

DR LEE FAWCETT

MAS1343
COMPUTATIONAL
PROBABILITY AND
STATISTICS
(WITH R)

NEWCASTLE UNIVERSITY

Timetable

- Times:
 - Tuesday @ 4pm to 5pm: Lecture in LT3.
 - Thursday @ 2pm to 4pm: Lecture in LT3.
 - Friday @ 1pm to 2pm: Computer lab in Herschel PC cluster
- Attendance will be taken in all labs
- Any box with a ✕ indicates no class that day.
- Feedback and Problems classes:
 - Thursday will not *always* be a two hour lecture. It will often include time for problems and feedback.
- This timetable may change, so check Blackboard and your University email regularly.
- CBAs: This module has a single CBA.

Practicals: Solutions to all practical work should be handed in by 4pm at the School office. Deadlines are shown in the calendar. For the practicals, use a cover sheet found in the Maths & Stats reception area, (remember to stamp it with the date), and put it in the post box. You can hand these practicals in early if you wish. However, I would strongly recommend that you hand them in at most 3 days early (otherwise they may get lost in the ‘system’).

Class Schedule

TUESDAY	THURSDAY	FRIDAY
Jan 28th 1 Lecture: §1	30th 2 Lecture: §2	31st 3 ✗
Feb 4th 4 ✗	6th 5 Lecture: §3	7th 6 Lab: Practical 1
11th 7 ✗	13th 8 Lecture: §4 & §5 Hand-in: Practical 1	14th 9 Lab: Practical 2 Hand-in: CBA 1
18th 10 ✗	20th 11 Lecture: §5 Hand-in: Practical 2	21st 12 ✗
25th 13 ✗	27th 14 Lecture: §5	28th 15 Lab: Practical 3
Mar 4th 16 ✗	6th 17 Lecture: §6 Hand-in: Practical 3	7th 18 Lab: Practical 4
11th 19 ✗	13th 20 Lecture: §7 Hand-in: Practical 4	14th 21 ✗
18th 22 Tutorial	20th 23 Lecture: §7	21st 24 ✗
25th 25 ✗	27th 26 Lecture: §8 Hand-in: Practical 5	28th 27 Lab: Practical 6

TERM 2 SCHEDULE.

CBA 1 practice week

CBA 1 assessed week

TUESDAY	THURSDAY	FRIDAY
Apr 29th 1 ✗	May 1st 2 Lecture: §9 Hand-in: Practical 6	2nd 3 ✗
6th 4 ✗	8th 5 Lecture: Revision	9th 6 ✗
13th 7 ✗	15th 8 ✗	16th 9 ✗

TERM 3 SCHEDULE.

Contents

<i>1</i>	<i>Introduction and Housekeeping</i>	5
<i>2</i>	<i>Introduction to R</i>	9
<i>3</i>	<i>Summary Statistics</i>	17
<i>4</i>	<i>Graphical Presentation of Data</i>	23
<i>5</i>	<i>Control Statements and Functions</i>	29
<i>6</i>	<i>Random Number Generation</i>	37
<i>7</i>	<i>Simulating Discrete Random Numbers</i>	44
<i>8</i>	<i>Monte Carlo Methods</i>	50
<i>9</i>	<i>Kernel Density Estimation</i>	54

1

Introduction and Housekeeping

1.1 Housekeeping

This course will use lectures, problem classes, tutorials, drop-in sessions, computing practicals and feedback sessions. Since this course contains a higher component of course work, we will have **fewer** lectures.

1.1.1 Lecturer information

The lecturer for this module is Dr Lee Fawcett. If you have any questions, comments or feedback on this course I can be contacted via email at lee.fawcett@ncl.ac.uk. More complicated questions about the course are best dealt with face-to-face.

1.1.2 Office hours

The scheduled office hour for this module is Thursday 12–1. I also have the following office hour for MAS2317, so you can drop in then too: Friday 2–3. In general, I am happy to see students anytime (if I'm free), except on Wednesdays and Friday after 3pm.

1.1.3 Lectures

We will have fewer lectures than the other first year modules, but will have more computer labs. I will hand out copies of the lecture notes and practicals in class.

Lectures will also be recorded. You can access the recordings through BB or at the following webpage:

<http://www.mas.ncl.ac.uk/~nlf8/teaching/mas1343/>

The lecture slides will (eventually) be put on BB.

A copy of the notes and practicals can also be download from Blackboard.

1.1.4 Assessment

- End of semester examination worth 60% of your overall mark. In order to pass this module you must get at least 35% in the exam.
- Assessment of course-work worth 40%:
 - A **single** CBA - due week 3;

- Practicals;
- Demonstrations of computer skills at the lab.

Not all practicals are worth the same marks:

Practical	1	2	3	4	5	6
Weight(%)	5	4	9	9	5	5

1.1.5 Computing practicals

During each computer practical, your general computer skills will be assessed. In particular

- Are your files organised in a sensible manner? For example, do you have a directory for MAS1343 and sub-directories for each practical?
- Are your files named in a structured manner?
- Do you use the correct text editor?

To obtain these marks, you must attend at least five computer labs. These marks will form part of your final assessment.

IN THIS COURSE, all our computer practicals are held in the Herschel cluster. However, you can use computers in other parts of the University while doing your coursework. A list of free computers can be found:

<http://www.ncl.ac.uk/iss/teaching/clusters/usage.php>

and the mobile site is:

<https://m.ncl.ac.uk/iss/>

If you use a machine in a cluster and it is missing the necessary software, contact Anthony Youd: a.j.youd@ncl.ac.uk.

1.1.6 Late work policy

It is not possible to extend submission deadlines for coursework in this module and no late work can be accepted. For details of the policy (including procedures in the event of illness etc.) please look at the School web site

<http://www.ncl.ac.uk/maths/students/teaching/homework/>

1.2 What is R?

R is a computer package that is widely used for statistical software development and data analysis.¹ R uses a command line interface, though several graphical user interfaces are available. The system provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, and others) and graphical techniques.

Warning: last year a few students passed the exam, but failed the module due to poor course-work marks.

Table 1.1: Approximate coursework weightings. On average, 1% of coursework will take 1 hour (assuming you are familiar with your lecture notes). CBA 1 is worth 3% of your final mark.

¹ For example Google, IBM, Shell, Thomas Cook, Facebook. The recent NY times article at <http://tinyurl.com/9m5qyh> is worth a read.

Title	Year	Length	Budget	Voting statistics					Movie genre							
				Rating	Votes	r1	...	r10	mpaa	Action	Animation	Comedy	Drama	Documentary	Romance	Short
A.k.a. Cassius	1970	85	-1	5.7	43	4.5	...	14.5	PG	0	0	0	0	1	0	0
AKA	2002	123	-1	6.0	335	24.5	...	1.5	R	0	0	0	1	0	0	0
Alien Vs. Pred	2004	102	45000000	5.4	14651	4.5	...	4.5	PG-13	1	0	0	0	0	0	0
Abandon	2002	99	25000000	4.7	2364	4.5	...	4.5	PG-13	0	0	0	1	0	0	0
Abendland	1999	146	-1	5.0	46	14.5	...	24.5	R	0	0	0	0	0	0	0
Aberration	1997	93	-1	4.8	149	14.5	...	4.5	R	0	0	0	0	0	0	0
Abilene	1999	104	-1	4.9	42	0.0	...	24.5	PG	0	0	0	1	0	0	0
Ablaze	2001	97	-1	3.6	98	24.5	...	14.5	R	1	0	0	1	0	0	0
Abominable Dr	1971	94	-1	6.7	1547	4.5	...	14.5	PG-13	0	0	0	0	0	0	0
About Adam	2000	105	-1	6.4	1303	4.5	...	4.5	R	0	0	1	0	0	1	0

Table 1.2: The first ten rows of the movie data set. **Credit:** This data set was initially constructed by Hadley Wickham at <http://had.co.nz/>.

R is highly extensible through the use of user-submitted libraries for specific functions or specific areas of study. A particular strength of R is its graphical facilities, which produce quality graphs that can include mathematical symbols. Although R is mostly used by statisticians and other practitioners requiring an environment for statistical computation and software development, it can also be used as a general matrix calculation toolbox with comparable benchmark results to many other software packages.

You will use R throughout your degree at Newcastle.

1.2.1 Accessing R

R is installed on all University machines. In this class we will be using Rstudio, which is an R IDE.² Both R and Rstudio are free, so you can install them on your own computer.³ See

<http://www.ncl.ac.uk/maths/students/teaching/installingr/>

for more details.

² Integrated development environment (IDE).

³ Versions of R and Rstudio are available for Windows, Apple Mac and Linux.

1.2.2 Previous computing knowledge

This course is intended to teach you the basics of programming. No previous programming knowledge is assumed. Overall, the programming related aspects probably accounts for 50% of the final mark in this course - there are programming questions in the exam! It is crucial that you come to all the practicals for this course.

1.2.3 Recommended R textbooks

At <http://goo.gl/nzQTK>, I have constructed list of suitable books (with comments) on R programming. Since you will be using R throughout your degree, then it may be worthwhile buying a good R textbook.

1.3 Movie data set

The internet movie database, <http://imdb.com/>, is a website devoted to collecting movie data supplied by studios and fans. It claims to be the biggest movie database on the web and is run by amazon. More about information imdb.com can be found online:

http://imdb.com/help/show_leaf?about

including information about the data collection process

http://imdb.com/help/show_leaf?infosource

IMDB makes their raw data available at <http://uk.imdb.com/interfaces/>.

Movies were selected for inclusion if they had a known length, had been rated by at least one imdb user and had an mpaa rating. The dataset contains the following fields:

- Title. Title of the movie.
- Year. Year of release.
- Budget. Total budget in US dollars. If the budget isn't known, then it is stored as '-1'.
- Length. Length in minutes.
- Rating. Average IMDB user rating.
- Votes. Number of IMDB users who rated this movie.
- r1-10. Percentage(to nearest 5%) of users who rated this movie a 1, ..., 10
- mpaa. MPAA rating.
- Action, Animation, Comedy, Drama, Documentary, Romance, Short. Binary variables representing if movie was classified as belonging to that genre. A movie can belong to more one genre. See for example the film *Ablaze* in Table 1.2.

There are a total of 24 variables and 4847 films. The first few rows are given in Table 1.2. We will use this dataset to illustrate the concepts covered in this class.

This is **only a subset** of the data, the actual data set contains information on over 50,000 movies.

1.4 Module R package

This module has an associated R package. Installing this package is straightforward:

```
> install.packages("mas1343",
+                   repos="http://R-Forge.R-project.org",
+                   type="source")
```

To load the package, use

```
> library(mas1343)
```

2

Introduction to R

In this chapter we will play about with R and learn about the basics.

2.1 A simple R session

At its most basic R, can be used as a calculator, for example for multiplication and subtraction¹

```
> 5*5
[1] 25
> 10.2/6
[1] 1.7
```

¹ The # symbol is used for commenting. We use comments to describe what a piece of code is doing. That way, when we look at the code in a few months/years we can figure out what is going on.

and more ‘advanced’ operations:

```
> 2^3
[1] 8
> exp(1.5)
[1] 4.481689
> log(10)
[1] 2.302585
> 4 %% 3
[1] 1
```

2.1.1 Assignment operations

In the practicals we will use assignment, i.e. `x = 5`. However, computer assignment is different from typical mathematical assignment. For example,

In mathematics, $x = x + 1$ implies that $0 = 1$.

```
> x = 5
> x = x + 1
> x
[1] 6
```

Notice that when we type `x = 5`, R doesn’t display or print any output to the screen.² If we want to see what value has been assigned to the variable, we type `x`. An equivalent way is to surround the expression with brackets.³ For example

```
> (x = 2*x)
[1] 12
```

² Don’t confuse this with R not “doing anything.”

³ In these notes, I will surround expressions with brackets so you can see what R has done.

You can also use the `<-` operator for assignment. This is, for almost all situations, identical to the `=` operator.

2.1.2 Data types

R has a variety of data types:

```
> (v = TRUE)
[1] TRUE
> (w = "fred")
[1] "fred"
> (x = 5.0)
[1] 5
```

and also some “special” data types:

```
> (y = 5/0)
[1] Inf
> (z = y-y)
[1] NaN
```

Another important data type in R is `NA`. This is used to represent missing values. A list of data types is given in Table 2.1.

Type	Example 1	Example 2	Example 3	Example 4
Doubles	2	3.1242	-45.6	4e-10
Logicals/Boolean	TRUE	FALSE		
Characters	“FRED”	“x”	“Male”	“TRUE”
Infinity	Inf	5/0		

Table 2.1: Summary of data types in R.

2.2 The R workspace

Once you create a variable, R stores that variable in memory for reuse. You can view available variables with the `ls()` command:

```
> ls()
[1] "movies"
```

To delete a variable in R, we use the `rm` function. For example,

```
> x = 0
> y = 1
> z = 2
> ls()
[1] "movies" "x"      "y"      "z"
> rm(x)
> ls()
[1] "movies" "y"      "z"
```

We can remove everything in the workspace using `rm(list=ls())`:

```
> rm(list=ls())
```

I would recommend running `rm(list=ls())` at the beginning of each new R session. This stops you relying on previously stored variables and makes your code more portable.

2.3 Vectors

Vectors are the most basic of all data structures, but are used in almost all R code. An R vector contains n values of the same type, where n can be zero. For example

```
> c(0, 1, 2, 3, 4, 5)
[1] 0 1 2 3 4 5
> (my_first_vec = c(0, 1, 2, 3, 4, 5))
[1] 0 1 2 3 4 5
> (my_second_vec = c("Male", "Female", "Male"))
[1] "Male" "Female" "Male"
```

In the above code, we create a vector of **doubles**, in line 3 we assigned the vector to the variable `my_first_vec`. We can create vectors of any data type. For example, `my_second_vec` is a vector of **characters**.

In R, when we type:

```
> x = 5
> y = "Fred"
```

we have actually created a vector of doubles and characters (of length one). There are special functions in R to determine type of a variable:

```
> x = 5
> is.double(x)
[1] TRUE
> is.character(x)
[1] FALSE
> is.vector(x)
[1] TRUE
```

To determine the length of vector in R, we use the `length` function:

```
> length(my_first_vec)
[1] 6
> length(my_second_vec)
[1] 3
```

To create sequences of numbers we use the `seq` command. For example,

```
> (x1 = seq(1, 6))
[1] 1 2 3 4 5 6
> (x2 = seq(-4, 4, by=2))
[1] -4 -2 0 2 4
```

Table 2.2 gives a few more useful R functions.

Command description	Example	Result
Length	<code>length(x)</code>	4
Reverse order	<code>rev(x)</code>	3,5,5,1
Sort	<code>sort(x)</code>	1,3,5,5
Sum	<code>sum(x)</code>	14
Extract unique elements	<code>unique(x)</code>	1,5,3
Indices of particular elements	<code>which(x==5)</code>	2,3

Table 2.2: Useful vector functions. In the above examples, `x = c(1,5,5,3)`. Check the associated R help for further information.

2.3.1 Vector operations

When our data is in a vector structure, we can apply standard operations to the entire vector. For example,

```
> (x = seq(-4, 4))
[1] -4 -3 -2 -1 0 1 2 3 4
> x*x
[1] 16 9 4 1 0 1 4 9 16
> x - 5
[1] -9 -8 -7 -6 -5 -4 -3 -2 -1
> x + x
[1] -8 -6 -4 -2 0 2 4 6 8
```

2.3.2 Extracting elements from vectors

R has a number of useful methods that we use to extract subsets of our data. For example to pick out particular elements:

```
> my_first_vec[2]
[1] 1
> my_second_vec[2:3]
[1] "Female" "Male"
> my_first_vec[4:2]
[1] 3 2 1
```

We can also use other arguments. For example to remove the last entry in the vector, we use the `length` function:

```
> l = length(my_first_vec)
> my_first_vec[1:(l-1)]
[1] 0 1 2 3 4
```

We determine the length of the vector using the `length` function and select particular elements using the `[.]` operator.

2.4 Logical vectors

R supports the logical elements: `TRUE` and `FALSE`. Boolean algebra tells us how to evaluate the truth of compound statements. Table 2.3 gives a summary of R operations and compares them to the notation used in MAS1341. So for example,⁴

```
> A = TRUE
> B = FALSE
> !A
[1] FALSE
> !B
[1] TRUE
> A & B
[1] FALSE
> A | B
[1] TRUE
```

⁴ Read `!A` as **NOT** A.
Read `A & B` as A **AND** B.
Read `A | B` as A **OR** B.

Boolean	A	B	$\bar{A}$	$\bar{B}$	$A \cap B$	$A \cup B$
R	A	B	!A	!B	A & B	A B
	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE

Table 2.3: Truth table for Boolean operations.

2.4.1 Using logicals for sub-setting vectors

We can construct vectors of logical operators and use them to take subsets of vectors. For example,

```
> (logic1 = c(TRUE, FALSE, TRUE, FALSE))
[1] TRUE FALSE TRUE FALSE
> (vec1 = seq(1, 4))
[1] 1 2 3 4
> vec1[logic1]
[1] 1 3
```

2.4.2 Relational operators

When programming it is often necessary to test relations for equality and inequality. To do this in R we use the relation operators. First let's define some variables:

```
> x = 5
> y = 7
```

To test for equality we use ==:

```
> x == 5
[1] TRUE
> x == y
[1] FALSE
```

Similarly, to test for inequality we use !=:

```
> x != 5
[1] FALSE
> y != x
[1] TRUE
```

There are also commands for greater/less than:

```
> y > 6
[1] TRUE
> x >= 5
[1] TRUE
> x <= y
[1] TRUE
```

Table 2.4 gives a summary of the commands.

Operator	Tests for	Example	Result
==	Equality	x == 5	TRUE
!=	Inequality	x != 5	FALSE
<	Less than	x < 5	FALSE
<=	Less or equal	x <= 5	TRUE
>	Greater	x > 5	FALSE
>=	Greater or equal	x >= 5	TRUE

Table 2.4: Summary of R relational operators. The example is for $x = 5$.

WE CAN also apply these techniques to vectors. For example:

```
> (vec2 = seq(0, 10, by=2.5))
[1] 0.0 2.5 5.0 7.5 10.0
> vec2 > 3
[1] FALSE FALSE TRUE TRUE TRUE
> vec2 < 9
[1] TRUE TRUE TRUE TRUE FALSE
> (vec2 > 3) & (vec2 < 9)
[1] FALSE FALSE TRUE TRUE FALSE
```

We can also combine logical operators:

```
> vec2 > 3
[1] FALSE FALSE TRUE TRUE TRUE
> !(vec2 > 3)
[1] TRUE TRUE FALSE FALSE FALSE
```

2.4.3 Vector partitions

We can construct vectors of logical operators and use them to take subsets of vectors. For example,

```
> (logic1 = c(TRUE, FALSE, TRUE, FALSE))
[1] TRUE FALSE TRUE FALSE
> (vec1 = seq(1, 4))
[1] 1 2 3 4
> vec1[logic1]
[1] 1 3
```

```
> vec2
[1] 0.0 2.5 5.0 7.5 10.0
> vec2 > 3 & vec2 < 9
[1] FALSE FALSE TRUE TRUE FALSE
> vec2[vec2 > 3 & vec2 < 9]
[1] 5.0 7.5
```

Using relational operators allows us to extract subsets of data very easily. Consider the movie budgets:⁵

```
> length(Budget)
[1] 4847
```

To select movies where the budget is known, we use the following command:

⁵ To load movie budgets, use the following commands: `library(mas1343); data(Budget); data(Length)`

```
> non_zero_b = Budget[Budget != -1]
> length(non_zero_b)
[1] 1785
```

and to select movies where the movie length is greater than 60 mins but shorter than 90 mins

```
> m_l = Length[Length > 60 & Length < 90]
```

2.5 Data frames

A data frame is a special kind of object. We use data frames for storing and managing data sets that have a rectangular structure. Typically the rows correspond to *cases* and the columns to *variables*. The crucial difference between a data frame and a `matrix` is that all values in a matrix must be of the same type. The next code segment constructs a simple data frame. First, we construct three vectors:

```
> age = c(24, 26, 25, 21)
> sex = c("Male", "Female", "Male", "Female")
> respond = c(TRUE, FALSE, FALSE, FALSE)
```

Then we combine them using the `data.frame` function:

```
> (df1 = data.frame(age=age, gender=sex, respond=respond))
  age gender respond
1  24   Male    TRUE
2  26 Female   FALSE
3  25   Male   FALSE
4  21 Female   FALSE
```

The data frame `df1` has three columns and four rows. Once we put our data into a data frame, then data manipulation is easier. To calculate the dimensions of a data frame we use `dim`:

```
> dim(df1)
[1] 4 3
```

To extract the first column we use square brackets:

```
> df1[, 1]
[1] 24 26 25 21
```

Similarly, we can get the first row

```
> df1[1, ]
  age gender respond
1  24   Male    TRUE
```

The column names are also easily manipulated

```
> colnames(df1)
[1] "age"      "gender"   "respond"
> (colnames(df1) = c("Age", "Sex", "Respond"))
[1] "Age"      "Sex"      "Respond"
```

WHEN WE download the `movies` data set, we automatically create a data frame:

```
> dim(movies)
[1] 4847 24
> colnames(movies)[1:4]
[1] "Title" "Year" "Length" "Budget"
> movies[4061, 1:4]
```

	Title	Year	Length	Budget
4061	Star Wars: Episode I - The Phantom Menace	1999	133	1.15e+08

2.5.1 Subsets of data frames

We can also retrieve subsets from the data frame. For example, if we wanted only female responses, then:

```
> (female_only = df1$Sex=="Female")
[1] FALSE TRUE FALSE TRUE
> (df2 = df1[female_only, ])
  Age  Sex Respond
2  26 Female  FALSE
4  21 Female  FALSE
```

or people 25 and over,

```
> over_25 = df1$Age>=25
> (df3 = df1[over_25, ])
  Age  Sex Respond
2  26 Female  FALSE
3  25  Male  FALSE
```

2.5.2 Example: movie data

We can select movies where the budget is greater than \$100,000

```
> m1 = movies[movies$Budget > 100000,]
> dim(m1)
[1] 1738 24
```

or movies that cost more than \$100,000 but are not R rated:

```
> m2 = movies[movies$Budget > 100000
+ & movies$mpaa != "R",]
> dim(m2)
[1] 727 24
```

or movies that are either PG or PG-13:

```
> m3 = movies[
+ movies$mpaa == "PG" | movies$mpaa == "PG-13",]
> dim(m3)
[1] 1515 24
```

3

Summary Statistics

3.1 Measures of location

3.1.1 Sample mean

One of the most important and widely used measures of location is the (arithmetic) mean:

$$\bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n} = \frac{1}{n} \sum_{i=1}^n x_i .$$

So if our data set was $\{0, 3, 2, 0\}$, then $n = 4$. Hence,

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{0 + 3 + 2 + 0}{4} = 1.25 .$$

3.1.2 Sample median

The sample median is the middle observation when the data is ranked in ascending order. Denote the ranked observations as $x_{(1)}, x_{(2)}, \dots, x_{(n)}$. The sample median is defined as:

$$\text{Sample median} = \begin{cases} x_{(n+1)/2}, & n \text{ odd;} \\ \frac{1}{2}x_{(n/2)} + \frac{1}{2}x_{(n/2+1)}, & n \text{ even .} \end{cases}$$

Remember that $x_{(n+1)/2}$ is the $(n+1)/2^{\text{th}}$ ordered observation.

The median is more robust than the sample mean, but has less useful mathematical properties.

FOR OUR simple data set $\{0, 3, 2, 0\}$, to calculate the median we re-order it to: $\{0, 0, 2, 3\}$; then take the average of the middle two observations, to get 1.

3.1.3 Sample mode

The mode is the value which occurs with the greatest frequency. It only makes sense to calculate or use it with discrete data. In R we use the `table` function to calculate the mode.

Warning: in R the function `mode` doesn't give you the sample mode. Use `table` instead.

3.1.4 Examples

- Calculate the mean and median of $\{1, 7, 13, 14, 20\}$.

Solution

The mean is:

$$\bar{x} = \frac{1}{5}(1 + 7 + 13 + 14 + 20) = 11$$

The median is the middle value, so we get 13.

- Calculate the mean and median of $\{7, 13, 14, 20\}$

Solution

- Calculate the mean and median of $\{1, 7, 13, 14, 200\}$

Solution

3.2 Measures of spread

As well as knowing the location statistics of a data set, we also need to know how variable or ‘spread-out’ our data are.

3.2.1 Range

The range is easy to calculate. It is simply the largest minus the smallest.

$$\text{Range} = x_{(n)} - x_{(1)} .$$

So for our data set of $\{0, 3, 2, 0\}$, the range is $3 - 0 = 3$. It is very useful for data checking purposes, but in general it’s not very robust.

3.2.2 Sample variance and standard deviation

The sample variance, s^2 , is defined as

$$\begin{aligned} s^2 &= \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \\ &= \frac{1}{(n-1)} \left\{ \left(\sum_{i=1}^n x_i^2 \right) - n\bar{x}^2 \right\} . \end{aligned}$$

The second formula is easier for calculations. So for our data set, $\{0, 3, 2, 0\}$, we have

$$\sum_{i=1}^4 x_i^2 = 0^2 + 3^2 + 2^2 + 0^2 = 13$$

so,

$$s^2 = \frac{1}{n-1} \left\{ \left(\sum_{i=1}^n x_i^2 \right) - n\bar{x}^2 \right\} = \frac{1}{3} (13 - 4 \times 1.25^2) = 2.25.$$

The sample standard deviation, s , is the square root of the sample variance, i.e. for our toy example $s = \sqrt{2.25} = 1.5$.

3.2.3 Examples

- Calculate the variance of $\{1, 7, 13, 14, 20\}$.

Solution

We calculate

$$\sum_{i=1}^5 x_i^2 = 1^2 + 7^2 + 13^2 + 14^2 + 20^2 = 815$$

and $n\bar{x}^2 = 605$. So

$$s^2 = \frac{815 - 605}{4} = 52.5 .$$

When you get a new data set, calculating the range is useful when checking for obvious data-inputting errors.

Obviously, the range can be distorted by outliers or extreme observations.

In statistics, the mean and variance are used most often. This is mainly because they have nice mathematical properties, unlike the median, say.

The divisor is $n - 1$ rather than n in order to correct for the bias which occurs because we are measuring deviations from the sample mean rather than the “true” mean of the population we are sampling from - more on this in MAS1341.

The standard deviation is preferred as a summary measure as it is in the units of the original data. However, it is often easier from a theoretical perspective to work with variances.

- Calculate the variance of $\{7, 13, 14, 20\}$.

Solution

- Calculate the variance of $\{1, 7, 13, 14, 200\}$.

Solution

3.2.4 Quartiles and the interquartile range

The upper and lower quartiles are defined as follows:

- Q1: Lower quartile = $(n+1)/4^{\text{th}}$ smallest observation.
- Q3: Upper quartile = $3(n+1)/4^{\text{th}}$ smallest observation.

We can linearly interpolate between adjacent observations if necessary. The interquartile range is the difference between the third and first quartile, i.e.

$$IQR = Q3 - Q1.$$

Examples

For the following data sets, calculate the inter-quartile range:

1. $\{5, 6, 7, 8\}$ ¹ The lower quartile is the $5/4 = 1.25^{\text{th}}$ smallest observation, i.e. $x_{(1.25)}$. The value of $x_{(1)} = 5$ and $x_{(2)} = 6$. So ¹ So $n = 4$.

$$x_{(1.25)} = x_{(1)} + 0.25 \times (x_{(2)} - x_{(1)}) = 5 + \frac{6 - 5}{4} = 5.25.$$

Similarly, the upper quartile is the $3 \times (4 + 1)/4 = 3.75^{\text{th}}$ smallest observation. So

$$x_{(3.75)} = x_{(3)} + 0.75 \times (x_{(4)} - x_{(3)}) = 7.75.$$

Therefore,

$$IQR = 7.75 - 5.25 = 2.5.$$

2. $\{10, 15, 20, 25, 50\}$ ². The lower quartile is the $(5 + 1)/4 = 1.5^{\text{th}}$ smallest observation, i.e. $x_{1.5}$, so ² So $n = 5$

$$x_{(1.5)} = x_{(1)} + 0.5 \times (x_{(2)} - x_{(1)}) = 12.5$$

Similarly, the upper quartile is the $3 \times (5 + 1)/4 = 4.5^{\text{th}}$ smallest observation. So

$$x_{(4.5)} = x_{(4)} + 0.5 \times (x_{(5)} - x_{(4)}) = 37.5.$$

Thus,

$$IQR = 37.5 - 12.5 = 15.$$

3. $\{1, 7, 13, 14, 20\}$.

4. $\{7, 13, 14, 20\}$.

3.3 Using R

FOR THE movie data in section §1.3, we can easily use R to calculate the summary statistics. First we load the data from the `mas1343` package:

```
> library(mas1343)
> data(movies)
```

To calculate location measures, we use the `mean` and `median` functions:

```
> mean(movies$Budget)
[1] 10286893
> median(movies$Budget)
[1] -1
```

Notice that the budget mean and median are substantially different.

WE CAN also calculate measures of spread:

```
> range(movies$Budget)
[1] -1e+00 2e+08
> var(movies$Budget)
[1] 5.308283e+14
> sd(movies$Budge)
[1] 23039711
```

To get the quartiles from R we use the `quantile` command, i.e.

```
> quantile(movies$Rating, type=6)
 0%  25%  50%  75% 100%
1.0  4.6  5.7  6.6  9.1
> summary(movies$Rating)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.000  4.600   5.700   5.523  6.600   9.100
```

Note the default in R gives you a slightly different quartile range, i.e. if you don't enter the `type=6` argument. As $n \rightarrow \infty$, the different quartile functions converge.

Command	Comment	Example
<code>mean</code>	Calculates the mean of a vector	<code>mean(x)</code>
<code>sd</code>	Calculates the standard deviation of a vector	<code>sd(x)</code>
<code>var</code>	Calculates the variance of a vector	<code>var(x)</code>
<code>quantile</code>	The vector quartiles. Make sure you use <code>type=6</code>	<code>quantile(x, type=6)</code>
<code>range</code>	Calculates the vector range	<code>range(x)</code>
<code>summary</code>	Calculates the quartiles	<code>summary(x)</code>
However , it doesn't use <code>type=6</code> quartiles.		

Table 3.1: Summary of R commands in this chapter.

4

Graphical Presentation of Data

4.1 Introduction

Graphical displays of data can be very useful for showing the main features of a data set. The appropriate form of graph depends on the nature of the variables being displayed and what aspects are to be shown. However it should always be borne in mind that the object is to provide a clear and truthful representation of the data, not to distort and not to impress with unnecessary “fancy” features.

4.2 Qualitative data: bar charts

The most useful way to display qualitative data is usually with a bar chart. The length of each bar is proportional to the frequency of the corresponding value of the variable in the sample of data. Note that the widths of the bars should be equal to avoid giving a false impression, as should the width *between* bars.¹

Figure 4.1 shows the breakdown in MPAA ratings for the movie data set. To create a bar chart in R we use the `table` command...

```
> table(movies$mpaa)
NC-17    PG PG-13     R
   16   526   989  3316
```

... inside the `barplot` function:

```
> barplot(table(movies$mpaa), xlab="MPAA Rating",
+          ylab="Frequency", border = "black",
+          col="mistyrose")
```

Remember to load the data first!

4.3 Histograms

To represent the distribution of a sample of values of a continuous variable we can use a histogram. The range of values of the variable is divided into intervals, known as *classes*, and the frequencies in classes are represented by columns. As the variable is continuous, there are no gaps between neighbouring columns, unlike a bar chart.² Note also that, strictly speaking, it is the *area* of the column which is proportional to

¹ Type `colours()` into R to get a list of colours.

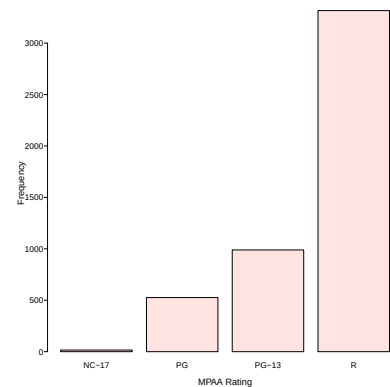


Figure 4.1: Barchart of the mpaa ratings for 4847 films.

² Unless, of course, a particular class has zero frequency

Rule	Formula	R command	Comment
Sturges'	$k_{ST} = \lceil \log_2 n + 1 \rceil$	Default	Tends not to be very good for $n > 30$.
Scott's	$h_{SC} = 3.49 \times s \times n^{-1/3}$	breaks = "Scott"	
Freedman-Diaconis	$h_{FR} = 2 \times IQR(x) \times n^{-1/3}$	breaks = "FD"	When the distribution is symmetric, this is very similar to Scott's rule.

the frequency, not the height. The reason for this is that columns need not be of the same width. Computer software tends to use columns of the same width. However, this default can be overridden in R if you really want to do this.

Figure 4.2 shows histograms of the film budgets. When dealing with densities (relative frequency), we can easily work out the height using this formula:

$$\text{Height} = \frac{\text{frequency}}{n \times \text{Bin-width}}.$$

When the y -axis is labelled with density or relative frequencies, the area under the histogram is one. Bin widths should be chosen so that you get a good idea of the distribution of the data, without being swamped by random variation.

To generate Figure 4.2 in R we use the following commands:

```
> hist(movies$Budget, col="grey",
+       main="Mean film budget",
+       freq=FALSE, xlab="Budget ($)")
```

4.3.1 How many bins should we have?

First we will define the notation we will use:

- n : the sample size;
- k : the number of bins in the histogram;
- h : the bin-width.

Then the number of bins we will use to construct a histogram is:

$$k = \left\lceil \frac{\max(x) - \min(x)}{h} \right\rceil \quad (4.1)$$

where $\lceil \cdot \rceil$ is the ceiling function.

Table 4.1 gives a summary of the different rules. Briefly, R uses Sturges' rule by default, which isn't always that good. Notice that Sturges' rule gives you k , but the other rules give you the bin width. Typically, it is best to go with Scott's rule or the Freedman-Diaconis rule.

To use the different rules, we use the breaks argument. For example, the following piece of code:

```
> hist(movies$Rating, col="wheat",
+       main="Mean user rating",
+       freq=TRUE, xlim=c(1,10),
+       xlab="Rating", breaks="FD")
```

Table 4.1: Standard bin width rules in R.

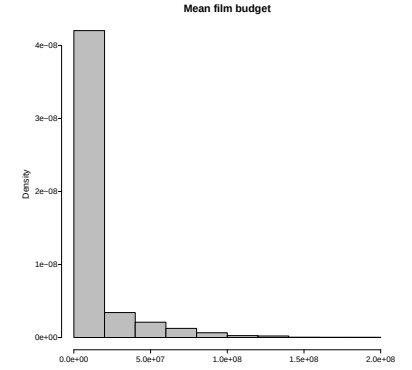


Figure 4.2: Histogram of film ratings and budgets.

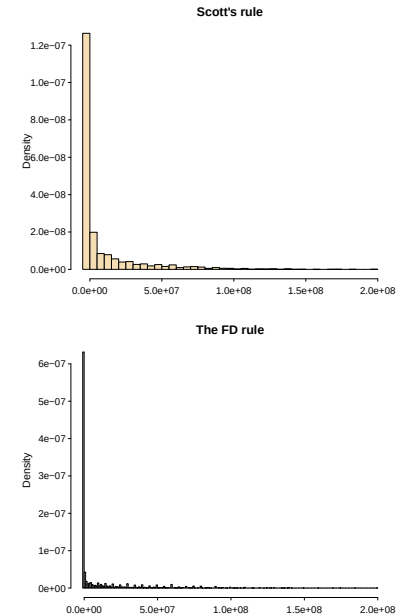


Figure 4.3: Histogram of (a) movie ratings and (b) movie budgets using the Freedman-Diaconis rule. Compare these histograms to Figure 4.2 which uses Sturges' rule (default).

uses the Freedman-Diaconis rule - see Figure 4.3. When we compare Figure 4.3a with 4.2, we have used many more bins, which results in a *better* histogram; however, it is clear that Figure 4.3b uses too *many* bins! In practical 2, you will get a chance to experiment with the different rules yourself.

4.3.2 Example: movie data

Variable	Range	s	IQR	Number of bins		
				Sturges	Scott's	Free-Dia
Budget	2×10^8	23039711	8×10^6	14	43	212
Length	249.0	17.3	17.0	14	70	124
Rating	8.1	1.5	2.0	14	28	35

Table 4.2: Bin sizes for the movie data.
In all cases $n = 4847$.

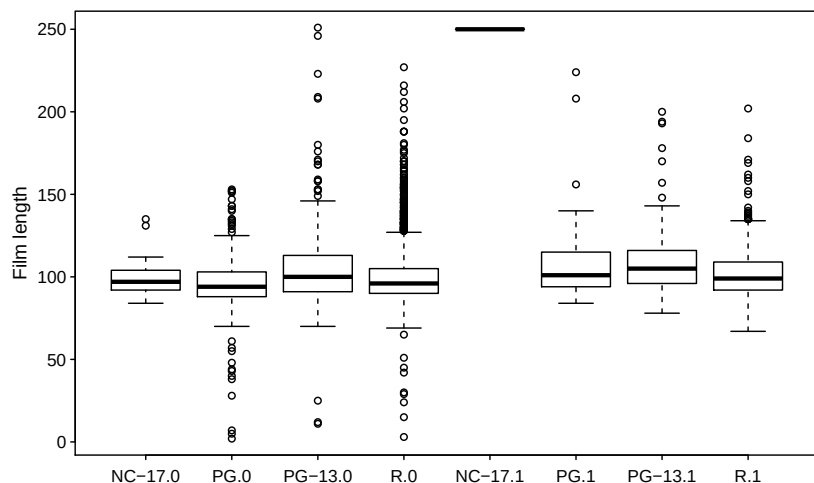


Figure 4.4: Box and whisker plots movie length split according to mpaa rating and whether the film was a romance.

4.4 Box and whisker plots

A box and whisker plot, often referred to simply as a boxplot, is another way to represent continuous data. This kind of plot is particularly useful for comparing two or more groups, by placing the boxplots side-by-side. Figure 4.4 and figure 4.5 shows boxplots of film length for different categories of film.

The central bar in the “box” is the sample *median*. The top and bottom of the box represent the upper and lower sample *quartiles*, respectively. Just as the median represents the 50% point of the data, the lower and upper quartiles represent the 25% and 75% points respectively.

The lower whisker is drawn from the lower end of the box to the smallest value that is no smaller than $1.5IQR$ below the lower quartile. Similarly, the upper whisker is drawn from the middle of the upper end of the box to the largest value that is no larger than $1.5IQR$ above the upper quartile. Points outside the whiskers are classified as outliers.

To do this in R we use the following commands:

```
> par(mfrow=c(2, 1))
> boxplot(movies$Length, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa + movies$Romance,
+         ylab="Film length")
```

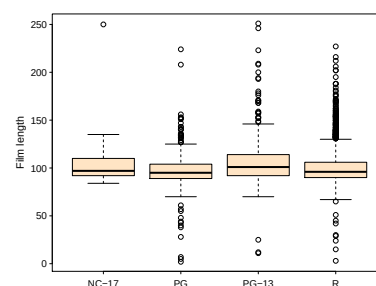
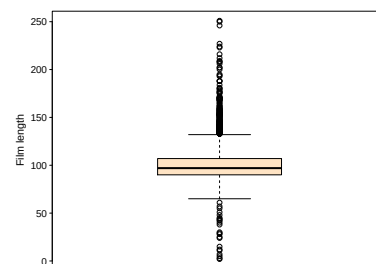


Figure 4.5: Box and whisker plots of (a) film length (b) film length split according to the mpaa rating.

4.4.1 Boxplot example 1

For the data set,

0.1	0.1	0.2	0.4	0.4	0.8	0.8	0.8
0.9	0.9	1.0	1.4	1.6	2.0	2.4	3.5

construct a boxplot.

Solution

First we calculate the median and quartiles

Median	1 st quartile	3 rd quartile	IQR
0.85	0.4	1.55	1.15

Table 4.3: An example data set.

Table 4.4: Summary statistics for the first example data in Table 4.3.

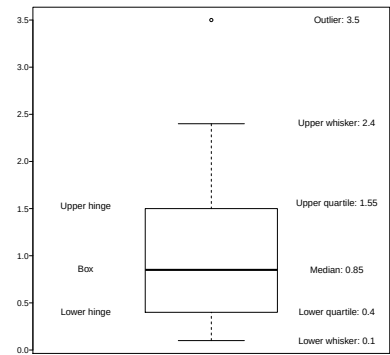


Figure 4.6: Boxplot for the first example data set in table 4.3 and summary statistics in table 4.4.

4.4.2 Boxplot example 2

For the following data:

9.0	32.8	33.0	34.9	35.4	39.7	41.6	42.0
42.3	43.2	46.9	49.2	51.6	51.7	55.0	81.0

Table 4.5: An example data set.

construct a boxplot.

Solution

First we calculate the necessary summary statistics:

Median	1 st quartile	3 rd quartile	IQR	W_L	W_U
42.1	35.02	51.00	15.98		

Table 4.6: Summary statistics for the second example data in table 4.5.

Command	Comment	Example
<code>table</code>	Contingency table	<code>table(x)</code>
<code>barplot</code>	Generate a bar chart	<code>barplot(table(x))</code>
<code>hist</code>	Histogram	<code>hist</code>
<code>plot</code>	Scatter plot. See Practical 2.	<code>plot(x, y)</code>
<code>points</code>	Add points to a plot. See Practical 2.	<code>points(x,y)</code>
<code>lines</code>	Add lines to a plot. See Practical 2.	<code>lines(x,y)</code>
<code>boxplot</code>	Box and whiskers plot	<code>boxplot(x)</code>

Table 4.7: Summary of R commands in this chapter.

5

Control Statements and Functions

5.1 Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or *arguments*) and return a *single* value. Let's look at some simple functions.

5.1.1 Basic functions

This function takes in a single argument x and returns x^2 :

```
> Fun1 = function(x) {  
+   return (x*x)  
+ }
```

The key elements in the function call are:

- The word **function**;
- The brackets **()** which enclose the **argument** list. This list may be empty.
- A sequence of statements in curly braces **{ }**.
- A **return** statement.

We *call* Fun1 in the following manner:

```
> Fun1(5)  
[1] 25  
> y = Fun1(10)  
> y  
[1] 100  
> z = c(1, 2, 3, 4)  
> Fun1(z)  
[1] 1 4 9 16
```

Of course, the old saying ‘Garbage in, Garbage out’ is true:

```
#Incorrect function calls.  
> Fun1()  
Error in Fun1() : argument "x" is missing, with no default  
> Fun1("5")  
Error in x * x : non-numeric argument to binary operator
```

The error messages give you an idea of what went wrong. Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
> Fun2()
[1] 1
> Fun2(4)
[1] 16
> Fun3 = function(x, y) {
+   return (x*y)
+ }
> Fun3(3, 4)
[1] 12
```

5.1.2 A more useful function

Here the function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
> Investigate = function(values) {
+   hist(values)
+   m_std = c(mean(values), sd(values))
+   return(m_std)
+ }
```

Once we have created our function, we can put it to good use¹:

```
> Investigate(movies$Rating)
[1] 5.522715 1.451864
```

¹ Obviously, a histogram would also be created – it's just not shown here

5.1.3 Variable scope

When we call a function, R first looks for *local* variables, then *global* variables. For example, `Fun4` uses a global variable:

```
> blob = 5
> Fun4 = function() {
+   return(blob)
+ }
> Fun4()
[1] 5
```

R scoping rules are actually a bit more complicated than described below. R uses something called *lexical* scope, but this doesn't affect us.

However, in `Fun5`, we use a local variable:

```
> Fun5 = function() {
+   blob = 6
+   return(blob)
+ }
> Fun5()
[1] 6
> blob
[1] 5
```

As a general rule, functions should only use **local** variables. This makes your code more portable and less likely to have bugs.

5.2 The *cat* command

A useful function to help debugging is the `cat` function. This function is used to print messages to the screen. For example,

```
> x = 5
> cat(x, "\n")
5
> (y = cat(x, "\n"))
5
NULL
```

We will use the `cat` function in the next section.

5.3 Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to **TRUE** or **FALSE**. They are used in almost all computer programs.

5.3.1 If statements

The basic structure of an **if** statement is:

```
> if(expr) {
+   ##do something
+ }
```

where `expr` is evaluated to be either **TRUE** or **FALSE**. The following example illustrates **if** statements in R:

```
> x = 5
> y = 5
> if(x<5) {
+   y = 0
+ }
> y
[1] 5
```

In this code chunk, `x < 5` evaluates to be **FALSE** so the following brackets are not evaluated. We test for greater than in a similar manner:

```
> x = 5
> y = 5
> if(x > 0) {
+   y = 0
+ }
> y
[1] 0
```

Here $x > 0$ evaluates to be TRUE so, y is set equal to 0. If we wanted to test for equality with zero, then we would use `==`.

5.3.2 If else statements

We can link together a number of if statements

```
> x = 0
> if(x > 0) {
+   cat("x is greater than zero")
+ } else if(x < 0) {
+   cat("x is less than zero")
+ } else {
+   cat("x must be zero!")
+   cat("\n")
+ }
x must be zero!
```

The final `else` is optional. We can also use `if` statements in functions, for example to check that our data is negative we can construct the following function:

```
> IsNegative = function(value) {
+   is_pos = FALSE
+   if(value < 0) {
+     is_pos = TRUE
+   }
+   return(is_pos)
+ }
> IsNegative(1)
[1] FALSE
> IsNegative(-5.6)
[1] TRUE
```

A more sophisticated function could be:

```
> IsGreaterThan = function(value1, value2) {
+   is_greater_than = FALSE
+   if(value1 > value2) {
+     is_greater_than = TRUE
+   }
+   return(is_greater_than)
+ }
```

Which we can then call:

```
> IsGreaterThan(-5, -6)
[1] TRUE
> IsGreaterThan(10, 10)
[1] FALSE
```

5.4 Control statements

At times we would like to perform some operation on a vector or a data frame. Often R has built-in functions that will do this for you, e.g. `mean`, `sd`,... Other times we have to write our own functions. For example, suppose we want to calculate $\sum_{i=1}^{10} i^2$. In R we can use a `for` loop:

```
> x = 0
> for(i in 1:10) {
+   x = x + i^2
+ }
> x
[1] 385
```

Or perhaps $\sum_{j=-5}^{-1} e^j / j^2$, then:

```
> total = 0
> for(j in -5:-1) {
+   total = total + exp(j)/j^2
+ }
> total
[1] 0.4086594
```

A more tricky example is, calculate $\sum e^k / k^2$, for $k = 3, 6, 9, \dots, 21$:

```
> total = 0
> for(i in 1:7) {
+   k = i*3
+   total = total + exp(k)/k^2
+ }
> total
[1] 3208939
```

Exercise: Using the inbuilt R function `sum`, calculate the above summations without using `for` loops.

5.5 Putting it all together

Rather than have to constantly write R code to solve the summations in §5.4 we can create a function to solve the general form:

$$\sum_{i=i_s}^{i_e} \frac{e^i}{i^2} \quad \text{for } i = i_s, i_s + j, i_s + 2j, \dots, i_e.$$

So in R we have:

```

> Summation1 = function(i_s, i_e, j) {
+   total = 0
+   for(i in 1:(i_e/j)) {
+     k = i*j
+     total = total + exp(k)/k^2
+   }
+   return(total)
+ }
> Summation1(-5, -1, 1)
[1] Inf
> Summation1(3, 21, 3)
[1] 3208939

```

5.6 The *apply* family

R has been designed with manipulating data in mind. Due to this, there are two important functions that are unique to R.

Probably not unique, but not common in other programming languages.

5.6.1 The *apply* function

We use the `apply` function when we want to *apply* the same function to every row or column of a data frame. For example, suppose we have a data frame with three columns:

```

> (df4 = data.frame(c1 = 1:4, c2 = 4:7, c3 = 2:5))
  c1 c2 c3
1  1  4  2
2  2  5  3
3  3  6  4
4  4  7  5

```

The `apply` function takes (at least) three arguments. The first argument is the data frame, the second the number 1 or 2 indicating row or column and the third a function to apply to each row or column. So

```

> apply(df4, 1, mean)
[1] 2.333333 3.333333 4.333333 5.333333

```

calculates the mean value of every row, while

```

> apply(df4, 2, sd)
      c1      c2      c3
1.290994 1.290994 1.290994

```

calculates the standard deviation of every column.

Suppose one of the columns was non-numeric

```

> (df5 = data.frame(c1 = 1:3, c2 = 4:6,
+   c3 = LETTERS[1:3]))
  c1 c2 c3
1  1  4  A
2  2  5  B
3  3  6  C

```

then taking the mean doesn't really make sense:

```
> apply(df5, 1, mean)
[1] NA NA NA
```

Instead, we remove the column, then calculate the mean:

```
> apply(df5[, 1:2], 1, mean)
[1] 2.5 3.5 4.5
```

5.6.2 The *tapply* function

The function `tapply` is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
> tapply(movies$Length, movies$mpaa, mean)
      NC-17      PG      PG-13      R 
110.18750  97.23384 104.97877 100.18818
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110mins. With `tapply` we can do very interesting things. For example, in the next piece of code, we plot the average movie rating conditional on its length:

```
> tapply(movies$Length, movies$Rating, mean)[1:6]
      1      1.2    1.3    1.4    1.5    1.6 
85.5 93.0 87.5 85.0 67.0 86.0 
> rating_by_len = tapply(movies$Length, movies$Rating,
+   mean)
> plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce Figure 5.1 in Excel!

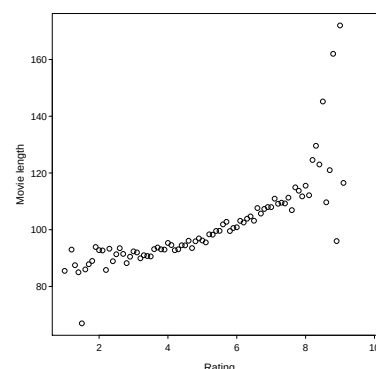


Figure 5.1: Plot of movie length conditional on its rating.

5.7 Help

R has a very good help system. If you need information about a particular function – say `plot` – then typing `?plot` in a R terminal will bring up the associated help page.

The internet is another very good source of R help. Unfortunately, using Google isn't particularly useful since the letter “R” appears on most web pages! However, you can use

<http://www.rseek.org/>

Using this search engine limits searches to R web-pages.

Summary of R commands

Command	Comment
<code>for</code>	A for loop. See §5.4
<code>if</code> or <code>else</code>	A conditional statement. See §5.3.
<code>function</code>	An R function constructor. See §5.1
<code>cat</code>	Print command. See §5.2
<code>apply</code> or <code>tapply</code>	See §5.6

Table 5.1: Summary of R commands in this chapter.

6

Random Number Generation

6.1 Randomness: quantifying uncertainty

The concepts of uncertainty and randomness have intrigued humanity for a long time. The world around us is not deterministic and we are faced continually with chance occurrences. Uncertainty is inherent in nature; for example, the behaviour of fundamental physical particles, genes and chromosomes in biology, and individuals in society under stress or strain. The methodology for exploring uncertainty involves the use of *random numbers*.

6.2 Pseudo-random numbers

Suppose we need to obtain a list of random digits $0, 1, 2, \dots, 9$. How might we go about this? There are several options:

- **Fair ten-sided die:**

If the sides are labelled from 0 to 9 then tosses of this die will yield the required digits.

- **Other physical devices:**

It is possible to construct more complicated mechanical devices than a coin or a die, such as ‘wheels of fortune’, or physical devices based on gamma rays emitted from radioactive elements;

A disadvantage of the mechanical and electronic devices is that the sequence of random numbers generated is not reproducible. So we use *Pseudo-random numbers generators* (rng), that is an algorithm for generating a sequence of numbers that approximates the properties of random numbers.

The German Federal Office for Information Security (BSI) has established criteria for quality rng:

1. A sequence of random numbers has a high probability of containing no identical consecutive elements.

2. A sequence of numbers which is indistinguishable from ‘true random’ numbers (tested using statistical tests).
3. It should be impossible to calculate, or guess, from any given subsequence, any previous or future values in the sequence.
4. It should be impossible, for all practical purposes, for an attacker to calculate, or guess the values used in the random number algorithm.

http://en.wikipedia.org/wiki/Statistical_randomness

Points 3 and 4 are crucial for many applications¹.

¹ Everytime you make a phone call, contact to a wireless point, pay using your credit card random numbers are used.

As we will see in this course, it is important to be able to reproduce a sequence of random numbers. The advent of digital computers gave rise to a number of techniques that use a *recursive relation*, that is the number r_i in a sequence is a function of the preceding numbers $r_{i-1}, r_{i-2}, \dots$.

6.3 Congruential generators

Consider the set $\mathbb{N}^0$ of non-negative integers. That is, $\mathbb{N}^0 = 0, 1, 2, \dots$. Let ‘mod’ represent the *modulo* operation, so that for $x, m \in \mathbb{N}^0, x \neq 0$, $(x) \bmod m$ means that x is divided by m and the remainder is taken as the result.

6.3.1 Example: modulo operations

1. What is $13 \bmod 4$? Answer =
2. What is $19 \bmod 5$? Answer =
3. What is $2008 \bmod 3$? Answer =
4. What is $10,008 \bmod 11$? Answer =

Now consider the relation

$$r_i = (ar_{i-1} + b) \bmod m, \quad i = 1, 2, \dots, m \quad (6.1)$$

where r_0 is the initial number, known as the *seed*, and $a, b, m \in \mathbb{N}^0$ are the *multiplier*, *additive constant* and *modulo* respectively.

The modulo operation means that at most m different numbers can be generated before the sequence must repeat – namely the integers $0, 1, 2, \dots, m-1$. The actual number of generated numbers is $h \leq m$, called the *period* of the generator.

6.3.2 Example: Congruential generators

Selecting $a = 17$, $b = 0$, $m = 100$, $r_0 = 13$ in relation (6.1) generates the following sequence:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
r_i	13	21	57			41			33	61	37	29	93	81	77			1	17	

The next value, r_{20} , is found to be 13 so that this sequence then repeats. Thus, this sequence has period 20.

6.3.3 Example: Bad random number generators

In the 1970's a popular random generator used was RANDU, where $M = 2^{31}$, $a = 65539$ and $b = 0$. Unfortunately this is a spectacularly bad choice of parameters. On noting that $a = 65,539 = 2^{16} + 3$, then

$$r_{i+1} = ar_i = 65539 \times r_i = (2^{16} + 3)r_i.$$

So

$$r_{i+2} = a r_{i+1} = (2^{16} + 3) \times r_{i+1} = (2^{16} + 3)^2 r_i.$$

On expanding the square, we get

$$r_{i+2} = (2^{32} + 6 \times 2^{16} + 9)r_i = [6(2^{16} + 3) - 9]r_i = 6r_{i+1} - 9r_i.$$

Note: all these calculations should be to the mod 2^{31} . So there is a large correlation between the three points! What does this mean in practice, well let's consider triplets from this random generator and compare them to R's standard random number generator.

The RANDU demo simulates 1000 RNs from RANDU and from R's standard random number generator. It then plots the results.

```
> require(mas1343)
> demo(RANDU)
```

This gives Figure 6.1.

Today's random number generators a (bit) more sophisticated than what we come across on this course. Newer methods also have quite cool names: **Mersenne-Twister**² and **Super-Duper**.

Computer generation of pseudo-random numbers - remarks

1. As computers essentially use numbers to base 2, generators generally use $m = 2^k$, where k is a very large number ($k \in \mathbb{N}$).
2. We want the period of the sequence to be as large as possible.

6.4 Assessing random number generators

Theorem

For the relation

(6.2)

the maximum period, m , is achieved for $b > 0$ if, and only if:

- (i)
- (ii)
- (iii)

Proof: Beyond the scope of this course.

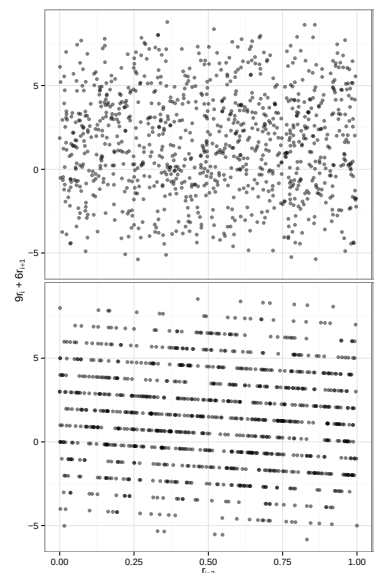


Figure 6.1: Comparison of the Randu algorithm and a standard R algorithm

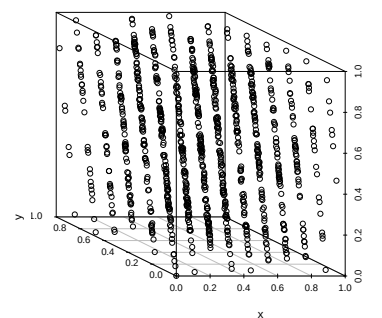


Figure 6.2: 3d scatterplot of Randu triples.

² http://en.wikipedia.org/wiki/Mersenne_twister

Remarks

1. If $m = 2^k$, then if $a = 4c + 1$ for some positive integer c , (ii) and (iii) will hold.
2. Similarly, for (i) to be true then b must be a positive odd integer if $m = 2^k$.
3. As a real example, the Numerical Algorithms Group (NAG) **Fortran** library uses $k = 59$, $b = 0$ and $a = 13^{13}$ in one its random number generator.

6.4.1 Example: Maximum periods of rng

Check to see if the maximum period can be achieved if the Congruential method with the following parameters is used to generate a sequence of pseudo-random numbers:

1. **a=16, b=5, m=20**
 - All three conditions must be satisfied for the maximum period to be obtained, so we check each in turn. Condition (i): False. $b = 5$ and $m = 20$ have a common factor, 5. Hence the maximum period is *not* achieved.
2. **a=16, b=3, m=20**
 - Condition (i): True. b and m have no common factors other than 1.
 - Condition (ii): False. $(a - 1) = 15$, and this is *not* divisible by 2. But 20 *is* divisible by 2.
 - Hence the maximum period is *not* achieved.
3. **a=11, b=3, m=20**
 - Condition (i):
 - Condition (ii):
 - Condition (iii):
 -
4. **a=21, b=3, m=20**
 - Condition (i):

- Condition (ii):
- Condition (iii):
-

6.5 Random numbers in R

6.5.1 The `runif` function

R has a number of commands that generate and manipulate random numbers. One function that we use throughout this course is.

The default random number generator used by R is the Mersenne-Twister

```
> runif(n, min=0, max=1)
```

This function will generate `n` random numbers between the values of `min` and `max`. If the arguments `min` or `max` are omitted, then the default values are 0 and 1 respectively. For example,

```
> runif(1)
[1] 0.528087
> runif(1)
[1] 0.9433197
> runif(5)
[1] 0.7552255 0.5012424 0.7879546 0.8537302 0.4291212
> runif(1, 6, 7)
[1] 6.759506
```

Notice that calling `runif` returns different values. If we wish to rerun a computer experiment, then we need repeatability. In this case we use the command `set.seed`, e.g.

The function `set.seed` is a user friendly version that sets the seeds of all possible underlying random number generators. This function is actually an interface to `.Random.seed`. Don't worry about this.

```
> set.seed(12345)
> runif(1)
[1] 0.7209039
> runif(1)
[1] 0.8757732
> set.seed(12345)
> runif(1)
[1] 0.7209039
```

If we want to generate integers, say 0, 1, ..., 9, then we could simply take the first value after the decimal place.

6.5.2 The `sample` function

Another important R function that we will use is the `sample` function:

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

This takes the following arguments

- **x**: a list of values
- **size**: non-negative integer giving the number of items to choose.
- **replace**: Should sampling be with replacement? Default: **FALSE**.
- **prob**: A vector of probability weights. Default: All values equally likely.

See `?sample` for help.

Example usage of `sample`

Suppose we wish to sample five numbers from $\{1, 2, 3, 4, 5, 6\}$, then

```
> set.seed(1)
> x = c(1, 2, 3, 4, 5, 6)
> sample(x, 5)
[1] 2 6 3 4 1
```

In `sample` the default is without replacement, i.e. no value more than once

We can also sample with replacement

```
> sample(x, 5, replace=TRUE)
[1] 6 6 4 4 1
```

This means that values **may** appear more than once

6.5.3 Simulating the World Cup draw

We are in the semi-finals of the World cup³, and need to organise the draw for the final stage. The remaining teams are:

³ Unfortunately, England didn't make it through. Shame

Scotland, Spain, Brazil, France.

Here's how we do this in R

```
> set.seed(2)
> teams = c("Scotland", "Spain", "Brazil", "France")
> sample(teams, 4)
[1] "Scotland" "Brazil"    "Spain"    "France"
```

So we have 'Scotland vs Brazil' and 'Spain vs France'. However, Scotland will likely get beaten by Brazil, so let's rig the voting:

```
> prob_weights = c(0.4, 0.4, 0.05, 0.2)
> sample(teams, 4, prob=prob_weights)
[1] "France"    "Brazil"    "Spain"     "Scotland"
```

That's better!

Summary of R commands

Command	Comment	Example
<code>sample</code>	Sample discrete numbers	<code>sample(c(1,2,3))</code>
<code>runif</code>	Generate a random number between 0 & 1	<code>runif(1)</code>
<code>set.seed</code>	Set the seed of the random number generator	<code>set.seed(10)</code>

Table 6.1: Summary of R commands in this chapter.

7

Simulating Discrete Random Numbers

7.1 Simulating a Bernoulli random variable

The Bernoulli random variable $I \sim \text{Bern}(p)$ has already been encountered in MAS1341. It is perhaps the simplest random variable and has the probability mass function

i	0	1
$\Pr[I = i]$		

To simulate such a quantity, we generate an observation u from a uniform $U(0,1)$ distribution and set

7.1.1 Example: Bernoulli random numbers

Suppose we generate a number from a uniform $U(0,1)$ distribution:

0.332, 0.739, 0.653, 0.110, 0.587, 0.144

and we wish to use these to simulate six independent observations from I , a $\text{Bern}(0.8)$ random variable. Here $p = 0.8$, so we convert using:

We've not actually said how we'd generate a $U(0,1)$ number yet. We'll cover this in §8.1. The reason for the delay is for MAS1341 to cover sufficient material.

to obtain the sequence 1, 1, 1, 0, 1, 0 as our Bernoulli random sample.

7.1.2 Using R to simulate a Bernoulli R.V.

To simulate a Bernoulli variable using R is straightforward. The easiest way is just to use the sample command:

```
> p = 0.5
> sample(c(0, 1), 1, prob=c(1-p, p), replace=TRUE)
[1] 0
> sample(c(0, 1), 10, prob=c(1-p, p), replace=TRUE)
[1] 1 1 1 0 1 0 0 1 1 1
```

7.2 *Simulation of discrete random variables*

You would use this technique described in this section to simulate from the Poisson and binomial distributions. Hint: assignment 1.

7.2.1 Example: discrete random numbers

Simulate a random variable with the following probability mass function:

x	1	2	3	4
$\Pr[X = x]$	0.2	0.15	0.25	0.4

Solution

We calculate the CDF of the distribution

x	1	2	3	4
$\Pr[X = x]$	0.2	0.15	0.25	0.4
$\Pr[X \leq x]$	0.2	0.35	0.60	1.0

Then we generate an observation u from $U \sim \text{Uniform}(0,1)$, so

- if $u < 0.2$, set $X = 1$;
- if $0.2 \leq u < (0.2 + 0.15) = 0.35$, set $X = 2$;
- if $0.35 \leq u < (0.2 + 0.15 + 0.25) = 0.6$, set $X = 3$;
- if $u \geq 0.6$, set $X = 4$.

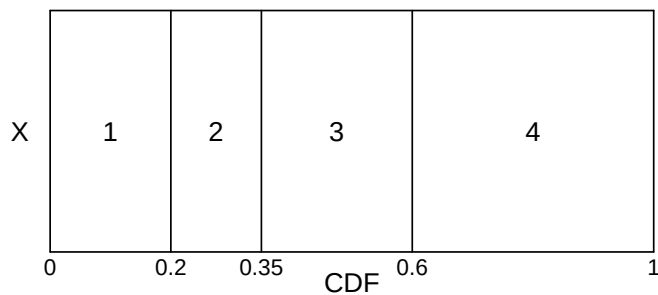


Figure 7.1: Simulating discrete random numbers.

Using R

To simulate the above distribution in R, there are two (obvious) methods that we can use: the `sample` command

```
> x = c(1, 2, 3, 4)
> prob = c(0.2, 0.15, 0.25, 0.4)
> sum(prob)
[1] 1
> sample(x, 1, prob, replace=TRUE)
[1] 2
```

or use a bunch of `if` statements:

```

> u = runif(1)
> if(u <= 0.2) {
+   X = 1
+ } else if(u <= (0.2+0.15)) {
+   X = 2
+ } else if(u <= (0.2+0.15+0.25)) {
+   X = 3
+ } else {
+   X = 4
+ }

```

7.2.2 Simulating a Poisson random variable

For the uniform random numbers,

0.253, 0.588, 0.789

simulate three random numbers from a Poisson distribution with mean $\lambda = 2$.

Solution

The pdf of the poisson distribution is

$$\Pr[X = x] = \frac{e^{-\lambda} \lambda^x}{x!}$$

where $\lambda > 0$. So we have

x	0	1	2	3	4
$\Pr[X = x]$	0.135	0.271	0.271	0.180	0.090
$\Pr[X \leq x]$	0.135	0.406	0.677	0.857	0.947

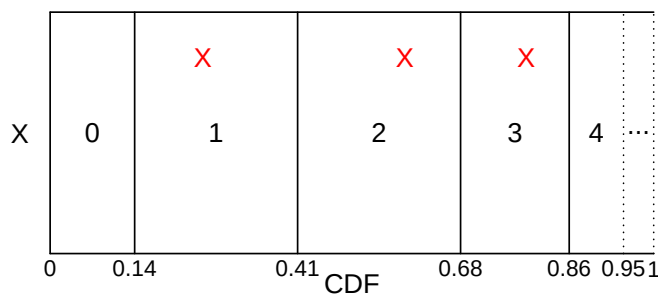


Figure 7.2: Simulating discrete random numbers from the Poisson distribution.

Hence our random numbers are 1, 2 and 3.

7.3 Simulating a Geometric random variable

Let $X \sim \text{Geom}(p)$, so that X takes the random values $1, 2, 3, \dots$ with probabilities $p, (1-p)p, (1-p)^2p, \dots$. Hence,

$$\Pr[X = k] = p(1-p)^{k-1}.$$

The technique described in this section only works for the geometric distribution!

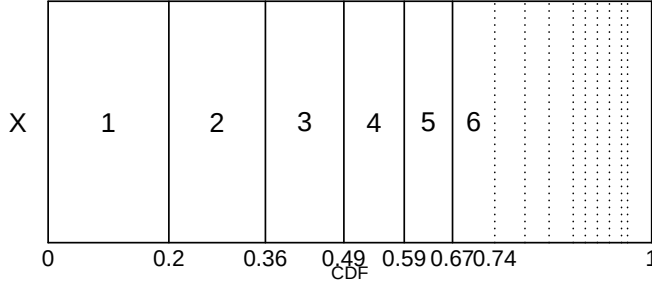


Figure 7.3: Diagram illustrating on simulating discrete random numbers from the Geometric ($p=0.2$) distribution.

The Geometric distribution is the distribution of the number of independent Bernoulli trials until the first success is encountered. For each individual trial, the probability of success is p . We know from §7.2 that we will obtain the value $X = k$ (say) when simulating an observation from X using a value of U if

$$\Pr[X = 1] + \Pr[X = 2] + \dots + \Pr[X = k - 1] \leq U < \Pr[X = 1] + \dots + \Pr[X = k],$$

i.e.

$$\sum_{j=1}^{k-1} \Pr[X = j] \leq U < \sum_{j=1}^k \Pr[X = j],$$

i.e.

$$\sum_{j=1}^{k-1} (1-p)^{j-1} p \leq U < \sum_{j=1}^k (1-p)^{j-1} p. \quad (7.1)$$

We know (for $0 < r < 1$) that

$$r \sum_{k=0}^n (1-r)^k = 1 - (1-r)^{n+1}$$

i.e. a Geometric progression. So

$$\begin{aligned} \sum_{j=1}^{k-1} (1-p)^{j-1} p &= p \sum_{j=1}^{k-1} (1-p)^{j-1} \\ &= p \sum_{j=0}^{k-2} (1-p)^j \\ &= 1 - (1-p)^{k-1}. \end{aligned}$$

Hence for expression 7.1 we have

$$1 - (1-p)^{k-1} \leq U < 1 - (1-p)^k,$$

i.e.

$$(1-p)^{k-1} \geq 1-U > (1-p)^k,$$

i.e.

$$(k-1) \ln(1-p) \geq \ln(1-U) > k \ln(1-p),$$

i.e.

R's Geometric distribution: R defines the geometric distribution slightly different than how we do: $\Pr[X = k] = p(1-p)^x$. This means that when simulating geometric random numbers from R we have to add 1.

$$k - 1 \leq \frac{\ln(1 - U)}{\ln(1 - p)} < k,$$

with the inequality sign reversing since $\ln(1 - p) < 0$. Thus, we observe $X = k$ if

$$1. \quad X > \frac{\ln(1 - U)}{\ln(1 - p)}$$

$$2. \quad X - 1 \leq \frac{\ln(1 - U)}{\ln(1 - p)}, \text{ i.e. } X \leq 1 + \frac{\ln(1 - U)}{\ln(1 - p)}.$$

Both (1) and (2) are satisfied by $X = 1 + \left\lfloor \frac{\ln(1 - U)}{\ln(1 - p)} \right\rfloor$.

The symbol $\lfloor \cdot \rfloor$ denotes “the integer part” (e.g. $\lfloor 3.246 \rfloor = 3$, $\lfloor 5.685 \rfloor = 5$). In R, we use the `floor` function.

7.3.1 Examples: geometric random numbers

- Given $u = 0.2179$, simulate a $\text{Geom}(0.2)$ random variable X .

Solution

$$\begin{aligned} X &= 1 + \left\lfloor \frac{\ln(1 - U)}{\ln(1 - p)} \right\rfloor \\ &= 1 + \left\lfloor \frac{\ln(1 - 0.2179)}{\ln(1 - 0.2)} \right\rfloor \\ &= 1 + \left\lfloor \frac{-0.2458}{-0.2231} \right\rfloor \\ &= 1 + \lfloor 1.1014 \rfloor \\ &= 1 + 1 = 2. \end{aligned}$$

- Given $u = 0.8923$, simulate a $\text{Geom}(0.4)$ random variable X .

Solution

$$\begin{aligned} X &= 1 + \left\lfloor \frac{\ln(1 - 0.8923)}{\ln(1 - 0.4)} \right\rfloor \\ &= 1 + \left\lfloor \frac{-2.228406}{-0.5108256} \right\rfloor \\ &= 1 + \lfloor 4.362361 \rfloor \\ &= 1 + 4 = 5. \end{aligned}$$

8

Monte Carlo Methods

8.1 The continuous uniform $U(0,1)$ distribution

We have already used this random variable a great deal: the Uniform distribution on the interval $(0,1)$ - see Figure 8.1. This is denoted by $U(0,1)$ and denotes the random variable which has probability density function (p.d.f.):

$$f_X(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Up to now, we have used $U(0,1)$ random variables to simulate from many discrete distributions, and taken the $U(0,1)$ random variables for granted - but how do we simulate them in the first place?

The key is in the accuracy to which the numbers are generated. Generating $U(0,1)$ random variables *precisely* would be very difficult! (Impossible?!?) However we can get three decimal places by generating a random integer x from the set

$$\{0, 1, 2, \dots, 999\},$$

with all outcomes equally likely, and then put $u = x/1000$. Then u is a simulation from a $U(0,1)$ random variable recorded to three decimals. The simulation of x is ‘easy’, e.g. using a Congruential generator with $m = 1000$ and maximum period, as we saw in Chapter 5. So for a full period Congruential generator with $m = 2^{32}$ we get set of integers:

$$\{0, 1, 2, \dots, 2^{32} - 1\}.$$

Setting $u = x/2^{32}$ would give a pseudo-random number from the $U(0,1)$ distribution, recorded to about 8 decimal places.

8.2 Monte Carlo

The term “Monte Carlo” is used to describe any simulation study which involves random numbers. The name is a reference to the famous Monte Carlo Casino in Monaco, where repetition of random events is the order of the day!

8.2.1 What is a simulation study?

For our purposes, a simulation study is any study where we study the properties of a system using random numbers. We have already

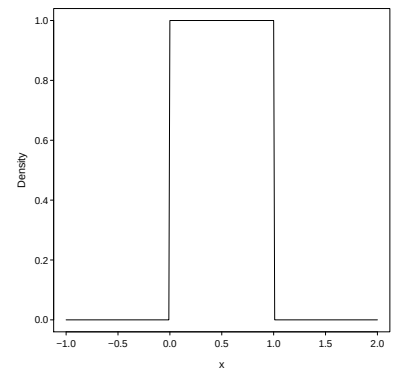


Figure 8.1: PDF of the uniform distribution.

seen simulation studies, for example the monopoly practical. Often simulation studies involve estimating the probability of an event (e.g. in Practical 4 the probability of landing on a particular square). In general, suppose we do an experiment which has the event A as one possible outcome. We would like to estimate the probability of A , denoted by $\Pr[A]$. Then by repeatedly simulating the experiment, it is simple to estimate $\Pr[A]$, the probability of the event A , using $P_F(A)$, where:

$$P_F(A) = \frac{\text{No. of times } A \text{ occurs}}{\text{Number of times experiment simulated}}.$$

Here $P_F(A)$ is the *frequency estimate* of $\Pr[A]$. Why does this work? Well suppose we denote the number of times we simulate the experiment by n , then $P_F(A)$ has the following important property:

$$\text{as } n \rightarrow \infty, \text{ then } P_F(A) \rightarrow \Pr[A].$$

This means that the more simulations we do, the more *accurate* our estimate of $\Pr[A]$.

8.3 Approximation of integrals

Suppose we have a complicated function $f(x)$ defined on the interval $(0, 1)$, and also suppose that $f(x) \in (0, 1)$.

We would like to evaluate $\int_0^1 f(x) dx$, but the function may be too complicated to integrate. We can find an approximate answer by noting that the integral is equal to the area under the curve, and using Monte Carlo methods. We design an experiment which would work in general, even if the function was defined on a general range (a, b) , and if $f(x) \in (0, c)$, for any positive value c . We generate a random data point from a *simulation grid*. Let

$$A = \{\text{the data point lies below the curve}\}.$$

Then

$$\Pr[A] = \frac{\text{area under curve}}{\text{area of simulation grid}},$$

so that

$$\begin{aligned} \int_a^b f(x) dx &= [\text{area under curve}] \\ &= \Pr[A] \times [\text{area of simulation grid}]. \end{aligned}$$

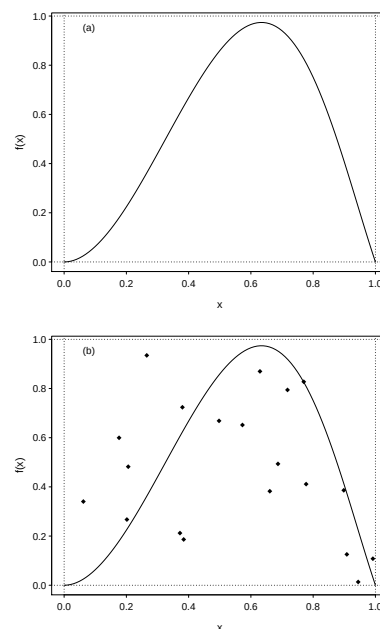


Figure 8.2: (a) An example function. (b) Twenty points randomly placed on the graph.

8.3.1 Example

Consider the function we saw earlier in Figure 8.2, defined on $(0, 1)$, and with $f(x) \in (0, 1)$. Estimate $\int_0^1 f(x) dx$ using Monte Carlo methods.

Solution:

1. We simulate n data points from the simulation grid; here this is the unit square $(0, 1) \times (0, 1)$.
2. Each of the coordinates x and y are generated using a $U(0, 1)$ random variable. Note the fact that

$$A = \{\text{data point } (x, y) \text{ lies below the curve}\} = \{y < f(x)\}.$$
3. If r points lie below the curve, then $\Pr[A] \simeq P_F(a) = r/n$, and thus

$$\begin{aligned} \int_0^1 f(x) dx &= \Pr[A] \times [\text{area of simulation grid}] \\ &= \Pr[A] \times 1 \\ &= \Pr[A] \\ &\simeq P_F(A) \\ &= r/n. \end{aligned}$$

4. In our case, from the figure above we estimate:

$$\int_0^1 f(x) dx \simeq \frac{r}{n} = \frac{14}{20} = 0.7.$$

Using R

We can also easily implement the above algorithm in R. As an extreme example, consider the function

$$f(x) = |\sin\{\sin[\sin(x^4)]\}|.$$

We wish to calculate

$$\int_0^2 f(x) dx = \int_0^2 |\sin\{\sin[\sin(x^4)]\}| dx,$$

which according to maple evaluates to approximately 0.71875. First plot the function to determine the necessary region:

```
> x = seq(0, 2, by=0.001)
> plot(x, abs(sin(sin(sin(x^4))))),
+       xlab="x", ylab="f(x)", type="l",
+       xlim=c(-1,3), ylim=c(0,1))
> abline(h=0, col=2, lty=2)
> abline(h=1, col=2, lty=2)
> abline(v=0, col=2, lty=2)
> abline(v=2, col=2, lty=2)
```

The plot generated in the code above is shown in figure 8.2. Then we use a `for` loop to simulate lots of random numbers

```
> set.seed(1)
> N = 100000
> no_of_hits = 0
> for(i in 1:N) {
+   x = runif(1, 0, 2); y = runif(1)
+   if(abs(sin(sin(sin(x^4)))) > y) {
+     no_of_hits = no_of_hits + 1
+   }
+ }
> area_under_curve = no_of_hits/N*2
> area_under_curve
[1] 0.71524
```

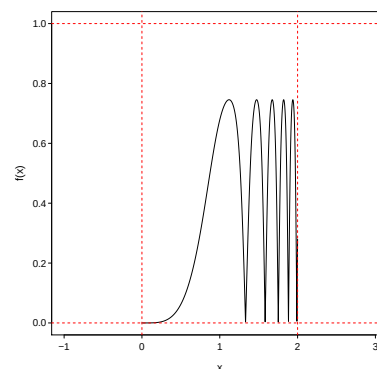


Figure 8.3: A plot of $|\sin\{\sin[\sin(x^4)]\}|$ with the sampling region.

9

Kernel Density Estimation

9.1 Introduction

The goal of density estimation is to approximate the probability density function of a random variable given a sample of observations. One of the most popular methods is kernel density estimators. For example, suppose we observe ten values from the $N(0,1)$, shown in Figure 9.1a. In general, we don't know the true underlying distribution. So how do we go about constructing these estimators?

9.2 Definition

A kernel is a non-negative, real-valued integrable function K that satisfies the following two requirements:

$$(9.1)$$

and

$$(9.2)$$

Expression 9.1 ensures that the kernel is a pdf, whilst expression 9.2 makes the distribution symmetric about 0. Standard kernels are:

Epanechnikov:

Uniform:

Triangular:

Gaussian:

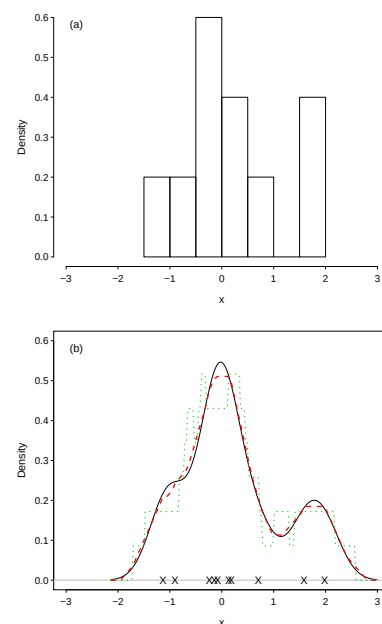


Figure 9.1: (a) Histogram of ten values sampled from a $N(0,1)$ distribution. (b) Three different Kernel density estimators. The data are the X's.

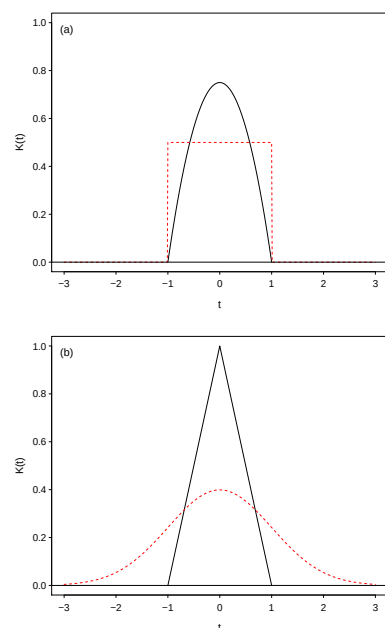


Figure 9.2: (a) The Epanechnikov and Uniform kernels. (b) The triangular and Gaussian kernel.

9.3 The general idea

Kernel density estimation can be summarised in four steps:

1. We have some sample data. In Figure 9.3a we have three points, so $n = 3$.
2. Around each of the data points, we draw a kernel. In Figure 9.3b we have used a Gaussian kernel. However, we could have used a Uniform, triangular, or Epanechnikov kernel.
3. Next we combine the kernels - the blue dashed line in Figure 9.3c.
4. The final step is to normalise the distribution. In our example, since we have three points, the total area under the blue dashed curve is 3. Hence, we recover a density we divide by 3 to get the black curve in Figure 9.3d.

Run `demo(kerneldensity)` to get figures 9.3 a-d.

9.4 Estimation

Let K be a kernel and suppose our sample contains n values: $x_1, \dots, x_n$. Then our estimate of the true pdf $f(x)$ is

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^n K(x - x_i) . \quad (9.3)$$

Notice we use $K(x - x_i)$, since we draw a kernel around *each* x_i . It's fairly straightforward to see that $\hat{f}(x)$ is also a pdf, namely

$$\begin{aligned} \int_{-\infty}^{\infty} \hat{f}(x) dx &= \frac{1}{n} \sum_{i=1}^n \int_{-\infty}^{\infty} K(x - x_i) dx \\ &= \frac{1}{n} \sum_{i=1}^n \int_{-\infty}^{\infty} K(y) dy \\ &= \frac{1}{n} \sum_{i=1}^n 1 \\ &= 1 . \end{aligned} \quad (9.4)$$

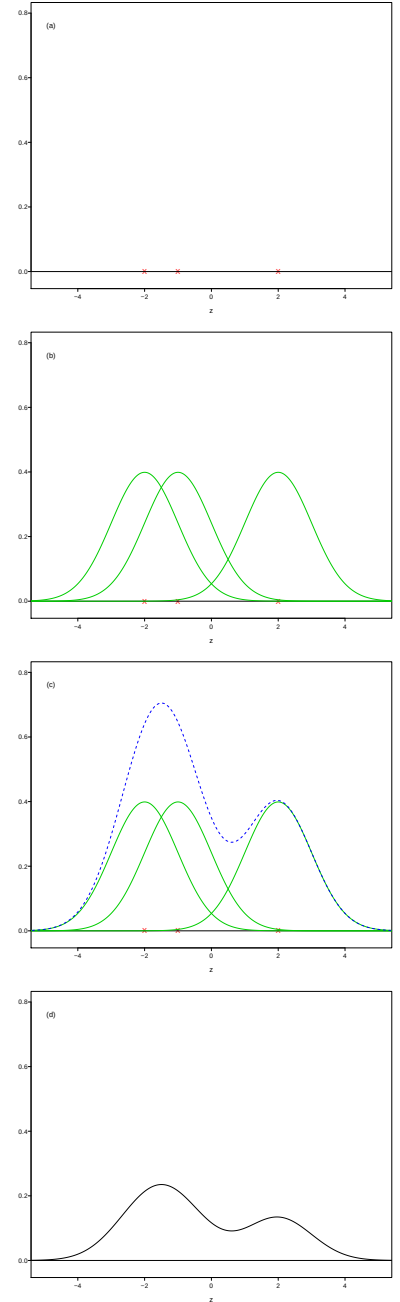


Figure 9.3: (a) Plot showing the data. (b) A Gaussian kernel is drawn round each point. (c) The kernels are combined. (d) The function is scaled an area of one.