than “one-class learning”. Perhaps, in the future, you will tell your monitoring system, instance by instance, whether it is correctly classifying behavior, until it “learns” – from you – the difference between good and bad.

In this paper, we expose a language and set of operators for building new kinds of monitoring filters that – while not particularly new to statisticians – seems to be new to the monitoring community, and represents a different world-

view than that of traditional monitoring. What is left to be done is not just to develop tools that do this for the masses, but also to develop practices for use of this new way of thinking. Unlike thresholding, the practices here require some feel for the data and its relationships. These practices, and this kind of “feel” for the technique and its powers and limitations, must grow hand-in-hand with any tools. As yet, the nature of these practices is unclear.

One important goal of machine learning is for machines to learn to manage themselves through instance-based reinforcement; that they can learn what is important and what is not, and that administrators can become partners rather than masters. We should be able to say “next time this thing happens, please tell me about it” and “never wake me up at 2 am about this ever again”, but *without* further ill effects. This goal is far from realistic at present, however, due to the limited data involved in such interactions. One sysadmin, acting alone, cannot hope to provide a machine learning algorithm with enough instances from which to paint a full picture of normalcy. This would be a full-time job, with no time left over to manage systems.

It is more likely that the machine learning of the future will involve a more “global” idea of normalcy than just sampling instances from one network. While one sysadmin cannot define normalcy to a machine’s satisfaction, all of us – working together – would have a chance. A second strategy, however, is to utilize analogy; machine B is normal if it is acting approximately like machine A. This requires, however, a finer understanding of how differences in machines affect behavior, and which behaviors are in fact invariant of hardware and software variations that occur naturally. In short, employing this technology in practice will require a better and more thorough understanding of the “shape of causality” that defines system behavior.

References

- [1] K. Begnum and M. Burgess. Principle components and importance ranking of distributed anomalies. *Mach. Learn.*, 58(2-3):217–230, 2005.
- [2] J. D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proceedings of the 14th USENIX Conference on System Administration*, pages 139–146, Berkeley, CA, USA, 2000. USENIX Association.
- [3] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.
- [4] M. Burgess. Computer immunology. *Proceedings of the 12th USENIX Conference on System Administration*, page 283, 1998.
- [5] M. Burgess. Theoretical system administration. In *Proceedings of the 14th USENIX Conference on System Administration*, pages 1–14, Berkeley, CA, USA, 2000. USENIX Association.

- [6] M. Burgess. Two dimensional time-series for anomaly detection and regulation in adaptive systems. In *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 169–180. Springer-Verlag, 2002.
- [7] M. Burgess. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming*, 60(1):1–26, 2006.
- [8] M. Burgess, H. Haugerud, S. Straumsnes, and T. Reitan. Measuring system normality. *ACM Transactions on Computer Systems*, 20:125–160, 2002.
- [9] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software practice and experience*, 27:1083, 1997.
- [10] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [11] M. Chiarini, J. Danziger, and A. Couch. Improving troubleshooting: Dynamic analysis of dependencies. 2006 USENIX Annual Technical Conference, Poster Session.
- [12] J. Hart and J. D’Amelia. An analysis of rpm validation drift. In *Proceedings of the 16th USENIX Conference on System Administration*, pages 155–166, Berkeley, CA, USA, 2002. USENIX Association.
- [13] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vialta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Conference on Knowledge Discovery in Data*, pages 426–435, Washington, D.C., 2003. ACM Press.
- [14] Y. Sun and A. Couch. Global impact analysis of dynamic library dependencies. In *Proceedings of the 15th USENIX Conference on System Administration*, pages 145–150, Berkeley, CA, USA, 2001. USENIX Association.