TIME SERIES ANOMALY DETECTION

A practical guide to detecting anomalies in time series using artificial intelligence concepts

Kamran Vatanabadi, April 2017
Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1. Time Series</td>
<td>4</td>
</tr>
<tr>
<td>2. Anomaly</td>
<td>9</td>
</tr>
<tr>
<td>3. System</td>
<td>13</td>
</tr>
<tr>
<td>4. Data</td>
<td>16</td>
</tr>
<tr>
<td>5. Introduction to Time Series Anomaly Detection</td>
<td>21</td>
</tr>
<tr>
<td>6. Using Data Distribution</td>
<td>28</td>
</tr>
<tr>
<td>7. Clustering and Classification</td>
<td>34</td>
</tr>
<tr>
<td>8. Temporal Patterns</td>
<td>37</td>
</tr>
<tr>
<td>9. What is learning?</td>
<td>41</td>
</tr>
<tr>
<td>10. Real time Anomaly Detection</td>
<td>45</td>
</tr>
<tr>
<td>11. Window processing</td>
<td>50</td>
</tr>
<tr>
<td>12. Anomaly Index</td>
<td>54</td>
</tr>
<tr>
<td>13. Coarse Coding</td>
<td>57</td>
</tr>
<tr>
<td>14. Distribution and Behavior Comparison</td>
<td>60</td>
</tr>
<tr>
<td>15. Time Series Anomaly Detection Algorithm</td>
<td>65</td>
</tr>
<tr>
<td>16. Basics of Time Series Anomaly Detection Engine</td>
<td>79</td>
</tr>
<tr>
<td>18. Software structure</td>
<td>108</td>
</tr>
</tbody>
</table>
TIME SERIES ANOMALY DETECTION
A practical guide to detecting anomalies in time series using AI concepts

Introduction
This white paper is about finding anomalies in time series, which we encounter in almost every system. I usually keep notes when I work on projects, and this paper is based on my experiences and the notes I took while working on anomaly detection systems.

There are hundreds of papers containing different methods and algorithms for finding anomalies in time series. However, as someone who has worked on many of these algorithms, I must say that there is no single algorithm that can catch all anomalies. Instead, we must choose combinations of algorithms to achieve particular goals.

I have seen many software engineers and programmers struggling with understanding algorithms in papers or having problems with using them in their applications. This work provides a general overview of using artificial intelligence methods in any application, not just in time series, so that everyone with intermediate knowledge of programming, mathematics, and statistics can understand them and use them in software applications.

I have used Java as the programming language to implement the algorithms, and I explain things as simply as possible to make the algorithms easier to implement. I have included examples mostly from computer networking, but also from different businesses and industries.

Chapters 1 to 14 describe the theoretical basis and practice behind anomaly detection algorithms. In each of these short chapters, I discuss a single subject with examples. I have used words as well as formulas to help keep my explanations as clear and simple as possible. In the last four chapters, we focus on building a simple but functional anomaly detection engine.

If you have already studied AI, you will find most of the chapter topics familiar, because the primary goal of this paper is describing the concepts used in AI and building a statistical machine learning anomaly detection engine. You will not need a third-party library to build your software because this paper will help you understand and implement your own learning algorithm and anomaly detection engine.

Finally, this paper is like an open lab report written in a conversational style. Feel free to contact me at kamran@sleptons.com if you have any questions.
1. Time Series

1.1. What is a time series?
A series is a series in which the items are changing over the time. This paper focuses on numeric time series. For example, you may want to monitor and analyze your diet by measuring how many calories you consume every day. To do that, every day you would write down a new item of information containing a date and how many calories you consumed. After three months, you would have around 90 items or data points. These points can be drawn on an x-y coordinate system with time on the x-axis and calories on the y-axis. The trajectory on this graph may make some patterns obvious. For example, it might show that, during weekends, you consume more calories than on working days, or that the calories you consume per day are the lowest on Mondays.

\[
\text{Consumed Calories Time Series : } (d_1, c_1), (d_2, c_2), \ldots
\]

1-1- Calories consumed per day as a series containing dates and caloric values.

1.2. Sensor
Time series are everywhere. Collecting time series data requires only something that measures the value of the variable of interest and sends it on to be stored or processed on a periodic basis. A device that interacts with the system, measures one or more variables, and gives you the result is called a sensor.

For example, to keep track of the temperature of a room, all you need is a digital thermometer controlled by a computer. It is easy enough to write an application that sends a message to the digital thermometer every so often to get it to return the temperature of the room. If the temperature is taken every minute, every day 1440\(^1\) data points are returned. These points show the room temperature with a resolution of one minute.

Sometimes a physical sensor is not needed to collect the value of the variable. For example, a database administrator (DBA) of a company might like to analyze the load on a database. All the DBA needs to do is to write a simple script that gets the CPU usage of the database server every minute and stores it somewhere for later processing using an SNMP\(^2\) call. Network administrators can do the same to collect incoming and outgoing traffic usage too, such as the series in 1-2.

---

\(^1\) (1440 samples per day) = (60 samples per hour) x (24 hours)
\(^2\) SNMP is the abbreviation for “Simple Network Management Protocol”. Plenty of useful measures of device activity are easy to obtain by running the “snmpwalk” command on almost any operating system or networked device.
Network Traffic Time Series: \((m_1, [i_1, o_1]), (m_2, [i_2, o_2]), ...\)

1-2- Each data point in a network traffic usage time series contains a time measurement (minute) as well as measures of incoming and outgoing traffic.

So, no matter what the industry, if the required sensors are available, it is possible to measure any parameters of a system periodically and collect time series data from it.

1.3. Regular vs. Irregular\(^3\)
Until now, we have described time series as a sequence of numeric values of a measure or measures. If the sampling events occur at equal intervals in time, the result is called a regular time series. Otherwise, it is called irregular. For example, the examples we used above of calories eaten per day or network traffic usage per minute are both regular time series, whereas a time series that tracks credit card usage would be an irregular time series because there is not necessarily any consistent pattern to when people make purchases. The general form of a time series applies to both regular and irregular cases\(^4\) is like 1-3.

\[
\text{Time Series} : (t_1, v_1), (t_2, v_2), ...
\]

\[
\text{if (Time Series is regular) then } \{ t_i - t_{i-1} = \text{constant} \}
\]

1-3- In a regular time series, points are collected in a fixed time space.

1.4. Univariate vs. Multivariate
The temperature example used above is a univariate time series in which each item is a pair of variables (time, number) while the network traffic is a multivariate time series because each item is a pair of variables (time, vector), with the vector containing values for both incoming and outgoing traffic.

\[
\text{Time Series} : (t_1, V_1), (t_2, V_2), ...
\]

\[
V_i = [v_{i1}, v_{i2}, ...]
\]

1-4- In multivariate time series, each point contains a time value and a vector of other data.

\(^3\) Sometimes regular time series are called evenly spaced time series, and irregular ones are called unevenly spaced time series.

\(^4\) It is possible to convert an irregular time series to regular by using a longer timeframe. In the credit card example, for instance, if credit card activity is tracked per day rather than per transaction it can work as regular time series. It is also possible to produce null data whenever there is no updated or new sample data.
In this paper, we focus on and describe regular univariate time series, though we do give some hints on how to analyze and process multivariate or irregular time series. So, from now on, unless we strictly emphasize that we are discussing multivariate or irregular time series, when we mention “time series,” we mean univariate regular time series.

1.5. Periodicity and Cycles

If there is a repeating pattern over some time period in a time series, the time series can be called periodic. Most of the measures of the resources that people use every day are periodic because of the nature of human life. For example, road traffic or electricity usage in a city, or internet usage in a company, all are examples of periodic patterns. In these examples of time series, usage goes up in the morning and falls in the afternoon or evening, and then the same pattern repeats the next day.

Weekly patterns are also easy to find because the usage of resources usually decreases during weekends. And yearly patterns are also common because the seasonal changes in weather affect how we live in different months, and there are also holidays and celebrations in some months. So periodicity can affect the standard behavior of a time series over different time scales. We will talk a bit more about periodicity later when we introduce methods of decomposition of time series.

We must mention that sometimes time series have cycles that repeat over time. Cycles are different from periodic patterns because they do not have fixed periods. For example, consider an economic or financial index that shows purchasing power. This index could have some fluctuation over an extended period because it depends on the overall economic condition of the world, which is not constant. We know if that index hits its maximum it will go down after a while, but there is no fixed period either for hitting the maximum or for returning to its average value because the situation depends on so many independent variables.

1.6. Trends

Trends in a time series are behavior of the time series over a period longer than its longest period. If the longest period of a time series is a year, for example, then, if the average of the series is taken for at least one or even two years, the result will show the trend. For example, consider the average price of a house in a city from 1900 to 2017. The prices may have had some ups in the warmer months and downs in the cooler months every year, however, because of inflation, the average price of a house between 1900 and 2017 has had a positive slope or upward trend. Note again, that to spot a trend, the duration examined must be greater than the longest period, otherwise, periodic changes can mask a trend.
1.7. Irregularity

Once the trend and periodic behaviour are removed from a time series, the remaining part is called the residual or the irregular component. This part fluctuates almost randomly over the period analyzed.

---

The better you define and extract trend and periodic behaviour in a time series, the less regularity will be left in the residual.

---

The residual can be considered to be something like noise or a random variable, although sometimes there might be a reason for this component’s behaviour. Consider a distributed denial of service (DDOS) cyberattack on a server. The time series graph of the server’s traffic volume would show some unusual behavior at that point, and if its time series’ usual trend and periodic behaviour were removed, the attack’s impact would be easy to see in its residuals—the attack reduces the randomness of the residual, or decreases its entropy.

1.8. Example

Look at the 200 sample points of the time series shown in Figure 1-5. It is evident that it has some periodicity, an upward trend and a residual.

---

1-5- Sample time series, showing the trend, periodicity, and residual components.
We will introduce methods to extract these components in next few chapters. For now, just look at Figure 1-6 and try to figure out how to decompose this time series into these three parts\(^5\). We must mention that not all time series exhibit clear periodicity or trends as smooth as the one in this example, but the decomposition methods are usually powerful enough to extract these components.

\[\text{Components of the time series}\]

1-6- The components of the time series in Figure 1-5: residual (bottom, irregular line), trend (smooth line with positive slope) and periodicity (wave-like line).

\(^5\) You can consider the positive offset of seasonality (here about 15 units) for the trend so that the seasonality fluctuates around zero and the trend starts at a higher level. It just depends on your decomposition algorithm.
2. Anomaly

2.1. What is an Anomaly?

An anomaly in general and in time series context, is something that happens with a very high deviation from what we expect. The reason for using the words “high deviation” is that we want to emphasize the difference between an anomaly and something unusual. This definition lets us use probability theory to model experiments and recognize anomalies.

Example 1: One day, you wake up in the morning and look through the window and see the sky is green. This is something you never expected, so it is an anomaly.

Example 2: You flip a fair coin, and it lands on its edge instead of showing heads or tails. The first time this happens, you might think that it was an accident, but then, in subsequent experiments, again and again, it does the same thing. This strange phenomenon is an anomaly.

Example 3: While your monthly utility bill has been around $60 for many years, this month’s bill is $230! You have never had a bill higher than $80, and you have not bought any new electrical devices, so it is an anomaly.

Example 4: Same scenario as in Example 3, but now there are rumors that people have been getting erroneous sums on their utility bills this month. Now there is an explanation for the unusual bills, so, you do not get excited as much as in Example 3 and do not consider it to be an anomaly, just something unusual.\(^6\)

Example 5: You are the administrator of a network, and every day in the morning you check the previous night’s usage traffic. You usually see very low traffic between 2:00 am and 6:00 am, but one day you see a small peak at 4:30 am. Now, based on your previous experience, it can be considered unusual and not an anomaly. But if there were a large peak at 4:30 am, you might consider that to be an anomaly.

2.2. The close relation between anomalies and probability

Let us continue our discussion of the coin flipping experiment and suppose some alien from another planet who does not know anything about the outcomes of coin flipping (but knows probability and statistics!) starts to do experiments with a coin.

He flips a fair coin, and it lands heads up, so he assumes that if you flip a coin it lands heads up all the time. He tries the experiment for another 42 times and, unexpectedly (for us), every time he observes a head outcome. So, up to now, the evidence for his theory is that if you flip a coin, it lands heads up with the following probability:

\[^6\] That is in cat using Bayesian inference unconsciously, because you update the probability when you get a new evidence.
\[ P(\text{head}) = \frac{42}{42} \]

2-1- The probability of landing heads up is 100%, based on getting heads on 42 out of 42 tries.

He continues the experiment, and, suddenly, on the 125th flip, he gets his first tails. This outcome is an anomaly for him because he has never seen a coin land tails up. However, after this experiment, he understands that there is a small possibility of getting tails.

\[ P(\text{tail}) = \frac{1}{125} = 0.008 \]
\[ P(\text{head}) = \frac{124}{125} = 0.992 \]

2-2- The probabilities of heads and tails after 125 experiments

Now, he thinks it is possible but still unusual to get tails. But if he continues getting tails occasionally, he realizes that the outcome of a coin flipping experiment could be either heads or tails. Since the coin is fair, after 1,000 experiments, he might get a result of 495 heads and 505 tails.

\[ P(\text{tail}) = \frac{505}{1000} = 0.505 \]
\[ P(\text{head}) = \frac{495}{1000} = 0.495 \]

2-3- After 1,000 experiments, the alien understands that when you flip a coin, the probability of getting heads or tails is almost equal.

As this example demonstrates, it is a judgment call about how to define what is an anomaly and what is not, even when based on the calculated probability of an outcome. In some cases, it may also be necessary to define normal as an outcome with a minimum threshold of probability and anything with a probability lower than that threshold as an anomaly. No one way is correct all the time.

When variables are monitored, and probed, there might be hazards or noise in the data. If, for example, any outcome with probability lower than 0.01 is considered to be an anomaly, then the system would recognize even those hazards as anomalies. Since we usually look for malicious
activities, and these activities do not have the same kind of pattern as hazards or noise, it is better to define a probability range for anomalies, such as \([0.001 \ldots 0.010]\).\(^7\)

In this way, every outcome with a probability of less than 0.001 is a hazard, every probability between 0.001 and 0.010 is an anomaly, and everything else is a normal outcome.\(^8\) Now, back to our alien story: if he flips the coin 5,000 times and gets only one instance of the coin landing on its edge and not more, we can assume that this outcome is a hazard and not an anomaly, but if it happens 5 or 10 times, it counts as an anomaly.

### 2.3. Context matters

Context does matter. Some experimental outcomes that would be anomalies in one system might be normal in another system. For example, in Figure 2-4, the left side shows a typical single period of a time series. The situation from around \(t = 90\) to \(t = 120\) on the right side is considered to be an anomaly based on the fact that a smooth curve is expected.

![Smooth data](chart1.png) ![Smooth data with anomaly](chart2.png)

2-4- Rapid fluctuations in a smooth curve is anomalous behaviour.

Now compare Figure 2-4 with Figure 2-5; the same situation as seen in the anomalous section between \(t = 90\) and \(t = 120\) on the right of Figure 2-4 is normal behavior here, but in this case, a smooth signal, as shown in the right side of Figure 2-5 is something abnormal.

---

\(^7\) \([0.001 \ldots 0.010]\) means \(\{ x \mid x \in \mathcal{R}, 0.001 \leq x < 0.010 \}\).

\(^8\) We will see in next chapters that another way to solve this problem is to look at the sequence of observation results, not just at one.

\(^9\) As we said, it all depends on the system being probed and the tolerance for unexpected outcomes. For instance, in building an anomaly detection system which monitors the vital signs of a patient in the hospital, every outcome with a probability lower than 0.50 may be considered to be an anomaly.
2-5- Smoothness in a curve that always has some fluctuation is an anomalous behavior.

Figure 2-6 shows how a drop to zero is also an anomalous behavior for both time series.

2-6- A sudden drop in signal in both smooth and fluctuating time series is an anomalous behavior.

To end this chapter, let us reiterate that the above reasoning can be explained using the definition of an anomaly we gave based on probabilities. With this reasoning, the fluctuating part of Figure 2-4 is an anomaly because our experience shows that the probability of having that kind of fluctuation in a day is zero, and the same explanation goes for Figure 2-5, in which the probability of having a smooth curve is zero. In Figure 2-6, the probability of having such a steep drop is also zero because, in both time series, no drop of this kind has ever happened before\(^\text{10}\).

\(^{10}\) Note that if such a drop continues to happen, then, after some time, its probability gets large enough to consider it something unusual or even usual or normal.
3. System

3.1. What is a system?

The variables of interest we discuss here do not exist in isolation; they are parameters in systems we want to measure and monitor to see if everything is going well or not! A human body, a computer network, a car, a city, the world economy, etc. all are examples of systems. Whenever there are components exchanging data, and working together, it is a system. The boundaries of a system are defined by the people studying it, and the boundaries can be defined from different perspectives. Consider a car as a system: it is possible to measure and monitor parameters such as oil and gas levels, brakes, and engine health to make sure that the car is up and running. It is also possible to measure and monitor the bumpers and airbag to make sure that a driver is also safe. Sometimes it can also be useful to measure and monitor the interactions of a system with its environment, especially in cases where interest lies in an ecosystem containing many systems working together as an even bigger system.

3.2. Static and Dynamic Models

First, what is a model? A model is anything that tries to describe a system. All we are doing in this paper is trying to find a way to model the behavior of a system and find its anomalies. It is relatively easy for someone to look at a time series graph and guess what part of a time series has an anomaly if the ups and downs of the series set up an expectation or model in their mind and if some parts of the time series violate that model. We need to find a way to formulize this idea to be able to use it as a model in our control or monitoring programs. So, back to our coin flipping experiment: if someone flips a coin every hour and an observer of this system saves the result to analyze this process, after a while, it becomes clear that, no matter what time or day or month it is, the result is almost 50% heads and 50% tails. We call this a static model because the outcomes of the observation are independent of time, as shown below:

\[ P(\text{tail}) = P(\text{head}) = \frac{1}{2} \]

3-1- In a static system, the probabilities of the outcomes do not change with time.

In other words, if the mathematical description of the system shows no time dependency, the model is static. Let us describe why we say that the Formula in 3-1 outlines a static system. The experiment of flipping a coin has just two possible states, that is, either heads or tails, and Formula 3-1 shows that the state of that system does not depend on time.

Now consider a system that models a person’s pulse. People’s pulses are higher when they are awake, and lower when they are asleep. Suppose you wearing a sensor which reads your pulse every hour, and its resolution is 10 beats per minute. So, during the daytime, it might show 70 bpm or 80 bpm, or even 90 bpm or 100 bpm when exercising, and during the nights, when you are asleep, 60 bpm or 70 bpm, as described in Formula 3-2.
\[ \text{pulse}(t) = \begin{cases} 
70,80,90,100 & 6:00 < t < 24:00 \\
60,70 & t < 7:00 
\end{cases} \]

3-2- A model to describe the pulse rate of a human as a function of time.

If probabilities are taken into account, the model better describes the system. The model would then be something like Formula 3-3:

\[ \text{pulse}(t) = \begin{cases} 
P(70) = 0.6, P(80) = 0.38, P(90) = 0.01, P(100) = 0.01 & 6:00 < t < 24:00 \\
P(60) = 0.7, P(70) = 0.3 & t < 7:00 
\end{cases} \]

3-3- A model to describe probability of the pulse rate of a human as a function of time.

The model shown in Formula 3-3 provides better information about the behavior of your pulse. For example, it shows that your pulse is unlikely to stay at 100 bpm for a long time because history shows that, between 6:00 and 24:00, the probability of reaching that pulse rate is just one percent.

### 3.3. More information yields a better model

In both Formula 3-2 and Formula 3-3 we modeled pulse rate as a function of time, like this:

\[ \text{pulse} = f(\text{time}) \]

3-4- Pulse rate as a function of time.

However, if we could gather more information, we would be able to build a better model. For example, if we consider collecting other data, including what you are doing—let us call it “action”—your pulse rate can be described using Formula 3-5:

\[ \text{pulse} = f(\text{time}, \text{action}) \]

3-5- Pulse rate as a function of time and action.

Now our model tries to describe pulse rate based on two different parameters. Although the actions we do every day are usually related to the time of day, if we have access to both parameters, our model potentially would catch anomalies more accurately, for example, when you go to work at 3:00 am or if you are working and experiencing a heartbeat of 100 bpm! Both states are anomalies, which we could not catch if our model were just a function of time.
3.4. State of a system

The set of the values of variables that describes a system is called the state of a system. In coin flipping, the observation has just two states: heads or tails. For the system, we modeled in Formula 3-5, the combinations of different values of time and action determine the various possible states of the system. So, if we have three actions, such as “working”, “exercising” and “sleeping,” 19 pulse rate variations (i.e. from 0 bpm to 180 bpm with a resolution of 10 bpm) and 24 hours, at any time, the system could be in any of $3 \times 19 \times 24 = 1,368$ possible states.

However, most of the time, systems are not likely to return all of the possible states with equal probability. Instead, they return just a portion of all possible states, and the probability associated with each of these states shows the system’s behavior or its nature. The pattern that results is, in fact, a basis for recognition (clustering and classification) which our brains use every day. We will talk about this later.

For example, there are over one billion websites on the internet, but how many of them do you use every day? What is the likelihood that you will look at a specific website at a particular time? Most of us do not visit more than 50 specific websites daily. Now, if we collect information on when and how many times you access these sites, we can build a model to describe your web surfing behavior. Your model is perhaps different from someone else’s model, but it is not likely to be unique in the world. More state variables, such as how long you stay on each website, the sequence in which you visit them, the searches you usually do, etc., need to be added to the model to describe your unique web surfing behavior11.

We must be careful to understand why we build models in first place. Having a model containing a hundred variables is not necessarily useful because processing that much information to find anomalies in real time requires an enormous amount of CPU power and memory. The more parameters that get added to a model, the more complicated it gets, until, at some point, the complexity of the model may approach the complexity of the system itself12.

---

11 User behavior analysis tool is a software that gets this kind of information and builds a behavioral model or fingerprint for every user of the network. It is supposed to recognize people based on their computer usage behaviour.

12 In machine learning, when a learning algorithm gives around 100% correct answers with the data it trained on, there is a high risk that the algorithm will produce incorrect answers with real data because the model is over-fitted. A similar thing happens when a model is too general, also known as under-fitted.
4. Data

4.1. Data generator function

Collecting real data can take a long time, and sometimes things don’t go as planned, such as when systems or sensors crash or stop working. Even when everything works perfectly, the data still needs to be cleaned. Sometimes it is more efficient to use manually generated data for experiments. Based on what we talked about in Chapter 1, all that it takes to make appropriate data is to define the functions for the trend, periodicity, and residual for the simulation. For example, there could be:

1. A daily pattern which has a morning high peak, afternoon low peak, and another peak before a new day starts.
2. A weekly pattern in which usage increases on weekends.
3. A small and smooth residual.
4. A trend level that increases about 30% per year.
5. A sampling rate of every 5 minutes.

Using this information, we have $24 \times 12 = 288$ samples per day and $7 \times 288 = 2016$ samples per week. So, if time starts from 0, then, using pure sine waves, we can write the periodic pattern, trend and residual functions as in Formulas 4-1.

\[
\text{residual}(t) = \frac{1 - 2 \text{rand}(t)}{2}
\]

\[
daily(t) = 2 \sin\left(\frac{2\pi}{288} t\right) - \sin\left(\frac{4\pi}{288} t\right) + \frac{1}{2} \sin\left(\frac{2\pi}{72} t\right)
\]

\[
\text{weekly}(t) = \begin{cases} 
0, & \frac{(t\mod 2016)}{288} \leq 5 \\
4 \sin\left(\frac{(t\mod 2016) - 5 \times 288}{2 \times 288}\right), & \frac{(t\mod 2016)}{288} > 5
\end{cases}
\]

\[
trend(t) = 15 - \frac{5t}{100000}
\]

4-1- Components of the sample time series variable.

The “$|$” sign used above is for the modulo operator, which gives the remainder of time divided by 2016 ( = $12 \times 24 \times 7$). Any periodic function can be used to simulate the required periodicities; the key is being able to define the right period for daily and weekly values. Using these components and adding them together gives us the graph in Figure 4-2 for a single day.
4-2- One day sample of the components used in Formula 4-1.

Figure 4-3 below shows the effect of the weekly periodicity, in which the last two days of the week have more usage than the working days:

4-3- A week’s sample of the data used in Formula 4-1.

Now this function works to generate data for the time series, and anomalies can be added in to test the system.
4.2. Collecting latency data

Almost all operating systems have a ping utility used to check the latency between a user’s computer and some remote point. For example, in Mac OS or Linux, it’s simple to check the connection latency with Google via the command in Terminal, shown in Listing 4-4:

```
$ ping google.com -c 10
PING google.com (172.217.6.110): 56 data bytes
64 bytes from 172.217.6.110: icmp_seq=0 ttl=54 time=25.878 ms
64 bytes from 172.217.6.110: icmp_seq=1 ttl=54 time=29.438 ms
64 bytes from 172.217.6.110: icmp_seq=2 ttl=54 time=28.864 ms
64 bytes from 172.217.6.110: icmp_seq=3 ttl=54 time=73.999 ms
64 bytes from 172.217.6.110: icmp_seq=4 ttl=54 time=27.296 ms
64 bytes from 172.217.6.110: icmp_seq=5 ttl=54 time=26.867 ms
64 bytes from 172.217.6.110: icmp_seq=6 ttl=54 time=24.696 ms
64 bytes from 172.217.6.110: icmp_seq=7 ttl=54 time=27.373 ms
64 bytes from 172.217.6.110: icmp_seq=8 ttl=54 time=25.345 ms
64 bytes from 172.217.6.110: icmp_seq=9 ttl=54 time=32.693 ms
--- google.com ping statistics ---
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 24.696/32.247/73.999/14.089 ms
```

4-4- Using the ping command to measure the latency between a computer and a remote host.

To have only one ping result and the Unix timestamp, use this command:

```
$ echo `date +%s`,`ping -c 1 google.com|tail -1|awk '{print $4}'|cut -d '/' -f 2`
1481747355,25.929
```

4-5- How to get only one ping result, with a time stamp.

The next step is to put this command in a bash file and call it every 5 minutes in a crontab. The result of the echo command can also be added to a file. After a while, this collected data will show the real latency pattern between a machine and google.com or any other website over time.

4.3. Monitoring MySQL connections

Just like with the ping command, it’s possible to gather information about any parameters of any services running on your servers. For example, if you have MySQL running on one of your servers, you can collect the number of connected users, as shown below, and use the same method we described before to save the result in a file:

```
$ mysql -e "show status like '%Max_used_connections%'" | grep "Max" | awk '{print $2}'
16
```

4-6- Collecting the number of users connected to MySQL, in this case, 16.

---

13 crontab (cron table) is a file in Linux systems which contains the schedule of tasks that must automatically run.
4.4. Established TCP connections from a computer

In the same way that we used the ping and MySQL commands above, you can gather information about the number of established TCP connections from your computer using the command below:

```bash
$ netstat -an | grep "ESTABLISHED" | wc -l
```

4-7- A simple command to return the number of established connection from your computer.

You can start playing with time series with any of the measures mentioned above. The only thing you need to make sure of is that we prefer regular univariate time series, so sample them regularly.

4.5. Pulling versus pushing

From the ADE (anomaly detection engine) side, you can pull information or have the system you are observing pushes the information to your engine, as shown in Figure 4-8.

![Diagram of Pushing vs. Pulling data models](image)

4-8- Pushing vs. pulling data models.

Pushing is when the system which generates data simply sends a copy of the data to the ADE whenever it has them ready. It is a bit safer for the system because no external source is required to poke it. With the pulling method, the ADE asks the system in every time interval to get a copy of the updated data.

Using the HTTP or HTTPS protocol can be helpful in either of the methods. However, since we are not sure whether the monitored system can terminate these kinds of connections, we suggest using the push method. So, all we need to do is to design and implement a listener for the ADE and ask the system to send the data to the ADE server. Moreover, in most cases, you do not even need to write any agent application to do it, a simple script can send data to your ADE listener, as shown in Listing 4-9.

```bash
$ curl http://www.myADE.com/data?time=`date +%s`
data=`mysql -e "show status like '%Max_used_connections%'" | grep "Max" | awk '{print $2}'`
```

4-9- The command to call in each time interval to send data to your ADE listener.
The curl command calls the data servlet of your ADE service and passes the time and updated value of the number of MySQL connected users. That is like calling the servlet with “time=1481821906&data=21” parameters. If you are collecting data from a real working system, the collected data could be something like the data shown in Figure 4-10.

![Graph](image)

4-10- Sample of 15 days of real data from a sensor sending data every two minutes, or 720 times per day.

Over the 15-day period shown above, the pattern of the variable for the five workdays is different than the pattern over the weekends, suggesting a weekly pattern. Importantly, the time series is not smooth, but instead it is spiky and has some large fluctuations. This is completely normal, in fact, it usually happens when collecting aggregated data from a system where the count of aggregating sources is small.

For example, if you assume that Figure 4-10 shows the internet usage of an ISP, then it clearly shows that the ISP has a small number of users. If it had a large user base, then the usage pattern would be smooth. Visually analyzing and understanding the anomalies in this kind of time series is difficult, so we will start first with smoother patterns, and then discuss spikier patterns in later chapters.
5. Introduction to time series anomaly detection

5.1. Using static thresholds

After monitoring variables’ changes for a while, you can usually get an idea of how much each variable can go up or down, so it is usually possible to define maximum and minimum safe values. Then, whenever the updated data goes above or below these values, your software can assume that something strange is happening and can generate an alarm. This model works well for some systems, such as controlling the temperature of a room or maintaining a minimum number of items in inventories, etc.

![Sample Data](image)

5-1- Sample time series data showing unusual activity between t = 25 and t = 50 (circled).

Figure 5-1 shows some real data from a time series. If we put a high threshold at 0.7 and a low threshold of 0.4, then there are three points when the variable is above or below the defined thresholds. In this model, we have assumed that normal behavior is when the variable has some value between thresholds, so based on our definition of the normal range of values, our system works well. But if we carefully study the above series, you will find that there is something unusual going on between the times of 25 and 50; the variable is experiencing a lower level than its normal value. Our simple threshold based anomaly detection cannot detect this situation. We can also see that the local minima seem to have a periodic or at least cyclic rhythm. Again, this threshold-based model is not able to detect or control the rhythm of these local minima because we have not defined the pattern of the local minima as critical or defined the system to operate so as to keep its average at a certain level. We, therefore, cannot expect our simple system understand or catch these anomalies.
5.2. Baselines and deviation

Many people prefer to define a baseline for their time series, and then calculate the deviation from that baseline. The idea is correct in a variety of circumstances. In fact, whenever we make a choice between different options, we make a comparison by calculating the deviation between each option and the baseline or the ideal option in our mind, and then we choose the one which is closer to the baseline.

For example, consider what you would do if you were the manager of a company and wanted to hire an accountant for the enterprise. You have interviewed some people, have their resumes on your desk, and want to choose one of them. You unconsciously have an idea (or model) of what a good accountant is in your mind. You just go through the resumes and remember the corresponding interview, or even the face of each candidate, and then compare this information with your ideal model. You do this for all of them and choose the best fit, the one with minimum deviation from the model you have in your mind. Even if you think you do this by your gut instincts, the model is what is in your gut instincts. So, note that a baseline does not necessary mean a literal line, even though it is easy to show a line in graphs. I prefer to use the term model instead of baseline to describe the norm or what is expected.

5.3. Moving average

Suppose you have a time series like the one in 5-2. Then, at any time, you can calculate the average of the recently received values of the series, as shown in Formula 5-3.

\[
v(t) : (t_1, v_1), (t_2, v_2), \ldots, (t_i, v_i) \ldots
\]

5-2- A general univariate time series.

\[
mAvg(t_i) = \mu_{i-k+1:i} = \frac{1}{k} \sum_{j=i}^{i-k+1} v_j
\]

5-3- Moving average of the time series, for a window of k items.

Formula 5-3 calculates the average of the time series based on the last \( k \) items; we call this a moving average\(^{14} \). If you calculate the moving average whenever new updated data comes in, since most variables behaving normally do not experience a sudden dramatic change in their movements, then you can estimate the next value likely to come in by allowed deviation margin over recent average. Figure 5-4 shows the series we had in Figure 5-1, but with margins and the moving average added.

\(^{14}\) There are other methods to calculate the moving average which we will discuss later in this chapter.
Here the moving average is based on the last 10 data items, and the margins are 10%. The upper and lower margins do not need to be equal; it depends on your business and needs. If you compare this model to our first simple static threshold method, you can see that this method can be interpreted as a dynamic threshold model, a model which updates and adjusts its safe margin thresholds as time passes.

![Moving average and dynamic threshold](image)

5.4. The magic of our brains

You may wonder why, when you look at a time series, even if you have not seen many days or months of the series, you can find almost all of its anomalies. No magic or complicated processing happens when you look that time series; the actual magic has been happening for your whole life and is still happening now.

From the moment, you born, and perhaps even before that, your eyes and brain work together to gather information from around you. Let us do a simple calculation. If you are 35 years old, multiply your age by 365 days in each year, 15 hours of open eyes per day, 60 minutes per hour, and at least 30 captured pictures. This gives us about 350 billion processed pictures. This is not the whole story, either, because, in each picture, there are different objects with different shapes and movement patterns. Now, look through the window or look around; how many objects do you see? 10, 50, 100, 500? Any of these objects contains more details, and most of the time smaller objects. Let’s assume we process an average of 100 objects in each picture. Then, the number of objects you have processed by the age of 35 comes to 35 trillion!

---

15 Naturally, it starts from recognizing dark and light.
16 These are just static objects or patterns. In real life, there are many sequence and temporal patterns that we also learn, know, and use every day.
17 35 trillion is like number of seconds in 189 thousand years!
We have categorized every single one of these objects in our brains and have assigned some relations between them, too. So, it is not surprising that, when we see a time series, it is easy for us to pick out its anomalies, while it is a big deal for a computer program to do so. The important thing to think about is that:

You cannot expect your model (or baseline) to answer any questions if you have not provided the required knowledge or information to it in advance.

The above quote is a fact that we usually forget and expect our learning system understands everything we understand. So, in our recent model using moving average, the only thing our model knows is the average of its most recent 10 data points and a constant margin for deviation. If we assume one new data point per minute, the model knows only some limited information—single numbers—about what happened in the last 10 minutes, and nothing more. If we compare this very narrow knowledge with the knowledge and intelligence we each have gathered and built upon in our minds over the years, the differences between our abilities to detect anomalies and computers’ abilities to detect anomalies make more sense.

5.5. A model for daily patterns

A moving average model monitors a time series to make sure its changes are close to the recent moving average of the data coming in. It cannot predict the future, and it does not know the differences between morning and night or working days and weekends. But it is simple to build a model that incorporates information about the time of the day and working days vs. weekends.

To do this, the first step is to gather enough information and let our ADE train on it. Suppose you are sending 12 samples per hour to your ADE. After four weeks, you will have samples of $4 \times 5$ working days and $4 \times 2$ weekends\(^{18}\), which is enough to start building a weekly model. To start, let us show our daily time series as in Formula 5-5 because it is regular and contains $24 \times 12 = 288$ samples.

\[
dailyData = [v_1, v_2, v_3, ..., v_{288}]
\]

5-5- A single day regular univariate time series with 12 samples per hour.

Now, based on the gathered information, you can calculate the average of the data for every sample in a day as follows\(^{19}\):

\(^{18}\) We may want to build seven different time series models—one for every day of the week—but here we simplify it by modelling only two different kinds of days, namely, working days and holidays.

\(^{19}\) Note that this is not a complete model, but is better than what we have discussed till now.
\[
\text{workingDaysAverageSeries} = \left\{ \frac{1}{20} \sum_{i \in \text{WeekDays}} v_{i1}, \frac{1}{20} \sum_{i \in \text{WeekDays1}} v_{i2}, \ldots, \frac{1}{20} \sum_{i \in \text{WeekDays}} v_{i288} \right\}
\]

\[
\text{weekendsAverageSeries} = \left\{ \frac{1}{8} \sum_{i \in \text{Weekends}} v_{i1}, \frac{1}{8} \sum_{i \in \text{Weekends}} v_{i2}, \ldots, \frac{1}{8} \sum_{i \in \text{Weekends}} v_{i288} \right\}
\]

5-6- Average series for working days and weekends.

Using 5-6 and the four weeks of sample data, an ADE can build an average time series for working days and weekends. It can also be called a model, typical behaviour, or a baseline. Now, the ADE has some expectation of how the time series should behave on working days or weekends at any time of day. When new data comes in, the ADE chooses one of the two average models based on what day it is, and calculates the difference between the incoming data and its corresponding item in the average series. The ADE still needs those static thresholds or deviation percentages to make sure that the variable is in its safe region. This is one of the simplest ways to model different days’ behaviour and have some ability to predict future behaviour.

### 5.6. Basic ADE design

Systems usually change their behaviour over the time because they do not work in stable and fixed ecosystems, so their parameters and measures of behavior change too. If we keep the four-week average models we introduced in Formulas 5-6 and do not update them, after a while, when the system changes its behaviour, almost every new data point could count as an anomaly. For example, if we build these daily models for a city’s electricity usage in a given year, after months or years when population increases, almost any incoming data will be higher than the safe margin, and the system will consider every incoming data point as anomalous even though they are not. Now look at Figure 5-7:

5-7- Basic design for Anomaly Detection Engine.
The basic design for our anomaly detection engine should contain a switch that allows the ADE to update its model whenever new data comes in. The processing model is something like this: first, the ADE should compare the newly-received information with the current model, and then, if it is required, it should update the model. The order of these tasks is essential; you cannot first update the model and then compare the data with the model, because updating the model with the received data before comparison would change the model toward the new data and potentially could reduce the deviation.

Now, consider a new data point that comes in at time=125 in a working day. Our ADE:

- Selects the corresponding working day model,
- Selects the 125th average item in the series (or daily model),
- Calculates the difference between the given data and the selected item of the series,
- If the deviation is above the defined margin, it generates an alarm,
- If the “Update Model” is checked, ADE should update the 125th item in the working day average series.

The only problem with this process is updating the items in the average series. While it may look like the ADE needs to have access to the all previous data to do this, but there is another way. The ADE can calculate the average the way we demonstrate in 5-8 below, by just keeping and updating the average value along with a counter that shows the population of the data that has been used in calculating the average:

$$newAverage = \frac{(N \times oldAverage) + newData}{N + 1}$$

5-8- How to update an average when you do not have access to the previous samples.

Although calculating the average as in 5-8 works and looks logical, the concept of an average itself does have a problem. Consider what happens after many years, when we update the average repeatedly. In a case like this, N gets so large that \(\frac{N \times oldAverage}{N+1}\) grows to become almost equal to \(oldAverage\), and \(\frac{newData}{N+1}\) shrinks down almost to 0, so the new data does not have enough weight to change the average or the model. The model then gets stuck, and cannot adapt itself to the new behavior of the system. The simplest way to solve this problem is to use an exponential moving average, which we will describe in the following chapters when we discuss learning mechanism.
5.7. Using other descriptive analytics measures

Other descriptive analytics methods can also be used to find outliers or anomalies. For example, you can calculate the standard score as shown in 5-9 to see if recent data is off when compared to the last set of observed data. If the score is greater than 3, or any other defined threshold, you can consider the last element as an outlier or anomaly.

\[
\text{series} : \ldots, v_{i-k+1}, v_{i-k+2}, \ldots, v_{i-1}, v_i
\]

\[
z = \frac{v_i - \mu_{v_{i-k+1:vi}}}{\sigma_{v_{i-k+1:vi}}}
\]

5-9- How to calculate the z-score for the recent window of the series item.

Bolton and Hand’s breakpoint analysis is another way to detect sudden changes in the behavior by defining two different windows with sizes of \(k\) and \(l\) as below, calculating their averages, and then using a method to compare these averages, such as the student’s t-test:

\[
\begin{align*}
\text{recent window of size } k & : \ v_{i-k+1}, v_{i-k+2}, \ldots, v_{i-1}, v_i \\
\text{model window of size } l & : \ v_{i-k-l+1}, v_{i-k-l+2}, \ldots, v_{i-k-1}, v_{i-k}
\end{align*}
\]

\[
t = \frac{\mu_{v_{i-k+1:vi}} - \mu_{v_{i-k-l+1:vi-k}}}{\sqrt{(k - 1) \times \sigma_{v_{i-k+1:vi}}^2 + (l - 1) \times \sigma_{v_{i-k-l+1:vi-k}}^2} \times \frac{1}{\sqrt{k + l - 2}}}
\]

5-10- How to calculate the t-score for two different window size of the series.

It is possible to describe the static behaviour of a time series using measures other than the z-score and t-test described above. But, no matter which measure you use, you need to write a simple test program and observe how the calculated index of the series changes over time. The next step is to find a method to use these measures so that the model adapts to the system and learns the correct behavior of the series over time.

---

20 Many of these indexes work only on data with a normal distribution, but you can always assume your data distribution is normal in small windows.
21 Richard J. Bolton and David J. Hand
6. Using Data Distribution

6.1. The need for more thresholds

Look at Figure 6-1 and try to figure out its pattern. Though it may seem to have some kind of periodicity, it mostly looks like noise—but it is not. Let us investigate why not.

We usually tend to draw time series as line graphs, but the truth is that, since we gather discrete data, it is sometimes better to draw them as scatter plots like the one shown in Figure 6-2.

As Figure 6-2 shows, at any point the data could be around 11, 21 or 31. There are many system variables that could behave like this, for example, a tristate switch.
None of the models or indices we have talked about yet works in this situation, so we need to use a different type of logic to represent these states, like the one used in 6-3.

\[
\text{isAnomaly}(v_i) = \begin{cases} 
\text{TRUE}, & v_i < 10 \text{ or } 12 \leq v_i < 20 \text{ or } 22 \leq v_i < 30 \text{ or } v_i \leq 32 \\
\text{FALSE}, & 10 \leq v_i < 12 \text{ or } 20 \leq v_i < 22 \text{ or } 30 \leq v_i < 32 
\end{cases}
\]

6-3- A logical statement used to find anomalies in the time series shown in Figure 6-2.

However, it is not sufficient to manually write logical statements to find anomalies; we need to clearly define the logic used to find the anomalies. In the case of the data shown in Figure 6-2, we can do this by asking ourselves how to define these regions on the graph where data appears.

6.2. Using data distribution

When you look at Figure 6-2 what you are probably doing is finding patterns by mentally grouping the data that are alike or close to each other. We call these groups clusters. We can recognize three clusters in the given data between 11±1, 21±1 and 31±1. The simplest thing we can do to find the data distribution is to define some data ranges, for example 0–2, 2–4, … 10–12, … 30–32, … and assign each of them an integer counter and set them all to zero. Then, for the entire given data set, increase the corresponding counter of the data value to produce a histogram:

![Histogram of the data in Figure 6-2.](6-4)

The histogram shows the distribution of the data. Figure 6-4 shows, almost 50% of the data items have values around 21, 25% around 11 and another 25% around 31.

Drawing a histogram is like looking at the scatter plot from the left side’s vertical axis. Figure 6-4 is therefore exactly like what you would see if you could look along Figure 6-2 from its Y axis. Histograms can be generated to determine the data distribution for any time series, even with analogue data. Consider an analogue time series changing between zero and 2,500. If you choose intervals of 25, then you have 100 clusters or counters at most.

Now whenever new data comes in, you can simply divide it by 25 and use the integer part of the result as the cluster ID or counter index, and then just increase that counter by 1. This method is
one of the widely-used clustering methods for single dimension data in which the clusters’ width is predefined. Listing 6-5 shows a piece of code that finds these clusters; it uses Java’s HashMap structure as the cluster counter.

```java
HashMap<Integer, Integer> getTimeSeries1DClusters(double[] timeSeries, double clusterWidth) {
    if (clusterWidth == 0 || timeSeries == null || timeSeries.length == 0) {
        return null;
    }
    HashMap<Integer, Integer> clusters = new HashMap<Integer, Integer>();
    Double data;
    Integer clusterIndex;
    for (double sample : timeSeries) {
        data = sample / clusterWidth;
        clusterIndex = data.intValue();
        if (!clusters.containsKey(clusterIndex)) {
            clusters.put(clusterIndex, 0);
        }
        clusters.put(clusterIndex, clusters.get(clusterIndex) + 1);
    }
    return clusters;
}
```

6-5- Simple Java function to return the histogram of a time series.

Note that the code in Listing 6-5 does not care about the maximum or minimum values of the data. Generally, though, it is better not to have negative numbers. In next few chapters, we introduce better ways to cluster both negative and positive data values.

### 6.3. Two-dimensional distribution

Suppose your time series has some periodic behavior that is important to monitor. All the methods we have discussed so far ignore the time of data measurement and have no direct sense of time. The average models we discussed in Chapter 5 do account for time in their calculations because we keep unique mean value for the time quanta of the series. However, we can do better and find anomalies in two dimensions at once, time and value. Consider the time series we have in Figure 6-6, which has some repetitive patterns; it looks like after each 19th or 20th data point, the series experiences a local minimum. These minima appear to be part of the system’s behavior and not anomalies.
If we pass the data in Figure 6-6 to the procedure we introduced in Listing 6-5 with a \textit{clusterWidth} of 1, then we have only two clusters with a non-zero population. Let’s call the data in [1,2) cluster C1 and the data in [2,3) cluster C2. The population density of the cluster C2 is about 19/20, and for the cluster, C1 is about 1/20. Now, if we get a new data sample, such as (t=148, v=0.3), it goes into cluster 0. Since this cluster is empty, we should assume it is an anomaly—and this is OK. However, if we get a new data sample of (t=148, v=1.7), that data point goes into cluster C1 and it fits with our model and looks correct, though you know that it is not! The solution to make the model time aware is easy; we need to use a two-dimensional distribution.

6-7- A two-dimensional distribution of a time series with a repetitive pattern.
In Figure 6-7, we have both vertical and horizontal segmentation, which lets us do two-dimensional data clustering. If we name our clusters using rows and columns, then clusters C2.0, C2.1, … C2.6 all have around 19 members each, and clusters like C1.0, C1.1, … C1.6 have under 3 members each. If we find this data distribution over enough samples, we will expect to see the same distribution in the future. So, if we get the most recent 20 data items and determine their distribution, it should comply with the expected distribution, that is, what we already found, which means the ratio of data in the range [1..2) to [2..3) should be 0.1 or even zero; anything else could potentially be an anomaly.

We can use the same idea in Listing 6-5 to write a program to build a two-dimensional distribution of a time series. The only difference is that now we also need to use clusterTimeWidth and clusterValueWidth instead of just clusterWidth. For this purpose, it is easier to use <String, Integer> for HasMap’s entry elements.

class DataPoint {
    long timestamp;
    double value;
}

HashMap<String, Integer> getTimeSeries2DClusters(ArrayList<DataPoint> timeSeries, double clusterValueWidth, long clusterTimeWidth) {
    if (clusterTimeWidth == 0 || clusterValueWidth == 0 || timeSeries == null ||
        timeSeries.size() == 0) {
        return null;
    }

    HashMap<String, Integer> clusters = new HashMap<String, Integer>();
    Double value;
    Integer valueIndex;
    long timeIndex;
    String clusterIndex;

    for (DataPoint sample : timeSeries) {
        value = sample.value / clusterValueWidth;
        valueIndex = value.intValue();
        timeIndex = sample.timestamp / (24 * 60 * 60);
        timeIndex = sample.timestamp - timeIndex * (24 * 60 * 60);
        timeIndex = timeIndex / clusterTimeWidth;
        clusterIndex = valueIndex + "." + timeIndex;

        if (!clusters.containsKey(clusterIndex)) {
            clusters.put(clusterIndex, 0);
        }
        clusters.put(clusterIndex, clusters.get(clusterIndex) + 1);
    }

    return clusters;
}

6-8- Sample HasMap Java code to build a two-dimensional distribution of time series data.
Here in Listing 6-8, each element of the time series is an instance of the DataPoint class, and in the procedure, we calculate two indices, one for value and one for time. To simplify the process, we assume the given time is in Linux time format with seconds resolution. In the process of calculating the timeIndex, we first find the number of whole days in the given time period, then calculate the number of seconds from midnight of the given timestamp. We are going to give more examples of two-dimensional clustering in next chapters. To close this chapter, we need to emphasize that:

Our primary and only goal of using distributions is to find out the behavior or pattern of data in a period. We are not interested in and will not use any known parametric statistical distributions in our process of finding anomalies.
7. Clustering and Classification

7.1. What is clustering?

This paper is not about machine learning or artificial intelligence, so we are not going to discuss the details of clustering or describe the hundreds of methods available here. However, we will still describe the concepts required to build our time series anomaly detection engine (ADE).

Clustering is nothing other than organizing or grouping according to similarities. If you want to cluster all the people in the world into two groups, for example, you could put them into two clusters of any kind: children and adults, or short people and tall people, or men and women, etc. In other words, we do not need to know in advance what information to use to form clusters; we have choices about which features we use to cluster data. Data clustering can also be done on the fly, with no prior knowledge about how the data might or should be grouped.

7.2. Distance function

The most important function in data clustering is the distance function, the output of which is a measure of how similar two different objects are. To start clustering, using almost any method, you need to choose some objects to act as the centers of your clusters. Then you must measure the distance between other objects and these central points. The central point closest to a data point indicates the cluster where that data belongs. Distance functions should be based on the properties of the objects you want to compare. In one-dimensional numeric data clustering, the distance function could be just the absolute value of the difference. Given two different objects with values \( x \) and \( y \), the distance between them can be expressed as:

\[
\text{distance}(x, y) = |x - y|
\]

7-1- Sample single-dimensional numeric data distance function, also called Manhattan distance.

For two-dimensional numeric data in which \( X \) and \( Y \) are vectors, we may choose to use a Euclidean distance function instead:

\[
\text{distance}(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}
\]

7-2- Sample two-dimensional numeric data distance function, also called Euclidean distance.

---

22 Or centroids

23 The general form is: \( \text{distance}(X,Y) = \sum_{i=1}^{n} (x_i - y_i)^2 \).
You can define your distance function to suit your needs. For example, in some machine learning algorithms, any given text document can be mapped to a single vector in multidimensional space. The distance between two documents is then usually calculated by a distance function called cosine similarity, as shown below. This function measures the cosine of the angle between the two vectors:

\[
distance(X, Y) = \frac{X \cdot Y}{\|X\| \|Y\|} = \frac{x_1y_1 + x_2y_2 + \cdots}{\sqrt{x_1^2 + x_2^2 + \cdots} \sqrt{y_1^2 + y_2^2 + \cdots}}
\]

7-3- Cosine similarity as a distance function.

In 7-3, \(X \cdot Y\) is the dot product of two vectors, and \(\|X\| \|Y\|\) is the product of the vectors’ magnitudes. You can see that you have a lot of choice in defining distance functions that will work with your data. That said, in defining a distance function, the following conditions must be met:

1. \(d(x, y) \geq 0\)
2. \(x = y \iff d(x, y) = 0\)
3. \(d(x, y) = d(y, x)\)
4. \(d(x, y) + d(y, z) \geq d(x, z)\)

7-4- Conditions that any distance function must meet.

The first condition is that the distance should be greater than or equal to zero; all the examples above yield a positive result. The second condition is that, if two objects are equal, the distance between them should be zero, and vice versa. The third condition is that the distance function should be symmetrical. The fourth and last condition is that the distances should satisfy the triangle inequality. We must therefore be careful when defining a distance function for our specific clustering project. If a function does not completely comply with all of the conditions in 7-4, its output may be wrong.

7.3. Time series distance function

Since every element of a time series is a point defined by a time and a value, it may seem that we can simply use the Euclidean distance function to recognize points that are close to each other and group them together. However, whether this is a good idea depends on how we define the similarity or distance between two points. We may decide to consider an item like (10:56, 206) closer to (10:23, 200) than to (11:01, 200), even though the Euclidean distance gives another result, depending on how large the time windows are and what the centres of the time clusters are.
For instance, consider the distance function in 7-5, in which x and y are two different points from the time series, \( v() \) and \( \text{hour}() \) functions return the value and hour of the passed point and \( \text{MaxV} \) determines the maximum possible value of the measure:

\[
d(x,y) = \begin{cases} 
\text{MaxV} \times |\text{hour}(x) - \text{hour}(y)| + |v(x) - v(y)|, & \text{hour}(x) \neq \text{hour}(y) \\
|v(x) - v(y)|, & \text{hour}(x) = \text{hour}(y)
\end{cases}
\]

7-5- Simple distance function for time series.

The function in 7-5 assigns new data into the cluster in the hour during which the data came in, otherwise it assigns it to closest hour and value.

### 7.4. What is classification?

As we discussed in 7.1, in AI and machine learning terminology, the act of grouping objects without any prior knowledge of how they should be grouped is called clustering. However, when you already have some data points labeled to show where they belong, there are methods that can assign any new given data to those groups; this is called classification or supervised learning. The difference between clustering and classification is that with classification, the groups that data is put in are known in advance—it is “supervised.” For example, you may be asked to classify people into two particular groups, such as people who work for themselves and people who work for someone else. Alternatively, in a time series, for example, we may define seven different classes to accommodate each day of the week, and assign data to these classes. To build an ADE, we may need to use either clustering or classification, or both methods.
8. Temporal patterns

8.1. What is a pattern?

A pattern is an arrangement of things that repeats. Although this a general description, it covers almost every possible pattern we observe. Square, triangle, and circle are patterns because we see many things that are in the shape of a square, a triangle or a circle, around us.

Almost everything in the world is dependent on patterns. Any decision you make is based on the patterns you have stored in your mind. How we talk is based on the patterns in the languages we speak, the elements in the periodic table are also based on the patterns of how subatomic particles stick together, etc.

If you observe a bus station and see that a bus stops there every half an hour, that is a pattern. You may then find out that this particular bus does not come to the station on weekends; then the fact that it stops there on working days and not on weekends is another pattern on top of the hourly pattern, showing that patterns can have a hierarchy.

8.2. Why are patterns important?

If you find a pattern in something, then it is likely to repeat in the same way, and you can use this to improve your power of prediction. For example, we know that winter comes after fall every year because there is a pattern to the changes of the seasons, so we prepare for winter in the fall.

Prediction is the key to finding anomalies, but the patterns used in predictions are not necessarily time-dependent. Suppose you are examining a used piano keyboard you want to buy. You start playing the keys from left to right; there are 88 keys, and after playing 20 or 30 keys, you get an idea of the piano’s timbre. The timbre is a complicated pattern of variations in sound waves that you have now learned. As you continue playing keys, you expect to hear the same timbre, but in another pitch. If the timbre is different, though, then there is something wrong with the piano.

8.3. Temporal and spatial patterns

Patterns that repeat in space are called spatial patterns, like what we see in geometric tiles or the shapes of different kinds of trees, or the appearance of different people from different countries, like height, body shape, eyes or noses form, etc.

Most importantly for time series, though, when you continuously observe a system, you may find that specific kinds of changes happen repeatedly. This type of repeat is called a temporal pattern. Music is one of the simplest possible examples of a temporal pattern. Each note has a duration and pitch; the pitch is a sound wave with a particular wavelength, and there are specific gaps in wavelengths between neighbouring notes. Changes in pitch, duration, volume, etc. over time build musical patterns, such as motifs, melodies, themes, etc.
Time series behave like music. For example, Picture 8-1 is the first four bars of Beethoven’s “Ode to Joy.” Even if you don’t know how to read music, if you know that time passes from left to right, you can see patterns.

![First four bars of Beethoven’s “Ode to Joy,” 9th Symphony, Movement IV](image)

So, studying time series is analogous to studying music; we look for melodies and harmonic patterns interacting with each other and changing over time.

### 8.4. How is time different from space?

When you look at ordinary two-dimensional data, like a painting, you can move your attention from the top to the bottom, from the bottom to the top, right to left or left to right, observing the data, finding patterns, building models and even looking for anomalies. But time is special for us in because of its unidirectional behaviour. Consider time series as two-dimensional data, with one time dimension and one spatial dimension, but with the restriction that only present and past data can affect us or let us predict the future.

There are various theories and beliefs about time; some physicists believe that time doesn’t exist by itself, and that it is just the entropy increment. However, we usually represent time with a unidirectional axis as the fourth dimension, perpendicular to the other three spatial dimensions, even though, since we technically can’t go back in time, the time dimension is probably fundamentally different from other dimensions.

Another special feature of time, especially when it comes to time series, is that we often tolerate time shifts in our data. An afternoon traffic peak—rush hour—could start at 5:30 pm or 5:15 pm or even 5:45 pm, and, most of the time, the exact moment it starts does not matter to us. However, we would care if it started much earlier or much later, such as at 4:00 pm or 6:30 pm, because it would affect our daily routine. Another example could be that you probably do not care whether your workplace deposits your salary at 23:00 pm or 20:00 pm on the last day of the month, but you would care if they paid you less than what you expected or paid it two days later.
8.5. Markov Chains

If, every half hour, you observe and store information about whether you are walking or sitting, from 7:00 am when you wake up until 10:00 pm when you go to bed, after a week, you will have a sequence with $7 \times 16 \times 2 = 224$ items in it. A state diagram showing these two states—walking and sitting—could have four possible transitions in it. Now, if you count each state transition and divide the total by the total number of transitions logged, i.e., $223 = (244-1)$, you could represent it as shown in Figure 8-2 which shows the probabilities of staying in the current state or changing it:

![Markov chain for walking and sitting states.](image)

Markov chain models are usually used for systems in which the next state depends only on the current state; in other words, systems that have no memory of the more distant past. The probabilities shown in Figure 8-2 indicate that, whether you are sitting or walking, you are more likely to maintain the same state over the next time interval than to change it. The state dependency we discussed can be defined as:

$$\text{State}(t_{i+1}) = F(\text{State}(t_i))$$

8-3- In a Markov chain, next state of a system depends only on its current state.

We have used capital letters in Formula 8-3 because we can characterize the state of a system at any time as a vector. While we are not going to show, or prove it here, we can also write Equation 8-3 using Matrix operations, as shown in 8-4:

$$\text{StateMatrix}(t_{i+1}) = \text{TransitionMatrix} \times \text{StateMatrix}(t_i)$$

8-4- Markov Chain expressed with matrix operations.
So, based on the diagram in Figure 8-2 we can write the following equation, which describes the same idea but in mathematical form. This ability to represent Markov chain relationships using vectors is helpful because we can use it in our computer programs later:

\[
State(t_{i+1}) = \begin{bmatrix} 0.70 & 0.05 \\ 0.30 & 0.95 \end{bmatrix} \times State(t_i)
\]

8-5- Sample representation of a Markov Chain using vectors.

In Equation 8-5, the state vectors have only one column and two rows each; the first row is the probability of being in the walking state and the second row indicates the probability of being in the sitting state. If the initial state is sitting, the next two state vectors are as shown below in 8-6. Note that, in any state, the sum of the probabilities must add up to one. The state vector of the system shows the superposition of the possible states, which means that the system could be in either of the two possible states, with the probabilities shown, until you observe the system to confirm exactly what it is doing:

\[
State(t = 1) = \begin{bmatrix} 0.70 & 0.05 \\ 0.30 & 0.95 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.70 \times 1 + 0.05 \times 0 \\ 0.30 \times 1 + 0.95 \times 0 \end{bmatrix} = \begin{bmatrix} 0.70 \\ 0.30 \end{bmatrix}
\]

\[
State(t = 2) = \begin{bmatrix} 0.70 & 0.05 \\ 0.30 & 0.95 \end{bmatrix} \times \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 0.70 \times 0.7 + 0.05 \times 0.3 \\ 0.30 \times 0.7 + 0.95 \times 0.3 \end{bmatrix} = \begin{bmatrix} 0.505 \\ 0.495 \end{bmatrix}
\]

8-6- Calculating the second and third states of the system shown in Figure 8-2, assuming the initial state of 1 (sitting).

We can use Markov chains in our anomaly detection system to learn about expected behaviours during specific subsets of time, for instance, one for each of midnight, morning, noon and night, and to predict the next state of the system in each time segment. As an example, suppose you are collecting data as in the previous example, but, this time, for a network’s traffic. This time, the Markov chain diagram looks like Figure 8-7. This diagram shows that a transition directly from low to high traffic has not happened before, so if it does happen, it could potentially be an anomaly.

8-7- Markov chain of observed network traffic.
9. What is learning?

9.1. Learning

For the purposes of time series analysis as discussed in this paper, we can define learning as the process of updating and changing a model so that it describes a system more accurately and comprehensively.

Learning starts with the first model you build of the system you are observing. In our coin flipping example, as soon as we flip the first coin and get heads, we learn that the outcome is heads. At this moment, what we have learned is not correct for all future instances, but if we continue doing the experiment and updating our learning model, we learn that the outcome of flipping a coin could be heads or tails with almost equal probabilities.

9.2. Moving Average

Whenever we calculate the average of the recent values of a time series, we are also updating our model’s expectations about future incoming data. This is what learning means, regardless of how accurate the model is; we constantly update our model in order to make better predictions. In Figure 9-1, the dashed line shows the average of the last ten samples, which follows the behaviour of the changing variable with a small margin.

![Moving Average](image)

9-1- The dashed line shows the average of the last ten samples, while the grey line shows the incoming data itself.
The size of the window we use when looking back and updating the average depends on our strategy. With a small window, the model will show more recent behaviour of the system, while a large window shows long term behaviour. We can also get an idea of safe the margins by calculating the maximum or average deviation or standard deviation. It is up to us to consider what difference from the average or range of standard deviation counts as normal behaviour. For a time series, we can have the moving average and margin as shown in 9-2 for a window size of $k$, where $\mu$ and $\sigma$ are the average and standard deviation for the data in the window, respectively:

$$mAvg(t_i) = \mu_{i-k+1:i} = \frac{1}{k} \sum_{j=i}^{i-k+1} v_j$$

$$safeDeviation(t_i) = 3 \times \sigma_{i-k+1:i}$$

9-2 - Moving average for a window of last $k$ items of a time series and a suggestion for its safe margin, using mean ($\mu$) and standard deviation ($\sigma$).

### 9.3. Exponential Moving Average

Consider a time series in which we are going to define the exponential moving average (EMA) as shown in 9-3. The formula for the EMA says that, whenever a new data point comes in, the new value of the EMA will equal $\lambda$ portions of the newly arrived data plus $(1 - \lambda)$ portions of the previous EMA.

$$time series : (t_1, v_1), (t_2, v_2), \ldots, (t_i, v_i), \ldots$$

$$0 \leq \lambda \leq 1, \ ema(t_1) = v_1$$

$$ema(t_{i+1}) = (1 - \lambda) \cdot ema(t_i) + \lambda \cdot v_{i+1}$$

9-3 - Exponential moving average or $ema()$ function with learning factor of $\lambda$.

Most of the literature about EMA uses the $\alpha$ symbol as for the smoothing factor, but here we prefer to reserve $\lambda$ for use as the learning factor variable.

### 9.4. Learning and Forgetting

Regardless of what you learn, good or bad, right or wrong, learning happens when you update a model of some kind. The update process is essentially adding something new and removing something old. Look again at Formula 9-3; when new data comes in, we learn from the new data at the rate of $\lambda$ and forget the old data at the same rate when we multiply it by $(1 - \lambda)$. If you choose $\lambda = 0.5$ then the 9-3 formula gets simpler, as shown in Formula 9-4, where the model learns half of the new data and forgets half of the data it knew before:
\[ ema(t_{i+1}) = \frac{ema(t_i) + v_{i+1}}{2} \]

9-4- EMA for \( \lambda = 0.5 \).

If you choose \( \lambda = 0 \), then Formula 9-3 formula becomes simplified even more and turns into Formula 9-5, in which the model stops learning after its first observation.

\[ ema(t_{i+1}) = ema(t_i) = \cdots = ema(t_1) = v_1 \]

9-5- EMA for \( \lambda = 0 \).

If you choose \( \lambda = 1 \), then, again, it is possible to simplify Formula 9-3 even more, as shown in Formula 9-6. In Formula 9-6, the model always uses the most recent value as its predicted value.

\[ ema(t_i) = v_i \]

9-6- EMA for \( \lambda = 1.0 \).

To get a better understanding of how \( \lambda \) affects the process of learning, consider a situation with an initial value of 100, in which you want to see how many times the model needs to observe a new value of 200 in order to make the prediction that the next data point that comes in is going to be 200.

As you see in Figure 9-7 for \( \lambda = 0 \), the model never notices that the second data point in the series, 200, is different from the first data point, 100, and it never predicts that the next value in the series will be 200. On the other hand, for \( \lambda = 1.0 \), the model exactly follows the data, so the moment it receives the new value of 200, it updates its prediction for the next value to 200.
Between these outer learning values, the system learns from the new given data at different rates. By choosing a particular value for the learning factor, we can build a fast or slow learning model or anything between them. Slow learning is generally a better strategy because, over the long term, it better captures the normal behaviour of a system. In other words, if you do not want your model to rapidly update its expectation whenever it receives new data, you must choose a learning factor close to zero, whereas, if you want a faster-learning model, pick a learning factor closer to one.

Both the moving average and exponential moving average require just two memory locations to keep track of a variable change, that is last calculated average and number of calculations or last calculated average and learning factor. However, when it comes to learning the different patterns of a variable through the time, we need more than just two memory locations or data structures.
10. Real-time anomaly detection

10.1. Problems with deal-time detection

We saw in earlier sections that, to detect anomalies in a time series, we need to learn the behavioural patterns of the system we are observing over time, and clustering helps us to approach this goal. If you run a search on the internet, you will see that most of the classic data mining or machine learning algorithms require access to the complete dataset. For example, most algorithms need you to gather all the data you want analyzed in a CSV file, and then, with some Python or R script or an API, you can run different clustering algorithms on the data and analyze the results. This is what many data scientists do every day. Most of these algorithms have an iterative part that requires them to go back and start doing something again, such as changing the center of each cluster. Let us see whether this type of model works for our time series.

Suppose you are monitoring the number of the transactions of a bank in a particular country to see whether anything is going wrong. The implemented sensor or system sends out new data containing the time and the number of transactions every second, so, you collect $24 \times 60 \times 60 = 86,400$ integer numbers per day. By the end of the first day you have 86,400 numbers, and by the end of the first week, you have 604,800 numbers, and 2,592,000 after one month, etc. You see even for a single measure, after a month you have a huge number of data to process real-time. In other words, there is no energy efficient, affordable or quick way to catch the anomalies of a system with algorithms that need to process all the stored data. We need to come up with a design for a system that can do this for us.

10.2. A System is required

Most of us are inclined to look for a single algorithm to fulfill all our goals for anomaly detection, but,

There is no single algorithm for anomaly detection!

By an algorithm, I mean any known clustering, classification or statistical measure. Instead, we need a system in which many different components work together to achieve our goals, and we need to use all the techniques we discussed before in that system.
10.3 How our brains find time series anomalies

Let’s take a look at the time series in Figure 10-1, which shows a single period of a measure containing 400 samples. When we look at this graph, we instantly notice something different, possibly an anomaly, around time = 50, but why and how is it happening?

We do not study or process all the information in a picture in a single glance, as we described before when looking at a painting. Instead, you may look at different parts of it one at a time, then step back, look at the whole again, zoom in on another part, and so on. The same thing happens when you study a graph, time series, etc. When you look at the graph in Figure 10-1, it is easy to divide it into three horizontal and four vertical sections, which gives a total of 12 different areas. Let us number them from bottom to top and left to right, from 0 to 11, as shown above. Now, when we look at these different regions, we find that:

- Area 0, 1, and 11 are empty.
- Areas that have data points in them each have an average value, for example, in area 5, the data values average around 16 and 17; in area 6, around 15, in area 8, around 21, etc.
- Each area with data points in it has either an upward or downward slope, for example, the data in area 4 slopes down, and, in area 5, it slopes up, etc.
- Some areas have maxima or minima in them, for example, maxima in areas 8 and 9, and minima at the border of areas 4 and 5, and at the border of areas 2 and 3.
- The overall shape of the line is smooth and has a continuous slope in all areas except for areas 4 and 8. In these two areas, we see jumps in value and dramatic changes in the slope of the line.
The process outlined above is called feature extraction because we use the available set of raw data points—their times and values—in each area to build information about, or extract, features of the data that can help us to understand the behaviour of the series. The amount of information we require for each area depends on the accuracy and precision we want to have for later use. In some cases, only the average value of each region may be important, whereas, if you want more detail to use in later comparisons, you may choose to extract not only the average value of the data, but also the slope, the locations of maxima and minima, and even the distribution of these values, etc., as we will discuss below.

### 10.4. More on feature extraction

When you look at the letter A, you do not remember the bitmap information that makes up the letter; instead you probably remember that the letter A has two short lines connected at the top making a caret-like symbol, and another, even shorter horizontal line in the middle. Characterizing the lines that make up the letter is more efficient than making note of all the individual pixels that make it up. Likewise, it is efficient to use vector elements to extract features from time series data.

If we build a $3 \times 4$ matrix with vector elements to show the extracted features of each area in Figure 10-1, it will look like this:

$$
\text{Features}(\text{area}_i) = [\text{avgValue}_i, \text{avgSlope}_i, \text{maxPointCount}_i, \text{minPointCount}_i]
$$

10-2- Sample features we can extract for each area of the series shown in Figure 10-1.

If we collect the information mentioned in 10-2 for each area of Figure 10-1, the matrix will then make up a high-level representation of the time series, which is enough to use to compare the data from Figure 10-1 with another time series’ area information and to calculate how much these two series are alike. All we need to do to compare the two series is define a distance function, the way we did in Section 7.2 above.

Collecting enough features lets you make comparisons of time series with whatever precision and confidence you want; even using just the four simple features shown in 10-2, it is possible to compare time series with very good precision and determine how close they are to the time series in Figure 10-1. If you would like more precision, you can add more features to the analysis. For instance, to distinguish between A and $\bar{A}$, you may need to extract line thickness or skewness as well.

Let us now turn to comparing the same area of two different time series, using the features of shown vector 10-2 as an example.

---

24 “Feature selection” is a related term, but it is used only when there are many measures, and you need to select the best of them for your work. In regular univariate time series, we only have the value of the measure.
10.5. Distance between two areas

Here, we introduce a distance function to compare two areas from two different time series. It is not used to evaluate the distance between a given point in a time series and available clusters or areas, but the idea behind defining the function is the same. Any function with a positive slope in its domain will have a distance function that complies with the conditions we discussed in 7-4. Here in 10-3 we have used Euclidean distance:

\[
Features(a_{1i}) = [avg_{1i}, slp_{1i}, mxc_{1i}, mnc_{1i}]
\]

\[
Features(a_{2i}) = [avg_{2i}, slp_{2i}, mxc_{2i}, mnc_{2i}]
\]

\[
dstnc(a_{1i}, a_{2i}) = \sqrt{(avg_{1i} - avg_{2i})^2 + (slp_{1i} - slp_{2i})^2 + (mxc_{1i} - mxc_{2i})^2 + (mnc_{1i} - mnc_{2i})^2}
\]

10-3- Defining a distance function to compare two areas from two different time series.

If two areas have the same vector values, then the distance function returns zero. It does not necessarily mean that the two graphs’ areas are the same, only that their features’ vectors are the same. As we mentioned before, the more independent features you extract, the more accurate and precise comparisons you can make, but increased accuracy and/or increased precision do not necessarily mean that a system works better. In artificial intelligence (AI) and machine learning (ML), this is an important concept; when you see your friends in the office and compare them with their models in your mind, you are 100% sure that they are your friends. But the fact is they are not physically 100% same as what they have been before. You would get a result of none of them are your friend if you could collect and compare thousands of features with high precision and accuracy. The more feature or higher precision you choose to use the more dimension and area you are facing in distance calculation, and this leads to a phenomenon called the curse of dimensionality we will talk about later.

10.6. An idea for real-time comparison

We will discuss the process of real-time anomaly detection later, but for now, we know enough to understand the core concept of our ADE. First, we will need to build a model of the time series we want to monitor. Let’s say we collect 30 days of its data and build a matrix with elements containing each area’s features. If we divide the vertical axis into 10 units and a single day into 6 units of 4 hours each, then we have \(10 \times 6\) matrix for each day. For our model to more accurately reflect the different patterns of data we obtain each day of the week, we set up seven different matrices, one for each day of the week. These seven matrices can be considered as seven elements in a single row matrix in our model of the time series, as shown in Equation 10-4:

---

25 It means when objects physically get closer, the distance returns less value no matter if it is linear.
weeklyModel = \[
\begin{pmatrix}
    d_{1,1} & \cdots & d_{1,6} \\
    \vdots & \ddots & \vdots \\
    d_{10,1} & \cdots & d_{1,6}
\end{pmatrix}, \ldots, \\
\begin{pmatrix}
    d_{7,1} & \cdots & d_{7,6} \\
    \vdots & \ddots & \vdots \\
    d_{7,10} & \cdots & d_{7,6}
\end{pmatrix}
\]

10-4- Weekly model for a time series. Each element represents a single day’s model.

Now when we get a new data point, we find the index of the element in the seven-element row of the matrix that corresponds with the relevant day of the week, and, based on the time element of the data, we can point to the corresponding column of the $10 \times 6$ matrix. At this point, we now have a vector column of 10 different areas. We must figure out which of these 10 areas is where the given value belongs. If it belongs to an area in which the model usually experiences such a value, the point is not an anomaly, but if it belongs to an area where the model rarely or never has experienced anything similar, then it is unusual or an anomaly.

Although it can work well in special cases, this model is a little too simple for everyday life. In most cases, a single point is not enough to extract the features we will need to detect anomalies. We will introduce a method to address this issue in the next few chapters.
11. Window processing

11.1. Processing a Time Window

As we discussed in chapter 10, we usually do not have enough time, processing power, or even memory to process all the information we have received before a new data sample of the time series comes in. In order to operate efficiently, we must keep only the minimum amount of information we need and, even then, process only a minimal portion of that, and never look back.

Many features we need to extract and learn from in a time series cannot be calculated from the single and most recently received sample alone. However, the features that help us learn the behaviour of a time series—or any other system—do not need to be extracted from the entire life of the series. Consider the time series in Formula 11-1. If we accept that we are allowed to extract only features based on a time window which ends at the last given sample, then all we need is a buffer to keep the data from this time window.

\[
\text{time series: } d_1, d_2, d_3, \ldots, d_{i-k}, d_{i-k+1}, d_{i-k+2}, \ldots, d_{i-1}, d_i, \ldots
\]

\[
\text{Features}(t_i) = F(t_i, d_{i-k+1}, d_{i-k+2}, \ldots, d_{i-1}, d_i)
\]

11-1- A sample time series with features we can extract from only the latest \( k \) samples.

So, for a time window with a length of \( k \), we just need to keep the \( k \) most recent samples, and no more. And at any time, the extracted features, or it is better to say, feature vector is a function of the time and the \( k \) most recent samples.

11.2. An analogy

I never understand complex mathematical models or algorithms until I find a proper analogy for them in daily life. For example, a time series is a dynamic system because its only measure changes over time, and human behaviour also changes over time, so that you cannot judge someone’s personality just by capturing a single instant’s worth of information. It is only possible to make an assessment after gathering data for a while, processing them, extracting different patterns, seeing a behavioural pattern, and comparing it with already known behavioural patterns.

Note that the time window we extract features from is different from the vertical divisions we used to build the areas in Figure 10-1. For example, if you want to analyze your behaviour over a week by logging your behaviour every half an hour, the feature extraction window would be half an hour, but you may use divisions (or windows) of six hours each—that is, early morning, morning, afternoon, and night—to divide the day in four. After collecting data for a while, you can build a model that shows the probability of each of your behavioural patterns occurring on any day of the week in any of those four time periods, as outlined in 11-2:
Sample entry of a behavioural model for Monday afternoon.

The complete model for 11-2 will have $7 \times 4$ areas or better to say clusters. For each day and chunk of time, we should have a probability distribution of behaviours that looks something like 11-2. The same model works for time series, too; if you take another look at Figure 10-1, you can see that, for each area, we can build a probability distribution of possible feature values showing what our expectations of the feature values are. So, for the graph in Figure 10-1, we might use this model:

$$Behaviour(day = \text{Monday}, \text{time} = \text{afternoon}) = \begin{cases} 
p(\text{angry}) = 0.1 \\
p(\text{happy}) = 0.53 \\
p(\text{bored}) = 0.2 \\
p(\text{delighted}) = 0.1 \\
p(\text{cranky}) = 0.07 
\end{cases}$$

11-3- Sample feature information we can extract for each time region of a time series.

Extracting the features mentioned in 11-3 is not hard because we have access to the information we need, namely, the average value, the number of peaks, and the slope of the graph in each division of time.

11.3. Time series as the motion of an object

If we consider any given time series to be the distance traveled by an object over time, then any pair of data point with a time and value component can be considered as the distance this object has traveled ($d_i$) by $t_i$. From classical mechanics, we know that the speed of an object can be defined as shown in Equation 11-4, in other words, as the change in distance divided by the change in time. If we assume a time quantum as the unit of time, then, for two recent sample data points, the amount of the change over time is the unit of time:

$$v(t_i) \approx \frac{\Delta d_i}{\Delta t_i} = \frac{d_i - d_{i-1}}{t_i - t_{i-1}} = d_i - d_{i-1}$$

11-4- The slope of a time series at any point can be described as a velocity.
As shown in Figure 11-5, when the time series is trending up, the velocity is positive, and when it is trending down, the velocity is negative. At maximum or minimum points, the velocity is zero.

A time series (solid line) can be thought of as the distance traveled by a moving object, and its first derivative (dashed line) as the speed of the object.

You can also see that a small change in the overall pattern can have a large impact on the velocity. Just by considering the sign of the velocity, we can find out if the graph is moving up or down. We can also use the second derivative of the distance, or first derivative of the velocity with respect to time, to get an idea of the acceleration, or the curvature of the time series.

The curvature of a time series at any point can be described as its acceleration.

Figure 11-7 shows the sign of the calculated second derivative based on the formula introduced in 11-6. As shown below, the sign clearly shows the moments that the series is convex or concave.

---

26 Higher order derivatives also give more information on the dynamics of motion.
As shown in Figures 11-5 and 11-7, both slope and curvature give us information about the behaviour of a time series. When we consider them along with the average value of the data, these three features can give us sufficient information to process most of the behaviours exhibited by smooth curves.

11.4. Defining anomaly

So far, we have addressed equations that can help us detect anomalies in smooth curves, but what should we do if our data is spiky like the data in Figure 2-5 or even 4-10? As it turns out, we can use the above equations to learn the behaviour of such series, depending on the definition of the anomaly. For the sample data in Figure 11-7, we can implicitly assume that any behaviour that changes the average value, the smoothness, the slope or the curvature of the series dramatically, is an anomaly, so these techniques would work.

However, if our time series is like the left picture shown in 2-5, which generally has some fluctuation, we may assume that the important features are the average value of the series, the average number of ups and downs per time region, and the curvature of the smoothed series. In a case like that, the ADE should have an idea of how many ups and downs the graph usually has over an any given time interval and report back if it changes dramatically. Or, for the traffic trends of a small network like the one shown in 4-10, perhaps the only thing that matters is the average value of the data, and nothing else.

The best way to help anomaly detection work better is to define relevant features based on your understanding of what anomalies are in the given system.
12. Anomaly index or anomaly score

12.1. Anomaly index for a single feature model

Back in Chapter 2, we discussed the relationship between the outcome of an experiment and anomalies in a coin flipping experiment. An anomaly index, also known as an anomaly score, is a useful way to characterize how much an outcome of an experiment or observation is anomalous. Suppose we run an experiment which generates the outcomes described in the set $O$, shown below:

$$O = \{o_1, o_2, o_3, \ldots, o_i, \ldots, o_n\}$$

12-1- An experimental outcome set.

When an anomaly occurs, it is an observation outcome with a low probability, so we can define the raw anomaly index of a system at the time we experience an outcome, such as $o_i$, in this way:

$$raw\text{AnomalyIndex}(o_i) = 1 - P(o_i)$$

12-2- Anomaly index of an outcome as the probability of something other than the outcome happening.

It can also be defined based on the ratio of the probability of the outcome divided by the maximum probability of any possible outcome:

$$raw\text{AnomalyIndex}(o_i) = 1 - \frac{P(o_i)}{Max(P(o_j))}, j \in O$$

12-3- Anomaly index of an outcome as a ratio of probabilities.

By the definitions introduced in 12-2 and 12-3, the smaller the probability of an outcome, the bigger the anomaly index. So, when there is a very low probability of something happening—like the probability of a flipped coin landing on its edge—its raw anomaly index is very close to 1.

We call the index raw because, for the coin flipping experiment, the probability of a coin landing either on heads or tails is close to 50% in each case. It is better to have an index value of zero when the outcome is not an anomaly. As we discussed in Chapter 2, we need to define an interval for our anomaly index in order to consider whether an outcome is an anomaly. Once we define an interval (or a series of intervals), we can map the anomaly index in 12-2 to it to get a better understanding of how likely the outcome is to be an anomaly:
\[ 0 \leq lt \leq \text{rawAnomalyIndex} < ut \leq 1 \]

\[
\text{anomalyIndex} = \begin{cases} 
0, & \text{rawAnomalyIndex} < lt \text{ or rawAnomalyIndex} \geq ut \\
\frac{\text{rawAnomalyIndex} - lt}{ut - lt}, & lt \leq \text{rawAnomalyIndex} < ut 
\end{cases}
\]

12-5- Anomaly index definition based on the lower and upper thresholds, \( lt \) and \( ut \), respectively.

We can also use logarithmic transformation functions to highlight what happens to an anomaly index when it gets close to 1 or when it starts to grow from zero.

\[ \text{logarithmicAI} = -\frac{\log_{10}(1 - \text{anomalyIndex})}{2} \]

\[ \text{logarithmicAI} = 1 + \frac{\log_{10}(\text{anomalyIndex})}{2} \]

12-5- Two different logarithmic transformation functions for anomaly index value between zero and one \((0,1)\).

As we mentioned earlier, defining an index for a particular kind of anomaly is up to you; it just has to measure the anomaly in question.

12.2. Anomaly index for a model with multiple features

When you are dealing with multiple selected or extracted features, you can either consider any one of them as a single feature on its own or you can look at the combination of them as a single new feature. If you use a combination, you could use Formula 12-2 or 12-3 to calculate the single anomaly index for whole features, but I don’t recommend it—the number of combinations grows exponentially as the number of your dimensions increases. (Even if you were to choose broad or large clusters, you would still have to deal with many of them. This situation is called “the curse of dimensionality”).

Alternatively, you can process each feature on its own and then make your decision based on the anomaly index. To understand this process better, consider this analogy; when you see someone on the other side of the street and want to make sure she is your friend, you usually do some measurements and comparisons. These measurements and comparisons are like feature extraction (or selection) and anomaly index (or probability) calculations. However, how do you get the result based on these calculated indices?
You might gather the following information from your observations:

1. Her hair style is almost like my friend’s, but the color is not.
2. She is as tall as my friend.
3. She is dressed like my friend.
4. She looks a bit bigger than my friend.
5. I spoke with my friend on the phone this morning and she said she’s on vacation in another city.

Based on your comparison, you could then generate an anomaly index for each component of this information, but for an overall result, you need to combine these four numbers:

1. You could average the anomaly indexes of the components. If you find that the average is low, you can assume that the person is your friend.
2. You could consider the largest number of all the component anomaly indexes to be the most important. If it is higher than your expectation, the person is not your friend.
3. Alternatively, you could consider using a weighted average of the component anomaly indexes because the four data points you gathered are not equal in informational value. For example, since your friend recently told you that she was on vacation, you would probably assign this piece of data a much higher weight than the others.

All of these methods are easy to implement, and you can modify them or build your own reasoning method to calculate the result for your system based on your business needs. You may even use a neural network black box (which I do not recommend!) and train it to learn how to report anomalies based on the given anomaly indexes, which takes far less time than you might think if the input data is binary and there is a single output. Here are mathematical formalization of the described reasoning process.

\[
\text{anomalyIndex} = \frac{1}{n} \sum_{i=1}^{n} \text{anomalyIndex}_i
\]

\[
\text{anomalyIndex} = \sum_{i=1}^{n} \text{weight}_i \times \text{anomalyIndex}_i , \; \sum_{i=1}^{n} \text{weight}_i = 1
\]

\[
\text{anomalyIndex} = \max(\text{anomalyIndex}_i) , \; i \in \{1..n\}
\]

12-6- Your business needs can be used to define how to combine partial anomaly indexes to get the index for the whole system.

We almost have discussed all the required concepts to build our ADE; there are only two more topics left, coarse coding and distribution and behaviour comparison.
13. **Coarse coding**

13.1. **Introduction**

It is hard if not impossible, to judge people accurately over a short period of time; you need to get to know them for a while. Otherwise, your observations give you only shallow knowledge that might not exactly reflect the behaviour they are capable of. The same is true for any other system.

Consider the traffic usage of a computer network that usually has its daily peak around 11:30 pm. If you build your model with a low time resolution, it might generate an anomaly alarm if the peak happens at 11:40 pm, which would not be correct, at least for a variable like network traffic. When you look at the network’s traffic, a more important question is whether the difference between 845 MB and 860 MB is significant when your traffic goes from a daily minimum of around 500 KB up to 1 GB. The answer is that the difference is not significant at all.

Almost in every system, if we use the wrong measurement or calculation units, then our comparison potentially could produce false positive or false negative results. So, we need to make sure to choose an appropriate unit of measurement.

13.1. **The power of coarse coding**

Let’s say we want to locate the position of a point in a coordinate system, like the one on the left in Figure 13-1, with a resolution of U:

![Figure 13-1](image)

13-1- How we can get a result like a high-resolution Cartesian coordinate system with two or more low-resolution coordinate systems.

---

27 Also, known as “coarse classification.”

28 This is the width of the region not the feature extraction window.
To find the position of the point, you need to measure the coordinates of that point with a tool of a resolution of no more than U. In the left picture of Figure 13-1, the point is at (2,7). In the middle pair of coordinate systems shown above, the resolution unit for both X and Y coordinates is 2U. It is evident that measuring a distance with a tool with a resolution of 2U is much easier than doing it with a tool with a resolution of just U. However, we can still locate the same point with a resolution of U by locating the point in two coordinate systems at once, like this: \{ (2,3), (1,4) \}.

The same logic works for the system shown to the right; it is in 3U units, but it is still possible to locate the point with a resolution of U while using a measurement of 3U by describing the point as \{ (1,3), (1,2), (1,3) \}.

With these three systems, we demonstrated how to locate the position of a point with a certain resolution using either a high-resolution measurement tool and system and measuring once or using a lower resolution measurement tool and system but doing multiple measurements. The idea works almost everywhere, and, in fact, it is something our brain uses every day: you never recognize your friends by examining one unique thing about them, such as their complex fingerprint patterns, or by measuring their exact weight, height, RGB values of their skin color, and so on, do you? Instead, without even thinking about it, you simply collect information about 10 or 20 features from every one of them and store some approximate values of those features in your memory. The combination of these low-resolution measurements can give you the same accurate result.

We can and should use coarse classification in our ADE application—it is almost the only way we can find anomalies with enough accuracy and precision. We cannot count on any one single sample value and consider it to be an anomaly if it goes above or below a threshold because it might be a device error or a hazard instead. If we did that, we’d end up with false positives. We can only believe it to be an anomaly if we find enough other supporting evidence.
13.2. Coarse coding for feature extraction

If you receive a sample every 5 minutes, you will get 288 samples per day; and as we talked about window processing, if you extract your features based on last half hour of data or 6 recent samples then you have to wait half an hour for features update, which is not good in many cases.

But there is another way to have updated features as every new data comes in, you just keep a copy of last 6 data. Figure 13-2 represents this idea by showing sample blocks along the top and processing windows as bars underneath them. The overlapping of processing windows over time, as shown in Figure 13-2, has another benefit; it helps to maintain any possible shift in time when something happens near the borders. Any event close to the left or right border of a given time window is in the middle of at least one other window, allowing you to process any change in a time series so long as you have access to the data from immediately before and after it.

13-2- Using the previous 30 minutes of data to extract features while new data comes in every 5 minutes.
14. Distribution and behaviour comparison

14.1. Distribution

In our discussion of time series, distribution means the density of the points in each horizontal unit per vertical time unit. Figure 14-1 shows one of our earlier examples in Chapter 9. You can calculate the vertical distribution of the points in every time column as shown, so that these five numbers define the expected probability of finding time series data in each coarse time unit in the future.

![Vertical Distribution of data](image)

14-1- Vertical distribution of data for a time series. The numbers in each of the five vertical segments of each column represent the likelihood that data points will occur in each one in the future.

The sum of the probabilities for any column is one and as we discussed in Chapters 10 and 11, we can build the above distribution not just for values of a time series, but for any other extracted features; the vertical axis could also show slope, curvature, the maximum number of points, or any other features extracted from a time series.

Now the question is, if your model has a distribution of, say, [0.00, 0.20, 0.40, 0.40, 0.00] and your recent data distribution shows [0.00, 0.20, 0.45, 0.35, 0.00], how much are these two distributions alike? As we discussed earlier, you can use any distance function to compare the two distributions because you can simply consider them as two different points in a multidimensional vector space and calculate the Euclidean distance between them, or you could use other methods. Before explaining how these methods work, we need to review the meaning of entropy.
14.2. Entropy

Entropy can be used to show changes in the dynamics of almost every system. It illustrates the degree of disorder or uncertainty in the physical world or the world of information. In the physical world, for example, the entropy of gasses is greater than the entropy of liquids, and fluids have more entropy than solids, the reason is the molecules of solids are more ordered than liquids and liquids are more ordered than gasses. The same is true in the world of information; the outcome of flipping a coin is one of two possibilities, either heads or tails, so its entropy is lower than the entropy involved in tossing a six-sided die, which has more possible outcomes.

We can use Shannon’s information theory to quantify the entropy of a system. It says that, if the states of a system or the possible outcomes of an experiment can be expressed as $s_1, s_2, s_3, \ldots s_n$ the following formula yields the entropy of the system:

$$H = - \sum_{i=1}^{n} P(s_i) \cdot \log_2(P(s_i))$$

14-2- Entropy of a system with $n$ states or an experiment with $n$ different possible outcomes.

Using Shannon’s information theory, the coin example with its two different possible outcomes is a system with two states, so only one bit of information is needed to keep track of them. For a six-sided die, though, you would need $\log_2(6)$ or 2.58 or 3 bits of information to specify the state of the system.

Entropy itself can be used as a feature to extract from a recent window of a time series. The ability to show how much order or disorder is in the data in a time window can be used in anomaly detection systems. For example, if you have a feature which demonstrates the entropy of port usage in a network, a DOS or even a DDOS attack will cause a decrease in entropy because port usage during the attack will become more ordered.

---

29 Claude Elwood Shannon (April 30, 1916–February 24, 2001) was an American mathematician, electrical engineer, and cryptographer known as “the father of information theory.”
14.3. Information gain

We can use information gain to find the difference in entropy between two distributions. Information gain is not a distance function, but we can use it as a measure to show how much a distribution differs from the ideal or model we expect. This measure is also called the Kullback–Leibler divergence. So, if you have a model distribution of $P$ and a newly observed distribution of $Q$, you can calculate the information gain via the following formula:

$$D_{KL}(P||Q) = \sum_{i=1}^{n} P(s_i) \cdot \log_2 \left( \frac{P(s_i)}{Q(s_i)} \right)$$

This is the Kullback-Leibler divergence for the model distribution of $P$ and observed distribution of $Q$.

14.4. More on patterns comparison

Consider you are collecting information on how you spend your time when you are working on the computer. Sometimes you use a browser to read the news, or search for information, check the stock market, do your banking, or watch YouTube—or you might use a word processor or spreadsheet at work, etc. You can build a daily distribution pattern for any one of these tasks over different times, as we described earlier. You would then have ten or twenty different daily distributions which shows how you spend your time with any of the aforementioned applications.

You could also look at this scenario from another angle and find some correlations between these distributions or application usage patterns. Consider a time unit of half an hour; in some of these time units, you might use the word processor 100% of the time; or spend 50% of your time with the word processor and 50% using a spreadsheet; in others, you might spend 60% of your time on YouTube, 30% banking and 10% web surfing, etc. Any of these combinations can be called one of your working behaviour patterns.

With this definition, working behaviour has more than one dimension that you can extract from the data, such as what we can have for a time series like:

1. $F_1$: average level of the data in window { very low, low, medium, high, very high }
2. $F_2$: average slope in window { uphill, downhill, flat }
3. $F_3$: fluctuation in window { high, medium, low }
4. $F_4$: curvature in window { convex / flat / concave }

Using these features to model your time series, the result of feature extraction will resemble 14-4, in which the values of $f_{1i}, f_{2i}, f_{3i}$ or $f_{4i}$ belong to the corresponding sets defined as $F_1, F_2, F_3,$ and $F_4$.

30 Solomon Kullback (April 3, 1907–August 5, 1994) and Richard A. Leibler (March 18, 1914–October 25, 2003) both American mathematicians.

31 You can skip this section if you are not interested in multivariate time series.
\((t_1,[f_{1,1}, f_{2,1}, f_{3,1}, f_{4,1}]),(t_2,[f_{1,2}, f_{2,2}, f_{3,2}, f_{4,2}]), \ldots (t_i,[f_{1,i}, f_{2,i}, f_{3,i}, f_{4,i}])\)

14-4- Sample time series data after extracting four features.

Like in the computer usage example above, you can build pattern distributions for any time window for the observed values of a single feature, as in 14-5 which shows the recent \(k\) feature number 3 values. And the result of these \(k\) values distribution is going to be something like 14-6, which shows the probability of each possible outcomes.

\[ \{ f_{3,i-k+1}, f_{3,i-k+2}, \ldots f_{3,i-1}, f_{3,i} \} \]

14-5- A sample single feature observation in a window of size \(k\).

\[ D_{3,i} = [P_{3,i}(low), P_{3,i}(medium), P_{3,i}(high)] \]

14-6- A sample distribution pattern for a window of size \(k\) for feature number 3.

You may call 14-6 as a behavioural pattern, because it shows and could predict the way feature number 3 might change. After a while, you will have different distributions which show different behavioural patterns of this feature for different time units. You will probably see hundreds of patterns; many of which will be basically the same but with small differences. Now, by clustering these different behavioural patterns, you can limit the number of behavioural patterns you need to work with, and make it easier to find high-frequency behavioural patterns in the time series, or when they are likely to happen.

You can also extract sequenced patterns from these clusters of distributions to build more long-term behavioural patterns like the one shown in 14-7, or use a vertical combination of different clusters of different features to build hybrid behavioural patterns like 14-8\(^{32}\), this one has a better meaning if we use features from different measures. The point is that you only need some basic methods of pattern extraction, classification and clustering to analyze or understand how a time series or any other system works. We will talk about how to apply these techniques in practice in the next chapter.

\(^{32}\) Note shown patterns in 14-7 and 14-8 are different. The first is derived from same feature patterns and the second is derived from the combination of different features pattern.
$[C_{3,1}, C_{3,15}, C_{3,11}]$

14-7- A sample sequenced cluster pattern for a window of size $3k$. Each of these clusters $C_{i,j}$ can represent many distributions that are alike.

$[C_{1,8}, C_{2,11}, C_{3,10}, C_{4,6}]$

14-8- A sample vertical pattern for a window of size $k$.

Figure 14-9 gives a about the process of extracting behavioural patterns from multiple features.

14-9- The process of extracting behavioural patterns from multiple features
15. Time series anomaly detection algorithm

15.1. Assumptions and scenario

There are two separate things we need to work on to build our ADE: first, the detecting algorithm, and, second, the design of a machine that gets the data, runs the algorithm, and gives the results. Before starting to work on the details of these two parts, let us review the scenario of how this ADE is expected to work and the assumptions behind it. We have discussed these assumptions before, and we should now be familiar with them:

1. The processing data is a regular univariate time series.
2. We need to have the following configuration data organized before running the algorithm:
   a. The longest period, for instance, a day, a week, a month, etc.
   b. The sampling time space, for example, 60, 120 or 1,800 seconds.
   c. A ratio used to scale down (or up) the value of the time series, such as 1,000:1 (or 1:1,000) to help our clustering or classification work on a sensible range of numbers for coarse classification.
   d. The time unit over which we want to analyze the behaviour of the series, for coarse classification, for instance, half an hour (1,800 seconds), 4 hours (14,400 seconds), or 6 hours (12,600 seconds), etc.
   e. A predefined range over which the calculated anomaly index is considered to indicate an anomaly.
   f. Finally, a learning factor to allow the algorithm to be set as a fast or slow learner, or anything in between.

3. The ADE should also fulfil the following requirements:
   a. It should learn from a given data set and constantly update its model unless we force it not to.
   b. The process of updating the model or calculating the anomaly index should continue even if nobody asks for a report.
   c. The data generating system should push updates to the ADE.
   d. The system should process data only on the fly and should not be able to use stored historical data for real time processing.

The configuration parameters depend on what the ADE will be used for, but if you have access to enough data before starting the ADE, you can calculate most of these initial parameters. For now, just consider these basic setup parameters are already defined.
15.2. Basic Structures
Consider the time series as shown in 15-1, in which any given data in the series has a time stamp and value, and the updating interval is $q$.

\[ d_1, d_2, ..., d_i, ... \]
\[ d_i = (t_i, v_i) \]
\[ q = t_i - t_{i-1} \]

15-1- Time series specification.

You need to maintain a list of the last $k$ element of the data, as in 15-2, so that you can extract features at any given time.

\[ recentList_i = [d_{i-k+1}, d_{i-k+2}, ..., d_i] \]

15-2- recentList is a list containing the last $k$ given data points.

With this definition, you can extract features at any given time, as shown in 15-3. When you write your ADE program, the return value of the functions could be of any type, just note if you extract $m$ features, you will need $m$ different feature extraction scalar functions, such as extracting, value, slope, maximum count, curvature, etc.

\[ FeatureVector_i = FV(recentList_i) \]
\[ FeatureVector_i = [f_1(recentList_i), f_2(recentList_i), ..., f_m(recentList_i)] \]

15-3- All extracted features can be shown as a single vector as a function of the list of recent data points.

Now you must decide which features will be the most useful to have and which will work best in your context. Here we introduce some features you can use in your ADE.

15.3. Value feature function
The first feature we usually need to extract is the average value of the series at the current time. As we described in the configuration requirements, we need to have a ratio that tells us how to scale the values so that we can use them in coarse classification. Let us assume there is a constant such as $vsr$ (value scale ratio) that, when you multiply it by the given value, the result is the scaled value. The scaled value can be calculated using 15-4, supposing that $floor()$ function returns the integer part of the passed argument.
\[ levelFeature_i = \text{floor}(recentList_i[k] \times vsr) = \text{floor}(d_i \times vsr) \]

Equation 15-4- A function that returns the scaled integer part of given value.

Note that \( recentList \) is a vector with elements from one to \( k \), and the \( i \) index shows the state of this vector at time \( i \). Equation 15-4 may return negative values, depending on the data input which is not a deal but if we somehow already knew the minimum possible value of the series, we would add an offset to the result so that every value that is returned would be greater than or equal to zero, but the truth is that we do not care if they get negative. The algorithm is still going to work because the returned numbers are used only to identify that feature value, so we can keep it simple.

We may need to consider smoothed average values of the series if the series has continuous fluctuation. In this case, 15-5 is the solution, in which \( sws \) (smoothing window size) is the number of entries in the list we use to calculate the average. If we choose \( sws = 5 \), we just need to calculate the average of the most recent 5 points, scale it, and then return its integer part as the smoothed value feature.

\[ \text{smoothedLevelFeature}_i = \text{floor}\left(\frac{1}{sws} \sum_{j=k-sw+1}^{k} recentList_i[j] \times vsr\right) \]

Equation 15-5- A function that returns the smoothed value of the series.

Alternatively, any other smoothing method can be used here.

**15.4. Slope feature function**

The slope feature gives us an idea of in which direction the series is moving, so it is not necessarily the mathematical implementation of slope. The simplest way to determine which way the series is moving is by using the \( \text{sign}() \) function, which returns -1, or 0, or +1, depending on whether the newly received data is lower than, equal to, or higher than the last received data point.

\[ \text{slopeFeature}_i = \text{sign}(recentList_i[i] - recentList_i[i-1]) \]

Equation 15-6- Very simple slope feature extraction function.

We can have a better idea of which direction the series is moving in by calculating the slope over a longer period, such as in 15-7, in which \( lws \) is the slope window size.

\[ \text{slopeFeature}_i = \text{sign}\left(\frac{1}{lws} \sum_{j=k-lws+1}^{k} recentList_i[j] - \frac{1}{lws} \sum_{j=k-2 \times lws+1}^{k-lws} recentList_i[j] \right) \]

Equation 15-6- A slope feature extraction function over a wider time window.
Instead of using the \textit{sign()} function, we can use our own function to return a higher resolution for the slope. For example, when you move from point a to point b, the function in 15-7 can give you better information about how fast the series is going up or down:

\[
slope(a, b) = \text{floor}(\arctan(|b - a|) \times 20) \times \text{sign}(b - a)
\]

15-7- A better slope function which returns different values.

As mentioned earlier, we are not interested in high-resolution feature values because we want to use information from many features at once. Figure 15-8 shows the result of using Formula 15-7 for one of our previous examples.

Figure 15-8 shows that, regardless of the average value of the series at any time, depending on how fast the series goes up or down, the slope follows the series’ slope behaviour within a limited range of digitized values. The resolution of the steps is up to you—you can modify it by changing the factor that increases the arc value in Equation 15-7.
15.5. Local maximum and minimum feature function

In Figure 15-9, the series has around 5 to 6 minimum points every 50 samples. To learn the behaviour of this series, we need to extract a feature which shows the number of local minima or maxima per feature extraction window. Equation 15-10 works for this purpose. Note that $mws$ is the minimum window size in which we look for local minima.

$$isMinimum(rcntList_{t,j}) = \begin{cases} 1, & (rcntList_{t[j-1]} > rcntList_{t[j]} \text{ and } rcntList_{t} < rcntList_{t[j+1]} ) \\ 0, & (rcntList_{t[j-1]} \leq rcntList_{t[j]} \text{ or } rcntList_{t} \geq rcntList_{t[j+1]} ) \end{cases}$$

$$minimumCount_{t} = \sum_{j=k-mws+2}^{k-1} isMinimum(recentList_{t,j})$$

15-10- Formula to calculate number of local minima in a window.

We can also use the same idea shown in 15-10 to extract the number of the local maxima in a window. It is good to get a sense of how Equation 15-10 can be used to increase our understanding of time series before we continue discussing the algorithm. Whenever a new data point comes in, the ADE’s algorithm calls the 15-10 function to determine how many local minima exist in the $mws$ window. After a while, it understands, via the distribution we talked in Chapters 6 and 14, that there is a particular probability that there are 4 or 5 or 6 of them.

Now, if in one of the calls it returns 10 or 0, the ADE looks to the historical model and finds that this number of minima has never been experienced before, so it assumes that these values are potential anomalies. The same applies to the local maxima: if an ADE observes that usually there are no local maxima, when it observes some around time = 551, it assumes that they are potential anomalies.
15.6. Should we decompose our time series?

Whether or not we should decompose a time series depends on our needs, the behaviour of the time series, and the amount of the historical data we have access to, as we discussed in Chapter 1 with respect to trends, periodicity and residuals.

For example, to remove the trend from a series, you need to have access to at least the average value of the series over one or two of the longest periods. (Refer back to Formula 5-8 for how to update the average of the last N samples without keeping all of the previous N samples in memory). For a typical time series, such as network traffic usage, city water usage, or electricity usage that has patterns over days, weeks, and even months, you need to have one or two years of data. Then all you need to do is subtracting the last year or two years’ averages from any given data set. Doing this has no effect on irregular or seasonal components because technically over that long time the average of both should be zero. Alternatively, if you want to remove the monthly patterns from the series, you need to subtract the average of the last two or three months of data from any new data sample.

But since the method we are using is going to learn from the data and adapt itself to long-term trends, we do not need to decompose the series. In fact, AI does not use decomposition. Instead, the algorithm is going to learn the behaviour of the time series, and since it keeps its focus on recent data, the long-term trend does not have any effect on the algorithm. The distributions of features gradually adapt to the long-term trend, so the model changes gradually. It is like how you may not pay attention to the changes that happen to your face over many years, but you notice even a small scratch on your face when you look in the mirror.

15.7. How many features do we need?

The number of features we need for our ADE depends on the time series we are observing. The features described above—value, slope, and how many maxima or minima there are—are almost enough for most series, though they need to be fine-tuned based on the behaviour of your particular series.

However, if you think you might need more features, you can write additional feature extraction functions. There are many features you can extract, including the entropy of a series, or you could compare what happens in one time window with what happens in other, predefined windows, etc.

For example, if you want to extract the sum of values in a particular time window, then all you need to do is write a function that returns the sum of the values in the desired window (\( \text{aws} \)) like the one shown in Equation 15-11. The sum of values in over a time period is useful for monitoring series that have irregular fluctuations, like the series shown in Figure 15-12.

---

33 But a friend who has not seen you for years would notice the all these changes because his mental model of your face has not been updated during that time.
\[ accValue_t = floor\left( \sum_{j=k-\text{aws}+1}^{k} \text{recentList}_t[j] \times \text{vsr} \right) \]

15-11- A feature extraction function to return the sum of recent values over a period of time (\text{aws}).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{time_series.png}
\caption{A time series with irregular fluctuations and spikes.}
\end{figure}

15.8. The Structure of the model

Now that we have our feature extraction functions, we can calculate a feature vector as described in 15-3. The next step is to build and update our model when new data comes in. As we mentioned earlier in Chapters 9 and 14, there is no single model good for all purposes, as different people think in different ways when they try to solve the same problem. However, the first thing we need to do is splitting the longest time period of the series into shorter time segments and study the behaviour of the series in each of these windows.

My experience with different companies shows that a period of a week works for most business measurements. Because a week includes seven different days, and we generally choose time windows of 2, 3, 4, or even 6 hours, our model could describe the specific behaviour of the series over each day of the week, and, within each day, in 12, 8, 6 or 4 different time windows. So, if we consider 6-hour time windows, then, for a week, we would have \(7 \times 4 = 28\) different windows. To describe our model, let us consider that:

1. The series takes one sample per minute.
2. The recent feature extraction window size is \(k = 60\), so we have access to last hour’s data, and we store it in a vector with element indexes from 1 to 60.
3. We extract only two features, as in the derivation below (15-13):
\[ \begin{align*}
d_1, d_2, \ldots, d_i, \ldots \\
q &= t_i - t_{i-1} = 60 \\
rcntLst_i &= [d_{i-59}, d_{i-58}, \ldots, d_i] \\
lf_i &= \text{smoothedLevelFeature}_i = \text{floor} \left( \frac{1}{3} \sum_{j=58}^{60} rcntLst_i[j] \times vsr \right) \\
sf_i &= \text{slope}_i = \text{floor} \left( \frac{1}{59} \sum_{j=2}^{60} 20 \times \arctan(|rcntLst_i[j] - rcntLst_i[j]|) \right) \\
\times \text{sing}(rcntLst_i[j] - rcntLst_i[j]) \\
FeatureVector_i &= [lf_i, sf_i] \\
\end{align*} \]

15-13- Deriving a two-dimensional feature vector from a time series.

The derivation in 15-13 shows a way to extract a new feature vector every minute, that is, whenever new data comes in. It calculates the average of the slope over the last 60 minutes, so the very first vector is ready only after we receive the 60th data point. Until then, we cannot extract any features from our time series. The model is a 28 \times 2 matrix, with 28 columns for different time windows over the week and two different distributions for each feature in time window. In model \( M \) in 15-14, \( LD \) is a value feature probability distribution, and \( SD \) is slope probability distribution.

\[
M = \begin{bmatrix}
LD_1 & LD_2 & \ldots & LD_{28} \\
SD_1 & SD_2 & \ldots & SD_{28}
\end{bmatrix}
\]

15-14- A vector model for a weekly pattern with 6-hour time windows.

\( LD \) and \( SD \) are vectors too, and since we have no idea what the time series data is in this example, we do not know how many elements these probability distribution vectors have, but because of the nature of our feature extraction function, the result vector should have a small number of elements. So, for example, if we start counting our days from Monday (with Monday=1, Tuesday=2, Wednesday=3, etc.) and data comes in at the unixtime equals to 1,483,291,330, we see it is on the 4th day of the week, that is, Thursday. And since the corresponding time is 17:22:10, it is in the 3rd daily time window, or its weekly index is \( 4 \times (4 - 1) + 3 = 15 \), so we have to get information from \( LD_{15} \) and \( SD_{15} \).
Figure 15-15 gives you an idea of how the 15-14 vector model looks like, and finally note that building and updating the distribution of elements of $M$ is necessary for our system to learn. We therefore need to use a learning factor in order for our system to avoid getting stuck in the past, and since we are not going to update our probability distributions by counting events, it is better to call them belief distributions. Because the is the way ADE thinks should update the distribution.

15-15- A three-dimensional coordination system showing the way we build and maintain the time series model. For each time window, we have probability distributions of the different features. We call these distributions belief distributions because they represent the ADE’s expectation that an event will occur, based on historical observations.
15.9. Updating the model

If we were to build a histogram distribution or a normal probability distribution, as we show below in 15-16, we could build a vector or table with elements or cells, each representing the total number of the observation or the probability that it will have a specific value based on normal counting.

\[
\text{featureOutcomes} : [d_1, d_2, ..., d_n]
\]

\[
\text{ordinaryProbabilityDistribution} : [p_1, p_2, ..., p_n] = [P(d_1), P(d_2), ..., P(d_n)]
\]

\[
P(p_i) = \frac{p_i}{\sum_{i=1}^{n} p_i}
\]

15-16- An ordinary probability distribution.

As simple as building a vector built on a probability distribution is, this procedure has a problem we talked in Section 5.6; after tens or hundreds of thousands of updates, during which \(n\) gets larger and larger, any new element would be so small relative to the accumulated data that it could barely affect the distribution or the ADE’s belief. Consider, for example, a time space of 60 seconds, or one update per minute, which collects 1,440 data points per day, 43,000 data points per month, and 525,000 data points per year. Even if there is a week of dramatic changes in the behaviour of the time series, there will have already been so much data accumulated under normal conditions that a new behaviour would not have the power to change a belief that has been shaped for years, if it continues to happen.

Instead, if we design our ADE to learn new data with learning factor (\(\lambda\)) of 0.01, it means that, when new information comes in, the ADE will give its old belief a value of 0.99 and the new one a value of 0.01. Formula 15-17 shows how we can update the ADE’s belief distribution elements whenever an update comes in:

\[
\text{new } p_i = \begin{cases} 
\text{old } p_i \times (1 - \lambda) + \lambda, & i = j \\
\text{old } p_i \times (1 - \lambda), & i \neq j
\end{cases}
\]

15-17- How to update belief distribution items in an ADE.

Note that, in 15-17, since the sum of the belief distribution probabilities is 1, applying the forgetting factor of \((1 - \lambda)\) to all items and then adding \(\lambda\) will keep the sum at 1 and maintain a valid distribution. If we then assume that both the belief distribution of the feature and the new given data’s feature value are vectors\(^{34}\), then Equation 15-18 below can represent the way we update the belief distribution matrix for any value of \(\lambda\):

---

\(^{34}\) The new given data is a vector with all zeros except for the new point.
\[
FeatureBeliefDistribution_{t+1} = (1 - \lambda) \times FeatureBeliefDistribution_t + \lambda \times FeatureData
\]

15-18- The ADE can learn quickly or slowly, depending on \( \lambda \).

So, since the elements of model \( M \), described in 15-14, are probability distributions, and we know how to update distributions with learning factor of \( \lambda \), we can update the whole model.

**15.10. Dynamic learning factor**

Up to now, whenever we discussed the idea of learning, we had a constant learning factor \( \lambda \) and a forgetting factor of \((1 - \lambda)\). But, is this the way we human beings learn things? No! We do not update our beliefs or the models we have in our minds with the same learning ratio all the time. Instead, we usually update our beliefs based on the value we give to the source of information.

For example, if your friend starts talking about the idea of sending a tiny and lightweight probe to distant galaxies, you probably would not pay attention and might even laugh at him. But, if you read that Stephen Hawking had the idea of sending a lightweight Nano spaceship to distant galaxies, you might instead think, “Wow, that is a great idea.”

We can apply the same idea of using different learning factors to our ADE algorithm, too. The only information we have available to evaluate the situation and modify the learning factor is the given data and our model, nothing else\(^35\). However, this is enough; we can reduce the impact of the learning factor based on the anomaly index we calculate for the new data coming in. If the new data is not an anomaly, then we can learn it using the regular learning factor of \( \lambda \), but if it is an anomaly, with an index value of \( \text{anomalyIndex} \), then we can use the multiplication \( \lambda \times (1 - \text{anomalyIndex}) \) as the learning factor. We can also use a factor to limit the diminishing effect of the \( \text{anomalyIndex} \). If we call this limiting factor \( \delta \) \((0 \leq \delta < 1)\), then the final dynamic learning factor is \( \lambda \times (1 - \delta \times \text{anomalyIndex}) \), and the combination of \( \lambda \) and \( \delta \) now shapes the way the ADE’s algorithm learns from the data source. By adjusting these two parameters, you can make the ADE learn faster or slower and adjust how much it trusts the data.

**15.11. Comparing the new sample with the stored model**

We will now introduce three different ways to compare the recently extracted features with the stored model. The overall process is straightforward, we are familiar with it, and we have a detailed understanding of steps 1, 2 and 4:

1. Put the new data in the recent data list.
2. Calculate and extract features from the updated recent list, and build the feature vector.
3. Compare the feature vector with the stored model, and calculate the anomaly index.
4. Update model.

\(^{35}\) We will see in Chapter 18 that we can use other data sources to make sure what we are learning is correct.
15.11.1 Model 1: Using the most recent feature vector

The first option is to compare the most recent feature vector with the model directly. In our example, the model matrix $M$ in 15-14 has two different rows for each of the value and slope features. Each row contains the entire time windows’ distribution, which means that each of the $LD$ and $SD$ elements are single data distributions for a specific time window. The result of the value feature extraction can be any of $n$ possible outcomes and the result of the slope feature can be any of $m$ possible outcomes, as shown in 15-19:

$$levelFeatureOutcomes : \{ l_1, l_2, ..., l_n \}$$

$$slopeFeatureOutcomes : \{ s_1, s_2, ..., s_m \}$$

15-19- Extracted result sets for value and slope features.

Based on 15-19, our $LD$ or $SD$ can be expressed using the general form shown in 15-20:

$$levelBeliefDistribution : \{ P(l_1), P(l_2), ..., P(l_n) \}$$

$$slopeBeliefDistribution : \{ P(s_1), P(s_2), ..., P(s_m) \}$$

15-20- Value and slope belief (probability) distributions.

And whenever a new data point comes in, our algorithm extracts the corresponding feature vector, as shown in 15-21:

$$featureVector : [ l_u, s_v ]$$

$$u \in [1..n], v \in [1..m]$$

15-21- New data sample feature vector.

Now the question is, how can we compare 15-21 with its corresponding time window belief distribution in 15-20? Based on what we discussed in Chapter 12, we can calculate the raw anomaly index as $1 - P(l_u)$ and $1 - P(s_v)$. If we consider the thresholds we already have from 12-3, we can calculate the anomaly indexes for both features; let us call them $levelAI$ and $slopeAI$. 

Now we can use any of the methods from 12-5 to assign an anomaly index to the current state of the system, as shown below:

\[
anomalyIndex = \frac{levelAI + slopeAI}{2}
\]

\[
anomalyIndex = levelWeight \times levelAI + slopeWeight \times slopeAI
\]

\[
anomalyIndex = \text{max}(levelAI, slopeAI)
\]

15-22- Some of the many ways you can calculate the overall anomaly index based on the features’ anomaly indexes.

Now you can use the anomaly index to update the model: all you need to do is use 15-17 to update the probabilities of LD and SD.

15.11.2. Method 2: Using recent feature vectors and the Kullback-Leibler comparison

The model we described in Section 15.11.1 was based on feature values as they come in; we can make a more behavioural comparison by keeping an updated list of recent feature vectors, like what we did for recentList, which contains the time series data. We can use a recentFeatureVectorList which always contains recent feature vectors—the size of the list can be anything, depending on how long-term the behavioural patterns are that we want to compare.

Now, whenever we receive new data, we extract the feature vector, update the recentFeatureVectorList and build a distribution from it. This distribution shows the recent behavioural pattern of the system, and we can use the Kullback-Leibler comparison measure discussed in Section 14.3 to calculate the divergence; the result can be considered as the raw anomaly index. We can use the same threshold model to define the safe zone and get a normalized anomaly index. The distribution elements at the middle section of Figure 14-9 gives you a better idea of what we are talking about.

15.11.3. Method 3: Using the entire feature vector’s element

In this approach, we build our model based on the entire feature vector or a combination of all extracted features. However, to get around the curse of dimensionality discussed in 12.2, we need to cluster different data combinations to prevent the number of clusters from growing to the point where it gets out of control. Extracting or selecting 10s or 100s of features works best.

Suppose you have extracted ten features, and any of them could have five different values. In the worst-case scenario, your distribution could have \(5^{10} = 9,765,625\) possibilities in each region—but we cannot work with that, it requires large memory and processing power. But extracting ten features that could have five states each is quite ordinary\(^{36}\), so there must be another way to handle this data.

\(^{36}\) Consider the feature we use to recognize cars.
For example, at some point you may extract a vector like [1,1,4,3,4,2,4,3,5,2] and then another one with values like [1,1,4,3,4,2,4,3,2,2]. These two vectors are identical except for their 9th elements, so we can simply say that they are 90% similar, this is called Hamming distance. Note that, since we do not have any idea of what these ten features are representing, we simply count the similar values and then calculate the ratios. However, when you design your algorithm, you will be aware of what the features are and can use this knowledge to design a better distance function for your system, using the processes described in Section 7.2.

For example, if you already knew that 9th element represents some numeric value, then, instead of saying that these vectors have \( \frac{1}{10} = 0.1 \) difference or \( 1 - 0.1 = 0.9 \) similarity, you could say they have a whole unit difference. Now, instead of saying that the difference between the 9th element values of 5 and 2 is 1.00, we can say that it is \( \frac{5-2}{5-1} = 0.75 \), so the distance between these two vectors is \( \frac{0.75}{10} = 0.075 \), or the similarity is \( 1 - 0.075 = 0.925 \), or 92.5%.

There are different distance functions you can use to find the distance between two vectors, sequences or strings, such as the Manhattan distance described in Section 7.2, the Hamming distance described above, etc. You can also use the combination of them or your own definition to see how similar or different two different extracted feature vectors are.

Now, to build or update the distribution, you need to choose a level of similarity between feature vectors, this level works like clusters radius. You may decide, for example, that 85% similarity is enough. Then, whenever a new feature vector is extracted, you calculate the similarity between that vector and the vectors already stored in the distribution, and if all of them have a similarity of less than 85%, then you add the new vector as a new item to the distribution and start a new counter for it. However, if there are some items with a similarity above 85%, then you would increase or adjust that counter.

Note that, in this method, the distribution is like `HashMap<FeatureVector, Double>`, and what you do is real-time clustering, like building a dictionary of feature vectors that have at least 85% difference. You can use any other clustering method if you are sure it works for the volume of the data or the processing resources you have. Now, since we build just one belief distribution for each time window, and we calculate only one anomaly index for it, there is no need to use the methods outlined in 12-5 to get an aggregate index. Vertical patterns at the right side of the Figure 14-9 gives you a better idea of how this model works.

---

37 Here adjustment means increasing based on learning factor.
16. Basics of time series ADE

16.1. Basic data structures

Before discussing the high-level design of the engine, let us work on some basic data structures and classes, and see if what we discussed before really works. To do this, we are going to introduce two basic data structures that we need, and then write two feature extraction functions and test them all to see how they work.

The first structure we will use is a simple time series of data points, as shown below in 16-1. The next structure is a class that we will use in our recent list implementations:

```java
public class DataPoint {
    private long time;
    private double value;

    DataPoint(long time, double value) {
        this.time = time;
        this.value = value;
    }

    public long getTime() {
        return this.time;
    }

    public double getValue() {
        return this.value;
    }
}

public class RecentDataPoints {
    private final static int minSize = 5;
    private int size = RecentDataPoints.minSize;
    private ArrayList<DataPoint> recentList = new ArrayList<DataPoint>();

    RecentDataPoints(int size) {
        if (size >= RecentDataPoints.minSize) {
            this.size = size;
        }
    }

    RecentDataPoints add(DataPoint dataPoint) {
        if (this.recentList.size() == this.size) {
            this.recentList.remove(0);
        }
        this.recentList.add(dataPoint);
        return this;
    }

    ArrayList<DataPoint> getRecentList() {
        if (this.recentList.size() < this.size) {
            return null;
        } else {
            return this.recentList;
        }
    }
}
```

16-1- Data structures we use to keep time series data and recent data points.
16.2. Level and slope feature extraction classes

The code in 16-2 shows two of the features we described in Chapter 15 that we will use to extract the average value as level and the slope of the series. To do this, we need to pass the recent data list to a feature extraction function, and it will return the feature value. The get() method gives this value in both classes.

```java
public class LevelFeature {
    private final int minWindowSize = 3;
    private final double defaultScaleRatio = 1;
    private int windowSize = this.minWindowSize;
    private double scaleRatio = this.defaultScaleRatio;

    LevelFeature(int windowSize, double scaleRatio) {
        if (windowSize > this.minWindowSize) {
            this.windowSize = windowSize;
        }
        if (scaleRatio > 0) {
            this.scaleRatio = scaleRatio;
        }
    }

    Integer get(ArrayList<DataPoint> recentList) {
        if (recentList.size() < this.windowSize) {
            return null;
        }
        Double result = 0.0;
        for (int i = 0; i < this.windowSize; i++) {
            result += recentList.get(recentList.size() - 1 - i).getValue() * this.scaleRatio;
        }
        result /= this.windowSize;
        return (Integer) result.intValue();
    }
}

public class SlopeFeature {
    private final int minWindowSize = 6;
    private final int halfWindowSize = this.minWindowSize / 2;
    private final int scaleRatio = 20;

    Integer get(ArrayList<DataPoint> recentList) {
        if (recentList.size() < this.minWindowSize) {
            return null;
        }
        double a = 0; double b = 0;
        for (int i = 0; i < this.halfWindowSize; i++) {
            a += recentList.get(i).getValue();
            b += recentList.get(i + this.halfWindowSize).getValue();
        }
        a /= this.halfWindowSize;
        b /= this.halfWindowSize;
        Double result = Math.atan(Math.abs(b - a)) * this.scaleRatio * Math.signum(b - a);
        return (Integer) result.intValue();
    }
}
```

16-2- Value and slope feature extraction classes.
16.3. Feature extraction test

If you use the data generation function `getDataPoint()` shown in 16-3, then the output of the generated time series will be like Figure 16-4.

```java
public class Test {
    final static long startTime = 1483974094;
    final static long period = 24 * 60;
    final static int shownPeriod = 2;

    double getDataPoint(int timeIndex) {
        double result = 15 + 7 * Math.sin(2 * Math.PI * timeIndex / period) - 4
                        - Math.pow(Math.cos(2 * Math.PI * timeIndex / period), 2);
        result = (double) 100 * Math.round(100 * result);
        return result;
    }

    static void showData() {
        Test test = new Test();
        for (int timeIndex = 0; timeIndex < shownPeriod * period;
            timeIndex++) {
            System.out.println(test.getDataPoint(timeIndex));
        }
    }

    public static void main(String[] args) {
        showData();
    }
}
```

16-3- Sample data generator and test method to see how feature extraction works.

16-4- Two periods of the time series generated by the sample code in 16-3.

You can consider the time series in 16-4 to be a daily pattern with one sample per minute, that is, or 1,440 samples per day. It has a peak around 360 (or 06:00), reaches its minimum around 900 (or 15:00), has a small local maximum around 1,080 (or 18:00), and returns to its starting point at 1,440 (or 24:00).

Before looking at the features, let us introduce another useful class which keeps track of the histogram data in 16-5. We use this class to get the probability distribution of feature values.
public class Histogram {
    private HashMap<Integer, Double> data = new HashMap<Integer, Double>();
    private long sum = 0;

    Histogram update(Integer key) {
        if (!this.data.containsKey(key)) {
            this.data.put(key, 0.0);
        }
        this.data.put(key, this.data.get(key) + 1);
        this.sum += 1;
        return this;
    }

    Set<Integer> getKeys() {
        return this.data.keySet();
    }

    double getProbability(Integer key) {
        if (this.data.containsKey(key)) {
            return (double) Math.round(this.data.get(key) * 1000 / this.sum) / 1000;
        } else {
            return 0;
        }
    }
}

16-5- A class to keep track of a histogram data of features distribution.

Now if you add the showLevel() in your Test class and call it in main(), you have the generated feature values and the histogram data, as shown in Figures 16-7 and 16-8.

static void showLevel() {
    Test test = new Test();
    RecentDataPoints rdp = new RecentDataPoints(10);
    LevelFeature lF = new LevelFeature(6, 0.0001);
    Histogram lH = new Histogram();

    for (int timeIndex = 0; timeIndex < Test.shownPeriod * Test.period;
        timeIndex++) {
        DataPoint dp = new DataPoint(Test.startTime + timeIndex * 60,
            test.getDataPoint(timeIndex));
        rdp.add(dp);
        ArrayList<DataPoint> recentList = rdp.getRecentList();
        if (recentList != null) {
            System.out.print(lF.get(recentList));
            lH.update(lF.get(recentList));
        } else {
            System.out.print(0);
        } System.out.println();
    } System.out.println("-------");

    for (Integer key : lH.getKeys()) {
        System.out.println(key + "," + lH.getProbability(key));
    }
}

16-6- A test method to get the extracted features and their histogram for the average value of the data series.
16-7 - Feature extraction result for level feature, based on the data in Figure 16-4.

16-8 - Distribution of different feature values.

Figure 16-7 shows that the extracted values exactly follow the ups and downs of the series, but at a very low resolution. And as we have discussed we do not need a high-resolution comparison because we are going to use a multi-feature comparison method, so a comparison at even this level of precision gives us a good idea of how the series goes up or down during the day.

However, the key point is that we do not need the graph in 16-7 for our comparison process. The graph still represents a large amount of data, and we are going to compare the distributions themselves instead of doing a point-by-point comparison, so all we need is the data shown in Figure 16-8. We just need to extract same kind of histogram from another two periods of the time series, then compare them.

You may wonder how come a histogram, which is information-dense and has lost many details of the source data, could be useful in a comparison model. Well, let us consider how likely it is for the same system to generate some data which is different from the data in 16-7 but that produces the same histogram as in 16-8. There is no doubt that we can independently generate some fake
data to produce the same histogram using mathematics, but is the system you are monitoring capable of doing the same thing? What if we add another feature, like slope, to the data. Then how likely will it be that the system will generate some series of data with the same distribution of values and slope but with different behaviour?

The logic behind this method is that, if you increase the number of extracted independent features and then use this dense information for comparison, there is only a very small chance that two different time series will produce the same dense information. The method in 16-9 below extracts the slope feature and produces its corresponding histogram data.

```java
static void showSlope() {
    Test test = new Test();
    RecentDataPoints rdp = new RecentDataPoints(10);
    SlopeFeature sF = new SlopeFeature();
    Histogram sH = new Histogram();
    for (int timeIndex = 0; timeIndex < Test.shownPeriod * Test.period;
        timeIndex++) {
        DataPoint dp = new DataPoint(Test.startTime + timeIndex * 60,
            test.getDataPoint(timeIndex));
        rdp.add(dp);
        ArrayList<DataPoint> recentList = rdp.getRecentList();
        if (recentList != null) {
            System.out.print(sF.get(recentList));
            sH.update(sF.get(recentList));
        } else {
            System.out.print(" ");
        }
        System.out.println();
    }
    System.out.println("-----");
    for (Integer key : sH.getKeys()) {
        System.out.println(key + "," + sH.getProbability(key));
    }
}
16-9- A test method to get the extracted features and their histogram for slope.
```
As you see in Figures 16-10 and 16-11, the slope feature gives us information about how fast or slow the time series values go up and down over a particular time span. Compare this histogram with 16-8 and think about how likely it is that the system will generate some data which will give us exactly these two histograms.

In a nutshell, the method we have described in this paper is the same as using these two histograms; we just need to use narrow periods for our histograms and update them using the dynamic learning factor we discussed earlier. To finish this section, let us change the `getDataType()` method outlined in Listing 16-3 above and redraw all the graphs to see how these two histograms change.
double getDataPoint(int timeIndex) {
    double result = 15 + 7 * Math.pow(Math.sin(2 * Math.PI * timeIndex / Test.period), 3) - 4
    * Math.pow(Math.cos(2 * Math.PI * timeIndex / Test.period), 3);
    result = (double) 100 * Math.round(100 * result);
    return result;
}

16-12- Another data generation function.

As you see in Figure 16-13, the general behaviour of the new data is not very different from the data shown in Figure 16-4, but the generated features and histograms are totally different and easily distinguishable.
So, with the introduced classes and methods we can build the probability distribution of different features of time series in different time windows and use the for our comparison or anomaly detection process.
16.4. Belief distributions and model classes

Using a histogram for probability distribution, which is essentially repeating an experiment and counting the results, is just one of the ways we can build a model—and there are many reasons to avoid using this kind of method on its own. Sometimes, for example, you cannot run fair experiments, or the environment is biased, or the experiments are done by different sources, and we do not trust them equally. This is where we can use the dynamic learning factor we introduced in Section 15.10 as below where $\lambda$ is the learning factor and $\delta$ is the anomaly learning impact limiting factor:

$$\text{dynamicLearningFactor} = \lambda \times (1 - \delta \times \text{anomalyIndex})$$

16-15- Definition of dynamic learning factor.

We will use this dynamic learning factor in Formula 15-17 to update our model. This process gives us a simple class to support our belief probability distribution. You can see the class source code in 16-17 and how to use it in 16-16. It is like our histogram class in 16-5, but with these differences:

1. There is a constructor that helps set the default values for $\lambda$ and $\delta$.
2. There are two update methods: one accepts only the feature value and is unaffected by the anomaly index, and the other one also accepts the anomaly index, and takes the impact of the anomaly index into account.
3. The method uses exactly the same dynamic learning factor model as 16-15 and the same process we described in Section 15.10 to update the belief model.

```java
static void testLevelBelief() {
    Test test = new Test();
    RecentDataPoints rdp = new RecentDataPoints(10);
    LevelFeature lF = new LevelFeature(6, 0.0001);
    BeliefDistribution lB = new BeliefDistribution();

    for (int timeIndex = 0; timeIndex < Test.shownPeriod * Test.period; timeIndex++) {
        DataPoint dp = new DataPointTest.startTime + timeIndex * 60, test.getDataPoint(timeIndex));
        rdp.add(dp);
        ArrayList<DataPoint> recentList = rdp.getRecentList();
        if (recentList != null) {
            System.out.print(lF.get(recentList));
            lB.update(lF.get(recentList));
        } else {
            System.out.print(0);
        }
        System.out.println();
    }
    System.out.println("-------");
    for (Integer key : lB.getKeys()) {
        System.out.println(key + "," + lB.getProbability(key));
    }
}
```

16-16- How to use belief probability distribution class.
public class BeliefDistribution {
    private double learningFactor = 0.001;
    private double dimishLimit = 0.2;
    private HashMap<Integer, Double> data = new HashMap<Integer, Double>();
    private boolean firstData = true;

    BeliefDistribution () {
    }

    BeliefDistribution (double learningFactor, double dimishLimit) {
        if (learningFactor > 0 && learningFactor < 1) {
            this.learningFactor = learningFactor;
        }

        if (dimishLimit >= 0 && dimishLimit <= 1) {
            this.dimishLimit = dimishLimit;
        }
    }

    BeliefDistribution update(Integer key) {
        return this.update(key, 0);
    }

    BeliefDistribution update(Integer key, double anomalyIndex) {
        double dynamicLearningFactor = this.learningFactor * (1 - this.dimishLimit * anomalyIndex);
        this.decreaseAll(dynamicLearningFactor);
        if (!this.data.containsKey(key)) {
            if (this.firstData) {
                this.data.put(key, 1.0);
                this.firstData = false;
            } else {
                this.data.put(key, dynamicLearningFactor);
            }
        } else {
            this.data.put(key, this.data.get(key) + dynamicLearningFactor);
        }
        return this;
    }

    Set<Integer> getKeys() {
        return this.data.keySet();
    }

    double getProbability(Integer key) {
        if (this.data.containsKey(key)) {
            return this.data.get(key);
        } else {
            return 0;
        }
    }
}
private void decreaseAll(double dynamicLearningFactor) {
    Iterator<Map.Entry<Integer, Double>> iterator =
        this.data.entrySet().iterator();
    while (iterator.hasNext()) {
        Map.Entry<Integer, Double> item = iterator.next();
        item.setValue(item.getValue() * (1 -
                     dynamicLearningFactor));
    }
}

16-17- A sample class to support the idea of belief probability distribution.

If you run one, two or three periods of data using a histogram class like the one described in Section
16.3, the results will always be the same, but using a belief distribution will not produce the same
results every time, even when the data does not change its behaviour. The reason for this is that,
with a belief probability distribution, the system trusts old data more than new data. But in the long
term, this strategy will reveal the dominant behaviour of the system. For example, Figure 16-18
shows how the belief distribution changes over several periods.

16-18- The belief probability distribution changes over time.

16-19 demonstrates a class that supports multiple belief distributions over the longest period of a
time series. Here is a brief description of its methods and specifications:

- The Model class needs to know the duration of the time window to convert a given point’s time
  stamp to a time window.
- The default time window is 3 hours.
- You can change the learning factor and the diminish limit ($\delta$) when you instantiate an object of this
class.
- A HashMap supports a separate belief distribution class for every time window.
- Like BeliefDistribution in 16-16 and 16-17, there are two update methods that can be used to update
  the belief of the model.
- There are also two different methods to return the probability of any feature having a particular
  value: one uses the time window ID and the other one uses a Unix time stamp.
- For internal usage, two private functions to convert the Unix time stamp to the time window ID are
  also included.
public class Model {
    private final int maxRegionWidthInHour = 12;
    private int regionWidthInHour = 3;
    private double learningFactor = 0.001;
    private double dimishLimit = 0.5;

    private HashMap<Integer, BeliefDistribution> model = new HashMap<Integer, BeliefDistribution>();

    Model() {
    }

    Model(int regionWidthInHour) {
        if (regionWidthInHour > 0 && regionWidthInHour <= this.maxRegionWidthInHour) {
            this.regionWidthInHour = regionWidthInHour;
        }
    }

    Model(int regionWidthInHour, double learningFactor, double dimishLimit) {
        if (learningFactor > 0 && learningFactor < 1) {
            this.learningFactor = learningFactor;
        }
        if (dimishLimit >= 0 && dimishLimit <= 1) {
            this.dimishLimit = dimishLimit;
        }
        if (regionWidthInHour > 0 && regionWidthInHour <= this.maxRegionWidthInHour) {
            this.regionWidthInHour = regionWidthInHour;
        }
    }

    Model update(long time, int featureValue) {
        return this.update(time, featureValue, 0);
    }

    Model update(long time, int featureValue, double anomalyIndex) {
        Integer region = this.getRegionId(time);
        if (!this.model.containsKey(region)) {
            this.model.put(region, new BeliefDistribution(this.learningFactor, this.dimishLimit));
        }
        this.model.get(region).update(featureValue, anomalyIndex);
        return this;
    }

    Set<Integer> getKeys(Integer region) {
        if (this.model.containsKey(region)) {
            return this.model.get(region).getKeys();
        } else {
            return null;
        }
    }

    boolean contains(Integer region) {
        return this.model.containsKey(region);
    }

    double getProbability(Integer region, Integer key) {
        if (this.model.containsKey(region)) {
            return this.model.get(region).getProbability(key);
        } else {
            return 0;
        }
    }

    double getProbability(long time, Integer key) {
        Integer region = this.getRegionId(time);
        return this.getProbability(region, key);
    }
}
private int hourOfWeek(long time) {
    Calendar cal = Calendar.getInstance();
    cal.setTimeInMillis(time * 1000);
    return cal.get(Calendar.HOUR_OF_DAY) + (cal.get(Calendar.DAY_OF_WEEK) - 1) * 24;
}

private int getRegionId(long time) {
    return hourOfWeek(time) / this.regionWidthInHour;
}

16-19- Sample model class code to handle multiple belief distributions for a feature in different time windows.

The model class introduced in 16-19 works exactly like what we discussed before; whenever you call its `update()` method, it finds the time window index of the data and updates its belief via the assigned `BeliefDistribution` class for that window, which itself updates its belief based on the calculated learning factor, anomaly index and $\delta$. You can use it exactly in the same way that you use the Histogram or `BeliefDistribution` classes as described above. The charts below in 16-20 show some of the belief distributions that happen when you feed 14 days of the defined data in Section 16.3 into this model class, with a time window of 6 hours.

16-20- Four samples of belief probability distribution, each for a single 6-hour time window during the week. And as you see since the time series behaviour is different in these time windows the results are different.
It is important to remember that the size of the time window depends on your business. For a variable that depends on people’s daily usage, choosing a window of 6 or 4 or 3 or even 2 hours works well. However, if you are working on monitoring a variable that changes over a period of an hour, then the time window you choose should be well under an hour, perhaps 5 minutes, or a minute, etc.

16.5. The anomaly detection class

The code in 16-21 is a suggestion for how to implement an anomaly detection class. There’s nothing new here, just things we have discussed in previous chapters, and the code is also like our earlier tests. The only differences are that we have added two new methods to our value and slope feature extraction in order to calculate the anomaly index because we need to do that in order to update the model. Since the new value still is not in the recent data points list, we have added these two new methods shown in 16-22 to feature classes.

```java
public class AnomalyDetection {
    private final double levelScaleRatio = 0.0001;
    private final int levelWindowSize = 6;
    private double highThreshold = 1.0;
    private double lowThreshold = 0.99;
    private Model levelModel = null;
    private Model slopeModel = null;
    private RecentDataPoints recentDataPoints = new RecentDataPoints(this.levelWindowSize);
    private LevelFeature levelFeature = new LevelFeature(this.levelWindowSize, this.levelScaleRatio);
    private SlopeFeature slopeFeature = new SlopeFeature();

    AnomalyDetection() {
        this.levelModel = new Model();
        this.slopeModel = new Model();
    }

    AnomalyDetection(int regionWidthInHour) {
        this.levelModel = new Model(regionWidthInHour);
        this.slopeModel = new Model(regionWidthInHour);
    }

    AnomalyDetection(int regionWidthInHour, double learningFactor, double dimishLimit) {
        this.levelModel = new Model(regionWidthInHour, learningFactor, dimishLimit);
        this.slopeModel = new Model(regionWidthInHour, learningFactor, dimishLimit);
    }

    AnomalyDetection setHighThreshold(double highThreshold) {
        if (highThreshold <= 1 && highThreshold > 0 && highThreshold > this.lowThreshold) {
            this.highThreshold = highThreshold;
        }
        return this;
    }

    AnomalyDetection setLowThreshold(double lowThreshold) {
        if (lowThreshold <= 1 && lowThreshold > 0 && lowThreshold < this.highThreshold) {
            this.lowThreshold = lowThreshold;
        }
        return this;
    }
}
```
double getAnomalyIndex(long time, double value) {
    double anomalyIndex = 1;
    ArrayList<DataPoint> recentList = this.recentDataPoints.getRecentList();
    if (recentList != null) {
        Integer levelValue = this.levelFeature.get(recentList, value);
        Integer slopeValue = this.slopeFeature.get(recentList, value);
        if (levelValue != null && slopeValue != null) {
            double levelAnomalyIndex = 1 -
                this.levelModel.getProbability(time, levelValue);
            double slopeAnomalyIndex = 1 -
                this.slopeModel.getProbability(time, slopeValue);
            anomalyIndex =
                Math.max(levelAnomalyIndex, slopeAnomalyIndex);
        }
    }
    return anomalyIndex;
}

AnomalyDetection update(long time, double value) {
    return this.update(time, value, 0.0);
}

AnomalyDetection update(long time, double value, double anomalyIndex) {
    this.recentDataPoints.add(new DataPoint(time, value));
    ArrayList<DataPoint> recentList = this.recentDataPoints.getRecentList();
    if (recentList != null) {
        Integer levelValue = this.levelFeature.get(recentList);
        Integer slopeValue = this.slopeFeature.get(recentList);
        this.levelModel.update(time, levelValue, anomalyIndex);
        this.slopeModel.update(time, slopeValue, anomalyIndex);
    }
    return this;
}

16-21- A suggestion for an anomaly detection class.

Integer get(ArrayList<DataPoint> recentList, double value) {
    if (recentList.size() < this.minWindowSize) {
        return null;
    }
    double a = 0;
    double b = value;
    for (int i = 1; i < this.halfWindowSize + 1; i++) {
        a += recentList.get(i).getValue();
    }
    for (int i = this.halfWindowSize + 1; i < recentList.size(); i++) {
        b += recentList.get(i).getValue();
    }
    a /= this.halfWindowSize;
    b /= this.halfWindowSize;
    Double result = Math.atan(Math.abs(b - a)) * this.scaleRatio * Math.signum(b - a);
    return (Integer) result.intValue();
}
Integer get(ArrayList<DataPoint> recentList) {
    if (recentList.size() < this.windowSize) {
        return null;
    }
    Double result = 0.0;
    for (int i = 0; i < this.windowSize; i++) {
        result += recentList.get(recentList.size() - i - 1).getValue();
    }
    result *= this.scaleRatio;
    result /= this.windowSize;
    return (Integer) result.intValue();
}

16-22- Two new methods we need to use in `SlopeFeature` and `LevelFeature` classes for anomaly detection.

Before we test the `AnomalyDetection` class with its default parameters, let’s go over how it works. First, it learns the system’s behaviour over time with a dynamic learning factor\(^{38}\) so if you test the system, it will initially have some anomalies because it will be encountering low-probability behaviours for the first time, but, after a while, it will learn that these behaviours are within the normal range. Second, you either need to extract many features to cover any series behaviour, or choose and fine tune the required parameters we introduced in the sample codes for your specific time series. It is very important to choose time windows appropriate for your system. Narrower time windows lead to better sensitivity if your time series pattern is smooth, or if you want to catch time shifts in the series.

Finally, let us mention again that it is up to you and the needs of your business and chosen method of analysis to decide how to calculate the overall anomaly index. In our sample class—look Listing 16-21—we have used the maximum value of anomalies of the features.

Figure 16-23 shows a test result after about twenty periods of the time series have passed. The horizontal line at 250,000 shows the anomalies. As you see, it is defined so that it catches those abnormal behaviours. You may also want to define a threshold to make sure that the situation is a real threat, like the dashed line in the picture. Catching this kind of anomaly is much easier with a feature that gives you the count of local maxima or minima in a window. If you are sure that your series is smooth, use that feature.

\(^{38}\) Here with a maximum of 0.001, based on what we defined in the Model class.
16.6. Testing with fluctuating data

The features described above will still work for a rapidly fluctuating time series like the one shown in Figure 16-24, with some tuning. The data shown below always has random fluctuations, so if we use the defined slope feature on it, it will not have any effect on its distribution pattern—it will always have many ups and downs. So, even if we use slope feature, it should not report a high anomaly unless the time series unexpectedly becomes smooth!

For this kind of time series, we need to define what an anomaly is before we can look for a solution. We do not have any exact idea of how high the next data point will be, but we do know that, over a period of 30 samples (for example), we should have a certain number of high and low values. Perhaps the best way to check the number of high and low values is to use a window of 30 for level feature extraction\(^{39}\). If we do this, then our ADE will see the time series in a way more like what is shown in 16-25, which is a more predictable pattern.

\(^{39}\) Change this line in AnomalyDetection class: `private final int levelWindowSize = 30;`
If we use a window size of 30 for the value feature extraction, the ADE looks at the data in a more like this smoothed time series.

I also used a single hour window size for modeling this time series. After about 30 days of data, Figure 16-26 shows the result of its anomaly detection. After time = 1,161, I manually put in some unusual data, and around time = 6,526, I set some data to zero. The system understands these unusual behaviours and reports them as anomalies. But there are two more anomalies, around time = 3,626 and time = 6,671—what are they? We will discuss them in next section.

Anomaly detection for a time series data like the one in 16-24, after about 30 days.

By changing this line in Model class: `private int regionWidthInHour = 1;`

sleptons.com
16.7. Deterministic, but hardly predictable!

To give a convincing answer to why a given point is an anomaly, we need to:

- Completely understand how the ADE’s algorithm works.
  - We need to understand how the ADE updates its belief probability distributions, extracts features, and calculates the overall anomaly index.
- And we must either
  - have access to the state of the entirety of the ADE’s belief probability distributions at the time the anomaly has been calculated for that point, or
  - have access to the entire data set the ADE has processed throughout its life.

Even with the above information, we would still need to do the calculations step by step to reach the point we want to know about to see why it was reported as an anomaly.

If you think about it, it is almost impossible for us to understand why something is reported as an anomaly, especially when ADE uses many features or when it is dealing with time series with unfamiliar patterns. However, for a simple model, if you were using a threshold or some margin above or below the average, the anomaly results would be easy to describe. But just because anomalies are easy to describe using these methods does not mean that using them is a good idea; they only work under special circumstances, and will not work with fluctuating series such as the one in 16-26 or many other cases.

So, if you want to understand the reason why the system has marked the point around time = 3,626 in Figure 16-26 as an anomaly, you need to go step-by-step through the entire 30 days of data, moving through the ADE’s belief probability distributions states one step at a time. Mathematicians call systems like this deterministic because there is no randomness in the process of determining their future states. Our algorithm is deterministic, but does this mean that you can jump to any place in the time index to see whether the point there is an anomaly? The answer is no: because the belief is shaped over time, step by step, it is deterministic but hardly predictable. This is something that can be said about any learning system.

16.8. Next steps

When you know your time series and are aware of how exactly it should behave, then you have enough information to define what anomaly is, and you can use that information to decide which features to extract. You can use the sample code we introduced in this chapter to extract your features, build a belief probability distribution model, and calculate your anomaly index. However, there will be times when you will not have this level of knowledge about your series, and yet you will still need to monitor a time series and see whether something goes wrong. You can do it by building a pattern dictionary from your time series on the fly, which we will talk about in the next chapter.
17. Using patterns in ADE

17.1. Extracting distribution patterns

We talked about temporal patterns and why they are important in Chapter 8. One of the easiest ways to extract patterns from a time series is to build a model of small time intervals of the time series and then see whether they repeat. Figure 17-1 shows a time series broken into small intervals. Let us consider the intervals to be one minute long.

Let us choose a small number of intervals, in this case, 6 samples. Over a short time period like this, there must be no difference between the average value or frequency of the distribution of the values within [3,3,5,5,3] and [3,3,3,5,5] or even [3,5,5,3,3], so the data distributions we build for them will all be the same, that is: \( \{3=4, 5=2\} \)\(^{41}\) and this distribution can also be used to identify similar patterns. And of course, you can choose shorter or longer intervals.

Now if you put 5 of these aggregated patterns together, as shown in Figure 17-1, you can build a pattern for half an hour. However, it is not a good idea to aggregate data for this length of time, because you might lose too much information. Instead, their exact sequence can be considered as a model for a half hour pattern.

---

\(^{41}\) This can also be expressed as \( P(4) = \frac{4}{6}, P(5) = \frac{2}{6} \).
So, if you build aggregated patterns such as a1, a2, a3, ... then your sequenced patterns could be s1, s2, s3, ... and s1 again. S1, for example, could be a sequence of [a1, a6, a14, a15, a14]. We can observe both aggregated and sequenced patterns on the fly for any time series, learn when and how many times they usually appear, and use them to find anomalies in the series.

```java
public class Pattern {
    private ArrayList<Integer> values = new ArrayList<Integer>();
    private HashMap<Integer, Integer> valuesCounter = new HashMap<Integer, Integer>();
    private PatternType patternType = PatternType.AGGREGATED;
    private int id = 0;
    private String stringId = "";
    enum PatternType { SEQUENCE, AGGREGATED }

    Pattern(ArrayList<Integer> values, PatternType patternType) {
        this.patternType = patternType;
        if (this.patternType == PatternType.SEQUENCE) {
            for (Integer i : values) {
                this.values.add(i);
            }
        } else {
            for (Integer i : values) {
                if (this.valuesCounter.containsKey(i)) {
                    this.valuesCounter.put(i, this.valuesCounter.get(i) + 1);
                } else {
                    this.valuesCounter.put(i, 1);
                }
            }
        }
        this.stringId = this.getRawId();
        this.id = this.stringId.hashCode();
    }

    private String getRawId() {
        String id = "";
        if (this.patternType == PatternType.SEQUENCE) {
            for (Integer value : this.values) {
                id += "." + value;
            }
        } else {
            ArrayList<Integer> sortedValues = new ArrayList<Integer>();
            this.valuesCounter.keySet().
            Collections.sort(sortedValues);
            for (Integer featureValue : sortedValues) {
                id += "." + featureValue + "=" + this.valuesCounter.get(featureValue);
            }
        }
        return id;
    }
}
double getSimilarity(Pattern pattern) {
    double similarity = 0;
    if (this.patternType == PatternType.SEQUENCE) {
        if (this.values.size() == pattern.size()) {
            for (int i = 0; i < this.values.size(); i++) {
                if (this.values.get(i) == pattern.getValue(i)) {
                    similarity += 1;
                }
            }
        }
    } else {
        for (Integer value : this.valuesCounter.keySet()) {
            if (this.getCounter(value) == pattern.getCounter(value)) {
                similarity += 1;
            }
        }
    }
    similarity /= this.values.size();
    return similarity;
}

int getId() {
    return id;
}

String getStringId() {
    return this.stringId;
}

double size() {
    return this.values.size();
}

Integer getValue(int index) {
    if (index > this.values.size()) {
        return 0;
    } else {
        return this.values.get(index);
    }
}

Integer getCounter(Integer value) {
    if (this.valuesCounter.containsKey(value)) {
        return this.valuesCounter.get(value);
    } else {
        return 0;
    }
}

17-2- A class to aggregate data and build patterns from time series.

The class introduced in 17-2 works for aggregated or sequenced patterns. If you want to build patterns for your time series data, you can use the LevelFeature class we introduced in Chapter 16 to extract the average value of the time series as integers. Now, whenever a new data point comes in, you can add the extracted value feature to a list of integers using ArrayList<Integer>, and when the size reaches 6, the class will build an aggregated pattern from that.

If you write a simple code and run this process over one or two or 100 periods of the time series we introduced in Figure 16-4, the test result shows just 23 different patterns, like {17=1, 18=5} or {12=4, 13=2} or {14=6} or {18=4, 19=2}, etc.
You can instantiate the `Pattern` class by passing it an array list of coarse level values. The constructor accepts another parameter to indicate whether you want to produce an aggregated pattern or sequenced one. It assigns a string ID and its corresponding integer hash ID to the aggregated or sequenced data as the pattern’s identification. So, if you give the class the data `[5,5,3,3,3,3]` and instantiate either an aggregated or sequenced pattern, the results are going to be different, depending on the pattern model used: the code will assign different string and integer hash IDs to the pattern and use different comparison methods, as shown below:

```java
Integer[] featureValues = { 5, 5, 3, 3, 3, 3 };
ArrayList<Integer> featureValuesList = new ArrayList<Integer>();
Collections.addAll(featureValuesList, featureValues);
Pattern pattern1 = new Pattern(featureValuesList, PatternType.SEQUENCE);
System.out.println("P1: " + pattern1.getStringId() + " , " + pattern1.getId());
Pattern pattern2 = new Pattern(featureValuesList, PatternType.AGGREGATED);
System.out.println("P2: " + pattern2.getStringId() + " , " + pattern2.getId());
```

```
P1: .5.5.3.3.3.3 , 1381576546
P2: .3=4.5=2 , -1621231176
```

17-3- Sample code for the Pattern class

This very simple method works well enough for many projects, though you can use any method to compare an incoming pattern with another pattern. In fact, you can build a dictionary of patterns, and give the stream of data to the dictionary to extract, categorize or cluster them based on a defined similarity or distance between the patterns. Sample code for this purpose is presented in 17-4 below. It is easy to use; just pass the required degree of similarity to its constructor, and then start adding patterns. When you add a new pattern by calling `addPattern()`, the built-in similarity function determines which pattern in the dictionary is the most similar to the incoming pattern. If the new pattern and most similar pattern in the dictionary are close enough to satisfy the configured minimum similarity, the code does not add the pattern, and just returns the closest stored pattern ID. However, if there is no pattern with the required degree of similarity in the dictionary, it adds the new pattern to the dictionary. Every time you call the `addPattern()` method and it matches a new pattern with one of the existing patterns, it increases a counter. We use this counter to see how many times each pattern repeats.
17-4- Sample Dictionary class code

Giving the time series in Figure 16-4 to the above dictionary, configured for a 6-sample window with 100% similarity (or 1.0) returns only 23 aggregated patterns and 31 sequenced patterns. These patterns are listed in Table 17-5.
<table>
<thead>
<tr>
<th>Aggregated Pattern</th>
<th>Count</th>
<th>Sequenced Pattern</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>.7=6</td>
<td>92</td>
<td>.7.7.7.7.7</td>
<td>92</td>
</tr>
<tr>
<td>.8=6</td>
<td>84</td>
<td>.8.8.8.8.8</td>
<td>84</td>
</tr>
<tr>
<td>.21=6</td>
<td>52</td>
<td>.21.21.21.21.21</td>
<td>52</td>
</tr>
<tr>
<td>.9=6</td>
<td>32</td>
<td>.9.9.9.9.9</td>
<td>32</td>
</tr>
<tr>
<td>.10=6</td>
<td>23</td>
<td>.10.10.10.10.10</td>
<td>23</td>
</tr>
<tr>
<td>.20=6</td>
<td>20</td>
<td>.20.20.20.20.20</td>
<td>20</td>
</tr>
<tr>
<td>.11=6</td>
<td>19</td>
<td>.11.11.11.11.11</td>
<td>19</td>
</tr>
<tr>
<td>.17=6</td>
<td>16</td>
<td>.17.17.17.17.17</td>
<td>16</td>
</tr>
<tr>
<td>.16=6</td>
<td>16</td>
<td>.16.16.16.16.16</td>
<td>16</td>
</tr>
<tr>
<td>.15=6</td>
<td>16</td>
<td>.15.15.15.15.15</td>
<td>16</td>
</tr>
<tr>
<td>.18=6</td>
<td>12</td>
<td>.18.18.18.18.18</td>
<td>12</td>
</tr>
<tr>
<td>.7=3.8=3</td>
<td>8</td>
<td>.8.8.8.7.7.7</td>
<td>4</td>
</tr>
<tr>
<td>.17=1.18=5</td>
<td>4</td>
<td>.7.7.7.8.8</td>
<td>4</td>
</tr>
<tr>
<td>.12=4.13=2</td>
<td>4</td>
<td>.22.22.22.21.21</td>
<td>2</td>
</tr>
<tr>
<td>.20=5.21=1</td>
<td>4</td>
<td>.21.21.22.22.22</td>
<td>2</td>
</tr>
<tr>
<td>.21=3.22=3</td>
<td>4</td>
<td>.21.20.20.20.20</td>
<td>2</td>
</tr>
<tr>
<td>.13=5.14=1</td>
<td>4</td>
<td>.20.20.20.20.20</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.19.19.18.18.18</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.18.18.18.18.19</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.18.18.18.18.17</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.17.18.18.18.18</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.13.13.13.13.14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.13.13.12.12.12</td>
<td>2</td>
</tr>
</tbody>
</table>

17-5- Different aggregated and sequenced patterns extracted from Figure 16-4 for a single period of the series.
Now consider the time series mentioned above as within your system’s normal range of behaviour. Even if there are some common changes in the behaviour of your system, after a while, the total number of collected patterns may grow to 50 or 100 or 200. If you build a time window vector like the ones used in our previous models for the longest period of the system and assign a dictionary for each of these windows, then you have fewer options for every time window and it is easier to calculate each pattern’s probability and catch newly-observed patterns. It is like having a hash map of the dictionary which maps each time window to its own dictionary by using `HashMap<Integer, Dictionary>`. In a nutshell, this is how to build patterns using only the values of the time series at certain times and observing their behaviour over time.

If your time series is not smooth but instead looks more like Figure 16-24, then the differences between data such as [26, 36, 10, 62, 19, 32] and [26, 19, 32, 36, 10, 62] are not very significant. In cases like this, you can use another similarity function or use a wider window for aggregation. if you pass 100 periods of the data from 16-24 to the Dictionary class, the results will be something like this:

<table>
<thead>
<tr>
<th>Window Size</th>
<th>Similarity</th>
<th>Sequenced Patterns</th>
<th>Aggregated Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2,917</td>
<td>1,579</td>
</tr>
<tr>
<td>6</td>
<td>0.95</td>
<td>2,884</td>
<td>1,561</td>
</tr>
<tr>
<td>6</td>
<td>0.90</td>
<td>2,755</td>
<td>1,508</td>
</tr>
<tr>
<td>6</td>
<td>0.80</td>
<td>1,360</td>
<td>1,486</td>
</tr>
<tr>
<td>6</td>
<td>0.65</td>
<td>552</td>
<td>392</td>
</tr>
</tbody>
</table>

17-6- Number of generated patterns for different similarities in the data from 16-24.

Table 17-6 shows that the number of sequenced patterns remains high until the similarity index drops to 0.80 and then to 0.65. But is it OK to use a similarity as low as 0.80 or 0.65? The answer is like most answers when it comes to anomaly detection in time series: yes and no. If you insist on using the value feature with a series like the one in 16-24, yes, you must use a low similarity, but if you use another feature, you may not need to do so. Table 17-6 also clearly shows why we cannot completely understand anomaly detection results for a time series like 16-24: unlike the small number of patterns generated from a smooth series shown in Table 17-3, series with many fluctuations such as 16-24 produce many patterns.

---

Extracting more patterns from a system’s behaviour, potentially gives us more power to catch anomalies. But as a human, since we cannot get all the existing patterns in a system, some of the anomalies may look odd to us.
I always give an example of the way we recognize potatoes; potatoes do not all have the same shape, so a shape-similarity comparison method does not work as well for them as it would work for something like bananas, which are all similar in shape. If you want to use a shape comparison to recognize potatoes, you would have to use a low similarity factor for shape and also examine some other features, like their colour or skin texture, so that they would not be confused with other things with overlapping ranges of shapes. The same goes for a series like the one in Figure 16-24.

### 17.2. Modeling with predefined patterns

If you know the usual behaviour of your time series, you may be able to use a few predefined patterns to model the series. For example, returning to Figure 16-4, see whether you can spot the predefined patterns shown in Table 17-7 within the 10-minute time windows:

<table>
<thead>
<tr>
<th>Pattern ID</th>
<th>Description</th>
<th>Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Horizontal movement</td>
<td>→</td>
</tr>
<tr>
<td>B</td>
<td>Uphill</td>
<td>↑</td>
</tr>
<tr>
<td>C</td>
<td>Downhill</td>
<td>↓</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

17-7- Some predefined patterns we can use to model the time series in Figure 16-4.

It is easy to write a function that accepts 10 samples and then determines which one of these predefined patterns matches best. Using this technique, the time series in 16-4 can be described as a sequence of short but known patterns, such as:


17-8- A model based on predefined patterns in Table 17-5 for time series 16-4.

If we need more precision or a higher resolution, we can either increase the sampling rate or define more patterns. For example, instead of just one Uphill pattern, you may define uphill segments with different angles, such as Uphill15, Uphill30, Uphill45, Uphill60, and so on. The important thing is that now you have more options for anomaly detection:

1. If you cannot match an incoming data sample to any of the predefined patterns with a minimum degree of similarity, you can raise an anomaly alarm.
2. You can build time windows and their corresponding distribution and find the anomalies as we described before.
Also, instead of analyzing distributions, you can build a Markov Chain or use the HMM\(^{42}\) for each time window to develop an expectation about the next incoming pattern. These options are helpful for when you do not want to store the whole pattern in memory.

### 17.3. Long term behaviour patterns and hierarchies

The method we discussed in the previous section, can be used to build a long-term behavioural pattern dictionary, which can be is important in analysis of the behaviour of a series. For example, based on the patterns defined in Table 17-5, we can describe 30 minutes of behaviour as B,A,C (going up to a maximum and then dropping) or C,A,B (going down to a minimum and then rising) or A,A,A (staying level), etc. We can use this idea to build a hierarchy of patterns: you can build 30-minute patterns based on 10-minute patterns, then hourly patterns based on 30-minute patterns, etc. Using a dictionary is crucial because it allows us to recognize the most frequently observed patterns in the time series at the different levels of the hierarchy. Table 17-7 shows how the use of dictionaries can help us understand long-term patterns.

---

\(^{42}\) Hidden Markov Model
18. Software structure

18.1. General design

We explained earlier how a general anomaly detection engine should work and saw that the following steps should be followed in order:

1. Preparing data, coarse classification, and feature extraction,
2. Comparing the extracted features with the model and calculating the anomaly index,
3. Updating the model, if it is required.

Now we need a software design that supports the above process and works real-time.

18.2. Feature anomaly detection

Figure 18-1 shows a general idea of a Feature Anomaly Detection (FAD) module. Any of the boxes in the diagram can be assumed to be a method of a class. So, using classes such as Feature, Model, AnomalyCalculation and Result, etc. Having these methods, lets us to write a simple code that continuously extracts features, calculates the anomaly index, and updates the model, as shown in 18-2.

If you review the classes used in 18-2, almost all of them have been covered in this paper, but some parts are new. observeSystem is a variable that controls the general process of feature anomaly detection; dataSource is an object that gives us new data, and there is also sleepOneSecond(), a static method that waits one second; if you do not use it, you probably will experience a high CPU usage. Note that the Result should be thread-safe to let another thread read the result whenever the UI asks for the update.

18-1- A detail model for feature anomaly detection and its encapsulation model.
109

```java
Feature feature = new Feature(...);
Model m = new Model(...);
AnomalyCalculationUnit acu = new AnomalyCalculationUnit(...);
Result result = new Result(...);

while (observeSystem) {
    if (dataSource.isAvailable()) {
        DataPoint dataPoint = dataSource.getData();
        Integer featureValue = feature.getValue(dataPoint);
        Double anomalyIndex = acu.process(model, dataPoint, featureValue);
        result.update(dataPoint, anomalyIndex);
        model.update(dataPoint, featureValue);
    }
    sleepOneSecond();
}
```

18-2- Java pseudocode to for the process shown in Figure 18-1.

The whole process can be wrapped up and used as a single component called FAD and used as follows:

```java
FAD fda = new FDA(...);
Result result = new Result(...);

while (observeSystem) {
    if (dataSource.isAvailable()) {
        DataPoint dataPoint = dataSource.getData();
        Double anomalyIndex = fda.process(dataPoint);
        result.update(dataPoint, anomalyIndex);
        model.update(dataPoint, featureValue);
    }
    sleepOneSecond();
}
```

18-3- Java pseudocode to for the process shown in 18-1 as a single component.

18.3. Multi feature anomaly detection

If you need your engine to work on more than one feature, you must use multiple instances of the FDA and also include a unit to decide whether the calculated anomaly indexes indicate an anomaly. Figure 18-5 gives an idea of how it can be implemented. The sample data must be passed to all the instances of the FDA and, since these instances are stateless with respect to the caller’s scope, you can use any method to run them in parallel. A map-reduce method or simple Java’s lambda could work in either case, as shown in 18-4:

```java
ArrayList<FAD> fdaList = new ArrayList<FAD>();

fdaList.forEach(fda -> {
    fda.process(dataPoint);
});
```

18-4- Java pseudocode showing how to process all features anomaly in parallel
18.5- A detail model for multi-feature anomaly detection and its encapsulation model.

After the lambda expression in 18-4 is complete, all FDA instances will have updated information, and you can call their report methods to pass results to the unit responsible for making the final decision. As with FDA, you can wrap the process of multi-feature anomaly detection in a class named ADE and use it exactly how you used FDA.

18.4. Ensemble anomaly detection

The truth is that no one anomaly detection algorithms produces 100% satisfying results 100% of the time. There will always be some false positives, some false negatives, or both, mainly because our definition or understanding of anomalies is not comprehensive, and/or the algorithm(s) used to detect anomalies is effective only for a portion of them. So, if you are looking for a better anomaly detection system, you should use more than one model of ADE at once. Even if you do not want to implement more than one algorithm, it is worthwhile to run more than one instance of the algorithm with different configuration parameters.

18.6- Using multiple ADE instances with different algorithm or configurations.
Each ADE unit shown in 18-6 makes its decision about the anomaly index independently and feeds its output to the decision maker unit, which then decides if the current state is an anomaly or not. You can use a weighted average to calculate the final index, or, if you do not have any idea of how to choose the weights, you may start with equal weights. After observing the output for some time, you may see that one of the instances produces better results, and you can then increase its weight. This is, in fact, what a neural network does: adjusting the weights to get closer to the required result.

18.9. Helping ADE to learn better

By using some reward or punishment as positive or negative feedback mechanism based on the correct or incorrect results, we can help the learning mechanism to build a better understanding of the monitoring system. When we talked about dynamic learning in Section 15.10, we discussed this idea, but at that time it was the ADE itself that was deciding how much it should learn from the given data. Now, consider that the system has another input source that somehow lets the ADE know which of its decisions have been correct or incorrect, and “rewards” it according to its behaviour. This is called reinforcement learning. We can implement reinforcement learning in many ways. Here are four examples:

1. **Unlearning:** Unlearning means learning with negative learning factor. It is a useful way to reduce or eliminate false positives. To make an ADE unlearn something, you multiply all distributions by $1 - \lambda$ and then add a fraction of $\lambda$ based on their probability weight to all of them except for the one that was wrongly marked as an anomaly. In the simplest case, you may have a user interface indicate anomalies with alarms, so that the system administrator notices them and, then, based on his knowledge, either accepts or rejects the system’s assessment of whether something really is an anomaly. There are different ways to apply administrators’ judgment of flagged anomalies to the system, and it depends on what kind of information we keep about every alarm generated. If administrator indicates that an alarm is a false positive, then you need to find the time window where it happened, choose its belief probability distribution for all the features, and have the system unlearn the data. If it has been a true positive, ADE can learn the pattern or feature values again or just does nothing. The earlier the administrator validates alarms, the less the model must be modified.

2. **Using pre-moderated data:** Another reinforcement learning technique is letting the ADE learn from pre-moderated data. All you do is just let the ADE work and give alarms every day, but never let the ADE learn from the data unless an administrator has already verified the old processed results. The benefit of this model is that the ADE never learns corrupt data, but the problem with it is that the data and the detection results need to be stored somewhere until the administrator moderates them, and the model can be safely updated.

---

43 You might need to review Section 15.9.
3. **Monitoring multiple measures**: You can almost always find a measure that somehow has a direct correlation with the measure you are monitoring. For example, consider you are monitoring the outgoing traffic of a network with an ADE instance. You can use another instance of ADE to monitor the incoming traffic, and whenever you see that these two instances are reporting anomalies in a short time window, you increase the reliability of the anomaly alarm. The reason is that there must be a correlation between these incoming and outgoing traffic, so we can use the knowledge to help ADE reduces false positive.

4. **Monitoring different data sources**: This one works same as monitoring multiple measures, but requires a separate measure from an entirely different source of data. For example, process the quality of service (QoS) as customer support says. Apply the QoS factor to the learning factor, the better the QoS, the more reliable data to be learned.

There many more methods we can use to support the decision making of our ADE, we just need to understand how human beings find solutions. How we support different hypotheses when trying to solve a problem; which in our case is just a simple binary classification: Is it an anomaly or not?
References and more study

Unfortunately, there are not many books on anomaly detection. However, if you want to learn the concepts of learning, pattern matching, etc., here are a few resources I found very valuable.