O'REILLY®

Anomaly Detection for Monitoring

Preetam Jinka & Baron Schwartz

4 Easy Ways to Stay Ahead of the Game

The world of web ops and performance is constantly changing. Here's how you can keep up:

1. **Download free reports** on the current and trending state of web operations, dev ops, business, mobile, and web performance. http://oreil.ly/free_resources
2. **Watch free videos and webcasts** from some of the best minds in the field—watch what you like, when you like, where you like. http://oreil.ly/free_resources
3. **Subscribe** to the weekly O'Reilly Web Ops and Performance newsletter. http://oreil.ly/getnews
4. **Attend the O'Reilly Velocity Conference**, the must-attend gathering for web operations and performance professionals, with events in California, New York, Europe, and China. http://velocityconf.com

For more information and additional Web Ops and Performance resources, visit **http://oreil.ly/Web_Ops**.

O'REILLY®

Anomaly Detection for Monitoring

A Statistical Approach to Time Series Anomaly Detection

Preetam Jinka & Baron Schwartz

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Anomaly Detection for Monitoring
by Preetam Jinka and Baron Schwartz

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Anderson
Production Editor: Nicholas Adams
Proofreader: Nicholas Adams
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition
2015-10-06: First Release
2016-03-09: Second Release

978-1-491-93578-1

[LSI]

Table of Contents

Foreword

Monitoring is currently undergoing a significant change. Until two or three years ago, the main focus of monitoring tools was to provide more and better data. Interpretation and visualization has too often been an afterthought. While industries like e-commerce have jumped on the data analytics train very early, monitoring systems still need to catch up.

These days, systems are getting larger and more dynamic. Running hundreds of thousands of servers with continuous new code pushes in elastic, self-scaling server environments makes data interpretation more complex than ever. We as an industry have reached a point where we need software tooling to augment our human analytical skills to master this challenge.

At Ruxit, we develop next-generation monitoring solutions based on artificial intelligence and deep data (large amounts of highly interlinked pieces of information). Building self-learning monitoring systems—while still in its early days—helps operations teams to focus on core tasks rather than trying to interpret a wall of charts. Intelligent monitoring is also at the core of the DevOps movement, as well-interpreted information enables sharing across organisations.

Whenever I give a talk about this topic, at least one person raises the question about where he can buy a book to learn more about the topic. This was a tough question to answer, as most literature is targeted toward mathematicians—if you want to learn more on topics like anomaly detection, you are quickly exposed to very advanced content. This book, written by practitioners in the space, finds the perfect balance. I will definitely add it to my reading recommendations.

—Alois Reitbauer,
Chief Evangelist, Ruxit

CHAPTER 1
Introduction

Wouldn't it be amazing to have a system that warned you about new behaviors and data patterns in time to fix problems before they happened, or seize opportunities the moment they arise? Wouldn't it be incredible if this system was completely foolproof, warning you about every important change, but never ringing the alarm bell when it shouldn't? That system is the holy grail of anomaly detection. It doesn't exist, and probably never will. However, we shouldn't let imperfection make us lose sight of the fact that useful anomaly detection is possible, and benefits those who apply it appropriately.

Anomaly detection is a set of techniques and systems to find unusual behaviors and/or states in systems and their observable signals. We hope that people who read this book do so because they believe in the promise of anomaly detection, but are confused by the furious debates in thought-leadership circles surrounding the topic. We intend this book to help demystify the topic and clarify some of the fundamental choices that have to be made in constructing anomaly detection mechanisms. We want readers to understand why some approaches to anomaly detection work better than others in some situations, and why a better solution for some challenges may be within reach after all.

This book is not intended to be a comprehensive source for all information on the subject. That book would be 1000 pages long and would be incomplete at that. It is also not intended to be a step-by-step guide to building an anomaly detection system that will work well for all applications—we're pretty sure that a "general solu-

tion" to anomaly detection is impossible. We believe the best approach for a given situation is dependent on many factors, not least of which is the cost/benefit analysis of building more complex systems. We hope this book will help you navigate the labyrinth by outlining the tradeoffs associated with different approaches to anomaly detection, which will help you make judgments as you reach forks in the road.

We decided to write this book after several years of work applying anomaly detection to our own problems in monitoring and related use cases. Both of us work at VividCortex, where we work on a large-scale, specialized form of database monitoring. At VividCortex, we have flexed our anomaly detection muscles in a number of ways. We have built, and more importantly discarded, dozens of anomaly detectors over the last several years. But not only that, we were working on anomaly detection in monitoring systems even before VividCortex. We have tried statistical, heuristic, machine learning, and other techniques.

We have also engaged with our peers in monitoring, DevOps, anomaly detection, and a variety of other disciplines. We have developed a deep and abiding respect for many people, projects and products, and companies including Ruxit among others. We have tried to share our challenges, successes, and failures through blogs, open-source software, conference talks, and now this book.

Why Anomaly Detection?

Monitoring, the practice of observing systems and determining if they're healthy, is hard and getting harder. There are many reasons for this: we are managing many more systems (servers and applications or services) and much more data than ever before, and we are monitoring them in higher resolution. Companies such as Etsy have convinced the community that it is not only possible but desirable to monitor practically everything we can, so we are also monitoring many more signals from these systems than we used to.

Any of these changes presents a challenge, but collectively they present a very difficult one indeed. As a result, now we struggle with making sense out of all of these metrics.

Traditional ways of monitoring all of these metrics can no longer do the job adequately. There is simply too much data to monitor.

Many of us are used to monitoring visually by actually watching charts on the computer or on the wall, or using thresholds with systems like Nagios. Thresholds actually represent one of the main reasons that monitoring is too hard to do effectively. Thresholds, put simply, don't work very well. Setting a threshold on a metric requires a system administrator or DevOps practitioner to make a decision about the correct value to configure.

The problem is, there is no correct value. A static threshold is just that: static. It does not change over time, and by default it is applied uniformly to all servers. But systems are neither similar nor static. Each system is different from every other, and even individual systems change, both over the long term, and hour to hour or minute to minute.

The result is that thresholds are too much work to set up and maintain, and cause too many false alarms and missed alarms. False alarms, because normal behavior is flagged as a problem, and missed alarms, because the threshold is set at a level that fails to catch a problem.

You may not realize it, but threshold-based monitoring is actually a crude form of anomaly detection. When the metric crosses the threshold and triggers an alert, it's really flagging the value of the metric as anomalous. The root of the problem is that this form of anomaly detection cannot adapt to the system's unique and changing behavior. It cannot learn what is normal.

Another way you are already using anomaly detection techniques is with features such as Nagios's flapping suppression, which disallows alarms when a check's result oscillates between states. This is a crude form of a low-pass filter, a signal-processing technique to discard noise. It works, but not all that well because its idea of noise is not very sophisticated.

A common assumption is that more sophisticated anomaly detection can solve all of these problems. We assume that anomaly detection can help us reduce false alarms and missed alarms. We assume that it can help us find problems more accurately with less work. We assume that it can suppress noisy alerts when systems are in unstable states. We assume that it can learn what is normal for a system, automatically and with zero configuration.

Why do we assume these things? Are they reasonable assumptions? That is one of the goals of this book: to help you understand your assumptions, some of which you may not realize you're making. With explicit assumptions, we believe you will be prepared to make better decisions. You will be able to understand the capabilities and limitations of anomaly detection, and to select the right tool for the task at hand.

The Many Kinds of Anomaly Detection

Anomaly detection is a complicated subject. You might understand this already, but nevertheless it is probably still more complicated than you believe. There are many kinds of anomaly detection techniques. Each technique has a dizzying number of variations. Each of these is suitable, or unsuitable, for use in a number of scenarios. Each of them has a number of edge cases that can cause poor results. And many of them are based on advanced math, statistics, or other disciplines that are beyond the reach of most of us.

Still, there are lots of success stories for anomaly detection in general. In fact, as a profession, we are late at applying anomaly detection on a large scale to monitoring. It certainly has been done, but if you look at other professions, various types of anomaly detection are standard practice. This applies to domains such as credit card fraud detection, monitoring for terrorist activity, finance, weather, gambling, and many more too numerous to mention. In contrast to this, in systems monitoring we generally do not regard anomaly detection as a standard practice, but rather as something potentially promising but leading edge.

The authors of this book agree with this assessment, by and large. We also see a number of obstacles to be overcome before anomaly detection is regarded as a standard part of the monitoring toolkit:

- It is difficult to get started, because there's so much to learn before you can even start to get results.
- Even if you do a lot of work and the results seem promising, when you deploy something into production you can find poor results often enough that nothing usable comes of your efforts.
- General-purpose solutions are either impossible or extremely difficult to achieve in many domains. This is partially because of the incredible diversity of machine data. There are also appa-

rently an almost infinite number of edge cases and potholes that can trip you up. In many of these cases, things appear to work well even when they really don't, or they accidentally work well, leading you to think that it is by design. In other words, whether something is actually working or not is a very subtle thing to determine.

- There seems to be an unlimited supply of poor and incomplete information to be found on the Internet and in other sources. Some of it is probably even in this book.
- Anomaly detection is such a trendy topic, and it is currently so cool and thought-leadery to write or talk about it, that there seem to be incentives for adding insult to the already injurious amount of poor information just mentioned.
- Many of the methods are based on statistics and probability, both of which are incredibly unintuitive, and often have surprising outcomes. In the authors' experience, few things can lead you astray more quickly than applying intuition to statistics.

As a result, anomaly detection seems to be a topic that is all about extremes. Some people try it, or observe other people's efforts and results, and conclude that it is impossible or difficult. They give up hope. This is one extreme. At the other extreme, some people find good results, or believe they have found good results, at least in some specific scenario. They mistakenly think they have found a general purpose solution that will work in many more scenarios, and they evangelize it a little too much. This overenthusiasm can result in negative press and vilification from other people. Thus, we seem to veer between holy grails and despondency. Each extreme is actually an overcorrection that feeds back into the cycle.

Sadly, none of this does much to educate people about the true nature and benefits of anomaly detection. One outcome is that a lot of people are missing out on benefits that they could be getting. Another is that they may not be informed enough to have realistic opinions about commercially available anomaly detection solutions. As Zen Master Hakuin said,

> Not knowing how near the truth is, we seek it far away.

Conclusions

If you are like most of our friends in the DevOps and web operations communities, you probably picked up this book because you've been hearing a lot about anomaly detection in the last few years, and you're intrigued by it. In addition to the previously-mentioned goal of making assumptions explicit, we hope to be able to achieve a number of outcomes in this book.

- We want to help orient you to the subject and the landscape in general. We want you to have a frame of reference for thinking about anomaly detection, so you can make your own decisions.
- We want to help you understand how to assess not only the meaning of the answers you get from anomaly detection algorithms, but how trustworthy the answers might be.
- We want to teach you some things that you can actually apply to your own systems and your own problems. We don't want this to be just a bunch of theory. We want you to put it into practice.
- We want your time spent reading this book to be useful beyond this book. We want you to be able to apply what you have learned to topics we don't cover in this book.

If you already know anything about anomaly detection, statistics, or any of the other things we cover in this book, you're going to see that we omit or gloss over a lot of important information. That is inevitable. From prior experience, we have learned that it is better to help people form useful thought processes and mental models than to tell them what to think.

As a result of this, we hope you will be able to combine the material in this book with your existing tools and skills to solve problems on your systems. By and large, we want you to get better at what you already do, and learn a new trick or two, rather than solving world hunger. If you ask, "what can I do that's a little better than Nagios?" you're on the right track.

Anomaly detection is not a black and white topic. There is a lot of gray area, a lot of middle ground. Despite the complexity and richness of the subject matter, it is both fun and productive. And despite the difficulty, there is a lot of promise for applying it in practice.

Somewhere between static thresholds and magic, there is a happy medium. In this book, we strive to help you find that balance, while avoiding some of the sharp edges.

CHAPTER 2
A Crash Course in Anomaly Detection

This isn't a book about the overall breadth and depth of anomaly detection. It is specifically about applying anomaly detection to solve common problems that the DevOps community faces when trying to monitor the types of systems that we manage the most.

One of the implications is that this book is mostly about time series anomaly detection. It also means that we focus on widely used tools such as Graphite, JavaScript, R, and Python. There are several reasons for these choices, based on assumptions we're making.

- We assume that our audience is largely like ourselves: developers, system administrators, database administrators, and DevOps practitioners using mostly open source tools.
- Neither of us has a doctorate in a field such as statistics or operations research, and we assume you don't either.
- We assume that you are doing time series monitoring, much like we are.

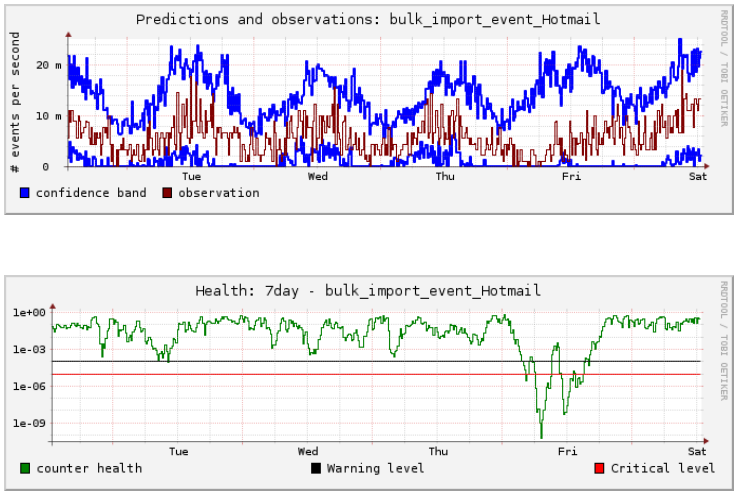
As a result of these assumptions, this book is quite biased. It is all about anomaly detection on metrics, and we will not cover anomaly detection on configuration, comparing machines amongst each other, log analysis, clustering similar kinds of things together, or many other types of anomaly detection. We also focus on detecting anomalies as they happen, because that is usually what we are trying to do with our monitoring systems.

A Real Example of Anomaly Detection

Around the year 2008, Evan Miller published a paper describing real-time anomaly detection in operation at IMVU.[1] This was Baron's first exposure to anomaly detection:

> At approximately 5 AM Friday, it first detects a problem [in the number of IMVU users who invited their Hotmail contacts to open an account], which persists most of the day. In fact, an external service provider had changed an interface early Friday morning, affecting some but not all of our users.

The following images from that paper show the metric and its deviation from the usual behavior.



They detected an unusual change in a really erratic signal. Mind. Blown. Magic!

The anomaly detection method was Holt-Winters forecasting. It is relatively crude by some standards, but nevertheless can be applied with good results to carefully selected metrics that follow predictable patterns. Miller went on to mention other examples where the same technique had helped engineers find problems and solve them quickly.

1 "Aberrant Behavior Detection in Time Series for Monitoring Business-Critical Metrics"

How can you achieve similar results on your systems? To answer this, first we need to consider what anomaly detection is and isn't, and what it's good and bad at doing.

What Is Anomaly Detection?

Anomaly detection is a way to help find signal in noisy metrics. The usual definition of "anomaly" is an unusual or unexpected event or value. In the context of anomaly detection on monitoring metrics, we care about unexpected *values* of those metrics.

Anomalies can have many causes. It is important to recognize that the anomaly in the metric that we are observing is not the same as the *condition in the system* that produced the metric. By assuming that an anomaly in a metric indicates a problem in the system, we are making a mental and practical leap that may or may not be justified. Anomaly detection doesn't understand anything about your systems. It just understands your definition of unusual or abnormal values.

It is also good to note that most anomaly detection methods substitute "unusual" and "unexpected" with "statistically improbable." This is common practice and often implicit, but you should be aware of the difference.

A common confusion is thinking that anomalies are the same as outliers (values that are very distant from typical values). In fact, outliers are common, and they should be regarded as normal and expected. Anomalies are outliers, at least in most cases, but not all outliers are anomalies.

What Is It Good for?

Anomaly detection has a variety of use cases. Even within the scope of this book, which we previously indicated is rather small, anomaly detection can do a lot of things:

- It can find unusual values of metrics in order to surface undetected problems. An example is a server that gets suspiciously busy or idle, or a smaller than expected number of events in an interval of time, as in the IMVU example.
- It can find changes in an important metric or process, so that humans can investigate and figure out why.

- It can reduce the surface area or search space when trying to diagnose a problem that has been detected. In a world of millions of metrics, being able to find metrics that are behaving unusually at the moment of a problem is a valuable way to narrow the search.
- It can reduce the need to calibrate or recalibrate thresholds across a variety of different machines or services.
- It can augment human intuition and judgment, a little bit like the Iron Man's suit augments his strength.

Anomaly detection *cannot* do a lot of things people sometimes think it can. For example:

- It cannot provide a root cause analysis or diagnosis, although it can certainly assist in that.
- It cannot provide hard yes or no answers about whether there is an anomaly, because at best it is limited to the probability of whether there might be an anomaly or not. (Even humans are often unable to determine conclusively that a value is anomalous.)
- It cannot prove that there is an anomaly in the system, only that there is something unusual about the *metric* that you are observing. Remember, the metric isn't the system itself.
- It cannot detect actual system faults (failures), because a fault is different from an anomaly. (See the previous point again.)
- It cannot replace human judgment and experience.
- It cannot understand the meaning of metrics.
- And in general, it cannot work generically across all systems, all metrics, all time ranges, and all frequency scales.

This last item is quite important to understand. There are pathological cases where every known method of anomaly detection, every statistical technique, every test, every false positive filter, *everything*, will break down and fail. And on large data sets, such as those you get when monitoring lots of metrics from lots of machines at high resolution in a modern application, you *will* find these pathological cases, guaranteed.

In particular, at a high resolution such as one-second metrics resolution, most machine-generated metrics are extremely noisy, and will

cause most anomaly detection techniques to throw off lots and lots of false positives.

Are Anomalies Rare?

Depending on how you look at it, anomalies are either rare or common. The usual definition of an anomaly uses probabilities as a proxy for unusualness. A rule of thumb that shows up often is three standard deviations away from the mean. This is a technique that we will discuss in depth later, but for now it suffices to say that if we assume the data behaves exactly as expected, 99.73% of observations will fall within three sigmas. In other words, slightly less than three observations per thousand will be considered anomalous.

That sounds pretty rare, but given that there are 1,440 minutes per day, you'll still be flagging about 4 observations as anomalous every single day, even in one minute granularity. If you use one second granularity, you can multiply that number by 60. Suddenly these rare events seem incredibly common. One might even call them noisy, no?

Is this what you want on every metric on every server that you manage? You make up your own mind how you feel about that. The point is that many people probably assume that anomaly detection finds rare events, but in reality that assumption doesn't always hold.

How Can You Use Anomaly Detection?

To apply anomaly detection in practice, you generally have two options, at least within the scope of things considered in this book. Option one is to generate alerts, and option two is to record events for later analysis but don't alert on them.

Generating alerts from anomalies in metrics is a bit dangerous. Part of this is because the assumption that anomalies are rare isn't as true as you may think. See the sidebar. A naive approach to alerting on anomalies is almost certain to cause a lot of noise.

Our suggestion is not to alert on most anomalies. This follows directly from the fact that anomalies do not imply that a system is in a bad state. In other words, there is a big difference between an anomalous observation in a metric, and an actual system fault. If you can guarantee that an anomaly reliably detects a serious prob-

lem in your system, that's great. Go ahead and alert on it. But otherwise, we suggest that you don't alert on things that may have no impact or consequence.

Instead, we suggest that you record these anomalous observations, but don't alert on them. Now you have essentially created an index into the most unusual data points in your metrics, for later use in case it is interesting. For example, during diagnosis of a problem that you have detected.

One of the assumptions embedded in this recommendation is that anomaly detection is cheap enough to do online in one pass as data arrives into your monitoring system, but that ad hoc, after-the-fact anomaly detection is too costly to do interactively. With the monitoring data sizes that we are seeing in the industry today, and the attitude that you should "measure everything that moves," this is generally the case. Multi-terabyte anomaly detection analysis is usually unacceptably slow and requires more resources than you have available. Again, we are placing this in the context of what most of us are doing for monitoring, using typical open-source tools and methodologies.

Conclusions

Although it's easy to get excited about success stories in anomaly detection, most of the time someone else's techniques will not translate directly to your systems and your data. That's why you have to learn for yourself what works, what's appropriate to use in some situations and not in others, and the like.

Our suggestion, which will frame the discussion in the rest of this book, is that, generally speaking, you probably should use anomaly detection "online" as your data arrives. Store the results, but don't alert on them in most cases. And keep in mind that the map is not the territory: the metric isn't the system, an anomaly isn't a crisis, three sigmas isn't unlikely, and so on.

CHAPTER 3

Modeling and Predicting

Anomaly detection is based on predictions derived from *models*. In simple terms, a model is a way to express your previous knowledge about a system and how you expect it to work. A model can be as simple as a single mathematical equation.

Models are convenient because they give us a way to describe a potentially complicated process or system. In some cases, models directly describe processes that govern a system's behavior. For example, VividCortex's Adaptive Fault Detection algorithm uses Little's law[1] because we know that the systems we monitor obey this law. On the other hand, you may have a process whose mechanisms and governing principles aren't evident, and as a result doesn't have a clearly defined model. In these cases you can try to *fit* a model to the observed system behavior as best you can.

Why is modeling so important? With anomaly detection, you're interested in finding what is unusual, but first you have to know what to expect. This means you have to make a prediction. Even if it's implicit and unstated, this prediction process requires a model. Then you can compare the observed behavior to the model's prediction.

Almost all *online* time series anomaly detection works by comparing the current value to a prediction based on previous values. Online means you're doing anomaly detection as you see each new value

1 *http://bit.ly/littleslaw*

appear, and online anomaly detection is a major focus of this book because it's the only way to find system problems as they happen. Online methods are not instantaneous—there may be some delay—but they are the alternative to gathering a chunk of data and performing analysis after the fact, which often finds problems too late.

Online anomaly detection methods need two things: past data and a model. Together, they are the essential components for generating predictions.

There are lots of canned models available and ready to use. You can usually find them implemented in an R package. You'll also find models implicitly encoded in common methods. Statistical process control is an example, and because it is so ubiquitous, we're going to look at that next.

Statistical Process Control

Statistical process control (SPC) is based on operations research to implement quality control in engineering systems such as manufacturing. In manufacturing, it's important to check that the assembly line achieves a desired level of quality so problems can be corrected before a lot of time and money is wasted.

One metric might be the size of a hole drilled in a part. The hole will never be *exactly* the right size, but should be within a desired tolerance. If the hole is out of tolerance limits, it may be a hint that the drill bit is dull or the jig is loose. SPC helps find these kinds of problems.

SPC describes a framework behind a family of methods, each progressing in sophistication. The *Engineering Statistics Handbook* is an excellent resource to get more detailed information about process control techniques in general.[2] We'll explain some common SPC methods in order of complexity.

Basic Control Chart

The most basic SPC method is a *control chart* that represents values as clustered around a mean and control limits. This is also known as the *Shewhart control chart*. The fixed mean is a value that we expect

2 *http://bit.ly/stathandbook*

(say, the size of the drill bit), and the control lines are fixed some number of standard deviations away from that mean. If you've heard of the *three sigma rule*, this is what it's about. Three sigmas represents three standard deviations away from the mean. The two control lines surrounding the mean represent an acceptable range of values.

The Gaussian (Normal) Distribution

A *distribution* represents how frequently each possible value occurs. Histograms are often used to visualize distributions. The Gaussian distribution, also called the normal distribution or "bell curve," is a commonly used distribution in statistics that is also ubiquitous in the natural world. Many natural phenomena such as coin flips, human characteristics such as height, and astronomical observations have been shown to be at least approximately normally distributed.[3] The Gaussian distribution has many nice mathematical properties, is well understood, and is the basis for lots of statistical methods.

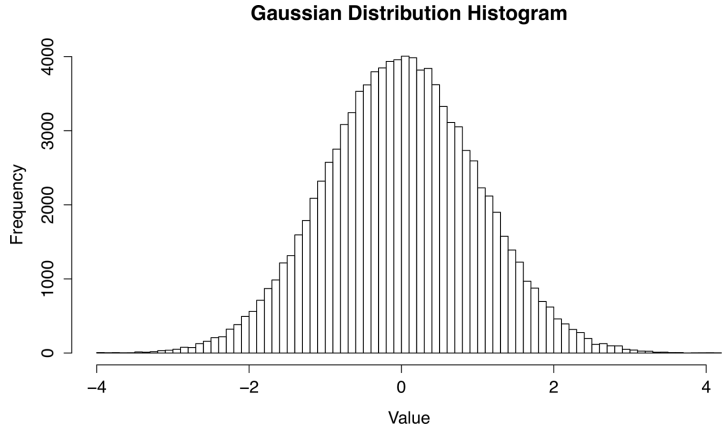


Figure 3-1. Histogram of the Gaussian distribution with mean 0 and standard deviation 1.

One of the assumptions made by the basic, fixed control chart is that values are stable: the mean and spread of values is constant. As a formula, this set of assumptions can be expressed as: $y = \mu + \varepsilon$. The

3 *History of the Normal Distribution*

letter μ represents a constant mean, and ε is a random variable representing noise or error in the system.

In the case of the basic control chart model, ε is assumed to be a Gaussian distributed random variable.

Control charts have the following characteristics:

- They assume a fixed or known mean and spread of values.
- The values are assumed to be Gaussian (normally) distributed around the mean.
- They can detect one or multiple points that are outside the desired range.

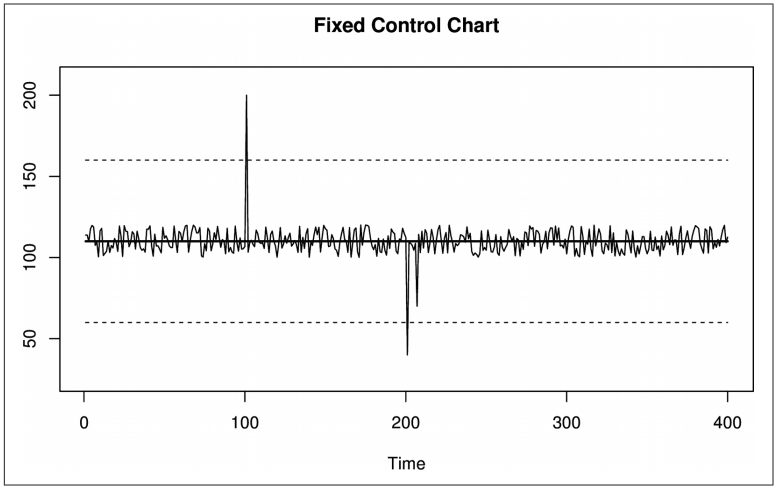


Figure 3-2. A basic control chart with fixed control limits, which are represented with dashed lines. Values are considered to be anomalous if they cross the control limits.

Moving Window Control Chart

The major problem with a basic control chart is the assumption of stability. In time series analysis, the usual term is *stationary*, which means the values have a consistent mean and spread over time.

Many systems change rapidly, so you can't assume a fixed mean for the metrics you're monitoring. Without this key assumption holding true, you will either get false positives or fail to detect true anoma-

lies. To fix this problem, the control chart needs to adapt to a changing mean and spread over time. There are two basic ways to do this:

- Slice up your control chart into smaller time ranges or *fixed windows*, and treat each window as its own independent fixed control chart with a different mean and spread. The values within each window are used to compute the mean and standard deviation for that window. Within a small interval, everything looks like a regular fixed control chart. At a larger scale, what you have is a control chart that changes across windows.
- Use a *moving window*, also called a *sliding window*. Instead of using predefined time ranges to construct windows, at each point you generate a moving window that covers the previous N points. The benefit is that instead of having a fixed mean within a time range, the mean changes after each value yet still considers the same number of points to compute the mean.

Moving windows have major disadvantages. You have to keep track of recent history because you need to consider all of the values that fall into a window. Depending on the size of your windows, this can be computationally expensive, especially when tracking a large number of metrics. Windows also have poor characteristics in the presence of large spikes. When a spike enters a window, it causes an abrupt shift in the window until the spike eventually leaves, which causes another abrupt shift.

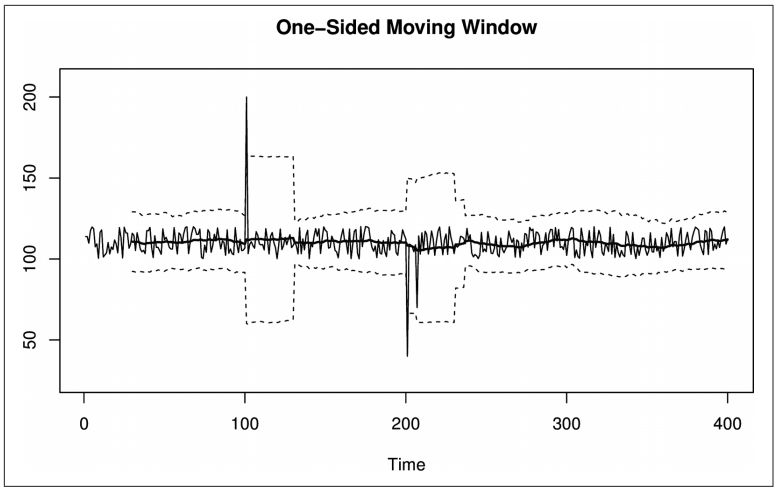


Figure 3-3. A moving window control chart. Unlike the fixed control chart shown in Figure 3-2, this moving window control chart has an adaptive control line and control limits. After each anomalous spike, the control limits widen to form a noticeable box shape. This effect ends when the anomalous value falls out of the moving window.

Moving window control charts have the following characteristics:

- They require you to keep some amount of historical data to compute the mean and control limits.
- The values are assumed to be Gaussian (normally) distributed around the mean.
- They can detect one or multiple points that are outside the desired range.
- Spikes in the data can cause abrupt changes in parameters when they are in the distant past (when they exit the window).

Exponentially Weighted Control Chart

An exponentially weighted control chart solves the "spike-exiting problem," where distant history influences control lines, by replacing the fixed-length moving windows with an infinitely large, gradu-

ally decaying window. This is made possible using an exponentially weighted moving average.

Exponentially Weighted Moving Average

An exponentially weighted moving average (EWMA) is an alternative to moving windows for computing moving averages. Instead of using a fixed number of values to compute an average within a window, an EWMA considers *all* previous points but places higher weights on more recent data. This weighting, as the name suggests, decays exponentially. The implementation, however, uses only a single value so it doesn't have to "remember" a lot of historical data.

EWMAs are used everywhere from UNIX load averages to stock market predictions and reporting, so you've probably had at least some experience with them already! They have very little to do with the field of statistics itself or Gaussian distributions, but are very useful in monitoring because they use hardly any memory or CPU.

One disadvantage of EWMAs is that their values are nondeterministic because they essentially have infinite history. This can make them difficult to troubleshoot.

EWMAs are continuously decaying windows. Values never "move out" of the tail of an EWMA, so there will never be an abrupt shift in the control chart when a large value gets older. However, because there is an immediate transition *into* the head of a EWMA, there will still be abrupt shifts in a EWMA control chart when a large value is first observed. This is generally not as bad a problem, because although the smoothed value changes a lot, it's changing in response to current data instead of very old data.

Using an EWMA as the mean in a control chart is simple enough, but what about the control limit lines? With the fixed-length windows, you can trivially calculate the standard deviation within a window. With an EWMA, it is less obvious how to do this. One method is keeping another EWMA of the *squares* of values, and then using the following formula to compute the standard deviation.

$$\text{StdDev}(Y) = \sqrt{\text{EWMA}\left(Y^2\right) - \text{EWMA}(Y)^2}$$

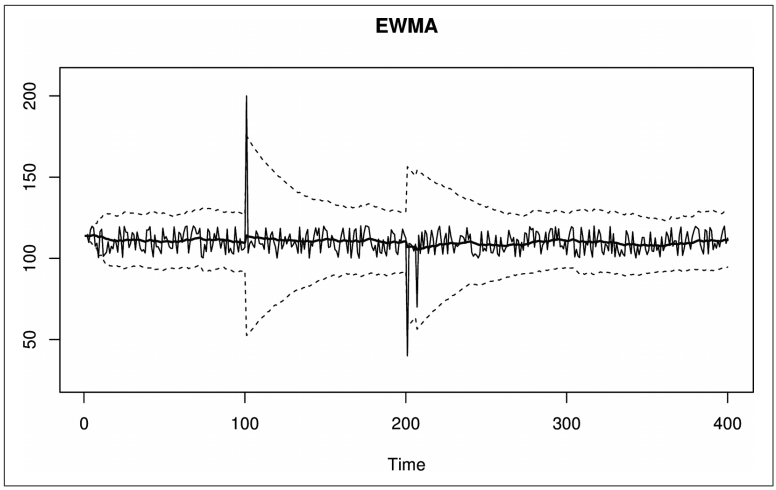


Figure 3-4. An exponentially weighted moving window control chart. This is similar to Figure 3-3, except it doesn't suffer from the sudden change in control limit width when an anomalous value ages.

Exponentially weighted control charts have the following characteristics:

- They are memory- and CPU-efficient.
- The values are assumed to be Gaussian (normally) distributed around the mean.
- They can detect one or multiple points that are outside the desired range.
- A spike can temporarily inflate the control lines enough to cause missed alarms afterwards.
- They can be difficult to debug because the EWMA's value can be hard to determine from the data itself, since it is based on potentially "infinite" history.

Window Functions

Sliding windows and EWMAs are part of a much bigger category of *window functions*. They are window functions with two and one sharp edges, respectively.

There are lots of window functions with many different shapes and characteristics. Some functions increase smoothly from 0 to 1 and

back again, meaning that they smooth data using both past and future data. Smoothing bidirectionally can eliminate the effects of large spikes.

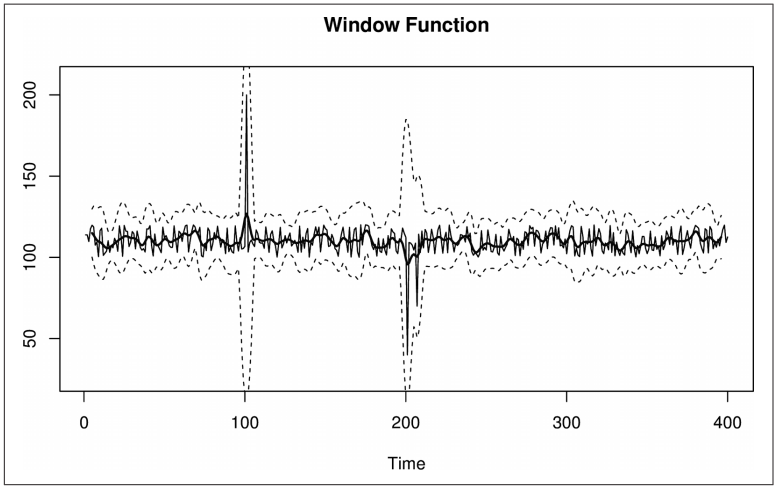


Figure 3-5. A window function control chart. This time, the window is formed with values on both sides of the current value. As a result, anomalous spikes won't generate abrupt shifts in control limits even when they first enter the window.

The downside to window functions is that they require a larger time delay, which is a result of not knowing the smoothed value until enough future values have been observed. This is because when you center a bidirectional windowing function on "now," it extends into the future. In practice, EWMAs are a good enough compromise for situations where you can't measure or wait for future values.

Control charts based on bidirectional smoothing have the following characteristics:

- They will introduce time lag into calculations. If you smooth symmetrically over 60 second-windows, you won't know the smoothed value of "now" until 30 seconds—half the window—has passed.
- Like sliding windows, they require more memory and CPU to compute.

- Like all the SPC control charts we've discussed thus far, they assume Gaussian distribution of data.

More Advanced Time Series Modeling

There are entire families of time series models and methods that are more advanced than what we've covered so far. In particular, the ARIMA family of time series models and the surrounding methodology known as the Box-Jenkins approach is taught in undergraduate statistics programs as an introduction to statistical time series. These models express more complicated characteristics, such as time series whose current values depend on a given number of values from some distance in the past. ARIMA models are widely studied and very flexible, and form a solid foundation for advanced time series analysis. The *Engineering Statistics Handbook* has several sections[4] covering ARIMA models, among others. *Forecasting: principles and practice* is another introductory resource.[5]

You can apply many extensions and enchancements to these models, but the methodology generally stays the same. The idea is to fit or train a model to sample data. Fitting means that parameters (coefficients) are adjusted to minimize the deviations between the sample data and the model's prediction. Then you can use the parameters to make predictions or draw useful conclusions. Because these models and techniques are so popular, there are plenty of packages and code resources available in R and other platforms.

The ARIMA family of models has a number of "on/off toggles" that include or exclude particular portions of the models, each of which can be adjusted if it's enabled. As a result, they are extremely modular and flexible, and can vary from simple to quite complex.

In general, there are lots of models, and with a little bit of work you can often find one that fits your data extremely well (and thus has high predictive power). But the real value in studying and understanding the Box-Jenkins approach is the method itself, which

4 *http://bit.ly/arimamod*

5 *https://www.otexts.org/fpp/8*

remains consistent across all of the models and provides a logical way to reason about time series analysis.

Parametric and Non-Parametric Statistics and Methods

Perhaps you have heard of *parametric* methods. These are statistical methods or tools that have coefficients that must be specified or chosen via fitting. Most of the things we've mentioned thus far have parameters. For example, EWMAs have a decay parameter you can adjust to bias the value towards more recent or more historical data. The value of a mean is also a parameter. ARIMA models are full of parameters. Common statistical tools, such as the Gaussian distribution, have parameters (mean and spread).

Non-parametric methods work independently of these parameters. You might think of them as operating on dimensionless quantities. This makes them more robust in some ways, but also can reduce their descriptive power.

Predicting Time Series Data

Although we haven't talked yet about prediction, all of the tools we've discussed thus far are designed for predictions. Prediction is one of the foundations of anomaly detection. Evaluating any metric's value has to be done by comparing it to "what it should be," which is a prediction.

For anomaly detection, we're usually interested in predicting one step ahead, then comparing this prediction to the next value we see. Just as with SPC and control charts, there's a spectrum of prediction methods, increasing in complexity:

1. The simplest one-step-ahead prediction is to predict that it'll be the same as the last value. This is similar to a weather forecast. The simplest weather forecast is *tomorrow will be just like today*. Surprisingly enough, to make predictions that are subjectively a lot better than that is a hard problem! Alas, this simple method, "the next value will be the same as the current one," doesn't work well if systems aren't stable (stationary) over time.
2. The next level of sophistication is to predict that the next value will be the same as the *recent central tendency* instead. The term *central tendency* refers to *summary statistics*: single values that

attempt to be as descriptive as possible about a collection of data. With summary statistics, your prediction formula then becomes something like "the next value will be the same as the current average of recent values." Now you're predicting that values will most likely be close to what they've typically been like recently. You can replace "average" with median, EWMA, or other descriptive summary statistics.

3. One step beyond this is predicting a likely range of values centered around a summary statistic. This usually boils down to a simple mean for the central value and standard deviation for the spread, or an EWMA with EWMA control limits (analogous to mean and standard deviation, but exponentially smoothed).
4. All of these methods use parameters (e.g., the mean and standard deviation). Non-parametric methods, such as histograms of historical values, can also be used. We'll discuss these in more detail later in this book.

We can take prediction to an even higher level of sophistication using more complicated models, such as those from the ARIMA family. Furthermore, you can also attempt to build your own models based on a combination of metrics, and use the corresponding output to feed into a control chart. We'll also discuss that later in this book.

Prediction is a difficult problem in general, but it's especially difficult when dealing with machine data. Machine data comes in many shapes and sizes, and it's unreasonable to expect a single method or approach to work for all cases.

In our experience, most anomaly detection success stories work because the specific data they're using doesn't hit a pathology. Lots of machine data has simple pathologies that break many models quickly. That makes accurate, robust[6] predictions harder than you might think.

6 In statistics, robust generally means that outlying values don't throw things for a loop; for example, the median is more robust than the mean.

Evaluating Predictions

One of the most important and subtle parts of anomaly detection happens at the intersection between predicting how a metric should behave, and comparing observed values to those expectations.

In anomaly detection, you're usually using *many standard deviations from the mean* as a replacement for *very unlikely*, and when you get far from the mean, you're in the tails of the distribution. The fit tends to be much worse here than you'd expect, so even small deviations from Gaussian can result in many more outliers than you theoretically should get.

Similarly, a lot of statistical tests such as hypothesis tests are deemed to be "significant" or "good" based on what turns out to be statistician rules of thumb. Just because some p-value looks really good doesn't mean there's truly a lot of certainty. "Significant" might not signify much. Hey, it's statistics, after all!

As a result, there's a good chance your anomaly detection techniques will sometimes give you more false positives than you think they will. These problems will always happen; this is just par for the course. We'll discuss some ways to mitigate this in later chapters.

Common Myths About Statistical Anomaly Detection

We commonly hear claims that some technique, such as SPC, won't work because system metrics are not Gaussian. The assertion is that the only workable approaches are complicated non-parametric methods. This is an oversimplification that comes from confusion about statistics.

Here's an example. Suppose you capture a few observations of a "mystery time series." We've plotted this in Figure 3-6.

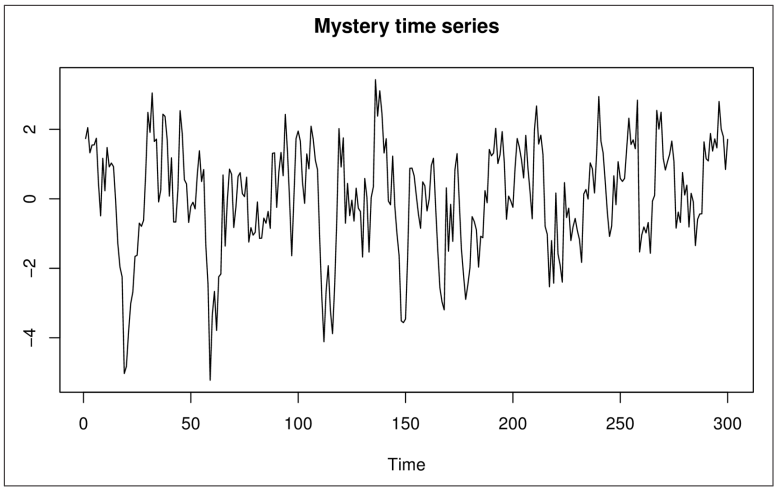


Figure 3-6. A mysterious time series about which we'll pretend we know nothing.

Is your time series Gaussian distributed? You decide to check, so you start up your R environment and plot a histogram of your time series data. For comparison, you also overlay a normal distribution curve with the same mean and standard deviation as your sample data. The result is displayed in Figure 3-7.

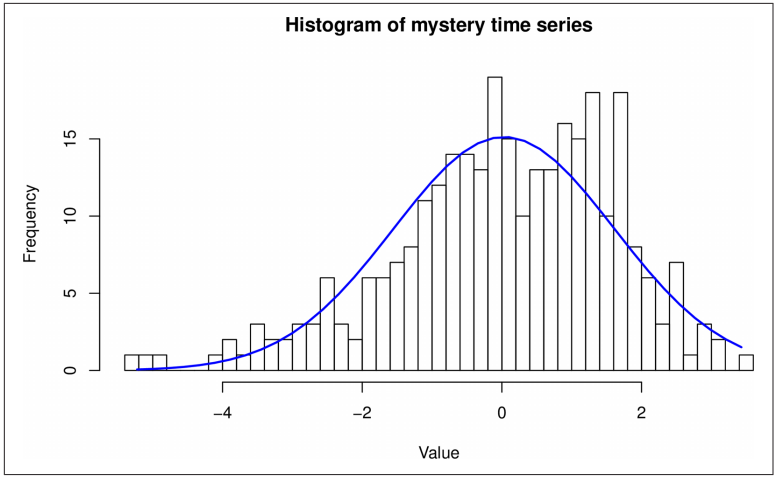


Figure 3-7. Histogram of the mystery time series, overlaid with the normal distribution's "bell curve."

Uh-oh! It doesn't look like a great fit. Should you give up hope?

No. You've stumbled into statistical quicksand:

- It's not important that the *data* is Gaussian. What matters is whether the *residuals* are Gaussian.
- The histogram is of the *sample* of data, but the *population*, not the sample, is what's important.

Let's explore each of these topics.

The Data Doesn't Need to Be Gaussian

The residuals, not the data, need to be Gaussian (normal) to use three-sigma rules and the like.

What are residuals? Residuals are the errors in prediction. They're the difference between the predictions your model makes, and the values you actually observe.

If you measure a system whose behavior is log-normal, and base your predictions on a model whose predictions are log-normal, and the errors in prediction are normally distributed, a standard SPC control chart of the results using three-sigma confidence intervals can actually work very well.

Likewise, if you have multi-modal data (whose distribution looks like a camel's humps, perhaps) and your model's predictions result in normally distributed residuals, you're doing fine.

In fact, your data can look any kind of crazy. It doesn't matter; *what matters is whether the residuals are Gaussian.* This is super-important to understand. Every type of control chart we discussed previously actually works like this:

- It models the metric's behavior somehow. For example, the EWMA control chart's implied model is "the next value is likely to be close to the current value of the EWMA."
- It subtracts the prediction from the observed value.
- It effectively puts control lines on the residual. The idea is that the residual is now a stable value, centered around zero.

Any control chart can be implemented either way:

- Predict, take the residual, find control limits, evaluate whether the residual is out of bounds

- Predict, extend the control lines around the predicted value, evaluate whether the value is within bounds

It's the same thing. It's just a matter of doing the math in different orders, and the operations are commutative so you get the same answers.[7]

The whole idea of using control charts is to find a model that predicts your data well enough that the residuals are Gaussian, so you can use three-sigma or similar techniques. This is a useful framework, and if you can make it work, a lot of your work is already done for you.

Sometimes people assume that any old model automatically guarantees Gaussian residuals. It doesn't; you need to find the *right* model, and check the results to be sure. But even if the residuals aren't Gaussian, in fact, a lot of models can be made to predict the data well enough that the residuals are very small, so you can still get excellent results.

Sample Distribution Versus Population Distribution

The second mistake we illustrated is not understanding the difference between sample and population statistics. When you work with statistics you need to know whether you're evaluating characteristics of the sample of data you have, or trying to use the sample to infer something about the larger population of data (which you don't have). It's usually the latter, by the way.

We made a mistake when we plotted the histogram of the sample and said that it doesn't look Gaussian. That sample is going to have randomness and will not look exactly the same as the full population from which it was drawn. "Is the sample Gaussian" is not the right question to ask. The right question is, loosely stated, "how likely is it that this sample came from a Gaussian population?" This is a standard statistical question, so we won't show how to find the answer here. The main thing is to be aware of the difference.

Nearly every statistical tool has techniques to try to infer the characteristics of the population, based on a sample.

7 There may be advantages to the first method when dealing with large floating-point values, though; it can help you avoid floating-point math errors.

As an aside, there's a rumor going around that the Central Limit Theorem guarantees that samples from any population will be normally distributed, no matter what the population's distribution is. This is a misreading of the theorem, and we assure you that machine data is not automatically Gaussian just because it's obtained by sampling!

Conclusions

All anomaly detection relies on predicting an expected value or range of values for a metric, and then comparing observations to the predictions. The predictions rely on models, which can be based on theory or on empirical evidence. Models usually use historical data as inputs to derive the parameters that are used to predict the future.

We discussed SPC techniques not only because they're ubiquitous and very useful when paired with a good model (a theme we'll revisit), but because they embody a thought process that is tremendously helpful in working through all kinds of anomaly detection problems. This thought process can be applied to lots of different kinds of models, including ARIMA models.

When you model and predict some data in order to try to detect anomalies in it, you need to evaluate the quality of the results. This really means you need to measure the prediction errors—the residuals—and assess how good your model is at predicting the system's data. If you'll be using SPC to determine which observations are anomalous, you generally need to ensure that the residuals are normally distributed (Gaussian). When you do this, be sure that you don't confuse the sample distribution with the population distribution!

CHAPTER 4

Dealing with Trends and Seasonality

Trends and seasonality are two characteristics of time series metrics that break many models. In fact, they're one of two major reasons why static thresholds break (the other is because systems are all different from each other). Trends are continuous increases or decreases in a metric's value. Seasonality, on the other hand, reflects periodic (cyclical) patterns that occur in a system, usually rising above a baseline and then decreasing again. Common seasonal periods are hourly, daily, and weekly, but your systems may have a seasonal period that's much longer or even some combination of different periods.

Another way to think about the effects of seasonality and trend is that they make it important to consider whether an anomaly is *local* or *global*. A local anomaly, for example, could be a spike during an idle period. It would not register as anomalously high overall, because it is still much lower than unusually high values during busy times. A global anomaly, in contrast, would be anomalously high (or low) no matter when it occurs. The goal is to be able to detect **both** kinds of anomalies. Clearly, static thresholds can only detect global anomalies when there's seasonality or trend. Detecting local anomalies requires coping with these effects.

Many time series models, like the ARIMA family of models, have properties that handle trend. These models can also accomodate seasonality, with slight extensions.

Dealing with Trend

Trends break models because the value of a time series with a trend isn't stable, or *stationary*, over time. Using a basic, fixed control chart on a time series with an increasing trend is a bad idea because it is guaranteed to eventually exceed the upper control limit.

A trend violates a lot of simple assumptions. What's the mean of a metric that has a trend? There is no single value for the mean. Instead, the mean is actually a function with time as a parameter.

What about the distribution of values? You can visualize it using a histogram, but this is misleading. Because the values increase or decrease over time due to trend, the histogram will get wider and wider over time.

What about a simple moving average or a EWMA? A moving average should change along with the trend itself, and indeed it does. Unfortunately, this doesn't work very well, because a moving average *lags* in the presence of a trend and will be consistently above or below the typical values.

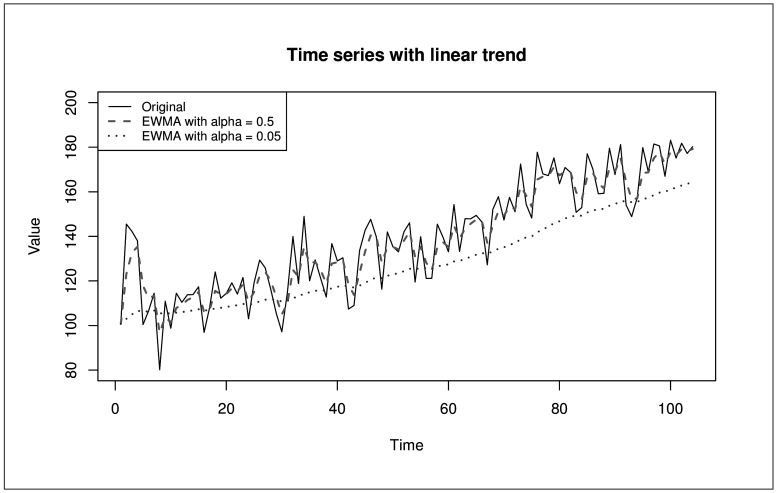


Figure 4-1. A time series with a linear trend and two exponentially weighted moving averages with different decay factors, demonstrating that they lag the data when it has a trend.

How do you deal with trend? First, it's important to understand that metrics with trends can be considered as *compositions* of other metrics. One of the components is the trend, and so the solution to deal-

ing with trend is simple: find a model that describes the trend, and subtract the trend from the metric's values! After the trend is removed, you can use the models that we've previously mentioned on the remainder.

There can be many different kinds of trend, but linear is pretty common. This means a time series increases or decreases at a constant rate. To remove a linear trend, you can simply use a *first difference*. This means you consider the differences between consecutive values of a time series rather than the raw values of the time series itself. If you remember your calculus, this is related to a derivative, and in time series it's pretty common to hear people talk about first differences as derivatives (or deltas).

Dealing with Seasonality

Seasonal time series data has cycles. These are usually obvious on observation, as shown in Figure 4-2.

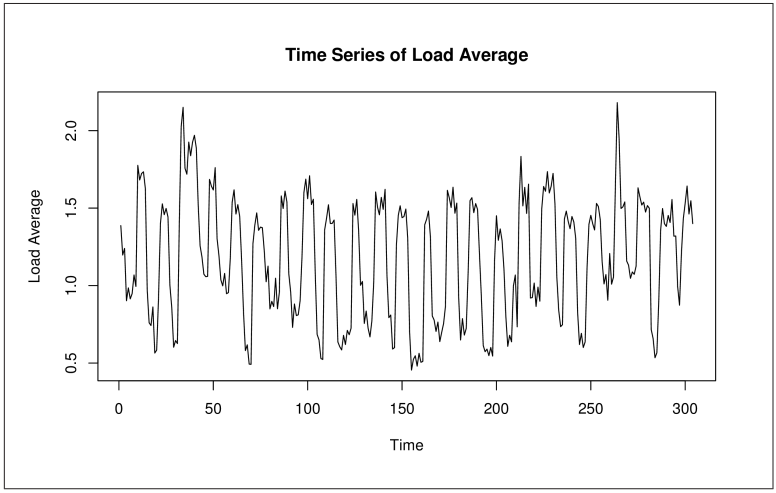


Figure 4-2. A server's load average, showing repeated cycles over time.

Seasonality has very similar effects as trend. In fact, if you "zoom into" a time series with seasonality, it really looks like trend. That's because seasonality is variable trend. Instead of increasing or decreasing at a fixed rate, a metric with seasonality increases or decreases with rates that vary with time. As you can imagine, things like EWMAs have the same issues as with linear trend. They lag behind, and in some cases it can get so bad that the EWMA is com-

pletely out of phase with the seasonal pattern. This is easy to see in Figure 4-3.

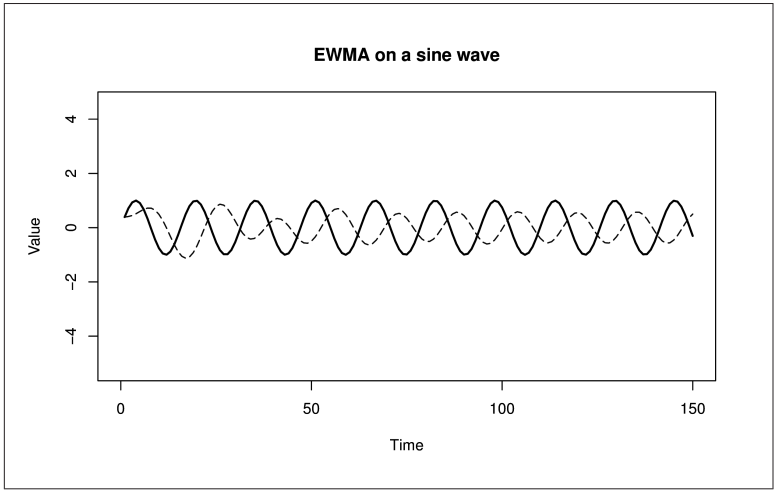


Figure 4-3. A sine wave and a EWMA of the sine wave, showing how a EWMA's lag causes it to predict the wrong thing most of the time.

Coping with seasonality is exactly the same as with trend: you need to decompose and subtract. This time, however, it's harder to do because the model of the seasonal component is much more complicated. Furthermore, there can be multiple seasonal components in a metric! For example, you can have a seasonal trend with a daily period as well as a weekly period.

Multiple Exponential Smoothing

Multiple exponential smoothing was introduced to resolve problems with using a EWMA on metrics with trend and/or seasonality. It offers an alternative approach: instead of modifying a metric to fit a model by decomposing it, it updates the model to fit the metric's local behavior. *Holt-Winters* (also known as the Holt-Winters triple exponential smoothing method) is the best known implementation of this, and it's what we're going to focus on.

A multiple exponential smoothing model typically has up to three components: an EWMA, a trend component, and a seasonal component. The trend and seasonal components are EWMAs too. For example, the trend component is simply an EWMA of the differences between consecutive points. This is the same approach we

talked about when discussing methods to deal with trend, but this time we're doing it to the model instead of the original metric. With a single EWMA, there is a single smoothing factor: α (alpha). Because there are two more EWMAs for trend and seasonality, they **also** have their own smoothing factors. Typically they're denoted as β (beta) for trend and γ (gamma) for seasonality.

Predicting the current value of a metric is similar to the previous models we've discussed, but with a slight modification. You start with the same "next = current" formula, but now you also have to add in the trend and seasonal terms. Multiple exponential smoothing usually produces much better results than naive models, in the presence of trend and seasonality.

Multiple exponential smoothing can get a little complicated to express in terms of mathematical formulas, but intuitively it isn't so bad. We recommend the "Holt-Winters seasonal method" section[1] of the *Forecasting: principles and practice* for a detailed derivation. It definitely makes things harder, though:

- You have to know the period of the seasonality beforehand. The method can't figure that out itself. If you don't get this right, your model won't be accurate and neither will your results.
- There are three EWMA smoothing parameters to pick. It becomes a delicate process to pick the right values for the parameters. Small changes in the parameters can create large changes in the predicted values. Many implementations use optimization techniques to figure out the parameters that work best on given sample data.

With that in mind, you can use multiple exponential smoothing to build SPC control charts just as we discussed in the previous chapter. The advantages and disadvantages are largely the same as we've seen before.

Potential Problems with Predicting Trend and Seasonality

In addition to being more complicated, advanced models that can handle trend and seasonality can still be problematic in some com-

1 *https://www.otexts.org/fpp/7/5*

mon situations. You can probably guess, for example, that outlying data can throw off future predictions, and that's true, depending on the parameters you use:

- An outage can throw off a model by making it predict an outage again in the next cycle, which results in a false alarm.
- Holidays often aren't in-sync with seasonality.
- There might be unusual events like Michael Jackson's death. This actually might be something you want to be alerted on, but it's clearly not a system fault or failure.
- There are annoying problems such as daylight saving time changes, especially across timezones and hemispheres.

In general, the Achilles heel of predictive models is the same thing that gives them their power: they can observe predictable behavior and predict it, but as a result they can be fooled into predicting the wrong thing. This depends on the parameters you use. Too sensitive and you get false positives; too robust and you miss them.

Another issue is that their predictive power operates at large time scales. In most systems you're likely to work with, the seasonality is hourly, daily, and/or weekly. If you're trying to predict things at higher resolutions, such as second by second, there's so much mismatch between the time scales that they're not very useful. Last week's Monday morning spike of traffic may predict this morning's spike pretty well in the abstract, but not down to the level of the second.

Fourier Transforms

It's sometimes difficult to determine the seasonality of a metric. This is especially true with metrics that are compositions of multiple seasonal components. Fortunately, there's a whole area of time series analysis that focuses on this topic: spectral analysis, which is the study of frequencies and their relative intensities. Within this field, there's a very important function called the *Fourier transform*, which decomposes any signal (like a time series) into separate frequencies. This makes use of the very interesting fact that any signal can be broken up into individual sine waves.

The Fourier transform is used in many domains such as sound processing, to decompose, manipulate, and recombine frequencies that

make up a signal. You'll often hear people mention an FFT (Fast Fourier Transform). Perfectionists will point out that they probably mean a DFT (Discrete Fourier Transform).

Using a Fourier transform, it's possible to take a very complicated time series with potentially many seasonal components, break them down into individual frequency peaks, and then subtract them from the original time series, keeping only the signal you want. Sounds great, pun intended! However, most machine data (unlike audio waves) is not really composed of strong frequencies. At least, it isn't unless you look at it over long time ranges like weeks or months. Even then, only some metrics tend to have that kind of behavior.

One example of the Fourier transform in action is Netflix's Scryer,[2] which is used to predict (or forecast) demand based on decomposed frequencies along with other methods. That said, we haven't seen Fourier transforms used practically in anomaly detection *per se*. Scryer predicts, it doesn't detect anomalies.

In our opinion, the useful things that can be done with a DFT, such as implementing low- or high-pass filters, can be done using much simpler methods. A low-pass filter can be implemented with a moving average, and a high-pass filter can be done with differencing.

Conclusions

Trend and seasonality throw monkey wrenches into lots of models, but they can often be handled fairly well by treating metrics as sums of several signals. Predicting a metric's behavior then becomes a matter of decomposing the signals into their component parts, fitting models to the components, and subtracting the predictable components from the original.

Once you've done that, you have essentially gotten rid of the non-stationary parts of the signal, and, in theory, you should be able to apply standard techniques to the stationary signal that remains.

2 *http://bit.ly/netflixscryer*

CHAPTER 5

Practical Anomaly Detection for Monitoring

Recall that one of our goals for this book is to help you actually get anomaly detection running in production and solving monitoring problems you have with your current systems.

Typical goals for adding anomaly detection probably include:

- To avoid setting or changing thresholds per server, because machines differ from each other
- To avoid modifying thresholds when servers, features, and workloads change over time
- To avoid static thresholds that throw false alerts at some times of the day or week, and miss problems at other times

In general you can probably describe these goals as "just make Nagios a little better for some checks."

Another goal might be to find all metrics that are abnormal without generating alerts, for use in diagnosing problems. We consider this to be a pretty hard problem because it is very general. You probably understand why at this point in the book. We won't focus on this goal in this chapter, although you can easily apply the discussion in this chapter to that approach on a case by case basis.

The best place to begin is often where you experience the most painful monitoring problem right now. Take a look at your alert history

or outages. What's the source of the most noise or the place where problems happen the most without an alert to notify you?

Is Anomaly Detection the Right Approach?

Not all of the alerting problems you'll find are solvable with anomaly detection. Some come from alerting on the wrong thing, period. Disks running out of space is a classic noisy alert in a lot of environments, for example. Disks reach the typical 95% threshold and then drop back below it by themselves multiple times every day, or they stay at 98% forever and that's okay, or they go from 5% to 100% in a matter of minutes. But these problems are not because you're setting the threshold to the wrong value or you need adaptive self-learning behavior. The problem is that you need to alert on predicted time to running out of space instead of alerting on fullness. This isn't about anomalies.

Another common source of monitoring noise is a non-actionable alert. If the alert signals a problem that might not even be a real issue, or has no solution or remedy, and doesn't imply the need for someone to actually do something, the solution is to delete the alert. Or at least rethink it completely. Replication delay in databases is an example we've seen a lot. If you have no good answer to someone who asks you why they should care about the metric's value (a "so what?" question) at 2 a.m., you might not have a good alert.

This implies three important things:

- You need a clear and unambiguous definition of the problem you're trying to detect. It is also best if there are direct ways to measure it happening.
- The best indicators of problems are usually directly related to one or more business goals or desired outcomes. KPIs (key performance indicators) that relate to actual impact, in other words.
- To detect a more complicated problem, you might need to relate the system behavior to a known model, such as physical laws or queueing theory or the like. If you have no model that describes how the system should behave, how do you know that your definition of a problem is correct?

If you can get close to that, you might have a pretty good shot at using anomaly detection to solve your alerting problem.

Choosing a Metric

It's important to be sure that the problem you're trying to detect has a reliable signal. It's not a good idea to use anomaly detection to alert on metrics that looked weird during that one outage that one time. One of the things we've learned by doing this ourselves is that metrics are weird constantly during normal system operation. You need to find metrics (or combinations of metrics) that are always normal during healthy system behavior, and always abnormal when systems are in trouble. Here "normal" and "abnormal" are in comparison with the local behavior of the metric, because if there were a reliable global good/bad you could just use a threshold.

It's usually best to look at a single, specific API endpoint, web page or other operation. You could detect anomalies globally—for example, over all API endpoints—but this causes two problems. First, a single small problem can be lost in the average. Second, you'll get multi-modal distributions and other complex signals. It's better to check each different kind of thing individually. If this is too much, then just pick one to begin with, such as the "add to cart" action for example.

Example metrics you could check include:

- Error rate
- Throughput
- Latency (response time), although this is tricky because latency almost always has a complex multi-modal distribution
- Concurrency, service demand, backlog, queue length, utilization, and similar metrics of load or saturation of capacity; all also usually have characteristics that are difficult to analyze with standard statistical tools unless you find an appropriate model

The Sweet Spot

Now that you've chosen a metric you want to check for anomalies, you need to analyze its behavior and see if you can do that with relatively simple tools in your toolbox. You're free to do something

more sophisticated if you want, but our suggestions in this chapter are geared toward getting useful results with low effort, not inventing something patentable.

The easiest tools to apply are, in order, exponentially weighted moving control charts[1] and Holt-Winters prediction "out of the box." You can get a lot done with these methods if you either choose the right metrics or apply them to the output of a simple model.

Both rely on the residuals being Gaussian distributed, so you need to begin by checking for that. Run an appropriate amount of the metric's history, such as a week or so, through the tool of choice to create a new series, then subtract this new series from the original to get the residuals and check the distribution of the result.

In some cases you will almost certainly not get usable results. Concurrency, for example, tends to look log-normal in our experience[2] This leads to an easy transformation: just take the log of concurrency, and try again.

Another simple technique is to look at the first difference (derivative) of the metric. Subtract each observation from the next and try again. Recall that this essentially asserts that the metric's behavior can be predicted with the equation: next=current+noise. One downside of this technique, however, is that it transforms the shape of sudden changes in the metric's value. Consider a metric that's always 500, then drops to 200 for one observation and back to 500 again. That's a single sharp downward spike. If you difference this metric, the result will be a metric that stays at 0, spikes down to -200, flips up to +200, and then back again. Differencing causes spikes to turn into zigzags, which might be more complicated to analyze.

You can also try finer or coarser time resolution, such as five minute intervals instead of second by second or minute by minute. This is a low-pass filter that removes noise, at the cost of discarding potentially useful information too.

---

1 Simple control charts also work well, but again if you can use them you can use static thresholds instead.

2 We tried to see if queuing theory predicts this, but were unable to determine whether the underlying model of any type of queue would result in a particular distribution of concurrency. In cases such as this, it's great to be able to prove that a metric should behave in a specific way, but absent a proof, as we've said, it's okay to use a result that holds even if you don't know why it does.

We previously discussed how frequent anomalies are when detected with standard three sigma math. Because of this, it is pretty common for people to suppress anomalies unless they occur at a given frequency within an interval. Say, five or more anomalies at one minute resolution in a 30 minute period.

Finally, you can look at simple models such as linear relationships between multiple metrics. For example, if a database's query throughput is closely correlated to the server's CPU utilization, you can try to divide utilization by throughput and look for anomalies in the result. This is a simple way to check whether CPU utilization is abnormal relative to the work the server is being asked to do.

Servers and their metrics are as different as fingerprints. One of these suggestions might work for you and completely fail for your friend.

All of these techniques are trial and error ways of checking whether there is some context or predictability that can describe normal behavior for the metric. We're trying to relate current observations to nearby ones, past behavior at a seasonal interval, other metrics, or combinations of these.

Provided that you can find a usable combination of metrics and methods or models, you can just sprinkle three sigmas on it and you're done. To repeat, the key is to be sure that:

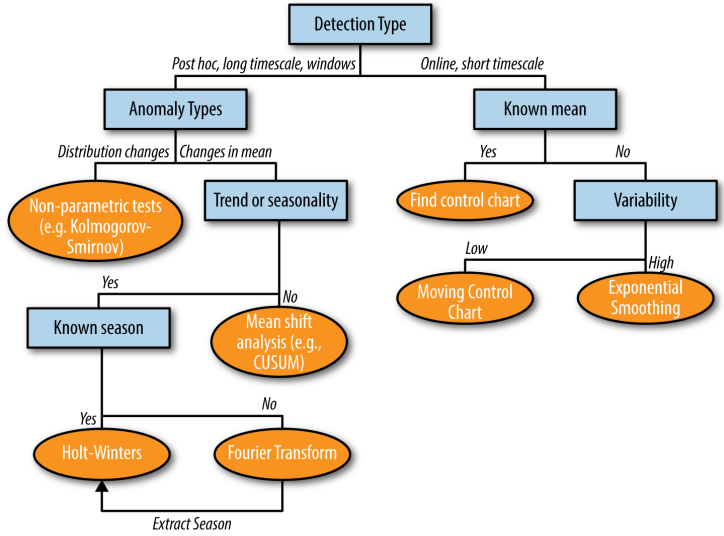
- The metric reliably indicates a condition that matters
- The model produces Gaussian distributed residuals, so the three sigma method actually works
- Out of bounds values pass the "so what?" test, i.e., abnormal is bad

A final note to add to the last point. Do you care about abnormal values in general, or only abnormally large or small values? Abnormally small error rates probably aren't a problem. Make sure you don't blindly use three sigma tests if you only care about half of the results.

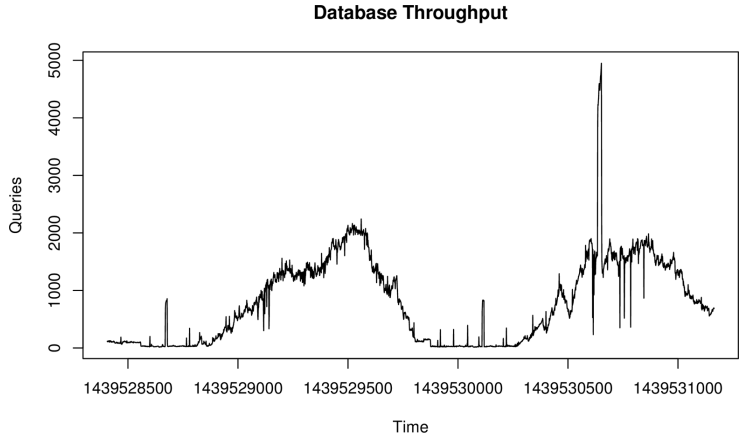
A Worked Example

In this section, we'll go through a practical example demonstrating some of the techniques we've covered so far. We're going to use a database's throughput (queries per second) as our metric.

To summarize our thought process, we've created the following flowchart that you can use as a decision tree. This is an extreme simplification, and a little bit biased towards our own experiences, but it should be enough to get you started and orient yourself in the space of anomaly detection techniques.



We'll use this decision tree to pick a suitable anomaly detection technique for the example. On that note, let's take a look at the data we're going to be dealing with. The following plot shows the throughput for a period of approximately 46 hours.

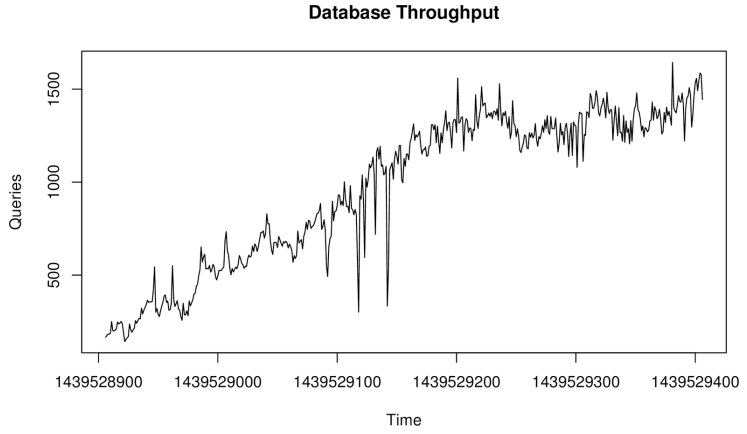


First, observe how odd this metric is on this server. On a database server like this, throughput is the best metric to represent the work being done. You can see that it would be difficult to describe what normal throughput would look like on this server because its workload is dynamic. Sometimes it's doing nothing, and sometimes it's doing a lot. Is the giant spike near the right an anomaly? Probably. Is it "bad?" Well… it's doing the work it's been asked to do, isn't it? So is it an anomaly or not?

You can't just ask "where are the anomalies?" on this data set. That's not a well-posed question, not even for a human. We need more context to form a better question.

One more specific question that we could answer about this dataset with anomaly detection is "when does throughput drop significantly low for short periods of time?" This simple behavior on a metric can have many causes, such as resource contention or locking. We can answer this question by finding *local* anomalies in the metric. This is more doable.

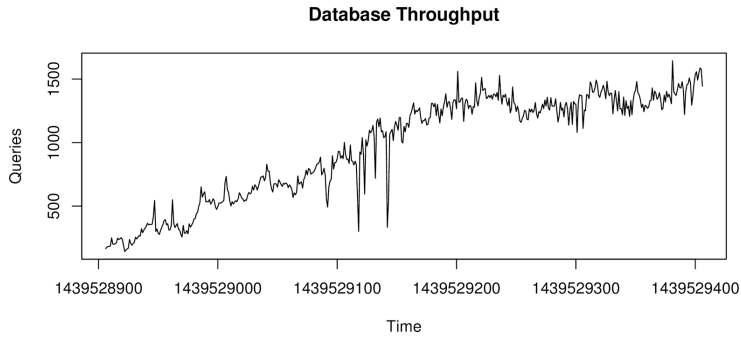
We'll zoom in to a more interesting section of the data set that shows the types of anomalies that we're interested in detecting. This smaller section is over an 8-hour interval.

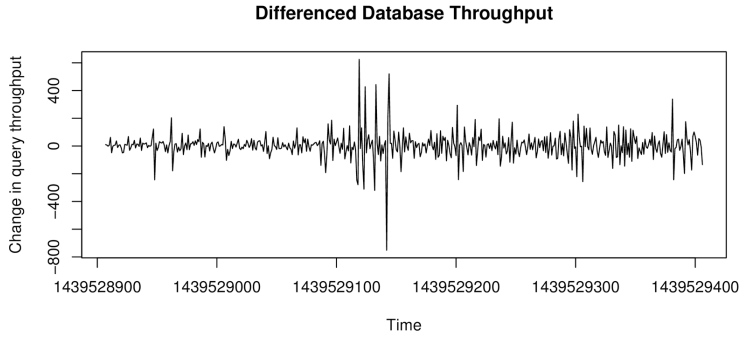


In this plot, you'll notice that throughput increases with an approximately linear trend, with a slight drop around the two-thirds mark. In the middle of the plot, you can see large drops in throughput. These drops, which happen on a short timescale, are the anomalies that we've decided to try detecting.

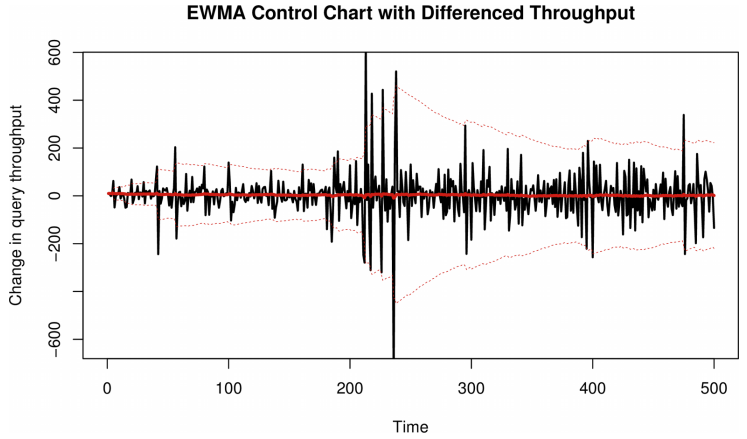
Now that we know the scope of our anomaly detection problem, let's try to use the decision tree to determine what kinds of methods might be useful. For "Detection Time," we know we're interested in a short timescale method so we'll follow the path on the right. With a linear trend, we don't have a fixed mean, so we'll follow the "no" path for "Known mean." Now we're down to "Variability." The metric is highly variable, and exponential smoothing tends to generate better results in situations like this.

If our data set didn't have trend, we may have ended up with a fixed control chart method, which is simpler than using exponential smoothing. Previously, we mentioned that trend is something that can be removed with a first difference or derivative. Let's try that out.





After differencing, it looks like we eliminated the trend. Great! Now we have a known, fixed mean at 0 and we can use an ordinary control chart. Because the spread changes over time and isn't something we know *a priori*, we'll use an EWMA control chart, which uses a EWMA to draw the limits based mostly on recent data values.



Here's what the histogram of residuals with a Gaussian distribution curve looks like.

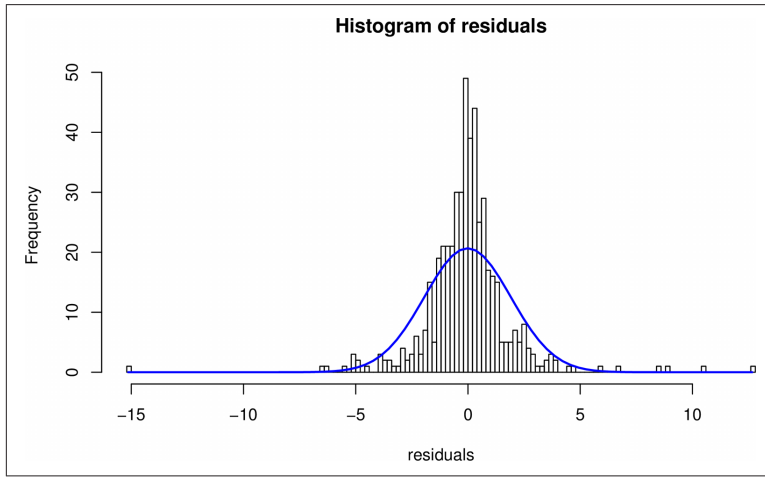
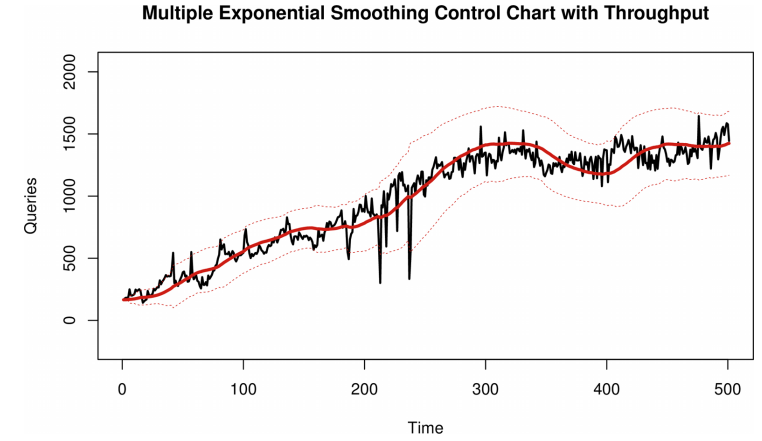


Figure 5-1. Histogram of residuals from the control chart on differenced data.

Where would we signal anomalies? It's not so easy to tell when the metric has short, significant drops. When throughput drops abnormally low and then returns to its original value, the differenced metric has zigzags, which are complicated to inspect. So although differencing takes care of the linear trend, it makes prediction a bit harder.

Perhaps, instead of differencing, we should try an alternative: multiple exponential smoothing on the original metric. This should cope well with the trend and avoid transforming the metric as differencing does.



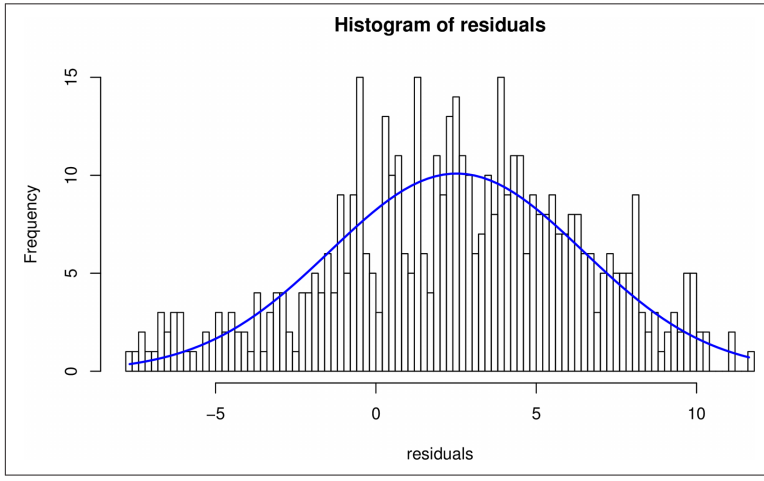


Figure 5-2. Histogram of residuals from the exponential smoothing control chart on the raw data.

Now it's easy to see throughput drops when the metric falls below the lower control line. It's much easier to interpret, visually at least.

Multiple exponential smoothing is a little more complicated, but produces much better results in this example. It has a trend compo-

nent built into its model, so you don't have to do anything special to handle metrics with trend; it trains itself, so to speak, on the actual data it sees.

This is a tradeoff. You can either transform your data to use a better model, which may hurt interpretability, or try to develop a more complicated model.

It's worth noting that based on Figure 5-1 and Figure 5-2, neither method seems to produce perfectly Gaussian residuals. This is not a major issue. At least with the exponential smoothing control chart, we're still able to reasonably predict and detect the anomalies we're interested in.

Keep in mind that this is a narrowly focused example that only demonstrates one path in our decision tree. We started with a very specific set of requirements (short timescale with significant spikes) that made our final solution work, but it won't work for everything. If we wanted to look at a larger time scale, like the full data set, we'd have to look at other techniques.

Conclusions

This chapter demonstrates relatively simple techniques that you can probably apply to your own problems with the tools you have at hand already, such as RRDTool, simple scripts, and Graphite. Maybe a Redis instance or something if you really want to get fancy.

The idea here is to get as much done with as little fuss as possible. We're not trying to be data scientists, we're just trying to improve on a Nagios threshold check.

What makes this work? It's mostly about choosing the right battle, to tell the truth. Throughput is about as simple a KPI as you can choose for a database server. Then we visualized our results and picked the simplest thing that could possibly work.

Your mileage, needless to say, will vary.

CHAPTER 6

The Broader Landscape

As we've mentioned before, there is an extremely broad set of topics and techniques that fall into anomaly detection. In this chapter, we'll discuss a few, as well as some popular tools that might be useful. Keep in mind that nothing works perfectly out-of-the-box for all situations. Treat the topics in this chapter as hints for further research to do on your own.

When considering the methods in this chapter, we suggest that you try to ask, "what assumptions does this make?" and "how can I assess the meaning and trustworthiness of the results?"

Shape Catalogs

In the book *A New Look at Anomaly Detection* by Dunning and Friedman, the authors write about a technique that uses shape catalogs. The gist of this technique is as follows. First, you have to start with a sample data set that represents the time series of a metric without any anomalies. You break this data set up into smaller windows, using a window function to mask out all but a specific region, and catalog the resulting shapes. The assumption being made is that any non-anomalous observation of this time series can be reconstructed by rearranging elements from this shape catalog. Anything that doesn't match up to a reasonable extent is then considered to be an anomaly.

This is nice, but most machine data doesn't really behave like an EKG chart in our experience. At least, not on a small time scale.

Most machine data is much noisier than this on the second-to-second basis.

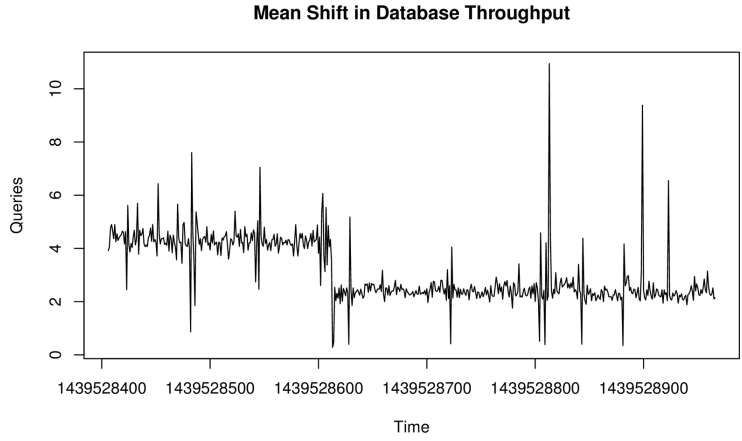
Mean Shift Analysis

For most of the book, we've discussed anomaly detection methods that try to detect large, sudden spikes or dips in a metric. Anomalies have many shapes and sizes, and they're definitely not limited to these short-term aberrations. Some anomalies manifest themselves as slow, yet significant, departures from some usual average. These are called *mean shifts*, and they represent fundamental changes to the model's parameters.[1] From this we can infer that the system's state has changed dramatically.

One popular technique is known as *CUSUM*, which stands for cumulative sum control chart. The CUSUM technique is a modification to the familiar control chart that focuses on small, gradual changes in a metric rather than large deviations from a mean.

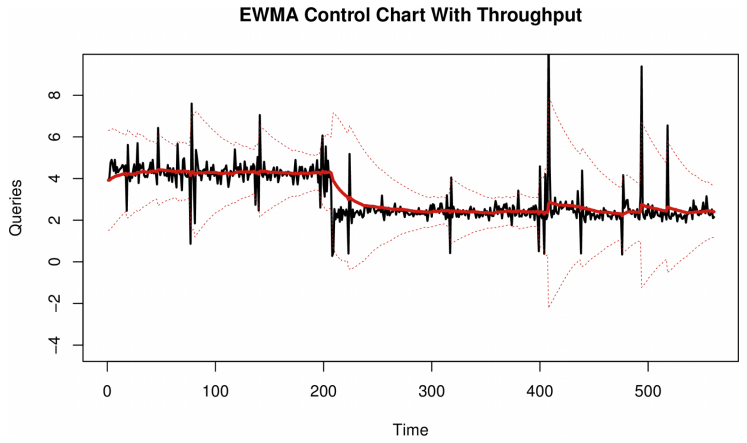
The CUSUM technique assumes that individual values of a metric are evenly scattered across the mean. Too many on one side or the other is a hint that perhaps the mean has changed, or *shifted*, by some significant amount.

The following plot shows throughput on a database with a mean shift.

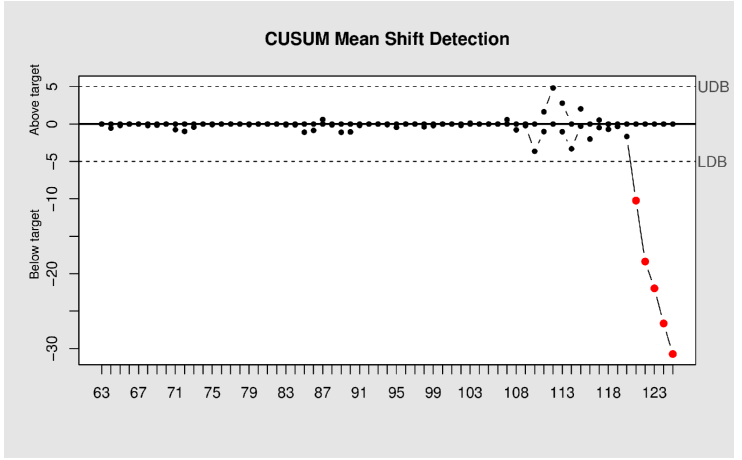


1 Mean-shift analysis is not a single technique, but rather a family. There's a Wikipedia page on the topic, where you can learn more: *http://bit.ly/mean_shift*

We could apply a EWMA control chart to this data set like in the worked example. Here's what it looks like.



This control chart definitely could detect the mean shift since the metric falls underneath the lower control line, but that happens often with this highly variable data set with lots of spikes! An EWMA control chart is great for detecting spikes, but not mean shifts. Let's try out CUSUM. In this image we'll show only the first portion of the data for clarity:

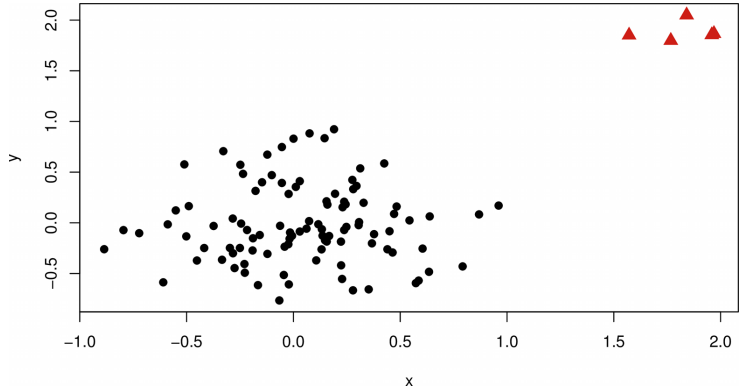


Much better! You can see that the CUSUM chart detected the mean shift where the points drop below the lower threshold.

Clustering

Not all anomaly detection is based on time series of metrics. Clustering, or *cluster analysis* is one way of grouping elements together to try to find the odd ones out. Netflix has written about their anomaly detection methods based on cluster analysis.[2] They apply cluster analysis techniques on server clusters to identify anomalous, misbehaving, or underperforming servers.

K-Means clustering is a common algorithm that's fairly simple to implement. Here's an example:



Non-Parametric Analysis

Not all anomaly detection techniques need models to draw useful conclusions about metrics. Some avoid models altogether! These are called *non-parametric* anomaly detection methods, and use theory from a larger field called non-parametric statistics.

The Kolmogorov-Smirnov test is one non-parametric method that has gained popularity in the monitoring community. It tests for changes in the distributions of two samples. An example of a type of question that it can answer is, "is the distribution of CPU usage this week significantly different from last week?" Your time intervals don't necessarily have to be as long as a week, of course.

We once learned an interesting lesson while trying to solve a sticky problem with a non-Gaussian distribution of values. We wanted to

2 "Tracking down the Villains: Outlier Detection at Netflix"

figure out how unlikely it was for us to see a particular value. We decided to keep a histogram of all the values we'd seen and compute the percentiles of each value as we saw it. If a value fell above the 99.9th percentile, we reasoned, then we could consider it to be a one-in-a-thousand occurrence.

Not so! For several reasons, primarily that we were computing our percentiles from the sample, and trying to infer the probability of that value existing in the population. You can see the fallacy instantly, as we did, if you just postulate the observation of a value much higher than we'd previously seen. How unlikely is it that we saw that value? Aside from the brain-hurting existential questions, there's the obvious implication that we'd need to know the distribution of the population in order to answer that.

In general, these non-parametric methods that work by comparing the distribution (usually via histograms) across sets of values can't be used online as each value arrives. That's because it's difficult to compare single values (the current observation) to a distribution of a set of values.

Grubbs' Test and ESD

The Grubbs' Test is used to test whether or not a set of data contains an outlier. This set is assumed to follow an approximately Gaussian distribution. Here's the general procedure for the test, assuming you have an appropriate data set *D*.

1. Calculate the sample mean. Let's call this μ.
2. Calculate the sample standard deviation. Let's call this *s*.
3. For each element *i* in *D*
4. Calculate abs($i - \mu$) / *s*. This is the number of standard deviations away *i* is from the sample mean.
5. Now you have the distance from the mean for each element in *D*. Take the maximum.
6. Now you have the maximum distance (in standard deviations) any single element is away from the mean. This is the test statistic.
7. Compare this to the critical value. The critical value, which is just a threshold, is calculated from some significance level, i.e. some coverage proportion that you want. In other words, if you

want to set the threshold for outliers to be 95% of the values from the population, you can calculate that threshold using a formula. The critical value in this case ends up being in units of standard deviations. If the value you calculated in step 5 is larger than the threshold, then you have statistically significant evidence that you have an outlier.

The Grubbs' test can tell you whether or not you have a single outlier in a data set. It should be straightforward to figure out which element is the outlier.

The ESD test is a generalization that can test whether or not you have up to *r* outliers. It can answer the question, "How many outliers does the data set contain?" The principle is the same—it's looking at the standard deviations of individual elements. The process is more delicate than that, because if you have two outliers, they'll interfere with the sample mean and standard deviation, so you have to remove them after each iteration.

Now, how is this useful with time series? You need to have an approximately Gaussian (normal) distributed data set to begin with. Recall that most time series models can be decomposed into separate components, and usually there's only one random variable. If you can fit a model and subtract it away, you'll end up with that random variable. This is exactly what Twitter's BreakoutDetection[3] R package does. Most of their work consists of the very difficult problem of automatically fitting a model that can be subtracted out of a time series. After that, it's just an ESD test.

This is something we would consider to fall into the "long term" anomaly detection category, because it's not something you can do online as new values are observed.

For more details, refer to the "Grubbs' Test for Outliers" page in the *Engineering Statistics Handbook*.[4]

Machine Learning

Machine learning is a meta-technique that you can layer on top of other techniques. It primarily involves the ability for computers to

3 *https://github.com/twitter/BreakoutDetection*

4 *http://bit.ly/grubbstest*

predict or find structure in data without having explicit instructions to do so. "Machine learning" has more or less become a blanket term these days in conversational use, but it's based on well-researched theory and techniques. Although some of the techniques have been around for decades, they've gained significant popularity in recent times due to an increase in overall data volume and computational power, which makes some algorithms more feasible to run. Machine learning itself is split into two distinct categories: unsupervised and supervised.

Supervised machine learning involves building a training set of observed data with labeled output that indicates the right answers. Thes answers are used to train a model or algorithm, and then the trained behavior can predict the unknown output of a new set of data. The term supervised refers to the use of the known, correct output of the training data to optimize the model such that it achieves the lowest error rate possible.

Unsupervised machine learning, unlike its supervised counterpart, does not try to figure out how to get the right answers. Instead, the primary goal of unsupervised machine learning algorithms is to find patterns in a data set. Cluster analysis is a primary component of unsupervised machine, and one method used is K-means clustering.

Ensembles and Consensus

There's never a one-size-fits-all solution to anomaly detection. Instead, some choose to combine multiple techniques into a group, or *ensemble*. Each element of the ensemble casts a vote for the data it sees, which indicates whether or not an anomaly was detected. These votes are then used to form a *consensus*, or overall decision of whether or not an anomaly is detected. The general idea behind this approach is that while individual models or methods may not always be right, combining multiple approaches may offer better results on average.

Filters to Control False Positives

Anomaly detection methods and models don't have enough context themselves to know if a system is actually anomalous or not. It's your task to utilize them for that purpose. On the flip side, you also need to know when to *not* rely on your anomaly detection frame-

work. When a system or process is highly unstable, it becomes extremely difficult for models to work well. We highly recommend implementing filters to reduce the number of false positives. Some of the filters we've used include:

- Instead of sending an alert when an anomaly is detected, send an alert when N anomalies are detected within an interval of time.
- Suppress anomalies when systems appear to be too unstable to determine any kind of normal behavior. For example, the variance-to-mean ratio (index of dispersion), or another dimensionless metric, can be used to indicate whether a system's behavior is stable.
- If a system violates a threshold and you trigger an anomaly or send an alert, don't allow another one to be sent unless the system resets back to normal first. This can be implemented by having a reset threshold, below which the metrics of interest must dip before they can trigger above the upper threshold again.

Filters don't have to be complicated. Sometimes it's much simpler and more efficient to just simply ignore metrics that are likely to cause alerting nuisances. Ruxit recently published a blog post titled "Parameterized anomaly detection settings"[5] in which they describe their anomaly detection settings. Although they don't call it a "filter," one of their settings disables anomaly detection for low traffic applications and services to avoid unnecessary alerts.

Tools

You generally don't have to implement an entire anomaly detection framework yourself. As a significant component of monitoring, anomaly detection has been the focus of many monitoring projects and companies which have implemented many of the things we've discussed in this book.

5 *http://bit.ly/ruxitblog*

Graphite and RRDTool

Graphite and RRDTool are popular time series storage and plotting libraries that have been around for many years. Both include Holt-Winters forecasting, which can be used to detect anomalous observations in incoming time series metrics. Some monitoring platforms such as Ganglia, which is built on RRDTool, also have this functionality. RRDTool itself has a generic anomaly detection algorithm built in, although we're not aware of anyone using it (unsurprisingly).

Etsy's Kale Stack

Etsy's Skyline project, which is part of the Kale stack, includes a variety of different algorithms used for anomaly detection. For example, it has implementations of the following, among others:

- Control charts
- Histograms
- Kolmogorov-Smirnov test

It uses an ensemble technique to detect anomalies. It's important to keep in mind that not all algorithms are appropriate for every data set.

R Packages

There are plenty of R packages available for many anomaly detection methods such as forecasting and machine learning. The downside is that many are quite simple. They're often little more than reference implementations that were not intended for monitoring systems, so it may be difficult to implement them into your own stack.

Twitter's anomaly detection R package,[6] on the other hand, actually runs in their production monitoring system. Their package uses time series decomposition techniques to detect point anomalies in a data set.

Commercial and Cloud Tools

Instead of implementing or incorporating anomaly detection methods and tools into your own monitoring infrastructure, you may be

6 *https://github.com/twitter/BreakoutDetection*

more interested in using a cloud-based anomaly detection service. For example, companies like Ruxit, VividCortex, AppDynamics, and other companies in the Application Performance Management (APM) space offer anomaly detection services of some kind, often under the rubric of "baselining" or something similar.

The benefits of using a cloud service are that it's often much easier to use and configure, and providers usually have rich integration into notification and alerting systems. Anomaly detection services might also offer better diagnostic tools than those you'll build yourself, especially if they can provide contextual information. On the other hand, one downside of cloud-based services is that because it's difficult to build a solution that works for everything, it may not work as well as something you could build yourself.

APPENDIX A

Appendix

Code

Control Chart Windows

Moving Window

```
function fixedWindow(size) {
  this.name = 'window';
  this.ready = false;
  this.points = [];
  this.total = 0;
  this.sos = 0;

  this.push = function(newValue) {
    if (this.points.length == size) {
      var removed = this.points.shift();
      this.total -= removed;
      this.sos -= removed*removed;
    }
    this.total += newValue;
    this.sos += newValue*newValue;
    this.points.push(newValue);
    this.ready = (this.points.length == size);
  }

  this.mean = function() {
    if (this.points.length == 0) {
      return 0;
    }
    return this.total / this.points.length;
  }
```

```
  this.stddev = function() {
    var mean = this.mean();
    return Math.sqrt(this.sos/this.points.length - mean*mean);
  }
}

var window = new fixedWindow(5);
window.push(1);
window.push(5);
window.push(9);
console.log(window);
console.log(window.mean());
console.log(window.stddev()*3);
```

EWMA Window

```
function movingAverage(alpha) {
  this.name = 'ewma';
  this.ready = true;

  function ma() {
    this.value = NaN;
    this.push = function(newValue) {
      if (isNaN(this.value)) {
        this.value = newValue;
        ready = true;
        return;
      }
      this.value = alpha*newValue + (1 - alpha)*this.value;
    };
  }

  this.MA = new ma(alpha);
  this.sosMA = new ma(alpha);

  this.push = function(newValue) {
    this.MA.push(newValue);
    this.sosMA.push(newValue*newValue);
  };

  this.mean = function() {
    return this.MA.value;
  };

  this.stddev = function() {
                  return     Math.sqrt(this.sosMA.value     -
this.mean()*this.mean());
  };
}
```

```
var ma = new movingAverage(0.5);
ma.push(1);
ma.push(5);
ma.push(9);
console.log(ma);
console.log(ma.mean());
console.log(ma.stddev()*3);
```

Window Function

```
function kernelSmoothing(weights) {
  this.name = 'kernel';
  this.ready = false;
  this.points = [];
  this.lag = (weights.length-1)/2;

  this.push = function(newValue) {
    if (this.points.length == weights.length) {
      var removed = this.points.shift();
    }
    this.points.push(newValue);
    this.ready = (this.points.length == weights.length);
  }

  this.mean = function() {
    var total = 0;
    for (var i = 0; i < weights.length; i++) {
      total += weights[i]*this.points[i];
    }
    return total;
  };

  this.stddev = function() {
    var mean = this.mean();
    var sos = 0;
    for (var i = 0; i < weights.length; i++) {
      sos += weights[i]*this.points[i]*this.points[i];
    }
    return Math.sqrt(sos - mean*mean);
  };
}

var ksmooth = new kernelSmoothing([0.3333, 0.3333, 0.3333]);
ksmooth.push(1);
ksmooth.push(5);
ksmooth.push(9);
console.log(ksmooth);
console.log(ksmooth.mean());
console.log(ksmooth.stddev()*3);
```

About the Authors

Baron Schwartz is the founder and CEO of VividCortex, a next-generation database monitoring solution. He speaks widely on the topics of database performance, scalability, and open source. He is the author of O'Reilly's bestselling book *High Performance MySQL*, and many open-source tools for MySQL administration. He's also an Oracle ACE and frequent participant in the PostgreSQL community.

Preetam Jinka is an engineer at VividCortex and an undergraduate student at the University of Virginia, where he studies statistics and time series.

Acknowledgments

We'd like to thank George Michie, who contributed some content to this book as well as helping us to clarify and keep things at an appropriate level of detail.