Package animation: Animated Statistics Using R[1]

Yihui Xie[2]

May 31, 2008

[1]Important Notice: *this vignette has stopped being updated since May 31, 2008. The more useful resource to know about ideas and changes of the package animation is this wiki:* `http://animation.yihui.name`

[2]School of Statistics, Renmin University of China, Beijing, 100872, China; Tel: 86-10-82509086; Fax: 86-10-82509086; Email: `cat('xieyihui', 'gmail.com', sep = '@')`; Homepage: `http://www.yihui.name`; You may visit my project "AniWiki: Animations in Statistics" at `http://animation.yihui.name` for a variety of animations in statistics in web pages.

Abstract

Animated graphs are undeniably both interesting and intuitional. This vignette mainly gives a brief overview to a large variety of animations in statistics, which could probably aid in teaching statistics, data analysis, and the presentation of statistical reports. The methods of making animations are also introduced. Animations can add insight and interest to traditional static approaches to teaching statistics, making statistics a more interesting and appealing subject.

Contents

List of Figures

Chapter 1

Introduction

The concept of statistics, viewed from an analytical way, can be defined as "the study of algorithms for data analysis" ([1]). Nowadays statistical methods and models are increasing at an exploding speed, leading to more and more difficulties for people to understand those abstract mathematical and statistical algorithms. Basically this happens because sometimes it is really hard to imagine what on earth has happened behind a statistical algorithm, or how it works in processing data. While on the other hand, the size and complexity of data are also increasing, which results in the other problem for knowledge discovery. In the mean time, traditional *static* statistical reports (printed on paper in presses) for these complex data can be rather unsatisfactory for explanations of statistical results, and we need a more *active* way to present the fruits of our analysis.

> We're drowning in information and starving for knowledge.
>
> – Rutherford D. Rogers

To solve these problems, I adopted the approach of animation at last, because the human visual cortex is arguably the most powerful computing system we have access to, and visualization (especially animation) allows us to put information into a form which allows us to use the power of this computing system. Thus by virtue of our visual system we may be able to quickly understand a somewhat complicated method or result (usually in a simplified case).

However, there is currently very few work in such a literature for animations in statistics[1], therefore this vignette provides an integrated discussion on the animating of statistical models and data in the environment of R language ([7]).

[1]Most of work has been contributed to the computer science, media and entertainment industries.

Chapter 2

Tools for Animation

Perhaps most people would think of GIF images as the first choice for making animations, because it is well known that GIF is one of the only few image formats that has the ability to create animations[1], nevertheless ultimately I didn't adopt this format as the primary means for creating animations because of several reasons[2]. Anyway, it will be briefly introduced in section 2.3.

Actually it is not so convenient to make animated image files in R, whereas we still have other choices, among which I list two main tools I have employed as follows in section 2.1 and 2.2, and other possible means are mentioned in section 2.3.

2.1 R Graphical Devices

The R package grDevices has offered a variety of graphics devices, and it's really a great help when we need to produce single image files – there are several choices such as PNG, JPEG, BMP, PDF, PS, TEX/LATEX and WMF, etc. All of them work very well when producing images files one by one, but the essential problem is that none of them is able to make animation files *directly*. At most what we can do is to produce a sequence of images.

Nevertheless, we may as well just use the Windows graphics devices (under Windows) or X Window System graphics devices (under Linux) or MacOS X Quartz devices (under MacOS X) inside R to make animations, i.e. draw graphs one after another in these devices. Again, there is an obvious drawback: it's inconvenient for users who don't have R installed in their computers to watch the animations, as the pictures are displayed *inside* R and we need an independent platform to show our animations.

For users who have installed R, animations can be made with the function *Sys.sleep()* in a loop. Obviously this function is intentionally used to slow down the loop so that we can see the whole process clearly. For example, we can show

[1] There are other formats such as APNG (Animated Portable Network Graphics), MNG (Multiple-image Network Graphics) and SVG (Scalable Vector Graphics), etc, but less popular (SVG might be promising in animations).

[2] You may read this page: `http://r.yihui.name/misc/gif_pdf_grDev.htm`

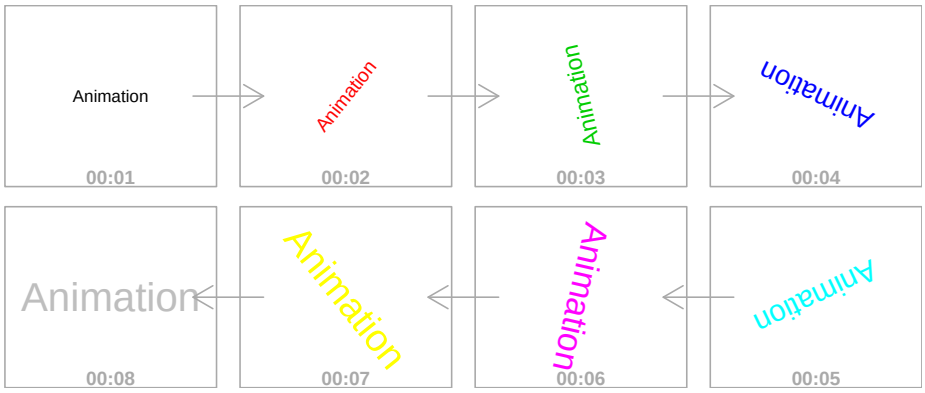

Figure 2.1: An illustration of the process of animations.

the process of rotating the word "Animation" in the loop below[3] (Figure 2.1 shows some sample frames of this animation):

```
> for (i in 1:360) {
+     plot(1, ann = F, type = "n", axes = FALSE)
+     text(1, 1, "Animation", srt = i, col = rainbow(360)[i],
+         cex = 7 * i/360)
+     Sys.sleep(0.01)
+ }
```

For detailed instructions and explanations in R graphics system, you may refer to the book "R graphics" ([6]) by Paul Murrell, or read R-help on graphics functions (e.g. packages graphics, grDevices, etc) carefully.

2.2 HTML & JavaScript

Why use HTML & JavaScript? I believe there are at least three reasons:

- R already has built-in functions for reading and writing text files, so we can create HTML files easily, e.g. use *cat()* with arguments `file` and `append`;
- Although R has no devices for image formats such as GIF, there are still many other "static" image formats which can be well shown in *web pages*, e.g. JPEG, PNG, ...;
- Generally speaking, no additional programs are needed in order to display the animations, as long as the web browser supports JavaScript – surely most browsers can meet such a simple requirement.

[3] This code is used in the header of `http://R.yihui.name`. Argument `srt` controls the rotating degree, `col` for colors, and `cex` for magnification.

The work to do is just to create single image *frames* and try to show them in an HTML page. And how can we fulfill this? The answer is through JavaScript[4]. This idea has already been implemented in this package animation: all animation functions has a special argument `saveANI`, which determines whether to generate animation files or just to show animations inside R.

2.3 Other Tools

As mentioned above, we can take GIF images as a medium for the expression of animations. However, it is likely that we need a certain special software to help with the generation of GIF images. The function *saveMovie()* in the package "animation" has provided a wrapper to create GIF animations with the help of "ImageMagick":

Usage

```
saveMovie(expr, interval = 1, moviename = "movie",
    movietype = "gif", loop = 0, dev = png,
    filename = "Rplot", fmt = "%03d", outdir = tempdir(), ...)
```

Currently there are still at least other two choices for animations: the first one is to use the package rgl which takes advantage of the OpenGL system to make 3D visualizations. The user may conveniently interact with 3D elements in the plot (drag and rotate, etc). And the second one is Scalable Vector Graphics (SVG), which is a language for describing two-dimensional graphics and graphical applications in XML. SVG files are compact and provide high-quality graphics on the Web, in print, and on resource-limited handled devices. In addition, SVG supports scripting and animation, so is ideal for interactive, data-driven, personalized graphics. Besides, SVG is a royalty-free vendor-neutral open standard developed under the W3C (World Wide Web Consortium) Process. Currently there are a few packages supporting the creation of SVG files, e.g. Cairo, cairoDevice, and RSvgDevice, etc[5].

Besides, this package might be a little bit like TeachingDemos, which contains several demonstrations for teaching and learning, however, clearly animation is focused on demonstrations that can be *animated*, and what's more, the tools for animations of these two packages are different, too[6]. The package animation also aims to cover many more fields in statistics.

And there is still another project related to the package animation. That is "the R Movies Gallery". Currently I haven't checked it carefully, but I see at least one difference: special software has to be installed to watch the movies, which is not necessary for the package animation.

[4] Refer to this page again for details: `http://r.yihui.name/misc/gif_pdf_grDev.htm`

[5] As far as I know, there is one other package gridSVG by Paul Murrell, but it is still highly experimental.

[6] TeachingDemos mainly takes advantage of the function *locator()*, the package rgl and Tcl/Tk to interact with users.

Chapter 3

Statistics and Animations

So what is the connection between statistics and animations? Generally there are three areas in which animations can be used, namely:

- algorithms involving iterations, e.g. K-Means cluster algorithm
- methods relevant to random numbers, e.g. simple random sampling
- statistics with dynamic trends, e.g. time series

Below I'll explain with examples how they work in animations respectively.

3.1 Iterative Algorithms

There are a large number of algorithms in statistics which involve iterations to optimize certain functions. For example, in the K-Means cluster analysis, the basic steps are described in Figure 3.1. What animation can do is to show the output of *each* iteration.

Here I use a cluster problem containing two numerical variables as a simplified case; actually the main reason is for the convenience of making scatterplots on the 2D plane: the x-axis and y-axis denote the two variables respectively. During the process of iterations, we may present these two elements at each iteration:

- Location of each center: just average x-y locations within each cluster.
- Temporary cluster results: annotate each cluster by different point symbols.

As the iteration goes on, both the centers and cluster membership will change – this is just the source of animation. To sum up, the animation steps may be: show centers, compute distances and cluster, show cluster membership, re-compute centers and move, re-compute distances and change cluster membership, and so one and so forth. Figure 3.2 gives two iterations of locating centers and computing distances.

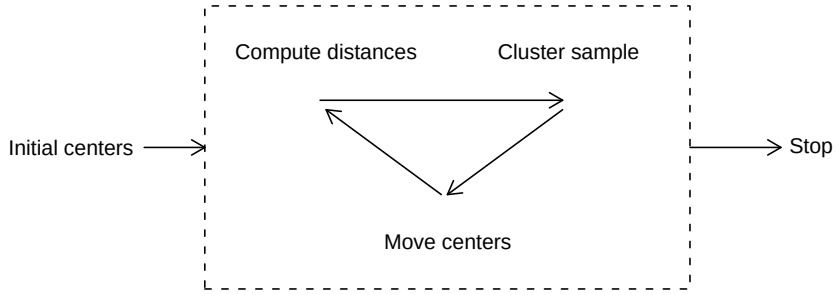

Figure 3.1: Basic steps of K-Means cluster algorithm: the iteration in the middle box will go on and on until maximum number of steps is achieved or clusters are converged.

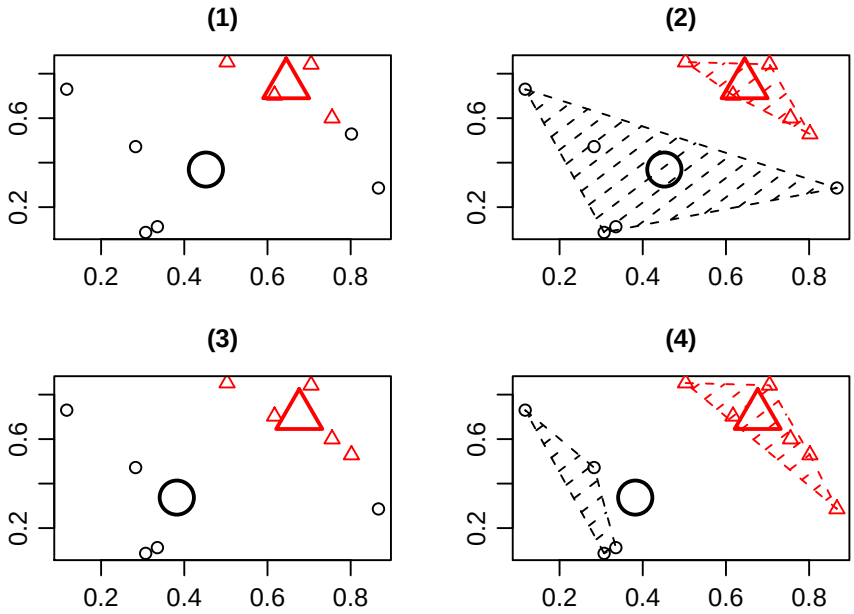

Figure 3.2: Sample iterations of K-Means cluster algorithm: (1) locate cluster centers based on the average of the last step; (2) compute distances and determine cluster membership again (centers are not moved!); (3) locate cluster centers again based on the result of (2); (4) compute distances and determine cluster membership again (centers not moved). Check carefully for the changes especially from (2) to (3) and from (3) to (4).

The animation function in the package animation is *kmeans.ani()*; see the help files for detailed usage (Section 5.5.1).

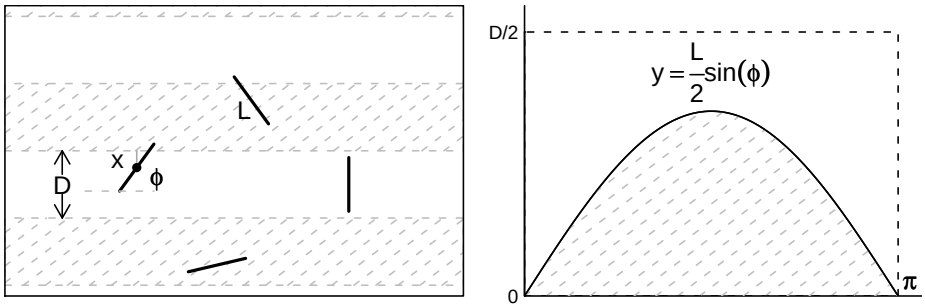

Figure 3.3: The problem of Buffon's Needle. (1) The left plot: bold segments stand for "needles"; D is the distance between lines; L is the length of needle; x is the distance from the middle of the needle to the nearest line; ϕ is the angle at which the needle falls; (2) The right plot: the needle will cross the lines if and only if $x \leq (L/2)\sin(\phi)$, i.e. the point (ϕ, x) should fall into the shadow area.

3.2 Random Numbers

Surely statistics cannot survive without randomness. We can see random numbers in subjects such as probability theory, survey sampling, and numerical simulation/optimization, etc. In areas which involve with random numbers, we may generate random numbers again and again to do simulations and get corresponding results – this is just where animations can play an important role.

Let's take the Buffon's Needle for example. This is one of the oldest problems in the field of geometrical probability and it's familiar to people who have basic knowledge of probability theory, so I wouldn't repeat the background here.

The critical parts for the simulation of this problem are:

- Randomly generate a location where the needle falls (only the middle point of the needle is needed).
- Randomly generate an angle ϕ at which the needle falls (a number in $[0, \pi]$).

After these two elements have been decided, we'll immediately know whether the needle will cross the lines or not. The problem and solution are explained in Figure 3.3, while the simulation is illustrated in Figure 3.4.

The animation function in the package animation is *buffon.needle()*; see the help files for detailed usage (Section 5.1.2).

3.3 Dynamic Trends

We are always exploring relationships among our variables, as descriptions for single variables are far from enough in statistics; therefore the method of *conditioning* is fairly important. For instance, we may want to examine how a

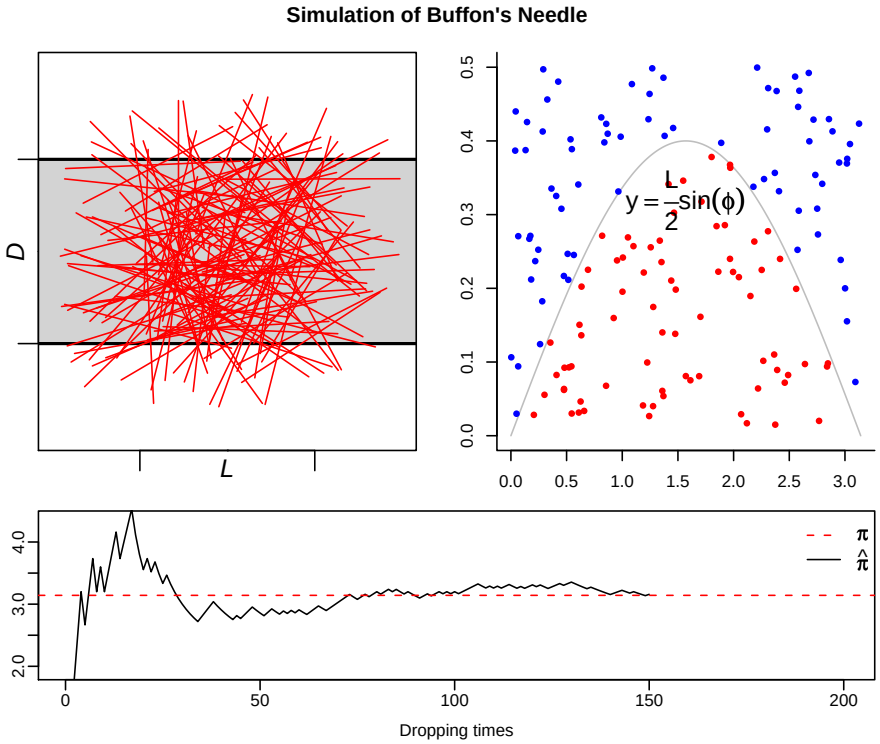

Figure 3.4: Simulation of Buffon's Needle. (1) Top left: simulation of dropping needles; (2) Top right: corresponding point pairs (ϕ, x); (3) Bottom: values of π calculated from the above simulations; actually this is the 150th animation frame taken from the whole process of 200 needle falls.

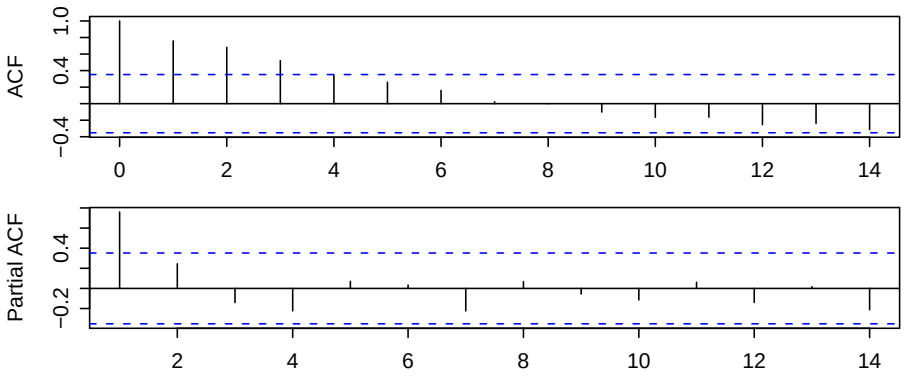

Figure 3.5: ACF and PACF for the number of visits to Yihui's website from Oct 1 to 31, 2007.

certain statistic A varies conditioned on a variable B. This "variation" (or simply "change") just builds the connection between statistics and the application of animations.

The most common case is the subject of time series[1], in which the conditioning variable is usually time.

For example, in time series analysis we often use the whole data set to fit an ARIMA model to examine the relationship between X_t and corresponding lagged terms such as $X_{t-1}, X_{t-2}, \ldots$, however, if we want to know how such a relationship varies over time, this single model surely cannot help. Here I simply employ an intuitional method called "*Moving Window Regression*" (MWR) to fulfill this idea. MWR is able to show the changes of coefficients over time, and what I do next is rather naive (just for demonstration). Further topics can be found in [8], etc.

The time series data is from the dataset `pageview` in animation, and we may have a look at the ACF and PACF plots in Figure 3.5 just for data between Oct 1 to 31, 2007.

```
> library(animation)
> data(pageview)
> x = pageview$visits[11:41]
> par(mfrow = c(2, 1))
> acf(x)
> pacf(x)
```

Not going further in the traditional ARIMA analysis, I just use an AR(1) model for computation. Suppose there are n observations $\{x_1, x_2, \ldots, x_n\}$, and the MWR method is just to split the data into $n - k + 1$ subsets depending on the window width k: $\{x_1, \ldots, x_k\}$, $\{x_2, \ldots, x_{k+1}\}$, ..., $\{x_{n-k+1}, \ldots, x_n\}$, and at last compute AR(1) models on each subset. In the code below, I computed the coefficient ϕ for $n - k + 1$ AR(1) models $x_t = \phi x_{t-1} + \epsilon_t$ using *arima()* in package stats, and during the moving (in a loop), I marked out observations used in MWR by rectangles with different colors, and plotted the corresponding coefficients as well as "$\phi \pm 2$*s.e." in the lower part of the graph. The last line `Sys.sleep(1)` is to slow down the process of moving windows so that we can clearly see the "real moving"[2].

```
> library(animation)
> data(pageview)
> x = pageview$visits[11:41]
> k = 15
> base = 0:(k - 1)
> sx = 2.5 * (x - min(x))/(max(x) - min(x)) + 1.6
> plot(sx, ylim = c(-0.3, 4.2), cex = 1.5, yaxt = "n")
> axis(2, c(0, 0.6, 1.2), col.axis = "red")
> axis(2, seq(1.6, 4.1, length = 4), seq(min(x), max(x),
```

[1] But the applications are absolutely not limited to time series!

[2] Just for demonstration; in practical applications there's no need to slow down the computation.

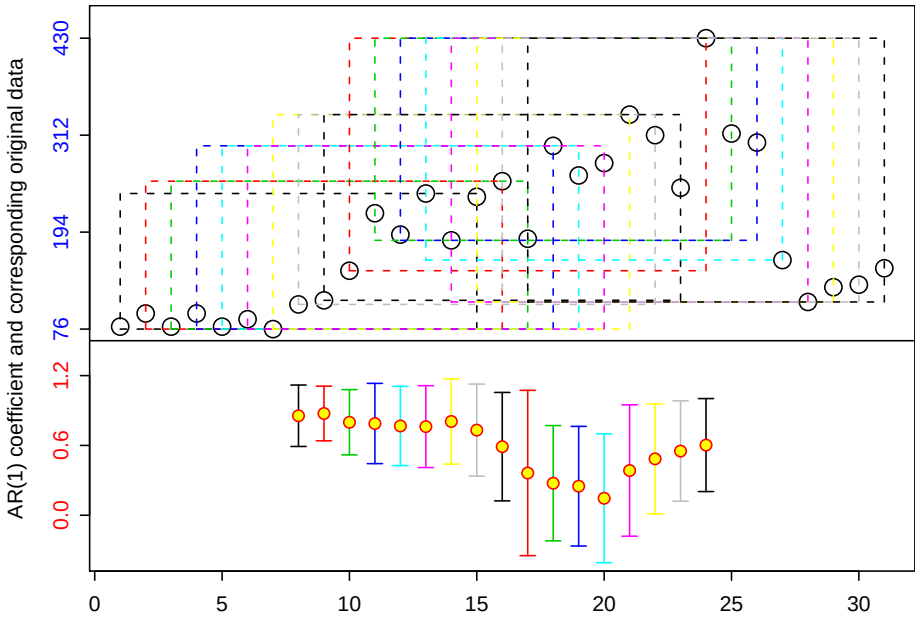

Figure 3.6: Illustration for Moving Window Auto-Regression with real data for a window width of 15 days.

```
+     length = 4), col.axis = "blue")
> abline(h = 1.5)
> for (i in 1:(length(x) - k + 1)) {
+     idx = base + i
+     m = arima(x[idx], order = c(1, 0, 0))
+     phi = coef(m)["ar1"]
+     se = sqrt(vcov(m)[1, 1])
+     rect(i, min(sx[idx]), i + k - 1, max(sx[idx]), lty = 2,
+         border = i)
+     arrows(i + k/2 - 0.5, phi - 2 * se, i + k/2 - 0.5,
+         phi + 2 * se, angle = 90, code = 3, length = 0.05,
+         col = i)
+     points(i + k/2 - 0.5, phi, pch = 21, col = "red",
+         bg = "yellow")
+     Sys.sleep(1)
+ }
```

Figure 3.6 shows the eventual result; we can roughly observe that the AR(1) coefficient ϕ is stable first, and begins to decrease in the 15 days centered at Oct 15, then increases from about Oct 13 (those 15 days are centered at Oct 20).

A general animation function for "Moving Window Auto-Regression" *mwar.ani()* is available since the package version 0.1-3. See Section 5.7.1 for usage.

Chapter 4

Package **animation** Overview

The package animation is based on the most primitive idea of animation: make picture *frames* one after another with a certain duration (time interval between frames) specified. And this dull method has also been implemented in an HTML animation page using JavaScript to animate the image frames.

Currently there are two ways for animation: one is just to show animations in a graphical device (Windows, X Window, etc), and the other is to make animations in an HTML page so that people without R installed are also able to view the animations.

There are some common arguments in each animation function controlling the way to make animations, for example, whether to save PNG files during the animation demonstration (`saveANI`), the time interval `interval` between each frame of a whole animation, and the `height` and `width` of the animation frames if they are to be saved. These arguments are controlled by a special function *ani.control()*:

Usage

```
ani.control(saveANI = FALSE, interval = 1, nmax = 50,
    width = 480, height = 480, ...)
```

The meaning of these arguments are very easy because the way of making animations is quite naive, e.g., for `saveANI`:

`saveANI = TRUE` Convert the animation frames into PNG files, which will be used in the HTML animation page.

`saveANI = FALSE` Don't generate animation files: just show animation inside R.

Nevertheless, the interpretation for `nmax` is not so apparent: it depends on the specified animation function; `nmax` is usually equal to the number of animation frames (e.g. for *brownian.motion()*) but not *always*! The reason is that sometimes there are more than one frame recorded in a single step of a

loop, for instance, there are 2 frames generated in each step of *kmeans.ani()*, and 4 frames in *knn.ani()*, etc.

To make an HTML animation page, you have to start a page first by *ani.start()*, then use any animation functions to generate PNG files in the `images` directory relative the HTML page, and at last use *ani.stop()* to complete writing the page. By default, *ani.stop()* will automatically open a web browser to view the HTML animation page[1].

Here is a sample session:

```
> library(animation)
> ani.start()
> op = par(mar = c(3, 3, 2, 0.5), mgp = c(2, 0.5, 0), tcl = -0.3,
+     cex.axis = 0.8, cex.lab = 0.8, cex.main = 1)
> brownian.motion(control = ani.control(saveANI = TRUE,
+     interval = 0.01, nmax = 100))
> par(op)
> ani.stop()
```

There a plenty of examples in the help pages of each animation functions. Just try them if you like.

Beside the approach of HTML animation pages, the function *saveMovie()* also provides another way for animations as mentioned before. Actually we may generate movies of any formats as long as they are supported by "ImageMagick", such as GIF (*.gif) or MPEG (*.mpg). Here are two examples:

```
> library(animation)
> saveMovie(for (i in 1:10) plot(runif(10), ylim = 0:1),
+     loop = 1)
> saveMovie(brownian.motion(nmax = 40), width = 600, height = 600)
```

Having provided a mechanism for generating animations, next I shall go into the huge project of statistical animations in the many branches.

[1] Use the function *browseURL()* in utils.

Chapter 5

Statistical Animations Gallery

In section 3 I have explained some basic connections between animation and the discipline of statistics. In this section I just give a summary of the animation functions in the package animation. This gallery will be supplemented day by day.

5.1 Probability Theory

Probability theory is a subject relevant to randomness. As mentioned in section 3.2, animation can be closely related to this subject.

5.1.1 Probability in Flipping Coins

In the first class of learning probability we usually begin with the probability in flipping coins or tossing dice, and the function *flip.coin()* gives a simple simulation. Here the concept of a "coin" is actually abstract: it can be anything, and you just have to specify the true probabilities (or take the default `NULL`) for each "face".

Usage

```
flip.coin(faces = 2, prob = NULL, border = "white", grid = "white",
    col = 1:2, type = "p", pch = 21, bg = "transparent",
    digits = 3, control = ani.control(interval = 0.2, nmax = 100),
    ...)
```

For example, we toss the coin for 50 times, and considering that sometimes the result of flipping a coin is neither head or tail (the coin just stands on the table!), we specify there are three possible results `Head, Tail, Stand` with probabilities `0.45, 0.1, 0.45` respectively, as the result `Stand` is not likely to happen. Figure 5.1 shows the result of flipping this coin.

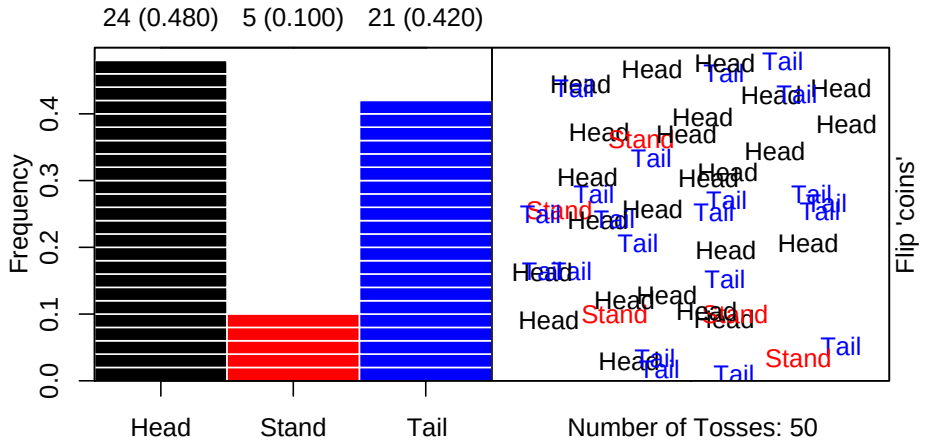

Figure 5.1: Probability of flipping a coin: Head? Tail? Or just stand on the table? Run in R to watch the animation.

```
> flip.coin(faces = c("Head", "Stand", "Tail"), interval = 0.2,
+     nmax = 50, type = "n", prob = c(0.45, 0.1, 0.45),
+     col = c(1, 2, 4))
```

You may set larger times of flipping `nmax` to check whether the frequencies will approximate to the true probabilities.

5.1.2 Buffon's Needle

This problem has been mentioned in section 3.2, so I will not repeat again here.

Usage

```
buffon.needle(l = 0.8, d = 1, redraw = TRUE,
    control = ani.control(interval = 0.05, nmax = 100), ...)
```

5.1.3 Brownian Motion

Brownian Motion, a.k.a "random walk", characterizes the trace of a point moving in a line or a plane (or in higher dimensions). Suppose the current location of the point is x_t, then the next location will be $x_{t+1} = x_t + \epsilon_{t+1}$ with i.i.d $\epsilon_t \sim N(\mu, \sigma^2)$.

It is very easy to simulate this process in R. If the initial location is 0, the next k locations can be computed simply by `cumsum(rnorm(k))`. The function *rnorm()* generates k i.i.d random numbers following Normal distribution, and *cumsum()* computes cumulative sums for these numbers, which is essentially the moving process of Brownian Motion.

The function *brownian.motion()* in animation has provided a simulation for Brownian Motion with animations.

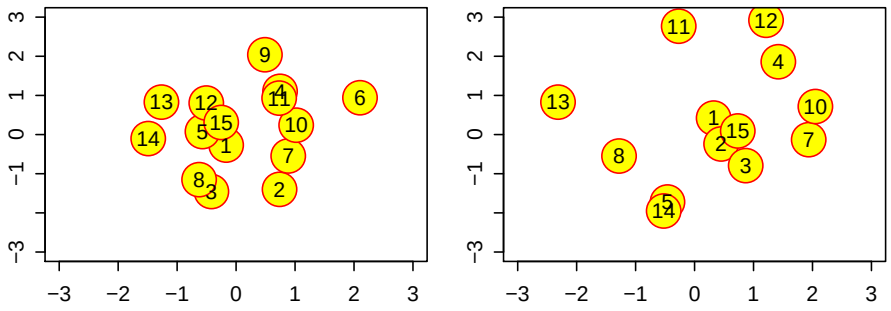

Figure 5.2: Two sample frames of Brownian Motion.

Usage

```
brownian.motion(n = 10, main = "Demonstration of Brownian Motion",
    xlim = c(-20, 20), ylim = c(-20, 20), pch = 21, cex = 5,
    col = "red", bg = "yellow", control = ani.control(nmax = 100,
    interval = 0.05), ...)
```

For example, the code below shows the traces of 15 points moving in the 2D plane for 100 steps. Figure 5.2 shows two sample frames of the animation.

```
> brownian.motion(control = ani.control(interval = 0.05,
+     nmax = 100))
```

5.1.4 Law of Large Numbers

The law of large numbers (LLN) is a theorem in probability that describes the long-term stability of a random variable. Given a sample of independent and identically distributed random variables with a finite population mean, the average of these observations will eventually approach and stay close to the population mean.

The function *lln.ani()* can illustrate this phenomenon in animations: for a series of sample sizes (denoted by i) from 1 to n, check the behavior of the sample mean corresponding to this sample size i by sampling `np` times from a certain distribution (given by the argument `FUN`). According to LLN, the sample mean will finally approach to the population mean (specified by the argument `mu`). Figure 5.3 has revealed this fact.

Usage

```
lln.ani(FUN = rnorm, mu = 0, np = 30, pch = 20,
    control = ani.control(interval = 0.3), ...)
```

Note that the arugment `FUN` is a function with two arguments `n` and `mean`, i.e. in the form of `FUN(n, mean)`. We may extend the distribution in this function by defining custom functions like:

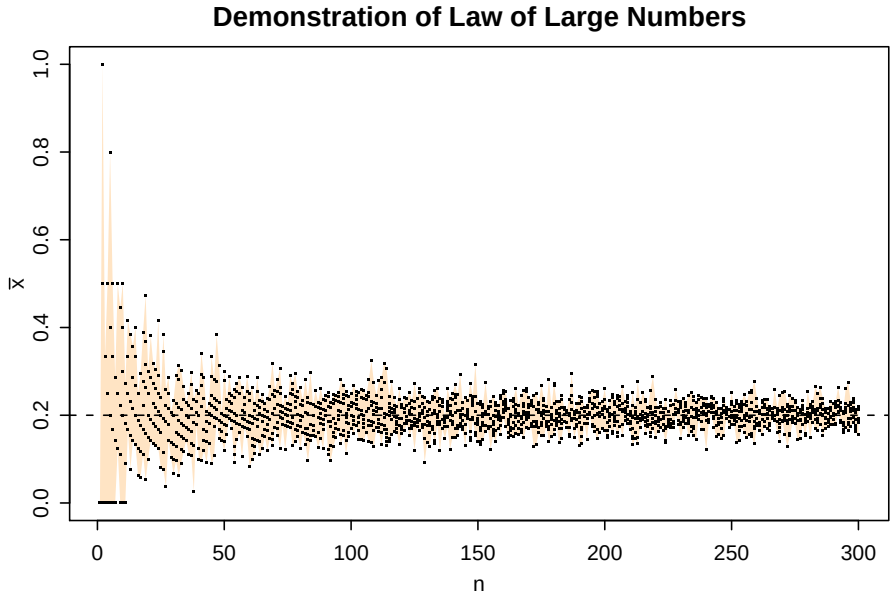

Figure 5.3: A demonstration of Law of Large Numbers. For the sample size i changing from 1 to $n = 300$: compute the sample mean for i random numbers from the Bernoulli(p) distribution with $p = 0.2$, repeat this step for 10 times to get 10 means and plot them. Apparently, as the sample size n grows, the sample mean will approach to the true mean p.

```
> f = function(n, mean) rbinom(n, 1, mean)
> lln.ani(FUN = f, mu = 0.2, np = 10, pch = ".", nmax = 100)
```

5.1.5 Monte-Carlo Simulation for Computing Areas

TODO...

Monte-Carlo integration.

5.1.6 Central Limit Theorem

The Central Limit Theorem (CLT) is one of the most famous theorems in probability theory, and here we may check the distribution of the sample mean $\bar{X}_n = \sum X_i/n$ when the sample size n is large enough. Relative theories can be easily found in most textbooks in probability theory or mathematical statistics (e.g. [5] and [2]).

The function *clt.ani()* provides a demonstration for CLT in the following schema: First of all, a number of obs observations are generated from a certain distribution for each variable X_i ($i = 1, 2, \cdots, n$), then the sample means are computed, and at last the density of these sample means is plotted as the sample size n increases.

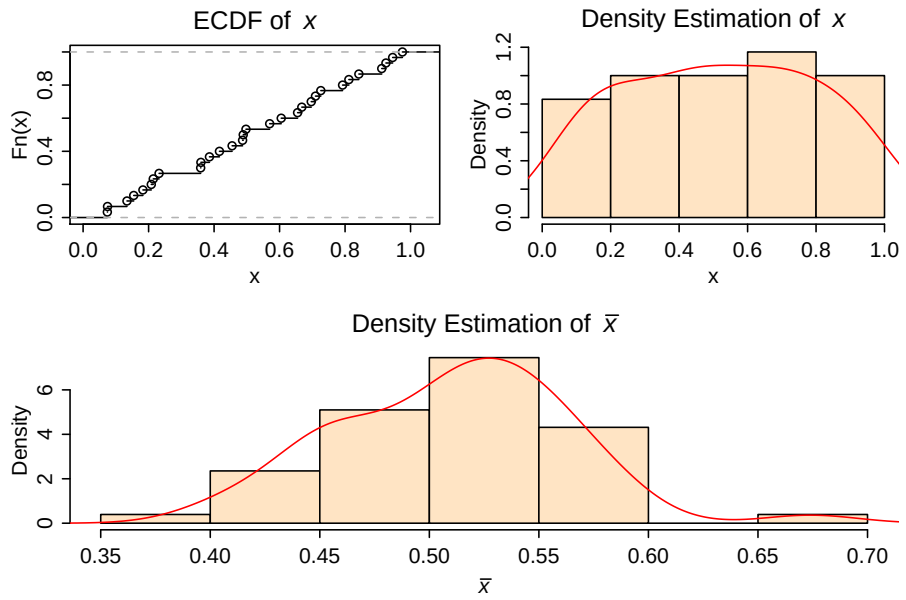

Figure 5.4: A CLT simulation for samples from the Uniform distribution: we know the original Uniform distribution is "flat", while the density of the sample mean is the well-known "bell-shaped" curve.

Usage

```
clt.ani(obs = 30, FUN = runif, control = ani.control(interval = 0.1),
    ...)
```

The function for the argument `FUN` is flexible, so we may just define it as we wish. For example, we can generate random numbers from $\chi^2(5)$ as follows:

```
> f = function(n) rchisq(n, 5)
> clt.ani(FUN = f)
```

Figure 5.4 shows a simulation for samples from the Uniform distribution. Other distributions such as Cauchy distribution (which has no finite variance[1]) or Log Normal distribution may also be tried.

5.2 Sampling Survey

Sampling survey is also a subject based on random numbers: the process of sampling is essentially generating random numbers for *indexing* the sampling frame. Therefore the problem behind sampling is just the manner to generate random numbers.

[1]This is an interesting example to show a non-Gaussian distribution when the conditions are not satisfied.

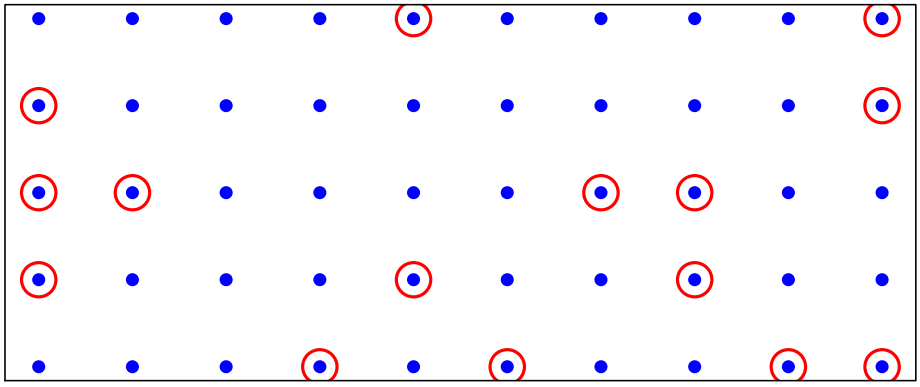

Figure 5.5: A possible result of simple random sampling: 13 points are sampled from a population of 50 points.

5.2.1 Simple Random Sampling

Simple Random Sampling is the purest form of probability sampling. Each member of the population has an equal and known chance of being selected. When there are very large populations, it is often difficult or impossible to identify every member of the population, so the pool of available subjects becomes biased.

In most cases, we conduct the sampling in a "*without-replacement*" manner, i.e. we don't put back the sample points once we pick them out. Correspondingly there is another way "sampling *with* replacement": every time before we do the sampling, we put all the individuals back again; although this is rare in practical sampling work, it's extremely important and closely related to the idea of bootstrapping (see section 5.10.1).

Here we only discuss the case of "Simple Random Sampling Without Replacement" (SRSWOR). The function *sample()* is convenient for us to conduct the sampling.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)
```

To randomly sample 10 individuals from a population of 100 elements, `sample(100, 10)` is enough for indexing. Actually the other kinds of sampling are also based on this useful function.

If we keep on sampling from a population, the samples will also change randomly, so this is the base of animations. The function *sample.simple()* provides such animations for SRSWOR.

Usage

```
sample.simple(nrow = 10, ncol = 10, size = 15,
```

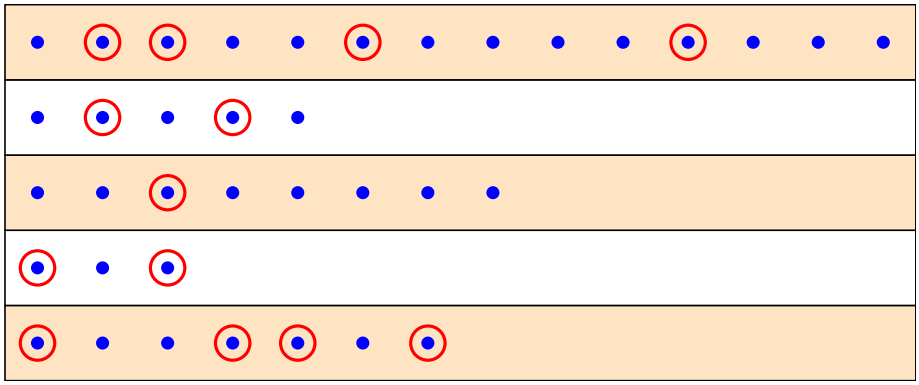

Figure 5.6: A possible result of stratified sampling: 14 points are sampled from a population of 5 stratum.

```
control = ani.control(interval = 0.2), ...)
```

Figure 5.5 is one possible result for simple random sampling.

Web page: `http://r.yihui.name/stat/sampling_survey/simple_random/`

5.2.2 Stratified Sampling

Stratified Sampling is commonly used probability method that is superior to random sampling because it reduces sampling error. A stratum is a subset of the population that share at least one common characteristic. Examples of stratums might be males and females, or managers and non-managers. The researcher first identifies the relevant stratums and their actual representation in the population. Random sampling is then used to select a sufficient number of subjects from each stratum. "Sufficient" refers to a sample size large enough for us to be reasonably confident that the stratum represents the population. Stratified sampling is often used when one or more of the stratums in the population have a low incidence relative to the other stratums.

The function *sample.strat()* provides the demonstration of stratified sampling:

Usage

```
sample.strat(pop = ceiling(10 * runif(10, 0.5, 1)),
    size = ceiling(pop * runif(length(pop), 0, 0.5)),
    control = ani.control(interval = 0.2), ...)
```

Figure 5.6 is one possible result for stratified sampling.

Web page: `http://r.yihui.name/stat/sampling_survey/stratified/`

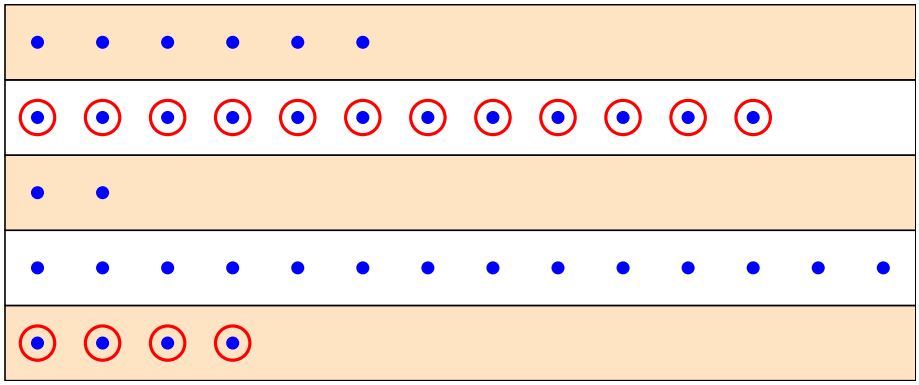

Figure 5.7: A possible result of cluster sampling: 2 clusters are sampled from a population of 5 clusters.

5.2.3 Cluster Sampling

Sometimes it is cheaper to "cluster" the sample in some way e.g. by selecting respondents from certain areas only, or certain time-periods only. (Nearly all samples are in some sense "clustered" in time – although this is rarely taken into account in the analysis.)

The function *sample.cluster()* provides the demonstration of cluster sampling:

Usage

```
sample.cluster(pop = ceiling(10 * runif(10, 0.2, 1)),
    size = 3, control = ani.control(interval = 0.2), ...)
```

Figure 5.7 is one possible result for cluster sampling.

Web page: `http://r.yihui.name/stat/sampling_survey/cluster/`

5.2.4 Systematic Sampling

Systematic Sampling is often used instead of random sampling. It is also called an Nth name selection technique. After the required sample size has been calculated, every Nth record is selected from a list of population members. As long as the list does not contain any hidden order, this sampling method is as good as the random sampling method. Its only advantage over the random sampling technique is simplicity. Systematic sampling is frequently used to select a specified number of records from a computer file.

The function *sample.system()* provides the demonstration of systematic sampling. The sample points with equal intervals are drawn out according to a random starting point.

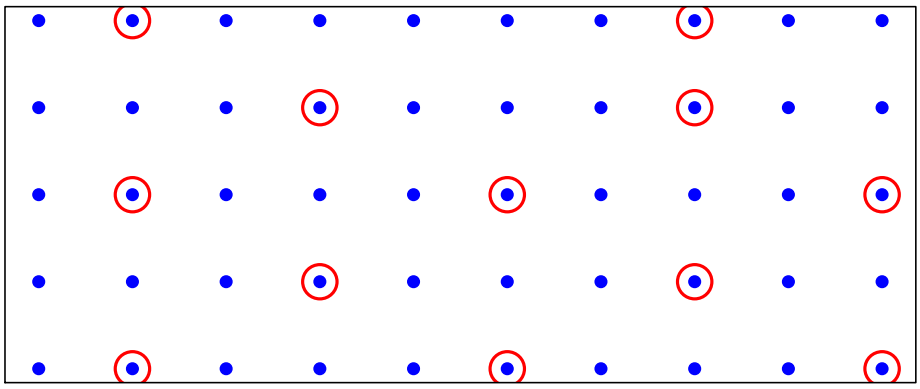

Figure 5.8: A possible result of systematic sampling: 12 points are sampled from a population of 50 points. Please note that the point at (8, 5) is the starting point!

Usage

```
sample.system(nrow = 10, ncol = 10, size = 15,
    control = ani.control(interval = 0.2), ...)
```

Figure 5.8 is one possible result for systematic sampling.

Web page: `http://r.yihui.name/stat/sampling_survey/systematic/`

5.3 Mathematical Statistics

5.3.1 Confidence Intervals

The concept of confidence intervals is quite important in mathematical statistics, but it is also confusing at the same time. A typical wrong interpretation for a confidence interval is that $P\{L(\mathbf{x}) < \theta < U(\mathbf{x})\} = 1 - \alpha$ where $\mathbf{x}$ denotes the observations. For the *observed* values of a sample, this interval is *non-random*, so there is no meaning speaking of the probability of coverage, as the probability for a fixed interval to cover the true parameter is either 1 or 0. The fact is that the intervals are random quantities, i.e. functions of the random $\mathbf{X}$ rather than the observed $\mathbf{x}$. Based on this point, the function *conf.int()* gives a illustration for the interval estimation of the parameter μ from $N(\mu, \sigma)$. For further knowledge of confidence intervals, please refer to textbooks such as [2].

Usage

```
conf.int(level = 0.95, size = 50,
    control = ani.control(interval = 0.3), ...)
```

Figure 5.9 is the result of drawing samples from $N(0, 1)$ for 50 times and plotting the corresponding intervals with segments. Obviously some segments

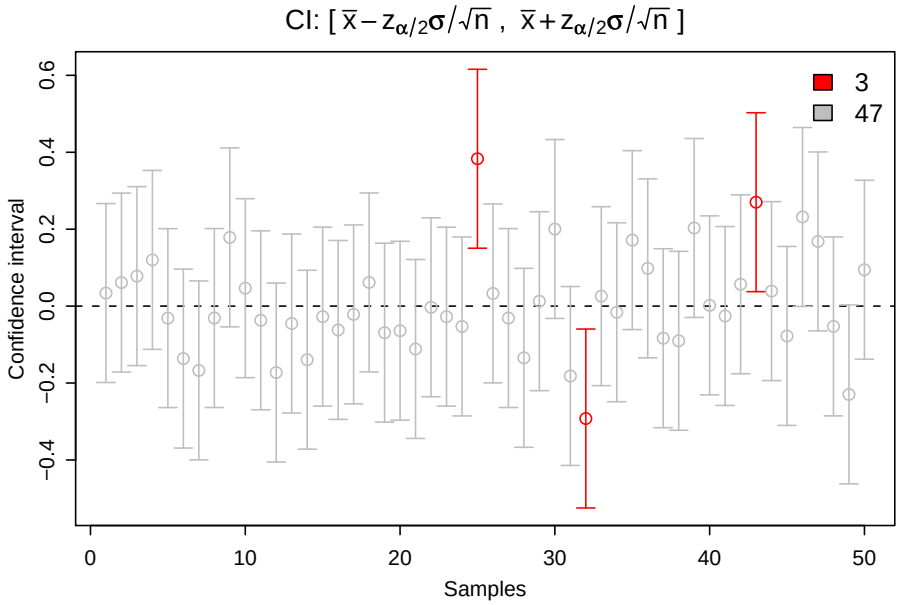

Figure 5.9: The interpretation of confidence intervals. Take samples from the Normal distribution $N(\mu, \sigma)$ (with σ known) for example: the confidence interval for the population mean μ is $[\bar{x} - z_{\alpha/2}\sigma/\sqrt{n}, \bar{x} + z_{\alpha/2}\sigma/\sqrt{n}]$, so every time we get a sample we can compute such an interval. If we keep on sampling from $N(\mu, \sigma)$, we may obtain a series of intervals because $\bar{x}$ will change according to different samples. Some intervals may cover the true mean, while other ones may not. However, in the end, we can see that this coverage rate will be close to $1 - \alpha$. In this example, the coverage rate is $47/50 = 94\%$ and the true $1 - \alpha = 95\%$.

cover the true parameter 0 and some do not. Theoretically there should be $50 * 0.95 = 47.5$ intervals covering the true parameter.

5.4 Linear Models

5.4.1 Subset Selection

5.5 Multivariate Statistics

5.5.1 K-Means Cluster Analysis

This algorithm has already been discussed in section 3.1.

Usage

```
kmeans.ani(x = matrix(runif(100), ncol = 2), centers = 2,
    control = ani.control(interval = 2, nmax = 30), ...)
```

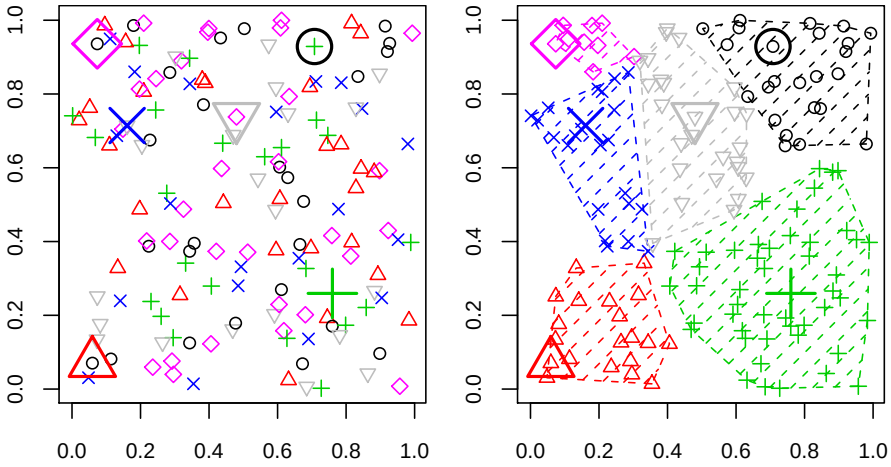

Figure 5.10: The first iteration for K-Means cluster analysis.

You may try several examples for yourself.

```
> x = matrix(runif(100), ncol = 2)
> kmeans.ani(x, centers = 2, interval = 1)
> x = matrix(runif(300), ncol = 2)
> kmeans.ani(x, centers = 6, interval = 0.5)
```

Figure 5.10 shows the first iteration in the K-Means algorithm: random centers are selected in the left plot, then distances are computed to determine the cluster membership; next we shall calculate the cluster centers again and repeat the steps till the maximum number of iterations is reached or the cluster membership is stable.

5.6 Nonparametric Statistics

5.6.1 Kernel Density Estimation

5.7 Time Series Analysis

I'm not sure whether everyone knows the famous Hans Rosling presentation[2] in which a large number of animations were displayed and the talk was a great success (at least it seemed to be) due to the exciting moving pictures (surely as well as his excellent skills at giving presentations). Hans just showed the changing relationship of several variables over time.

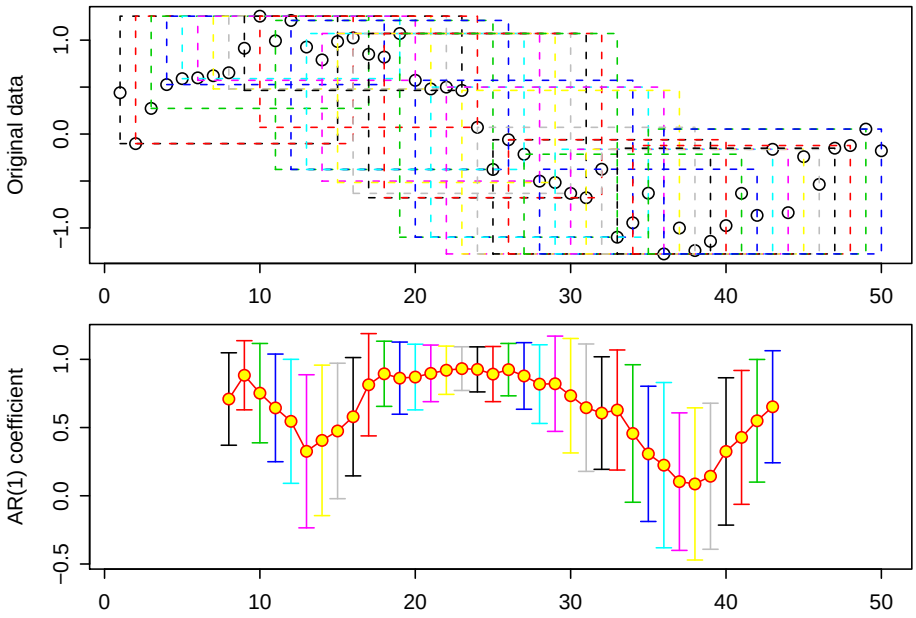

Figure 5.11: A general illustration for Moving Window Auto-Regression: the top plot is the original data points, and the bottom plot shows the AR(1) coefficients (with confidence intervals) changing along with time.

5.7.1 Moving Window Auto-Regression

As described in Section 3.3, we may check the variation of some statistics over time, and MVR is just one kind of such methods.

Usage

```
mwar.ani(x, k = 15, conf = 2, control = ani.control(), ...)
```

This function just fulfills a very naive idea about moving window regression using rectangles to denote the "windows" and move them, and the corresponding AR(1) coefficients as long as rough confidence intervals are computed for data points inside the "windows" during the process of moving.

Figure 5.11 demonstrates a process of moving windows and computing AR(1) coefficients for a random sample. Both the colors and locations of the above "windows" are corresponding to the confidence intervals below.

You may try more examples like:

```
> mwar.ani(interval = 0.3)
> data(pageview)
> mwar.ani(pageview$visits, k = 30, interval = 0.2)
```

[2]It was entitled "Debunking third-world myths with the best stats you've ever seen".

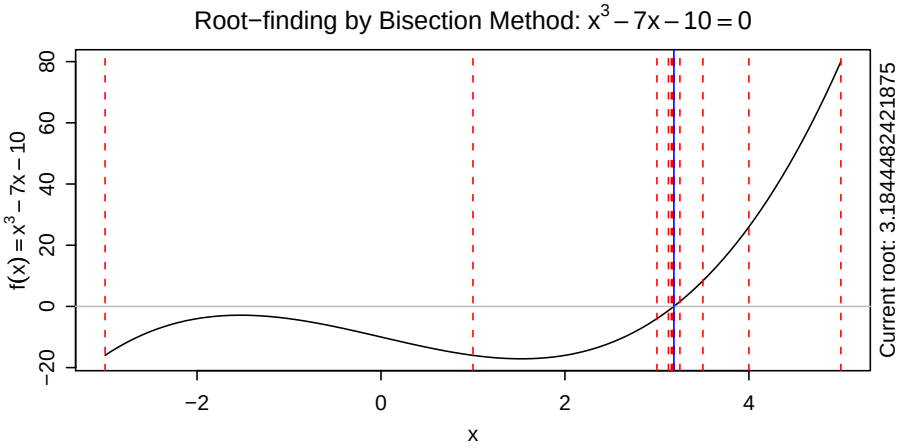

Figure 5.12: Bisection method for root-finding on an interval. The equation to solve is $x^3 - 7x - 10 = 0$, and the initial interval is $[-3, 5]$.

```
> ani.start()
> mwar.ani(interval = 0, width = 600, height = 500)
> ani.stop()
```

5.8 Computational Statistics

Computational Statistics is the area of specialization within statistics that includes statistical visualization and other computationally-intensive methods of statistics. Computational statistics is built on mathematical theories and statistical methods, and includes visualization, statistical computing, and Monte Carlo methods, etc.

One of the characteristics of computational statistics is that it often involves with iterations and random numbers, which have been mentioned in section 3.1 and 3.2 respectively. A large proportion of demonstrations in this section have shown these two features (especially iterations). Usually the frames of animations are in accordance with iterations in the algorithms.

5.8.1 Bisection Method

In mathematics, the bisection method is a root-finding algorithm which works by repeatedly dividing an interval in half and then selecting the subinterval in which a root exists.

Suppose we want to solve the equation $f(x) = 0$. Given two points a and b such that $f(a)$ and $f(b)$ have opposite signs, we know by the *intermediate value theorem* that f must have at least one root in the interval $[a, b]$ as long as f is continuous on this interval. The bisection method divides the interval in two by computing $c = (a + b)/2$. There are now two possibilities: either $f(a)$ and

$f(c)$ have opposite signs, or $f(c)$ and $f(b)$ have opposite signs. The bisection algorithm is then applied recursively to the sub-interval where the sign change occurs.

The function *bisection.method()* gives a visual demonstration of this process of finding the root of an equation $f(x) = 0$. During the process of searching, the mid-points of subintervals are annotated in the graph by both texts (indicating the values) and blue straight lines, and the end-points are denoted in dashed red lines. The root of each iteration is also plotted in the right margin of the graph.

Usage

```
bisection.method(FUN = function(x) x^2 - 4, rg = c(-1, 10),
    tol = 0.001, interact = FALSE, control = ani.control(), ...)
```

Figure 5.12 is an example for finding the root of the equation $x^3 - 7x - 10 = 0$.

```
> f = function(x) x^3 - 7 * x - 10
> xx = bisection.method(f, c(-3, 5))
```

5.8.2 Gradient Descent Algorithm

Gradient descent is an optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or the approximate gradient) of the function at the current point. If instead one takes steps proportional to the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

Gradient descent is based on the observation that if the real-valued function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$. It follows that, if

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a}) \tag{5.1}$$

for $\gamma > 0$ a small enough number, then $F(\mathbf{a}) \geq F(\mathbf{b})$. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),\ n \geq 0. \tag{5.2}$$

We have

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \ldots, \tag{5.3}$$

so hopefully the sequence $\{\mathbf{x}_n\}$ converges to the desired local minimum.

The function *grad.desc()* has provided a visual illustration for this process of minimization:

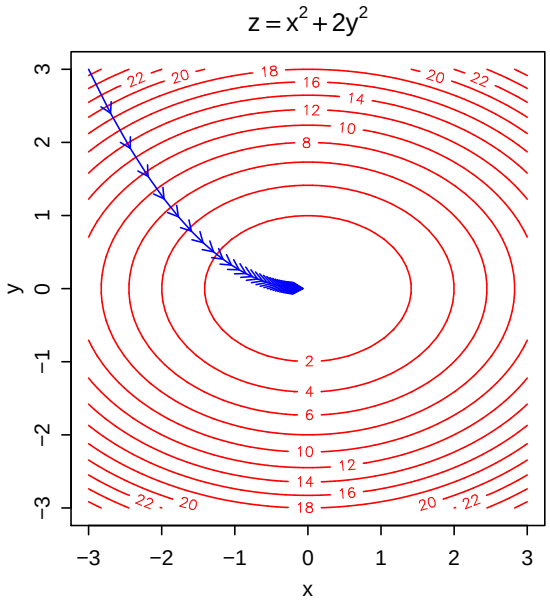

Figure 5.13: Gradient descent algorithm for a 2-D case: $z(x, y) = x^2 + 2y^2$. By default, the iteration starts at the point $(-3, 3)$. The arrows are indicating the detailed process of minimization. Theoretically the point $(0, 0)$ is the best solution, and we can see from the figure that the solution in the end is quite close to this point.

Usage

```
grad.desc(FUN = function(x, y) x^2 + 2 * y^2, rg = c(-3, -3, 3, 3),
    init = c(-3, 3), gamma = 0.05, tol = 0.001, len = 50,
    interact = FALSE, control = ani.control(interval = 0.1), ...)
```

Note that the iteration will stop if the change in the `FUN` values is smaller than the threshold `tol`. Figure 5.13 shows a illustration of minimizing the function $z(x, y) = x^2 + 2y^2$. Actually we may try even more complex function, for example:

```
> f1 = function(x, y) x^2 + 3 * sin(y)
> xx = grad.desc(f1, pi * c(-2, -2, 2, 2), c(-2 * pi, 2))
> xx$persp(col = "lightblue", theta = 30, phi = 30)
> f2 = function(x, y) sin(1/2 * x^2 - 1/4 * y^2 + 3) *
+     cos(2 * x + 1 - exp(y))
> xx = grad.desc(f2, c(-2, -2, 2, 2), c(-1, 0.5), gamma = 0.1,
+     tol = 1e-04, nmax = 200, interval = 0)
> xx = grad.desc(f2, c(-2, -2, 2, 2), interact = TRUE,
+     tol = 1e-04, nmax = 200, interval = 0)
> xx$persp(col = "lightblue", theta = 30, phi = 30)
```

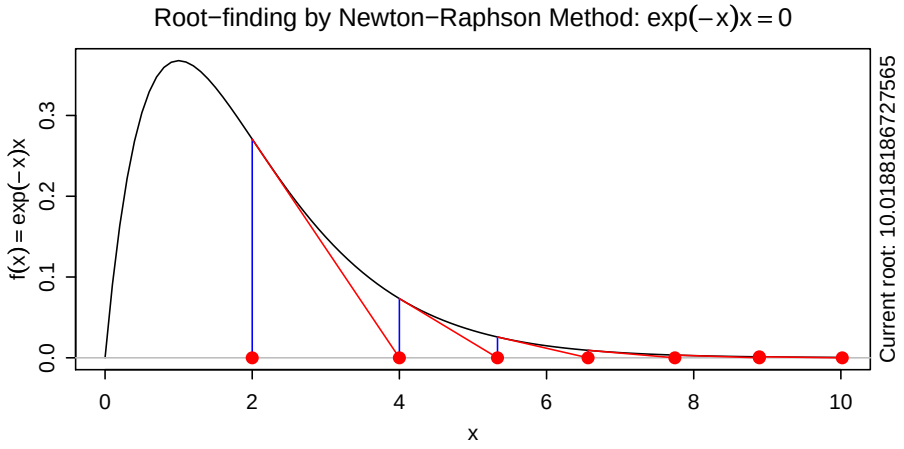

Figure 5.14: Newton-Raphson method to solve an equation $e^{-x}x = 0$ from a starting value $x_0 = 2$.

5.8.3 Newton's Method

In numerical analysis, Newton's method (also known as the Newton-Raphson method or the Newton-Fourier method) is an efficient algorithm for finding approximations to the zeros (or roots) of a real-valued function. As such, it is an example of a root-finding algorithm. It produces iteratively a sequence of approximations to the root, their rate of convergence to the root is quadratic. It can also be used to find a minimum or maximum of such a function, by finding a zero in the function's first derivative.

The function *newton.method()* is a very simple implementation of Newton's method. The iteration in the code follows this formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{5.4}$$

From the starting value x_0, blue vertical lines and red points are plotted to show the location of the sequence of iteration values $x_1, x_2, \ldots$; red tangent lines are drawn to illustrate the relationship between successive iterations; the iteration values are in the right margin of the plot.

Usage

```
newton.method(FUN = function(x) x^2 - 4, init = 10,
    rg = c(-1, 10), tol = 0.001, interact = FALSE,
    control = ani.control(interval = 2), ...)
```

Figure 5.14 shows the process of finding the root of $e^{-x}x = 0$ by Newton's method from a starting value $x_0 = 2$.

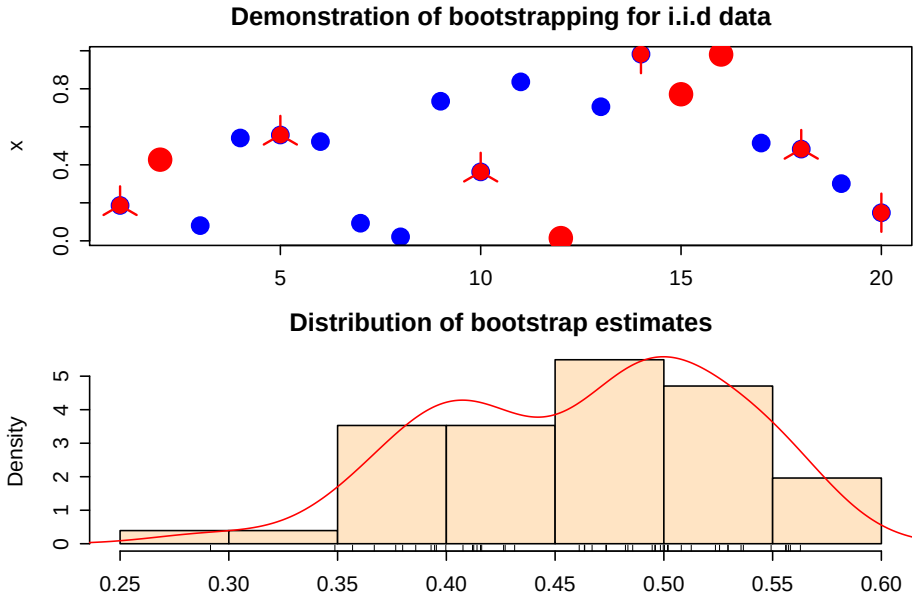

Figure 5.15: Bootstrapping for i.i.d data.

```
> newton.method(function(x) exp(-x) * x, rg = c(0, 10),
+     init = 2)
```

5.9 Data Mining

5.10 Machine Learning

5.10.1 Bootstrapping

What I am to introduce here is rather superficial; for further knowledge about bootstrapping, please refer to [3] for theories, and [9] can also be a simple guide to implementations in S language.

The two critical points for bootstrapping are: (1) data generating mechanism; (2) plug-in principle. The first tells us how to re-generate data from a sample, while the latter point tells us how to make estimations. The idea of bootstrapping is based on the method of resampling to a large degree. In the real world, we only have one sample, say, n sample points $x_1, x_2, \ldots, x_n$, then the problems we must face (when making inferences) are:

- How to guarantee the population distribution which we have assumed is correct?
- How to derive the expression of the point estimate or confidence interval of a parameter if the population distribution is too complicated?

- Or how can we obtain the distribution of a statistic when the population distribution is complicated?

We have always been deriving mathematical formulae... for this statistic... for that statistic... under perfect but unwarranted assumptions...

Why not re-generate some samples (resample the original sample *with* replacement) and re-compute the values of our statistic of interest? Then we can get a series of estimations of a certain parameter and in a result, we are able to make inferences based on these numbers using the plug-in principle, e.g. we may compute the standard error of a parameter by compute the corresponding standard error of that series of numbers (please do note the factual computation is not exactly so; read the references to learn the details), and estimate the quantiles of a statistic just by computing the corresponding quantiles of that series of numbers, etc. If you are confused by my description here, just keep on to the below animation example.

The function *boot.iid()* is provided for bootstrapping i.i.d data since the package version 0.1-2.

Usage

```
boot.iid(x = runif(20), statistic = mean, m = length(x),
    control = ani.control(), ...)
```

We just resample `m` points from `x` for `nmax` times, and in each time we compute the statistics of interest (e.g. mean, median, quantiles, etc). In the end, the approximate distribution of the statistic is illustrated in a histogram with a density line.

The blue points denote the original dataset, while the red points with (possible) leaves denote sample points being resampled; the number of leaves in the sunflower scatter plot just means how many times these points are resampled, as bootstrap samples *with* replacement.

Figure 5.15 is a demonstration of bootstrapping 20 random numbers following $U(0, 1)$ for the distribution of the sample mean. Here are some more examples:

```
> boot.iid()
> boot.iid(x = rchisq(15, 5), statistic = median)
> ani.start()
> boot.iid(saveANI = TRUE, width = 600, height = 500, interval = 0)
> ani.stop()
```

Web page: `http://r.yihui.name/stat/machine_learning/bootstrapping/`

5.10.2 k-fold / Leave-one-out Cross-validation

Cross-validation, sometimes called rotation estimation ([4]), is the statistical practice of partitioning a sample of data into subsets such that the analysis is

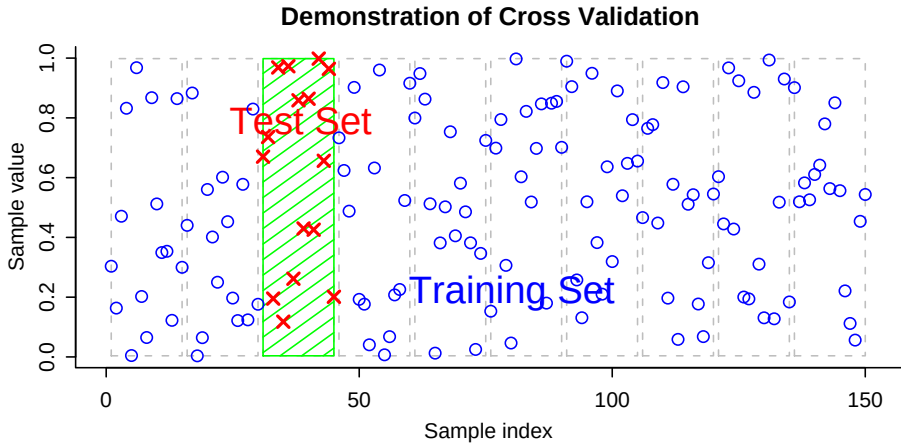

Figure 5.16: An illustration of 10-fold cross-validation.

initially performed on a single subset, while the other subset(s) are retained for subsequent use in confirming and validating the initial analysis.

The initial subset of data is called the *training set*; the other subset(s) are called *validation sets* or *testing sets*.

The theory of cross-validation was inaugurated by Seymour Geisser. It is important in guarding against testing hypotheses suggested by the data ("Type III error"), especially where further samples are hazardous, costly or impossible (uncomfortable science) to collect.

The function *cv.ani()* provides an illustration for k-fold cross-validation. Computation of the sizes of subsets is based on the function *kfcv()*. When `k` is specified as `length(x)`, the k-fold cross-validation will become "leave-one-out cross-validation".

Usage

```
cv.ani(x = runif(150), k = 10, control = ani.control(interval = 2,
    nmax = 50), ...)
```

Figure 5.16 shows a possible partition of the whole data set into a training set and test set (10-fold cross-validation). The test set can move from the first part to the last part, and this is the base for animations.

5.10.3 k-Nearest Neighbor Classification

The k-nearest neighbor algorithm is amongst the simplest of all machine learning algorithms. It is a supervised learning algorithm where the result of new instance query is classified based on majority of k-nearest neighbor category. The purpose of this algorithm is to classify a new object based on attributes and training samples. The classifiers do not use any model to fit and only based

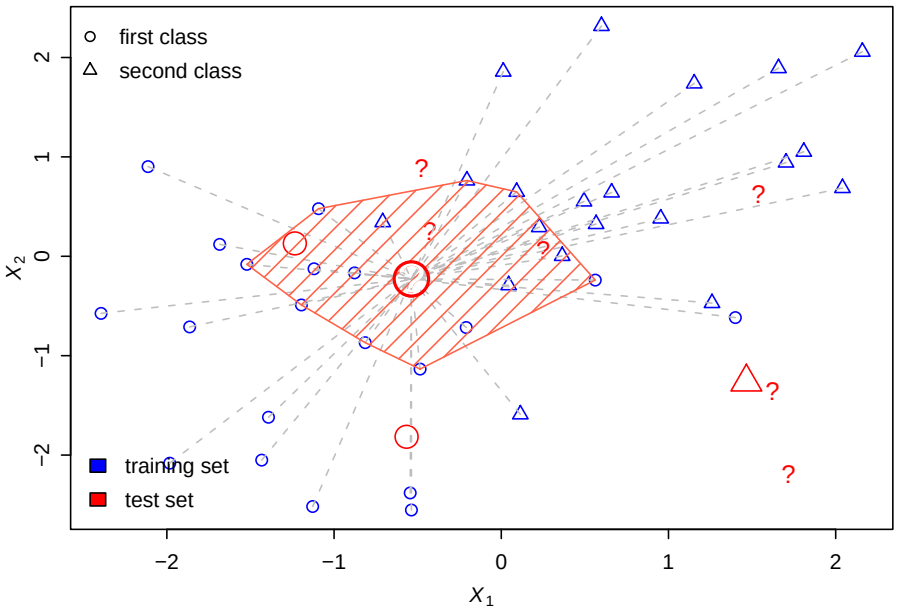

Figure 5.17: kNN algorithm in the 2D plane: gray dashed lines stand for "distances" so that neighbors can be decided; the red polygon means the k nearest neighbors (15 in this example); at last let these neighbors vote for the classification, and the symbol (classification) was changed according to the majority vote. Points marked by question marks "?" are the remaining points in the test set with unknown classifications yet.

on memory. Given a query point, we find k number of objects or (training points) closest to the query point. The classification is using majority vote among the classification of the k objects. Any ties can be broken at random. k-nearest neighbor algorithm used neighborhood classification as the prediction value of the new query instance.

The function *knn.ani()* provides the animated demonstration for kNN algorithm in the 2D case.

Usage

```
knn.ani(train, test, cl, k = 10, interact = FALSE,
    control = ani.control(), ...)
```

You may either provide a test set or specify `interact = TRUE` so that you can simply use mouse-click to decide the test set. Figure 5.17 is an intermediate result of the whole classification process.

Acknowledgements

I'm grateful to Dr Paul Murrell for his instructions and suggestions on the initial idea of my animation package. I'd like to thank Gregor Gorjanc for reminding me of creating an RSS feed for my web site `http://R.yihui.name`, and John Maindonald for valuable comments as well as a large number of ideas for statistical animations (I'm still working on them).

Appendix A

R Graphics

Appendix B

Misc Functions in animation

Some miscellaneous functions are provided in the package animation too: some might be of help with using R, some can fulfill certain tasks in the operating system, and some are just for fun (e.g. visual illusions).

B.1 Functions for R

B.1.1 Tidy up the Source Code

For people who are lazy to type spaces and tabs in the source code, this simple function *tidy.source()* might be of a little help, which mainly uses *parse()* to get the parsed code.

Usage

```
tidy.source(source, ...)
```

For example, this is the original (ugly) code:

```
pdf('kmeansframe.pdf',height=3)
par(mfrow=c(1,2),mar=c(2,2,0,1),cex.axis=.8,cex.main=1,ann=F)
x= matrix(runif(300), ncol = 2)
kmeans.ani(x,6,interval=0,nmax=1)
dev.off()
```

Proper spaces, tabs, and indents will be added after *tidy.source()* is used to the file:

```
pdf("kmeansframe.pdf", height = 3)
par(mfrow = c(1, 2), mar = c(2, 2, 0, 1), cex.axis = 0.8,
    cex.main = 1, ann = F)
x = matrix(runif(300), ncol = 2)
kmeans.ani(x, 6, interval = 0, nmax = 1)
dev.off()
```

B.1.2 Generate R Definition File for Highlight

The default definition file for R in the software Highlight[1] is somewhat incomplete, and this function *highlight.def()* is to dynamically generate such a file according to packages in the search path.

Usage

```
highlight.def(file = "r.lang")
```

Just copy the output to the directory `langDefs` of Highlight, and you will be able to convert your R code into other formats such as HTML, which is used almost everywhere in `http://R.yihui.name`.

B.2 Functions for Systems

TODO

rename a sequence of files *rename.seq()*

B.3 Functions for the Web (HTML/XML/RSS)

Here are some functions related to web pages (HTML/XML/RSS):

B.3.1 Create RSS Feed from a CSV Data File

The function *write.rss()* can create an RSS feed given an appropriate data file; I choose the CSV format because it's relatively convenient to edit with other software.

Usage

```
write.rss(file = "feed.xml", entry = "rss.csv", xmlver = "1.0",
    rssver = "2.0", title = "What's New?",
    link = "http://R.yihui.name",
    description = "Animated Statistics Using R",
    language = "en-us", copyright = "Copyright 2007, Yihui Xie",
    pubDate = Sys.time(), lastBuildDate = Sys.time(),
    docs = "http://R.yihui.name",
    generator = "Function write.rss() in R package animation",
    managingEditor = "xieyihui[at]gmail.com",
    webMaster = "xieyihui[at]gmail.com",
    maxitem = 10, ...)
```

[1] Highlight is a freeware for converting source code to formatted text with syntax highlighting by André Simon; it is released under the terms of the GNU GPL license. It supports more than 100 programming languages, including R.

title	link	author	description	pubDate	guid	category
...	...	...	...	...	...	...

Table B.1: Common elements of an item in an RSS feed.

The structure of the CSV data file is just like Table B.1. Note the order of items in the CSV file: newer items are added to the end of the file. But this order will be *reversed* in the RSS file! Just refer to `http://cyber.law.harvard.edu/rss/rss.html` for the specification of an RSS file. Here is a simple example:

```
> write.rss(entry = system.file("js", "rss.csv", package = "animation"))
```

The result might be like this:

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>What's New in 'Animated Statistics Using R'?</title>
    <link>http://R.yihui.name</link>
    <description>Animated Statistics Using R</description>
    <language>en-us</language>
    <pubDate>Mon, 17 Dec 2007 13:19:15 GMT</pubDate>
    <lastBuildDate>Mon, 17 Dec 2007 13:19:15 GMT</lastBuildDate>
    <docs>http://R.yihui.name</docs>
    <generator>write.rss() in R package animation</generator>
    <managingEditor>xieyihui[at]gmail.com</managingEditor>
    <webMaster>xieyihui[at]gmail.com</webMaster>
    <item>
      <title>The Way of Animation Found</title>
      <link>http://r.yihui.name/misc/java.htm</link>
      <author>Yihui Xie, xieyihui[at]gmail.com</author>
      <description>
      <![CDATA[
      <p>I spent really a lot of time on searching for a proper
      way for animations. After <a href="../misc/gif_pdf_grDev.htm">
      several trials</a>, I found <a href="../misc/java.htm">
      JavaScript could help me</a>.
      <acronym title="The HTML Document Object Model">HTML DOM
      </acronym> is really important. </p>
      <p>Cheers! Users won't need R or any other special programs
      to see my animations now -- just a browser supporting
      JavaScript is enough. </p>
      ]]>
      </description>
      <pubDate>Sun, 14 Oct 2007 00:00:00 GMT</pubDate>
      <guid>2</guid>
      <category>Technique</category>
    </item>
    <item>
```

```
      <title>Creation of This Website</title>
      <link>http://r.yihui.name/news/index.htm</link>
      <author>Yihui Xie, xieyihui[at]gmail.com</author>
      <description>
      <![CDATA[
      <p>Personally I don't like those complicated mathematical
      theories... And I believe many people hold the same opinion
      with me. I want to find some simpler approaches to learn
      statistics. That's my original motivation to create such a
      website.</p>
      <p>Sure, I'm too lazy. Simulation and graphics alone cannot
      contribute to mathematics and statistics directly. So, we
      should face the reality anyway... We should <em>prove</em>
      the effect of gradient descent algorithm <em>X<sub>n+1</sub>
      = X<sub>n</sub> - &gamma;F'(X<sub>n</sub>)</em> instead of
      just giving an illustration. </p>
      ]]>
      </description>
      <pubDate>Sat, 06 Oct 2007 00:00:00 GMT</pubDate>
      <guid>1</guid>
      <category>Web</category>
    </item>
   </channel>
</rss>
```

B.4 Visual Illusions

I write these functions just for fun – they don't have anything to do with statistics. The names of these functions are like *vi.*()*.

B.4.1 Lilac Chaser

Just stare at the center cross for a few (say 30) seconds to experience the phenomena of the illusion.

Usage

```
vi.lilac.chaser(np = 16, col = "magenta", bg = "gray",
    cex.p = 7, cex.c = 5, control = ani.control(interval = 0.05,
    nmax = 30), ...)
```

For details please refer to `http://en.wikipedia.org/wiki/Lilac_chaser`.

B.4.2 Grid Illusions

A grid illusion is any kind of grid that deceives a person's vision. The two most common types of grid illusions are Hermann grid illusions and Scintillating grid

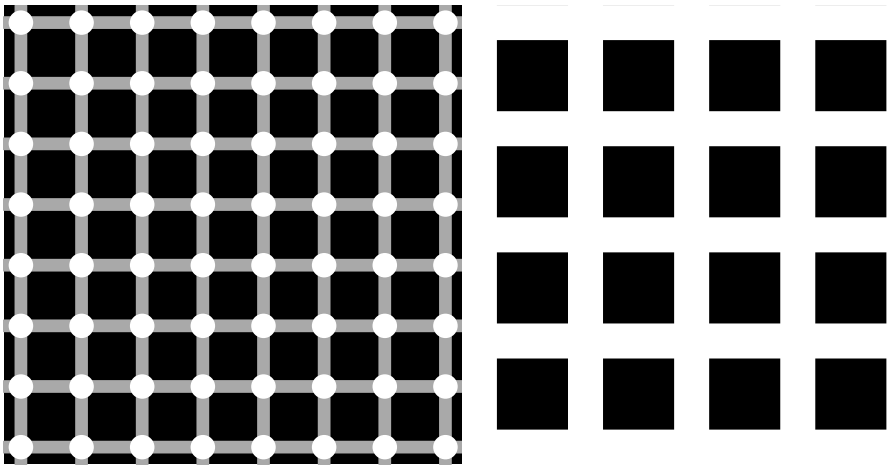

Figure B.1: Scintillating grid illusions and Hermann grid illusions: are there really some moving dots?

illusions. This function *vi.grid.illusion()* provides illustrations for both illusions as in Figure B.1.

Usage

```
vi.grid.illusion(nrow = 8, ncol = 8, lwd = 8, cex = 3,
    col = "darkgray", type = c("s", "h"))
```

For details please refer to `http://en.wikipedia.org/wiki/Grid_illusion`.

Bibliography

[1] Rudolf Beran. The impact of the bootstrap on statistical algorithms and theory. *Statistical Science*, 18(2):175–184, 2003.

[2] George Casella and Roger L. Berger. *Statistical Inference*. Duxbury Press, 2th edition, 2001.

[3] Bradley Efron and Robert Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1994.

[4] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 2(12):1137–1143, 1995.

[5] E. L. Lehmann. *Elements of Large-Sample Theory*. Springer-Verlag, New York, 1999.

[6] Paul Murrell. *R Graphics*. Chapman & Hall/CRC, 2005.

[7] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007. ISBN 3-900051-07-0.

[8] Robert A. Meyer, Jr. Estimating coefficients that change over time. *International Economic Review*, 13(3):705–710, 1972.

[9] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, 4th edition, 2002.