Simulation and Exploratory Data Analysis

STAT2005

Chapter 2

Simulation

- One of the major applications of computer in statistics is simulation.
- Simulation becomes an inevitable tool in statistics nowadays. It can be used in testing and assessing our statistical methodologies.
- Through simulation, we can test our statistical procedures under different distributional assumptions.
- Using R, we can generate pseudorandom numbers from various statistical distributions.

Pseudorandom numbers

- In R and most other common computer programming language, an algorithm is used to generate a sequence of pseudorandom numbers whose properties approximate the properties of sequences of random numbers.
- Pseudorandom number generators are important in practice for their speed in number generation and their reproducibility.

Pseudorandom numbers

So the random numbers R generates are not actually random?

- Correct. They behave very similarly (in a statistical sense) to genuinely random numbers, i.e., they exhibit independence and follow prescribed probability distributions
- “Nevertheless, we should be aware that the apparently random numbers at the heart of a simulation are in fact produced by completely deterministic algorithms” – Chapter 2 of “Monte Carlo Methods in Financial Engineering”, by Glasserman.

Pseudorandom numbers

Here's a brief hint of how pseudorandom numbers can be generated.

The linear congruential generator uses a multiplier a, modulus m and seed x_0

- For example, let $a = 6,\ m = 11,\ x_0 = 1$
- Then the next number is given by $x_1 = 6 \times 1 \pmod{11} = 6$ (the remainder of $6 \times 1 = 6$ on division by 11 is 6)
- Then $x_2 = 6 \times 6 \pmod{11} = 3$ (the remainder of $6 \times 6 = 36$ on division by 11 is 3
- Then $x_3 = 6 \times 3 \pmod{11} = 7$
- And so on. What are $x_3, x_4, x_5, x_6, x_7, x_8, x_9$ and x_{10}? What happens after them?

In practice, one may choose $m = 2^{32} - 1$ and $a = 7^5$.

Simulation using the `sample()` function

The function `sample()` generates random samples or permutations of data.

```
> sample(10)
# random permutation of integers from 1 to 10
 [1]  3 10  4  5  9  8  7  1  2  6
> sample(10,replace=T) # sample with replacement
 [1] 5 9 3 4 7 3 9 5 9 3
# 20 random drawings from a given discrete
distribution
> sample(c(-1,0,1),size=20,
prob=c(0.25,0.5,0.25),replace=T)
 [1]  0  0 -1  0  0 -1  0  1 -1 -1  0  0  0
 1  0  0  0  0  0  0
```

Let us increase the size from 20 to 10000 and compute the proportion of -1, 0 and 1 in the output respectively.

```
# increase size to 10000
> d<-sample(c(-1,0,1), size=10000,
prob=c(0.25,0.5,0.25), replace=T)
> sum(d==-1)/10000
# compute the proportion of -1
[1] 0.2528
> sum(d==0)/10000
# compute the proportion of 0
[1] 0.4962
> sum(d==1)/10000
# compute the proportion of 1
[1] 0.251
```

- Note that the proportion is close to the 0.25, 0.5 and 0.25 in the `sample()` function. The difference is due to the sampling error.
- As the simulation size increases, this sampling error decreases and closer to the true probabilities.
- We get different sequence of pseudorandom numbers each time when we execute the function `sample()`. It is because these pseudorandom numbers are generated using a random initial seed.
- If we want to produce the same sequence of pseudorandom numbers each time (for verification purpose), we can set the initial seed manually by using `set.seed`.

Let us illustrate this by the following

```
> set.seed(12345)# set initial seed to 12345
> sample(10)# generate first sample
[1]  3  8  2  5 10  9  7  6  4  1
> sample(10)# generate second sample
[1] 10  1  8  7  6  9  4  2  3  5
> sample(10)# generate third sample
[1]  9  4  2  7  6  8 10  3  5  1
> set.seed(12345)
> # reset the initial seed again to 12345
> sample(10)# generate first sample
[1]  3  8  2  5 10  9  7  6  4  1
> sample(10)# generate second sample
[1] 10  1  8  7  6  9  4  2  3  5
```

Random walk

- Imagine that we are playing a game with a fair coin. We win $1 if the result is head or loss $1 if it is tail.
- We perform a simulation of this game by generating 100 random numbers from a discrete uniform distribution. Then we plot the money we win or loss.
- Let us use `sample()` function to generate a random walk. This random walk is a time series starts with pre-assigned initial value equals 10.

- Let our initial wealth be $W_0 = w_0$, our wealth after t tosses be W_t and the outcome of the tth toss be R_t.
- Then we have

$$W_t = W_{t-1} + R_t,$$

where

$R_t = 1$ with probability p (the probability of tossing a head) or

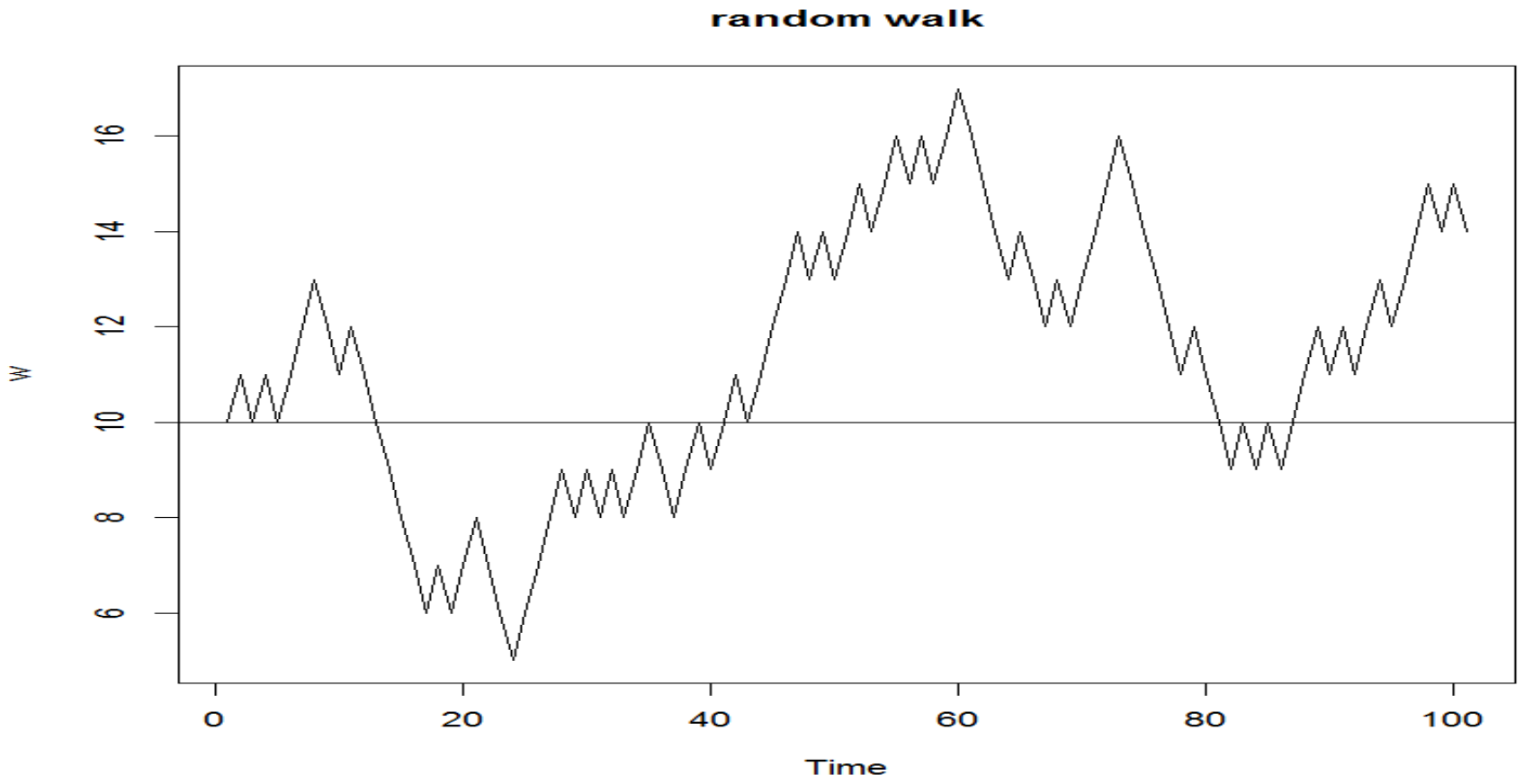
$R_t = -1$ with probability $1 - p$ (the probability of tossing a tail)

- For a fair coin, $p = 0.5$.

```
> set.seed(13579)
# set the initial seed manually
# 100 random drawings with Pr{R=-1} = Pr{R=1} = 0.5
> r<-sample(c(-1,1),size=100,replace=T,prob=c(0.5,0.5))
> (r<-c(10,r))
# append the starting value (10) to r


  [1] 10  1 -1  1 -1  1  1  1 -1 -1  1 -1 -1 -1 -1 -1 -1
1 -1  1  1 -1 -1 -1  1
 [26]  1  1  1 -1  1 -1  1 -1  1  1 -1 -1  1  1 -1  1  1
-1  1  1  1  1 -1  1 -1
 [51]  1  1 -1  1  1 -1  1 -1  1  1 -1 -1 -1 -1  1 -1 -1
1 -1  1  1  1  1 -1 -1
 [76] -1 -1 -1  1 -1 -1 -1  1 -1  1 -1  1  1  1 -1  1 -1
1  1 -1  1  1  1 -1  1
[101] -1
```

```
> (w<-cumsum(r))
# cumulative sum of vector r
  [1] 10 11 10 11 10 11 12 13 12 11 12 11 10  9  8  7  6
7  6  7  8  7  6  5  6
 [26]  7  8  9  8  9  8  9  8  9 10  9  8  9 10  9 10 11
10 11 12 13 14 13 14 13
 [51] 14 15 14 15 16 15 16 15 16 17 16 15 14 13 14 13 12
13 12 13 14 15 16 15 14
 [76] 13 12 11 12 11 10  9 10  9 10  9 10 11 12 11 12 11
12 13 12 13 14 15 14 15
[101] 14
> w<-as.ts(w)
# transform w into a time series object
> plot(w,main="random walk")
# plot w with main title
> abline(h=10)
# add a horizontal line (see help(abline) for details)
```


random walk
W
6
8
10
12
14
16
0
20
40
60
80
100
Time

Exercise

1. The player goes bankrupt when his/her worth level is lower than 0 at any time point. Create a logical value which equals TRUE if the player has bankrupted.

2. What do you expect when the number of toss is significantly increased.

3. How can we simulate a random walk with a biased coin?

Monte Carlo simulation

- One of the most general computer-based methods for approximating properties of random variables is the Monte Carlo method.
- To approximate the mean $\mu = E(X)$ using the Monte Carlo method, we generate m independent and identically distributed (i.i.d.) copies of X, namely $X_1, \ldots, X_m$, and use the sample mean

$$\overline{X} = \frac{1}{m} \sum_i X_i$$

as an estimate of $E(X)$.

- For large values of m, $\bar{X}$ gives a good approximation to $E(X)$
- Furthermore, when m is large, the distribution of the sample mean, $\bar{X}$, can be approximated by a normal distribution with mean μ and variance σ^2/m, where σ^2 is the variance $Var(X)$.
- This allows us to construct approximate confidence intervals for μ.

Random number between a and b

The `runif()` function can be used to generate random number from the uniform distribution.

```
runif(n, min = a, max = b)
```

Execution of this command produces n pseudorandom uniform number on the interval $[a, b]$. The default values are $a = 0$ and $b = 1$.

```
> runif(5)
[1] 0.9502223 0.3357378 0.1330718 0.4901114 0.0607455
> runif(10, min = -3, max = -1)
[1] -2.284105 -2.545768 -2.199852 -1.126908 -1.324746 -
2.744848
[7] -1.549739 -1.445740 -2.834744 -1.372574
```

Built-in probability distribution functions

In R, there are four fundamental items that can be computed for a statistical distribution, namely:

- Probability density function (pdf) or Probability mass function (pmf), (e.g. `dnorm`)
- Cumulative probability distribution function (cdf), (e.g. `pnorm`)
- Quantiles, (e.g. `qnorm`)
- Pseudorandom numbers. (e.g. `rnorm`)

Related concepts in statistics

For a discrete random variable X, the probability of $X = x$,

$$\Pr\{X = x\} = p(x)$$

where $p(x)$ is the **probability mass function** (pmf) of X.

Similarly, for a continuous random variable X, the probability of $a < X < b$,

$$\Pr\{a < X < b\} = \int_a^b f(x)dx$$

where $f(x)$ is the **probability density function** (pdf) of X.

The **cumulative probability distribution function** (cdf) $F(x)$ gives the probability of X less than or equal to x, i.e.

$$F(x) = \Pr\{X \leq x\}$$

In other words,

$$F(x) = \sum_{t=-\infty}^{x} p(t)$$

for discrete X or

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

for continuous X.

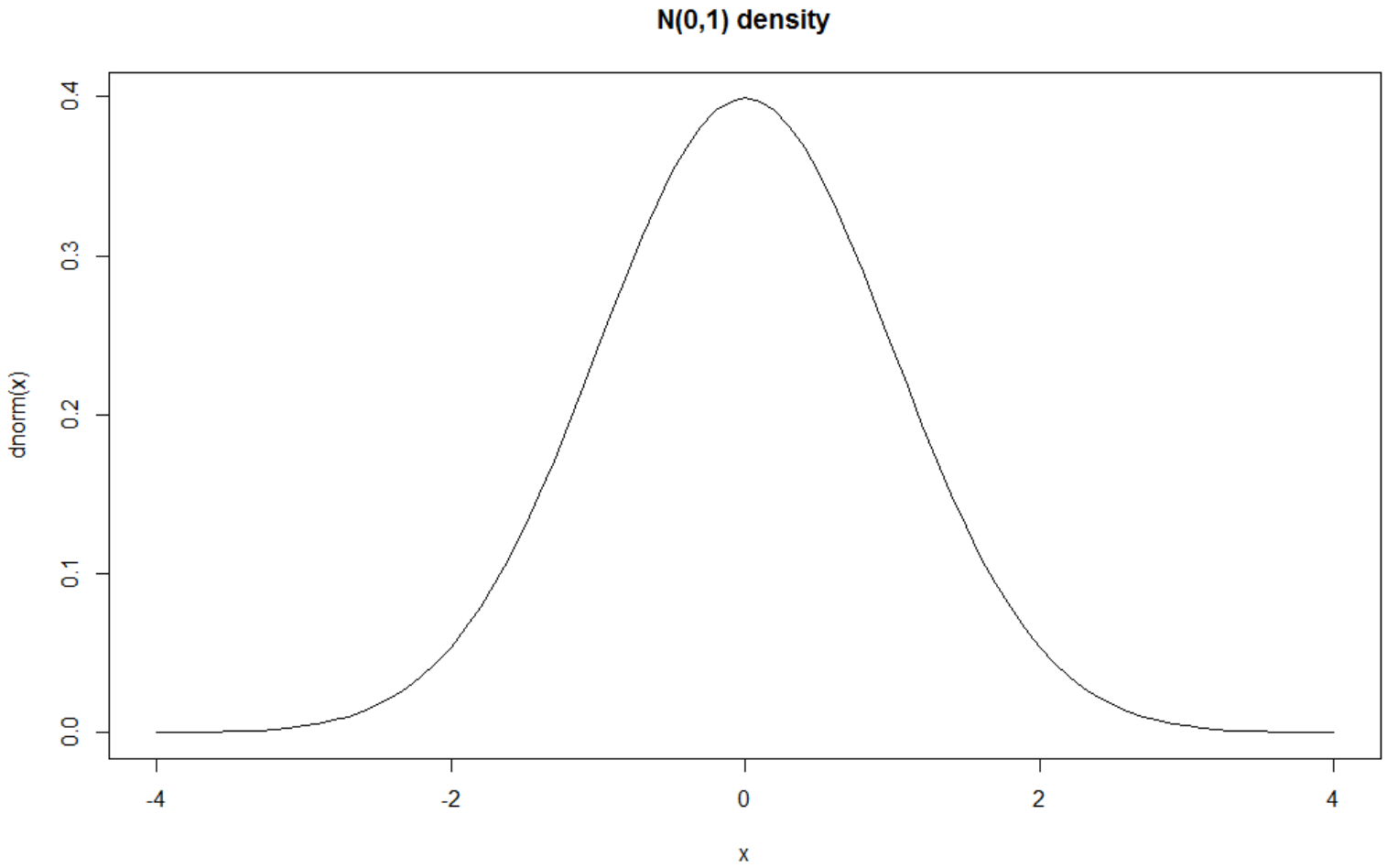
The **quantile function** is the inverse of the cumulative distribution function. The q-th quantile of X is the smallest value of x, denoted x_q, such that

$$\Pr\{X \leq x_q\} \geq q.$$

We can plot the well-known bell-shape normal density curve with the following

```
> x<-seq(-4,4,0.1)
> plot(x,dnorm(x),type="l",main="N(0,1) density")
```

The first command generates a sequence from -4 to 4 with increment 0.1. The `dnorm(x)` gives the density of x, `type="l"` causes the plot using lines instead of points.


N(0,1) density
dnorm(x)
0.0
0.1
0.2
0.3
0.4
-4
-2
0
2
4
x

We can use the function `pnorm()` to find out the probability that $\Pr\{X < 1.96\}$ by

```
> pnorm(1.96)
[1] 0.9750021
```

To illustrate the quantile function, we first defines the probabilities and then compute the corresponding quantiles of the standard normal distribution by `qnorm()` function.

```
> q<-c(0.025,0.05,0.5,0.95,0.975)
> qnorm(q)
[1] -1.959964 -1.644854  0.000000  1.644854  1.959964
```

Finally, we can generate 100 random numbers from N(0,1) by

```
> rnorm(100)
```

Note that the naming convention in R, `dnorm`, `pnorm`, `qnorm` and `rnorm` stands for the density, cdf, quantiles and random number of the standard normal distribution.

We try another well-known discrete distribution, the binomial distribution with $n = 20$ and $p = 1/4$. The pmf is

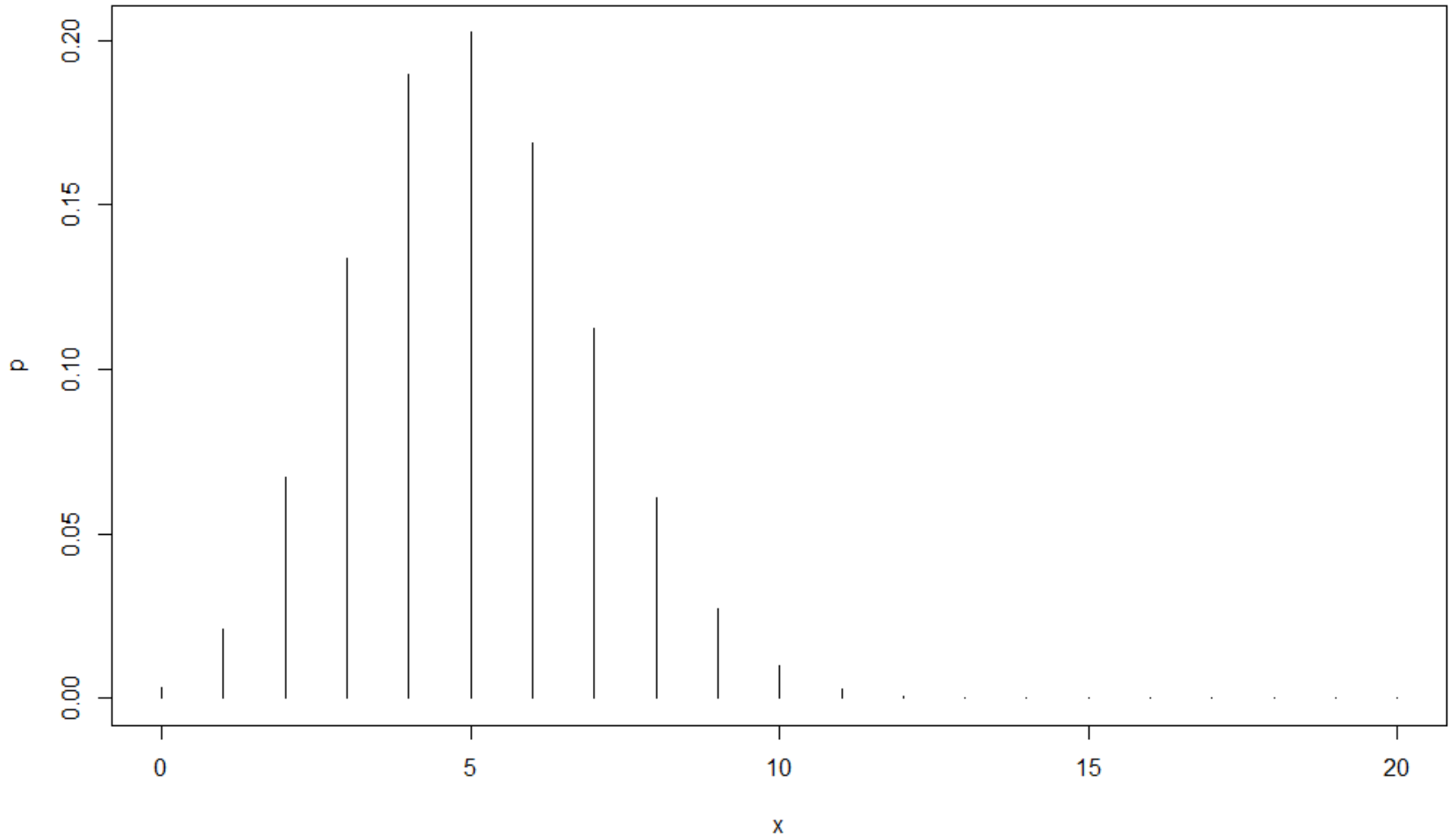
$$\Pr\{X = x\} = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0,1,\ldots,20$.

```
> x<-0:20
# generate an integer sequence from 0 to 20
> p<-dbinom(x,size=20,prob=1/4)
# p is a prob vector, p=Pr(X=x)

> round(p,digits=4)
# display the vector p
 [1] 0.0032 0.0211 0.0669 0.1339 0.1897 0.2023
0.1686 0.1124 0.0609 0.0271 0.0099 0.0030
[13] 0.0008 0.0002 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
> plot(x,p,type="h")        # plot the pmf
```

Note that the option `type="h"` to plot the pmf using vertical lines.


p
0.00
0.05
0.10
0.15
0.20
0
5
10
15
20
x

To find the probability that X is greater than 8 (tail probability, i.e. $\Pr\{X > 8\}$), we can use

```
> 1-pbinom(8,size=20,prob=1/4)
# Pr{X>8} = 1 – Pr{X<=8}
[1] 0.04092517
```

We can also obtain this number from p by the following:

```
> 1-sum(p[1:9])    # Pr{X>8} = 1- Pr{X<=8}
[1] 0.04092517
```

To find the 0^{th}, 10^{th}, 20^{th} ,…, 100^{th} quantiles of X, (i.e., the smallest x_q such that $Pr\{X \le x_q\} \ge q$), we can use

```
> (q<-seq(0,1,0.1))
# create and display q
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> qbinom(q,size=20,prob=1/4)
# quantiles from bin(n,p) corresponds to q
 [1]  0  3  3  4  4  5  5  6  7  8 20
```

We can confirm the above quantiles by the following:

```
> round(cumsum(p),digits=4)
 [1] 0.0032 0.0243 0.0913 0.2252 0.4148 0.6172 0.7858
0.8982 0.9591 0.9861 0.9961 0.9991
[13] 0.9998 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 1.0000
```

Note that $Pr\{X \le 2\} = 0.0913$ and $Pr\{X \le 3\} = 0.2252$, therefore, the 10th quantile is 3.

The following table gives a list of all built-in function for various statistical distributions in R. The syntax of these functions are placing the letter d, p, q and r in front of these function's name.

Distribution	R name	Additional arguments
Beta	beta	shape1, shape2, ncp
Binomial	binom	size, prob
Cauchy	cauchy	location, scale
Chi-square	chisq	df, ncp
Exponential	exp	rate
F	f	df1, df2, ncp
Gamma	gamma	shape, scale
Geometric	geom	prob
Hypergeometric	hyper	m, n, k
Log-normal	lnorm	meanlog, sdlog
Logistic	logis	location, scale
Negative binomial	nbinom	size, prob
Normal	norm	mean, sd
Poisson	pois	lambda
Student's t	t	df, ncp
Uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Example

Suppose that we want to estimate

$$E[|X|], \text{ where } X \sim N(0,1).$$

This can be done using Monte Carlo simulation.

Note: the theoretical value of $E[|X|]$ is $\sqrt{2/\pi}$.

```
n <- 1000
X <- rnorm(n)
Est <- mean(abs(X))    # estimate
SE <- sd(abs(X))/sqrt(n) # standard error
CI95 <- c(Est-qnorm(0.975)*SE, Est+qnorm(0.975)*SE)
# 95% confidence interval
c(Est, sqrt(2/pi), CI95)
```

Exploratory data analysis

- An important advantage of R is that we can analyse our data interactively. That means we can perform our analysis step by step.
- We can perform some exploratory data analysis, look at the results and decide what to do next.
- The first step in data analysis is to read in our data. Data are usually stored in a file with rows representing observations and columns representing variables.
- There are many formats used in storing data. The most simple and common one is the ASCII (American Standard Code for Information Interchange) format, with file extension `.dat` or `.txt`.

Working directory

- If you want to read or write files from a specific location you will need to set working directory in R.
- To see the current working directory, we can use

```
getwd()
```

- To change the working directory, we press `File -> Change dir…` from the menu bar and then choose your desired folder.
- Alternatively, we can set working directory using the command

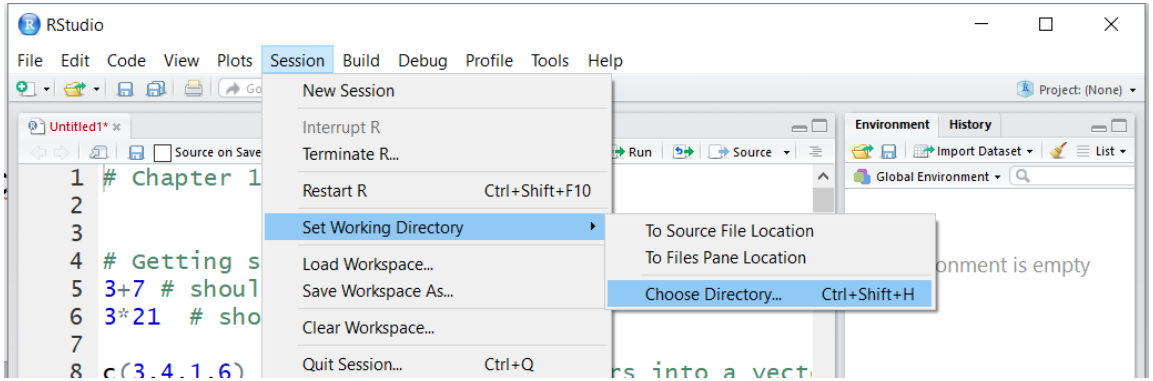
```
setwd("C:/Folder")
```

to set the directory to `Folder` on C drive.

On other platform…

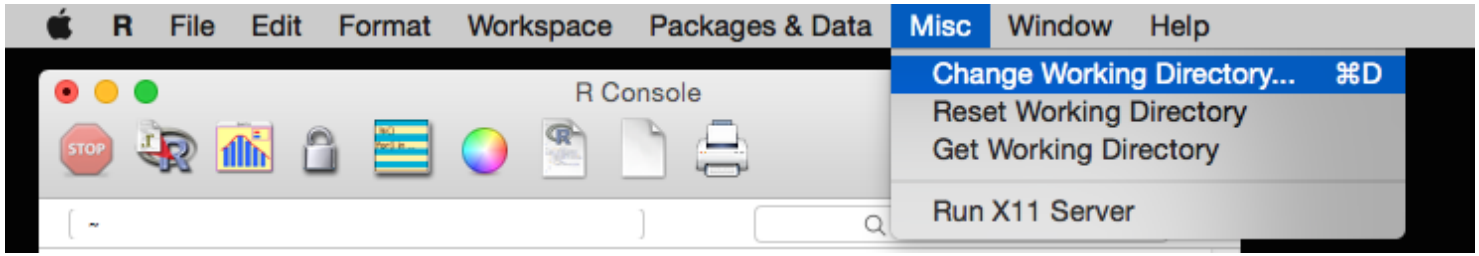
In RStduio, we can change directory by

`Session -> Set Working Directory -> Choose Directory`



On macOS R Console, we can change directory by

`Misc -> Choose Directory`



Executing commands from a source file

- So far, we have entered the commands in R interactively. Actually, commands can be executed from an ASCII file using the source command.
- First, we can enter the following commands using a text editor and save it as text (ASCII) file or simply using the R command

```
edit(file="ch2-apply.r")
```

- This command will invoke the built-in text editor and open an ASCII file name "`ch2-apply.r`".

Now enter the following commands and save it.

```
set.seed(123)      # set initial seed
x<-runif(500,0,20)
# generate 500 uniform(0,20) random numbers
y<-matrix(x,ncol=10)
# put x into a 50 by 10 matrix
# compute the mean and sd of each column and
# store the results in m and s respectively.
m<-apply(y,2,mean)
# apply the function mean in the 2nd margin, ie.
Column
# equivalently we can use colMeans(y)
s<-apply(y,2,sd)
# compute column sd (see help(apply) for more
details)
```

- These commands will generate 500 random numbers from Uniform(0,20) and rearrange them into a 50 by 10 matrix.
- In data analysis, we usually take each column as variable and each row as observation.
- Now we compute the sample mean and sample standard deviation of each column (variable) using the function `mean()` and `sd()` and store them in vector `m` and `s` respectively.
- Note that there are three arguments in the `apply()` function. The first argument is a matrix and the third argument is a function name that we want to apply. The second argument indicates the margin of the matrix (in our example, column is the 2nd margin) that we want to apply the function to.
- Now we can execute these commands contained in the file "`ch2-apply.r`" by typing

```
> source("ch2-apply.r",echo=T)
```

Note that `echo=T` will echo these commands to the screen.

```
> m  # display the value stored in m
 [1] 10.401819  9.540540 10.307797 10.261198
10.253117
 [6]  9.188950  9.849096  9.838603  9.813296
9.602321

> s  # display the value stored in s
 [1] 5.885765 5.533551 5.880352 4.652955 5.462771
 [6] 6.292274 5.749865 6.060184 5.583015 6.017083
```

Note also that if X follows Uniform(a,b), then E(X) = (a+b)/2 and Var(X) = $(b-a)^2/12$.

In our example, E(X) = 10, Var(X) = 400/12 and SD(X) = 5.7735.

Inputting ASCII data file

Suppose we have an ASCII data file "popden.dat" contains the population density (thousand/sq. km) of 19 districts of Hong Kong in 1986 and 1990.

district	year86	year90	Region				
Islands	290	293	NT	Wan_Chai	20182	18209	HK
Sai_Kung	365	1026	NT	Central/West	20854	20479	HK
Tai_Po	1033	1496	NT	Kwai_Tsing	21464	21158	KL
North	1074	1211	NT	Eastern	27387	30316	KL
Yuen_Long	1545	1664	NT	Yau_Tsim	45355	33232	KL
Tuen_Mun	3611	4711	NT	Wong_Tai_Sin	46940	41331	KL
Tsuen_Wan	4159	4581	NT	Kowloon_City	47156	41759	KL
Sha_Tin	5402	7378	NT	Sham_Shui_Po	56875	48822	KL
Southern	6380	6701	HK	Kwun_Tong	60826	52562	KL
				Mong_Kok	142718	116531	KL

We can read in this data file using

```
> d<-read.table("popden.dat",header=T,stringsAsFactors=T)
# read in data file popden.dat with header
# character variables are stored as factor
> names(d)
# display the labels in d
[1] "district" "year86"   "year90"   "Region"
```

Note that the first row in the data file is the header containing the label of each variable.

Therefore we need to use `read.table()` with the option `header=T`.

Another useful and handy function is `head()` which display the first 6 lines in the data matrix with the variable names. Similarly, `tail()` will display the last 6 lines.

```
> head(d)   # display the first 6 lines in d
   district year86 year90 Region
1   Islands    290    293     NT
2  Sai_Kung    365   1026     NT
3    Tai_Po   1033   1496     NT
4     North   1074   1211     NT
5 Yuen_Long   1545   1664     NT
6  Tuen_Mun   3611   4711     NT
```

Variable type

- In data analysis, we usually classify variables into continuous or categorical variables.
- In the example above, `District` and `Region` are categorical while `year86` and `year90` are continuous.
- Note that the class of `year86` and `year90` are integer while the class of `Districts` and `Region` are factor.

```
> class(d$year86)          # display class of year86
[1] "integer"
> class(d$Region)          # display class of Region
[1] "factor"
> d$Region              # Region has 3 levels
 [1] NT NT NT NT NT NT NT NT HK HK HK KL KL KL KL KL KL
KL KL
Levels: HK KL NT
```

Data frame

- The object `d` is a `data.frame` object. It looks like a matrix but each column could contain different type of object, such as numeric, logical and character values.
- For example, `district` and `Region` are factor while `year86` and `year90` are numeric.
- We can change a matrix into data frame using the command `data.frame()`.

```
> class(d)
[1] "data.frame"
```

Suppose we want to find the mean of `year90` in each `Region`, we can use `tapply()`:

```
> (t<-tapply(d$year90,d$Region,mean))
# save result to t and display
      HK       KL       NT
15129.67 48213.88  2795.00
```

The function `split()` will split the data into a list with components according to some factor. For example, the following split `year90` into a list with 3 components according to `Region`:

```
> (s<-split(d$year90,d$Region))
# save result to s and display
$HK
[1]  6701 18209 20479
$KL
[1]  21158  30316  33232  41331  41759  48822  52562 116531
$NT
[1]  293 1026 1496 1211 1664 4711 4581 7378
```

We can check the mean of each region by

```
> mean(s$HK)
[1] 15129.67
> mean(s$KL)
[1] 48213.88
> mean(s$NT)
[1] 2795
```

We can use the `table()` function to find out how many cases are there in each region:

```
> table(d$Region)
# display frequency count in each Region
HK KL NT
 3  8  8
```

The following `by()` function computes the mean of `year86` and `year90` in each region:

```
> by(d[,c(2,3)],d$Region,colMeans)
# sample mean by each Region

d$Region: HK
  year86   year90
15805.33 15129.67
------------------------------------------------
d$Region: KL
  year86   year90
56090.12 48213.88
------------------------------------------------
d$Region: NT
  year86   year90
2184.875 2795.000
```

We can define our own categorical variable as well.

For example, if we define a categorical (or binary) variable `dense=1` if both `year86` and `year90` are greater than 10,000, and `dense=0` otherwise. We can use the following to create `dense` as well as a "two-way" table for `dense` and `Region` and save the result to `t`.

```
> dense<-as.numeric(
     (d$year86>10000)&(d$year90>10000)
)
# create binary variable dense
> (t<-table(dense,d$Region))
# save the two-way table and display

dense HK KL NT
    0  1  0  8
    1  2  8  0
```

```
> dense
# display dense
 [1] 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
```

Alternatively, we can create extra variables for a data frame using the function `with()`.

```
> d$dense<-with(d,
    as.numeric((year86>10000)&(year90>10000))
)
# add the variable dense to d
```

aggregate() function

The aggregate() function is a more flexible way to split and apply transformation to data.

To split the variable year86 by Region, we can use

```
aggregate(year86~Region,d,mean)
```

To split both year86 and year90 by Region and dense, we can use

```
aggregate(cbind(year86,year90)~Region+
dense,d,mean)
```

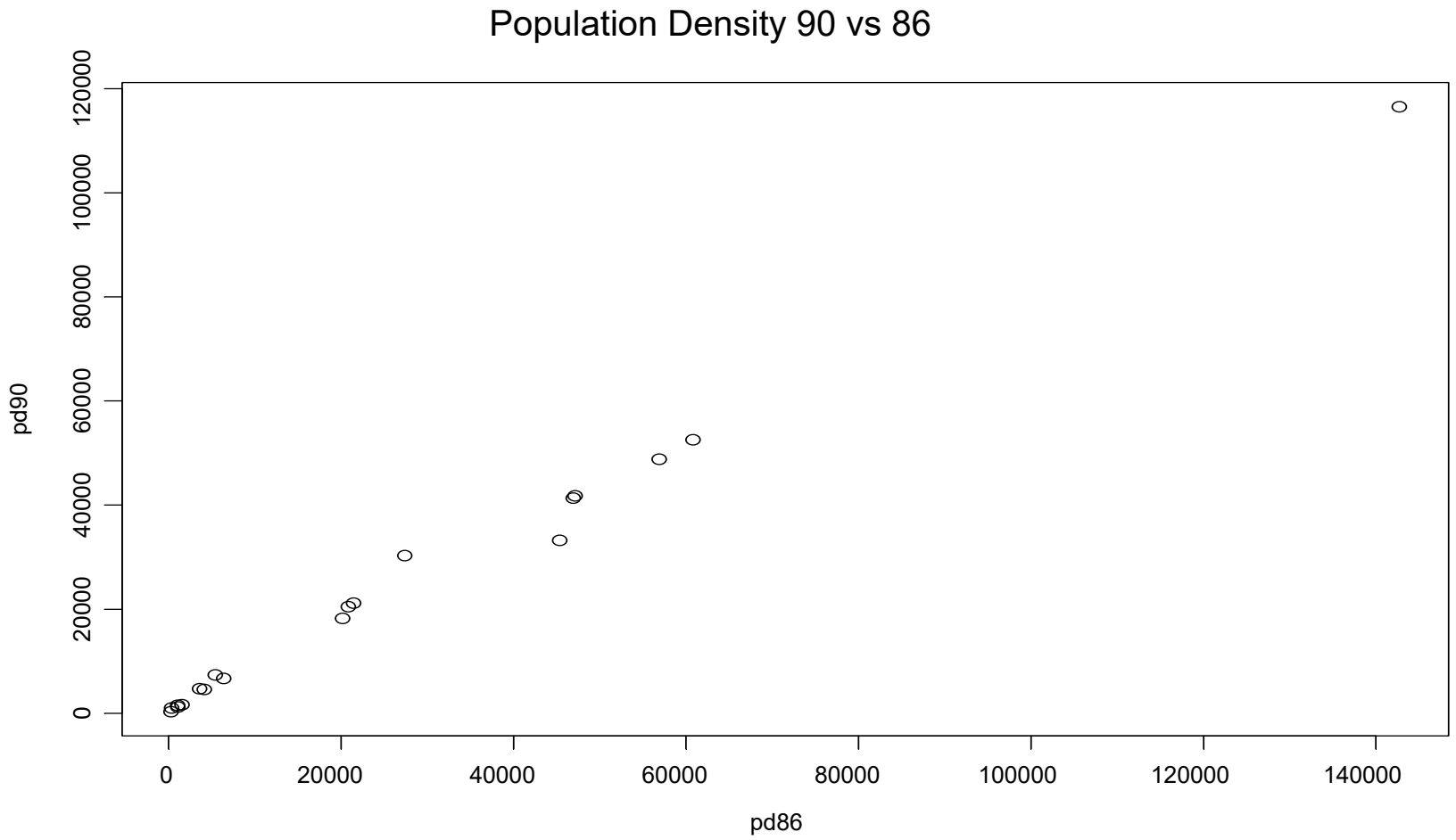
```
> aggregate(year86~Region,d,mean)
  Region    year86
1     HK 15805.333
2     KL 56090.125
3     NT  2184.875

> aggregate(cbind(year86,year90)~Region+dense,d,mean)
  Region dense    year86   year90
1     HK     0  6380.000  6701.00
2     NT     0  2184.875  2795.00
3     HK     1 20518.000 19344.00
4     KL     1 56090.125 48213.88
```

Simple scatter plot

- We can use scatter plot to study the relationship between two or more continuous variables.
- For example, if we want to know the relationship between `year86` and `year90`, we can use the following commands:

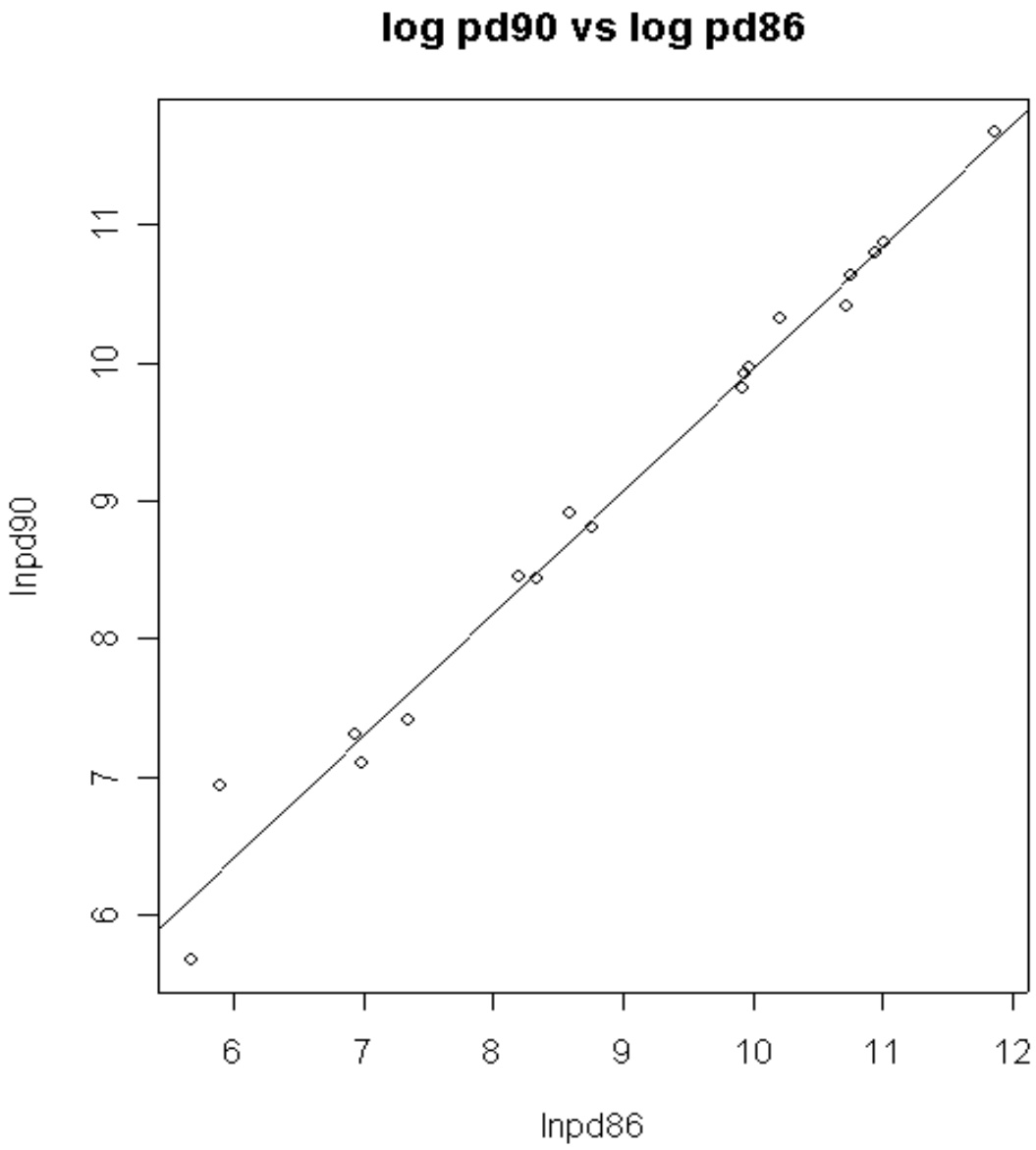
```
> pd86<-d$year86
# assign the 2nd column to pd86
> pd90<-d$year90
# assign the 3rd column to pd90
> plot(pd86,pd90,main="Population Density 90 vs 86")
# plot pd90 vs pd86
```


Population Density 90 vs 86
pd90
0
20000
40000
60000
80000
100000
120000
pd86
0
20000
40000
60000
80000
100000
120000
140000

Note that the range in the data is large and the plot suggested a log transformation is needed.

We can produce the plot with the log-transformed data and add the least squares line to the plot by

```
> lnpd86<-log(pd86)
# natural log transformation on pd86
> lnpd90<-log(pd90)
# natural log transformation on pd90
> plot(lnpd86,lnpd90,main="log pd90 vs log pd86")
# plot log pd90 vs log pd86
> abline(lsfit(lnpd86,lnpd90))
# add least square line
```



- This least square line is the line that closest to the data point (minimizing the error sum of squares).
- The built-in function `lsfit()` will returns the intercept and slope of this least square line.

Saving an R plot

- We will often want to save the output of an R plot.
- We can simply copy the plot window and paste it into an appropriate file type, such as a Word document.
- Alternatively, we use the `pdf( )` function to create a pdf, and the `jpeg( )` function to create a jpeg.

```
pdf("Figure.pdf")
plot(lnpd86,lnpd90,main="log pd90 vs log pd86")
abline(lsfit(lnpd86,lnpd90))
dev.off()
```

Saving data file

Now suppose we want to save the original data together with the transformed data. We use the `cbind()` function to combine the columns of matrix and vectors to an object and then write it to an ASCII file.

```
> x<-cbind(d[,1:4],lnpd86,lnpd90)
# combined with lnpd86 and lnpd90
> names(x)
# display the labels in x
[1] "district" "year86"   "year90"   "Region"   "lnpd86"
"lnpd90"
> write.table(x,file="popden1.dat",row.names=F)
# write x to an ASCII file
```

- Next time when we read in this file with `read.table("popden1.dat")`, we will have two more columns of log-transformed population density.
- Note that the option `row.names=F` is important. Without this option, R will automatically add in the row number in `x`.
- Another commonly used data file format is comma separated value (CSV) file. This format can be imported to and exported from Excel directly.
- Let us continue with the population density example. We can save the object `x` (which contains `popden.dat` and two extra columns `lnpd86` and `lnpd90`) in CSV format by

```
> write.csv(x,file="popden1.csv", row.names=F)
```

- Now we have an extra file `"popden1.csv"` that can be imported directly into Excel. We can also read this CSV file in R using

```
> d1<-read.csv("popden1.csv")
```

R data editor

R has a built-in data editor to help us to enter the data like using Excel.

Suppose we want to edit the dataset `d1`, try the following commands:

```
> x<-edit(d1)
```

Now, the data editor window appears and we can edit the data. We can change the labels in each column by clicking the column header.

Close the window when finish and we will have our data stored in `x`.

Data Editor

	district	year86	year90	Region	lnpd86	lnpd90	var7	var8
1	Islands	290	293	NT	5.669881	5.680173		
2	Sai_Kung	365	1026	NT	5.899897	6.933423		
3	Tai_Po	1033	1496	NT	6.940222	7.31055		
4	North	1074	1211	NT	6.979145	7.099202		
5	Yuen_Long	1545	1664	NT	7.342779	7.41698		
6	Tuen_Mun	3611	4711	NT	8.19174	8.457655		
7	Tsuen_Wan	4159	4581	NT	8.33303	8.429673		
8	Sha_Tin	5402	7378	NT	8.594525	8.906258		
9	Southern	6380	6701	HK	8.760923	8.810012		
10	Wan_Chai	20182	18209	HK	9.912546	9.809671		
11	Central/West	20854	20479	HK	9.945301	9.927155		
12	Kwai_Tsing	21464	21158	KL	9.974132	9.959773		
13	Eastern	27387	30316	KL	10.21782	10.31943		
14	Yau_Tsim	45355	33232	KL	10.72228	10.41127		
15	Wong_Tai_Sin	46940	41331	KL	10.75663	10.62937		
16	Kowloon_City	47156	41759	KL	10.76122	10.63967		
17	Sham_Shui_Po	56875	48822	KL	10.94861	10.79594		
18	Kwun_Tong	60826	52562	KL	11.01577	10.86975		
19	Mong_Kok	142718	116531	KL	11.86863	11.66591		