R프로그래밍

7장. 데이터 다루기

박 혜 승 교 수

7장. 데이터 다루기

7.1 데이터 읽고 쓰기

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ 데이터를 저장하는 데 사용되는 모든 파일 형식 중에서 가장 널리 사용되는 것은 csv.
  ✓ 일반적인 csv 파일에서 첫 번째 행은 열의 헤더.
  ✓ 다음 줄부터 나오는 행들은 쉼표로 구분된 열이 있는 데이터 레코드.
  ✓ 다음은 이 형식으로 작성된 학생 기록의 예

persons.csv

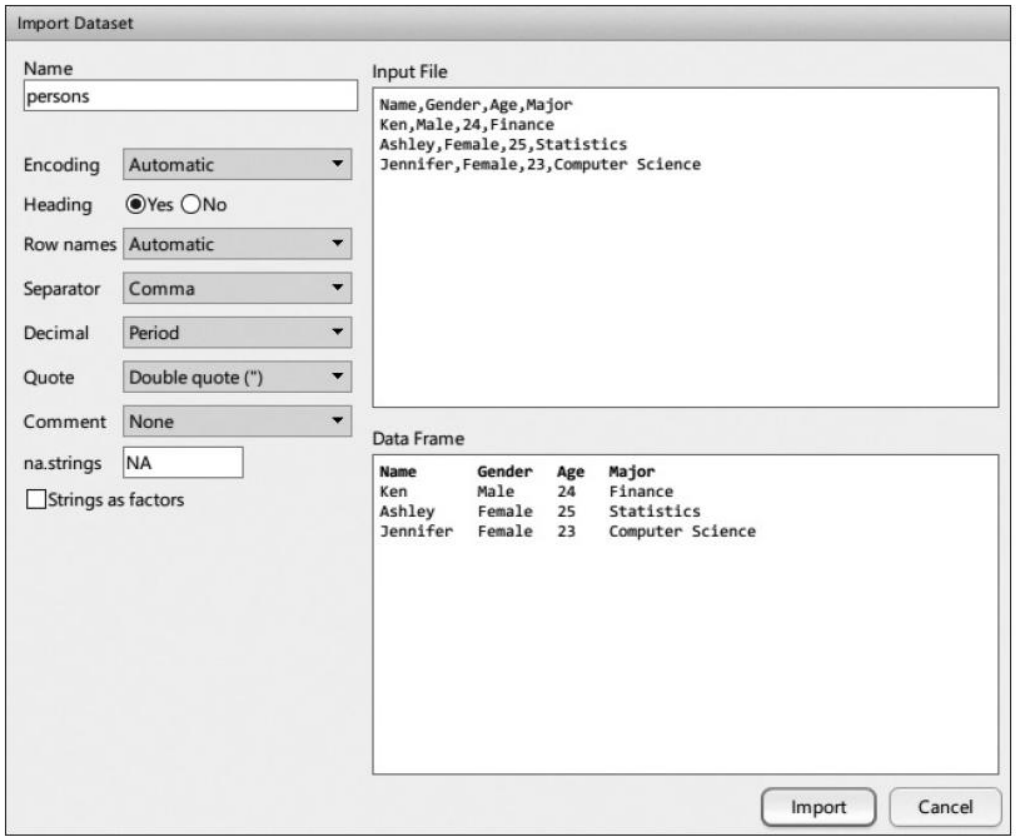
```
Name,Gender,Age,Major
Ken,Male,24,Finance
Ashley,Female,25,Statistics
Jennifer,Female,23,Computer Science
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

RStudio IDE로 데이터 가져오기

[그림 7-1] RStudio에서 데이터 가져오기



7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ 각 열의 문자열을 일부러 요소로 변환할 때는 Strings as factors를 체크.
  ✓ 파일 가져오기는 마술을 부리는 것이 아니라 파일 경로와 옵션을 R 코드로 변환하는 것.
  ✓ 데이터를 가져오는 인수들을 설정하고 Import를 누르면 read.csv() 함수를 호출.
  ✓ 이 대화 창을 이용하면 데이터를 편리하게 가져올 수 있으며, 처음 데이터 파일을 불러올 때 많이 하는 실수를 피하는 데 도움이 됨

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

내장 함수를 사용하여 데이터 가져오기

✓ 스크립트를 작성할 때마다 사용자가 파일 가져오기 대화 창을 사용하리라고 기대하기는 힘듦.

✓ 생성된 코드를 스크립트에 복사하여 스크립트를 실행할 때마다 자동으로 이 코드가 동작하게 할 수 있음.

✓ 내장 함수를 사용하여 데이터를 가져오는 방법은 매우 유용.

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ 데이터를 가져오는 가장 간단한 내장 함수는 readLines()
  ✓ 이 함수는 텍스트 파일을 읽어서 줄을 문자형 벡터로 반환.

```
> readLines("data/persons.csv")
[1] "Name,Gender,Age,Major"                "Ken,Male,24,Finance"
[3] "Ashley,Female,25,Statistics"          "Jennifer,Female,23,Computer Science"
```

  ✓ 기본적으로 파일에 있는 모든 행을 읽음.
  ✓ 처음 두 줄만 미리보려면 다음 코드를 실행.

```
> readLines("data/persons.csv", n = 2)
[1] "Name,Gender,Age,Major" "Ken,Male,24,Finance"
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

✓ 실제 데이터를 불러오는 많은 경우 readLines() 함수는 너무 단순.
  ❖ 행을 데이터 프레임으로 변환하기보다는 행을 문자열로 읽어 들이는 식으로 동작.

✓ 앞 코드처럼 CSV 파일에서 데이터를 가져오려면 read.csv() 함수를 직접 호출해야 함.

```
> persons1 <- read.csv("data/persons.csv", stringsAsFactors = FALSE)
> str(persons1)
'data.frame': 3 obs. of  4 variables:
 $ Name  : chr  "Ken" "Ashley" "Jennifer"
 $ Gender: chr  "Male" "Female" "Female"
 $ Age   : int  24 25 23
 $ Major : chr  "Finance" "Statistics" "Computer Science"
```

✓ 문자열 값을 그대로 유지하기 원한다면 문자열이 요소로 바뀌지 않도록 함수를 호출할 때 stringsAsFactors = FALSE를 설정.

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기
  - ✓ read.csv() 함수는 사용자가 원하는 대로 가져오기를 설정할 수 있게 다양하고 유용한 인수를 제공.
  - ✓ colClasses로는 명시적으로 열의 타입을 지정.
  - ✓ col.names로는 데이터 파일의 원래 열 이름을 바꿀 수 있음.

```
> persons2 <- read.csv("data/persons.csv", colClasses = c("character", "factor",
"integer", "character"),
+     col.names = c("name", "sex", "age", "major"))
> str(persons2)
'data.frame': 3 obs. of  4 variables:
 $ name : chr  "Ken" "Ashley" "Jennifer"
 $ sex  : Factor w/ 2 levels "Female","Male": 2 1 1
 $ age  : int  24 25 23
 $ major: chr  "Finance" "Statistics" "Computer Science"
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  - ✓ CSV는 특수한 구분 기호가 있는 데이터 형식.
  - ✓ 기술적으로 CSV 형식은 쉼표(,)를 사용하여 열을 구분하고, 새로운 줄로 행을 구분하는 데이터 형식.
  - ✓ 더 일반적으로는 쉼표 말고 다른 문자 역시도 열 구분 문자와 행 구분 문자가 될 수 있음.
  - ✓ 많은 데이터셋이 구분 문자가 탭인 형태로 저장.
  - ✓ 즉, 탭 문자를 사용하여 열을 구분.
  - ✓ 이때는 read.csv() 함수를 기본으로 구현한 좀 더 일반적인 버전인 read.table() 함수를 사용해 볼 수 있음.

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

readr 패키지를 사용하여 데이터 가져오기

✓ 지금까지 여러 가지 이유에서 read.* 함수들은 때로는 일관성이 없고, 어떤 상황에서는 그리 친절하지 않음.

✓ readr 패키지는 빠르고 일관성 있게 테이블 형식의 데이터를 가져오는 좋은 방법.

✓ 패키지를 설치하려면 install.packages("readr")을 실행.

✓ read_* 계열의 함수들을 사용하여 테이블 형식의 데이터를 가져옴.

```
> install.packages("readr")
> persons3 <- readr::read_csv("data/persons.csv")
Parsed with column specification:
cols(
  Name = col_character(),
  Gender = col_character(),
  Age = col_double(),
  Major = col_character()
)
```

```
> str(persons3)
Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 3 obs. of  4 variables:
 $ Name  : chr  "Ken" "Ashley" "Jennifer"
 $ Gender: chr  "Male" "Female" "Female"
 $ Age   : num  24 25 23
 $ Major : chr  "Finance" "Statistics" "Computer Science"
 - attr(*, "spec")=
  .. cols(
  ..   Name = col_character(),
  ..   Gender = col_character(),
  ..   Age = col_double(),
  ..   Major = col_character()
  .. )
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ library(readr)을 실행하여 read_csv를 직접 호출하는 대신 readr::read_csv로 함수를 사용.
  ✓ read_csv 함수와 내장 함수 read.csv가 조금 달라서 이 둘을 쉽게 혼동하기 때문.
  ✓ read_csv 함수의 기본 동작은 대부분의 상황을 처리할 수 있을 만큼 똑똑함.
  ✓ 내장 함수와 대조를 이루고자 불규칙한 형식의 데이터 파일(data/persons.txt)을 가져옴.

data/persons.txt

```
Name     Gender Age Major
Ken      Male   24  Finance
Ashley   Female 25  Statistics
Jennifer Female 23  Computer Science
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

✓ 파일 내용은 얼핏 보기에 상당히 표준에 맞고 테이블 형식인 것처럼 보이지만, 각 열 사이의 공백 개수는 행마다 동일하지 않으므로 sep = " "로 read.table() 함수를 사용할 수 없음.

```
> read.table("data/persons.txt", sep = " ")
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec,  :
  line 1 did not have 12 elements
```

✓ read.table()을 사용하여 데이터를 가져올 때, 제대로 동작하기 위해 올바른 인수를 파악하는 데 시간을 많이 허비할 수도 있음.

✓ readr 패키지의 read_table 함수 기본 동작이 충분히 똑똑하니까 동일한 입력으로도 시간을 절약하는 데 도움이 됨.

```
> readr::read_table("data/persons.txt")
Parsed with column specification:
cols(
  Name = col_character(),
  Gender = col_character(),
  Age = col_double(),
  Major = col_character()
)
```

```
# A tibble: 3 x 4
      Name     Gender      Age             Major
     <chr>      <chr>    <dbl>             <chr>
1      Ken       Male       24           Finance
2   Ashley     Female       25        Statistics
3 Jennifer     Female       23  Computer Science
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기
  - ✓ 이것이 바로 readr을 강력하게 추천하는 이유임
  - ✓ readr의 함수는 빠르고 똑똑하고 일관성이 있으며, 훨씬 쉽게 사용할 수 있는 내장 함수들의 기능을 지원함
  - ✓ 자세한 readr 패키지 내용은 https://github.com/tidyverse/readr을 참고

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

데이터 프레임 파일에 쓰기

✓ 데이터 분석의 일반적인 절차는 데이터 소스에서 데이터를 가져오고 변환.
✓ 적절한 도구와 모델을 적용하고, 마지막으로 의사 결정을 위해 저장할 새로운 데이터를 만드는 것.
✓ 파일에 데이터를 쓰는 인터페이스는 데이터를 읽는 인터페이스와 매우 유사.
✓ 우리는 `write.*` 함수를 사용하여 데이터 프레임을 파일로 내보냄.

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ 예를 들어 임의의 데이터 프레임을 만든 후 이것을 csv 파일에 저장 가능.

```
> some_data <- data.frame(
+     id = 1:4,
+     grade = c("A", "A", "B", NA),
+     width = c(1.51, 1.52, 1.46, NA),
+     check_date = as.Date(c("2016-03-05", "2016-03-06", "2016-03-10", "2016-03-11")))
> some_data
    id    grade    width    check_date
1    1        A     1.51        3/5/16
2    2        A     1.52        3/6/16
3    3        B     1.46       3/10/16
4    4     <NA>       NA       3/11/16
> write.csv(some_data, "data/some_data.csv")
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

  ✓ csv 파일에 누락된 값과 날짜가 올바르게 유지되는지 확인하려고 원본 텍스트로 출력 파일을 읽어 올 수 있음.

```
> cat(readLines("data/some_data.csv"), sep = "\n")
"","id","grade","width","check_date"
"1",1,"A",1.51,2016-03-05
"2",2,"A",1.52,2016-03-06
"3",3,"B",1.46,2016-03-10
"4",4,NA,NA,2016-03-11
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

✓ 데이터는 일치하더라도 때로는 데이터를 저장하는 데 다른 표준을 적용할 수도 있음.
- ❖ write.csv() 함수를 사용하여 파일에 쓰는 동작을 수정할 수 있음.
- ❖ 앞 결과에 불필요한 부분이 있다고 생각할 수 있음.
- ❖ 예를 들어 id가 이미 있기 때문에 행 이름이 약간 중복되어 보임.
- ❖ 이때는 행 이름을 파일로 내보내길 원하지 않음.
- ❖ 문자열 값을 따옴표로 묶을 필요가 없음.
- ❖ 누락된 값을 NA 대신에 -라고 표시하길 원함.

✓ 다음 코드를 실행하여 우리가 원하는 동작과 표준을 사용해서 동일한 데이터 프레임을 파일로 내보낼 수 있음.

```
> write.csv(some_data, "data/some_data.csv", quote = FALSE, na = "-", row.names = FALSE)
```

```
> cat(readLines("data/some_data.csv"), sep = "\n")
id,grade,width,check_date
1,A,1.51,2016-03-05
2,A,1.52,2016-03-06
3,B,1.46,2016-03-10
4,-,-,2016-03-11
```

7.1 데이터 읽고 쓰기

- 파일에서 텍스트 데이터 읽고 쓰기

✓ readr::read_csv() 함수를 사용하여 다음과 같이 실측값과 날짜 열이 있는 csv 파일을 가져올 수 있음.

```
> readr::read_csv("data/some_data.csv", na = "-")
Parsed with column specification:
cols(
  id = col_double(),
  grade = col_character(),
  width = col_double(),
  check_date = col_date(format = "")
)
```

```
# A tibble: 4 x 4
     id grade   width check_date
  <dbl> <chr>   <dbl>      <date>
1     1     A    1.51 2016-03-05
2     2     A    1.52 2016-03-06
3     3     B    1.46 2016-03-10
4     4    NA      NA 2016-03-11
```

✓ - 표시가 실측값으로 올바르게 변환되고 날짜 열이 날짜 객체로 올바르게 인식.

✓ 다음과 같이 해당 파일을 삭제할 수 있음.

```
> file.remove("data/some_data.csv")
[1] TRUE
```

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  ✓ csv 같은 텍스트 형식의 데이터를 사용하는 장점은 소프트웨어의 중립성.

  ❖ 특정 소프트웨어를 사용하여 데이터를 읽지 않아도 되며, 파일을 사람이 직접 읽을 수 있다는 장점.

  ✓ 내용이 순수 텍스트이므로 편집기에 표시된 데이터에서 바로 계산을 수행할 수 없는 단점.

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  ✓ 테이블 형태의 데이터를 저장하는 다른 보편적인 형식은 엑셀(Excel) 통합 문서.
  ✓ 엑셀 통합 문서에는 하나 이상의 워크시트가 존재.
  ✓ 각 워크시트는 테이블을 만들기 위해 텍스트나 임의의 값을 채울 수 있는 일종의 격자판.
  ✓ 이것을 사용하면 테이블 내에서 테이블 간에, 워크시트 간에 손쉽게 계산 수행 가능.
  ✓ 마이크로소프트 엑셀은 강력한 소프트웨어이지만 데이터 형식(엑셀 97~2003 버전은 .xls, 엑셀 2007 이후 버전은 .xlsx)은 사람이 직접 읽을 수 없음.

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  ✓ 예를 들어 data/prices.xlsx 파일은 다음과 같이 생긴 간단한 엑셀 통합 문서임

[그림 7-2] prices.xlsx 파일

	A	B	C	D
1	Date	Price	Growth	
2	3/1/16	85		
3	3/2/16	88	3.5%	
4	3/3/16	84	-4.5%	
5	3/4/16	81	-3.6%	
6	3/5/16	83	2.5%	
7	3/6/16	87	4.8%	
8				

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  ✓ 엑셀 통합 문서를 읽을 수 있게 제공되는 내장 함수는 없음.
  ✓ 다만 이를 위해 설계된 여러 가지 R패키지 존재.
  ✓ 가장 간단한 것은 readxl(https://github.com/tidyverse/readxl)
  ✓ 엑셀 통합 문서의 워크시트에 저장된 표를 훨씬 쉽게 추출 가능.

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  ✓ CRAN에서 패키지를 설치하려면 install.packages("readxl")을 사용.

	A	B	C	D
1	Date	Price	Growth	
2	3/1/16	85		
3	3/2/16	88	3.5%	
4	3/3/16	84	-4.5%	
5	3/4/16	81	-3.6%	
6	3/5/16	83	2.5%	
7	3/6/16	87	4.8%	
8				

```
> readxl::read_excel("data/prices.xlsx")
# A tibble: 6 x 3
                  Date Price  Growth
                 <dttm> <dbl>   <dbl>
1 2016-03-01 00:00:00    85      NA
2 2016-03-02 00:00:00    88   0.035
3 2016-03-03 00:00:00    84  -0.045
4 2016-03-04 00:00:00    81  -0.036
5 2016-03-05 00:00:00    83   0.025
6 2016-03-06 00:00:00    87   0.048
```

  ✓ 앞의 데이터 프레임에서 보듯이 readxl::read_excel() 함수는 엑셀 날짜를 R의 날짜로 자동 변환.

  ✓ Growth 열에 있던 실측값 역시 올바르게 보존.

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

✓ 엑셀 통합 문서를 작업할 수 있는 또 다른 패키지는 openxlsx.
✓ 이 패키지는 readr이 설계된 것보다 더 포괄적으로 XLSX 파일을 읽고 쓰고 편집 가능.
✓ 패키지를 설치하려면 install.packages("openxlsx")를 실행.
✓ openxlsx에서는 read.xlsx() 함수를 호출하여 readr::read_excel()처럼 지정된 통합 문서의 데이터를 데이터 프레임으로 읽을 수 있음.

```
> openxlsx::read.xlsx("data/prices.xlsx", detectDates = TRUE)
   Date Price Growth
1 42430    85     NA
2 42431    88  0.035
3 42432    84 -0.045
4 42433    81 -0.036
5 42434    83  0.025
6 42435    87  0.048
```

✓ 날짜 값을 정확히 가져오려면 detectDates = TRUE를 지정.
  ❖ 그렇지 않으면 날짜는 숫자로 인식됨.
✓ openxlsx는 데이터를 읽는 것 외에도 데이터 프레임을 엑셀 문서로 만들 수 있음.

```
> openxlsx::write.xlsx(mtcars, "data/mtcars.xlsx")
```

7.1 데이터 읽고 쓰기

- 엑셀 워크시트 읽기와 쓰기

  - ✓ 엑셀 통합 문서를 작업할 수 있게 설계된 또 다른 패키지가 있음
  - ✓ XLConnect(http://cran.r-project.org/web/packages/XLConnect/)는 또 다른 엑셀 커넥터.
  - ✓ 모든 플랫폼에서 사용 가능.
  - ✓ 마이크로소프트의 엑셀 프로그램이 아닌 자바 런타임 환경(JRE)이 기존 설치된 것에 의존.
  - ✓ RODBC(http://cran.r-project.org/web/packages/RODBC/)는 윈도에 설치된 ODBC 드라이버를 사용하여 액세스 데이터베이스와 엑셀 통합 문서를 연결할 수 있는 일반적인 데이터베이스 커넥터.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  - ✓ 이전 절에서는 CSV 파일이나 엑셀 통합 문서를 읽고 쓰는 기능을 소개.
  - ✓ 이들은 R 입장에서 본연 그대로가 아닌, 즉 네이티브하지 않은 데이터 형식.
  - ✓ 즉, 원래 데이터 객체와 출력된 파일 사이에 차이가 있음.
  - ✓ 예를 들어 서로 다른 타입의 열이 많은 데이터 프레임을 CSV 파일로 내보낼 때는 열 타입 정보가 사라짐.
  - ✓ 열이 수치형인지, 문자형인지, 날짜인지 여부와 상관없이 항상 텍스트 형식으로 표시.
  - ✓ 사람은 출력된 파일에서 데이터를 직접 쉽게 읽을 수 있지만, 컴퓨터는 각 열의 타입을 유추하는 방법에 의존할 수밖에 없음.
  - ✓ 다시 말해 쓰기 과정에서 이동성(즉, 다른 소프트웨어에서도 이 파일을 읽을 수 있게 됨)의 대가로 열 타입을 잃어버리게 되어 CSV 형식의 데이터를 원래 데이터 프레임과 똑같은 데이터 프레임으로 복구하는 것이 어려울 수 있음.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  ✓ 이동성을 고려하지 않고 R만 사용하여 데이터를 처리할 때는 네이티브 형식으로 데이터를 읽고 쓸 수 있음.

  ✓ 더 이상 일반적인 텍스트 편집기나 다른 소프트웨어로 데이터를 읽을 수는 없지만, 단일 객체 또는 전체 환경을 매우 효율적으로 데이터 손실 없이 쉽게 읽고 쓸 수 있음.

  ✓ 즉, 네이티브 형식을 사용하여 객체를 파일로 저장하면 열의 결측 값, 타입이나 클래스, 속성 같은 문제를 걱정하지 않아도 정확히 동일한 데이터를 그대로 복구할 수 있음.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

네이티브 형식의 단일 객체 읽기와 쓰기

✓ 네이티브 데이터 형식과 관련한 작업에는 크게 두 가지 그룹의 함수가 존재.
✓ 한 그룹은 단일 객체를 RDS 파일에 쓰거나 RDS 파일에서 단일 객체를 읽도록 설계.
✓ RDS 파일은 단일 R 객체를 직렬화하여 저장하는 파일 형식.
✓ 또 다른 그룹은 R 객체를 여러 개 다루는데, 다음 절에서 자세히 설명 예정.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

네이티브 형식의 단일 객체 읽기와 쓰기

✓ 다음 코드에서는 some_data 객체를 RDS 파일에 쓰고, 해당 파일을 다시 읽어 데이터 프레임 2개가 정확히 동일한지 확인.

✓ 먼저 saveRDS() 함수를 사용하여 some_data 객체를 data/some_data.rds에 저장.

```
> saveRDS(some_data, "data/some_data.rds")
```

✓ 이후 readRDS() 함수를 통해 동일한 파일에서 데이터를 읽고, 그 결과 데이터 프레임을 some_data2에 저장.

```
> some_data2 <- readRDS("data/some_data.rds")
```

✓ 마지막으로 identical() 함수를 사용하여 두 데이터 프레임이 정확히 일치하는지 확인.

```
> identical(some_data, some_data2)
[1] TRUE
```

✓ 예상한 대로 두 데이터 프레임이 정확히 일치하는 것을 볼 수 있음.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

✓ 네이티브 형식에는 두 가지 주목할 만한 장점이 존재.
  - ❖ 공간 효율성
  - ❖ 시간 효율성

✓ 다음 코드에서는 난수 20만 행으로 구성된 대형 데이터 프레임을 생성.

✓ 이후 이 데이터 프레임을 CSV와 RDS 파일로 각각 저장하는 프로세스를 수행.

```
> rows <- 200000
> large_data <- data.frame(id = 1:rows, x = rnorm(rows), y = rnorm(rows))
> system.time(write.csv(large_data, "data/large_data.csv"))
   사용자   시스템  elapsed
   0.568    0.041    0.629

> system.time(saveRDS(large_data, "data/large_data.rds"))
   사용자   시스템  elapsed
   0.255    0.010    0.270
```

✓ saveRDS() 함수가 write.csv() 함수보다 쓰기 측면에서 효율성이 훨씬 우수.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  ✓ file.info() 함수를 사용하여 두 출력 파일의 크기를 비교해 보자.

```
> fileinfo <- file.info("data/large_data.csv", "data/large_data.rds")
> fileinfo[, "size", drop = FALSE]
                          size
data/large_data.csv 10241085
data/large_data.rds  3498456
```

  ✓ 파일 크기에 큰 차이가 있는 것을 볼 수 있다. CSV가 RDS 파일에 비해서 거의 3배 정도 큼.

  ✓ 네이티브 형식이 저장 공간의 효율성 면에서 더 뛰어나다는 것을 보여줌.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

✓ 마지막으로 CSV와 RDS 파일을 읽을 때의 성능을 살펴보자.

✓ CSV 파일을 읽을 때는 내장 함수인 read.csv() 함수와 readr 패키지에서 제공하는 좀 더 빠른 read_csv() 함수를 사용.

```
> system.time(read.csv("data/large_data.csv"))
   사용자   시스템 elapsed
     1.09    0.00    1.09
> system.time(readr::read_csv("data/large_data.csv"))
   사용자   시스템 elapsed
     0.15    0.09    0.25
```

✓ read_csv() 함수가 내장 함수인 read.csv에 비해서 최소 8배 이상 빠름.

✓ 하지만 네이티브 형식에 비하면 이 CSV 함수들은 비교가 안됨.

```
> system.time(readRDS("data/large_data.rds"))
   사용자   시스템 elapsed
     0.03    0.00    0.03
```

✓ 네이티브 형식은 쓰기 측면에서도 훨씬 더 효율적임.

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  ✓ saveRDS나 readRDS 함수는 데이터 프레임뿐만 아니라 모든 R 객체에 대해 동작.

  ✓ 예를 들어 실측값을 포함하는 수치형 벡터와 중첩 구조가 있는 리스트 객체를 만들고, 해당 객체를 별도의 RDS 파일에 저장하자.

```
> nums <- c(1.5, 2.5, NA, 3)
> list1 <- list(x = c(1, 2, 3),
+     y = list(a = c("a", "b"),
+     b = c(NA, 1, 2.5)))
> saveRDS(nums, "data/nums.rds")
> saveRDS(list1, "data/list1.rds")
```

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  ✓ 이제 이 RDS 파일들을 다시 읽어 원래 객체들이 그대로 복원되는지 살펴보자

```
> readRDS("data/nums.rds")
[1] 1.5 2.5  NA 3.0
> readRDS("data/list1.rds")
$x
[1] 1 2 3

$y
$y$a
[1] "a" "b"

$y$b
[1] NA 1.0 2.5
```

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

작업 환경 저장하고 복원하기

✓ RDS 형식은 단일 R 객체를 저장하는 데 사용.

✓ RData 형식은 여러 R 객체를 저장하는 데 사용.

✓ save() 함수를 호출하여 some_data, nums, list1을 단일 RData 파일에 저장.

```
> save(some_data, nums, list1, file = "data/bundle1.RData")
```

✓ 객체 3개를 잘 저장하고 다시 복구할 수 있는지 확인하려면, 먼저 이 객체들을 제거하고 load() 함수를 호출하여 파일에서 다시 복구.

```
> rm(some_data, nums, list1)
> load("data/bundle1.RData")
```

7.1 데이터 읽고 쓰기

- 네이티브 데이터 파일 읽기와 쓰기

  ✓ 세 객체가 모두 완전히 복구된 것을 확인할 수 있음.

```
> some_data
    id    grade    width    check_date
1    1        A     1.51    2016-03-05
2    2        A     1.52    2016-03-06
3    3        B     1.46    2016-03-10
1    4     <NA>       NA    2016-03-11
> nums
[1] 1.5 2.5  NA 3.0
> list1
$x
[1] 1 2 3

$y
$y$a
[1] "a" "b"

$y$b
[1] NA 1.0 2.5
```

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  - ✓ R에서는 이미 기본 데이터셋을 여러 개 제공하고, 주로 예제와 테스트 목적으로 쉽게 가져와 사용 가능.
  - ✓ 기본으로 제공하는 데이터셋은 대부분 데이터 프레임이며, 자세한 세부 사항과 함께 제공.
  - ✓ 예를 들어 iris와 mtcars는 아마도 R에서 가장 유명한 데이터셋.
  - ✓ ?iris와 ?mtcars라고 입력하면 각 데이터셋에 관한 자세한 설명을 읽을 수 있음.
    - ❖ 일반적으로 데이터 설명은 매우 구체적.
    - ❖ 데이터에 있는 내용, 수집 및 형식 지정 방법, 각 열의 의미를 알려 줄 뿐만 아니라 관련 출처와 참고 문서도 제공함.
  - ✓ 일단 R이 준비되면 이러한 데이터셋을 즉시 사용할 수 있기 때문에 아주 편하게 내장 데이터셋을 활용하여 데이터 분석 도구 테스트 가능.
  - ✓ 예를 들어 어딘가에서 명시적으로 데이터를 불러오지 않고 iris나 mtcars를 직접 사용 가능.

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  ✓ 다음 코드는 iris 데이터의 첫 여섯 행을 출력한 결과.

```
> 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          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         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  ✓ 다음 코드는 이 데이터셋의 구조를 보여줌

```
> 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 ...
```

  ✓ iris를 콘솔 창에서 출력하여 전체 데이터 프레임을 보거나 View(iris)를 사용하여 미리보기 창에서 데이터 확인 가능.

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  ✓ 이제 mtcars의 첫 여섯 행을 보고 그 구조를 확인해보자.

```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.8  2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

```
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  - ✓ iris와 mtcars는 작고 단순.
  - ✓ 실제로 대부분의 기본 데이터셋은 행 수십 개 또는 수백 개와 열 몇 개로 구성.
  - ✓ 종종 특정 데이터 분석 도구의 사용법을 보여 주려고 이들을 사용.
  - ✓ 더 큰 데이터를 테스트하고 싶다면 데이터셋을 함께 제공하는 일부 R 패키지 사용 가능.
  - ✓ 예를 들어 가장 유명한 데이터 시각화 패키지인 ggplot2는 수많은 다이아몬드의 가격과 기타 속성을 포함하는 diamonds라는 데이터셋을 제공.
  - ✓ 이 데이터의 세부 사항을 더 알고 싶다면 ?ggplot2::diamonds를 입력.
  - ✓ 패키지를 설치하지 않았다면 install.package("ggplot2")를 실행하여 설치.

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  ✓ 패키지의 데이터를 불러오려면 data() 함수를 사용.

```
> data("diamonds", package = "ggplot2")
> dim(diamonds)
[1] 53940    10
```

  ✓ 이 결과는 diamonds 객체에 행 53,940개와 열 10개가 있다는 것을 보여줌.

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

✓ 다음 코드는 데이터 일부를 미리보기 한 결과.

```
> head(diamonds)
# A tibble: 6 x 10
  carat       cut   color clarity depth table price     x     y     z
  <dbl>     <ord>   <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23     Ideal       E     SI2  61.5  55.0   326  3.95  3.98  2.43
2  0.21   Premium       E     SI1  59.8  61.0   326  3.89  3.84  2.31
3  0.23      Good       E     VS1  56.9  65.0   327  4.05  4.07  2.31
4  0.29   Premium       I     VS2  62.4  58.0   334  4.2   4.23  2.63
5  0.31      Good       J     SI2  63.3  58.0   335  4.34  4.35  2.75
6  0.24 Very Good       J    VVS2  62.8  57.0   336  3.94  3.96  2.48
```

7.1 데이터 읽고 쓰기

- 내장 데이터셋 가져오기

  ✓ 유용한 기능을 제공하는 패키지 외에 데이터셋만 제공하는 패키지도 존재.
  ✓ 예를 들어 nycflights13과 babynames 패키지는 각각 여러 데이터셋만 포함.
  ✓ 데이터를 가져오는 방법은 이전 예제와 완전히 동일.
  ✓ 두 패키지를 설치하려면 install.packages(c("nycflights13", "babynames"))를 실행.

7.2 데이터 시각화하기

7.2 데이터 시각화하기

- 데이터 시각화하기

  ✓ 모델을 선택한 뒤에 선택한 모델이 데이터 적용에 적합한지에 대한 검사를 수행하는 첫 번째 방법은 데이터의 경계나 패턴을 보고 시각적으로 검사하는 것.

  ❖ 즉, 먼저 데이터를 시각화해야 한다는 것.

  ✓ 이 절에서는 주어진 데이터셋을 시각화하는 기본적인 그래픽 함수를 학습.

  ✓ nycflights13과 babynames 패키지에 있는 데이터셋을 사용할 것.

  ✓ 아직 설치하지 않았다면 다음 코드를 실행.

```
> install.packages(c("nycflights13", "babynames"))
```

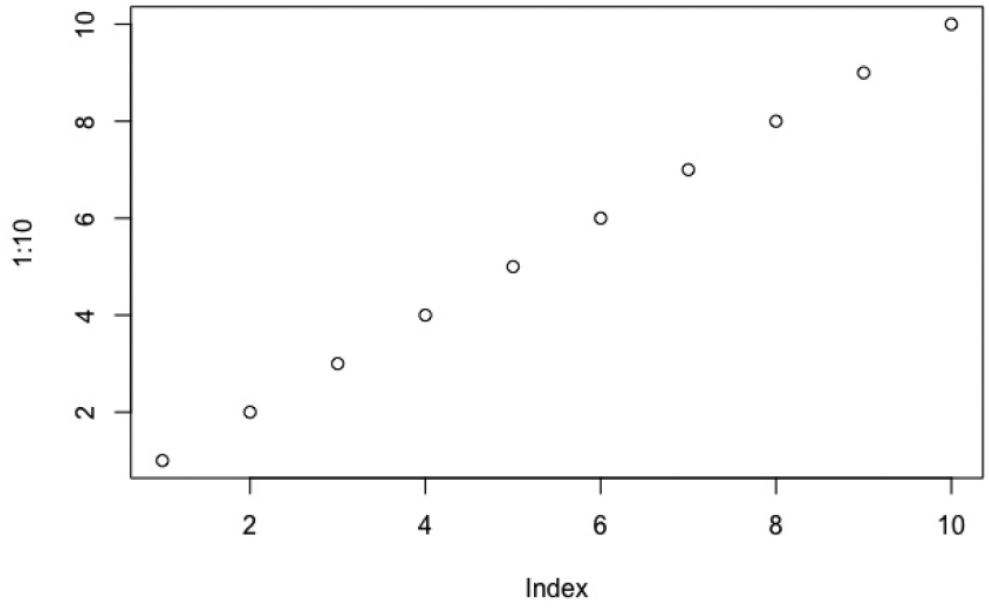
7.2 데이터 시각화하기

- 산점도 만들기

  ✓ R에서 데이터를 시각화하는 기본 함수는 plot()
  ✓ 단순히 숫자나 정수 벡터를 plot() 함수에 입력하면 인덱스에 따른 값의 산점도를 생성.
  ✓ 예를 들어 다음 코드는 순서대로 증가하는 점 10개로 된 산점도를 생성.

```
> plot(1:10)
```

[그림 7-3] 1:10을 입력으로 한 산점도



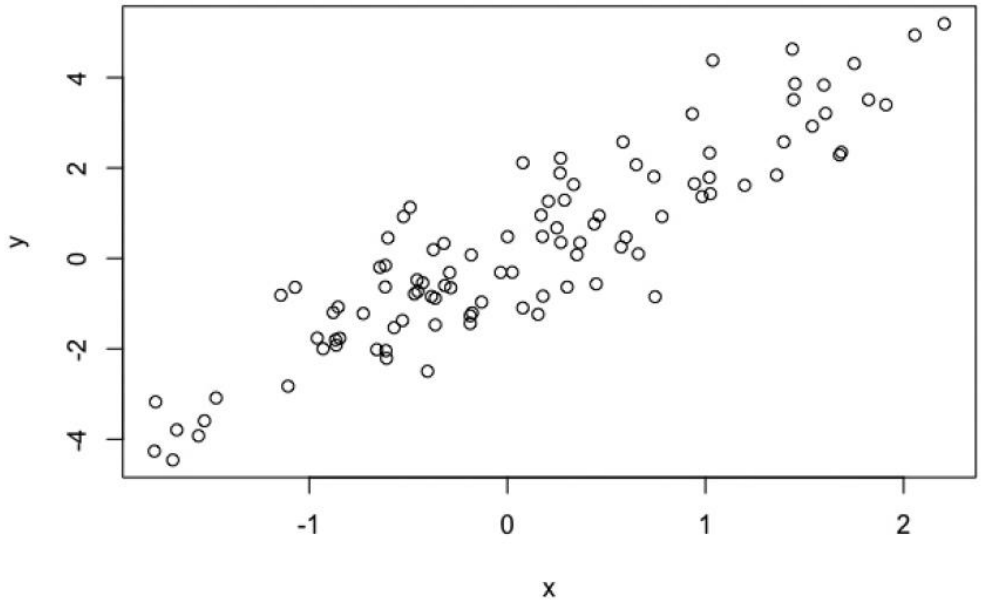
7.2 데이터 시각화하기

- 산점도 만들기

✓ 선형적 상관관계가 둘 있는 임의의 숫자 벡터를 생성하여 좀 더 현실적인 산점도 생성 가능.

```
> x <- rnorm(100)
> y <- 2 * x + rnorm(100)
> plot(x, y)
```

[그림 7-4] 난수 발생을 이용한 산점도



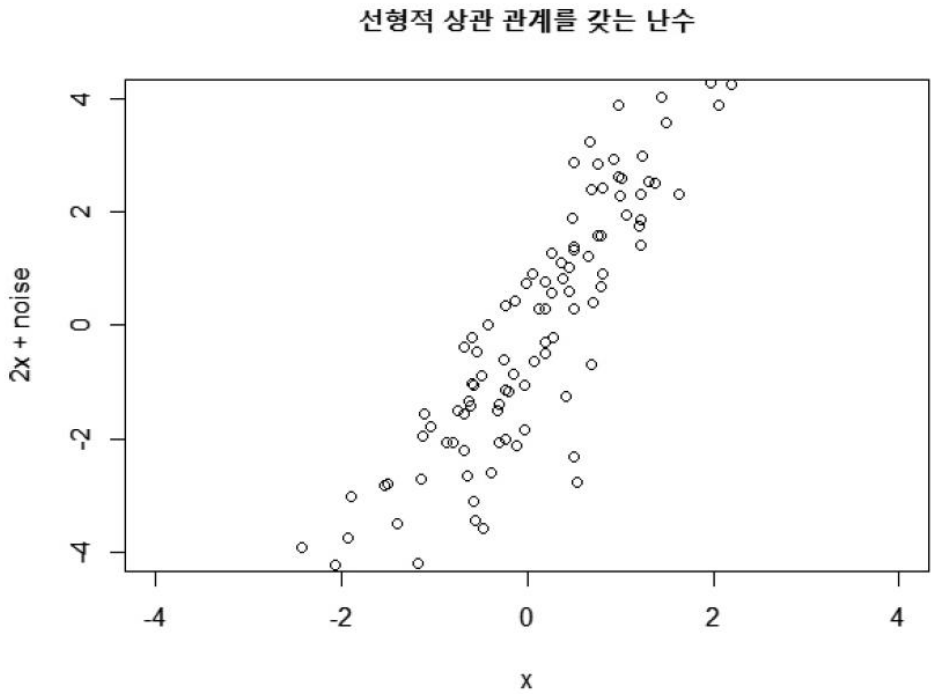
7.2 데이터 시각화하기

- 산점도 만들기

차트 요소 사용자 정의하기

✓ 플롯에서 사용자 지정이 가능한 가장 일반적인 요소들은 제목(main 인수나 title() 함수), x축 레이블(xlab), y축 레이블(ylab), x축 범위(xlim), y축 범위(ylim)

```
> plot(x, y,
+      main = "선형적 상관 관계를 갖는 난수",
+      xlab = "x", ylab = "2x + noise",
+      xlim = c(-4, 4), ylim = c(-4, 4))
```



7.2 데이터 시각화하기

- 산점도 만들기

  ✓ 차트 제목은 main 인수 혹은 별도의 title() 함수를 호출하여 지정 가능.
  ✓ 앞 코드는 다음 코드와 같이 작성 가능.

```
> plot(x, y,
+      xlim = c(-4, 4), ylim = c(-4, 4),
+      xlab = "x", ylab = "2x + noise")
> title("선형적 상관 관계를 갖는 난수")
```

7.2 데이터 시각화하기

- 산점도 만들기

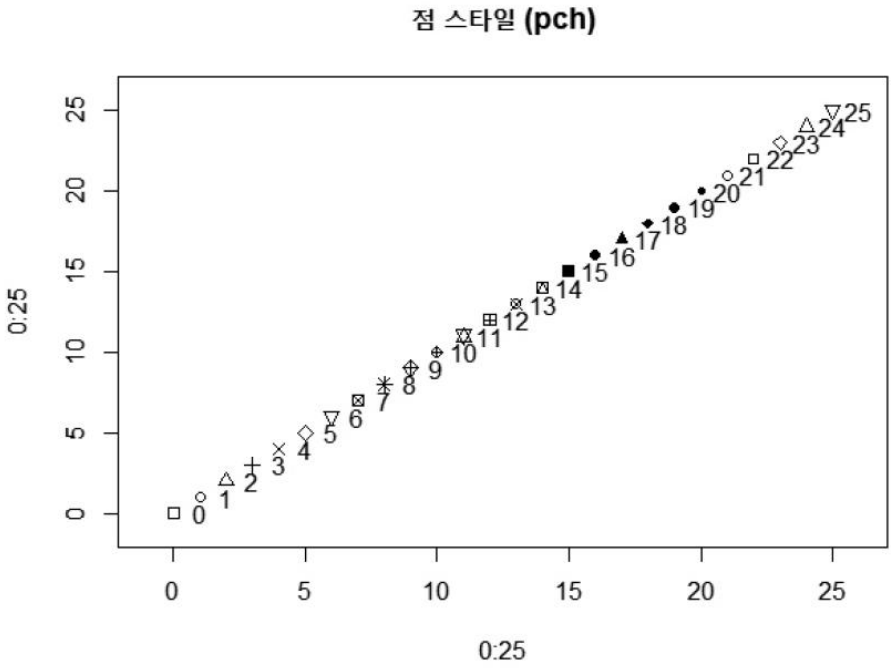
점 스타일 사용자 정의하기

✓ 산점도의 점 스타일은 기본적으로 원.

✓ pch 인수(플로팅 문자)를 지정하면 점 스타일 변경 가능.

✓ 26가지 점 스타일 사용 가능.

```
> plot(0:25, 0:25, pch = 0:25,
+      xlim = c(-1, 26), ylim = c(-1, 26),
+      main = "점 스타일 (pch)")
> text(0:25+1, 0:25, 0:25)
```

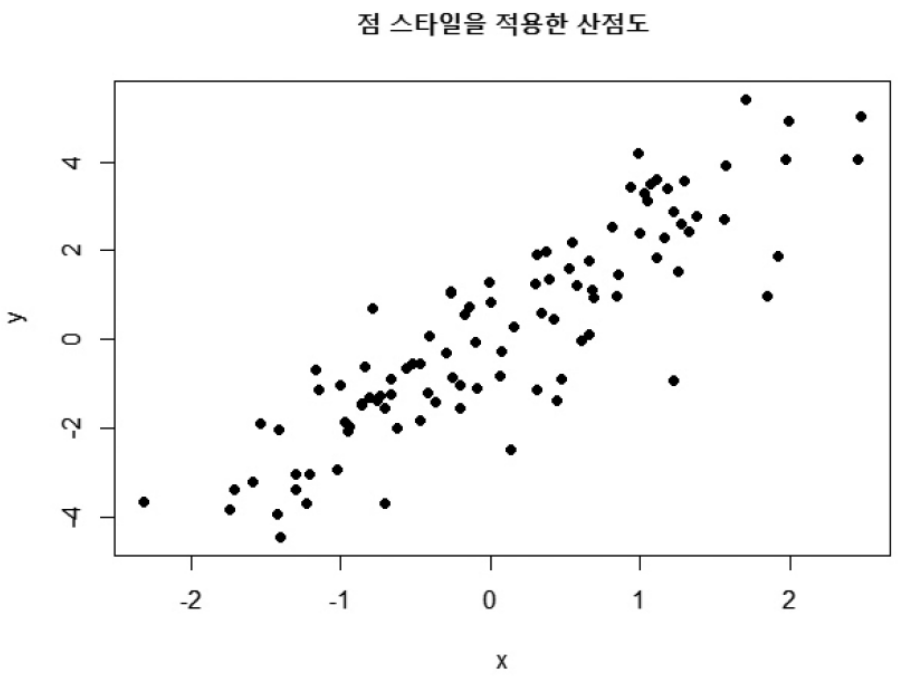


7.2 데이터 시각화하기

- 산점도 만들기

✓ 다른 내장 함수들과 마찬가지로, plot() 함수는 pch를 비롯한 다른 인수와 관련하여 벡터화.
✓ 산점도의 각 점 스타일을 사용자가 정의 가능.
✓ 예를 들어 가장 간단하게는 pch = 16을 설정하여 모든 점에 기본 설정과 다른 한 가지 점 스타일을 사용해보자.

```
> x <- rnorm(100)
> y <- 2 * x + rnorm(100)
> plot(x, y, pch = 16,
+      main = "점 스타일을 적용한 산점도")
```

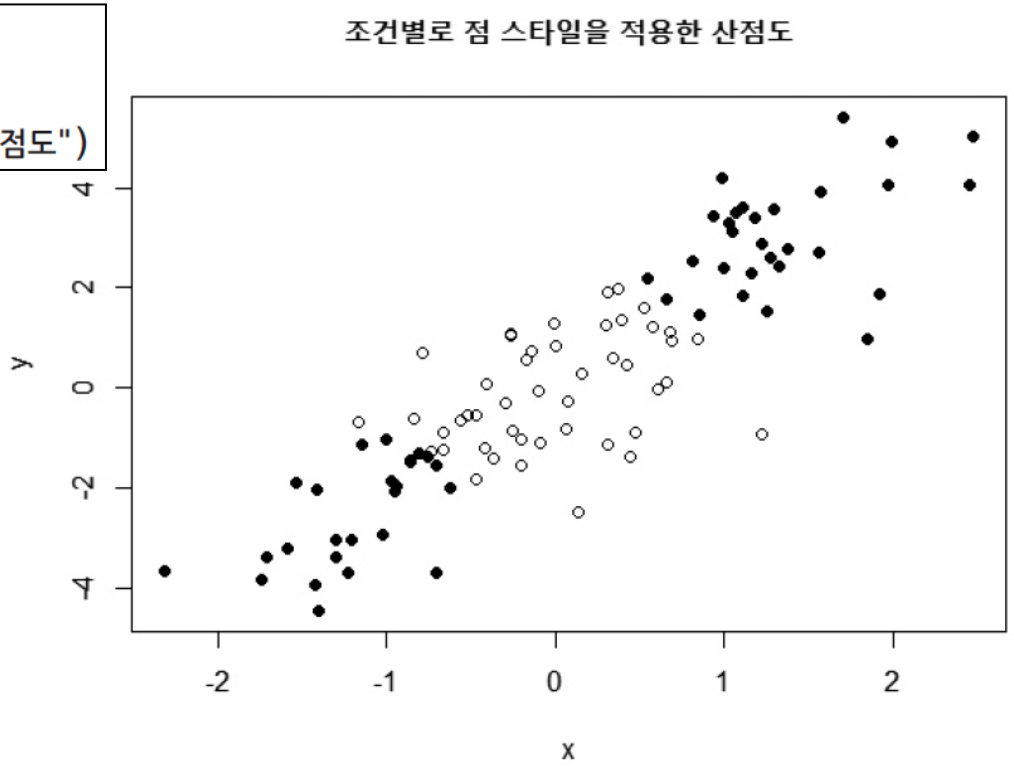


7.2 데이터 시각화하기

- 산점도 만들기

✓ 조건에 따라 두 그룹의 점을 구별해야 하는 경우가 존재.
✓ ifelse() 함수를 통해 점들이 조건을 만족하는지 검사해서 각 관측치의 점 스타일 지정 가능.
✓ 다음 코드에서는 x * y > 1을 만족하는 점에 pch = 16을 적용하고, 그렇지 않으면 pch = 1을 적용.

```
> plot(x, y,
+      pch = ifelse(x * y > 1, 16, 1),
+      main = "조건별로 점 스타일을 적용한 산점도")
```

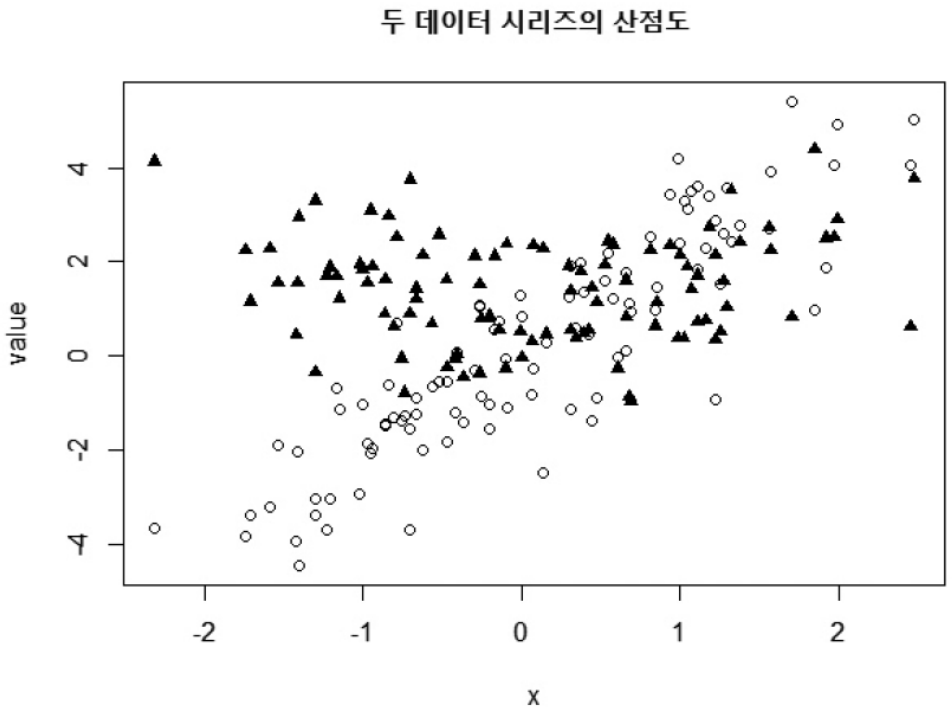


7.2 데이터 시각화하기

- 산점도 만들기

✓ plot()과 points() 함수를 사용하여 동일한 x축을 공유하는 서로 다른 데이터셋 2개를 한곳에 시각화할 수도 있음.
✓ 앞 예제에서 우리는 정규 분포를 갖는 랜덤 벡터 x와 선형 관계를 갖는 랜덤 벡터 y를 생성.
✓ 이제 x와 비선형 관계를 갖는 다른 임의의 z 벡터를 생성해보자.
✓ x에 대해 y와 z를 서로 다른 점 스타일로 적용하여 이 모두를 그래프로 출력.

```
> z <- sqrt(1 + x ^ 2) + rnorm(100)
> plot(x, y, pch = 1,
+      xlim = range(x), ylim = range(y, z),
+      xlab = "x", ylab = "value")
> points(x, z, pch = 17)
> title("두 데이터 시리즈의 산점도")
```



7.2 데이터 시각화하기

- 산점도 만들기

  ✓ z를 생성한 후 먼저 x와 y의 그래프를 그리고, 다른 pch 설정으로 z점들을 추가.
  ✓ ylim = range(y, z)를 지정하지 않으면 그래프는 y의 범위만 고려.
  ❖ 결국 y축은 z의 범위보다 좁은 범위를 가질 수 있음.
  ✓ points()는 plot() 함수가 생성한 그래프의 축 범위를 자동으로 조정해 주지 않음.
  ❖ 축 범위를 벗어나는 모든 점은 그래프에서 사라짐.
  ✓ 앞 코드에서는 y축 범위를 적절히 설정하여 플롯에 y와 z의 모든 점을 표시 가능.

7.2 데이터 시각화하기

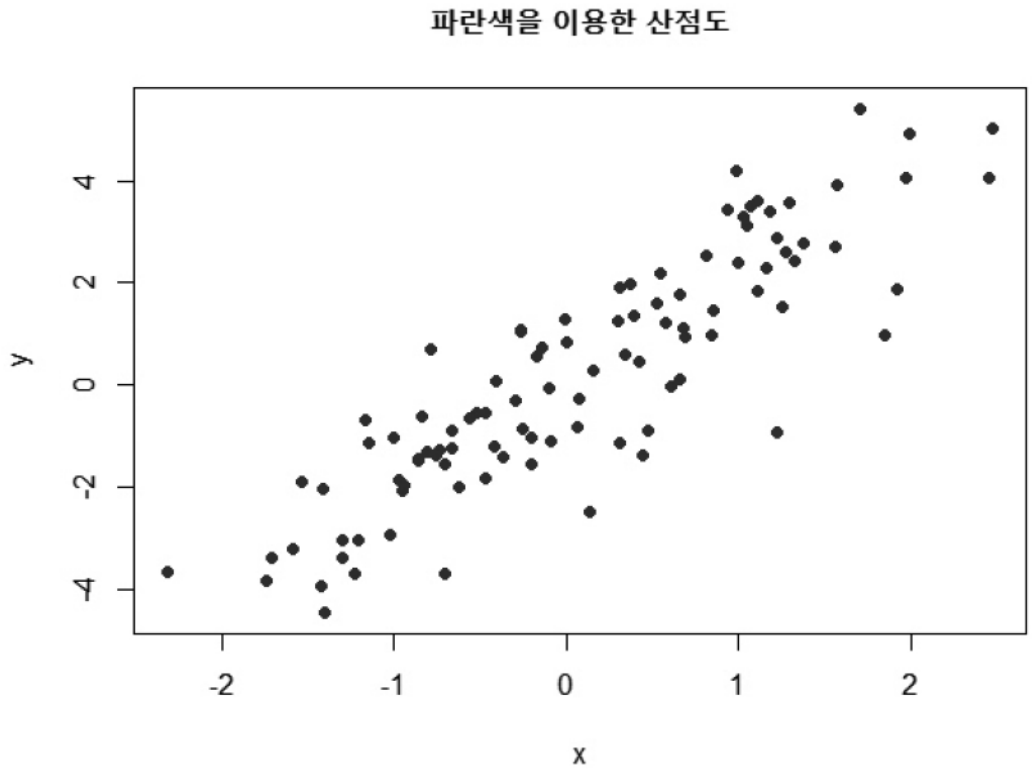
- 산점도 만들기

포인트 색상 사용자 정의하기

✓ 이 그래픽은 그레이 스케일(흑백) 이미지를 출력하는 것에 제한되지 않음.

✓ plot()의 col 인수를 사용하면 다른 색상 적용 가능.

```
> plot(x, y, pch = 16, col = "blue",
+      main = "파란색을 이용한 산점도")
```



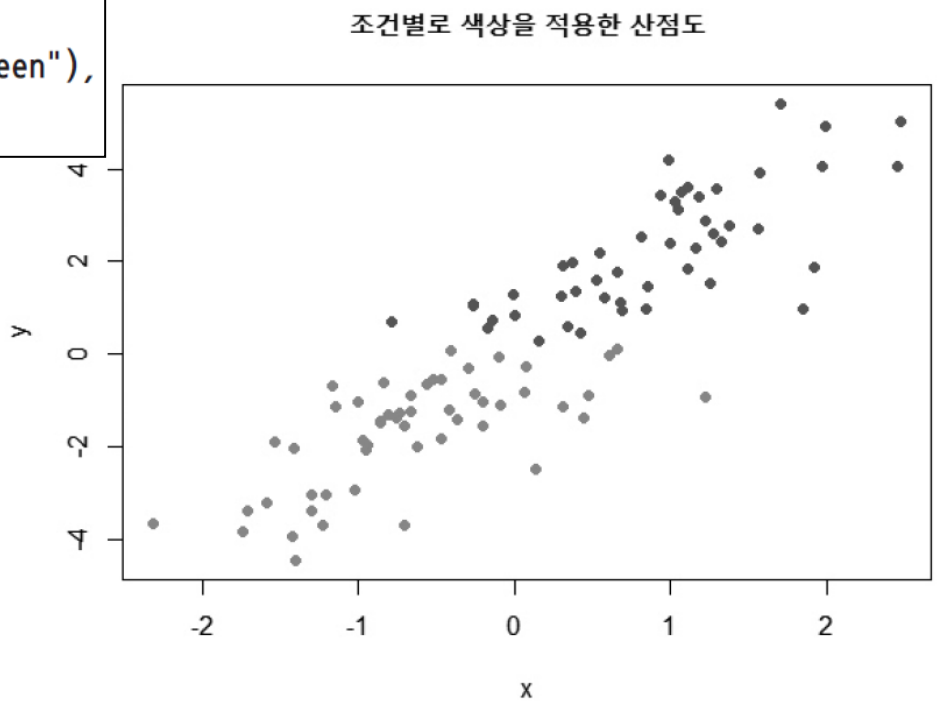
7.2 데이터 시각화하기

- 산점도 만들기

  ✓ pch와 마찬가지로 col도 벡터화된 인수.

  ✓ 동일한 방법으로 특정 조건을 만족하는지 여부에 따라 서로 다른 두 가지 범주로 점들을 구분하고자 다른 색상 적용 가능.

```
> plot(x, y, pch = 16,
+      col = ifelse(y >= mean(y), "red", "green"),
+      main = "조건별로 색상을 적용한 산점도")
```

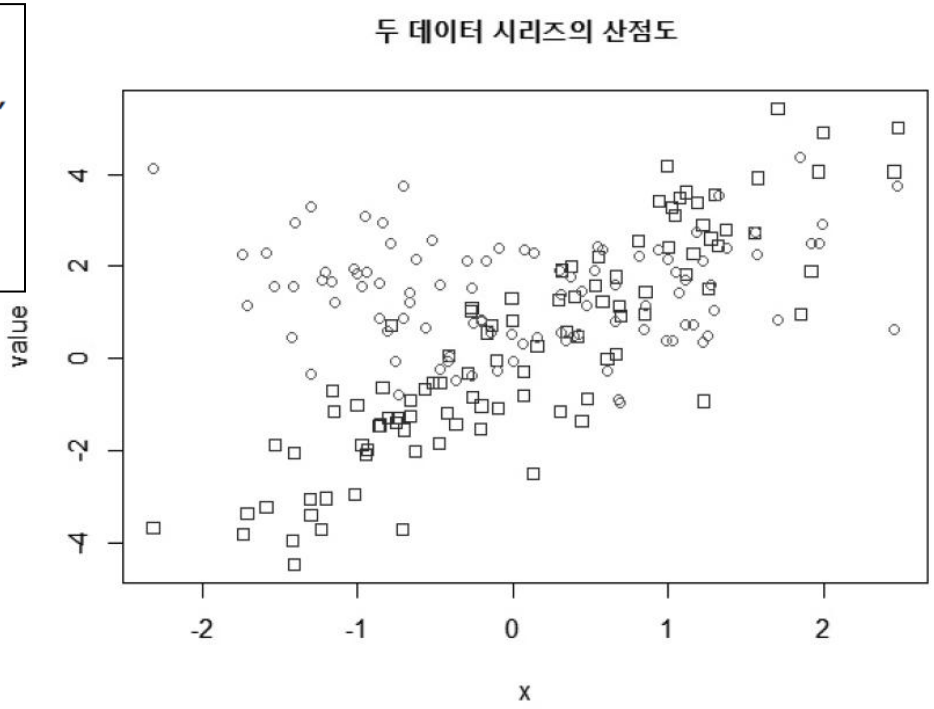


7.2 데이터 시각화하기

- 산점도 만들기

✓ 산포도를 그레이 스케일로 인쇄하면 이 두 색상은 농도가 서로 다른 회색으로 보임.
✓ 역시 다른 그룹의 점들을 구분하고자 다른 col 인수로 다음과 같이 plot()과 points() 함수를 다시 한 번 사용 가능.

```
> plot(x, y, col = "blue", pch = 0,
+      xlim = range(x), ylim = range(y, z),
+      xlab = "x", ylab = "value")
> points(x, z, col = "red", pch = 1)
> title("두 데이터 시리즈의 산점도")
```



7.2 데이터 시각화하기

- 산점도 만들기

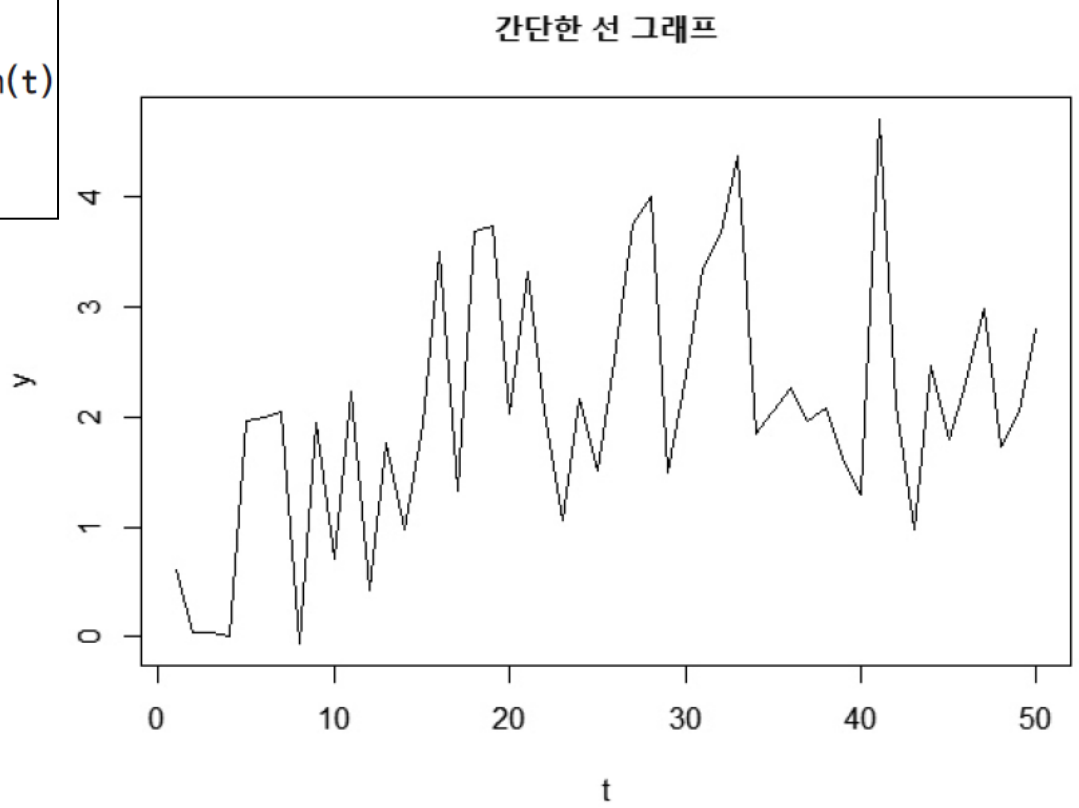
  ✓ R은 일반적으로 사용되는 색상 이름과 그 외 많은 색상(총 657가지)을 지원.
  ✓ colors() 함수를 호출하여 R이 지원하는 전체 색상 목록 확인 가능.

7.2 데이터 시각화하기

- 선 그래프 만들기

✓ 시계열 데이터는 선 그래프가 시간에 따른 추세와 변동을 보여 주기에 유용.
✓ 선 그래프를 만들려면 plot()을 호출할 때 type = "l"로 설정.

```
> t <- 1:50
> y <- 3 * sin(t * pi / 60) + rnorm(t)
> plot(t, y, type = "l",
+      main = "간단한 선 그래프")
```



7.2 데이터 시각화하기

- 선 그래프 만들기

선 종류와 너비 사용자 정의하기

✓ 선 그래프에서는 선의 종류를 지정하는 데 lty 인수를 사용.

✓ 다음 코드는 R이 지원하는 여섯 가지 유형의 미리보기를 보여줌.

```
> lty_values <- 1:6
> plot(lty_values, type = "n", axes = FALSE, ann = FALSE)
> abline(h = lty_values, lty = lty_values, lwd = 2)
> mtext(lty_values, side = 2, at = lty_values)
> title("선 종류 (lty)")
```

7.2 데이터 시각화하기

- 선 그래프 만들기

  - ✓ 앞 코드에서는 type = "n"을 설정하여 적절한 축 범위를 갖는 빈 캔버스를 만들고, 축과 다른 라벨은 모두 없앰.
  - ✓ elements.abline() 함수를 사용하여 선의 종류는 다르지만 선 폭은 동일한 lwd = 2 수평선을 그림.
  - ✓ mtext() 함수는 여백에 텍스트를 그림.
  - ✓ abline()과 mtext() 함수는 각자의 인수에 대해 벡터화되어 있음.
  - ✓ 각 선과 여백 텍스트를 차례로 그리려고 for 루프를 사용할 필요가 없음.

7.2 데이터 시각화하기

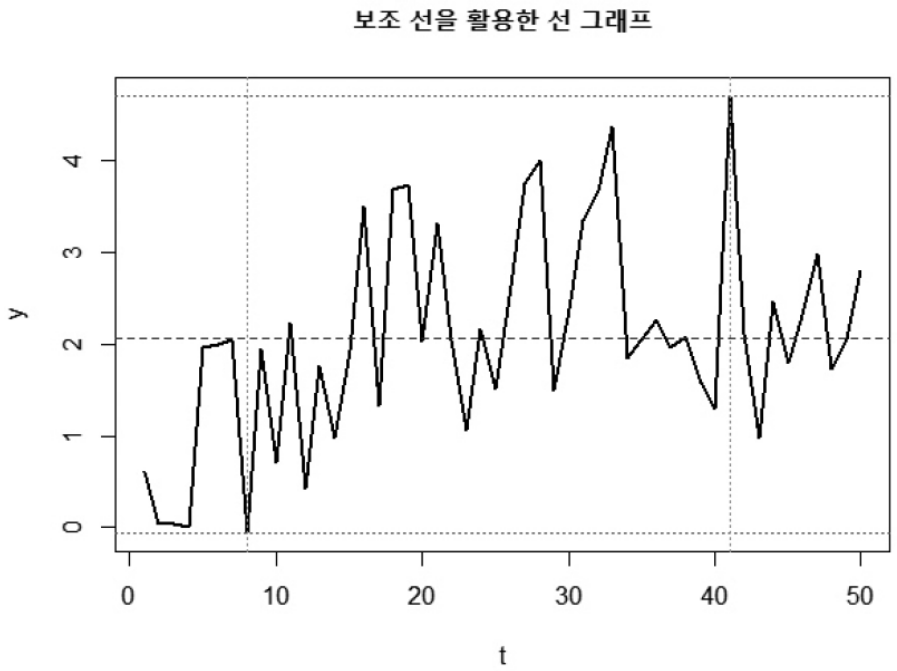
- 선 그래프 만들기

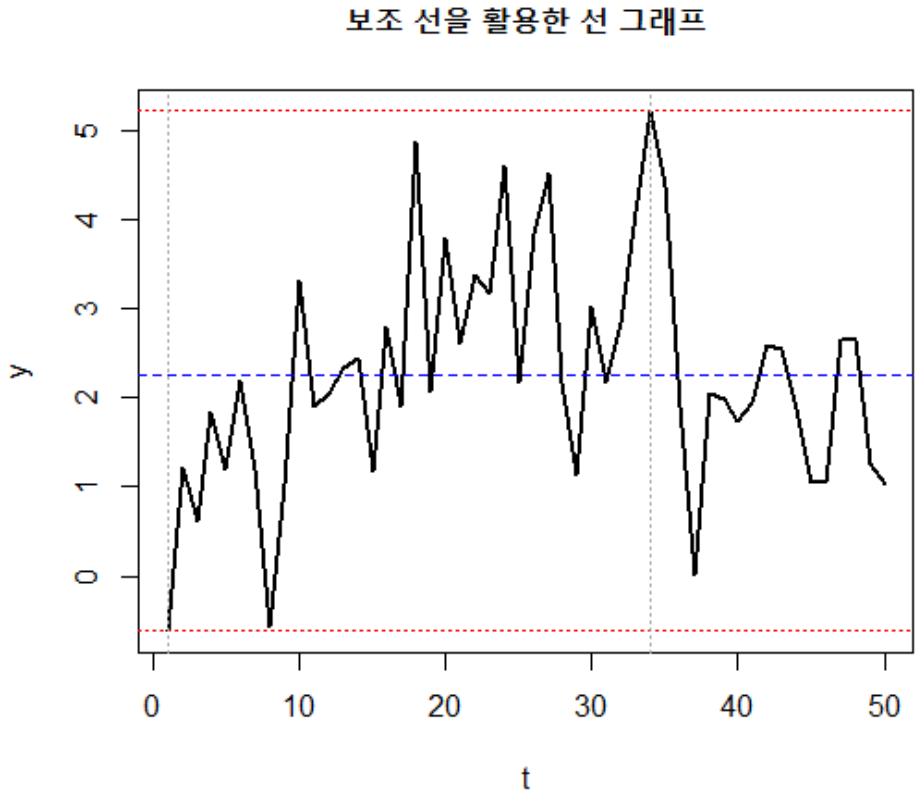
  - ✓ 다음 예제는 그래프에 보조선을 그릴 때 abline() 함수가 유용하다는 것을 보여줌.
  - ✓ 먼저 선 그래프를 그리기 전에 시간 t에 대한 y 값의 선 그래프를 생성.
  - ✓ y의 평균값과 시간에 따른 최댓값과 최솟값 범위를 이 그래프에 함께 나타내길 원한다고 가정해 보자.
  - ✓ abline() 함수를 사용하면 다양한 선 종류와 색상을 활용하여 분명하게 구분할 수 있는 여러 가지 보조선을 쉽게 그릴 수 있음.

```
> plot(t, y, type = "l", lwd = 2)
> abline(h = mean(y), lty = 2, col = "blue")
> abline(h = range(y), lty = 3, col = "red")
> abline(v = t[c(which.min(y), which.max(y))], lty = 3, col = "darkgray")
> title("보조 선을 활용한 선 그래프")
```

7.2 데이터 시각화하기

- 그림 7-15 보조선을 활용한 선 그래프





7.2 데이터 시각화하기

- 선 그래프 만들기

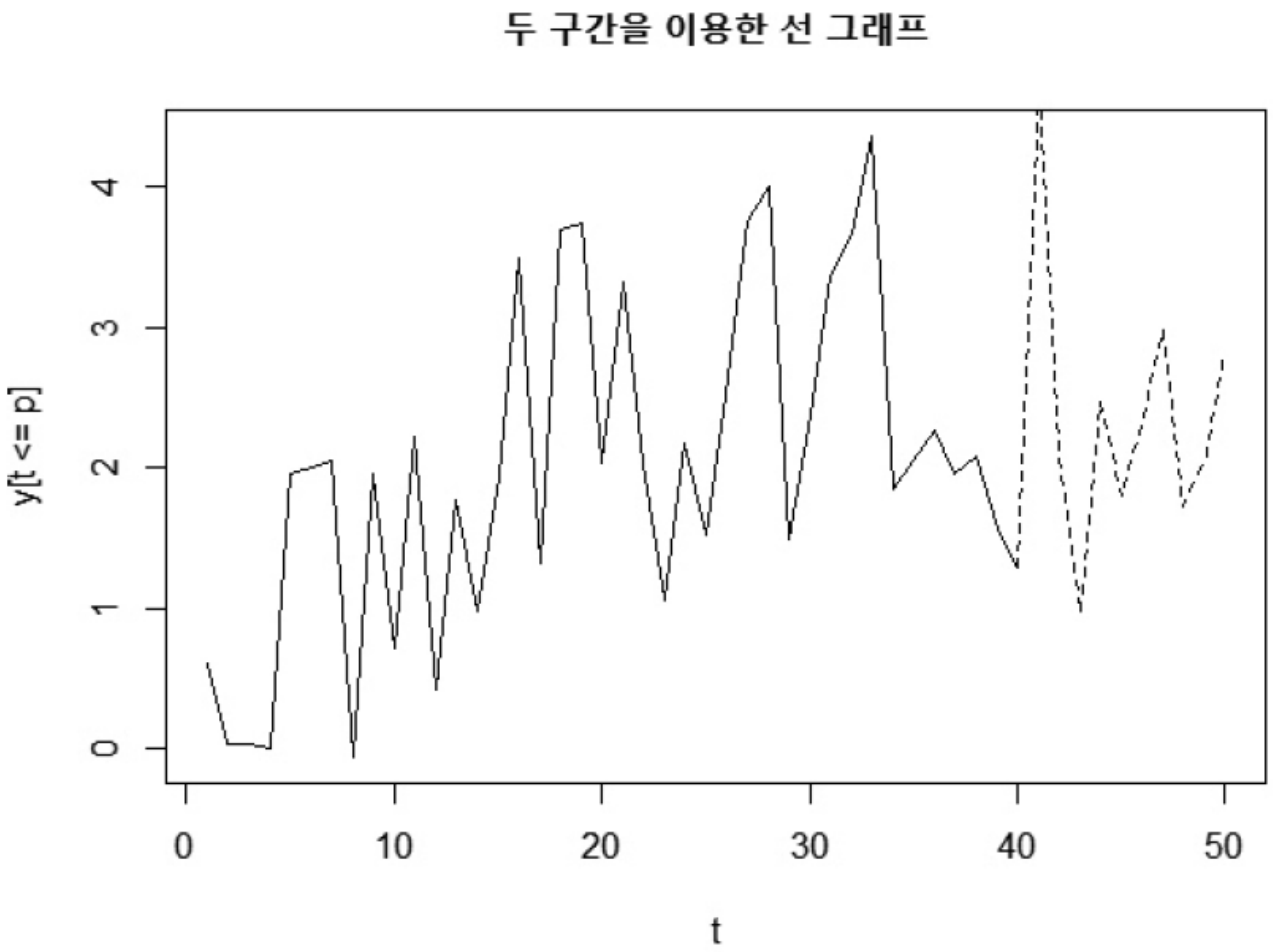
다중 구간에 선 그리기

✓ 다른 유형의 선들이 섞여 있는 또 다른 종류의 선 그래프는 바로 다중 구간 선 그래프.
✓ 보통 처음 구간은 과거 데이터를 나타내고 다음 구간은 예측 데이터를 의미.
✓ 첫번째 구간은 처음 관측 데이터 40개를 포함하고, 나머지 점들은 과거 데이터를 기반으로 한 예측 결과라고 가정하자.
✓ 과거 데이터를 나타내는 데 실선을 사용하고, 예측 구간을 나타내는 데 점선을 사용.
✓ 첫 번째 구간의 데이터를 먼저 그리고, 두 번째 구간의 데이터는 점선을 추가하여 그림.
✓ lines() 함수는 선 그래프를 그리고, points() 함수는 산점도를 그림.

```
> p <- 40
> plot(t[t <= p], y[t <= p], type = "l",
+     xlim = range(t), xlab = "t")
> lines(t[t >= p], y[t >= p], lty = 2)
> title("두 구간을 이용한 선 그래프")
```

7.2 데이터 시각화하기

그림 7-16 다중 구간을 표시한 선 그래프



7.2 데이터 시각화하기

- 선 그래프 만들기

점으로 선 그리기

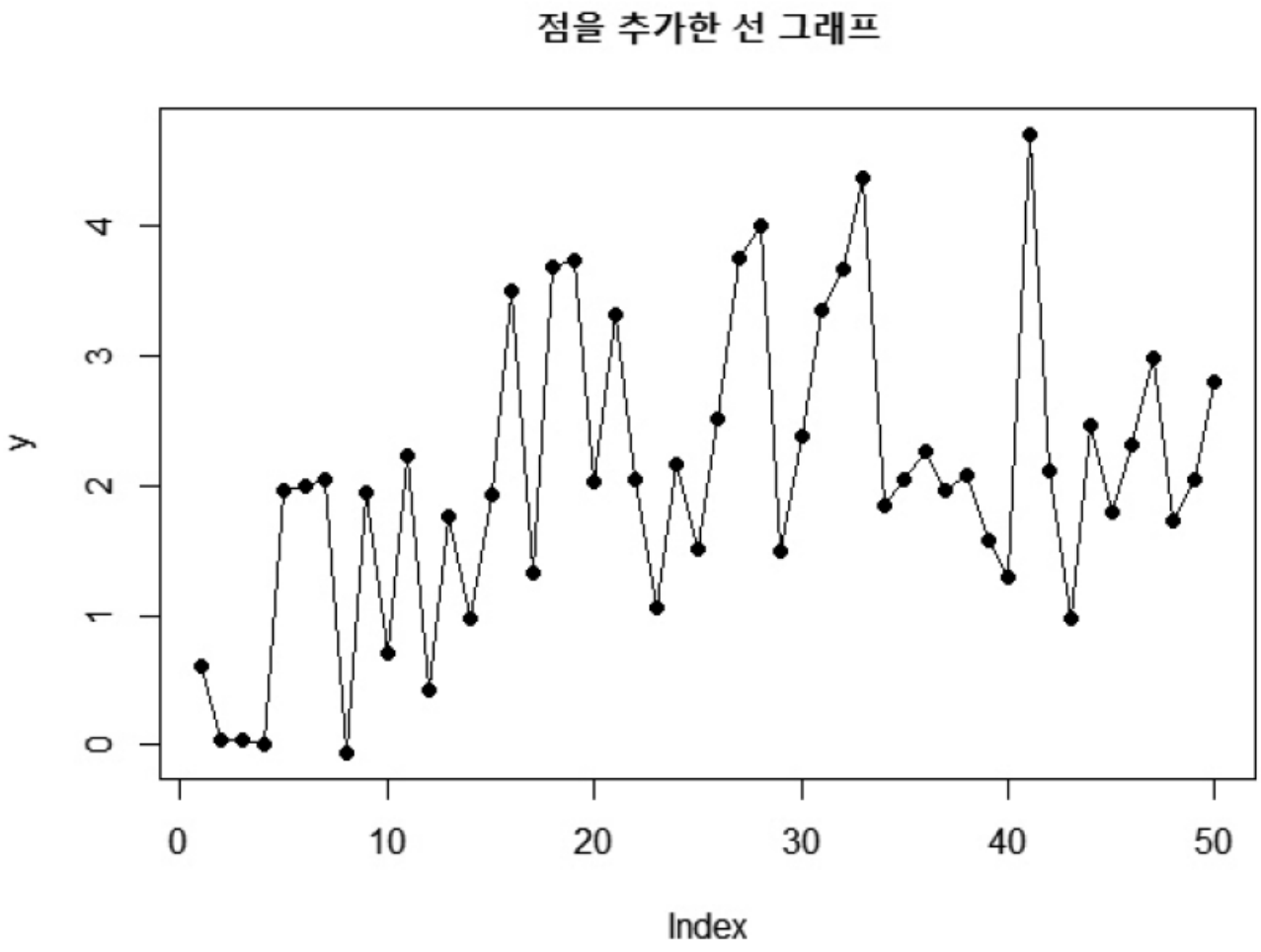
✓ 때로는 동일한 그래프에 선과 점을 모두 사용하여 관측 데이터가 차트에서 명확하게 보이도록 강조하는 것이 더 유용.

✓ 선 그래프를 먼저 그리고, points() 함수를 사용하여 동일한 데이터를 그래프에 다시 추가하면 됨.

```
> plot(y, type = "l")
> points(y, pch = 16)
> title("점을 추가한 선 그래프")
```

7.2 데이터 시각화하기

그림 7-17 점을 추가한 선 그래프



7.2 데이터 시각화하기

- 선 그래프 만들기

  ✓ 이를 수행하는 동일한 방법은 먼저 plot() 함수로 산점도를 그린 후 lines() 함수로 똑같은 데이터의 선 그래프를 다시 추가하는 것.

  ✓ 다음 코드는 앞 예제와 동일한 그래프를 만듦.

```
> plot(y, pch = 16)
> lines(y)
> title("점을 추가한 선 그래프")
```

7.2 데이터 시각화하기

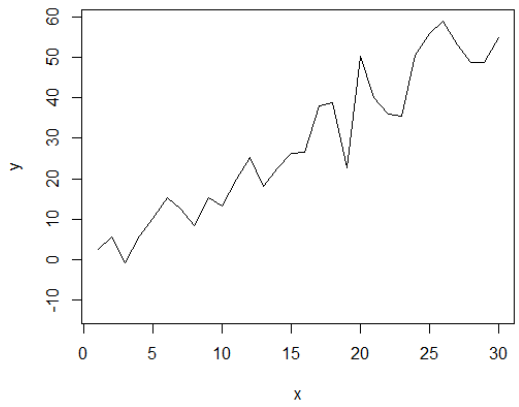
- 선 그래프 만들기

범례가 있는 다중 곡선 그래프 그리기

✓ 제대로 된 다중 곡선 그래프에는 선과 점으로 표시된 여러 곡선과 각 곡선을 설명하는 범례가 포함되어야 함.

✓ 다음 코드는 시간 x에 따른 y와 z 계열을 무작위로 생성하고, 이들을 함께 표시한 그래프를 만듦.

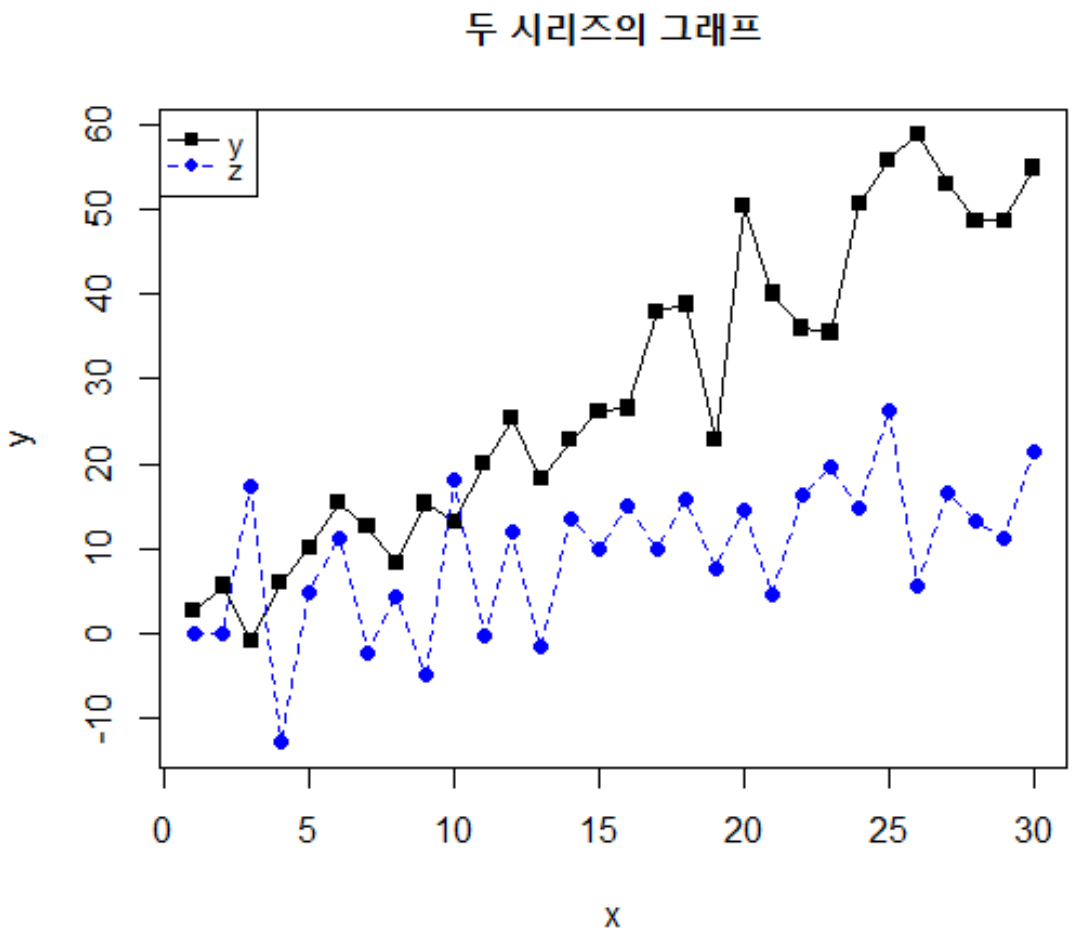
```
> x <- 1:30
> y <- 2 * x + 6 * rnorm(30)
> z <- 3 * sqrt(x) + 8 * rnorm(30)
> plot(x, y, type = "l",
+      ylim = range(y, z), col = "black")
```




```
> points(y, pch = 15)
> lines(z, lty = 2, col = "blue")
> points(z, pch = 16, col = "blue")
> title("두 시리즈의 그래프")
> legend("topleft",
+      legend = c("y", "z"),
+      col = c("black", "blue"),
+      lty = c(1, 2), pch = c(15, 16),
+      cex = 0.8, x.intersp = 0.5, y.intersp = 0.8)
```

7.2 데이터 시각화하기

- 그림 7-18 범례를 추가한 다중 곡선 그래프



7.2 데이터 시각화하기

- 선 그래프 만들기

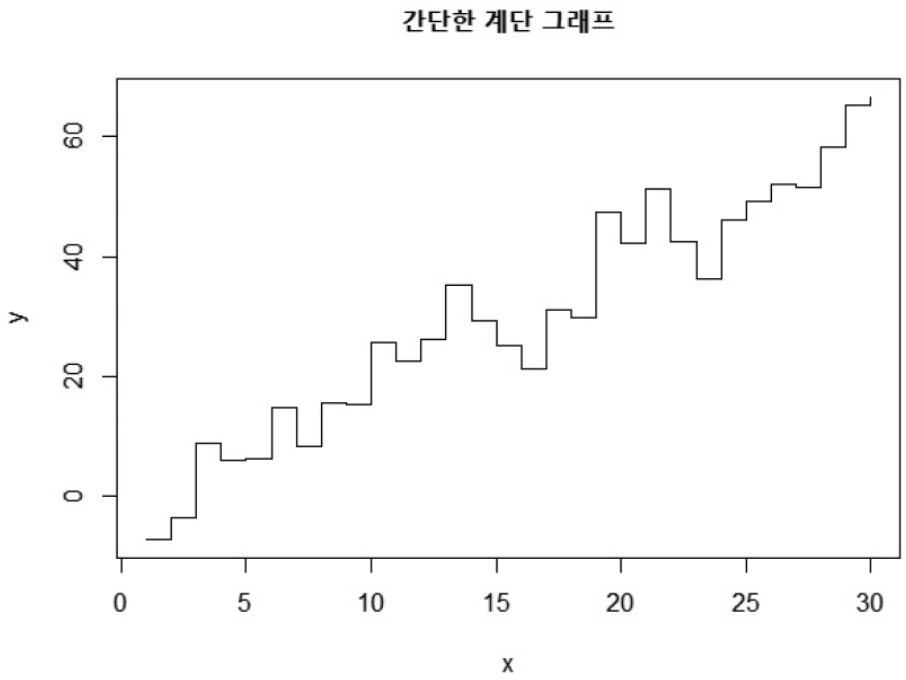
  - ✓ 앞 코드는 plot() 함수를 사용하여 y의 그래프를 먼저 만들고, lines()와 points() 함수로 z 그래프를 추가.
  - ✓ legend() 함수로 왼쪽 위에 범례를 추가하여 y와 z의 선과 점 스타일을 각각 보여줌.
  - ✓ cex는 범례의 글꼴 크기를 조정하는 데 사용.
  - ✓ x.intersp와 y.intersp는 범례를 미세하게 조정하는 데 사용.

7.2 데이터 시각화하기

- 선 그래프 만들기

  ✓ 또 다른 유용한 선 그래프의 종류는 계단 모양 그래프.
  ✓ 우리는 plot()과 lines() 함수에서 type = "s"를 사용하여 계단 선 그래프를 만듦.

```
> plot(x, y, type = "s", main = "간단한 계단 그래프")
```

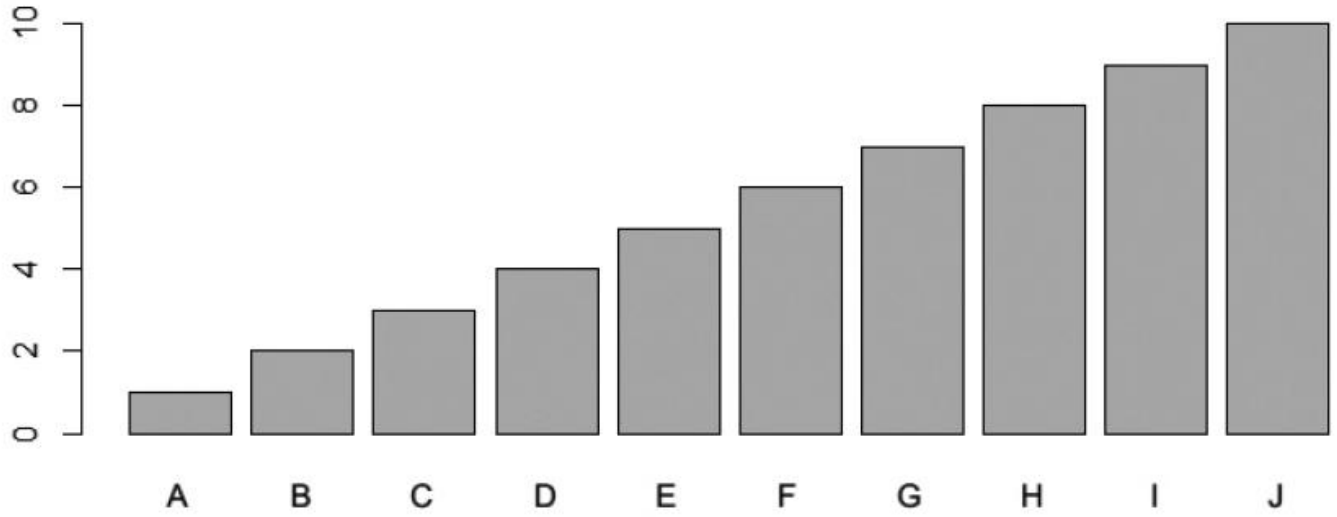


7.2 데이터 시각화하기

- 막대 그래프 그리기

✓ 막대 그래프(bar chart)는 많이 사용하는 그래프 중 하나.
✓ 막대 그래프는 막대 높이를 사용하여 서로 다른 범주의 양을 한눈에 비교 가능.
✓ 가장 쉽게 만들 수 있는 막대 그래프는 다음과 같음.
✓ 여기에서는 plot() 함수 대신 barplot() 함수를 사용.

```
> barplot(1:10, names.arg = LETTERS [1:10])
```

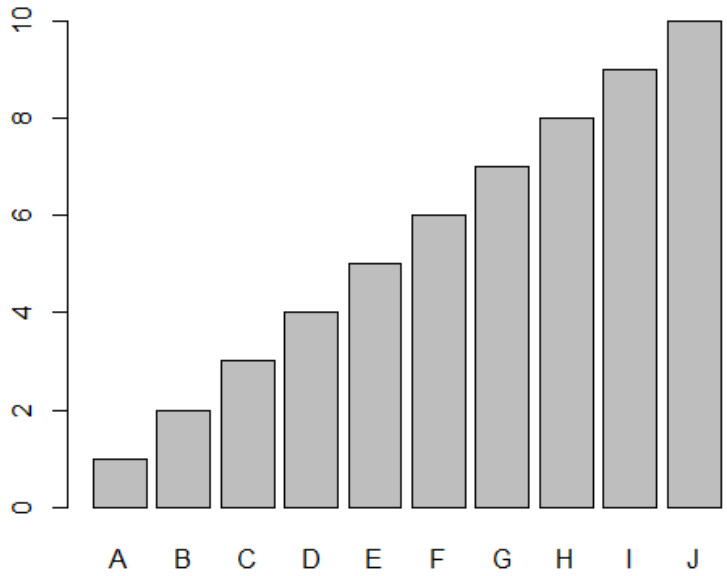


7.2 데이터 시각화하기

- 막대 그래프 그리기

  ✓ 수치형 벡터에 이름이 있다면 이 이름들을 x축에 자동으로 사용.
  ✓ 다음 코드도 이전 코드와 마찬가지로 똑같은 막대 그래프를 생성.

```
> ints <- 1:10
> names(ints) <- LETTERS[1:10]
> barplot(ints)
```



7.2 데이터 시각화하기

- 막대 그래프 그리기

✓ 이제 nycflights13에서 얻은 항공편 데이터셋을 사용하여 기록 중에서 가장 많은 항공편을 가진 상위 8개 항공사에 대한 막대 그래프를 만들 수 있음.

✓ 다음 코드에서는 각 항공사의 항공기 편수를 세는 데 table() 함수를 사용.

```
> data("flights", package = "nycflights13")
> carriers <- table(flights$carrier)
> carriers
   9E     AA    AS     B6     DL     EV    F9    FL    HA     MQ   OO     UA     US    VX
   WN     YV
18460  32729   714  54635  48110  54173   685  3260   342  26397   32  58665  20536  5162
12275    601
```

7.2 데이터 시각화하기

- 막대 그래프 그리기

✓ 다음 코드에서는 carriers를 내림차순(decreasing = TRUE)으로 정렬.

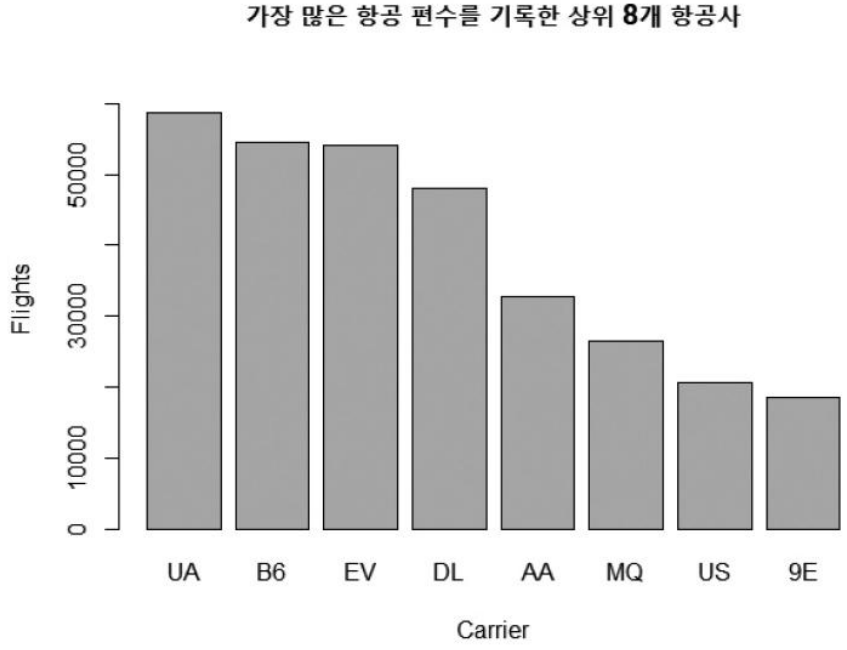
```
> sorted_carriers <- sort(carriers, decreasing = TRUE)
> sorted_carriers
   UA    B6    EV    DL    AA    MQ    US    9E    WN    VX    FL    AS    F9
   YV    HA    OO
58665 54635 54173 48110 32729 26397 20536 18460 12275  5162  3260   714   685
  601   342    32
```

7.2 데이터 시각화하기

- 막대 그래프 그리기

✓ 이 테이블에서 첫 원소 8개만 추출하여 막대 그래프를 그림

```
> barplot(head(sorted_carriers, 8),
+     ylim = c(0, max(sorted_carriers) * 1.1),
+     xlab = "Carrier", ylab = "Flights",
+     main = "가장 많은 항공 편수를 기록한 상위 8개 항공사")
```

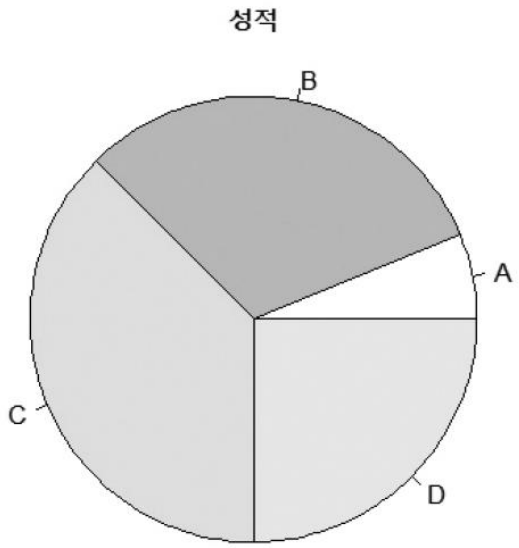


7.2 데이터 시각화하기

- 원 그래프 만들기

  ✓ 원 그래프(파이 차트(pie chart))는 또 다른 형태의 유용한 그래프.
  ✓ pie() 함수는 barplot() 함수와 비슷한 방식으로 원 그래프를 그림.
  ✓ 수치형 벡터에 라벨을 적용하여 사용 가능.
  ✓ 물론 이름을 갖고 있는 수치형 벡터를 바로 사용할 수도 있음.

```
> grades <- c(A = 2, B = 10, C = 12, D = 8)
> pie(grades, main = "성적", radius = 1)
```



7.2 데이터 시각화하기

- 히스토그램과 밀도 그래프 그리기

  - ✓ 산점도와 선 그래프는 데이터셋의 관측 값을 직접적으로 보여줌
  - ✓ 막대 그래프와 원 그래프는 일반적으로 서로 다른 범주의 데이터 요소에서 대략적인 요약 정보를 표시하는 데 사용함
  - ✓ 산점도와 선 그래프는 너무 많은 정보를 전달하여 데이터에 대한 통찰을 끌어내기가 어려움
  - ✓ 반대로 막대 그래프와 원 그래프는 많은 정보를 놓칠 수 있어 이 정보만으로 확신을 갖고 어떤 결정적인 판단을 내리기가 어려울 수 있음

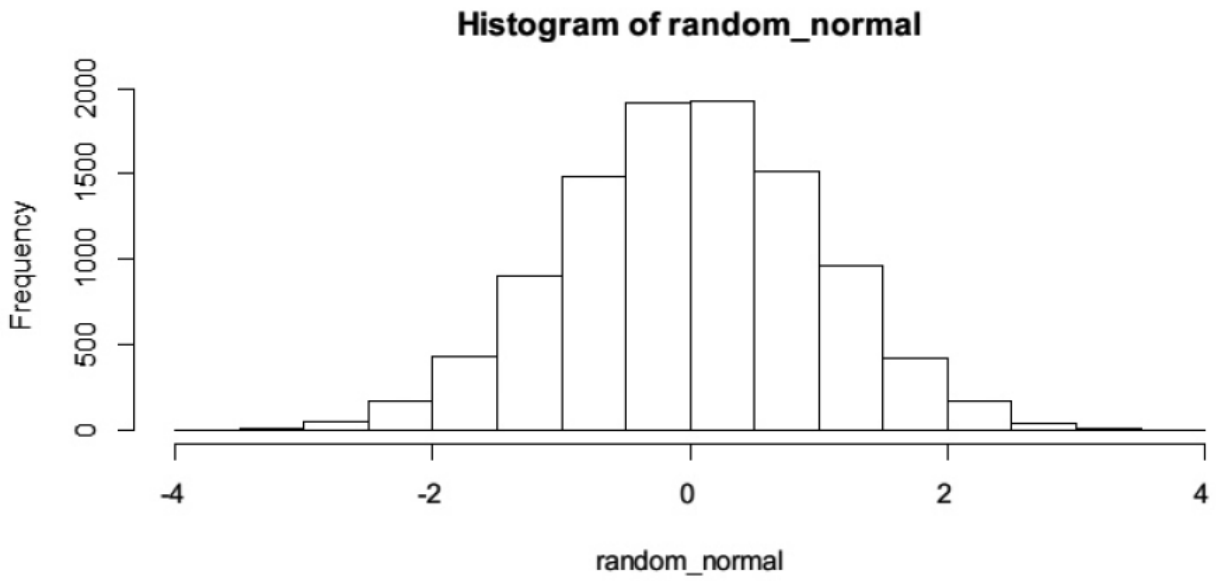
7.2 데이터 시각화하기

- 히스토그램과 밀도 그래프 그리기

  ✓ 히스토그램은 수치형 벡터의 데이터 분포를 보여 주며, 정보를 많이 잃어버리지 않고 데이터를 요약해 주기 때문에 더 쉽게 사용 가능.

  ✓ 다음 코드는 hist() 함수를 사용하여 정규 분포에서 추출한 난수 벡터로 정규 분포 밀도 함수의 히스토그램을 생성하는 방법을 보여줌.

```
> random_normal <- rnorm(10000)
> hist(random_normal)
```

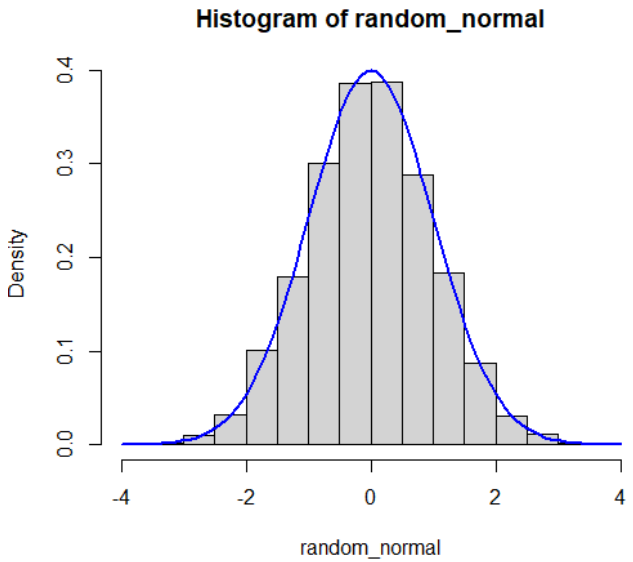


7.2 데이터 시각화하기

- 히스토그램과 밀도 그래프 그리기

  ✓ 기본적으로 히스토그램의 y축은 해당 범위에서 데이터 값의 빈도.
  ✓ random_normal 객체를 만드는 데 사용한 표준 정규 분포와 이 히스토그램이 매우 가깝다는 것을 검증 가능.
  ✓ 표준 정규 분포의 확률 밀도 함수 곡선을 표시하는 데 dnorm() 함수를 사용.
  ✓ 히스토그램의 y축은 확률을 의미하고 히스토그램에 곡선을 추가.

```
> hist(random_normal, probability = TRUE, col = "lightgray")
> curve(dnorm, add = TRUE, lwd = 2, col = "blue")
```

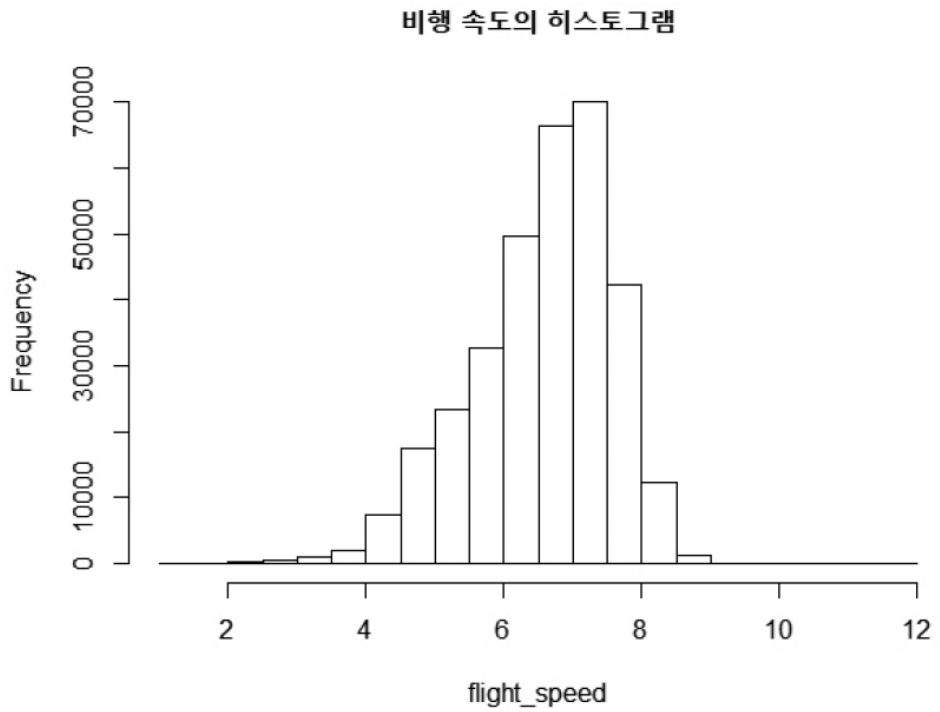


7.2 데이터 시각화하기

- 히스토그램과 밀도 그래프 그리기

  ✓ 이제 비행 중인 항공기 속도에 대한 히스토그램을 만들어 보자.
  ✓ 비행 중인 항공기의 평균 속도는 비행 거리(distance)를 대기 시간(air_time)으로 나눈 값.

```
> flight_speed <- flights$distance / flights$air_time
> hist(flight_speed, main = "비행 속도의 히스토그램")
```

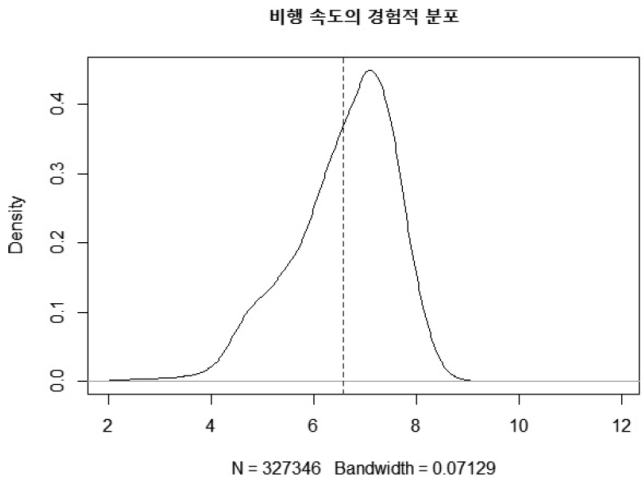


7.2 데이터 시각화하기

- 히스토그램과 밀도 그래프 그리기

  ✓ 이 히스토그램은 이전의 정규 분포와는 조금 차이가 있어 보임.
  ✓ 이때 density() 함수를 사용하여 속도의 경험적 분포를 추정하고, 그중에서 아주 부드러운 확률 분포 곡선을 그림.
  ✓ 모든 관측 값의 전체 평균을 나타내는 수직선을 추가.

```
> plot(density(flight_speed, from = 2, na.rm = TRUE),
+     main = "비행 속도의 경험적 분포")
> abline(v = mean(flight_speed, na.rm = TRUE),
+     col = "blue", lty = 2)
```



7.2 데이터 시각화하기

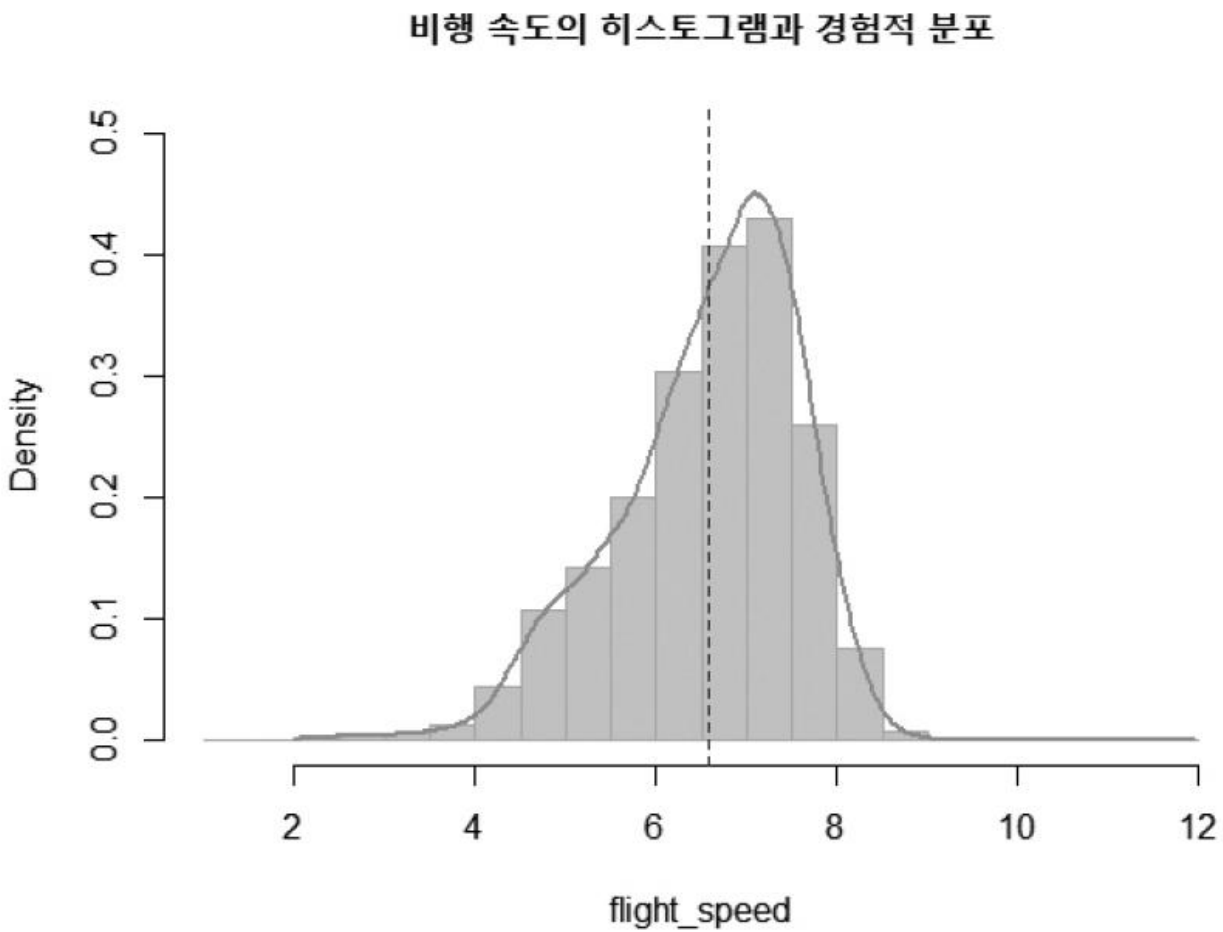
- 히스토그램과 밀도 그래프 그리기

  ✓ 앞서 히스토그램과 곡선을 그렸을 때와 마찬가지로 데이터를 더 잘 보려고 두 그래프를 하나로 합칠 수 있음

```
> hist(flight_speed,
+      probability = TRUE, ylim = c(0, 0.5),
+      main = "비행 속도의 히스토그램과 경험적 분포",
+      border = "gray", col = "lightgray")
> lines(density(flight_speed, from = 2, na.rm = TRUE),
+      col = "darkgray", lwd = 2)
> abline(v = mean(flight_speed, na.rm = TRUE),
+      col = "blue", lty = 2)
```

7.2 데이터 시각화하기

그림 7-27 항공기의 비행 속도에 대한 히스토그램과 경험적 분포를 함께 표시한 그래프

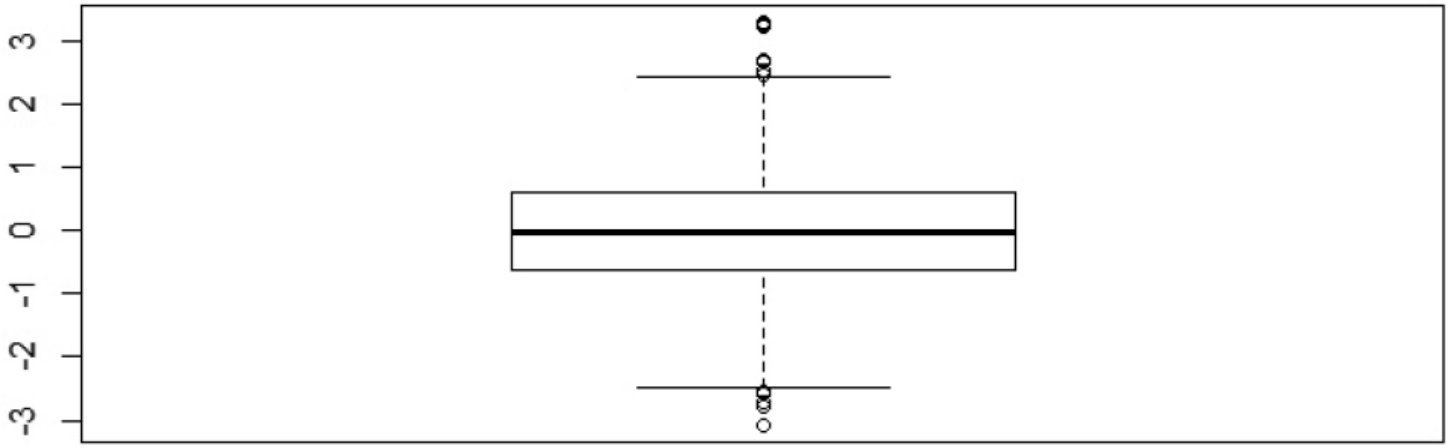


7.2 데이터 시각화하기

- 상자 그림 그리기

✓ 히스토그램과 확률 분포 곡선은 데이터 분포를 보여 주는 대표적인 방법.
✓ 일반적으로는 전체 분포에 대한 인상을 얻는 데 여러 중요한 분위수만 있어도 됨.
✓ 상자 그림(box plot) 또는 상자 수염 그림은 이를 수행하는 간단한 방법.
✓ 임의로 생성된 수치형 벡터에 대해 상자 그림을 그리기 위해 boxplot() 함수 호출.

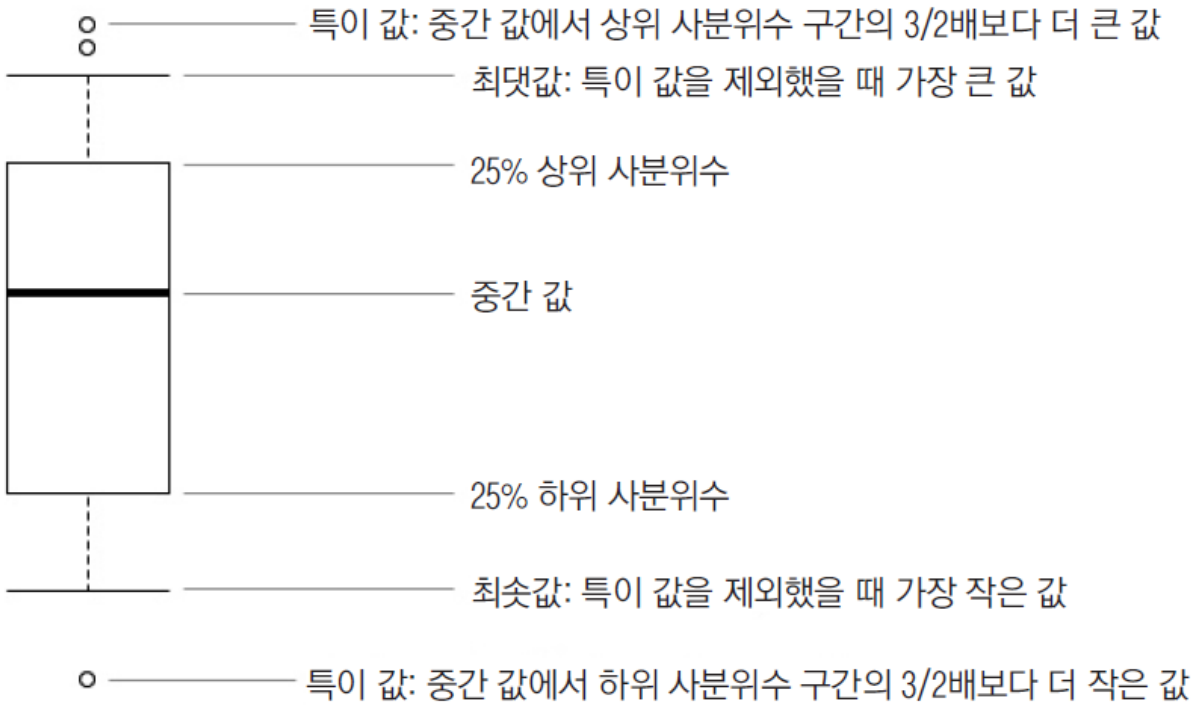
```
> x <- rnorm(1000)
> boxplot(x)
```



7.2 데이터 시각화하기

- 상자 그림 그리기

✓ 상자 그림은 데이터의 중요한 사분위 값과 특이 값을 보여 주는 몇 가지 구성 요소를 포함.
✓ 다음은 상자 그림이 의미하는 바를 명확하게 설명.

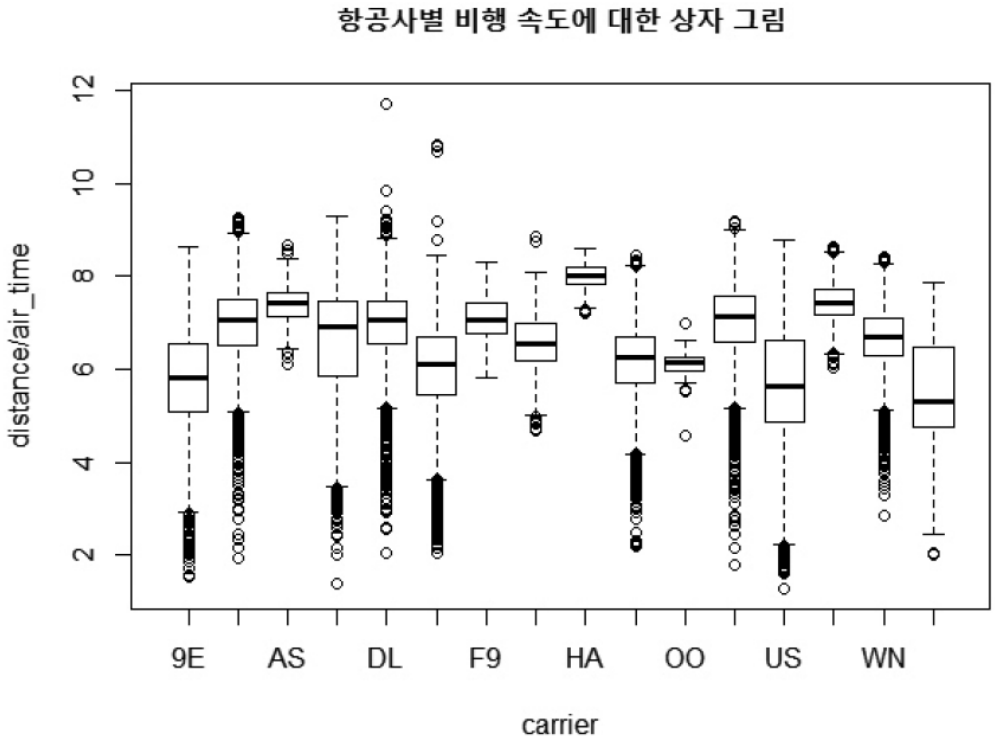


7.2 데이터 시각화하기

- 상자 그림 그리기

✓ 다음 코드는 각 항공사의 비행 속도에 대한 상자 그림을 만듦.
✓ 한 그래프 안에 상자 16개가 있으므로 서로 다른 항공사의 대략적인 분포를 쉽게 비교 가능.

```
> boxplot(distance / air_time ~ carrier, data = flights,
+     main = "항공사별 비행 속도에 대한 상자 그림")
```



7.2 데이터 시각화하기

- 상자 그림 그리기

  - ✓ boxplot() 함수에서는 그래픽을 생성하는 포뮬라(formula) 인터페이스를 사용.
  - ✓ distance / air_time ~ carrier는 기본적으로 y축은 거리 / 비행 시간, 즉 비행 속도를 나타내고 x축은 항공사를 나타냄.
  - ✓ data = flights는 boxplot() 함수에 지정한 포뮬라에서 기호를 찾을 위치를 알려줌.
  - ✓ 결과적으로 항공사별 비행 속도를 나타내는 상자 그림이 만들어짐.
  - ✓ 데이터를 시각화하고 분석하는 포뮬라 인터페이스는 표현력이 매우 풍부하고 강력함.

7.3 데이터 분석하기

7.3 데이터 분석하기

- 데이터 분석하기

  - ✓ 실제 데이터 분석에서는 데이터 정리, 즉 원본 데이터(또는 원시 데이터)를 필터링하고 분석하기 쉬운 형태로 변환하는 데 대부분의 시간을 소비.
  - ✓ 필터링과 변환 과정을 데이터 조작이라고도 함.
  - ✓ 이 절에서는 분석할 데이터가 이미 준비되었다고 가정함.
  - ✓ 이 절에서는 모델을 자세히 다루지는 않겠지만, 데이터에 적합한 모델을 어떻게 만드는지, 이렇게 얻은 모델과 어떻게 상호 작용하는지, 예측을 위해 이 모델들을 어떻게 적용하는지를 대략 파악하려고 간단한 모델을 활용할 예정.

7.3 데이터 분석하기

- 선형 모델 피팅하기

  ✓ R에서 가장 간단한 모델은 선형 모델.
  ✓ 즉, 어떤 가정하에서 두 무작위 변수 사이의 관계를 설명하려고 선형 함수를 사용.
  ✓ 다음 예제에서는 먼저 x를 3 + 2 * x에 매핑하는 선형 함수를 만듦.
  ✓ 정규 분포를 따르는 난수 벡터 x를 생성하고 f(x)에 독립적인 노이즈를 더하여 y를 생성.

```
> f <- function(x) 3 + 2 * x
> x <- rnorm(100)
> y <- f(x) + 0.5 * rnorm(100)
```

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 어떻게 x에서 y가 만들어졌는지 모른다고 했을 때, 선형 모델로 이 관계를 복구할 수 있을까?
✓ 다시 말해 선형 함수의 계수를 다시 구할 수 있을까?
✓ 다음 코드는 x와 y를 선형 모델(lm: linear model)에 맞추려고 lm() 함수를 사용.
✓ y ~ x 포뮬라는 종속 변수 y와 단일 회귀 변수 x 사이에 선형 회귀가 있다는 것을 lm() 함수에 전달하는 표현.

```
> model1 <- lm(y ~ x)
> model1

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      2.969        1.972
```

✓ 실제 계수는 3(절편)과 2(기울기)였고, 표본 데이터 x와 y를 가지고 피팅한 모델 계수는 2.9692146(절편)과 1.9716588(기울기)로 실제 계수 값과 매우 비슷.
✓ model1 객체에 모델을 저장.

7.3 데이터 분석하기

- 선형 모델 피팅하기

  ✓ 다음 코드를 사용하여 모델 계수를 얻음.

```
> coef(model1)
(Intercept)           x
   2.969215    1.971659
```

  ✓ model1 객체는 기본적으로 리스트이기 때문에 model1$coefficients 사용 가능.

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 선형 모델의 통계 특성에 대한 자세한 내용을 보려면 summary() 함수를 호출.

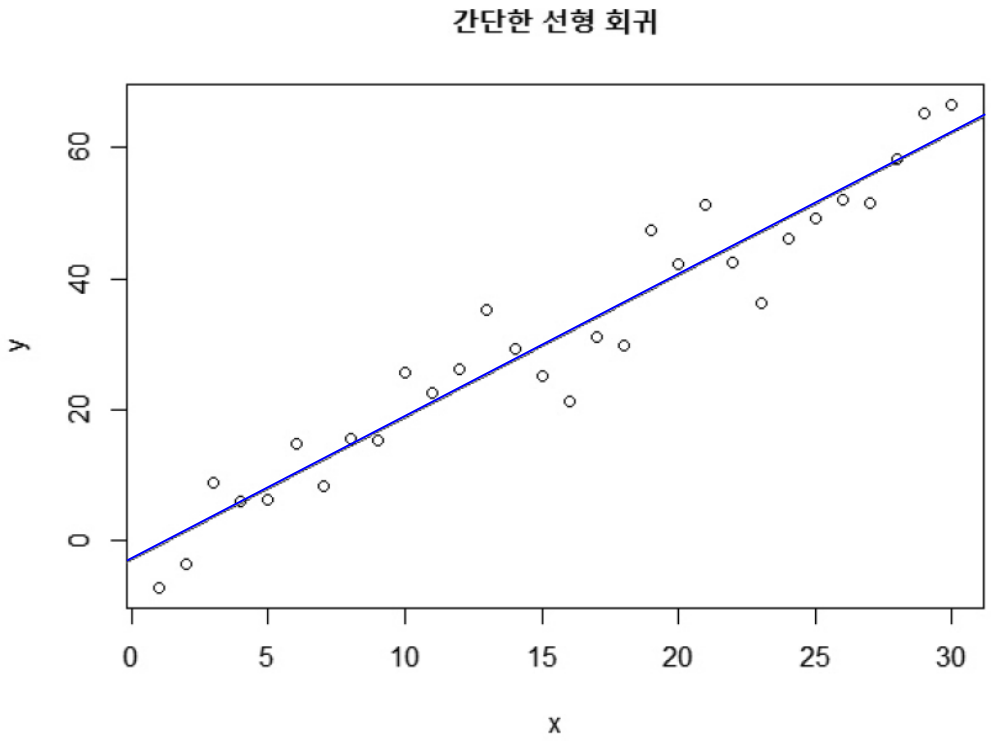
```
> summary(model1)

Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-0.96258 -0.31646 -0.04893  0.34962  1.08491

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.96921    0.04782    62.1   <2e-16 ***
x            1.97166    0.05216    37.8   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.476 on 98 degrees of freedom
Multiple R-squared:  0.9358,     Adjusted R-squared:  0.9352
F-statistic:  1429 on 1 and 98 DF,  p-value: < 2.2e-16
```

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 다음 코드로 데이터와 피팅한 모델이 함께 들어간 그래프를 만듦.

```
> plot(x, y, main = "간단한 선형 회귀")
> abline(coef(model1), col = "blue")
```



7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 앞 코드에서는 abline() 함수에 회귀 계수를 직접 입력하여 원하는 회귀 선을 그렸음.
✓ predict() 함수를 호출하여 적합 모델을 사용해서 예측 가능.
✓ x = -1이나 x = 0.5일 때, 표준 오차 이내에서 y를 예측할 수 있게 다음 코드를 실행하자.

```
> predict(model1, list(x = c(-1, 0.5)), se.fit = TRUE)
$fit
        1         2
0.9975559 3.9550440

$se.fit
         1          2
0.06730363 0.05661319

$df
[1] 98

$residual.scale
[1] 0.4759621
```

✓ 예측 결과는 y의 예측 값($fit), 예측 값의 표준 오차($se.fit), 자유도($df), $residual.scale로 된 리스트 객체.

7.3 데이터 분석하기

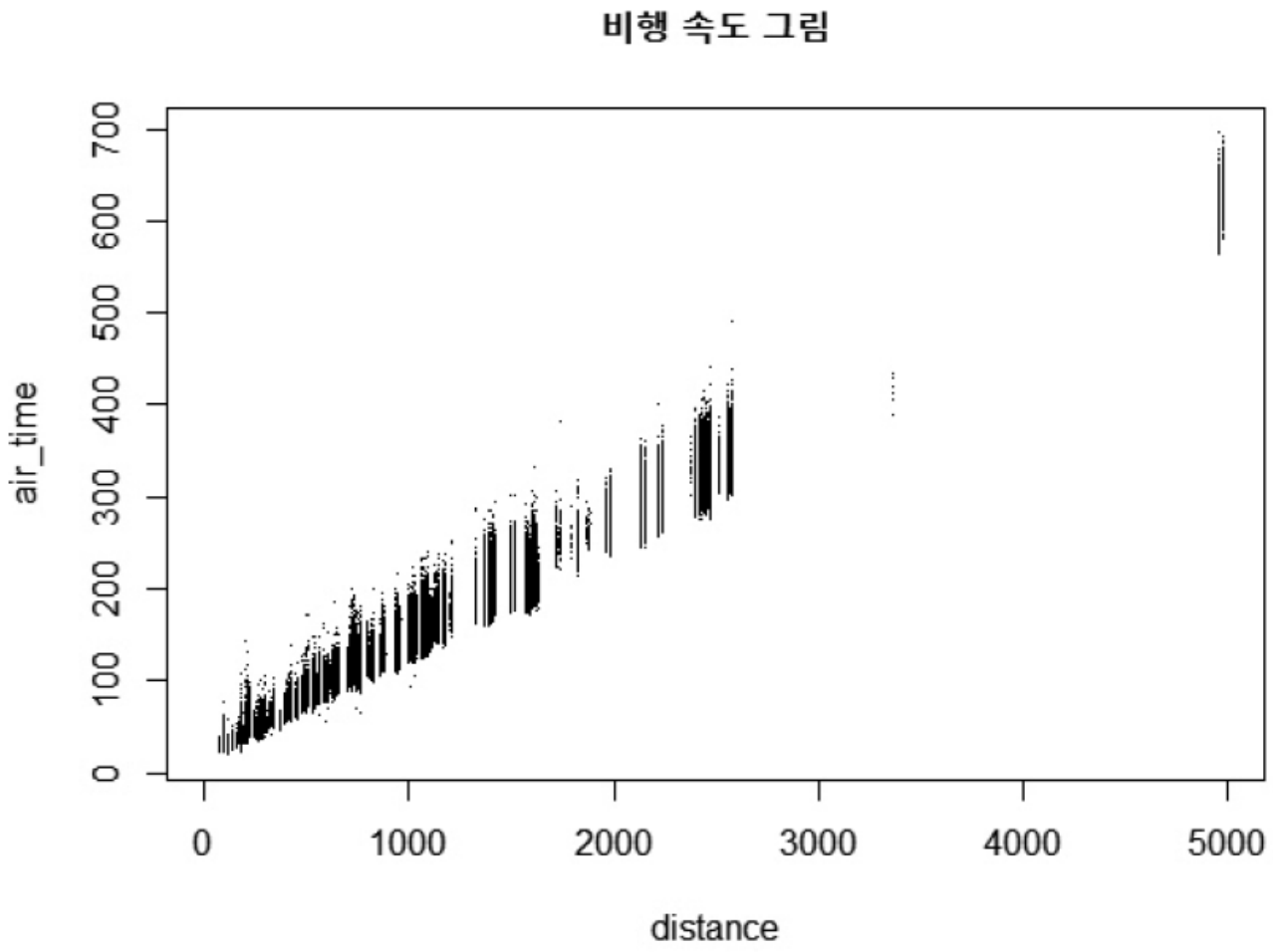
- 선형 모델 피팅하기

✓ 주어진 데이터에 선형 모델을 적용하는 기본 방법을 알아보았음.
✓ 이제 실제 데이터를 살펴볼 차례.
✓ 다음 예제에서는 서로 복잡성이 다른 선형 모델을 사용하여 비행 대기 시간을 예측하려고 함.
✓ 비행 시간을 예측하는 데 가장 도움이 되는 변수는 거리.
✓ 먼저 데이터셋을 가져오고 distance와 air_time을 사용하여 산점도를 그림.
✓ 우리는 pch = "."를 사용.
✓ 데이터셋의 레코드 개수가 많기 때문에 각 포인트를 매우 작게 표시.

```
> data("flights", package = "nycflights13")
> plot(air_time ~ distance, data = flights,
+      pch = ".",
+      main = "비행 속도 그림")
```

7.3 데이터 분석하기

그림 7-32 비행 거리와 시간을 나타내는 산점도



7.3 데이터 분석하기

- 선형 모델 피팅하기

  - ✓ 이 그래프는 distance와 air_time이 양의 상관관계를 갖는다는 것을 분명히 보여줌.
  - ✓ 두 변수 사이에 선형 모델을 적용하는 것이 당연.
  - ✓ 전체 데이터셋을 선형 모델에 적용하기 전에 데이터셋을 먼저 훈련 세트와 테스트 세트의 두 부분으로 나눔.
  - ✓ 데이터셋을 나누는 목적은 표본 내 평가뿐 아니라 표본 외 평가도 수행하기 위해서임.
  - ✓ 좀 더 구체적으로는 데이터의 75%를 훈련용 집합에 넣고, 나머지 25%는 테스트 집합에 넣음.

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 다음 코드에서는 sample() 함수를 사용하여 원본 데이터에서 무작위로 75% 레코드를 추출한 후 setdiff() 함수로 나머지 레코드를 가져옴.

```
> rows <- nrow(flights)
> rows_id <- 1:rows
> sample_id <- sample(rows_id, rows * 0.75, replace = FALSE)
> flights_train <- flights[sample_id,]
> flights_test <- flights[setdiff(rows_id, sample_id), ]
```

✓ setdiff(rows_id, sample_id)는 sample_id가 아닌 rows_id의 인덱스를 반환.

7.3 데이터 분석하기

- 선형 모델 피팅하기

  ✓ flights_train은 훈련 세트고, flights_test는 테스트 세트.
  ✓ 이렇게 분할된 데이터셋을 사용한 모델 피팅과 모델 평가 절차는 간단.
  ✓ 먼저 훈련 세트를 사용하여 모델을 피팅한 후 표본 내 예측을 수행하여 훈련 데이터에서 오차가 얼마나 큰지 확인.

```
> model2 <- lm(air_time ~ distance, data = flights_train)
> predict2_train <- predict(model2, flights_train)
> error2_train <- flights_train$air_time - predict2_train
```

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ evaluate_error() 함수를 정의하여 평균 절대 오차와 오차의 표준 편차를 계산해서 오차 크기를 평가.

```
> evaluate_error <- function(x) {
+     c(abs_err = mean(abs(x), na.rm = TRUE),
+       std_dev = sd(x, na.rm = TRUE))
+ }
```

✓ 이 함수를 사용하여 model2의 표본 내 예측 오차를 계산할 수 있음.

```
> evaluate_error(error2_train)
  abs_err   std_dev
 9.413836 12.763126
```

✓ 절대 평균 오차는 평균적으로 예측 값이 정확한 값에서 약 9.41분 정도 차이가 났으며, 표준 편차는 12.7분 정도임을 의미.

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ 테스트 세트에서 예측을 수행하려고 모델을 사용하여 표본 외 평가를 간단하게 수행.

```
> predict2_test <- predict(model2, flights_test)
> error2_test <- flights_test$air_time - predict2_test
> evaluate_error(error2_test)
   abs_err   std_dev
 9.482135 12.838225
```

✓ 앞 결과에서 predict 함수는 결과적으로 예측 값들로 구성된 수치형 벡터를 생성.

✓ 절대 평균 오차와 표준 편차는 모두 약간씩 올라감.

✓ 이는 표본 외 예측 품질이 크게 나쁘지 않다는 것을 나타내며, model2가 과적합되지 않았다는 것을 보여줌.

7.3 데이터 분석하기

- 선형 모델 피팅하기

✓ model2는 회귀 변수 distance만 고려했는데, 회귀 변수를 더 추가하면 예측도 향상될까?

✓ 다음 코드는 거리뿐만 아니라 항공사(carrier), 월(month), 출발 시간(dep_time)을 회귀 변수로 사용하여 새로운 선형 모델을 구현.

```
> model3 <- lm(air_time ~ carrier + distance + month + dep_time,
+     data = flights_train)
> predict3_train <- predict(model3, flights_train)
> error3_train <- flights_train$air_time - predict3_train
> evaluate_error(error3_train)
  abs_err   std_dev
 9.312961 12.626790
```

✓ 표본 내 오차는 평균과 편차 모두 약간 더 낮아짐.

```
> predict3_test <- predict(model3, flights_test)
> error3_test <- flights_test$air_time - predict3_test
> evaluate_error(error3_test)
 abs_err    std_dev
 9.38309   12.70168
```

✓ model2에 비해서 표본 외 오차 역시 살짝 더 좋아짐

7.3 데이터 분석하기

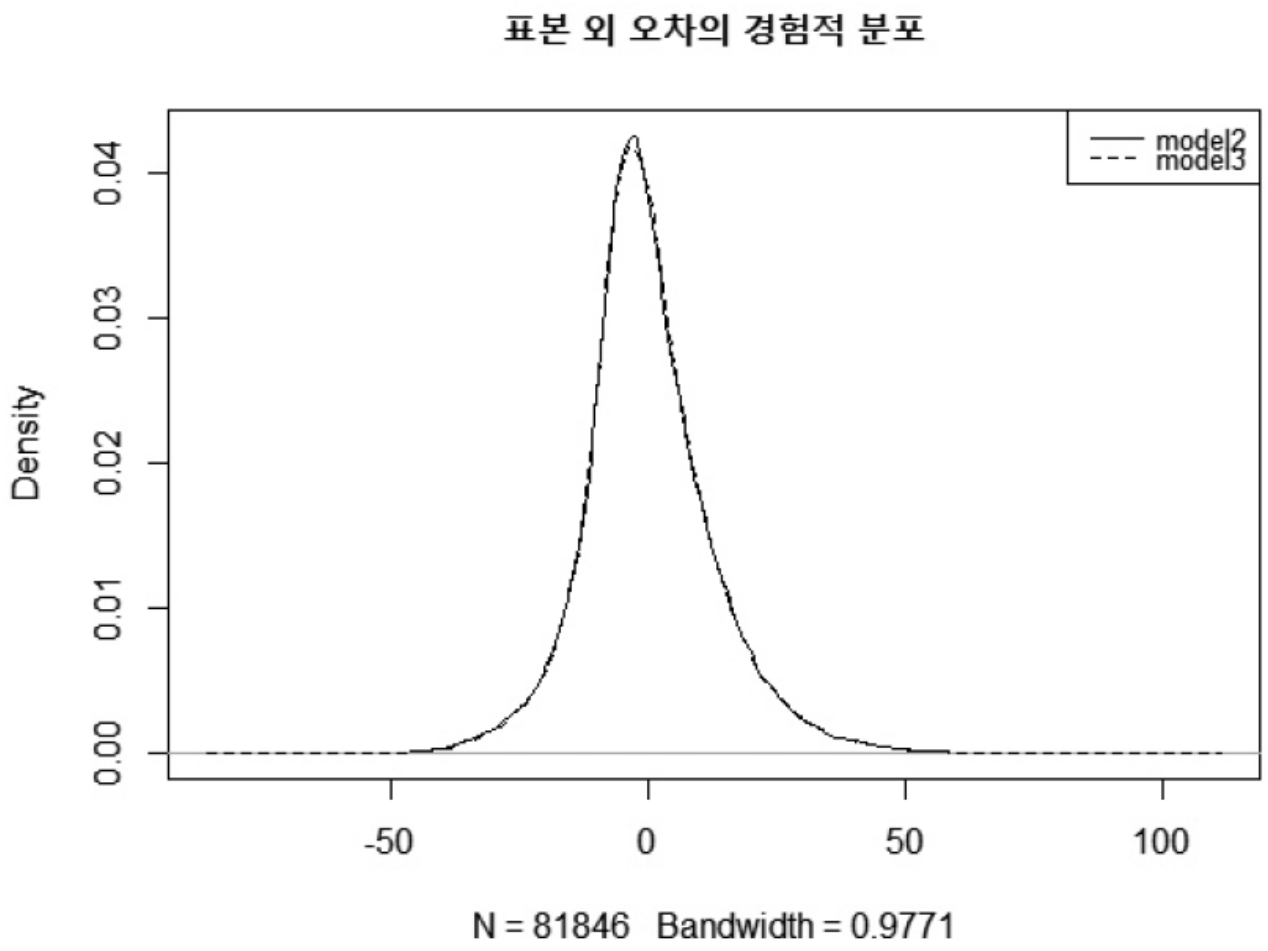
- 선형 모델 피팅하기

✓ 새로운 회귀 변수를 추가하기 전과 후의 표본 외 오차 분포를 비교하고자 두 밀도 곡선을 한 그래프에 그려 보자.

```
> plot(density(error2_test, na.rm = TRUE),
+      main = "표본 외 오차의 경험적 분포")
> lines(density(error3_test, na.rm = TRUE), lty = 2)
> legend("topright", legend = c("model2", "model3"),
+      lty = c(1, 2), cex = 0.8,
+      x.intersp = 0.6, y.intersp = 0.6)
```

7.3 데이터 분석하기

그림 7-33 표본 외 오차의 경험적 분포



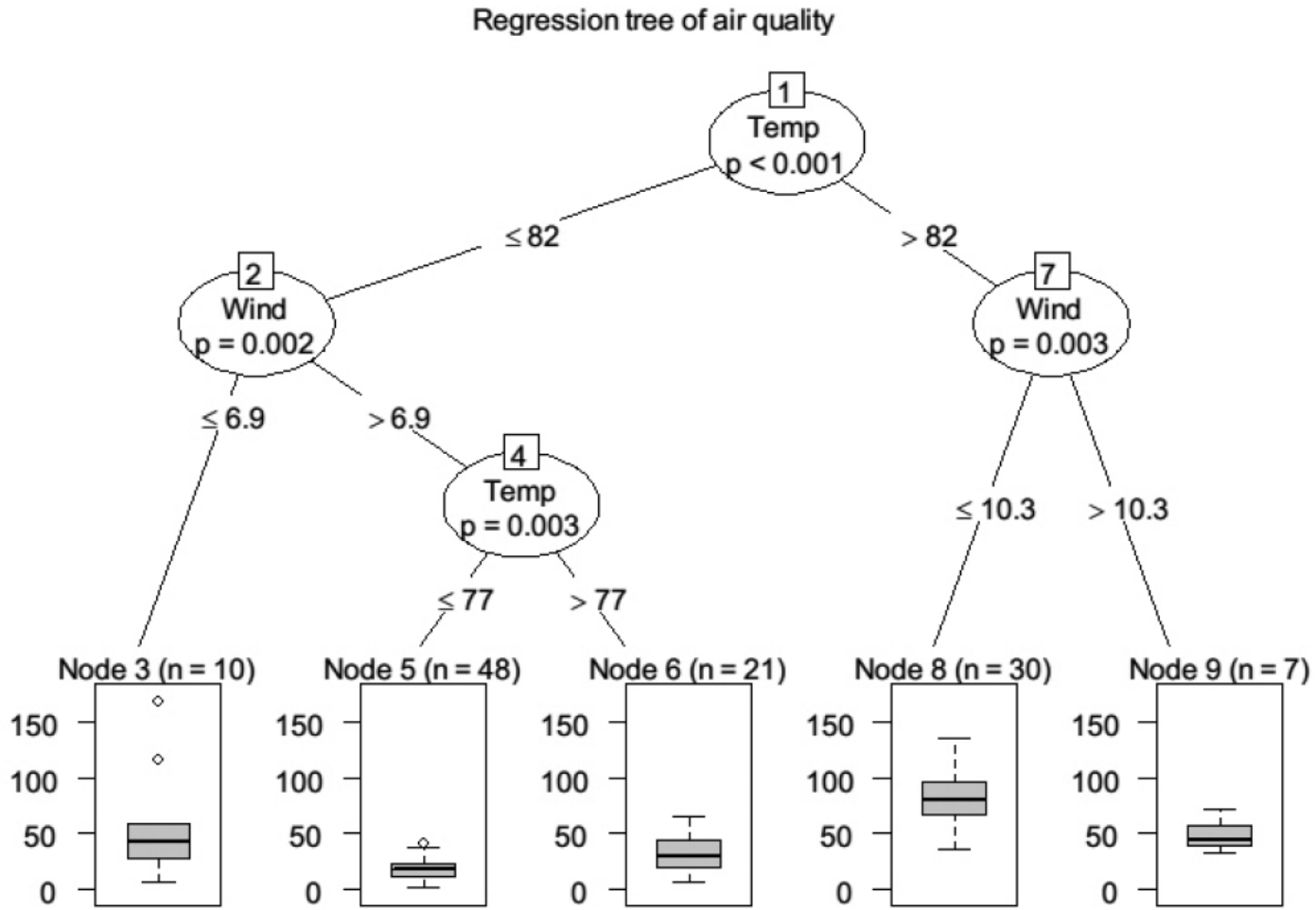
7.3 데이터 분석하기

- 회귀 트리 피팅하기

  - ✓ 이 절에서는 데이터를 피팅하는 데 서로 다른 모델을 사용.
  - ✓ 이 모델을 회귀 트리(https://en.wikipedia.org/wiki/Decision_tree_learning)라고 하며, 머신 러닝 모델 중 하나임.
  - ✓ 단순 선형 회귀는 아니지만, 결정 트리를 사용하여 데이터를 피팅.
  - ✓ 태양 복사(Solar.R), 평균 풍속(Wind), 일일 최대 온도(Temp)에 따라 일일 대기 질(Ozone)을 예측한다고 가정하자.

7.3 데이터 분석하기

그림 7-34 대기질을 예측하는 회귀 트리



7.3 데이터 분석하기

- 회귀 트리 피팅하기

  ✓ 트리에서 원은 가능한 대답이 두 가지인 질문을 의미.
  ✓ 일일 대기질을 예측하려면 위에서 아래로 그래프의 노드를 따라 질문.
  ✓ 결국 각 관찰은 그림 아래쪽에 있는 예 중 하나에 속함.
  ✓ 상자 그림으로 표시된 아래쪽의 각 노드는 다른 노드들과는 분포가 다름.
  ✓ 각 상자의 중간 값이나 평균값은 각 경우에 대한 어느 정도 합리적 예측이라고 볼 수 있음.

7.3 데이터 분석하기

- 회귀 트리 피팅하기

✓ 의사 결정 트리 학습 알고리즘을 구현한 패키지는 많음.
✓ 이 절에서는 party(https://cran.r-project.org/web/packages/party)라고 하는 간단한 패키지를 사용함.
✓ 아직 설치하지 않았다면 install.packages("party")를 실행하자.
✓ 이제 회귀 트리 모델을 학습하는 데 동일한 포뮬라와 데이터를 사용.
✓ ctree는 응답 변수에 실측 값을 허용하지 않으므로 air_time 값이 있는 하위 집합을 가져옴.

```
> model4 <- party::ctree(air_time ~ distance + month + dep_time,
+     data = subset(flights_train, !is.na(air_time)))
> predict4_train <- predict(model4, flights_train)
> error4_train <- flights_train$air_time - predict4_train[, 1]
> evaluate_error(error4_train)
  abs_err   std_dev
 7.418982 10.296528
```

✓ model4가 model3보다 성능이 더 좋아 보임.

7.3 데이터 분석하기

- 회귀 트리 피팅하기

✓ 표본 외 성능을 확인해 보자.

```
> predict4_test <- predict(model4, flights_test)
> error4_test <- flights_test$air_time - predict4_test[, 1]
> evaluate_error(error4_test)
  abs_err   std_dev
 7.499769 10.391071
```

✓ 결과는 회귀 트리가 평균적으로 이 문제에서 더 나은 예측을 할 수 있다는 것을 보여줌.

7.3 데이터 분석하기

- 회귀 트리 피팅하기

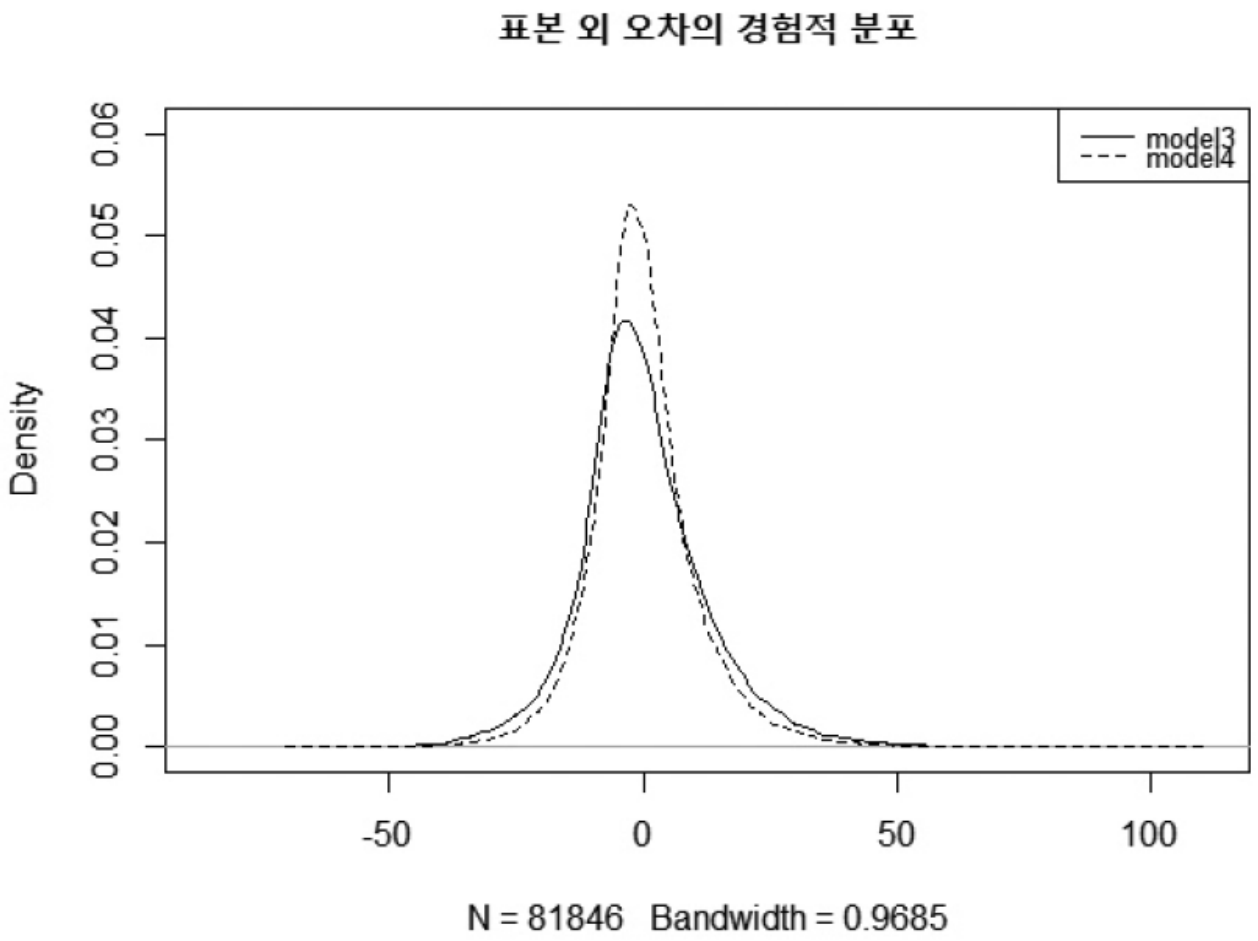
✓ 다음 밀도 그래프는 model3과 model4의 표본 외 예측 오차 분포 차이를 보여줌.

```
> plot(density(error3_test, na.rm = TRUE),
+      ylim = range(0, 0.06),
+      main = "표본 외 오차의 경험적 분포")
> lines(density(error4_test, na.rm = TRUE), lty = 2)
> legend("topright", legend = c("model3", "model4"),
+      lty = c(1, 2), cex = 0.8,
+      x.intersp = 0.6, y.intersp = 0.6)
```

✓ 뒤 그래프를 살펴보면 model4의 예측 오차 분산이 model3보다 작다는 것을 알 수 있음.

7.3 데이터 분석하기

그림 7-35 model3와 model4의 표본 외 오차 경험적 분포



7.3 데이터 분석하기

- 회귀 트리 피팅하기

  - ✓ 이 예제에서는 데이터를 깊이 있게 검사하지 않고 바로 선형 모델과 머신 러닝 모델을 적용했기 때문에 많은 문제가 생길 수 있음.
  - ✓ 이 절의 요점은 모델에 관한 것이 아니라 R에서 모델을 피팅하는 일반적인 절차와 인터페이스를 보여 주는 것.
  - ✓ 실제 문제에서는 데이터를 무작정 임의의 모델에 넣어 결론을 내기보다는 조심스럽게 분석할 필요가 있음.

7.4 마치며

7.4 마치며

- 마치며

  - ✓ 다양한 형식으로 데이터를 읽고 쓰는 방법, 플롯 함수를 사용하여 데이터를 시각화하는 방법, 데이터에 기본 모델을 적용하는 방법을 배움.
  - ✓ 이제 데이터 작업에 필요한 기본 도구와 인터페이스도 알았음.
  - ✓ 다른 자료를 참고하여 더 많은 데이터 분석 도구를 배울 수도 있음.
  - ✓ 통계 및 계량경제학 모델은 참고서뿐 아니라 통계 분석에 초점을 맞춘 R 관련 도서를 읽는 것을 권장함.
  - ✓ 인공 신경망, 서포트 벡터 머신, 랜덤 포레스트 같은 머신 러닝 모델은 머신 러닝 관련 도서를 읽는 것을 권장함.
  - ✓ <CRAN Task View: Machine Learning & Statistical Learning(CRAN 작업 보기:머신 러닝 & 통계 학습)>(https://cran.rproject.org/web/views/MachineLearning.html)을 참고하길 추천함