R-intro-ko

KRUG

원문출처: http://cran.r-project.org/doc/manuals/R-intro.html

An Introduction to R (한국어 번역판 . 버전 0.73 - September. 19. 2011)

이 문서는 GNU S 라고 불리우는 R 에 대한 기초 안내서입니다. 본래 R 은 Bell 연구실에서 John Chambers 와 그의 동료들에 의해서 선형, 비선형, statistical tests, 시계열, classification, clustering 과 같은 다양한 통계분석 및 시각화를 목적으로 개발된 S 라는 시스템과 매우 유사합니다.

이 문서는 이러한 목적으로 사용하고자 하는 초보자를 대상으로 데이터형, 프로그래밍의 기본요소들, 그리고 통계학적 모델링 및 그래픽에 대한 정보를 제공합니다.

이 문서의 버전은 2.13.1 (2011-07-08) 입니다.

ISBN 3-900051-12-7

본 안내서는 S 와 S-Plus 에 대해서 Bill Venables 와 David M. Smith 가 Adelaide 대학에서 1990년부터 1992년까지 작성한 내용을 토대로 하고 있습니다. 그러나, 우리는 R 과S 프로그램간의 차이를 고려하여 본래의 내용을 적절히 수정 및 변경을 하거나 추가하였습니다. 이러한 수정 및 재배포를 수락하고, R의 후원자가 되어주신 Bill Venables 와 David Smith 에게 깊은 감사의 뜻을 표합니다. 만약, 내용상의 오류를 발견하셨다면, R-core@R-project.org 으로 이메일을 보내주세요.

처음 접하는 사용자에게

만약 이 문서를 읽고 있는 독자가 R을 처음 접하는 것이라면 R 이 어떻게 동작하는지를 즉각적으로 이해하기 위해서 Appendix A 를 가장 먼저 보기를 권합니다.

많은 사용자들이 R을 그래픽을 위한 목적으로 사용하고 있기 때문에 그래픽에 관련한 주제를 Graphics 라는 섹션에 따로 모아 정리하여 필요할 때마다 찾아 볼 수 있도록 하였습니다.

목차

Introduction and preliminaries

The R environment

R 은 데이터의 관리, 수치 연산, 그리고 시각화를 통합적으로 지원하는 소프트웨어입니다.

효과적인 데이터 관리 및 저장 기능,행렬에 특화된 배열기반의 수치연산 기능들, 데이터 분석의 중간과정을 모두 확인할 수 있도록 지원하는 방대한 양의 기능들이 일관성이 있게 잘 통합되어 있으며, 데이터 분석 및 시각화된 결과물을 컴퓨터와 하드카피로 출력이 가능하도록 다양한 기능을 지원하고, 조건문, 반복문, 사용자 정의 함수, 및 입출력을 단순하면서도 효율적으로 제어할 수 있는 'S' 프로그래밍 언어를 제공합니다. (실제로 내장되어 있는 대다수의 함수들은 S 언어로 작성되어 있습니다). 데이터 분석을 위한 프로그램 작성시 제약사항이 많거나, 혹은 추가적인 개발 및 지원을 받아야만 하는 다른 분석도구들과는 달리, R은 논리적으로 설계가 매우 잘 되어 있는 "개발환경"을 지원하고자 합니다.

대화식 데이터 분석방법을 가지고 있는 R은 방대한 양의 packages (패키지)를 통하여 빠른 속도로 개발환경을 확장할 수 있고 새로운 방법론을 즉각적으로 적용할 수 있는 강점이 있으나, 단 하나의 데이터 분석을 위해 개발된 프로그램들이기에 범용성에 제한이 있다는 것을 기억해야 합니다.

Related software and documentation

독자는 R을 Bell 연구실에서 근무하던 Rick Becker, John Chambers 그리고 Allan Wilks 에 의해 재구성된 S 시스템으로 간주하셔도 무방합니다. 실은 S 언어는 S-Plus 를 구성하고 있는 기본 시스템이기도 합니다.

만약, S에 대해서 더 알고 싶으신 독자가 있으시다면, John Chambers 와 그의 동료들이 쓴 네권의 책을 찾아보세요. R 에 대해서는 Richard A. Becker, John M. Chambers and Allan R. Wilks 가 저술한 The New S Language: A Programming Environment for Data Analysis and Graphics 를 찾아보시길 바랍니다. 원본에 주석처리 되어 있는 내용에 따르면, S의 세번째 버전인 S3가 1988 년에 발표되었었다고 합니다. 1991년에 새로이 릴리즈된 S의 새로운 기능들에 대해서는 John M. Chambers and Trevor J. Hastie가 편집한 Statistical Models in S 를 참고하시길 바랍니다. methods에 내장되어 있는 메소드와 클래스들은 John M. Chambers 가 저술한 Programming with Data 를 토대로 하여 작성되었습니다. 더 자세한 사항은, See References.

현재 R을 이용한 데이터 분석과 통계학에 관련된 많은 서적들이 나와 있으며, S/S-Plus에 관련된 문서들 역시 R 을 사용하는데 도움이 될 것입니다. 그러나, 독자들은 S 언어가 R과 S-Plus에서 다르게 구현되어 있다는 것을 반드시 명심해야 합니다. 더 자세한 사항은 [See What documentation exists for R?] 문서를 참고하시기 바랍니다.

R and statistics

R의 개발환경에 대한 이 안내서에서는 statistics (통계학)을 다루고 있지 않지만, 많은 독자들이 R 을 통계 시스템으로서 활용하고 있습니다. 우리는 독자들이 R 이라는 개발환경을 과거부터 현대에 이르기까지 개발되어온 많은 통계학적 방법론을 한데 묶어 놓은 것으로서 이해하길 권장합니다. 일부 방법론들은 base 라는 R 개발환경안에 포함되어 있으나, 대부분의 방법론들은 packages 의 형태로서 제공되어 집니다. 기본

적으로 , “standard" 혹은 “recommended" 타입으로 제공되는 R 은 25개의 패키지로 구성되어 있고, 그 외의 패키지들은 인터넷 저장소인 CRAN 을 통하여 설치하실 수 있습니다. 인터넷 주소는 http://CRAN.R-project.org 입니다. 더 자세한 내용은 추후에 see Packages 라는 섹션에서 설명합니다.

예전부터 최근에 이르기까지 개발되어 온 대부분의 통계적 방법들을 R 에서 사용할 수 있으나, 이를 위해서는 약간의 노력이 필요할 수 도 있습니다.

독자들은 S (즉, R)이 다른 통계시스템들과 근본적으로 다르다는 것을 인지하고 있어야 합니다. 예를 들면, S에서의 통계분석은 기본적으로 일련의 분석 및 연산 과정을 통하여 얻은 결과물들을 object (객체) 라는 중간 매개체에 저장합니다. 따라서, SAS 혹은 SPSS가 회귀분석이나 판별분석을 수행하여 한 번에 방대한 양의 결과를 제공하는 것과는 달리, R 은 최소한의 결과물만을 제공하고, 더 자세한 중간 수행과정 및 결과물을 알고 싶다면 부가적인 R 함수들을 사용해야 합니다.

R and the window system

R 은 그래픽을 활용하는 windowing system 을 갖춘 환경에서 가장 손쉽게 활용할 수 있습니다. 우리는 이 안내서를 읽고 있는 사용자의 개발환경이 X window system 을 갖추고 있다고 가정합니다. 또한, 우리는 사용자의 운영체제가 UNIX 라고 간주하고 이 안내서를 작성하였으므로, 윈도우즈 혹은 Mac 을 사용하고 계신다면 몇가지 명령어에 대해서 약간의 도움이 필요할 것입니다. (몇가지를 제외한 모두의 명령어들은 운영체제에 관계없이 사용할 수 있습니다. 만약, 안내서에서 기술하고 있는 명령어의 사용법이 UNIX 와 윈도우즈에서 다르다면 번역자가 최대한 배려할 것입니다).

최적화된 성능의 R 을 사용하고 싶으시다면 약간의 기술적인 조정이 필요하지만, 이 안내서는 이러한 기술적인 부분에 대해서는 다루지 않습니다.

Using R interactively

먼저, R 프로그램을 실행시키면, 사용자의 입력을 기다리는 표시 (‘〉’)를 볼 수 있을 것입니다. 이 표시를 프롬프트라고 하고, 사용자가 원한다면 다른 모양의 프롬프트를 사용할 수 도 있습니다. 그러나, R 에서 사용되는 프롬프트는 유닉스의 쉘 프롬프트와 표시가 동일하므로, 우리는 내용의 명확성을 위해서 유닉스의 프롬프트를 ‘$’ 로 표시할 것입니다.

유닉스내에서 R 을 실행시키기 위해서는 다음을 따라해주세요.

먼저 R 에서 사용할 데이터 파일들이 저장된 곳에 work 라는 서브 디렉토리를 생성합니다. 이 서브 디렉토리를 working directory (워킹 디렉토리)라고 하며, 이 워킹 디렉토리에서 여러분은 R 을 이용하여 통계분석을 하게 될 것입니다.

```
$ mkdir work
$ cd work
```

R 프로그램을 실행시키위해서 다음의 명령어를 입력하세요.

```
$ R
```

이제부터 여러분들은 R 명령어들을 이용하여 작업을 하실 수 있습니다. R 프로그램을 종료하기 위해서는 다음의 명령어를 사용합니다.

```
> q()
```

종료 명령어를 입력하면 R 프로그램은 여러분에게 현재 R 세션 내에서 작업을 하고 있던 데이터를 저장 (yes), 저장안함 (no) 혹은 종료 명령을 취소 (cancel) 할 것인지를 물어볼 것입니다. 이 저장여부의 메시지는 시스템에 따라서 대화창으로 보여질 수 도 있고, 종료명령어 다음줄에 텍스트 형식으로 나타날 수 도 있습니다. 만약 저장을 선택한다면, 저장된 데이터는 다음번 R 프로그램을 실행했을때 다시 사용할 수 있습니다.

새로운 R 세션을 시작하는 것은 간단합니다.

위에서 R 프로그램을 실행했던 것과 같이 work 라는 워킹 디렉토리를 만들고 R 이라고 명령어를 쉘 프롬프트에 입력하면 됩니다.

```
$ cd work
$ R
```

R 프로그램을 사용한 뒤, 종료를 하고 싶다면 세션 마지막에 q() 이라고 입력하면 됩니다. 윈도우즈에서 R의 실행은 유닉스에서의 실행방법과 동일합니다. 워킹 디렉토리와 동일한 개념의 폴더를 생성한 뒤에, Start In 메뉴에서 R 바로가기를 바탕화면에 생성합니다. 이제 바탕화면의 바로가기 아이콘을 이용하여 R을 실행시킬 수 있습니다.

An introductory session

R을 처음 접하여 익숙하지 않은 사용자들에게는 A sample session 에 있는 introductory session 을 꼭 따라해 보실 것을 권합니다.

Getting help with functions and features

R은 UNIX 에서 man 이란 명령어를 이용하여 도움말을 볼 수 있는 것과 같은 기능을 제공합니다. 예를 들어 solve라는 함수의 사용법이 궁금할 경우에는 다음의 명령어를 입력합니다.

```
> help(solve)
```

다음과 같은 방법으로도 동일한 도움을 얻을 수 있습니다.

```
> ?solve
```

special character (특수문자)의 기능들을 알고 싶을 때에는 반드시 큰 따옴표 혹은 작은 따옴표를 함께 사용하여 “character string" 형태를 갖추어 주어야 합니다.

```
> help("[[")
```

그러나 우리는 큰 따옴표를 사용하기를 권장합니다. 그 이유는 "It's important" 와 같은 문자열을 표현할 경우도 있기 때문입니다.

R을 설치할 때 HTML 형식의 도움말도 함께 설치되며 아래와 같이 입력하면 그 내용을 볼 수있습니다

```
> help.start()
```

위 명령에 따라 웹 브라우저가 실행되며 하이퍼링크로 구성된 도움말 화면을 볼 수 있습니다.help.start()로 로딩된 페이지에 있는 'Search Engine and Keywords' 링크를 통해 나타나는 페이지가 특히 유용한데 여기에서 키워드, 타이틀, 개념, 함수 등에 관한 검색이 가능합니다. 이를 통해 자신의 R 시스템에 대한 이해 정도를 확인하고 R이 제공해 줄 수 있는 것이 무엇인지 알 수 있습니다.

help.search 명령(줄여서 ??)은 좀더 다양한 방식으로 도움말을 검색할 수 있게 해 줍니다. 예를 들어,

```
> ??solve
```

더 자세한 내용과 예제를 찾아보기 위해서 ?help.search 기능을 사용해 보세요.

특별히 찾고자 하는 주제와 관련된 예제들은 다음의 명령어를 통하여 찾아 볼 수 있습니다.

```
> example(topic)
```

윈도우즈 버전의 R은 또 다른 방법의 도움말 찾기 기능이 있습니다. 이에 대해서 더 알고 싶으시다면 다음의 명령어를 입력하세요.

```
> ?help
```

R commands, case sensitivity, etc.

R은 아주 간단한 문법을 가진 expression language (문자식을 기초로 하는 언어) 입니다.

제일 먼저, 유닉스를 기반으로 하는 다른 패키지들과 같이 대소문자를 구별하기 때문에 A 와 a는 서로 다른 문자로 인식되어 서로 다른 변수를 가리킬 수 도 있습니다.

R 문자식에 사용될 수 있는 기호들은 사용자의 운영체제 및 언어 설정 (locale) 에 따라 달라질 수 있습니다. 일반적인 문자식에는 알파벳 (A-Za-z) 과 숫자 (0-9)의 조합한 형태가 이용되며, '.' 혹은 '_' 를 문자식안에 함께 사용할 수 도 있습니다. 단, 문자식의 첫 문자가 '.' 이라면 두번째 문자는 반드시 숫자가 와서는 안됩니다. R 2.13.0 이전 버전들 은 표현식의 길이가 256 bytes 를 넘을 수 없었으나, R 2.13.0 버전부터는 제약이 없습니다.

기본적인 명령어란 expressions (문자식) 과 assignment (할당) 이라는 두가지 요소로 구성됩니다. 만약 문자식이 명령어처럼 할당없이 홀로 쓰여진다면, 이는 명령어처럼 사용되거나, 어떠한 값도 문자식에 저장되지 않습니다. 그러나, 문자식에 할당이 이루어진다면, 문자식은 할당된 특정한 값을 가지게 되며, 그 할당의 결과는 보여지지 않습니다.

일반적으로 명령어들은 세미콜론 (';') 에 의해서 구분이 되거나, 새로운 행에 의해서 구분되어 집니다. 이러한 명령어들은 중괄호 ('{ ' 와 '} ') 를 이용하여 한개의 작업단위를 구분할 수 있도록 그룹을 형성할 수 있습니다.

Comments (주석)은 프로그램상의 어느 곳에나 놓일 수 있으며, '#' (해쉬마크) 을 이용하여 행 전체를 주석처리 합니다.

만약 명령어가 현재의 행에서 끝나지 않고, 다음 행으로 넘어간다면 R 은 다음의 프롬프트 기호를 이용하

여 명령어가 끝나지 않았음 (continuation)을 표현할 것입니다.

```
+
```

콘솔내에서 입력될 수 있는 최대 명령어의 길이는 4095 bytes 입니다

Recall and correction of previous commands

R은 이전에 사용되었던 명령어들을 다시 불러와서 재실행을 시킬 수 있는 기능을 가지고 있습니다. 이전에 실행된 명령어들의 기록 (command history)을 확인하기 위해서는 단순히 키보드의 세로방향의 화살표를 이용하며, 재실행하고자 하는 명령어를 찾았다면 가로방향의 화살표 키를 이용하여 변경하고자 하는 위치에 커서를 두고 'DEl' 키를 사용하면 명령어를 지우거나 다른 키를 이용하여 추가 및 변경도 가능합니다. 더 자세한 사항은 see The command-line editor.

만약, 독자가 UNIX 상에서 작업을 한다면 readline 이라는 라이브러리를 이용하여 사용했던 명령어들을 다시 불러들이고, 편집할 수 있습니다. 또한, Emacs 라는 텍스트 에디터 는 ESS (즉, Emacs Speaks Statistics) 라는 보조시스템과 함께 R 을 파워풀하게 사용할 수 있는 환경을 제공해줍니다. 이에 대해서는 See R and Emacs.

Executing commands from or diverting output to a file

예를들어, 작업을 하려면 R 명령문들을 포함하는 commands.R 이라는 이름을 가진 파일이 현재 워킹 디렉토리인 work 에 저장이 되어 있다면, 언제든지 다음의 명령어를 이용하여 현재 실행되고 있는 R 세션내에서 실행시킬수 있습니다.

```
> source("commands.R")
```

Source 라는 명령어는 윈도우즈에서도 마찬가지로 File 메뉴를 통하여 사용될 수 있습니다.

```
> sink("record.lis")
```

위에서 사용된 sink 라는 함수는 record.lis 이라는 외부파일을 생성하고, 콘솔에서 출력되는 모든 결과물을 record.lis 에 저장합니다. 이때, 콘솔에서는 아무런 결과를 출력하지 않습니다.

```
> sink()
```

sink라는 명령어를 다시 입력하면 결과물을 다시 콘솔내에서 확인할 수 있으며, 결과물들은 더 이상 record.lis 에 저장되지 않습니다.

Data permanency and removing objects

R 프로그램을 작성하면서 생성되어지고 다루어지는 모든것들을 객체 (objects) 라고 합니다. 변수 (variables), 숫자형 배열 (numerical array), 문자열 (character strings), 함수 (functions), 그리고 이러한 것들로 이루어진 좀 더 일반적인 형태의 모든 것이 객체에 해당합니다.

R 세션내에서, 객체는 주어진 문자식 (혹은 name) 에 의하여 생성되고, 저장되어 집니다. (우리는 이것에

대해서 다음 세션에서 더 자세히 설명할 것입니다). 현재 R 세션내에서 사용되는 객체들을 확인하고자 한다면 다음의 명령어를 사용해보세요.

```
> objects()
```

ls() 라는 명령어 역시 현재 R 세션내에서 사용되는 모든 객체들의 이름을 보여줍니다. 그리고, 이렇게 모든 객체를 저장하고 있는 곳을 workspace 라고 합니다.

현재 작업하고 있는 workspace 내의 객체들은 rm 이라는 명령어를 이용하여 삭제할 수도 있습니다.

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

현재 R 세션내에서 생성되고 작업하던 모든 객체들은 .RData 라고 불리는 파일에 영구적으로 저장되고, 사용되었던 모든 명령어들은 .Rhistory 라는 또 다른 파일에 따로 저장되어 다음 번 R 세션이 새로이 시작되었을때 불러내어 다시 사용될 수 있습니다.

만약, 같은 디렉토리 안에서 R 이 다시 실행된다면, R은 자동적으로 이전에 세션에서 사용되었던 객체와 명령어들을 .RData 와 .Rhistory으로부터 불러들입니다.

우리는 R을 이용하여 통계분석을 하고자 하는 독자들에게 여러개의 다른 이름의 워킹 디렉토리를 생성하여 작업하기를 권장합니다. 그 이유는 x 와 y 같은 객체이름을 사용하는 것은 매우 흔하며, 이런 명칭을 이용하여 분석을 하는것은 단순한 작업에는 빠르게 작업할 수 있으나, 복잡한 작업을 수행할 경우 동일한 워킹 디렉토리내에서 이러한 객체이름을 사용하는 것은 큰 혼란을 초래하기 때문입니다.

Simple manipulations; numbers and vectors

Vectors and assignment

R은 data structure (데이터구조)를 바탕으로 작동합니다. 이것의 가장 단순한 형태는 일련의 숫자가 하나로 묶여있는 수치형 벡터 (vector) 입니다. 예를들어, 10.4, 5.6, 3.1, 6.4, 그리고 21.7 이라는 일련의 숫자로 구성된 x 라는 이름을 가진 벡터를 생성하기 위해서는 다음과 같은 R 명령어를 입력합니다.

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

위에서 사용된 문자식을 assignment statement (할당문) 이라고 하며, c() 라는 함수 (function) 를 이용하여 입력되어진 숫자들을 하나의 벡터 형태로 묶어주는 역할을 하고 있습니다. 이때 함수 c() 안에서 컴마로 구분된 개별 숫 하나하나를 함수의 인자 (argument) 라고 하고, 이 예제에서는 숫자 하나하나가 각각 길이가 1인 벡터로서 간주되었습니다.

또한, 띄어쓰기 없이 '<' 와 '-'를 한데 묶어 사용된 '<-' 라는 기호는 할당연산자 (assignment operator) 라고 불리며, 문자식에 의해 생성된 객체의 값을 지시하도록 한다는 것을 의미합니다. 대부분의 경우, '=' 라는 기호를 사용하기도 합니다.

할당연산은 또한 assign() 이라는 함수를 이용하여 이루어 질 수 도 있습니다. 위에서 수행한 동일한 할당연산을 아래와 같이 수행할 수 있습니다.

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

여러분들은 <- 라는 기호를 단순히 할당연산을 위한 약속이라고 기억하시면 될 것입니다.

또한, 기호의 방향을 바꾸어 아래와 같은 방법으로도 할당문을 표현할 수 있습니다.

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

만약 이러한 문자식이 완전한 형태의 명령문 형식으로 사용된다면 결과물은 출력이되나, 값들을 잃어버리게 됩니다.

만약 우리가 다음의 명령문을 실행한다면,

```
> 1/x
```

이것은 x 에 할당되어 있는 5개의 수치에 대하여 각각의 역수를 터미널을 통하여 보여줄 것입니다. 이때, x 의 값은 변하지 않습니다.

다음의 할당문은 2개의 x 벡터사이에 0 이 끼워진 형태로 총 11개의 일련의 숫자로 이루어진 y 라는 새로운 벡터를 생성해냅니다.

```
> y <- c(x, 0, x)
```

Vector arithmetic

벡터 (vector) 는 요소 (element) 단위의 산술 연산에 활용될 수 있습니다. 이때, 꼭 벡터의 길이 (length)가 같을 필요는 없습니다. 여기에서 길이란 벡터를 구성하는 요소의 개수입니다. 만약, 길이가 다르다면 상대적으로 짧은 길이를 가지는 벡터는 길이가 긴 벡터와의 길이를 맞추기 위해서 자동으로 recycled (반복)되어지게 됩니다. 만약, 벡터와 벡터의 길이가 1개인 상수 (constant)간의 연산시에는 상수를 벡터의 길이만큼 늘여 맞춘후에 요소단위로 연산을 수행합니다.

```
> v <- 2*x + y + 1
```

위의 명령어에 대하여 R은 다음과 같이 연산을 수행합니다. R 따라서 x 는 2.2배 반복될 것입니다. x 앞에 있는 상수 2 는 11 번 반복된 후 각각의 요소끼리 연산 된 후, y 와의 연산을 수행합니다. 마지막으로 상수 1 역시 11번 반복 된 후 이전 연산의 결과와 새로운 연산을 요소 단위로 수행한 뒤 그 결과를 v 에 할당합니다.

기초 산술 연산자는 +, −, *, / 그리고 ^ (자승) 입니다. 또한, R은 log, exp, sin, cos, tan, sqrt 와 같은 수학에서 자주 쓰이는 모든 산술에 관계된 함수를 제공합니다.

max 와 min는 벡터가 가지는 요소들을 확인하여 최대값과 최소값을 알려주는 함수입니다.

range 는 주어진 벡터 x 의 최소값과 최대값을 알려주는 c(min(x), max(x)) 형식으로 알려주는 함수입니다. length(x) 는 벡터 x의 길이, 즉, 요소의 개수를 알려주는 함수 입니다. sum(x) 는 x의 모든 요소들을 더한 값을 알려주는 함수이고, prod(x) 는 x의 모든 요소들을 곱한 값을 알려주는 함수입니다. mean(x) 는 벡터 x가 가지는 모든 요소들에 대한 평균 (sample mean)을 알려주는 함수이며, sum(x)/length(x)와 동일한

값을 가집니다. 또한, var(x) 는 벡터 x 가 가지는 모든 요소들에 대한 분산 (sample variance)을 알려주는 함수이고, 아래와 같은 연산을 통하여 얻은 값과 동일합니다.

```
sum((x-mean(x))^2)/(length(x)-1)
```

만약, var() 함수에서 괄호안에 받아들이는 인자가 n-by-p 크기의 행렬이라면, var() 은 p-by-p 크기의 공분산 (sample covariance matrix) 를 계산해줍니다. 여기에서 n-by-p 라는 것은 n개의 관측대상들로부터 p개의 독립변수를 측정한 값들을 모아 놓은 값들의 모임을 수학적으로 표현한 것이며, p 개의 독립변수를 가졌다고 하고, 각각의 독립변수들은 n 개의 관측치를 가졌다고 해석합니다.

sort(x) 는 벡터 x를 인수로 받아 x 의 모든 요소들을 오름차순 (ascending order, 즉 작은 거에서 큰거로)으로 재 배열하여 얻어진 벡터를 돌려줍니다. 만약, 인자로 사용된 x 벡터가 오름차순으로 정렬된 순서를 알고 싶을 경우에 order() 혹은 sort.list()란 함수를 사용하면 알 수 있습니다. pmax 와 pmin 이라는 함수는 여러개의 벡터로부터 각각의 포지션에서 병렬 (parallel) 으로 max 와 min 을 이용한 최대 최소값을 알려줍니다. 대부분의 경우 사용자들은 수치형 벡터 (numeric vector) 에 사용되는 요소가 정수(integer), 실수 (real), 혹은 복소수 (complex) 인지를 신경쓰지 않습니다. R 은 내부적으로 입력된 벡터가 실수라면 double형 실수로, 복소수라면 double 형 복소수로 연산을 수행합니다. 단, 복소수를 연산을 수행하고자 할 때는 복소수 부분을 정확히 명시해 주어야 합니다. 다음의 예는 이러한 경우를 보여줍니다.

```
sqrt(-17)
```

루트안에 음수가 사용되었으므로, R 은 NaN 이라는 경고를 해줄 것입니다. 여기에서 NaN 은 Not a Number (숫자가 아냥) 이라는 메시지입니다.

```
sqrt(-17+0i)
```

루트안에 정확하게 복소수를 표현해 준다면 R 은 에러없이 연산을 수행합니다.

Generating regular sequences

R 은 시퀀스 (혹은 수열) 를 다양한 기능을 가지고 있습니다. 시퀀스란 일련의 숫자가 어떤 규칙에 의해서 나열된 벡터의 형태를 의미합니다. 예를 들어, 1:30 이란 명령어는 c(1,2, ..., 29, 30) 이란 벡터를 생성합니다.

콜론 (colon) 연산자는 어떤 수식보다도 최우선권을 가집니다. 예를 들면, 2*1:15 라는 명령어는 c(2,4, ..., 28, 30) 이라는 벡터를 생성합니다. 그럼, n <- 10 을 명령하고 나서 1:n-1 와 1:n-1 라는 두 시퀀스를 비교해 보시길 바랍니다.

또한, R 은 역으로 나열되는 수열도 30:1 이라는 명령을 통해서 표현할 수 있습니다.

seq() 라는 함수는 시퀀스를 좀 더 일반적인 방식으로 생성하는 기능을 가지고 있습니다. seq() 함수는 5개의 인자를 가지는데, 처음 두개의 인자는 시퀀스의 처음값과 마지막값을 지정합니다. 만약, 두개의 인자만 입력되었다면, 이는 콜론 연산자를 사용한 것과 동일한 벡터를 생성합니다. 즉, seq(2,10) 에서 생성된 시퀀스는 2:10 와 동일합니다.

R에서 제공하는 모든 함수는 사용자가 인자 입력의 실수를 피하기 위해서 다음과 같은 기능을 지원합니다. 예를 들어, seq(1,30) 이라는 시퀀스를 말로 표현하면 제일 처음 값이 1 이고, 맨 마지막값이 30 인 수의 나열이라고 표현할 수 있으므로, R은 사용자가 seq() 이라는 함수를 사용할 때 실수를 범하지 않도록 맨 처음

인자는 from=value 이고, 두번째 인자는 to=value 라는 인자명 (named form) 을 이용한 입력 기능을 제공합니다. 즉, seq(1,30)과 seq(from=1, to=30), 그리고 1:30 모두 같은 시퀀스를 생성합니다. 여지까지, seq() 란 함수의 처음 2개의 인자만을 설명했습니다. 다음 2개의 인자는 by=value와 length=value입니다. by=value 는 시퀀스에서 나열된 숫자들간의 차이 (step size) 를 지정하고, length=value 는 시퀀스의 길이를 지정합니다. 만약, 사용자가 by=value 를 지정하지 않는다면, R 은 기본적으로 by=1 로 가정한 상태에서 시퀀스를 생성합니다.

예를 들어, 다음의 명령어는 c(−5.0, −4.8, −4.6, ..., 4.6, 4.8, 5.0) 이라는 시퀀스 객체를 생성한 뒤, s3 라는 변수명을 가진 객체에 할당합니다.

```
> seq(-5, 5, by=.2) -> s3
```

이와 비슷한 논리로, 다음의 명령어는 s3 과 동일한 시퀀스를 가지는 s4 란 객체를 생성해 낼 수 있습니다.

```
> s4 <- seq(length=51, from=-5, by=.2)
```

seq() 함수의 다섯번째 인자는 along=vector 입니다. 이것은 다른 인자들의 입력없이 독립적으로 사용되어지며, 1, 2, ..., length(vector) 라는 시퀀스를 생성해 냅니다.

R 은 seq()와 비슷하지만 약간 다른 기능의 rep() 이란 함수를 제공합니다. 이것은 매우 다양한 방법으로 객체를 반복시킬 수 있습니다. 가장 간단한 예제는 다음과 같습니다.

```
> s5 <- rep(x, times=5)
```

위의 명령어는 x 란 벡터를 5번 반복시켜 생성한 시퀀스 객체를 s5 에 할당하라는 의미입니다. 또한, 아래의 명령어는 또 다른 사용법을 보여주고 있습니다.

```
> s6 <- rep(x, each=5)
```

이것은 x란 벡터가 가지고 있는 각각의 요소들을 5번씩 반복한 시퀀스를 생성하란 의미입니다.

Logical vectors

수치형 벡터와 마찬가지로 R은 논리형 연산 또한 가능합니다. 논리형 벡터란 벡터의 요소들이 TRUE, FALSE, 혹은 NA 와 같은 논리형 값을 가지는 것을 말합니다. 여기에서 NA 는 Not applicable (판별불능 혹은 적용불가능) 을 의미합니다. 또한, TRUE 와 FALSE 는 약어로 T 혹은 F 사용해도 무방합니다. 그러나, T 혹은 F 은 사용자에 의한 변수명으로도 사용될 수 있기 때문에, R 은 TRUE 와 FALSE 이라는 예약어로서 참과 거짓을 구분합니다.

논리형 벡터는 조건문 (conditions) 에 의해서 생성됩니다.

예를 들면, 다음의 명령문은 x 벡터가 가진 각각의 요소들이 13 보다 크다는 조건을 만족하면 TRUE, 만약 만족하지 않는다면 FALSE 를 저장한 새로운 벡터의 개체를 temp 라는 변수에 할당하게 합니다.

```
> temp <- x > 13
```

R은 <, <=, >=, ==, != (같지 않음) 과 같은 논리연산자를 제공합니다. 또한, R은 “and” (그리고) 와 “or”

(또한) 같은 논리연산을 통하여 집합의 연산을 가능하게 합니다. “and” 는 & 로 표시하고, “or”는 | 로 표시합니다. 예를들어, c1과 c2 가 논리형 벡터라면, 합집합은 c1 & c2, 교집합은 c1 | c2, c1의 컴플리먼트 (한국말 모름)는 c1 로 표현할 수 있습니다. R 은 논리연산자를 이용하여 산술연산 또한 할 수 있습니다. R 은 내부적으로 TRUE 와 FALSE 이라는 값을 1 과 0 으로 강제변환 (coerced) 된 숫자형으로 인식하기 때문입니다. 그러나, 때로는 이러한 논리연산이 수치연산과 맞지 않는 경우가 있는데, 이에 대해서는 다음 섹션에서 다루겠습니다.

Missing values

가끔 벡터를 구성하는 요소의 값을 알 수 없는 경우가 있습니다. 통계학적인 의미에서 이런 경우는 두가지가 있습니다. 하나는 어떤 연산의 결과를 얻지 못한 경우이고, 또 다른 하나는 데이터 수집시의 데이터 결측인 경우입니다. 두 가지 모두 사용자가 벡터에 넣어주어야 할 값을 사전에 알 수 없기 때문에 Not Available (정보가 없엉) 라는 의미의 약자인 NA 로서 표기합니다. 따라서 벡터 x 에 NA 값이 있는지 확인하고 싶은 경우에 여러분은 R 프로그램에 다음과 같이 물어볼려고 생각할 것입니다. 벡터 x가 NA야? = Is x NA?. 즉, is.na(x) 라는 함수는 벡터 x 의 모든 구성요소를 확인하여 값이 있다면 TRUE를, 없다면 FALSE라는 값을 가지고 있는 새로운 벡터를 생성해줍니다. 다음의 명령문을 이용하여 확인해 보세요.

```
> z <- c(1:3,NA);  ind <- is.na(z)
```

그러나, NA는 단순히 벡터를 구성하는 요소의 값이 있는지 없는지를 나타내는 마커 (혹은 인디케이터) 이므로, x == NA 라는 논리적 표현은 is.na(x) 와 다르다는 것을 알고 있어야 합니다. 따라서, x == NA 란 논리연산은 수행되어 질 수 없으므로 모든 요소의 값이 NA을 가지는 벡터를 생성하게 될 것입니다.

사용자는 NA와 NaN 을 혼돈하지 말아야 합니다. 이는 산술연산시 알 수 있는데, NaN 은 어떤 연산이 수학적인 조건을 만족하지 못하기 때문에 계산할 수가 없으므로 값을 생성하지 못하는 경우를 의미합니다. 따라서, NaN 은 Not a Number (숫자가 아냥) 란 말입니다. 반면에, NA 는 사용자로부터의 정보가 없기 때문에 결측처리의 의미로서의 Not available 이라고 합니다.

다음 명령어는 언제 NaN 이 나옴을 알 수 있습니다.

```
> 0/0
```

또한 Inf 역시 NaN 에 속합니다. 여기에서 Inf 란 infinite 의 약자로 수학적 값이 무한대, 즉 정확한 값을 지정할 수 없을때 사용합니다. 따라서 Inf (양의 무한값) 그리고 -Inf (음의 무한값) 은 NaN 에 해당합니다.

```
> Inf - Inf
```

is.na(xx) 의 경우와 마찬가지로 x 벡터가 NaN 값을 가지고 있는 경우를 확인하고 싶다면 is.nan(xx) 란 함수를 이용하시면 됩니다. 어떤 경우에는 결측치가 〈NA〉로 표시될 때가 있습니다. 이경우는 결측치가 숫자가 아닌 문자형이기 때문입니다.

Character vectors

문자열은 주로 R 에서 그래픽처리시 라벨링을 할 때 자주 사용되며, 큰 따옴표를 사용하여 "x-values" 혹은 "New iteration results" 사용하여 R 프로그램이 큰 따옴표 안에 있는 내용을 문자열로 인식할 수 있게 해줍니다. 그러나, 결과물의 출력시에는 큰 따옴표는 보여지지 않습니다. 이것은 R 내부적으로 C 언어와 같은 이스케이프 시퀀스 (escape sequence) 를 사용하여 문자열을 처리하기 때문입니다. 따라서 큰 따옴표,

작은 따옴표, 혹은 백슬래쉬 등 숫자나 문자가 아닌 그 외의 기호들을 표시하기 위해서는 원하는 기호 앞에 ₩ 기호를 붙여 같이 사용해야 합니다. 유용한 이스케이트 시퀀스는 ₩n (다음행으로 이동), ₩t (탭 넣기), ₩b (백스페이스) 같은 것들이 있습니다. 만약, 더 알고 싶으시다면 ?Quotes 를 입력해 보세요. 모든 이스케이프 시퀀스의 리스트를 확인 할 수 있습니다.

또한 c() 함수를 이용하여 새로운 문자열을 생성해 낼 수도 있습니다. 이것에 대한 예제는 앞으로 보게 될 것입니다.

paste() 라는 함수는 임의의 개수의 인자를 입력받아 이들을 한데 묶어 하나의 문자열을 생성해 줍니다. 이 때, 인자가 숫자형일지라도 문자형으로 강제변환됩니다. paste() 함수는 입력받은 인자를 하나로 묶어 출력할 때 한개의 빈 공간 (white space 또는 blank space) 을 각각 인자사이에 추가하여 보여줍니다. 만약, 이 화이트 스페이스를 이용한 출력을 원하지 않는다면, sep=string 이라는 옵션을 사용하여 변경할 수 도 있습니다. 다음 명령어를 통하여 어떻게 문자열이 처리되는지 확인해 보세요.

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

여기에서 paste() 란 함수는 c("X", "Y") 라는 문자형 벡터와 1:10 이라는 숫자형 벡터를 한 데 묶을 것인데, 두개의 길이가 다르므로 c("X", "Y") 는 5 번 반복 될 것입니다. 따라서, 위의 명령어는 다음의 명령어를 수행한 것과 동일합니다.

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Index vectors; selecting and modifying subsets of a data set

벡터의 일부분만을 이용하여 분석 작업을 하고자 한다면, 벡터의 이름 바로 뒤에 대괄호 ([]) 와 함께 인덱스 벡터 를 활용하시길 바랍니다. 인덱스 벡터를 활용하는 방법은 다음과 같습니다.

논리연산값을 가지는 인덱스 벡터. 인덱스 벡터가 TRUE 혹은 FALSE 와 같은 논리형 값을 가지는 것을 의미합니다. TRUE 값에 해당하는 원래의 벡터의 요소만을 남겨두고, FALSE 에 해당하는 요소는 삭제합니다. 다음의 예제는 벡터 x 의 구성요소 중 NA가 아닌 값만을 선택하여 새로운 벡터 y 를 생성합니다. 따라서, 만약 벡터 x가 NA 값을 포함하고 있다면, 생성된 벡터 y 의 길이는 벡터 x 의 길이보다 잛을 것입니다.

```
> y <- x[!is.na(x)]
```

다음의 예제는 벡터 x의 구성요소들의 값이 NA이 아니고, 양수인 구성요소들만 고른 후, 골라진 요소들에 각각 1 을 더하여 새로운 벡터 z 를 생성합니다. 여기에서 인덱싱 연산자 (대괄호) 는 어떠한 산술연산 (즉, 소괄호) 보다도 우선합니다.

```
> (x+1)[(!is.na(x)) & x>0] -> z
```

양의 정수로만 이루어진 인덱스 벡터 여기에서 인덱스란 벡터의 구성요소의 위치를 나타내므로 정수입니다. 따라서 인덱스 벡터가 가질 수 있는 구성요소의 최대값은 lenght(x) 와 같습니다. 즉, 다음의 예제는 벡터 x가 가지는 구성요소중 첫번째부터 열번째에 해당하는 요소만을 선택하여 새로운 벡터를 생성하는 것을 의미합니다.

```
> x[1:10]
```

또한 다음의 예제처럼 응용하여 사용할 수 있습니다. rep(c(1,2,2,1)) 은 "x", "y", "y", "x" 와 동일한데, 이를 4번 반복하라는 옵션을 주었으므로, 이것은 총 16개의 구성요소를 가진 벡터를 생성할 것입니다.

```
> c("x","y")[rep(c(1,2,2,1), times=4)]
```

음의 정수로 구성된 인덱스 벡터 이것은 선택한 인덱스를 제외하고자 할 때 사용되어 집니다. 다음의 예제는 벡터 x 가 가지는 구성요소의 첫번째부터 다섯번째까지 뺀 그 나머지를 한데 묶어 새로운 벡터 y 를 생성하는 것을 의미합니다.

```
> y <- x[-(1:5)]
```

문자열로 이루어진 인덱스 벡터 이것은 인덱스 벡터를 활용하는 특수한 경우에 해당합니다. 다음의 예제를 살펴보면, fruit 이라는 수치형 벡터에 names() 라는 함수를 이용하여 각각의 구성요소에 "orange", "banana", "apple", "peach" 를 부여하였습니다. 그리고, 당신이 fruit 이라는 벡터의 세번째와 첫번째 구성요소를 주어진 이름 (즉, 문자열) 을 이용하여 선택하고자 할 경우에 사용되는 인덱스 벡터를 의미하는 것입니다.

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banana", "apple", "peach")
> lunch <- fruit[c("apple","orange")]
```

이렇게 문자열을 이용하여 벡터를 활용하는 것은 정수를 이용한 인덱싱 방법보다 편리합니다. R 은 이 기능을 다음장에서 보게 될 data frame (데이터 프레임) 에 활용합니다.

다음은 인덱스연산자를 활용하는 한 가지 방법입니다.

```
> x[is.na(x)] <- 0
```

이것은 벡터 x의 구성요소들 중에 NA 값을 가지는 것들을 0 값으로 대체 합니다.

```
> y[y < 0] <- -y[y < 0]
```

이것은 아래에 표현된 abs() 함수를 사용한 것과 동일합니다.

```
> y <- abs(y)
```

Other types of objects

벡터는 R 시스템에 있어서 가장 중요한 객체이므로, 벡터를 활용하는 방법을 이해하는 것이 핵심입니다. 앞으로 우리가 다루게 될 섹션들에서는 다른 형태의 개체에 대해서 설명할 것이나, 실은 모두 벡터를 기반으로 하고 있습니다.

matrices (행렬) 또는 일반적으로 arrays (배열)이라는 것은 단순히 다차원 벡터를 의미하는 것입니다. 벡터란 숫자를 일렬로 나열해 놓은 것을 한데 묶어 놓은 1차원적인 형태입니다. 그러나, 행렬과 배열 이라는 것은 동일하게 1 개의 벡터이나, 이를 행과 열, 그리고 차원이라는 추가적인 요소를 이용하여 다차원적으로 표현한 것을 말합니다. 배열과 행렬에 대해서는 See Arrays and matrices. 통계자료 입력 및 분석시, 벡터에 입력된 자료가 연속형 (즉, 실수형) 일수도 있지만 카테고리형일 수 도 있습니다. factors 라는 것은 이러한

카테고리형 데이터를 다룰때 데이터가 카테고리 타입이라는 것을 R 이 인식하도록 합니다. 자세한 내용은 See Factors. lists 라는 것은 벡터의 가장 일반적인 형태로서, 벡터의 구성요소가 될 수 있는 형태에 제한이 없습니다. 여지까지, 설명한 벡터는 모두 동일한 형태의 숫자 혹은 문자형의 구성요소를 가졌습니다. 그러나, lists (리스트) 란 리스트의 구성요소가 문자열일 수 도 있고, 숫자형일 수 도 있으면, 행렬이 입력될 수 있도 있습니다. 올바른 리스트의 활용은 굉장히 유연한 통계연산을 할 수 있도록 도와줍니다. 이에 대해서는 See Lists. data frames (데이터 프레임) 이라는 것은 행렬과 같은 형식을 가지고 있으나, 각각의 열이 1 개의 독립적인 다른 벡터로 이루어진 형태입니다. 이는 통계 데이터 입력 및 분석에 활용되는 가장 일반적인 형태의 데이터 행렬이라 해도 무방합니다. 이에 대해서는 See Data frames. functions (함수) 라는 것은 그 자신이 워크스페이스에 저장되어 있는 일종의 객체입니다. 따라서, 함수가 현재 작업하고 있는 워크스페이스에 있다면, R 은 이를 인식하고 분석에 활용할 수 있습니다. 이러한 개념은 R에서 사용자가 직접 사용자 정의 함수를 작성하여 쓸 수 있도록 하고, 또한 패키지 (즉, 추가적인 함수들의 집합) 을 설치하여 R 을 사용자측에서 편리하게 확장할 수 있는 것입니다. 사용자 정의 함수에 대해서 알고 싶으시다면, See Writing your own functions.

Objects, their modes and attributes

Intrinsic attributes: mode and length

R은 objects (객체)라는 것을 시스템 내부에서의 모든 정보처리 (연산 및 수행) 의 기본단위로 합니다. 따라서, 객체에 대한 이해는 R 을 사용함에 있어서 필수적이기 때문에 , 이번섹션에서는 객체에 대해서 구체적으로 다루도록 하겠습니다.

먼저, 여지까지 우리가 언급해 온 다양한 종류의 벡터들은 객체의 한가지 예입니다. 이미 눈치를 챈 독자가 있겠지만, 벡터의 종류는 벡터의 구성요소가 가지는 데이터의 유형 (mode) 에 의해서 실수형 (numeric), 복소수형 (complex), 논리형 (logical), 문자형 (character) 으로 나뉘어집니다. 이때, 벡터의 구성요소가 가지는 값의 유형을 모드 (mode) 라고 하며, 이 것은 또한 객체를 구성하는 최소단위 기본형 (atomic) 형태라고 지칭합니다. 따라서, 벡터는 그 구성요소들이 모두 동일한 모드를 가지고 있어야 합니다. (단, 이 규칙에 대해서 Quantities Not available (수치를 알수가 없엉) 를 의미하는 NA 는 예외입니다).

또한, 벡터는 공집합과 같이 아무런 요소를 가지지 않을 수 도 있는데, 이 또한 모드를 가진다는 사실을 알고 있어야 합니다. 만약 문자열 벡터가 아무런 요소를 가지지 않는다면 character(0) 라고 표시될 것이고, 실수형이라면 numeric(0) 라고 나타납니다.

R 에서 사용되는 list 라는 객체는 구성요소의 모드가 list 자신인 특수한 형태의 벡터입니다. 즉, list 는 각각의 구성요소가 다른 종류의 모드를 가지는 객체들을 한데 묶은 특수한 형태의 객체인 것입니다. 이것은 list 자신이 그 자신을 구성하기 위한 요소로서의 사용되어지기 때문에 재귀적 (“recursive") 형태를 가진다고 지칭합니다.

재귀적 형태를 가지는 또 다른 타입의 객체가 존재하는데 이는 함수 (function)와 통계적 모델링에 사용되는 expression 인 model formula (모델식) 입니다. 우리는 나중에 R 시스템의 일부분을 구성하는 중요한 객체인 사용자 정의 함수에 대해서 다룰 것이지만, 객체로서의 expression은 R 에서도 어려운 개념에 해당되기 때문에 model formula를 어떻게 작성하는가를 제외하고는 이 안내서에서는 다루지 않을 것입니다.

위에서 언급한 것과 같이 모드란 것은 객체를 구성하는 기본 데이터 유형을 의미합니다. 그러나, 모드란 것은 실제로 객체가 특징을 나타내는 여러가지의 속성 (property) 들 중에 한가지 입니다. 객체가 가지는 또 다른 속성은 length (길이) 라는 것입니다. 만약, 독자가 임의의 객체가 가지는 mode 와 length 를 알고 싶다면 mode(object) 와 length(object) 라는 함수를 사용하여 확인할 수 있으며, 객체의 특징을 정의하는 더 많은 종류의 속성들에 확인해 보고 싶으시다면 attributes(object) 라는 함수를 이용해 보시길 바랍니다. 더 많은 것을 알고 싶으시다면, see Getting and setting attributes.

예를 들어, 벡터 z가 100개의 복소수 데이터형을 가진 구성요소로 되어 있다면, mode(z) 는 "complex" 를 보여줄 것이고, length(z) 는 100 이라는 값을 보여줄 것입니다.

R 시스템은 객체의 데이터형을 변환시켜주는 기능도 가지고 있습니다.

```
> z <- 0:9
```

예를 들면 z가 0 부터 9까지의 숫자로 된 벡터라면, 아래의 명령문은 숫자형 벡터 z를 문자형 벡터로 변환해 줍니다. 그 결과는 c("0", "1", "2", ..., "9") 와 동일하며, 문자형 벡터를 입력할때는 반드시 큰 따옴표와 함께 사용하는 것을 기억해 두어야 합니다.

```
> digits <- as.character(z)
```

문자형 벡터 z 는 다시 아래의 명령문을 이용하여 숫자형 벡터로 변환시킬 수 있습니다. 따라서 d 와 z 는 동일한 벡터입니다.

```
> d <- as.integer(digits)
```

이와 같이 데이터형의 변환을 가능케 해주는 (즉, 객체가 가지고 있는 속성을 변환할 수 있도록 해주는) as.something 형태의 많은 함수가 제공됩니다. 여러분들은 이러한 함수들의 이용에 친숙해지기 위해서 도움말 파일을 잘 활용하셔야 합니다.

Changing the length of an object

아무런 구성요소를 가지고 있지 않은 객체일 지라도 모드를 가지고 있습니다.

```
> e <- numeric()
```

위의 명령문은 구성요소를 가지고 있지 않으나, 모드는 수치형을 가지고 있는 객체 e 를 생성하라는 의미입니다. 이와 유하하게 character() 란 함수 역시 구성요소는 없으나, 모드는 문자형을 가진 객체를 생성하라는 의미입니다. 이렇게 생성되어진 객체들은 임의의 사이즈를 가지는 벡터를 생성할 수 있습니다.

예를 들어, 세개의 구성요소를 가진 벡터 e 를 생성하고자 한다면, 다음과 같이 인덱스를 사용하여 벡터의 길이를 지정할 수 있습니다. 한개의 인덱스만을 이용했기 때문에 지정된 인덱스 3 만큼 벡터의 길이가 생성되고, 3번째 구성요소에 17이라는 값이 할당되었으나, 첫번째와 두번째의 구성요소가 가질 수 있는 값에 대해서 아무런 정보도 제공하지 않았기 때문에 NA 라는 값을 가지게 됩니다.

```
> e[3] <- 17
```

종종 이렇게 자동으로 객체의 길이가 조정되어질 필요가 있습니다. 그 대표적인 예로서는 입력기능에 관련된 scan() 라는 함수입니다. scan() 함수에 대한 더 자세한 내용은 see The scan() function.

이와 반대로 객체의 길이 (즉, 사이즈)를 줄이는 방법은 오로지 할당 (assignment) 을 이용한 방법밖에는 없습니다. 만약 alpha라는 벡터는 길이가 10 이라면, 다음의 명령문은 alpha의 인덱스 1 부터 10 중에서 짝수에 해당하는 인덱스들만을 골라 길이가 5인 새로운 객체를 생성한뒤 alpha 에 재할당하라는 의미입니다.

```
> alpha <- alpha[2 * 1:5]
```

벡터 alpha 의 길이를 변경하는 또 다른 방법은 length() 함수를 이용하는 것입니다. 다음의 명령어는 alpha 벡터의 길이를 3 으로 변경했기 때문에 벡터 alpha는 처음 세개의 구성요소만을 가지는 벡터를 생성하라는 의미와 동일합니다.

```
> length(alpha) <- 3
```

Getting and setting attributes

attributes(object) 라는 함수는 객체가 가지는 모든 속성의 이름을 보여주는 함수입니다. 이렇게 객체가 가지는 속성의 이름을 알고 그 속성의 모드를 확인하고자 할 때 attr(object, name) 의 형식으로 함수를 사용합니다. 그러나 실제적으로 이러한 함수는 특수한 경우를 제외하고는 많이 사용되지는 않지만, 매우 중요한 개념입니다. 그 이유는 이 속성이라는 개념을 이용하여 R은 모든 객체 관리를 하기 때문입니다.

다음의 명령어는 z 라는 벡터가 가지고 있는 dim (차원)이라는 속성의 값을 c(10,10) (즉, 10행 10열) 으로 정의하거나 혹은 변경하는 것입니다.

```
> attr(z, "dim") <- c(10,10)
```

The class of an object

R 에서 사용되는 모든 객체들은 class (클래스) 라고 하는 원형을 가지고 있습니다. 여기에서 원형이란, 데이터 혹은 객체들이 특정한 방식으로 동작하도록 하는 미리 정해놓은 형식 혹은 일종의 규칙을 의미합니다. 예를들어, 벡터는 모든 구성요소가 수치, 논리, 그리고 문자와 같은 동일한 데이터 형을 가져야 한다는 제약조건 한가지만 있습니다. 이에 반해, 행렬, 요인, 혹은 데이터 프레임이라는 것을 정의하고 사용하기 위해서는 데이터형 이외에도 차원과 같은 부가적인 조건을 가져야만 합니다.

객체가 가지는 이러한 클래스란 개념은 컴퓨터 사이언스에서 소위 말하는 메소드라는 개념과 함께 객체지향 (an object-oriented) 스타일의 프로그래밍을 가능하도록 해줍니다. 2 예를 들어, 사용자가 "data.frame" 이라는 클래스를 가진 객체를 그래픽적으로 출력하고자 plot() 이라는 함수를 이용한다면, 그 객체는 plot() 이라는 함수내에서 미리 정해져 있는 방식대로 데이터를 불러 들여와지고, 이 불러들여온 데이터를 어떻게 시각화 할 것인지 미리 정해놓은 규칙대로 그래픽 처리를 할 것입니다. 또한, generic 타입의 summary() 함수의 결과물 출력은 객체가 가지는 클래스에 따라서 매우 달라집니다. 만약, winter 라는 객체가 데이터프레임이라는 클래스를 가지고 있다면, 데이터프레임이라는 클래스 내에 미리 정해놓은 출력방식을 따라서 결과를 보여줄 것입니다.

```
> winter
```

데이터 프레임이란 클래스는 리스트 형식의 데이터벡터를 입력받아, 이를 행렬의 형식으로 변환후 변수명과 함께 출력하도록 약속되어 있습니다. 따라서, 아래와 같이 unclass()라는 함수를 이용한다면, 클래스내에 정의되어 있는 출력형식을 더 이상 따르지 않고, 본래의 리스트 형식으로 출력하게 됩니다. 즉, 행렬의 형식으로 변환하고 변수명과 함께 출력하라는 클래스 내의 출력형식에 대한 규칙을 무시하게 됩니다.

```
> unclass(winter)
```

이런 클래스의 활용은 매우 특수한 경우에만 사용되지만, 현재로서는 단순히 클래스와 generic 함수라는 개념에만 익숙해지시면 됩니다. 우리는 이것들에 대해서 조금 더 다룰 예정이지만, R 에서도 어려운 내용에 해당하므로 이 문서에서 자세히 다루지는 않을 것입니다. Object orientation.

Ordered and unordered factors

요인 (factor) 이라는 벡터는 다른 벡터를 분류 (classification 혹은 grouping) 하는데 쓰여지는 벡터를 말합니다. 이 요인이 가지고 있는 레벨이 어떤 순서를 가지고 있는가 없는가에 따라서 순서형 (ordered) 혹은 명목형 (unordered) 로 구분합니다. 예를 들어, 성별이라는 요인의 남 혹은 여는 명목형이고, 월소득수준이 100만원 이하, 100~200, 200~300, 300~400, 500 이상을 5 부터 1등급으로 분류한 것은 순서형입니다.

요인이라는 벡터를 실제 분석에서 어떻게 활용하는지는 see Contrasts 에서 더 자세히 다룰 것이지만, 여기에서는 단순히 개념을 이해하는데 중점을 두겠습니다.

A specific example

예를 들어, 오스트리아에 있는 30명의 회계사들에 대한 정보를 가지고 있는 샘플이 있다고 가정합니다. 그리고, 그들의 근무지에 대한 정보가 state 라는 변수에 다음과 같이 입력되어 있습니다.

```
> state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
             "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
             "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
             "sa",  "act", "nsw", "vic", "vic", "act")
```

factor() 함수는 먼저 state 벡터가 가지는 문자형 구성요소들을 알파벳순으로 정렬한뒤, state 가 가지고 있는 특별한 값들의 개수만큼 levels 를 생성합니다. 그리고, 벡터 state의 길이만큼 각각의 구성요소들이 어떤 level에 속하는가에 대한 그 분류결과를 가지고 있는 벡터를 생성해 줍니다.

```
> statef <- factor(state)
```

print() 함수를 이용하여 statef 의 결과를 확인하여 보시면 그 결과가 state라는 벡터와 동일한 것을 알 수 있을 것입니다. 그러나, 사용자는 state 벡터는 원래의 데이터가 입력되어 있는 벡터이고, statef 라는 벡터는 state 라는 벡터의 분류결과를 포함하는 벡터임을 알고있어야 합니다.

```
> statef
[1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa
[16] tas sa  nt  wa  vic qld nsw nsw wa  sa  act nsw vic vic act
Levels:  act nsw nt qld sa tas vic wa
```

요인이 몇개의 분류를 가지고 있는지 확인하고자 한다면 levels() 이라는 함수를 이용하시면 됩니다 .

```
> levels(statef)
[1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

The function tapply() and ragged arrays

The function tapply() and ragged arrays

이전에 사용된 예제를 계속 이용하여 factor가 어떻게 활용되는가에 대한 기본적인 응용 예제를 tapply()를 통하여 알아보겠습니다. 우리는 동일한 회계사들의 수입을 income 이라는 벡터에 아래와 같이 저장합니다.

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
              61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
              59, 46, 58, 43)
```

각 근무지별로 그들의 평균 소득을 계산하고 싶다면 아래와 같이 tapply() 라는 함수를 이용하시길 바랍니다. 첫번째 인자는 30명의 회계사들의 소득을 포함하고 있는 incomes 이라는 벡터를 불러들이고, 두번째 인자는 회계사들의 근무지가 어디에 속하는지에 대한 결과가 포함된 statef라는 벡터가 이용되며, 마지막으로 세번째 인자는 각 근무지별로 분류된 데이터에 평균을 산출해 내는 mean 이라는 함수명이 사용되었습니다.

```
> incmeans <- tapply(incomes, statef, mean)
```

그 결과는 다음과 같습니다.

```
     act    nsw     nt    qld     sa    tas    vic     wa
  44.500 57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

만약, 사용자가 평균소득에 대한 편차 (standard error) 를 알고 싶다면, 이를 계산하는 R 함수를 다음과 같이 작성해야 할 것입니다.

```
  > stderr <- function(x) sqrt(var(x)/length(x))
```

(사용자가 이처럼 직접 작성하는 함수에 대해서는 Writing your own functions에서 더 다룰 것입니다. 그러나, R 은 사용자의 편의를 위하여 이미 방대한 양의 통계량을 산출하는 함수들을 작성해 두었습니다. 이를 내장함수라고 하며, 표준편차는 sd() 라는 함수에 의해 구해질수 있습니다. 위에서 사용자가 미리 정의한 함수 stderr() 를 이용하여 근무지별 평균소득에 대한 편차를 다음과 같이 구할 수 있습니다.

```
  > incster <- tapply(incomes, statef, stderr)
  > incster
  act    nsw  nt    qld     sa tas   vic     wa
  1.5 4.3102 4.5 4.1061 2.7386 0.5 5.244 2.6575
```

아마 사용자는 평균소득에 대한 95% 신뢰구간을 찾아야 할 수 도 있습니다. 이것에 대한 것은 여러분들이 연습을 위해서 남겨두지만, 이를 위해서는 여러분들은 아마도 tapply(), length(), qt() 라는 함수들이 필요할 수 있습니다. length() 라는 함수는 샘플 사이즈를, qt() 라는 함수는 t-분포에서 95% 에 해당하는 계수를 찾아 줄 것입니다.

위의 예제에서 살펴 본 것과 같이 요인(factor)는 그룹핑에 매우 유용하게 사용될 수 있습니다. 만약, 회계사의 평균 소득을 근무지와 성별이라는 요인에 따라 분석을 해 본다고 가정한다면, tapply() 라는 함수는 각각의 요인에 의해서 그룹화된 자료에 분석해 내고자 하는 통계량을 산출해 낼 수 있는 함수를 각 그룹에 일괄적으로 적용하는 편리한 방법을 제공한다고 할 수 있습니다.

Ordered factors

요인의 레벨 (level) 은 기본적으로 알파벳 순서로 저장되나, 사용자에 따라 그 순서가 지정될 수 도 있습니다. 어떤 경우에는 레벨들이 우리가 원하고자 했던 대로 기록되어 있거나, 통계분석시 필요로 하는 데이터

형태를 미리 가지고 있을 수도 있습니다. ordered() 라는 함수는 레벨들을 순차적으로 정리하는 기능외에는 정렬되지 않은 factor와 다른것이 없으나, 적합된 선형모델을 이용하여 추정을 할 경우 다른 결과값을 산출하므로 주의해야 합니다.

Arrays and matrices

Arrays

배열 (array) 란 벡터안에 저장된 데이터들의 값들과 개개의 데이터가 벡터내에서 어디에 위치하고 있는가에 대한 위치 정보를 함께 지닌 형태를 의미합니다. 이 위치는 위에서 설명했던 인덱스 벡터를 활용하는데, 벡터가 1차원 인덱스 벡터를 가지는 반면에, 배열은 다차원 인덱스 벡터를 가질 수 있습니다. 예를 들어, 벡터는 1차원적으로 숫자들을 일렬로 나열한 형태입니다. 이에 반해, 행렬이라는 것은 이렇게 일렬로 나열된 숫자들을 행과 열이라는 두가지 정보를 이용하여 벡터를 2차원 공간에 재배열한 형태를 지니고 있습니다. 좀 더 일반화 시키면, 벡터는 2차원 공간이 여러 번 겹친 육면체의 형태인 3차원공간에 재배열 될 수 있습니다. 이렇게 벡터가 재배열될 수 있는 공간의 크기를 차원이라고 하고, 벡터가 가지는 구성요소가 그 공간내에서 정확히 어디에 있는가에 대한 위치정보를 각 차원별 인덱스를 이용하여 표현합니다. R 에서는 공간의 크기를 차원 (dimension 혹은 디멘션) 벡터라고 하고, 디멘션 벡터의 길이가 k 라면 k-차원을 가지는 배열이라고 합니다. 그리고, 각각의 차원은 1부터 디멘션 벡터에서 지정한 해당 차원의 크기만큼의 인덱스를 양의 정수 형태로 가질 수 있습니다.

예를들어, 1500개의 수치형 자료로 이루어진 벡터 z가 있다고 가정합니다. 다음에 사용된 dim 이라는 명령어는 z 라는 벡터가 가지고 있는 dim(차원의 약자)라는 속성을 변경함으로서 3행-5열-10차원 공간에 재배열하도록 합니다.

```
> dim(z) <- c(3,5,100)
```

R 은 특히 행렬 (matrices) 와 배열 (arrays) 를 생성하고 다루기 위한 많은 함수들을 제공하고 있는데, 이에 대해서는 The array() function.

사용자는 배열의 차원벡터에 의해서 벡터의 구성요소들이 어떻게 재배열되는지 궁금해 할 것입니다. 이는 FORTRAN (포트란) 시스템에서 이용되는 방식과 동일합니다. 즉, 벡터의 구성요소들은 배열의 디멘션 벡터에서 지정된 각각의 차원의 크기만큼 열방향으로 나열해 나갑니다. 예를 들어, c(3,4,2) 라는 디멘션벡터를 가진 a이라는 임의의 배열이 존재한다면, a 라는 배열은 총 개의 데이터를 입력받기를 요구할 것입니다. 따라서 데이터가 입력되는 순서는 a[1,1,1], a[2,1,1], ..., a[2,4,2], a[3,4,2] 가 됩니다 . 즉, 차원벡터의 맨처음 인덱스가 가장 빠르게 갱신되면서 데이터를 입력받고, 차원벡터의 맨 마지막 인덱스가 가장 천천히 업데이트되면서 데이터를 보관하게 됩니다.

이문서에서 우리는 용어의 혼란을 피하기 위하여 편의상 1차원 배열을 벡터, 2차원 벡터를 행렬이라고 하고, 3차원 이상인 경우 배열이라고 합니다.

Array indexing. Subsections of an array

배열이 가진 개개의 데이터는 배열의 이름 뒤에 붙여 쓰는 대괄호 안의 위치 지시자를 이용하여 표현할 수도 있습니다. 또한, 배열의 일부분만을 따로 추출해 내고자 하는 경우에는 index vectors (인덱스 벡터)를 이용할 수도 있습니다. 만약, 대괄호 내부에서 컴마들 사이에 위치해야할 인덱스 벡터가 빈공간으로 남는다면, 이는 해당 차원의 모든 인덱스를 지정한 것과 같습니다. 여러분의 이해를 돕기 위해서 이전에 사용된 3행-4열-2차원의 크기를 가진 a 라는 벡터를 계속 이용하도록 하겠습니다.

a 라는 배열은 3행-4열-2차원의 크기를 가지고 있는데, a[2,,]이라는 것은 3행-4열-2차원 육면체 공간에서 2행에 해당하는 모든 데이터들을 뽑아 내라는 의미입니다. 따라서 다음에 해당하는 데이터들을 가지게 됩니다.

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],
  a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2])
```

a[, ,] 라고 표시하는 것은 a라는 배열이 가지는 전체 데이터를 의미하므로 단독적으로 a 라는 배열의 이름을 사용하는 것과 동일한 역할을 합니다.

만약, 여러분이 Z 라는 임의의 배열에 대한 디멘션을 알고 싶으시다면, dim(Z) 라는 함수를 이용하시면 됩니다.

Index matrices

배열뒤에 오는 대괄호안에 사용되는 위치 지시자에 원하고자 하는 위치에 데이터의 값을 할당 혹은 추출을 위해서 ₩인덱스 행렬을 이용할 수 도 있습니다. 다음의 예제는 여러분들의 이해를 도울 것입니다.

예를 들어, 1부터 20까지의 숫자를 가지는 4행 5열인 2차원 배열 X 가 있고, 다음의 작업을 수행하고자 합니다.

X[1,3], X[2,2], 그리고 X[3,1] 에 해당하는 데이터만을 뽑아내어 새로운 벡터를 생성하고, 데이터를 뽑아낸 위치에 해당하는 값들을 0 으로 교체 하고 싶습니다. 이러한 작업을 효율적으로 하기 위해서는 여러분은 3 행 2 열로 된 인덱스행렬이 필요할 것입니다.

```
> x <- array(1:20, dim=c(4,5))   # 4행 5열로 구성된 2차원 배열을 생성합니다.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1), dim=c(3,2))
> i                             # i 이라는 3행 2열의 인덱스행렬을 생성했습니다.
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                          # 인덱스행렬에 해당하는 값만 뽑아냅니다
[1] 9 6 3
> x[i] <- 0                     # 인덱스 행렬에 해당하는 값들을 0으로 교체합니다.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
>
```

인덱스 행렬에서 음의 값을 지시자 (index) 를 사용할 수는 없지만, NA 와 0 은 사용할 수 있습니다. 만약, 인덱스 행렬이 0을 포함하고 있는 행이 있다면 이는 무시 될 것이고, NA 를 포함하고 있다면 모두 NA 를 출력할 것입니다.

다소 쉬운 예제는 아니지만 , b 개의 레벨이 있는 blocks 이라는 요인과 v 개의 레벨이 있는 varieties 라는 요인을 이용하여, n개의 플랏에 적용될 실험에 사용될 디자인 매트릭스는 다음의 절차를 통하여 얻어질 수 있습니다.

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

만약 N 이라는 the incidence matrix (인시던스 매트릭스)를 생성하고자 한다면 다음과 같이 하면 됩니다.

```
> N <- crossprod(Xb, Xv)
```

위와 동일한 인시던스 매트릭스는 table() 이라는 함수를 이용하면 더욱 쉽게 생성할 수 있습니다.

```
> N <- table(blocks, varieties)
```

인덱스 행렬은 반드시 수치형으로 구성되어야 합니다.

The array() function

배열은 벡터에 dim 이라는 속성을 부여하여 얻는 방법외에도, 아래와 같은 방법으로 array 라는 함수를 이용하여 생성할 수 있습니다.

```
> Z <- array(data_vector, dim_vector)
```

예를 들어 벡터 h가 24개의 수치형 원소를 가진다면, 아래의 명령어는 3행 4열로 구성된 2개의 행렬을 나란히 3차원 공간에 배열 한 뒤 이를 Z 에 할당했음을 의미합니다.

```
> Z <- array(h, dim=c(3,4,2))
```

위에서는 배열 Z 를 array() 란 함수를 이용하였으나, 아래의 명령어는 벡터 h 의 속성을 조정하여 동일한 배열 Z 를 생성함을 보여줍니다.

```
> Z <- h ; dim(Z) <- c(3,4,2)
```

그런데, 만약 벡터 h 가 24개보다 적은 수의 원소를 가진다면, 24개라고 지정된 배열의 크기를 맞추기 위해서 벡터 h의 첫번째 원소부터 재사용되게 됩니다. 이것에 대해서는 (see The recycling rule). 그러나, dim(h) <- c(3,4,2) 와 같이 속성을 조정하고자 할 때에는 에러를 보여줄 것입니다.

다음은 배열 Z의 모든 원소가 0 이라는 값을 가지는 특수한 경우입니다. 여기에서의 dim(Z) 는 차원벡터를 의미하고, Z[1:24] 는 h 와 동일한 데이터벡터를 의미합니다. Z[] 와 같이 아무런 서브스크립트를 가지지 않거나 단순히 Z 라고 배열이름을 사용하는 것은 배열 전체를 의미합니다.

```
> Z <- array(0, c(3,4,2))
```

배열에서의 산술연산은 기본적으로 배열을 구성하는 원소단위에서 이루어집니다. 이때, 각각의 배열이 가지는 dim 이라는 차원속성이 모두 동일해야 하며, 동일한 차원의 결과값을 얻을 수 있게 됩니다.

따라서 아래의 명령문은 A, B, 그리고 C 가 모두 같은 dim 차원속성을 가지고 있다는 전제하에서, 원소단위의 연산을 수행한뒤 D 에 할당함을 의미합니다. 그러나, 배열과 벡터가 혼합된 경우의 연산을 수행할 경우에는 많은 주의가 요구됩니다.

```
> D <- 2*A*B + C + 1
```

Mixed vector and array arithmetic. The recycling rule

벡터와 배열이 혼합된 연산을 수행하는 것을 설명하기에는 다소 복잡하지만, 우리의 경험으로 아래와 같은 내용을 알려드립니다.

수식연산은 기본적으로 좌에서 우로 진행합니다. 만약, 연산에 이용되는 벡터들의 길이가 서로 다를 경우, 길이가 짧은 벡터는 원소를 재사용하여 길이가 긴 벡터의 길이 만큼 맞추게 됩니다. 만약, 동일한 dim 속성을 가지지 않는 배열끼리 연산될 경우에는 에러가 출력됩니다. 만약, 벡터의 길이가 행렬과 배열의 길이보다 큰 경우에도 에러가 출력됩니다.

The outer product of two arrays

배열의 연산에서 가장 중요한 것은 %o% 라는 연산자기호를 이용하여 연산을 수행하는 outer product (한국말 모름) 이라는 것입니다. 예를 들어 a와 b 라는 두개의 수치형 배열이 있다면, outer product 의 결과로 얻어지는 차원속성은 c(dim(a), dim(b)) 이고, 데이터 벡터는 a 의 모든 원소와 b 의 모든 원소로 얻어진 모든 조합의 곱입니다.

```
> ab <- a %o% b
```

다음은 outer() 란 함수를 이용하여 동일한 outer product 라는 연산을 수행하는 것입니다.

```
> ab <- outer(a, b, "*")
```

outer product 는 두개의 변수를 가지고 있는 임의의 함수에 대한 결과값을 연산하고자 할때 유용하게 사용됩니다. 예를 들어, x- 와 y- 좌표평면상에 있는 모든 좌표에 대하여 f(x;y)=cos(y)/(1+x^2) 라는 함수가 가질 수 있는 모든 값은 다음의 명령을 수행함으로서 쉽게 얻을 수 있습니다.

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

An example: Determinants of 2 by 2 single-digit matrices

다소 인위적이긴 하지만 이해가 쉬운 예를 하나 들어보겠습니다. 먼저, [a, b; c,d] 라는 2행 2열의 행렬이 있다고 가정합니다. 그리고, 0 부터 9까지 정수가 각각의 원소 a, b, c, d에 할당될 확률은 균등하며, 독립적으로 할당됩니다. 이때, ad-bc 라는 행렬계수 (determinant) 의 분포를 구하고자 한다면, 다음과 같이 두번의 outer() 함수를 사용함으로 해결할 수 있습니다.

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
       xlab="Determinant", ylab="Frequency")
```

여기에서 한가지 주의 할 점은 fr 이라는 상대도수표에서 행렬계수가 가지는 범위를 나타내는 name 이라는 속성을 수치형으로 강제변환해야 한다는 것입니다. 가장 확실한 방법은 for 라는 반복문을 이용하는 것인데, 이것은 비효율적이므로 실용적이지 못합니다. 그러나, 반복문에 대해서는 Loops and conditional execution 에서 다루어 질 것입니다.

Generalized transpose of an array

aperm(a, perm) 이라는 함수는 배열의 차원을 재배열하는 역할을 합니다. 이것의 가장 쉬운예제는 행렬의 전치를 들 수 있습니다. 행렬 A의 차원벡터 dim(A)는 c(1,2) 이므로, 이것의 순서를 c(2,1) 로 바꾸는 것은 행렬 A를 전치 (transpose) 한 t(A) 와 동일한 결과를 줍니다. (즉, B <- t(A)).

```
> B <- aperm(A, c(2,1))
```

Matrix facilities

위에서 언급했던것과 같이, 행렬은 단순히 2개의 서브스크립트로 이루어진 배열의 특수한 경우입니다. R 은 다양하고 풍부한 행렬의 연산기능을 제공하고 있습니다. 예를 들면, t(X) 는 행렬 X 의 전치입니다. 함수 nrow(A) 과 ncol(A) 은 행렬 A 의 행의 개수와 열의 개수를 알려줍니다.

Matrix multiplication

%*% 라는 연산자는 행렬의 곱에 사용됩니다. 본래 1개의 행 또는 1개의 열로 구성된 행렬은 정확히 말하면, n 개의 요소를 가진 행 벡터 혹은 열벡터입니다. 그러나, 행렬곱셈에서 사용되는 벡터는 수학적 표현이 논리적으로 정확하다면 행 혹은 열벡터로 적절하게 변경되어 연산을 수행합니다.

만약, A 와 B 가 정방행렬이라면, 다음의 표현은 원소단위의 곱을 수행합니다.

```
> A * B
```

그러나, 다음의 표현은 원소의 곱이 아닌 행렬곱셈을 의미합니다.

```
> A %*% B
```

만약, 아래와 같이 행렬곱셈에서 사용되는 x가 벡터라면 스칼라 값을 돌려주는 qaudratic (한국말 모름) 연산을 수행합니다.

```
> x %*% A %*% x
```

crossprod() 이라는 함수는 연산속도가 더 효율적이라는 점만 제외한다면 t(X)%*%y 와 동일한 연산을 수행합니다. 만약, 두번째 인자와 첫번째 인자와 동일하다면 두번째 인자는 생략해도 무방합니다.

diag() 라는 함수가 v 라는 벡터를 인자를 가진다면 대각원소가 v인 대각 행렬을 생성합니다. 하지만, diag() 의 인자가 M 이라는 행렬이라면, 행렬 M 의 대각원소를 출력하게 됩니다. 이것은 Matlab 에서 사용되는 diag() 이라는 함수의 기능과 동일합니다. 다소 혼돈스러울 수도 있으나, diag() 의 인자가 k라는 스칼라라면 k 행 k열을 가지는 단위행렬 (identity matrix) 를 생성합니다.

Linear equations and inversion

다음의 표현과 같은 선형 방정식이 존재할 때, A 의 계수값들과 b 의 값을 미리 알고 있다고 가정한다면, x 를 찾아내는 것을 선형방정식의 해를 찾는다고 합니다.

```
> b <- A %*% x
```

R 은 다음의 명령어를 통하여 선형방정식의 해 x 를 구합니다.

```
> solve(A,b)
```

즉, 이것은 x = A^{-1} %*% b 입니다. 여기에서 A^{-1} 은 A 의 역행렬이며, 다음의 명령어를 수행하여 얻을 수 있습니다.

```
solve(A)
```

하지만, 위와 같이 역행렬을 이용하여 x <- solve(A) %*% b 와 같은 연산을 하는 것은 solve(A,b) 라는 함수를 이용하는 것보다 비효율적입니다.

Eigenvalues and eigenvectors

eigen(Sm) 이라는 함수는 대칭행렬 Sm 의 기저값 (eigenvalues) 와 기저벡터 (eigenvectors) 를 계산해줍니다. 예를들어, 아래와 같이 이 함수를 적용한 결과는 values 라는 기저값으로 구성된 벡터와 vectors 라는 기저벡터로 구성된 행렬을 반환하는 리스트의 형식을 가지고 있습니다. 따라서 기저값은 ev$val을, 기저벡터는 ev$vec을 통하여 얻을 수 있습니다.

```
> ev <- eigen(Sm)
```

만약 사용자가 단순히 기저값만을 원한다면 아래와 같이 표현할 수 있습니다. 즉, evals 은 기저값에 대한 결과만을 가지게 되고 기저벡터에 대한 결과는 저장되지 않습니다.

```
> evals <- eigen(Sm)$values
```

만약 아래와 같이 단순히 함수를 적용하는 것은 기저값과 기저벡터의 결과를 동시에 출력해줍니다.

```
> eigen(Sm)
```

행렬의 크기가 크고, 기저벡터보다는 기저값만 요구될 경우 연산속도를 높이기 위해서 아래와 같이 사용할 수 도 있습니다.

```
> evals <- eigen(Sm, only.values = TRUE)$values
```

Singular value decomposition and determinants

svd(M) 이라는 함수는 M 이라는 임의의 행렬을 인자로 가지며 the singular value decomposion (한국말

모름) 을 수행합니다. 이 함수로부터 세가지의 결과를 얻어낼 수 있습니다. 첫번째는 M 행렬의 열을 기초로 한 orthonormal matrix U이고, 두번째는 M 행렬의 행을 기초로 한 orthonormal matrix V 이며, 세번째는 M = U %*% D %*% t(V)라는 조건을 만족하는 D 라는 대각행렬 계산해냅니다. svd(M) 의 실제 계산결과는 위의 세가지를 한데 묶은 리스트의 형식으로 d, u, v 라는 이름으로 저장되어집니다.

예를 들어 행렬 M의 행렬계수(determinant)의 절대값을 구하고자 한다면, singular value decomposition 을 이용하여 아래와 같이 쉽게 얻을 수있습니다.

```
> absdetM <- prod(svd(M)$d)
```

만약 사용자가 위와 같은 행렬계수 (determinant)의 절대값을 자주 사용해야 한다면, R은 아래와 같이 사용자가 정의하는 함수 absdet() 를 작성하여 사용할 수 있도록 도와줍니다.

```
> absdet <- function(M) prod(svd(M)$d)
```

또 다른 직관적인 예로 여러분은 함수의 trace (한국말 모름) 을 계산해주는 tr() 이라는 singular decomposition 을 이용하여 얻어낸 대각행렬과 diag()이라는 함수를 이용하여 쉽게 작성할 수 도 있습니다.

실은 R은 행렬계수 (determinant)를 연산하기 위한 det 를 제공하고 있으며, 행렬계수 외에도 계수의 부호와 로그값이 취해진 모듈러스 (modulus) 의 결과를 함께 계산해주는 determinant 라는 함수또한 제공하고 있습니다.

Least squares fitting and the QR decomposition

아래의 표현은 만약 여러분이 관측치들을 y 라는 벡터에 저장하고, X 라는 디자인 매트릭스를 지정했을때, lsfit() 이라는 함수가 최소제곱법 (least square)을 이용하여 적합된 (fitted) 된 값을 ans 에 할당하는 것을 보여줍니다.

```
> ans <- lsfit(X, y)
```

더 자세한 사항은 도움말을 참조하세요. 그리고 최소제곱법을 이용한 회귀분석을 통한 모델진단에 대해서 조금 더 알고 싶으시다면 이와 연관된 ls.diag() 라는 함수를 살펴보시길 바랍니다. 그러나, 여러분들이 회귀분석을 수행하고자 한다면 lsfit() 이라는 함수를 이용하는 것보다 lm(.) (see Linear models) 을 사용하는 것이 더 편리합니다.

아래의 코드는 lsfit() 이라는 함수를 통한 회귀분석을 qr() 이라는 함수를 이용하여 동일한 작업을 할 수 있음을 보여줍니다.

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

여기에서 b 는 회귀계수를, fit 은 벡터 y 가 X 에 직교영사 (orthogonal projection) 된 값 (즉, fitted 된 값)을, 그리고 res 란 값은 오차 (the projection onto the orthogonal complement)를 가지고 있습니다.

여러분들은 디자인 매트릭스 X 가 꼭 full column rank (한국말 모름) 이라는 조건을 만족해야 하는 것에 대해서 걱정할 필요는 없습니다. 만약 full column rank 가 아니라면, R 은 자동으로 redundancies (한국말

모름) 을 수행합니다.

Forming partitioned matrices, cbind() and rbind()

위에서도 잠깐 언급했었지만, cbind() 와 rbind() 이라는 함수는 벡터와 행렬을 열방향 혹은 행방향으로 묶어 새로운 행렬을 생성할 때 이용됩니다.

```
> X <- cbind(arg_1, arg_2, arg_3, ...)
```

위에서 사용된 cbind() 라는 함수는 임의의 크기를 가진 벡터나 행렬을 인자로 가질 수 있으나, 모든 인자들의 행의 길이는 같아야만 합니다.

rbind()이라는 함수역시 cbind() 라는 함수와 사용방법은 동일하지만, 모든 인자들의 열의 길이는 같아야만 합니다.

예를 들어, X1 과 X2 이라는 두 행렬이 같은 행의 길이를 가지고 있다고 가정합니다. 아래의 명령문은 1 이란 인자는 X1 이 가진 행의 길이만큼 1 을 리사이클링 시켜서 1 이라는 열벡터를 생성한뒤, 1, X1, X2 를 열방향으로 한데 묶어 X 라는 새로운 벡터를 생성시켜 줍니다.

```
> X <- cbind(1, X1, X2)
```

The concatenation function, c(), with arrays

만약, cbind()와 rbind() 이라는 함수가 행렬의 dim 속성을 기초로 여러개의 행렬을 한데 묶어준다고 한다면, c() 이라는 함수는 dim 과 dimnames 라는 속성을 모두 무시한채 결합을 수행한다고 할 수 있습니다. c() 를 이용하는 것이 때때로 더 유용한 경우가 있습니다.

원래 이렇게 속성들을 무시하고, 배열을 단순 벡터로 강제변환하는 방법은 아래와 같이 as.vector()이라는 함수를 이용하는 것입니다.

```
> vec <- as.vector(X)
```

위에서 얻은 동일한 결과를 아래와 같이 c()라는 함수를 이용해서도 얻을 수 있습니다. 실은 약간 다른 점은 있으나, 어느 것을 사용하는가는 프로그래밍 스타일의 차이일 뿐입니다 .

```
> vec <- c(X)
```

Frequency tables from factors

위에서 요인 (factor) 라는 것은 데이터를 요인이 가진 레벨에 따라서 분류할 때 사용한다고 했습니다. 만약 2개의 요인이 존재한다면 2차원 분류 (two way cross classification) 가 가능할 것이고, 더 나아가 k 개의 요인이 있다면 k-차원의 분류가 가능할 것입니다.

함수 table() 은 배열을 이용하여 이러한 분류가 가능하도록 도와줍니다. 예를 들어, 여러개의 레벨을 가진 지역코드란 요인에 대한 정보가 statef 에 저장되어 있다면, 다음의 코드는 지역코드별 도수분포를 statefr 에 저장하게 됩니다. 이때 각 레벨별 도수는 레벨이 가진 이름순으로 정렬됩니다.

```
> statefr <- table(statef)
```

이와 동일한 역할을 수행하는 또 다른 편리한 방법은 아래와 같습니다.

```
> statefr <- tapply(statef, statef, length)
```

지역별 코드의 경우에는 데이터값이 이산형이지만, 소득수준의 경우는 연속형의 데이터를 가집니다. 이런 경우, 소득수준에 대한 구간을 나누는 편리한 방법은 아래와 같이 cut() 이라는 함수를 사용하는 것입니다.

```
> factor(cut(incomes, breaks = 35+10*(0:7))) -> incomef
```

아래의 코드는 cut() 이라는 함수에 의해 구간화된 incomef 이라는 요인을 이용하여 지역코드별 소득수준에 대한 도수분포를 보여줍니다.

```
> table(incomef,statef)
          statef
incomef   act nsw nt qld sa tas vic wa
  (35,45]   1   1  0   1  0   0   1  0
  (45,55]   1   1  1   1  2   0   1  3
  (55,65]   0   3  1   3  2   2   2  1
  (65,75]   0   1  0   0  0   0   1  0
```

Lists and data frames

Lists

R 에서 list 라는 것은 여러개의 컴포넌트 (components) 라고 불리는 객체들의 집합입니다. 리스트가 벡터, 매트릭스, 혹은 배열과 구분되는 점은 컴포넌트들의 데이터형이 같지 않아도 된다는 것입니다. 예를들어, 수치형 벡터의 모든 원소는 수치형 데이터를 가져야만 했습니다. 그러나 리스트는 각각의 컴포넌트가 수치형 벡터, 논리형 벡터, 행렬, 배열, 그리고 함수로 구성될 수 있습니다. 다음은 리스트를 생성하는 간단한 예를 보여줍니다.

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
              child.ages=c(4,7,9))
```

리스트의 구성요소인 각각의 컴포넌트를 구분하는 방법은 컴포넌트 인덱스를 사용하는 것입니다. 예를 들어, 위와 같이 4개의 컴포넌트들로 구성된 리스트가 있다면, 각각의 컴포넌트는 Lst1, Lst2, Lst3, Lst4 에 의해서 엑세스 되어 집니다. 더 나아가, 마지막 네번째 컴포넌트 Lst4 의 첫번째 구성요소의 값을 알고자 한다면, Lst4[1] 이라고 하면 됩니다.

만약 사용자가 접한 리스트가 몇개의 컴포넌트로 구성된지 모를 경우 length() 명령어를 사용하면 컴포넌트의 개수를 알수 있습니다. 위에 사용된 Lst 이라는 리스트를 예를 들면, length(Lst) 은 4란 값을 알려줍니다.

리스트를 구성하는 컴포넌트들은 인덱스를 사용하여 엑세스 하는 방법외에도 일반적으로 아래와 같이 컴포넌트의 이름을 사용하여 엑세스 할 수 도 있습니다. 아래에서 name 이라는 것은 리스트의 이름에 해당되고 스트링 ($) 이라는 문자 다음에 오는 component_name 이라는 것은 컴포넌트의 이름입니다.

```
> name$component_name
```

이렇게 이름을 사용하는 방법은 사용자가 컴포넌트 인덱스를 일일이 기억하지 않도록 도와줍니다. 위에서 사용된 예제를 다시 보면, Lst$name 이라는 것은 Lst1와 동일하고, "Fred" 라는 값을 가집니다. Lst$wife 는 Lst2와 동일하고 "Mary"라는 값을 가집니다. Lst$child.ages[1] 역시 Lst4[1] 동일하며, 4 라는 값을 가집니다.

추가적으로, Lst$name 과 Lst"name" 은 Lst 라는 리스트의 첫번째 컴포넌트를 엑세스 하기 위한 동일한 표현법입니다. 이것은 아래와 같이 리스트의 이름을 다른 변수에 따로 저장하여 활용 할 수 있도록 하는 아주 유용한 기능입니다.

```
> x <- "name"; Lstx
```

리스트를 활용함에 있어서 Lst1 와 Lst[1] 를 구분하는 것은 매우 중요합니다. '...' 은 리스트에서 컴포넌트를 구별하기 위해서 사용되고, '[...]' 은 컴포넌트가 가지는 원소들을 구분하기 위해서 사용됩니다. 그러므로, 리스트 이름뒤에는 항상 '...' 이 먼저 오고, '[...]' 이 나중에 와야 합니다.

만약 리스트를 구성하는 컴포넌트들의 이름이 길지만, 동일한 이름이 없다면, R 은 컴포넌트들을 구분할 수 있는 약어기능을 제공합니다. 예를들면, Lst 라는 리스트가 coefficients 와 covariance 라는 두개의 컴포넌트를 가지고 있다면, 굳이 Lst$coefficients 혹은 Lst$covariance 이라는 이름을 다 사용할 필요가 없습니다. R 은 Lst$coe 와 Lst$cov 만으로도 충분히 두개의 컴포넌트를 구분해 낼 수 있습니다.

Constructing and modifying lists

만약 여러분이 이미 작업하고 있던 여러개의 객체를 가지고 있었다면, list() 라는 함수를 이용하여 새로운 리스트를 생성할 수 도 있습니다. 아래의 명령문은 object_1 이라는 객체를 첫번째 컴포넌트로 name_1 이라는 이름과 함께 사용할 것이며, 이와 같은 방법으로 m 개의 객체를 새로운 리스트를 정의하겠다는 의미입니다.

```
> Lst <- list(name_1=object_1, ..., name_m=object_m)
```

만약, 컴포넌트에 사용할 이름이 정해지지 않았다면, R 은 자동으로 몇 번째 컴포넌트인지 숫자를 이름으로 정합니다. 새로운 리스트가 생성될 때 기존의 객체들은 단순히 컴포넌트로서 복사 되는 것이기 때문에 새로운 리스트가 정의되어도 기존의 객체들은 아무런 영향을 받지 않습니다.

만약, 리스트에 새로운 컴포넌트를 추가하고 싶을 경우에는 아래의 명령문과 같이 서브스크립트를 이용하면 편리합니다.

```
> Lst[5] <- list(matrix=Mat)
```

Concatenating lists

만약 여러개의 리스트를 한데 묶고 싶다면 아래와 같이 함수 c() 를 이용하면 사용된 인자의 순서대로 한데 묶인 새로운 리스트를 생성해 줍니다.

```
> list.ABC <- c(list.A, list.B, list.C)
```

Data frames

데이터 프레임 (data frame) 이라는 것은 특수한 형태의 리스트로서 "data.frame" 라는 클래스로 따로 구분됩니다. 데이터 프레임은 리스트와 구분되는 몇가지 특징이 있습니다.

데이터 프레임에 입력될 수 있는 자료는 반드시 (수치형, 문자형, 혹은 논리형) 벡터, 수치형 행렬, 리스트, 또는 다른 데이터 프레임이어야만 합니다. 데이터 프레임을 구성하는 각각의 열은 변수로 인식되므로, 새로운 데이터 프레임은 입력되는 행렬의 열, 리스트의 컴포넌트, 혹은 데이터 프레임의 변수의 개수만큼 자동으로 확장되어 생성되어 집니다. 데이터 프레임에 입력되는 수치형, 문자형, 그리고 논리형 벡터는 모두 요인으로 강제변환되어지고, 입력된 벡터내에 중복되지 않은 값들이 요인의 레벨로 간주되어 집니다. 데이터 프레임 내에 있는 모든 변수 (즉, 벡터) 는 동일한 길이를 가져야 합니다. 데이터 프레임을 단순히 추가적인 모드와 속성을 가진 행렬로 보아도 무방합니다. 행렬에서 인덱스 벡터와 인덱스 행렬을 이용하여 행렬의 서브셋을 구한 것과 동일한 방법으로 데이터프레임의 일부분을 서브셋 할 수 있습니다.

Making data frames

데이터 프레임은 아래의 명령문에서 보여지는 바와 같이 data.frame이라는 함수내에 데이터 프레임에서 사용될 각각의 열에 대한 변수명과 이에 상응하는 데이터 벡터를 지정함으로서 생성할수 있습니다. 위의 예제에서 사용된 statef 는 이산형 데이터를 가지는 벡터이고, incomes 는 연속형 데이터를 가지는 벡터, 그리고 incomef는 구간화 된 값을 가지는 벡터입니다. 따라서 새로이 생성될 데이터 프레임 accountants 은 home, loot, 그리고 shot 이라는 세가지 변수를 가질 것인데, home 이라는 변수는 statef의 데이터가, loot 라는 변수에는 incomes의 데이터가, 그리고 shot 이라는 변수에는 incomef 이라는 데이터 벡터가 이용될 것이라는 것을 의미하게 됩니다.

```
> accountants <- data.frame(home=statef, loot=incomes, shot=incomef)
```

만약 사용자가 이미 벡터로 구성된 리스트를 가지고 있다면, as.data.frame() 이라는 함수를 이용하여 리스트를 데이터 프레임의 형식으로 강제변환 시킬 수 있습니다. 데이터 프레임을 생성하는 가장 편리한 방법은 read.table() 이라는 함수를 이용하여 컴퓨터에 저장되어 있는 데이터 파일을 불러들어오는 것입니다. 이것에 대해서는 Reading data from files 에서 자세히 소개합니다.

attach() and detach()

여러분은 스트링 ($) 연산자를 이미 리스트를 구성하는 개별 컴포넌트를 엑세스할때 사용해 보았습니다. 그러나, 데이터 프레임 accountants 에 정의 되어 있는 home 이라는 변수를 엑세스 하기 위해서 매번 accountants$home와 같이 사용하는것은 많은 불편함이 있습니다. 따라서 R 은 attach() 라는 함수를 제공하여 이러한 불편함을 줄이고자 했습니다.

예를들어 lentils$u, lentils$v, lentils$w라는 세개의 변수로 구성된 lentils 이라는 데이터 프레임이 있다고 가정합니다. 아래의 명령문에 사용된 attach() 란 함수의 사용은 R의 내부탐색경로를 자동으로 position 1 (워크 스페이스에 있는 객체를 검색하는 최우선 탐색경로) 에서 position 2 (워크 스페이스 내에 존재하는 특정 객체의 내부를 검색하는 차선 탐색경로)로 변경하는 역할을 수행합니다. 따라서, attach(lentils) 라는 명령은 워크스페이스 내의 특정한 lentils 라는 데이터 프레임의 내부에서 이용되고 있는 u, v, w 이라는 객체들을 데이터 프레임의 이름없이 사용할 수 있도록 해줍니다. (만약, 전산지식이 있는 분이시라면 네이스페이스의 개념으로 이해하시는 것이 편리합니다).

```
> attach(lentils)
```

attach() 를 사용하기 전에 반드시 알아야 할 점이 있습니다. 예를들어, 아래의 명령은 lentils 내 에 정의되어 있는 변수 v 와 w 를 이용하여 연산을 수행한 결과를 u 에 할당할 것을 의미합니다. 그러나 연산결과는 데이터 프레임 lentils 내에 정의된 u 에 저장되지 않습니다. 연산결과의 할당은 최우선 탐색경로 position 1 로 재변경되어 새로운 u 라는 벡터를 생성해 내기 때문입니다.

```
> u <- v+w
```

만약, lentils 라는 데이터 프레임내에 있는 u 라는 변수에 연산결과를 할당하고 싶다면 반드시 아래의 명령문에서 보이는 것과 같이 $ 연산자를 사용하여야 합니다. 그러나, 변경된 u 의 값들은 lentils 라는 데이터 프레임이 detach() 라는 함수가 사용될 때까지는 육안으로 확인할 수 없습니다.

```
> lentils$u <- v+w
```

아래의 detach() 란 함수는 현재 사용하고 있는 특정 데이터 프레임에 대한 탐색경로 position 2 를 position 1 로 재변경을 해주는 역할을 합니다.

```
> detach()
```

여러 개의 데이터의 프레임을 동시에 attach()라는 함수를 이용하여 사용할 수 있기 때문에 detach(lentils) 혹은 ("lentils") 라고 데이터 프레임의 이름을 정확히 명시하는게 프로그래밍의 오류를 줄이는 방법입니다.

Note: In R lists and data frames can only be attached at position 2 or above, and what is attached is a copy of the original object. You can alter the attached values via assign, but the original list or data frame is unchanged.

Working with data frames

만약 여러분들이 여러 가지 분석작업을 동일한 워킹 디렉토리에서 수행하고 있다면, 아래에 나열한 몇 가지 방법들이 오류를 최소화하는데 도움이 될 것입니다.

문제를 세분화하여, 문제별로 의미가 분명한 데이터 프레임의 이름을 부여하고, 데이터 프레임내에 정의된 변수들도 그 의미를 쉽게 알수있는 변수명을 사용하도록 합니다. 특정 데이터 프레임을 position 2 로 이동시켜 작업을 하되, 임시변수의 생성 및 연산은 position 1 에서 수행하도록 합니다. position 2 에서의 특정 데이터 프레임과 연관된 연산이 끝났다면 $ 를 이용하여 그 연산결과를 저장한뒤 detach()를 이용하여 position 1 으로 돌아옵니다. 마지막으로 워킹 디렉토리에 있는 임시적으로 사용했던 변수들은 모두 삭제하도록 합니다.

Attaching arbitrary lists

실제로 attach() 는 제네릭 함수 (generic function) 이므로 데이터 프레임을 특정탐색경로에 연결하는 작업 외에도 다른 크래스의 객체와의 연결에도 사용이 가능합니다. 특히 객체의 모드가 "list" 이라면 데이터 프레임과 동일한 방법으로 사용이 됩니다.

```
> attach(any.old.list)
```

즉, attach() 에 의해서 연결된 리스트형의 객체는 detach 라는 함수에 의해서 그 연결이 종료됩니다.

Managing the search path

아래에서 사용된 search 라는 함수는 현재의 객체탐색경로를 보여줌으로서 사용자가 현재 연결하여 작업하고 있는 데이터 프레임 및 리스트를 확인할 수 있도록 해줍니다. 만약, 어떠한 객체도 연결되지 않았다면 아래와 같은 결과를 보여줍니다. 여기에서 .GlobalEnv 이라는 것은 현재 작업하고 있는 워크 스페이스를 의미합니다. 3

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

만약 lentils 이라는 데이터 프레임이 연결되었다면, 아래와 같은 결과를 보여지게 됩니다.

```
> search()
[1] ".GlobalEnv"   "lentils"      "Autoloads"    "package:base"
```

```
> ls(2)
[1] "u" "v" "w"
```

ls 혹은 objects 함수를 이용하여 우리는 lentils 라는 데이터 프레임이라는 어떤 객체가 있는지 확인해 볼 수 있습니다. 여기에서 함수 ls() 안에 사용된 인자 2 이라는 것은 2단계 탐색경로를 의미하는 것입니다.

마지막으로 사용하고 있던 lentils 라는 데이터 프레임을 탐색경로로부터 연결해제를 한다면 아래와 같은 결과를 볼 수 있습니다.

```
> detach("lentils")
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

Reading data from files

용량이 큰 데이터는 키보드로 입력하는 것보다는 외부에 저장되어 있는 데이터파일을 현재 작업하고 있는 R 세션으로 불러들이는 것이 편리하고 빠른 방법입니다. R 의 입력기능은 간단하지만, 다소 불편한 점이 없는 것은 아닙니다. R 의 개발자는 사용자가 파일 에디터 혹은 Perl 이라는 프로그래밍 언어를 이용하여 R 이 데이터를 읽어들일수 있는 형식으로 변경할 수 있을 것이라는 가정하에 입력기능을 개발하였습니다. 4 번거로워 보이지만, 실제로 이 과정은 매우 간단합니다.

우리는 R이 데이터를 쉽게 불러올 수 있도록 사용자가 데이터 파일을 데이터프레임과 같은 형식을 가지도록 편집하는 것을 권장합니다. 만약, 데이터가 데이터 프레임과 같이 정리가 되어 있다면 read.table() 이라는 함수를 이용하여 데이터 파일안에 있는 모든 데이터를 R 세션내로 불러들일 수 있습니다. read.table() 외에도 scan() 이라는 함수를 이용하여 직접적으로 불러올 수 도 있습니다.

데이터의 입출력에 관한 더 자세한 사항에 대해서는 R Data Import/Export 이라는 문서를 참조하시길 바랍니다.

The read.table() function

데이터 프레임 전체를 한번에 불러오기 위해서는, 외부 파일이 특정한 형식을 가지고 있어야만 합니다.

데이터 파일내의 첫번째 줄은 데이터 프레임 안으로 불러올 데이터들의 변수명이 저장되어 있어야 합니다. 데이터 파일내의 변수명 다음 줄 부터는 행번호와 함께 변수값이 저장되어 있어야 합니다. 따라서 우리는 사용자가 데이터 파일을 아래와 같은 데이터 프레임의 형식을 갖출 것을 권장합니다.

```
Input file form with names and row labels:

     Price    Floor     Area   Rooms     Age  Cent.heat
01   52.00    111.0      830     5       6.2      no
02   54.75    128.0      710     5       7.5      no
03   57.50    101.0     1000     5       4.2      no
04   57.50    131.0      690     6       8.8      no
05   59.75     93.0      900     5       1.9     yes
...
```

기본적으로 R이 데이터 파일을 읽어올때, 행번호를 제외하고서 변수가 수치형 값을 가지면 수치형으로 변수가 수치형이 아니면 요인 (factor) 의 형식으로 자동변환되어 집니다. 위에 사용된 데이터 파일을 예로 들면, Price, Floor, Area, Rooms, Age 는 수치형으로 읽어들이지만, Cent.heat 라는 변수는 요인으로 읽어들입니다. 따라서 위와 같은 데이터 프레임의 형식을 가지고 있는 데이터 파일이라면 read.table() 이라는 함수를 아래와 같이 사용하여 데이터를 불러들이면 됩니다.

```
> HousePrice <- read.table("houses.data")
```

아래에 보이는 것과 같이 종종 데이터 파일에는 행번호가 없는 경우가 있거나, 혹은 사용자가 행번호를 원하지 않는 경우가 존재합니다.

```
Input file form without row labels:

Price    Floor     Area   Rooms     Age  Cent.heat
52.00    111.0      830     5       6.2      no
54.75    128.0      710     5       7.5      no
57.50    101.0     1000     5       4.2      no
57.50    131.0      690     6       8.8      no
59.75     93.0      900     5       1.9     yes
...
```

이런경우에는 아래와 같이 read.table() 이라는 함수를 사용할 때 header 라는 옵션을 이용하면 됩니다.

```
> HousePrice <- read.table("houses.data", header=TRUE)
```

The scan() function

먼저 input.dat 이라는 데이터 파일이 길이가 같은 세개의 벡터를 가지고 있다고 가정합니다. 또한, 이 세개의 벡터중 하나는 문자형이고, 나머지 두개는 수치형이라고 가정합니다. 아래의 코드는 scan() 함수를 이용하여 input.dat 이라는 데이터 파일을 읽어들이는 방법을 보여주고 있습니다.

```
> inp <- scan("input.dat", list("",0,0))
```

위에서 보이는 것과 같이 scan() 함수의 두번째 인자는 리스트라는 형식을 요구합니다. 또한, 이 리스트의 인자는 세개의 변수가 가지는 데이터형을 지정합니다. 큰 따옴표는 문자형을 의미하고, 0 이란 수치형을 의미합니다. 따라서, scan() 함수를 통해서 불러들인 input.dat 데이터 파일은 inp 이라는 리스트에 데이터를 저장하게 되므로 아래와 같은 방법으로 각각의 변수를 엑세스 할 수 있습니다.

```
> label <- inp1; x <- inp2; y <- inp3
```

위에서 소개한 방법보다 더 편한 방법은 아래의 코드에서 보이는 것과 같이 리스트를 통해 불러올때 인자에 바로 이름을 부여하는 것입니다.

```
> inp <- scan("input.dat", list(id="", x=0, y=0))
```

이처럼 불러들어온 데이터가 리스트 형이므로 아래와 같은 방법으로 각각의 변수를 엑세스 할 수 있습니다.

```
> label <- inp$id; x <- inp$x; y <- inp$y
```

또는 inp 이라는 리스트를 제 2 탐색경로 (즉, position 2) 에 연결시켜 엑세스 할 수도 있습니다. (see Attaching arbitrary lists).

Accessing builtin datasets

R 은 기본적으로 대략 100여개의 데이터셋을 datasets 이라는 패키지를 통하여 제공하고 있습니다. 만약 R 이 제공하는 데이터셋의 목록을 확인하고 싶으시다면 아래의 명령어를 이용하면 됩니다.

```
data()
```

R-2.0.0 버전부터 제공되는 모든 데이터셋은 아래에서 보여지는 것과 같이 데이터셋의 이름을 통하여 바로 엑세스 할 수 있습니다.

```
data(infert)
```

Loading data from other R packages

만약 특정 패키지 안에서 사용되는 데이터셋에 대해서 알고 싶으시다면 아래에 보이는 것과 같이 data() 함수 내에 있는 package 라는 인자를 이용하시면 됩니다.

```
data(package="rpart")
data(Puromycin, package="datasets")
```

만약 특정 패키지를 library 라는 함수를 통해서 현재 작업하고 있는 워크스페이스에 연결시켰다면, 패키지에 포함된 데이터셋들은 자동으로 연결되어 검색되어집니다.

Editing data

만약 사용자가 데이터 프레임이나 행렬의 일부내용을 수정 및 편집을 하고 싶을때 edit 이라는 함수는 윈도우즈 기반의 스프레드시트 형식의 편집기능을 제공해 주므로 매우 편리하게 작업을 할 수 있습니다.

```
> xnew <- edit(xold)
```

위에서 사용된 edit() 이라는 함수는 xold 이라는 데이터셋을 편집한 뒤, 편집된 새로운 내용을 xnew 라는 데이터셋으로 저장한다는 의미입니다. 만약, 사용자가 xold 라는 원본 데이터셋 자체를 변경하고자 한다면 fix(xold) 라는 함수를 이용하시길 바랍니다. 이것은 xold <- edit(xold) 와 동일한 표현입니다.

만약에 여러분이 비어있는 데이터 프레임으로부터 윈도우즈 기반의 스프레드시트를 통하여 새로운 데이터셋을 생성하고자 한다면 아래에서 보여지는 것과 같이 명령어를 입력하시면 됩니다.

```
> xnew <- edit(data.frame())
```

Probability distributions

R as a set of statistical tables

R의 많은 장점들 중 한가지는 다양한 통계분포표를 제공하는 것입니다. 이것은 P(X <= x)를 만족하는 누적확률계산, 주어진 확률함수로부터 밀도계산, 주어진 q에 해당하는 퀀타일 (quantile)을 계산, P(X <= x) > q)를 만족하는 x 값 찾기, 그리고 주어진 분포로부터의 난수를 생성하는데 사용되어집니다.

확률분포함수	R 함수이름	필요한 인자들
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	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
signed rank	signrank	n
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

여기에 나열된 R 함수의 이름앞에 'd'를 붙여 사용하면 확률밀도를 구하기 위한 함수명이 되고, 'p' 를 붙이면 누적확률값을 구할 수 있는 함수명이 되고, 'q' 를 붙이면 퀀타일을 구할수 있는 함수명이 되며, 'r' 을 붙이는 것은 난수를 생성하는 함수명을 가지게 됩니다. 한가지 주의할 점은 R 이 제공하고 있는 모든 함수가 non-centrality parameter (한국말 모름)을 지원하는 것은 아닙니다. 더 자세한 사항은 온라인 도움말을 참고해주세요.

'p'로 시작하는 확률함수 pxxx와 'q'로 시작하는 확률함수 qxxx 들은 lower.tail와 log.p 라는 인자들을 가집니다. 또한 'd' 로 시작하는 확률함수 dxxx는 log 라는 인자를 가집니다. 아래의 코드가 보여주는 것과 같이 이러한 인자들의 활용은 생존분석에서 사용되는 H(t) = − log(1 − F(t))라는 해저드 (hazard) 함수의 연산을 가능하게 해줍니다.

```
- pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
```

또한, dxxx(..., log = TRUE)) 와 같은 인자의 설정은 더 정교한 로그라이클리후드 (log-likelihood) 값을 얻을 수 있도록 도와줍니다.

더 나아가 정규분포로부터 얻은 샘플에 대한 스튜던타이즈된 범위에 대한 분포 (the distribution of the studentized ranged of samples from a normal distribution) 은 ptukey 와 qtukey 라는 함수를 통하여 얻을 수 있습니다. 또한, 멀티노미얼 (multinomial) 분포에 대한 밀도값과 난수생성은 dmultinom 과 rmultinom 이라는 함수를 통하여 얻을 수 있습니다. 더 많은 분포에 대해서는 SuppDists 라는 패키지를 검색해보시길 바랍니다.

밀도함수를 활용하는 두 가지 예제입니다.

```
> ## 2-tailed p-value for t 분포
> 2*pt(-2.43, df = 13)
> ## 상위 1% point for an F(2, 7) 분포
> qf(0.01, 2, 7, lower.tail = FALSE)
```

R이 어떻게 난수 생성을 하는가에 대해서 알고 싶으시다면 온라인 도움말 RNG (random number generator의 약자) 를 참고해주세요.

Examining the distribution of a set of data

우리는 많은 방법을 통하여 주어진 (일변량) 데이터의 분포를 확인해 볼 수 있습니다. 가장 쉬운 방법은 summary 혹은 fivenum이라는 함수를 이용하여 수치적 요약정보를 보는 것입니다. 혹은 stem 이라는 함수를 이용하여 줄기-잎 그림을 통하여 확인해 볼수도 있습니다.

```
> attach(faithful)
> summary(eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 000000335778888800223355557778
  42 | 033355557788002333335555777778
  44 | 022223355577800000000233333577788888
  46 | 000023335770000002357 8
```

```
    48 ¦ 00000022335800333
    50 ¦ 0370
```

실은 줄기-잎 그림은 R 에서 제공하는 hist라는 함수가 제공하는 히스토그램과 비슷합니다.

```
> hist(eruptions)
## 구간을 좀 더 좁게하고, 추정된 밀도함수를 히스토그램위에 오버랩해서 보여주기
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # 실제 데이터 포인트도 함께 출력하기
```

위의 예제에서 사용된 것과 같이 히스토그램보다 좀 더 정교한 확률분포를 살펴보기 위해서 density 라는 함수에 의해서 계산되는 밀도플랏 (density plot) 을 함께 사용하는 것이 좋습니다. density 함수가 가지는 인자중에서 bw 이라는 bandwidth (한국말모름) 가지는 0.1 이라는 값은 기본으로 설정된 값이 너무 smoothing 이 되어서 trial-and-error 에 의해서 정해진 것입니다. (Bandwidth의 선택에 있어서 자동화된 방법이 있는데, bw='SJ' 를 이용해보시길 바랍니다).

확률분포를 확인해보기 위해서 우리는 또한 ecdf 라는 함수에 의해서 얻어지는 empirical cumulative distribution function (정확한 한국말 모름)을 이용해 볼 수 있습니다.

```
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

당연히 empirical 누적분포는 그 어떠한 표준분포를 따르지 않습니다. 하지만, 화산폭발이 3분이상 지속될 경우에 대한 분포는 어떨까요? 3분이상의 데이터으로부터의 생성된 empirical 누적분포를 만들어 본뒤 그 위에 정규분포와 오버레이시켜서 비교해 봅니다.

```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```

퀀타일-퀀타일 (Q-Q) 플랏을 이용하여 화산폭발 3분이상의 데이터에 대한 정규성을 검토해보는 것도 도움이 됩니다.

```
par(pty="s")       # arrange for a square figure region
qqnorm(long); qqline(long)
```

Q-Q 플랏으로부터 정규성을 보임을 확인할 수 있었지만, 오른쪽 꼬리 부분이 생각보다 정규분포를 따르지 않는 것 같습니다. 그럼 이것을 t분포로부터 생성된 난수의 분포와는 어떤 관계를 가질까요?

```
x <- rt(250, df = 5)
qqnorm(x); qqline(x)
```

t분포로부터 얻어진 난수의 분포가 정규분포보다도 더 긴 꼬리를 가짐을 볼 수 있습니다.

```
qqplot(qt(ppoints(250), df = 5), x, xlab = "Q-Q plot for t dsn")
qqline(x)
```

아마도 그 다음 과정은 R이 제공하는 Shpiro-Wilk 라는 테스트를 통하여 이 분포가 정규성을 얼마나 따르는지 확인하는 것입니다.

```
> shapiro.test(long)

        Shapiro-Wilk normality test

data:  long
W = 0.9793, p-value = 0.01052
```

Shpiro-Wilk 라는 테스트 외에도 Kolmogorov-Smirnov 라는 테스트를 이용하여 정규성을 확인해 볼 수 있습니다.

```
> ks.test(long, "pnorm", mean = mean(long), sd = sqrt(var(long)))

        One-sample Kolmogorov-Smirnov test

data:  long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

One- and two-sample tests

여지까지 우리는 하나의 샘플을 정규분포와 비교해 보았습니다. 두개의 샘플을 비교하고자 한다면 더 많은 작업이 요구되나, 많은 작업이 공통적으로 적용되므로 많은 중복작업을 하는것과 같습니다. 다음의 예제에 사용되는 통계적 테스트를 포함하여 모든 전형적인 통계테스트는 기본적으로 stats 이라는 패키지에 포함되어 있습니다.

1995년에 Rice가 the latent heat of the fusion of ice 에 대해서 연구한 논문의 490 쪽에 게재된 내용입니다. 다음 두개의 데이터셋이 주어졌습니다.

```
Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
          80.05 80.03 80.02 80.00 80.02
Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

박스플랏 (boxplot) 은 두개의 샘플에 대한 정보를 시각적으로 비교해 줍니다.

```
A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02

B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

boxplot(A, B)
```

첫번째 그룹이 두번째 그룹보다 더 높은 결과를 보여주는 것같습니다.

각각의 데이터의 평균이 일치하는가를 테스트 하기 위해서 우리는 아래와 같이 unpaired t-테스트를 수행해봅니다.

```
> t.test(A, B)

        Welch Two Sample t-test
```

```
	data:  A and B
	t = 3.2499, df = 12.027, p-value = 0.00694
	alternative hypothesis: true difference in means is not equal to 0
	95 percent confidence interval:
	 0.01385526 0.07018320
	sample estimates:
	mean of x mean of y
	 80.02077  79.97875
```

테스트의 결과는 평균의 차이가 정규분포를 따른다고 가정할때 그 평균의 차이는 통계적으로 의미가 있음을 나타내고 있습니다. 여기에서 한가지 알아야 할 점은 R 은 기본적으로 두 데이터의 분산이 같다고 가정하지 않는다는 것입니다. 이것은 S-Plus 에서 제공하는 t.test 와 동일합니다. 따라서 우리는 두 데이터의 분산이 같은지에 대해서 F-테스트를 시행해 봅니다. 이때, 두 데이터는 정규분포로부터 얻어진 샘플이라고 가정해야 합니다.

```
	> var.test(A, B)

	        F test to compare two variances

	data:  A and B
	F = 0.5837, num df = 12, denom df =  7, p-value = 0.3938
	alternative hypothesis: true ratio of variances is not equal to 1
	95 percent confidence interval:
	 0.1251097 2.1052687
	sample estimates:
	ratio of variances
	         0.5837405
```

테스트의 결과는 두 데이터의 분산은 통계학적으로 차이가 없음을 의미하므로, 우리는 두 데이터의 분산이 같다고 가정하는 전형적인 t-테스트를 사용해도 무방합니다.

```
	> t.test(A, B, var.equal=TRUE)

	        Two Sample t-test

	data:  A and B
	t = 3.4722, df = 19, p-value = 0.002551
	alternative hypothesis: true difference in means is not equal to 0
	95 percent confidence interval:
	 0.01669058 0.06734788
	sample estimates:
	mean of x mean of y
	 80.02077  79.97875
```

우리가 사용한 모든 테스트들이 두 데이터가 정규분포로부터 얻어진 샘플임을 가정하고 있습니다. 그러나 two-sample Wilcoxon 혹은 Mann-Whitney 테스트는 단순히 연속형 분포만을 가정합니다.

```
	> wilcox.test(A, B)

	        Wilcoxon rank sum test with continuity correction

	data:  A and B
	W = 89, p-value = 0.007497
	alternative hypothesis: true location shift is not equal to 0

	Warning message:
	Cannot compute exact p-value with ties in: wilcox.test(A, B)
```

테스트를 수행한 결과에 경고가 있음을 알 수 있습니다. 이것은 아마도 각각의 샘플안에 동일한 값을 가지는 데이터들이 존재하기 때문일것입니다. 따라서 우리는 연속형 분포를 가정하기 보다는 데이터가 이산형

분포로부터 나왔다고 생각하는게 올바를 것입니다. (혹은 이것은 rounding 으로부터 나온 결과일 수 도 있습니다).

두개의 샘플을 시각적으로 비교해 보는데 여러가지 방법이 있을 수 있습니다. 그 중에서 두개의 박스플랏을 동시에 비교해보는 것은 이미 해 보았습니다.

```
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

위에서 사용된 코드는 두개의 emprical CDFs (누적확률분포)를 비교할 것입니다. 또한 qqplot함수를 이용하여 두 개의 샘플에 대한 Q-Q 플랏을 생성할 수도 있습니다. 아래의 코드는 두개의 ecdf(empirical CDF)의 차이에 대하여 Kolmogorov-Smirnov 테스트를 수행합니다.

```
> ks.test(A, B)

        Two-sample Kolmogorov-Smirnov test

data:  A and B
D = 0.5962, p-value = 0.05919
alternative hypothesis: two-sided

Warning message:
cannot compute correct p-values with ties in: ks.test(A, B)
```

Grouping, loops and conditional execution

Grouped expressions

R은 결과값을 반환하는 함수를 사용하고, 이 결과를 재사용할 수 있도록 표현식에 할당하며, 이 표현식이 다시 재사용될 수 있다는 점에서 표현식을 기반으로 하는 언어라고 할 수 있습니다. 특히 다중 할당도 가능하다는 것이 특징입니다. 또한, {expr_1; ...; expr_m} 와 같이 여러개의 명령어들을 중괄호로 한데 묶어 이것을 또한 표현식으로도 사용할 수 있습니다 이것을 그룹화된 표현식이라고 하고, 이 그룹화된 표현식의 결과는 그룹내의 맨 마지막 명령문이 실행되었을때 얻어지게 됩니다. 이렇게 그룹화된 표현식 역시 괄호안의 일부분 혹은 더 큰 표현식의 일부분으로도 사용될 수 있습니다.

Control statements

Conditional execution: if statements

조건문은 아래와 같은 형식을 가집니다.

```
> if (expr_1) expr_2 else expr_3
```

expr_1 은 반드시 한개의 참 혹은 거짓 이라는 논리형 값을 가져야만 합니다.

"short-circuit" 이라고 불리는 && 와 || 이라는 연산자는 종종 if 라는 조건문과 함께 쓰입니다.

R 은 벡터라이즈(vectorized) 된 버전의 if/else 조건문인 ifelse 라는 함수를 제공하고 있습니다. 이것은 ifelse(condition, a, b) 의 형식을 가지고 만약 condition[i] 이 참이면 a[i]를, 거짓이면 b[i]를 결과값으로 가지게 됩니다.

Repetitive execution: for loops, repeat and while

다음은 for를 이용한 반복문에 대한 형식입니다.

```
> for (name in expr_1) expr_2
```

name은 1:20 과 같이 벡터로서 표현된 expr_1에 정의된 범위내에서 expr_2라는 표현식의 반복을 컨트롤하는 지시자, 즉, 루프 변수 (loop variable) 입니다. expr_2 은 종종 그룹화된 표현식으로 오는 경우가 있습니다.

예를들어, ind가 그룹멤버쉽을 나타낸다고 가정할때 그룹별로 y 와 x의 관계를 살펴 볼 수 있는 플랏 (plot)을 생성하고 싶은 경우를 생각해 봅시다. 이것을 수행하기 위한 한가지 옵션은 어떤 특정요인이 가지고 있는 레벨별로 플랏을 생성해주는 함수 coplot(),5 을 이용하는 것입니다. 아래의 코드는 coplot() 을 사용하지 않고도 이를 수행하는 방법을 보여줍니다.

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
    plot(xci, yci)
    abline(lsfit(xci, yci))
  }
```

위의 코드에서 사용된 split() 이란 함수는 데이터를 특정요인이 가지는 레벨별로 분리해 낸 뒤, 각각의 레벨별 데이터를 컴포넌트로 가지는 리스트를 생성하는 기능을 가지고 있습니다. 이것은 주로 박스플랏과 함께 사용되는데 매우 유용한 함수입니다. 더 자세한 사항은 도움말 기능을 이용하시길 바랍니다.

R을 이용하여 프로그래밍을 할때 for() 를 이용한 반복문은 자주 사용되지 않습니다. 그 이유는 전체 데이터를 이용하여 벡터라이징 테크닉을 적절히 활용하는 것이 for() 를 이용한 반복문보다도 더 빠른 연산속도를 가지며, 프로그래머의 입장에서 프로그램을 직관적으로 읽기 쉽고 이해가 빠르게 도와줍니다. R 은 반복기능에 있어서 다음과 같은 repeat 와 while 이라는 기능을 제공하고 있습니다.

```
> repeat expr
```

```
> while (condition) expr
```

break 와 next 라는 두 명령문은 반복문을 중단하고자 쓰이지만, 중요한 차이가 있습니다. break 는 반복문을 무조건 중단하고 반복문 이후의 명령들을 실행하고자 할때 쓰입니다. 따라서 repeat 을 사용하여 expr을 반복하던 중, 이를 중단하고자 할때는 break 이라는 명령문을 사용하면 됩니다. 그러나, next 라는 명령어는 현재 진행하고 있는 반복 단계만을 건너뛰고 다음 단계의 반복단계로 넘어가라는 의미입니다.

실제로 프로그램의 흐름을 제어하는 것은 대개 사용자정의함수를 작성할때 다루어집니다. 이에 관련된 예제들에 대해서는 Writing your own functions 에서 다룰 것입니다.

Writing your own functions

R은 방대한 양의 함수를 지원하고 있으나, 사용자가 직접 function(함수)라는 모드를 가진 객체를 생성할 수 있도록 지원합니다. 이처럼 R이 사전에 미리 제공하는 함수들은 built-in (내장) 함수라고 하며, 사용자가 직접 정의하는 함수를 사용자 정의함수라고 합니다. 사용자 정의 함수를 작성하는 것은 여러분들이 R

을 더 효율적으로, 기능적으로, 그리고 생산적인 통계분석처리를 할 수 있도록 도와줍니다.

실제로 R시스템에 미리 내장되어 있는 많은 함수들 중의 대부분은 사용자 정의 함수입니다. 예를 들면, mean(), var(), postscript() 과 같은 것들이 대표적인 경우입니다.

다음을 함수를 정의하는 방법입니다.

```
> name <- function(arg_1, arg_2, ...) expression
```

위에서 사용된 expression 이란 arg_1, arg_2, ..., 을 이용하여 작성되는 R의 표현식 혹은 그룹화된 표현식입니다. 또한, name이라는 함수가 arg_1, arg_2, ..., 라는 인자들을 이용하여 expression 을 수행함으로 얻어진 최종적인 결과는 name 함수의 반환값이라고 합니다.

사용자가 정의한 함수를 사용하는 방법은 일반적인 내장함수를 사용하는 것과 동일하며, R 시스템 어느곳에서나 name(expr_1, expr_2, ...)이라고 명령하면 됩니다.

Simple examples

첫번째 (인위적인) 예제입니다. 두개의 샘플로부터 t-통계량을 구하는 함수를 다음과 같이 정의할 수 있습니다.

```
> twosam <- function(y1, y2) {
    n1  <- length(y1); n2  <- length(y2)
    yb1 <- mean(y1);   yb2 <- mean(y2)
    s1  <- var(y1);    s2  <- var(y2)
    s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
    tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
    tst
  }
```

이렇게 함수가 정의되고 나면, 여러분은 아래와 같이 정의한 함수 twosam 를 사용할 수 있습니다.

```
> tstat <- twosam(data$male, data$female); tstat
```

두번째 예제는 Matlab을 이용하여 회귀분석시 최소제곱법을 이용하여 회귀계수를 구해주는 백슬래쉬(backslash)라는 명령어를 구현하는 것입니다. 본래 이것은 qr() 함수를 이용하여 할 수도 있으나, 어떤 경우에는 직접적으로 활용하기가 곤란한 경우가 있습니다. 따라서, 우리는 다음과 같은 방식을 사용하는 것을 권장합니다.

만약, n개의 원소를 가지는 열벡터 y 가 있고, 사이즈가 n행 p열을 가진 디자인 매트릭스 X 가 있다면 X ₩ y 라는 것은 (X'X)^{-}X'y, 이라고 정의됩니다. 여기에서 (X'X)^{-}라는 것은 X'X의 역행렬입니다.

```
> bslash <- function(X, y) {
  X <- qr(X)
  qr.coef(X, y)
}
```

이렇게 정의된 bslash 라는 함수는 아래와 같이 regcoeff 라는 객체에 결과를 저장하게 됩니다.

```
> regcoeff <- bslash(Xmat, yvar)
```

이와 동일한 기능은 R에서 제공하는 lsfit() 함수를 이용하여 수행할 수 있는데, lsfit() 함수의 이용이 좀 더 효율적일 수 있습니다. 6. 실제로, qr() 과 qr.coef() 라는 함수의 사용은 회귀계수를 구하는데 있어서 직관적이라고 보기 어려우며, 사용에 번거롭기도 합니다. 따라서, 우리는 이러한 번거로움을 줄이고 매우 편리한 방법으로 사용자가 직접 정의해서 사용하는 binary operator (사용자정의 바이너리 연산자)라는 기능을 제공합니다.

Defining new binary operators

위에서 bslash()이라는 함수를 작성하였습니다. 그러나 이러한 연산의 수행은 함수를 작성하는 것보다 연산자의 형태를 지닌다면 훨씬 편리할 것입니다. 따라서 우리는 binary operator 라는 사용자정의 바이너리 연산자 기능을 아래와 같이 제공합니다.

```
%anything%
```

예를 들어, anything 이라는 부분에 느낌표 (!) 를 이용하여 %!% 이라는 새로운 바이너리 연산자를 정의하고자 한다면 아래와 같이 큰 따옴표를 활용하여 연산자의 이름을 정의하면 됩니다.

```
> "%!%" <- function(X, y) { ... }
```

이렇게 새로이 정의된 연산자는 X %!% y와 같이 사용할 수 있습니다.

행렬의 곱셈을 수행하는 연산자 %*%, outer product 를 수행하는 연산자 %o%, 그리고 그외의 연산자들은 바이너리 연산자의 대표적인 예입니다.

Named arguments and defaults

Generating regular sequences에서 설명했던 것과 같이 함수에 사용되는 인자들은 “name=object” 와 같은 형식을 가지고 있으며, 이렇게 함수를 정의할때 인자의 이름을 굳이 정의하는 것은 프로그래밍의 논리적 오류를 줄이고자 하는 노력이라고 보시면 됩니다.

예를들어, 아래와 같이 정의된 fun1 이라는 함수가 있다고 가정합니다.

```
> fun1 <- function(data, data.frame, graph, limit) {
    [function body omitted]
  }
```

위에서 정의된 함수는 아래와 같이 세가지 방법으로 사용이 가능합니다.

```
> ans <- fun1(d, df, TRUE, 20)
> ans <- fun1(d, df, graph=TRUE, limit=20)
> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

그런데, R 에서 제공하는 모든 함수들은 아래에서 보여지는 것과 같이 함수를 정의할 때 함수의 인자에 기본값이라는 것을 부여합니다.

```
> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }
```

이처럼 함수의 인자에 대해 기본값이 미리 정의되어 있기 때문에 아래와 같이 함수의 모든 인자를 입력하

지 않아도 함수는 기본값을 이용하여 연산을 수행하게 됩니다.

```
> ans <- fun1(d, df)
```

아래의 코드는 limit 라는 인자의 기본값을 20 에서 10 으로 바꾸어서 실행하라는 의미입니다.

```
> ans <- fun1(d, df, limit=10)
```

인자에 주어지는 기본값이 여기 예제에서 사용된 것과 같이 굳이 상수를 가질 필요는 없으며, 그 어떠한 표현식을 사용해도 무방합니다.

The '...' argument

함수를 정의할 때 '...' 라는 기능을 알아두면 매우 유용합니다. 예를들어, plot()과 같이 그래픽에 관련된 많은 함수들은 par() 라는 함수를 사용합니다. par()라는 함수에는 그래픽의 결과물을 정교히 조정하기 위한 방대한 양의 인자들이 포함되어 있습니다. par() 라는 함수에 대해서 자세히 알고 싶으시다면 See The par() function. 그러나 실제로는, 그래픽에 관련된 함수들은 단순히 par()라는 함수에 있는 인자를 사용함으로서 결과물의 수정 및 조정을 조정할 뿐, 이러한 역할을 위한 인자를 따로 정의하고 있지 않습니다. 이렇게 외부에 정의에 되어 있는 함수 및 객체들의 인자들을 정의하고자 하는 함수로 불러들여와 활용하고자 할 때, 아래의 코드에서 보이는 것과 같이 '...'를 함수 선언부에 나열된 인자들의 목록뒤에 붙여주면 됩니다.

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {
  [omitted statements]
  if (graph)
    par(pch="*", ...)
  [more omissions]
}
```

Assignments within functions

함수 내부에서 사용되는 변수가 가지는 어떠한 값이라도 그 값들은 함수 내부에서만 유효합니다. 이렇게 변수가 가지는 값이 지정된 함수 내부에서만 사용되는것을 지역할당(local assignment) 이라고 합니다. 이것을 이해하기 위해서는 R시스템내에서 할당범위 (the scope of assignment)라는 개념을 이해해야만 하며, 이것은 이 안내서의 수준을 벗어나므로 설명하지 않겠습니다. 그러나, 함수 내부에서 연산된 어떤 값을 함수 외부 및 작성하는 프로그램 전체에서 사용하고 싶다면, <<- 이라는 "superassignment" 이라는 연산자를 사용하거나, assign() 이라는 함수를 이용하시면 됩니다. 이렇게 변수가 가지는 값이 특정 함수내에서 국한되지 않고 전체에서 활용되는 개념을 전역할당 (global assignment 혹은 permanent assignment) 라고 합니다. 이것에 대한 더 자세한 사항은 도움말을 참조하시길 바랍니다.

S-Plus 사용자는 <<- 라는 연산자가 R 에서 다른 의미로 사용된다는 점을 반드시 알고 있어야 합니다. 이러한 다른 점들은 Scope 에서 다루어집니다.

More advanced examples

Efficiency factors in block designs

함수에 대해서 좀 더 자세히 다루기 위해서 이번에는 블락 디자인에서 사용되는 efficiency factor (한국말

모름)를 찾아보도록 하겠습니다. 이 문제에 대한 일부의 내용은 이미 Index matrices에서 다룬바 있습니다.

b레벨을 가진 blocks이라는 요인과 v레벨을 가진 varieties라는 요인을 이용한 블락 디자인을 생각해봅시다. 만약 R과 K이 각각 v행 v열을 가진 replications 행렬과 b행 b열을 가진 block size 행렬이라고 합시다. efficiency factors 는 아래와 같은 행렬의 기저값들 (eigenvalues)로 정의될 수 있습니다. E = I_v - R^{-1/2}N'K^{-1}NR^{-1/2} = I_v - A'A, 여기에서 A = K^{-1/2}NR^{-1/2} 입니다. efficiency factor를 산출하는 함수를 작성하는 것은 다음과 같습니다.

```
> bdeff <- function(blocks, varieties) {
    blocks <- as.factor(blocks)             # minor safety move
    b <- length(levels(blocks))
    varieties <- as.factor(varieties)       # minor safety move
    v <- length(levels(varieties))
    K <- as.vector(table(blocks))           # remove dim attr
    R <- as.vector(table(varieties))        # remove dim attr
    N <- table(blocks, varieties)
    A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
    sv <- svd(A)
    list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)
}
```

실제로 이경우에는 the singular value decomposition 을 이용하는 것이 기저값 (eigenvalue)를 이용한 방법보다 보다 나은 연산결과를 줍니다.

위에서 작성된 함수는 the efficiency factors 만을 결과값으로 돌려주는 것만이 아니라 block 과 variety 라는 요인의 canonical contrast (한국말 모름) 또한 알려줍니다. 그 이유는 이 두가지 결과값들이 때때로 유용한 qualitative information (한국말 모름) 을 제공하기 때문입니다.

Dropping all names in a printed array

사이즈가 큰 행렬 혹은 배열의 출력이 주목적인 경우, 행렬과 배열의 구성요소의 위치를 알려주는 행, 열, 혹은 차원에 대한 정보를 나타내지 않고 오로지 데이터값만을 출력하는 것이 유용한 경우가 있습니다. 이런 경우에는 dimnames 이라는 속성이 가지는 문자열값들을 초기화 시켜주면 됩니다. 예를 들어 행렬 X를 행과 열의 정보 없이 데이터만 출력하고자 할 경우, 아래와 같이 코드를 작성하시면 됩니다.

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))
> temp; rm(temp)
```

우리는 또한 이와 같은 기능을 아래와 같이 no.dimnames()이라는 사용자정의 함수를 작성함으로서 수행할 수 도 있습니다. 이것은 또한 이러한 유용한 기능을 위한 사용자정의함수가 꼭 길지 않아도 됨을 잘 보여주고 있습니다.

```
no.dimnames <- function(a) {
  ## Remove all dimension names from an array for compact printing.
  d <- list()
  l <- 0
  for(i in dim(a)) {
    d[[l <- l + 1]] <- rep("", i)
  }
  dimnames(a) <- d
  a
}
```

위와 같이 정의된 함수는 아래와 같이 사용함으로서 배열이 가지는 데이터 값들만 출력해낼수 있습니다.

```
> no.dimnames(X)
```

이번 섹션에서 다룬 내용은 배열 데이터 값 자체 보다 배열의 값들이 가지는 패턴을 분석하고자 할때 더 유용하므로 주로 사이즈가 큰 정수형 배열을 다룰때 사용됩니다 .

Recursive numerical integration

함수는 재귀적인 용법으로도 사용할 수 있으며, 또한 정의하고자 하는 함수 내부에 필요한 함수를 정의하고 사용할 수 있습니다. 이렇게 정의하고자 하는 함수 내부에서 새로이 정의된 함수들은 제1탐색경로에서 검색될 수 있는 객체로 인식되지 않는다는 점에 명심해야 합니다. 정의하고자 하는 함수 내부에서 새로이 정의된 함수의 사용범위는 단순히 정의하고자 하는 함수 내부로만 제한됩니다.

아래의 예제는 1개의 변수만을 가지는 함수를 적분하는 가장 기초적인 방법에 대해서 다루고 있습니다. 적분을 하는 기본 개념은 먼저 적분하고자 하는 범위의 양쪽 끝점과 그 중간점에 해당하는 함수들의 값을 얻어낸 후, 사다리꼴 공식을 이용하여 단일 패널에서 얻어진 결과가 이중패널에서 얻어진 결과와 매우 유사할 경우 이중패널로부터 얻어진 결과를 적분값으로 결정합니다. 만약, 그렇지 않다면 각각의 패널에 위에서 묘사한 프로세스를 다시 적용하되 값의 변화가 없을때까지 반복합니다. 아래의 예제는 R프로그래밍을 하는데 있어서 다소 복잡한 부분에 해당하므로 이해가 어려울 수 도 있습니다.

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {
    ## function `fun1' is only visible inside `area'
    d <- (a + b)/2
    h <- (b - a)/4
    fd <- f(d)
    a1 <- h * (fa + fd)
    a2 <- h * (fd + fb)
    if(abs(a0 - a1 - a2) < eps || lim == 0)
      return(a1 + a2)
    else {
      return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
             fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
    }
  }
  fa <- f(a)
  fb <- f(b)
  a0 <- ((fa + fb) * (b - a))/2
  fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}
```

Scope

이번 섹션에서 다루어질 내용은 이 문서의 다른 부분들보다는 다소 기술적인 부분에 해당합니다. 그러나, 이 부분은 R과 S-Plus를 구분짓는 많은 다른 점들중에서 함수와 관련된 부분에 대해서 기술합니다.

먼저 함수의 본체에 사용되는 심볼에는 formal parameters(한국말 모름), local variables(지역변수), 그리고 free variables (자유변수) 라는 세가지 종류가 있습니다. formal parameters 라는 것은 함수 선언시 아직 값이 부여되지 않은 인자들의 목록을 의미합니다. 따라서 formal parameters는 실제로 함수가 사용될 때 함수가 받아들이는 인자의 값을 가지게 됩니다. 이에 반해, 지역변수라는 것은 함수 본체에서 사용된 표현식의 결과값들을 가지게 됩니다. 만약, 어떤 변수가 formal parameters도 아니고 지역변수도 아닌 경우, 이러한 변수를 바로 자유변수라고 합니다. 그러나, 상황에 따라서 자유변수에 값이 부여될 경우 이는 지역변수가 되기도 합니다. 더 정확한 개념의 이해를 위해서 다음의 예제를 살펴봅시다.

```
f <- function(x) {
  y <- 2*x
```

```
  print(x)
  print(y)
  print(z)
}
```

여기 정의되어 있는 f 란 함수는 x 라는 formal parameter 를 가지고, y 란 지역변수를 가지는데 반해, z 라는 것은 자유변수입니다.

R에서 자유변수에 대한 값의 할당은 변수가 관리가 되고 있는 함수에 우선권이 부여되며, 이것을 렉시컬 스코프 (lexical scope)라고 합니다. 이해를 돕기 위해서 먼저 cube라는 함수를 정의해봅니다.

```
cube <- function(n) {
  sq <- function() n*n
  n*sq()
}
```

sq라는 함수 안에 있는 변수 n 은 인자가 아니므로 자유변수이며, 이 자유변수가 어떠한 값을 가져야 하는가에 대해서 스코핑 규칙 (scoping rules)이 적용됩니다. 스태틱 스코핑 규칙을 가지는 S-Plus 에서는 자유변수가 가지는 값은 n 이라는 이름을 가진 전역변수 (global variable)에 의해서 그 값이 결정됩니다. 이에 반해, 렉시컬 스코핑 규칙을 가지는 R은 자유변수에 cube라는 함수가 실행될 때 가지는 인자의 값을 할당합니다. 그 이유는 n은 sq라는 함수가 정의되기 '바로 이전에' cube라는 함수의 내부에서 정의되어 관리되는 변수이기 때문입니다. 이것은 R과 S-Plus를 구분짓는 큰 특징들중의 하나 입니다. 즉, R의 기본적인 스코핑 규칙은 전역변수에 의한 관리가 아닌 지역에 촛점을 맞춘 렉시컬 스코핑을 따르게 됩니다.

```
## first evaluation in S
S> cube(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18
## then the same function evaluated in R
R> cube(2)
[1] 8
```

렉시컬 스코핑은 또한 변수가 가지는 값을 자동으로 변경 (mutable state) 해주기도 합니다.

다음의 예제는 R의 렉시컬 스코핑이 얼마나 효율적으로 은행계좌를 구현할 수 있는가를 보여줍니다. 은행계좌를 구현하기 위해서는 총계 (balance 혹은 total), 출금 (withdrawal), 입금 (deposits), 그리고 현재잔고 (current balance)이 필요합니다. 먼저, 출급, 입금, 그리고 잔고에 대한 함수를 먼저 만든 뒤, 이를 리스트형식으로 한데 묶어 account 라는 함수를 정의합니다. account라는 함수가 실제적으로 실행되었을때 total라는 파라미터는 특정한 값을 가진 인자가 되어 리스트에 정의되어 있는 출금, 입금, 그리고 잔고라는 함수에 그 값을 전달하게 됩니다. 그 이유는 바로 total이라는 인자는 account라는 함수에 의해서 정의되어 관리되는 변수이기 때문입니다.

아래의 코드에서 total이라는 변수와 연관된 값의 변화를 위해서 특별한 할당 연산자인 <<-를 사용하였습니다. 이 연산자의 역할은 입금, 출금, 잔액이라는 함수내부의 total이라는변수의 값을 변경하는 것이 아니라, account 라는 함수내에 정의되어 관리되어지는 total이라는 변수의 값을 변경하는 것입니다. 만약, 이 연산자를 사용하였으나 가장 상위에 정의된 함수 혹은 프로그램 전역에서도 total 이라는 변수를 찾을수가 없다면, total 이라는 변수는 새로이 생성이 되고, 새로이 생성된 변수에 그 값을 부여하게 됩니다. 7.

```
open.account <- function(total) {
  list(
    deposit = function(amount) {
      if(amount <= 0)
```

```
        stop("Deposits must be positive!\n")
      total <<- total + amount
      cat(amount, "deposited.  Your balance is", total, "\n\n")
    },
    withdraw = function(amount) {
      if(amount > total)
        stop("You don't have that much money!\n")
      total <<- total - amount
      cat(amount, "withdrawn.  Your balance is", total, "\n\n")
    },
    balance = function() {
      cat("Your balance is", total, "\n\n")
    }
  )
}

ross <- open.account(100)
robert <- open.account(200)

ross$withdraw(30)
ross$balance()
robert$balance()

ross$deposit(50)
ross$balance()
ross$withdraw(500)
```

Customizing the environment

사용자는 사용자의 작업환경을 여러가지 방법으로 커스터마이징 할 수 있습니다. 먼저, 사이트 초기화 파일이라는 것이 있고, 각각의 디렉토리마다 특정역할을 하는 초기화 파일이 있습니다. 그리고, 이들과 연관된 특수한 함수 .First와 .Last라는 함수가 있습니다.

주로 패키지의 다운로드에 관련된 저장소에 대한 정보와 연관이 있는 사이트 초기화 파일의 위치는 R_PROFILE 이라는 환경변수의 값에 의해 정해집니다. 만약, 아무런 값이 정해지지 않았다면 R의 홈 디렉토리의 서브들 중에서 etc라는 디렉토리 내에 있는 Rprofile.site이라는 파일이 대신 사용되어 집니다. 두번째로 사용자의 워크스페이스 관리, 프로그램 사용도중 워킹 디렉토리 변경, 그리고 프로그램 시작시 워킹디렉토리의 설정에 대한 내용은 R_PROFILE_USER라는 환경변수의 값에 정해지는데, 만약 이러한 환경변수의 값이 정해지지 않았다면, .Rprofile이라는 파일을 찾게 됩니다. 주로 이 사용자에 대한 프로파일은 R 스타트업 디렉토리에 존재하지만 실제로 R홈 디렉토리내의 어떠한 서브에 위치해도 관계는 없습니다 8. 만약, .Rprofile이 스타트업 디렉토리에서 발견되지 않는다는 R은 사용자의 홈디렉토리에서 .Rprofile 이라는 것을 찾을 것입니다 (만약 존재할 경우에만 해당합니다).

사이트 프로파일, 사용자 프로파일, 그리고 .RData라는 이미지 내부에 .First()라고 이름이 붙은 함수들은 특정한 값을 가지고 있습니다. 이것들은 R세션이 시작할 때 자동으로 작업환경을 초기화할 때 사용되게 됩니다. 아래의 코드를 살펴 보신다면 더 빠른 이해를 가지실 수 있습니다. 예를들어, 아래에 있는 .First 라는 함수의 정의중 prompt 라는 옵션을 $ 에서 다른 심볼로 바꾸어 준다면, 여러분은 R을 사용하는 세션내내 사용자가 변경한 심볼을 이용하여 prompt를 사용하게 됩니다.

일반적으로 사용자의 환경설정을 위한 파일의 실행순서는 Rprofiles.site, 그리고 사용자 프로파일, .RData, 그리고 .First() 입니다.

```
> .First <- function() {
  options(prompt="$ ", continue="+\t")  # $ is the prompt
  options(digits=5, length=999)         # custom numbers and printout
  x11()                                 # for graphics
  par(pch = "+")                        # plotting character
  source(file.path(Sys.getenv("HOME"), "R", "mystuff.R"))
                                        # my personal functions
  library(MASS)                         # attach a package
}
```

위의 .First()와 유사하게 .Last()라는 함수가 정의된다면 아래와 같습니다. 그리고 이것은 보통 세션이 종료될때 사용될 것입니다.

```
> .Last <- function() {
  graphics.off()                        # a small safety measure.
  cat(paste(date(),"\nAdios\n"))        # Is it time for lunch?
}
```

Classes, generic functions and object orientation

객체에 대한 클래스라는 것은 generic (제네릭)함수가 어떻게 그 객체를 처리할 것인지 대한 정보를 제공합니다. 즉, 제네릭 함수라는 것이 객체에 대한 클래스정보를 인자로 가지기 때문에 만약 클래스 정보가 없거나, 혹은 제네릭 함수내에서 해당 클래스에 대한 처리방법이 정의되지 않았다면 default action (한국말 모름)이라는 것을 수행하게 됩니다.

이러한 클래스 메커니즘은 사용자가 제네릭 함수를 디자인하고 새로 정의할 수 있도록 도와줍니다. 단편적인 예로 plot() 이라는 제네릭 함수는 여러가지 객체의 종류에 따른 다양한 시각화 방법을 제공하고, summary()라는 함수는 다양한 종류의 객체에 대한 통계분석의 결과를 요약해 줍니다. 또한 anova() 라는 제네릭 함수는 통계 모델을 비교할 때 사용됩니다.

클래스를 특정한 방법으로 처리하는 제네릭 함수의 종류는 매우 많습니다. 예를 들어, 데이터프레임이라는 클래스의 객체를 처리하고자 한다면 아래와 같은 처리방법들에 대해서 포함하고 있을 것입니다.

```
[     [[<-    any     as.matrix
[<-   mean    plot    summary
```

따라서 어떤 처리방법들이 제네릭 함수안에 구현되어 있는지를 확인하고 싶으시다면 아래와 같이 methods()라는 함수를 클래스명과 함께 이용하시길 바랍니다.

```
> methods(class="data.frame")
```

이와 반대로, 제네릭 함수가 처리해 낼 수 있는 클래스의 종류 또한 많습니다. 예를 들어, plot()이라는 함수는 기본적으로 default method이라는 처리방식을 가지며, "data.frame", "density", "factor", 그 밖의 클래스들로부터 파생되어진 다양한 종류의 객체들을 처리할 수 있습니다. 만약, 어떠한 종류의 객체들을 처리할 수 있는지 확인해 보고 싶으시다면 아래와 같이 methods()를 함수의 이름과 함께 사용하시면 됩니다.

```
> methods(plot)
```

일반적으로 제네릭 함수의 본체는 아래에서 보는 것과 같이 매우 짧습니다. 여기에서 UseMethod라는 메시지는 coef라는 함수가 제네릭 함수임을 의미하는 것입니다.

```
> coef
function (object, ...)
UseMethod("coef")
```

따라서 coef라는 제네릭 함수에 대한 처리방식에 대한 내용을 확인하고 싶으시다면 methods()를 사용하시면 됩니다.

```
> methods(coef)
```

```
[1] coef.aov*         coef.Arima*       coef.default*     coef.listof*
[5] coef.nls*         coef.summary.nls*

   Non-visible functions are asterisked
```

위의 예제는 6가지의 처리방식이 있으나, 위에서 보는 바와 같이 메소드들이 어떻게 작성되었는지에 대해서는 보여지지 않는다고 *를 이용하여 표기하고 있습니다. 그러나, 우리는 이러한 메소드들에 대한 알고리즘의 구현방법을 아래와 같이 확인할 수 있는 함수를 제공합니다.

```
> getAnywhere("coef.aov")
A single object matching 'coef.aov' was found
It was found in the following places
  registered S3 method for coef from namespace stats
  namespace:stats
with value

function (object, ...)
{
    z <- object$coef
    z[!is.na(z)]
}

> getS3method("coef", "aov")
function (object, ...)
{
    z <- object$coef
    z[!is.na(z)]
}
```

이 섹션의 내용을 더 정확히 이해하기 위해서는 우리는 사용자가 R Language Definition라는 문서를 참고하길 바랍니다.

Statistical models in R

이번 세션에서 다루어지는 내용은 여러분이 회귀분석 (regression analysis) 혹은 분산분석 (analysis of variance)를 어느 정도 알고 있다고 가정한 상태에서 작성되었습니다. 그리고 이 섹션의 후반부에서는 여러분들이 일반 선형화 모델 (generalised linear model)과 비선형 모델 (nonlinear model) 을 이미 알고 있다고 전제합니다.

R은 여러분들이 일반적인 통계모형을 개발하는데 도움을 주고자 다양한 기능을 제공하고 있습니다. 이 문서의 처음부분에서 언급한 것과 같이 R은 최소한의 결과물만을 제공하기 때문에, 분석에 대한 더욱 상세한 내용이 필요한 경우에는 분석결과의 출력에 관련된 출력함수 (extractor functions)들을 이용하여 확인하셔야 합니다.

Defining statistical models; formulae

선형 회귀 모델 (linear regression model)은 다음과 같은 일반적인 형식을 가집니다.

```
y_i = sum_{j=0}^p beta_j x_{ij} + e_i,     i = 1, ..., n,
```

이때 e_i란 오차항 (error term) 을 의미하며, 오차항은 평균이 0이고 분산이 sigma^2인 정규분포를 따르되 각 관측치는 독립 (independent) 이며 동질적 (homoscedastic) 입니다. 그리고 이것을 오차항 (e_i) 은 NID (0, sigma^2)을 따른다고 씁니다. 독립이란 말의 의미는 i번째 관측치는 i번째 관측치가 아닌 다른 관측치들

과 관계가 없으며, 동질적이란 말의 의미는 모든 관측치가 같은 분포로부터 얻어졌다는 것입니다.

위에서 표현된 회귀분석의 일반식은 개개의 관측치를 기준으로 쓰여졌다면, 아래에 보이는 수식은 모든 관측치들에 대해여 행렬의 형식으로 표현한 것입니다. 따라서 y는 모든 관측치를 한개의 열벡터의 형식을 가진 종속변수 (response variable 혹은 outcome variable) 이라고 하고, X은 회귀분석의 모델을 정의하는 행렬로서 모델매트릭스 (model matrix) 혹은 디자인매트릭스 (design matrix)라고 합니다. 이 디자인 매트릭스는 그 사이즈가 n행 p열로 정의되는데, 각 열은 p개의 설명변수(explanatory variable)로서 x_0, x_1, ..., x_p 표시합니다. 일반적으로 x_0 은 intercept term (한국말 모름)으로서 x_0라는 열 벡터의 모든 구성요소가 1로 이루어진 것으로 정의합니다.

```
y = X  beta + e
```

Examples

여기에서 다루는 내용은 사용자가 R을 이용하여 통계분석의 수행시 필요한 정보를 담고 있습니다.

먼저 y, x, x0, x1, ... 가 모두 수치형 열벡터라고 가정합니다. 또한, X 는 행렬이고, A, B, C, ... 는 요인 들(factors)을 나타낸다고 가정합니다.

그럼 이제부터 선형 회귀 모형 (linear equation 혹은 model formula)을 정의해봅니다. (역자는 model formula라는 영문을 그대로 사용할 것입니다. 그 이유는 model formula가 R에서 내부적으로 통일된 용어이기 때문입니다).

y ~ x	두가지 표현 모두 종속변수 y를 x 라는 한개의 설명변수를 이용하여 설명하고자 하는 단순선형회귀분석입니다. 가장 기본적인 수학적 모델이 인터셉트를 포함하므로 R은 기본적으로 분석을 수행할 때 인터셉트를 포함한다고 가정합니다.
y ~ 1 + x	세가지 표현 모두 단순선형회귀 모형을 나타내지만 인터셉트항을 가지고 있지 않을 경우에 사용합니다.
y ~ 0 + x	
y ~ -1 + x	
y ~ x - 1	
log(y) ~ x1 + x2	log(y), 에 의해 변환된 종속변수의 값을 2개의 설명변수 x1 과 x2를 이용하여 설명하고자 하는 multiple regression (정확한 한국말 모름)입니다.
y ~ poly(x,2)	이것은 종속변수 y를 설명변수 x의 다항식 (polynomial) 으로서 설명하고자 하는 polynomial regression (정확한 한국말 모름)입니다. 첫번째 표현은 직교다항모형 (orthogonal polynomials)를 의미하며, 두번째 표현은 x^0, x^1, x^2가 직접적인 기저 (즉, basis)로 하는 다항모형을 의미합니다.
y ~ 1 + x + I(x^2)	
	이 표현식은 요인 A를 이용하여 y에 대한 1차 분산분석 (one-way analysis of variance)을 의미합니다. 여기에서 A이라는 요인은 여러개의 레벨로 구성되어 있음을 기억하셔야 합니다.
y ~ A	model formula 에 사용되는 ~ (틸다) 라는 연산자는 통계학적 의미로 연관시킨다는 의미를 가지고 있습니다. 그 이유는 수학과 같이 정확한 함수적 관계가 있지 않기 때문입니다. 따라서, R 에서 사용하는 일반 선형모형은 다음과 같은 model formula 의 형식을 가집니다.

```
response ~ op_1 term_1 op_2 term_2 op_3 term_3 ...
```

각각의 항들에 대한 설명은 다음과 같습니다.

response는 종속변수가 1개로 이루어진 열벡터가 될 수도 있고, 혹은 여러개의 종속변수들로 이루어진 행렬이 될 수도 있습니다. op_i이라는 것은 + 혹은 -이라는 연산자를 의미합니다. 연산자 +는 설명변수 term_i를 회귀분석에 포함시킨다느느 의미이고 -는 포함시키지 않는다는 의미입니다. term_i라는 것은 다음의 항목들중에 한가지가 될 수 있습니다. 벡터 혹은 행렬, 또는 1 요인 (factor) 요인, 벡터, 행렬등을 +와 -라는 연산자를 이용하여 복합적으로 표현한 항 위의 모든 경우들을 행렬식으로 나타낼 경우, 각각의 term_i은 디자인 매트릭스에 의해서 선형회귀분석에 포함될 것인지 말 것인지가 결정되어 집니다. 1 이라는 열벡터는 인터셉트를 의미하는데, 이 열벡터는 특별히 지정하지 않아도 항상 디자인 매트릭스에 포함되어 있다고 생각하셔야 합니다.

formula operators는 GLIM 혹은 Genstat 에서 사용되는 Wilkinson과 Rogers라는 연산자의 개념과 유사합니다. 그러나 다른점이 있다면 GLIM이나 Genstat에서는 '.'이라는 연산자를 사용하지만 이것은 R에서는 ':'이라는 심볼을 사용합니다. 그 이유는 '.'심볼은 R에서 변수이름을 사용하는 하나의 문자로 사용되기 때문입니다.

다음은 Chambers & Hastie (1991, p.29)의 내용을 기초로 model formula내에서 사용되는 formula operators들에 대해서 정리한 것입니다.

Y ~ M	~ (틸다)는 Y를 M에서 정의된 모델에 관계시킨다는 의미입니다. 여기에서 M 이라는 것은 model formula에서 정의한 모델의 약자로 기억하시는 것이 편리합니다.
M_1 + M_2	+ (플러스)는 model formula 작성시, 설명 변수 M_1과 또 다른 설명변수 M_2를 동시에 포함하겠다는 의미입니다.
M_1 - M_2	- (마이너스)는 model formula 정의시, 설명변수 M_1은 포함하되 M_2는 포함시키지 않는다는 의미입니다.
M_1 : M_2	: (세미콜론)은 model formula 정의시 , M_1과 M_2의 tensor product에 해당하는 모든 값을 활용한다는 의미입니다. 만약, M_1과 M_2가 요인이라면, tensor product의 결과는 M_1과 M_2가 가지는 모든 레벨들의 조합으로 만들어진 모든 서브클래스들입니다.
M_1 %in% M_2	이것은 위의 tensor product 와 동일한 표현입니다.
M_1 * M_2	
M_1 + M_2 + M_1:M_2.	
M_1 / M_2	
M_1 + M_2 %in% M_1.	
M^n	All terms in M together with "interactions" up to order n

I(M)	I() 라는 연산자는 insulate (즉, 따로 보관함)의 약자로서 model formula 정의시, 괄호안에 작성되는 표현식의 결과를 하나의 변수로서 사용한다는 의미입니다. 일반적으로 R은 괄호안에서 사용되는 표현식에 포함된 연산자들은 그 수학적 의미를 그대로 가질 수 있도록 하였습니다. 따라서, model formula 정의시 사용되는 + 혹은 - 라는 연산자는 model operators로서 설명변수를 model formula 에서 포함할 것인지 말것인지를 결정하는데 사용되지만, model formula를 구성하는 한개의 항에 사용된 I(M_1+M_2)이라는 것은 변수 M_1과 변수 M_2를 활용한 그 수학적 결과를 하나의 변수로서 model formula에 이용하겠다는 의미입니다. model formula는 디자인 매트릭의 열을 지정하는 것과 동일하며, 각각의 열에 부응하는 모델파라미터들이 존재한다고 가정해야 합니다. 그리고 우리는 이 모델파라미터들을 구하는데 R을 사용하는 것입니다. 그러나 비선형 모델에서는 이러한 규칙들이 동일하게 적용되지 않으므로 주의해야 합니다.

Contrasts

우리는 model formula가 포함하는 항들이 디자인 매트릭스가 가지고 있는 열과 어떠한 관계를 가지고 있는지에 대한 최소한의 지식을 가지고 있어야만 합니다. 만약, 연속형 데이터를 가지는 변수들만 있을 경우 이 관계에 대해서 생각하는것은 단순히 개개의 항들이 각각의 열들에 상응하기 때문에 어렵지 않습니다.

그러나, k개의 레벨로 구성된 A라는 요인을 모델하고자 할 경우에는 model formula와 디자인 매트릭스와의 관계는 순서형 요인인지 아닌지에 따라서 크게 달라집니다. 만약 순서형 요인이 아닌 경우 (unordered factor), 디자인 매트릭스의 각 열은 요인의 레벨에 해당하게 됩니다. 따라서 두번째 열부터 마지막 k번째 열까지는 요인의 레벨을 나타내는 지시자 (indicator)로서 모델을 표현하게 됩니다 (만약 이 표현이 익숙하지 않다면 더미변수의 생성으로 생각하시길 바랍니다). 이러한 관계는 통계분석의 결과에 대해서 각 레벨에 해당하는 통계치는 첫번째 레벨에 해당하는 통계치에 대해서 상대적이란 해석을 하도록 만듭니다. 이와 다르게, 순서형 요인 (ordered factor)인 경우에는 k-1개의 열들은 상수항을 제외한 1, ...,k에 해당하는 orthogonal polynomials (정확한 한국말 모름)을 의미하게 됩니다.

그러나 우리는 model formula와 디자인매트릭스의 관계에 대해서 모두 다 다룬것은 아닙니다. 먼저 요인을 포함하고 있지만 인터셉트가 없는 모델을 생각해 볼 수가 있습니다. 이 경우 첫번째 항은 k개의 열들이 요인의 레벨을 나타낼수 있도록 변형되어져야 합니다. 또 다른 경우는 주어진 모델의 전체적인 선형변환과 관계되는 contrasts (한국말모름)을 수행하는 방식이 R과 S-Plus가 서로 다르다는 점입니다. R은 기본적으로 아래와 같이 보이는대로 contr.treatment이라는 contrast 방식을 사용합니다.

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

이에 반해 S는 아래에서 보이는 것과 같이 contr.helmert라는 contrast방식을 사용하기 때문에, 만약 순서형이 아닌 요인을 가진 모델이 있다면 R과 S-Plus는 서로 다른 값을 줍니다. 사용자가 S-Plus를 기초로 하여 쓰여진 책이나 논문을 사용하여 R에서 확인해 보고자 한다면 contrast에 대한 옵션을 아래와 같이 contr.helmert를 선택해주어야 합니다.

```
options(contrasts = c("contr.helmert", "contr.poly"))
```

이렇게 R이 S-Plus와 다른 contrast방식을 사용하는 이유는 R을 처음 접하는 사용자들에 해석상의 편의를 제공하기 위해서 입니다. 그러나, 디자인 매트릭스의 선형변환을 수행하는 contrast라는 함수와 C라는 함수를 아직 우리는 다루지 않았습니다. 또한 우리는 열들의 곱으로서 표현되는 교차항 (interaction terms)에 대해서도 다루지 않았습니다. 이러한 세부사항들은 너무 복잡하므로 여기에서 다루기는 적합하지 않지만, R은 통계전문가들이 요구하는 복잡한 모형을 모델할 수 있도록 많은 기능적요소들을 제공합니다.

Linear models

보통 여러개의 설명변수를 가지는 선형모형을 적합하기 위한 가장 간편한 방법은 아래와 같이 lm()이라는 함수를 이용하는 것입니다.

```
> fitted.model <- lm(formula, data = data.frame)
```

예를 들면, 아래의 코드는 종속변수 y를 설명변수 x1과 x2로 설명하고자 하는 multiple regression model (정확한 한국말 모름, 다항회귀?)입니다. 이때 인터셉트항이 포함되어 있다는 점에 대해서 명심하세요. 또한 lm() 함수에 사용되는 인자들 중 data=production이라는 인자는 데이터셋을 의미하는데, 이 데이터셋 인자는 attach()에 의해 변수탐색경로와 연결여부를 떠나서 반드시 입력되어야 합니다. 더 나아가서, 이 production이라는 데이터셋은 선형모형을 모델함에 있어서 필요한 모든 변수들을 포함하고 있어야 하고 반드시 데이터프레임의 형식을 가지고 있어야 합니다.

```
> fm2 <- lm(y ~ x1 + x2, data = production)
```

Generic functions for extracting model information

함수 lm()의 사용은 적합된 선형모형의 결과를 돌려줍니다. 기술적으로 정확히 말한다면 "lm"이라는 클래스의 결과물들이 한데 묶인 리스트의 형식으로 반환됩니다. 이때, "lm"이라는 클래스의 결과물이란 적합된 선형모형의 결과를 저장하는 객체들입니다. 이렇게 lm()함수로부터 반환된 리스트는 출력에 관계된 제네릭 함수들을 이용하여 적합된 선형모형의 결과를 출력하고 시각화하는데 이용됩니다.

```
add1    deviance   formula     predict  step
alias   drop1      kappa       print    summary
anova   effects    labels      proj     vcov
coef    family     plot        residuals
```

다음은 가장 일반적으로 많이 사용되는 제네릭 함수들에 대한 설명입니다.

anova(object_1, object_2)	이것은 두개의 적합된 모델을 비교하는데 사용되며, 분산분석표를 제공합니다. 조금 더 자세히 설명하면, model formula 1을 이용하여 lm()을 수행하여 얻은 객체를 object_1 이라고 하고, model formula 2를 이용하여 얻은 객체 object_2라고 합시다. 이때 model formula 1 과 2는 반드시 동일한 데이터에 적용되어야 합니다. anova(object_1)이라고 하면 model formula 1을 이용하여 적합된 선형모형의 분산분석표를 보여줍니다. 그리고 anova(object_1, object_2)라고 하면 두 모델의 비교에 대한 분석분석표를 제공합니다.
coef(object)	이것은 선형모형에 대한 선형계수를 행렬의 폼으로 출력시켜줍니다. 여기에서 coef()는 coefficients()라는 함수의 약어로 표현된 함수명일뿐입니다. 따라서 coefficients(object)이 원래의 사용법입니다.

deviance(object)	이것은 deviance (한국말모름)통계량을 출력시켜줍니다. 데비언스란 제곱된 잔차(residual = 관측값 - 적합된 값)들의 합을 의미합니다.
formula(object)	이것은 적합에 사용된 선형모형 model formula를 출력합니다.
plot(object)	이것은 잔차, 적합된 값들, 그리고 모델진단을 위한 4가지 종류의 그래프를 생성해 줍니다.
predict(object, newdata=data.frame)	새로운 데이터를 적합된 모델에 적용하여 예측값을 구하며, 그 결과는 벡터 혹은 행렬이 될 수 있습니다. 이때 적용되는 새로운 데이터는 반드시 적합된 모델에 이용되었던 데이터와 같은 변수명들을 가지고 있어야 합니다.
print(object)	이것은 객체를 출력하는 함수인데, 실제로 다른 함수들 속에 정의되어 출력기능을 수행합니다.
residuals(object)	이것은 적합된 모형의 잔차를 출력해줍니다. residual()이라는 함수명이 길다고 생각되시면 단순히 residual()의 약어로 표현된 함수인 resid(object)을 사용하시면 됩니다.
step(object)	이것은 stepwise search (한국말모름)이라는 방법으로서 AIC (Akaike's Information Criterion)의 값을 근거로 하여 가장 데이터에 적합한 모형을 찾아주는 함수입니다. 이 방법은 자동으로 모형에 새로운 변수를 추가하거나 제거하는 과정을 반복하므로서 가장 작은 값의 AIC를 가지는 모형을 알려줍니다.
summary(object)	회귀분석의 결과를 간결한 형태로 요약하여 보여줍니다.
vcov(object)	이것은 적합된 회귀계수의 분산-공분산 (variance-covariance) 행렬을 출력해줍니다.

Analysis of variance and model comparison

lm()이라는 함수의 사용방법과 그 작동원리가 매우 비슷하나, 분산분석이라는 통계적 방법을 이용하여 데이터를 적합하는데 가장 손쉽게 사용되는 aov(formula, data=data.frame)이라는 함수가 있습니다. 이 함수로부터 반환되는 리스트형식의 결과는 Generic functions for extracting model information에서 설명한 제네릭 함수들을 그대로 적용하여 출력해 낼 수 있습니다. 그러나, aov()라는 함수에 대해서 반드시 강조해야할 장점이 있습니다. 그것은 split plot experiment (한국말 모름) 혹은 블럭간의 정보를 추가적으로 활용해야 하는 balanced incomplete block design (한국말모름)와 같이 에러항이 다중구조를 가지는 통계적 방법을 손쉽게 구현할 수 있다는 것입니다. 이것은 multiple stratum experiment (한국말 모름)와 같은 통계적 방법에서 error strata (한국말 모름) 때문에 생기는 에러항에 대한 복합적 다중구조를 함수 aov() 내부

에서 함께 쓰이는 함수 Error()의 strata.formula라는 인자를 통하여 쉽게 정의할 수 있기 때문입니다. 가장 간단한 예로, strata.formula는 요인의 수준간 (between)와 수준내 (within) 분석이 허용하는 two strata experiment (한국말 모름)을 정의할 수 있습니다.

```
response ~ mean.formula + Error(strata.formula)
```

예를들어, 아래와 같은 모형은 v + n*p*k이라는 mean model (정확한 한국말 모름)과 Error(farms/blocks)로 표현되는 오차항의 합으로 표현되는 실험계획임을 알 수 있는데, 여기에서 사용된 함수 Error()는 오차항에 대하여 “between farms”, “within farms”, 그리고 “within block”이라는 다중구조를 정의해 줍니다.

```
> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks), data=farm.data)
```

ANOVA tables

aov()란 함수는 여러개의 적합된 모델에 대한 분산분석표를 제공해 줍니다. 일반적으로 항을 하나씩 추가함으로서 얻어지는 여러개의 적합된 모델의 Sum of Squares (SS)은 잔차제곱의 합이 감소함을 보여줍니다. 특히, orthogonal experiment (한국말 모름)의 경우에는 항을 추가하는 순서가 중요하지 않습니다. Multistratum experiments (한국말 모름)의 경우 역시 종속변수의 값을 오차항이 가지는 계층에 순차적으로 mean model 을 영사시켜 나감으로서 분산분석이 가능해집니다. 이에 대한 자세한 사항은 Chambers & Hastie (1992)의 책을 참고하시길 바랍니다. 이러한 통계분석이 요구될 경우 한개의 모델에 대한 적합결과를 완전한 분산분석표를 통하여 보는 것보다 2개 혹은 그 이상의 모델을 아래와 같이 anova()함수를 이용하여 종합적으로 한번에 비교해 본다면 매우 편리할 것입니다. 그러나 이러한 방법을 취하는 것은 단순히 모형에 대한 이해를 구조화함으로서 적합한 모델을 손쉽게 찾기 위한 방법이외의 다른 목적이 없다는 것을 사용자는 반드시 인지하고 있어야 합니다.

```
> anova(fitted.model.1, fitted.model.2, ...)
```

Updating fitted models

아래에 예제에서 사용된 update()이라는 함수는 이전에 적합된 모형에 몇 개의 항을 추가 혹은 제거를 통하여 새로운 모형으로서 적합시키고자 할 때 매우 유용한 함수입니다.

```
> new.model <- update(old.model, new.formula)
```

앞서 언급했던 것과 같이 R은 변수명에 ‘.’을 사용하는 것을 허용하므로 위의 예제와 같이 new.model이라는 변수명을 사용함에 아무런 문제가 없습니다. 그러나, 함수 update()를 사용함에 있어서 ‘.’를 사용할 때는 이전 적합시 사용한 모델의 동일한 부분을 다시 사용함을 R시스템이 내부적으로 인식하도록 하는 역할을 합니다. 좀 더 확실한 이해를 이해서 아래의 예제를 살펴봅니다.

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = production)
> fm6  <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

먼저, production이라는 데이터가 가지고 있는 5개의 설명변수를 가지고 y라는 종속변수를 설명하고자 적

합된 모델의 객체가 fm05에 저장되었습니다. 이렇게 이미 적합에 사용된 모형에 새로운 6번째 설명변수를 추가하여 모형을 재적합하고 싶을 때, '.'를 사용합니다. 이때 data=production이라는 인자를 다시 입력할 필요는 없습니다. 그 이유는 update()이라는 함수가 이미 production이라는 데이터를 재적합을 위해서 홀딩하고 있기 때문입니다. 위의 코드의 맨 마지막라인은 square root라는 변환된 종속변수 y의 값을 6개의 설명변수로 설명하고자 하는 모델을 적합한다는 의미입니다.

그러나, '.'는 아래의 코드에서 보여지는 것과 같이 다른 용도로 사용될 수 있기 때문에 사용자에게 '.'를 사용함에 있어서 주의가 요구됩니다. 이것은 '.'가 종속변수 y를 설명함에 있어서 production이라는 데이터안에 있는 모든 변수를 다 이용하여 모형을 적합하겠다는 의미입니다.

```
> fmfull <- lm(y ~ . , data = production)
```

순차적으로 여러개의 모형을 살펴보기 위한 함수로서 add1(), drop1(), 그리고 step()이라는 함수들이 있는데, 이들은 이름 그대로의 의미를 가집니다. 더 자세한 사항은 온라인 도움말을 살펴보시길 바랍니다.

Generalized linear models

일반화선형모델 (generalized linear model) 이란 정규분포를 따르지 않는 종속변수 (non-normal response distributions)에 변수변환 (transformation)이라는 과정을 통하여 모형의 선형성 (linearity)를 찾는 방법으로 선형모델 (linear model)의 일반화라고 생각하시면 됩니다 여기에서부터 역자는 일반선형화모델을 GLM 이라는 약자를 사용하도록 하겠습니다. GLM은 아래와 같은 일련의 내용을 가정합니다.

먼저 우리가 분석하고자 하는 y라는 반응 혹은 종속변수 (response)가 있고, 이 반응변수의 분포 (distribution of response variable)에 영향을 미치는 여러개의 설명변수 (stimulus variables) x_1, x_2, ..., 이 있다고 가정합니다. 또한 우리는 설명변수들이 y라는 반응변수의 분포에 영향을 미칠때 ,이 영향이 반드시 변수간의 선형관계 (a single linear function, only)를 통하여 영향을 준다고 가정합니다. 따라서, 이 영향을 linear predictor (한국말 모름)이라고 말하고 아래와 같이 표현합니다.

```
      eta = beta_1 x_1 + beta_2 x_2 + ... + beta_p x_p,
```

그러므로, 만약 i번째의 설명변수 x_i의 선형계수 beta_i 가 0 이라면 반응변수의 분포에 어떠한 영향을 미치지 않는다는 점을 알 수 있습니다.

GLM을 사용하기 위해서는 우리는 또한 반응변수 y의 분포가 아래와 같이 표현된다고 가정을 합니다.

```
      f_Y(y; mu, phi)
        = exp((A/phi) * (y lambda(mu) - gamma(lambda(mu))) + tau(y, phi))
```

위에 표현된식에서 phi 란 scale parameter (스케일 파라미터)이고, 이 값은 아마도 사전에 알고 있을 수 있거나 미리 주어짐으로서 상수로서의 역할을 하게 됩니다. 또한 이 스케일 파라미터는 모든 관측치에 대해서 동일하게 적용된다고 가정합니다. A라는 것은 prior weight (사전 가중치)를 나타내는데, 이 사전 가중치는 관측치마다 모두 다를 수도 있다고 가정합니다. 그리고, $₩mu$라는 것은 y의 평균값입니다. 따라서 y의 분포는 y의 평균과 스케일파라미터로서 결정된다고 가정할 수가 있게 됩니다.

마지막으로 우리는 아래에서 표현한 것과 같이 반응변수 y의 평균과 linear predictor가 역함수 관계가 성립한다고 가정을 합니다. 그리고 이 역함수를 link function(링크함수)라고 부르며, 아래의 표현식에서 이 링크함수를 ell()로 표현하였습니다.

```
mu = m(eta),    eta = m^{-1}(mu) = ell(mu)
```

위에서 언급된 가정들은 매우 많아 보이지만, 실제로 이것들은 통계실무에서 사용되는 광범위한 범위의 모델을 모두 포괄할 수 있는 최소한의 가정입니다. GLM 은 이러한 최소한의 가정들을 가지고 추정(estimation)과 추론(inference)을 위한 하나의 통일된 방법을 개발할 수 있도록 도와줍니다. 만약, GLM 에 관심이 있는 독자라면 McCullagh & Nelder (1989) 과 Dobson (1990)을 찾아보시길 바랍니다. 이 두권의 책은 GLM의 거의 모든것에 대하여 매우 상세히 설명하고 있습니다.

Families

R은 GLM에서 사용되는 종속변수 y의 분포가 gaussian, binomial, poisson, inverse gaussian, 그리고 gamma인 경우를 모두 다룰 수 있는 기능을 제공하고 있으며, 종속변수의 분포를 함수식으로 정확히 쓸 수 없는 경우에 적용되는 quasi-likelihood (쿼시-라이클리후드) 모델 역시 지원됩니다. 일반적으로 GLM을 적합하기 위해서는 variance function (한국말 모름)을 알고 있어야 합니다. 쿼시-라이클리후드 모델이 아닌 경우에는 종속변수의 분포로부터 variance function를 알 수 있으나, 쿼시-라이클리후드 모델의 경우에는 variance function가 반드시 종속변수의 평균에 관한 함수로서 표현되어져야만 합니다.

종속변수에 대한 각각의 분포들은 아래에 나열된 다양한 링크함수를 통하여 linear predictor와 함수적 관계를 가지게 됩니다.

Family name	Link functions
binomial	logit, probit, log, cloglog
gaussian	identity, log, inverse
Gamma	identity, inverse, log
inverse.gaussian	1/mu^2, identity, inverse, log
poisson	identity, log, sqrt
quasi	logit, probit, cloglog, identity, inverse, log, 1/mu^2, sqrt

위에서 보여지는 표의 내용과 같이 종속변수의 분포, 링크함수, 그리고 모형에 필요한 다른 정보들과 조합은 GLM에서 family라는 클래스를 결정하게 됩니다.

The glm() function

GLM은 종속변수의 분포를 설명변수의 조합을 오로지 선형관계로 설명하므로, 선형회귀분석에서 모델을 적합하는데 사용된 방식이 GLM에서도 선형관계를 설명하는데 이용될 수 있습니다. R은 GLM 을 수행하기 위해서 아래와 같은 형식을 가지는 glm()이라는 함수를 제공하고 있습니다. 함수의 이용방법은 family라는 클래스를 지정하는 것을 제외하고는 lm()의 사용방식과 동일합니다.

```
> fitted.model <- glm(formula, family=family.generator, data=data.frame)
```

여기에서 family를 지정하기 위해서 family.generator라는 인자가 사용되는데, 이것은 GLM이 모형을 정의하고 추정하는 그 모든 과정을 수행하기 위해 필요한 제네릭함수들을 불러오기 위해서 사용됩니다. 이는 매우 복잡한 것같이 생각되지만, 실제로 family의 이름만을 결정하면 되기 때문에 매우 GLM을 매우 간편하게 수행할 수 있도록 해줍니다. 여기에서 family.generator가 가질 수 있는 이름은 Families이라는 섹션에서 우리가 family에 대해서 언급할 때 사용한 표에 "Family Name"이라는 열에 나열되어 있습니다. 또한 이 표는 각 family에서 사용이 가능한 링크함수들에 대해서도 나열하고 있습니다. glm()함수를 사용시, 특정 링크 함수의 사용은 family.generator의 이름뒤에 괄호와 함께 링크함수의 이름을 지정해 줌으로 사용이

가능합니다. quasi family의 경우, variance function 역시 같은 방법으로 지정하게 됩니다.

아래의 예제는 여러분이 glm()함수를 이용하여 GLM 을 수행하는 방법을 보다 쉽게 이해할 수 있도록 도와줄 것입니다.

The gaussian family

아래에서 보이는 것과 같이 glm()함수를 이용하여 gaussian (가우시안) family에 해당하는 문제를 해결하는 것은 lm()을 이용하여 문제를 해결하는 것과 수치적으로 아무런 차이는 없으나 비효율적입니다. 그 이유는 family.generator에 어떤 특정한 링크함수를 사용할 것인가를 알려주지 않았기 때문입니다. 또한, 사용자가 가우시안 페밀리에서 nonstandard (비표준) 링크함수를 이용하여 문제를 풀고자 할 경우, 이는 quasi 페밀리를 통하여 해결해야 합니다. 이 방법은 qausi 페밀리를 소개할때 설명하도록 하겠습니다.

```
> fm <- glm(y ~ x1 + x2, family = gaussian, data = sales)
> fm <- lm(y ~ x1+x2, data=sales)
```

The binomial family

Silvey (1970)에 등장하는 간단한 예를 살펴보도록 하겠습니다.

Kalythos 라는 에게해에 있는 한 섬에는 남성들이 선천성 안질환을 가지고 있습니다. 이 안질환은 주로 나이가 들어감에 따라서 진행되어집니다. 아래의 데이터는 이 섬에 거주하는 다양한 연령대의 남성들의 실명여부를 검사한 결과를 기록한 것입니다.

Age: 20 35 45 55 70 No. tested: 50 50 50 50 50 No. blind: 6 17 26 37 44 우리는 이 데이터에 logistic과 probit이라는 두가지 모델을 적합시키고 각각의 모형으로부터 남성거주자의 실명확률이 50%가 되는 나이 (LD50)를 추정하고자 합니다.

만약 y가 연령 x에서 테스트로부터 확인한 실명한 사람의 수 n을 나타낸다면, 두 모형 모두 y ~ B(n, F(beta_0 + beta_1 x)) 이라는 형식을 가지게 됩니다. 단, probit 모델의 경우 F(z) = Phi(z) 가 표준정규분포를 따르며, R에서 디폴트로 사용되는 logit 모델인 경우에는 F(z) = e^z/(1+e^z) 입니다. 두 경우 모두, LD50은 LD50 = − beta_0/beta_1 가 되며, 이것은 분포함수 F(z)의 값을 0으로 만들어 주는 z에 해당하게 됩니다.

이 문제를 접근하는 가장 첫번째 단계는 모형에서 다룰 데이터를 아래와 같이 데이터프레임의 형식을 갖도록 하는 것입니다.

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n = rep(50,5),
                         y = c(6,17,26,37,44))
```

glm()함수를 이용하여 binomial model (이항모델)을 적합하고자 한다면 아래의 세가지 경우 중 한 가지에 해당될 것입니다.

종속변수가 0과 1만을 데이터값으로 가지는 열벡터인 경우입니다. 종속변수가 첫번째 열은 각 시행에서의 성공회수 그리고 두번째 열은 실패회수의 정보를 가지고 있는 행렬의 형식을 가지고 있을 수도 있습니다. 또한, 종속변수는 요인의 형식을 가지는 경우도 있을 수 있습니다. 이때, 요인은 두개의 레벨을 가지며, 첫번째 레벨은 실패를 의미하는 0의 값을 가지고 두번째 레벨은 성공을 의미하는 1의 값을 가질 것입니다. 여기에서 우리는 두번째의 경우를 다룰 것입니다. 따라서 우리는 아래와 같은 작업을 통하여 데이터프레임에 실패의 정보를 포함하고 있는 열을 추가합니다.

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

모형을 적합하는 방법은 아래와 같습니다.

```
> fmp <- glm(Ymat ~ x, family = binomial(link=probit), data = kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

만약 아무런 링크함수를 지정하지 않는다면 R은 기본적으로 logit 링크라고 가정합니다. 적합된 모형의 결과를 보기 위해서 아래와 같이 실행합니다.

```
> summary(fmp)
> summary(fml)
```

두모형 모두 적합결과가 좋습니다. LD50을 추정하기 위해서 우리는 다음과 같은 간단한 함수를 작성합니다.

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fml)); c(ldp, ldl)
```

이 데이터로부터의 추정치는 각각 43.663 과 43.601 입니다.

Poisson models

Poisson family 의 경우 log를 기본링크함수로 가정합니다. 주로 Poisson family는 Poisson log-linear model (포이송 로그-리니어모델)을 이용하여 빈도데이터 (frequency data)를 적합하는데 사용됩니다. 역자는 추후 문서작성 방향을 고려하여 빈도데이터라는 용어보다는 카운트 데이터라고 사용할 것입니다. 실제로 이런 카운트 데이터는 주로 multinomial distribution을 따르는데, 이것은 매우 중요하고도 방대한 주제이므로 더 이상 이문서에서 다루지는 않습니다. 포이송 페밀리를 이용하는 것은 가우시안페밀리에 해당하지 않는 GLM의 거의 모든 경우에 해당하므로 매우 중요합니다.

과거에는 실무에서 가끔 포이송데이터를 로그 혹은 스퀘어루트를 이용하여 변환된 데이터를 가우시안 분포를 따르는 데이터와 같이 분석을 수행하기도 했었지만, 현재는 아래와 같이 포이송 페밀리를 이용한 GLM을 수행합니다.

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt),
              data = worm.counts)
```

Quasi-likelihood models

GLM의 모든 페밀리들이 종속변수의 분산은 평균에 대한 함수이며, scale parameter는 단순히 multiplier 이라는 공통점을 발견할 수 있을 것입니다. 분산과 평균과의 상관관계는 종속변수분포의 특징이기도 합니다. 예를 들어, 포아송 분포는 평균과 분산이 일치합니다. (즉, Var(y) = mu).

Quasi-likelihood 를 이용한 추정과 추론은 정확한 종속변수의 분포를 찾아낸다는 것보다 다소 링크함수와 variance function이 평균과 어떤 관계를 가지고 있는지와 관련이 있습니다. quasi-likelihood를 이용한 추정은 가우시안 분포를 사용하는 경우와 동일한 방법을 사용하므로, 비표준링크함수 혹은 variance funtion을 이용하여 가우시안 모형의 적합을 가능하게 해줍니다.

예를 들어, 비선형회귀모형인 y = theta_1 z_1 / (z_2 - theta_2) + e 에 대한 적합을 생각해 봅시다. 이모

형은 다시 y = 1 / (beta_1 x_1 + beta_2 x_2) + e 로도 표현이 가능하며, 이 때 x_1 = z_2/z_1, x_2 = −1/z_1, beta_1 = 1/theta_1, and beta_2 = theta_2/theta_1. 입니다.

만약, 사용자가 모형에 대응하는 데이터 프레임을 가지고 있다면, 위에서 정의한 비선형회귀 모형은 아래와 같이 적합될 수 있습니다.

```
> nlfit <- glm(y ~ x1 + x2 - 1,
               family = quasi(link=inverse, variance=constant),
               data = biochem)
```

만약, 더 자세한 설명이 필요하다면 매뉴얼과 도움말을 이용하시길 바랍니다.

Nonlinear least squares and maximum likelihood models

특정한 형식을 가지고 있는 비선형모델은 GLM (glm())에 의하여 적합될 수 있습니다. 그러나, 대부분의 비선형 모델의 경우 적합이란 문제는 수치해석측면의 최적화 문제로서 접근해야만 합니다. R은 이런 비선형 최적화 문제를 위해서 optim()과 nlm()이라는 기능을 제공하고 있습니다. 특히, R 2.2.0 버전부터는 S-Plus에서 사용할 수 있었던 ms()과 nlminb()와 같은 기능의 nlminb()을 지원합니다. 이러한 함수들의 기본 동작원리는 여러가지 모수의 값을 반복적으로 적용하여 적합결여지표 (the index of lack-of-fit)의 값을 최소화하는 모수의 값을 추정하는 것입니다. 그러나, 선형회귀에서 사용되는 모수의 추정과는 달리 비선형 최적화 방법을 통한 모수의 추정은 만족스러운 수준의 추정치에 수렴하지 않을 수도 있습니다. 그 이유는 여기에 사용되는 수치해석법이 모수에 대한 초기값을 지정해야 하고, 추정치에 대한 수렴이라는 과정은 시작점의 선택에 따라서 크게 달라질 수 있기 때문입니다.

Least squares

이렇게 비선형 모형을 적합하는 한가지 방법은 오차 혹은 잔차들의 제곱합 (the sum of the squared errors or residuals)을 최소화시키는 것입니다. 이러한 최소제곱법은 관측된 오차들이 정규분포와 유사할 경우 권장됩니다.

여기에서 우리는 Bates & Watts (1988), page 51 에 수록된 예제를 하나 살펴보도록 하겠습니다 . 데이터가 아래와 같이 주어졌습니다.

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56,
         1.10, 1.10)
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

최소화 시켜야 할 적합기준을 아래와 같이 정의합니다.

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] + x))^2)
```

모형적합을 위해서 우리는 모수에 대한 초기값이 필요합니다. 이러한 초기값을 결정하는 한가지 방법은 데이터를 플랏시켜보아 데이터와 비슷한 유사한 패턴을 생성해주는 분포의 모수를 추측해 보는 것입니다.

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 200 * xfit/(0.1 + xfit)
> lines(spline(xfit, yfit))
```

초기값에 대해서 물론 더 나은 추측값들이 있겠지만 200 과 0.1 이 적당하다고 생각하므로 이들을 이용하

여 모형을 적합시켜 보았습니다.

```
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

적합된 모형으로부터 우리는 out$minimum으로부터 SSE를, out$estimate로부터 최소제곱법을 이용하여 추정된 모수의 값을 확인할 수 있습니다. 추정된 모수의 근사 표준편차는 아래와 같이 계산됩니다.

```
> sqrt(diag(2*out$minimum/(length(y) - 2) * solve(out$hessian)))
```

위의 공식에서 사용된 숫자 2는 모수의 개수를 의미합니다. 모수의 추정치에 대한 95%의 신뢰구간을 구하기 위해서 +/- 1.96 SE 를 계산합니다. 최소제곱법으로 적합된 모형을 이용하여 새로운 플랏을 생성하여 데이터와 비교해봅니다.

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 212.68384222 * xfit/(0.06412146 + xfit)
> lines(spline(xfit, yfit))
```

R시스템에 내장되어 있는 표준패키지인 stats는 최소제곱방식을 이용하여 비선형모형을 적합하는 방대한 양의 함수들을 제공하고 있습니다. 우리는 위에서 Michaelis-Menten방법을 이용하여 모델을 적합하였으나, 다음과 같은 방법으로도 가능합니다.

```
> df <- data.frame(x=x, y=y)
> fit <- nls(y ~ SSmicmen(x, Vm, K), df)
> fit
Nonlinear regression model
  model:  y ~ SSmicmen(x, Vm, K)
   data:  df
        Vm            K
212.68370711   0.06412123
 residual sum-of-squares:  1195.449
> summary(fit)

Formula: y ~ SSmicmen(x, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
Vm 2.127e+02  6.947e+00  30.615 3.24e-11
K  6.412e-02  8.281e-03   7.743 1.57e-05

Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:
      Vm
K 0.7651
```

Maximum likelihood

Maximum likelihood (한국말모름, ML) 이라는 방법 역시 비선형모델을 적합하는데 쓰이며, 오차항이 정규분포를 따르지 않을 경우에도 이용될 수 있습니다. 이 방법의 핵심은 로그-라이클리후드 (log-likelihood) 함수를 최대화 시켜주는 모수를 찾는것이며, 이것은 네가티브 로그-라이클리후드 (negative log-likelihood)를 최소화 시켜주는 것과 동일합니다. 우리는 여기에서 Dobson(1990), pp.108 - 111 에 나오는 예제를 하나 살펴보도록 하겠습니다. 이 예제는 dose-response 데이터에 로지스틱 모형을 적합하는 것이며 glm() 함수를 활용할 수 있습니다. 데이터는 아래와 같습니다.

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113,
```

```
          1.8369, 1.8610, 1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

아래와 같이 정의된 negative log-likelihood 함수를 최소화 하고자 합니다.

```
> fn <- function(p)
   sum( - (y*(p[1]+p[2]*x) - n*log(1+exp(p[1]+p[2]*x))
           + log(choose(n, y)) ))
```

이를 수행하기 위해서 우리는 그럴듯한 초기값 (starting values)를 지정합니다.

```
> out <- nlm(fn, p = c(-50,20), hessian = TRUE)
```

적합된 모형으로부터 최소화 된 negative log-likelihood 값을 얻고자 하면 out$minimum을 이용하시고, 모수의 ML추정치를 확인하고 싶으시면 out$estimate을 통하여 얻을 수 있습니다. 추정치에 대한 근사 SE는 아래와 같이 얻어집니다.

```
> sqrt(diag(solve(out$hessian)))
```

95% 신뢰구간은 추정된 모수값에 +/-1.96 SE 를 계산하시면 됩니다.

Some non-standard models

이번 장을 마무리 하기 위해서 우리는 R이 특별한 회귀모형 및 데이터 분석에 어떠한 기능을 제공하는지 간단히 정리해 보았습니다.

Mixed models. mixed effect model(한국말 모름) 이란 선형 및 비선형 회귀모형 중 회귀계수의 일부가 random effect (한국말 모름)을 가지는 모델입니다. 이러한 선형 및 비선형 mixed models을 적합하기 위해서 사용자는 nlme이라는 패키지에서 제공하는 lme()과 nlme()이라는 함수를 이용하시면 됩니다. Local approximating regressions. loess()이라는 함수는 가중회귀모형을 이용하고자 하는 일부 비모수회귀모형의 적합에 사용됩니다. 이러한 회귀분석은 messy data 에 대한 특정 추세를 파악하거나, 사이즈가 큰 데이터에 대해서 데이터 리덕션 (data reduction)을 하고자 할 때 이용될 수 있습니다. loess()라는 함수는 R이 제공하는 기본패키지인 stats란 패키지에 포함되어 있습니다. Robust regression. R은 데이터에 포함된 이상치(extreme outliers)들의 영향력을 줄여 안정적인 회귀모형을 적합하기 위하여 몇가지 함수를 제공하고 있습니다. 현재로서는 MASS라는 패키지에 포함되어 있는 lqs라는 함수가 다른 함수들에 비하여 이상치들의 영향을 상대적으로 적게 받는 가장 안정적인 모형을 적합해주는 것으로 알려져 있습니다. 또한 MASS패키지는 이상치에 대한 모형적합의 안정성이 lqs보다는 조금 떨어지지만 통계적으로 좀 더 나은 결과를 주는 rlm이라는 함수도 포함하고 있습니다. Additive models. This technique aims to construct a regression function from smooth additive functions of the determining variables, usually one for each determining variable. Functions avas and ace in package acepack and functions bruto and mars in package mda provide some examples of these techniques in user-contributed packages to R. An extension is Generalized Additive Models, implemented in user-contributed packages gam and mgcv. Tree-based models. 일반적인 선형모델방법이 전체적인 데이터의 특징을 설명하고 예측할 수 있는 선형모델을 찾는데 목적이 있다고 한다면, tree-based models (한국말 모름)은 데이터 파티션에 주 목적이 있습니다. 데이터 파티셔닝이란 주어진 데이터를 여러개의 동질적인 그룹들로 나누어 주되, 각 그룹간의 이질성은 최대화 시켜주는 방법을 말합니다. 이것은 데이터를 그룹간의 차이를 가장 잘 구분할 수 있는 모형을 찾아 그 모형대로 데이터를 반복적으로 이분화함으로 이루어집니다. 이러한 데이터 파티셔닝은 종종 다른 데이터 분석법으로부터는 찾을 수 없었던 데이터가 가지고 있는 특징을 발견해 내기도 합니다. 이러한 tree-based

models에 의한 데이터 파티셔닝은 tree()라는 함수를 통하여 이루어집니다. 또한 이 분석법에 의한 결과는 효율적인 시각화를 위하여 plot()과 text()와 같은 제네릭함수와 잘 호환되도록 설계되었습니다.

대개 데이터 파티셔닝은 rpart와 tree같은 user-contributed packages (사용자 기여 패키지)에 의해 수행됩니다.

Graphical procedures

R에서 제공하는 그래픽 기능은 R시스템에서 매우 중요한 비중을 차지하며 많은 역할을 수행하고 있습니다. 이러한 기능을 통하여 통계자료 정리 및 분석을 위한 시각화를 할 수 있을 뿐만아니라, 새로운 종류의 그래프를 생성할 수도 있습니다.

그래픽 기능들은 인터랙티브 (interactive) 혹은 배치 (batch)모드 모두 사용될 수 있으나, 대부분의 경우 인터랙티브한 모드에서 사용하는 것이 더욱 효과적입니다. 그 이유는 R프로그램이 시작될때 인터랙티브 그래픽 (interactive graphic)을 위한 그래픽 윈도우(graphic windows)를 연결시켜주는 디바이스 드라이버 (device driver)가 초기화 되기 때문입니다. 물론 이 그래픽 디바이스 드라이버가 자동초기화 되기는 하지만, 이 기능을 하는 명려어를 알아두면 더욱 편리한 사용을 할 수 있습니다. 이것은 유닉스에서는 X11()이라는 명령어이고, 윈도우에서는 windows()이며, Mac OS X 에서는 quartz()입니다. 이러한 명령어에 의하여 그래픽 디바이스 드라이버가 구동을 시작하게 되면, 플랏과 관계된 명령어들에 의하여 다양한 종류의 자료의 정리 및 분석에 대한 시각화가 가능해집니다.

플랏에 관계된 명령문은 아래와 같이 크게 세가지로 구분됩니다.

High-level (하이레벨) 플랏 함수들은 축, 레이블, 제목 등을 조정하여 새로운 플랏을 생성합니다.
Low-level (로우레벨) 플랏 함수들은 이미 생성된 플랏에 추가로 점, 선, 레이블 등을 추가하여 좀 더 많은 정보를 표현할 수 있도록 도와줍니다. Interactive (인터랙티브) 그래픽 함수들은 이미 생성된 플랏에 마우스와 같은 포인팅 디바이스 (pointing device)를 이용하여 사용자와의 인터랙티브한 과정을 통하여 정보를 추가하거나 혹은 추출합니다. 또한, R은 그래픽 파라미터들을 조작하여 사용자가 원하는대로 플랏을 수정 및 변경하는 기능을 제공하고 있습니다. 그러나, 이 메뉴얼에서는 base패키지 안에 있는 가장 기초적이며 기본적인 그래픽 기능들을 다룰 것입니다. base패키지와는 별개로 grid라는 패키지 안에는 별도의 서브시스템은 좀 더 강력한 그래픽 기능들을 제공하지만 사용에 다소 어려움이 있습니다. 또한, grid패키지에 있는 그래픽 기능들을 활용하여 제작된 lattice라는 패키지는 S의 Trellis (트렐리스)시스템에서 제공하는 멀티패널 플랏과 같은 고급 그래픽처리가 가능한 함수들을 포함하고 있습니다.

High-level plotting commands

하이레벨 플랏 함수들은 함수에 인수로서 받아들여지는 데이터에 대한 완전한 시각화 처리를 위해 고안되었습니다. 완전한 시각화 처리란 사용자가 특별히 따로 지정하지 않아도 적절한 위치에 축, 레이블, 그리고 제목들이 자동으로 생성되어짐을 말합니다. 하이레벨 플랏은 현재 그래픽 디바이스에 출력되어 있는 플랏을 지우고, 항상 새로운 플랏을 생성하는 기능을 수행합니다.

The plot() function

R에서 가장 많이 사용되는 플랏함수는 plot()입니다. 이것은 제네릭 함수로서 생성되는 플랏의 형태는 함수의 첫번째 인자로서 입력받는 객체의 클래스에 의해서 결정되어 집니다.

plot(x, y) plot(xy) 만약, x와 y가 둘다 벡터이면 plot(x, y)라는 명령은 x에 대한 y의 산점도를 생성합니다. 두번째 명령문 역시 동일한 산점도를 출력하지만, xy라는 인자가 입력받는 데이터 형식에 주의해야합니다.

이것은 x와 y라는 변수들이 각각의 컴포넌트로 구성된 하나의 리스트 형을 입력받아야 하거나, 혹은 두개의 열을 가지는 행렬의 형식을 입력받기를 요구함을 의미합니다. plot(x) 만약, x가 시계열 데이터 인경우, 이 명령문은 자동으로 시계열 플랏을 생성합니다. 단, x가 수치형 벡터인 경우, 이것은 x의 인덱스에 대한 x의 값들을 플랏하게 됩니다. 만약, x가 복소수라면 이것은 실수부분에 해당하는 값에 대하여 허수의 부분에 해당하는 값들을 산점도의 형식으로 출력하게 됩니다. plot(f) plot(f, y) 만약, f가 요인이고, y가 수치형 벡터이면, 첫번째 명령어는 f에 대한 바 플랏을 생성하고, 두번째 명령어는 f가 가지는 레벨별로 y에 대한 박스플랏을 생성하게 됩니다. plot(df) plot(~ expr) plot(y ~ expr) 만약, df가 데이터 프레임이고, y은 어떠한 객체가 될 수 도 있으며, expr라는 것은 a+b+c와 같이 +를 이용하여 연결된 하나의 객체들의 집합이라고 가정합시다. 첫번째 명령문은 데이터프레임의 각 변수에 대해서 다른 변수들에 대한 분포를 보여주는 산점도를 출력하고, 두번째 명령문은 expr 내에서 사용된 변수들만을 이용한 분포를 보여줍니다. 마지막 명령문은 y의 분포를 expr에 사용된 변수벼로 플랏시켜줍니다.

Displaying multivariate data

R은 다변량 데이터의 시각화를 위해 두가지 매우 유용한 함수들을 제공합니다. 만약, X가 수치형 행렬 혹은 데이터 프레임이라면, 아래의 명령문은 각각의 모든 열에 대해서 산점도 (scatterplot)을 출력합니다. 이것은 X의 한 열 혹은 변수에 대해서 다른 열들에 대한 플랏을 생성하므로, 한 화면에 총 n(n-1)개의 플랏들이 동일한 형식을 가진 행렬의 형태로 출력되게 됩니다.

```
> pairs(X)
```

만약, 사용자가 세개 혹은 네 개의 변수만을 이용하여 분석을 수행하는 경우 coplot이 더 효과적일 수 있습니다. 예를들어, a와 b가 수치형 벡터이고, c가 수치형 벡터 혹은 a와 b와 같은 길이를 가지는 요인벡터라면 다음의 명령어를 사용해 보시길 바랍니다.

```
> coplot(a ~ b | c)
```

이것은 c가 가지는 각각의 값별로 b에 대한 a의 산점도를 출력시켜줍니다. 만약 c가 요인이라면 이 함수는 c의 각 수준별 a와 b에 대한 산점도를 출력해주는 것입니다. 더 나아가, 이 함수는 c가 수치형 벡터일 경우 c를 구간화하여 각 구간별 a와 b에 대한 산점도를 출력할 수 있으며, 구간에 대한 조정은 given.values=이라는 인자를 조절함함으로서 이루어집니다. coplot()을 사용할 때, co.intervals()이라는 함수를 함께 사용하는 것은 구간을 선택할 때 매우 유용합니다. 또한, 아래와 같이 c와 d와 두개의 변수가 가지는 값들의 조합으로부터, 각 조합의 수준별 a와 b에 대한 산점도 출력 역시 가능합니다.

```
> coplot(a ~ b | c + d)
```

coplot()과 pairs()라는 함수 모두 panel=이라는 인자를 가지고 있습니다. 이 panel=이라는 인자는 각각의 패널에 어떤 종류의 플랏을 생성할 것인지를 결정하며, 그 기본형은 points()라는 산점도를 출력하는데 이용되는 함수입니다. 만약, 사용자가 산점도외의 다른 종류의 플랏을 생성하고자 한다면, x와 y라는 벡터를 이용하는 로우레벨의 그래픽 함수를 이용하면 됩니다. 예를 들어, coplot()에 사용될 있는 플랏은 panel.smooth()란 함수가 있습니다.

원본 주소 "http://r-project.kr/w/index.php?title=R-intro-ko&oldid=2785"
