4. 데이터 조작 (1) : 벡터 기반 처리와 외부 데이터 처리

4.1 아이리스 데이터

본격적으로 데이터 조작을 알아보기에 앞서, 앞으로 데이터 처리 및 기계 학습 기법의 예제로 사용할 아이리스 데이터 셋에 대해 살펴보자. 이 데이터에는 3가지 종(setosa, versicolor, virginica)에 대해 꽃받침과 꽃잎의 길이를 정리한 데이터다. 이 데이터는 R에 내장되어 있고, 이해하기 쉽다.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

> str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1
...
```

iris에는 붓꽃 데이터가 데이터 프레임으로 저장되어 있는 반면, iris3에는 3차원 배열 형태로 저장되어 있다.

이외에도 R에는 다양한 데이터 셋이 준비되어 있다. datasets 패키지에 있는 데이터 셋은 R에 기본적으로 포함된 데이터들이며, 이 데이터의 목록은 library(help=datasets) 명령으로 살펴볼 수 있다.

```
> data(mtcars)
> head(mtcars)
                   mpg cyl disp  hp
Mazda RX4         21.0   6  160 110
Mazda RX4 Wag     21.0   6  160 110
Datsun 710        22.8   4  108  93
Hornet 4 Drive    21.4   6  258 110
Hornet Sportabout 18.7   8  360 175
Valiant           18.1   6  225 105
                  drat    wt  qsec vs
Mazda RX4         3.90 2.620 16.46  0
Mazda RX4 Wag     3.90 2.875 17.02  0
Datsun 710        3.85 2.320 18.61  1
Hornet 4 Drive    3.08 3.215 19.44  1
Hornet Sportabout 3.15 3.440 17.02  0
Valiant           2.76 3.460 20.22  1
                  am gear carb
Mazda RX4          1    4    4
Mazda RX4 Wag      1    4    4
Datsun 710         1    4    1
Hornet 4 Drive     0    3    1
```

```
Hornet Sportabout  0    3    2
Valiant            0    3    1
```

4.2 파일 입출력

이 절에서는 CSV로 저장된 외부 데이터 파일을 데이터 프레임으로 불러들이는 방법, 데이터 프레임을 CSV파일로 저장하는 방법을 설명한다. 프로그램 수행 중에 긴 시간이 소요되는 작업을 실행한 뒤에는 중간 중간 실행 결과를 저장해놓고 싶을 때가 있다. 이때 사용할 수 있는 R객체를 바이너리 형태로 파일에 저장했다가 불러들이는 방법에 대해서도 설명한다.

4.2.1 CSV 파일 입출력

CSV 파일을 데이터 프레임으로 읽으려면 read.csv()를, 데이터 프레임을 CSV로 저장하려면 write.csv()를 사용한다. 다음 표에 이 함수들의 프로트타입을 보였다.

read.csv : CSV 파일을 데이터 프레임으로 읽어들인다.

```
read.csv(
    file, #파일명
    header = FALSE, #파일의 첫 행을 헤더로 처리할 것인지 여부, 기본값은 TRUE
    na.strings = "NA",
    #데이터에 결측치가 포함되어 있을 경우 R의 NA에 대응시킬 값을 지정
    #기본값은 "NA"로, "NA"로 저장된 문자열들은 R의 NA로 저장된다.
    stringAsFactors = defalut.stringAsFactor()
    #문자열을 팩터로 저장할지 또는 문자열로 저장할지 여부를 지정하는 데 사용한다.
    #별다른 설정을 하지 않았다면 기본값은 보통 True이다.
)
#반환 값은 데이터 프레임이다.
```

write.csv : 데이터 프레임을 CSV로 저장한다.

```
write.csv(
    x,          #파일에 저장할 데이터프레임 또는 행렬
    file = "",  #데이터를 저장할 파일명
    row.names = TRUE #TRUE면 행 이름을 CSV 파일에 포함하여 저장한다.
    )
```

read.csv(), write.csv()를 사용한 파일 입출력을 알아보자. 다음과 같은 a.csv파일이 있다고 하자.

이 파일의 첫 행은 열의 이름이다. read.csv()를 사용해 파일을 읽어보자.

```
(x<-read.csv("C:/Users/김대현/Desktop/data/R_ex_data/a.csv"))
  id   name score
1  1 Mr.Foo    95
2  2 Ms.Bar    97
3  3 Mr.Baz    92

> str(x)
'data.frame':   3 obs. of  3 variables:
 $ id   : int  1 2 3
 $ name : Factor w/ 3 levels "Mr.Baz","Mr.Foo",..: 2 3 1
 $ score: int  95 97 92
```

보다시피 읽어들인 파일은 데이터 프레임으로 반환된다. b.csv 파일에는 다음과 같이 헤더 행이 없다고 가정해보자.

다음은 a.csv와 달리 헤더파일이 없는 b.csv 파일이 있다고 가정해보자. 이 경우에는 다음과 같이 header = FALSE를 지정한다. 헤더가 없어 컬럼의 이름이 주어지지 않게 되므로, 다음 예에서 보인바와 같이 names()를 사용해 별도로 컬럼 이름을 지정해야 한다.

```
> (x<-read.csv("C:/Users/김대현/Desktop/data/R_ex_data/b.csv"))
  X1 Mr.Foo X95
1  2 Ms.Bar  97
2  3 Mr.Baz  92
> names(x)<-c("id","name","score")
> x
  id   name score
1  2 Ms.Bar    97
2  3 Mr.Baz    92
> str(x)
'data.frame':   2 obs. of  3 variables:
 $ id   : int  2 3
 $ name : Factor w/ 2 levels "Mr.Baz","Ms.Bar": 2 1
 $ score: int  97 92
```

위에서 데이터를 읽어들인 결과를 보면 name 컬럼이 모두 팩터 형태로 변환되었다. 그러나 이름은 범주형 변수가 아니므로 다음과 같이 다시 문자열로 변환해줘야 한다.

```
> x<-read.csv("C:/Users/김대현/Desktop/data/R_ex_data/a.csv",stringsAsFactors =
F)
> str(x)
'data.frame':   3 obs. of  3 variables:
 $ id   : int  1 2 3
 $ name : chr  "Mr.Foo" "Ms.Bar" "Mr.Baz"
 $ score: int  95 97 92
```

또는 처음부터 문자열을 팩터가 아닌 문자열 타입으로 읽도록 stringsAsFactors = FALSE를 지정해도 된다.

때에 따라서는 다음에 보인 c.csv파일처럼 데이터에 NA를 지정하는 문자열이 저장되어 있을 수 있다. NA를 NIL이라는 문자열로 자동 저장하는 파일이 있다고 가정해보자.

read.csv()로 읽으면 NIL은 문자열이므로 팩터로 저장된다. 그 결과 score 컬럼 전체는 팩터가 된다.

이러한 결과를 피하려면 na.strings 인자를 사용하여 na.strings의 기본값은 "NA"로, "NA"라는 문자열이 주어지면 이를 R이 인식하는 NA로 바꿔준다. 이 예시에서는 na.strings = c("NIL")을 사용하여 NIL을 NA로 지정할 수 있다. na.strings에 지정하는 값은 벡터이므로 여러 문자열을 벡터로 지정하면 벡터 내 모든 문자열이 NA로 저장된다.

```
> is.na(x$score)
[1] FALSE  TRUE FALSE
```

is.na()로 확인한 결과 NIL이 NA로 잘 변환되었음을 볼 수 있다. 데이터를 CSV파일로 저장하려면 wrote.csv()를 사용한다. 다음 예에서는 row.names를 FALSE로 지정하여 행 이름은 제외하고 파일에 저장한다.

```
write.csv(x, "d.csv",row.names = FALSE)
```

다음은 row.names의 F와 T에 대한 차이다. T일 때는 행에 번호가 매겨진다. 이 CSV 파일의 첫 번째 컬럼은 행 번호 1,2,3을 의미한다. 그러나 이 값이 데이터에 꼭 필요한 정보가 아니면 생략하는 것이 낫다.

```
write.csv(x, "e.csv",row.names = FALSE)
write.csv(x, "e_1.csv",row.names = T)
> (read.csv("e.csv"))
  id   name score
1  1 Mr.Foo    95
2  2 Ms.Bar    NA
3  3 Mr.Baz    92

> (read.csv("e_1.csv"))
  X id   name score
1 1  1 Mr.Foo    95
2 2  2 Ms.Bar    NA
3 3  3 Mr.Baz    92
```

4.2.2 객체의 파일 입출력

데이터를 다양한 알고리즘으로 장시간 처리한 뒤 파일에 저장해두면 나중에 계산을 반복할 필요가 없어 효율적이다. 바이너리 파일로 R 객체를 저장하고 불러들이는 함수에는 save(), load()가 있다.

save() : 메모리에 있는 객체를 파일에 저장한다.

```
save(
    ..., #저장할 객체의 이름
    list #저장할 객체의 이름을 벡터로 지정할 경우 ... 대신 사용
    file #파일명
    )
```

load() : 파일로부터 객체를 메모리로 읽어들인다.

```
load(
    file #파일명
    )
#반환 값은 파일에서 읽어들인 객체의 이름들을 저장한 벡터다.
```

다음은 두 벡터 x,y를 xy.RData 파일에 저장하는 예다.

```
x<-1:5
y<-6:10
save(x,y,file="xy.RData")
```

메모리에 있는 모든 객체를 저장하고 싶다면 메모리에 있는 객체 목록을 조회하는 함수 ls()의 결과를 list 인자에 지정할 수 있다. 다음은 a,b,c 객체를 파일 abc.RData에 저장하는 예이다.

```
rm(list=ls())
a<-1:5
b<-6:10
c<-11:15
save(list=ls(),file="abc.RData")
```

파일로 부터 데이터를 불러들이는 함수는 load()이다. 다음 코드는 abc.RData 파일로부터 a,b,c 객체를 불러들이는 예를 보여준다.

```
> rm(list=ls())
> ls()
character(0)
> load("abc.RData")
> ls()
[1] "a" "b" "c"
> a
[1] 1 2 3 4 5
> b
[1]  6  7  8  9 10
> c
[1] 11 12 13 14 15
```

4.3 데이터 프레임의 행과 컬럼 합치기

rbind()와 cbind()는 각각 행 또는 컬럼 형태로 주어진 벡터, 행렬, 데이터 프레임을 합쳐서 결과로 행렬 또는 데이터 프레임을 만드는 데 사용한다. 이들 함수는 분리되어 저장된 데이터를 합치는 데 유용하게 사용할 수 있다.

함수	의미
rbind()	지정한 데이터들을 행으로 취급해 합친다.
cbind()	지정한 데이터들을 컬럼으로 취급해 합친다.

예를 들어, c(1,2,3), c(4,5,6)이라는 두 벡터는 rbind()를 사용해 각 벡터를 한 행으로 하는 행렬로 합칠 수 있다.

```
> rbind(c(1,2,3),c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

마찬가지로 데이터 프레임 역시 rbind()를 사용하여 행을 합칠 수 있다. 다음은 2개 행을 저장한 데이터 프레임 x와 새로운 값을 저장한 벡터 c(3,"c")를 rbind()로 합쳐서 새로운 데이터 프레임 y에 저장하는 예이다.

```
> (x<-data.frame(id=c(1,2),name=c("a","b"),stringsAsFactors=F))
  id name
1  1    a
2  2    b
> str(x)
'data.frame':   2 obs. of  2 variables:
 $ id  : num  1 2
 $ name: chr  "a" "b"
> (y<-rbind(x,c(3,"c")))
  id name
1  1    a
2  2    b
3  3    c
```

위 코드의 첫 행에서 stringsAsFactors 는 name 컬럼의 데이터를 팩터가 아닌 문자열로 취급하기 위해 필요하다. 만약 stringsAsFactors를 지정하지 않으면 "a", "b"가 팩터 데이터로 취급되어 이름을 표현하려는 컬럼의 목적에 어긋나게 된다.

cbind()는 주어진 인자를 컬럼으로 취급하여 데이터를 합친다. 다음은 두 벡터를 합쳐 행렬로 만드는 예다.

```
> cbind(c(1,2,3),c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

마찬가지 방법으로 cbind()를 사용해 데이터 프레임에 새로운 컬럼을 추가할 수 있다.

```
> (y<-cbind(x,greek=c("alpha", "beta")))
  id name greek
1  1    a alpha
2  2    b  beta
> str(y)
'data.frame':   2 obs. of  3 variables:
 $ id   : num  1 2
 $ name : chr  "a" "b"
 $ greek: Factor w/ 2 levels "alpha","beta": 1 2
> y<-cbind(x, greek = c("alpha","beta"),stringsAsFactors = F)
> str(y)
'data.frame':   2 obs. of  3 variables:
 $ id   : num  1 2
 $ name : chr  "a" "b"
 $ greek: chr  "alpha" "beta"
```

위 코드에서도 확인할 수 있듯, stringsAsFactors를 FALSE로 지정하면 새로 추가된 greek 컬럼이 문자열 데이터가 되지만, 이를 생략하면 범주형 데이터인 팩터가 된다.

데이터 프레임에 새로운 컬럼을 추가할 때는 cbind()를 사용하지 않고 df$colname<- data 형태로도 추가할 수 있다.

4.4 apply 계열 함수

R에는 벡터, 행렬 또는 데이터 프레임에 임의의 함수를 적용한 결과를 얻기 위한 apply 계열 함수가 있다. 이 함수들은 데이터 전체에 함수를 한 번에 적용하는 벡터 연산을 수행하므로 속도가 빠르다. 다음은 apply 계열 함수를 요약한 것이다.

함수	설명	다른 함수와 비교했을 때의 특징
apply()	배열 또는 행렬에 주어진 함수를 적용한 뒤 그 결과를 벡터, 배열 또는 리스트로 반환	배열 또는 행렬에 적용
lapply()	벡터, 리스트 또는 표현식에 함수를 적용하여 그 결과를 리스트로 반환	결과가 리스트
sapply()	lapply와 유사하지만 결과를 벡터, 행렬 또는 배열로 반환	데이터를 그룹으로 묶은 뒤 함수를 적용
tapply()	벡터에 있는 데이터를 특정 기준에 따라 그룹으로 묶은 뒤 각 그룹마다 주어진 함수를 적용하고 그 결과를 반환	데이터를 그룹으로 묶은 뒤 함수를 적용
mapply()	sapply()의 확장된 버전으로, 여러 개의 벡터 또는 리스트를 인자로 받아 함수에 각 데이터의 첫째 요소들을 적용한 결과, 둘째 요소들을 적용한 결과, 셋째 요소들을 적용한 결과 등을 적용	여러 데이터를 함수의 인자로 적용

4.4.1 apply()

apply()는 행렬의 행 또는 열 방향으로 특정 함수를 적용하는데 사용한다.

```
apply(
    X, #배열 또는 행렬
    MARGIN, #함수를 적용하는 방향. 1은 행방향, 2는 열 방향
            #c(1,2)는 행과 열 방향 모두를 의미
    FUN # 적용할 함수
)
#반환 값은 FUN이 길이 1인 벡터들을 반환한 경우 벡터, 1보다 큰 벡터들을 반환한 경우 행렬,
# 서로 다른 길이의 벡터를 반환한 경우 리스트다.
```

apply()가 적용된 결과가 벡터, 배열, 리스트 중 어떤 형태로 반환될 것인지는 데이터 X의 데이터 타입과 함수 FUN의 반환 값에 따라 대부분 자연스럽게 예상할 수 있으므로, 반환 값의 데이터 타입에 대해 크게 걱정할 필요는 없다. 또, 반환 값을 str()로 검토하면 데이터 타입을 알아낼 수 있다는 점을 기억하기 바란다.

합을 구하는 함수 sum()을 apply()에 적용하는 예에 대해 알아보자. sum()은 인자로 주어진값들의 합을 구하는 간단한 함수다. 예를 들어, 다음은 1부터 10까지의 합을 구하는 코드이다.

```
> sum(1:10)
[1] 55
```

이를 사용해 apply()로 행렬에 저장된 데이터의 합을 구해보자. 예를 들어, 다음과 같은 행렬이 있다고하자.

```
> d<-matrix(1:9, ncol=3)
> d
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

이 행렬의 각 행의 합 (즉, 1+4+7, 2+5+8, 3+6+9)을 구하려면 apply를 행 별로(즉, MARGIN에 1을 지정) 처리하되 각 행에 대해 sum함수를 호출하면 된다.

```
> apply(d,1,sum)
[1] 12 15 18
> apply(d,2,sum)
[1]  6 15 24
```

apply()를 사용하여 아이리스 데이터의 컬럼의 합을 구해보자 .다음 코드에서 iris[,1:4]는 아이리스 데이터 프레임의 모든 행에서 1~4컬럼에서만 가져온다는 의미다.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length
1          5.1         3.5          1.4
2          4.9         3.0          1.4
3          4.7         3.2          1.3
4          4.6         3.1          1.5
5          5.0         3.6          1.4
6          5.4         3.9          1.7
  Petal.Width Species
1         0.2  setosa
2         0.2  setosa
3         0.2  setosa
4         0.2  setosa
5         0.2  setosa
6         0.4  setosa
> apply(iris[,1:4],2,sum)
Sepal.Length  Sepal.Width Petal.Length
       876.5        458.6        563.7
 Petal.Width
       179.9
```

이와 같은 행 또는 열의 합 또는 평균의 계산은 빈번히 사용되므로 rpwSums(), rowMeans(), colSums(), colMeans() 함수가 미리 정의되어 있다.

rowSums : 숫자 배열 또는 데이터 프레임에서 행의 합을 구한다.

```
rowSums{
    x, # 배열 또는 숫자를 저장한 데이터 프레임
    na.rm = FALSE # NA를 제외할지 여부
}
#반환 값은 행 방향에 저장된 값의 합이다.
```

다음은 앞서 apply()를 통해 구한 계산을 colSums()로 수행하는 예이다.

```
> colSums(iris[,1:4])
Sepal.Length  Sepal.Width
       876.5        458.6
Petal.Length  Petal.Width
       563.7        179.9
```

4.4.2 lapply()

lapply()는 리스트를 반환하는 특징이 있는 apply 계열 함수다.

lapply() 및 그 결과를 처리하기 위한 함수

lapply : 벡터, 리스트, 표현식, 데이터 프레임 등에 함수를 적용하고 그 결과를 리스트로 반환한다.

```
lapply(
    X, #벡터,리스트, 표현식 또는 데이터 프레임
    FUN, #적용할 함수
    ... #추가 인자, 이 인자들은 FUN에 전달된다.
    )
#반환 값은 X와 같은 길이의 리스트다.
```

리스트보다는 벡터 또는 데이터 프레임이 사용하기에 직관적인 면이 있으므로 lapply()의 결과를 벡터 또는 데이터 프레임으로 변환할 필요가 있다. 이 경우 다음과 같은 함수를 사용한다.

unlist : 리스트 구조를 벡터로 변환한다.

```
unlist(
    x, #R 객체,보통 리스트 또는 벡터
    recursive = FALSE, #x에 포함된 리스트 역시 재귀적으로 변환할지 여부
    use.name = TRUE #리스트 내 값의 이름을 보존할지 여부
    )
#반환 값은 벡터다.
```

do.call : 함수를 리스트로 주어진 인자에 적용하여 결과를 반환한다.

```
do.call(
    what, #호출할 함수
    args #함수에 전달할 인자의 리스트
    )
#반환 값은 함수 호출 결과다.
```

c(1,2,3) 벡터가 있을 때, 각 숫자를 2배한 값을 lapply()를 통해 구해보자. 앞서 '2.5 리스트'절에서 살펴봤듯이, 리스트의 각 값은 [[n]] 형태로 접근한다는 점을 기억하기 바란다.(이때 n은 접근할 요소의 색인이다)

```
> (result<-lapply(1:3,
+                 function(x){
+         x*2}
+         ))
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
[1] 6
```

위의 예처럼 lapply()의 결과는 리스트다. 이 결과를 다시 벡터로 변환하고 싶다면 unlist()를 사용한다.

```
> unlist(result)
[1] 2 4 6
```

lapply()는 인자로 리스트를 받을 수 있다. 다음은 a에는 c(1,2,3), b에는 c(4,5,6)이 저장된 리스트에서 각 변수마다 평균을 계산한 예다.

```
> (x<-list(a=1:3,b=4:6))
$a
[1] 1 2 3

$b
[1] 4 5 6

> lapply(x,mean)
$a
[1] 2

$b
[1] 5
```

데이터 프레임에도 곧바로 lapply()를 적용할 수 있다. 아이리스 데이터의 숫자형 데이터에 대한 평균을 구해보자.

```
> lapply(iris[,1:4], mean)
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

앞서 설명했듯 이를 colMeans로 구할 수도 있다.

```
> colMeans(iris[,1:4])
Sepal.Length  Sepal.Width Petal.Length
    5.843333     3.057333     3.758000
 Petal.Width
    1.199333
```

데이터 프레임을 처리한 결과를 리스트로 얻은 뒤 해당 리스트를 다시 데이터 프레임으로 변환할 필요가 있을 수 있다. 이 변환은 몇 단계를 거쳐서 처리해야한다.

1. unlist()를 통해 리스트를 벡터로 변환한다.
2. matrix()를 통해 벡터를 행렬로 변환한다.
3. as.data.frame()을 사용해 행렬을 데이터 프레임으로 변환한다.
4. names()를 사용해 리스트로부터 변수명을 얻어와 데이터 프레임의 각 컬럼에 이름을 부여한다.

다음은 아이리스 데이터의 각 컬럼에 대한 평균을 lapply()로 구한 뒤 이 결과를 다시 데이터 프레임으로 변환한 예다.

```
> d<-as.data.frame(matrix(unlist(lapply(iris[,1:4],mean)),
                    ncol =4, byrow = T))
> names(d)<- names(iris[,1:4])
> d
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.843333    3.057333        3.758    1.199333
```

또는 do.call()을 사용해 변환할 수도 있다. 지금 살펴보는 예제의 경우에는 lapply()가 반환한 리스트 내의 각 컬럼별 계산 결과가 들어 있다. 따라서 이를 새로운 데이터 프레임의 컬럼들로 합치기 위해 cbind()를 사용한다. 다음 코드는 do.call()을 사용해 lapply()의 결과로 나온 리스트 내 요소 각각을 cbind()의 인자들로 넘겨준다.

앞에서 살펴본 두 가지 방법 중 unlist() 후 matrix()를 거쳐 데이터 프레임으로 변환하는 방법에는 한가지 문제가 있다. unlist()는 벡터를 반환하는데, 벡터에는 한 가지 데이터 타입만 저장할 수 있기에 변환 과정에서 하나의 데이터 타입으로 데이터가 모두 형 변환되어버리기 때문이다.

다음 예에서는 문자열과 숫자가 혼합된 경우 unlist()가 문자열을 모두 엉뚱한 값으로 바꿔버리는 것을 볼 수 있다.

```
> x <- list(data.frame(name = "foo", value = 1),
+           data.frame(name = "bar", value = 2))
> unlist(x)
 name value  name value
    1     1     1     2
```

따라서 데이터 타입이 혼합된 경우에는 do.call()을 사용해야 한다. 다음 예에서는 리스트의 각 요소가 한 행에 해당하므로 rbind를 호출한다.

```
> x <- list(data.frame(name = "foo", value = 1),
+           data.frame(name = "bar", value = 2))
> do.call(rbind, x)
  name value
1  foo     1
2  bar     2
```

이것만으로 끝이라면 좋겠지만 아쉽게도 do.call(rbind, ...) 방식은 속도가 느리다는 단점이 있다. 따라서 대량의 데이터를 변환해야 한다면 rbindlist()를 사용해야 한다.

4.4.3 sapply()

sapply()는 lapply()와 유사하지만 리스트 대신 행렬, 벡터 등의 데이터 타입으로 결과를 반환하는 특징이 있는 함수다.

sapply : 벡터, 리스트, 표현식, 데이터 프레임 등에 함수를 적용하고 그 결과를 벡터 또는 행렬로 반환한다.

```
sapply(
    X, #벡터, 리스트, 표현식 또는 데이터 프레임
    FUN, #적용할 함수
    ..., #추가 인자, 이 인자들은 FUN에 전달된다.
)
#반환 값은 FUN의 결과가 길이 1인 벡터들이면 벡터, 길이가 1보다 큰 벡터들이면 행렬이다.
```

예를 들어, 아이리스의 컬럼별로 평균을 구하는 경우를 살펴보자. 다음 코드에서 볼 수 있듯이 lapply()는 결과를 리스트로 반환하지만, sapply()는 벡터를 반환한다.

```
> lapply(iris[,1:4], mean)
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333

> sapply(iris[,1:4],mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
> class(sapply(iris[,1:4],mean))
[1] "numeric"
```

sapply()에서 반환한 벡터는 as.data.frame()을 사용해 데이터 프레임으로 변환할 수 있다. 이때 t(x)를 사용해 벡터의 행과 열을 바꿔주지 않으면 기대한 것과 다른 모양의 데이터 프레임을 얻게 된다.

다음은 아이리스의 컬럼별 평균 sapply()를 사용해 벡터로 구한 뒤 이를 다시 데이터 프레임으로 변환하는 예이다.

```
> x <- sapply(iris[,1:4],mean)
> as.data.frame(x)
                    x
Sepal.Length 5.843333
Sepal.Width  3.057333
Petal.Length 3.758000
Petal.Width  1.199333
> as.data.frame(t(x))
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.843333    3.057333        3.758    1.199333
```

다수의 컬럼을 포함하는 데이터 프레임을 처리하다 보면 종종 각 컬럼의 데이터 타입을 알고 싶을 때가 있다. 예를 들면, 어떤 컬럼에 숫자가 저장되어 있는지를 판단하는 경우이다.

이때 sapply()를 유용하게 사용할 수 있다. 다음은 아이리스 데이터에서 각 컬럼의 데이터 타입을 구하는 예다.

```
> sapply(iris, class)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   "numeric"    "numeric"    "numeric"    "numeric"
     Species
    "factor"
```

지금까지 설명한 sapply()의 사용 예에서 FUN은 1개 값을 반환했다. 예를 들어, mean 함수의 반환 값은 1개 값이다. 이 경우 sapply()의 결과는 이들 값을 모은 벡터가 된다. 하지만 sapply()에 인자로 주어진 함수의 출력이 길이가 1보다 큰 벡터들이라면 sapply()는 행렬을 반환한다. 다음 예에서는 아이리스의 숫자형 값들에 대해 각 값이 3보다 큰 지 여부를 판단한다. sapply()에 넘긴 FUN의 반환 값이 길이가 1보다 큰 벡터기 때문에 sapply() 수행 결과는 행렬이다.

```
> y <- sapply(iris[,1:4],function(x){x>3})
> class(y)
[1] "matrix"
> head(y)
     Sepal.Length Sepal.Width Petal.Length
[1,]         TRUE        TRUE        FALSE
[2,]         TRUE       FALSE        FALSE
[3,]         TRUE        TRUE        FALSE
[4,]         TRUE        TRUE        FALSE
[5,]         TRUE        TRUE        FALSE
[6,]         TRUE        TRUE        FALSE
     Petal.Width
[1,]       FALSE
[2,]       FALSE
[3,]       FALSE
[4,]       FALSE
[5,]       FALSE
[6,]       FALSE
```

sapply()는 한 가지 타입만 저장 가능한 데이터 타입인 벡터 또는 행렬을 반환하므로, sapply()에 인자로 준 함수 FUN의 반환 값에 여러 가지 데이터 타입이 섞여 있으면 안된다. sapply()에 인자로 준 함수 FUN의 반환 값에 여러 가지 데이터 타입이 섞여 있으면 안된다. 만약 각 컬럼에 대해 수행한 함수의 결과 데이터 타입이 서로 다르다면, 리스트를 반환하는 lapply()나 리스트 또는 데이터 프레임을 반환할 수 있는 plyr 패키지를 사용해야 한다.

4.4.4 tapply()

tapply()는 그룹별로 함수를 적용하기 위한 apply 계열 함수다.

tapply: 벡터 등에 저장된 데이터를 주어진 기준에 따라 그룹으로 묶은 뒤 각 그룹에 함수를 적용하고 그 결과를 반환한다.

```
tapply(
    X, # 벡터
    INDEX, #데이터를 그룹으로 묶을 색인, 팩터를 지정해야 하며 팩터가 아닌 타입이 지정되면
           #팩터로 형 변환된다.
    FUN, #각 그룹마다 적용할 함수
    ..., #추가 인자, 이 인자들은 FUN에 전달된다.
    )
#반환 값은 배열이다.
```

다음과 같은 예를 생각해보자. 1부터 10까지의 숫자가 있고 이들이 모두 한 그룹에 속해 있을 때,

이 그룹에 속한 데이터의 합을 구하면 55가 될 것이다.

```
> tapply(1:10,rep(1,10),sum)
 1
55
```

위 코드에서 rep(1,10)는 1을 10회 반복하는 것을 의미한다. 따라서 숫자 1,2,..., 10에 대해 동일한 소속번호 1,1,1,... 1 을 부여한 것이다. 그러므로 그룹 1에 속한 데이터의 합은 55(1+2+3+...+10 =55)이다.

이번에는 1부터 10까지의 숫자를 홀수별, 짝수별로 묶어서 합을 구해보자. INDEX에 홀수와 짝수별로 다른 팩터 값이 주어지도록 %%2를 사용했다.

```
> tapply(1:10,1:10%%2 == 1, sum)
FALSE  TRUE
   30    25
```

수행 결과 짝수의 합이 30, 홀수의 합이 25로 구해졌다. 아이리스 데이터에서 Specise 별로 Sepal.Length의 평균을 구해보자.

```
> tapply(iris$Sepal.Length, iris$Species, mean)
    setosa versicolor  virginica
     5.006      5.936      6.588
```

이번에는 조금 더 복잡한 그룹화를 다뤄보자. 계절별 성별로 정리된 판매량 데이터가 다음과 같이 주어졌다고 하자.

```
> m <- matrix(1:8, ncol=2,
+             dimnames = list(c("spring", "summer"
,"fall","winter"),
+                             c("male","female")))

> m
       male female
spring    1      5
summer    2      6
fall      3      7
winter    4      8
```

행렬의 행은 봄, 여름, 가을, 겨울을 뜻하고 열은 성별을 의미한다.

이때 반기별, 셩별 셀의 합을 구해보자. 즉, 상반기의 남성 셀 합(1+2)과 여성 셀 합(5+6),

하반기 남성 셀(3+4), 하반기 여성 셀(7+8)을 구하는 것이 목표다.

이 연산을 수행하기 위한 성별, 분기별 그룹은 다음과 같이 구성할 수 있다. 그림에서 (n,m) 값은 tapply()에 INDEX로 주어질 값들을 의미한다.

▼ 표 4-13 성별, 분기별 그룹을 위한

	male		female	
spring	(1, 1)	1	(1, 2)	5
summer	(1, 1)	2	(1, 2)	6
fall	(2, 1)	3	(2, 2)	7
winter	(2, 1)	4	(2, 2)	8

INDEX 를 실제로 지정할 때는 (n,m)에서 n을 먼저 나열한 뒤 m값을 나열한다. 즉, 그룹(n1, m1),(n2, m2)는 list(c(n1,n2),c(m1,m2))로 표현한다. 다음은 이러한 방식으로 tapply()를 사용한 분기별, 성별 합을 구한 예다.

```
> tapply(m,list(c(1,1,2,2,1,1,2,2),
+               c(1,1,1,1,2,2,2,2)),sum)
  1  2
1 3 11
2 7 15
```

tapply()는 클러스터링 알고리즘을 수행한 후 같은 클러스터에 속한 데이터들의 x좌표의 평균, y좌표의 평균을 계산하는 데 사용할 수 있는데 바로 이때 위와 같은 방식으로 색인을 부여한다. 따라서 조금 복잡해 보여도 여기서 배운 내용을 꼭 알고 넘어가야 한다.

4.4.5 mapply()

이제 apply() 계열 함수의 마지막 변형으로 mapply()에 대해 알아보자. mapply()는 sapply()의 유사하지만 다수의 인자를 함수에 넘긴다는 점에서 차이가 있다. 주요 사용 목적은 다수의 인자를 받는 함수 FUN()이 있고 FUN()에 넘겨줄 인자들이 데이터로 저장되어 있을 때, 데어터에 저장된 값들을 인자로 하여 함수를 호출하는 것이다. 다음 표에 mapply()에 대해 보였다.

mapply : 함수에 리스트 또는 벡터로 주어진 인자를 적용한 결과를 반환한다.

```
mapply(
        FUN, #실행할 함수
        ..., #적용할 인자
        )
#...에 주어진 여러 데이터가 있을 때 FUN에 이들 데이터 각각의 첫째 요소를 인자로 전달하여
#실행한 결과, 각각의 둘째 요소를 인자로 전달하여 실행한 결과 등을 반환한다.
```

여기에서는 정규 분포를 따르는 난수를 생성하는 rnorm() 함수에 mapply()를 적용하는 예를 살펴볼 것이다. R에는 다양한 난수 생성함수가 있다. 이 함수들은 'r{분포명}()' 형태를 띈다.

함수	의미
rnorm(n,mean = 0, sd =1)	평균이 n,표준편차가 sd인 정규 분포를 따르는 난수 n개 발생
runimf(n, min=0, max = 1)	최솟값이 min, 최댓값이 max인 균등 분포를 따르는 난수 n개 발생
rpois(n, lambda)	람다 값이 lambda인 포아송 분포를 따르는 난수 n개 발생
rexp(n, rate = 1)	람다가 rate 인 지수 분포를 따르는 난수 n개 발생

mapply()를 사용해 rnorm()을 호출하는 방법에 대해 알아보자. rnorm()은 n, mean, sd 세 인자를 받는 함수다. 예를 들어 평균 0, 표준 편차 1을 따르는 난수 10개는 다음과 같이 생성시킬 수 있다.

```
> rnorm(10,0,1)
 [1]  0.7898706 -0.2299939 -0.8185025  0.4997342
 [5]  0.1591923  0.5426264 -0.1566451  0.4387933
 [9]  1.4878706  0.0601651
```

rnorm()을 다음 세 가지 조합에 대해 호출할 필요가 있다고 해보자 .

n	mean	sd
1	0	1
2	10	1
3	100	1

이를 수행하기 위해 rnorm을 세번 호출해도 되지만 또 다른 방법은 mapply()를 활용하는 것이다.

mapply()는 인자로 주어진 데이터들의 첫 번째 요소들을 묶어서 FUN을 호출하고, 두 번째 인자들을 묶어 FUN을 호출하는 식으로 동작한다. 따라서 다음과 같이 mapply()를 사용할 수 있다.

```
> mapply(rnorm,
+        c(1,2,3), # n
+        c(0,10,100), # mean
+        c(1,1,1)) # sd
[[1]]
[1] -0.8490129

[[2]]
[1] 12.339693  9.878797

[[3]]
[1]  98.04979 100.53871 101.69351
```

위 실행 결과에서 [[1]]은 rnorm(n=1, mean = 0, sd = 1), [[2]]은 rnorm(n=2, mean = 10, sd = 1),

[[3]]은 rnorm(n=3, mean = 100, sd = 1)에 해당한다.

또 다른 예로 아이리스의 각 컬럼 평균을 구하는 경우를 살펴보자.

```
> mapply(mean, iris[,1:4])
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

mapply()의 인자로 iris[,1:4]가 주어졌다. 따라서 mapply()에는 iris의 모든 행이 나열되어 인자로 주어졌다고 볼 수 있다. mapply()가 주어진 인자들을 하나씩 묶어 mean()을 호출해주므로 각 행의 첫 번째 컬럼끼리 묶어 평균을 구하고, 다시 두 번째 컬럼 끼리 묶어 평균을 구하는 작업을 반복한다. 그 결과 모든 컬럼의 평균이 계산된다.

4.5 데이터를 그룹으로 묶은 후 함수 호출하기

데이터 분석에서는 데이터 전체에 대해 함수를 호출하기보다는 데이터를 그룹별로 나눈 뒤 각 그룹별로 함수를 호출하는 일이 흔하다. 앞 절에서 설명한 tapply() 외에도 이런 목적에 특화된 패키지들이 있는데, doBy는 그중 잘 알려진 패키지다.

doBy 패키지에는 summaryBy(), orderBy(), sampleBy()와 같이 특정 값에 따라 데이터를 처리하는 유용한 함수들이 있다. 다음은 이 함수들의 특징을 요약한 것이다.

함수	특징
doBy::summaryBy()	데이터 프레임을 컬럼 값에 따라 그룹으로 묶은 후 요약 값 계산
doBy::orderBy()	지정된 컬럼 값에 따라 데이터 프레임을 정렬
doBy::sampleBy()	데이터 프레임을 특정 컬럼 값에 따라 그룹으로 묶은 후 각 그룹에서 샘플(sample) 추출

doBy를 설치 및 로드해보자.

```
install.packages('doBy')
library(doBy)
```

4.5.1 summaryBy()

summaryBy()는 그룹별로 그룹을 특징짓는 통계적 요약 값을 계산하는 함수다. summaryBy()에 대해 살펴보기에 앞서 R에 기본으로 포함되어 있는 base 패키지 내의 summary() 함수, 즉 base::summary() 함수에 대해서 먼저 살펴보자.

base::summary: 다양한 모델링 함수의 결과에 대해 요약 결과를 반환한다.

```
summary(
    object #요약할 객체
    )
#반환 값은 요약 결과며, 데이터 타입은 object 차입에 따라 다르다.
```

doBy::summaryBy : 포뮬러에 따라 데이터를 그룹으로 묶고 요약한 결과를 반환한다.

```
doBy::summaryBy(
    formula,  #요약을 수행할 포뮬러
    data = parent.frame() #포뮬러를 적용할 데이터
    )
#반환 값은 데이터 프레임이다.
```

summary()는 데이터에 대한 간략한 통계 요약을 보기 위해 사용하는 일반 함수다. 일반 함수는 주어진 인자에 따라 다른 동작을 수행하는 함수로, summary()의 경우에는 데이터가 인자로 주어지면 간략한 통계 요약을 내놓고, 모델이 인자로 주어지면 모델에 대한 요약을 보여주는 방식으로 동작한다.

다음 코드는 아이리스 데이터에 대한 통계 요약을 summary()로 살펴본 예다.

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length
 Min.   :4.300   Min.   :2.000   Min.   :1.000
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
 Median :5.800   Median :3.000   Median :4.350
 Mean   :5.843   Mean   :3.057   Mean   :3.758
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
 Max.   :7.900   Max.   :4.400   Max.   :6.900
  Petal.Width          Species
 Min.   :0.100   setosa    :50
 1st Qu.:0.300   versicolor:50
 Median :1.300   virginica :50
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500
```

위 결과에 보인 것처럼 summary()는 Sepal.Length 등과 같은 수치형 데이터에 대해서는 최솟값, 1사분위수, 중앙값, 평균, 3사분위수, 최댓값을 보여준다. 팩터인 Species에 대해서는 각 레벨 마다 몇 개의 값이 있는지를 보여준다.

Note_ 수치형 데이터의 분포는 quantile()을 통해서도 알아볼 수 있다. 다음 코드는 데이터를 크기 순서대로 나열 할 때 0,25,50,75,100%에 해당하는 값과 0,10,20, ... ,100%에서의 값을 보여주는 예이다.

```
> quantile(iris$Sepal.Length)
  0%  25%  50%  75% 100%
 4.3  5.1  5.8  6.4  7.9
> quantile(iris$Sepal.Length, seq(0,1,0.1))
  0%  10%  20%  30%  40%  50%  60%  70%  80%  90%
4.30 4.80 5.00 5.27 5.60 5.80 6.10 6.30 6.52 6.90
100%
7.90
```

이제 doBy 패키지에 대해 알아보자.

```
> summaryBy(Sepal.Width + Sepal.Length~Species, iris)
     Species Sepal.Width.mean Sepal.Length.mean
1     setosa            3.428             5.006
2 versicolor            2.770             5.936
3  virginica            2.974             6.588
```

위 코드에서 'Sepal.Width + Sepal.Length~Species' 부분은 포뮬러(수식)이라고 하는데, 처리할 데이터를 일종의 수학 공식처럼 표현하는 방법이다. 이 예에서는 Sepal.Width와 Sepal.Length를 +로 연결해, 이 두 가지에 대한 값을 결과에 각 컬럼으로 놓고, 각 행에는 ~Species를 사용해 Species를 놓았다. 즉, 위 결과는 Sepal.Width와 Sepal.Length를 Species 별로 요약한 것이다.

NOTE_ 포뮬러 해석하기

포뮬러의 일반적인 표현은 다음과 같다.

$$Y1 + Y2 + \ldots + Yn \ \sim X1 + X2 + \ldots + Xm$$

포뮬러의 정확한 의미는 사용하는 함수에 따라 다르나 일반적으로 '(Y1,Y2,...Yn)의 순서쌍을 (X1, X2, ..., Xn)의 순서쌍으로 모델링한다' 고 볼 수 있다. 이때 상수항은 임시적으로 허용된다.

이에 선형 회귀에서 Y ~ X1은 Y = a * X1 + b를 의미한다.

+는 여러 변수를 연결하기 위한 목적 또는 실제로 합을 의미하는 목적으로 사용한다. 이외에도 - , :, |,* 등의 연산자를 사용할 수 있으며 이들의 의미는 다음 표에 정리했다.

연산자	예	의미
+	Y~X1+X2	Y를 X1,X2로 모델링, 상수항은임시적으로 허용한다. 따라서 선형 회귀에 이 포뮬러를 사용하면 Y= a*X1+b * X2 + c를 의미한다.
-	Y~X1-X2	Y를 X1로 모델링 하되 X2를 제외한다. 특히 선형 회귀에서 Y~X1+X2-1은 Y를 X1과 X2로 모델링하되, 상수항은 제외한다는 의미이다. 즉, Y = a*X1*+*b*X2를 의미한다.
|	Y~X1|X2	X2의 값에 따라 데이터를 그룹으로 묶은 후 각 그룹별로 Y~X1을 적용한다.
:	Y~X1:X2	Y를 X1과 X2의 상호 작용에 따라 모델링한다. 상호 작용은 Y = a*X1*X2 + b와 같이 X1과 X2가 동시에 Y값에 영향을 주는 상황을 말한다. 특히 영향을 주는 방식이 Y = a*X1+b*X2 + c와 같은 합의 형태와는 구분된다.
*	Y~X1*X2	Y~X1 + X2 + X1:X2의 축약형 표현이다.

4.5.2 orderBy()

orderBy()는 데이터 프레임을 정렬하는 목적으로 사용한다. 다른 함수들과 다르게 데이터를 그룹으로 묶는 기능은 없다. orderBy() 역시 base 패키지의 order() 함수에 대응한다. 이 두가지 함수를 다음 표에 정리했다.

base::order : 데이터를 정렬하기 위한 순서를 반환한다.

```
order(
..., #정렬할 데이터
    #na.last는 NA값을 정렬한 결과의 어디에 둘 것인지를 제어한다.
    #기본값인 na.last = T는 NA값을 정렬한 결과의 마지막에 둔다.
    #na.last = F는 정렬한 값의 처음에 둔다.
    na.last = T,
    decreasing = F #내림차순 여부
)
#반환 값은 order()와 동일하다.
```

doBy::orderBy : 표뮬러에 따라 데이터를 정렬한다.

```
doBy::orderBy(
    formular, #정렬할 기준을 지정한 표뮬러
              #~의 좌측은 무시하며, ~우측에 나열한 이름에 따라 데이터가 정렬된다.
    data,     #정렬한 데이터
    )
#반환 값은 order()와 동일하다.
```

order() 함수는 주어진 값을 정렬하기 위한 색인을 순서대로 반환하는데, 이를 사용해 정돈된 결과를 얻을 수 있다. 다음 예는 아이리스 데이터를 Sepal.Length에 따라 정렬했을 때 61행이 가장 처음, 63행이 두 번째, 69번째 세 번째 등 해당함을 보여준다.

```
> head(order(iris$Sepal.Width))
[1]  61  63  69 120  42  54
```

이 순서를 데이터 프레임의 각 행에 대한 색인으로 사용하면 정렬된 결과를 얻을 수 있다.

```
> head(iris[order(iris$Sepal.Width),])
    Sepal.Length Sepal.Width Petal.Length Petal.Width
61           5.0         2.0          3.5         1.0
63           6.0         2.2          4.0         1.0
69           6.2         2.2          4.5         1.5
120          6.0         2.2          5.0         1.5
42           4.5         2.3          1.3         0.3
54           5.5         2.3          4.0         1.3
       Species
61  versicolor
63  versicolor
69  versicolor
120  virginica
42      setosa
54  versicolor
```

여러 컬럼을 기준으로 데이터를 정렬하고자 한다면 해당 컬럼들을 order()에 나열한다. 다음은 Sepal.Length, Sepal.Width 순으로 데이터를 정렬한 예다.

```
> head(iris[order(iris$Sepal.Length, iris$Sepal.Width),])
   Sepal.Length Sepal.Width Petal.Length Petal.Width
14          4.3         3.0          1.1         0.1
9           4.4         2.9          1.4         0.2
39          4.4         3.0          1.3         0.2
43          4.4         3.2          1.3         0.2
42          4.5         2.3          1.3         0.3
4           4.6         3.1          1.5         0.2
   Species
14  setosa
9   setosa
39  setosa
43  setosa
42  setosa
4   setosa
```

orderBy는 order와 비슷하지만 정렬할 데이터를 포뮬러로 지정할 수 있다는 점이 편리하다.

다음 예는 모든 데이터를 Sepal.Width로 배열한다. orderBy()에서 ~의 좌측은 무시하므로 적지 않는다.

```
> head(orderBy(~Sepal.Width, iris))
    Sepal.Length Sepal.Width Petal.Length Petal.Width
61           5.0         2.0          3.5         1.0
63           6.0         2.2          4.0         1.0
69           6.2         2.2          4.5         1.5
120          6.0         2.2          5.0         1.5
42           4.5         2.3          1.3         0.3
54           5.5         2.3          4.0         1.3
       Species
61  versicolor
63  versicolor
69  versicolor
120  virginica
42      setosa
54  versicolor
```

다음은 모든 데이터를 Species, Sepal.Width 순으로 정렬한 예다.

```
> head(orderBy(~Species+Sepal.Width, iris))
   Sepal.Length Sepal.Width Petal.Length Petal.Width
42          4.5         2.3          1.3         0.3
9           4.4         2.9          1.4         0.2
2           4.9         3.0          1.4         0.2
13          4.8         3.0          1.4         0.1
14          4.3         3.0          1.1         0.1
26          5.0         3.0          1.6         0.2
   Species
42  setosa
9   setosa
2   setosa
13  setosa
14  setosa
26  setosa
```

4.5.3 sampleBy()

samplyBy()는 데이터를 그룹으로 묶은 후, 각 그룹에서 샘플을 추출하는 함수다. -이 함수는 base::sample()에 대응하므로 함께 알아보도록 하자.

base::sample : 샘플링을 수행한다.

doBy::sampleBy : 포뮬러에 따라 데이터를 그룹으로 묶은 후 샘플을 추출한다.

sample()은 주어진 데이터에서 임의로 샘플(표본)을 추출하는 목적으로 사용된다. 복원 추출 여부는 replace로 지정하며, 기본 값은 비복원 추출(replace = FALSE)이다. 다음은 1에서 10까지의 숫자 중 5개를 샘플링한 결과다. replace = TRUE일 경우 6이 2회 추출된 것을 볼 수 있다.

sample()의 한가지 흥미로운 적용방법은 데이터를 무작위로 섞는 데 사용하는 것이다. 예를 들어, 다음과 같이 1~10의 숫자에서 10개의 샘플을 뽑으면 1부터 10까지의 숫자를 무작위로 섞는 것이 된다.

```
> sample(1:10,5)
[1] 7 4 3 5 1
> sample(1:10,5,replace = T)
[1] 9 4 5 4 9
> sample(1:10,10)
 [1]  7  1 10  4  5  6  8  3  2  9
```

이를 사용하면 iris 데이터 역시 무작위로 섞을 수있다. 다음 코드에서 NROW()는 주어진 데이터 프레임 또는 벡터의 행의 수 또는 길이를 반환하는 함수다.

```
> head(iris[sample(NROW(iris), NROW(iris)),])
    Sepal.Length Sepal.Width Petal.Length Petal.Width
47           5.1         3.8          1.6         0.2
7            4.6         3.4          1.4         0.3
125          6.7         3.3          5.7         2.1
58           4.9         2.4          3.3         1.0
94           5.0         2.3          3.3         1.0
109          6.7         2.5          5.8         1.8
       Species
47      setosa
7       setosa
125  virginica
58  versicolor
```

```
94  versicolor
109  virginica
```

샘플링은 주어진 데이터를 훈련 데이터와 테스트 데이터로 분리하는 데 유용하게 사용 할 수 있다.

훈련 데이터로부터 모델을 만든 뒤 테스트 데이터에 모델을 적용하면 모델의 정확성을 평가할 수 있다. 모델의 정확성을 잘 평가하려면 데이터에서 예측 대상이 되는 값별로 샘플을 균일하게 뽑을 필요가 있다.

예를 들어, 아이리스의 Sepal.Width, Sepal.Length 등으로 부터 Species를 예측하는 모델을 만드는 경우를 생각해보자. 문약 훈려 데이터에는 setosa만 들어 있고, 테스트 데이터에는 vericolor 와 virginica 만 들어 있다면 제대로된 모델링과 모델의 평가가 이뤄질 수 없다. 평가를 올바르게 하려면 훈련 데이터와 테스트 데이터에 Species 값 별로 데이터의 수가 균일한 것이 좋다. 바로 이런 경우에 sampleBy()가 유용하다.

다음은 아이리스 데이터에서 각 Species 별로 10%의 데이터를 추출한 예다. sampleBy()에서 Species 를 지정했으므로 각 종별로 5개씩 데이터가 정확히 추출된 것을 볼 수 있다.

```
> sampleBy(~Species, frac = 0.1, data=iris)
              Sepal.Length Sepal.Width Petal.Length
setosa.1               5.1         3.5          1.4
setosa.13              4.8         3.0          1.4
setosa.18              5.1         3.5          1.4
setosa.19              5.7         3.8          1.7
setosa.25              4.8         3.4          1.9
versicolor.71          5.9         3.2          4.8
versicolor.93          5.8         2.6          4.0
versicolor.96          5.7         3.0          4.2
versicolor.99          5.1         2.5          3.0
versicolor.100         5.7         2.8          4.1
virginica.104          6.3         2.9          5.6
virginica.119          7.7         2.6          6.9
virginica.123          7.7         2.8          6.7
virginica.127          6.2         2.8          4.8
virginica.147          6.3         2.5          5.0
              Petal.Width    Species
setosa.1              0.2     setosa
setosa.13             0.1     setosa
setosa.18             0.3     setosa
setosa.19             0.3     setosa
setosa.25             0.2     setosa
versicolor.71         1.8 versicolor
versicolor.93         1.2 versicolor
versicolor.96         1.2 versicolor
versicolor.99         1.1 versicolor
versicolor.100        1.3 versicolor
virginica.104         1.8  virginica
virginica.119         2.3  virginica
virginica.123         2.0  virginica
virginica.127         1.8  virginica
virginica.147         1.9  virginica
```

4.6 데이터 분리 및 병합

이 절에서는 주어진 데이터를 조건에 따라 분리하는 split(), subset() 함수 그리고 분리되어 있는 데이터를 공통된 값에 따라 병합하는 merge() 함수에 대해 설명한다. 데이터를 분리하는 함수를 사용하면 조건에 만족하는 데이터를 미리 선택할 수 있어 이어지는 처리를 쉽게 할 수 있다. 또, 분리된 데이터는 merge()로 재병합 할 수 있다. 다음 표에 이들 함수의 특징을 요약했다.

함수	특징
split()	주어진 조건에 따라 데이터를 분리한다.
subset()	주어진 조건을 만족하는 데이터를 선택한다.
merge()	주어진 조건을 만족하는 데이터를 병합한다.

4.6.1 split()

split()은 조건에 따라 데이터를 분리하는 데 사용된다.

```
> head(split(iris, iris$Species))
$setosa
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa

$versicolor
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
51           7.0         3.2          4.7         1.4 versicolor
52           6.4         3.2          4.5         1.5 versicolor

$virginica
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
101          6.3         3.3          6.0         2.5 virginica
102          5.8         2.7          5.1         1.9 virginica
```

split()의 실행 결과가 리스트므로, split() 후 lapply()를 적용하면 iris의 종별 Sepal.Length의 평균을 구할 수 있다. 다음 코드를 앞서 설명한 tapply()의 경우와 비교해보길 바란다.

```
> lapply(split(iris$Sepal.Length,iris$Species),mean)
$setosa
[1] 5.006

$versicolor
[1] 5.936

$virginica
[1] 6.588
```

4.6.2 subset()

subset()은 split()과 유사하지만 전체를 부분으로 구분하는 대신 조건을 만족하는 특정 부분만 취하는 용도로 사용한다.

다음은 iris에서 setosa 종만 뽑아내는 예이다.

```
> head(subset(iris, Species == 'setosa'))
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

앞서 '벡터 연산' 절에서 살펴봤듯이 벡터 간 연산에서의 AND, OR는 &&, ||가 아니라 &,|를 사용해야 한다. 따라서 subset()에서 2개 이상의 조건을 조합할 때는 &나 |를 사용한다. 다음은 setosa 종에서 Sepal.Length가 5.0이상인 행을 추출하는 예다.

```
> head(subset(iris, Species == 'setosa' & Sepal.Length > 5.0))
   Sepal.Length Sepal.Width Petal.Length Petal.Width
1           5.1         3.5          1.4         0.2
6           5.4         3.9          1.7         0.4
11          5.4         3.7          1.5         0.2
15          5.8         4.0          1.2         0.2
16          5.7         4.4          1.5         0.4
17          5.4         3.9          1.3         0.4
   Species
1   setosa
6   setosa
11  setosa
15  setosa
16  setosa
17  setosa
```

subset에 select 인자를 지정하면 특정 컬럼을 선택하거나 제외할 수 있다. 다음은 Sepal.Length와 Species 컬럼을 iris에서 선택하여 추력한 예다.

```
 head(subset(iris, select = c(Sepal.Length, Species)))
  Sepal.Length Species
1          5.1  setosa
2          4.9  setosa
3          4.7  setosa
4          4.6  setosa
5          5.0  setosa
6          5.4  setosa
> head(subset(iris, Species == 'setosa' & Sepal.Length > 5.0,
select = c(Sepal.Length, Species)))
   Sepal.Length Species
1           5.1  setosa
6           5.4  setosa
11          5.4  setosa
15          5.8  setosa
16          5.7  setosa
17          5.4  setosa
```

반대로 특정 컬럼을 제외하고자 한다면 -를 컬럼 이름 앞에 붙인다.

```
> head(subset(iris, select = -c(Sepal.Length, Species)))
  Sepal.Width Petal.Length Petal.Width
1         3.5          1.4         0.2
2         3.0          1.4         0.2
3         3.2          1.3         0.2
4         3.1          1.5         0.2
5         3.6          1.4         0.2
6         3.9          1.7         0.4
```

select에서 컬럼을 제외하는 방법을 names()와 %in%을 사용하여 제외하는 방법과 비교해서 알아두기 바란다.

```
> head(iris[,!names(iris) %in% c('Sepal.Length', 'Species')])
  Sepal.Width Petal.Length Petal.Width
1         3.5          1.4         0.2
2         3.0          1.4         0.2
3         3.2          1.3         0.2
4         3.1          1.5         0.2
5         3.6          1.4         0.2
6         3.9          1.7         0.4
```

4.6.3 데이터 병합

merge()는 두 데이터 프레임을 공통된 값을 기준으로 묶는 함수다. merge()는 DB에서 join과 같은 역활을 한다.

다음은 이름 컬럼을 기준으로 수학점수가 저장된 데이터 프레임과 영어 점수가 저장된 데이터 프레임을 병합한 예이다. x와 y에 name 값이 서로 다른 순서로 저장되어 있으나 공통된 name 값을 기준으로 점수가 잘 병합된 것을 볼 수 있다.

```
> x<-data.frame(name=c('a','b','c'),math = c(1,2,3))
> y<-data.frame(name=c('a','b','c'),eng = c(4,5,6))
> merge(x,y)
  name math eng
1    a    1   4
2    b    2   5
3    c    3   6
```

merge()는 cbind()와는 다르다. 앞의 코드는 두 개의 데이터 프레임을 합칠 때 공통되는 컬럼인 name을 기준으로 데이터를 합치지만 cbind()는 다음에 보인 것처럼 단순히 컬럼을 합칠 뿐이다.

```
> cbind(x,y)
  name math name eng
1    a    1    a   4
2    b    2    b   5
3    c    3    c   6
```

merge() 수행 시 공통된 값이 한쪽에만 있는 경우에는 반대편 데이터가 비게 되고 이 경우 해당 행은 병합 결과에서 빠진다. 그러나 이 경우에도 데이터가 비어 있는 쪽의 값을 NA로 채우면서 전체 데이터를 모두 병합하고 싶다면 all 인자에 TRUE를 지정한다.

다음에 all = FALSE(기본값)인 경우와 all = TRUE 인 경우를 비교해봤다.

```
>  x<-data.frame(name=c('a','b','c'),math = c(1,2,3))
>  y<-data.frame(name=c('a','b','d'),eng = c(4,5,6))
> merge(x,y)
  name math eng
1    a    1   4
2    b    2   5
> merge(x,y, all=T)
  name math eng
1    a    1   4
2    b    2   5
3    c    3  NA
4    d   NA   6
```

4.7 데이터 정렬

이 절에서는 데이터를 정렬하는 함수인 sort()와 order()를 설명한다. sort()는 주어진 데이터를 직접 정렬해주는 함수며, order()는 데이터를 정렬했을 때의 순서를 반환한다.

4.7.1 sort()

sort()주어진 벡터를 정렬한 결과를 반환한다.

다음은 임의의 값이 저장된 벡터를 각각 오름 차순과 내림차순으로 정렬한 예이다.

```
> x <- c(20,11,33,50,47)
> sort(x)
[1] 11 20 33 47 50
> sort(x, decreasing = T)
[1] 50 47 33 20 11
> x
[1] 20 11 33 50 47
```

여기서 알 수 있는 점이 있다. sort()는 값을 정렬한 결과를 반환할 뿐이지 인자로 받은 벡터 자체를 변경하지는 않는다.

4.7.2 order()

order()는 주어진 인자를 정렬하기 위한 각 요소의 색인을 반환한다. 다음은 임의의 값을 저장한 벡터 x를 정렬하기 위한 순서를 반환한 예다. order(x)는 x[order(x)]가 정렬되어 있게 하기 위한 색인이다.

```
> x <- c(20,11,33,50,47)
> order(x)
[1] 2 1 3 5 4
```

내림차순으로 정렬한 결과를 얻고 싶다면 decresing = T를 지정한다.

```
> order(x, decreasing = T)
[1] 4 5 3 1 2
```

order()가 정렬된 순서를 반환한다는 점을 이용해 데이터 프레임을 정렬할 수 있다. 다음은 아이리스 데이터를 Sepal.Length에 따라 정렬한 예다.

```
> head(iris[order(iris$Sepal.Length),])
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width
14          4.3         3.0          1.1         0.1
9           4.4         2.9          1.4         0.2
39          4.4         3.0          1.3         0.2
43          4.4         3.2          1.3         0.2
42          4.5         2.3          1.3         0.3
4           4.6         3.1          1.5         0.2
   Species
14  setosa
9   setosa
39  setosa
43  setosa
42  setosa
4   setosa
```

위 결과를 보면 Sepal.Length같은 다 같다. 이에 동률 일 때, Petal.Length를 기준으로 다시 정렬하고자 할 때는 다음과 같이 한다.

```
> head(iris[order(iris$Sepal.Length, iris$Petal.Length),])
   Sepal.Length Sepal.Width Petal.Length Petal.Width
14          4.3         3.0          1.1         0.1
39          4.4         3.0          1.3         0.2
43          4.4         3.2          1.3         0.2
9           4.4         2.9          1.4         0.2
42          4.5         2.3          1.3         0.3
23          4.6         3.6          1.0         0.2
   Species
14  setosa
39  setosa
43  setosa
9   setosa
42  setosa
23  setosa
```

3.8 데이터 프레임 컬럼 접근

데이터 프레임에 저장된 컬럼을 매번 df$colname과 같은 형식으로 접근하면 매번 데이터 프레임 이름 df와 $를 반복하게 되어 코드가 불필요하게 복잡해진다. 이는 리스트인 경우에도 마찬가지다. 이 절에서는 데이터 프레임 또는 리스트의 필드들을 df$colname이 아니라 colname만 적어도 접근할 수 있게 해주는 with(), within() 그리고 attach(), detach()에 대해 알아본다. 다음 표에 이들 함수의 특징을 정리했다.

데이터 프레임 접근 함수

함수	특징
with()	코드 블록 안에서 필드 이름만으로 데이터를 곧바로 접근할 수 있게 한다.
within()	with() 동일한 기능을 제공하지만 데이터에 저장된 값을 손쉽게 변경하는 기능을 제공함
attach()	attach() 이후 코드에서는 필드 이름만으로 데이터를 곧바로 접근할 수 있게 한다.
detach()	attach()의 반대 역활로 detach() 이후 코드에서 더 이상 필드 이름으로 데이터를 곧바로 접근 할 수 없게 한다.

Note_ with()와 attach()의 차이

with(), within()은 명시된 데이터를 환경으로 하여 표현식을 평가한다.

반면 attach(), detach()는 이름을 찾는 검색 경로를 수정하는 방식으로 실행된다.

3.8.1 with()

with()는 데이터 프레임 또는 리스트 내 필드를 필드 이름만으로 접근할 수 있게 해주는 함수다.

with : 데이터 환경에서 주어진 표현식을 평가한다.

```
with(
    data, #환경을 만들 데이터
    expr, #평가할 표현식. expr의 예에는 코드 블록 {...}을 들 수 있다.
    ...   #이후 함수들에 전달할 인자
    )
#반환 값은 expr의 평갓값이다.
```

예를 들어, 아이리스의 Sepal.Length, Sepal.Width의 평균을 다음과 같이 구했다고 해보자.

```
> print(mean(iris$Sepal.Length))
[1] 5.843333
> print(mean(iris$Sepal.Width))
[1] 3.057333
```

위의 두 명령은 컬럼을 접근할 때 마다 매번 iris$colname 형태로 코드를 적어야 했다. with를 사용하면 각 컬럼을 곧바로 접근할 수 있다.

```
> with(iris, {
+     print(mean(Sepal.Length))
+     print(mean(Sepal.Width))
+ })
[1] 5.843333
[1] 3.057333
```

3.8.2 within()

within()은 with()와 비슷하지만 데이터를 수정하는 기능을 제공한다는 차이가 있다.

within : 데이터 환경에서 주어진 표현식을 평가한다.

```
within(
    data, #환경을 만들 데이터
    expr, #평가할 표현식, expr의 예에는 코드 블록 {...}를 만들 수 있다.
    ... #이후 함수들에 전달될 인자
    )
    #반환 값은 expr의 평가에 따라 수정된 데이터다
```

다음은 벡터에서 몇 개 데이터가 결측치 일 때 이를 중앙값으로 치환하는 예다.

```
> (x<-data.frame(val=c(1,2,3,4,NA,5,NA)))
  val
1   1
2   2
3   3
```

```
4   4
5  NA
6   5
7  NA

> x<- within(x, {
+     val <- ifelse(is.na(val), median(val, na.rm=T), val)
+ })
> x<-within(x,{})x

> x
  val
1   1
2   2
3   3
4   4
5   3
6   5
7   3
```

위 코드에서 median() 함수 호출 시에 na,rm = TRUE 를 지정했다. 이는 NA값이 포함된 새로 median()을 호출하면 결과로 NA가 나오기 때문이다.

참고로 위에서 보인 within() 호출 코드는 다음과 같이 표현할 수 도 있다.

```
> x$val[is.na(x$val)]<- median(x$val, na.rm = T)
> x
  val
1   1
2   2
3   3
4   4
5   3
6   5
7   3
```

이번에는 조금 더 복잡하지만 더 실제적인 상황을 다해보자. 아이리스 내 일부 데이터가 결측치 일 때, 결측치를 해당 종의 중앙값으로 바꾸는 예다.

```
> data(iris)
> iris[1,1] = NA
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1           NA         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

```
> median_per_species <- sapply(split(iris$Sepal.Length,iris$Species)
                               ,median, na.rm =T)
> median_per_species
    setosa versicolor  virginica
       5.0        5.9        6.5

> iris <- within(
+     iris, {
+         Sepal.Length<-ifelse(is.na(Sepal.Length),
+                              median_per_species[Species],Sepal.Length)
+     }
+ )

> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.0         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

종별 중앙값 median_per_species를 구하는 sapply() 부분이 조금 복잡하므로 풀어서 살펴보자. sapply() 안에서 사용한 split()은 Sepal.Length를 Species 별로 나누는 역활을 한다.

```
> split(iris$Sepal.Length, iris$Species)
$setosa
 [1] 5.0 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8
[14] 4.3 5.8 5.7 5.4 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0
[27] 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4
[40] 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0

$versicolor
 [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0
[14] 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6
[27] 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6
[40] 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7

$virginica
 [1] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8
[14] 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
[27] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0
[40] 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

split()의 결과에서 Species 별 중앙값을 구하기 위해 sapply()로 median()을 호출하되, median() 호출 시 na.rm = TRUE를 추가적인 인자로 넘겨준다.

within()은 이 결과를 사용해 NA를 중앙값으로 치환한 것이다.

3.8.3 attach(), detach()

with(), within()은 인자로 넘긴 expr 안에서만 데이터 프레임의 컬럼들을 직접 접근할 수 있게 해준다. 반면 attach(), detach()는 이들 함수 호출 후 모든 코드에서 컬럼들을 직접 접근할 수 있게 한다.

R 객체와 이를 검색 경로(search path)

다음은 iris를 attach() 하여 iris의 모든 필드를 직접 접근할 수 있게 했다가 detach()로 해제하는 예이다.

```
> Sepal.Width
Error: object 'Sepal.Width' not found
> attach(iris)
> head(Sepal.Width)
[1] 3.5 3.0 3.2 3.1 3.6 3.9
> detach(iris)
> head(Sepal.Width)
Error in head(Sepal.Width) : object 'Sepal.Width' not found
```

이처럼 이름으로 데이터를 곧바로 접근할 수 있는 이유는 R 객체를 찾는 검색 경로가 attach()를 통해 수정되기 때문이다. search() 사용해보면 attach(iris) 이후 검색 경로의 두 번째에 iris가 추가됨을 볼 수 있다.

```
> search()
 [1] ".GlobalEnv"        "package:doBy"
 [3] "tools:rstudio"     "package:stats"
 [5] "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"
 [9] "package:methods"   "Autoloads"
[11] "package:base"
> attach(iris)
> search()
 [1] ".GlobalEnv"        "iris"
 [3] "package:doBy"      "tools:rstudio"
 [5] "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"
 [9] "package:datasets"  "package:methods"
[11] "Autoloads"         "package:base"
> detach(iris)
> search()
 [1] ".GlobalEnv"        "package:doBy"
 [3] "tools:rstudio"     "package:stats"
 [5] "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"
 [9] "package:methods"   "Autoloads"
[11] "package:base"
```

한 가지 주의할 점은 attch()한 후 이뤄진 변수의 수정은 detach() 시 원래의 데이터 프레임에는 반영되지 않는다는 것이다. 다음 예에서는 iris를 attach() 한 후, Sepal.width의 값을 변경하더라도 이 결과가 iris 에는 반영되지 않음을 보여준다.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.0         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
```

```
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
> attach(iris)
> Sepal.Width[1] = -1
> head(Sepal.Width)
[1] -1.0  3.0  3.2  3.1  3.6  3.9
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.0         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

4.9 조건에 맞는 데이터의 색인 찾기

주어진 데이터에서 조건에 맞는 데이터를 찾을 때, 조건에 맞는 데이터 값 자체를 찾을 때는앞서 설명한 subset()을 사용하거나 데이터 프레임을 접근할 때 조건문을 지정하면 된다. 그러나 때에 따라서는 조건을 만족하는 데이터의 색인 자체를 구할 필요가 있다. which(), which.max(), which.min()은 이런 목적으로 사용할 수 있는 함수들이다.

예를 들어, iris데이터에서 Species가 setosa 인 행은 subset()으로 다음과 같이 찾을 수 있다.

```
> head(subset(iris, Species == 'setosa'))
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.0         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

또는 데이터 프레임을 접근할 때 색인에 진릿값이 지정되도록 조건을 적을 수 있다.

```
> head(iris[iris$Species=='setosa',])
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.0         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
```

이 두가지 방법은 모두 조건을 만족하는 행을 반환하는 역활을 한다. 반면 which()는 조건을 만족하는 행의 색인을 반환한다.

```
> which(iris$Species == 'setosa')
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

which.mini() 과 which.max()는 주어진 벡터에서 최솟값 또는 최댓값이 저장된 색인을 찾는다.

iris에서 Sepal. Length의 최솟값, 최댓값이 속한 행의 색인을 찾아보자.

```
> which.min(iris$Sepal.Length)
[1] 14
> which.max(iris$Sepal.Length)
[1] 132
```

whick.min()과 which.max() 함수는 다양한 파라미터에 따라 모델을 만들고 모델을 선택하는 데 사용할 수 있다. 예를 들어, 기계 학습 모델을 만들고, 모델의 기능도를 벡터에 저장해뒀다면 which.max() 함수를 사용해 기능도가 가장 큰 모델을찾을 수 있다.

4.10 그룹별 연산

doBy가 데이터를 그룹별로 나눈 후 특정 계산을 적용하기 위한 함수들의 패키지인 반면 aggregate()는 좀 더 일반적인 그룹별 연산을 위한 함수다. aggregate()를 사용하면 데이터를 그룹으로 묶은 후 임의의 함수를 그룹에 적용할 수 있다.

그룹별 연산

aggregate : 데이터를 분할하고 각 그룹마다 요약치를 계산한다.

```
aggregate(
    x, #R 객체
    by, #그룹으로 묶을 값의 리스트
    FUN #그룹별로 요약치 계산에 사용할 함수
    )

aggregate(
    formula, #y ~ x 형태로 y는 계산에 사용될 값이며, x는 그룹으로 묶을 때 사용할 기준값
    data, #formula를 적용할 데이터
    FUN
    )
```

이 절에서는 aggregate()에 포뮬러를 적용한 형태의 예를 살펴본다. 다음은 아이리스 데이터에서 종별 Sepal.Width의 평균 길이를 구하는 예다.

```
> aggregate(Sepal.Width ~ Species , iris, mean) #y~x에서 x별로 y를 분류한다.
     Species Sepal.Width
1     setosa       3.428
2 versicolor       2.770
3  virginica       2.974
```

같은 일을 tapply()를 통해서 수행하면 다음과 같다.

```
 tapply(iris$Sepal.Width, iris$Species, mean)
    setosa versicolor  virginica
     3.428      2.770      2.974
```

얻는 결과는 같지만 포뮬러를 사용한 aggregate() 쪽이 코드 작성하기 편리하다.

4.11 편리한 처리를 위한 데이터의 재표현

약물 A,B,C를 실험하고 그 효과를 측정했다고 하자. 이러한 실험의 측정 결과는 보통 다음과 같은 스프레드시트 형태의 표로 정리한다.

A	B	C
3	5	4
2	3	5
9	2	7

그런데 위와 같은 형태는 그래프를 그린다거나 데이터를 조작하는 등의 측면에서 불편한 면이 있다. 예를 들어, 앞서 살펴본 summeryBy()를 위의 데이터에 적용한다고 생각해보자. 만약 효과를 약물별로 요약하고 싶다면 summaryBy(value~category, data) 형태로 명령을 줄 수 있어야 하는데 위의 데이터는 그러한 명령에 적합한 형태가 아니다. summaryBy()를 쉽게 적용하려면 데이터를 다음과 같은 포멧으로 변환해야 한다.

Medicine	Value
A	3
A	2
A	9
B	5
B	3
B	2
C	4
C	5
C	7

변환된 데이터는 summaryBy(Value ~ Medicine, data) 명령으로 쉽게 분석 할 수 있다.

이와 같은 방식으로 정리된 형태의 데이터를 'Tidy Data'라고 부르며, Tidy Data는 조작, 모델링, 시각화 구현이 편하다는 장점이 있다. Tidy Data의 정의는 다음과 같다.

- **각 변수는 하나의 컬럼에 해당한다.**
- **각 관찰은 한 행에 해당한다.**
- **한 관찰 유형은 하나의 테이블을 형상한다.**

스프레드시트 형태로 정리된 데이터와 Tidy Data 형태의 데이터 사이의 변환은 stack(), unstack()으로 수행할 수 있다.

Tidy Data 형태로의 데이터 변환

stack : 다수의 벡터를 하나의 벡터로 합치면서 관측값이 온 곳을 팩터로 명시한다.

```
stack(
    x #리스트 또는 데이터 프레임
)
#반환값은 데이터 프레임이며, values에는 x가 하나로 합쳐진 값들이 저장된다.
#ind에는 관측값이 온 곳을 팩터로 명시
```

앞서 설명한 약물 실험 결과의 데이터 변환을 해보자.

```
> x <- data.frame(a = c(1,2,3),
+                 b = c(5,3,2),
+                 c = c(4,5,7))
> (x_stacked <- stack(x))
  values ind
1      1   a
2      2   a
3      3   a
4      5   b
5      3   b
6      2   b
7      4   c
8      5   c
9      7   c
```

다음은 각 약물별 평균 효과를 stack()을 수행한 데이터에 적용한 예다.

```
> summaryBy(values ~ ind , x_stacked)
  ind values.mean
1   a    2.000000
2   b    3.333333
3   c    5.333333
```

unstacked()은 이름에서 짐작할 수 있듯이 stack()을 통해 변환된 데이터를 원래 상태로 되돌리는 데 사용된다. stacked_x에서 데이터는 values, 각 관찰이 나온 그룹은 ind 에 저장되어 있다. 따라서 'values ~ ind' 포뮬러를 사용한다.

```
> unstack(x_stacked, values ~ ind)
  a b c
1 1 5 4
2 2 3 5
3 3 2 7
```