DOI:10.1145/2890780

Article development led by acmqueue
queue.acm.org

Applying statistical techniques to operations data.

BY HEINRICH HARTMANN

Statistics for Engineers

MODERN IT SYSTEMS collect an increasing wealth of data from network gear, operating systems, applications, and other components. This data needs to be analyzed to derive vital information about the user experience and business performance. For instance, faults need to be detected, service quality needs to be measured and resource usage of the next days and month needs to be forecasted.

Rule #1: Spend more time working on code that analyzes the meaning of metrics, than code that collects, moves, stores and displays metrics. —Adrian Cockcroft[1]

Statistics is the art of extracting information from data, and hence becomes an essential tool for operating modern IT systems. Despite a rising awareness of this fact within the community (see the quote above), resources for learning the relevant statistical methods for this domain are hard to find.

The statistics courses offered in universities usually depend on their students having prior knowledge of probability, measure, and set theory, which is a high barrier of entry. Even worse, these courses often focus on parametric methods, such as t-tests, that are inadequate for this kind of analysis since they rely on strong assumptions on the distribution of data (for example, *normality*) that are not met by operations data.

This lack of relevance of classical, parametric statistics can be explained by history. The origins of statistics reach back to the 17th century, when computation was expensive and data was a sparse resource, leading mathematicians to spend a lot of effort to avoid calculations.

Today the stage has changed radically and allows different approaches to statistical problems. Consider this example from a textbook[2] used in a university statistics class:

A fruit merchant gets a delivery of 10,000 oranges. He wants to know how many of those are rotten. To find out he takes a sample of 50 oranges and counts the number of rotten ones. Which deductions can he make about the total number of rotten oranges?

The chapter goes on to explain various inference methods. The example translated to the IT domain could go as:

A DB admin wants to know how many requests took longer than one second to complete. He measures the duration of all requests and counts the number of those that took longer than one second. Done.

The abundance of computing resources has completely eliminated the need for elaborate estimations.

Therefore, this article takes a different approach to statistics. Instead of presenting textbook material on inference statistics, we will walk through four sections with *descriptive* statistical methods that are accessible and relevant to the case in point. I will discuss several visualization methods, gain a precise understanding of how to summarize data with histograms, visit classical summary statistics, and see how to replace them with robust, quantile-based alternatives. I have tried to keep prerequisite mathematical knowledge to a minimum (for example, by provid-

IMAGE BY ANDRIJ BORYS ASSOCIATES/SHUTTERSTOCK

ing source code examples along with the formulas wherever feasible. (Disclaimer: The source code is deliberately inefficient and serves only as an illustration of the mathematical calculation. Use it at your own risk!)

Visualizing Data

Visualization is the most essential data-analysis method. The human brain can process geometric information much more rapidly than numbers or language. When presented with a suitable visualization, one can almost instantly capture relevant properties such as typical values and outliers.

This section runs through the basic visualization plotting methods and discusses their properties. The Python tool chain (IPython,[8] matplotlib,[12] and Seaborn[17]) is used here to produce the plots. The section does not demonstrate how to use these tools. Many alternative plotting tools (R, MATLAB) with accompanying tutorials are available online. Source code and datasets can be found on GitHub.[4]

Rug plots. The most basic visualization method for a one-dimensional dataset $X = [x_1, ..., x_n]$ is the rug plot (Figure 1). It consists of a single axis on which little lines, called *rugs*, are drawn for each sample.

Rug plots are suitable for all questions where the temporal ordering of the samples is not relevant, such as common values or outliers. Problems occur if there are multiple samples with the same sample value in the dataset. Those samples will be indistinguishable in the rug plot. This problem can be addressed by adding a small random displacement (jitter) to the samples.

Despite its simple and honest character, the rug plot is not commonly used. Histograms or line plots are used instead, even if a rug plot would be more suitable.

Histograms. The histogram is a popular visualization method for one-dimensional data. Instead of drawing rugs on an axis, the axis is divided into bins and bars of a certain height are drawn on top of them, so that the number of samples within a bin is proportional to the area of the bar (Figure 2).

The use of a second dimension often makes a histogram easier to comprehend than a rug plot. In particular, questions such as "Which ratio of the samples lies below *y*?" can be effectively estimated by comparing areas. This convenience comes at the expense of an extra dimension used and additional choices that have to be made about value ranges and bin sizes.

Figure 1. Rug plot of Web-request rates.

Figure 2. Histogram of Web-request rates.

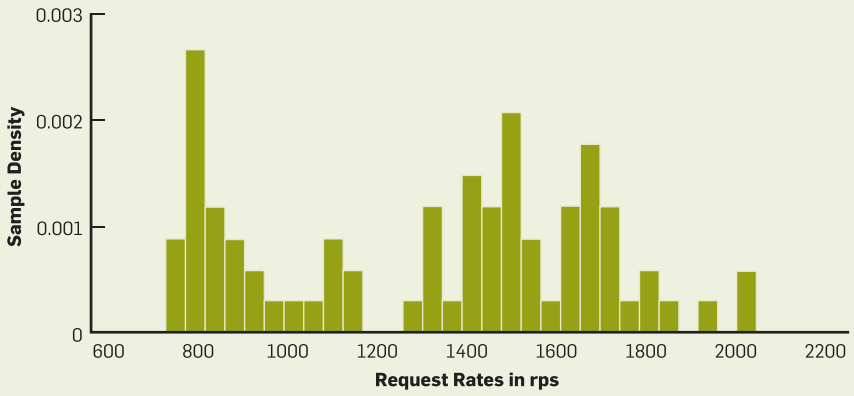
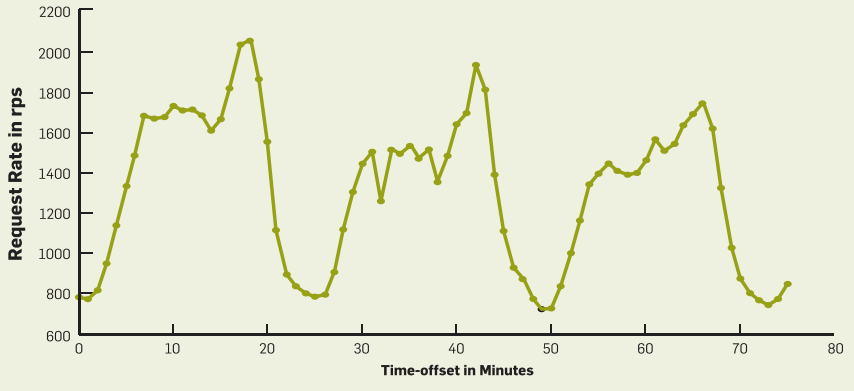


Figure 3. Line plot of Web-request rates.



Histograms are addressed in more detail later.

Scatter plots. The scatter plot is the most basic visualization of a two-dimensional dataset. For each pair of values *x*,*y* a point is drawn on a canvas that has coordinates (*x*,*y*) in a Cartesian coordinate system.

The scatter plot is a great tool to compare two metrics. Figure 4 plots the request rates of two different database nodes in a scatter plot. In the plot shown on top the points are mainly concentrated on a diagonal line, which means that if one node serves many requests, then the other is doing so as well. In the bottom plot the points are scattered all over the canvas, which represents a highly irregular load distribution, and might indicate a problem with the db configuration.

In addition to the fault-detection scenario outlined above, scatter plots are also an indispensable tool for capacity planning and scalability analysis.[3,15]

Line plots. The line plot is by far the most popular visualization method seen in practice. It is a special case of a scatter plot, where time stamps are plotted on the *x*-axis. In addition, a line is drawn between consecutive points. Figure 3 shows an example of a line plot.

The addition of the line provides the impression of a continuous transition between the individual samples. This assumption should always be challenged and taken with caution (for example, just because the CPU was idle at 1:00PM and 1:01PM, this does not mean it did not do any work in between).

Sometimes the actual data points are omitted from the visualization altogether and only the line is shown. This is a bad practice and should be avoided.

The line plot is a great tool to surface time-dependent patterns such as periods or trends. For time-independent questions—typical values, for example—other methods such as rug plots might be better suited.

Which one to use? Choosing a suitable visualization method depends on the question to be answered. Is time dependence important? Then a line plot is likely a good choice. If not, then rug plots or histograms are likely better tools. Do you want to compare different metrics with each other? Then consider using a scatter plot.

Producing these plots should become routine. Your monitoring tool might be able to provide you with some of these methods already. To get the others, figure out how to export the relevant data and import it into the software tool of your choice (Python, R, or Excel). Play around with these visualizations and see how your machine data looks.

To discover more visualization methods, check out the Seaborn gallery.[18]

Histograms

Histograms in IT operations have two different roles: as visualization method and as aggregation method.

To gain a complete understanding of histograms, let's start by building one for the Web request-rate data discussed previously. The listing in Figure 5 contains a complete implementation, discussed step by step here.

1. The first step in building a histogram is to choose a range of values that should be covered. To make this choice you need some prior knowledge about the dataset. Minimum and maximum values are popular choices in practice. In this example the value range is [500, 2200].

2. Next the value range is partitioned into bins. Bins are often of equal size, but there is no need to follow this convention. The bin partition is represented here by a sequence of bin boundaries (line 4).

3. Count how many samples of the given dataset are contained in each bin (lines 6–13). A value that lies on the boundary between two bins will be assigned to the higher bin.

4. Finally, produce a bar chart, where each bar is based on one bin, and the bar height is equal to the sample count divided by the bin width (lines 14–16). The division by bin width is an important normalization, since otherwise the bar area is not proportional to the sample count. Figure 5 shows the resulting histogram.

Different choices in selecting the range and bin boundaries of a histogram can affect its appearance considerably. Figure 6 shows a histogram with 100 bins for the same data. Note that it closely resembles a rug plot. On the other extreme, choosing a single bin would result in a histogram with a single bar with a height equal to the sample density.

Figure 4. Scatter plots of request rates of two database nodes.

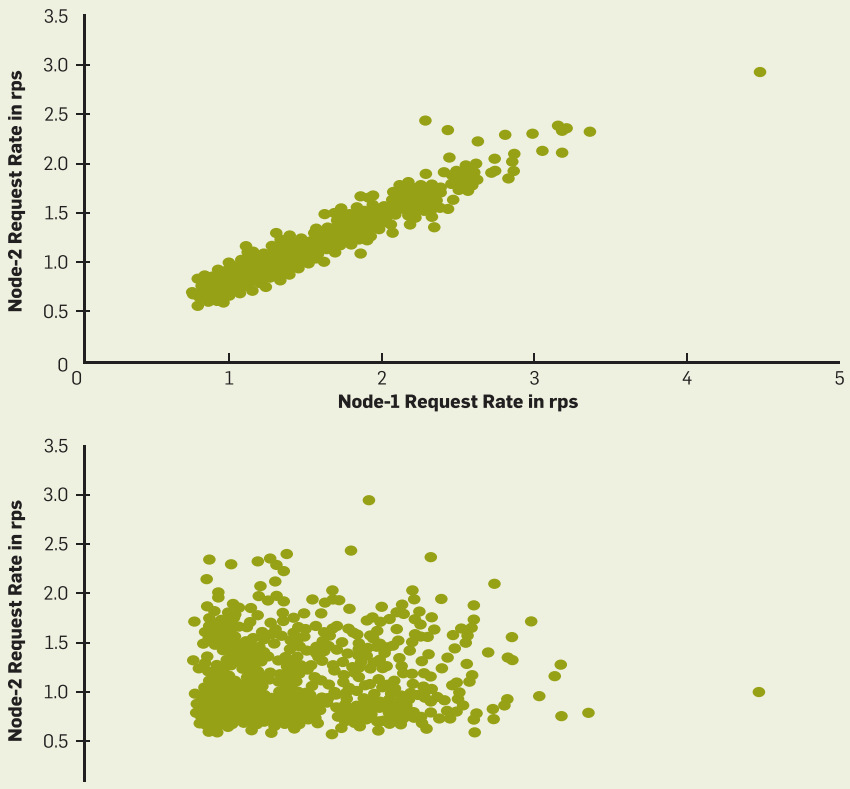


Figure 5. Result of a manual histogram implementation.

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3  X = np.genfromtxt("DataSets/RequestRates.csv", delimiter=",")[:,1]
4  bins = [500, 700, 800, 900, 1000, 1500, 1800, 2000, 2200]
5  bin_count = len(bins) - 1
6  sample_counts = [0] * bin_count
7  for x in X:
8      for i in range(bin_count):
9        if (bins[i] <= x) and (x < bins[i + 1]):
10           sample_counts[i] += 1
11 bin_widths = [ float(bins[i] - bins[i-1]) for i in range
   (1, bin_count) ]
12 bin_heights = [ count/width for count, width in zip(sample_counts,
   bin_widths) ]
13 plt.bar(bins[:bin_count-1], width=bin_widths, height=bin_heights);
```

Software products make default choices for the value range and bin width. Typically the value range is taken to be the range of the data, and equally spaced bins are used. Several formulas exist for selecting the number of bins that yield "ideal" results under certain assumptions—in particular, $n^{1/2}$ (Excel) and $3.5\sigma/n^{1/3}$ (Scott's rule[7]). In practice, these choices do not yield satisfying results when applied to operations data, such as request latencies, that contain many outliers.

Histogram as aggregation method. When measuring high-frequency data such as I/O latencies, which can arrive at rates of more than 1,000 samples per second, storing all individual samples is no longer feasible. If you are willing to forget about ordering and sacrifice some accuracy, you can save massive amounts of space by using histogram data structures.

The essential idea is, instead of storing the individual samples as a list, to use the vector of bin counts that occurs as an intermediate result in the histogram computation. The example in Figure 5 arrived at the following values:

```
sample_count = [0, 10, 8, 4, 25, 23, 4, 2]
```

The precise memory representation used for storing histograms varies. The important point is that the sample count of each bin is available.

Histograms allow approximate computation of various summary statistics, such as mean values and quantiles that will be covered next. The precision depends on the bin sizes.

Also, histograms can be aggregated easily. If request latencies are available for each node of a database cluster in histograms with the same bin choices, then you can derive the latency distribution of the whole cluster by adding the sample counts for each bin. The aggregated histogram can be used to calculate mean values and quantiles over the whole cluster. This is in contrast to the situation in which mean values or quantiles are computed for the nodes individually. It is not possible to derive, for example, the 99th-percentile of the whole cluster from the 99th-percentiles of the individual nodes.[14]

High-dynamic range histogram. An high-dynamic range (HDR) histogram provides a pragmatic choice for bin width that allows a memory-efficient representation suitable for capturing data on a very wide range that is common to machine-generated data such as I/O latencies. At the same time HDR histograms tend to produce acceptable visual representations in practice.

An HDR histogram changes the bin width dynamically over the value range. A typical HDR histogram has a bin size of 0.1 between 1 and 10, with bin boundaries: 1, 1.1, 1.2,..., 9.9, 10. Similarly, between 10 and 100 the bin size is 1, with boundaries 10, 11, 12, ..., 100. This pattern is repeated for all powers of 10, so that there are 90 bins between 10^k and 10^{k+1}.

The general definition is a little bit more complex and lengthy, so it is not

Figure 6. Histogram plot with value range (500, 2,200) and 100 equally sized bins.

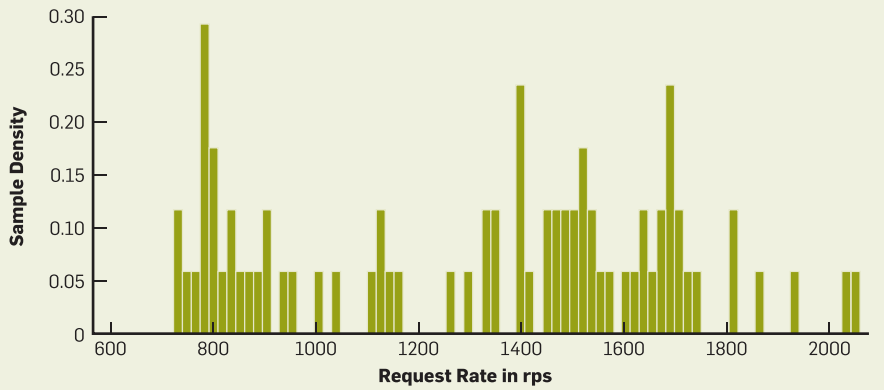


Figure 7. Request rate histogram (50 bins) presented as a heat map.

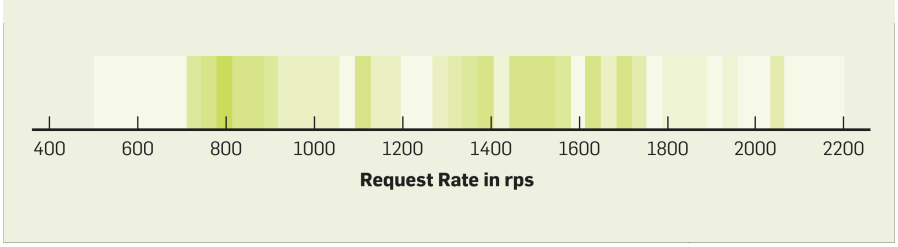


Figure 8. Request latency heat map over time in Circonus.

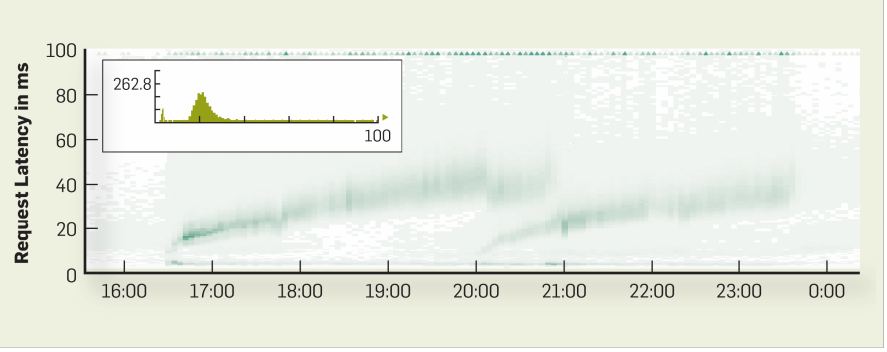
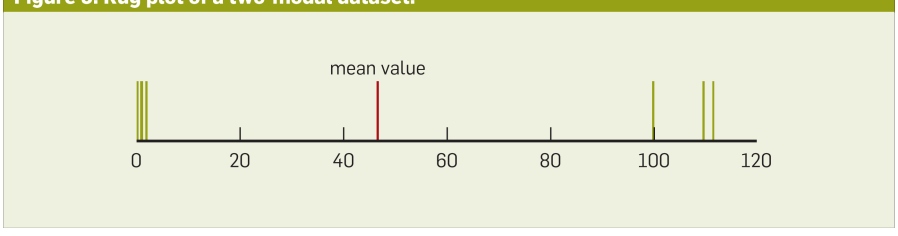


Figure 9. Rug plot of a two-modal dataset.



provided here, but interested readers can refer to http://hdrhistogram.org/[6] for more details and a memory-efficient implementation.

From the previous description it should be apparent that HDR histograms span an extremely large range of values. Bin sizes are similar to float-number precisions: the larger the value, the less precision is available. In addition, bin boundaries are independent of the dataset. Hence, the aggregation technique described earlier applies to HDR histograms.

Histograms as heat maps. Observing the change of data distributions over time requires an additional dimension on the histogram plot. A convenient method of doing so is to represent the sample densities as a heat map instead of a bar chart. Figure 7 shows the request-rate data visualized in such a way. Light colors mean low sample density, dark colors signal high sample density.

Multiple histogram heat maps that were captured over time can be combined into a single two-dimensional heat map.

Figure 8 shows a particularly interesting example of such a visualization for a sequence of HDR histograms of Web-request latencies. Note that the distribution of the data is bimodal, with one mode constant around ~5ms and another more diffuse mode ascending from ~10ms to ~50ms. In this particular case the second mode was caused by a bug in a session handler, which kept adding new entries to a binary tree. This tree had to be traversed for each incoming request, causing extended delays. Even the logarithmic growth of the average traversal time can be spotted if you look carefully.

Classical Summary Statistics

The aim of summary statistics is to provide a summary of the essential features of a dataset. It is the numeric equivalent of an "elevator pitch" in a business context: just the essential information without all the details.

Good summary statistics should be able to answer questions such as "What are typical values?" or "How much variation is in the data?" A desirable property is robustness against outliers. A single faulty measurement should not change a rough description of the dataset.

This section looks at the classical summary statistics: mean values and standard deviations. In the next section we will meet their quantile-based counterparts, which are more robust against outliers.

Figure 10. Ping latency spike on a view range of 6H vs. 48H.

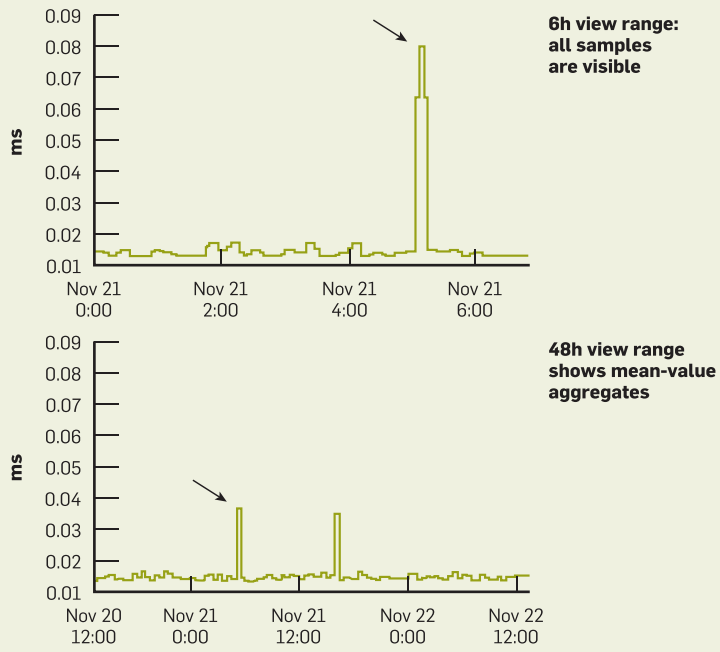
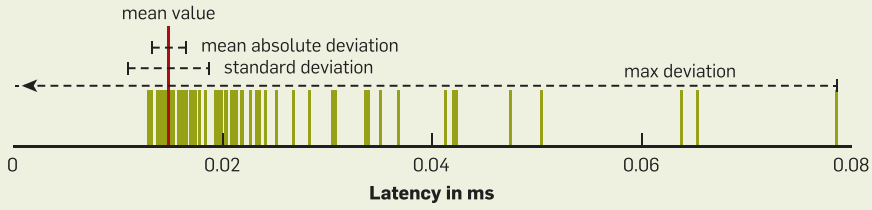


Figure 11. A request latency dataset.



Mean value or *average* of a dataset $X = [x_1,..., x_n]$ is defined as

$$\mu(x_1,...,x_n) = \frac{1}{n}\Sigma_{i=1}^{n} x_i$$

or when expressed as Python code:

```
def mean(X): return sum(X) / len(X)
```

The mean value has the physical interpretation of the center of mass if weights of equal weight are placed on the points x_i on a (mass-less) axis. When the values of x_i are close together, the mean value is a good representation of a typical sample. In contrast, when the samples are concentrated at several centers, or outliers are present, the mean value can be far from each individual data point (Figure 9).

Mean values are abundant in IT operations. One common application of mean values is data rollup. When multiple samples arrive during a sampling period of one minute, for example, the mean value is calculated as a "one-minute rollup" and stored instead of the original samples. Similarly, if data is available for every minute, but you are interested only in hour intervals, you can "roll up the data by the hour" by taking mean values.

Despite their abundance, mean values lead to a variety of problems when measuring performance of services. To quote Dogan Ugurlu from

Figure 12. The cumulative distribution function for a dataset of request rates.

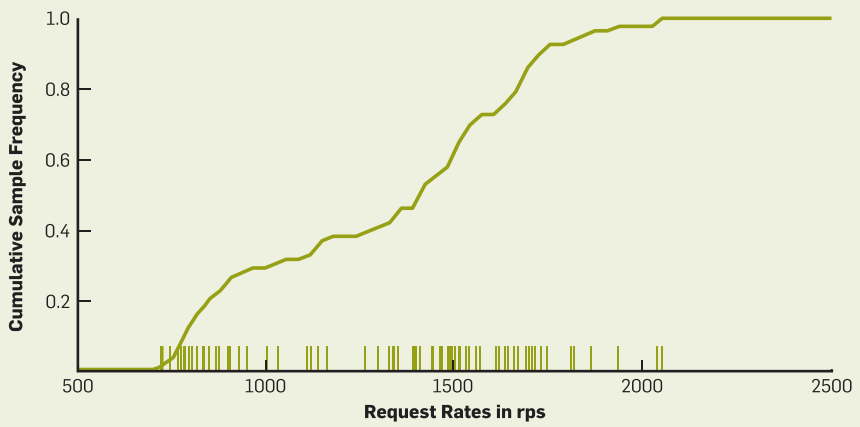


Figure 13. Histogram metric with quantile (QMIN(0.8) over 1H windows.

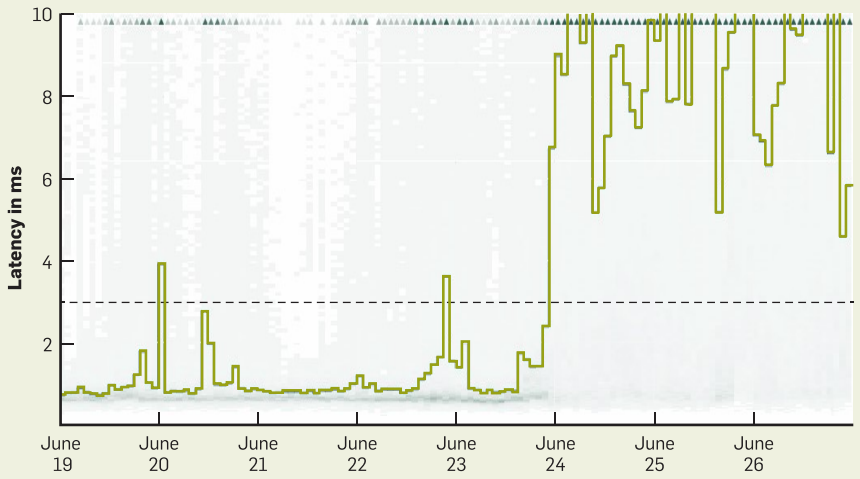
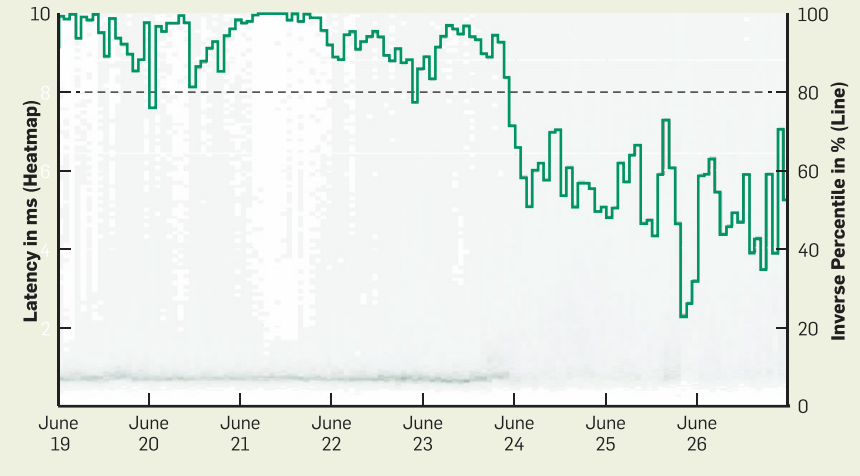


Figure 14. Histogram metric with inverse quantile CF D(3ms) over 1H windows.



Optimizely.com: "Why? Because looking at your average response time is like measuring the average temperature of a hospital. What you really care about is a patient's temperature, and in particular, the patients who need the most help."[16] In the next section we will meet median values and quantiles, which are better suited for this kind of performance analysis.

Spike erosion. Viewing metrics as line plots in a monitoring system often reveals a phenomenon called *spike erosion*.[5] To reproduce this phenomenon, select a metric (for example, ping latencies) that experiences spikes at discrete points in time and zoom in on one of those spikes and read the height of the spike at the *y*-axis. Now zoom out of the graph and read the height of the same spike again. Are they equal?

Figure 10 shows such a graph. The spike height has decreased from `0.8` to `0.35`.

How is that possible? The result is an artifact of a rollup procedure that is commonly used when displaying graphs over long time ranges. The amount of data gathered over the period of one month (more than 40,000 minutes) is larger than the amount of pixels available for the plot. Therefore, the data has to be rolled up to larger time periods before it can be plotted. When the mean value is used for the rollups, the single spike is averaged with an increasing number of "normal" samples and hence decreases in height.

How to do better? The immediate way of addressing this problem is to choose an alternative rollup method, such as max values, but this sacrifices information about typical values. Another more elegant solution is to roll up values as histograms and display a two-dimensional heat map instead of a line plot for larger view ranges.

Deviation measures. Once the mean value μ of a dataset has been established, the next natural step is to measure the deviation of the individual samples from the mean value. The following three deviation measures are often found in practice.

The *maximal deviation* is defined as $maxdev(x_1,\ldots,x_n) = max\{|x_i - \mu| \mid i = 1,\ldots,n\}$, and gives an upper bound for the distance to the mean in the dataset.

The *mean absolute deviation* is defined as $mad(x_1,\ldots,x_n) = \frac{1}{n}\Sigma_{i=1}^{n}|x_i - \mu|$ and is the most direct mathematical translation of a typical deviation from the mean.

The standard deviation is defined as $stddev(x_1,\ldots,x_n) = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(x_i-\mu)^2}$

While the intuition behind this definition is not obvious, this deviation measure is very popular for its nice mathematical properties (as being derived from a quadratic form). In fact, all three of these deviation measures fit into a continuous family of *p*-deviations,[11] which features the standard deviation in a "central" position.

Figure 11 shows the mean value and all three deviation measures for a request latency dataset. You can immediately observe the following inequalities:

$$mad(x_1,\ldots,x_n) \le stddev(x_1,\ldots,x_n) \le maxdev(x_1,\ldots,x_n)$$

This relation can be shown to hold true in general.

The presence of outliers affects all three deviation measures significantly. Since operations data frequently contains outliers, the use of these diviation measures should be taken with caution or avoided all together. More robust methods (for example, interquartile ranges[a]) are based on quartiles, which we discuss below.

Caution with the standard deviation. Many of us remember the following rule of thumb from school:

- 68% of all samples lie within one standard deviation of the mean.
- 95% of all samples lie within two standard deviations of the mean.
- 99.7% of all samples lie within three standard deviations of the mean.

These assertions rely on the crucial assumption the data is normally distributed. For operations data this is almost never the case, and the rule fails quite drastically: in the previous example more than 0.97% lie within one standard deviation of the mean value.

The following war story can be found in P.K. Janert's book:[10]

An SLA (service level agreement) for a database defined a latency outlier as a value outside of three standard deviations. The programmer who implemented the SLA check remembered the above rule naively and computed the latency of the slowest 0.3 percent of the queries instead.

This rule has little to do with the original definition in practice. In particular, this rule labels 0.3% of each dataset blindly as outliers. Moreover, it turned out that the reported value captured long-running batch jobs that were on the order of hours. Finally, the programmer hard-coded a seemingly reasonable threshold value of ~50 seconds, and that was reported as the "three standard deviations," regardless of the actual input.

The actual SLA was never changed.

Despite their abundance, mean values lead to a variety of problems when measuring performance of services.

Quantiles and Outliers

The classical summary statistics introduced in the previous section are well suited for describing homogeneous distributions but are easily affected by outliers. Moreover, they do not contain much information about the tails of the distribution.

A quantile is a flexible tool that offers an alternative to the classical summary statistics, which is less susceptible to outliers.

Before introducing quantiles, we need to recall the following concept. The (empirical) cumulative distribution function $CDF(y)$ for dataset X, at a value y, is the ratio of samples that are lower than the value y:

$$CDF(X,y) = \#\{i|x_i \le y\}/\#X$$

Or expressed in Python code:

```
def CDF(X,y):
 lower_cnt = 0.0
 for x in X:
  if x<=y: lower_cnt += 1
 return lower_cnt / len(X)
```

Figure 12 shows an example for a dataset of request rates. Note $CDF(X,y)$ takes values between 0 and 1 and is monotonically increasing as a function of y.

Now we turn to the definition of quantiles. Fix a number q between 0 and 1 and a dataset X of size n. Roughly speaking, a q-quantile is a number y that divides X into two sides, with a ratio of q samples lying below y and the remaining ratio of $1 - q$ samples lying above y.

More formally, a q-quantile for X is a value y such that:

a http://en.wikipedia.org/wiki/Interquartile_range

▸ at least $q \times n$ samples are less than or equal to y;

▸ at least $(1-q) \times n$ samples are greater than or equal to y.

Familiar examples are the minimum, which is a 0-quantile; the maximum, which is a 1-quantile; and the median, which is a 0.5-quantile. Common names for special quantiles include percentiles for k/100-quantiles and quartiles for k/4-quantiles.

Note that quantiles are not unique. There are ways of making them unique, but those involve a choice that is not obvious. Wikipedia lists nine different choices that are found in common software products.[13] Therefore, if people talk about the q-quantile or the median, one should always be careful and question which choice was made.

As a simple example of how quantiles are non-unique, take a dataset with two values $X = [10,20]$. Which values are medians, 0-quantiles, 0.25-quantiles? Try to figure it out yourself.

The good news is that q-quantiles always exist and are easy to compute. Indeed, let S be a sorted copy of the dataset X such that the smallest element X is equal to $S[0]$ and the largest element of X is equal to $S[n-1]$. If $d = \text{floor}(q \times (n-1))$, then $S[d]$ will have d+1 samples $S[0],\ldots,S[d]$, which are less than or equal to $S[d]$, and $n - d + 1$ samples $S[d],\ldots, S[n]$, which are greater than or equal to $S[d]$. It follows that $S[d] = y$ is a q-quantile. The same argument holds true for $d = \text{ceil}(q \times (n-1))$.

The following listing is a Python implementation of this construction:

```
def quantile_range(X,q):
 S = sorted(X)
 r = q * (len(X)-1)
 return (
  S[int(math.floor(r))],
  S[int(math.ceil(r))]
 )
```

It is not difficult to see this construction consists of the minimal and maximal possible q-quantiles. The notation $Q_{min}(X, q)$ represents the minimal q-quantile. The minimal quantile has the property $Q_{min}(X, q) \leq y$ if and only if at least $n \times q$ samples of X are less than or equal to y. A similar statement holds true for the maximal quantile when checking ratios of samples that are greater than y.

Quantiles are closely related to the cumulative distribution functions discussed in the previous section. Those concepts are inverse to each other in the following sense: If $CDF(X, y) = q$, then y is a q-quantile for X. Because of this property, cumulative distribution function values are also referred to as *inverse quantiles*.

Applications to Service-Level Monitoring

Quantiles and CDFs provide a powerful method to measure service levels. To see how this works, consider the following SLA that is still commonly seen in practice: "The mean response time of the service shall not exceed three milliseconds when measured each minute over the course of one hour."

This SLA does not do a good job of capturing the service experience of consumers. First, the requirement can be violated by a single request that takes more than 90ms to complete. Also, a long period where low overall load causes the measured request to finish within 0.1ms can compensate for a short period where lots of external requests are serviced with unacceptable response times of 100ms or more.

An SLA that captures the quality of service as experienced by the customers looks like this: 80% of all requests served by the API within one hour should complete within 3ms.

Not only is this SLA easier to formulate, but it also avoids the above problems. A single long-running request does not violate the SLA, and a busy period with long response times will violate the SLA if more than 20% of all queries are affected.

To check the SLA, here are two equivalent formulations in terms of quantiles and CFDs:

▸ The minimal 0.8-quantile is at most 3ms: $Q_{min}(X_{1h}, 0.8) \leq 3ms$.

▸ The 3-ms inverse quantile is larger than 0.8: $CDF(X_{1h}, 3ms) \geq 0.8$.

Here X_{1h} denotes the samples that lie within a one-hour window. Both formulations can be used to monitor service levels effectively. Figure 13 shows $Q_{min}(X_{1h}, 0.8)$ as a line plot. Note how on June 24 the quantile rises above 3ms, indicating a violation of the SLA. Figure 14 shows a plot of the inverse quantile $CDF(X_{1h}, 3ms)$, which takes values on the right axis between 0% and 100%. The SLA violation manifests as the inverse quantile dropping below 80%.

Hence, quantiles and inverse quantiles give complementary views of the current service level.

Conclusion

This article has presented an overview of some statistical techniques that find applications in IT operations. We discussed several visualization methods, their qualities and relations to each other. Histograms were shown to be an effective tool for capturing data and visualizing sample distributions. Finally, we have seen how to analyze request latencies with (inverse) percentiles. C

References

1. Cockcroft, A. Monitorama—Please, no more Minutes, Milliseconds, Monoliths or Monitoring Tools, 2014; http://de.slideshare.net/adriancockcroft/monitorama-please-no-more
2. Georgii, H.-O. *Stochastik*. DeGruyter, 2002.
3. Gunther, N.J. *Guerrilla Capacity Planning*. Springer-Verlag, Berlin, 2007.
4. Hartmann, H. Statistics for Engineers, 2015; https://github.com/HeinrichHartmann/Statistics-for-Engineers.
5. Hartmann, H. Show Me the Data, 2016; http://www.circonus.com/spike-erosion.
6. HDR Histogram: A high dynamic range histogram; http://hdrhistogram.org/.
7. Histogram; https://en.wikipedia.org/wiki/Histogram.
8. IPython; http://ipython.org.
9. Izenman, A.J. *Modern Multivariate Statistical Techniques*. Springer-Verlag, New York, 2008.
10. Janert, P.K. *Data Analysis with Open Source Tools*. O'Reilly, 2010.
11. L^p space; https://en.wikipedia.org/wiki/Lp_space.
12. Matplotlib; http://matplotlib.org.
13. Quantile; https://en.wikipedia.org/wiki/Quantile.
14. Schlossnagle, T. The problem with math: why your monitoring solution is wrong, 2015; http://www.circonus.com/problem-math/.
15. Schwarz, B. Practical Scalability Analysis with the Universal Scalability Law, 2015; https://www.vividcortex.com/resources/universal-scalability-law.
16. Ugurlu, D. The Most Misleading Measure of Response Time: Average, 2013; https://blog.optimizely.com/2013/12/11/why-cdn-balancing/
17. Waskom, M. Seaborn: statistical data visualization, 2012-2015; http://stanford.edu/~mwaskom/software/seaborn/.
18. Waskom, M. Seaborn example gallery, 2012–2015; http://stanford.edu/~mwaskom/software/seaborn/examples/index.html.

Heinrich Hartmann is chief data scientist for the Circonus monitoring platform, leading the efforts to make Circonus the best tool to monitor APIs and services. Previously, he worked as an independent consultant for a number of different companies and research institutions.