Chapter 3. Statistics

Applying the basic principles of statistics to data science provides vital insight into our data. Statistics is a powerful tool. Used correctly, it enables us to be sure of our decision-making process. However, it is easy to use statistics incorrectly. One example is Anscombe's quartet (Figure 3-1), which demonstrates how four distinct datasets can have nearly identical statistics. In many cases, a simple plot of the data can alert us right away to what is really going on with the data. In the case of Anscombe's quartet, we can instantly pick out these features: in the upper-left panel, x and y appear to be linear, but noisy. In the upper-right panel, we see that x and y form a peaked relationship that is nonlinear. In the lower-left panel, x and y are precisely linear, except for one outlier. The lower-right panel shows that y is statistically distributed for $x = 8$ and that there is possibly an outlier at $x = 19$. Despite how different each plot looks, when we subject each set of data to standard statistical calculations, the results are identical. Clearly, our eyes are the most sophisticated data-processing tool in existence! However, we cannot always visualize data in this manner. Many times the data will be multidimensional in x and perhaps y as well. While we can plot each dimension of x versus y to get some ideas on the characteristics of the dataset, we will be missing all of the dependencies between the variates in x.

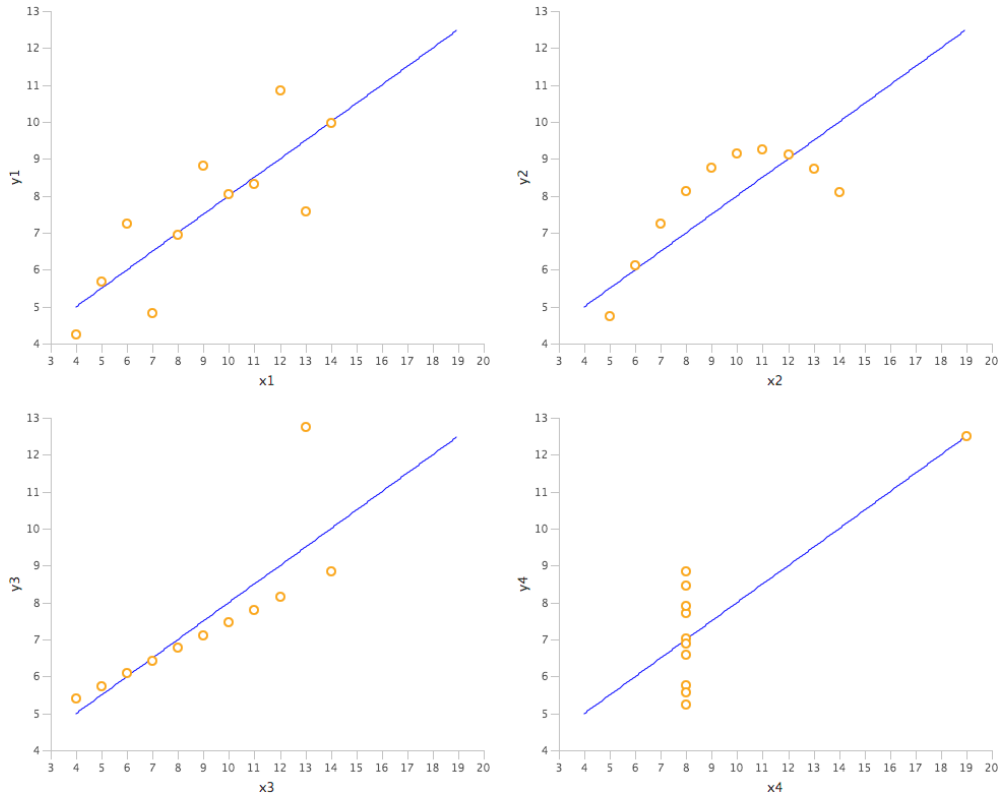

Figure 3-1. Anscombe's quartet

The Probabilistic Origins of Data

At the beginning of the book, we defined a *data point* as a recorded event that occurs at that exact time and place. We can represent a datum (data point) with the dirac delta δ(**x**), which is equal to zero everywhere except at **x** = 0, where the value is ∞. We can generalize a little further with $\delta(x - x_i)$, which means that the dirac delta is equal to zero everywhere except at $x = x_i$, where the value is ∞. We can ask the question, is there something driving the occurrence of the data points?

Probability Density

Sometimes data arrives from a well-known, generating source that can be described by a functional form $f(x)$, where typically the form is modified by some parameters θ and is denoted as $f(x;\ \theta)$. Many forms of $f(x)$ exist, and most of them come from observations of behavior in the natural world. We will explore some of the more common ones in the next few sections for both continuous and discrete random-number distributions.

We can add together all those probabilities for each position as a function of the variate *x*:

$$f(x) = \sum_{i=1}^{n} p_i \delta(x - x_i)$$

Or for a discrete integer variate, *k*:

$$f(x) = f(k)\delta(x - k)$$

Note that *f(x)* can be greater than 1. Probability density is not the probability, but rather, the local density. To determine the probability, we must integrate the probability density over an arbitrary range of *x*. Typically, we use the cumulative distribution function for this task.

Cumulative Probability

We require that probability distribution functions (PDFs) are properly normalized such that integrating over all space returns a 100 percent probability that the event has occurred:

$$F = \int_{-\infty}^{\infty} f(x)\,\mathrm{d}x = 1$$

However, we can also calculate the cumulative probability that an event will occur at point *x*, given that it has not occurred yet:

$$F(x) = \int_{-\infty}^{x} f(x')\,\mathrm{d}x'$$

Note that the cumulative distribution function is monotonic (always increasing as *x* increases) and is (almost) always a sigmoid shape (a slanted *S*). Given that an event has not occurred yet, what is the probability that it will occur at *x*? For large values of *x*, P = 1. We impose this condition so that we can be sure that the event definitely happens in some defined interval.

Statistical Moments

Although integrating over a known probability distribution $f(x)$ gives the cumulative distribution function (or 1 if over all space), adding in powers of *x* are what defines the statistical moments. For a known statistical distribution, the statistical moment is the expectation of order *k* around a central point *c* and can be evaluated via the following:

$$\mu_k = \int_{-\infty}^{\infty} (x - c)^k f(x)\,\mathrm{d}x$$

Special quantity, the expectation or average value of *x*, occurs at c = 0 for the first moment *k* = 1:

$$\mu = \int_{-\infty}^{\infty} x f(x)\,\mathrm{d}x$$

The higher-order moments, *k* > 1, with respect to this mean are known as the *central moments* about the mean and are related to descriptive statistics. They are expressed as follows:

$$\mu_{k>1} = \int_{-\infty}^{\infty} (x - \mu)^k f(x)\, \mathrm{d}x$$

The second, third, and fourth central moments about the mean have useful statistical meanings. We define the variance σ^2 as the second moment:

$$\sigma^2 = \mu_2$$

Its square root is the standard deviation σ, a measure of how far the data is distributed from the mean. The skewness γ is a measure of how asymmetric the distribution is and is related to the third central moment about the mean:

$$\gamma = \frac{\mu_3}{\sigma^3}$$

The *kurtosis* is a measure of how fat the tails of the distribution are and is related to the fourth central moment about the mean:

$$\kappa = \frac{\mu_4}{\sigma^4}$$

In the next section, we will examine the normal distribution, one of the most useful and ubiquitous probability distributions. The normal distribution has a kurtosis kappa = 3. Because we often compare things to the normal distribution, the term *excess kurtosis* is defined by the following:

$$\kappa = \frac{\mu_4}{\sigma^4} - 3$$

We now define *kurtosis* (the fatness of the tails) in reference to the normal distribution. Note that many references to the kurtosis are actually referring to the excess kurtosis. The two terms are used interchangeably.

Higher-order moments are possible and have varied usage and applications on the fringe of data science. In this book, we stop at the fourth moment.

Entropy

In statistics, *entropy* is the measure of the unpredictability of the information contained within a distribution. For a continuous distribution, the entropy is as follows:

$$\mathcal{H}(p) = \int_{-\infty}^{\infty} p(x)\log_b(p(x))\,\mathrm{d}x$$

For a discrete distribution, entropy is shown here:

$$\mathcal{H}(p) = -\sum_i p(x_i)\log_b(p(x_i))$$

In an example plot of entropy (Figure 3-2), we see that entropy is lowest when the probability of a 0 or 1 is high, and the entropy is maximal at $p = 0.5$, where both 0 and 1 are just as likely.

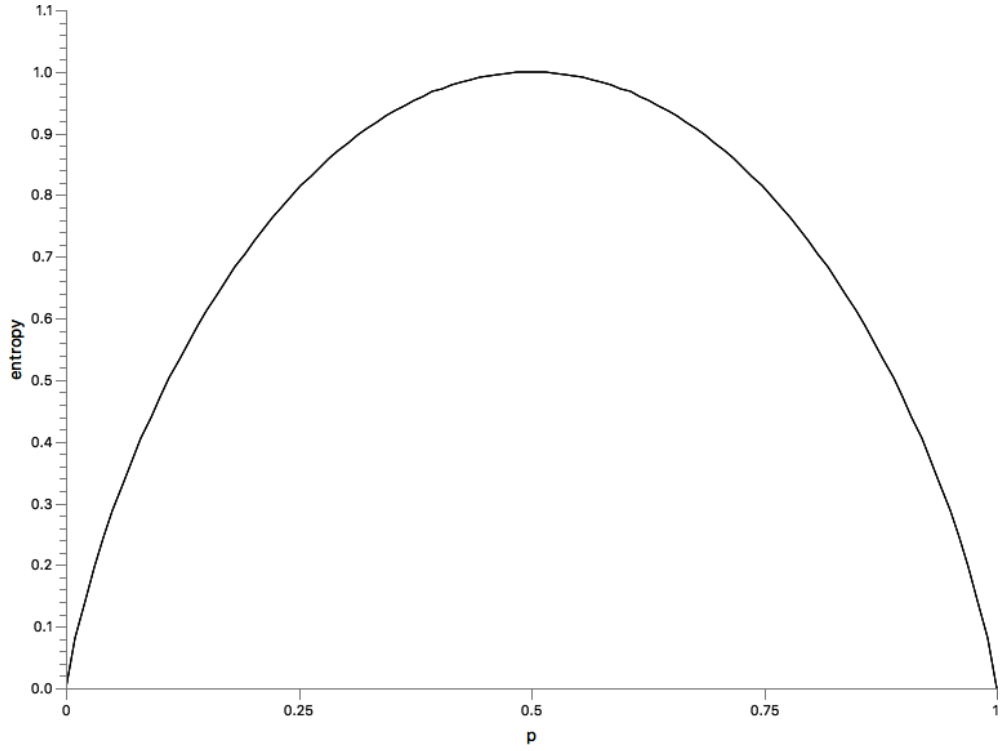

Figure 3-2. Entropy for a Bernoulli distribution

We can also examine the entropy between two distributions by using cross entropy, where *p(x)* is taken as the true distribution, and *q(x)* is the test distribution:

$$\mathcal{H}(p,\ q) = \int_{-\infty}^{\infty} p(x)\log_b(q(x))\,\mathrm{d}x$$

And in the discrete case:

$$\mathcal{H}(p, q) = -\sum_i p(x_i)\log_b(q(x_i))$$

Continuous Distributions

Some well-known forms of distributions are well characterized and get frequent use. Many distributions have arisen from real-world observations of natural phenomena. Regardless of whether the variates are described by real or integer numbers, indicating respective continuous and discrete distributions, the basic principles for determining the cumulative probability, statistical moments, and statistical measures are the same.

UNIFORM

The *uniform distribution* has a constant probability density over its supported range $x \in [a, b]$, and is zero everywhere else. In fact, this is just a formal way of describing the random, real-number generator on the interval `[0,1]` that you are familiar with, such as `java.util.Random.nextDouble()`. The default constructor sets a lower bound $a = 0.0$ and an upper bound of $b = 1.0$. The probability density of a uniform distribution is expressed graphically as a *top hat* or *box* shape, as shown in Figure 3-3.

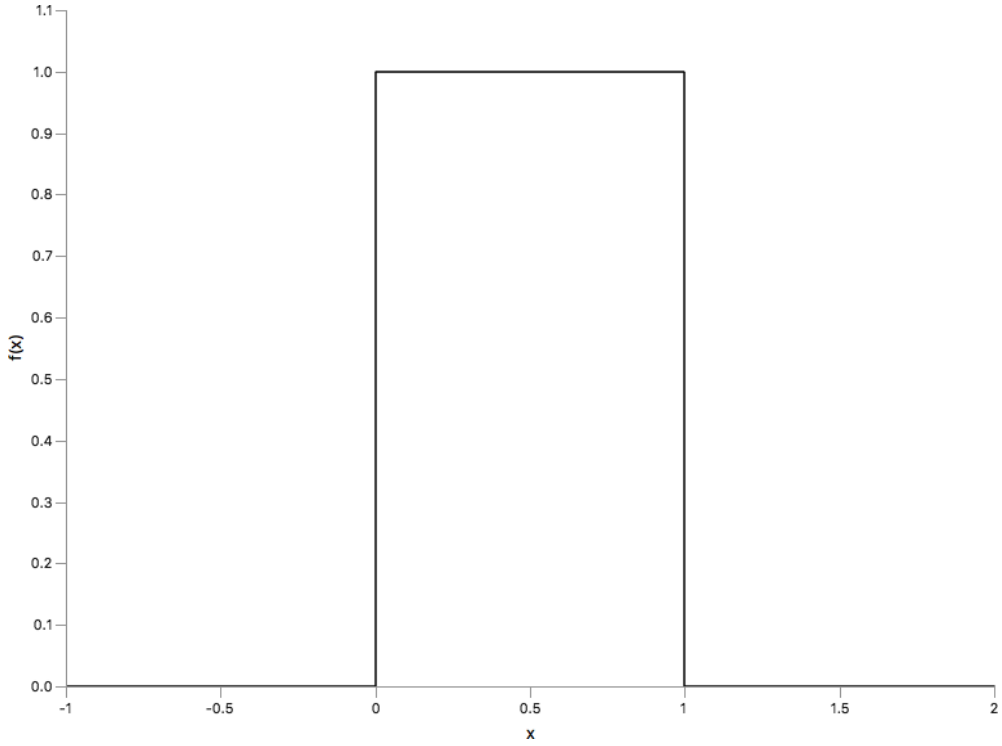

Figure 3-3. Uniform PDF with parameters $a = 0$ *and* $b = 1$

This has the following mathematical form:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

The cumulative distribution function (CDF) looks like Figure 3-4.

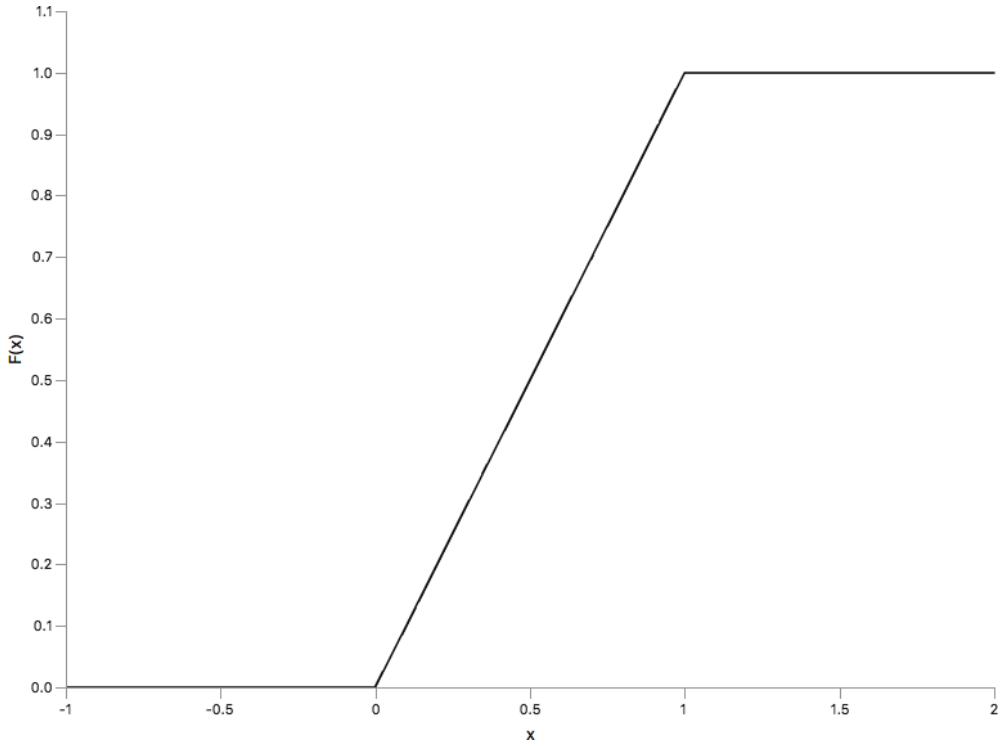

Figure 3-4. Uniform CDF with parameters $a = 0$ *and* $b = 1$

This has the form shown here:

$$F(x) = \begin{cases} 0 & \text{for } x < a \\ \frac{x-a}{b-a} & \text{for } x \in [a,\ b) \\ 1 & \text{for } x \geq b \end{cases}$$

In the uniform distribution, the mean and the variance are not directly specified, but are calculated from the lower and upper bounds with the following:

$$\mu = \frac{1}{2}(a+b)$$

$$\sigma^2 = \frac{1}{12}(b-a)^2$$

To invoke the uniform distribution with Java, use the class `UniformDistribution(a, b)`, where the lower and upper bounds are specified in the constructor. Leaving the constructor arguments blank invokes the standard uniform distribution, where $a = 0.0$ and $b = 1.0$.

```
UniformRealDistribution dist = new UniformRealDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // 1.0
double mean = dist.getNumericalMean();
double variance = dist.getNumericalVariance();
double standardDeviation = Math.sqrt(variance);
double probability = dist.density(0.5));
double cumulativeProbability = dist.cumulativeProbability(0.5);
double sample = dist.sample(); // e.g., 0.023
double[] samples = dist.sample(3); // e.g., {0.145, 0.878, 0.431}
```

Note that we could reparameterize the uniform distribution with $a = \mu - \delta$ and $b = \mu + \delta$, where μ is the center point (the mean) and δ is the distance from the center to either the lower or upper bound. The variance then becomes $\sigma^2 = \frac{\delta^2}{3}$ with standard deviation $\sigma = \frac{\delta}{\sqrt{3}}$. The PDF is then

$$f(x) = \begin{cases} \frac{1}{2\delta} & \text{for } x \in [\mu - \delta, \mu + \delta] \\ 0 & \text{otherwise} \end{cases}$$

and the CDF is

$$F(x) = \begin{cases} 0 & \text{for } x < \mu - \delta \\ \frac{1}{2}(1 + \frac{x - \mu}{\delta}) & \text{for } x \in [\mu - \delta, \mu + \delta] \\ 1 & \text{for } x \geq \mu + \delta \end{cases}$$

To express the uniform distribution in this centralized form, calculate $a = \mu - \delta$ and $b = \mu + \delta$ and enter them into the constructor:

```
/* initialize centralized uniform with mean = 10 and half-width = 2 */
double mean = 10.0
double hw = 2.0;
double a = mean - hw;
double b = mean + hw;
UniformRealDistribution dist = new UniformRealDistribution(a, b);
```

At this point, all of the methods will return the correct results without any further alterations. This reparameterization around the mean can be useful when trying to compare distributions. The centered, uniform distribution is naturally extended by the normal distribution (or other symmetric peaked distributions).

NORMAL

The most useful and widespread distribution, found in so many diverse use cases, is the normal distribution. Also known as the *Gaussian distribution* or the *bell curve*, this distribution is symmetric about a central peak whose width can vary. In many cases when we refer to something as having an average value with plus or minus a certain amount, we are referring to the normal distribution. For example, for exam grades in a classroom, the interpretation is that a few people do really well and a few people do really badly, but most people are average or right in the middle. In the normal distribution, the center of the distribution is the

maximum peak and is also the mean of the distribution, μ. The width is parameterized by σ and is the standard deviation of the values. The distribution supports all values of $x \in [-\infty, \infty]$ and is shown in Figure 3-5.

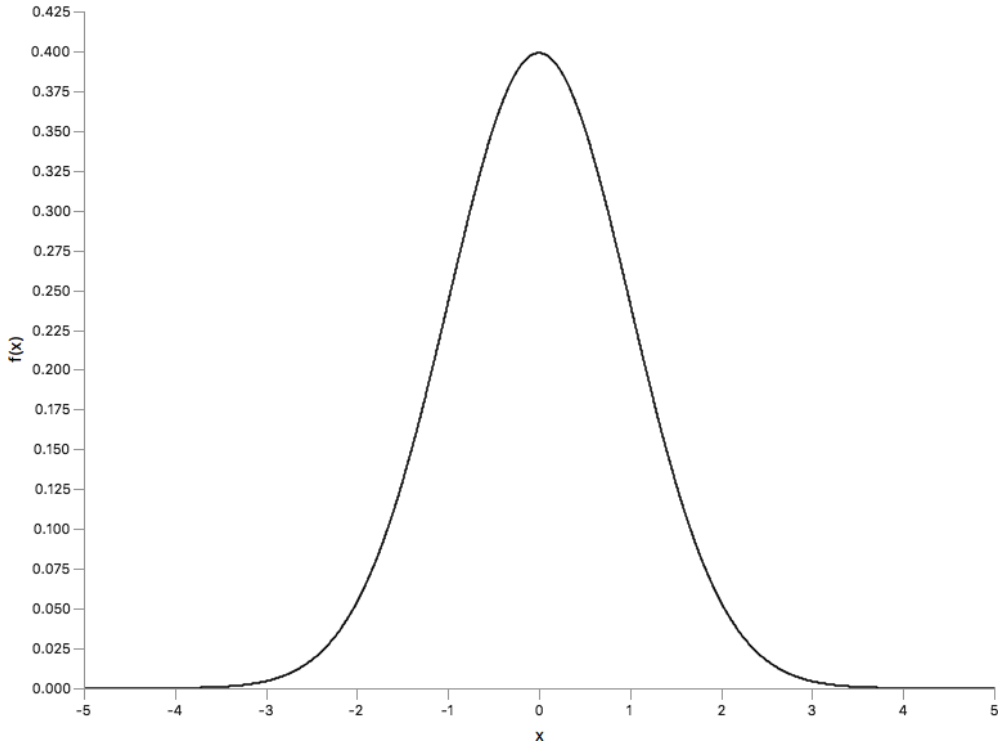

Figure 3-5. Normal PDF with parameters $\mu = 0$ *and* $\sigma = 1$

The probability density is expressed mathematically as follows:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The cumulative distribution function is shaped like Figure 3-6.

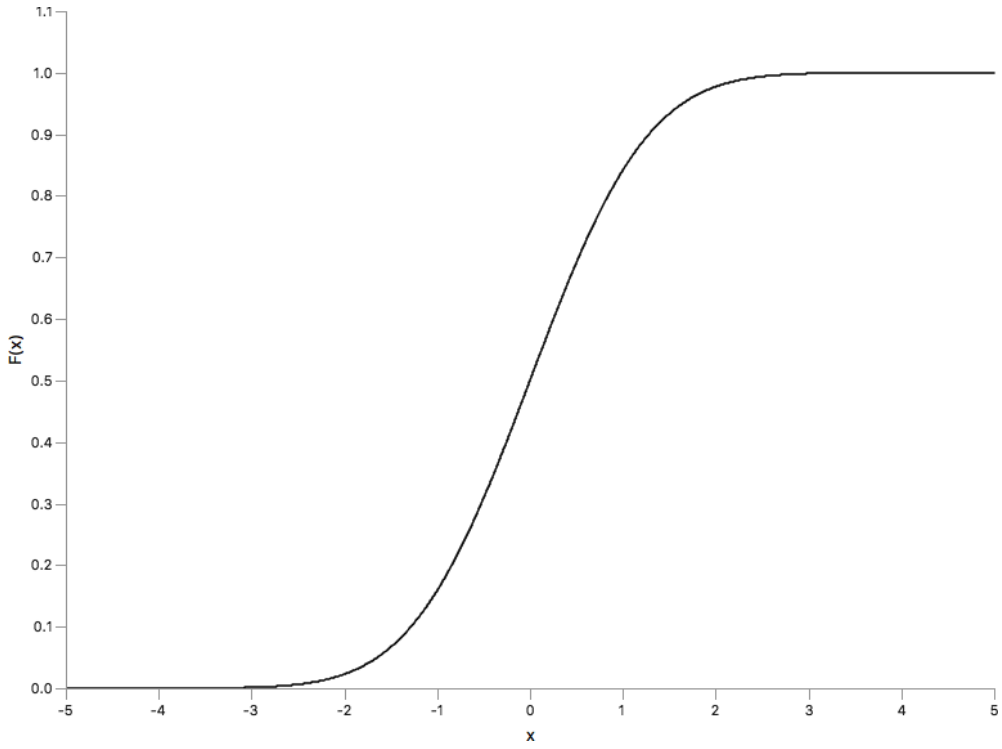

Figure 3-6. Normal CDF with parameters $\mu = 0$ *and* $\sigma = 1$

This is expressed with the error function:

$$F(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x - \mu}{\sigma\sqrt{2}}\right)\right]$$

This is invoked via Java, where the default constructor creates the standard normal distribution with $\mu = 0.0$ and $\sigma = 1.0$. Otherwise, pass the parameters μ and σ to the constructor:

```
/* initialize with default mu=0 and sigma=1 */
NormalDistribution dist = new NormalDistribution();
double mu = dist.getMean(); // 0.0
double sigma = dist.getStandardDeviation(); // 1.0
double mean = dist.getNumericalMean(); // 0.0
double variance = dist.getNumericalVariance(); // 1.0
double lowerBound = dist.getSupportLowerBound(); // -Infinity
double upperBound = dist.getSupportUpperBound(); // Infinity
/* probability at a point x = 0.0 */
double probability = dist.density(0.0);
/* calc cum at x=0.0 */
double cumulativeProbability = dist.cumulativeProbability(0.0);
double sample = dist.sample(); // 1.0120001
double samples[] = dist.sample(3); // {.0102, -0.009, 0.011}
```

MULTIVARIATE NORMAL

The normal distribution can be generalized to higher dimensions as the *multivariate normal* (a.k.a. *multinormal*) distribution. The variate $\mathbf{x}$ and the mean $\boldsymbol{\mu}$ are vectors, while the covariance matrix $\boldsymbol{\Sigma}$ contains the variances on the diagonal and the covariances as i,j pairs. In general, the multivariate normal has a squashed ball shape and is symmetric about the mean. This distribution is perfectly

round (or spherical) for unit normals when the covariances are 0 and variances are equivalent. An example of the distribution of random points is shown in Figure 3-7.

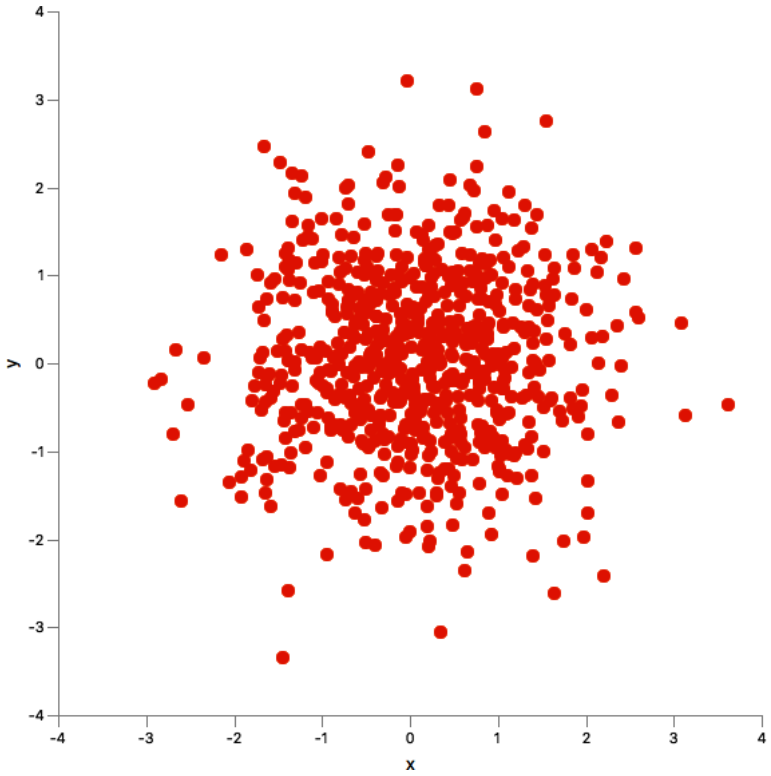

Figure 3-7. Random points generated from 2D multinormal distribution

The probability distribution function of a p dimensional multinormal distribution takes the form

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

Note that if the covariance matrix has a determinant equal to zero such that $|\Sigma| = 0$, then $f(\mathbf{x})$ blows up to infinity. Also note that when $|\Sigma| = 0$, it is impossible to calculate the required inverse of the covariance Σ^{-1}. In this case, the matrix is termed *singular*. Apache Commons Math will throw the following exception if this is the case:

```
org.apache.commons.math3.linear.SingularMatrixException: matrix is singular
```

What causes a covariance matrix to become singular? This is a symptom of co-linearity, in which two (or more) variates of the underlying data are identical or linear combinations of each other. In other words, if we have three dimensions and the covariance matrix is singular, it may mean that the distribution of data could be better described in two or even one dimension.

There is no analytical expression for the CDF. It can be attained via numerical integration. However, Apache Commons Math supports only univariate numerical integration.

The multivariate normal takes means and covariance as arrays of doubles, although you can still use `RealVector`, `RealMatrix`, or `Covariance` instances with their `getData()` methods applied:

```
double[] means = {0.0, 0.0, 0.0};
double[][] covariances = {{1.0, 0.0, 0.0},{0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}};
```

```
MultivariateNormalDistribution dist =
  new MultivariateNormalDistribution(means, covariances);

/* probability at point x = {0.0, 0.0, 0.0} */
double probability = dist.density(x); // 0.1
double[] mn = dist.getMeans();
double[] sd = dist.getStandardDeviations();
/* returns a RealMatrix but can be converted to doubles */
double[][] covar = dist.getCovariances().getData();
double[] sample = dist.sample();
double[][] samples = dist.sample(3);
```

Note the special case in which the covariance is a diagonal matrix. This occurs when the variables are completely independent. The determinant of $\mathbf{\Sigma}$ is just the product of its diagonal elements $\sigma_{i,i}$. The inverse of a diagonal matrix is yet another diagonal matrix with each term expressed as $1 / \sigma_{i,i}$. The PDF then reduces to the product of univariate normals:

$$f(\mathbf{x}) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}\right)$$

As in the case of the unit normal, a unit multivariate normal has a mean vector of 0s and a covariance matrix equal to the identity matrix, a diagonal matrix of 1s.

LOG NORMAL

The *log normal distribution* is related to the normal distribution when the variate *x* is distributed logarithmically—that is, *ln(x)* is normally distributed. If we substitute *ln(x)* for *x* in the normal distribution, we get the log normal distribution. There are some subtle differences. Because the logarithm is defined only for positive *x*, this distribution has support on the interval $x \in [0, \infty]$, where $x > 0$. The distribution is asymmetric with a peak near the smaller values of *x* and a long tail stretching, infinitely, to higher values of *x*, as shown in Figure 3-8.

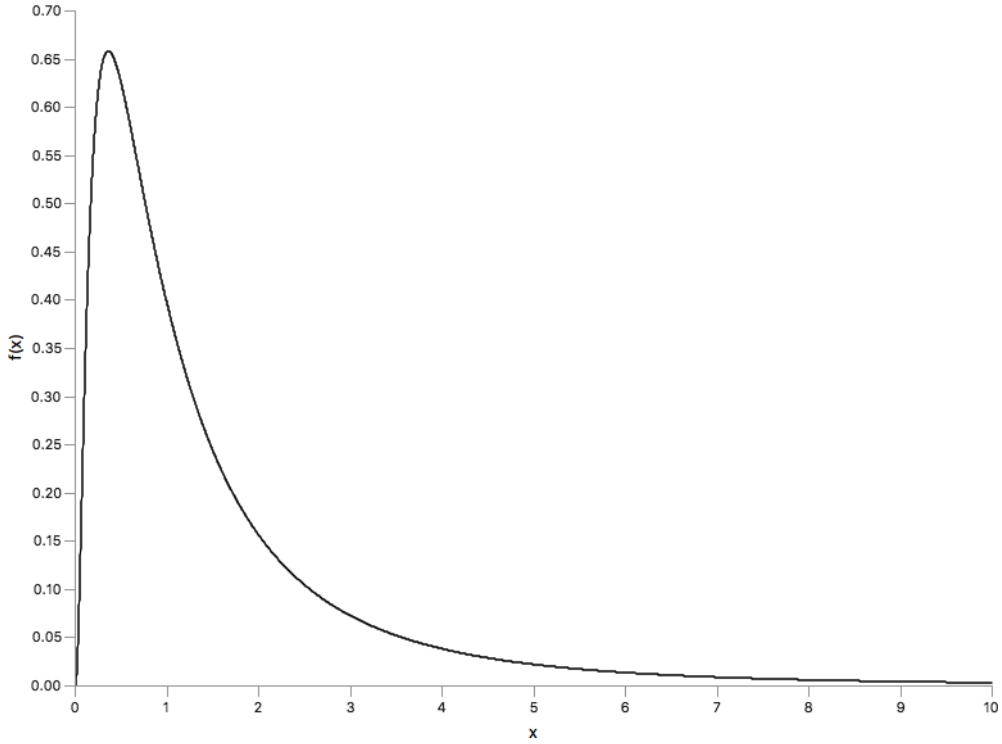

Figure 3-8. Log normal PDF with parameters $m = 0$ *and* $s = 1$

The location (scale) parameter *m* and the shape parameter *s* rise to the PDF:

$$f(x) = \frac{1}{\sqrt{2\pi}xs} \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right)$$

Here, *m* and *s* are the respective mean and standard deviation of the logarithmically distributed variate *X*. The CDF looks like Figure 3-9.

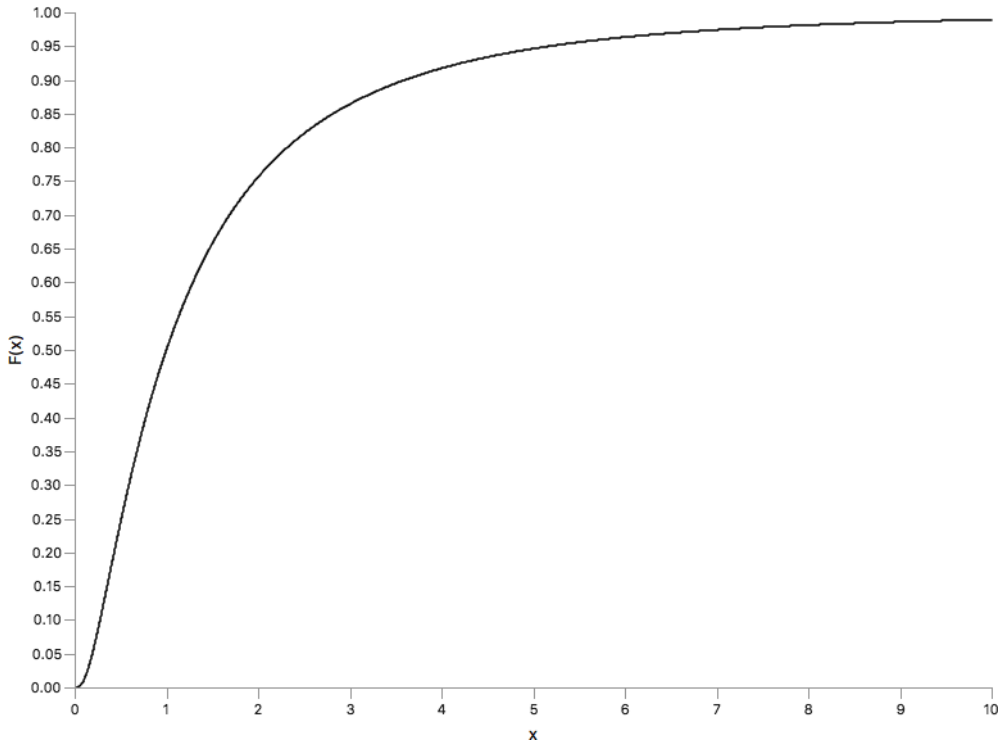

Figure 3-9. Log normal CDF with parameters $m = 0$ *and* $s = 1$

This has the following form:

$$F(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{\ln x - m}{s\sqrt{2}}\right)\right]$$

Unlike the normal distribution, the *m* is neither the mean (average value) nor the mode (most likely value or the peak) of the distribution. This is because of the larger number of values stretching off to positive infinity. The mean and variance of *X* are calculated from the following:

$$\mu = \exp\left(m + s^2/2\right)$$

$$\sigma^2 = \left(\exp\left(s^2\right) - 1\right)\exp\left(2m + s^2\right)$$

We can invoke a log normal distribution with this:

```
/* initialize with default m=0 and s=1 */
NormalDistribution dist = new NormalDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // Infinity
double scale = dist.getScale(); // 0.0
double shape = dist.getShape(): // 1.0
double mean = dist.getNumericalMean(); // 1.649
double variance = dist.getNumericalVariance(); // 4.671
double density = dist.density(1.0); // 0.3989
```

```
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.5
double sample = dist.sample(); // 0.428
double[] samples = dist.sample(3); // {0.109, 5.284, 2.032}
```

Where do we see the log normal distribution? The distribution of ages in a human population, and (sometimes) in particle size distribution. Note that the log normal distribution arises from a multiplicative effect of many independent distributions.

EMPIRICAL

In some cases you have data, but do not know the distribution that the data came from. You can still approximate a distribution with your data and even calculate probability density, cumulative probability, and random numbers! The first step in working with an empirical distribution is to collect the data into bins of equal size spanning the range of the dataset. The class `EmpiricalDistribution` can input an array of doubles or can load a file locally or from a URL. In those cases, data must be one entry per line:

```
/* get 2500 random numbers from a standard normal distribution */
NormalDistribution nd = new NormalDistribution();
double[] data = nd.sample(2500);

// default constructor assigns bins = 1000
// better to try numPoints / 10
EmpiricalDistribution dist = new EmpiricalDistribution(25);
dist.load(data); // can also load from file or URL !!!
double lowerBound = dist.getSupportLowerBound(); // 0.5
double upperBound = dist.getSupportUpperBound(); // 10.1
double mean = dist.getNumericalMean(); // 5.48
double variance = dist.getNumericalVariance(); // 15.032
double density = dist.density(1.0); // 0.357
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.153
double sample = dist.sample(); // e.g., 1.396
double[] samples = dist.sample(3); // e.g., [10.098, 0.7934, 9.981]
```

We can plot the data from an empirical distribution as a type of bar chart called a histogram, shown in Figure 3-10.

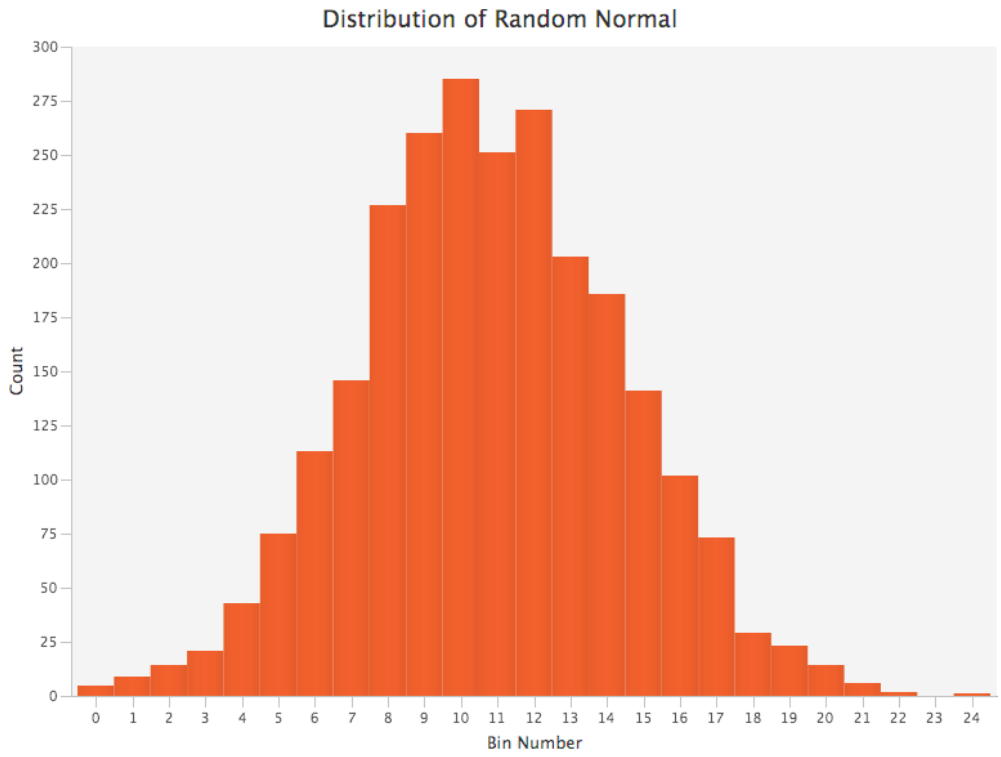

Figure 3-10. Histogram of random normal with parameters $\mu = 0$ *and* $\sigma = 1$

The code for a histogram uses the `BarChart` plot from Chapter 1, except that we add the data directly from the `EmpiricalDistribution` instance that contains a `List` of all the `SummaryStatistics` of each bin:

```
/* for an existing EmpiricalDistribution with loaded data */
List<SummaryStatistics> ss = dist.getBinStats();
int binNum = 0;
for (SummaryStatistics s : ss) {
    /* adding bin counts to the XYChart.Series instance */
    series.getData().add(new Data(Integer.toString(binNum++), s.getN()));
}
// render histogram with JavaFX BarChart
```

Discrete Distributions

There are several discrete random-number distributions. These support only integers as values, which are designated by k.

BERNOULLI

The *Bernoulli distribution* is the most basic and perhaps most familiar distribution because it is essentially a coin flip. In a "heads we win, tails we lose" situation, the coin has two possible states: tails ($k = 0$) and heads ($k = 1$), where $k = 1$ is designated to have a probability of success equal to p. If the coin is perfect, then $p = 1/2$; it is equally likely to get heads as it is tails. But what if the coin is "unfair," indicating $p \neq 1/2$? The probability mass function (PMF) can then be represented as follows:

$$f(k) = \begin{cases} 1 - p & \text{for } k = 0 \\ p & \text{for } k = 1 \end{cases}$$

The cumulative distribution function is shown here:

$$F(k) = \begin{cases} 0 & \text{for } k < 0 \\ (1-p) & \text{for } 0 \leq k < 1 \\ 1 & \text{for } k \geq 1 \end{cases}$$

The mean and variance are calculated with the following:

$$\mu = p$$

$$\sigma^2 = p(1-p)$$

Note that the Bernoulli distribution is related to the binomial distribution, where the number of trials equals $n = 1$. The Bernoulli distribution is implemented with the class `BinomialDistribution(1, p)` setting $n = 1$:

```
BinomialDistribution dist = new BinomialDistribution(1, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 1
int numTrials = dist.getNumberOfTrials(); // 1
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
double mean = dist.getNumericalMean(); // 0.5
double variance = dist.getNumericalVariance(); // 0.25
// k = 1
double probability = dist.probability(1); // 0.5
double cumulativeProbability = dist.cumulativeProbability(1); // 1.0
int sample = dist.sample(); // e.g., 1
int[] samples = dist.sample(3); // e.g., [1, 0, 1]
```

BINOMIAL

If we perform multiple Bernoulli trials, we arrive at the binomial distribution. For n Bernoulli trials, each with probability of success p, the distribution of successes k has the form in Figure 3-11.

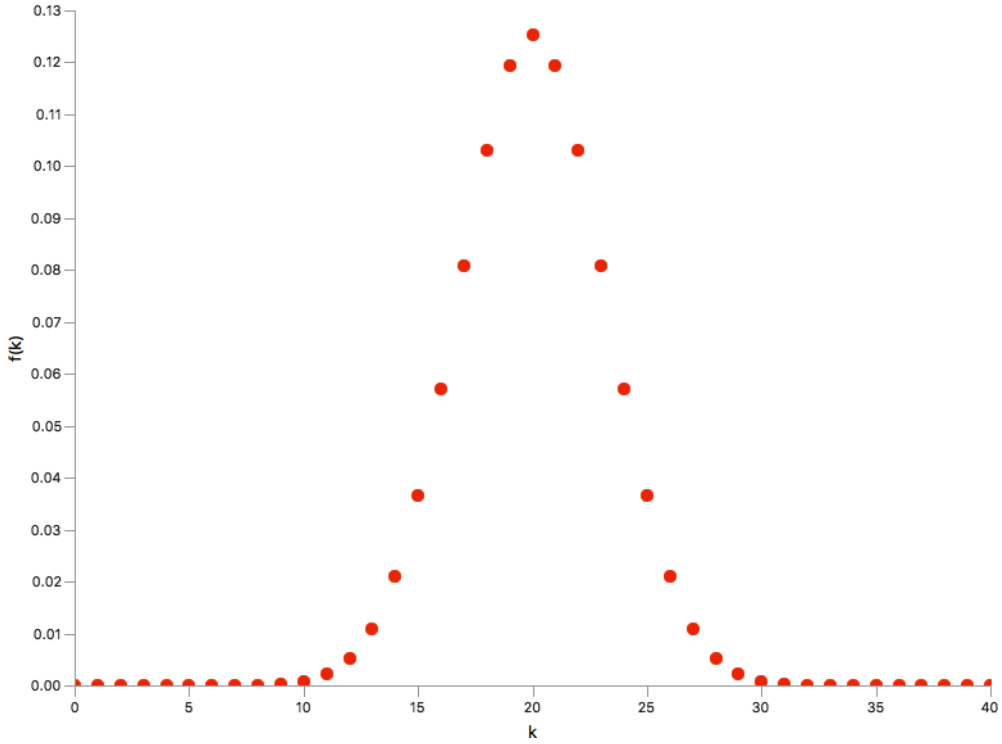

Figure 3-11. Binomial PMF with parameters $n = 40$ *and* $p = 0.5$

The probability mass function is expressed with the following:

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

The CDF looks like Figure 3-12.

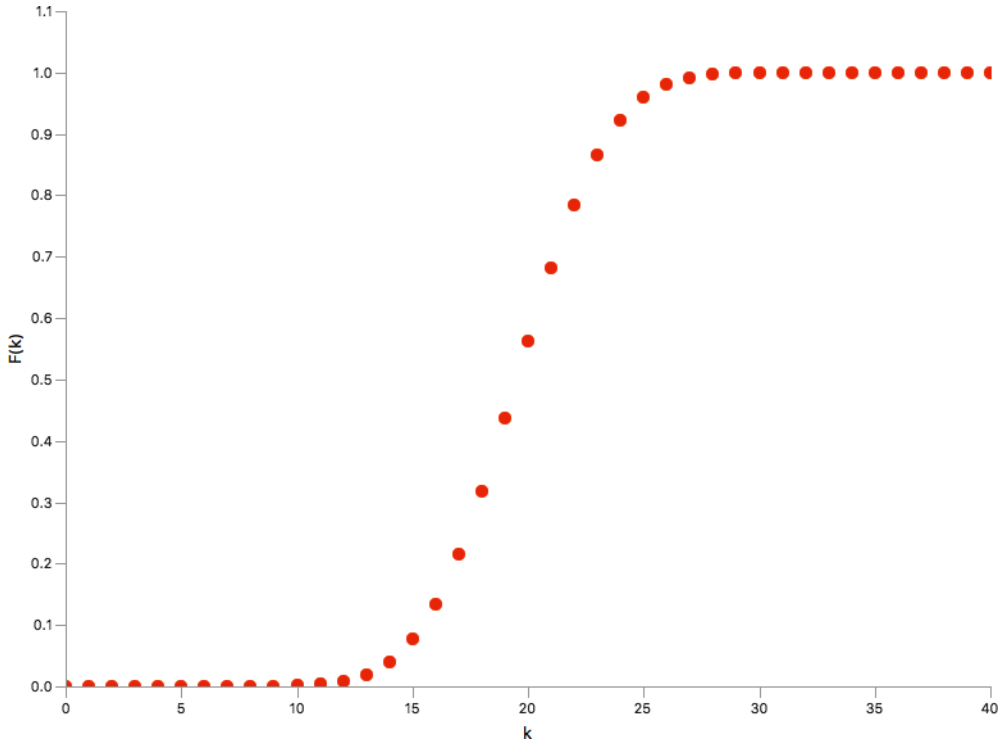

Figure 3-12. Binomial CDF with parameters $n = 40$ *and* $p = 0.5$

The CDF takes the form

$$F(k) = I_{1-p}(n - k, 1 + k)$$

I_{1-p} is the regularized incomplete beta function. The mean and variance are computed via the following:

$$\mu = np$$

$$\sigma^2 = np(1 - p)$$

In Java, `BinomialDistribution` has two required arguments in the constructor: *n*, the number of trials; and *p*, the probability of success for one trial:

```
BinomialDistribution dist = new BinomialDistribution(10, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 10
int numTrials = dist.getNumberOfTrials(); // 10
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
double mean = dist.getNumericalMean(); // 5.0
double variance = dist.getNumericalVariance(); // 2.5
// k = 1
double probability = dist.probability(1); // 0.00977
double cumulativeProbability = dist.cumulativeProbability(1); // 0.0107
int sample = dist.sample(); // e.g., 9
int[] samples = dist.sample(3); // e.g., [4, 5, 4]
```

POISSON

The *Poisson distribution* is often used to describe discrete, independent events that occur rarely. The number of events are the integers $k \geq 0$ that occur over some interval with a constant rate $\lambda > 0$ gives rise to the PMF in Figure 3-13.

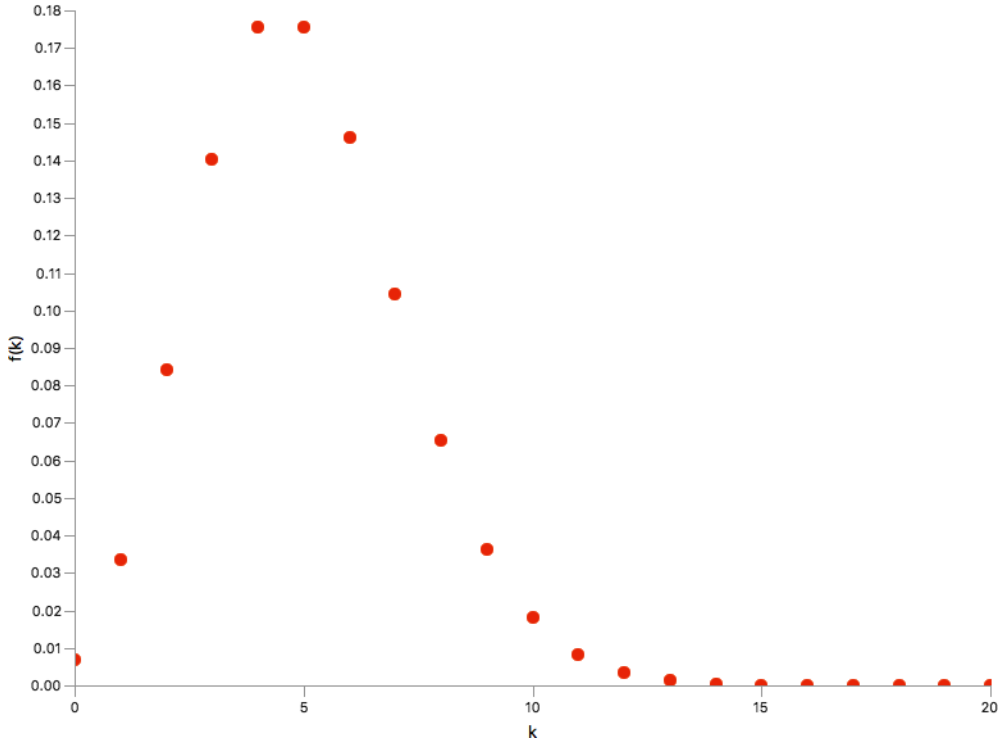

Figure 3-13. Poisson PMF with parameters $\lambda = 5$

The form of the PMF is

$$f(k) = \frac{\lambda^k \exp(-\lambda)}{k!}$$

Figure 3-14 shows the CDF.

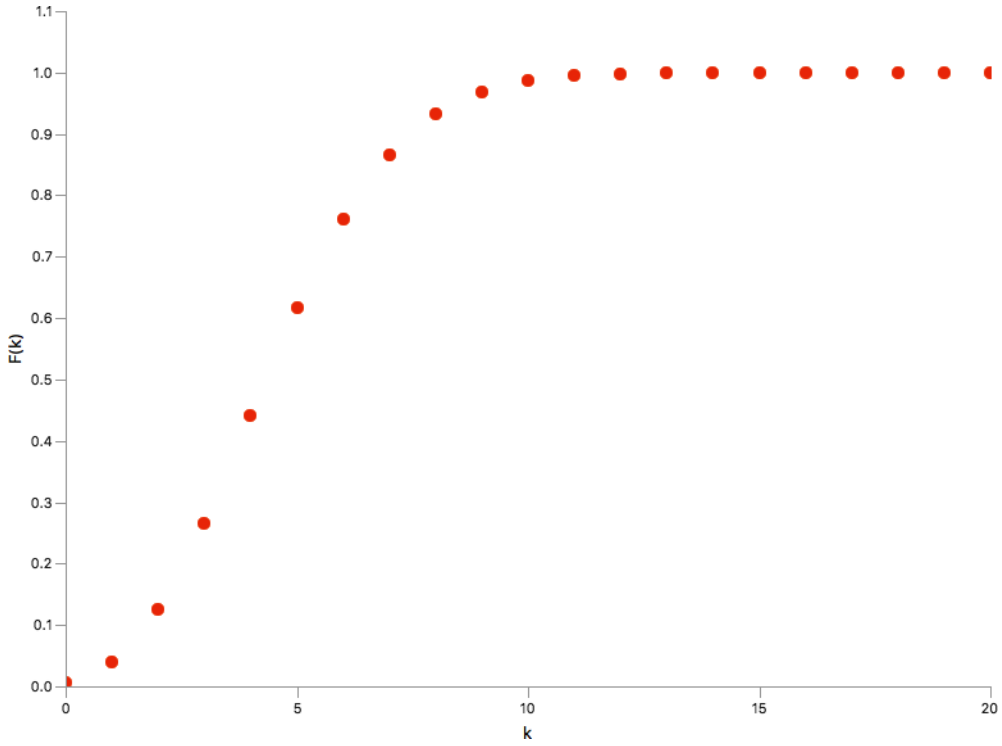

Figure 3-14. Poisson CDF with parameters $\lambda = 5$

The CDF is expressed via

$$F(k) = \frac{\Gamma(\lfloor k+1 \rfloor, \lambda)}{\lfloor k \rfloor !}$$

The mean and variance are both equivalent to the rate parameter λ as

$$\mu = \lambda$$

$$\sigma^2 = \lambda$$

The Poisson is implemented with the parameter λ in the constructor and has an upper bound at `Integer.MAX` k = $2^{32} - 1$ = 2147483647:

```
PoissonDistribution dist = new PoissonDistribution(3.0);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 2147483647
double mean = dist.getNumericalMean(); // 3.0
double variance = dist.getNumericalVariance(); // 3.0
// k = 1
double probability = dist.probability(1); // 0.1494
double cumulativeProbability = dist.cumulativeProbability(1); // 0.1991
int sample = dist.sample(); // e.g., 1
int[] samples = dist.sample(3); // e.g., [2, 4, 1]
```

Characterizing Datasets

Once we have a dataset, the first thing we should do is understand the character of the data. We should know the numerical limits, whether any outliers exist, and whether the data has a shape like one of the known distribution functions. Even if we have no idea what the underlying distribution is, we can still check whether two separate datasets come from the same (unknown) distribution. We can also check how related (or unrelated) each pair of variates is via covariance/correlation. If our variate *x* comes with a response *y*, we can check a linear regression to see whether the most basic of relationships exists between *x* and *y*. Most of the classes in this section are best for small, static datasets that can fit entirely into memory, because most of the methods in these classes rely on stored data. In the following section, we deal with data that is so large (or inconvenient) that it cannot fit in memory.

Calculating Moments

In the previous section where we discussed statistical moments, formulas were presented for calculating moments and their various statistical outcomes when we know the probability distribution function *f(x)*. When dealing with real data, we usually do not know the form of *f(x)* and so we must estimate the moments numerically. The moment calculations have another critical feature: robustness. Estimating statistical quantities can result in numerical error as extreme values are encountered. Using the method of updating moments, we can avoid numerical imprecision.

SAMPLE MOMENTS

In the case of real data, where the true statistical distribution function may not be known, we can estimate the central moments:

$$m_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^k$$

Here, the estimated mean $m_1 = \bar{x}$ is calculated as follows:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n}x_i$$

UPDATING MOMENTS

Without the factor 1/n for the estimate of the central moment, the form is as follows:

$$M_k = \sum_{i=1}^{n}(x_i - \bar{x})^k$$

In this particular form, the unnormalized moments can be split into parts via straightforward calculations. This gives us a great advantage that the unnormalized moments can be calculated in parts, perhaps in different processes or even on different machines entirely, and we can glue back together the pieces later. Another advantage is that this formulation is less sensitive to extreme values. The combined unnormalized central moment for the two chunks is shown here:

$$M_k = M_{k,1} + M_{k,2} + \sum_{j=1}^{k-2} \binom{j}{k} \left[\left(-\frac{n_2}{n} \right)^j M_{k-j,1} + \left(\frac{n_1}{n} \right)^j M_{k-j,2} \right] \delta_{2,1}^j + \left(\frac{n_1 n_2}{n} \delta_{2,1} \right)^k \left[\frac{1}{n_2^{k-1}} - \left(\frac{-1}{n_1} \right)^{k-1} \right]$$

$\delta_{2,1} = \bar{x}_2 - \bar{x}_1$ is the difference between the means of the two data chunks. Of course, you will also need a method for merging the means, because this formula is applicable only for $k > 1$. Given any two data chunks with known means and counts, the total count is $n = n_1 + n_2$, and the mean is robustly calculated as follows:

$$\bar{x} = \bar{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

If one of the data chunks has only one point x, then the formulation for combining the unnormalized central moments is simplified:

$$M_k = M_{k,1} + \sum_{j=1}^{k-2} \binom{j}{k} M_{k-j,1} \left(\frac{-\delta}{n} \right)^j + \left(\frac{n-1}{n} \delta \right)^k \left[1 - \left(\frac{-1}{n-1} \right)^{k-1} \right]$$

Here, $\delta = x - \bar{x}_1$ is the difference between the added value x and the mean value of the existing data chunk.

In the next section, we will see how these formulas are used to robustly calculate important properties to statistics. They become essential in distributed computing applications when we wish to break statistical calculations into many parts. Another useful application is storeless calculations in which, instead of performing calculations on whole arrays of data, we keep track of moments as we go, and update them incrementally.

Descriptive Statistics

We instantiate `DescriptiveStatistics` with no argument to add values later, or start it out with an array of doubles (and then can still add values later). Although you can use `StatUtils` static methods, this is not Java-like, and although there is nothing wrong with it, it's probably wiser to use `DescriptiveStatistics` instead. Some of the formulas in this section are not stable, and a more robust method is described in the next section. Indeed, some of those methods are used in the descriptive stats methods as well. In Table 3-1, we display the data from Anscombe's quartet for further analysis in this chapter.

Table 3-1. Anscombe's quartet data

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

We can then create `DescriptiveStatistics` classes with these datasets:

```
/* stats for Anscombe's y1 */
DescriptiveStatistics descriptiveStatistics = new DescriptiveStatistics();
descriptiveStatistics.addValue(8.04);
descriptiveStatistics.addValue(6.95);
//keep adding y1 values
```

However, you may already have all the data that you need, or maybe it's just an initial set that you will add to later. You can always add more values with `ds.addValue(double value)` if needed. At this point, you can display a report of stats with either a call to the method, or by printing the class directly:

```
System.out.println(descriptiveStatistics);
```

This produces the following result:

```
DescriptiveStatistics:
n: 11
min: 4.26
max: 10.84
mean: 7.500909090909091
std dev: 2.031568135925815
median: 7.58
skewness: -0.06503554811157437
kurtosis: -0.5348977343727395
```

All of these quantities (and more) are available via their specific getters, as explained next.

COUNT

The simplest statistic is the count of the number of points in the dataset:

```
long count = descriptiveStatistics.getN();
```

SUM

We can also access the sum of all values:

```
double sum = descriptiveStatistics.getSum();
```

MIN

To retrieve the minimum value dataset, we use this:

```
double min = descriptiveStatistics.getMin();
```

MAX

To retrieve the maximum value in the dataset, we use this:

```
double max = descriptiveStatistics.getMax();
```

MEAN

The average value of the sample or mean is calculated directly:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

However, this calculation is sensitive to extreme values, and given $\delta = x - \overline{x}$, the mean can be updated for each added value *x*:

$$\overline{x} = \overline{x}_1 + \frac{x - \overline{x}_1}{n}$$

Commons Math uses the update formula for mean calculation when the `getMean()` method is called:

```
double mean = descriptiveStatistics.getMean();
```

MEDIAN

The middle value of a sorted (ascending) dataset is the *median*. The advantage is that it minimizes the problem of extreme values. While there is no direct calculation of the median in Apache Commons Math, it is easy to calculate by taking the average of the two middle members if the array length is even; and otherwise, just return the middle member of the array:

```
// sort the stored values
double[] sorted = descriptiveStatistics.getSortedValues();
int n = sorted.length;
double median = (n % 2 == 0) ? (sorted[n/2-1]+sorted[n/2])/2.0 : sorted[n/2];
```

MODE

The mode is the most likely value. The concept of *mode* does not make sense if the values are doubles, because there is probably only one of each. Obviously, there will be exceptions (e.g., when many zeros occur) or if the dataset is large and the numerical precision is small (e.g., two decimal places). The mode has two use cases then: if the variate being considered is discrete (integer), then the mode can be useful, as in dataset four of Anscombe's quartet. Otherwise, if you have created bins from empirical distribution, the mode is the max bin. However, you should consider the possibility that your data is noisy and the bin counts may erroneously identify an outlier as a mode.The `StatUtils` class contains several static methods useful for statistics. Here we utilize its mode method:

```
// if there is only one max, it's stored in mode[0]
// if there is more than one value that has equally high counts
// then values are stored in mode in ascending order
double[] mode = StatUtils.mode(x4);
//mode[0] = 8.0
double[] test = {1.0, 2.0, 2.0, 3.0, 3.0, 4.0}
//mode[0] = 2.0
//mode[1] = 3.0
```

VARIANCE

The *variance* is a measure of how much the data is spread out and is always a positive, real number greater than or equal to zero. If all values of x are equal, the variance will be zero. Conversely, a larger spread of numbers will correspond to a larger variance. The variance of a known population of data points is equivalent to the second central moment about the mean and is expressed as follows:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

However, most of the time we do not have all of the data—we are only sampling from a larger (possibly unknown) dataset and so a correction for this bias is needed:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

This form is known as the *sample variance* and is most often the variance we will use. You may note that the sample variance can be expressed in terms of the second-order, unnormalized moment:

$$s^2 = \frac{1}{n-1}M_2$$

As with the mean calculation, the Commons Math variance is calculated using the update formula for the second unnormalized moment for a new data point x, an existing mean $\overline{x}_1$ and the newly updated mean $\overline{x}$:

$$M_2 = M_{2,1} + (x - \overline{x}_1)\delta$$

Here, $\delta = x - \bar{x}_1$.

Most of the time when the variance is required, we are asking for the bias-corrected, sample variance because our data is usually a sample from some larger, possibly unknown, set of data:

```
double variance = descriptiveStatistics.getVariance()
```

However, it is also straightforward to retrieve the population variance if it is required:

```
double populationVariance = descriptiveStatistics.getPopulationVariance();
```

STANDARD DEVIATION

The variance is difficult to visualize because it is on the order of x^2 and usually a large number compared to the mean. By taking the square root of the variance, we define this as the standard deviation s. This has the advantage of being in the same units of the variates and the mean. It is therefore helpful to use things like $\mu +- \sigma$, which can indicate how much the data deviates from the mean. The standard deviation can be explicitly calculated with this:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

However, in practice, we use the update formulas to calculate the sample variance, returning the standard deviation as the square root of the sample variance when required:

```
double standardDeviation = descriptiveStatistics.getStandardDeviation();
```

Should you require the population standard deviation, you can calculate this directly by taking the square root of the population variance.

ERROR ON THE MEAN

While it is often assumed that the standard deviation is the error on the mean, this is not true. The standard deviation describes how the data is spread around the average. To calculate the accuracy of the mean itself, s_x, we use the standard deviation:

$$s_x = \frac{s}{\sqrt{n}}$$

And we use some simple Java:

```
double meanErr = descriptiveStatistics.getStandardDeviation() /
                        Math.sqrt(descriptiveStatistics.getN());
```

SKEWNESS

The *skewness* measures how asymmetrically distributed the data is and can be either a positive or negative real number. A positive skew signifies that most of the values leans toward the origin ($x = 0$), while a negative skew implies the values are distributed away (to right). A skewness of 0 indicates that the data is perfectly distributed on either side of the data distribution''s peak value. The skewness can be calculated explicitly as follows:

$$\omega = \frac{1}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s} \right)^3$$

However, a more robust calculation of the skewness is achieved by updating the third central moment:

$$M_3 = M_{3,1} - 3M_{2,1}\frac{\delta}{n} + (n-1)(n-2)\frac{\delta^3}{n^2}$$

Then you calculate the skewness when it is called for:

$$\omega = \frac{1}{(n-1)(n-2)} \frac{M_3}{s^3}$$

The Commons Math implementation iterates over the stored dataset, incrementally updating M3 and then performing the bias correction, and returns the skewness:

```
double skewness = descriptiveStatistics.getSkewness();
```

KURTOSIS

The kurtosis is a measure of how "tailed" a distribution of data is. The sample kurtosis estimate is related to the fourth central moment about the mean and is calculated as shown here:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s} \right)^4$$

This can be simplified as follows:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \frac{M_4}{s^4}$$

A kurtosis at or near 0 signifies that the data distribution is extremely narrow. As the kurtosis increases, extreme values coincide with long tails. However, we often want to express the kurtosis as related to the normal distribution (which has a kurtosis = 3). We can then subtract this part and call the new quantity the *excess kurtosis*, although in practice most people just refer to the excess kurtosis as the kurtosis. In this definition, $\kappa = 0$ implies that the data has the same peak and tail shape as a normal distribution. A kurtosis higher than 3 is called *leptokurtic* and has wider tails than a normal distribution. When κ is less than 3, the distribution is said to be

platykurtic and the distribution has few values in the tail (less than the normal distribution). The excess kurtosis calculation is as follows:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)}\frac{M_4}{s^4} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

As in the case for the variance and skewness, the kurtosis is calculated by updating the fourth unnormalized central moment. For each point added to the calculation, M_4 can be updated with the following equation as long as the number of points $n >= 4$. At any point, the skew can be computed from the current value of M_4:

$$M_4 = M_{4,1} - 4\,M_{3,1}\frac{\delta}{n} + 6\,M_{2,1}\left(\frac{\delta}{n}\right)^2 + (n-1)(n^2-3n+3)\frac{\delta^4}{n^3}$$

The default calculation returned by `getKurtosis` is the excess kurtosis definition. This can be checked by implementing the class `org.apache.commons.math3.stat.descriptive.moment.Kurtosis`, which is called by `getKurtosis()`:

```
double kurtosis = descriptiveStatistics.getKurtosis();
```

Multivariate Statistics

So far, we have addressed the situation where we are concerned with one variate at a time. The class `DescriptiveStatistics` takes only one-dimensional data. However, we usually have several dimensions, and it is not uncommon to have hundreds of dimensions. There are two options: the first is to use the `MultivariateStatisticalSummary` class, which is described in the next section. If you can live without skewness and kurtosis, that is your best option. If you do require the full set of statistical quantities, your best bet is to implement a `Collection` instance of `DescriptiveStatistics` objects. First consider what you want to keep track of. For example, in the case of Anscombe's quartet, we can collect univariate statistics with the following:

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);
...
List<DescriptiveStatistics> dsList = new ArrayList<>();
dsList.add(descriptiveStatisticsX1);
dsList.add(descriptiveStatisticsX2);
...
```

You can then iterate through the `List`, calling statistical quantities or even the raw data:

```
for(DescriptiveStatistics ds : dsList) {

    double[] data = ds.getValues();
    // do something with data or

    double kurtosis = ds.getKurtosis();
    // do something with kurtosis
}
```

If the dataset is more complex and you know you will need to call specific columns of data in your ensuing analysis, use `Map` instead:

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsY1 = new DescriptiveStatistics(y1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);

Map<String, DescriptiveStatistics> dsMap = new HashMap<>();
dsMap.put("x1", descriptiveStatisticsX1);
dsMap.put("y1", descriptiveStatisticsY1);
dsMap.put("x2", descriptiveStatisticsX2);
```

Of course, now it is trivial to call a specific quantity or dataset by its key:

```
double x1Skewness = dsMap.get("x1").getSkewness();
double[] x1Values = dsMap.get("x1").getValues();
```

This will become cumbersome for a large number of dimensions, but you can simplify this process if the data was already stored in multidimensional arrays (or matrices) where you loop over the column index. Also, you may have already stored your data in the `List` or `Map` of the data container classes, so automating the process of building a multivariate `Collection` of `DescriptiveStatistics` objects will be straightforward. This is particularly efficient if you already have a data dictionary (a list of variable names and their properties) from which to iterate over. If you have high-dimensional numeric data of one type and it already exists in a matrix of double array form, it might be easier to use the `MultivariateSummaryStatistics` class in the next section.

Covariance and Correlation

The covariance and correlation matrices are symmetric, square $m \times m$ matrices with dimension m equal to the number columns in the original dataset.

COVARIANCE

The covariance is the two-dimensional equivalent of the variance. It measures how two variates, in combination, differ from their means. It is calculated as shown here:

$$\sigma_{i,j}^2 = \frac{1}{n-1}\sum_{k=1}^{n}(x_{k,i} - \bar{x}_i)(x_{k,j} - \bar{x}_j)$$

Just as in the case of the one-dimensional, statistical sample moments, we note that the quantity

$$C_2 = \sum_{k=1}^{n}(x_{k,i} - \bar{x}_i)(x_{k,j} - \bar{x}_j)$$

can be expressed as an incremental update to the co-moment of the pair of variables x_i and x_j, given a known means and counts for the existing dimensions:

$$C_2 = C_{2,1} + \frac{n_1 n_2}{n}(x_i - \bar{x}_i)(x_j - \bar{x}_j)$$

Then the covariance can be calculated at any point:

$$\sigma_{i,j}^2 = \frac{1}{n-1}C_2$$

The code to calculate the covariance is shown here:

```
Covariance cov = new Covariance();

/* examples using Anscombe's Quartet data */
double cov1 = cov.covariance(x1, y1); // 5.501
double cov2 = cov.covariance(x2, y2); // 5.499
double cov3 = cov.covariance(x3, y3); // 5.497
double cov4 = cov.covariance(x4, y5); // 5.499
```

If you already have the data in a 2D array of doubles or a `RealMatrix` instance, you can pass them directly to the constructor like this:

```
// double[][] myData or RealMatrix myData
Covariance covObj = new Covariance(myData);
// cov contains covariances and can be accessed
// from RealMatrix.get(i,j) retrieve elements
RealMatrix cov = covObj.getCovarianceMatrix();
```

Note that the diagonal of the covariance matrix $\sigma_{i,j}^2$ is just the variance of column i and therefore the square root of the diagonal of the covariance matrix is the standard deviation of each dimension of data. Because the population mean is usually not known, we use the biased covariance with the sample mean. If we did know the population mean, the unbiased correction factor $1/n$ would be used to calculate the unbiased covariance:

$$\sigma_{i,j}^2 = \frac{1}{n} C_2$$

PEARSON'S CORRELATION

The Pearson correlation coefficient is related to covariance via the following, and is a measure of how likely two variates are to vary together:

$$\rho_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j}$$

The correlation coefficient takes a value between –1 and 1, where 1 indicates that the two variates are nearly identical. –1 indicates that they are opposites. In Java, there are once again two options, but using the default constructor, we get this:

```
PearsonsCorrelation corr = new PearsonsCorrelation();

/* examples using Anscombe's Quartet data */
double corr1 = corr.correlation(x1, y1)); // 0.816
double corr2 = corr.correlation(x2, y2)); // 0.816
double corr3 = corr.correlation(x3, y3)); // 0.816
double corr4 = corr.correlation(x4, y4)); // 0.816
```

However, if we already have data, or a `Covariance` instance, we can use the following:

```
// existing Covariance cov
PearsonsCorrelation corrObj = new PearsonsCorrelation(cov);
// double[][] myData or RealMatrix myData
PearsonsCorrelation corrObj = new PearsonsCorrelation(myData);
// from RealMatrix.get(i,j) retrieve elements
RealMatrix corr = corrObj.getCorrelationMatrix();
```

> **WARNING**
>
> Correlation is not causation! One of the dangers in statistics is the interpretation of correlation. When two variates have a high correlation, we tend to assume that this implies that one variable is responsible for causing the other. This is not the case. In fact, all you can assume is that you can reject the idea that the variates have nothing to do with each other. You should view correlation as a fortunate coincidence, not a foundational basis for the underlying behavior of the system being studied.

Regression

Often we want to find the relationship between our variates **X** and their responses y. We are trying to find a set of values for β such that $y = \mathbf{X}\hat{\beta}$. In the end, we want three things: the parameters, their errors, and a statistic of how good the fit is, R^2.

SIMPLE REGRESSION

If **X** has only one dimension, the problem is the familiar equation of a line $y = \hat{\alpha} + \hat{\beta}x$, and the problem can be classified as simple regression. By calculating σ_x^2, the variance of x and $\sigma_{x,y}^2$, and the covariance between x and y, we can estimate the slope:

$$\hat{\beta} = \frac{\sigma_{x,y}^2}{\sigma_x^2}$$

Then, using the slope and the means of x and y, we can estimate the intercept:

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

The code in Java uses the `SimpleRegression` class:

```
SimpleRegression rg = new SimpleRegression();

/* x-y pairs of Anscombe's x1 and y1 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/* get regression results */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

We can then interpret these results as $y = 3.0 + 0.5x$, or more specifically as $y = (3.0 \pm 1.12) + (0.5 \pm 0.12)x$. How much can we trust this model? With $R^2 = 0.67$, it's a fairly decent fit, but the closer it is to the ideal of $R^2 = 1.0$, the better. Note that if we perform the same regression on the other

three datasets from Anscombe's quartet, we get identical parameters, errors, and R^2. This is a profound, albeit perplexing, result. Clearly, the four datasets look different, but their linear fits (the superposed blue lines) are identical. Although linear regression is a powerful yet simple method for understanding our data as in case 1, in case 2 linear regression in *x* is probably the wrong tool to use here. In case 3, linear regression is probably the right tool, but we could sensor (remove) the data point that appears to be an outlier. In case 4, a regression model is most likely not appropriate at all. This does demonstrate how easy it is to fool ourselves that a model is correct if we look at only a few parameters after blindly throwing data into an analysis method.

MULTIPLE REGRESSION

There are many ways to solve this problem, but the most common and probably most useful is the ordinary least squares (OLS) method. The solution is expressed in terms of linear algebra:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

The `OLSMultipleLinearRegression` class in Apache Commons Math is just a convenient wrapper around a QR decomposition. This implementation also provides additional functions beyond the QR decomposition that you will find useful. In particular, the variance-covariance matrix of β is as follows, where the matrix *R* is from the QR decomposition:

$$\sigma^2_{\hat{\beta}} = (X^T X)^{-1} = (R^T R)^{-1}$$

In this case, *R* must be truncated to the dimension of beta. Given the fit residuals $\hat{\epsilon} = y - X\hat{\beta}$, we can calculate the variance of the errors $s^2_{err} = \hat{\epsilon}^T \hat{\epsilon} / (n - p)$ where n and p are the respective number of rows and columns of X. The square root of the diagonal values of $\sigma^2_{\hat{\beta}}$ times the constant s_{err} gives us the estimate of errors on the fit parameters:

$$\delta\hat{\beta}_i = s_{err} \sqrt{\sigma^2_{\hat{\beta}_{i,i}}}$$

The Apache Commons Math implementation of ordinary least squares regression utilizes the QR decomposition covered in linear algebra. The methods in the example code are convenient wrappers around several standard matrix operations. Note that the default is to include an intercept term, and the corresponding value is the first position of the estimated parameters:

```
double[][] xNData = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] yNData = {-1, 0.2, 0.9, 2.1};
// default is to include an intercept
OLSMultipleLinearRegression mr = new OLSMultipleLinearRegression();
/* NOTE that y and x are reversed compared to other classes / methods */
mr.newSampleData(yNData, xNData);
double[] beta = mr.estimateRegressionParameters();
// [-0.7499, 1.588, -0.5555]
double[] errs = mr.estimateRegressionParametersStandardErrors();
// [0.2635, 0.6626, 0.6211]
double r2 = mr.calculateRSquared();
// 0.9945
```

Linear regression is a vast topic with many adaptations—too many to be covered here. However, it is worth noting that these methods are relevant only if the relations between *X* and *y* are actually linear. Nature is full of nonlinear relationships, and in Chapter 5 we will address more ways of exploring these.

Working with Large Datasets

When our data is so large that it is inefficient to store it in memory (or it just won't fit!), we need an alternative method of calculating statistical measures. Classes such as `DescriptiveStatistics` store all data in memory for the duration of the instantiated class. However, another way to attack this problem is to store only the unnormalized statistical moments and update them one data point at a time, discarding that data point after it has been assimilated into the calculations. Apache Commons Math has two such classes: `SummaryStatistics` and `MultivariateSummaryStatistics`.

The usefulness of this method is enhanced by the fact that we can also sum unnormalized moments in parallel. We can split data into partitions and keep track of moments in each partition as we add one value at a time. In the end, we can merge all those moments and then find the summary statistics. The Apache Commons Math class `AggregateSummaryStatistics` takes care of this. It is easy to imagine terabytes of data distributed over a large cluster in which each node is updating statistical moments. As the jobs complete, the moments can be merged in a simple calculation, and the task is complete.

In general, a dataset $\mathbf{X}$ can be partitioned into k smaller datasets: $\mathbf{X}_1, \mathbf{X}_2, \bullet\bullet\bullet \mathbf{X}_k$. Ideally, we can perform all sorts of computations on each partition $\mathbf{X}_i$ and then later merge these results to get the desired quantities for $\mathbf{X}$. For example, if we wanted to count the number of data points in $\mathbf{X}$, we could count the number of points in each subset and then later add those results together to get the total count:

$$n = n_1 + n_2 \cdots + n_k$$

This is true whether the partitions were calculated on the same machine in different threads, or on different machines entirely.

So if we calculated the number of points, and additionally, the sum of values for each subset (and kept track of them), we could later use that information to calculate the mean of $\mathbf{X}$ in a distributed way.

$$\overline{x} = \frac{\sum_{x \in \mathbf{X}_1} x + \sum_{x \in \mathbf{X}_2} x + \cdots + \sum_{x \in \mathbf{X}_k} x}{n_1 + n_2 \cdots + n_k}$$

At the simplest level, we need only computations for pairwise operations, because any number of operations can reduce that way. For example, $\mathbf{X} = (\mathbf{X}_1 + \mathbf{X}_2) + (\mathbf{X}_3 + \mathbf{X}_4)$ is a combination of three pairwise operations. There are then three general situations for pairwise algorithms: first, where we are merging two partitions, each with $n_i > 1$; the second, where one partition has $n_i > 1$ and the other partition is a singleton with $n_i = 1$; and the third where both partitions are singletons.

Accumulating Statistics

We saw in the preceding chapter how stats can be updated. Perhaps it occurred to you that we could calculate and store the (unnormalized) moments on different machines at different times, and update them at our convenience. As long as you keep track of the number of points and all relevant statistical moments, you can recall those at any time and update them with a new set of data points. While the `DescriptiveStatistics` class stored all the data and did these updates in one long chain of calculations, the `SummaryStatistics` class (and `MultivariateSummaryStatistics` class) do not store any of the data you input to them. Rather, these classes store only the relevant n, M_1, and M_2. For massive datasets, this is an efficient way to keep track of stats without incurring the huge costs of storage or processing power whenever we need a statistic such as mean or standard deviation.

```
SummaryStatistics ss = new SummaryStatistics();

/* This class is storeless, so it is optimized to take one value at a time */
ss.addValue(1.0);
ss.addValue(11.0);
ss.addValue(5.0);

/* prints a report */
System.out.println(ss);
```

As with the `DescriptiveStatistics` class, the `SummaryStatistics` class also has a `toString()` method that prints a nicely formatted report:

```
SummaryStatistics:
n: 3
min: 1.0
max: 11.0
sum: 17.0
mean: 5.666666666666667
geometric mean: 3.8029524607613916
variance: 25.333333333333332
population variance: 16.88888888888889
second moment: 50.666666666666664
sum of squares: 147.0
standard deviation: 5.033222956847166
sum of logs: 4.007333185232471
```

For multivariate statistics, the `MultivariateSummaryStatistics` class is directly analogous to its univariate counterpart. To instantiate this class, you must specify the dimension of the variates (the number of columns in the dataset) and indicate whether the input data is a sample. Typically, this option should be set to `true`, but note that if you forget it, the default is `false`, and that will have consequences. The `MultivariateSummaryStatistics` class contains methods that keep track of the covariance between every set of variates. Setting the constructor argument `isCovarianceBiasedCorrected` to `true` uses the biased correction factor for the covariance:

```
MultivariateSummaryStatistics mss = new MultivariateSummaryStatistics(3, true);

/* data could be 2d array, matrix, or class with a double array data field */
double[] x1 = {1.0, 2.0, 1.2};
double[] x2 = {11.0, 21.0, 10.2};
double[] x3 = {5.0, 7.0, 0.2};

/* This class is storeless, so it is optimized to take one value at a time */
mss.addValue(x1);
mss.addValue(x2);
mss.addValue(x3);

/* prints a report */
System.out.println(mss);
```

As in `SummaryStatistics`, we can print a formatted report with the added bonus of the covariance matrix:

```
MultivariateSummaryStatistics:
n: 3
min: 1.0, 2.0, 0.2
max: 11.0, 21.0, 10.2
mean: 5.666666666666667, 10.0, 3.866666666666667
geometric mean: 3.8029524607613916, 6.649399761150975, 1.3477328201610665
sum of squares: 147.0, 494.0, 105.52
sum of logarithms: 4.007333185232471, 5.683579767338681, 0.8952713646500794
standard deviation: 5.033222956847166, 9.848857801796104, 5.507570547286103
covariance: Array2DRowRealMatrix{{25.3333333333,49.0,24.3333333333},
{49.0,97.0,51.0},{24.3333333333,51.0,30.3333333333}}
```

Of course, each of these quantities is accessible via their getters:

```
int d = mss.getDimension();
long n = mss.getDimension();
double[] min = mss.getMin();
double[] max = mss.getMax();
double[] mean = mss.getMean();
double[] std = mss.getStandardDeviation();
RealMatrix cov = mss.getCovariance();
```

NOTE

At this time, third- and fourth-order moments are not calculated in `SummaryStatistics` and `MultivariateSummaryStatistics` classes, so skewness and kurtosis are not available. They are in the works!

Merging Statistics

The unnormalized statistical moments and co-moments can also be merged. This is useful when data partitions are processed in parallel and the results are merged later when all subprocesses have completed.

For this task, we use the class `AggregateSummaryStatistics`. In general, statistical moments propagate as the order is increased. In other words, in order to calculate the third moment M_3 you will need the moments M_2 and M_1. It is therefore essential to calculate and update the highest order moment first and then work downward.

For example, after calculating the quantity $\delta_{2,1}$ as described earlier, update M_4 with

$$M_4 = M_{4,1} + M_{4,2} + n_1 n_2 (n_1^2 - n_1 n_2 + n_2^2) \frac{\delta_{2,1}^4}{n^3} + 6(n_1^2 M_{2,2} - n_2^2 M_{2,1}) \frac{\delta_{2,1}^2}{n^2} + 4(n_1 M_{3,2} - n_2 M_{3,1}) \frac{\delta_{2,1}}{n}$$

Then update M_3 with

$$M_3 = M_{3,1} + M_{3,2} + n_1 n_2 (n_1 - n_2) \frac{\delta_{2,1}^3}{n^2} + 3(n_1 M_{2,2} - n_2 M_{2,1}) \frac{\delta_{2,1}}{n}$$

Next, update M_2 with:

$$M_2 = M_{2,1} + M_{2,2} + n_1 n_2 \frac{\delta_{2,1}^2}{n}$$

And finally, update the mean with:

$$\bar{x} = \bar{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

Note that these update formulas are for merging data partitions where both have $n_i > 1$. If either of the partitions is a singleton ($n_i = 1$), then use the incremental update formulas from the prior section.

Here is an example demonstrating the aggregation of independent statistical summaries. Note that here, any instance of `SummaryStatistics` could be serialized and stored away for future use.

```
// The following three summaries could occur on
// three different machines at different times

SummaryStatistics ss1 = new SummaryStatistics();
ss1.addValue(1.0);
ss1.addValue(11.0);
ss1.addValue(5.0);
```

```
SummaryStatistics ss2 = new SummaryStatistics();
ss2.addValue(2.0);
ss2.addValue(12.0);
ss2.addValue(6.0);

SummaryStatistics ss3 = new SummaryStatistics();
ss3.addValue(0.0);
ss3.addValue(10.0);
ss3.addValue(4.0);

// The following can occur on any machine at
// any time later than above

List<SummaryStatistics> ls = new ArrayList<>();
ls.add(ss1);
ls.add(ss2);
ls.add(ss3);

StatisticalSummaryValues s = AggregateSummaryStatistics.aggregate(ls);

System.out.println(s);
```

This prints the following report as if the computation had occurred on a single dataset:

```
StatisticalSummaryValues:
n: 9
min: 0.0
max: 12.0
mean: 5.666666666666667
std dev: 4.444097208657794
variance: 19.75
sum: 51.0
```

Regression

The `SimpleRegression` class makes this easy, because moments and co-moments add together easily. The Aggregates statistics produce the same result as in the original statistical summary:.

```
SimpleRegression rg = new SimpleRegression();

/* x-y pairs of Anscombe's x1 and y1 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/**/
double[][] xyData2 = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

SimpleRegression rg2 = new SimpleRegression();
rg2.addData(xyData);

/* merge the regression from rg with rg2 */
rg.append(rg2);

/* get regression results for the combined regressions */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

In the case of multivariate regression, `MillerUpdatingRegression` enables a storeless regression via `MillerUpdatingRegression.addObservation(double[] x, double y)` or

`MillerUpdatingRegression.addObservations(double[][] x, double[] y)`.

```
int numVars = 3;
boolean includeIntercept = true;
MillerUpdatingRegression r =
    new MillerUpdatingRegression(numVars, includeIntercept);
double[][] x = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] y = {-1, 0.2, 0.9, 2.1};
r.addObservations(x, y);
RegressionResults rr = r.regress();
double[] params = rr.getParameterEstimates();
double[] errs = rr.getStdErrorOfEstimates();
double r2 = rr.getRSquared();
```

Using Built-in Database Functions

Most databases have built-in statistical aggregation functions. If your data is already in MySQL you may not have to import the data to a Java application. You can use built-in functions. The use of `GROUP BY` and `ORDER BY` combined with a `WHERE` clause make this a powerful way to reduce your data to statistical summaries. Keep in mind that the computation must be done somewhere, either in your application or by the database server. The trade-off is, is the data small enough that I/O and CPU is not an issue? If you don't want the DB performance to take a hit CPU-wise, exploiting all that I/O bandwidth might be OK. Other times, you would rather have the CPU in the DB app compute all the stats and use just a tiny bit of I/O to shuttle back the results to the waiting app.

> **WARNING**
>
> In MySQL, the built-in function `STDDEV` returns the population's standard deviation. Use the more specific functions `STDDEV_SAMP` and `STDDEV_POP` for respective sample and population standard deviations.

For example, we can query a table with various built-in functions, which in this case are example revenue statistics such as AVG and STDDEV from a sales table:

```
SELECT city, SUM(revenue) AS total_rev, AVG(revenue) AS avg_rev,
    STDDEV(revenue) AS std_rev
    FROM sales_table WHERE <some criteria> GROUP BY city ORDER BY total_rev DESC;
```

Note that we can use the results as is from a JDBC query or dump them directly into the constructor of `StatisticalSummaryValues(double mean, double variance, long count, double min, double max)` for further use down the line. Say we have a query like this:

```
SELECT city, AVG(revenue) AS avg_rev,
             VAR_SAMP(revenue) AS var_rev,
             COUNT(revenue) AS count_rev,
MIN(revenue) AS min_rev, MAX(revenue) AS max_rev
    FROM sales_table WHERE <some criteria> GROUP BY city;
```

We can populate each `StatistialSummaryValues` instance (arbitrarily) in a `List` or `Map` with keys equal to `city` as we iterate through the database cursor:

```
Map<String, StatisticalSummaryValues> revenueStats = new HashMap<>();

Statement st = c.createStatement();
ResultSet rs = st.executeQuery(selectSQL);
while(rs.next()) {
    StatisticalSummaryValues ss = new StatisticalSummaryValues(
        rs.getDouble("avg_rev"),
        rs.getDouble("var_rev"),
        rs.getLong("count_rev"),
        rs.getDouble("min_rev"),
```

```
            rs.getDouble("max_rev") );

        revenueStats.put(rs.getString("city"), ss);
    }
    rs.close();
    st.close();
```

Some simple database wizardry can save lots of I/O for larger datasets.

PREV
2. Linear Algebra

NEXT
4. Data Operations