

# R을 이용한 데이터 분석 실무

## R for Practical Data Analysis

서민구  
[minkoo.seo@gmail.com](mailto:minkoo.seo@gmail.com)

이 책의 최신판은 <http://r4pda.co.kr/> 또는  
<http://ihelp.r-forge.r-project.org/contributedDocs.html>에서 보실 수 있습니다.  
본 문서의 최종 수정일은 2014년 3월 2일입니다.

## 저작권 고지

이 책은 무료로 인터넷에 공개됩니다. 그러나 무료 배포가 저작권의 소멸을 뜻하지는 않으며, 저자는 이 책의 저작권을 갖습니다. 따라서 대한민국 저작권법에 따른 보호를 받습니다.

## 저자에게 피드백을 제공하는 방법

이 책에 있는 오류 또는 개선 건의 사항에 대한 의견은 [ihelp-r4pda@lists.r-forge.r-project.org](mailto:ihelp-r4pda@lists.r-forge.r-project.org)로 이메일을 주시기 바랍니다.

# 차례

<b>제1장 서문</b>	<b>16</b>
1 왜 R인가 . . . . .	16
2 이 책의 목적 . . . . .	17
<b>제2장 R 시작하기</b>	<b>18</b>
1 설치 . . . . .	18
1.1 Windows 상에서의 설치 . . . . .	19
1.2 Linux 에서의 설치 . . . . .	20
1.3 OSX 상에서의 설치 . . . . .	25
2 R 시작하기 . . . . .	27
3 도움말 보기 . . . . .	28
4 IDE . . . . .	30
5 배치 실행 . . . . .	33
6 패키지의 사용 . . . . .	34
<b>제3장 데이터 타입</b>	<b>36</b>
1 변수 . . . . .	36
2 스칼라 . . . . .	37
2.1 숫자 . . . . .	37
2.2 NA . . . . .	37
2.3 NULL . . . . .	38
2.4 문자열 . . . . .	38
2.5 진리값 . . . . .	38
2.6 요인(Factor) . . . . .	40
3 벡터(Vector) . . . . .	41
3.1 벡터의 정의 . . . . .	41
3.2 벡터내 데이터 접근 . . . . .	43

---

3.3 벡터 연산 . . . . .	44
3.4 seq . . . . .	45
3.5 rep . . . . .	45
4 리스트(List) . . . . .	46
4.1 리스트의 정의 . . . . .	46
4.2 리스트내 데이터 접근 . . . . .	47
5 행렬(Matrix) . . . . .	48
5.1 행렬의 정의 . . . . .	48
5.2 행렬내 데이터 접근 . . . . .	49
5.3 행렬의 연산 . . . . .	50
6 배열 . . . . .	53
6.1 배열 정의 . . . . .	53
6.2 배열 데이터 접근 . . . . .	54
7 데이터 프레임(Data Frame) . . . . .	54
7.1 데이터 프레임 정의 . . . . .	55
7.2 데이터 프레임 접근 . . . . .	56
8 타입 판별 . . . . .	59
9 타입 변환 . . . . .	60
<b>제 4 장 제어문, 연산, 함수</b>	<b>63</b>
1 IF, FOR, WHILE 문 . . . . .	63
2 행렬 연산 . . . . .	64
3 NA의 처리 . . . . .	65
4 함수의 정의 . . . . .	66
5 스코프(Scope) . . . . .	68
6 벡터 연산 . . . . .	71
7 값에 의한 전달 . . . . .	73
8 객체의 불변성 . . . . .	75
9 모듈(Module) 패턴 . . . . .	76
9.1 큐(Queue) . . . . .	77
9.2 큐(Queue) 모듈의 작성 . . . . .	78
10 객체의 삭제 . . . . .	81
<b>제 5 장 데이터 조작 I</b>	<b>82</b>
1 iris 데이터 . . . . .	82

---

2 파일 입출력 . . . . .	84
2.1 CSV파일 입출력 . . . . .	84
3 save(), load() . . . . .	88
4 rbind(), cbind() . . . . .	88
5 apply 함수들 . . . . .	90
5.1 apply() . . . . .	90
5.2 lapply() . . . . .	92
5.3 sapply() . . . . .	95
5.4 tapply . . . . .	97
5.5 mapply() . . . . .	99
6 doBy 패키지 . . . . .	99
7 split() . . . . .	104
8 subset() . . . . .	105
9 merge() . . . . .	106
10 sort(), order() . . . . .	108
11 with(), within() . . . . .	109
12 attach(), detach() . . . . .	112
13 which(), which.max(), which.min() . . . . .	113
14 aggregate() . . . . .	114
15 stack(), unstack() . . . . .	114
16 RMySQL 패키지 . . . . .	116
<b>제 6 장 데이터 조작 II</b> . . . . .	<b>119</b>
1 sqldf 패키지 . . . . .	119
2 plyr 패키지 . . . . .	121
2.1 adply() . . . . .	122
2.2 ddply() . . . . .	124
2.3 transform(), summarise(), subset() . . . . .	126
2.4 m*ply() . . . . .	129
3 reshape2 패키지 . . . . .	130
3.1 melt() . . . . .	130
3.2 dcast() . . . . .	133
4 data.table 패키지 . . . . .	139
4.1 데이터 테이블 생성 . . . . .	139
4.2 데이터 접근과 그룹 연산 . . . . .	140

---

4.3	key를 사용한 탐색 . . . . .	144
4.4	key를 사용한 데이터 테이블 병합 . . . . .	145
4.5	참조를 사용한 데이터 수정 . . . . .	147
4.6	rbindlist . . . . .	148
5	foreach . . . . .	152
6	doMC . . . . .	154
6.1	프로세스의 수 설정 . . . . .	154
6.2	plyr의 .parallel 옵션 . . . . .	155
6.3	foreach에서 %dopar%의 사용 . . . . .	157
7	테스팅과 디버깅 . . . . .	157
7.1	testthat . . . . .	157
7.2	test_that을 사용한 테스트 그룹화 . . . . .	158
7.3	테스트 파일 구조 . . . . .	160
7.4	디버깅 . . . . .	161
8	section:browser . . . . .	165
9	코드 수행 시간 측정 . . . . .	168
9.1	system.time()을 사용한 시간 측정 . . . . .	168
9.2	Rprof()를 사용한 코드 프로파일링 . . . . .	169

<b>제 7 장</b>	<b>그래프</b>	<b>172</b>
1	산점도 . . . . .	172
2	그래픽 옵션 . . . . .	174
2.1	축 이름(xlab, ylab) . . . . .	174
2.2	그래프 제목(main) . . . . .	175
2.3	점의 종류(pch) . . . . .	176
2.4	점의 크기(cex) . . . . .	177
2.5	색상(col) . . . . .	178
2.6	좌표축 값의 범위(xlim, ylim) . . . . .	179
2.7	type . . . . .	180
3	그래프의 배열(mfrow) . . . . .	183
4	지터(jitter) . . . . .	184
5	점(points) . . . . .	185
6	선(lines) . . . . .	187
7	직선(abline) . . . . .	189
8	곡선(curve) . . . . .	191

---

9	다각형(polygon) . . . . .	192
10	문자열(text) . . . . .	195
11	그래프상에 그려진 데이터의 식별 . . . . .	196
12	범례(legend) . . . . .	197
13	행렬에 저장된 데이터 그리기(matplot, matlines, matpoints) . . . . .	198
14	상자 그림(boxplot) . . . . .	199
15	히스토그램(hist) . . . . .	203
16	밀도 그림(density) . . . . .	206
17	막대 그림(barplot) . . . . .	208
18	파이 그래프(pie) . . . . .	209
19	모자이크 플롯(mosaicplot) . . . . .	211
20	산점도 행렬(pairs) . . . . .	214
21	투시도(persp), 등고선 그래프(contour) . . . . .	216
<b>제 8 장 통계 분석</b>		<b>220</b>
1	난수 생성 및 분포 함수 . . . . .	220
2	기초 통계량 . . . . .	223
2.1	평균, 표본 분산, 표본 표준편차 . . . . .	223
2.2	다섯 수치 요약 . . . . .	224
2.3	최빈값(mode) . . . . .	225
3	표본추출 . . . . .	226
3.1	단순 임의 추출(Random Sampling) . . . . .	226
3.2	총화 임의 추출(Stratified Random Sampling) . . . . .	226
3.3	계통 추출(Systematic Sampling) . . . . .	229
4	분할표(Contingency Table) . . . . .	231
4.1	분할표의 작성 . . . . .	231
4.2	독립성 검정(Independence Test) . . . . .	233
4.3	피셔의 정확 검정(Fisher's Exact Test) . . . . .	235
4.4	맥니마 검정(McNemar Test) . . . . .	236
5	적합도 검정(Goodness of Fit) . . . . .	239
5.1	Chi Square Test . . . . .	239
5.2	Shapiro-Wilk Test . . . . .	239
5.3	Kolmogorov-Smirnov Test . . . . .	240
5.4	Q-Q Plot . . . . .	241

---

6	상관 계수 . . . . .	244
6.1	피어슨 상관계수(Pearson Correlation Coefficient) . . . . .	245
6.2	스피어만 상관계수(Spearman's Rank Correlation Coefficient) . . . . .	247
6.3	肯달의 순위 상관 계수(Kendal's Rank Correlation Coefficient) . . . . .	248
6.4	상관 계수 검정(Correlation Test) . . . . .	249
7	추정 및 검정 . . . . .	250
7.1	일표본 평균 . . . . .	251
7.2	독립 이표본 평균 . . . . .	253
7.3	짝지은 이표본 평균 . . . . .	257
7.4	이표본 분산 . . . . .	259
7.5	일표본 비율 . . . . .	260
7.6	이표본 비율 . . . . .	262
제9장 선형 회귀(Linear Regression) . . . . .		264
1	단순 선형 회귀(Simple Linear Regression) . . . . .	264
1.1	모델 생성 . . . . .	265
1.2	선형회귀 결과 추출 . . . . .	266
1.3	예측과 신뢰구간 . . . . .	267
1.4	모형 평가 . . . . .	269
1.5	ANOVA 및 모델간의 비교 . . . . .	271
1.6	모델 평가 차트 . . . . .	273
1.7	회귀 직선의 시각화 . . . . .	275
2	중선형회귀(Multiple Linear Regression) . . . . .	277
2.1	모델 생성 및 평가 . . . . .	277
2.2	범주형 변수 . . . . .	279
2.3	중선형회귀모형의 시각화 . . . . .	282
2.4	표현식을 위한 I()의 사용 . . . . .	284
2.5	변수의 변환 . . . . .	286
2.6	상호 작용 . . . . .	287
3	이상치(outlier) . . . . .	293
4	변수 선택 . . . . .	295
4.1	단계적 변수 선택 . . . . .	295
4.2	모든 경우에 대한 비교 . . . . .	300

---

<b>제 10 장 분류 알고리즘(Classification Algorithms)</b>	<b>303</b>
1 데이터 탐색 . . . . .	303
1.1 기술 통계 . . . . .	304
1.2 데이터 시각화 . . . . .	306
2 전처리(Preprocessing) . . . . .	311
2.1 데이터 변환 . . . . .	311
2.2 결측값(NA)의 처리 . . . . .	315
2.3 변수 선택(Feature Selection) . . . . .	318
3 모델 평가 방법 . . . . .	323
3.1 평가 메트릭(metric) . . . . .	323
3.2 ROC 커브 . . . . .	325
3.3 교차 검증(cross validation) . . . . .	328
4 로지스틱 회귀모형(Logistic Regression) . . . . .	336
5 다항 로지스틱 회귀분석(Multinomial Logistic Regression) . . . . .	340
6 나무 모형(Tree Models) . . . . .	343
6.1 rpart . . . . .	343
6.2 party::ctree . . . . .	347
6.3 Random Forest . . . . .	348
7 신경망(Neural Networks) . . . . .	354
7.1 Formula를 사용한 모델 생성 . . . . .	354
7.2 X와 Y의 직접 지정 . . . . .	355
8 SVM(Support Vector Machine) . . . . .	357
9 클래스 불균형(Class Imbalance) . . . . .	359
10 문서 분류(Document Classification) . . . . .	364
10.1 코퍼스와 문서 . . . . .	364
10.2 문서 변환 . . . . .	365
10.3 문서의 행렬 표현 . . . . .	366
10.4 문서 분류 . . . . .	369
10.5 파일로부터 Corpus 생성 . . . . .	371
10.6 메타 데이터 . . . . .	373
<b>제 11 장 타이타닉 데이터를 사용한 머신 러닝 연습</b>	<b>378</b>
1 타이타닉 데이터 형식 . . . . .	378
2 데이터 불러오기 . . . . .	379
2.1 데이터 탑입 지정 . . . . .	380

---

2.2 테스트 데이터(Test Data)의 분리 . . . . .	382
2.3 교차 검증 준비 . . . . .	383
3 데이터 탐색 . . . . .	386
4 평가 메트릭 . . . . .	391
5 rpart 모델 . . . . .	392
5.1 rpart의 교차 검증 . . . . .	392
5.2 Accuracy 평가 . . . . .	394
6 ctree 모델 . . . . .	395
7 또 다른 특징(Feature)의 발견 . . . . .	396
7.1 ticket을 사용한 가족 식별 . . . . .	397
7.2 생존 확률 예측 . . . . .	398
7.3 가족 ID 부여 . . . . .	399
7.4 가족 구성원 생존 확률의 병합 . . . . .	401
7.5 가족 정보를 사용한 ctree() 모델링 . . . . .	404
7.6 성능 평가 . . . . .	406
8 교차 검증의 병렬화(Parallelization) . . . . .	409
8.1 10겹 교차 검증의 3회 반복 수행 . . . . .	409
8.2 foreach()와 %dopar%를 사용한 병렬화 . . . . .	411

# 그림 차례

2.1	www.r-project.org에서 R 다운로드 . . . . .	18
2.2	Windows 상에서 R의 실행 화면 . . . . .	19
2.3	Ubuntu 다운로드 링크 . . . . .	20
2.4	Ubuntu 다운로드 링크 . . . . .	21
2.5	App Store에서 XCode 검색 . . . . .	26
2.6	Command Line Tools 설치 . . . . .	26
2.7	R GUI 작동화면 . . . . .	31
2.8	RStudio . . . . .	32
2.9	Vim R Plugin . . . . .	33
4.1	객체의 불변성 . . . . .	75
6.1	.parallel=TRUE를 지정한 멀티코어의 활용중 thrashing이 발생한 경우 . . . . .	156
7.1	Sandburg(V8)와 El Monte(V9)지역의 온도 . . . . .	174
7.2	xlab, ylab의 사용 . . . . .	175
7.3	main을 사용한 타이틀 지정 . . . . .	176
7.4	pch를 사용한 점 모양의 지정 . . . . .	177
7.5	cex=0.1을 사용한 점의 크기 조정 . . . . .	178
7.6	col='FF0000'을 사용한 점 색상의 지정 . . . . .	179
7.7	xlim, ylim의 사용 . . . . .	180
7.8	plot(cars) . . . . .	181
7.9	plot(cars, type="l") . . . . .	181
7.10	plot(cars, type="o") . . . . .	182
7.11	주행속도별 평균 제동거리 . . . . .	183
7.12	par(mfrow=c(1, 2)) . . . . .	184
7.13	jitter() . . . . .	185
7.14	points() . . . . .	186
7.15	plot(type="n") . . . . .	187

---

7.16	lines() . . . . .	188
7.17	lines() . . . . .	189
7.18	abline(a=-5, b=3.5) . . . . .	190
7.19	abline(h=...), abline(v=...) . . . . .	191
7.20	curve()를 사용해 그린 sin 곡선 . . . . .	192
7.21	polygon() . . . . .	195
7.22	text() . . . . .	196
7.23	identify() 호출 후 몇개의 점을 클릭한 결과 . . . . .	197
7.24	legend() . . . . .	198
7.25	matplot() . . . . .	199
7.26	boxplot() . . . . .	200
7.27	boxplot()에 text()를 사용한 outlier 표시 . . . . .	202
7.28	두개의 boxplot . . . . .	203
7.29	hist() . . . . .	204
7.30	hist(x, freq=FALSE) . . . . .	205
7.31	plot(density(x)) . . . . .	207
7.32	밀도그림과 히스토그램 . . . . .	207
7.33	rug() . . . . .	208
7.34	barplot() . . . . .	209
7.35	pie() . . . . .	211
7.36	mosaicplot() . . . . .	213
7.37	mosaicplot(formula, data) . . . . .	214
7.38	pairs() . . . . .	215
7.39	persp() . . . . .	218
7.40	contour() . . . . .	219
8.1	rnorm()으로 구한 샘플의 밀도 그림 . . . . .	222
8.2	사건 전후의 Test 결과 . . . . .	237
8.3	rnorm() 데이터에대한 정규화플롯 . . . . .	243
8.4	rcauchy() 데이터에대한 정규화플롯 . . . . .	244
8.5	iris 데이터로 그린 corrgram . . . . .	246
9.1	선형 모델의 평가 . . . . .	273
9.2	Cook's Distance와 Leverage . . . . .	275
9.3	Cars 데이터와 회귀직선 . . . . .	276
9.4	회귀직선의 신뢰대(Confidence Band) . . . . .	277

9.5	iris 데이터에 대한 Species별 선형회귀모형의 시각화 . . . . .	284
9.6	변수간의 상호작용과 그에 따른 선형 회귀 모형 . . . . .	287
9.7	Tree에 따른 circumference의 상자 그림 . . . . .	290
9.8	Tree, age, circumference의 상호 작용 그림(interaction plot) . . . . .	290
9.9	regsubsets() 결과로부터 구한 Adjusted R squared . . . . .	302
10.1	plot(iris) . . . . .	306
10.2	plot(iris\$Sepal.Length) . . . . .	307
10.3	plot(iris\$Species) . . . . .	308
10.4	plot()과 formula의 사용 . . . . .	309
10.5	plot()과 col의 사용 . . . . .	310
10.6	featurePlot()을 iris에 적용한 예 . . . . .	311
10.7	iris데이터의 PCA후 Scree Plot . . . . .	313
10.8	ROCR 패키지를 사용한 ROC 커브 . . . . .	326
10.9	ROCR 패키지를 사용한 Accuracy/Cutoff 차트 . . . . .	327
10.10	과적합의 예 . . . . .	328
10.11	훈련데이터와 검증 데이터를 사용한 모델링 과정 . . . . .	330
10.12	8겹 교차검증에서 K=1, 2, 3일 때 검증 데이터 . . . . .	331
10.13	iris 데이터에 대한 rpart() 수행 결과 . . . . .	345
10.14	prp()를 사용한 rpart 시각화 . . . . .	346
10.15	iris 데이터에 대한 ctree() 수행 결과 . . . . .	348
10.16	randomForest를 사용한 변수 중요도 평가 . . . . .	351
11.1	View(titanic)의 실행화면 . . . . .	380
11.2	featurePlot()의 실행 결과 . . . . .	389
11.3	mosaicplot()의 실행 결과 . . . . .	390
11.4	rpart()와 ctree()의 Accuracy 비교 . . . . .	396

# 표 차례

5.1 대조군, 실험군에서의 약물 반응 결과 . . . . .	115
8.1 확률 분포 및 난수 발생 함수 . . . . .	221
8.2 확률 분포 및 관련 함수 . . . . .	222
9.1 가변수를 사용한 범주형 변수의 표현 . . . . .	280
11.1 타이타닉 데이터 . . . . .	378

## 1 왜 R인가

R[1]은 데이터 분석을 위한 통계 및 그래픽스를 지원하는 자유 소프트웨어 환경이다. 그 뿐만 아니라 벨 연구소에서 만들어진 통계 분석 언어 S에 두고 있는데, R은 S언어를 근간으로 뉴질랜드의 University of Auckland에서 Ross Ihaka와 Robert Gentleman이 만든것이 그 시작이다.

R은 현재 데이터 분석을 위한 도구로 많은 인기를 누리고 있다. 한가지 사례로 kdnugget에서 실시한 ‘지난 12개월간 실제로 사용한 분석, 데이터 마이닝, 빅 데이터 소프트웨어’에 대한 설문 조사를 볼 수 있다.<sup>1)</sup> 그 내용에 따르면 R은 2012년 현재 Rapid Miner, Weka, SAS, MATLAB 등의 쟁쟁한 경쟁자를 물리치고 데이터 분석 소프트웨어 1위로 자리매김하고 있다.

R은 하나의 컴퓨터 언어이자 다양한 패키지(또는 라이브러리)의 집합이다. 따라서 자유롭게 데이터 분석을 R안에서 수행할 수 있다는 장점이 있다. 또한 R은 통계, 머신러닝, 금융, 바이오인포머틱스, 그래픽스에 이르는 다양한 통계패키지를 갖고 있으며 이 모든 것이 무료로 제공된다. 거기에 더해 최근 시류에 발맞춰 R은 멀티프로세서에서 손쉽게 병렬화하여 실행할 수 있고, RHive를 사용하면 최근 인기를 끌고 있는 Hive 환경에서 R을 사용할 수 있다.

인기가 좋다는 것은 알겠는데, 그렇다고 꼭 R을 배워야 할까. 다른 언어 또는 환경은 없을까. 물론 다른 언어나 환경도 있다. WEKA는 자바로 작성된 데이터 마이닝 소프트웨어이며 자바 언어에서 연결해 사용하기에 편리하다. 또한 매우 훌륭한 책[2]이 나와있어 쉽게 시작할 수 있는 환경이다. 그러나 WEKA는 기본적으로 GUI를 사용한 도구이지 하나의 언어체계는 아니다.

Python에는 pydata라 불리는 numpy, scipy, pandas, matplotlib, scikit-learn 의 라이브러리들이 있다. Python for Data Analysis[3]처럼 numpy, pandas 등을 잘 설명한 책도 있다. 그러나 pydata에는 R의 다양한 통계 기능에 대응되는 기능이 없다. 비록 statsmodel 등의 라이브러리가 있으나 문서화의 수준이 빈약하다.

R이 좋은 이유가 무엇인지에 대한 질문에 딱 한가지만 답해야 한다면, 필자는 셀수 없이

<sup>1)</sup><http://www.kdnuggets.com/polls/2012/analytics-data-mining-big-data-software.html>

많은 R을 사용한 통계 분석 서적, 머신러닝과 관련한 서적을 들고싶다. 기초부터 발전된 주제까지 포괄한 서적과 문서화덕에 이론과 실제를 배울 수 있는 환경이 가장 잘 갖추어져있는 것이 R이다.

## 2 이 책의 목적

이 책은 소프트웨어 개발에 능숙하며 통계 및 머신 러닝 기법을 알고 있는 독자들이 빠르게 R을 배울 수 있게 도와주는 목적으로 쓰여졌다. 따라서 가장 중요하다고 생각되는 R의 함수나 패키지에 대해서만 다루고, 이들을 짧은 시간내에 살펴볼 수 있게 했다. 또 코드는 가능한 그 자체로 독립적이게 했고, 코드 실행 화면을 충실히 포함시키기 위해 애썼다.

한편 이런 목적을 만족시키기위해 이 책에서는 프로그래밍 언어의 기본적인 내용인 변수, 변수의 스코프, 반복문의 의미와 같은 기초적인 프로그래밍 개념은 설명하고 있지 않다. 또한 통계나 머신 러닝의 원리에 대한 설명 역시 생략되어 있다. 마지막으로 이 책은 R언어의 내부 구조나, R언어를 사용한 라이브러리를 만드는 방법을 다루고 있지 않다.

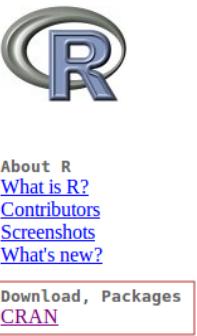
이 책은 계속적으로 갱신되며 <http://r4pda.co.kr/>에서 최근 파일을 다운로드 받을 수 있다.

## R 시작하기

### 1 설치

R은 공개 소프트웨어로 <http://www.r-project.org/>에서 다운로드 받아 설치가 가능하다. 만약 Ubuntu나 Mac을 사용하고 있다면 apt-get, brew 등으로 설치하면 된다.

브라우저에서 <http://www.r-project.org/>을 연 뒤 화면 왼쪽의 "Download, Packages" 밑에 있는 CRAN을 클릭한다. 그 뒤 나오는 목록에서 Korea 밑에 있는 링크를 클릭한다. 그림 2.1을 참고하기 바란다.



<a href="#">About R</a> <a href="#">What is R?</a> <a href="#">Contributors</a> <a href="#">Screenshots</a> <a href="#">What's new?</a>  <a href="#">Download, Packages</a> <b>CRAN</b>	Ireland <a href="http://cran.um.ac.ir/">http://cran.um.ac.ir/</a> Italy <a href="http://ftp.heanet.ie/mirrors/cran.r-project.org/">http://ftp.heanet.ie/mirrors/cran.r-project.org/</a> Japan <a href="http://cran.mirror.garr.it/mirrors/CRAN/">http://cran.mirror.garr.it/mirrors/CRAN/</a> <a href="http://cran.stat.unipd.it/">http://cran.stat.unipd.it/</a> <a href="http://dssm.unipa.it/CRAN/">http://dssm.unipa.it/CRAN/</a> Korea <a href="http://cran.nexr.com/">http://cran.nexr.com/</a> <a href="http://biostat.cau.ac.kr/CRAN/">http://biostat.cau.ac.kr/CRAN/</a> Lebanon <a href="http://rmirror.lau.edu.lb/">http://rmirror.lau.edu.lb/</a> Mexico <a href="http://cran.itam.mx/">http://cran.itam.mx/</a> <a href="http://www.est.colpos.mx/R-mirror/">http://www.est.colpos.mx/R-mirror/</a> Netherlands	Ferdowsi University of Mashhad HEAnet, Dublin Garr Mirror, Milano University of Padua Universita degli Studi di Palermo Hyogo University of Teacher Education Institute of Statistical Mathematics, Tokyo University of Tsukuba NexR Corporation, Seoul Chung-Ang University, Seoul Lebanese American University, Byblos Instituto Tecnologico Autonomo de Mexico Colegio de Postgraduados, Texcoco
--	--	---

그림 2.1: www.r-project.org에서 R 다운로드

그 뒤 화면에서 자신의 운영체제에 맞는 버전을 선택해 다운로드 받아 설치한다.

## 1.1 Windows 상에서의 설치

다음은 Windows를 선택한 경우의 화면이다. 그림 ??에서 base 링크를 클릭해 설치 파일을 다운로드 받은 뒤 설치를 진행한다.

설치 화면에서는 별다른 옵션 선택없이 ‘다음’만 선택해서 설치해도 된다. 설치 후 바탕화면의 R 아이콘을 클릭하면 R GUI 화면이 뜰 것이다.

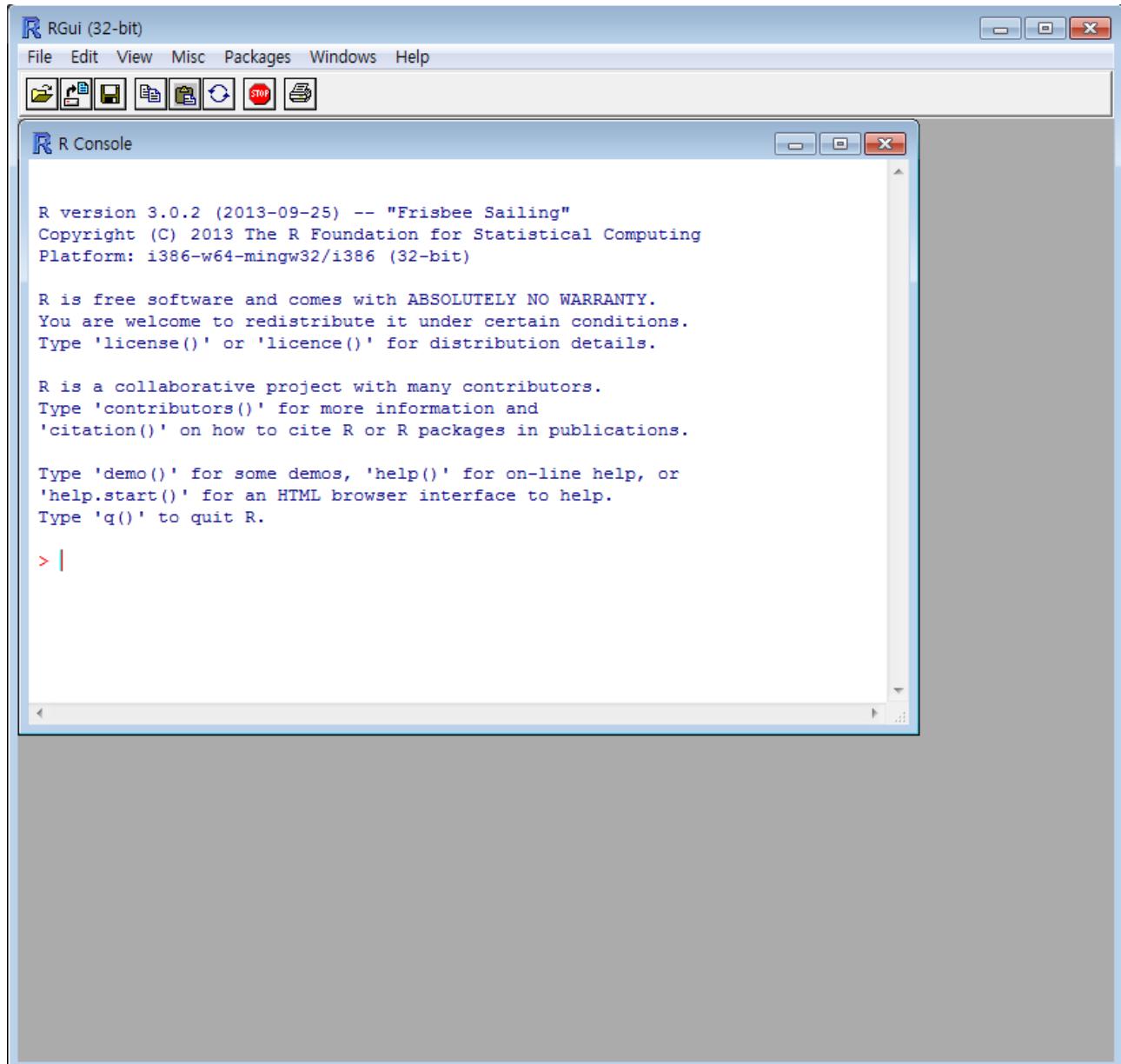


그림 2.2: Windows 상에서 R의 실행 화면

## 1.2 Linux에서의 설치

Linux 배포판 중 하나인 Ubuntu에서도 2.1에 있는 다운로드 화면에서 ‘Download R for Linux’를 클릭해 설치한다. 링크를 클릭한 뒤 그림 2.3에 있는 ubuntu 링크를 클릭한다.

### Index of /bin/linux

Name	Last modified	Size	Description
 <a href="#">Parent Directory</a>		-	
 <a href="#">debian/</a>	26-Oct-2013 00:14	-	
 <a href="#">redhat/</a>	26-Nov-2009 02:01	-	
 <a href="#">suse/</a>	16-Feb-2012 23:09	-	
 <a href="#">ubuntu/</a>	18-Oct-2013 11:03	-	

*Apache/2.2.3 (CentOS) Server at cran.nexr.com Port 80*

그림 2.3: Ubuntu 다운로드 링크

링크를 클릭하면 그림 2.4에 보인 것과 같은 apt-get 의 소스 목록이 나타난다. 이들 중 자신의 배포판에 맞는 항목을 골라 apt-get 소스 파일을 업데이트 해야한다.

## INSTALLATION

To obtain the latest R packages, add an entry like

```
deb http://<my.favorite.cran.mirror>/bin/linux/ubuntu saucy/
```

or

```
deb http://<my.favorite.cran.mirror>/bin/linux/ubuntu raring/
```

or

```
deb http://<my.favorite.cran.mirror>/bin/linux/ubuntu quantal/
```

or

```
deb http://<my.favorite.cran.mirror>/bin/linux/ubuntu precise/
```

or

```
deb http://<my.favorite.cran.mirror>/bin/linux/ubuntu lucid/
```

그림 2.4: Ubuntu 다운로드 링크

예를 들어 Ubuntu precise의 사용자는 다음과 같이 /etc/apt/sources.list.d 디렉토리 밑에 r.list라는 파일을 작성한다.

```
$ cd /etc/apt/sources.list.d
$ cat > r.list
deb http://cran.nexr.com/bin/linux/ubuntu precise/
^D (CTRL+D)
```

설치에 필요한 명령들은 다음과 같다. 이 중 apt-key는 새로 추가한 apt-get 소스를 접근하기 위해 필요한 명령으로 Linux 다운로드 페이지 하단에 설명되어 있다.

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
E084DAB9
$ sudo apt-get update
$ sudo apt-get install r-base
```

설치가 완료되었다면 콘솔에서 R 을 입력해 잘 실행되는지 확인한다.

```
$ R
```

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

이것으로 기본적인 설치는 끝났지만 R의 퍼포먼스를 더 높힐 수 있는 방법이 있어 소개한다. 리눅스에서는 BLAS(Basic Linear Algebra System)의 구현체로 레퍼런스구현(libblas), ATLAS 구현(libatlas), OpenBLAS 구현(libopenblas)가 있다. 이들 목록은 다음과 같은 명령으로 찾아볼 수 있다.

```
$ sudo apt-cache search --names-only libblas
libblas-dev - Basic Linear Algebra Subroutines 3, static library
libblas-doc - Basic Linear Algebra Subroutines 3, documentation
libblas3gf - Basic Linear Algebra Reference implementations, shared
             library
libatlas-base-dev - Automatically Tuned Linear Algebra Software,
                     generic static
libatlas3gf-base - Automatically Tuned Linear Algebra Software,
                     generic shared
libblas-test - Basic Linear Algebra Subroutines 3, testing programs
```

```
libopenblas-base - Optimized BLAS (linear algebra) library based on
  GotoBLAS2
libopenblas-dev - Optimized BLAS (linear algebra) library based on
  GotoBLAS2
```

BLAS 구현중 어느것을 사용하는지에따라 성능이 크게 차이 날 수 있다. 설정 변경에 앞서 R의 벤치마크 프로그램을 다운로드 받아 실행해보자.

```
$ wget http://r.research.att.com/benchmarks/R-benchmark-25.R
$ cat R-benchmark-25.R | time R --slave
```

필자의 머신에서 실행결과는 다음과 같다.

```
Loading required package: Matrix
Loading required package: SuppDists
Warning messages:
1: In remove("a", "b") : object 'a' not found
2: In remove("a", "b") : object 'b' not found

R Benchmark 2.5
=====
Number of times each test is run-----: 3

I. Matrix calculation
-----
Creation, transp., deformation of a 2500x2500 matrix (sec):0.9046666666666667
2400x2400 normal distributed random matrix ^1000---- (sec):0.5706666666666667
Sorting of 7,000,000 random values----- (sec):0.6710000000000001
2800x2800 cross-product matrix (b = a' * a)----- (sec):1.866
Linear regr. over a 3000x3000 matrix (c = a \ b')--- (sec):1.018
-----
Trimmed geom. mean (2 extremes eliminated) 0.851764683213596

II. Matrix functions
-----
FFT over 2,400,000 random values----- (sec):0.3840000000000002
Eigenvalues of a 640x640 random matrix----- (sec):0.7633333333333333
Determinant of a 2500x2500 random matrix----- (sec):1.122
Cholesky decomposition of a 3000x3000 matrix----- (sec):0.9120000000000004
Inverse of a 1600x1600 random matrix----- (sec):1.0506666666666667
-----
Trimmed geom. mean (2 extremes eliminated):0.900999755960927

III. Programming
-----
3,500,000 Fibonacci numbers calculation (vector calc)(sec):0.6346666666666665
```

```

Creation of a 3000x3000 Hilbert matrix (matrix calc) (sec):0.320333333333335
Grand common divisors of 400,000 pairs (recursion)--- (sec):0.886333333333335
Creation of a 500x500 Toeplitz matrix (loops)----- (sec):0.570666666666668
Escoufier's method on a 45x45 matrix (mixed)----- (sec):0.3539999999999999
-----
Trimmed geom. mean (2 extremes eliminated):0.504247575798334

Total time for all 15 tests----- (sec):12.0283333333333
Overall mean (sum of I, II and III trimmed means/3) (sec):0.728723386126362
--- End of test ---

```

실제 숫자는 사용자 머신에 따라 다르겠지만 위 수치와 비교해 너무 큰 값이 나온다면 BLAS 설정 변경으로 성능을 향상시킬 수도 있다. 필자의 머신에서는 libatlas의 성능이 좋았고 이를 설치하는 방법은 다음과 같다.

먼저 libatlas를 찾는다.

```

$ sudo apt-cache search --names-only libatlas
libatlas-base-dev - Automatically Tuned Linear Algebra Software, generic static
libatlas-cpp-0.6-1 - The protocol library of the World Forge project - runtime libs
libatlas-cpp-0.6-1-dbg - The protocol library of the World Forge project - debugging
    libs
libatlas-cpp-0.6-dev - The protocol library of the World Forge project - header files
libatlas-cpp-doc - The protocol library of the World Forge project - documentation
libatlas-dev - Automatically Tuned Linear Algebra Software, C header files
libatlas-doc - Automatically Tuned Linear Algebra Software, documentation
libatlas-test - Automatically Tuned Linear Algebra Software, test programs
libatlas3gf-base - Automatically Tuned Linear Algebra Software, generic shared

```

그리고 필요해 보이는 패키지들을 설치한다.

```
$ sudo apt-get install libatlas-base-dev libatlas-dev libatlas3gf-base
```

설치가 되면 우분투에서 다양한 구현체 중 하나의 구현체를 선택하게 해주는 update-alternatives 가 라이브러리의 우선순위에 따라 BLAS를 선택하게 된다. 만약 설치한 BLAS 구현체의 우선순위가 낮다면 선택이 안될 수도 있는데 이때는 직접 원하는 구현체를 선택할 수 있다.

```

$ sudo update-alternatives --config liblapack.so.3gf
There are 2 choices for the alternative liblapack.so.3gf (providing /usr/lib/
    liblapack.so.3gf).

      Selection    Path                                Priority      Status
      -----      -----
* 0            /usr/lib/atlas-base/atlas/liblapack.so.3gf    35          auto mode
    1            /usr/lib/atlas-base/atlas/liblapack.so.3gf    35          manual mode
    2            /usr/lib/lapack/liblapack.so.3gf                10          manual mode

```

```

Press enter to keep the current choice[*], or type selection number:

$ sudo update-alternatives --config libblas.so.3gf
There are 2 choices for the alternative libblas.so.3gf (providing /usr/lib/libblas.so
.3gf).

Selection      Path          Priority      Status
-----
* 0            /usr/lib/atlas-base/atlas/libblas.so.3gf    35      auto mode
  1            /usr/lib/atlas-base/atlas/libblas.so.3gf    35      manual mode
  2            /usr/lib/libblas/libblas.so.3gf           10      manual mode

Press enter to keep the current choice[*], or type selection number:

```

선택을 마쳤으면 벤치마킹을 다시 실행해 성능 향상 여부를 검토한다.

### 1.3 OSX 상에서의 설치

OSX는 다운로드 페이지에서 R-버전명.pkg 를 다운로드 받은 뒤 더블클릭하는 것만으로 간단히 설치할 수 있다.

또는 OSX 사용자들이 GNU 소프트웨어 설치 등을 위해 자주 사용하는 패키지 관리자 중 하나인 homebrew를 사용해 설치하는 것 역시 가능하다. homebrew를 사용해 설치하면 설치 파일이 /usr/local 로 분리되고 향후 업데이트도 brew update 명령등으로 간단히 수행할 수 있어 편리하다. 혹시 homebrew를 사용해 본 경험이 없는 OSX 사용자라면 이 기회에 배워보기 바란다.

homebrew 사용시 가장 먼저 해야 할 일은 XCode를 설치하는 것이다. XCode는 OSX 상의 App Store를 실행해 xcode를 검색한 뒤 그림 2.5에 있는 아이콘을 클릭해 설치한다. XCode는 용량이 커서 설치에 시간이 걸린다.

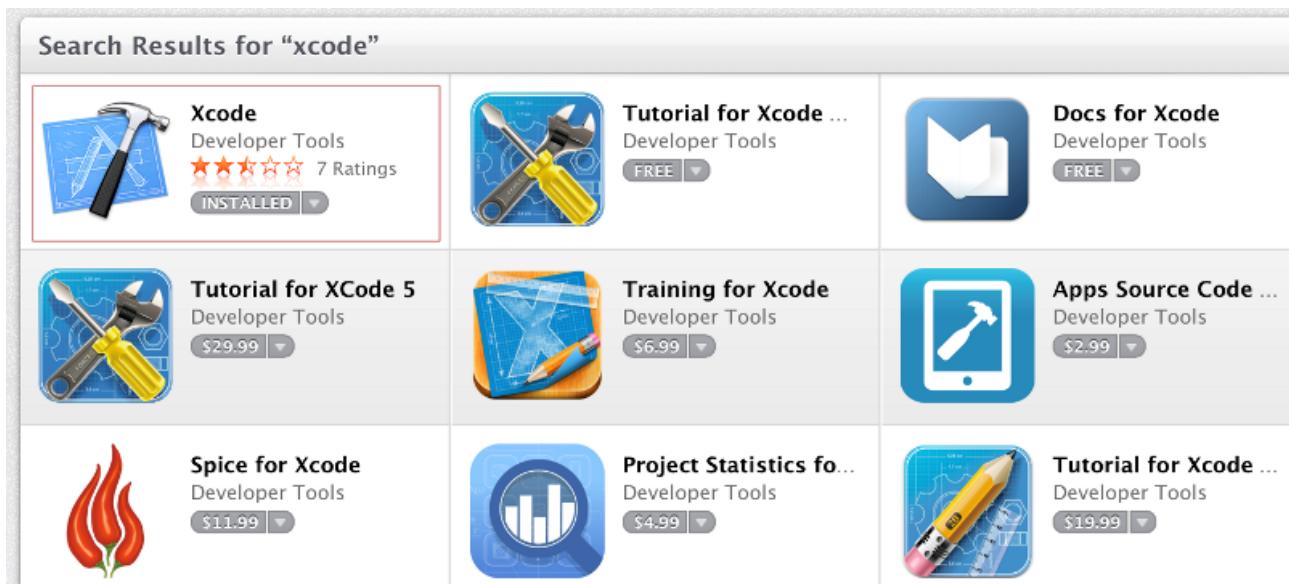


그림 2.5: App Store에서 XCode 검색

XCode 설치가 끝났으면 이를 실행해 잘 동작하는지 확인한다. 다음은 커맨드 라인에서 사용할 다양한 개발자 도구의 설치가 필요하다. <http://developer.apple.com/downloads>를 방문해 Apple 아이디로 로그인한다. 다음, 화면에 보여진 소프트웨어 목록에서 Command Line Tools를 다운로드 받아 설치한다. 이 때 자신의 운영체제에 맞는 파일을 잘 선택하기 바란다. 2.6에 보인 화면의 경우 Mountain Lion과 Mavericks 가 둘 다 나열된 것을 볼 수 있다.

The screenshot shows the Apple Developer Downloads page for Command Line Tools. The page lists several packages:

- Command Line Tools (OS X Mountain Lion) for Xcode - October (Oct 23, 2013)
- Kernel Debug Kit OS X Mavericks GM Seed build 13A603 (Oct 23, 2013)
- Xcode 5.0.1 (Oct 23, 2013)
- Graphics Tools for Xcode - October 2013 (Oct 23, 2013)
- Command Line Tools (OS X Mavericks) for Xcode - Late October (Oct 23, 2013)

The "Command Line Tools (OS X Mavericks) for Xcode - Late October 2013" item is expanded, showing its description and download link:

This package enables UNIX-style development via Terminal by installing command line developer tools, as well as OS X SDK frameworks and headers. Many useful tools are included, such as the Apple LLVM compiler, linker, and Make.

[Command Line Tools \(OS X Mavericks\) for Xcode - Late October 2013](#)  
.dmg(97.36 MB)

그림 2.6: Command Line Tools 설치

이제 homebrew를 사용할 준비가 끝났다. <http://brew.sh/> 하단에 있는 다음 명령을 사용해 homebrew를 설치한다. (주의: sudo 를 사용하지 않는다)

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/homebrew/go/install)"
```

brew 사용에는 주의할 점이 몇가지가 있다. 첫째는 위 명령으로 brew를 설치한 뒤 ‘brew doctor’ 명령을 실행해 잘못된 점이 없는지 확인해야한다는 것이다. 둘째는 brew 명령은 sudo 없이 사용한다는 점이다. 마지막으로 brew는 소프트웨어를 /usr/local 아래에 있는 디렉토리들에 설치하는데 sudo 없이 명령을 수행해야하므로 혹시 에러가 난다면 해당 디렉토리를 현재 사용자의 소유로 바꾸어 주어야 할 필요가 있을 수 있다.

예를들어 다음 명령은 /usr/local/bin을 현재 사용자의 소유로 변경한다.

```
$ chown $USER -R /usr/local/bin
```

brew 설치가 완료되면 <https://xquartz.macosforge.org/landing/>를 방문해 XQuartz를 다운로드 받아 설치한다. dmg 파일이 업로드 되어있으므로 파일을 다운로드 받은 뒤 링크를 더블클릭하는 것으로 손쉽게 설치가 가능하다.

이제 homebrew/science를 등록한 뒤 R을 설치한다.

```
$ brew update
$ brew tap homebrew/science
$ brew install gfortran
$ brew install r --with-openblas
```

가장 마지막 줄에서 --with-openblas 옵션은 BLAS(Basic Linear Algebra System)의 구현체 중 openblas를 사용하도록 하는 옵션으로 OSX상에서 성능이 우수하므로 --with-openblas를 사용하기를 권장한다.

## 2 R 시작하기

설치가 끝나면 R을 입력하거나 아이콘을 클릭해 R 프로그램을 시작한다.

```
$ R

R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.
```

Natural language support but running in an English locale

```
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.
```

>

여기서 “>” 가 앞으로 계속해서 보게될 프롬프트이다. 잘 작동되는지 확인하기 위해 “hello”를 화면에 출력해본다.

```
> print("hello")  
[1] "hello"
```

만약 여러줄에 걸쳐 명령어를 입력하면 > 대신 + 가 프롬프트로 나타난다.

```
> print(  
+ "hello"  
+ )  
[1] "hello"
```

### 3 도움말 보기

R에서 도움말은 ‘?’ 뒤에 찾고자 하는 명령어를 입력하거나 help(명령어) 형태로 입력하여 볼 수 있다. 예를들어 앞서서 사용한 print에 대한 도움말은 다음과 같이 볼 수 있다.

```
> ?print  
  
print           package:base          R Documentation
```

```
Print Values
```

Description:

```
print prints its argument and returns it _invisibly_ (via
invisible(x)). It is a generic function which means that new
printing methods can be easily added for new ''classes.
```

Usage:

```
print(x, ...)
```

....

Examples:

```
require(stats)

## print is the "Default function" --> print.ts(.) is called
ts(1:20)
for(i in 1:3) print(1:i)

## Printing of factors
attenu$station ## 117 levels -> 'max.levels' depending on width

## ordered factors: levels "11 < 12 < .."
esoph$agegp[1:12]

...
```

도움말에는 위와같이 예제(Example 섹션)가 포함된 경우가 많다. 예제 부분은 example()을 사용해 손쉽게 수행해 볼 수 있다. 예를들어 print에 대한 예제는 example(print)를 입력하면 자동으로 수행된다.

만약 정화한 함수나 기능 등에서 검색을 수행하려면 ‘??’ 뒤에 찾고자 하는 문자열을 입력하면 된다. 또는 help.search(“찾고자 하는 문자열”)을 입력한다.

```
> ??print
```

```
Vignettes with name or keyword or title matching '',
print using fuzzy matching:
```

```
Rcpp::Rcpp-introduction
    Rcpp-introduction
```

```
Type 'vignette("FOO", package="PKG")' to inspect
entries 'PKG::FOO'.
```

```
Demos with name or title matching ''print using fuzzy
matching':
```

```
RGtk2::printing      Demonstrates rendering a document
                     and printing it via a dialog
tcltk::tkcanvas      Creates a canvas widget showing a
                     2-D plot with data points that can
                     be dragged with the mouse.
```

```
vcd::hullternary
    Demo for adding data point hulls to
    a ternary plot
```

```
...
```

도움말의 시작페이지는 `help.start()` 명령을 사용하여 볼 수 있다. 또한 공식 설명서인 [1, 4] 도 좋은 출발점이다. 이 중 Introduction to R은 <http://ihelp.r-forge.r-project.org/doc/manual-ko/R-intro-ko.html>에 한국어로의 번역이 게시되어있다. 중요 문서중 하나이니 챙겨보기 바란다.

## 4 IDE

R을 실행한뒤 TAB 키를 누르면 명령어를 자동완성시켜주고, 키보드 화살표 위 아래를 눌러서 이전/이후 history의 명령어를 실행할 수 있으므로 콘솔에서의 기본적인 지원도 그리 나쁘지

않다.

그래도 한번쯤은 반드시 찾아보는 것이 IDE가 아닐까 한다. R에는 기본적인 GUI 도구(그림 2.7)가 떨려오지만, 그다지 많은 기능이 지원되지는 않는다.

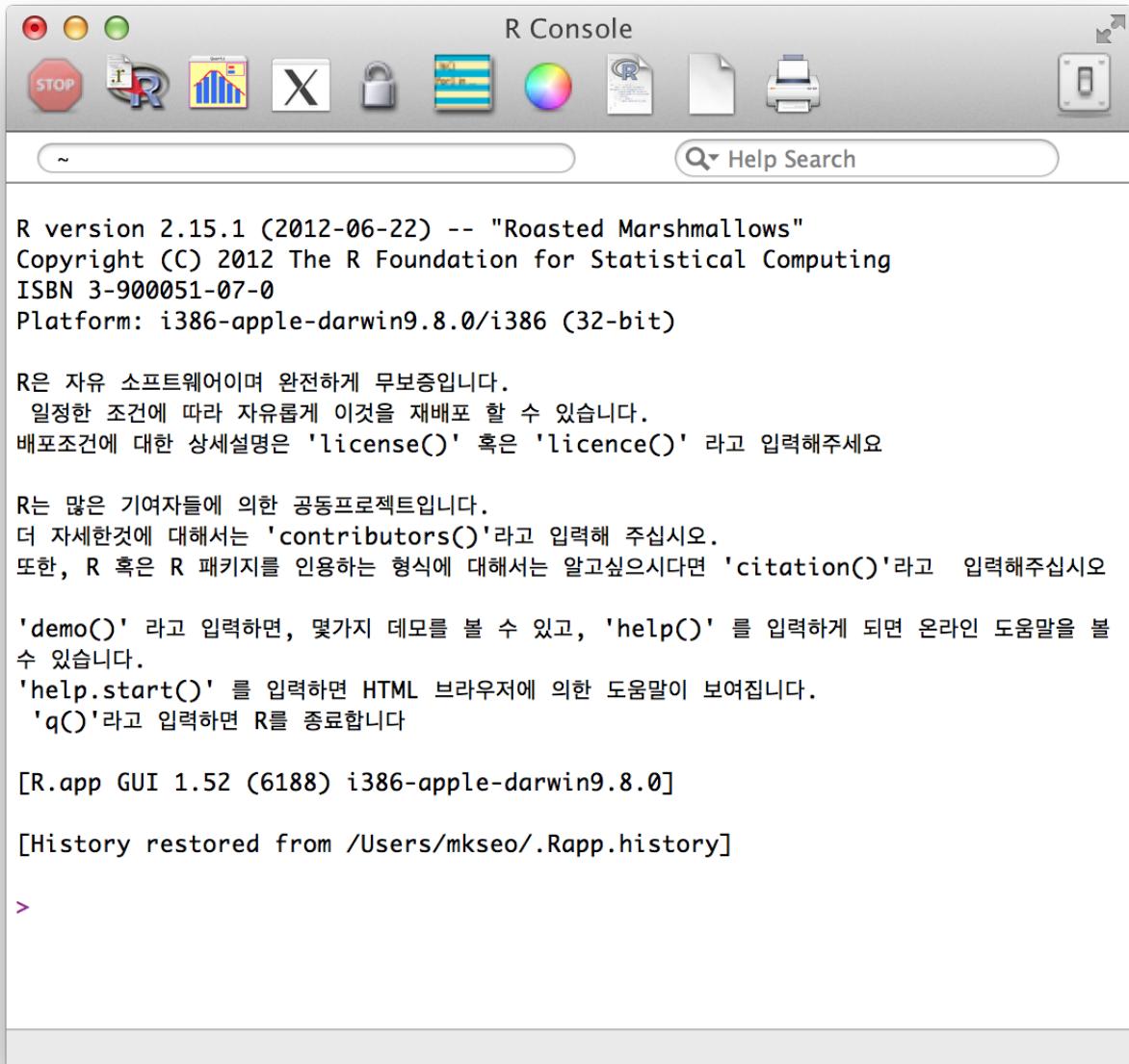


그림 2.7: R GUI 작동화면

만약 GUI와 메뉴 등이 필요하다면 기본 제공 GUI 보다는 RStudio(그림 2.8)가 많이 언급되고 있고 또 사용하기에도 괜찮은 편이다.

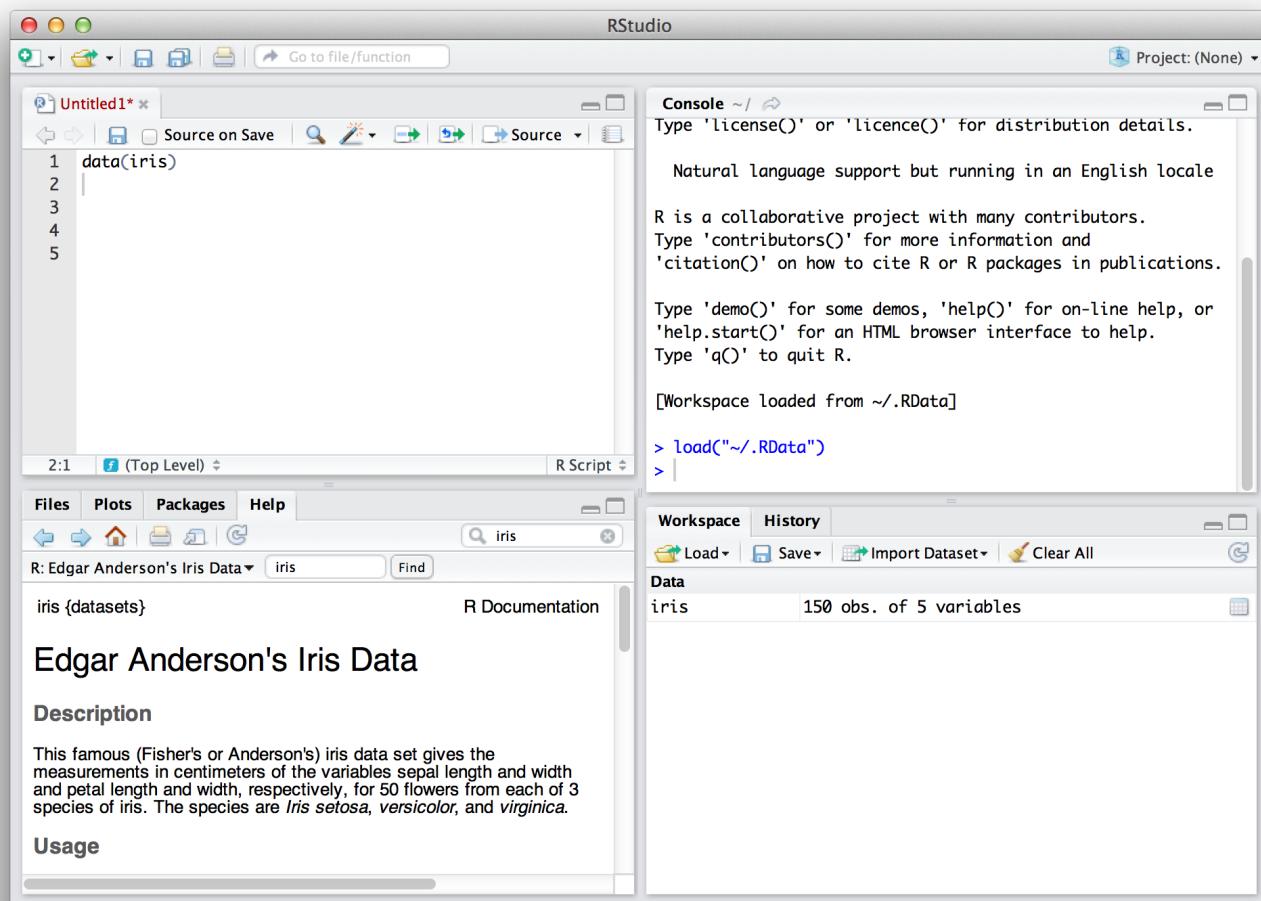
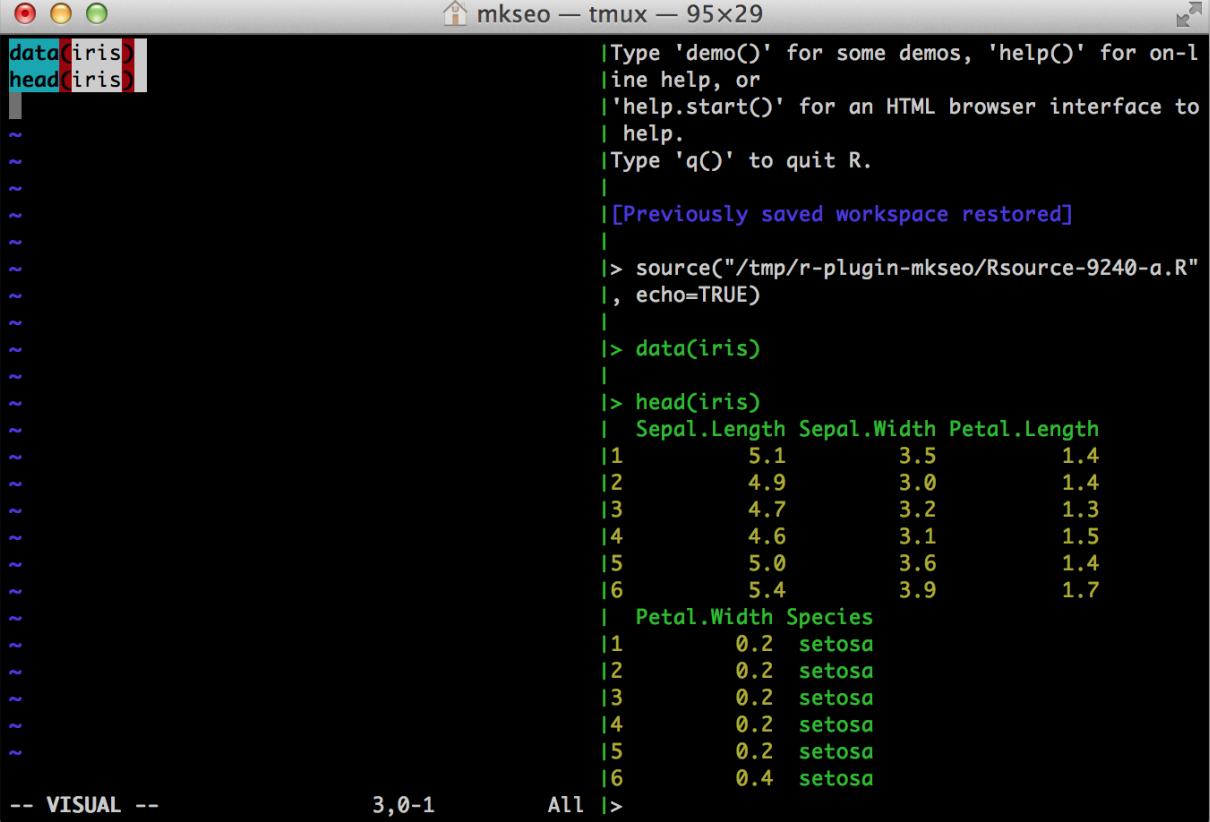


그림 2.8: RStudio

RStudio는 명령 콘솔외에 파일 편집, 데이터 보기, History, 차트 등을 손쉽게 접근할 수 있게 해준다. 만약 R 프로그램을 편집기에서 입력하면서 동시에 방금 입력한 내용을 곧바로 콘솔에 보내는 방법으로 코딩하고 싶다면 메뉴에서 ‘Code’부분을 살펴보면 된다. 코드의 처음부터 현재줄까지, 또는 에디터의 코드 한줄씩을 쉽게 실행할 수 있다.

Vim 사용자라면 Vim-R-Plugin을 사용할 수 있다. Vim-R-Plugin은 screen 또는 tmux와 함께 사용하는 도구로, 화면을 에디터(vim)/R 콘솔/객체 브라우저로 나누어 작업할 수 있게 해준다. 그림 2.9은 vim r plugin을 사용해 특정 코드 블럭을 R콘솔로 보내 실행하는 화면이다.



The screenshot shows a tmux session titled 'mkseo' with a window size of 95x29. The terminal window displays R code being run. The code starts with `data(iris)` and `head(iris)`, followed by several blank lines. Then it runs `source()` to restore a workspace, and finally prints the first six rows of the iris dataset. The output shows Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species columns for the first six entries. The terminal status bar at the bottom indicates 'VISUAL' mode, '3,0-1', and 'All >'.

```

data(iris)
head(iris)

[Previously saved workspace restored]

> source("/tmp/r-plugin-mkseo/Rsource-9240-a.R"
, echo=TRUE)

> data(iris)
>
> 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

-- VISUAL --
3,0-1      All >

```

그림 2.9: Vim R Plugin

Emacs 사용자라면 Emacs-ess를 사용할 수 있다.

이 외에도 [Geany](#)와 같은 IDE 환경을 사용하거나 간단하게는 리눅스 데스크탑 환경 중 하나인 GNOME에 기본적으로 설치되는 [gedit](#)를 사용할 수 도 있다.

이 중 한가지 툴을 추천하자면 RStudio 가 가장 낫다고 생각된다. 무료로 제공되는데다가 사용하기에도 편리하다.

## 5 배치 실행

데이터 분석은 R 명령어를 인터랙티브 환경에서 실행하면되지만, 데이터 분석을 장시간 수행해야한다거나 반복적인 작업을 종종 해야한다면 배치파일을 만들 수 있다. 다음은 Rscript를 사용해 코드를 .R 파일에 저장하고 배치로 실행하는 예이다.

```
$ cat > x.R
```

```
#!/usr/bin/env Rscript
print("hello")
^D

$ chmod u+x x.R
$ ./x.R
[1] "hello"
```

또는 R코드를 .R 확장자의 파일에 저장한뒤 다른 R 스크립트에서 source(“파일명.R”) 형태로 실행할 수도 있다. 이를 활용해 데이터 로딩은 data\_load.R에 구현하고, 데이터 분석은 data\_analysis.R에 저장한뒤 main.R을 다음과 같이 구성할 수 있다.

```
source("data_load.R")
source("data_analysis.R")
```

## 6 패키지의 사용

R의 가장 큰 장점중 하나는 사용자들이 구축해 놓은 방대한 라이브러리이다. 이 라이브러리들은 목록은 CRAN에 접속하여 좌측 메뉴에서 ‘Packages’를 눌러 확인해 볼 수 있다. 특히 CRAN Task Views 페이지로 들어가면 각 패키지들을 Bayesian, ChemPhys, Cluster, Differential Equations, ... 등으로 분류해 놓을 것을 볼 수 있는데 각 분류를 눌러 들어가면 어떤 상황에서 어떤 패키지를 사용할 수 있는지 보다 쉽게 확인할 수 있다.

한 예로 R에서 random forest 알고리즘을 지원하는 randomForest 패키지를 사용하는 경우를 생각해보자. 먼저 다음과 같이 install.packages(“패키지명”) 명령어를 사용해 randomForest를 설치한다. install.packages()명령어를 입력하면 mirror를 선택하는 화면이 나오는데 Korea 또는 원하는 mirror를 선택하면 된다.

```
> install.packages("randomForest")
--- Please select a CRAN mirror for use in this session ---
Loading Tcl/Tk interface ... done
trying URL 'http://cran.nexr.com/bin/macosx/leopard/contrib/2.15/
  randomForest_4.6-7.tgz'
Content type 'application/x-gzip' length 206287 bytes (201 Kb)
opened URL
=====
downloaded 201 Kb
```

```
The downloaded binary packages are in
  /var/folders/3t/kmb3lgcn5bxf6m020rhnpbst0000gp/T//RtmpdmgXpE/
    downloaded_packages
>
```

패키지를 설치한 뒤 시간이 지나 새로운 버전이 나왔다면 `update.packages()`를 사용해 패키지들을 업데이트 할 수 있다. 이 명령을 (인자없이) 입력하면 설치된 패키지들을 확인해 최신 버전을 설치해준다.

설치된 패키지를 사용하려면 `library(패키지명)`, 또는 `require(패키지명)`을 사용해 패키지를 불러들인다.

```
> library(randomForest)
randomForest 4.6-7
Type rfNews() to see new features/changes/bug fixes.
```

이제 `randomForest` 패키지에 있는 다양한 함수를 사용할 준비가 끝났다. 실제 `randomForest` 패키지내에 어떤 함수가 있고 이들을 어떻게 사용되는지는 cran 홈페이지의 패키지 설명 페이지를 참고한다. 예를들어 `randomForest`는 <http://cran.r-project.org/web/packages/randomForest/index.html>에 설명 페이지가 있고 여기서 Reference Manual 링크를 눌러 함수의 목록과 도움말을 볼 수 있다.

경우에 따라서는 vignettes(R에서 메뉴얼 격에 해당하는 문서를 부르는 용어)를 제공하는 경우도 있다. `randomForest`에는 이런 문서가 없지만, 또다른 머신 러닝 관련 패키지인 `caret`에서는 vignette를 제공한다. Vignette는 <http://cran.r-project.org/web/packages/caret/index.html>의 경우처럼 패키지 설명 페이지에서 Vignette 밑에 나열된 문서를 확인하면 된다.

[The R Journal](#) 또는 [Journal of Statistical Software](#)에는 R패키지, 또는 그 패키지를 사용한 데이터 분석에 대한 좋은 글이 많이 실려있다. 이들 사이트를 확인해보는 것도 새로운 패키지를 배우는데 많은 도움이 된다.

## 데이터 타입

R에서는 일반적인 프로그래밍 언어에서 흔히 사용되는 정수, 부동소수, 문자열이 기본적으로 지원된다. 그외에도 자료처리에 적합한 자료구조인 벡터(vector), 행렬(matrix), 데이터 프레임(data frame), 리스트(list) 등이 있다.

### 1 변수

먼저 간단히 R에서 사용되는 변수에 대해서 알아보자. R의 변수명은 알파벳, 숫자, ‘\_’, ‘.’로 구성되며 첫글자는 문자 또는 ‘\_’으로 시작해야한다. 만약 ‘.’으로 시작한다면 ‘.’ 뒤에는 숫자가 올 수 없다. 예를들어 다음은 모두 올바른 변수명이다.

```
a  
b  
a1  
a2  
.x
```

그러나 다음은 올바르지 않은 변수명이다.

```
2a  
.2
```

한가지 흥미로운 사실은 R 1.9.0 이전에는 ‘\_’ 가 변수명에 사용될 수 없었다는 점이다. 이런 이유로 다른 언어에서 흔히 ‘\_’ 을 사용할만한 상황에서 R은 ‘.’ 을 사용하고는 한다. 예를들어 training\_data, validation\_data 같은 변수명 대신 data.training, data.validation 과 같이 마치 객체의 속성을 접근하는 듯한 명명 규칙을 사용하고는 한다.

변수에 값을 할당할때는 <- 또는 <<- 또는 = 를 사용한다. 어떤 연산자를 사용해야하는지는 다분히 논쟁적인 주제이다. 이 책에서는 <- 를 사용하였는데, 그 이유중 하나는 = 는 경우에

따라 사용될 수 없는 경우가 있기 때문이다<sup>1)</sup>.

## 2 스칼라

R의 기본형은 벡터[5, 6]이므로 이를 스칼라자료는 길이 1의 벡터로 볼 수 있다. 그러나 실제로는 편하게 스칼라 자료라고 생각해도 큰 문제는 없다.

### 2.1 숫자

정수, 부동 소수 등이 자연스럽게 지원된다.

```
> a <- 3
> b <- 4.5
> c <- a + b
> print(c)
[1] 7.5
```

위에서 ‘<-’ 는 값을 변수에 할당함을 의미한다. 따라서 위 코드는 a 변수에 3을 저장하고, b 변수에 4.5를 저장한다. c에는 a와 b의 합을 저장하고, 마지막으로 c변수값을 출력한다. 출력을 위해서는 print() 함수를 사용했는데, 함수 호출 방법은 이처럼 다른 언어와 큰 차이가 없다.

### 2.2 NA

만약 데이터에 값이 존재하지 않는다면 NA로 표시할 수 있다. 예를들어 4명의 시험점수가 있을 때 그 점수가 각각 80, 90, 75 이지만 4번째 사람의 점수를 모른다면 NA로 표현한다.

```
> one <- 100
> two <- 75
> three <- 80
> four <- NA
> is.na(four)
[1] TRUE
```

위에 보인바와 같이 어떤 변수에 NA가 저장되어있는지는 is.na() 함수로 확인할 수 있다.

---

<sup>1)</sup><http://stat.ethz.ch/R-manual/R-patched/library/base/html/assignOps.html>

## 2.3 NULL

NULL은 NULL 객체를 뜻하는데, 변수가 초기화 되지 않은 경우 등에 사용할 수 있다. 이는 결측치(NA)와 구분하여 생각해야한다. 어떤 변수에 NULL이 할당되었는지는 `is.null()`을 사용하여 판단할 수 있다.

```
> x <- NULL
> is.null(x)
[1] TRUE
> is.null(1)
[1] FALSE
> is.null(NA)
[1] FALSE
> is.na(NULL)
logical(0)
Warning message:
In is.na(NULL) : is.na() applied to non-(list or vector) of type 'NULL'
'
```

## 2.4 문자열

R은 C 등의 언어에서 볼 수 있는 한개 문자에 대한 데이터 타입은 없다. 대신 문자열로 모든 것을 처리한다. 문자열은 ‘this\_is\_string’ 또는 “this\_is\_string”과 같이 어느 따옴표로 묶어도 된다.

```
> a <- "hello"
> print(a)
[1] "hello"
```

## 2.5 진리값

`TRUE`, T는 모두 참값을 말한다. `FALSE`, F는 거짓을 말한다. 진리값에는 `&` (AND), `|` (OR), `!` (NOT) 연산자를 사용할 수 있다.

```
> TRUE & TRUE
[1] TRUE
> TRUE & FALSE
[1] FALSE
```

```
> TRUE | TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> !TRUE
[1] FALSE
> !FALSE
[1] TRUE
```

그러나 좀 더 염밀히 말하면 TRUE와 FALSE는 예약어(reserved words)이며 T, F는 각각 TRUE와 FALSE로 초기화된 전역 변수이다. 따라서 다음과 같이 T에 FALSE를 할당하는 것이 가능하다! 반면 TRUE에는 FALSE를 할당할 수 없다.

```
> T <- FALSE
> TRUE <- FALSE
Error in TRUE <- FALSE : invalid (do_set) left-hand side to assignment
```

AND나 OR연산자에는 &, | 외에도 &&와 || 가 있다. 이들의 차이점은 &, |는 boolean으로 저장된 벡터(Vector) (페이지 41)끼리의 연산시 각 원소간 계산을 한다는 점이다. 예를들어 다음 코드를 보자.

```
> c(TRUE, TRUE) & c(TRUE, FALSE)
[1] TRUE FALSE
```

반면 && 은 벡터의 요소간 계산을하기 위함이 아니라 TRUE && TRUE 등의 경우와 같이 한개의 boolean 값끼리의 연산을 하기 위한 연산자이다. 이는 ||와 |도 마찬가지이다. 예를들어 다음 코드를 보면 한개의 값만 반환됨을 볼 수 있다.

```
> c(TRUE, FALSE) && c(TRUE, FALSE)
[1] TRUE
```

또 &&, || 는 short-circuit을 지원한다. 따라서 A && B 형태의 코드가 있을때 A가 만약 TRUE라면 B도 평가하지만 A가 FALSE라면 B를 평가하지 않는다.

언뜻보기에는 && 나 ||를 더 많이 사용해야할 것 같지만, R에서는 벡터 또는 리스트내 원소를 한번에 비교하는 연산이 많으므로 오히려 &나 |가 더 유용하다. 이에 대해서는 뒤에 벡터 연산 (페이지 71)에서 다시 다룬다.

## 2.6 요인(Factor)

요인(Factor)은 범주형(Categorical) 변수를 위한 데이터 탐색이다. 예를 들어, 성별은 다음과 같이 지정할 수 있다.

```
> sex <- factor("m", c("m", "f"))
> sex
[1] m
Levels: m f
```

위 코드에서 ‘sex’ 변수를 출력해보니 값은 m 이었고, 이 변수가 가질 수 있는 수준은 m과 f의 2개 였다.

첫줄의 `factor()` 는 범주형 변수 값을 표현하기 위해 호출되었다. `factor()` 에 주어진 첫번째 인자는 ‘sex’ 변수에 저장되는 값으로서 여기서는 m이다. 그리고 이 범주형 변수는 `c("m", "f")`에 표현된 바와 같이 2개의 값 m과 f가 가능하다. 다시 말해 `sex`에는 m, f의 2개 값을 갖는 범주에서 m이 선택되어 저장되었다. `c()`의 표현이 생소한데 이것은 벡터를 표현하는 방식으로 아래에서 설명한다.

Factor 변수는 `nlevels()`로 범주의 수를 구할 수 있고, `levels()`로 범주 목록을 알 수 있다.

```
> nlevels(sex)
[1] 2

> levels(sex)
[1] "m" "f"
```

이를 응용하면 각 범주의 값을 다음과 같이 구할 수 있다. R에서 색인은 0이 아니라 1부터 시작함에 유의하기 바란다.

```
> levels(sex)[1]
[1] "m"
> levels(sex)[2]
[1] "f"
```

Factor 변수에서 level의 값을 직접적으로 수정하고자 한다면 `levels()`에 값을 할당하면 된다. ‘m’을 ‘male’, ‘f’를 female로 바꿔보자.

```
> sex
[1] m
Levels: m f
```

```
> levels(sex) <- c('male', 'female')
> sex
[1] male
Levels: male female
```

factor()는 기본적으로 데이터에 순서가 없는 명목형 변수(Nominal)를 만든다. 만약 범주형 데이터지만 ‘나쁨, 조금 나쁨, 보통, 조금 좋음, 아주 좋음’ 등과 같이 순서가 있는 값일 경우는 순서형(Ordinal) 변수로 만들기 위해 ordered()를 사용하거나 factor() 호출시 ordered=TRUE<sup>2)</sup>를 지정해준다.

```
> ordered(c("a", "b", "c"))
[1] a b c
Levels: a < b < c
> factor(c("a", "b", "c"), ordered=TRUE)
[1] a b c
Levels: a < b < c
```

앞서와 달리 Levels에 순서가 <로 표시되어 있음을 볼 수 있다.

### 3 벡터(Vector)

#### 3.1 벡터의 정의

벡터는 다른 프로그래밍 언어에서 흔히 보는 배열의 개념으로, 다음과 같이 c() 안에 원하는 인자들을 나열하여 정의한다.

```
> x <- c(1, 2, 3, 4, 5)
> x
[1] 1 2 3 4 5
```

나열하는 인자들은 한가지 유형의 스칼라 타입이어야 한다. 만약 다른 타입의 데이터를 섞어서 벡터에 저장하면 이들 데이터는 한가지 타입으로 자동 형변환 된다. 예를 들어 아래 코드에서 숫자형 자료인 2는 “2”의 문자열 형태로 자동으로 변환되어 x 안에는 문자열 형태의 자료만 나열되었다.

```
> x <- c("1", 2, "3")
> x
```

<sup>2)</sup>T는 TRUE와 같으므로 ordered=T라고 해도 된다.

```
[1] "1" "2" "3"
```

벡터는 중첩될 수 없다. 따라서 벡터 안에 벡터를 정의하면 이는 단일 차원의 벡터로 변경된다. 중첩된 구조가 필요하다면 뒤에서 다룰 [리스트\(List\)](#) (페이지 46)를 사용해야한다.

```
> c(1, 2, 3)
[1] 1 2 3
> c(1, 2, 3, c(1, 2, 3))
[1] 1 2 3 1 2 3
```

숫자형 데이터의 경우 start:end 형태로 시작값부터 끝값까지의 값을 갖는 벡터를 만들 수 있다. 또는 seq(from, to, by) 형태 역시 가능하다.

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> x <- 5:10
> x
[1] 5 6 7 8 9 10
> seq(1, 10, 2)
[1] 1 3 5 7 9
```

seq\_along()은 인자로 주어진 데이터의 길이만큼 1, 2, 3, ..., N으로 구성된 벡터를 반환한다. seq\_len()은 N값이 인자로 주어지면 1, 2, ..., N으로 구성된 벡터를 반환한다.

```
> seq_along(c('a', 'b', 'c'))
[1] 1 2 3
> seq_len(3)
[1] 1 2 3
```

벡터의 각 셀에는 이름을 부여할 수 있다. names()에 원하는 이름을 벡터로 넘겨주면 된다.

```
> x <- c(1, 3, 4)
> names(x) <- c("kim", "seo", "park")
> x
kim    seo  park
1      3      4
```

### 3.2 벡터내 레이터 접근

벡터는 [ ] 안에 인덱스를 적어서 각 요소를 가져올 수 있다. 이때, 인덱스는 다른 언어와 달리 1로 시작한다.

```
> x <- c("a", "b", "c")
> x[1]
[1] "a"
> x[3]
[1] "c"
```

또 ‘-인덱스’와 같이 음수의 인덱스를 사용해 특정 요소만 제외할 수도 있다.

```
> x <- c("a", "b", "c")
> x[-1]
[1] "b" "c"
> x[-2]
[1] "a" "c"
```

위의 코드에서 x[-1]은 제일 첫번째 요소인 “a”를 제외한 경우이고, x[-2]는 두번째 요소인 “b”를 제외한 경우이다.

벡터의 여러 위치에 저장된 값을 가져오려면 ‘벡터명[색인 벡터]’의 형식을 사용한다.

```
> x <- c("a", "b", "c")
> x[c(1, 2)]
[1] "a" "b"
> x[c(1, 3)]
[1] "a" "c"
```

‘x[start:end]’를 사용해 start부터 end까지 (end에 위치한 요소 포함)의 데이터를 볼 수도 있다.

```
> x <- c("a", "b", "c")
> x[1:2]
[1] "a" "b"
> x[1:3]
[1] "a" "b" "c"
```

벡터의 각 셀에 names()를 사용해 이름을 부여해 두었다면, 이 이름을 사용해 데이터를 접근할 수 있다.

```
> x <- c(1, 3, 4)
> names(x) <- c("kim", "seo", "park")
> x
  kim   seo  park
  1     3     4
> x["seo"]
seo
3
> x[c("seo", "park")]
  seo  park
  3     4
```

벡터에 부여된 이름만 보려면 이름을 부여할 때와 마찬가지로 `names()`를 사용한다. 다음은 벡터의 두번째 요소에 부여된 이름이 “seo”임을 보여준다.

```
> names(x)[2]
[1] "seo"
```

벡터의 길이는 `length()` 또는 `NROW()` (대문자임에 주의!) 를 통해 알 수 있다. 본래 `nrow()`는 행렬(Matrix) (페이지 48)의 행의 수를 알려주는 함수이지만 `nrow()`의 변형인 `NROW()`는 인자가 벡터인경우 벡터를 n행 1열의 행렬로 취급해 길이를 반환한다.

```
> x <- c("a", "b", "c")
> length(x)
[1] 3
> nrow(x) # nrow()는 행렬만 가능
NULL
> NROW(x) # NROW()는 벡터와 행렬 모두 사용 가능
[1] 3
```

### 3.3 벡터 연산

`%in%` 연산자는 어떤 값이 벡터에 포함되어있는지를 알려준다.

```
> "a" %in% c("a", "b", "c")
[1] TRUE
> "d" %in% c("a", "b", "c")
[1] FALSE
```

벡터를 집합(set)으로 취급해 집합간 합집합, 교집합, 차집합을 계산할 수 있다.

```
> setdiff(c("a", "b", "c"), c("a", "d"))
[1] "b" "c"
> union(c("a", "b", "c"), c("a", "d"))
[1] "a" "b" "c" "d"
> intersect(c("a", "b", "c"), c("a", "d"))
[1] "a"
```

집합간 비교는 setequal()을 사용한다.

```
> setequal(c("a", "b", "c"), c("a", "d"))
[1] FALSE
> setequal(c("a", "b", "c"), c("a", "b", "c", "c"))
[1] TRUE
```

### 3.4 seq

1부터 31까지의 숫자를 저장한 벡터를 표현하려면 c(1, 2, 3, ..., 31)과 같이 숫자를 모두 나열해야한다. 이러한 번거로운 일을 피하기 위해 seq()함수를 쓸 수 있다. seq(시작값, 마지막값, 증가치)의 형태로 호출하면 된다. 이때 증가치는 생략가능하다.

```
> seq(1, 5)
[1] 1 2 3 4 5
> seq(1, 5, 2)
[1] 1 3 5
```

보다 짧게 표현하려면 시작값:마지막값 형태로 적어도 된다.

```
> 1:5
[1] 1 2 3 4 5
```

### 3.5 rep

특정 값들이 반복된 벡터를 손쉽게 만드려면 rep()를 사용한다. 다음은 1, 2를 총 5회 반복한 벡터를 만드는 예이다.

```
> rep(1:2, 5)
[1] 1 2 1 2 1 2 1 2 1 2
```

이와 달리 1이 5회, 2가 5회 반복된 벡터를 만들기 위해서는 다음과 같이 하면된다.

```
> rep(1:2, each=5)
[1] 1 1 1 1 1 2 2 2 2 2
```

## 4 리스트(List)

리스트는 다른언어에서 흔히 보는 해싱 또는 딕셔너리에 해당하며, (키, 값) 형태의 데이터를 담는 연관 배열(associative array)이다.

### 4.1 리스트의 정의

리스트는 list(키=값, 키=값, ...) 형태로 데이터를 나열해 정의한다.

```
> x <- list(name="foo", height=70)
> x
$name
[1] "foo"

$height
[1] 70
```

이 때 각 값이 반드시 스칼라일 필요는 없다. 다음처럼 벡터를 저장할 수도 있다.

```
> x <- list(name="foo", height=c(1, 3, 5))
> x
$name
[1] "foo"

$height
[1] 1 3 5
```

이처럼 리스트에는 다양한 값을 혼합해서 저장할 수 있다. 따라서 리스트 안에 리스트를 중첩하는 것이 가능하다.

```
> list(a=list(val=c(1, 2, 3)), b=list(val=c(1, 2, 3, 4)))
```

```
$a
$a$val
[1] 1 2 3

$b
$b$val
[1] 1 2 3 4
```

## 4.2 리스트내 데이터 접근

앞에서 본바와 같이 리스트를 출력해보면 ‘\$키’ 형태로 각 키들이 나열된다. 데이터는 ‘리스트 변수명\$키’와 같이 접근한다. 또는 각 요소를 순서대로 ‘리스트변수[[인덱스]]’와 같이 접근할 수도 있다.

```
> x <- list(name="foo", height=c(1, 3, 5))
> x$name
[1] "foo"
> x$height
[1] 1 3 5
> x[[1]]
[1] "foo"
> x[[2]]
[1] 1 3 5
```

주의할점은 값을 가져오기 위해서는 ‘[[인덱스]]’의 형태로 적어야지 ‘[인덱스]’로 해서는 안된다는 점이다. ‘[인덱스]’의 형태는 각 값이 아니라 ‘(키, 값)’을 담고 있는 서브 리스트를 반환한다. 예를들어 다음의 코드를 보자.

```
> x[1]
$name
[1] "foo"

> x[2]
$height
[1] 1 3 5
```

보다시피 x[1]는 ‘(name, foo)’ 를 갖고 있는 리스트를 반환한다.

## 5 행렬(Matrix)

벡터와 마찬가지로 행렬에는 한가지 유형의 스칼라만 저장할 수 있다. 따라서 모든 요소가 숫자인 행렬은 가능하지만, ‘1열은 숫자, 2열은 문자열’과 같은 형태는 불가능하다.

### 5.1 행렬의 정의

행렬은 `matrix( )`를 사용해 표현한다. 다음은 1, 2, 3, 4, 5, 6, 7, 8, 9로 구성된 3행 3열의 행렬을 만드는 방법이다.

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
[,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol=3)
[,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

위의 예에서 보다시피 행렬값을 나열한뒤 `ncol`을 사용해 열의수를 지정하거나 `nrow`를 사용해 행의 수를 지정하면 된다. 위 경우에는 행렬값이 좌측 열부터 채워져오는데, 대신 행렬값을 위쪽 행부터 채우고 싶다면 `byrow`를 사용한다.

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3, byrow=T)
[,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

위 코드에서 `byrow=T`의 T는 TRUE를 뜻한다. 따라서 `byrow=TRUE`로 지정해도 효과는 같다.

행렬의 행과 열에 명칭을 부여하고 싶다면 `dimnames()`를 사용한다.

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3,
+         dimnames=list(c("item1", "item2", "item3"),
+                         c("feature1", "feature2", "feature3")))
          feature1 feature2 feature3
```

item1	1	4	7
item2	2	5	8
item3	3	6	9

## 5.2 행렬내 데이터 접근

행렬의 각 요소는 ‘행렬이름[행인덱스, 열인덱스]’로 접근할 수 있다. 이 때, 인덱스는 벡터의 경우와 마찬가지로 1부터 시작한다.

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

역시 벡터와 마찬가지로 ‘-인덱스’와 같이 표현해 특정 행/열을 제외하거나 ‘시작인덱스:끝인덱스’로 특정 범위의 데이터만 가져올 수도 있다. 만약 특정 행이나 열의 전부를 가져오고 싶다면 행이나 열쪽에는 아무런 인덱스도 기재하지 않으면 된다.

다음 예는 1, 2행의 데이터만 추출한다.

```
> x[1:2, ]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

반대로 3행을 제외하는 방법을 쓸 수도 있다.

```
> x[-3, ]
     [,1] [,2] [,3]
```

```
[1,] 1 4 7  
[2,] 2 5 8
```

또는 인덱스를 벡터에 넣어 나열할 수도 있다. 다음은 1행, 3행, 1열, 3열의 값만 추출하는 경우이다.

```
> x[c(1, 3), c(1, 3)]  
[,1] [,2]  
[1,] 1 7  
[2,] 3 9
```

만약 dimnames를 통해 행과 열에 이름을 부여했다면 그 이름을 직접 사용할 수도 있다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9),  
+                 nrow=3,  
+                 dimnames=list(c("item1", "item2", "item3"),  
+                               c("feature1", "feature2", "feature3")))  
> x  
      feature1 feature2 feature3  
item1      1        4        7  
item2      2        5        8  
item3      3        6        9  
> x["item1", ]  
feature1 feature2 feature3  
1        4        7
```

### 5.3 행렬의 연산

행렬에 스칼라를 곱하거나 나누는 연산은 일반적인 프로그래밍 언어에서 하듯이 \* 나 / 를 사용한다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)  
> x * 2  
[,1] [,2] [,3]  
[1,] 2 8 14  
[2,] 4 10 16  
[3,] 6 12 18  
> x / 2
```

```
[ ,1] [ ,2] [ ,3]
[1,] 0.5 2.0 3.5
[2,] 1.0 2.5 4.0
[3,] 1.5 3.0 4.5
```

행렬끼리의 덧셈이나 뺄셈은 + 나 - 를 사용한다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
> x + x
[ ,1] [ ,2] [ ,3]
[1,] 2 8 14
[2,] 4 10 16
[3,] 6 12 18
> x - x
[ ,1] [ ,2] [ ,3]
[1,] 0 0 0
[2,] 0 0 0
[3,] 0 0 0
```

행렬 곱은 %\*% 를 사용한다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
> x
[ ,1] [ ,2] [ ,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
> x %*% x
[ ,1] [ ,2] [ ,3]
[1,] 30 66 102
[2,] 36 81 126
[3,] 42 96 150
```

역행렬은 solve(행렬)로 계산한다.

```
> x <- matrix(c(1, 2, 3, 4), ncol=2)
> x
[ ,1] [ ,2]
[1,] 1 3
```

```
[2,]      2      4  
> solve(x)  
[ ,1] [ ,2]  
[1,]    -2   1.5  
[2,]     1  -0.5  
> x %*% solve(x)  
[ ,1] [ ,2]  
[1,]     1     0  
[2,]     0     1
```

전치행렬은 t()로 구한다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)  
> x  
[ ,1] [ ,2] [ ,3]  
[1,]     1     4     7  
[2,]     2     5     8  
[3,]     3     6     9  
> t(x)  
[ ,1] [ ,2] [ ,3]  
[1,]     1     2     3  
[2,]     4     5     6  
[3,]     7     8     9
```

행렬의 차원은 ncol() 또는 nrow()로 알 수 있다.

```
> x <- matrix(c(1, 2, 3, 4, 5, 6), ncol=3)  
> x  
[ ,1] [ ,2] [ ,3]  
[1,]     1     3     5  
[2,]     2     4     6  
> ncol(x)  
[1] 3  
> nrow(x)  
[1] 2
```

## 6 배열

### 6.1 배열 정의

matrix가 2차원 행렬이라면 배열(array)은 n차원 행렬이다. 예를 들어  $3 \times 4$  matrix는 다음과 같이 array를 사용해 표현할 수 있다.

```
> matrix(1:12, ncol=4)
 [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> array(1:12, dim=c(3, 4))
 [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

이번에는 이 데이터로  $2 \times 2 \times 3$  차원의 배열을 만들어보자.

```
> array(1:12, dim=c(2, 2, 3))
, , 1
 [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2
 [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3
 [,1] [,2]
[1,]    9   11
[2,]   10   12
```

## 6.2 배열 데이터 접근

행렬의 경우와 유사한 방식으로 각 요소에 접근할 수 있다. 또, 배열의 차원은 `dim()` 함수를 통해 알 수 있다.

```
> x <- array(1:12, dim=c(2, 2, 3))
> x
, , 1
 [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2
 [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3
 [,1] [,2]
[1,]    9   11
[2,]   10   12
> x[1, 1, 1]
[1] 1
> x[1, 2, 3]
[1] 11
> x[, , 3]
[,1] [,2]
[1,]    9   11
[2,]   10   12
> dim(x)
[1] 2 2 3
```

## 7 데이터 프레임(Data Frame)

데이터 프레임은 R에서 가장 중요한 자료형이다. 데이터 프레임은 행렬과 마찬가지의 모습을 하고 있지만 행렬과 달리 다양한 변수, 관측치(observations), 범주 등을 표현하기 위해 특화되어 있다.

## 7.1 데이터 프레임 정의

데이터 프레임은 `data.frame()` 을 사용해 정의한다.

```
> d <- data.frame(x=c(1, 2, 3, 4, 5), y=c(2, 4, 6, 8, 10))
> d
  x   y
1 1   2
2 2   4
3 3   6
4 4   8
5 5  10
```

각 열의 데이터 타입이 같다면 여러가지 데이터 타입을 혼용해서 사용할 수 있다. 다음은 위와 같은 데이터에서 Factor 형태의 z 열이 추가된 예이다.

```
> d <- data.frame(x=c(1, 2, 3, 4, 5),
+                   y=c(2, 4, 6, 8, 10),
+                   z=c('M', 'F', 'M', 'F', 'M'))
> d
  x   y   z
1 1   2   M
2 2   4   F
3 3   6   M
4 4   8   F
5 5  10   M
```

만약 이미 정의된 데이터 프레임에 새 열을 추가하고자한다면 ‘\$열이름 <- ...’ 과 같은 형식으로 새로운 데이터를 추가할 수 있다.

```
> d <- data.frame(x=c(1, 2, 3, 4, 5),
+                   y=c(2, 4, 6, 8, 10),
+                   z=c('M', 'F', 'M', 'F', 'M'))
> d$v <- c(3, 6, 9, 12, 15)
> d
  x   y   z   v
1 1   2   M   3
2 2   4   F   6
3 3   6   M   9
```

4 4 8 F 12
5 5 10 M 15

## 7.2 데이터 프레임 접근

데이터 프레임의 각 열은 \$변수명으로 접근할 수 있으며, 행이나 열의 인덱스를 사용해서도 데이터에 접근할 수 있다.

```
> d <- data.frame(x=c(1, 2, 3, 4, 5), y=c(2, 4, 6, 8, 10))
> d$x
[1] 1 2 3 4 5
> d[1,]
  x  y
1 1  2
> d[1,2]
[1] 2
```

벡터로 인덱스를 지정하거나, 또는 제외할 행 또는 열을 - 로 표시할 수 있다.

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

> d[-1, -c(2, 3)]
[1] 2 3
```

또는 컬럼명을 지정할 수도 있다.

```
> d[, c("x", "y")]
  x  y
1 1  2
2 2  4
3 3  6
4 4  8
5 5 10

> d[, c("x")]
[1] 1 2 3 4 5
```

위 코드에서 x 컬럼만 선택했을 때는 data.frame의 일반적인 모양인 표 형태가 아닌 벡터처럼 결과가 나온 것을 볼 수 있다. 이는 컬럼의 차원이 1이 되면 자동으로 데이터 프레임이 아닌 벡터로 값을 반환하기 때문이다. 이를 피하려면 다음과 같이 drop=FALSE 옵션을 지정한다.

```
> d[, c("x"), drop=FALSE]
  x
1 1
2 2
3 3
4 4
5 5
```

데이터 프레임부터는 구조가 복잡해지므로 str() 과 head() 함수를 알아둘 필요가 있다. str() 함수는 R 객체의 내부 구조를 보는데 사용되는 함수로 데이터 프레임을 인자로 받으면 데이터 타입 등을 잘 정리해서 한눈에 보기 쉽게 나타낸다. 아래 예에서는 x, y 열은 숫자형 데이터 타입이며 z는 Factor 데이터임을 알 수 있다.

```
> str(d)
'data.frame': 5 obs. of 3 variables:
 $ x: num 1 2 3 4 5
 $ y: num 2 4 6 8 10
 $ z: Factor w/ 2 levels "F","M": 2 1 2 1 2
```

통상 R에서의 처리는 데이터 프레임을 기본 타입으로 하여 진행되므로 데이터 프레임에 많은 양의 데이터를 저장하게 된다. 이때 만약 단순히 데이터를 저장한 변수명을 입력하면 전체 데이터가 모두 출력되므로 데이터의 일부를 읽어보기가 어렵다. 이 경우 쉽게 데이터의 제일 앞부분만 살펴보는데 head()를 사용할 수 있다.

```
> d <- data.frame(x=1:1000)
> d
  x
1 1
2 2
3 3
4 4
5 5
6 6
... 중략 ...
```

```
995    995
996    996
997    997
998    998
999    999
1000   1000
> head(d)
  x
1 1
2 2
3 3
4 4
5 5
6 6
```

데이터 프레임의 행 이름, 열 이름은 각각 rownames(), colnames() 함수로 지정할 수 있다.

```
> x <- data.frame(1:3)
> x
  X1.3
1    1
2    2
3    3
> colnames(x) <- c('val')
> x
  val
1 1
2 2
3 3
> rownames(x) <- c('a', 'b', 'c')
> x
  val
a 1
b 2
c 3
```

또는 names()를 사용해도 colnames()와 같은 결과를 얻는다.

주어진 값이 벡터에 존재하는지를 판별하는 `%in%` 연산자를 이용하면 특정 열만 선택하는 작업을 보다 손쉽게 할 수 있다. 예를 들어 다음과 같이 a, b, c 열이 있는 데이터 프레임에서 b, c 열만 선택하는 경우를 보자.

```
> (d <- data.frame(a=1:3, b=4:6, c=7:9))
   a b c
1 1 4 7
2 2 5 8
3 3 6 9
> d[, names(d) %in% c("b", "c")]
   b c
1 4 7
2 5 8
3 6 9
```

위 코드에서 첫 줄의 (`d <- data.frame...`)에는 명령의 처음과 끝에 괄호가 사용되었다. 이와 같은 방식으로 명령을 기술하면 해당 명령을 수행함과 동시에 결과값(여기서는 `d`)를 출력하는 작업까지 동시에 하게 된다.

반대로 ! 연산자를 사용해 특정 값들만 제외해서 열을 선택할 수도 있다.

```
> d[, !names(d) %in% c("a")]
   b c
1 4 7
2 5 8
3 6 9
```

## 8 타입 판별

데이터를 처리하기 위해 여러 함수를 호출하다보면 반환된 결과의 타입이 무엇인지 분명하지 않을 때가 있다. 이때는 다음과 같이 `class()`를 사용할 수 있다.

```
> class(c(1, 2))
[1] "numeric"
> class(matrix(c(1, 2)))
[1] "matrix"
> class(list(c(1,2)))
[1] "list"
```

```
> class(data.frame(x=c(1,2)))
[1] "data.frame"
```

위 코드에서 벡터는 numeric(숫자)이라고 class가 반환되었는데, 벡터에 저장된 데이터 타입에 따라 이 값은 logical, character, factor 등이 될 수 있다.

또는 다음과 같이 데이터 타입을 str()문으로도 확인해 볼수 있다. 벡터와 행렬의 결과가 유사해보이지만 벡터의 경우 차원이 [1:2](1차원에 값이 2개)라고 표시되어있는 반면, 행렬은 차원이 [1:2, 1](2차원이고 2행 1열)로 표시되어 있는 점이 다르다.

```
> str(c(1, 2))
num [1:2] 1 2
> str(matrix(c(1,2)))
num [1:2, 1] 1 2
> str(list(c(1,2)))
List of 1
$ : num [1:2] 1 2
> str(data.frame(x=c(1,2)))
'data.frame': 2 obs. of 1 variable:
$ x: num 1 2
```

is.factor, is.numeric (숫자 벡터), is.character(문자열 벡터), is.matrix, is.data.frame 등의 ‘is.\*’ 형태의 함수들을 사용해 데이터의 타입을 확인 할 수 있다. 다음 코드에 몇가지 예를 보였다.

```
> is.numeric(c(1, 2, 3))
[1] TRUE
> is.numeric(c('a', 'b', 'c'))
[1] FALSE
> is.matrix(matrix(c(1, 2)))
[1] TRUE
```

## 9 타입 변환

타입간의 변환은 각 타입에 인자로 변환할 데이터를 넘기거나, ‘as.\*’ 종류의 함수를 사용하여 수행할 수 있다. 다음은 행렬을 data.frame()의 인자로 넘겨 데이터 프레임으로 형변환 하는 예이다.

```
> x <- data.frame(matrix(c(1, 2, 3, 4), ncol=2))
> x
  X1  X2
1  1  3
2  2  4
> colnames(x) <- c("X", "Y")
> x
  X  Y
1 1 3
2 2 4
```

다음은 리스트를 데이터 프레임으로 변환한 예이다.

```
> data.frame(list(x=c(1, 2), y=c(3, 4)))
  x  y
1 1 3
2 2 4
```

또는 as.numeric, as.factor, as.data.frame, as.matrix 등의 함수를 사용한 형 변환이 가능하다. 예를 들어 다음은 문자열 벡터를 Factor로 변환했다가 다시 숫자형 벡터로 변환하는 코드이다.

```
> x <- c("m", "f")
> as.factor(x)
[1] m f
Levels: f m
> as.numeric(as.factor(x))
[1] 2 1
```

f가 알파벳 순서상 m보다 앞이므로 as.factor(c("m", "f"))의 결과에서 levels의 "f m" 으로 정해진다. 따라서 as.numeric()에 "m", "f"의 factor들을 주면 levels 순서에 따라 2, 1이 반환된다.

만약 levels의 순서를 "m f"로 하고 싶다면 다음과 같이 factor() 함수를 써야한다.

```
> factor(c("m", "f"), levels=c("m", "f"))
[1] m f
Levels: m f
```

as.factor()는 levels 파라미터가 존재하지 않으므로, 이와 같은 순서 지정은 factor()를 통해 서만 가능하다. 따라서 상황에 따라 적절한 함수를 사용해 형 변환을 수행해야한다.

# 4

## 제어문, 연산, 함수

R의 기본적인 제어문, 연산, 함수는 기본적으로 다른 언어와 유사하지만 저장된 데이터를 한번에 다루는 벡터연산을 수행한다는 점, 데이터 분석이기 때문에 결측치(NA)가 중요한 주제로 떠오른다는 점이 다르다.

### 1 IF, FOR, WHILE 문

R의 IF문 구조는 다른 언어와 큰 차이가 없다.

```
if (TRUE) {  
  print('TRUE')  
  print('hello')  
} else {  
  print('FALSE')  
  print('world')  
}
```

위 코드의 결과는 TRUE, hello이다.

아래 코드에 보인 FOR문은 변수 i가 주어진 벡터에 있는 1, 2, 3, …, 10을 차례로 출력한다.

```
> for (i in 1:10) {  
+   print(i)  
+ }  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5
```

```
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

WHILE문 역시 다른 언어와 유사하다.

```
> i <- 0  
> while (i < 10) {  
+   print(i)  
+   i <- i + 1  
+ }  
[1] 0  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9
```

## 2 행렬 연산

행렬의 기본적인 연산인 합, 곱, 전치행렬(transpose), 역행렬을 다음과 같이 계산할 수 있다.

```
> x  
     [,1] [,2]  
[1,]    1    2  
[2,]    3    4  
> x + x  
     [,1] [,2]  
[1,]    2    4  
[2,]    6    8  
> x %*% x
```

```
[ ,1] [ ,2]
[1,]    7   10
[2,]   15   22
> t(x)
[ ,1] [ ,2]
[1,]    1    3
[2,]    2    4
> solve(x)
[ ,1] [ ,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

### 3 NA의 처리

결측치가 데이터에 포함되어 있을 경우 연산결과가 다음과 같이 NA로 바뀌어버리므로 주의가 필요하다.

```
> NA & TRUE
[1] NA
> NA + 1
[1] NA
```

이러한 문제점을 해결하기위해 많은 R 함수들이 na.rm 인자를 받는다. na.rm 은 NA값이 있을때 해당값을 제거할 것인지를 지정하기 위한 목적이다. 다음 예를 살펴보자.

```
> sum(c(1, 2, 3, NA))
[1] NA
> sum(c(1, 2, 3, NA), na.rm=T)
[1] 6
```

한편 잘알려진 머신 러닝 패키지 중 하나인 [caret: Classification and Regression Training](#)<sup>[7]</sup> 은 na.omit, na.pass, na.fail등을 na.action이라는 인자로 받아서 NA 처리를 어떻게 할지 결정 한다. 다음 예제는 이 셋의 차이를 보여준다.

```
> x <- data.frame(a=c(1, 2, 3), b=c("a", NA, "c"), c=c("a", "b", NA))
> x
  a     b     c
1 1     a     A
```

```

2 2 <NA>      B
3 3      c <NA>
> na.omit(x)
  a  b  c
1 1 a A
> na.pass(x)
  a      b      c
1 1      a      A
2 2 <NA>      B
3 3      c <NA>
> na.fail(x)
Error in na.fail.default(x) : missing values in object

```

즉, na.omit은 NA가 포함된 행을 제외하고, na.pass는 NA의 포함여부를 상관하지 않으며, na.fail은 데이터에 NA가 포함되어있을 경우 에러를 내보낸다. 따라서 NA를 어떻게 처리할지를 na.action으로 받았다면 ‘na.action(데이터 프레임)’을 실행해 현재 처리중인 데이터를 사용자가 원하는대로 정제할 수 있게 된다.

## 4 함수의 정의

함수는 ‘함수명 <- function(인자, 인자, ....) { 함수 본문 }’로 정의한다. 다음은 피보나치 함수를 구현한 예이다.

```

> fibo <- function(n) {
+   if (n == 1 || n == 2) {
+     return(1)
+   }
+   return(fibo(n - 1) + fibo(n - 2))
+ }
> fibo(1)
[1] 1
> fibo(5)
[1] 5

```

R에서 함수를 정의하는 방법은 이처럼 다른 언어의 함수 정의와 유사하지만 몇가지 차이점이 있다. 첫째는 값 반환시 ‘return 반환값’같은 형태가 아니라 함수 호출을 하듯이 ‘return(반환값)’을 적어주어야한다는 점이다. 두번째는 return()이 생략된다면 함수내 마지막 문장의 반

환값이 함수의 반환값이 된다는 점이다. 이 점을 사용하면 fibo()함수를 다음과 같이 고쳐쓸 수 있다.

```
> fibo <- function(n) {
+   if (n == 1 || n == 2) {
+     1
+   } else {
+     fibo(n - 1) + fibo(n - 2)
+   }
+ }
```

그러나 보통은 return()을 적어주는 편이 의도가 명확한 코드가 될 것이다.

함수를 호출할때는 인자의 위치를 맞춰서 값을 넘겨주는 방식, 또는 인자의 이름을 명명해서 넘겨주는 방식 두가지가 모두 가능하다. 다음은 이 두가지 방식을 모두 보여주는 예이다.

```
> f <- function(x, y) {
+   print(x)
+   print(y)
+ }
> f(1, 2)
[1] 1
[1] 2
> f(y=1, x=2)
[1] 2
[1] 1
```

R의 함수들의 도움말을 살펴보면 ‘...’를 인자 목록에 적은 경우를 종종 볼 수 있다. 함수의 인자에서 사용하는 ‘...’는 임의의 인자들을 받아서 다른 함수에 넘겨주는 용도로 주로 사용된다. 예를들어 다음에 보일 함수 g()는 인자 z는 자기 자신이 처리하지만 나머지 인자들은 함수 f로 넘긴다.

```
> f <- function(x, y) {
+   print(x)
+   print(y)
+ }
> g <- function(z, ...) {
+   print(z)
+   f(...)
```

```
+ }
> g(1, 2, 3)
[1] 1
[1] 2
[1] 3
```

함수안에 또 다른 함수를 정의하여 사용할 수 있다. 이를 중첩함수(Nested Function)이라고 부른다. 다음 코드는 함수 f()안에 함수 g()를 정의하고 함수 f()안에서 이를 호출하여 사용하는 경우를 보여준다.

```
> f <- function(x, y) {
+   print(x)
+   g <- function(y) {
+     print(y)
+   }
+   g(y)
+ }
> f(1, 2)
[1] 1
[1] 2
```

## 5 스코프(Scope)

코드에 기술한 변수 등을 지칭하는 이름이 어디에서 사용가능한지를 정하는 규칙을 스코프라고 한다. R에서는 대부분의 현대적인 프로그래밍 언어가 그러하듯이 lexical scope(또는 static scope라고도 함)를 사용한다.

콘솔에서 변수를 선언하면 모든 곳에서 사용 가능한 변수가 된다.

```
> n <- 1
> f <- function() {
+   print(n)
+ }
> f()
[1] 1
> n <- 2
> f()
[1] 2
```

함수 내부에서 변수 이름이 주어졌을 때 그 변수를 찾는 범위는 함수 내부가 우선된다.

```
> n <- 100
> f <- function() {
+   n <- 1
+   print(n)
+ }
> f()
[1] 1
```

만약 함수 내부에도, 전역 변수로도 없는 이름을 사용하면 에러가 된다. 다음 코드에서 rm(list=ls())는 모든 객체를 삭제하는 명령이다.

```
> rm(list=ls())
> f <- function() {
+   print(x)
+ }
> f()
Error in print(x) : object 'x' not found
```

마찬가지로 함수 내부에 정의한 이름은 바깥에서 접근 할 수 없다.

```
> rm(list=ls())
> f <- function() {
+   x <- 1
+ }
> f()
> x
Error: object 'x' not found
```

함수내에서 이름은 함수안의 변수들로 부터 먼저 찾는다. 같은 이유로 함수 인자의 변수명 역시 전역 변수에 우선한다.

```
> n <- 100
> f <- function(n) {
+   print(n)
+ }
> f(1)
```

```
[1] 1
```

중첩함수에도 같은 규칙이 적용된다. 다음 코드에서 함수 g 안에는 변수 a가 선언되어있지 않다. 따라서 R언어는 변수 a를 g함수를 감싼 함수 f로부터 찾게된다.

```
> f <- function(x) {
+   a <- 2
+   g <- function(y) {
+     print(y + a)
+   }
+   g(x)
+ }
> f(1)
[1] 3
```

그러나 만약 함수 f마저도 변수 a를 갖고 있지 않다면 전역 변수 a를 사용하게 된다.

```
> a <- 100
> f <- function(x) {
+   g <- function(y) {
+     print(y + a)
+   }
+   g(x)
+ }
> f(1)
[1] 101
```

이러한 스코프 규칙은 의도치 않은 결과를 낳을 수 있다. 예를들어 함수 f() 내에 변수 a가 있고 중첩된 함수 g에서 함수 f()내의 변수 a에 값을 할당하는 다음 코드를 살펴보자.

```
> f <- function() {
+   a <- 1
+   g <- function() {
+     a <- 2
+     print(a)
+   }
+   g()
+   print(a)
+ }
```

```
> f()
[1] 2
[1] 1
```

함수 g()에서 함수 f()에 선언된 변수 a에 2를 할당하려 하였지만 <-는 함수 g() 내부에 새로운 변수 a를 만들고 그 변수에 2를 할당한다. 따라서 f()에서 print(a)가 수행될 때 출력은 2가 아니라 1이 된다. 만약 함수 g()에서 함수 f()내의 변수 또는 전역 변수를 접근하려면 <<-를 사용해야 한다.

```
> b <- 0
> f <- function() {
+   a <- 1
+   g <- function() {
+     a <<- 2
+     b <<- 2
+     print(a)
+     print(b)
+   }
+   g()
+   print(a)
+   print(b)
+ }
> f()
[1] 2
[1] 2
[1] 2
[1] 2
```

g() 내에서 <<-를 사용한 값의 할당은 함수 f()에 선언된 a와 전역 변수인 b에 대해 이루어졌다.

## 6 벡터 연산

벡터 연산(Vectorized Computation 또는 [Array Programming](#))은 벡터 데이터 탑입을 지칭한다기보다는 벡터 또는 리스트를 한번에 연산하는 것을 말한다. 벡터 연산이 중요한 이유는 for문 등을 사용해 값을 하나씩 처리해나가는 대신 벡터나 리스트를 한번에 처리하는 것이 더 효율적이고 편리하기 때문이다. 가장 간단한 예로 다음과 같이 벡터내 저장된 값을 1씩 증가시

키는 경우를 살펴보자.

```
> x <- c(1, 2, 3, 4, 5)
> x + 1
[1] 2 3 4 5 6
```

벡터끼리 연산하는 것도 가능하다. 앞서 [진리값](#) (페이지 38)에서 설명하였듯이 벡터간 연산시에는 &&가 아니라 &를 사용한다.

```
> x <- c(1, 2, 3, 4, 5)
> x + x
[1] 2 4 6 8 10
> x == x
[1] TRUE TRUE TRUE TRUE TRUE
> x == c(1, 2, 3, 5, 5)
[1] TRUE TRUE TRUE FALSE TRUE
> c(T, T, T) & c(T, F, T)
[1] TRUE FALSE TRUE
```

R의 함수들은 기본적으로 이러한 벡터 기반 연산을 지원한다. 예를 들어 다음과 같이 sum(), mean(), median()등은 벡터를 곧바로 인자로 받을 수 있다.

```
> x <- c(1, 2, 3, 4, 5)
> sum(x)
[1] 15
> mean(x)
[1] 3
> median(x)
[1] 3
```

if - else문도 다음과 같이 한번에 적용가능하다. 다음은 2로 나눈 나머지를 사용해 짝수 (even), 홀수(odd)를 판별하는 예이다.

```
> x <- c(1, 2, 3, 4, 5)
> ifelse(x %% 2 == 0, "even", "odd")
[1] "odd"  "even" "odd"  "even" "odd"
```

이외에도 lapply등의 함수를 사용한 함수적 프로그래밍 방식도 유용하다. 이에 대해서는 [lapply\(\)](#) (페이지 92)에서 살펴본다.

벡터 연산을 사용하면 데이터 프레임에 저장된 데이터 중 원하는 정보를 쉽게 얻을 수 있다. 기본 원리는 데이터 프레임에 TRUE 또는 FALSE를 지정해 특정 행을 얻어올 수 있다는 점을 이용하는 것이다. 다음은 1행, 3행, 5행을 TRUE로 하여 해당 행들만 데이터 프레임에서 가져오는 예이다.

```
> (d <- data.frame(x=c(1, 2, 3, 4, 5), y=c("a", "b", "c", "d", "e")))
   x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
> d[c(TRUE, FALSE, TRUE, FALSE, TRUE), ]
   x y
1 1 a
3 3 c
5 5 e
```

따라서 행의 선택 기준이 되는 TRUE, FALSE를 벡터 연산으로 만들어주면 특정 행을 선택할 수 있게 된다. 다음은 x 값이 짝수인 행만 선택한 예이다. %% 연산자는 주어진 숫자로 나눈 나머지를 뜻하므로 2로 나눈 나머지가 0인 행들을 선택하고 있다.

```
> d[d$x %% 2 == 0, ]
   x y
2 2 b
4 4 d
```

## 7 값에 의한 전달

R에서는 모든 것이 객체이다. 또 객체는 함수 호출시 일반적을 값으로 전달된다. 이를 변수에 대한 참조(Reference)에 의한 방식에 대비해 값에 의한 전달(Pass By Value)이라고 한다. 값으로 전달된다는 말은 객체가 복사되어 함수로 전달된다는 의미이다<sup>1)</sup>.

예를 들어 다음과 같이 데이터 프레임을 함수에 주었을 때 함수 내부에서 수행한 변경은 원래 객체에는 반영되지 않는다. 왜냐하면 함수 f안에서의 df2란 f를 호출한 측에서 넘긴 df

---

<sup>1)</sup>모든 객체가 다 값에 의해 전달되지는 않는다. <http://homepage.stat.uiowa.edu/~luke/R/references.html>를 참조하기 바란다.

객체를 가리키는 참조가 아니라 넘겨받은 df를 복사한 새로운 df2이기 때문이다.

```
> f <- function(df2) {
+   df2$a <- c(1, 2, 3)
+ }
>
> df <- data.frame(a=c(4, 5, 6))
> f(df)
> df
  a
1 4
2 5
3 6
```

만약 인자로 받은 df 를 수정하여 이를 반영하려면 함수 f에서 다음과 같이 return을 해주어야한다.

```
> f <- function(df) {
+   df$a <- c(1, 2, 3)
+   return(df)
+ }
>
> df <- data.frame(a=c(4, 5, 6))
> df <- f(df)
> df
  a
1 1
2 2
3 3
```

특별한 객체(예를들어 네트워크 접속 connection이나 파일 입출력 connection 등)를 제외하고는 객체의 상태가 함수에 의해 직접적으로 수정되지 않는다. 따라서 어떤 함수를 호출하더라도 인자로 넘긴 객체가 수정되지 않음을 보장받는다. 더 관심있는 독자는 [Mutable objects in R](#)을 참고하기 바란다.

여기까지 읽은 독자는 ‘그렇다면 객체가 함수에 넘겨질때마다 새로운 객체를 만들어야하므로 메모리 사용량이 너무 많아지지 않을까?’라는 의심을 품게 될 것이다. 실제로 매번 복사가 일어나야 한다면 그러한 부담이 있는 것이 사실이다. 그러나 R을 포함한 현대적 언어들은 함수 호출시 곧바로 객체를 복사하기 보다는 객체의 값을 바꿀 필요가 있을 때만 복사를 수행하

는 [Copy On Write](#) 기법을 사용한다. Copy on Write에서는 데이터 프레임 등의 객체를 어떤 함수에 넘겼을 때 그 함수가 객체 내부 값을 바꾸는 시점에 이르러서야 값이 수정된 새로운 복사본을 만든다.

## 8 객체의 불변성

R의 객체는 (거의 대부분의 경우에) 불변이다. OOP(Object Oriented Programming. 객체 지향 프로그래밍)에 친숙하지 않은 독자라면, 객체라는 용어를 R에서 메모리에 할당된 데이터 구조들을 뜻한다고 생각하면 된다. 예를 들어 벡터를 하나 메모리에 만들었으면 그것이 객체이고, 리스트를 하나 만들었어도 객체이다.

이런 객체들은 불변(immutable)이다[8]. 프로그래밍에서 불변이란 데이터를 수정하는 것이 불가능하단 말이다. 따라서 마치 객체를 수정하는 것 처럼 보이는 다음 코드는 실제로 원래 객체를 변경하는 것은 아니다.

```
> a <- list()
> a$b <- c(1, 2, 3)
```

코드로부터 짐작되는 일은 리스트 a에 새로이 필드 b를 만들고 거기에 c(1, 2, 3)을 할당하는 수정처럼 보인다. 그러나 실제로 일어나는 일은 a를 복사한 새로운 객체 a'을 만들고 a'에 필드 b를 추가하고 그 필드에 c(1, 2, 3)을 채워넣은 다음, 변수명 a가 a'을 가리키도록 하는 것이다.

그림 4.1을 살펴보자. 최초의 변수 a는 어떤 객체에 붙여진 이름이었다. 그러나 그 객체의 값을 바꾸면 이전 객체는 버려지며 새로이 b=c(1, 2, 3)을 담고 있는 새로운 객체가 만들어지고 a는 이 새로운 객체에 대한 이름이 된다.

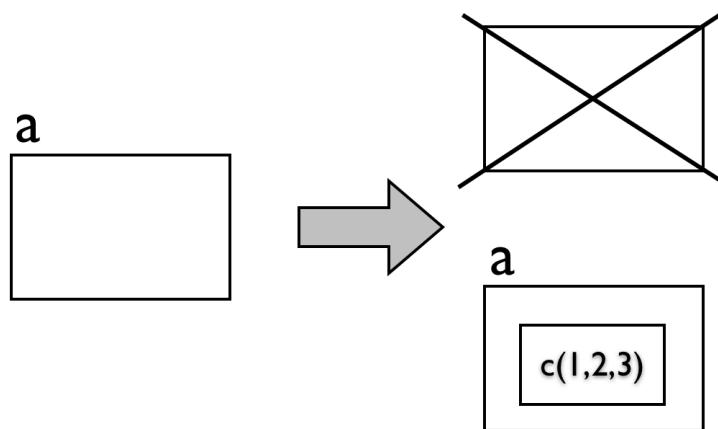


그림 4.1: 객체의 불변성

앞서 [벡터 연산](#) (페이지 71)에서 벡터 기반 연산을 사용하는 것이 for 등의 반복문을 사용하는 것보다 효율적이라고 한 바 있다. 그 이유 중 하나가 바로 객체가 불변이라는 점 때문이다. 예를 들어 for 문 안에서 벡터의 인자를 하나씩 바꾸는 다음 코드는  $v[i]$  값을 1씩 증가시킬 때마다  $i$ 번째 값이 수정된 벡터를 매번 새로 만들고 이를 매번  $v$ 에 할당한다. 따라서 1000개의 새로운 객체를 생성하는 비효율성이 발생한다.

```
> v <- 1:1000
> for (i in 1:1000) {
+   v[i] <- v[i] + 1
+ }
```

반면 다음 코드는  $v$  내 전체 값을 1 만큼 증가시킨 객체를 한 개 만든 다음 이를  $v$ 에 할당한다.

```
> v <- 1:1000
> v <- v + 1
```

이처럼 벡터 연산이 더 빠른 이유는 이러한 메모리 사용 최적화의 문제와도 관련이 있다[8]. 이를 확인하는 한 가지 방법은 다음 코드를 실행해 보는 것이다.

```
> rm(list=ls())
> gc()
> v <- 1:99999999
> for (i in 1:99999999) {
+   for (j in 1:99999999) {
+     v[j] <- v[j] + 1
+   }
+ }
```

위 코드의 첫 줄은 모든 선언된 변수를 삭제하기 위한 명령이다. 둘째 줄은 가ベ지 컬렉션 (Garbage Collection)을 호출해 현재 사용하지 않는 변수들을 메모리에서 삭제하기 위한 명령이다. for 문이 수행되는 동안 작업 관리자를 살펴보면 메모리 사용량이 서서히 증가함을 알 수 있다.

## 9 모듈(Module) 패턴

이 절에서는 모듈 패턴에 대해 살펴본다. 모듈이란 외부에서 접근할 수 없는 데이터와 그 데이터를 제어하기 위한 함수로 구성된 구조물을 말한다. 패턴이란 정형화된 코딩 기법을 뜻한다.

모듈 패턴을 사용하여 데이터를 외부에서 직접 접근 할 수 없게 하면 데이터의 내부 구현이 숨겨진다. 반대로 말하자면 모듈의 사용자가 모듈 내부에서 데이터가 어떻게 저장되는지 더 이상 신경 쓸 필요가 없게 된다는 의미이다.

또한 사전에 정의된 함수로만 그 데이터를 다룰 수 있게 하면 데이터의 내부 구조를 잘 모르는 사용자가 데이터를 함부러 건드려서 손상시키는 일을 막을 수 있다.

마지막으로 모듈의 사용자는 내부 구조는 건들 수 없고 외부로 노출된 함수만 불러쓰고 있는 상태가 되므로, 모듈의 제작자는 해당 함수가 이전과 같은 결과를 내놓기만 한다면 함수의 내부구조나 데이터 구조를 마음대로 바꿀 수 있다.

이 절에서는 잘 알려진 자료구조인 큐(Queue)를 모듈로 작성해본다.

## 9.1 큐(Queue)

큐는 먼저 들어온 데이터를 먼저 처리하는데 사용하는 자료구조이다. 사람들이 줄을 차례로 서고 줄의 제일 앞에 서있는 사람부터 자신의 일을 처리하는 모습을 연상하면 된다. 큐는 다음 다음 3개의 함수로 구현할 수 있다.

- Enqueue: 줄의 맨 뒤에 데이터를 추가한다.
- Dequeue: 줄의 맨 앞에 있는 데이터를 가져온다. 가져온 데이터는 줄에서 빠진다.
- Size: 줄의 길이, 즉 자료구조내에 저장된 데이터의 수를 반환한다.

이들 세가지 함수를 지원하는 큐를 작성해보자.

```
> q <- c()
> q_size <- 0
> enqueue <- function(data) {
+   q <<- c(q, data)
+   q_size <<- q_size + 1
+ }
> dequeue <- function() {
+   first <- q[1]
+   q <<- q[-1]
+   q_size <<- q_size - 1
+   return(first)
+ }
```

```
> size <- function() {
+   return(q_size)
+ }
```

위 코드에서 큐에 저장될 데이터는 벡터 q를 사용해 저장하고 있다. q\_size는 큐에 저장된 데이터 수를 기록하는 목적으로 쓰였다.

함수 enqueue()는 q에 이미 저장되어있는 데이터에 인자로 받은 데이터를 추가하여 다시 변수 q에 할당하였다. 이 때 <<-를 사용해 전역변수에 있는 q를 직접 접근하게 하였다. 마지막으로 q\_size의 값을 1 증가시켰다.

함수 dequeue()는 q에 저장된 데이터중 첫번째 요소를 first에 저장하고, q에는 이 데이터를 제외한 데이터를 저장한다음 first를 반환하였다. 이 때 q\_size 를 1 감소시킨다.

함수 size()는 q 의 길이인 q\_size를 반환한다.

위 코드는 다음과 같이 사용할 수 있다.

```
> enqueue(1)
> enqueue(3)
> enqueue(5)
> print(size())
[1] 3
> print(dequeue())
[1] 1
> print(dequeue())
[1] 3
> print(dequeue())
[1] 5
> print(size())
[1] 0
```

## 9.2 큐(Queue) 모듈의 작성

앞 절에서 작성한 큐 코드의 작성자는 큐의 맨뒤에 데이터를 저장하고 큐의 맨 앞에서 데이터를 가져오도록 설계하였다. 하지만 q라는 변수가 전역적으로 선언되어있으므로 이 함수를 거치지 않고 외부에서 데이터를 조작해버릴 수 있다. 그리고 이 때 데이터의 무결성이 다음과 같이 깨질 수 있다.

```
> enqueue(1)
```

```
> q <- c(q, 5)
> print(size())
[1] 1
> dequeue()
[1] 1
> dequeue()
[1] 5
> size()
[1] -1
```

보다시피 q의 외부에서 q에 직접적으로 값을 할당해버린 탓에 실제로는 q내부에 데이터가 2개 저장되어 있지만 size()를 호출했을 때 크기가 1로 나타나는 문제가 있다.

이러한 문제를 막기 위한 방법은 큐 코드 전체를 하나의 함수로 감추어 모듈화 시키는 것이다.

```
> queue <- function() {
+   q <- c()
+   q_size <- 0
+
+   enqueue <- function(data) {
+     q <<- c(q, data)
+     q_size <<- q_size + 1
+   }
+
+   dequeue <- function() {
+     first <- q[1]
+     q <<- q[-1]
+     q_size <<- q_size - 1
+     return(first)
+   }
+
+   size <- function() {
+     return(q_size)
+   }
+
+   return(list(enqueue=enqueue, dequeue=dequeue, size=size))
+ }
```

이 코드가 앞서와 다른 점이라면 q와 q\_size가 이제 queue 함수내의 지역변수라는 점이다. 따라서 이 변수들은 외부에서 접근 불가능하다. 또 함수 enqueue(), dequeue(), size()는 함수 queue()의 반환값이 되었고 이 값은 리스트로 반환되고 있다.

이렇게 만들어진 queue()는 다음과 같이 사용할 수 있다.

```
> q <- queue()
> q$enqueue(1)
> q$enqueue(3)
> q$size()
[1] 2
> q$dequeue()
[1] 1
> q$dequeue()
[1] 3
> q$size()
[1] 0
```

queue() 함수 호출시 만들어지는 queue() 함수 내부의 지역변수 q와 q\_size 가 생성되는 공간은 queue() 함수 호출때마다 매번 새로 생성된다. 즉 queue()를 다음과 같이 여러개 만들어서 사용해도 데이터가 서로 섞이지 않게 된다.

```
> q <- queue()
> r <- queue()
> q$enqueue(1)
> r$size()
[1] 0
> r$enqueue(3)
> q$dequeue()
[1] 1
> r$dequeue()
[1] 3
> q$size()
[1] 0
> r$size()
[1] 0
```

이러한 코딩 기법을 모듈 패턴이라고 하며 Software for Data Analysis[8]에 소개되어 있다. 그러나 이는 비단 R만의 전유물은 아니며 자바스크립트에서도 같은 이름으로 널리 사용되어

JavaScript Module Pattern: In-Depth 등의 문서에 소개되어 있다.

## 10 객체의 삭제

메모리 상에 만들어진 객체의 목록을 얻는 함수는 ls()이다.

```
> x <- c(1, 2, 3, 4, 5)
> ls()
[1] "x"
```

때에 따라서는 메모리 상에 생성해둔 객체를 삭제할 필요가 있을 수 있다. 예를 들어 객체를 너무 많이 만들어서 메모리가 부족해졌다거나, 또는 내 프로그램에서 변수 x를 사용하는 경우가 있다면 이미 메모리에 있는 객체 x를 실수로 가져다 쓰는 일이 없도록 하기 위해서의 경우를 생각해볼 수 있다. 객체의 삭제는 rm()을 사용한다.

```
> rm("x")
> ls()
character(0)
```

모든 객체를 한번에 메모리에서 삭제하는 방법은 다음과 같다.

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

## 데이터 조작 I

앞 장에서 기본적인 데이터 탑입에 대해서 살펴보았다. R은 C, C++, Python, Java, Ruby 같은 일반적인 목적의 언어와 달리 벡터 기반의 데이터 처리가 자주 사용된다. R에서의 벡터 기반의 처리는 개별 원소를 for 루프 등으로 하나씩 보면서 처리하는 방식보다 종종 빠르게 수행될 뿐만 아니라 손쉽게 병렬화가 가능하다.

R에서 데이터 처리를 잘 하려면 상당히 많은 함수에 능숙해질 필요가 있고, 데이터의 양이 많다면 그 처리 속도가 MySQL같이 대량 데이터에 특화된 도구를 다루는 경우에 비해 특별히 뛰어나지 않은 경우가 있다. 따라서 R이외에 다른 언어를 사용해 미리 데이터를 처리하여 CSV나 TSV형식으로 파일을 저장한 뒤 이를 R에서 읽어들이거나, R에서 MySQL에 직접 접근하면서 데이터를 처리하는 것도 한가지 방법이 될 수 있다.

### 1 iris 데이터

본격적으로 데이터 조작을 알아보기 위해 앞서, 앞으로 데이터 처리 및 머신 러닝 기법에 예제로 사용할 iris ([http://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](http://en.wikipedia.org/wiki/Iris_flower_data_set)) 데이터 셋에 대해 살펴보자. iris는 통계학자인 Fisher가 소개한 데이터로서, 붓꽃의 3가지 종(setosa, versicolor, virginica)에 대해 꽃받침(sepal)과 꽃잎(petal)의 길이를 정리한 데이터이다. iris의 각 행에 저장된 데이터는 다음과 같다.

- Species: 붓꽃의 종. setosa, versicolor, virginica의 세가지 값 중 하나를 저장한 범주형 변수.
- Sepal.Width: 꽃받침의 너비. Number 변수.
- Sepal.Length: 꽃받침의 길이. Number 변수.
- Petal.Width: 꽃잎의 너비. Number 변수.
- Petal.Length: 꽃잎의 길이. Number 변수.

iris에는 각 종별로 50개씩, 총 150개의 행이 저장되어있다. 이 데이터는 R에 기본적으로 포함되어 있어 다양한 통계 기법이나 머신 러닝 기법을 테스트해보기에 좋다. 다음은 iris데이터를 간략히 살펴본 예이다.

```
> 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 ...
```

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

```
> iris3
, , Setosa

  Sepal L. Sepal W. Petal L. Petal W.
[1,]     5.1     3.5     1.4     0.2
[2,]     4.9     3.0     1.4     0.2
...
, , Versicolor

  Sepal L. Sepal W. Petal L. Petal W.
[1,]     7.0     3.2     4.7     1.4
[2,]     6.4     3.2     4.5     1.5
...
```

```
, , Virginica

Sepal L. Sepal W. Petal L. Petal W.
[1,] 6.3 3.3 6.0 2.5
[2,] 5.8 2.7 5.1 1.9
...

```

이 외에도 R에는 다양한 데이터 셋이 준비되어 있다. 데이터 셋 목록은 ‘library(help=datasets)’ 명령을 통해 살펴볼 수 있다. 실제 데이터를 사용할 때는 ‘data(데이터 셋 이름)’의 형태로 하면 된다. 예를 들어 mtcars를 살펴보려면 다음과 같이 한다.

```
> data(mtcars)
> 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 3.85 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
```

mtcars 데이터 셋의 상세 내용을 알고 싶다면 ‘?mtcars’ 또는 ‘help(mtcars)’ 을 사용한다.

## 2 파일 입출력

### 2.1 CSV파일 입출력

CSV파일을 읽으려면 `read.csv()`를 사용한다. 이 함수는 파일명과 첫째행이 header인지를 구분하기 위한 파라미터를 받는다. 따라서 보통 `read.csv(파일명, header=TRUE)`와 같은 형식으로 호출하면 된다.

예를 들어 다음과 같은 CSV파일이 있다고 하자.

```
$ cat a.csv
id,name,score
1,"Mr. Foo",95
2,"Ms. Bar",97
3,"Mr. Baz",92
```

이 파일의 첫행은 각 컬럼명을 나타내는 레이블이다. `read.csv()`를 사용해 파일을 읽어보자.

```
> x <- read.csv("a.csv")
> x
  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
```

읽어들인 파일은 데이터 프레임으로 반환된다.

만약 csv 파일에 헤더행이 없다면 다음과 같이 `header=FALSE`를 지정한다. 이 경우 열의 이름이 주어지지 않게 되므로 다음 예에서 보인 바와 같이 `names()`를 사용해 별도로 열이름을 지정해야한다.

```
$ cat b.csv
1,"Mr. Foo",95
2,"Ms. Bar",97
3,"Mr. Baz",92

$ R

> x <- read.csv("b.csv")
> x
  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
```

또한가지 주의할 점이 있다. 위에서 데이터를 읽어들인 결과를 보면 “name” 열이 모두 Factor 형태로 변환되었다. 실제 데이터 분석을 할 때는 성명과 같은 문자열 속성을 사용할 일은 거의 없을것이므로 주어진 문자열을 Factor로 변환하는 것은 자연스러운 일이다. 그러나 위 예제 데이터의 목적에는 맞지 않으므로 다음과 같이 다시 문자열로 변환해주어야한다.

```
> x$name = as.character(x$name)
> 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 Factor가 아니라 문자열 타입으로 읽어들이도록 stringsAsFactors=TRUE를 지정해도 된다.

```
> x <- read.csv("a.csv", stringsAsFactors=FALSE)
> 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
```

때에 따라서는 데이터에 다음과 같이 NA를 지정하는 문자열이 저장되어 있을 때도 있다.

```
$ cat c.csv
id,name,score
1,"Mr. Foo",95
2,"Ms. Bar",NIL
3,"Mr. Baz",92
```

이 데이터를 read.csv()로 읽어들이면 다음에 보인 바와 같이 NIL이 문자열로 인식된다. 그 결과 95와 92가 모두 문자열로 변환되어버린다.

```
> x <- read.csv("c.csv")
> x
```

```

id      name  score
1  1 Mr. Foo     95
2  2 Ms. Bar     NIL
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: Factor w/ 3 levels " 92"," 95"," NIL": 2 3 1

```

이러한 결과를 피하려면 na.strings 인자를 사용한다. na.strings의 기본값은 “NA”로서 NA라는 문자열이 주어지면 이를 R이 인식하는 NA로 바꿔준다. 이 예에서는 na.strings=“NIL”을 다음과 같이 사용하면 된다.

```

> x <- read.csv("c.csv", na.strings=c("NIL"))
> 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 NA 92
> is.na(x$score)
[1] FALSE  TRUE FALSE

```

is.na()로 확인해본 결과 NIL이 NA로 잘 변환되었음을 볼 수 있다. help(read.csv)에는 더욱 많은 옵션이 설명되어있으니 필요시 참고하기 바란다.

데이터를 파일로 저장하려면 다음과 같이 하면 된다.

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

결과는 다음과 같다.

```

$ cat b.csv
"id","name","score"
1,"Mr. Foo",95
2,"Ms. Bar",97
3,"Mr. Baz",92

```

CSV형태로 파일을 저장할 때, rownames을 저장하지 않도록 지정했다. 만약 그런 설정을 빼고 CSV파일로 데이터를 저장하면 다음과 같은 결과를 얻는다.

```
$ cat b.csv
"", "id", "name", "score"
"1", 1, "Mr. Foo", 95
"2", 2, "Ms. Bar", 97
"3", 3, "Mr. Baz", 92
```

이 CSV파일의 첫열은 행번호 1, 2, 3을 의미하는데 이 값이 데이터 저장에 꼭 필요한 것은 아니므로 생략한 것이다.

### 3 save(), load()

데이터를 다양한 알고리즘으로 처리한 뒤 저장할 필요가 있다면 R 객체를 그대로 파일로 저장할 수 있다. 다음은 두 벡터를 xy.RData 파일에 저장하는 예이다.

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

파일로부터 데이터를 불러들이는 함수는 load()이다. 다음 코드는 메모리 상에 있는 모든 객체를 삭제한 뒤 파일로부터 x, y 객체를 불러들이는 예를 보여준다.

```
> rm(list=ls())
> x
Error: object 'x' not found
> y
Error: object 'y' not found
> load("xy.RData")
> x
[1] 1 2 3 4 5
> y
[1] 6 7 8 9 10
```

### 4 rbind(), cbind()

rbind()와 cbind()는 각각 행 또는 열형태로 주어진 데이터를 합쳐서 행렬 또는 데이터 프레임을 만드는데 사용된다. 예를 들어, [1, 2, 3], [4, 5, 6] 의 2개 벡터는 다음과 같이 하나의 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.

```
> x <- data.frame(id=c(1, 2), name=c("a", "b"), stringsAsFactors=F)
> x
  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"))
> y
  id name
1  1    a
2  2    b
3  3    c
```

위 코드에서 stringsAsFactors 는 name 컬럼의 데이터를 Factor가 아니라 문자열로 취급하기 위해 필요하다. 만약 stringsAsFactors를 지정하지 않으면 "a", "b"는 범주형 데이터로 취급되어 이름을 표현하려는 목적에 어긋나게 된다.

cbind()는 주어진 인자를 열(column)로 취급하여 데이터를 합친다.

```
> 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 행을 추가할 수 있다.

```
> y <- cbind(x, greek=c('alpha', 'beta'))
```

```

> y
  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 있듯이 stringsAsFactors를 FALSE로 지정하면 새로 추가된 greek 열이 문자열 자료가 되지만, 이를 생략하면 범주형 자료인 Factor가 된다.

데이터 프레임에 새로운 열을 추가할 때는 cbind() 를 사용하지 않고 ‘변수명\$컬럼명 <- 데이터’ 형태로도 열을 추가할 수 있다. 이 내용은 앞서 살펴본 [데이터 프레임\(Data Frame\)](#) (페이지 54)을 참고하기 바란다.

## 5 apply 함수들

R에는 다양한 벡터 또는 행렬데이터에 임의의 함수를 적용한 결과를 얻기 위한 apply류의 함수들이 있다. 이들 함수는 벡터, 행렬, 리스트, 데이터 프레임에 적용할 함수를 명시하는 형태로서, 함수형 언어 스타일에 가깝다고 볼 수 있다. 따라서 다른 언어에만 익숙한 사람들이 이 절에서는 이들 중 종종 사용되는 apply, lapply, sapply, tapply에 대해서 알아본다.

### 5.1 apply()

apply() 는 행렬의 행 또는 열방향으로 특정 함수를 적용하는데 사용되며, apply(행렬, 방향, 함수) 형태로 호출한다. 이때 ‘방향’은 1이 주어지면 행, 2가 주어지면 열을 뜻한다. apply()를 수행한 결과값은 벡터, 배열, 리스트 중 적합한 것으로 반환된다. 반환 값 유형에 대한 자세한 규칙은 ?apply를 참고하기 바란다.

apply()에 앞서 먼저 합을 구하는 함수 sum()에 대해서 알아보자. sum()은 인자로 주어진 값들의 합을 구하는 간단한 함수이다. 예를 들어 다음은 1부터 10까지의 합을 계산한다.

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

자 이제 `apply()`를 사용해 행렬에 저장된 데이터의 합을 구해보자. 예를 들어 다음과 같은 행렬이 있다고 하자.

```
> 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,  $1+4+7$ ,  $2+5+8$ ,  $3+6+9$ )를 구하려면 `apply`를 행별로(즉 방향에 1을 지정) 처리하되 각 행에 대해 `sum` 함수를 호출하면 된다.

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

마찬가지로 열방향 합( $1+2+3$ ,  $4+5+6$ ,  $7+8+9$ )은 다음과 같다.

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

`apply()`를 사용하여 `iris` 데이터의 Sepal.Length, Sepal.Width, Petal.Length, Petal.Width 컬럼의 합을 구해보자. 다음 코드에서 `iris[, 1:4]`는 `iris` 데이터 프레임의 모든 행에서 1~4열만 가져온다는 의미이다.

```
> 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
...
> apply(iris[, 1:4], 2, sum)
Sepal.Length Sepal.Width Petal.Length Petal.Width
           876.5      458.6      563.7      179.9
```

이와 같은 행 또는 열의 합 계산은 빈번히 사용되므로 `rowSums()`, `colSums()` 함수가 정의되어 있다. 다음은 앞서 `apply()`를 통해 구한 계산을 `colSums()`으로 수행하는 예이다.

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

행, 열방향의 평균역시 rowMeans() 또는 colMeans()를 통해서도 계산할 수 있다.

## 5.2 lapply()

lapply()는 lapply(X, 함수)의 형태로 호출하며 이때 ‘X’는 벡터 또는 리스트이고, ‘함수’는 ‘X’내 각 요소에 적용할 함수이다. 함수를 적용한 결과는 리스트로 반환된다.

예를들어 1, 2, 3으로 구성된 벡터가 있을때, 각 숫자를 2배한 값을 lapply()를 통해 구하는 예는 다음과 같다. 참고로 앞서 [리스트\(List\)](#) (페이지 46)에서 살펴보았듯이, 리스트의 각 값은 [[n]] (이때 n은 접근할 요소의 색인) 형태로 접근할 수 있다.

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

[[2]]
[1] 4

[[3]]
[1] 6

> result[[1]]
[1] 2
```

위의 예처럼 lapply()에 벡터를 입력으로 준 경우에는 리스트로 나온 결과를 다시 벡터로 변환하고 싶을때가 많다. 이때 사용할 수 있는 한가지 방법은 unlist()를 호출하는 것이다.

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

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

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

```
> x
$a
[1] 1 2 3

$b
[1] 4 5 6

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

$b
[1] 5
```

마찬가지로 데이터 프레임에도 곧바로 lapply()를 적용할 수 있다.

```
> 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 설명했듯이 각 열의 평균은 colMeans()로도 계산할 수 있다.

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

데이터 프레임을 처리한 결과를 리스트로 얻게되면, 그 리스트를 다시 데이터 프레임으로 변환할 필요가 있다. 이때는 몇단계를 거쳐서 처리해야한다.

- 1) unlist()를 통해 리스트를 벡터로 변환한다.

- 2) matrix()를 사용해 벡터를 행렬로 변환한다.
- 3) as.data.frame()<sup>1)</sup>을 사용해 행렬을 데이터 프레임으로 변환한다.
- 4) 마지막으로 names()를 사용해 리스트로 부터 변수명을 얻어와 데이터 프레임의 각 열에 이름을 부여한다.

```
> d <- as.data.frame(matrix(unlist(lapply(iris[, 1:4], mean)),  
+ ncol=4, byrow=TRUE))  
  
> 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 사용할 수도 있다. 형식은 ‘do.call(호출할 함수, 파라미터)’와 같은 방식이다. 지금 살펴보는 예제의 경우에는 lapply()가 반환한 리스트내에 각 컬럼별 계산결과가 들어있다. 따라서 이를 새로운 데이터 프레임의 컬럼들로 합치기 위해 cbind()를 사용한다. 다음 코드는 do.call()을 사용해 lapply()의 결과로 나온 리스트내 요소 하나 하나를 cbind()의 파라미터로 넘겨준다.

```
> data.frame(do.call(cbind, lapply(iris[, 1:4], mean)))  
  
Sepal.Length Sepal.Width Petal.Length Petal.Width  
1      5.843333     3.057333      3.758      1.199333
```

위에서 살펴본 두가지 방법중 unlist()후 matrix를 거쳐 데이터 프레임으로 변환하는 방법은 리스트에 저장된 데이터 프레임을 하나의 데이터 프레임으로 합치는 일반적인 방법으로 사용하기에는 한가지 문제가 있다. unlist()는 벡터를 반환하며 벡터에는 한가지 데이터 타입만 저장할 수 있기 때문에 변환 과정에서 한가지 데이터 타입으로 데이터가 모두 형변환 되어버리기 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
```

---

<sup>1)</sup>이와 유사한 함수로 주어진 객체를 벡터로 변환하는 as.vector(), 리스트로 변환하는 as.list(), factor로 변환하는 as.factor()등의 함수가 있다.

따라서 데이터 타입이 혼합된 경우에는 do.call()을 사용한다. 다음 예에서는 리스트의 각 요소가 한 행에 해당하므로 rbind를 호출하면 된다.

```
> 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, ...)는 속도가 매우 느리다는 단점이 있다. 뒤에서 살펴볼 것이지만 데이터 양이 많다면 [rbindlist](#) (페이지 148)를 사용해야 한다.

### 5.3 sapply()

sapply()는 lapply()와 유사하지만 리스트대신 행렬, 벡터 등으로 결과를 반환하는 함수이다. 입력으로는 벡터, 리스트, 데이터 프레임등이 쓰일 수 있다.

예를 들어 iris의 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()에 의해 반환된 벡터는 as.data.frame()을 사용해 데이터 프레임으로 변환할 수 있다. 이 때, t(x)를 사용해 벡터의 행과 열을 바꿔주지 않으면 기대한 것과 다른 모양의 데이터 프레임을 얻게 되므로 이를 참고하기 바란다.

```
> 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
>
```

또한 sapply()를 각 열에 저장된 데이터의 클래스를 알아내는데 사용할 수 있다.

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

만약 sapply()에 인자로 주어진 함수의 출력이 여러 행이라면 행렬이 반환된다. 다음 예에서는 iris의 숫자형 컬럼들에 대해 각 값이 3보다 큰지의 여부를 반환한다. 그리고 이때 반환된 결과의 클래스는 행렬이다.

```
> y <- sapply(iris[, 1:4], function(x) { x > 3 })
> class(y)
[1] "matrix"
> head(y)

  Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]        TRUE      TRUE     FALSE     FALSE
[2,]        TRUE     FALSE     FALSE     FALSE
[3,]        TRUE      TRUE     FALSE     FALSE
[4,]        TRUE      TRUE     FALSE     FALSE
[5,]        TRUE      TRUE     FALSE     FALSE
[6,]        TRUE      TRUE     FALSE     FALSE
...
```

sapply()는 한가지 타입만 저장가능한 데이터 타입인 벡터, 행렬, 배열을 반환하므로 sapply()에 인자로 주어진 함수의 반환값에 여러가지 데이터 타입이 섞여있어서는 안된다. 만약 각 컬럼들에 대해 수행한 함수의 결과 데이터 타입이 서로 다르다면, 리스트를 반환하는 lapply()나 리스트 또는 데이터 프레임을 반환할 수 있는 [plyr 패키지](#) (페이지 121)을 사용해야한다.

## 5.4 tapply

tapply()는 그룹별 처리를 위한 apply 함수로서 tapply(데이터, 색인, 함수)의 형태로 호출한다. 여기서 ‘색인’은 데이터가 어느 그룹에 속하는지를 표현하기 위한 factor 형 데이터이다. 즉, tapply는 데이터가 주어졌을 때 각 데이터가 속한 그룹별로 주어진 함수를 수행한다.

다음과 같은 예를 생각해보자. 1부터 10까지의 숫자가 있고 이들이 모두 한 그룹에 속해 있을 때 각 그룹에 속한 데이터의 합을 구한다면, 그 합은 55가 될 것이다.

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

위 코드에서 rep(1, 10)은 1을 10회 반복하는 것을 의미한다. 따라서 1, 2, 3, …, 10의 숫자에 대해 동일한 소속번호 1, 1, 1, …, 1 를 부여한 것이다. 그러므로 그룹 1에 속한 데이터의 합은 55( $=1+2+3+\dots+10$ )이다.

이번에는 1부터 10까지의 숫자를 홀수별, 짝수별로 묶어서 합을 구해보자.

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

위 코드에서 `%%` 는 나머지를 구하는 연산자이다<sup>2)</sup>. 수행 결과 짝수의 합이 30( $=2+4+6+8+10$ ), 홀수의 합이 25( $=1+3+5+7+9$ )로 구해졌다.

iris 데이터에서 Species별 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 다뤄보도록하자. 예를 들어 다음과 같은 행렬이 있다고 하자.

---

<sup>2)</sup> 물은 `%/%` 연산자를 사용해 구할 수 있다.

```

> 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. 이 때, 반기별 남성 셀의 값의 합과 여성셀의 합을 구해보자. 다시말해, 상반기(봄, 여름)의 남성 셀의 합( $=1+2$ ), 여성셀의 합( $=5+6$ )과 하반기(가을, 겨울)의 남성셀의 합( $=3+4$ ), 여성셀의 합( $=7+8$ )을 구하는 것이 목표이다. 다음은 이를 구현한 예이다.

```

> 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

```

위 코드에서 색인 파라미터내 첫번째 벡터는 상반기인지 하반기인지의 구분을 표현하였고, 두번째 벡터는 성별을 표시하고 있다. 그룹을 나타내는 색인은 데이터의 좌측에서 우측으로 상단에서 하단으로 가면서 부여된다. 따라서 색인의 첫번째 리스트의 1, 1, 2, 2, 1, 1, 2, 2는 차례대로 (spring, male), (summer, male), (fall, male), (winter, male), (spring, female), (summer, female), (fall, female), (winter, female)의 그룹을 나타내는 색인이다. 성별을 나타낸 1, 1, 1, 2, 2, 2, 2 색인 역시 같은 방식으로 먼저 male 데이터에 그룹을 부여하고 다음으로 female 데이터에 그룹을 부여한다. 최종적으로 sum이 동작할때는 같은 그룹에 속한 데이터끼리 연산이 수행되므로 성별/반기별 합이 계산된다.

tapply()는 같은 클러스터에 속한 데이터들의 x좌표의 평균, y좌표의 평균을 계산하는데 사용할 수 있는데, 바로 그때 위와 같은 방식으로 색인을 부여한다. 따라서 조금 복잡해 보여도 위 내용을 꼭 알고 넘어가도록 하자.

## 5.5 mapply()

이제 apply()함수들의 마지막 변형으로 mapply()에 대해 살펴보자. mapply()는 sapply()와 유사하지만 다수의 인자를 함수에 넘긴다는데서 차이가 있다. 예를 들어 숫자 1, 2, 3과 문자열 “a”, “b”, “c”를 짹지어서 (1, “a”), (2, “b”), (3, “c”)로 묶어 함수에 넘기는 다음 예를 살펴보자.

```
> mapply(function(i, s) {
+   sprintf("%d%s", i, s)
+ }, 1:3, c("a", "b", "c"))
[1] "1a" "2b" "3c"
```

위 코드에서 mapply()에 주어진 인자는 c(1, 2, 3)과 c(“a”, “b”, “c”)이다. mapply()는 주어진 인자들의 값을 하나씩 묶어 (1, “a”), (2, “b”), (3, “c”)로 만들고 이를 function(i, s)에 넘긴다. sprintf()는 데이터를 문자열로 변환하는데 사용하며 인자로 데이터를 어떻게 문자열로 변환할지를 지정하는 포맷팅 문자열, 그리고 문자열로 변환할 변수들을 받는다. 위 코드에서는 “%d%s”를 사용했는데 %d는 정수를, %s는 문자열을 뜻한다. 포맷팅 문자열에 적용할 변수는 i와 s가 주어졌는데 이들 두 변수는 차례대로 포맷팅 문자열에 맞게 정수, 문자열로 변환된다. 이런 이유로 출력이 “1a”, “2b”, “3c”로 변환된다.

또 다른 예로 iris의 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
```

mapply의 인자로 iris[1:4]가 주어졌다. 따라서 mapply에는 iris의 모든 행이 나열되어 인자로 주어졌다고 볼 수 있다. mapply가 주어진 인자들을 하나씩 묶어 mean을 호출해주기에 각 행의 첫번째 열끼리 묶어 평균을 구하고, 다시 두번째 열끼리 묶어 평균을 구하는 작업을 반복하게 된다. 그리고 그 결과 모든 열의 평균을 얻는다.

## 6 doBy 패키지

doBy 패키지에는 summaryBy(), orderBy(), splitBy(), sampleBy()와 같이 특정 값에 따라 데이터를 처리하는 유용한 함수들이 있다. 이 절에서는 이들 함수 뿐만 아니라 이들 함수의 기본이 되는 base패키지<sup>3)</sup>내에 있는 함수들에 대해서도 살펴본다.

---

<sup>3)</sup>R의 기본 패키지이며 앞서 살펴본 다양한 함수들이 대부분 이 패키지에 속해있다. 이외에도 stats 등이 R에서 기본적으로 제공되는 패키지이다.

summaryBy()에 대해 살펴보기 전에 앞서 base::summary()<sup>4)</sup> 함수에 대해서 살펴보자. summary()는 자료에 대한 간략한 통계 분석 또는 머신 러닝 모델에 대한 요약을 보기 위해 사용되는 generic function이다. Generic function은 주어진 인자에 따라 다른 동작을 수행하는 함수로서, summary()의 경우에는 자료가 인자로 주어지면 간략한 통계요약을 내놓고, 모델이 인자로 주어지면 모델에 대한 요약을 보여주는 방식으로 동작한다.

다음 코드는 iris 데이터에 대한 통계 자료를 summary()로 살펴본 예이다.

```
> summary(iris)

  Sepal.Length      Sepal.Width       Petal.Length      Petal.Width
Min.     :4.300    Min.     :2.000    Min.     :1.000    Min.     :0.100
1st Qu.:5.100    1st Qu.:2.800    1st Qu.:1.600    1st Qu.:0.300
Median   :5.800    Median   :3.000    Median   :4.350    Median   :1.300
Mean     :5.843    Mean     :3.057    Mean     :3.758    Mean     :1.199
3rd Qu.:6.400    3rd Qu.:3.300    3rd Qu.:5.100    3rd Qu.:1.800
Max.     :7.900    Max.     :4.400    Max.     :6.900    Max.     :2.500

  Species
setosa     :50
versicolor:50
virginica :50
```

Sepal.Length 등과 같은 수치형 자료에 대해서는 최소값, 1사분위수, 중앙값, 평균, 3사분위수, 최대값을 보여준다. Species는 factor형 자료이므로 각 Species마다 몇개의 값이 있는지를 보여준다.

참고로 수치형 자료의 분포는 quantile()을 통해서도 알아볼 수 있다. 다음 코드는 0, 25, 50, 75, 100%에서의 값을 살펴보고, 또 0, 10, 20, ..., 100%의 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, by=0.1))
  0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
 4.30 4.80 5.00 5.27 5.60 5.80 6.10 6.30 6.52 6.90 7.90
```

doBy 패키지의 summaryBy()는 그 이름에서 짐작할 수 있듯이 원하는 컬럼의 값을 특정

<sup>4)</sup>'::' 는 다른 언어와 마찬가지로 네임스페이스를 표현하는데 사용된다. 이 경우에는 base 패키지내에 있는 summary()함수를 가리킨다.

조건에 따라 요약하는 목적으로 사용된다. 예를 들어, Sepal.Length와 Sepal.Width를 Species에 따라 살펴보려면 다음과 같이 하면 된다.

```
> 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.Length + Sepal.Length ~ Species’ 부분은 formula라고 하는데, 처리할 데이터를 일종의 수식으로 표현하는 방법이다. 이 예에서는 Sepal.Width와 Sepal.Length를 +로 연결해 이 두 가지에 대한 값을 차례로 컬럼으로 놓고, 각 행에는 Species를 배열하기 위해 ~ Species를 붙여주었다.

orderBy()는 데이터를 정렬하기 위한 목적으로 사용한다. 첫 번째 예는 모든 데이터를 Sepal.Width로 배열하는 예이다. 모든 데이터를 보여줄 것이므로 ~ 좌측에는 아무것도 적지 않는데, 그러면 모든 컬럼을 의미하게 된다.

```
> orderBy(~ Sepal.Width, iris)
      Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
61            5.0       3.7          1.5        0.2    versicolor
63            5.1       3.0          1.9        0.3    versicolor
69            5.1       3.4          1.3        0.2    versicolor
120           5.8       4.3          1.5        0.2  virginica
42             4.3       3.0          1.3        0.1    setosa
...
...
```

다음은 모든 데이터를 Species, Sepal.Width 순으로 정렬한 예이다. 먼저 Species 순으로 정렬되고 그 안에서 Sepal.Width로 정렬되었음을 알 수 있다.

```
> orderBy(~ Species + Sepal.Width, iris)
      Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
42            4.5       3.0          1.4        0.2    setosa
9             4.3       3.4          1.3        0.2    setosa
...
34            5.5       4.5          1.5        0.2    setosa
16            5.0       4.3          1.3        0.1    setosa
61            5.0       3.4          1.5        0.2  versicolor
63            5.8       4.3          1.6        0.2  versicolor
```

...				
57	6.3	3.3	4.7	1.6 versicolor
86	6.0	3.4	4.5	1.6 versicolor
120	6.0	2.2	5.0	1.5 virginica
107	4.9	2.5	4.5	1.7 virginica
...				
118	7.7	3.8	6.7	2.2 virginica
132	7.9	3.8	6.4	2.0 virginica

보통 R코드에서는 orderBy() 함수보다는 R에 기본적으로 포함된 base 패키지의 order()함수를 많이 사용한다. order()함수는 주어진 값을 정렬했을때의 색인을 순서대로 반환하는데, 이를 사용해 정렬된 결과를 얻을 수 있다.

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

위의 결과는 iris에서 Sepal.Width를 순서대로 정렬하면 61행, 63행, 69행, ... 순으로 된다는 것을 뜻한다. 따라서 다음과 같이 정렬된 결과를 얻을 수 있다.

```
> iris[order(iris$Sepal.Width),]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
61          5.0      2.0       3.5      1.0 versicolor
63          6.0      2.2       4.0      1.0 versicolor
...
```

orderBy()보다 먼저로워 보이는 것을 사실이나 기본 패키지 내의 함수인 이유로 더 많이 사용된다.

남아있는 두 함수 중 splitBy()는 생략하고 sampleBy()에 대해서 살펴보자. 그리고 그에앞서 R의 기본 패키지인 base에 있는 sample()에 대해 먼저 살펴보자.

sample()은 주어진 데이터에서 임의로 샘플을 추출하는 목적으로 사용되는데, 중복을 허용하거나 허용하지 않고 샘플을 추출할 수 있다. 기본은 중복을 허용하지 않는 것이며 중복을 허용하려면 replace=TRUE를 지정하면 된다. 다음은 1에서 10까지의 숫자중 5개를 샘플한 결과이다.

```
> sample(1:10, 5)
[1] 10 3 8 2 6
> sample(1:10, 5, replace=TRUE)
```

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

sample()의 한가지 흥미로운 사용방법은 데이터를 무작위로 섞는데 사용하는 것이다. 예를 들어 다음과 같이 1부터 10까지의 숫자에서 10개의 샘플을 뽑으면 이는 1부터 10까지의 숫자를 무작위로 섞는 것이다.

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

이를 사용하면 iris 데이터역시 무작위로 섞을 수 있다. 다음 코드에서 NROW()는 주어진 데이터 프레임 또는 벡터의 행의 수 또는 길이를 반환하는 함수이다.

```
> iris[sample(NROW(iris), NROW(iris)),]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
121          6.9        3.2       5.7        2.3  virginica
99           5.1        2.5       3.0        1.1 versicolor
...
...
```

sample()은 데이터의 무작위 재배열보다는 주어진 데이터에서 일부를 training data, 일부를 validation data(또는 test data)로 분리하는데 더 유용하게 쓰일 수 있다. 위에 보인 sample()의 사용방법은 데이터를 랜덤 샘플링 하는 것이었다. 그러나 보통은 주어진 데이터에서 각 Species마다 10% 씩을 뽑아서 test data로 활용하고 싶은 경우가 더 많을 것이다. sampleBy()는 이런 경우에 유용하게 사용할 수 있다.

```
> sampleBy(~ Species, frac=0.1, data=iris)
   Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa.4          4.6        3.1       1.5        0.2
setosa.11         5.4        3.7       1.5        0.2
setosa.15         5.8        4.0       1.2        0.2
setosa.37         5.5        3.5       1.3        0.2
setosa.50         5.0        3.3       1.4        0.2
versicolor.58     4.9        2.4       3.3        1.0
versicolor.66     6.7        3.1       4.4        1.4
versicolor.72     6.1        2.8       4.0        1.3
versicolor.75     6.4        2.9       4.3        1.3
versicolor.78     6.7        3.0       5.0        1.7
```

virginica.114	5.7	2.5	5.0	2.0
virginica.120	6.0	2.2	5.0	1.5
virginica.128	6.1	3.0	4.9	1.8
virginica.136	7.7	3.0	6.1	2.3
virginica.144	6.8	3.2	5.9	2.3
Species				
setosa.4	setosa			
setosa.11	setosa			
setosa.15	setosa			
setosa.37	setosa			
setosa.50	setosa			
versicolor.58	versicolor			
versicolor.66	versicolor			
versicolor.72	versicolor			
versicolor.75	versicolor			
versicolor.78	versicolor			
virginica.114	virginica			
virginica.120	virginica			
virginica.128	virginica			
virginica.136	virginica			
virginica.144	virginica			

위 코드의 실행결과에서 Species는 종이 폭이 좁아 별도의 문단으로 표시된 것이므로 오해 없기 바란다.

## 7 split()

split()은 데이터를 분리하는데 사용된다. 형식은 ‘split(데이터, 분리조건)’이다.

```
> 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()이 반환한 값은 리스트이다. 따라서 위 결과에 lapply()를 적용해서 iris의 종별 Sepal.Length의 평균을 구할 수 있다. 다음코드를 97페이지에서 살펴본 tapply()를 사용한 경우와 비교해서 보기 바란다.

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

$versicolor
[1] 5.936

$virginica
[1] 6.588
```

## 8 subset()

subset()은 split()과 유사하지만 전체를 부분으로 구분하는 대신 특정 부분만 취하는 용도로 사용한다. 다음은 iris에서 setosa종만 뽑아내는 예이다.

```
> subset(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
...
```

벡터 연산 (페이지 71)에서 살펴보았듯이 벡터간 연산에서의 AND는 `&&`가 아니라 `&`를 사용해야한다. 따라서 `subset()`에서도 2개 이상의 조건을 나열할때는 `&`를 사용한다. 다음은 `setosa`종에서 `Sepal.Length`가 5.0 이상인 행을 추출하는 예이다.

```
> subset(iris, Species == "setosa" & Sepal.Length > 5.0)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1        3.5       1.4        0.2    setosa
6          5.4        3.9       1.7        0.4    setosa
...
...
```

`subset`에 `select` 인자를 지정하면 특정 열을 선택하거나 제외할 수 있다. 다음은 `Sepal.Length` 와 `Species`열을 `iris`에서 선택하여 출력한 예이다.

```
> 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
...
...
```

특정 열을 제외하고자한다면 '-'를 열이름 앞에 붙이면 된다.

```
> 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
...
...
```

## 9 merge()

`merge()`는 두 데이터 프레임을 공통된 값을 기준으로 묶는 함수로서 데이터베이스에서 `join`과 같은 역할을 한다. 다음은 이름을 기준으로 수학 점수가 저장된 데이터 프레임과, 영어 점수가

저장된 데이터 프레임을 결합한 예이다. x 와 y의 이름이 서로 다른 순서로 저장되어있으나 이름을 기준으로 점수를 잘 찾아서 결합한 것을 볼 수 있다.

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

이는 [rbind\(\)](#), [cbind\(\)](#) (페이지 88)에서 설명한 cbind()과는 다르다. 위 코드는 두개의 데이터 프레임을 합칠 때 공통되는 컬럼 name을 기준으로 데이터를 합치지만 cbind()은 다음에 보인바와 같이 단순히 열을 합칠 뿐이다.

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

만약 공통된 이름이 없는 경우에는 한쪽에 데이터가 없게되는데 이 경우 값을 NA로 채우면서 전체 데이터를 모두 합치려면 다음과 같이 all 인자에 TRUE를 지정한다<sup>5)</sup>.

```
> merge(x, y, all=TRUE)
  name math english
1     a     1       4
2     b     2       5
3     c     3      NA
4     d    NA       6
```

---

<sup>5)</sup>RDBMS에서 사용하는 용어로는 Full Outer Join에 해당한다.

## 10 sort(), order()

sort(), order()는 데이터의 정렬을 위한 함수들이다. 먼저 sort()에 대해서 알아보자. sort()는 다음과 같이 벡터를 정렬하는 목적으로 사용한다.

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

위 코드에서 알 수 있듯이, sort()는 값을 정렬한 그 결과를 반환할 뿐이지 인자로 받은 벡터 자체를 변경하지 않는다.

order()는 주어진 인자를 정렬하기 위한 각 요소의 색인을 반환한다.

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

order(x)는  $x[\text{order}(x)]$  가 정렬되어있게 하기 위한 색인을 반환한다. 위 예에서 주어진 x를 정렬하면 그 결과가 11, 20, 33, 47, 50이 되어야 할 것이다. 정렬한 값 순서대로 x에서 값을 찾으면 제일 처음에 11이 와야하므로 x에서 2번째 숫자를 취해야한다. 다음, 20을 가져오려면 x에서 첫번째 수를 취해야하고, 33을 가져오려면 x에서 세번째 수를 취해야한다. 따라서 order(x)는 2, 1, 3, ... 이다.

다시말해 order(x)가 2, 1, 3, 5, 4라는 뜻은 x를 순서대로 정렬했을 때 x값의 순서는 x[2], x[1], x[3], x[5], x[4]가 된다는 의미이다.

만약 큰수부터 정렬한 결과를 얻고싶다면 값에 -1 을 곱하면 된다.

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

이를 활용해 order()를 데이터 프레임을 정렬하는데 사용할 수 있다. 다음은 iris를 Sepal.Length에 따라 정렬한 예이다.

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

위 결과를 보면 9행, 39행, 43행의 Sepal.Length가 모두 같았다. 이와 같이 Sepal.Length가 같은 경우 Petal.Length의 순서에 따라 정렬하려면 다음과 같이 Petal.Length를 order()에 추가적인 인자로 넘기면 된다.

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

보다시피 9행, 39행, 43행의 순서가 Petal.Length의 크기에 따라 39행, 43행, 9행으로 바뀌었다.

## 11 with(), within()

with()는 데이터 프레임 또는 리스트내 필드를 손쉽게 접근하기 위한 함수이다. ‘with(data, expression)’ 형태로 작성한다.

예를들어 iris의 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
```

위 두 명령은 번거롭게 ‘iris\$변수명’의 형태로 코드를 적어야 했다. with를 사용하면 iris내에서 각 필드를 곧바로 접근할 수 있다.

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

within()은 with()와 비슷하지만 데이터를 수정하는데 사용한다. 다음은 벡터에서 몇데이터가 결측치일 때 이를 중앙값으로 치환하는 예이다.

```
> x <- data.frame(val=c(1, 2, 3, 4, NA, 5, NA))  
> x  
  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=TRUE), val)  
+ })  
> x  
  val  
1   1  
2   2  
3   3  
4   4  
5   3  
6   5  
7   3
```

위 코드에서 median함수 호출시에 na.rm=TRUE를 지정했다. 이는 NA값이 포함된채로 median()을 부르면 결과로 NA가 나오기 때문이다. 참고로 위에서 보인 within() 호출 코드는 다음과 같이 간략하게 작성할 수도 있다.

```
> x$val[is.na(x$val)] <- median(x$val, na.rm=TRUE)
```

이번에는 조금 더 복잡하지만 실제로 만나볼만한 상황을 다뤄보자. iris내 일부 데이터가 결측치인 경우일때, 결측치를 해당 종(Species)의 중앙값으로 바꾸는 경우이다.

```
> data(iris)
> iris[1, 1] = NA
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          NA        3.5       1.4        0.2   setosa
2         4.9        3.0       1.4        0.2   setosa
...
> median_per_species <- sapply(split(iris$Sepal.Length, iris$Species),
  median, na.rm=TRUE)
> 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 Species
1          5.0        3.5       1.4        0.2   setosa
2          4.9        3.0       1.4        0.2   setosa
...
```

sapply부분이 조금 복잡하므로 풀어서 살펴보자. split은 다음과 같이 데이터를 Species별로 나눠준다.

```
> split(iris$Sepal.Length, iris$Species)
$setosa
[1]  NA 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5
 .4 5.1 5.7 5.1 5.4 5.1 4.6 5.1
...
$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 6.1 5.6 6.7 5
 .6 5.8 6.2 5.6 5.9 6.1 6.3 6.1
...
```

```
$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 5.7 5.8 6.4 6
.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3
...
```

여기서 종별 중앙값을 구하기 위해 sapply를 사용해 종별로 median을 부르되 median호출시 na.rm=TRUE를 넘겨준다.

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

앞서의 within()문은 이 벡터를 참조한 것이다.

## 12 attach(), detach()

with()와 비슷하게 사용할 수 있는 함수가 attach()이다. attach는 인자로 주어진 데이터 프레임이나 리스트를 곧바로 접근할 수 있게 해준다. 이를 해제하려면 detach()를 사용한다.

```
> 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
> detach(iris)
> Sepal.Width
Error: object 'Sepal.Width' not found
```

주의할점은 attach()한 변수값은 detach()시 원래의 데이터 프레임에는 반영되지 않는다는 점이다.

```
> data(iris)
> 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
...
> attach(iris)
```

```

> Sepal.Width[1] = -1
> Sepal.Width
[1] -1.0  3.0  3.2  3.1  3.6  3.9  3.4  3.4  2.9  3.1  3.7  3.4  3.0
      3.0  4.0  4.4  3.9  3.5  3.8
...
> detach(iris)
> 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
...

```

보다시피 첫째행의 Sepal.Width값은 attach()시 변경한 -1이 아닌 본래의 값 3.5이다.

## 13 which(), which.max(), which.min()

which()는 벡터 또는 배열에서 주어진 조건을 만족하는 값이 있는 곳의 색인을 찾는다. c(2, 4, 6, 7, 10)의 벡터에서 2로 나눈 나머지가 0인 값이 저장된 색인과 그 값을 찾아보자.

```

> x <- c(2, 4, 6, 7, 10)
> x %% 2
[1] 0 0 0 1 0
> which(x %% 2 == 0)
[1] 1 2 3 5
> x[which(x %% 2 == 0)]
[1] 2 4 6 10

```

which.min()과 which.max()는 주어진 벡터에서 최소 또는 최대 값이 저장된 색인을 찾는 함수이다. 다음 예를 살펴보자.

```

> x <- c(2, 4, 6, 7, 10)
> which.min(x)
[1] 1
> x[which.min(x)]
[1] 2
> which.max(x)
[1] 5
> x[which.max(x)]

```

[1] 10
--------

이 두 함수는 다양한 파라미터에 따라 모델을 만들고 모델을 선택하는데 사용할 수 있다. 예를 들어 머신 러닝 모델을 만들고, 모델의 우도(likelihood)를 벡터에 저장해두었다면 which.max() 함수를 사용해 최선의 모델을 찾을 수 있다.

which.min(), which.max()는 [sort\(\)](#), [order\(\)](#) (페이지 108)에서 살펴 본 sort()로 구현할 수도 있다. 다음은 최소, 최대 값을 구하는 예이다.

```
> x <- c(2, 4, 6, 7, 10)
> sort(x)[1] # which.min()
[1] 2
> -sort(-x)[1] # which.max()
[1] 10
```

## 14 aggregate()

[doBy 패키지](#) (페이지 99)가 데이터를 그룹별로 나눈뒤 특정 계산을 적용하는 함수인 반면 aggregate()는 보다 일반적인 그룹별 연산을 위한 함수이다. 기본적인 형태는 aggregate(데이터, 그룹 조건, 함수) 또는 aggregate(formula, 데이터, 함수)이다. 이중 여기서는 더 편리한 formula 형태의 방법을 살펴보자.

다음은 iris 데이터에서 종별 Sepal.Width의 평균 길이를 구하는 예이다.

```
> aggregate(Sepal.Width ~ Species, iris, mean)
   Species Sepal.Width
1      setosa     3.428
2 versicolor    2.770
3 virginica     2.974
```

## 15 stack(), unstack()

예를 들어 약물 A, B, C를 대조군(control), 실험군(experiment)에 대해서 시행하여 그 효과를 측정했다고 하자. 그러면 보통 이 데이터는 표 5.1처럼 정리하게 된다.

Medicine	Control	Experiment
A	5	4
B	3	5
C	2	7

표 5.1: 대조군, 실험군에서의 약물 반응 결과

그런데 위와 같은 형태는 그래프를 그린다면 데이터를 조작한다던가 하는 측면에서 불편한 면이 있다. 예를 들어 앞서 [doBy 패키지](#) (페이지 99)에서 살펴봤던 `summaryBy()`의 경우를 생각해보자. 만약 약물 반응을 실험군, 대조군 별로 요약하고 싶다면 `summaryBy(value ~ category, data)`와 같은 형태로 명령을 줄 수 있어야하는데 위의 데이터는 그러한 명령에 적합한 형태가 아니다.

이 경우에는 다음과 같이 `stack()` 명령으로 데이터를 변환하고 `summaryBy()`를 적용할 수 있다.

```
> x <- data.frame(medicine=c("a", "b", "c"),
+                   ctl=c(5, 3, 2),
+                   exp=c(4, 5, 7))
> x
  medicine ctl  exp
1         a    5    4
2         b    3    5
3         c    2    7
> stacked_x <- stack(x)
Warning message:
In stack.data.frame(x) : non-vector columns will be ignored
> stacked_x
  values ind
1      5  ctl
2      3  ctl
3      2  ctl
4      4  exp
5      5  exp
6      7  exp
> library(doBy)
> summaryBy(values ~ ind, stacked_x)
  ind values.mean
```

```
1 ctl      3.333333
2 exp      5.333333
```

위 코드에서 중간에 나온 경고 메시지는 factor 컬럼은 stack이 되는 대상이 아니라는 의미이다. 이 예에서는 벡터에 해당하는 ctl, exp 컬럼만 변환하는 것이므로 무시했다.<sup>6)</sup>

unstack()은 그 이름에서 짐작할 수 있듯이 stack()을 통해 변환된 데이터를 다시 원래 상태로 되돌리는데 사용한다.

```
> unstack(stacked_x, values ~ ind)
ctl  exp
1    5    4
2    3    5
3    2    7
```

unstack()의 두번째 인자는 formula로서, values가 데이터 프레임에 저장될 값이며, 이 값을 ind에 있는 값(ctl, exp)을 컬럼으로 해서 저장해달라는 의미를 표현하는 일종의 모델 표현식이다.

## 16 RMySQL 패키지

RMySQL 패키지는 MySQL을 R에서 접근하는데 사용된다. 당연하지만 MySQL은 이미 설치가 되어 있어야하고, 다음과 같이 콘솔에서 MySQL을 로그인하는데 문제가 없어야 한다.

```
$ mysql -umkseo -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.1.66 Source distribution

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
```

<sup>6)</sup> 즉, 이 예에서는 medicine 컬럼이 사라졌다. 만약 medicine 컬럼도 보존하는 변환을 원한다면 [reshape 패키지](#)의 melt() 함수를 사용하면 된다.

```
Type `help;' or `\'h' for help. Type `\'c' to clear the current input
statement.

mysql> use mkseo;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
```

다음과 같이 이름, 점수가 기록될 score 테이블을 만들고 이 테이블을 사용한 다양한 작업을 살펴보자.

```
mysql> create table score(name varchar(20), score integer);
Query OK, 0 rows affected (0.02 sec)

mysql> show tables;
+-----+
| Tables_in_mkseo |
+-----+
| score           |
+-----+
1 row in set (0.00 sec)

mysql> insert into score values("a", 1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into score values("b", 3);
Query OK, 1 row affected (0.00 sec)
```

R 콘솔을 열고 RMySQL 패키지를 설치한다.

```
> install.packages("RMySQL")
> library(RMySQL)
```

컨넥션을 맺고 테이블 목록을 뽑아보자.

```
> con <- dbConnect(MySQL(), user="mkseo", password="1234", dbname="mkseo", host="127.0.0.1")
> dbListTables(con)
[1] "score"
```

dbGetQuery() 함수를 사용해 질의를 실행하고 그 결과를 데이터 프레임을 받는다.

```
> dbGetQuery(con, "select * from score");
   name score
1     a     1
2     b     3
```

처리할 데이터를 MySQL에 넣고, 복잡한 전처리를 미리 sql 스크립트 파일을 작성해 mysql 클라이언트에서 수행해 놓는다면, 이 정도의 명령만으로 mysql을 R에서 사용하는데 큰 문제가 없을 것이다. 만약 더 많은 함수(예를 들어 질의후 결과를 한줄씩 처리해야한다던가 등)가 필요하다면 ?RMySQL로 도움말을 살펴보거나 [RMySQL Reference](#)를 참고하기 바란다.

## 데이터 조작 II

머신 러닝 또는 데이터 마이닝 프로젝트에서 모델을 만드는데 소요되는 시간은 전체 프로젝트의 14% 정도에 해당하며, 오히려 그 외의 작업들(데이터 수집, 전처리 및 후처리)에 더 긴 시간이 소요된다. 심지어 데이터 전처리에 소요되는 시간의 중앙값은 다른 모든 단계들에서의 소요시간 중앙값 보다 더 크다<sup>1)</sup>. 이처럼 데이터 전처리는 중요하고 실제로 많은 시간이 걸리는 작업이다.

전처리단계에서 뿐만 아니라 머신 러닝 알고리즘을 R 코드로 작성하다보면 데이터를 원하는 형태로 변환해야하는 필요가 수없이 발생한다. 따라서 데이터를 원하는 형태로 바꾸거나 조작하는 능력은 R을 사용하는데 있어서 절대적으로 중요하다.

이러한 필요를 해결하기위하여 이 장에서는 복잡한 데이터 처리를 해결하는데 필요한 R의 중요 패키지 sqldf, plyr, reshape2, data.table, foreach를 설명한다. 또한 R 코드를 더 빠르게 실행하기 위한 멀티코어 병렬처리 패키지 doMC를 살펴본다. 마지막으로 다양한 데이터 조작 함수를 작성한 뒤에는 그 코드가 올바르게 작성되었는지, 그리고 빠르게 수행되는지 확인하는 것이 필요하다. 따라서 코드 테스트를 위해 사용하는 testthat 패키지와 browser(), 코드 성능 평가를 위한 system.time()과 Rprof()를 살펴본다.

이 장의 내용을 보면서 꼭 이런 패키지들을 다 알아야만 할까 의문이 드는 독자도 있을 것으로 짐작한다. 이에 대한 답은 ‘필요하다’라는 것이다. 앞서 데이터 조작 I 장에서 살펴본 기본 R 함수들도 충분히 훌륭하지만 실제 데이터를 다루는데 있어서는 이 장에서 살펴볼 내용에 비해 편의성이 많이 부족하기 때문이다.

### 1 sqldf 패키지

R에는 많은 데이터 처리 함수들이 있어 데이터를 편리하게 조작할 수 있다는 장점이 있다. 그러나 한편으로는 원하는 형태로 데이터를 만들기 위해서 여러가지 함수를 알아야 하는점이

<sup>1)</sup>M. Arthur Munson, A Study on the Importance of and Time Spent on Different Modeling Steps, ACM SIGKDD Explorations Newsletter, 2011.

부담이 되기도 한다. sqldf[9]는 이런 부담을 털어버리는데 큰 도움이 되는 패키지로 SQL문을 사용할 줄 아는 사용자들에게 더욱 쉽게 데이터를 접근할 수 있게 해준다.

sqldf는 SQL명령이 주어지면 자동으로 스키마를 생성하고 데이터를 테이블로 로딩한 뒤 SQL문을 수행한다. SQL의 실행 결과는 다시 R로 로딩된다. 이러한 작업은 자동으로 이루어지기 때문에 사용자가 힘들여 데이터베이스를 설치하고 환경을 설정하는 작업이 필요없다. 또한 성능 최적화가 최대한 이루어진 데이터 베이스기술을 활용하게 되어 데이터 처리 성능또한 상당히 우수하다.

sqldf를 사용하기 위해서 다음과 같이 패키지를 설치한다.

```
> install.packages("sqldf")
> library(sqldf)
```

sqldf()를 사용해 iris 데이터를 살펴보자. iris에는 세종류의 붓꽃 종류가 저장되어 있으며 붓꽃 종은 Species 열에 저장되어 있다. 이를 확인해보자.

```
> sqldf("select distinct Species from iris")
Loading required package: tcltk
      Species
1      setosa
2  versicolor
3  virginica
```

이번에는 setosa에 속하는 데이터에서 Sepal.Length의 평균을 구해보자.

```
> sqldf("select avg(Sepal_Length) from iris where Species='setosa'")
      avg(sepal_length)
1           5.006
```

R과 달리 SQL에서 ‘.’은 컬럼명이 될 수 없으므로 Sepal.Length가 아니라 Sepal.Length로 컬럼명을 적어야 한다. 또한 SQL에서 대소문자 구별은 없으므로 Sepal.Length 대신 sepal.length로 컬럼명을 적을 수 있다.

만약 위와 같은 작업을 split(), apply()등으로 해결하려 한다면 다음과 같이 좀 더 번거로운 명령들을 수행해야 한다.

```
> mean(subset(iris, Species == "setosa")$Sepal.Length)
[1] 5.006
```

각 종별 Sepal.Length의 평균을 구하려면 다음과 같이 한다.

```
> sqldf(
+   "select species, avg(sepal_length) from iris group by species")
  Species avg(sepal_length)
1      setosa          5.006
2 versicolor          5.936
3 virginica           6.588
```

비교를 위해 `split()`, `sapply()`를 사용해 같은 명령을 실행하려면 다음과 같은 코드를 사용해야 한다. R에 아직 익숙하지 않고, SQL에 익숙한 사용자에게는 `sqldf()`가 더 편한 인터페이스를 제공한다.

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

`sqldf` 패키지에서는 디스크를 저장소로 사용한다던가, 매번 `sqldf()`를 수행할 때마다 데이터베이스에 데이터 프레임을 저장했다가 처리하고 삭제하는 대신 한번 만들어둔 데이터를 재사용한다던가 하는 많은 최적화가 가능하다.

`sqldf`는 다양한 저장소를 사용할 수 있으며 기본 데이터 저장소로는 `sqlite`를 사용한다. 따라서 속도 향상을 위한 인덱스 등을 적절히 사용할 수도 있으며, 데이터베이스 기반 기술을 활용하므로 프로그램의 속도를 매우 빠르게 높힐 수 있다. `sqldf`의 성능을 보여주는 사례로 [R] conditionally merging adjacent rows in a data frame를 참고하기 바란다.

보다 자세한 내용은 [sqldf 프로젝트 홈페이지](#)에도 잘 설명이 되어있다.

## 2 plyr 패키지

`plyr`[10]은 데이터를 분할하고(`split`), 분할된 데이터에 특정 함수를 적용한 뒤(`apply`), 그 결과를 재 조합(`combine`)하는 세 단계로 데이터를 처리하는 함수들을 제공한다. `plyr`의 입력은 배열, 데이터 프레임, 리스트가 될 수 있다. 출력 역시 배열, 데이터 프레임, 리스트가 될 수 있으며 또는 아무런 결과도 출력하지 않을 수 있다.

`plyr`은 이처럼 데이터의 분할, 계산, 조합을 한번에 처리해 주어 여러 함수로 처리해야 할 일들을 짧은 코드로 대신해준다. 뿐만 아니라 입력과 출력에서 다양한 데이터 타입을 지원해주어 데이터 변환의 부담을 크게 덜어준다.

`plyr`의 데이터 처리 함수들은 `??ply` 형태의 5글자 함수명으로 이름지어져 있다. 첫번째 글자는 입력 데이터 타입에 따라 각각 배열(`a`), 데이터 프레임(`d`), 리스트(`l`)로 정해진다. 두번째

글자는 출력 데이터 타입으로서 마찬가지로 a, d, l 또는 -로 정해지는데 이 중 -은 아무런 출력도 내보내지 않음을 뜻한다.

이들 중 adply()와 같이 배열을 입력으로 받는 함수들은 각 행 또는 각 열마다 데이터를 처리하기 위해 margin을 인자로 받으며, ddply()와 같이 데이터 프레임을 입력으로 받는 함수들은 데이터를 분할하는데 사용할 그룹 변수를 인자로 받는다.

이 절에서는 plyr의 함수들 중 몇몇 특징적인 경우에 대해서 살펴보기로 한다.

## 2.1 adply()

adply()는 배열(a)를 받아 데이터 프레임(d)를 반환하는 함수이므로 그 이름이 adply로 명명되었다. 그러나 입력이 반드시 배열일 필요는 없다. 그보다는 주어진 입력을 숫자 색인으로 읽을 수 있는가하는 점이 중요하다. 이런 이유로 데이터 프레임도 숫자 색인으로 각 행이나 열을 접근 할 수 있어 adply()를 적용할 수 있다.

adply()는 인자로 데이터, margin, 함수를 입력으로 받는데, margin=1은 행방향, margin=2는 열방향으로 데이터를 처리함을 뜻한다.

물론 굳이 adply()를 쓰지 않고 apply()를 사용해도 데이터 프레임을 행 또는 열방향으로 처리할 수 있다. 그러나 apply()를 행방향으로 처리할 때 각 열에 서로 다른 데이터 타입이 섞여있다면 예상치 못한 타입 변환이 발생할 수 있다. 예를 들어 다음 코드에서 볼 수 있듯이 apply()에 숫자형 컬럼만 입력으로 주었을 경우에는 그 값이 제대로 숫자로 넘어오지만, 문자열이 섞이면 데이터가 모두 문자열로 바뀐다.

```
> apply(iris[, 1:4], 1, function(row) { print(row) })
Sepal.Length Sepal.Width Petal.Length Petal.Width
      5.1          3.5         1.4         0.2
Sepal.Length Sepal.Width Petal.Length Petal.Width
      4.9          3.0         1.4         0.2
...
> apply(iris, 1, function(row) { print(row) })
Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
      "5.1"        "3.5"       "1.4"       "0.2"      "setosa"
Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
      "4.9"        "3.0"       "1.4"       "0.2"      "setosa"
...
```

이러한 변환이 발생한 이유는 apply()가 한가지 타입만 저장할 수 있는 ‘행렬’로 결과를 반환했기 때문이다. 본래 apply()는 벡터, 행렬, 리스트 중 한가지 타입으로 결과를 반환할 수 있다. 그리고 결과가 한 행이라면 벡터를 반환하고, 여러행이라면 행렬을 반환하며, 각 행마다 컬럼

갯수가 다르다면 리스트를 반환한다. 위의 예에서는 각 행의 컬럼 갯수가 5로 모두 일치하므로 리스트를 반환하지 않고 행렬로 결과가 반환된 것이다.

반면 adply()를 사용해 결과를 데이터 프레임을 반환시키면 결과 타입이 문자열로 모두 바뀌는 현상을 피할 수 있다. 다음은 adply()를 사용해 데이터 프레임의 각 행을 보면서 Sepal.Length 가 5.0이상이고 Species가 setosa인지 여부를 확인한다음 그 결과를 새로운 컬럼 V1에 기록하는 예이다.

```
> install.packages("plyr")
> library(plyr)
> adply(iris,
+       1,
+       function(row) { row$Sepal.Length >= 5.0 &
+                      row$Species == "setosa"})
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species      V1
1          5.1        3.5         1.4        0.2    setosa    TRUE
2          4.9        3.0         1.4        0.2    setosa   FALSE
3          4.7        3.2         1.3        0.2    setosa   FALSE
4          4.6        3.1         1.5        0.2    setosa   FALSE
5          5.0        3.6         1.4        0.2    setosa    TRUE
...
...
```

위 예에서는 adply()에 인자로 넘긴 함수의 반환값이 단순한 boolean 값이었으므로 그 결과가 임의의 컬럼명 V1에 저장되었다. 그러나 최종 반환 값이 데이터 프레임인 경우 함수의 반환값을 데이터 프레임으로 하는 것이 안전하다. 이 경우 함수가 반환하는 데이터 프레임에 변수명을 적절히 지정할 수도 있다. 다음은 앞서와 같은 계산을 수행하지만 함수 내부에서 데이터 프레임을 반환하는 예이다.

```
> adply(iris,
+       1,
+       function(row) {
+         data.frame(
+           sepal_ge_5_setosa=c(row$Sepal.Length >= 5.0 &
+                               row$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
```

```

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
...
      sepal_ge_5_setosa
1      TRUE
2      FALSE
3      FALSE
4      FALSE
5      TRUE
...

```

## 2.2 ddply()

ddply()는 데이터 프레임(d)을 입력으로 받아 데이터 프레임(d)을 내보내는 함수이므로 이름이 ddply()로 명명되었다. ddply()는 인자로 데이터, 데이터를 그룹 지을 변수명, 데이터 처리에 사용할 함수를 받는다.

다음은 iris 데이터에서 Sepal.Length의 평균을 Species 별로 계산하는 예이다. 두번째 인자인 데이터를 그룹짓는 변수는 .( ) 안에 기록한다.

```

> ddply(iris,
+       .(Species),
+       function(sub) {
+         data.frame(sepal.width.mean=mean(sub$Sepal.Width))
+       })
      Species  sepal.width.mean
1      setosa          3.428
2  versicolor         2.770
3  virginica          2.974

```

여러 변수들로 그룹을 짓고자 한다면 .( ) 안에 조건들 또는 필드명들을 나열하면 된다.

```

> ddply(iris,
+       .(Species, Sepal.Length > 5.0),
+       function(sub) {
+         data.frame(sepal.width.mean=mean(sub$Sepal.Width))
+       })

```

	Species	Sepal.Length > 5	sepal.width.mean
1	setosa	FALSE	3.203571
2	setosa	TRUE	3.713636
3	versicolor	FALSE	2.233333
4	versicolor	TRUE	2.804255
5	virginica	FALSE	2.500000
6	virginica	TRUE	2.983673

plyr의 참고문헌 The Split-Apply-Combine Strategy for Data Analysis[10]에 실린 baseball 예제를 살펴보자. baseball 데이터는 야구 선수들의 기록 정보를 모은 데이터이다. 다음에 데이터의 일부를 보였다.

	> head(baseball)														
		id	year	stint	team	lg	g	ab	r	h	X2b	X3b	hr		
4	ansonca01	1871		1	RC1		25	120	29	39	11	3	0		
44	forceda01	1871		1	WS3		32	162	45	45	9	4	0		
68	mathebo01	1871		1	FW1		19	89	15	24	3	1	0		
99	startjo01	1871		1	NY2		33	161	35	58	5	1	1		
102	suttoez01	1871		1	CL1		29	128	35	45	3	7	3		
106	whitede01	1871		1	CL1		29	146	40	47	6	5	1		
						rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
4						16	6	2	2	1	NA	NA	NA	NA	NA
44						29	8	0	4	0	NA	NA	NA	NA	NA
68						10	2	1	2	0	NA	NA	NA	NA	NA
99						34	4	2	3	0	NA	NA	NA	NA	NA
102						23	3	1	1	0	NA	NA	NA	NA	NA
106						21	2	2	4	1	NA	NA	NA	NA	NA

데이터의 각 행에는 선수(id 컬럼)가 해당 년도(year 컬럼)에 기록한 성적이 들어있다. 다음은 선수 ansonca01의 기록을 살펴본 예이다.

	> head(subset(baseball, id=="ansonca01"))													
		id	year	stint	team	lg	g	ab	r	h	X2b	X3b	hr	
4	ansonca01	1871		1	RC1		25	120	29	39	11	3	0	
121	ansonca01	1872		1	PH1		46	217	60	90	10	7	0	
276	ansonca01	1873		1	PH1		52	254	53	101	9	2	0	
398	ansonca01	1874		1	PH1		55	259	51	87	8	3	0	

525	ansonca01	1875	1	PH1	69	326	84	106	15	3	0	
741	ansonca01	1876	1	CHN	NL	66	309	63	110	9	7	2
<hr/>												
	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp		
4	16	6	2	2	1	NA	NA	NA	NA	NA		
121	50	6	6	16	3	NA	NA	NA	NA	NA		
276	36	0	2	5	1	NA	NA	NA	NA	NA		
398	37	6	0	4	1	NA	NA	NA	NA	NA		
525	58	11	6	4	2	NA	NA	NA	NA	NA		
741	59	NA	NA	12	8	NA	NA	NA	NA	NA		

ddply()를 사용해 각 선수가 출전한 게임수의 평균을 구해보자.

```
> ddply(baseball, .(id), function(sub) { mean(sub$g) })
      id          V1
1 aaronha01 143.39130
2 abernte02  40.05882
3 adairje01  77.66667
4 adamsba01  25.36842
5 adamsbo03  85.40000
6 adcocjo01 115.23529
7 agostju01  36.20000
8 aguilri01  38.42105
9 aguirha01  27.93750
10 ainsmed01 63.41176
...
```

## 2.3 transform(), summarise(), subset()

이 절에서 살펴본 예에서는 adply() 또는 ddply()에 임의의 사용자 정의 함수를 넘겨주어 분석을 수행했다. 그러나 공통적으로 자주 사용되는 유형의 계산은 transform(), summarise(), subset()를 사용해 보다 간단히 표현 할 수 있다.

## transform()

base:::transform()<sup>2)</sup>은 변수값에 대한 연산결과를 데이터 프레임의 다른 변수에 저장하는 함수이다.

이를 사용해 baseball 데이터에 각 행이 선수의 몇년차 통계인지를 뜻하는 cyear 컬럼을 추가해보자. 다음 코드는 데이터를 선수 id로 분할 한 뒤, 선수별 분할에서 year의 최소값과 현재 행의 year 값의 차이를 cyear에 저장한다.

```
> head(ddply(baseball, .(id), transform, cyear=year - min(year) + 1))
      id year stint team lg   g  ab    r    h X2b X3b hr
1 aaronha01 1954      1  ML1 NL 122 468 58 131 27   6 13
2 aaronha01 1955      1  ML1 NL 153 602 105 189 37   9 27
3 aaronha01 1956      1  ML1 NL 153 609 106 200 34  14 26
4 aaronha01 1957      1  ML1 NL 151 615 118 198 27   6 44
5 aaronha01 1958      1  ML1 NL 153 601 109 196 34   4 30
6 aaronha01 1959      1  ML1 NL 154 629 116 223 46   7 39

      rbi  sb  cs  bb  so  ibb  hbp  sh  sf  gidp cyear
1   69   2   2  28  39    NA   3   6   4   13     1
2  106   3   1  49  61     5   3   7   4   20     2
3   92   2   4  37  54     6   2   5   7   21     3
4  132   1   1  57  58    15   0   0   3   13     4
5   95   4   1  59  49    16   1   0   3   21     5
6  123   8   0  51  54    17   4   0   9   19     6
```

plyr에는 transform()을 개선한 plyr:::mutate() 함수가 있다. 이 함수는 여러 컬럼을 데이터 프레임에 추가할 때 바로 앞서 추가한 컬럼을 뒤에 추가하는 컬럼에서 참조할 수 있어 편리하다. 예를 들어 아래 코드에서는 mutate를 이용해 cyear를 계산한 뒤 cyear를 참조하는 log\_cyear를 계산했다. 만약 mutate가 아닌 transform을 사용하면 이 경우 에러가 발생한다.

```
> head(ddply(baseball, .(id), mutate,
+             cyear=year - min(year) + 1, log_cyear=log(cyear)))
      id year stint team lg   g  ab    r    h X2b X3b hr
1 aaronha01 1954      1  ML1 NL 122 468 58 131 27   6 13
2 aaronha01 1955      1  ML1 NL 153 602 105 189 37   9 27
3 aaronha01 1956      1  ML1 NL 153 609 106 200 34  14 26
```

<sup>2)</sup>base:::transform 형태의 표현은 base 패키지에 있는 transform() 함수를 의미함

4	aaronha01	1957	1	ML1	NL	151	615	118	198	27	6	44
5	aaronha01	1958	1	ML1	NL	153	601	109	196	34	4	30
6	aaronha01	1959	1	ML1	NL	154	629	116	223	46	7	39
<hr/>												
1	69	2	2	28	39	NA	3	6	4	13	1	0.0000000
2	106	3	1	49	61	5	3	7	4	20	2	0.6931472
3	92	2	4	37	54	6	2	5	7	21	3	1.0986123
4	132	1	1	57	58	15	0	0	3	13	4	1.3862944
5	95	4	1	59	49	16	1	0	3	21	5	1.6094379
6	123	8	0	51	54	17	4	0	9	19	6	1.7917595

### summarise()

plyr::summarise()는 데이터의 요약 정보를 만드는데 사용하는 함수이다. transform()의 인자로 주어진 계산 결과를 새로운 컬럼에 추가한 데이터 프레임을 반환하는 반면 summarise()는 계산 결과를 담은 새로운 데이터 프레임을 반환한다.

baseball 데이터에서 각 선수의 최초 데이터가 몇년도에 해당하는지 살펴보는 다음 예를 보자. 아래 코드에서는 각 id마다 최소 year를 minyear로 갖는 데이터 프레임들이 summarise()에 의해 생성되고 ddply는 이를 데이터 프레임을 모아 하나의 데이터 프레임으로 반환한다.

```
> head(ddply(baseball, .(id), summarise, minyear=min(year)))
      id minyear
1 aaronha01    1954
2 abernte02    1955
3 adairje01    1958
4 adamsba01    1906
5 adamsbo03    1946
6 adcocjo01    1950
```

만약 여러 계산값들을 구하고 싶다면 인자를 계속 나열하면 된다. 다음은 minyear, maxyear를 구하는 예이다.

```
> head(ddply(baseball, .(id), summarise,
+             minyear=min(year), maxyear=max(year)))
      id minyear maxyear
1 aaronha01    1954    1976
2 abernte02    1955    1972
```

3	adairje01	1958	1970
4	adamsba01	1906	1926
5	adamsbo03	1946	1959
6	adcocjo01	1950	1966

### subset()

그 이름에서 쉽게 알 수 있듯이 subset()은 각 분할별로 데이터를 추출하는데 사용한다.

다음은 각 선수별로 최대 게임을 플레이한 해의 기록을 추출한다.

```
> head(ddply(baseball, .(id), subset, g==max(g)))
      id year stint team lg   g  ab   r    h  X2b  X3b hr
1 aaronha01 1963     1  ML1 NL 161 631 121 201   29    4 44
2 abernte02 1965     1  CHN NL  84 18   1    3   0    0  0
3 adairje01 1965     1  BAL AL 157 582 51 151   26    3  7
4 adamsba01 1913     1  PIT NL  43 114 13 33   6    2  0
5 adamsbo03 1952     1  CIN NL 154 637 85 180   25    4  6
6 adcocjo01 1953     1  ML1 NL 157 590 71 168   33    6 18
      rbi sb cs bb so ibb hbp sh sf gidp
1 130 31  5 78 94 18   0   0  5   11
2   2  0  0  0  7  0   1   3   0   0
3  66  6  4 35 65  7   2   4   2   26
4  13  0 NA  1 16 NA   0   3 NA   NA
5  48 11  9 49 67 NA   0   8 NA   15
6  80  3  2 42 82 NA   2   6 NA   22
```

코드에서 g==max(g)는 subset에서 조건을 지정하는데 사용되는 인자이므로 두개의 등호가 사용됨에 유의하기 바란다.

## 2.4 m\*ply()

m\*ply(), 즉 maply(), mdply(), mlply(), m\_ply() 함수는 데이터 프레임 또는 배열을 인자로 받아 각 컬럼을 주어진 함수에 적용한 뒤 그 실행 결과들을 조합한다.

예를 들어 다음과 같은 데이터 프레임을 가정해보자.

```
> x <- data.frame(mean=1:5, sd=1:5)
> x
  mean sd
```

1	1	1
2	2	2
3	3	3
4	4	4
5	5	5

mdply()를 사용하면 위 데이터 프레임의 각 행을 rnorm() 함수의 mean, sd에 대한 인자로 넘겨주어 실행한 뒤 그 결과를 데이터 프레임으로 모을 수 있다. 다음 예에서는 각 mean, sd 값에 대해 2개씩 난수를 발생시켰다.

```
> mdply(x, rnorm, n=2)
      mean   sd        V1        V2
1     1.00  1.00  0.8397891  0.5390922
2     2.00  2.00  4.9274472  6.5302294
3     3.00  3.00  5.2031884  1.9018554
4     4.00  4.00  7.5995477  5.0228909
5     5.00  5.00 14.2004666  5.1076554
```

### 3 reshape2 패키지

reshape2[11] (<http://had.co.nz/reshape/>)는 데이터의 모양을 바꾸는데 사용하는 함수이다. reshape2가 제공하는 변환은 크게 melt, cast이며 이 두가지 변환을 사용해 데이터의 모양을 바꾸거나, 데이터를 요약할 수 있다.

#### 3.1 melt()

melt() 함수는 인자로 데이터를 구분하는 식별자(id), 측정 대상 변수, 측정치를 받아 데이터를 간략하게 표현한다.

이 절에서 예제로 사용할 french\_fries 데이터를 살펴보자. french\_fries 데이터는 세가지 종류의 오일을 사용하여 프렌치 프라이를 만들었을 때 프렌치 프라이의 맛이 어떻게 달라지는가를 측정한 결과를 담고 있다. 다음은 데이터의 일부를 보인 예이다.

```
> library(reshape2)
> str(french_fries)
'data.frame': 696 obs. of 9 variables:
 $ time      : Factor w/ 10 levels "1","2","3","4",...: 1 1 1 1 1 1 ...
 $ treatment: Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
```

```
$ subject   : Factor w/ 12 levels "3","10","15",...: 1 1 2 2 3 3 4 ...
$ rep       : num  1 2 1 2 1 2 1 2 1 2 ...
$ potato    : num  2.9 14 11 9.9 1.2 8.8 9 8.2 7 13 ...
$ buttery   : num  0 0 6.4 5.9 0.1 3 2.6 4.4 3.2 0 ...
$ grassy    : num  0 0 0 2.9 0 3.6 0.4 0.3 0 3.1 ...
$ rancid   : num  0 1.1 0 2.2 1.1 1.5 0.1 1.4 4.9 4.3 ...
$ painty   : num  5.5 0 0 0 5.1 2.3 0.2 4 3.2 10.3 ...

> head(french_fries)

  time treatment subject rep potato buttery grassy rancid painty
61     1           1      3   1     2.9     0.0     0.0     0.0    5.5
25     1           1      3   2    14.0     0.0     0.0     1.1    0.0
62     1           1     10   1    11.0     6.4     0.0     0.0    0.0
26     1           1     10   2     9.9     5.9     2.9     2.2    0.0
63     1           1     15   1     1.2     0.1     0.0     1.1    5.1
27     1           1     15   2     8.8     3.0     3.6     1.5    2.3
```

french\_fries 데이터 프레임에서 time, treatment, subject, rep는 각각 실험을 수행한 시간, 오일의 종류, 대상 프렌치 프라이, 반복된 실험중에서 이 실험이 몇번째 반복인지를 담고 있다. potato, buttery, grassy, rancid, painty는 프렌치 프라이의 맛을 측정한 변수들이다. 이 데이터에서 각 데이터를 식별하는 식별자, 즉 id라고 볼 수 있는 부분은 time, treatment, subject, rep의 1:4 컬럼이고, 각 id마다 측정된 결과는 나머지 변수들이라고 볼 수 있다.

이런 사실을 활용해 1:4 컬럼을 id, 나머지 컬럼을 측정치로 놓고 melt() 변환을 수행하는 코드는 다음과 같다.

```
> m <- melt(id=1:4, french_fries)
> head(m)

  time treatment subject rep variable value
1     1           1      3   1     potato   2.9
2     1           1      3   2     potato  14.0
3     1           1     10   1     potato  11.0
4     1           1     10   2     potato   9.9
5     1           1     15   1     potato   1.2
6     1           1     15   2     potato   8.8
```

이 결과에서 볼 수 있듯이 측정 변수명은 ‘variable’, 측정치는 ‘value’에 저장되는 것이 melt()의 특징이다.

좀 더 간단한 사례로 smiths 데이터를 살펴보자. smiths 데이터는 id가 subject, time이며 나머지는 측정치이다. 측정치에는 NA로 기재된 결측치가 존재한다.

```
> smiths
  subject time age weight height
1 John Smith     1   33      90    1.87
2 Mary Smith     1    NA      NA    1.54

> melt(id=1:2, smiths)
  subject time variable value
1 John Smith     1       age 33.00
2 Mary Smith     1       age     NA
3 John Smith     1     weight 90.00
4 Mary Smith     1     weight     NA
5 John Smith     1     height  1.87
6 Mary Smith     1     height  1.54
>
```

smiths를 melt()로 변환한 결과에서 2행, 4행은 경우에 따라 불필요 할 경우도 있다. 이 경우 melt()에 na.rm을 지정해 NA가 저장된 행을 제외할 수 있다.

```
> melt(id=1:2, smiths, na.rm=TRUE)
  subject time variable value
1 John Smith     1       age 33.00
3 John Smith     1     weight 90.00
5 John Smith     1     height  1.87
6 Mary Smith     1     height  1.54
```

마찬가지로 french\_fries에도 NA가 기록된 행들이 있고 이를 melt()로 변환할 때 제외할 수 있다. 다음 코드에서 complete.cases() 함수는 해당 행의 모든 값이 NA가 아닌 경우 TRUE, 해당 행의 값이 하나라도 NA를 갖고 있는 경우 FALSE를 반환한다.

```
> french_fries[!complete.cases(french_fries), ]
  time treatment subject rep potato buttery grassy rancid painty
315     5          3     15    1     NA      NA      NA      NA
455     7          2     79    1     7.3     NA     0.0     0.7      0
515     8          1     79    1    10.5     NA     0.0     0.5      0
520     8          2     16    1     4.5     NA     1.4     6.7      0
```

563	8	2	79	2	5.7	0	1.4	2.3	NA
<pre>&gt; m &lt;- melt(id=1:4, french_fries, na.rm=TRUE)</pre>									
<pre>&gt; head(m)</pre>									
<pre>  time treatment subject rep variable value</pre>									
1	1	1	3	1	potato	2.9			
2	1	1	3	2	potato	14.0			
3	1	1	10	1	potato	11.0			
4	1	1	10	2	potato	9.9			
5	1	1	15	1	potato	1.2			
6	1	1	15	2	potato	8.8			

### 3.2 dcast()

cast()는 결과로 얻고자하는 데이터 타입에따라 dcast(), acast()로 구분하여 사용한다. dcast()는 결과로 데이터 프레임을 반환하며, acast()는 벡터, 행렬, 배열을 반환한다. 이 절에서는 dcast()만 살펴보도록 하겠다.

dcast()의 첫번째 인자는 melt()를 사용해 변환된 데이터(molten data)이며 두번째 인자는 데이터 변환 규칙을 지정한 formula이고 세번째 인자는 요약치를 계산할때 사용할 함수이다.

cast 함수의 formula는 다음과 같이 작성한다.

- “행 변수 ~ 열 변수”형태로 적는다. “~” 앞에 나열된 변수들은 행으로 지정되며, “~” 뒤에 적은 변수들은 열에 배열된다.
- 아무 변수도 지정하지 않으면 . 를 적는다.
- formula에 나열되지 않은 모든 변수를 표현하려면 ... 를 적는다.

### 데이터 형태 변환

실제 예를 통해 formula의 사례를 살펴보자. 다음 코드는 smiths 데이터를 melt 후 다시 원 데이터로 변환하는 예를 보여준다. 코드 마지막에 사용된 함수인 identical()은 두 데이터가 완전히 동일한 객체인지를 알려주는 함수이다.

> smiths
subject time age weight height
1 John Smith     1   33      90    1.87
2 Mary Smith     1    NA      NA    1.54

```

> (m <- melt(id=1:2, smiths))
  subject time variable value
1 John Smith     1      age 33.00
2 Mary Smith     1      age     NA
3 John Smith     1    weight 90.00
4 Mary Smith     1    weight     NA
5 John Smith     1   height  1.87
6 Mary Smith     1   height  1.54

> (x <- dcast(m, subject + time ~ ...))
  subject time age weight height
1 John Smith     1   33     90    1.87
2 Mary Smith     1    NA     NA    1.54

> identical(x, smiths)
[1] TRUE

```

다음 예에서 보다시피 만약 melt() 시에 na.rm을 사용했다고 해도 dcast()를 사용해 원상복귀한 데이터는 원본 데이터와 같다. cast시 셀에 지정할 값이 없다면 암시적으로 NA로 간주하기 때문이다.

```

> dcast(melt(id=1:2, smiths, na.rm=TRUE), subject + time ~ ...)
  subject time age weight height
1 John Smith     1   33     90    1.87
2 Mary Smith     1    NA     NA    1.54

```

다음은 french\_fries를 melt 후 원상복귀 시키는 예이다. identical() 호출전에 rownames() 를 NULL로 둔 것은 french\_fries 데이터에 특이하게 배정된 행이름(아래 코드에서 제일 좌측 61, 25, 62, 26, ... 부분) 등을 제외하기 위해 사용되었다.

```

> head(french_fries)
  time treatment subject rep potato buttery grassy rancid painty
61     1          1       3    1     2.9      0.0      0.0      0.0    5.5
25     1          1       3    2    14.0      0.0      0.0      1.1    0.0
62     1          1      10    1    11.0      6.4      0.0      0.0    0.0
26     1          1      10    2     9.9      5.9      2.9      2.2    0.0
63     1          1      15    1     1.2      0.1      0.0      1.1    5.1

```

```

27      1          1      15    2     8.8      3.0      3.6      1.5      2.3
> ffm <- melt(id=1:4, french_fries)
> head(ffm)
  time treatment subject rep variable value
1   1           1       3   1    potato  2.9
2   1           1       3   2    potato 14.0
3   1           1      10   1    potato 11.0
4   1           1      10   2    potato  9.9
5   1           1      15   1    potato  1.2
6   1           1      15   2    potato  8.8
> x <- dcast(ffm, time + treatment + subject + rep ~ variable)
> head(x)
  time treatment subject rep potato buttery grassy rancid painty
1   1           1       3   1     2.9     0.0     0.0     0.0     5.5
2   1           1       3   2    14.0     0.0     0.0     1.1     0.0
3   1           1      10   1    11.0     6.4     0.0     0.0     0.0
4   1           1      10   2     9.9     5.9     2.9     2.2     0.0
5   1           1      15   1     1.2     0.1     0.0     1.1     5.1
6   1           1      15   2     8.8     3.0     3.6     1.5     2.3
> rownames(french_fries) <- NULL
> rownames(x) <- NULL
> identical(french_fries, x)
[1] TRUE

```

identical()의 결과에서 보다시피 데이터의 melt()와 molten data로부터의 원데이터로의 복귀가 잘 이루어졌다.

## 데이터 요약

cast의 또 다른 유용성은 데이터를 요약하는 기능이 있다는 점이다. 요약을 수행 하려면 cast시 데이터를 원래 차원으로 복원 시키는 대신 더 작은 차원으로 복원 시키는 formula를 지정하면 된다.

다음 예는 ffm 을 만들때 id로 사용되었던 time, treatment, subject, rep 중 time만 행으로 배치하고 측정 변수를 열로 배치한 경우를 보여준다. 이 경우 같은 time값에 해당하는 여러개의 데이터가 존재하게 되며, cast는 자동적으로 length를 적용해 같은 셀에 모인 행의 갯수를 세게된다.

예를들어 아래 결과에서 time이 1일때 potato의 값은 72인데, 이는 melt된 데이터에서 time 값이 1일때 potato 값이 측정된 행의 수가 72개였음을 뜻한다. (이 예의 출력결과에는 'Aggregation function missing'이라는 에러메시지가 나왔다. 이에 대해서는 잠시 뒤 자세히 살펴본다.)

```
> ffm <- melt(id=1:4, french_fries)
> dcast(ffm, time ~ variable)
Aggregation function missing: defaulting to length
   time potato buttery grassy rancid painty
1     1      72      72      72      72      72
2     2      72      72      72      72      72
3     3      72      72      72      72      72
4     4      72      72      72      72      72
5     5      72      72      72      72      72
6     6      72      72      72      72      72
7     7      72      72      72      72      72
8     8      72      72      72      72      72
9     9      60      60      60      60      60
10    10     60      60      60      60      60
```

좀 더 이해하기 쉽게 R의 기본함수로 이를 계산해보자. time이 1일때 potato를 측정한 행의 수는 다음과 같다.

```
> NROW(subset(ffm, time==1 & variable=="potato"))
[1] 72
```

또는 plyr을 사용해 위에서 보인 값 전부를 다음과 같이 구해볼 수도 있다.

```
> ddply(ffm, .(time, variable), function(rows) { NROW(rows) })
  time variable V1
1     1    potato  72
2     1   buttery  72
3     1    grassy  72
4     1   rancid  72
5     1   painty  72
6     2    potato  72
7     2   buttery  72
8     2    grassy  72
```

```

9      2    rancid 72
10     2    painty 72
11     3    potato 72
12     3   buttery 72
...

```

어떤 방법을 사용해도 원하는 값은 구할 수 있으나, reshape2의 방법이 좀 더 직관적이다.

앞서 예에서는 여러개의 값이 하나의 행에 모일 경우 어떻게 요약할 것인지를 지정하지 않고 dcast를 수행하였기 때문에 ‘Aggregation function missing: defaulting to length’라는 경고 메시지가 나왔다. 그리고 length()가 암시적으로 적용되었다. 경고 메시지는 요약에 사용할 함수를 다음과 같이 명시적으로 지정하면 사라진다.

```

> dcast(ffd, time ~ variable, length)
  time potato buttery grassy rancid painty
1      1      72      72      72      72      72
2      2      72      72      72      72      72
3      3      72      72      72      72      72
4      4      72      72      72      72      72
5      5      72      72      72      72      72
6      6      72      72      72      72      72
7      7      72      72      72      72      72
8      8      72      72      72      72      72
9      9      60      60      60      60      60
10     10     60      60      60      60      60

```

좀 더 재미있는 통계는 length 보다는 sum, mean, 또는 임의의 함수를 적용해 구할 수 있다. 다음 데이터는 time에 따라 평균값이 어떻게 달라지는지를 보여준다.

```

> dcast(ffd, time ~ variable, mean)
  time potato buttery grassy rancid painty
1      1 8.562500 2.236111 0.9416667 2.358333 1.645833
2      2 8.059722 2.722222 1.1819444 2.845833 1.444444
3      3 7.797222 2.102778 0.7500000 3.715278 1.311111
4      4 7.713889 1.801389 0.7416667 3.602778 1.372222
5      5       NA        NA        NA        NA        NA
6      6 6.670833 1.752778 0.6736111 4.075000 2.341667
7      7 6.168056        NA 0.4208333 3.886111 2.683333
8      8 5.431944        NA 0.3805556 4.272222        NA

```

9	9	5.673333	1.586667	0.2766667	4.670000	3.873333
10	10	5.703333	1.765000	0.5566667	6.068333	5.291667

위 결과에서 NA 값들은 ffm에 value==NA인 행들이 있기에 발생하였다. 이는 melt를 할 때 na.rm=TRUE를 지정해 NA가 포함된 행을 제외하거나, 또는 cast시 mean 함수에 na.rm=TRUE를 지정해 피할 수 있다. 다음은 이 두가지 경우의 예를 각각 보였다.

```
> dcast(melt(id=1:4, french_fries, na.rm=TRUE), time ~ variable, mean)
   time potato buttery grassy rancid painty
1     1 8.562500 2.236111 0.9416667 2.358333 1.645833
2     2 8.059722 2.722222 1.1819444 2.845833 1.444444
3     3 7.797222 2.102778 0.7500000 3.715278 1.311111
4     4 7.713889 1.801389 0.7416667 3.602778 1.372222
5     5 7.328169 1.642254 0.6352113 3.529577 2.015493
6     6 6.670833 1.752778 0.6736111 4.075000 2.341667
7     7 6.168056 1.369014 0.4208333 3.886111 2.683333
8     8 5.431944 1.182857 0.3805556 4.272222 3.938028
9     9 5.673333 1.586667 0.2766667 4.670000 3.873333
10    10 5.703333 1.765000 0.5566667 6.068333 5.291667

> dcast(melt(id=1:4, french_fries), time ~ variable, mean, na.rm=TRUE)
   time potato buttery grassy rancid painty
1     1 8.562500 2.236111 0.9416667 2.358333 1.645833
2     2 8.059722 2.722222 1.1819444 2.845833 1.444444
3     3 7.797222 2.102778 0.7500000 3.715278 1.311111
4     4 7.713889 1.801389 0.7416667 3.602778 1.372222
5     5 7.328169 1.642254 0.6352113 3.529577 2.015493
6     6 6.670833 1.752778 0.6736111 4.075000 2.341667
7     7 6.168056 1.369014 0.4208333 3.886111 2.683333
8     8 5.431944 1.182857 0.3805556 4.272222 3.938028
9     9 5.673333 1.586667 0.2766667 4.670000 3.873333
10    10 5.703333 1.765000 0.5566667 6.068333 5.291667
```

다음은 treatment를 행으로하고 rep와 variable을 열로 하도록 변환한 뒤 평균값을 계산한 예이다. 결과에서 1\_potato는 rep가 1일때 potato 값을, 2\_potato는 rep가 2일때 potato 값을 뜻한다. 이처럼 cast에서는 다양한 방식의 데이터 축약이 가능하다.

```
> dcast(ffm, treatment ~ rep + variable, mean, na.rm=TRUE)
```

```

treatment 1_potato 1_buttery 1_grassy 1_rancid 1_painty
1          1 6.772414  1.797391 0.4456897 4.283621 2.727586
2          2 7.158621  1.989474 0.6905172 3.712069 2.315517
3          3 6.937391  1.805217 0.5895652 3.752174 2.038261

2_potato 2_buttery 2_grassy 2_rancid 2_painty
1 7.003448  1.762931 0.8525862 3.847414 2.439655
2 6.844828  1.958621 0.6353448 3.537069 2.597391
3 6.998276  1.631034 0.7706897 3.980172 3.008621

```

## 4 data.table 패키지

데이터 테이블[12]은 R의 기본 데이터 타입인 데이터 프레임을 대신하여 사용할 수 있는 더 빠르고 편리한 데이터 타입이다.

### 4.1 데이터 테이블 생성

데이터 테이블은 데이터 프레임을 만드는 것과 동일한 문법으로 생성한다. 또는 데이터 프레임과 데이터 테이블간 as.data.frame() 또는 as.data.table()를 사용해 상호 변환해도 된다.

다음은 iris를 데이터 테이블로 변환하는 예이다. 데이터 테이블을 출력했을 때 데이터 프레임과 달리 데이터의 일부만 잘려 보여지는 점은 상당히 편리하다. 더 많은 데이터를 출력해 보려면 iris\_table[1:n, ]와 같이 행 번호를 직접 지정하거나 print(your.data.table, nrow=Inf) 명령을 사용하면 된다.

```

> iris_table <- as.data.table(iris)
> iris_table
      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
---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica

```

149:	6.2	3.4	5.4	2.3	virginica
150:	5.9	3.0	5.1	1.8	virginica

다음은 데이터 테이블을 직접 생성하는 예이다.

```
> (x <- data.table(x=c(1, 2, 3), y=c("a", "b", "c")))
  x y
1: 1 a
2: 2 b
3: 3 c
```

이 절의 처음에서 언급한 바와 같이 데이터 테이블은 데이터 프레임과 동일하게 취급된다. 다음 코드는 데이터 테이블의 클래스가 data.frame을 포함하고 있음을 보여준다. 클래스가 data.frame 을 포함하므로 summary, print, plot 등의 데이터 프레임을 처리하는 함수들이 데이터 테이블에도 동일하게 동작한다. 다시말해 data.frame 을 인자로 기대하는 함수에 데이터 테이블을 넘겨도 많은 경우 별 문제 없이 동작한다. 만약 예외적인 상황이 발생한다면 데이터 테이블을 as.data.frame()를 사용해 데이터 프레임으로 변환시키면 된다.

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

이렇게 만든 데이터 테이블들의 목록은 table()로 열람할 수 있다.

```
> iris_table <- as.data.table(iris)
> x <- data.table(x=c(1, 2, 3), y=c("a", "b", "c"))
> tables()
      NAME      NROW MB
[1,] iris_table  150  1
[2,] x           3   1
      COLS          KEY
[1,] Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, Species
[2,] x, y
Total: 2MB
```

## 4.2 데이터 접근과 그룹 연산

데이터 테이블은 데이터 프레임이 사용되는 대부분의 곳에서 문제없이 동작한다. 하지만 데이터를 접근할 때는 몇가지 주의를 기울여야한다.

데이터 테이블의 데이터는 [행, 표현식, 옵션] 형태로 접근한다. 행은 ‘행번호’ 또는 ‘행을 선택할지를 나타내는 진리값’으로 지정한다. 이는 데이터 프레임과 동일하다. 다음 예를 보자.

```
> DT <- as.data.table(iris)

> DT[1,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1:          5.1         3.5        1.4       0.2   setosa

> DT[DT$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
3:          4.7         3.2        1.3       0.2   setosa
```

데이터 테이블 접근시 [ ] 의 두번째 인자에는 컬럼명을 그대로 적거나 또는 컬럼명에 대한 표현식을 적는다.

다음은 1행의 컬럼 Sepal.Length를 선택한 예이다. 컬럼명을 ‘Sepal.Length’가 아닌 Sepal.Length로 따옴표없이 지정하였음을 유의하기 바란다. 따옴표로 묶는 형식은 데이터 프레임에서 사용하는 문법이다.

```
> DT
  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
---
146:          6.7         3.0        5.2       2.3 virginica
147:          6.3         2.5        5.0       1.9 virginica
148:          6.5         3.0        5.2       2.0 virginica
149:          6.2         3.4        5.4       2.3 virginica
150:          5.9         3.0        5.1       1.8 virginica

> DT[1, Sepal.Length]
[1] 5.1
```

여러 컬럼을 선택하고자 한다면 list()안에 컬럼들을 나열한다.

```
> DT[1, list(Sepal.Length, Species)]
  Sepal.Length Species
1:          5.1   setosa
```

컬럼명을 그대로 사용해 연산을 수행할 수도 있다.

```
> DT[, mean(Sepal.Length)]
[1] 5.843333

> DT[, mean(Sepal.Length - Sepal.Width)]
[1] 2.786
```

그러나 데이터 테이블의 두번째 인자로 컬럼 이름을 담은 문자열 또는 컬럼 번호를 지정하고자 한다면 with=FALSE 옵션을 주어야한다. 그렇지 않으면 두번째 인자로 지정한 문자열 또는 컬럼 번호를 그대로 ‘연산식’ 취급해버리기 때문이다.

```
> DT <- as.data.table(iris)
> 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

> iris[1, 1]
[1] 5.1

> DT[1, 1]
[1] 1

> DT[1, 1, with=FALSE]
  Sepal.Length
1:          5.1
```

```
> iris[1, c("Sepal.Length")]
[1] 5.1

> DT[1, c("Sepal.Length")]
[1] "Sepal.Length"

> DT[1, c("Sepal.Length"), with=FALSE]
  Sepal.Length
1:           5.1
```

데이터 테이블의 세번째 인자에는 데이터를 그룹화할 변수를 지정할 수 있다. 예를 들어 Sepal.Length의 평균 값을 Species별로 구하는 계산은 세번째 인자로 by="Species"를 지정해 수행할 수 있다.

```
> DT[, mean(Sepal.Length), by="Species"]
    Species      V1
1:   setosa  5.006
2: versicolor 5.936
3: virginica 6.588
```

만약 그룹화할 변수가 여러개라면 by에 컬럼명을 계속 나열하면 된다. 다음 예를 살펴보자.

```
> DT <- data.table(x=c(1, 2, 3, 4, 5),
+                     y=c("a", "a", "a", "b", "b"),
+                     z=c("c", "c", "d", "d", "d"))

> DT
  x y  z
1: 1 a  c
2: 2 a  c
3: 3 a  d
4: 4 b  d
5: 5 b  d

> DT[, mean(x), by="y,z"]
  y z   V1
1: a c  1.5
2: a d  3.0
```

3: b d 4.5
------------

이처럼 연산을 손쉽게 표현할 수 있다는 점이 데이터 테이블의 장점이다.

### 4.3 key를 사용한 탐색

데이터 프레임에서 특정 컬럼에 특정 값이 들어있는 행을 찾는 작업은 모든 행의 값을 하나하나 검토하는 방식으로 이루어진다. 따라서 데이터 양이 많고 데이터 검색 작업의 횟수가 많다면 긴 수행시간이 걸린다.

다음 예는 x에 연속 균등 분포(uniform distribution)을 따르는 값을 저장하고, y에 각 알파벳을 10000회씩 반복한 값을 저장한 데이터 프레임에서 y값이 C인 행을 선택하는 예이다.

```
> DF <- data.frame(x=runif(260000), y=rep(LETTERS, each=10000))
> str(DF)
'data.frame': 260000 obs. of 2 variables:
 $ x: num 0.1405 0.0378 0.8146 0.3096 0.9899 ...
 $ y: Factor w/ 26 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 ...
> head(DF)
      x y
1 0.14046554 A
2 0.03776698 A
3 0.81462321 A
4 0.30959019 A
5 0.98985253 A
6 0.24352429 A

> system.time(x <- DF[DF$y=="C", ])
    user   system elapsed
0.034     0.000   0.033
```

system.time() 을 사용해 평가한 결과를 보면 총 수행시간(elapsed time)은 0.033초가 걸린 것으로 나타났다<sup>3)</sup>. 이 값이 특별히 크다고 할 수는 없으나 데이터 양이 많다면 얼마든지 더 많은 시간이 걸릴 수 있다.

이를 해결하는 방법은 y 값에 대해 미리 색인<sup>4)</sup>을 만들어두었다가 y값을 사용한 검색시 색인을 사용하여 검색하는 것이다. 색인을 만드는 것에도 시간이 소요되기는 하지만 데이터 검색

<sup>3)</sup>system.time()에 대해서는 [system.time\(\)을 사용한 시간 측정](#) (페이지 168)에서 자세히 다룬다. 여기서는 주어진 명령의 총 소요시간을 elapsed time 컬럼을 읽어 알 수 있다는 점만 알고 넘어가도록 하자.

<sup>4)</sup>이진 탐색 트리(binary search tree)가 사용된다.

횟수가 많다면 결과적으로 많은 속도 향상을 이룰 수 있다. 색인을 만들기위해서는 setkey() 명령을 사용한다. 다음 예를 살펴보자.

```
> DT <- as.data.table(DF)
> setkey(DT, y)
> system.time(x <- DT[J("C")], )
    user   system elapsed
0.002     0.000   0.002
```

보다시피 총 소요시간이 0.002초로 훨씬 빨라진 것을 볼 수 있다. 만약 여러 컬럼을 키로 만들어 두어야한다면 setkey(DT, 컬럼1, 컬럼2, ...) 형태로 다수의 키를 지정할 수 있으며 J(컬럼1, 컬럼2, ...) 형태로 키를 나열해 데이터를 검색하면 된다.

특히 데이터 테이블의 두번째 인자는 컬럼명을 사용한 표현식이 될 수 있기에 검색과 동시에 계산을 수행할 수 있다. 다음 예는 y 값이 C 인 행들을 찾은 뒤 x 의 평균을 구한 결과이다.

```
> DT[J("C"), mean(x)]
      y          V1
1: C 0.5033286
```

위 결과에서 V1이라는 컬럼 이름이 보기 싫다면 list()안에 표현식을 나열해 컬럼이름을 지정할 수 있다. 다음은 y값이 C인 행들에서 x의 평균과 표준편차를 계산한 예이다.

```
> DT[J("C"), list(x_mean=mean(x), x_std=sd(x))]
      y      x_mean      x_std
1: C 0.5033286 0.2904743
```

#### 4.4 key를 사용한 데이터 테이블 병합

key는 여러 데이터 테이블의 병합에도 사용할 수 있다. 예를 들어 DT1, DT2의 두개 데이터 테이블이 있을때 DT1[DT2, 표현식]는 DT1에서 DT2에 대응되는 데이터를 찾는 방식으로 데이터를 병합한다.

다음 두개 데이터 테이블을 가정해보자.

```
> DT1 <- data.table(x=runif(260000), y=rep(LETTERS, each=10000))
> DT2 <- data.table(y=c("A", "B", "C"), z=c("a", "b", "c"))
```

DT1[DT2, ]는 DT1으로부터 y값이 A, B, C인 행을 찾아 병합한다.

```
> setkey(DT1, y)
```

```
> DT1[DT2, ]
      y           x   z
 1: A 0.36912925 a
 2: A 0.60170742 a
 3: A 0.16972871 a
 4: A 0.27251097 a
 5: A 0.96946385 a
---
29996: C 0.66247989 c
29997: C 0.69217630 c
29998: C 0.02152985 c
29999: C 0.60682416 c
30000: C 0.07117150 c
```

반면 DT2[DT1, ]은 260000개의 행을 DT2로부터 검색하므로 그 결과가 총 260000행이된다.

```
> setkey(DT2, y)
> DT2[DT1, ]
      y   z           x
 1: A  a 0.36912925
 2: A  a 0.60170742
 3: A  a 0.16972871
 4: A  a 0.27251097
 5: A  a 0.96946385
---
259996: Z NA 0.04522689
259997: Z NA 0.50320326
259998: Z NA 0.01764672
259999: Z NA 0.51967080
260000: Z NA 0.78829015
```

데이터 테이블간의 병합은 색인을 활용하므로 그 속도가 빠르다. 또 데이터 테이블의 병합 시 DT1[DT2, 표현식]과 같이 얻고자 하는 결과의 표현식을 바로 지정할 수 있으므로 데이터 병합과 동시에 계산을 수행할 수 있는 장점이 있다.

편리해보이기는 하지만 data.frame과 merge()를 사용해도 같은 결과를 얻을 수 있지 않을까? 물론 merge()를 사용해서도 같은 결과를 얻을 수 있다. 그러나 data.table이 제공하는 테이블 병합 방식의 장점은 속도에 있다. 다음은 위 코드를 수행하는데 필요한 시간을 비교한

예이다.

```
> library(data.table)

> DT1 <- data.table(x=runif(260000), y=rep(LETTERS, each=10000))
> DT2 <- data.table(y=c("A", "B", "C"), z=c("a", "b", "c"))
> setkey(DT1, y)
> system.time(DT1[DT2, ])
    user   system elapsed
0.002    0.001   0.005

> DF1 <- as.data.frame(DT1)
> DF2 <- as.data.frame(DT2)
> system.time(merge(DF1, DF2))
    user   system elapsed
0.102    0.010   0.112
```

30,000 행의 출력을 만드는데 데이터 테이블은 총 0.005초가 소요되었고 데이터 프레임은 0.112초가 소요되었다. 즉 데이터 테이블이 20배 빠르게 결과를 내놓는다.

## 4.5 참조를 사용한 데이터 수정

[객체의 불변성](#) (페이지 75)절에서 R 객체는 수정되지 않으며, 값을 수정하는 것처럼 보이는 경우에도 실제로는 새로운 객체를 매번 만든다고 설명했다. 따라서 for 문안에서의 데이터 수정은 매우 긴 시간이 소요되고 이런 이유로 벡터화된 연산을 사용해야한다.

데이터 테이블은 데이터 프레임이 지원하는 일반적인 데이터 수정 연산외에도 := 연산자를 사용한 데이터 수정 기능을 제공한다. 방식은 DT[i, 변수명 := 값]의 형태이다. 다음은 데이터 프레임과 데이터 테이블을 각각 사용하여 for 문에서 첫번째 컬럼에 1부터 1000까지의 값을 지정하는 예이다. 이 예는 데이터 테이블 메뉴얼[12]에서 인용하였다.

```
> m = matrix(1, nrow=1000, ncol=100)
> DF <- as.data.frame(m)
> DT <- as.data.table(m)

> system.time({
+   for (i in 1:1000) {
+     DF[i, 1] <- i
+   }
})
```

```
+ })
  user    system elapsed
0.741    0.411   1.347

> system.time(for(i in 1:1000) {
+   DT[i, V1 := i]
+ })
  user    system elapsed
0.330    0.006   0.373
```

위 결과에서 볼 수 있듯이 데이터 테이블의 속도가 3.6배 빠르다. 그러나 모든 경우 데이터 테이블의 참조 연산 방식이 데이터 프레임에 비해 우위에 있지는 않다. 예를 들어 데이터의 다른 컬럼을 참조하는 경우 등에서는 데이터 프레임이 더 빠른 경우도 있으니 항상 속도를 직접 측정해보면서 적절한 방식을 택해나가길 바란다.

## 4.6 rbindlist

[lapply\(\)](#) (페이지 92)를 비롯해 많은 R 함수들이 결과를 리스트로 반환한다. 리스트를 반환값으로 채택한 이유중 하나는 아마도 결과값의 데이터 타입을 다양하게 설정할 수 있다는 점 때문일 것이다. 다음 예에서는 리스트의 첫번째 요소는 x, y, z를 모두 갖고 있는 반면 두번째 요소는 x, y만 갖고 있는 경우를 보여준다.

```
> x <- list()
> x[[1]] <- c(1, 2, 3)
> names(x[[1]]) <- c('x', 'y', 'z')
> x[[2]] <- c(1, 2)
> names(x[[2]]) <- c('x', 'y')

> x
[[1]]
x y z
1 2 3

[[2]]
x y
1 2
```

그러나 많은 데이터 모델링 또는 시각화 함수들은 그 인자로 데이터 프레임을 받는다. 또 아무래도 데이터 프레임이 보기 편하다. 이런 이유로 결과를 데이터 프레임으로 변환할 필요가 생기게 된다.

데이터 프레임을 함수 실행 결과로 얻는 가장 간단한 방법은 [plyr 패키지](#) (페이지 121)의 `ldply()` 등과 같이 결과를 데이터 프레임으로 출력하는 함수를 사용하는 것이다. 하지만 안타깝게도 `ldply()` 함수의 속도는 `llply()`와 같이 리스트를 결과로 출력하는 경우에 비해 좋지 않다. 다음은 `plyr`의 `ldply()`와 `llply()`의 성능을 비교한 예이다.

```
> system.time(x <- ldply(1:10000, function(x) {
+   data.frame(val=x,
+             val2= 2 * x,
+             val3= 2 / x,
+             val4 = 4 * x,
+             val5 = 4 / x)
+ }))
```

	user	system	elapsed
	5.757	0.739	6.575

```
> system.time(x <- llply(1:10000, function(x) {
+   data.frame(val=x,
+             val2= 2 * x,
+             val3= 2 / x,
+             val4 = 4 * x,
+             val5 = 4 / x)
+ }))
```

	user	system	elapsed
	3.452	0.012	3.466

위 결과를 보면 같은 명령을 수행하는 코드임에도 불구하고 `ldply()`의 실행 시간(elapsed time)이 `llply()`의 실행시간의 거의 2배에 가까운 것을 알 수 있다. 그 이유는 `ldply()`는 `llply()`를 사용해 리스트로 구성된 결과를 얻은 다음 이를 다시 데이터 프레임으로 변환하기 때문이다. 그리고 리스트를 데이터 프레임으로 만드는데는 생각보다 긴 시간이 소요된다.

이를 보다 직접적으로 알아보기 위해 다음 예를 살펴보자.

```
> x <- lapply(1:10000, function(x) {
+   data.frame(val=x,
+             val2= 2 * x,
```

```
+           val3= 2 / x ,
+           val4 = 4 * x ,
+           val5 = 4 / x)
+ })

> head(x)
[[1]]
  val  val2  val3  val4  val5
1     1      2      2      4      4

[[2]]
  val  val2  val3  val4  val5
1     2      4      1      8      2

[[3]]
  val  val2          val3  val4          val5
1     3      6  0.6666667    12  1.333333

[[4]]
  val  val2  val3  val4  val5
1     4      8   0.5     16      1

[[5]]
  val  val2  val3  val4  val5
1     5     10   0.4     20     0.8

[[6]]
  val  val2          val3  val4          val5
1     6     12  0.3333333    24  0.6666667

> system.time(y <- do.call(rbind, x))
  user  system elapsed
  1.894   0.461   2.429

> head(y)
  val  val2          val3  val4          val5
```

1	1	2 2.0000000	4 4.0000000
2	2	4 1.0000000	8 2.0000000
3	3	6 0.6666667	12 1.3333333
4	4	8 0.5000000	16 1.0000000
5	5	10 0.4000000	20 0.8000000
6	6	12 0.3333333	24 0.6666667

위 결과에서 보다시피 do.call(rbind, 데이터 프레임)<sup>5)</sup>을 사용해 리스트 안에 저장된 데이터 프레임을 하나로 합치는데 2.429초가 소요되었다. 2초 정도는 긴 시간이 아니지만 컬럼 수가 많아지거나 데이터가 많아지면 이 시간은 급격히 증가하며 심지어 데이터 처리의 대부분의 시간이 여기에만 소요되기도 한다.

이러한 문제를 해결해주는 data.table 패키지의 함수가 rbindlist()이다. rbindlist는 인자로 데이터 테이블의 리스트를 받아 하나의 데이터 테이블로 합쳐준다. 따라서 lapply()를 사용해 일단 데이터 테이블의 리스트를 만든 다음 이를 다시 하나의 데이터 테이블로 병합한다면 그 속도가 매우 빠를 것이다. 이를 다음 예에서 살펴본다.

```
> system.time(x <- lapply(1:10000, function(x) {
+   data.frame(val=x,
+             val2= 2 * x,
+             val3= 2 / x,
+             val4 = 4 * x,
+             val5 = 4 / x)
+ }))

 user    system elapsed
 5.763    0.909    6.901

> system.time(x <- lapply(1:10000, function(x) {
+   data.table(val=x,
+             val2= 2 * x,
+             val3= 2 / x,
+             val4 = 4 * x,
+             val5 = 4 / x)
+ }))

 user    system elapsed
 1.336    0.025    1.449
```

<sup>5)</sup>do.call(rbind, 데이터 프레임)을 사용한 변환에 대해서는 앞서 [lapply\(\)](#) (페이지 92)에서 설명한 바 있다.

```
> system.time(x <- rbindlist(x))
   user  system elapsed
0.004    0.000    0.004
```

보다시피 데이터 프레임을 대신 만들어주는 ldply()는 6.901초가 소요된 반면 llply()를 사용해 데이터 테이블들의 리스트를 먼저 만든 다음 이를 다시 rbindlist()로 합쳐 최종 데이터 테이블을 만드는 코드는 총 1.453초( $= 1.449 + 0.004$ )가 소요되어 4.6배 빠르게 수행되었다.

## 5 foreach

foreach[13]는 [apply 함수들](#) (페이지 90), for 문 등을 대체할 수 있는 루프문을 위한 함수이다. for 문과의 가장 큰 차이는 반환값이 있고, %do% 문을 사용해 블럭을 지정한다는 점이다.

1에서 5까지의 숫자를 루프를 돌면서 %do%안에서 반환하면 1부터 5까지의 숫자를 담은 리스트가 반환된다.

```
> foreach(i=1:5) %do% {
+   i
+ }
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4

[[5]]
[1] 5
```

foreach의 결과로 얻어진 리스트는 앞서 살펴본 [rbindlist](#) (페이지 148)를 사용해 하나의 데이터 프레임으로 변환할 수 있다.

또는 foreach()에 .combine=c를 지정해 처음부터 벡터로 결과를 받을 수도 있다.

```
> foreach(i=1:5, .combine=c) %do% {
+   i
+
[1] 1 2 3 4 5
```

.combine에는 이외에도 rbind, cbind 등의 함수를 지정할 수도 있다.

```
> foreach(i=1:5, .combine=rbind) %do% {
+   data.frame(val=i)
+
  val
1    1
2    2
3    3
4    4
5    5

> foreach(i=1:5, .combine=cbind) %do% {
+   data.frame(val=i)
+
  val  val  val  val  val
1    1    2    3    4    5
```

그러나 앞서 [rbindlist](#) (페이지 148)에서 보인 성능 문제가 발생할 수 있으니 일단 리스트를 결과로 받고 rbindlist()를 하는 방법에 대해서도 고려하기 바란다.

.combine에는 또한 연산자를 지정할 수 있다. 예를 들어 ‘+’를 지정하면 모든 결과를 합한 결과를 반환한다. 다음은 1부터 10까지의 합을 계산하는 예이다.

```
> foreach(i=1:10, .combine="+") %do% {
+   i
+
[1] 55
```

이처럼 .combine에 연산자를 지정하면 foreach를 사용해 함수형 언어에서 제공되는 Reduce를 구현할 수 있다.

## 6 doMC

doMC[14]는 멀티코어를 활용해 프로그램을 병렬적으로 수행할 수 있게 해준다. 멀티 코어를 활용한다는 뜻은 여러개의 프로세스를 실행하고 각 프로세스가 개별 CPU 코어에서 돌아가게 한다는 의미이다.

doMC는 [plyr 패키지](#) (페이지 121) 또는 [foreach](#) (페이지 152)와 결합해 편리하게 사용할 수 있다.

### 6.1 프로세스의 수 설정

doMC를 사용하기 위해 가장 먼저 할일은 몇개의 프로세스를 사용할 것인지 선택하는 것이다. 이 숫자는 크게 두가지 측면에서 살펴볼 수 있다.

첫째로 어느정도의 입출력 작업이 있는가 하는 점이다. 디스크, 데이터베이스, 네트워크 입출력이 발생하면 기기(디스크나 네트워크 인터페이스 카드)에서 응답이 올 때까지 해당 프로세스는 CPU를 사용하지 않는다. 따라서 이러한 입출력 작업이 많다면 코어보다 프로세스의 수를 더 많이 정해 입출력을 대기중인 프로세스가 CPU를 사용하지 않는 동안 다른 프로세스들이 해당 CPU를 사용할 수 있게 할 수 있다. 그러나 입출력 작업이 거의 없는 상태에서 다수의 프로세스를 실행하게되면, 하나의 CPU 코어가 여러 프로세스를 실행해야하고, 이 경우 하나의 CPU 자원을 위해 여러 프로세스가 경쟁하는 환경이 되어 오히려 성능이 떨어질 수 있다.

둘째로 여러 프로세스를 실행하면 그 각각이 메모리를 소요하게 된다는 점이다. 특히 사용 중인 알고리즘이나 모델링 패키지에 따라서 이 용량은 매우 커질 수 있다. 예를들어 어떤 데이터를 분석하는데 통상 100M의 메모리가 소요된다고 하자. 만약 프로세스가 4개라면 각각의 프로세스가 25M씩 소모하여 총 100M의 메모리를 차지할 것 같지만 실제로는 그보다 더 큰 메모리를 사용할 수 있다. 따라서 사용중인 컴퓨터의 메모리를 잘 관찰해 메모리 여분이 충분히 남는지, 그리고 불필요한 페이지ing이나 스와핑<sup>6)</sup>이 발생하지 않는지 등 메모리 사용량을 유심히 지켜볼 필요가 있다. 만약 페이지ing이나 스와핑이 발생한다면 오히려 디스크 입출력 작업으로 인해 성능이 더 떨어질 수 있다.

모든 경우에 맞는 프로세스의 수란 존재하지 않으며 상황에 따라 적절히 조절해나가야 한다. 만약 적절한 입출력이 있는 경우라면 코어갯수 \* 2로부터 시작할 것을 추천한다. 하지만 CPU 작업이 대부분이라면 코어갯수만큼 프로세스를 설정하거나 또는 registerDoMC()의 기본값처럼 코어갯수의 절반을 선택할 수도 있다.

다음은 registerDoMC()의 사용예이다. cores의 명칭이 마치 CPU 코어 수를 연상하게 하지 만 실제로는 실행할 프로세스의 수임에 유의하기 바란다.

---

<sup>6)</sup>메모리가 부족하면 운영체제는 사용중이지 않은 프로세스를 디스크로 옮기거나, 또는 디스크를 가상적인 메모리로 활용하게 된다. 어느쪽이든 메모리가 부족하면 급격히 발생하기 시작하며, 디스크 입출력으로 인해 오히려 성능이 저하될 수 있다.

```
> library(doMC)
> registerDoMC(cores=8)
```

만약 인자 없이 registerDoMC()를 실행하면 기본값인 코어 갯수의 절반이 프로세스 수로 설정된다.

## 6.2 plyr의 .parallel 옵션

plyr의 함수들 중 ??ply() 형태 함수들의 도움말을 살펴보면 .parallel 옵션이 있다. 이 옵션 값이 TRUE로 설정되면 registerDoMC() 를 사용해 설정한 만큼의 프로세스가 동시에 실행되어 데이터를 병렬적으로 처리한다. 그리고 그 결과들은 자동으로 하나의 결과로 병합되어 사용자에게 반환된다. 그러나 .parallel의 기본값은 FALSE이므로 이 옵션을 명시적으로 설정하지 않으면 registerDoMC()를 호출하였다 할지라도 병렬화가 사용되지 않는다.

.parallel=TRUE를 지정하는 예를 살펴보자. 다음 코드는 value에는 값, group에는 그룹을 나타내는 알파벳이 담긴 데이터 프레임을 만든 다음 각 그룹별 value의 평균을 구하는 예이다.

```
> big_data <- data.frame(
+   value=runif(NROW(LETTERS) * 2000000) ,
+   group=rep(LETTERS , 2000000))

> dplyr(big_data, .(group), function(x) {
+   mean(x$value)
+ },
+ .parallel=TRUE)
```

dplyr()를 실행한 다음 프로세스를 관찰해보면 그림 6.1에서 볼 수 있듯이 다수의 R 프로세스가 실행되어 데이터를 처리한다. 병렬로 작업을 실행했을 때에는 프로세스의 실행을 관찰하면서 동시에 시스템 전체의 가용 메모리가 얼마나 남아 있는지를 점검하면서 혹시라도 다수의 프로세스가 동시에 많은 메모리를 필요로 하여 혹시라도 **thrashing**을 발생시키지는 않는지 확인할 필요가 있다.

만약 thrashing이 발생한다면 다음 증상들을 볼 수 있다.

- 시스템의 가용메모리가 거의 남지 않게 된다.
- 디스크 입출력이 지속적으로 많이 발생한다.
- R 프로세스의 CPU 사용량이 낮다.

예를 들어 그림 6.1에서는 R 프로세스가 사용하는 메모리가 각 1.5G씩이고 프로세스의 수는 8개였다. 그러나 이 때 사용한 머신의 총 메모리 크기는 8G이어서 thrashing이 발생하게 되었다. 따라서 그림에서 보다시피 R 프로세스들의 CPU 사용량은 약 4.5% 수준에 머무르고 있었다. 또한 그림에는 보이지 않지만 디스크 입출력이 많이 일어나고 있었다. 이로 인해 전체적인 작업 진행 속도는 느리게 진행되었다. 이러한 경우 registerDoMC()에 cores를 더 크게 지정하거나 또는 경우에 따라서는 병렬화를 하지 않는 것이 낫다.

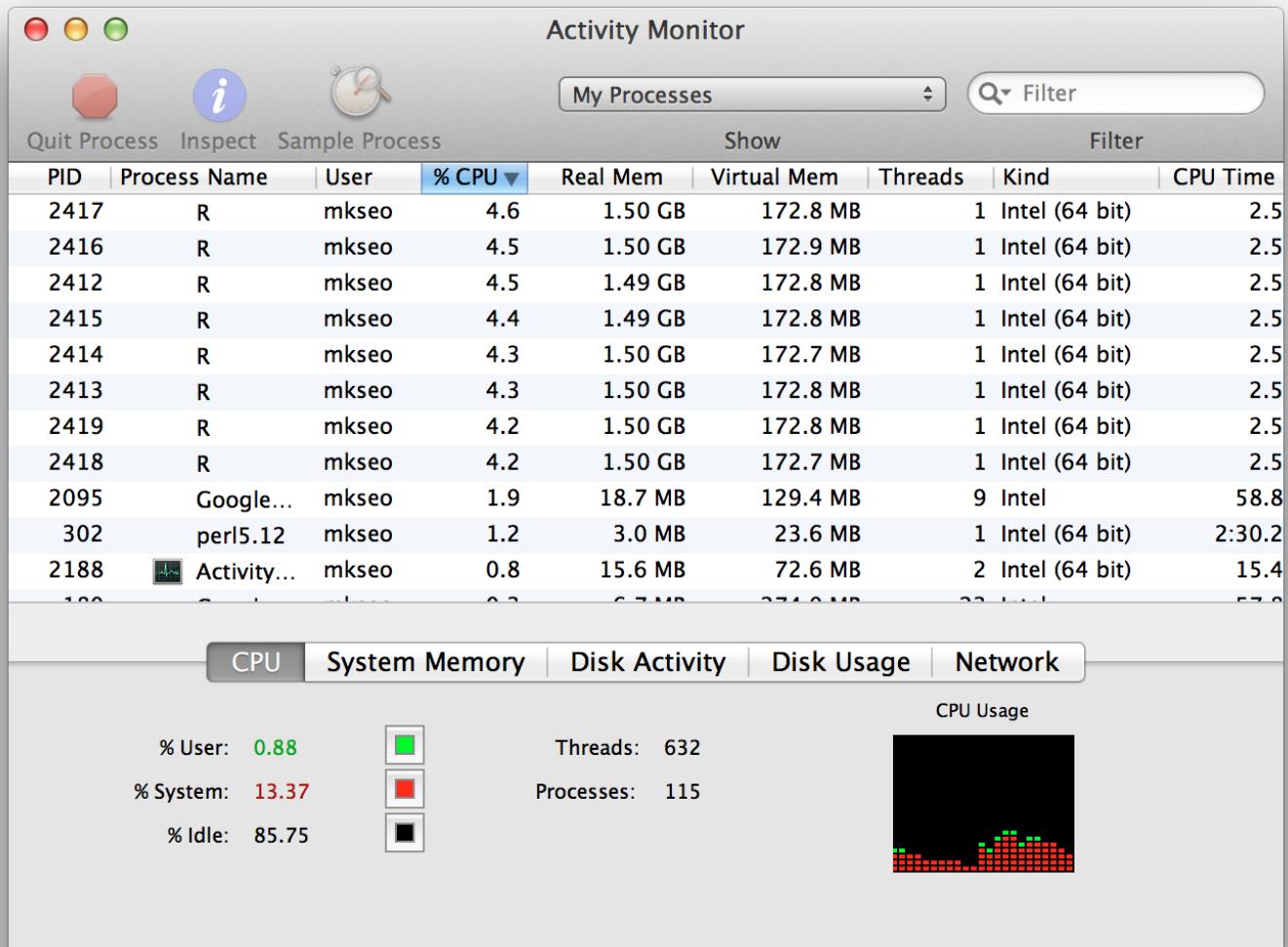


그림 6.1: .parallel=TRUE를 지정한 멀티코어의 활용 중 thrashing이 발생한 경우

### 6.3 foreach에서 %dopar%의 사용

foreach() 함수에는 %do%를 사용해 실행할 명령의 블럭을 지정해야함을 앞서 설명했다. 만약 foreach()에서 블럭내 명령을 동시에 수행하고자 한다면 %do% 대신 %dopar%를 지정하면 된다.

다음은 i 값이 1부터 800000까지 변할 때 이 값을 big\_data\$value에 더한다음 그 평균을 매 i마다 계산한 예이다.

```
> library(doMC)
> library(foreach)
> registerDoMC(cores=8)
> foreach(i=1:800000) %dopar% {
+   mean(big_data$value + i)
+ }
```

이 예에서는 설명을 위해 다소 작위적인 코드를 작성하였다. 보다 실제적인 예로는 random forest 모델을 만들 때 적절한 tree 갯수를 설정하기 위해 트리 크기를 바꿔가면서 모델을 여러개 만드는 예[13]를 들 수 있다. 다음에 이를 의사코드(pseudo code)로 표현하였다.

```
> foreach(ntree=c(10, 20, 30, 100, 1000)) %dopar% {
+   build_model(big_data, ntree=ntree)
+ }
```

## 7 테스팅과 디버깅

### 7.1 testthat

유닛 테스팅(Unit Testing)은 코드가 정확히 작성되었는지 확인하도록 해 줄 뿐만 아니라, 함수의 사용자가 함수로부터 어떤 기능을 기대할지를 미리 생각할 수 있게 해주는 테스트 주도 개발(TDD. Test Driving Developemnt)을 가능하게 해주어 그 중요성이 높다.

R에는 유닛 테스팅을 위한 패키지로 RUnit[15], testthat[16, 17] 등이 있으며, 이 절에서는 그 중 좀 더 사용하기 편리하다고 생각되는 testthat에 대해서 설명하도록 하겠다.

#### expect 함수들

유닛 테스팅은 작성한 코드에 어떤 입력을 주었을 때 예상되는 출력이 실제로 반환되는지 확인하는 것을 기본으로 한다. 기대값과 실제 반환값을 비교하기 위해 사용하는 함수에는 expect\_true(), expect\_false(), expect\_is(), expect\_equal() 등이 있다. 전체 함수의 목록은 참고문

현[17]을 읽어보거나 `help(expect_that)`, `help(expect_equal)`을 입력해서 도움말을 살펴보기 바란다.

`expect_equal`의 사용 방법을 피보나치 수열을 구하는 함수를 예로 들어 살펴보자. 피보나치 수열은 1, 1, 2, 3, 5, ... 와 같은 값을 반환해야하지만 다음 코드에는 일부러 버그를 만들어 두었다.

```
> library(testthat)

> fib <- function(n) {
+   if (n == 0) {
+     return(1)
+   }
+   if (n > 0) {
+     return(fib(n-1) + fib(n-2))
+   }
+ }
```

`expect_equal()` 을 사용해 `fibo(0)`의 값이 1인지 확인해보자.

```
> expect_equal(1, fib(0))
>
```

앞서 `fibo()` 함수에 버그를 만들어 두기는 했지만 피보나치 수열의 첫번째 값인 `fib(0)`을 구했을 때는 정답인 1을 제대로 반환한다. 따라서 `expect_equal`에 인자로 준 기대값 1과 `fib(0)`의 값이 같다. 따라서 아무런 경고 메시지도 출력되지 않는다.

그러나 수열의 두번째 값인 `fib(1)`을 확인해보면 버그로 인해 잘못된 값이 반환되고, 따라서 `fibo(1)`의 값이 1과 같지 않다는 에러 메시지가 출력된다.

```
> expect_equal(1, fib(1))
Error: 1 not equal to fib(1)
Numeric: lengths (0, 1) differ
```

이처럼 잘못된 값이 반환된 것을 찾았다면 이에 따라 `fibo()` 함수를 올바르게 수정해나가면 된다.

## 7.2 test-that을 사용한 테스트 그룹화

서로 관련된 내용을 테스트하는 `expect` 문장들은 `test_that()` 함수를 사용해 그룹으로 묶을 수 있다. 이 때 사용하는 형식은 `test_that('테스트에 대한 짧은 설명', { 테스트 명령들 })`을 따른다.

예를 들어 피보나치 수열의 처음 두개 값은 각각 1로 정해지는데 이를 base test라 명명해 다음과 같이 테스트 할 수 있다.

```
> test_that("base case", {
+   expect_equal(1, fib(0))
+   expect_equal(1, fib(1))
+ })
```

$\text{fib}(2)$  이상은 재귀적으로 실행되므로 이를 recursion test라고 명명해 따로 뮤을 수 있다. 다음은 버그가 없도록 수정한 `fibo()` 함수와 이를 base test와 recursion test로 나눠 테스트하는 예를 보여준다.

```
> library(testthat)

> fib <- function(n) {
+   if (n == 0 || n == 1) {
+     return(1)
+   }
+   if (n >= 2) {
+     return(fib(n-1) + fib(n-2))
+   }
+ }

> test_that("base test", {
+   expect_equal(1, fib(0))
+   expect_equal(1, fib(1))
+ })

> test_that("recursion test", {
+   expect_equal(2, fib(2))
+   expect_equal(3, fib(3))
+   expect_equal(5, fib(4))
+ })
```

테스트 중 아무런 오류도 발견되지 않았으므로 특별히 출력되는 내용은 없다.

### 7.3 테스트 파일 구조

보다 일반적인 파일구성은 비즈니스 로직이 되는 fibo() 를 하나의 소스파일에 넣고, 테스트 코드는 별도의 소스 파일에 넣는 것이다. 한가지 파일 구성의 예는 다음과 같다.

- fibo.R: fibo() 함수를 작성해 저장.
- run\_tests.R: 테스트들을 실행하기 위한 명령을 저장한 파일.
- tests/test\_fibo.R: tests 디렉토리에 fibo 함수에 대한 테스트 코드를 작성해 test\_fibo.R로 저장.

fibo.R 함수에는 다음과 같이 fibo() 함수 코드만 작성해 넣으면 된다.

```
fibo <- function(n) {
  if (n == 0 || n == 1) {
    return(1)
  }
  return(fibo(n - 1) + fibo(n - 2))
}
```

run\_tests.R은 다음과 같이 작성한다. 먼저 fibo.R을 source() 하여 해당 파일에 있는 명령을 실행한다. 그러면 그 결과로 fibo() 함수가 정의될 것이다. 테스트 수행은 test\_dir() 함수를 사용하는데, 아래 예에서는 tests 디렉토리에 있는 모든 테스트 파일들을 실행하고 그 결과의 요약을 얻게 하였다.

```
require(testthat)
source("fibo.R")

test_dir("tests", reporter="Summary")
```

tests 디렉토리의 test\_fibo.R의 첫줄에는 해당 파일내 테스트들이 무엇에 대한 테스트인지를 명시하기위해 context("fibonacci series")를 적는다. 이 예에서는 테스트 파일이 하나뿐이지만 여러 테스트 파일을 작성하게 된다면 context() 에 기록한 정보가 유용하게 사용된다. 그 뒤 test\_that() 들을 기술하는데, test\_that()은 앞서 설명한대로 base test와 recursion test의 두가지 경우로 나누어져 있다.

```
context("fibonacci series")

test_that("base test", {
```

```

expect_equal(1, fibo(0))
expect_equal(1, fibo(1))
})

test_that("recursion test", {
  expect_equal(2, fibo(2))
  expect_equal(3, fibo(3))
  expect_equal(5, fibo(4))
})

```

테스트를 위한 준비가 모두 끝났다. run\_tests.R 파일이 있는 디렉토리에서 R을 실행하고 source("run\_tests.R")을 수행하면 테스트가 실행되는 화면을 볼 수 있다. 테스트 실행시에는 context()로 지정한 “fibonacci series”라는 문자열이 출력되므로 어떤 테스트들이 수행중인지 보다 쉽게 알 수 있다.

```

> source("run_tests.R")
fibonacci series : .....

```

에러가 발생하는 경우의 출력을 살펴보기 위해 일부러 expect\_equal(0, fibo(5))를 recursion test에 추가해 다시 run\_tests.R을 실행해보자.

```

> source("run_tests.R")
fibonacci series : .....1

1. Failure: recursion test
-----
0 not equal to fibo(5)
Mean relative difference: 1

```

위 실행결과를 보면 fibonacci series 테스트 수행중 recursion test에서 에러가 발생하였고 fibo(5)의 반환값이 0이 아님을 알 수 있다.

## 7.4 디버깅

코드가 원하는대로 동작하지 않을때 그 이유를 확인하는 방법에는 크게 print(), sprint(), cat()을 사용해 메시지나 객체의 내용을 출력해보는 방법과 browser()를 사용한 코드 디버깅 방법이 있다.

**print()**

print()는 주어진 객체를 화면에 출력하는 역할을 한다. 따라서 print() 문을 코드 중간 중간에 삽입해 코드가 어떻게 동작하는지 쉽게 확인할 수 있다. 이 때 paste() 함수를 유용하게 사용할 수 있다. paste()는 인자로 주어진 값들을 하나의 문자열로 합쳐 출력한다. 다음 예를 보자.

```
> paste('a', 1, 2, 'b', 'c')
[1] "a 1 2 b c"
```

paste()는 위의 예에서 보다시피 각 인자간 공백을 넣는다. 이를 생략하고 싶다면 sep=""를 인자로 주거나 인자간 공백을 삽입하지 않는 paste0() 함수를 사용한다.

```
> paste('a', 1, 2, 'b', 'c', sep="")
[1] "a12bc"
> paste0('a', 1, 2, 'b', 'c')
[1] "a12bc"
```

다음 코드는 피보나치 수열을 구하는 함수가 잘 동작하는지 확인해보기 위하여 코드 사이사이에 print()를 넣은 예이다.

```
> fibo <- function(n) {
+   if (n == 1 || n == 2) {
+     print("base case")
+     return(1)
+   }
+   print(paste0("fibo(", n - 1, " + fibo(", n - 2, ")")))
+   return(fibo(n - 1) + fibo(n - 2))
+ }
>
> fibo(1)
[1] "base case"
[1] 1
> fibo(2)
[1] "base case"
[1] 1
> fibo(3)
[1] "fibo(2) + fibo(1)"
[1] "base case"
[1] "base case"
```

```
[1] 2
```

### sprintf()

sprintf()는 print()와 유사하지만 주어진 인자들을 특정한 규칙에 맞게 문자열로 변환해 출력한다. 형식은 sprintf("포맷팅 문자열", 인자1, 인자2, ..., 인자n)과 같은 방식이다. 이 때 포맷팅 문자열에는 주어진 인자를 어떤 방식으로 포맷팅할 것인지를 결정하는 특수한 문자열을 적을 수 있는데, 가장 중요한 포맷팅 문자열의 예는 다음과 같다.

- %d: 인자를 정수로 출력
- %f: 인자를 실수로 출력
- %s: 인자를 문자열로 출력

가장 기본적인 형태의 예는 다음과 같다. 첫번째 예는 주어진 인자를 그대로 숫자로 출력하는 예이며, 두번째 예는 'Number: ' 뒤에 정수를 출력한 예이다. 세번째 예에서는 'String: ' 뒤에 문자열을 출력했다.

```
> sprintf("%d", 123)
[1] "123"

> sprintf("Number: %d", 123)
[1] "Number: 123"

> sprintf("Number: %d, String: %s", 123, "hello")
[1] "Number: 123, String: hello"
```

%d나 %f는 %nd 또는 %.nf 형태로 적을 수 있는데 이 때 n은 1의 자리이상의 수를 몇자리 까지 출력할 것인지를, m은 소수점이하 자리의 수를 몇자리까지 출력할 것인지를 지정하는데 사용한다.

다음은 주어진 실수를 소수점 둘째자리까지만 출력하는 예이다.

```
> sprintf("%.2f", 123.456)
[1] "123.46"
```

다음은 1의 자리 이상의 수를 5자리로 고정한 예이다. 소수점이하를 지정한 경우와 달리 1의 자리 이상의 숫자가 5자리로 잘리지는 않으며, 다만 1의 자리 이상의 숫자를 5자리로의 고정폭으로 맞춰주는 역할만 한다.

```
> sprintf("%5d", 123)
```

```
[1] " 123"
> sprintf("%5d", 1234)
[1] " 1234"
> sprintf("%5d", 12345)
[1] "12345"
> sprintf("%5d", 123456)
[1] "123456"
```

이처럼 1의 자리 이상의 숫자의 자리수를 맞춰서 출력해주면 숫자가 주어진 자리수에 맞춰 우측 정렬되므로 보다 보기 좋은 출력물을 얻을 수 있다.

### cat()

print()나 sprintf()는 결과를 출력한 뒤 행바꿈이 일어난다. 반면 cat()은 주어진 입력을 출력하고 행을 바꾸지 않는다는 특징이 있다. 또한 cat()에는 여러개의 인자를 나열해 쓰면 그 인자들이 계속 연결되어 출력된다는 특징이 있다.

먼저, cat()이 행을 바꾸지 않는다는 점을 다음 코드를 통해 살펴보자.

```
> print("hi")
[1] "hi"
> cat("hi")
hi>
```

보다시피 print()가 수행되고나면 줄이 바뀌므로 R의 명령 프롬프트인 '>'가 [1] "hi"라는 결과의 다음줄에 보여졌다. 그러나 cat은 주어진 문자열을 그대로 출력하기만한다. 그런 이유로 'hi'라는 문자열 바로 뒤에 명령 표시줄이 붙어서 나타나 'hi>'가 결과로 나타났다.

cat()에서 줄 바꿈을 하려면 줄 바꿈을 뜻하는 문자열인 '\n'을 직접 출력하면 된다.

```
> cat(1, 2, 3, 4, 5, "\n")
1 2 3 4 5
```

이러한 특징때문에 cat() 을 사용해 데이터 처리가 어떻게 수행중인지를 다음과 같이 보다 보기 좋게 출력해 줄 수 있다.

```
> sum_to_ten <- function() {
+   sum <- 0
+   cat("Adding ... ")
+   for (i in 1:10) {
+     sum <- sum + i
```

```

+     cat(i, "...")
+
+     cat("Done!", "\n")
+
+   return(sum)
+
+ }
>
> sum_to_ten()
Adding ...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...Done!
[1] 55

```

`browser()`

## 8 section:browser

`browser()`가 호출되면 명령의 수행이 중지되고, 해당 시점부터 디버깅 모드가 시작된다. 다음 코드는 1에서 10까지의 합을 구하다가 `i`가 5 이상일 때 수행을 정지하고 `browser()`를 수행하는 예이다.

```

> sum_to_ten <- function() {
+
+   sum <- 0
+
+   for (i in 1:10) {
+
+     sum <- sum + i
+
+     if (i >= 5) {
+
+       browser()
+
+     }
+
+   }
+
+   return(sum)
+
+ }

```

이 코드를 실행하면 다음과 같이 ‘Browse’라는 새로운 명령 표시줄이 출력된다.

```

> sum_to_ten()
Called from: sum_to_ten()
Browse [1] >

```

이 프롬프트는 `i`가 5인 상황에서 정지된 그 환경을 그대로 갖고 있다. 따라서 `i`나 `sum`의 값을 출력해보거나 일반적인 R 명령 프롬프트에서 하는 모든 작업을 해 볼 수 있다. 다음은 `i`와 `sum` 값을 확인해보거나 `i`와 `sum`의 합을 구하는 예를 보여준다.

```
Browse [1] > i  
[1] 5  
Browse [1] > sum  
[1] 15  
Browse [1] > sum + i  
[1] 20
```

Browse 프롬프트에서는 c(ontinue) 또는 n(ext) 명령을 사용할 수 있다. c를 입력하면 다음 browser()가 호출될 때 까지 코드의 수행을 재개한다. 이 예에서는 i가 5이상일 때 매번 browser()가 호출되게 하였으므로 i가 5일때 정지된 상태에서 c를 입력하면 i가 6일때 다시 browser()로 진입한다.

```
Browse [1] > c  
Called from: sum_to_ten()  
Browse [1] > i  
[1] 6  
Browse [1] > sum  
[1] 21
```

n을 입력하면 browser()로 정지된 그 다음부터 코드를 하나씩 차례로 수행한다. n을 입력할 때마다 현재 수행중인 문장을 출력하여 보여주므로 어느 코드가 수행되었는지를 알 수 있다.

```
Browse [1] > n  
debug at #3: i  
Browse [2] > n  
debug at #4: sum <- sum + i  
Browse [2] > n  
debug at #5: if (i >= 5) {  
    browser()  
}  
Browse [2] > n  
debug at #6: browser()  
Browse [2] > n  
Browse [2] > n  
debug at #3: i  
Browse [2] > n  
debug at #4: sum <- sum + i
```

```

Browse [2] > n
debug at #5: if (i >= 5) {
    browser()
}

Browse [2] > n
debug at #6: browser()
Browse [1] > n
debug at #3: i
Browse [2] > n
debug at #4: sum <- sum + i
Browse [2] > n
debug at #5: if (i >= 5) {
    browser()
}
Browse [2] > n
debug at #6: browser()
Browse [2] > n
Browse [2] > n
debug at #3: i
Browse [2] > n
debug at #4: sum <- sum + i
Browse [2] > n
debug at #5: if (i >= 5) {
    browser()
}
Browse [2] > n
debug at #6: browser()

```

browser()의 수행을 마치려면 Q(uit)를 입력한다.

```

Browse [2] > Q
>

```

지금까지 디버깅을 위한 여러가지 방법에 대해 살펴보았다. 그러나 이를 방법에 의존하기보다는 [testthat](#) (페이지 157)과 같은 유닛 테스팅을 자주 또는 먼저 작성하여 보다 다양한 테스트를 수행하기를 권한다. 테스트를 작성하면 사전에 예상하지 못한 다양한 입력에 대해 코드를 검증해 볼 수 있으며, 코드를 수정할 때 새로운 버그가 생기지 않았음을 보다 쉽게 확인할 수 있기 때문이다.

## 9 코드 수행 시간 측정

코드 성능을 평가하는 방법은 system.time()을 사용해 간단히 함수의 수행 시간을 출력해보는 방법과 Rprof() 함수를 사용해 보다 본격적인 보고서를 출력해보는 방법이 있다.

### 9.1 system.time()을 사용한 시간 측정

system.time()은 인자로 주어진 명령이 수행되는데 걸린 시간을 측정한다. 다음은 1부터 N까지 더하는 함수인 sum\_to\_n()의 수행시간을 N=10,000, N=100,000, N=1,000,000인 경우에 대해 각각 측정해본 예이다.

```
> sum_to_n <- function(n) {
+   sum <- 0
+   for (i in 1:n) {
+     sum <- sum + i
+   }
+   return(sum)
+ }

>

> system.time(sum_to_n(10000))
  user  system elapsed
 0.004    0.000    0.003

> system.time(sum_to_n(100000))
  user  system elapsed
 0.041    0.000    0.042

> system.time(sum_to_n(1000000))
  user  system elapsed
 0.424    0.002    0.428
```

수행시간은 user time, system time, elapsed time으로 구분된다. 이 중 elapsed time이 가장 알기 쉬운 개념이다. 이 시간은 코드의 총 소요시간으로 코드를 시작하자마자부터 코드 수행이 끝날때까지의 시간을 초시계로 쟀을때 얼마만큼의 시간이 걸렸는지를 나타낸다. 통상적으로 이야기하는 코드 수행시간이 이에 해당한다.

user time, system time은 elapsed time을 구성하는 요소로서 각각 프로그램 코드 자체를 수행하는데 걸린시간과, 프로그램이 운영체제의 명령을 호출하였다면 그 때 운영체제가 명령을 수행하는데 걸린 시간을 의미한다.

프로그램이 운영체제의 명령을 호출하는 대표적인 예로는 파일 입출력이 있다. `save()`, `load()` (페이지 88)에서 살펴본 파일 입출력 명령을 사용해 큰 파일을 저장 및 로딩해보고 이 때 걸리는 시간을 측정해보자.

```
> x <- matrix(1:(10000*10000), ncol=10000)
> system.time(save(x, file="x.RData"))
  user  system elapsed
 56.832   0.378  57.377
> system.time(load(file="x.RData"))
  user  system elapsed
 2.560   0.132   2.692
```

위 결과를 보면 1부터 N까지의 숫자를 합하는 함수의 경우와 달리 `system.time`이 크게 나타난 것을 볼 수 있다. 이는 파일 입출력을 위해 R 환경이 운영체제의 기능을 많이 호출해 사용하기에 발생한 현상이다.

`system.time`은 코드를 수행하는 머신의 메모리가 부족해 운영체제가 디스크의 일부를 메모리로 사용하는 페이징이 빈번히 발생할 경우에도 증가할 수 있다. 따라서 `system.time`이 크다면 메모리가 혹시 부족하지는 않은지 운영체제의 작업관리자를 통해 잘 살펴볼 필요가 있다.

이처럼 코드의 수행시간을 측정할 수 있으면 다양한 구현 방법중 어떤 방법이 더 나은 성능을 보이는지 평가할 수 있다. 한가지 예로 `rbindlist` (페이지 148)에서는 `rbind()`보다 `rbindlist()`가 성능이 더 나음을 `system.time()` 함수를 사용해 보인바 있다.

## 9.2 Rprof()를 사용한 코드 프로파일링

`Rprof()`는 보다 본격적인 코드 수행 성능 평가를 위한 함수이다. ‘prof’는 Profiling을 의미하는데 프로그램의 동적인 성능, 즉 메모리나 CPU 사용량을 평가하는 작업을 소프트웨어 공학에서는 흔히 코드 프로파일링(Profiling)이라 부른다.

코드 파일링은 `Rprof()` 함수 호출로 시작한다. `Rprof()` 호출 이후에는 특정 시간 간격마다 현재 어떤 함수가 수행 중인지 샘플이 추출되어 파일에 저장된다. 파일에 기록된 결과는 이후 `summaryRprof()` 함수로 분석할 수 있다.

실제 예를 살펴보자. 코드 프로파일링을 시작하려면 `Rprof("샘플을 저장할 파일명")` 명령을 수행하고 종료하려면 `Rprof(NULL)`을 수행한다. 다음은 주어진 벡터에 1을 더하는 비효율적인 함수에 대해 코드 프로파일링을 수행하는 예를 보여준다. 코드에서 `seq.along()`은 인자로 주어진 벡터의 길이만큼 1, 2, 3, ..., N으로 구성된 숫자 벡터를 생성하는 함수이다.

```
> add_one <- function(val) {
+   return(val + 1)
```

```

+ }
>

> add_one_to_vec <- function(x) {
+   for (i in seq_along(x)) {
+     x[i] <- add_one(x[i])
+   }
+   return(x)
+ }

> Rprof("add_one.out")
> add_one_to_vec(1:1000000)
...
> Rprof(NULL)

```

위 코드의 실행 결과 add\_one.out이라는 파일이 생성된다. 결과의 분석은 summaryRprof()에 Rprof()가 생성한 파일을 넘겨주면 된다.

```

> summaryRprof("add_one.out")
$by.self
               self.time self.pct total.time total.pct
"add_one_to_vec"      1.62    62.79      2.58    100.00
"add_one"            0.82    31.78      0.96     37.21
"+"
```

	total.time	total.pct	self.time	self.pct
"add_one_to_vec"	2.58	100.00	1.62	62.79
"add_one"	0.96	37.21	0.82	31.78
"+"	0.14	5.43	0.14	5.43

  
\$sample.interval
[1] 0.02
  
\$sampling.time
[1] 2.58

summaryRprof()의 분석결과는 크게 두개 섹션 by.self와 by.total로 나뉘는데, 실은 이 두 섹션의 내용은 동일하다. 다만 by.self는 self.time으로 정렬된 표이고 by.total은 total.time으로 정렬된 표이다. 따라서 설명의 편의를 위해 여기서는 by.self를 기준으로 살펴보기로 하자.

```
> summaryRprof("add_one.out")$by.self
      self.time self.pct total.time total.pct
"add_one_to_vec"     1.62    62.79      2.58   100.00
"add_one"            0.82    31.78      0.96    37.21
"+"
```

표에서 self.time은 각 함수가 수행되는데 걸린시간이다. add\_one\_to\_vec은 함수내 코드를 수행하는데 1.62초가 소요되었으며 add\_one은 해당 함수내 코드를 수행하는데 0.82초가 소요되었다. self.time 시간의 비율은 self.pct를 통해 알 수 있다.

total.time은 각 함수내 코드를 수행하는데 걸린시간과 해당함수가 호출한 함수를 수행하는데 걸린 시간의 합이다. 예를들어 ‘+’ 연산은 self.time 이 0.14 초였다. ‘+’ 연산자 자체는 단독으로 수행되므로 total.time도 0.14초이다. 반면 add\_one 함수는 해당함수 내에서 val + 1 명령을 수행하느라 ‘+’를 호출하고 있다. 따라서 add\_one의 total.time은 자신의 self.time 0.82 초에 ‘+’의 self.time 0.14 초를 더한 0.96초이다. 마찬가지로 add\_one\_to\_vec은 해당 함수내 코드를 수행하는데 1.62초, add\_one을 호출하면서 0.82초, ‘+’을 호출하면서 0.14초를 소모하므로 total.time이 2.58초이다.

이처럼 summaryRprof()의 결과는 어떤 함수가 수행되는데 얼마만큼의 시간이 소모되는지를 보다 쉽고 분석적으로 나열해주므로 내 코드내 어떤 함수에서 가장 긴 시간이 소모되는지를 판단할 수 있다. 만약 R 코드의 성능이 만족스럽지 못하다면 Rprof()를 사용해 소모시간과 그 비율을 판단하고 가장 긴 시간이 걸리는 항목부터 개선해나감으로써 보다 과학적인 성능 개선을 이룰 수 있다.

Rprof()에는 여기에서 설명하지 않은 메모리 프로파일링 등의 다양한 옵션이 있으므로 코드 성능 평가시 help(Rprof)를 살펴보기 바란다.

## 그래프

R은 다양한 그래프 기능을 지원한다. 비록 처음 접할때는 그래프를 그릴때 사용하는 파라미터 이름등이 생소해서 어렵게 보이지만, 매번 사용해야하는 옵션의 갯수는 많지 않으므로 몇번 접하다 보면 쉽게 익힐 수 있다.

R에서 사용되는 그래픽스 기능으로는 크게 graphics, lattice, ggplot 패키지를 들 수 있다. 이번 장에서는 이들중 graphics 패키지에 대해서 살펴본다.

graphics 패키지는 R의 가장 기본이 되는 시각화 기능을 지원한다. 전체 함수의 목록은 library(help = 'graphics')를 통해 볼 수 있으며 여기서는 가장 자주 사용되는 함수들에 대해서 설명한다.

### 1 산점도

산점도는 주어진 데이터를 점으로 표시해 흩뿌리듯이 시각화한 그림이다. R에서 산점도는 plot() 함수로 그리는데, plot()은 산점도 뿐만 아니라 일반적으로 객체를 시각화하는데 모두 사용될 수 있는 일반 함수(Generic Function)이다. 여기서 일반함수란 주어진 데이터 타입에 따라 다른 종류의 plot() 함수의 변형이 호출됨을 뜻한다. plot()이 어떤 객체들을 그려줄 수 있는지는 다음 명령으로 볼 수 있다.

```
> methods("plot")
[1] plot.aareg*          plot.acf*           plot.cld*
[4] plot.confint.glm*    plot.correspondence* plot.cox.zph*
[7] plot.data.frame*     plot.decomposed.ts*   plot.default
[10] plot.dendrogram*    plot.density        plot.ecdf
[13] plot.factor*         plot.formula*       plot.function
[16] plot.glm*            plot.hclust*        plot.histogram*
[19] plot.HoltWinters*   plot.isoreg*        plot lda*
[22] plot.lm              plot.mca*          plot.medpolish*
```

```
[25] plot.mlm          plot.ppr*        plot.prcomp*
[28] plot.princomp*    plot.profile*    plot.profile.nls*
[31] plot.ridgelm*     plot.shingle*    plot.spec
[34] plot.spline*       plot.stepfun    plot.stl*
[37] plot.survfit*     plot.table*      plot.trellis*
[40] plot.ts            plot.tskernel*   plot.TukeyHSD
[43] plot.xyVector*
```

Non-visible functions are asterisked

예를들어 `plot.lm` 은 `lm`이라는 클래스에 정의된 `plot` 메소드로서 `plot(lm 객체)` 와 같은 방식으로 호출하면 자동으로 `lm` 클래스의 `plot`이 불려지게된다. 객체 지향에 대한 좀 더 자세한 내용은 차후에 다루기로 하고 여기서는 `plot()`이 주어진 객체에따라 다르게 처리된다는 정도만 알고 넘어가기로 한다.

`plot()` 함수를 사용한 가장 빈번한 예는 산점도(scatter plot)을 그리는 것이다. `mlbench` 패키지<sup>1)</sup>에 있는 `Ozone`데이터를 사용해 산점도를 그려보자.

```
> install.packages("mlbench")
> library(mlbench)
> data(Ozone)
> plot(Ozone$V8, Ozone$V9)
```

위 코드에서 `data()` 문은 `mlbench` 패키지 로딩뒤 `Ozone`데이터 셋을 읽어들이기 위해 사용한 명령이다. `mlbench` 패키지 뿐만 아니라 통상적으로 많은 통계 또는 머신 러닝 패키지들이 이러한 데이터 셋을 데모 목적으로 가지고 있으며, 각 데이터는 `data()`문을 통해 읽어들일 수 있다.

`Ozone`의 V8과 V9변수는 각각 캘리포니아 Sandburg와 El Monte에서 매일 측정한 온도이다. `Ozone` 데이터에 들어있는 각 필드의 설명은 `?Ozone` 또는 `help(Ozone)`명령으로 볼 수 있다.

마지막으로 `mlbench`에 포함된 전체 데이터는 `library(help = "mlbench")` 명령으로 살펴볼 수 있다. 또는 `mlbench` 의 reference manual 을 찾아서 읽어봐도 되는데, 보통 ‘R mlbench’ 정도의 키워드를 구글에서 찾으면 손쉽게 패키지 참조 문헌을 찾아 볼 수 있다.

위 프로그램의 실행 결과는 그림 7.1에 보였다. 그림에서 볼 수 있듯이 `plot`은 (x, y)의 순서로 입력을 받으며, x와 y가 숫자형 데이터의 경우 산점도를 그려준다.

<sup>1)</sup>UCI repository를 포함한 다양한 머신 러닝 벤치 마킹을 위한 데이터가 있는 패키지이다.

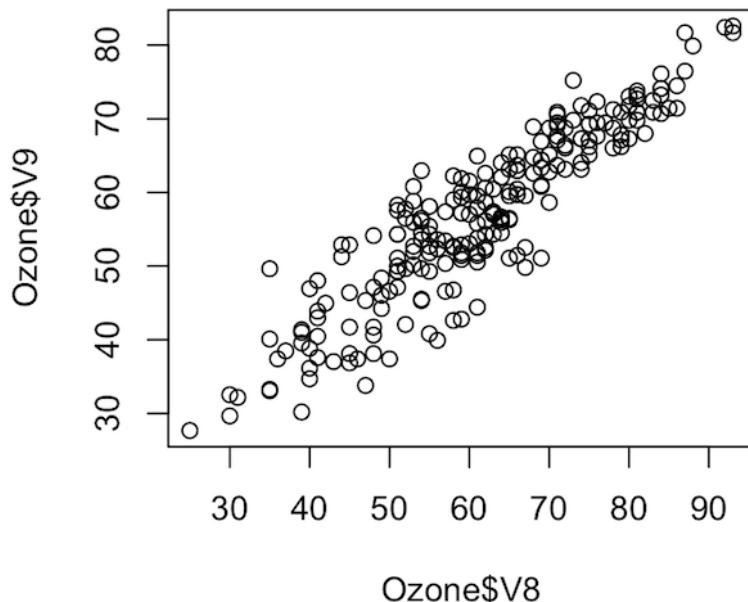


그림 7.1: Sandburg(V8)와 El Monte(V9)지역의 온도

## 2 그라프 옵션

`plot()` 과 같은 그래픽 함수들은 여러가지 파라미터들로 그 모양을 다듬을 수 있다. 이번 절에서는 그러한 파라미터들 중 자주 사용되는 옵션에 대해서 설명한다. 보다 완전한 리스트는 `?par` 또는 `help(par)`를 입력하여 볼 수 있다.

### 2.1 축 이름(xlab, ylab)

그림 7.1을 볼때 가장 먼저 눈에 띄는 점은 x축, y축의 레이블이 컬럼명이므로 그 의미를 잘 설명해주지 못한다는 것이다. 이를 해결하기위해 `xlab`, `ylab`으로 축 이름을 지정해보자.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature")
```

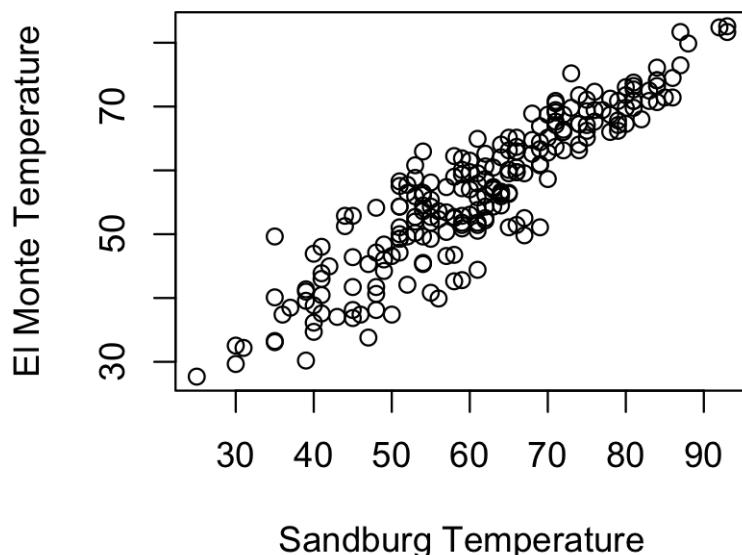


그림 7.2: xlab, ylab의 사용

## 2.2 그래프 제목(main)

이번에는 제목을 붙여보자. 제목은 main 파라미터로 지정한다.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone")
```

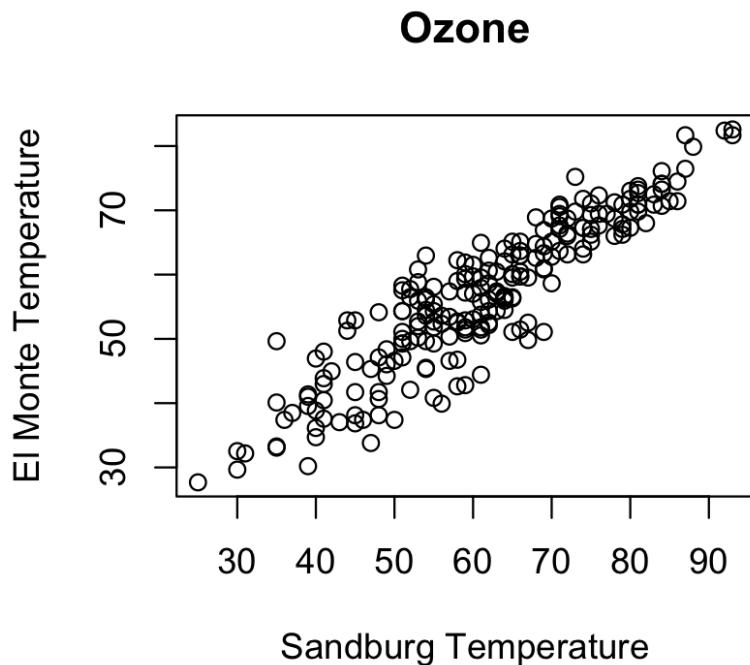


그림 7.3: main을 사용한 타이틀 지정

### 2.3 점의 종류(pch)

그래프에 보이는 점은 모양은 pch로 지정하는데 pch에 숫자를 지정하면 미리 지정된 심볼이 사용되고 문자(예를 들어 '+')를 지정하면 그 문자를 사용해 점을 표시한다.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone", pch=20)
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone", pch="+")
```

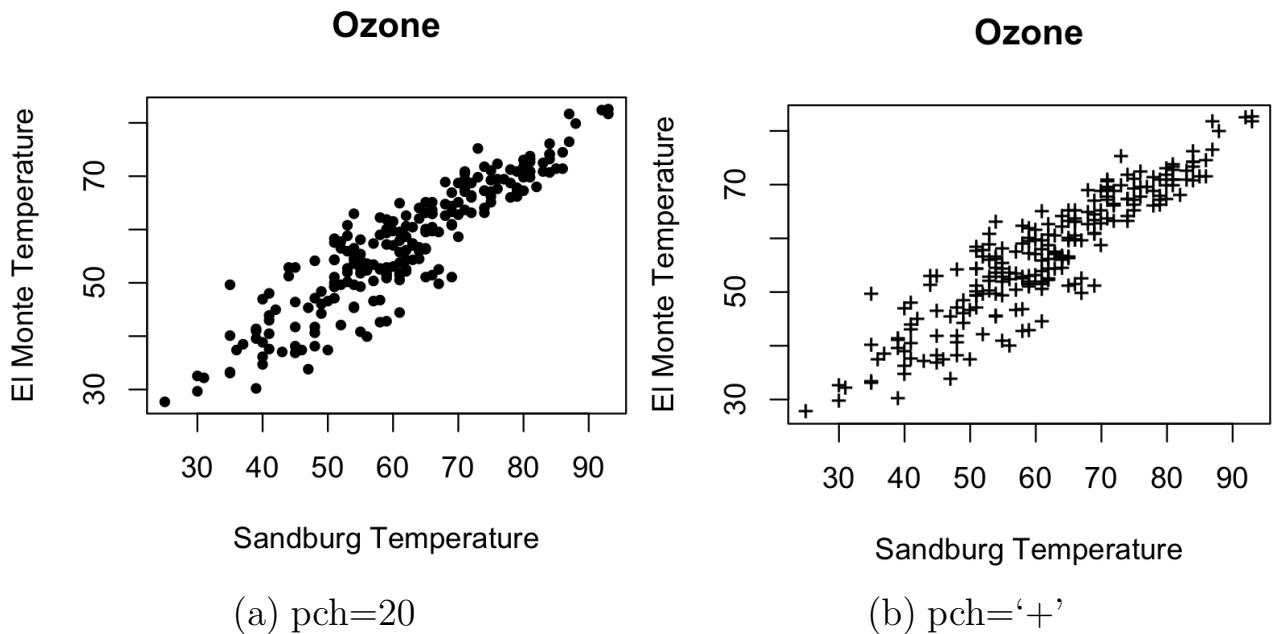


그림 7.4: pch를 사용한 점 모양의 지정

pch에 보여줄 수 있는 심볼의 목록은 구글에서 ‘r pch symbols’ 정도의 검색어를 사용하면 쉽게 찾아볼 수 있다.

## 2.4 점의 크기(cex)

산점도에 보인 점의 크기는 cex로 조정할 수 있다. 다음 코드는 cex=0.1을 주어 점의 크기를 작게 만들었다.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone", cex=.1)
```

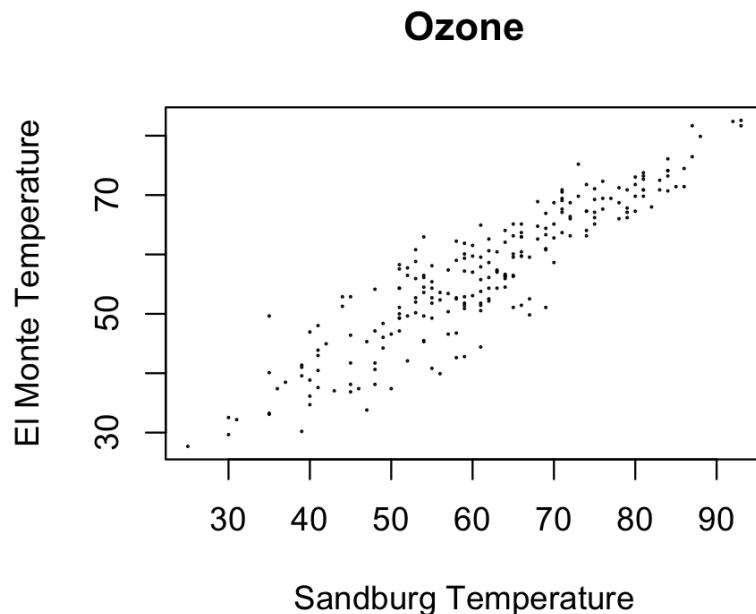


그림 7.5: cex=0.1을 사용한 점의 크기 조정

## 2.5 색상(col)

색상은 col 파라미터로 RGB값을 각각 두자리씩 지정한다. 다음 코드에서는 col="#FF0000"으로 빨간색 점을 지정했으나 col="red"처럼 색상 명칭을 사용해도 된다. 전체 색상 목록은 colors() 명령으로 볼 수 있다.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone", col="#FF0000")
```

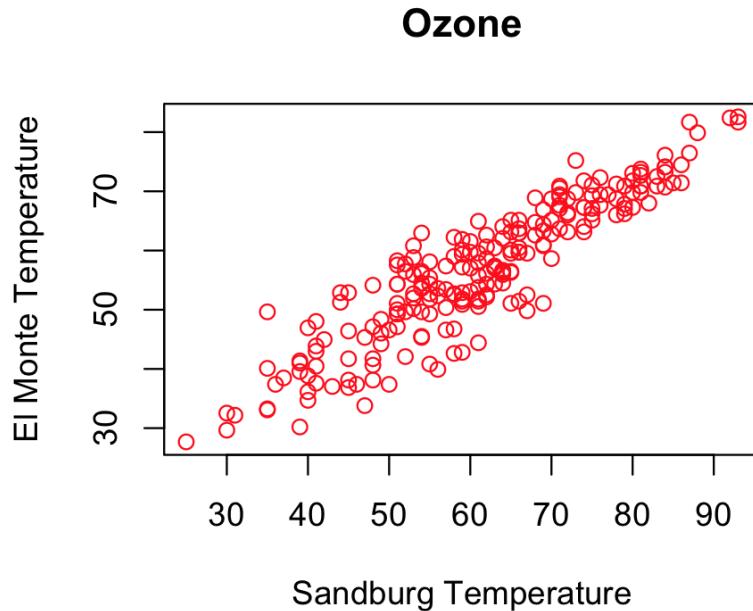


그림 7.6: col='FF0000'을 사용한 점 색상의 지정

이외에도 col.axis, col.lab 등의 옵션으로 그래프의 여러 부분에 색상 지정이 가능하다. ?par 명령으로 확인해보기 바란다.

## 2.6 좌표축 값의 범위(xlim, ylim)

필요하다면 그래프에 그려질 x 값의 범위, y값의 범위를 바꿔볼 수 있다. x축과 y축 각각 xlim, ylim을 사용하여 c(최소값, 최대값)의 형태로 각 인자에 값을 지정하면 된다. 다음 코드에서 보다시피 Ozone\$V8과 Ozone\$V9에는 NA값이 있다. 따라서 최대 값을 구할때 na.rm=TRUE를 사용했다. 그 뒤 적당한 값으로 x축과 y축의 값을 지정해보았다.

```
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone")
> max(Ozone$V8)
[1] NA
> max(Ozone$V8, na.rm=TRUE)
[1] 93
> max(Ozone$V9, na.rm=TRUE)
[1] 82.58
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone",
+       xlim=c(0, 100), ylim=c(0, 90))
```

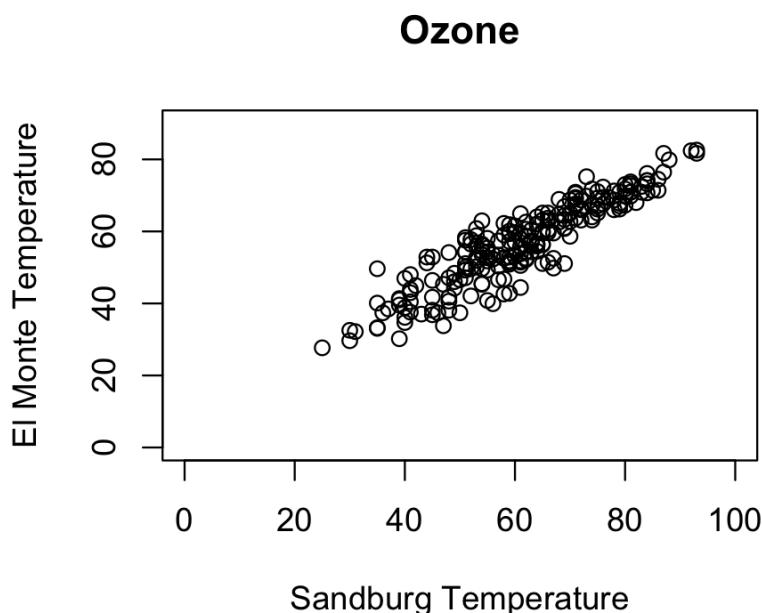


그림 7.7: xlim, ylim의 사용

## 2.7 type

type을 설명하기 위해 잠깐 cars 데이터셋에 알아보자. cars 데이터는 차량이 달리던 속도, 그리고 그 속도에서 브레이크를 잡았을 때 제동거리를 측정한 데이터이다.

```
> data(cars)
> str(cars)
'data.frame': 50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
> head(cars)
   speed dist
1      4     2
2      4    10
3      7     4
4      7    22
5      8    16
6      9    10
> plot(cars)
```

cars 데이터를 plot()한 결과는 다음과 같다.

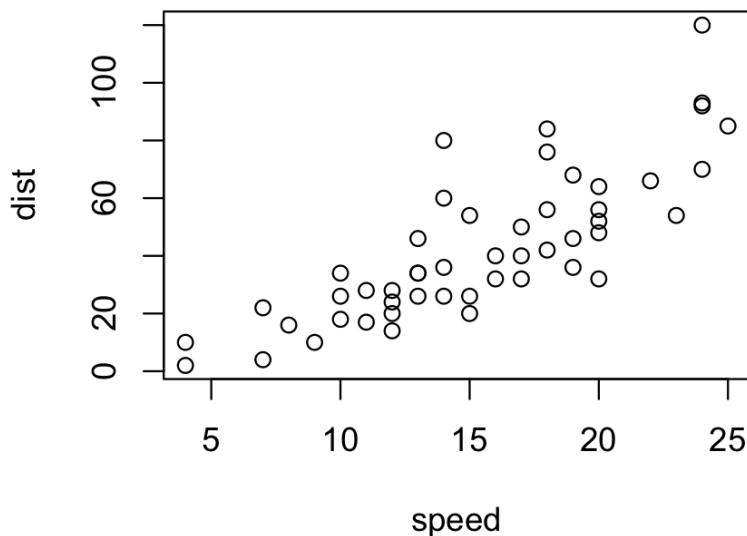


그림 7.8: plot(cars)

그런데 이처럼 속도와 거리가 있는 데이터에서는 점으로 데이터를 표시하는 것보다는 선으로 표시하는 것이 낫지 않을까? 이런 경우에는 type="l"을 지정하여 라인 그래프를 그릴 수 있다.

```
> plot(cars, type="l")
```

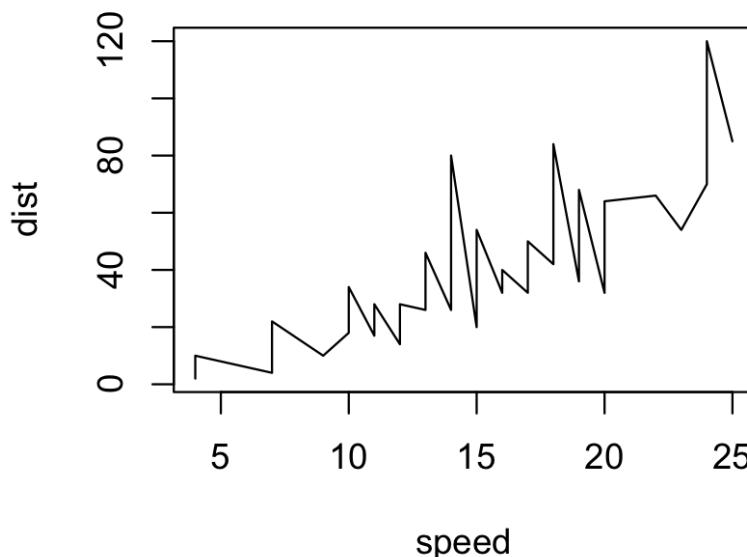


그림 7.9: plot(cars, type="l")

관찰된 점들과 선을 중첩하여(overlapped) 그리려면 type="o"를 지정한다. cex=0.5는 점의 크기를 조금 작게 그리기 위해서 사용했다.

```
> plot(cars, type="o", cex=0.5)
```

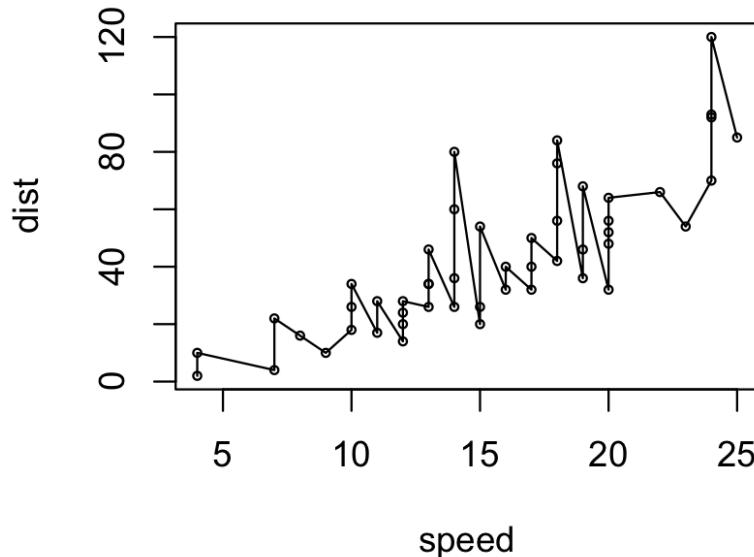


그림 7.10: plot(cars, type="o")

그런데 그림 7.10에는 같은 주행속도에 대해 두개 이상의 제동거리가 있는 경우가 많아 어색해보인다. 이 문제를 해결하기 위해 tapply (페이지 97)를 사용해보자. 각 speed마다 평균 dist를 tapply를 사용해 계산한 다음, 이를 plot()하면 된다.

```
> tapply(cars$dist, cars$speed, mean)
    4      7      8      9      10     11      12
6.00000 13.00000 16.00000 10.00000 26.00000 22.50000 21.50000
    13     14     15     16     17     18     19
35.00000 50.50000 33.33333 36.00000 40.66667 64.50000 50.00000
    20     22     23     24     25
50.40000 66.00000 54.00000 93.75000 85.00000
> plot(tapply(cars$dist, cars$speed, mean), type="o", cex=0.5,
+       xlab="speed", ylab="dist")
```

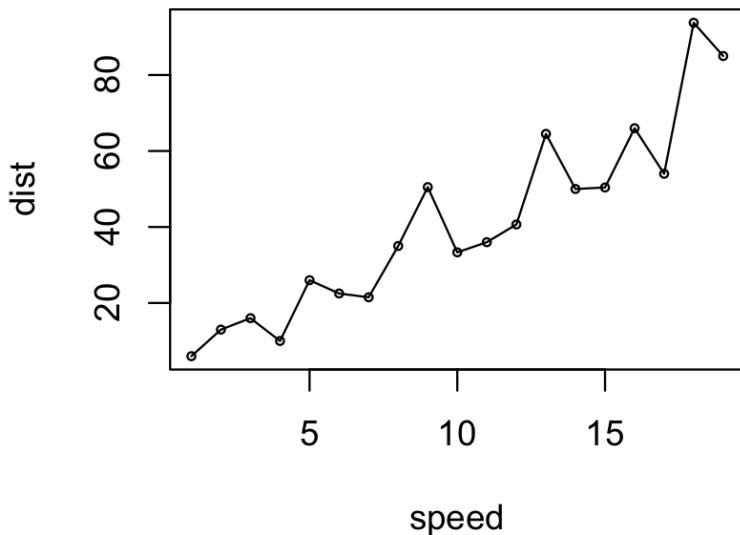


그림 7.11: 주행속도별 평균 제동거리

type에는 이외에도 다양한 옵션이 있으므로 ?par 명령을 통해 살펴보기 바란다.

### 3 그래프의 배열(mfrow)

plot() 명령으로 그래프를 그리면 매번 새로운 창이 뜨면서 그래프가 그려진다. 그러나 mfrow를 지정하면 한 창에 여러개의 그래프를 나열할 수 있다. mfrow를 지정하는 형식은 par(mfrow = c(nr, nc))이며 nr은 행의 수, nc는 열의 수를 뜻한다. 다음에 보인 코드에서는 mfrow=c(1, 2)를 지정하여 한 창에 그래프를 1행 2열로 배치하였다.

par() 문에 mfrow를 지정하면 이전에 저장된 par 설정이 반환된다. 코드의 마지막에서 par(opar)를 수행함으로써 mfrow 지정 이전의 par 설정으로 되돌렸다. 만약 이러한 설정을 하지 않는다면 다음에 그리는 plot() 명령에 의해 그려지는 그래프들도 계속해서 1행 2열로 그려지게 된다. Ozone 데이터를 사용해 이 기능을 알아보자.

```
> opar <- par(mfrow=c(1, 2))
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone")
> plot(Ozone$V8, Ozone$V9, xlab="Sandburg Temperature",
+       ylab="El Monte Temperature", main="Ozone2")
> par(opar)
```

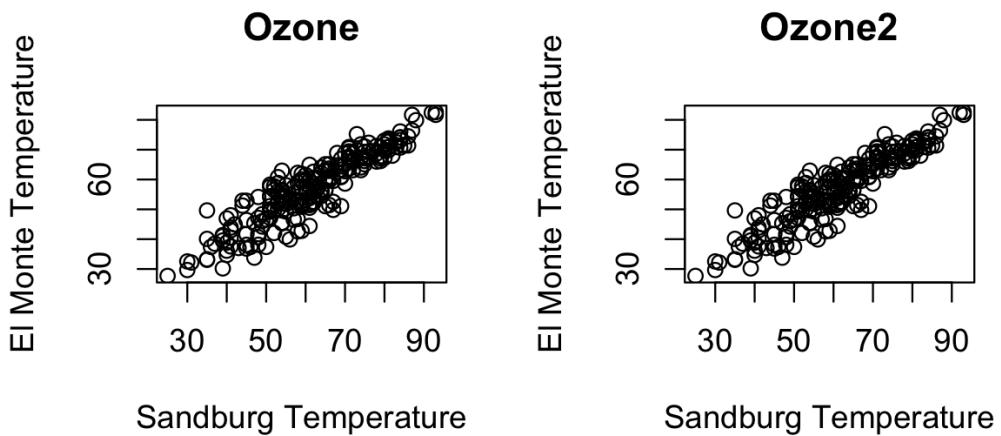


그림 7.12: par(mfrow=c(1, 2))

## 4 지터(jitter)

Ozone데이터의 V6와 V7은 각각 LAX에서의 풍속과 습도를 담고 있다. 그런데 이 둘은 자연수로 표시되므로 값이 같은 경우가 많다. 아래에 보인 것처럼 몇개의 데이터만 살펴봐도 풍속 4인 경우가 3행과 6행, 풍속 3인 경우가 4행과 5행에서 관찰된다.

```
> head(Ozone)
   V1 V2 V3 V4   V5 V6 V7 V8     V9 V10 V11 V12 V13
1  1  1  4  3 5480  8 20 NA    NA 5000 -15 30.56 200
2  1  2  5  3 5660  6 NA 38    NA    NA -14      NA 300
3  1  3  6  3 5710  4 28 40    NA 2693 -25 47.66 250
4  1  4  7  5 5700  3 37 45    NA  590 -24 55.04 100
5  1  5  1  5 5760  3 51 54 45.32 1450  25 57.02  60
6  1  6  2  6 5720  4 69 35 49.64 1568  15 53.78  60
```

이와 같은 경우에 (V6, V7)의 순서쌍을 좌표평면에 도시하면 여러 점들이 한 위치에 표시되어 서로 구분이 잘 되지 않는다. Jitter는 데이터 값을 조금씩 움직여서 같은 점에 데이터가 여러번 겹쳐서 표시되는 현상을 막는다.

다음 코드는 원본 데이터와 jitter를 사용한 경우를 각각 그리는 코드이다. 그림 7.13에서 볼 수 있듯이 jitter를 사용한 경우에 데이터가 몰리는 점을 더 쉽게 파악할 수 있다.

```
> plot(Ozone$V6, Ozone$V7, xlab="Windspeed", ylab="Humidity",
+       main="Ozone", pch=20, cex=.5)
> plot(jitter(Ozone$V6), jitter(Ozone$V7),
```

```
+     xlab="Windspeed", ylab="Humidity", main="Ozone",
+     pch=20, cex=.5)
```

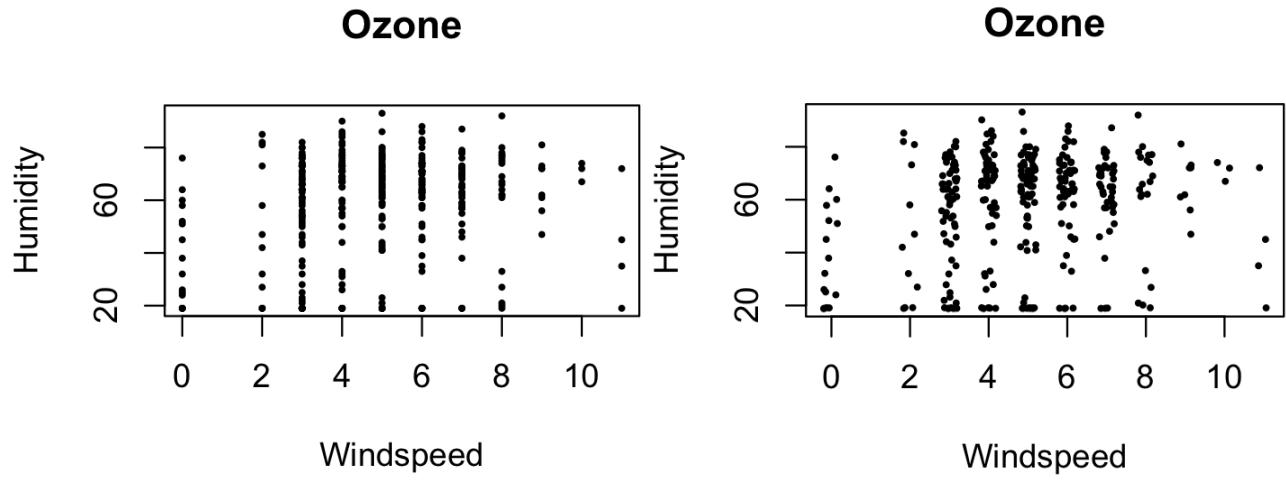


그림 7.13: jitter()

## 5 점(points)

points()는 점을 그리기 위한 함수이다. plot()을 연달아 호출하는 경우 매번 새로운 그래프가 그려지는 것과 달리 points()는 이미 생성된 plot에 점을 추가로 그려준다. 다음은 iris의 Sepal.Width, Sepal.Length을 plot으로 그린 뒤 Petal.Width, Petal.Length을 같은 그래프 위에 덧 그리는 코드이다.

```
> plot(iris$Sepal.Width, iris$Sepal.Length, cex=.5, pch=20,
+       xlab="width", ylab="length", main="iris")
> points(iris$Petal.Width, iris$Petal.Length, cex=.5,
+         pch="+", col="#FF0000")
```

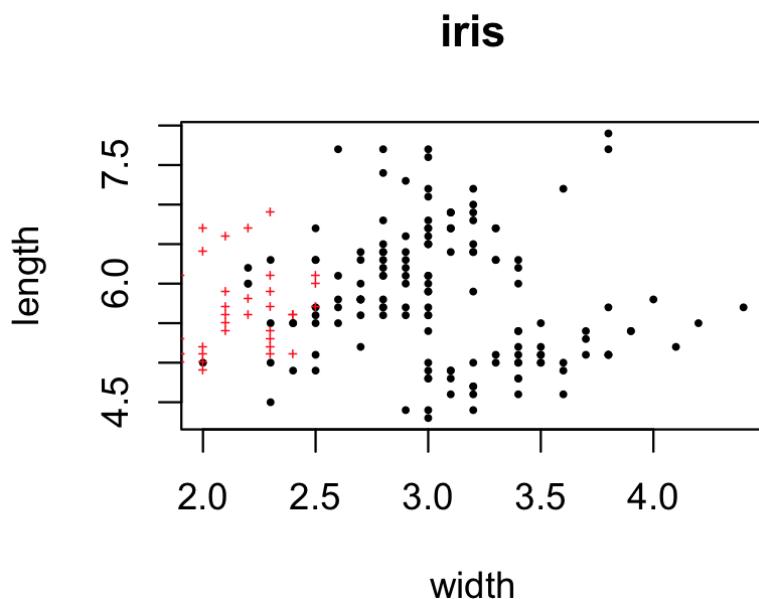


그림 7.14: points()

위의 코드처럼 iris가 연달아 나타날때는 [attach\(\), detach\(\)](#) (페이지 112)에서 본 바와 같이 attach()로 데이터를 불러들인 뒤 필드를 곧바로 접근할 수도 있다.

```
> attach(iris)
> plot(Sepal.Width, Sepal.Length, cex=.5, pch=20,
+       xlab="width", ylab="length", main="iris")
> points(Petal.Width, Petal.Length, cex=.5,
+         pch="+", col="#FF0000")
```

이 방법의 단점이라면 attach()한 데이터를 detach()하지 않을경우 Sepal.Length 등이 계속 접근 가능하게 남아있게 된다는 것이다. 만약 위 두 명령뒤에 이들 데이터를 다시 사용하지 않을 계획이라면 [with\(\), within\(\)](#) (페이지 109)에서 본바와 같이 with문을 써서 iris 데이터의 접근 범위를 보다 명시적으로 제한하여 코딩할 수도 있다.

```
> with(iris, {
+   plot(Sepal.Width, Sepal.Length, cex=.5, pch=20,
+         xlab="width", ylab="length", main="iris")
+   points(Petal.Width, Petal.Length, cex=.5,
+         pch="+", col="#FF0000")
+ })
```

`points()`는 이처럼 이미 그려진 plot에 추가로 점을 표시할 수 있다. 그런데 때에 따라서는 제일 처음 `plot()` 문을 수행할 때는 그래프에 표시할 데이터가 없다가, 이후 `points()` 명령을 수행할 시점에 표시할 데이터가 준비될 경우가 있다. 이럴때는 `type="n"`을 사용하여 `plot()`을 수행한다. 그러면 화면에 그려지는 데이터는 없으나 새로운 plot을 시작하여 `points()` 호출이 가능해진다. 다음은 앞서 보인 (`Sepal.Width`, `Sepal.Length`), (`Petal.Width`, `Petal.Length`)를 두 개의 `points()` 호출로 그리는 예이다.

```
> with(iris, {
+   plot(NULL, xlim=c(0, 5), ylim=c(0, 10),
+         xlab="width", ylab="length", main="iris", type="n")
+   points(Sepal.Width, Sepal.Length, cex=.5, pch=20)
+   points(Petal.Width, Petal.Length, cex=.5, pch="+", col="#FF0000")
+ })
```

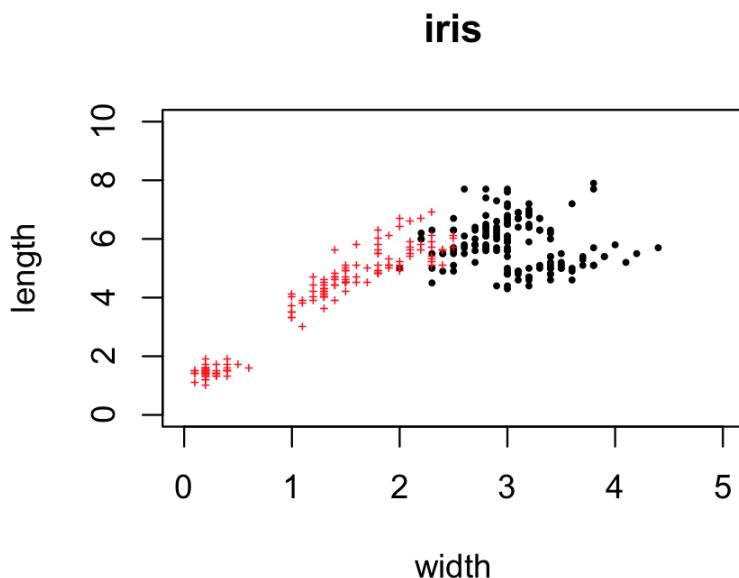


그림 7.15: `plot(type="n")`

이처럼 `type="n"`을 사용하여 점진적인 방법으로 그래프를 그려나갈 수 있다. 그러나 `xlim`과 `ylim`을 적절하게 설정해줘야하는 번거로움이 있다.

## 6 선(lines)

`lines()`는 `points()`와 마찬가지로 `plot()`으로 새로운 그래프를 그린 뒤 선을 그리는 목적으로 사용된다. 다음은  $[0, 2\pi]$ 까지  $\sin$  그래프를 그리는 예이다.

```
> x <- seq(0, 2*pi, 0.1)
> y <- sin(x)
> plot(x, y, cex=.5, col="red")
> lines(x, y)
```

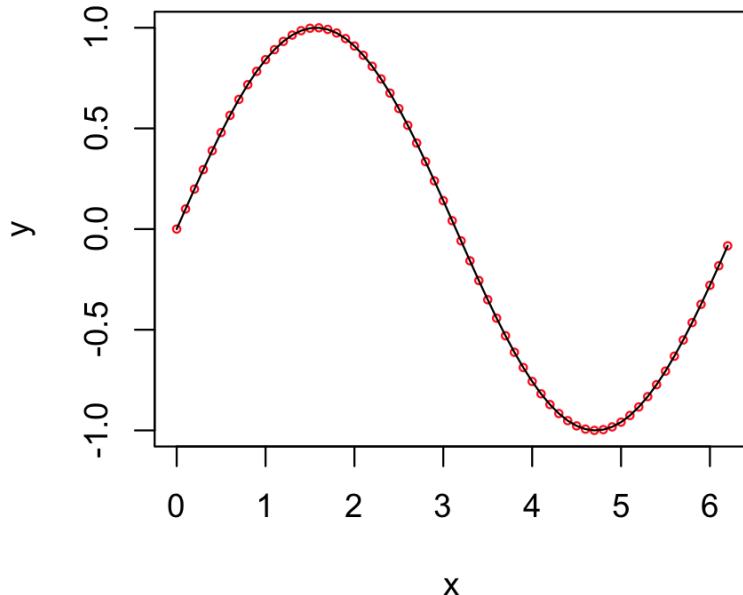


그림 7.16: lines()

example(lines)를 입력하면 cars 데이터에 LOWESS를 적용한 예를 볼 수 있다. LOWESS는 데이터의 각 점에서 linear model( $y = ax + b$ ) 또는 quadratic model( $y = ax^2 + bx + c$ )을 각각 적합하되, 각 점에서 가까운 데이터에 많은 weight를 주면서 regression을 수행한다. 이렇게 만들어진 결과물은 자료의 추세를 보여주는 선이된다. 자세한 내용은 참고자료[18, 19]를 보기 바란다.

cars 데이터에 대해 LOWESS를 수행해보자. cars 데이터는 자동차의 속도와 그 속도에서의 제동거리를 담고 있는 데이터이다.

```
> library(mlbench)
> data(cars)
> head(cars)
  speed dist
1      4    2
2      4   10
3      7    4
```

```

4      7     22
5      8     16
6      9     10

```

각각의 관찰값을 점으로 표시하고, LOWESS를 수행한 결과를 보이는 코드는 다음과 같다.

```

> plot(cars)
> lines(lowess(cars))

```

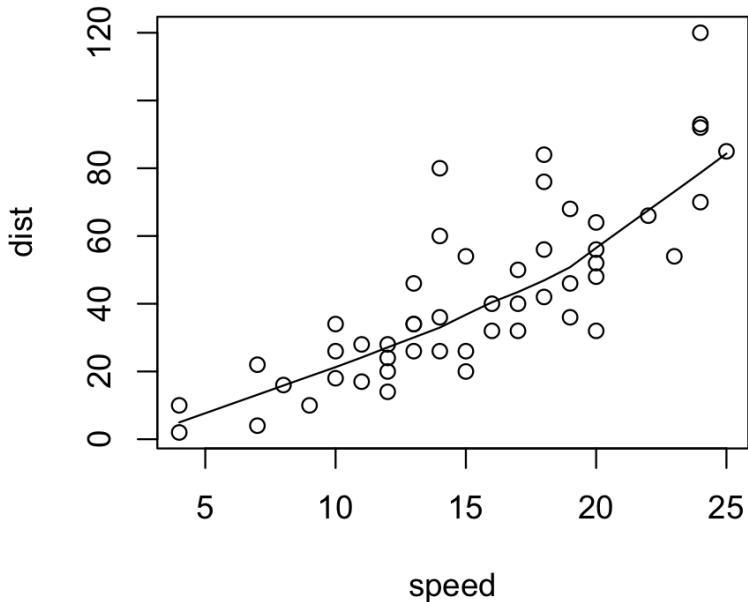


그림 7.17: lines()

R은 이외에도 loess(), ksmooth(), smooth.spline(), earth() 등의 비모수적 회귀 방법을 제공한다.

## 7 직선(abline)

abline은  $y = a + bx$  형태의 직선, 또는  $y = h$  형태의 가로로 그은 직선, 또는  $x = v$  형태의 세로로 그은 직선을 그래프에 그린다.

앞서보인 cars 데이터가  $dist = -5 + 3.5 \times speed$ 로 근사될 수 있다고 가정해보자. 그러면 다음과 같이 abline()을 사용하여 근사가 얼마나 잘 이루어지는지를 시각화 해 볼 수 있다.

```

> plot(cars, xlim=c(0, 25))
> abline(a=-5, b=3.5, col="red")

```

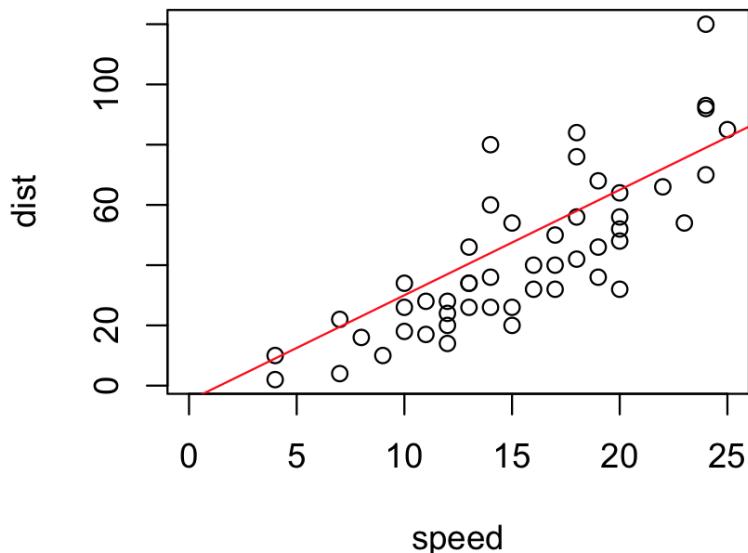


그림 7.18: abline(a=-5, b=3.5)

그래프에 speed와 dist의 평균까지 표시해보자. 다음 코드에서 lty는 선의 유형을 지정하는데 사용되며 2는 dashed line을 뜻한다.

```
> plot(cars, xlim=c(0, 25))
> abline(a=-5, b=3.5, col="red")
> abline(h=mean(cars$dist), lty=2, col="blue")
> abline(v=mean(cars$speed), lty=2, col="green")
```

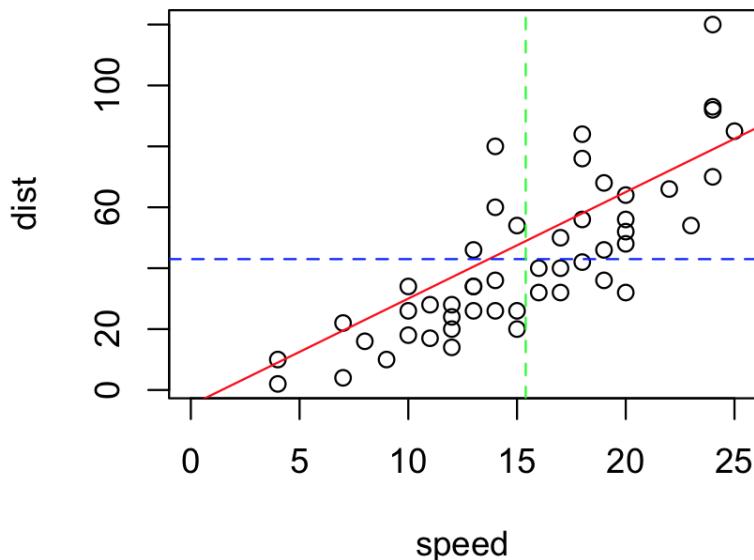


그림 7.19: abline(h=...), abline(v=...)

그래프를 그려본 결과  $dist = -5 + 3.5 \times speed$ 는 ( $x=speed$ 의 평균,  $y=dist$ 의 평균) 점을 지나지 않는다. 따라서 이 직선은 올바른 선형회귀 직선은 아니다.<sup>2)</sup>

## 8 곡선(curve)

curve는 주어진 표현식에 대한 곡선을 그리는 함수이다. 앞서  $\sin$  곡선을 lines()를 사용해 그리는 예를 [선\(lines\)](#) (페이지 187)에서 살펴보았지만, curve()를 사용해 이를 보다 직접적으로 표현할 수도 있다. 특히 curve()는 인자로 표현식, 시작점, 끝점을 받으므로 그 표현이 좀 더 편리하다.

다음은 0부터  $2\pi$ 까지의 구간에 대해  $\sin$  곡선을 그리는 예이다.

```
> curve(sin, 0, 2*pi)
```

이처럼 curve()의 첫번째 인자로는 함수명 또는 표현식을 사용할 수 있다. 자세한 내용은 ?curve를 참고하기 바란다.

다음은 위 코드의 수행 결과이다.

<sup>2)</sup><http://www.pmean.com/10/LeastSquares.html> 참고.

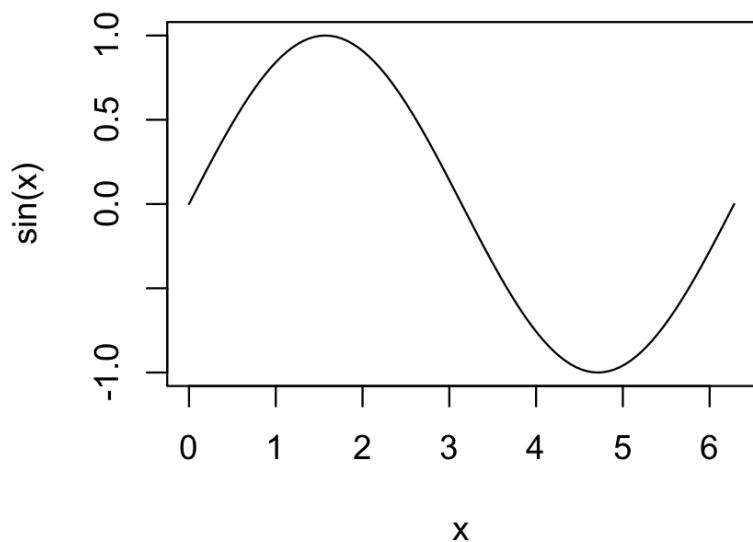


그림 7.20: curve()를 사용해 그린 sin 곡선

## 9 다각형(polygon)

polygon()은 다각형을 그리는데 사용하는 함수이다. 이번에는 cars 데이터에 선형 회귀를 수행하고, 신뢰구간(confidence interval)을 polygon() 함수를 사용해 그려보는 조금 복잡하지만 실제적인 예를 살펴보자.

선형 회귀는 lm() 함수로 수행하는데, 형식은 lm(formula, data=데이터)이다. 이 때 formula는 ‘종속변수 ~ 독립변수’의 형식을 취한다. 이렇게 만들어진 모델은 abline()으로 곧바로 그래프에 표시할 수 있으며, predict()를 사용해 예측을 수행할 수 있다. 특히 predict()에 인자로 interval=“confidence”를 주면 신뢰구간을 포함한 예측 결과를 돌려준다.

cars에 대한 선형 회귀를 수행해보자.

```
> m <- lm(dist ~ speed, data=cars)
> m
Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)      speed
-17.579        3.932
```

회귀 모형을 적합한 결과  $dist = -17.579 + 3.932 \times speed$ 의 식이 얻어졌다. 앞에서도 설명했듯이 lm()으로 만든 모델은 단순히 abline() 함수에 넘겨주는 것만으로 그래프에 표시할 수

잇다.

```
> abline(m)
```

예측은 predict()로 수행한다. 함수 호출시 인자로 interval="confidence"를 지정해 신뢰구간까지 구해보자.

```
> p <- predict(m, interval="confidence")
> head(p)

      fit        lwr        upr
1 -1.849460 -12.329543  8.630624
2 -1.849460 -12.329543  8.630624
3  9.947766  1.678977 18.216556
4  9.947766  1.678977 18.216556
5 13.880175  6.307527 21.452823
6 17.812584 10.905120 24.720047

> head(cars)

  speed dist
1      4     2
2      4    10
3      7     4
4      7    22
5      8    16
6      9    10
```

위 코드에서 p는 matrix이며 ‘fit’는 회귀모형으로 적합된 값, ‘lwr’은 신뢰구간의 하한, ‘upr’은 상한을 뜻한다. p의 각행은 cars의 각 행에 대응된다. 예를들어 p의 첫행에 있는 fit 값 -1.849460은 cars의 첫행에 있는 speed 값 4에 대한 dist 예측값이다.

polygon()으로 신뢰구간을 그리려면 그래프에 그릴 다각형의 x 좌표, y 좌표를 구해야한다. 이는 cars의 speed를 x좌표, 앞서 코드에서 구한 p의 lwr과 upr을 각각 y좌표로 한 점들을 나열해 구할 수 있다. 단, 닫혀있는 다각형을 그려야하므로 시작점과 끝점이 만나야 함에 유의한다. 이를 표현한 코드는 다음과 같다.

```
> x <- c(cars$speed,
+         tail(cars$speed, 1),
+         rev(cars$speed),
+         cars$speed[1])
> y <- c(p[, "lwr"],
```

```
+      tail(p[, "upr"], 1),
+      rev(p[, "upr"]),
+      p[, "lwr"] [1])
```

이 코드는 다음과 같이 이해할 수 있다. 먼저 (cars\$speed, p[, "lwr"])를 나열한다. 그러면 하한에 대한 선이 완성된다. 다음, (cars\$speed의 가장 마지막 값, p[, "upr"]의 가장 마지막 값)으로 선을 그린다. 이때 사용된 tail()은 head()의 반대되는 함수로 데이터의 가장 마지막 값을 얻어오는데 사용되었다. 이제 (cars\$speed, p[, "upr"])의 점들을 따라 선을 그리되 이것을 그래프의 우측에서 좌측으로 그려나간다. 이는 rev()함수를 사용해 수행되었다. 마지막으로 시작점인 (cars\$speed의 첫번째 값, p[, "lwr"]의 첫번째 값)으로 선을 그린다.

신뢰구간을 그래프에 그리는 것이므로 이 영역을 회색으로 색칠하는 것이 보기에 좋을 것이다. 그러나 col="grey"와 같이 색을 주게 되면 polygon() 호출 이전에 그래프에 그려진 내용이 모두 가려져버린다. 이때는 rgb() 함수에 alpha값을 지정해 투명한 색을 지정하면 된다.

전체 코드를 살펴보자.

```
> m <- lm(dist ~ speed, data=cars)
> p <- predict(m, interval="confidence")
> plot(cars)
> abline(m)
> x <- c(cars$speed,
+          tail(cars$speed, 1),
+          rev(cars$speed),
+          cars$speed[1])
> y <- c(p[, "lwr"],
+          tail(p[, "upr"], 1),
+          rev(p[, "upr"]),
+          p[, "lwr"] [1])
> polygon(x, y, col=rgb(.7, .7, .7, .5))
```

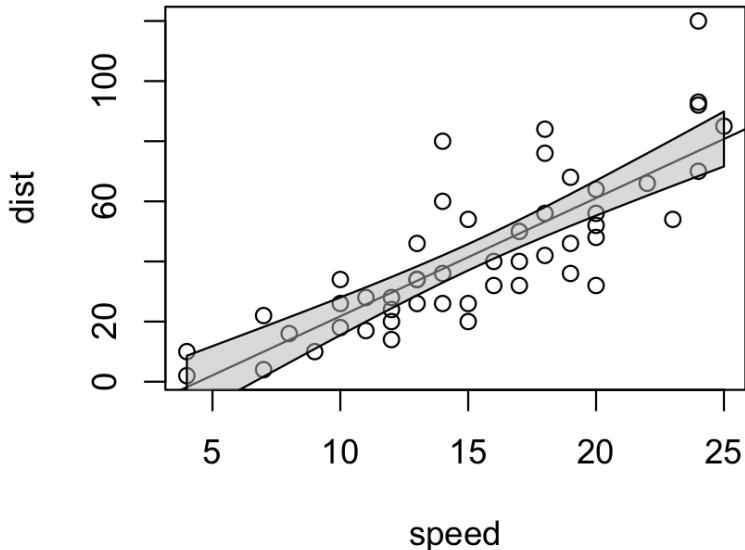


그림 7.21: polygon()

그런데 ‘통계에 강하다는 R을 사용하는데 이 정도의 그래프는 더 쉽게 가능해야 하지 않을까?’하는 생각이 자연스레 들 것이다. 이에 대한 답을 미리 보이자면, R에는 여러가지 그래픽 패키지들이 있고, 이 중 ggplot2를 사용하면 smoothing line과 신뢰 구간을 보다 더 편리하게 그릴 수 있다. 그러나 위의 코드에서 보인 방법은 그래프의 요소 요소를 하나씩 그려가기에 더 직접적인 그래픽 제어가 가능하다는 장점이 있다.

## 10 문자열(text)

text()는 그래프에 문자를 그리는데 사용하며 형식은 `text(x, y, labels)`이다. labels는 각 좌표에 표시할 문자들이며, `text()`함수에는 보여질 텍스트의 위치를 조정하기 위한 다양한 옵션이 있다. 아래 코드에서는 `pos=4`를 지정했는데, 이 경우 레이블을 주어진  $(x, y)$  좌표 우측에 표시한다.

`text` 함수의 도움말을 보기 위해 `?text` 또는 `help(text)`를 입력해보면 `labels` 값이 주어지지 않을 경우 기본값은 `seq_along(x)`임을 볼 수 있다. `seq_along(x)`은 1, 2, 3, ..., `NROW(x)` 까지의 정수를 반환하는 함수이다. 따라서  $(x, y)$  좌표들에 데이터 순서에 따라 번호를 붙이게 된다.

```
> plot(cars, cex=.5)
> text(cars$speed, cars$dist, pos=4, cex=.5)
```

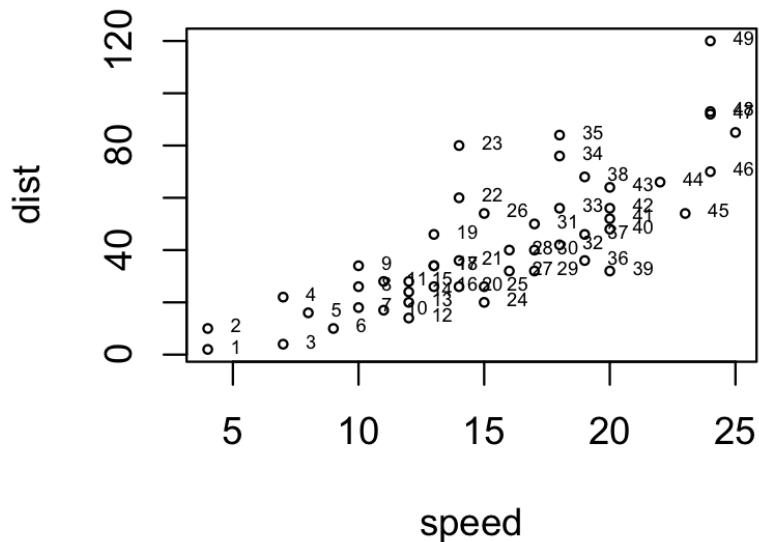


그림 7.22: text()

`text()`를 사용하면 점들에 번호를 붙여 각 점이 어느 데이터에 해당하는지 쉽게 알 수 있다. 그러나 데이터가 몰려있는 점에서는 각 점이 어느 데이터에 해당하는지 구분이 어려우며, 또 항상 모든 점에 레이블이 필요한 것은 아니다. 이런 경우에는 [그래프상에 그려진 데이터의 식별](#) (페이지 196) 를 사용한다.

## 11    그래프상에 그려진 데이터의 식별

`identify()`는 그래프상에서 특정 점을 클릭하면 클릭된 점과 가장 가까운 데이터를 그려준다. 다음 코드를 실행한 뒤 그래프 상의 점을 클릭하면 클릭된 점 옆에 각 점이 cars 데이터의 몇번째 데이터에 해당하는지 표시되는 것을 볼 수 있다. 마우스 클릭을 중단하려면 간단히 그래프가 보여진 창을 닫으면 된다.

```
> plot(cars, cex=.5)
> identify(cars$speed, cars$dist)
```

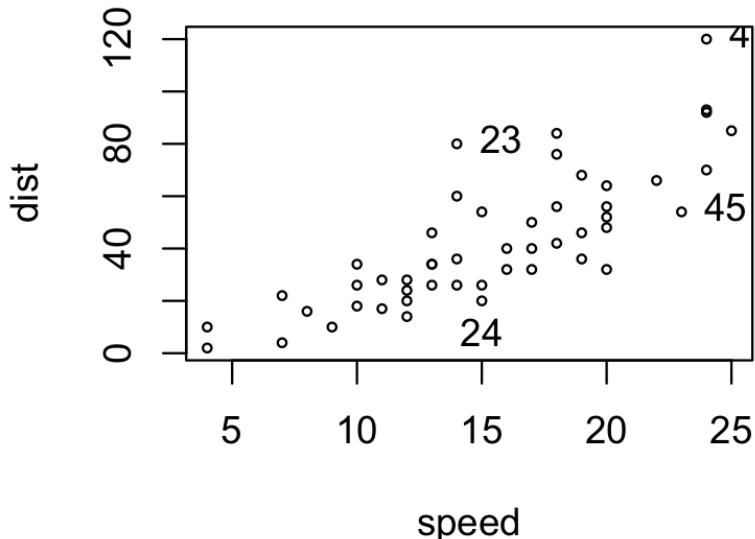


그림 7.23: identify() 호출 후 몇개의 점을 클릭한 결과

## 12 범례(legend)

legend()는 범례를 표시하는데 사용된다. 가장 기본적인 형식은 legend(x, y=NULL, legend)인데, 범례가 보여질 (x, y) 좌표를 지정할 수도 있고 사전에 정의된 키워드(bottomright, bottom, bottomleft, left, topleft, top, topright, right, center) 중 하나로 범례의 위치를 지정해도 된다.

다음에 보인 코드에서는 legend()의 위치를 topright로 했고, 범례는 Sepal, Petal을 각각 보였다. Sepal은 pch 20, Petal은 pch 43('+'의 아스키 코드)으로 지정했으며, 마찬가지로 Sepal은 색상 black, Petal은 색상 red로 지정했다. bg는 회색 배경을 의미한다. 한가지 주의할 점은 Petal을 points()로 그릴 때 pch="+"을 사용했음에도 불구하고 legend에서 pch를 지정할 때는 pch=c(20, "+")가 아니라 pch=c(20, 43)을 사용해야 한다는 것이다. 이는 앞서 [벡터\(Vector\)](#) (페이지 41)에서 살펴봤듯이 벡터는 한가지 타입의 인자만 받을 수 있기 때문이다<sup>3)</sup>.

```
> plot(iris$Sepal.Width, iris$Sepal.Length, cex=.5, pch=20,
+ xlab="width", ylab="length")
> points(iris$Petal.Width, iris$Petal.Length, cex=.5,
+ pch="+", col="#FF0000")
> legend("topright", legend=c("Sepal", "Petal"),
+ pch=c(20, 43), cex=.8, col=c("black", "red"), bg="gray")
```

<sup>3)</sup>c(20, "+")은 "20", "+"로 인식되어버린다.

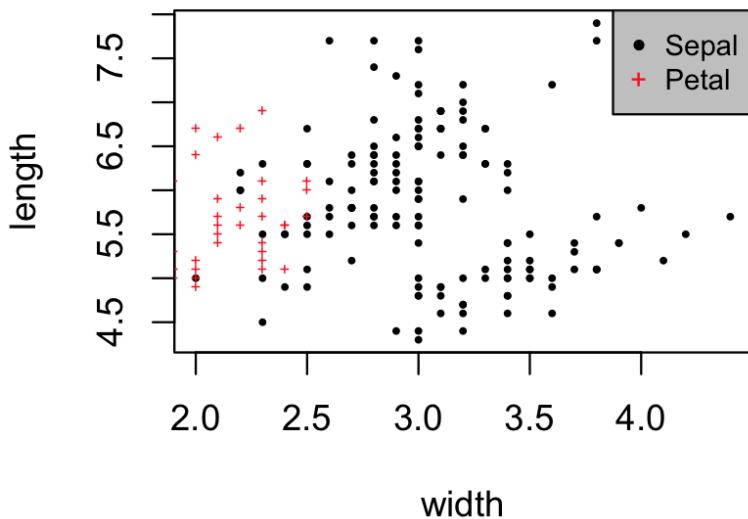


그림 7.24: legend()

## 13 행렬에 저장된 데이터 그리기(matplot, matlines, matpoints)

matplot(), matlines(), matpoints()는 각각 plot(), lines(), points() 함수와 유사하지만 행렬(matrix) 형태로 주어진 데이터를 그래프에 그린다는 점에서 차이가 있다.

$[-2\pi, 2\pi]$  구간에서의  $\cos(x)$ ,  $\sin(x)$  그래프를 matplot을 사용해 그려보자. x 값은 다음과 같이 만든다.

```
> x <- seq(-2*pi, 2*pi, 0.01)
> x
> x
> x
[1] -6.283185307 -6.273185307 -6.263185307 -6.253185307
[5] -6.243185307 -6.233185307 -6.223185307 -6.213185307
...
[1253] 6.236814693 6.246814693 6.256814693 6.266814693
[1257] 6.276814693
```

y 축은 행렬로 만든다.

```
> y <- matrix(c(cos(x), sin(x)), ncol=2)
```

위 코드에서  $c(\cos(x), \sin(x))$ 는  $\cos(x)$ 로 구해진 벡터와  $\sin(x)$ 로 구해진 벡터를 합한 새로운 벡터를 만든다. `matrix()`는 값을 열 순으로 채우므로 새로운 벡터는 2개 열을 가진 행렬이 된다. 마지막으로 `matplotlib()`으로 그래프를 그리고  $x=0$ ,  $y=0$ 의 두 축을 그려 넣는다.

```
> matplot(x, y, col=c("red", "black"), cex=.2)
> abline(h=0, v=0)
```

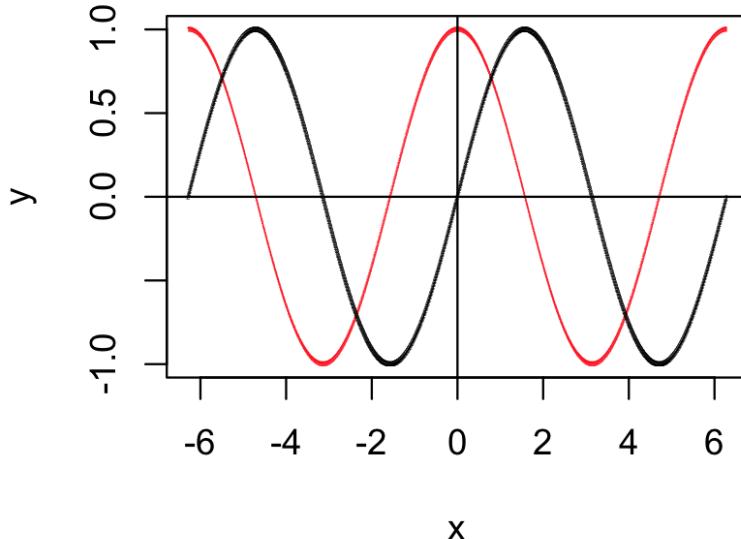


그림 7.25: `matplot()`

## 14 상자 그림(boxplot)

이 절 부터는 새로운 종류의 그래프에 대해서 살펴보자. 가장 먼저 보일 그래프는 상자그림이다. 상자 그림은 데이터의 분포를 보여주는 그림으로 가운데 상자는 제1사분위수, 중앙값, 제3사분위수를 보여준다. 상자의 좌우 또는 상하로 뻗어나간 선(whisker라고 부름)은 중앙값  $-1.5 * \text{IQR}$  보다 큰 데이터 중 가장 작은 값(lower whisker라고 부름), 중앙값  $+1.5 * \text{IQR}$  보다 작은 데이터 중 가장 큰 값(upper whisker)을 각각 보여준다. IQR은 Inter Quartile Range의 약자로 ‘제3사분위수 - 제1사분위수’로 계산한다. 그래프에 보여지는 점들은 outlier에 해당하는데 lower whisker 보다 작은 데이터 또는 upper whisker 보다 큰 데이터가 이에 해당한다. 다음은 `iris$Sepal.Width`에 대해 상자 그림을 그리는 예이다.

```
> boxplot(iris$Sepal.Width)
```

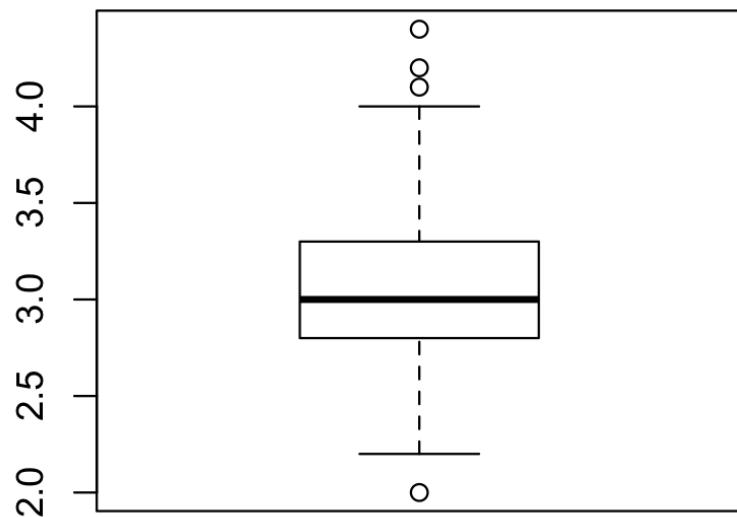


그림 7.26: boxplot()

그림에서 보다시피 제1사분위수는 약 2.8, 중앙값은 약 3.0, 제3사분위수는 약 3.3 정도였다. Lower whisker는 약 2.2 정도였고, lower whisker 보다 작은 outlier 데이터는 1개 발견되었다. 마찬가지로 upper whisker는 약 4.0 이었고 이 값보다 큰 outlier는 3개였다.

상자 그림을 그리면서 계산된 정확한 값을 보려면 boxplot()의 반환값을 보면 된다. 도움말 ?boxplot의 Value 항목을 보면 boxplot()의 반환값은 리스트로서 stats은 (lower whisker, 제1사분위수, 중앙값, 제3사분위수, upper whisker)<sup>4)</sup>를 갖고 있고, out은 outlier를 저장하고 있다. iris\$Sepal.Width에 대해 이 값들을 살펴보자.

```
> boxstats <- boxplot(iris$Sepal.Width)
> boxstats
$stats
[ ,1]
[1,] 2.2
[2,] 2.8
[3,] 3.0
[4,] 3.3
[5,] 4.0

$n
[1] 150
```

<sup>4)</sup>정확히는 제1, 3 사분위수가 아니라 lower hinge, upper hinge를 반환한다. 자세한 내용은 [다섯 수치 요약](#) (페이지 224)을 참고하기 바란다.

```
$conf
[,1]
[1,] 2.935497
[2,] 3.064503

$out
[1] 4.4 4.1 4.2 2.0

$group
[1] 1 1 1 1

$names
[1] "1"
```

보다시피 lower whisker는 2.2, 제1사분위수는 2.8, 중앙값은 3.0, 제3사분위수는 3.3, upper whisker는 4.0 이었다. outlier는 4.4, 4.1, 4.2, 2.0이었다. (이 외 값들의 의미는 도움말을 참고하기 바란다.)

이런 정보를 사용해 앞서 보였던 iris의 outlier 옆에 데이터 번호를 표시해보자.

```
> boxstats <- boxplot(iris$Sepal.Width, horizontal=TRUE)
> text(boxstats$out, rep(1, NROW(boxstats$out)), labels=boxstats$out,
       pos=1, cex=.5)
```

boxplot() 호출시 지정한 horizontal=TRUE는 상자 그림을 가로로 그리게 한다. text()를 호출할 때 y 좌표는 rep(1, NROW(boxstats\$out))을 사용했는데 이 값은 boxstats\$out의 길이가 4이므로 1이 4회 반복된 벡터인 c(1, 1, 1, 1)이 된다. 즉, 텍스트의 위치를 (outlier 값, 1)로 잡은 것이다. pos=1은 텍스트를 점의 하단에 표시하라는 의미이고 cex=.5는 글자 크기를 작게 하기 위해 사용하였다.

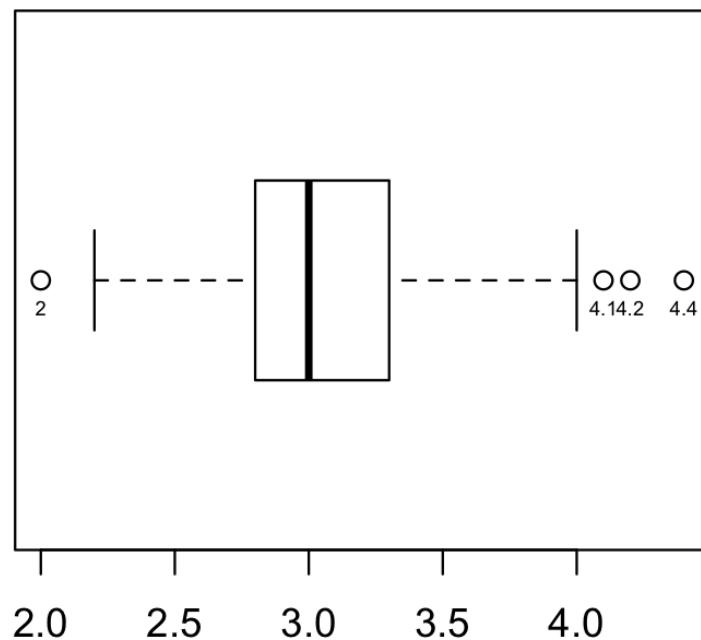


그림 7.27: boxplot()에 text()를 사용한 outlier 표시

boxplot()에는 또 한가지 흥미로운 notch 라는 인자를 지정할 수 있다. 이 값이 지정되면 median 값에 대한 신뢰구간이 오목하게 그려진다. 따라서 2개의 상자 그림을 나란히 그렸을 때 만약 두 상자 그림의 notch가 겹친다면 두 상자 그림의 중앙값이 다르지 않다고 볼 수 있다. 만약 겹치지 않는다면 두 상자 그림의 중앙값이 유의하게 다르다고 본다.

iris의 setosa종과 versicolor종의 Sepal.Width에 대한 상자 그림을 그린 뒤 이 두 종의 중앙값이 다른지 비교해보자.

```
> sv <- subset(iris, Species=="setosa" | Species=="versicolor")
> sv$Species <- factor(sv$Species)
> boxplot(Sepal.Width ~ Species, data=sv, notch=TRUE)
```

위 코드는 먼저 iris에서 Species가 setosa 또는 versicolor인 행을 선택했다. 이 때 사용하는 OR연산자는 || 가 아니라 | 임에 유의하자. |를 사용해야 하는 이유에 대해서는 [진리값](#) (페이지 38)를 참고하기 바란다.

코드의 두번째 줄에서는 sv\$Species를 다시 한번 factor로 변환했다. 이미 factor인데 어째서 factor도 다시 변환할까? 그 이유는 subset이 비록 두개의 Species만 선택하는 역할은 하지만, Species 변수 자체를 바꾸지는 않기 때문이다. 즉, sv\$Species는 subset() 후에도 여전히 level이 setosa, versicolor, virginica이기 때문이다. 따라서 실제 존재하는 level만 남기고 지우기 위해 다시한번 factor로 변환을 해주었다. 그림 7.28에는 가로축에 setosa와 versicolor만 나열되어 있다. 만약 남아있는 virginica를 없애주지 않으면 텅빈 virginica 컬럼이 추가로 보이게 된다.

마지막으로 세번째 줄에서는 Sepal.Width를 Species 마다 그렸다. 이는 ‘Sepal.Width ~ Species’로 표현되었다.

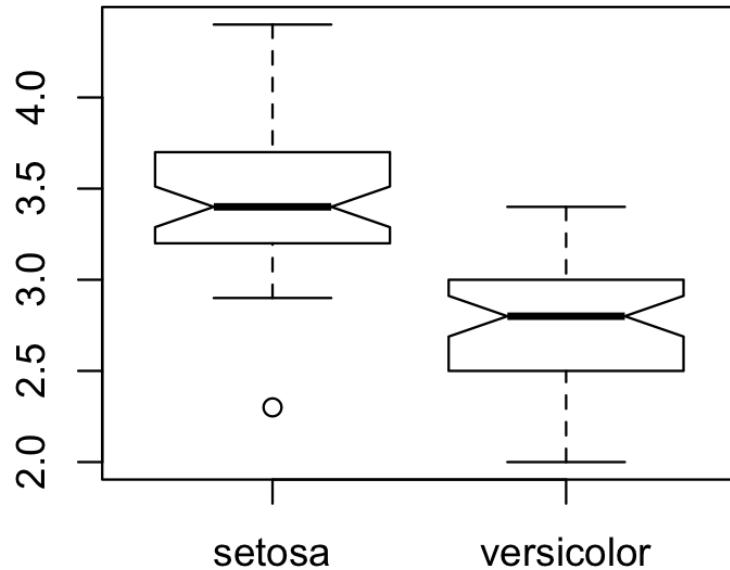


그림 7.28: 두개의 boxplot

위 그림을 보면 setosa종의 Sepal.Width가 전반적으로 versicolor종의 Sepal.Width보다 큼을 알 수 있다.

이번에는 두 상자 그림의 notch를 살펴보자. 그림을 보면 이전의 상자 그림과 달리 상자의 가운데 부분이 오목하게 들어가 있음을 볼 수 있다. 이 영역이 각각 setosa와 versicolor의 중앙값에 대한 신뢰구간이다. setosa와 versicolor의 신뢰구간이 겹치지 않으므로 이 두 그룹의 중앙값은 서로 다르다고 결론 내릴 수 있다.

## 15 히스토그램(hist)

자료의 분포를 알아보는데 유용한 또 다른 그래프는 히스토그램이다. 히스토그램은 hist(x) 명령으로 그린다.

```
> hist(iris$Sepal.Width)
```

## Histogram of iris\$Sepal.Width

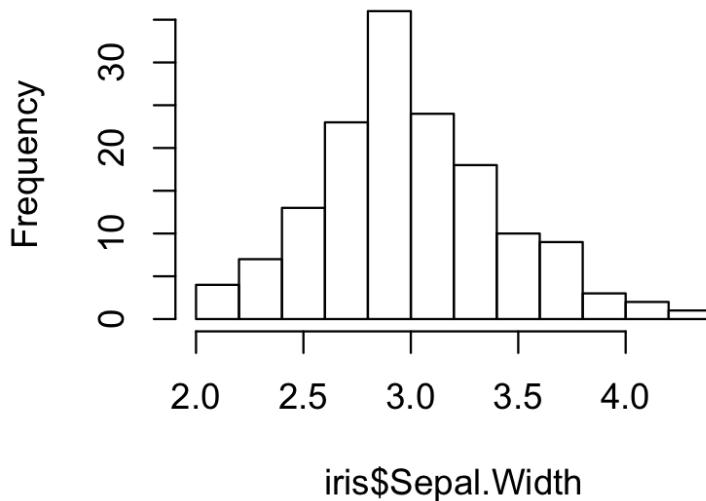


그림 7.29: hist()

히스토그램의 모양을 결정짓는 중요한 요소들중 한가지는 막대의 너비, 즉 각 막대가 나타내는 구간이다. `hist()`에는 `breaks`라는 파라미터가 있는데 이 파라미터에 적절한 인자를 설정하여 막대의 구간을 직접 지정할 수 있다. 기본값은 “Sturges”로 지정되어 있으며 Sturge 방법은 각 막대의 너비를  $\lceil \log_2(n) + 1 \rceil$ 로 지정한다. ( $n$ 은 데이터의 수)

`hist()` 의 또 다른 파라미터는 `freq`이다. `freq`의 기본 값은 `NULL`이며, 값이 지정되지 않으면 히스토그램 막대가 각 구간별 데이터의 갯수로 그려진다. 만약 이 값이 `FALSE`이면 다음 코드에서 보다시피 각 구간의 확률 밀도가 그려진다. 확률 밀도이므로 막대의 너비의 합이 1이 된다.

```
> hist(iris$Sepal.Width, freq=FALSE)
```

## Histogram of iris\$Sepal.Width

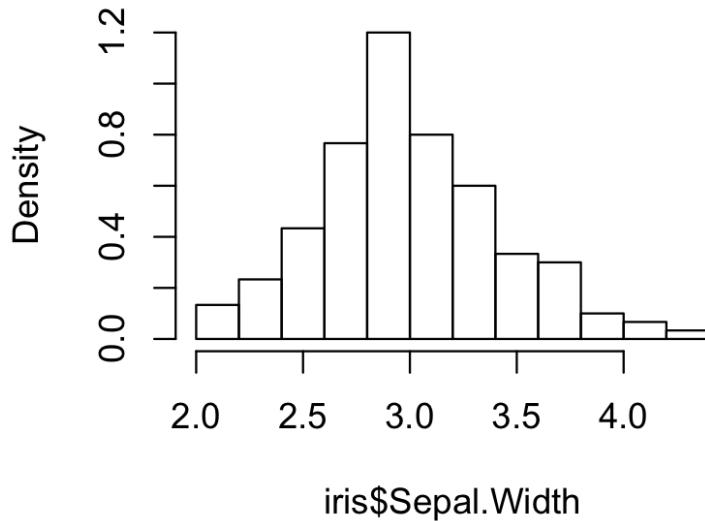


그림 7.30: hist(x, freq=FALSE)

그림 7.30의 중간쯤에 density 1.2 가 나온다고 오해하지 말기 바란다. 아래 코드에서 볼 수 있듯이 breaks에 나타난 막대의 너비가 0.2이기 때문에 density 는 1보다 클 수 있다. hist()가 그런 막대의 너비의 합이 1이면 된다.

```
> x <- hist(iris$Sepal.Width, freq=FALSE)
> x
$breaks
[1] 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4

$counts
[1] 4 7 13 23 36 24 18 10 9 3 2 1

$intensities
[1] 0.13333333 0.23333333 0.43333333 0.76666667 1.20000000
[6] 0.80000000 0.60000000 0.33333333 0.30000000 0.10000000
[11] 0.06666667 0.03333333

$density
[1] 0.13333333 0.23333333 0.43333333 0.76666667 1.20000000
[6] 0.80000000 0.60000000 0.33333333 0.30000000 0.10000000
[11] 0.06666667 0.03333333
```

```
$mids
[1] 2.1 2.3 2.5 2.7 2.9 3.1 3.3 3.5 3.7 3.9 4.1 4.3

$xname
[1] "iris$Sepal.Width"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"

> sum(x$density) * 0.2
[1] 1
```

위 코드에서 볼 수 있듯이 density의 합에 구간의 너비 0.2<sup>5)</sup>를 곱해보면 합이 1이다.

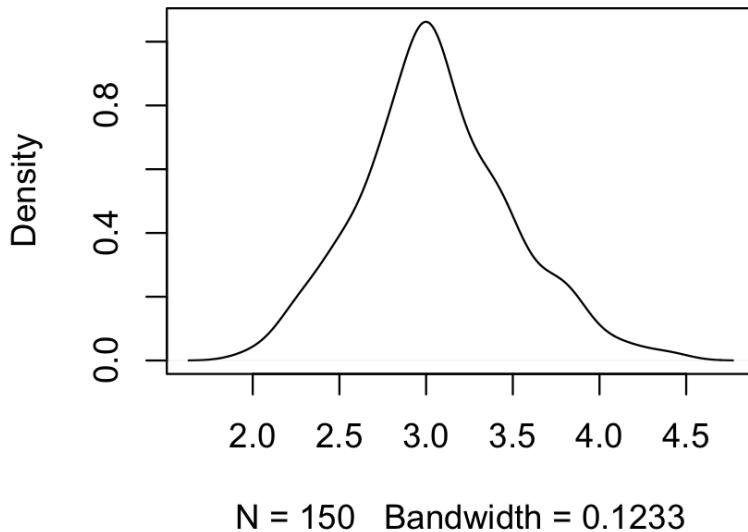
## 16 밀도 그림(density)

히스토그램은 분포를 살펴보기 위한 가장 잘 알려진 그래프이지만 bin의 너비를 어떻게 잡는지에 따라 전혀 다른 모양이 될 수 있다. bin의 경계에서 분포가 확연히 달라지지 않는 kernel density estimation<sup>6)</sup>에 의한 밀도 그림이다. 밀도 그림은 density() 함수를 사용한다.

```
> plot(density(iris$Sepal.Width))
```

<sup>5)</sup>breaks의 값의 간격이 0.2이다.

<sup>6)</sup>[http://en.wikipedia.org/wiki/Kernel\\_density\\_estimation](http://en.wikipedia.org/wiki/Kernel_density_estimation)

그림 7.31: `plot(density(x))`

밀도그림은 히스토그램과 같이 그릴 수도 있다.

```
> hist(iris$Sepal.Width, freq=FALSE)
> lines(density(iris$Sepal.Width))
```

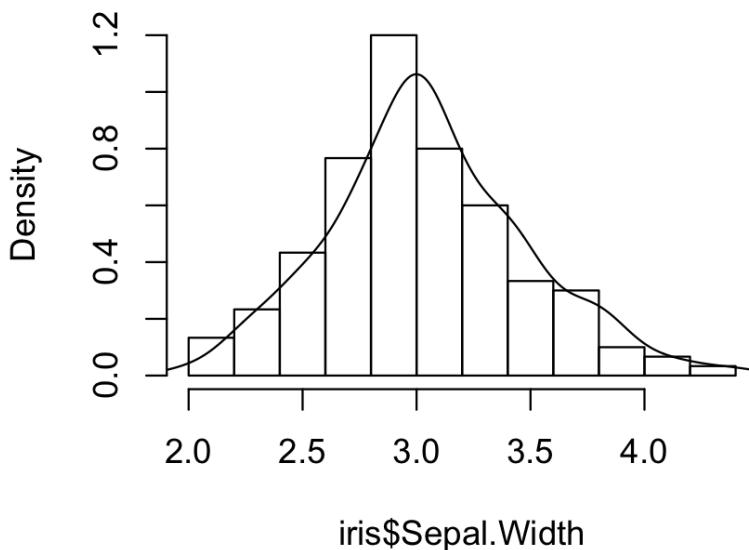


그림 7.32: 밀도그림과 히스토그램

밀도 그림에 `rug()` 함수를 사용해 실제 데이터의 위치를 표시할 수 있다. 이 때, 데이터가 중첩되는 경우가 많다면 `jitter()`을 같이 사용한다.

```
> plot(density(iris$Sepal.Width))
> rug(jitter(iris$Sepal.Width))
```

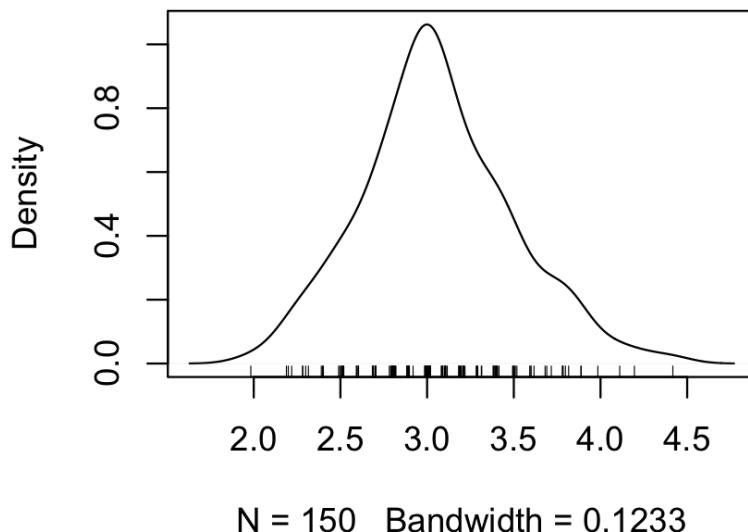


그림 7.33: rug()

## 17 막대 그림(barplot)

막대 그림은 barplot() 함수를 사용해 그린다. 다음은 Sepal.Width의 평균 값을 종별로 구하고 그 값을 막대 그림으로 나타낸 예이다. tapply (페이지 97)는 ‘데이터, 그룹 인덱스, 각 그룹별로 호출할 함수’를 인자로 받는다.

```
> barplot(tapply(iris$Sepal.Width, iris$Species, mean))
```



그림 7.34: barplot()

## 18 파이 그래프(pie)

파이 그래프는 pie() 함수를 사용해 그리며, 데이터의 비율을 알아보는데 적합하다.

이 절에서는 Sepal.Length의 구간별 비율을 파이 그래프로 그려보기로 한다. 구간으로 데이터를 나누기 위해서는 cut() 함수를 사용한다. cut()은 인자로 구간으로 나눌 데이터와 breaks를 받는다. breaks에는 나눌 구간을 직접 넘겨도 되고 나눌 구간의 수를 넘겨도 된다. 1부터 10까지의 수를 (0, 5], (5, 10] 의 두개 구간으로 나누는 다음 예를 보자.

```
> cut(1:10, breaks=c(0, 5, 10))
[1] (0,5]  (0,5]  (0,5]  (0,5]  (0,5]  (5,10] (5,10] (5,10] (5,10]
[10] (5,10]
Levels: (0,5] (5,10]
```

(0, 5]는  $0 < x \leq 5$ 의 형태이므로 breaks의 값을 c(1, 5, 10)으로 지정해서는 안된다. 이 경우 1이 어떤 구간에도 속하지 않게 되기 때문이다. 다음 코드는 1부터 10까지의 수를 3개의 구간으로 나누는 예이다.

```
> cut(1:10, breaks=3)
[1] (0.991,4] (0.991,4] (0.991,4] (4,7]      (4,7]      (4,7]
[7] (4,7]       (7,10]     (7,10]     (7,10]
Levels: (0.991,4] (4,7] (7,10]
```

이처럼 구간의 수를 지정하면 동일한 너비의 구간이 자동으로 구해지고, 각 구간별로 값이 나눠지게 된다. 이제 Sepal.Width를 10개의 구간으로 나눠보자.

```
> cut(iris$Sepal.Width, breaks=10)
[1] (3.44,3.68] (2.96,3.2] (2.96,3.2] (2.96,3.2] (3.44,3.68]
[6] (3.68,3.92] (3.2,3.44] (3.2,3.44] (2.72,2.96] (2.96,3.2]
[11] (3.68,3.92] (3.2,3.44] (2.96,3.2] (2.96,3.2] (3.92,4.16]
...
[141] (2.96,3.2] (2.96,3.2] (2.48,2.72] (2.96,3.2] (3.2,3.44]
[146] (2.96,3.2] (2.48,2.72] (2.96,3.2] (3.2,3.44] (2.96,3.2]
10 Levels: (2,2.24] (2.24,2.48] (2.48,2.72] (2.72,2.96] ... (4.16,4.4]
```

파이 그래프를 그리려면 이 factor 데이터를 그대로 사용할 수는 없으며, 나눠진 각 구간에 몇개의 데이터가 있는지 세야한다. table() 함수는 이러한 목적으로 사용되며 factor 값을 받아 분할표(Contingency Table. Cross Tabulation 또는 Cross Tab이라고도 부름)를 만든다. 이 함수의 이용법을 이해하기 위해 “a”, “b”, “b”, “c”, “c”, “c”로 구성된 벡터가 있을때 각 값의 수를 세는 다음 예를 보자.

```
> rep(c("a", "b", "c"), 1:3)
[1] "a" "b" "b" "c" "c" "c"

> table(rep(c("a", "b", "c"), 1:3))
a b c
1 2 3
```

table() 함수의 결과로 각 문자의 갯수가 구해졌음을 볼 수 있다. 마찬가지로 Sepal.Width를 10개의 구간으로 나눈 뒤 각 구간에 몇개의 데이터가 있는지는 다음 코드로 구할 수 있다.

```
> table(cut(iris$Sepal.Width, breaks=10))
(2,2.24] (2.24,2.48] (2.48,2.72] (2.72,2.96] (2.96,3.2] (3.2,3.44]
        4          7         22         24         50         18
(3.44,3.68] (3.68,3.92] (3.92,4.16] (4.16,4.4]
        10         11          2          2
```

이 책의 후반부 [분할표\(Contingency Table\)](#) (페이지 231)에서 table()에 대해 더 자세히 다루기로 하고, 여기서는 지금까지 구한 값을 사용해 파이 그래프를 그려보자.

```
> pie(table(cut(iris$Sepal.Width, breaks=10)), cex=.7)
```

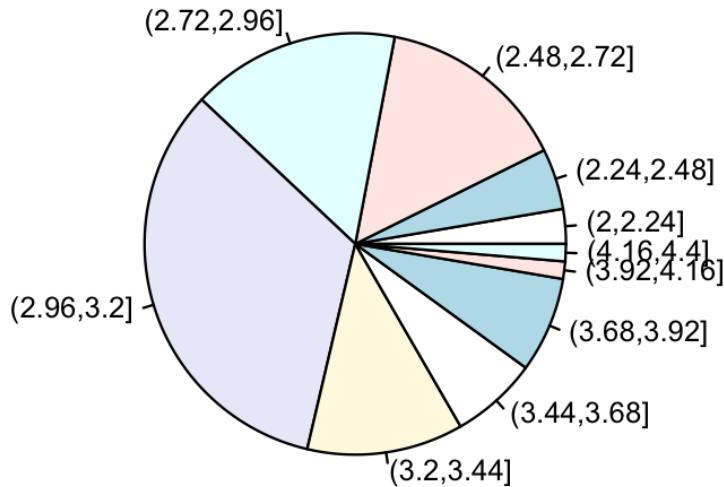


그림 7.35: pie()

## 19 모자이크 플롯(mosaicplot)

모자이크 플롯은 범주형 다변량 데이터를 표현하는데 적합한 그래프로 mosaicplot() 함수를 사용해 그린다. 모자이크 플롯에는 사각형들이 그래프에 나열되며, 각 사각형의 넓이가 각 범주에 속한 데이터의 수에 해당한다. 이 절에서는 타이타닉호 생존자의 정보를 담고 있는 Titanic 데이터를 사용해 모자이크 플롯을 그려본다. 먼저 Titanic 데이터의 형태를 살펴보자.

```
> str(Titanic)
table [1:4, 1:2, 1:2, 1:2] 0 0 35 0 0 0 17 0 118 154 ...
- attr(*, "dimnames")=List of 4
..$ Class    : chr [1:4] "1st" "2nd" "3rd" "Crew"
..$ Sex      : chr [1:2] "Male" "Female"
..$ Age       : chr [1:2] "Child" "Adult"
..$ Survived: chr [1:2] "No" "Yes"
```

Titanic은 table 클래스의 인스턴스이며 속성은 객실 구분(Class), 성별(Sex), 성인인지의 여부(Adult), 생존 여부(Survived)로 구성되어있다. 실제 데이터는 다음과 같다.

```
> Titanic
, , Age = Child, Survived = No
```

```
Sex
```

Class	Male	Female
1st	0	0
2nd	0	0
3rd	35	17
Crew	0	0

```
, , Age = Adult, Survived = No
```

#### Sex

Class	Male	Female
1st	118	4
2nd	154	13
3rd	387	89
Crew	670	3

```
, , Age = Child, Survived = Yes
```

#### Sex

Class	Male	Female
1st	5	1
2nd	11	13
3rd	13	14
Crew	0	0

```
, , Age = Adult, Survived = Yes
```

#### Sex

Class	Male	Female
1st	57	140
2nd	14	80
3rd	75	76
Crew	192	20

가장 간단한 모자이크 플롯을 그리는 방법은 이 표를 그대로 mosaicplot()에 넘기는 것이다. 다음 코드에서 color=TRUE는 그래프의 사각형들에 음영을 넣어 구분을 쉽게 하기 위해 사용되었다.

```
> mosaicplot(Titanic, color=TRUE)
```

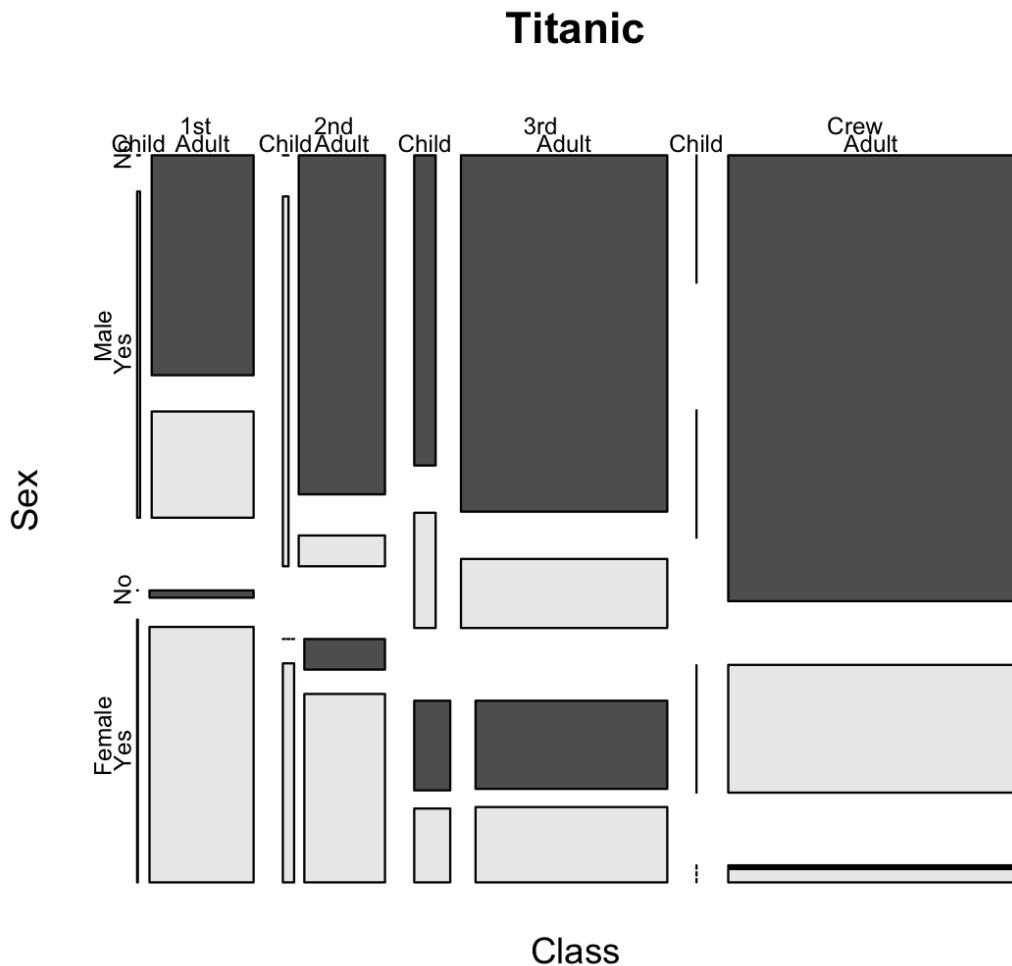


그림 7.36: mosaicplot()

그림에서 좌측 상단을 보면 성별이 남성(Male)인 경우 1등석(1st) 성인(Adult)의 사망률은 같은 그룹에서의 생존률보다 높음을 알 수 있다.

그러나 모든 조건을 다 나열해 그림을 그리면 오히려 개별 속성에 대한 분포를 살펴보기 불편하다. 일부 속성에 대해서만 살펴보려면 ‘mosaicplot(formula, data)’의 형태를 사용할 수 있다. 예를 들어 다음 코드는 객실(Class)과 생존(Survived)여부에 대한 모자이크 플롯을 그린다.

```
> mosaicplot(~ Class + Survived, data=Titanic, color=TRUE)
```

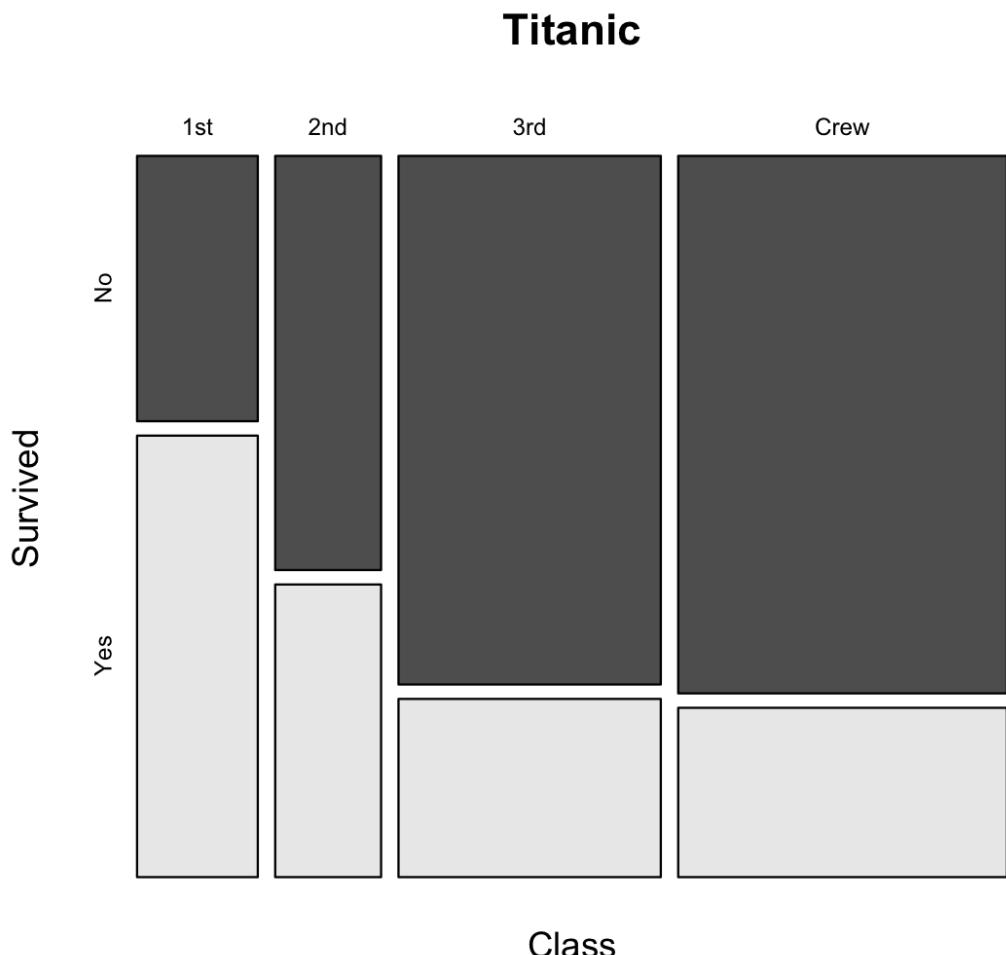


그림 7.37: mosaicplot(formula, data)

위 그림으로부터 높은 등급 객실에서의 생존률이 낮은 등급 객실에서의 생존률 보다 높음을 더 쉽게 알 수 있다.

## 20 산점도 행렬(pairs)

산점도 행렬(Scatter Plot Matrix)은 다변량 데이터에서 변수 쌍간의 산점도 행렬을 그린 그래프를 말한다. 산점도 행렬은 pairs() 함수를 이용해 그린다.

다음 코드는 iris 데이터에서 각 종별로 Sepal.Width, Sepal.Length, Petal.Width, Petal.Length 의 산점도 행렬을 그리는 예이다.

```
> pairs(~ Sepal.Width + Sepal.Length + Petal.Width + Petal.Length ,
+       data=iris, col=c("red", "green", "blue")[iris$Species])
```

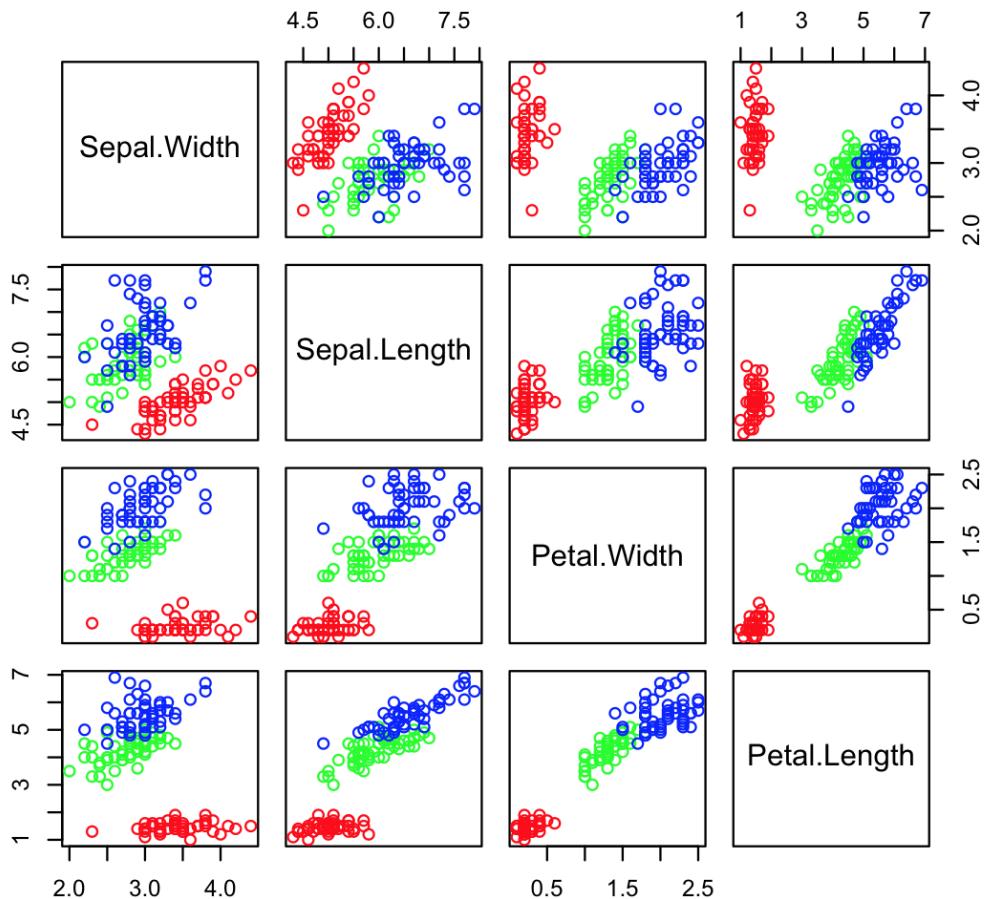


그림 7.38: pairs()

위 그림에서 setosa는 빨간색, versicolor는 녹색, virginica는 파란색으로 표시되었다. 이는 `levels()`로 범주의 목록을 살펴보면 알 수 있는데, 아래 코드에서 볼 수 있듯이 setosa가 1, versicolor가 2, virginica가 3에 해당해 각각 “red”, “green”, “blue” 색상을 사용하게 된다.

as.numeric()은 Factor를 숫자형 벡터로 바꾸는 함수로 Factor 레벨의 순서에따라 숫자값을 반환한다. 자세한 내용은 [타입 변환](#) (페이지 60)를 참고하기 바란다.

## 21 투시도(persp), 등고선 그래프(contour)

투시도는 3차원 데이터를 마치 투시한 것처럼 그린 그림으로 persp() 함수로 그린다. persp()는 인자로 X 그리드, Y 그리드, 그리고 각 grid 점에서의 Z 값을 인자로 받는다.

이 때 유용한 함수가 outer() 이다. outer는 outer(X, Y, FUN) 형태로 호출하며 FUN은 각 X, Y 조합에 대해 수행할 함수이다. 예를들어  $x=1:5$ ,  $y=1:3$  일때 모든 x, y 조합에 대해  $x+y$ 를 계산해보자.

```
> outer(1:5, 1:3, "+")
 [,1] [,2] [,3]
[1,]    2    3    4
[2,]    3    4    5
[3,]    4    5    6
[4,]    5    6    7
[5,]    6    7    8
```

위 예에서 outer()의 결과는 5행 3열의 행렬로서 (x, y)에는  $x+y$ 가 각각 저장되어 있음을 알 수 있다.

“+” 대신 함수를 기술해도 된다.

```
> outer(1:5, 1:3, function(x, y) { x + y })
 [,1] [,2] [,3]
[1,]    2    3    4
[2,]    3    4    5
[3,]    4    5    6
[4,]    5    6    7
[5,]    6    7    8
```

outer()를 사용해 X축 그리드 seq(-3, 3, .1), Y축 그리드 seq(-3, 3, .1)에 대해 이변량 정규분포(bivariate normal distribution)를 그려보자. persp()에 넘길 Z인자는 각 X, Y 그리드 점에 대해 확률 밀도인데, 다변량 정규분포(multivariate normal distribution)의 확률밀도는 dmvnorm() 을 사용해 계산한다. 다음은  $x=0$ ,  $y=0$ 에 대해 x, y의 평균이 각각 0이고 공분산 행렬(covariance matrix)이 2x2 크기의 단위행렬(identity matrix)일 때 확률 밀도를 구하는 예이다.

```
> library(mvtnorm)
> dmvnorm(c(0, 0), rep(0, 2), diag(2))
[1] 0.1591549
```

다음 코드는 seq(-3, 3, .1)의 X, Y축 그리드 조합에 대해 Z축 값을 구한다. dmvnorm()에서 평균의 기본값은 영행렬(zero matrix)이고, 공분산 행렬의 기본값은 단위행렬이다.

```
> x <- seq(-3, 3, .1)
> y <- x
> outer(x, y, function(x, y) { dmvnorm(cbind(x, y)) })
```

이제 모든 준비가 끝났다. 지금까지 살펴본 내용을 종합해 투시도를 그리는 코드는 다음과 같다.

```
> x <- seq(-3, 3, .1)
> y <- x
> f <- function(x, y) { dmvnorm(cbind(x, y)) }
> persp(x, y, outer(x, y, f), theta=30, phi=30)
```

코드에서 theta와 phi는 그림의 기울어진 각도를 지정하는 인자이다. 결과는 다음과 같다.

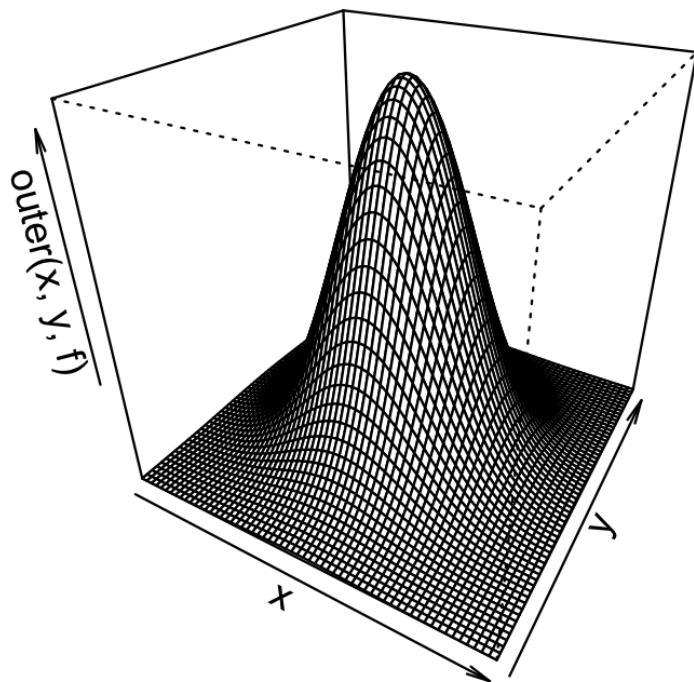


그림 7.39: persp()

등고선 그래프는 투시도와 유사하지만 투시한 3차원 그림대신 값이 같은 곳들을 선으로 연결한 등고선을 이용해 데이터를 표시한다. 등고선 그래프는 contour()를 사용해 그린다. 인자는 persp()를 그릴때와 마찬가지의 데이터를 사용하면 된다.

```
> contour(x, y, outer(x, y, f))
```

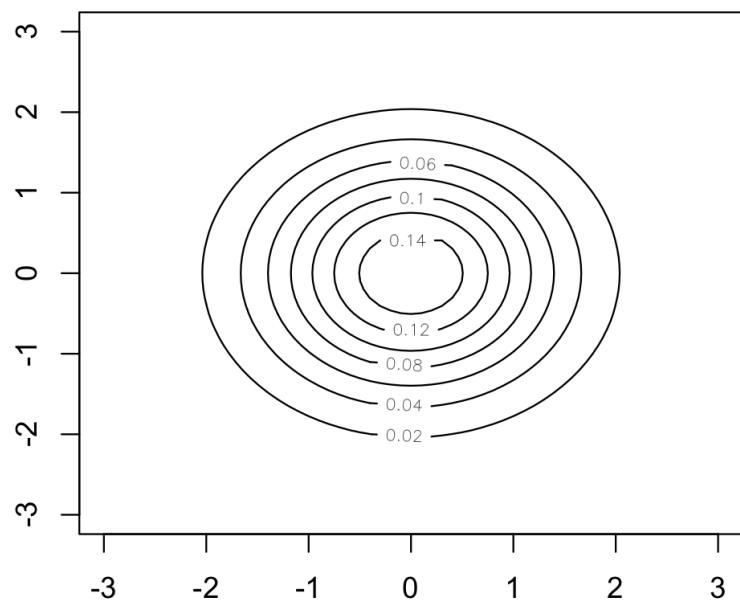


그림 7.40: contour()

# 8

## 통계 분석

이 장에서는 R언어가 특히 강점을 보이는 다양한 통계 분석 기법들을 활용하는 방법에 대해 살펴본다. 통계적 이론에 대해서는 위키피디아와 통계학 이론과 응용[20], 통계학개론[21], 수리통계학[22] 등의 통계학 서적의 내용이 인용되어있다. 부족한 내용은 참고 문서를 확인하기 바란다.

R을 사용한 통계 분석에 대한 책 역시 많이 나와 있다. 필자가 R을 배우면서 읽은 책중에는 R을 이용한 통계 프로그래밍 기초[23], R을 이용한 누구나 하는 통계분석[24], R을 이용한 비모수 통계학[25]을 추천드린다.

Linear Models with R[26]은 회귀분석을 수행하기 위한 실제적인 코드를 많이 담고 있는 추천할 만한 책이다. PDF 버전이 무료로 공개되어 있어 무료로 읽어볼 수 있다. 일반화 선형 모형에는 An Introduction to Generalized Linear Model[27], Extending the Linear Model with R[28]가 있다.

R Cookbook[29]은 소주제 별로 R의 기능들에 대해 정리한 책이지만 통계 분석을 수행하거나 그 결과를 읽는 방법에 대해서도 잘 정리되어있어 추천한다.

Wikipedia는 최고의 레퍼런스가 아닌가 생각된다. 대부분의 통계 기법에 대한 설명을 담고 있으며, 상당히 정확한 내용이 들어있다. 하지만 한글 위키피디아보다는 영문 위키피디아를 참고하기를 권한다.

## 1 난수 생성 및 분포 함수

R은 주어진 통계 분포를 따르는 난수를 발생시키는 다양한 함수를 갖고 있다. 이 함수들은 r 뒤에 분포명을 붙인 이름을 갖고 있다. 표 8.1에 몇몇 분포에 대한 함수를 정리했다. 좀 더 완벽한 목록은 <http://www.stat.umn.edu/geyer/old/5101/rlook.html#dist>을 참고하기 바란다.

확률 분포	난수 발생 함수
이항분포(Binomial)	rbinom
F 분포(F)	rf
기하분포(Geometric)	rgeom
초기하분포(Hypergeometric)	rhyper
음이항분포(Negative Binomial)	rnbnom
정규 분포(Normal)	rnorm
포아송 분포(Poisson)	rpois
t 분포(Student t)	rt
연속 균등 분포(Uniform)	runif

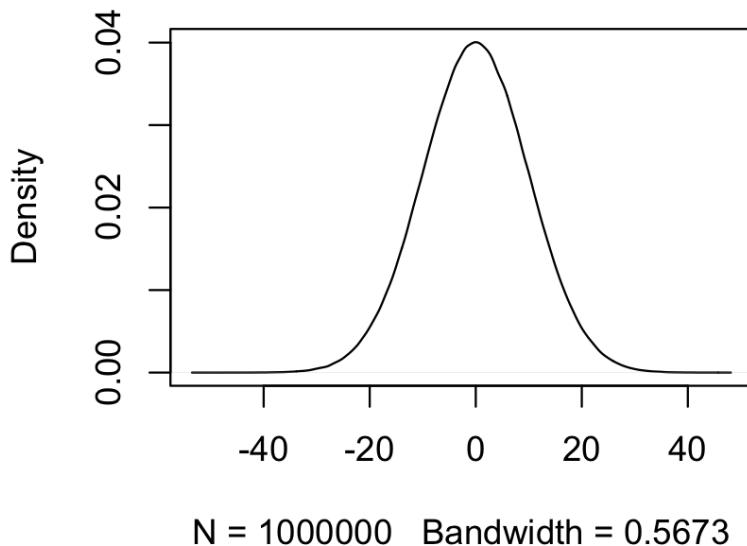
표 8.1: 확률 분포 및 난수 발생 함수

이들 함수는 함수명(발생시킬 난수의 수, 파라미터1, 파라미터2, ..., 파라미터N)의 형식으로 호출한다. 다음은 평균 0, 표준편차 10인 정규분로부터 100개의 난수를 뽑는다.

```
> rnorm(100, 0, 10)
[1] 6.35522264 -15.91675609  0.11219825  2.81311412  8.94825134
...
[96] -4.70195484 12.33659335 -15.98517300 -13.41173703 -12.91536521
```

많은 수의 샘플을 사용해 [밀도 그림\(density\)](#) (페이지 206)을 그려보면 데이터가 정규분포를 잘 따르고 있음을 볼 수 있다.

```
> plot(density(rnorm(1000000, 0, 10)))
```

그림 8.1: `rnorm()`으로 구한 샘플의 밀도 그림

정규분포의 누적 분포(Cumulative Distribution Function)을 구하는 함수들은 p 뒤에 분포명을 적은 형태의 함수 이름을 갖는다. 예를 들어 정규분포의 cdf는 `pnorm()`이다.

분위수(Quartile)는 q 뒤에 분포명을 적으면 정규분포의 경우 `qnorm()`으로 구한다. `qnorm()`이 `pnorm()`의 역함수에 해당하고 p 와 q가 알파벳에서 연속된 문자임을 상기하면 기억하기 쉽다.

확률 밀도(Probability Density)는 d 뒤에 분포명을 적는다. 정규분포의 경우 `dnorm()`을 사용한다.

위 세 가지 함수와 난수 발생 함수를 모두 정리해 표 8.2에 보였다.

확률 분포	난수	확률 밀도	누적 분포	분위수
이항분포(Binomial)	<code>rbinom</code>	<code>dnorm</code>	<code>pnorm</code>	<code>qnorm</code>
F 분포(F)	<code>rf</code>	<code>df</code>	<code>pf</code>	<code>qf</code>
기하분포(Geometric)	<code>rgeom</code>	<code>dgeom</code>	<code>pgeom</code>	<code>qgeom</code>
초기하분포(Hypergeometric)	<code>rhyper</code>	<code>dhyper</code>	<code>phyper</code>	<code>qhyper</code>
음이항분포(Negative Binomial)	<code>rnbnom</code>	<code>dnbinom</code>	<code>pnbinom</code>	<code>qnbinom</code>
정규 분포(Normal)	<code>rnorm</code>	<code>dnorm</code>	<code>pnorm</code>	<code>qnorm</code>
포아송 분포(Poisson)	<code>rpois</code>	<code>dpois</code>	<code>ppois</code>	<code>qpois</code>
t 분포(Student t)	<code>rt</code>	<code>dt</code>	<code>pt</code>	<code>qt</code>
연속 균등 분포(Uniform)	<code>runif</code>	<code>dunif</code>	<code>punif</code>	<code>qunif</code>

표 8.2: 확률 분포 및 관련 함수

위 함수들을 실제 코드로 연습해보자. 포아송 분포의 확률 질량 함수(Probability Mass Function)은 다음과 같다.

$$f(n; \lambda) = \frac{\lambda^n e^{-\lambda}}{n!} \quad (8.1)$$

$\lambda = 1$  일 때,  $f(3; 1)$ 을 구해보자.

```
> dpois(3, 1)
[1] 0.06131324
> (1^3 * exp(-1)) / (factorial(3))
[1] 0.06131324
```

실행 결과 dpois()에서 구한 값과 수식으로 구한 값이 일치했다.

다음으로  $N(0, 1)$ 의 정규 분포에서 누적 분포  $F(0)$ , 그리고 50%에 대한 분위수  $F^{-1}(0)$ 을 구해보자.

```
> pnorm(0)
[1] 0.5
> qnorm(0.5)
[1] 0
```

## 2 기초 통계량

### 2.1 평균, 표본 분산, 표본 표준편차

평균, 표본 분산, 표본 표준편차는 각각 mean(), var(), sd()로 구한다.<sup>1)</sup>

```
> mean(1:5)
[1] 3
> var(1:5)
[1] 2.5
> sum((1:5-mean(1:5))^2)/(5-1)
[1] 2.5
```

---

<sup>1)</sup> 표본에 대한 분산과 표준편차이므로 분모가  $n$ 이 아니라  $n - 1$ 이다.

## 2.2 다섯 수치 요약

다섯 수치 요약은 데이터를 ‘최소값, 제1사분위수, 중앙값, 제3사분위수, 최대값’으로 요약한다. 다섯 수치 요약을 구하는 함수는 `fivenum()`이다. `summary()`는 `fivenum()`과 유사하지만 다섯 수치 요약에 더해 평균까지 계산해준다.

```
> fivenum(1:11)
[1] 1.0 3.5 6.0 8.5 11.0
> summary(1:11)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
1.0      3.5      6.0      6.0      8.5      11.0
```

`fivenum()`과 `summary()`는 데이터의 수가 홀수개일 경우에는 위의 예처럼 동일한 결과를 보이지만 데이터의 갯수가 짝수일때는 다소 다른 결과를 출력한다.

```
> fivenum(1:4)
[1] 1.0 1.5 2.5 3.5 4.0
> summary(1:4)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
1.00    1.75    2.50    2.50    3.25    4.00
```

그 이유는 다음과 같다. `summary()`는 1:4에서 제1사분위수, 제3사분위수를 다음과 같이 수식적으로 계산한다.

$$\text{1st Qu.} = 1 + (4 - 1) \times (1/4)$$

$$\text{3rd Qu.} = 1 + (4 - 1) \times (3/4)$$

반면 `fivenum()`은 1, 2, 3, 4에서 25% 백분율인 위치가 1.75로 1과 2의 사이므로 제1사분위수를 1과 2의 평균인 1.5로 잡는다. 또 75% 백분율의 위치가 3.25로 3과 4 사이의 수이므로 3과 4의 평균인 3.5로 제3사분위수를 잡는다. 실은 제1 또는 제3사분위수 대신 이들 두 수를 `fivenum()`에서는 lower-hinge, upper-hinge로 지칭한다. [상자 그림\(boxplot\)](#) (페이지 199)에서 살펴본 `boxplot()` 함수에서도 같은 방식의 계산을 사용한다.

다섯 수치 요약의 값들을 하나씩 별도의 함수로 계산해보자. 아래 코드에서 `quantile()` 함수만 낯설어 보일텐데, `quantile()`은 백분위수를 구하는데 사용하는 함수로 `quantile(데이터, 백분율)` 형태로 호출한다.

```
> x <- 1:10
```

```
> c(min(x), quantile(x, 1/4), median(x), quantile(x, 3/4), max(x))
      25%           75%
1.00  3.25  5.50  7.75 10.00
```

사분위수를 살펴보았으니, 이와 관련된 IQR(Inter-Quartile Range. ‘제3사분위수 - 제1사분위수’의 값)을 계산해보자. IQR은 이름 그대로 IQR()함수를 사용해 계산한다. 아래 코드에서는 IQR()을 사용해 계산한 값을 직접 제1, 3사분위수를 구해 계산한 값을 비교해 보았다.

```
> IQR(1:10)
[1] 4.5
> quantile(1:10, c(1/4, 3/4))
 25% 75%
3.25 7.75
> 7.75 - 3.25
[1] 4.5
```

### 2.3 최빈값(mode)

최빈값은 [파이 그래프\(pie\)](#) (페이지 209)에서 잠시 살펴보았던 table()을 사용해 분할표 (Contingency Table)를 만들고, [which\(\)](#), [which.max\(\)](#), [which.min\(\)](#) (페이지 113)에서 살펴본 which.max()를 사용해 최대 값이 저장된 색인을 찾는 방법을 사용해 구할 수 있다.

다음 코드는 Factor에서 각 문자별 출현 횟수를 table()로 구한 다음, 출현 횟수가 가장 큰 색인을 which.max()로 찾는 예이다. 최빈값 자체를 구할 때는 names()를 사용해 분할표에서 각 셀의 이름을 구한 뒤 최대값이 저장된 셀을 선택했다.

```
> x <- factor(c("a", "b", "c", "c", "c", "d", "d"))
> x
[1] a b c c c d d
Levels: a b c d
> table(x)
x
a b c d
1 1 3 2
> which.max(table(x))
c
3
> names(table(x))[3]
```

```
[1] "c"
```

### 3 표본추출

컴퓨터를 사용하여 기계적으로 데이터를 처리하는 경우라 할지라도 분석할 데이터 역시 기하급수적으로 늘어나기에 올바른 표본의 추출 방법은 더없이 중요하다.

#### 3.1 단순 임의 추출(Random Sampling)

단순 임의 추출은 sample(데이터, 표본 크기, 복원추출여부, 가중치)의 형식으로 표본을 추출할 수 있다. 간단한 예로 1에서 10까지의 수중 5개의 수를 비복원 추출(Sampling without Replacement)로 뽑아보자.

```
> sample(1:10, 5)
[1] 4 5 6 10 9
```

1에서 10까지의 수에서 복원 추출(Sampling With Replacement)로 5개의 표본을 뽑아보자.

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

1에서 10까지의 수에 각각 1에서 10까지의 가중치를 주어 복원 추출 해보자. 아래 코드를 보면 가중치가 큰 표본이 더 많이 뽑히는 경향을 쉽게 확인할 수 있다.

```
> sample(1:10, 5, replace=TRUE, prob=1:10)
[1] 10 9 6 7 7
```

sample()은 주어진 데이터의 순서를 보존하지 않는다. 따라서 sample()을 주어진 데이터를 섞는(shuffle) 목적으로 사용할 수 있다.

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

#### 3.2 층화 임의 추출(Stratified Random Sampling)

데이터가 중첩 없이 분할 될 수 있는 경우(즉 disjoint 한 부분으로 나뉠 수 있는 경우) 그리고 각 분할의 성격이 명확히 다른 경우 층화 임의 추출을 수행하여 더 정확한 결과를 얻을 수 있다.

예를들어 남성 20%, 여성 80%로 구성된 집단이 있을 때 이 집단의 평균 키를 측정한다고 가정해보자. 성별에 따라 키의 차이가 명확히 존재할 것이므로 표본을 잘 추출하는 것이 무엇보다 중요할 것이다. 그런데 단순 임의 추출을 적용하게되면 남성이 우연히 20%보다 더 많이 추출되거나 또는 20%보다 더 적게 추출될 수 있다. 다시 말해 단순 임의 추출은 평균에 대한 추정의 정확도가 떨어지는 위험이 있다.

이런 경우 층화 추출을 사용해 데이터로부터 남성과 여성의 표본의 갯수를 2:8로 유지하여 뽑는다면 더 정확한 결과를 얻을 수 있게된다. 또한 층화 추출을 하게 되면 뽑힌 남성의 표본에 대해서도 평균 키를 측정할 수 있고, 여성의 표본에 대해서도 평균 키를 측정할 수 있게 된다. 다시말해 각 층에 대한 추정역시 가능해지는 장점이 있다. 자세한 설명은 참고문헌[30]을 보기 바란다.

직접 층화 임의 추출을 수행해보자. 층화 임의 추출은 sampling::strata() 함수를 사용한다. 다음 예는 iris데이터로부터 srswor(Simple Random Sampling Without Replacement. 비복원 단순 임의 추출)을 사용해 각 Species별로 3개씩 샘플을 추출한다. strata()가 반환한 값으로부터 실 데이터를 얻으려면 getdata() 함수를 사용한다. 아래 코드에서 getdata()의 출력이 마치 2개의 표처럼 보이지만 이는 종이 폭 탓이고 실제로는 하나의 데이터 프레임이다.

```
> install.packages("sampling")
> library(sampling)
> x <- strata(c("Species"), size=c(3, 3, 3), method="srswor",
+               data=iris)
> x
      Species ID_unit Prob Stratum
10     setosa      10  0.06      1
20     setosa      20  0.06      1
31     setosa      31  0.06      1
66 versicolor     66  0.06      2
75 versicolor     75  0.06      2
76 versicolor     76  0.06      2
123 virginica    123  0.06      3
125 virginica    125  0.06      3
138 virginica    138  0.06      3

> getdata(iris, x)
   Sepal.Length Sepal.Width Petal.Length Petal.Width     Species
10          4.9       3.1        1.5       0.1     setosa
20          5.1       3.8        1.5       0.3     setosa
```

31	4.8	3.1	1.6	0.2	setosa
66	6.7	3.1	4.4	1.4	versicolor
75	6.4	2.9	4.3	1.3	versicolor
76	6.6	3.0	4.4	1.4	versicolor
123	7.7	2.8	6.7	2.0	virginica
125	6.7	3.3	5.7	2.1	virginica
138	6.4	3.1	5.5	1.8	virginica
ID_unit	Prob	Stratum			
10	10	0.06	1		
20	20	0.06	1		
31	31	0.06	1		
66	66	0.06	2		
75	75	0.06	2		
76	76	0.06	2		
123	123	0.06	3		
125	125	0.06	3		
138	138	0.06	3		

strata() 함수가 편리한 점은 층별로 다른 수의 표본을 추출할 수 있다는 점이다. 다음 예에서는 setosa 종에서 3개의 샘플을 추출하고, 나머지 종에서는 각 1개씩 표본을 추출했다.

> strata(c("Species"), size=c(3, 1, 1), method="srswr", data=iris)
Species ID_unit Prob Stratum
5 setosa 5 0.06 1
38 setosa 38 0.06 1
46 setosa 46 0.06 1
89 versicolor 89 0.02 2
116 virginica 116 0.02 3

또한 strata()는 다수의 층을 기준으로 데이터를 추출할 수 있다. iris에 Species2라는 이름으로 또 다른 층을 만들고, (Species, Species2)의 각 층마다 1개씩 표본을 추출하는 예를 아래에 보였다.

> iris\$Species2 <- rep(1:2, 75)
> strata(c("Species", "Species2"), size=c(1, 1, 1, 1, 1, 1),
+ method="srswr", data=iris)
Species Species2 ID_unit Prob Stratum

49	setosa	1	49	0.04	1
26	setosa	2	26	0.04	2
59	versicolor	1	59	0.04	3
60	versicolor	2	60	0.04	4
127	virginica	1	127	0.04	5
114	virginica	2	114	0.04	6

각 층마다 동일한 갯수의 표본을 추출하고자한다면 doBy::sampleBy()를 사용할 수 있다. 형식은 sampleBy(Formula, 추출할 표본의 비율, 복원 추출 여부(기본값 FALSE), 데이터)이다.

```
> sampleBy(~ Species, frac=.06, data=iris)
      Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa.1           5.1        3.5       1.4        0.2
setosa.27          5.0        3.4       1.6        0.4
setosa.42          4.5        2.3       1.3        0.3
versicolor.63      6.0        2.2       4.0        1.0
versicolor.70      5.6        2.5       3.9        1.1
versicolor.98      6.2        2.9       4.3        1.3
virginica.111     6.5        3.2       5.1        2.0
virginica.118     7.7        3.8       6.7        2.2
virginica.147     6.3        2.5       5.0        1.9
      Species
setosa.1           setosa
setosa.27          setosa
setosa.42          setosa
versicolor.63      versicolor
versicolor.70      versicolor
versicolor.98      versicolor
virginica.111     virginica
virginica.118     virginica
virginica.147     virginica
```

결과에서 각 행은 ‘Species명.행 번호’로 이름 붙여져 있음을 볼 수 있다.

### 3.3 계통 추출(Systematic Sampling)

아침부터 밤까지 특정한 지역을 지나간 차량의 번호를 모두 조사하였고, 이들로부터 조사 대상을 뽑는 경우를 가정해보자. 가장 간단한 단순 임의 추출을 적용하여 차량 번호를 뽑는다면

우연히 아침시간에 지나간 차량을 더 많이 뽑거나 저녁시간에 지나간 차량을 더 많이 뽑는 편향이 발생할 수 있다. 계통추출은 이런 상황에서 해답이 될 수 있다.

계통 추출은 모집단의 임의 위치에서 시작해 매  $k$  번째 항목을 표본으로 추출하는 방법이다. 예를들어 1, 2, 3, ..., 10까지의 수에서 3개의 샘플을 뽑는다고 가정해보자.  $10/3 = 3.333\dots$  이므로  $k = 3$ 이다. 표본 추출 시작위치를 잡기위해  $1 \sim k$  사이의 수 하나를 뽑는다. 이 수가 2라하자. 나머지 두 수를 뽑기위해  $2 + k$  에 해당하는 5를 뽑는다. 다음,  $5 + k$ 에 해당하는 8을 뽑는다. 그러면 최종적으로 표본 ‘2, 5, 8’를 얻는다.

매우 단순한 방법이지만 랜덤 모집단의 경우에는 단순 임의 추출방법과 동일한 효과를 보이고, 만약 데이터가 순서대로 나열된 순서 모집단(Ordered Population. 예를들어 ‘1, 2, 3, ..., 10’과 같이 순서대로 나열된 모집단)의 경우 단순 임의 추출보다 효율성이 높다(추정량의 분산이 작아진다). 하지만 데이터에 일종의 주기성이 존재한다면 (예를들어 데이터가 ‘1, 2, 3, 1, 2, 3, ...’ 같이 반복되는 경우) 편향된 표본을 얻게 된다[30].

`sampleBy()`에 `systematic=TRUE` 옵션을 주어 손쉽게 계통 추출을 할 수 있다. 다음 예는 `1:10` 을 저장한 데이터 프레임에서 3개의 표본을 계통 추출로 뽑는 예이다. 코드에서 `sampleBy()`의 첫번째 인자는 ‘ $\sim 1$ ’이다. 그 이유는 첫번째 인자가 표본을 추출할 그룹을 지정하는 `formula`이기 때문이다. 만약 그룹 별로 데이터를 뽑는 충화 임의 추출이라면 그룹을 뜻하는 표현을 적어야하지만, 여기서는 그룹의 구분이 없으므로 상수 1을 사용하였다. 실행결과 ‘1, 4, 7’의 3개 표본이 뽑혔다.

```
> x <- data.frame(x=1:10)
> x
   x
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
> sampleBy(~1, frac=.3, data=x, systematic=TRUE)
 [,1] [,2] [,3]
1     1     4     7
```

## 4 분할표(Contingency Table)

분할표는 명목형(Categorical) 또는 순서형(Ordinal) 자료의 도수를 표 형태로 기록한 것이다. 분할표가 작성되면 Chi Square Test로 확률 변수(Random Variable. 이후 ‘변수’라고도 부르도록 하겠다.)간에 의존관계가 있는지 살펴보는 독립성 검정, 도수가 특정 분포를 따르는지에 대한 적합도 검정을 수행할 수 있다. 만약 표본의 수가 작다면 피셔의 정확 검정(Fisher's Exact Test)을 시행하고, 짹지은 자료의 경우에는 맥니마 검정(McNemar Test)를 한다.

### 4.1 분할표의 작성

분할표를 작성하는 기본 함수는 `table()`이다. 다음은 주어진 벡터에서 a, b, c의 출현 횟수를 세는 간단한 예이다.

```
> table(c("a", "b", "b", "b", "c", "c", "d"))
a b c d
1 3 2 1
```

`xtabs()`는 분할표를 만드는 또 다른 함수로 `table()`과 달리 `Formula`를 사용해 데이터를 지정할 수 있다. `Formula` 작성시에는 ‘~’ 앞에 도수가 저장된 변수를 적고 ‘~’ 뒤에 분류를 나타내는 범주형 변수를 ‘+’로 연결해 적는다. 예를 들어 x, y라는 두 가지 속성이 있고 (x, y)에 대한 도수가 num에 저장되어 있을 때 이 데이터로 부터 분할표를 만드는 `Formula`는 `num ~ x + y`이다. 다음 예를 보자.

```
> d <- data.frame(x=c("1", "2", "2", "1"),
+                   y=c("A", "B", "A", "B"),
+                   num=c(3, 5, 8, 7))
> d
   x y  num
1 1 A    3
2 2 B    5
3 2 A    8
4 1 B    7
> xt <- xtabs(num ~ x + y, data=d)
> xt
      y
x   A B
  1 3 7
  2 8 5
```

만약 도수를 나타내는 컬럼이 따로 없고, 각 관찰 결과가 서로 다른 행으로 표현되어 있다면 ‘~ 변수 + 변수 ...’ 형태로 formula를 작성한다. 다음 코드는 x 값이 A 또는 B인 각 경우의 수를 세는 예를 보여준다.

```
> d2 <- data.frame(x=c("A", "A", "A", "B", "B"),
+                     result=c(3, 2, 4, 7, 6))
> xtabs(~ x, d2)
x
A  B
3  2
```

분할표에서는 종종 행, 열의 합을 계산해 표시한다. margin.table()은 이런 목적으로 사용된다. 호출 형식은 margin.table(데이터, margin)의 형태이며 이때 ‘margin’의 값이 1이면 각 행에 있는 도수의 합, 2이면 각 열에 있는 도수의 합을 계산한다. 만약 ‘margin’을 지정하지 않으면 전체 도수의 합을 구한다.

```
> xt
  y
x   A  B
  1 3  7
  2 8  5
> margin.table(xt, 1)
x
  1  2
10 13
> margin.table(xt, 2)
y
A  B
11 12
> margin.table(xt)
[1] 23
```

prop.table()은 분할표로부터 각 셀의 비율을 계산한다. 호출형식은 margin.table()의 경우와 동일하다.

```
> xt
  y
x   A  B
```

```

1 3 7
2 8 5
> prop.table(xt, 1)
y
x           A           B
1 0.3000000 0.7000000
2 0.6153846 0.3846154
> prop.table(xt, 2)
y
x           A           B
1 0.2727273 0.5833333
2 0.7272727 0.4166667
> prop.table(xt)
y
x           A           B
1 0.1304348 0.3043478
2 0.3478261 0.2173913

```

위 코드에서 `prop.table(xt, 1)`의 결과를 확인해보자.  $x = 1, y = A$ 인 경우의 출력 값은 행의 합( $=3+7$ )에 대한  $x = 1, y = A$ 인 경우의 비이므로  $3/(3+7) = 0.333\dots$  이다. 마찬가지로  $x = 1, y = B$ 인 경우  $7/(3+7) = 0.7$ 이다.

`prop.table(xt)`는 모든 셀의 합( $=3+7+8+5$ )에 대한 각 셀의 비율이다. 예를 들어  $x = 1, y = A$ 인 경우  $3/(3+7+8+5) = 0.1304\dots$  이다.

범주형 변수에 대한 좀 더 다양한 요약 기능은 [reshape2 패키지](#) (페이지 130)를 참고하기 바란다.

## 4.2 독립성 검정(Independence Test)

분할표의 행에 나열된 속성과 열에 나열된 속성이 독립이라면  $(i, j)$  셀의 확률  $P(i, j)$ 에 대해 다음 식이 성립한다.

$$P(i, j) = P(i) \times P(j) \quad (8.2)$$

독립성 검정에는 Chi-Squared Test를 사용하며 이 때 사용되는 통계량은 다음과 같다.

$$\sum_{i=1}^r \sum_{j=1}^c \frac{O_{ij} - E_{ij}}{E_{ij}} \sim \chi^2(r-1)(c-1) \quad (8.3)$$

위 식에서  $r$ 은 행의 수,  $c$ 는 열의 수를 의미한다.  $O_{ij}$ 는  $(i, j)$ 셀에 대해 관찰된 관측값이며  $E_{ij}$ 는 귀무가설이 참일 때  $(i, j)$ 셀에 대한 기대값이다. 식 8.2에 의해 전체 데이터의 수가  $n$ 이라 할 때  $E_{ij} = n \times P(i, j) = n \times P(i) \times P(j)$ 가 성립한다.

학생 설문 조사 데이터를 담고 있는 MASS::survey를 사용해 학생들의 성별에 따른 운동량에 차이가 있는지 독립성 검정을 해보자. 그러기 위해 먼저 survey 데이터를 살펴본다.

```
> library(MASS)
> data(survey)
> str(survey)

'data.frame': 237 obs. of 12 variables:
 $ Sex      : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
 $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
 $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
 $ W.Hnd  : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
 $ Fold    : Factor w/ 3 levels "L on R","Neither",...: 3 3 1 3 2 1 1 3 3
               3 ...
 $ Pulse   : int  92 104 87 NA 35 64 83 74 72 90 ...
 $ Clap    : Factor w/ 3 levels "Left","Neither",...: 1 1 2 2 3 3 3 3 3 3
               ...
 $ Exer    : Factor w/ 3 levels "Freq","None",...: 3 2 2 2 3 3 1 1 3 3
               ...
 $ Smoke   : Factor w/ 4 levels "Heavy","Never",...: 2 4 3 2 2 2 2 2 2 2
               ...
 $ Height: num  173 178 NA 160 165 ...
 $ M.I    : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2
               2 ...
 $ Age     : num  18.2 17.6 16.9 20.3 23.7 ...
> head(survey[c("Sex", "Exer")])

  Sex Exer
1 Female Some
2   Male None
3   Male None
4   Male None
5   Male Some
6 Female Some
```

survey 데이터에서 성별은 Sex, 운동을 얼마나 하는지는 Exer 열에 저장되어 있다. Exer의

값은 Freq(Frequently), Some, None의 3가지 레벨로 구성된 Factor이다. 성별과 운동이 독립인지를 확인해보기위해 분할표를 만들어보자.

```
> xtabs(~ Sex + Exer, data=survey)
      Exer
Sex      Freq  None  Some
Female    49    11    58
Male      65    13    40
```

chisq.test()를 사용해 검정한다.

```
> chisq.test(xtabs(~ Sex + Exer, data=survey))

Pearson's Chi-squared test

data: xtabs(~Sex + Exer, data = survey)
X-squared = 5.7184, df = 2, p-value = 0.05731
```

p값이 0.05731이므로 0.05보다 커서 ' $H_0$ : 성별과 운동은 독립이다'라는 귀무가설을 기각할 수 없는 것으로 나타났다. 통계량  $\chi^2$ 는 5.7184이었으며 자유도(Degree of Freedom)는 2였다. 이 값은 식 8.3에서 보인  $(r - 1)(c - 1) = (2 - 1)(3 - 1) = 2$ 와 같다.

### 4.3 피셔의 정확 검정(Fisher's Exact Test)

샘플 수가 작다면 chisq.test()는 경고 메시지를 내보낸다. survey 데이터에서 손글씨를 어느 손으로 쓰는지와 박수를 칠 때 어느 손이 위로 가는지간의 분할표를 xtab()으로 구하면 다음과 같이 경고 메시지가 나온다.

```
> xtabs(~ W.Hnd + Clap, data=survey)
      Clap
W.Hnd   Left  Neither  Right
Left       9        5        4
Right     29       45      143

> chisq.test(xtabs(~ W.Hnd + Clap, data=survey))

Pearson's Chi-squared test

data: xtabs(~ W.Hnd + Clap, data = survey)
```

```
X-squared = 19.2524, df = 2, p-value = 6.598e-05

Warning message:
In chisq.test(xtabs(~ W.Hnd + Clap, data = survey)) :
  Chi-squared approximation may be incorrect
```

샘플 수가 작다는 기준은 특정 짓기 어렵지만 기대빈도가 작은 셀이 전체의 20% 이상인 경우 등이 이에 해당한다. 이런 경우에는 피셔의 정확 검정을 사용한다. 통계적인 계산식에 대한 설명은 <http://dermabae.tistory.com/175>와 [http://en.wikipedia.org/wiki/Fisher's\\_exact\\_test](http://en.wikipedia.org/wiki/Fisher's_exact_test)를 참고하기 바란다.

앞서 데이터에 대해 피셔의 정확 검정을 수행해보자.

```
> fisher.test(xtabs(~ W.Hnd + Clap, data=survey))

Fisher's Exact Test for Count Data

data: xtabs(~W.Hnd + Clap, data = survey)
p-value = 0.0001413
alternative hypothesis: two.sided
```

p-value가 0.05보다 작으므로 글씨를 쓰는 손과 박수를 칠때 위에 오는 손이 독립이라는 귀무 가설을 기각하고 둘간에 의존 관계가 있다는 대립가설을 채택한다.

#### 4.4 맥니마 검정(McNemar Test)

벌금을 부과하기 시작한 후 안전 벨트 착용자의 수, 유세를 하고난 뒤 지지율의 변화와 같이 응답자의 성향이 사건 전후에 어떻게 달라지는지를 알아보는 경우 맥니마 검정을 수행한다<sup>2)</sup>.

사건 전후에 Test를 수행하여 사건발생전 Test결과를 Test1, 사건 발생 후 Test결과를 Test2라고 명시한 다음 테이블을 보자<sup>3)</sup>.

<sup>2)</sup><http://wolfdack.hnu.ac.kr/lecture/fall01/stat4bus/stat4bus9-1.pdf> 참고

<sup>3)</sup> 표 그림은 [http://en.wikipedia.org/wiki/McNemar's\\_test](http://en.wikipedia.org/wiki/McNemar's_test)에서 인용하였다.

	Test 2 positive	Test 2 negative	Row total
Test 1 positive	$a$	$b$	$a + b$
Test 1 negative	$c$	$d$	$c + d$
Column total	$a + c$	$b + d$	$n$

그림 8.2: 사건 전후의 Test 결과

사건 전 후에 Test 결과의 변화가 없다면 Test1에서의 positive와 Test2에서의 positive가 동일해야하므로  $a + b = a + c$ 가 성립해야한다. 또 Test1의 negative와 Test2에서의 negative가 동일해야하므로  $c + d = b + d$ 가 성립해야한다. 이 둘을 정리하면 결과적으로  $b = c$ 의 여부를 검토해 사건 전후에 성향 변화가 생겼는지를 알 수 있다.

$b = c$ 가 성립하려면  $b, c$ 의 값은  $b + c$ 의 절반씩이 되므로  $b$ 는 이항분포를 따른다.

$$b \sim B(b + c, \frac{1}{2}) \quad (8.4)$$

만약  $b + c$ 가 크다면 정규분포로 근사화 할 수 있다.

$$b \sim N\left(\frac{b + c}{2}, \frac{b + c}{4}\right) \quad (8.5)$$

$b$ 를 표준화하여  $N(0, 1)$ 을 따르게 하고 연속성 수정(Continuity Correction)을 하면 다음이 성립한다[31].

$$\frac{(|b - c| - 1)^2}{b + c} \sim \chi^2(1) \quad (8.6)$$

대응 표본 검정은 mcnemar.test()로 수행한다. 다음은 help(mcnemar.test)에서 가져온 예시이다.

```
> ## Agresti (1990), p. 350.
> ## Presidential Approval Ratings.
> ## Approval of the President's performance in office in two
> ## surveys, one month apart, for a random sample of 1600
> ## voting-age Americans.
> Performance <-
+   matrix(c(794, 86, 150, 570),
+           nrow = 2,
+           dimnames = list(
```

```

+           "1st Survey" = c("Approve", "Disapprove"),
+           "2nd Survey" = c("Approve", "Disapprove")))

> Performance
    2nd Survey
 1st Survey   Approve Disapprove
  Approve       794      150
  Disapprove     86       570

> mcnemar.test(Performance)

McNemar's Chi-squared test with continuity correction

data:  Performance
McNemar's chi-squared = 16.8178, df = 1, p-value = 4.115e-05

```

결과에서 p-value < 0.05 가 나타나 사건 전후에 Approve, Disapprove에 차이가 없다는 귀무가설이 기각된다. 즉, 사건 전후에 Approve, Disapprove 비율에 차이가 발생하였다.

앞서 mcnear.test()는 이항분포로부터 나왔다고 하였다. 따라서 binom.test()를 사용해 1st Survey에서의 Disapprove와 2nd Survey에서의 Disapprove가 같은 값인지 확인할 수 있다. binom.test()는 베르누이 실험에서의 성공 비율에 대한 가설검정을 수행하며, 호출 형식은 binom.test(성공횟수, 전체횟수, 확률)이다. 다음 코드에서는 86이 86+150의 절반에 해당하는지를 검정하고 있다.

```

> binom.test(86, 86 + 150)

Exact binomial test

data:  86 and 86 + 150
number of successes = 86, number of trials = 236, p-value = 3.716e-05
alternative hypothesis: true probability of success is not equal to .5
95 percent confidence interval:
 0.3029404 0.4293268
sample estimates:
probability of success
                  0.364406

```

여기에서도 p-value < 0.05로 86이 86 + 150의 절반이라는 귀무가설이 기각되었다. 즉 사건 전후에 Approve, Disapprove 성향 차이가 발생하였다.

## 5 적합도 검정(Goodness of Fit)

통계 분석에서는 종종 데이터가 특정 분포를 따름을 가정한다. 특히 데이터의 크기가 일정 수 이상이라면 데이터가 정규성을 따름을 별 의심없이 가정하기도 하지만 실제 검정을 해 볼 수도 있다.

### 5.1 Chi Square Test

데이터가 특정 분포를 따르는지 살펴보기 위해 분할표를 만들고 식 8.3을 사용할 수 있다. 다만 독립성 검정과 달리  $E_{ij}$ 를 비교하고자 하는 분포로부터 계산하는 것이 차이점이다.

survey 데이터를 사용해 글씨를 왼손으로 쓰는 사람과 오른손으로 쓰는 사람의 비율이 30% : 70%인지의 여부를 분석해보자. 귀무가설( $H_0$ )은 분할표에 주어진 관측 데이터가 주어진 분포를 따른다는 것이다.

```
> table(survey$W.Hnd)

Left Right
 18    218

> chisq.test(table(survey$W.Hnd), p=c(.3, .7))

Chi-squared test for given probabilities

data: table(survey$W.Hnd)
X-squared = 56.2518, df = 1, p-value = 6.376e-14
```

p-value < 0.05 이므로 글씨를 왼손으로 쓰는 사람과 오른손으로 쓰는 사람의 비가 30% : 70%라는 귀무 가설을 기각한다.

### 5.2 Shapiro-Wilk Test

Shapiro Wilk Test는 표본이 정규분포로 부터 추출된 것인지 테스트하기 위한 방법이다. 검정은 shapiro.test() 함수를 사용하며 이 때 귀무가설은 주어진 데이터가 정규분포로부터의 표본이라는 것이다.

```
> shapiro.test(rnorm(1000))

Shapiro-Wilk normality test

data: rnorm(1000)
W = 0.9974, p-value = 0.1052
```

p-value > 0.05 이므로 데이터가 정규 분포를 따른다는 귀무가설을 기각할 수 없다.  
 패키지 nortest에는 이외에도 Anderson-Darling Test, Pearson Chi-Square Test 등을 사용해 정규성을 검정하는 다양한 함수들이 있으니 참고하기 바란다.

### 5.3 Kolmogorov-Smirnov Test

K-S Test(Kolmogorov-Smirnov Test)는 비모수 검정(Nonparameteric Test)으로 경험적 분포 함수(Empirical Distribution Function)와 비교대상이 되는 분포의 누적분포함수(Cumulative Distribution Function)간의 최대 거리를 통계량으로 사용한다. <http://www.physics.csbsju.edu/stats/KS-test.html> 페이지의 중간에 다양한 예시 그림이 있으니 참고하기 바란다.

K-S Test는 ks.test() 함수를 사용해 수행한다. 다음은 귀무가설로 ‘주어진 두 데이터가 동일한 분포로부터 추출된 표본이다’를 놓고 검정하는 예이다. 먼저 정규 분포로부터 구한 표본끼리 비교해보자.

```
> ks.test(rnorm(100), rnorm(100))

Two-sample Kolmogorov-Smirnov test

data: rnorm(100) and rnorm(100)
D = 0.1, p-value = 0.6994
alternative hypothesis: two-sided
```

보다시피 같은 분포라는 귀무가설을 기각할 수 없었다. 다음은 정규분포와 균등 분포(Uniform Distribution)간의 비교이다.

```
> ks.test(rnorm(100), runif(100))

Two-sample Kolmogorov-Smirnov test

data: rnorm(100) and runif(100)
```

```
D = 0.56, p-value = 4.807e-14
alternative hypothesis: two-sided
```

이 경우에는 서로 다른 분포로 판단되었다.

다음은 K-S Test를 사용해 주어진 데이터가 특정 분포로부터의 표본인지 검정하는 예이다. 아래 코드의 귀무가설은 ‘주어진 데이터가 평균 0, 분산 1인 정규분포로 부터 뽑은 표본이다’라는 것이다.

```
> ks.test(rnorm(1000), "pnorm", 0, 1)

One-sample Kolmogorov-Smirnov test

data: rnorm(1000)
D = 0.0399, p-value = 0.08342
alternative hypothesis: two-sided
```

귀무가설을 기각할 수 없어, 주어진 rnorm(100)은 평균 0, 분산 1인 정규분포로 부터의 표본이라고 결론내린다.

## 5.4 Q-Q Plot

자료가 특정 분포를 따르는지를 시각적으로 검토하기 위해 [Q-Q Plot](#)을 사용한다. Q-Q Plot은 Quantile-Quantile Plot의 약자로 비교하고자 하는 분포의 분위수끼리 좌표 평면에 표시하여 그린다.

예를 들어  $X$ 가 정규분포를 따르는지 살펴보고 싶다고 가정하자.  $X \sim N(\mu, \sigma^2)$ 라면 다음 관계가 성립한다.

$$Z = \frac{X - \mu}{\sigma} \sim N(0, 1) \quad (8.7)$$

정의에 의해 다음이 성립한다.

$$X = \mu + \sigma Z \quad (8.8)$$

$X$ 가 정규분포를 따른다면 식 8.8과 같은 직선이 나타나야 한다. Q-Q Plot은 이와 같은 직선 관계가 실제로 성립하는지 시각적으로 보여주는 도구이다. 이 직선관계를 살펴보기 위해서  $(X, Z)$ 를 좌표 평면에 그리고 이들 점들이 직선이 되는지 확인하면 된다.  $(X, Z)$ 에서  $X$ 는 주어진 샘플이므로 이미 알고 있는 값이다. 따라서  $X$ 에 대항하는  $Z$ 만 찾으면 된다. 이 때 분위수가 사용된다.

$X$ 에서 관측값  $x_1 < x_2 < \dots < x_n$ 이 있을 때 이 자료들의 분포를 표현하는 경험 분포 함수 (Empirical Distribution Function)가 다음과 같이 정의되었다고 가정해보자[32]<sup>4)</sup>.

$$G(x_i) = \frac{i - 3/8}{n + 1/4} \quad (8.9)$$

이 식은 샘플 데이터를 정렬했을 때  $i$  번째 데이터가 몇 % 분위수인지 알려주는 역할을 한다. 예를 들어 샘플의 크기  $n=20$ 이라면  $G(x_1) = (1 - 3/8)/(20 + 1/4) = 0.03$ ,  $G(x_2) = (2 - 3/8)/(20 + 2/4) = 0.07$ , ... 이 된다. 따라서  $x_1$ 은  $X$ 가 따르는 분포의 3% 분위수,  $x_2$ 는 7% 분위수이다.

$X$ 가 몇 % 분위수인지 알면  $Z$ 는 손쉽게 찾을 수 있다.  $Z$ 가 표준 정규 분포를 따르므로 3% 분위수인  $z_1$ , 7% 분위수인  $z_2$  등은  $Z$ 의 누적 분포 함수가  $\Phi$ 라 할 때  $z_1 = \Phi^{-1}(0.03)$ ,  $z_2 = \Phi^{-1}(0.07)$ 이다. 일반적으로  $X$ 가 정규분포를 따른다는 가정하에서 다음이 성립한다[32].

$$x_i = \mu + \sigma z_i = \mu + \sigma \Phi^{-1}(G(x_i)) \quad (8.10)$$

이제  $X$ 에 해당하는  $Z$ 를 찾았으니  $(X, Z)$ 를 그려볼 차례이다. 이 목적으로는 `qqnorm()` 함수를 사용한다. `qqline()`은 Q-Q Plot에서 데이터가 만족해야 하는 직선 관계를 그린다. 다음은  $N(10, 1)$ 의 정규분포로부터 1000개 크기의 샘플을 뽑아 Q-Q Plot을 그리는 예이다.

```
> x <- rnorm(1000, mean=10, sd=1)
> qqnorm(x)
> qqline(x, lty=2)
```

아래 그림은 위 코드의 실행결과로 그려진 Q-Q Plot이다. 정규 분포에 대한 그림이므로 정규화률플롯(Normal Q-Q Plot)이라고 한다. 그림에서 볼 수 있듯이 직선 관계가 잘 성립된다.

<sup>4)</sup> 참고문헌에 설명되어 있듯이 이 함수를  $i/n$ 으로 하지 않는 이유는  $i/n$ 은 분포함수의 일반적인 대칭성, 즉  $G(x_1) = 1 - G(x_n)$ 을 만족시키지 못하기 때문이다. R의 `qqnorm`은 `ppoints()`를 사용해 % 분위수를 판단한다. 콘솔에서 `ppoints`를 입력해 소스 코드를 확인해볼 수 있다.

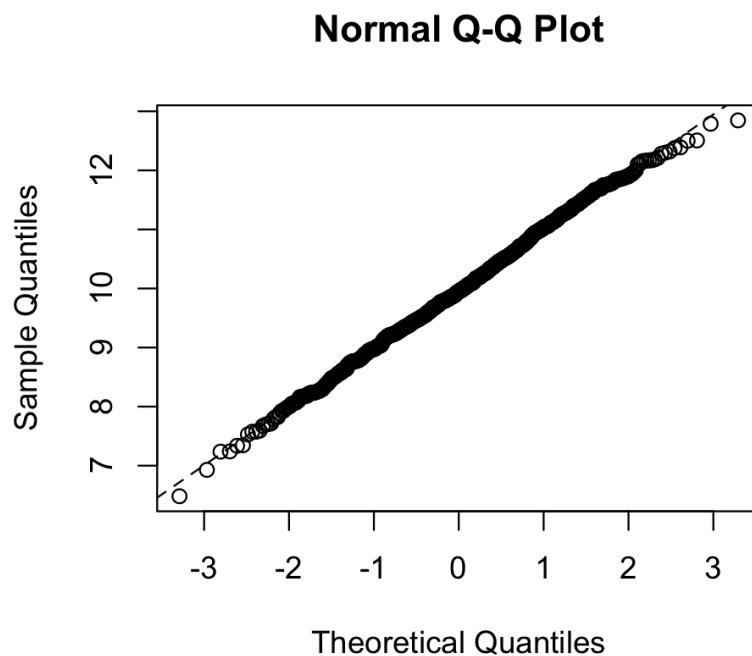


그림 8.3: `rnorm()` 데이터에 대한 정규화율플롯

비교를 위해 `rcauchy()`로부터 생성한 데이터로 정규화율플롯을 그려보자. 결과를 보면 한 눈에도 직선 관계가 성립하지 않음이 보인다.

```
> x <- rcauchy(1000)
> qqnorm(x)
> qqline(x, lty=2)
```

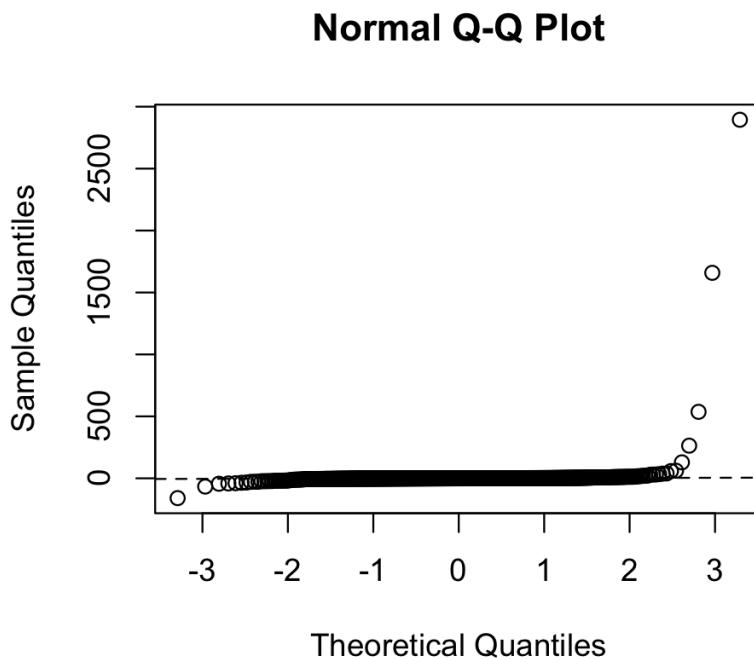


그림 8.4: rcauchy() 데이터에 대한 정규화률플롯

그러나 데이터의 정규성이 Q-Q Plot을 통해 항상 명확히 판단되는 것은 아니다. 어디서 구한 데이터이고 데이터가 정규성을 따를 이유가 있는지에 대한 고민이 더 필요하고, Q-Q Plot은 보조적으로 쓸 수 있는 방법이다.

정규화률플롯이 아닌 다른 분포에 대해서는 `qqplot()` 함수를 쓰거나 `car::qqPlot()` 함수를 사용한다.

## 6 상관 계수

상관계수는 두 확률 변수 사이의 관계를 파악하는 방식으로, 흔히 상관계수라고하면 피어슨 상관계수를 뜻한다.

상관 계수 값이 크면 데이터간의 연관 관계가 존재한다는 의미이다. 그러나 이것이 반드시 인과관계를 뜻하는 것은 아니다. A가 B를 야기한다고 판단했으나 실제로는 C<sup>5)</sup>가 B를 야기하고 있거나, A가 B의 원인이라고 예상했지만 실제로는 B가 A의 원인일 수도 있기 때문이다. 위키피디아에 [Correlation does not imply Causation](#)라는 제목의 글을 참고하기 바란다.

두 확률 변수간 다음 식이 성립하면 독립이라고 부른다.

$$P(X, Y) = P(X)P(Y) \quad (8.11)$$

<sup>5)</sup>교락변수(Confounding Variable)라고 부름.

변수가 서로 독립이라면 변수간 상관 계수는 0이다. 그러나 상관 계수가 0이라고 해서 두 변수가 독립임을 의미하지는 않는다. 변수간 독립성 확인은 앞서 [독립성 검정\(Independence Test\)](#) (페이지 233)에서 살펴 본 Chi Square Test를 사용할 수 있다.

## 6.1 피어슨 상관계수(Pearson Correlation Coefficient)

피어슨 상관계수는 [-1, 1] 사이의 값을 가진다. 0보다 큰 상관 계수 값은 한 변수가 커지면 다른 변수도 큰 값을 갖게 됨을 뜻하고, 음의 상관계수는 한 변수가 커지면 다른 변수가 작은 값을 갖게 됨을 뜻한다. 피어슨 상관계수는 선형 관계를 판단한다. 따라서 비선형 관계 (예를 들어  $Y = aX^2 + b$  형태)에 대해서는 제대로 판별하지 못할 수 있다..

피어슨 상관계수는 다음과 같이 정의된다.

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sigma_X \sigma_Y} = \frac{E[(X - E(X))(Y - E(Y))]}{\sigma_X \sigma_Y} \quad (8.12)$$

식에서  $cov(X, Y)$ 는  $X, Y$ 의 공분산,  $\sigma_X, \sigma_Y$ 는  $X, Y$ 의 표준편차이다.

피어슨 상관계수는 cor() 함수를 사용해 계산한다. 다음은 iris데이터에서 Sepal.Width, Sepal.Length의 피어슨 상관계수를 구하는 예이다.

```
> cor(iris$Sepal.Width, iris$Sepal.Length)
[1] -0.1175698
```

iris에서 Species를 제외한 모든 열의 피어슨 상관계수를 구해보자.

```
> cor(iris[, 1:4])
      Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000 -0.1175698   0.8717538   0.8179411
Sepal.Width     -0.1175698  1.0000000  -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401   1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259   0.9628654   1.0000000
```

만약 살펴봐야 할 열의 수가 많다면 한눈에 숫자가 잘 안들어올 수 있다. symnum() 함수는 특정 범위의 값을 문자로 치환하여 보기 쉽게 표시해준다.

```
> symnum(cor(iris[, 1:4]))
      S.L S.W P.L P.W
Sepal.Length 1
Sepal.Width   1
Petal.Length + .   1
Petal.Width   + .   B   1
```

```
attr("legend")
[1] 0 ` ' 0.3 ` . ' 0.6 ` , ' 0.8 ` +' 0.9 ` * ' 0.95 ` B ' 1
```

위에서 [0, 0.3)은 표시되지 않고, [0.3, 0.6)은 '.', [0.6, 0.8)은 ',', 등으로 표시되었음을 알 수 있다.

corrgram 패키지는 상관계수를 시각화하는데 유용한 패키지이다. 다음 코드는 iris의 상관계수를 그림의 우상단에 배치하고, 대각선에는 컬럼의 이름을 적고, 좌하단에는 상관계수를 그림으로 표현하는 예이다. 그림에서 파란색은 양의 상관계수를 뜻하고 빨간색은 음의 상관계수를 뜻한다. 색의 짙기는 상관계수의 크기를 뜻해 절대 값이 큰 상관계수일수록 더 짙은 파란색이나 더 짙은 빨간색을 띤다.

```
> install.packages("corrgram")
> library(corrgram)
> corrgram(cor(iris[,1:4]), type="corr", upper.panel=panel.conf)
```

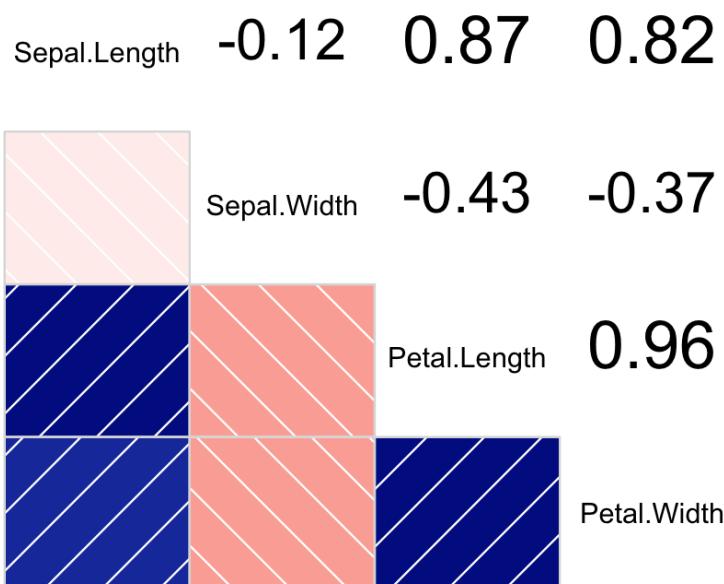


그림 8.5: iris 데이터로 그린 corrgram

앞서 피어슨 상관계수는 데이터의 선형 관계를 판단한다고 했다. 이 말의 의미는  $Y = X$ ,  $Y = 2X$  모두 같은 피어슨 상관 계수 1을 갖는다는 뜻이다. 피어슨 상관계수에서는 선형관계가 성립하면 1, 성립하지 않으면 1이 아닌 값을 갖게 된다. 다음 코드는  $Y = X$ 와  $Y = 2X$ 의 경우 모두 피어슨 상관계수가 1임을 보여준다.

```
> cor(1:10, 1:10)
```

```
[1] 1
> cor(1:10, 1:10*2)
[1] 1
```

## 6.2 스피어만 상관계수(Spearman's Rank Correlation Coefficient)

[스피어만 상관계수](#)는 상관계수를 계산할 두 데이터의 실제값 대신 두 값의 순위를 사용해 상관계수를 비교하는 방식이다. 계산 방법이 피어슨 상관계수와 유사해 이해가 쉽고, 피어슨 상관계수와 달리 비선형 관계의 연관성을 파악할 수 있다는 장점이 있다. 또한 순위만 매길 수 있다면 적용이 가능하므로 연속형(Continous) 데이터에 적합한 피어슨 상관계수와 달리 이산형(Discrete) 데이터, 순서형(Ordinal) 데이터에 적용이 가능하다.

예를 들어 국어 점수와 영어 점수간의 상관계수는 피어슨 상관계수로 계산할 수 있고, 국어 성적 석차와 영어 성적 석차의 상관계수는 스피어만 상관계수로 계산 가능하다.

스피어만 상관계수는 다음과 같이 정의한다.

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2} \sqrt{\sum_i (y_i - \bar{y})^2}} \quad (8.13)$$

위 식에서  $x_i$ 는  $X_i$ 의 순위,  $y_i$ 는  $Y_i$ 의 순위,  $\bar{x}$ 와  $\bar{y}$ 는 각각  $x_i$ 와  $y_i$ 의 평균을 뜻한다.

예를 들어 데이터 3, 4, 5, 3, 2, 1, 7, 5를 가정해보자. 이 데이터를 정렬하면 1, 2, 3, 3, 4, 5, 5, 7이 된다. 또 각 값의 순위는 1, 2, 3.5, 3.5, 5, 6.5, 8이 된다. 3, 3은 순위 3, 4에 해당하므로 이들의 평균인 3.5가 순위로 주어지고 5, 5는 순위가 6, 7위 이므로 평균인 6.5가 순위로 주어진 것이다. R 코드를 사용해 순위를 확인해보자<sup>6)</sup>.

```
> x <- c(3, 4, 5, 3, 2, 1, 7, 5)
> rank(sort(x))
[1] 1.0 2.0 3.5 3.5 5.0 6.5 6.5 8.0
```

이처럼 비교할 데이터로부터 순위를 구한 다음 식 8.13을 적용하면 된다. 이 모든 작업을 한번에 해주는 함수는 Hmisc::rcorr()이다. rcorr()을 사용해 피어슨 상관계수와 스피어만 상관계수를 비선형 관계의 데이터에 적용한 다음 코드를 살펴보자.

```
> m <- matrix(c(1:10, (1:10)^2), ncol=2)
> m
     [,1] [,2]
[1,]    1    1
```

<sup>6)</sup>rank() 함수를 사용하기 위해 sort()를 할 필요는 없다. 보기 편하게 하기 위해 정렬한 것일 뿐이다.

```
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
[9,]    9   81
[10,]   10  100
> rcorr(m, type="pearson")$r
      [,1]      [,2]
[1,] 1.0000000 0.9745587
[2,] 0.9745587 1.0000000
> rcorr(m, type="spearman")$r
      [,1] [,2]
[1,]    1    1
[2,]    1    1
```

보다시피 두 컬럼간 피어슨 상관계수는 0.97이었지만 스피어만 상관계수는 1.00으로 나타났다.

이 절의 시작에서 확률 변수간 독립이면 상관 계수가 0이지만 상관 계수가 0이라고해서 독립이지는 않음을 설명했다. 다음에 보인 코드에서는  $Y = X^2$  의 명확한 의존 관계가 있는  $X$ ,  $Y$ 간 상관계수가 0임을 보여준다.

```
> x <- -5:5
> y <- (-5:5)^2
> x
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> y
[1] 25 16  9  4  1  0  1  4  9 16 25
!!!!!! TODO !!!!!
```

### 6.3 켄달의 순위 상관 계수(Kendal's Rank Correlation Coefficient)

켄달의 순위 상관 계수는  $(X, Y)$  형태의 순서쌍으로 데이터가 있을 때  $x_i < x_j$ ,  $y_i < y_j$ 가 성립하면 concordant,  $x_i < x_j$  이지만  $y_i > y_j$ 이면 discordant라고 정의한다. 즉,  $x$ 가 클 때  $y$ 도

크면 concordant,  $x$ 가 크지만  $y$ 는 작다면 discordant로 보는 것이다.

肯달의 순위 상관계수는 다음과 같이 정의된다.

$$\tau = \frac{(\text{number of concordant pairs} - \text{number of discordant pairs})}{\frac{1}{2}n(n-1)} \quad (8.14)$$

분모는 모든  $(x_i, y_i), (x_j, y_j)$ 의 조합의 수이다. 따라서 위 식은 concordant과 discordant에 비해 얼마나 많은지 그 비율을 보는 것이다.

다음 예를 살펴보자.

```
> install.packages("Kendall")
> library(Kendall)
> Kendall(c(1, 2, 3, 4, 5), c(1, 0, 3, 4, 5))
tau = 0.8, 2-sided pvalue = 0.086411
```

## 6.4 상관 계수 검정(Correlation Test)

`corr.test()`를 사용해 상관 계수의 유의성을 판단할 수 있다. 이 때 귀무가설은 다음과 같다. ‘ $H_0$ : 상관 계수가 0이다.’

$c(1, 2, 3, 4, 5)$ 와  $c(1, 0, 3, 4, 5)$ 간의 피어슨 상관 계수, 스피어만 상관 계수,肯달의 상관 계수에 대해 검토해보자.

```
> cor.test(c(1, 2, 3, 4, 5), c(1, 0, 3, 4, 5), method="pearson")

Pearson's product-moment correlation

data: c(1, 2, 3, 4, 5) and c(1, 0, 3, 4, 5)
t = 3.9279, df = 3, p-value = 0.02937
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
0.1697938 0.9944622
sample estimates:
cor
0.9149914

> cor.test(c(1, 2, 3, 4, 5), c(1, 0, 3, 4, 5), method="spearman")

Spearman's rank correlation rho
```

```

data: c(1, 2, 3, 4, 5) and c(1, 0, 3, 4, 5)
S = 2, p-value = 0.08333
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.9

> cor.test(c(1, 2, 3, 4, 5), c(1, 0, 3, 4, 5), method="kendall")

Kendall's rank correlation tau

data: c(1, 2, 3, 4, 5) and c(1, 0, 3, 4, 5)
T = 9, p-value = 0.08333
alternative hypothesis: true tau is not equal to 0
sample estimates:
tau
0.8

```

피어슨 상관계수에서만 p-value 가 0.05보다 작아 상관 관계가 유의한것으로 나타났다. 만약 단측 검정을 원한다면 cor.test()에 인자로 alternative="greater" 또는 alternative="less"를 주어 상관 계수가 양의 값 또는 음의 값인지를 볼 수 있다.

이처럼 세가지 상관 계수의 같은 서로 다른 값이 될 수 있다. 이런 경우 더 작은 숫자를 사용하는 것이 바람직하다. 위의 경우라면 켄달의 상관 계수 0.8이 가장 작은 값이므로 이 값을 사용한다. 또, 세가지 값이 계산하는 것이 무엇인지 비교하여 사용할 필요가 있다<sup>7)</sup>.

## 7 추정 및 검정

R을 사용해 데이터의 추정 및 검정을 할 수 있다는 사실은 정말로 놀라운 일이다. 많은 수고가 필요한 일을 간단한 함수 호출만으로 해결해주시니 말이다.

이 절에서는 간략한 분포의 성질에 대한 설명과 함께 모집단의 평균, 분산, 비율을 추정하고 비교하는 방법을 일표본(확률 변수가 1개)인 경우와 이표본(확률 변수가 2개)인 경우로 나누어 살펴본다. 이 절의 내용에서 이론적 배경 부분은 위키피디아와 참고 도서[20, 21, 22]에서 결과만 추린 내용이다. 그러므로 부족한 통계적 설명은 언급한 참고 도서를 보기 바란다.

<sup>7)</sup>[http://www.unesco.org/webworld/idams/advguide/Chapt4\\_2.htm](http://www.unesco.org/webworld/idams/advguide/Chapt4_2.htm) 참고

## 7.1 일표본 평균

### 이론적 배경

확률 변수  $X_1, X_2, \dots, X_n$  가 서로 독립이고  $N(\mu, \sigma^2)$  를 따른다고 하자. 즉  $X_i$  는  $N(\mu, \sigma^2)$ 로부터 구한 표본(sample)이며 관찰 대상인 표본의 갯수는  $n$  이다. 이때 다음이 성립한다.

$$\frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \sim N(0, 1) \quad (8.15)$$

위에서 사용한  $\sigma^2$  는 모집단의 분산을 뜻한다. 이 값이 알려져 있으면 좋지만 보통은 미지의 값이다. 따라서 다음에 정의된 표본 분산을  $\sigma^2$  대신 사용한다.

$$S^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - E(X))^2 \quad (8.16)$$

식 8.15에  $S^2$  을 사용하면  $t$  분포를 따르게 된다. 이 때 자유도는  $n - 1$  이다.

$$\frac{\bar{X} - \mu}{S/\sqrt{n}} \sim t(n-1) \quad (8.17)$$

따라서 모평균에 대한 95% 신뢰 수준의 신뢰 구간은  $\alpha = 0.05$  일 때 다음과 같다.

$$(\bar{X} - t(n-1; \alpha/2)S/\sqrt{n}, \bar{X} + t(n-1; \alpha/2)S/\sqrt{n}) \quad (8.18)$$

위 식에서  $t(n-1; \alpha/2)$  는 자유도  $n-1$  인  $t$  분포의  $100(1-\alpha/2)\%$  분위수를 뜻한다. 즉, 다음이 성립한다.

$$P(t \geq t(n; \alpha)) = \alpha \quad (8.19)$$

### 추정 및 검정의 예

다음은  $N(0, 1)$ 로부터 30개의 표본을 뽑은 뒤 모평균의 구간을 추정한 예이다.

```
> x <- rnorm(30)
> x
[1] -0.031730453 -0.589826570  1.575581040 -0.146396717 -0.328761466
[6] -0.620959988  0.422465776  0.305835299 -0.314972917 -0.881650165
[11]  0.698445951 -0.293486558 -1.299123995  0.018505730 -0.510159586
[16]  1.398750247 -0.164320432  1.052527533 -0.017473444  0.415540231
[21] -0.815439177 -1.044640606  1.002270966 -0.594042081 -0.003546248
[26]  0.179795860  1.267979880 -0.028501269 -1.176802619 -1.860904735
> t.test(x)
```

## One Sample t-test

```

data: x
t = -0.5284, df = 29, p-value = 0.6012
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
-0.3872198 0.2282171
sample estimates:
mean of x
-0.07950135

```

실행 결과 모평균은 -0.07950135, 모평균의 95% 신뢰 구간은 (-0.3872198, 0.2282171)로 추정되었다. `t.test()`는 구간 추정과 가설 검증을 동시에 수행한다. 기본 인자에 의한 귀무가설은 ' $H_0$ : 모평균이 0이다'라는 것이고  $p\text{-value} > 0.05$ 이며 신뢰구간에 0이 포함돼 귀무가설을 기각하지 못한다. 즉, 모평균은 0으로 보인다.

다음은  $N(10, 1)$ 에서 30개의 표본을 뽑아 모평균의 구간을 추정한 예이다.

```

> x <- rnorm(30, mean=10)
> t.test(x, mu=10)

One Sample t-test

data: x
t = 0.1694, df = 29, p-value = 0.8666
alternative hypothesis: true mean is not equal to 10
95 percent confidence interval:
 9.688162 10.368171
sample estimates:
mean of x
10.02817

```

표본 평균은 10.02817, 평균의 신뢰구간은 (9.688162, 10.368171)이다.  $p\text{-value} > 0.05$ 이며 신뢰구간이 10을 포함하므로 귀무가설  $H_0 : \mu = 10$ 을 기각하지 못한다.

이 절의 방법은 데이터가 정규분포로 부터 나온 것임을 가정하고 식 8.17를 사용해 구간 추정을 수행했다. 위에 보인 예제 코드들에서는 `rnorm()`을 사용하여 데이터를 생성하였으므로 데이터가 정규분포로 부터 나온 표본임이 보장되었다. 그러나 데이터가 정규분포를 따르는지

불명확한 경우에는 Shapiro-Wilk Test (페이지 239) 또는 Q-Q Plot (페이지 241)을 사용해 데이터의 정규성을 검토할 수 있다.

## 7.2 독립 이표본 평균

### 이론적 배경

독립 이표본은 서로 독립인 두개의 표본 집단이 있는 경우를 지칭한다.  $X_1, X_2, \dots, X_m \sim N(\mu_x, \sigma_x^2)$ 로 부터의 표본이고  $Y_1, Y_2, \dots, Y_n \sim N(\mu_y, \sigma_y^2)$ 로 부터의 표본이라고 하자.  $X$ 와  $Y$ 는 독립이다.

$\bar{X}, \bar{Y}$ 는 다음의 정규 분포를 따른다.

$$\bar{X} \sim N(\mu_x, \frac{\sigma_x^2}{m}) \quad (8.20)$$

$$\bar{Y} \sim N(\mu_y, \frac{\sigma_y^2}{n}) \quad (8.21)$$

$X$ 와  $Y$ 가 독립이므로 다음이 성립한다.

$$\bar{X} - \bar{Y} \sim N(\mu_x - \mu_y, \frac{\sigma_x^2}{m} + \frac{\sigma_y^2}{n}) \quad (8.22)$$

$$\frac{\bar{X} - \bar{Y} - (\mu_x - \mu_y)}{\sqrt{\sigma_x^2/m + \sigma_y^2/n}} \sim N(0, 1) \quad (8.23)$$

일표본 평균의 경우와 마찬가지로  $\sigma_x^2, \sigma_y^2$  모두 모 집단의 분산이고 이 값은 보통 미지의 값이다. 따라서 표본 분산을 대신 사용하게 된다.

표본 분산을 구함에 있어 이표본의 경우에는  $X, Y$ 의 모분산이 같은 경우(즉,  $\sigma_x = \sigma_y$ )와  $X, Y$ 의 모분산이 다른 경우( $\sigma_x \neq \sigma_y$ )를 나누어 생각한다. 여기서는 모분산이 같은 경우만 알아보도록하자.

$\sigma_x = \sigma_y$ 라면<sup>8)</sup> 합동 표본 분산(Pooled Sample Variance)  $S_p^2$ 을 구하여 사용한다.

$$S_p^2 = \frac{(m-1)S_x^2 + (n-1)S_y^2}{m+n-2} \quad (8.24)$$

$S_p$ 를 식 8.23에 대입하면 다음과 같이 자유도가  $m+n-2$ 인 t 분포를 따르게 된다.

$$\frac{\bar{X} - \bar{Y} - (\mu_x - \mu_y)}{S_p \sqrt{1/m + 1/n}} \sim t(m+n-2) \quad (8.25)$$

<sup>8)</sup> 뒤에서 다룰 F-test를 사용해 실제로 분산의 차이가 있는지를 검정할 수 있다.

따라서 두개 표본으로부터 구한 모평균 차이의 95% 신뢰구간은  $\alpha = 0.05$  일 때 다음과 같다.

$$\begin{aligned} & (\bar{X} - \bar{Y} - t(m+n-2; \alpha/2)S_p\sqrt{1/m+1/n}, \\ & \bar{X} - \bar{Y} + t(m+n-2; \alpha/2)S_p\sqrt{1/m+1/n}) \end{aligned} \quad (8.26)$$

### 추정 및 검정의 예

`help(t.test)`에 있는 예제를 살펴보자. `t.test()` 예제는 다음에 보인 sleep 데이터를 사용하는데 각 컬럼의 의미는 다음과 같다.

- extra: 수면 시간의 증가량
- group: 사용한 수면제의 종류
- ID: 환자 식별 번호

다음은 sleep 데이터 전체의 내용이다.

```
> sleep
   extra group ID
1    0.7     1  1
2   -1.6     1  2
3   -0.2     1  3
4   -1.2     1  4
5   -0.1     1  5
6    3.4     1  6
7    3.7     1  7
8    0.8     1  8
9    0.0     1  9
10   2.0     1 10
11   1.9     2  1
12   0.8     2  2
13   1.1     2  3
14   0.1     2  4
15  -0.1     2  5
16   4.4     2  6
17   5.5     2  7
18   1.6     2  8
19   4.6     2  9
```

20 3.4 2 10

위 데이터에는 예를들어 1번 수면제를 사용했을때 환자1의 수면시간 증가량이 0.7이었고, 2번 수면제를 사용했을때 환자 1의 수면시간 증가량이 1.9였음을 보여준다. 이처럼 동일한 대상에 대해 서로 다른 처치를 하였을 때의 비교는 뒤에서 설명할 [깍지은 이표본 평균](#) (페이지 257) 형태이다. 그러나 여기서는 독립 이표본을 살펴보기 위해 이 데이터에 환자 식별번호가 없다고 가정해 보자.

```
> sleep2 <- sleep[, -3]
> sleep2
  extra group
 1     0.7     1
 2    -1.6     1
 3    -0.2     1
 4    -1.2     1
 5    -0.1     1
 6     3.4     1
 7     3.7     1
 8     0.8     1
 9     0.0     1
10    2.0     1
11    1.9     2
12    0.8     2
13    1.1     2
14    0.1     2
15   -0.1     2
16    4.4     2
17    5.5     2
18    1.6     2
19    4.6     2
20    3.4     2
```

이제 이 데이터의 의미가 달라진다. 첫째행은 어떤 환자인지는 모르지만 누군가가 수면제 1을 복용했더니 수면시간이 0.7 증가했다는 의미이다. 마찬가지로 11번 행 역시 누군가에게 수면제 2를 투여했더니 수면시간이 1.9 증가하였다는 의미이다. 바로 이러한 관찰 결과가 독립 이표본 검정에 해당한다.

수면제별 수면시간 증가량의 평균을 계산해보자. 다음은 [tapply](#) (페이지 97)를 사용한 예이다.

```
> tapply(sleep2$extra, sleep2$group, mean)
  1     2
0.75 2.33
```

[doBy 패키지](#) (페이지 99)의 `summaryBy()`를 사용할 수도 있다.

```
> library(doBy)
> summaryBy(extra ~ group, sleep2)
  group extra.mean
  1      1      0.75
  2      2      2.33
```

이 장에서는 모분산이 같은 경우만 살펴보기로 했으므로 모분산이 같은지 먼저 검정한다. 분산의 비교는 뒤에 [이표본 분산](#) (페이지 259)에서 다루기로하고, 여기서는 `var.test()`의 결과만 활용하자.

```
> var.test(extra ~ group, sleep2)

F test to compare two variances

data: extra by group
F = 0.7983, num df = 9, denom df = 9, p-value = 0.7427
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
0.198297 3.214123
sample estimates:
ratio of variances
0.7983426
```

`p-varlu`가 0.05보다 커서 귀무가설( $H_0$ : 분산의 비가 1이다)을 기각할 수 없다. 또는 신뢰구간의 개념을 활용해 분산비의 95% 신뢰구간이 (0.19827, 3.214123)으로 그 안에 1이 포함되어 분산의 비가 1임을 반박할 증거가 없다고 읽어도 된다.

`t.test()`를 적용해보자. `t.test()`의 주요 인자에는 `paired`, `var.equal`가 있다. `paired=FALSE`는 독립 이표본 검정을 뜻하고 `paired=TRUE`는 짹지은 이표본 검정을 뜻한다. `var.equal`은 두 집단의 모분산이 같은지의 여부를 뜻한다.

```
> t.test(extra ~ group, data=sleep2, paired=FALSE, var.equal=TRUE)

Two Sample t-test

data: extra by group
t = -1.8608, df = 18, p-value = 0.07919
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-3.363874 0.203874
sample estimates:
mean in group 1 mean in group 2
0.75          2.33
```

분석 결과  $p\text{-value} > 0.05$  이므로 ' $H_0$ : 모평균에 차이가 없다'는 가설을 기각할 수 없다. 마찬가지로 신뢰구간이 0을 포함해 평균에 차이가 없다고 읽어도 된다.

### 7.3 짹지은 이표본 평균

#### 이론적 배경

쫙지은 이표본은 두 개 표본이 짹지은 순서쌍처럼 구해진 경우이다. 예를들어 다이어트 약의 효과를 보기 위해 50 명의 표본을 조사하는데  $i$  번째 사람에 대해  $X_i$ 에는 약물 섭취전의 체중,  $Y_i$ 에는 약물 섭취 후의 체중을 측정해  $(X_i, Y_i)$  형태로 기록했다면 짹지은 이표본에 해당한다. 이처럼 짹지은 이표본은 특정 기능을 넣기 전과 후와 같이 실험 데이터의 분석에 직접 활용할 수 있어 유용하다.

쫙지은 이표본  $(X_1, Y_1), (X_2, Y_2), \dots, (X_n, Y_n)$ 이 있을 때  $D = X - Y$ 가 정규분포를 따른다고 가정하자. 그러면  $\bar{D}$ 는 다음의 정규분포를 따른다.

$$\bar{D} \sim N(\mu_D, \sigma_D^2/n) \quad (8.27)$$

그러나  $\sigma_D$ 는 모분산이므로 미지의 값이다. 이를 표본 분산으로 치환하면 t 분포를 따르게 된다.

$$\frac{\bar{D} - \mu_D}{S_D/\sqrt{n}} \sim t(n-1) \quad (8.28)$$

$\bar{D}$ 의 95% 신뢰구간은  $\alpha = 0.05$  일때 다음과 같다.

$$(\bar{D} - t(n-1; \alpha/2)S_D/\sqrt{n}, \quad \bar{D} + t(n-1; \alpha/2)S_D/\sqrt{n}) \quad (8.29)$$

### 추정 및 검정의 예

일표본 평균의 [추정 및 검정의 예](#) (페이지 254)에서 살펴본 sleep 데이터를 사용하자. 그룹별 평균을 구하는 방법도 해당 섹션을 참고하기 바란다.

sleep 데이터는 다음과 같이 수면제1(group == 1)과 수면제2(group==2) 각각에 대해 환자 ID가 오름차순으로 정렬되어 있다. 따라서 수면제별로 데이터를 잘라냈을때 수면제1과 수면제2의 환자가 동일한 순서로 오게된다.

```
> sleep
   extra group ID
1     0.7     1   1
2    -1.6     1   2
3    -0.2     1   3
4    -1.2     1   4
5    -0.1     1   5
6     3.4     1   6
7     3.7     1   7
8     0.8     1   8
9     0.0     1   9
10    2.0     1  10
11    1.9     2   1
12    0.8     2   2
13    1.1     2   3
14    0.1     2   4
15   -0.1     2   5
16    4.4     2   6
17    5.5     2   7
18    1.6     2   8
19    4.6     2   9
20    3.4     2  10
```

t.test()에 paired=TRUE를 지정해 짹지은 이표본 검정을 수행해보자. 앞서 설명한대로 그룹별로 데이터를 잘라내었을때 1, 2, 3, ..., 10 환자 순서로 t.test()의 인자로 넘겨지고 있다.

```
> with(sleep, t.test(extra[group==1], extra[group==2], paired=TRUE))

Paired t-test

data: extra[group == 1] and extra[group == 2]
t = -4.0621, df = 9, p-value = 0.002833
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-2.4598858 -0.7001142
sample estimates:
mean of the differences
-1.58
```

p-value < 0.05이므로 귀무가설 ' $H_0$ : 모평균의 차이가 0 이다'를 기각한다. 따라서 두 수면제의 수면시간 연장 정도가 다르다고 결론을 내린다.

이 결론은 sleep 데이터를 독립 이표본으로 본 경우와 다른 결과이다. 독립 이표본의 경우에는 신뢰구간이 (-3.363874, 0.203874)로 구간안에 0을 가까스로 포함한 형태였다. 짹지는 이표본 검정의 경우 독립 이표본 검정에 비해 추정의 정확도가 높아<sup>9)</sup> 신뢰구간이 좁아지면서 신뢰구간에서 0이 빠지게 되었다. 그 결과 수면제간 수면시간 연장정도에 차이가 존재함을 보이게 되었다.

## 7.4 이표본 분산

### 이론적 배경

일변수의 경우는 생략하기로 하고 여기서는 두개의 확률변수가 있을때 모분산을 비교하는 방법에 대해서 살펴본다. 확률변수  $X, Y$ 가 독립이며  $X \sim N(\mu_X, \sigma_X^2)$ ,  $Y \sim N(\mu_Y, \sigma_Y^2)$ 라 하자.  $m$ 은  $X$ 에서의 표본의 수,  $n$ 은  $Y$ 에서의 표본의 수라 할 때 다음이 성립한다.

$$\frac{S_X^2/\sigma_X^2}{S_Y^2/\sigma_Y^2} \sim F(m-1, n-1) \quad (8.30)$$

따라서 모분산 비에대한 95% 신뢰구간은  $\alpha = 0.05$ 라 할 때 다음과 같다.

$$\left( \frac{S_Y^2}{S_X^2} \frac{F(m-1, n-1; \alpha/2)}{F(n-1, m-1; \alpha/2)} \right) \quad (8.31)$$

---

<sup>9)</sup>추정의 정확도가 높다는 말은 추정치에 대한 신뢰구간이 좁다는 의미이고, 다시 말해 추정치에 대한 분산이 작다는 뜻이다.

## 추정 및 검정의 예

분산의 비교는 `var.test()` 함수를 사용한다. `iris`의 `Sepal.Width`와 `Sepal.Length`가 같은지 `var.test()`를 사용하여 검정해보자.

```
> with(iris, var.test(Sepal.Width, Sepal.Length))

F test to compare two variances

data: Sepal.Width and Sepal.Length
F = 0.2771, num df = 149, denom df = 149, p-value = 3.595e-14
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
0.2007129 0.3824528
sample estimates:
ratio of variances
0.2770617
```

수행결과 `p-value`가 매우작게 나타났다. 따라서 모분산에 차이가 없다는 귀무가설을 기각한다.

## 7.5 일표본 비율

### 이론적 배경

베르누이 시행을  $n$ 회 수행하여  $X$  번 성공을 관찰하였다고하자. 일표본 비율에서는 다음식에서  $p$ 를 구하는 것이 목표이다.

$$X \sim B(n, p) \quad (8.32)$$

$n$ 이 크면 이를 정규분포로 근사할 수 있게 된다.

$$X \sim N(np, np(1-p)) \quad (8.33)$$

모비율에 대한 추정값  $\hat{p}$ 은  $X/n$ 로 계산할 수 있으므로 위 식의 양변을  $n$ 으로 나누면 다음과 같다.

$$\hat{p} \sim N(p, (1-p)/n) \quad (8.34)$$

따라서  $\alpha = 0.05$  라 할 때 95% 신뢰 구간은 다음과 같다.

$$(\hat{p} - z_{\alpha/2} \sqrt{\hat{p}(1-\hat{p})/n}, \hat{p} + z_{\alpha/2} \sqrt{\hat{p}(1-\hat{p})/n}) \quad (8.35)$$

## 추정 및 검정의 예

비율은 `prop.test()`를 사용하여 검정한다. 동전을 100번 던졌을 때 앞면이 42번 나왔다고 하자. 이 때 동전의 앞면이 나오는 비율이 50%라고 할 수 있을까? 이는 `prop.test()` 함수로 확인할 수 있다.

```
> prop.test(42, 100)

1-sample proportions test with continuity correction

data: 42 out of 100, null probability 0.5
X-squared = 2.25, df = 1, p-value = 0.1336
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
0.3233236 0.5228954
sample estimates:
p
0.42
```

수행 결과  $p$ -value  $> 0.05$ 이고 신뢰구간이 0.5를 포함해 이 동전의 앞면이 나오는 확률이 0.5라는 귀무가설을 기각할 수 없다.

비율 구간 추정 또는 검정 시 반드시 정규분포 등으로 근사를 해야하는 것은 아니다. `binom.test()` 를 사용하면 이항분포에서 신뢰구간을 계산한다.

```
> binom.test(42, 100)

Exact binomial test

data: 42 and 100
number of successes = 42, number of trials = 100, p-value = 0.1332
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
0.3219855 0.5228808
```

```
sample estimates:
probability of success
0.42
```

이항분포를 통한 정확한 계산의 경우에도 동전의 앞면이 나올 확률이 0.5라는 귀무가설을 기각하지 못한다.

## 7.6 이표본 비율

### 이론적 배경

독립인 두 집단  $X, Y$ 에서 다음을 따른다고 하자.

$$X \sim B(n_1, p_1), \quad Y \sim B(n_2, p_2) \quad (8.36)$$

식 8.34를 참고해보면  $n$ 이 충분히 클 때  $X - Y$ 가 근사적으로 정규분포를 따름을 알 수 있다.

$$X - Y \sim N(p_1 - p_2, \frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2}) \quad (8.37)$$

따라서  $p_1 - p_2$ 의 95% 신뢰구간은  $\alpha = 0.05$ 라 할 때 다음과 같다.

$$(p_1 - p_2 - z_{\alpha/2} \sqrt{\frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2}}, p_1 - p_2 + z_{\alpha/2} \sqrt{\frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2}}) \quad (8.38)$$

### 추정 및 검정의 예

두개의 동전을 각각 100회, 90회 던졌을 때 각각 앞면이 45회, 55회 나왔다고 하자. 이 때 두 동전의 앞면이 나올 확률이 같은지 검정해보자.

```
> prop.test(c(45, 55), c(100, 90))

 2-sample test for equality of proportions with continuity correction

data: c(45, 55) out of c(100, 90)
X-squared = 4.3067, df = 1, p-value = 0.03796
alternative hypothesis: two.sided
95 percent confidence interval:
-0.31185005 -0.01037217
```

```
sample estimates:  
prop 1      prop 2  
0.4500000 0.6111111
```

$p < 0.05$ 가 나와 두 동전의 앞면이 나올 확률이 같다는 가설을 기각한다. 즉 두 동전의 앞면이 나올 확률은 유의미하게 다르다.

## 선형 회귀(Linear Regression)

이 장에서는 선형 회귀의 가장 기본적인 lm() 함수의 사용 방법과 formula를 올바로 사용하는 방법 및 만들어진 모델의 결과를 해석하는 방법에 대해 다룬다. 선형 회귀는 단순해 보이지만 여러권의 책으로도 모두 다루기 어려운 주제이다. 많은 내용이 생략되어 있고 통계적 배경에 대한 설명이 대부분 생략되어 있으므로 부족한 부분은 참고 문헌을 확인하기 바란다.

### 1 단순 선형 회귀(Simple Linear Regression)

선형 회귀는 독립변수와 종속변수간의 관계를 모델링하는 기법을 말한다. 이 때 독립변수가 하나인 경우 단순 선형 회귀(Simplie Linear Regression)라고 독립변수가 2개 이상인 경우 중선형회귀(Multiple Linear Regression) (페이지 277)(또는 다중 선형 회귀. Multiple Linear Regression)이라 한다.

선형 회귀는 다음과 같이 독립변수  $X$ 와 종속변수  $Y$ 를 표현한다

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i \quad (9.1)$$

선형 회귀에는 다음과 같은 전제가 따른다[23, 33, ?].

- 독립변수  $X$ 는 고정된 값이다.
- 오차항의 분산이 동일하다.
- 오차항간 상호 독립이다.
- 오차항은 평균이 0이며 분산은  $\sigma^2$ 인 정규 분포를 따른다.
- 독립변수간 독립이다.
- 종속 변수와 독립변수간에 수식 9.1이 성립해야한다.

그러나 선형 회귀를 적용할 때 이러한 조건이 저절로 성립하는 것은 아니므로 모델을 검증하여 부족한 점이 없는지 확인하거나, 또는 독립 변수나 종속 변수를 변환하여 이러한 조건이 만족하도록 할 필요가 있다.

## 1.1 모델 생성

선형 회귀는 lm() 함수를 사용하여 만들 수 있다. lm()은 인자로 선형 회귀식을 표현하는 formula와 모델링할 data를 받는다.

lm() 함수를 배우기에 앞서 선형 회귀의 예로 사용할 cars 데이터를 살펴보자.

```
> data(cars)
> head(cars)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
```

cars 데이터는 speed와 dist의 2개 컬럼으로 구성되어 있다. speed는 차량의 속도, dist는 해당 차를 멈추기 위해 브레이크를 잡았을 때 차량이 얼마나 더 전진했는지를 기록한 값이다. 간단히 다음과 같은 모델을 생각해보자.

$$dist = \beta_0 + speed \times \beta_1 + \epsilon \quad (9.2)$$

이를 Formula로 표시하면  $dist \sim speed$ 가 된다. 절편에 해당하는  $\beta_0$ 은 굳이 쓰지 않아도 항상 존재하는 것으로 취급된다. 다음은 lm()을 사용한 선형 회귀의 예이다.

```
> m <- lm(dist ~ speed, cars)
> m

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)      speed
-17.579        3.932
```

$\epsilon$ 을 무시하면 다음과 같은 수식을 얻게된다.

$$dist = -17.579 + 3.932 \times speed \quad (9.3)$$

## 1.2 선형회귀 결과 추출

lm()으로 구한 모델의 내용을 다음과 같은 함수를 사용해 살펴볼 수 있다.

### 회귀 계수

다음은 절편이 -17.579095, speed에 대한 기울기가 3.932409임을 보여준다.

```
> coef(m)
(Intercept)      speed
-17.579095     3.932409
```

### 예측 값(Fitted Values)

cars데이터의 각 speed 값에 대한 dist의 예측값은 fitted()로 구할 수 있다.

```
> fitted(m)[1:4]
 1          2          3          4
-1.849460 -1.849460  9.947766  9.947766
```

즉 이 값은  $-17.579095 + 3.932409 * \text{cars\$speed}$ 로 추정한 값에 해당한다.

### 잔차(Residuals)

예측값과 실제 dist간의 차이에 해당하는 잔차는 residual()로 구한다.

```
> residuals(m)[1:4]
 1          2          3          4
 3.849460 11.849460 -5.947766 12.052234
```

따라서 fitted(m) + residuals(m)은 cars\$dist와 같다.

```
> fitted(m)[1:4] + residuals(m)[1:4]
 1  2  3  4
 2 10  4 22
```

```
> cars$dist[1:4]
[1]  2 10  4 22
```

## 계수의 신뢰구간

단순 선형 회귀에서 절편과 speed의 기울기는 정규 분포를 따른다. 따라서 t 분포를 사용한 신뢰구간을 `confint()`를 사용해 다음과 같이 구할 수 있다.

```
> confint(m)
              2.5 %    97.5 %
(Intercept) -31.167850 -3.990340
speed        3.096964  4.767853
```

## 잔차 제곱 합

잔차의 제곱합, 즉  $\sum(Y - \hat{Y})^2$ 은 다음과 같이 구한다.

```
> deviance(m)
[1] 11353.52
```

식에따라 위 값은 다음과 같이 계산할 수도 있다.

```
> sum((cars$dist - predict(m, newdata=cars))^2)
[1] 11353.52
```

`predict()`의 사용에 대해서는 [예측과 신뢰구간](#) (페이지 267)을 참고하기 바란다.

## 1.3 예측과 신뢰구간

`lm()`을 통해 만들어진 모델은 `predict()`를 사용하여 예측할 수 있다. `predict`는 사실 generic function으로서 여러가지 다른 방식으로 모델을 만들었을 때 그 모델로부터 새로운 데이터에 대한 예측값을 구하는데 사용할 수 있다. 주어진 모델에따라 내부적으로 `predict.glm()`, `predict.lm()`, `predict.nls()` 등의 함수를 부르게되는데 선형 회귀의 경우 `predict.lm()`을 부르게 된다.

`predict()`는 인자로 모델, 그리고 예측을 수행할 새로운 데이터를 받는다. 다음 예를보자.

```
> m
```

```

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)      speed
-17.579        3.932

> predict(m, newdata=data.frame(speed=3))
1
-5.781869

> coef(m)
(Intercept)      speed
-17.579095    3.932409

> -17.579095 + 3.932409 * 3
[1] -5.781868

```

`m` 은 앞서 절에서 `cars`데이터로부터 만든 선형 모델이다. 이 모델을 사용해 `speed=3`일 경우의 예측을 `predict()`를 사용해 수행하였더니 그 결과가 `-5.781869`로 나타났다. 그리고 이 값은 `m` 모델의 계수를 사용해 직접 계산한 값과 일치했다<sup>1)</sup>.

절편과 기울기를 고려한 예측값의 신뢰구간은 다음과 같이 구한다. 다음 결과에서 `fit`은 예측값의 점 추정치, `lwr`와 `upr`은 각각 신뢰구간의 하한과 상한 값을 의미한다.

```

> predict(m, newdata=data.frame(speed=c(3)), interval="confidence")
      fit        lwr        upr
1 -5.781869 -17.02659  5.462853

```

이 값은 다음 식을 바탕으로 신뢰 구간을 구한 것이다.

$$dist = \beta_0 + \beta_1 \times speed \quad (9.4)$$

즉 특정 속도를 가진 평균적인 차량의 제동거리에 해당한다. 평균적인 차량에 대한 추정이므로 오차항은 고려되지 않는다. 왜냐하면 오차는 평균이 0인 정규분포로 가정되기 때문이다. 그러나 특정 속도를 가진 한대의 차량이 주어졌을 때 그 한대의 차량에 대한 신뢰구간을

<sup>1)</sup>직접 계산한 값은 계수와 절편값의 유효자리가 제한된 상태로 계산하였으므로 완전히 일치하지는 않는다

구하려면 다음과 같이 오차항을 고려해야한다.

$$dist = \beta_0 + \beta_1 \times speed + \epsilon \quad (9.5)$$

이를 반영한 신뢰구간은 interval="prediction"을 사용해 구한다.

```
> predict(m, newdata=data.frame(speed=c(3)), interval="prediction")
      fit        lwr        upr
1 -5.781869 -38.68565 27.12192
```

이 경우 오차항으로인해 신뢰구간의 크기가 커짐을 볼 수 있다.

## 1.4 모형 평가

summary()를 사용해 선형 회귀의 결과를 손쉽게 평가할 수 있다.

```
> summary(m)

Call:
lm(formula = dist ~ speed, data = cars)

Residuals:
    Min      1Q  Median      3Q     Max 
-29.069 -9.525 -2.272  9.215 43.201 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -17.5791    6.7584  -2.601   0.0123 *  
speed        3.9324    0.4155   9.464 1.49e-12 *** 
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438 
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

summary() 함수의 가장 처음에는 어떤 formula를 사용해 선형 회귀를 수행했는지 알려준다.

Residual 부분에서는 실제 데이터에서 관측된 잔차를 보여준다.

Coefficients에서는 모델의 계수와 이 계수들의 통계적 유의성을 알려준다.

마지막으로 R-squared와 Adjusted R-squared를 통해 모델이 데이터의 분산을 얼마나 설명하는지를 알려주며, F-statistics는 모델의 유의성을 보여준다.

### 설명 변수 평가

summary()의 출력중 다음 부분이 설명 변수를 평가한 부분이다.

**Coefficients :**

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-17.5791	6.7584	-2.601	0.0123 *
speed	3.9324	0.4155	9.464	1.49e-12 ***
<hr/>				
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1				

Estimate 열은 절편과 계수의 추정치를 보여준다. 위 결과에 따르면  $distance = -17.5791 + 3.9324 \times speed$ 의 식이 구해졌다.

Pr(> |t|)열은 t 분포를 사용하여 각 변수가 얼마나 유의한지를 알려준다. speed열의 경우 P 값이 1.49e-12로 나타나 매우 유의한 것으로 판단 되었다. 만약 유의하지 않은 결과가 나온다면 이는 해당 계수가 0이라는 귀무가설을 기각할 수 없음, 즉 계수를 0으로 봐야한다는 것을 뜻한다.

Pr(> |t|)열 바로 뒤에 \* 또는 \*\*\*로 표시된 문자열은 p value의 범위를 뜻한다. 만약 유의한 (즉 p 값이 0.05보다 작은) 계수가 있다면 . 또는 \*, \*\*, \*\*\* 등으로 그 값을 알기쉽게 표시해준다. 만약 아무런 표시가 없다면 계수가 통계적으로 유의하지 않음을 뜻한다.

### 결정계수와 F 통계량

summary()의 결과중 다음 부분은 모델의 결정계수와 F 통계량을 보여준다.

```
Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared: 0.6511, Adjusted R-squared: 0.6438
F-statistic: 89.57 on 1 and 48 DF, p-value: 1.49e-12
```

이 결과에서 R-squared( $R^2$ )와 Adjusted R-squared는 각각 0.6511, 0.6438로 나타났다. R-squared는 다음과 같이 정의된다[33].

$$SST = \sum (Y_i - \bar{Y})^2$$

$$SSR = \sum (\hat{Y}_i - \bar{Y})^2$$

$$R^2 = SST/SSR$$

SST는 총 제곱합(Total Sum of Squares), SSR은 회귀 제곱합(Sum of Squares due to Regression)을 의미한다. SST는 관측된  $Y_i$  값이  $Y_i$ 들의 평균( $\bar{Y}$ )으로부터 얼마나 떨어져있는지를 뜻하며 SSR은 추정치  $\hat{Y}_i$ 가 평균  $\bar{Y}$ 로부터 얼마나 떨어져있는지를 뜻한다. 따라서 이 둘의 비율은  $Y_i$ 를 단순히 평균으로 추정하는 경우의 변동 대비 회귀 모형이 얼마나 그 변동을 설명하는지를 알려준다.

그러나  $R^2$ 는 설명 변수가 늘어나면 그 값이 커지는 성질이 있으므로 이를 자유도로 나눈 Adjusted R-squared가 더 많이 사용된다. 위 결과에서도 R-squared보다 Ajudsted R-squared가 작은 값이 나타나 보수적인 추정치임을 보여주고 있다.

F-statistics은 MSR/MSE의 비율을 F 분포를 사용해 검정한 것이며, 또 이 값은  $dist = \beta_0 + \epsilon$ 인 축소 모형(reduced model)과  $dist = \beta_0 + \beta_1 \times speed + \epsilon$ 의 완전 모형(full model)간에 잔차 제곱합이 얼마나 유의하게 다른지 보는 방식과 같다. 다시 말해 F statistics은  $H_0: \beta_1 = 0$ 인지에 대한 가설 검증 결과이다.

더 자세한 설명은 참고문헌[33, 34]을 보기 바란다.

## 1.5 ANOVA 및 모델간의 비교

summary()가 보여주는 F 통계량은 anova() 함수를 사용해 직접 구할 수 있다.

```
> anova(m)
Analysis of Variance Table

Response: dist
          Df  Sum Sq Mean Sq F value    Pr(>F)
speed       1  21186 21185.5   89.567 1.49e-12 ***
Residuals  48   11354    236.5
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

또한 anova() 함수를 사용해 완전 모형과 축소 모형을 직접 비교해 볼 수도 있다. 다음은 완전 모형과 축소 모형을 만드는 예이다.

```

> full <- lm(dist ~ speed, data=cars)
> reduced <- lm(dist ~ 1, data=cars)

> full
Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)      speed
              -17.579       3.932

> reduced
Call:
lm(formula = dist ~ 1, data = cars)

Coefficients:
(Intercept)
              42.98

```

reduced 모델을 생성할 때 formula가 ‘dist ~ 1’임에 유의하자. 여기서 1은 절편(intercept)를 표현하기 위해 사용된 것이다.

다음은 anova로 두 모델을 비교한 결과이다.

```

> anova(reduced, full)
Analysis of Variance Table

Model 1: dist ~ 1
Model 2: dist ~ speed

  Res.Df   RSS Df Sum of Sq    F    Pr(>F)
1     49 32539
2     48 11354  1    21186 89.567 1.49e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

모델을 비교한 결과 F 통계량은 89.567이며 p 값은 아주 작게 나타났다. 따라서 reduced 모델과 full 모델간에는 유의한 차이가 있다. 이는 바꿔말하면 speed 열이 유의미한 설명변수임을 뜻한다.

## 1.6 모델 평가 차트

단순히 `plot(m)` 명령을 내리는 것만으로 선형 모델을 평가하는데 필요한 다양한 차트를 볼 수 있다.

```
> plot(m)
```

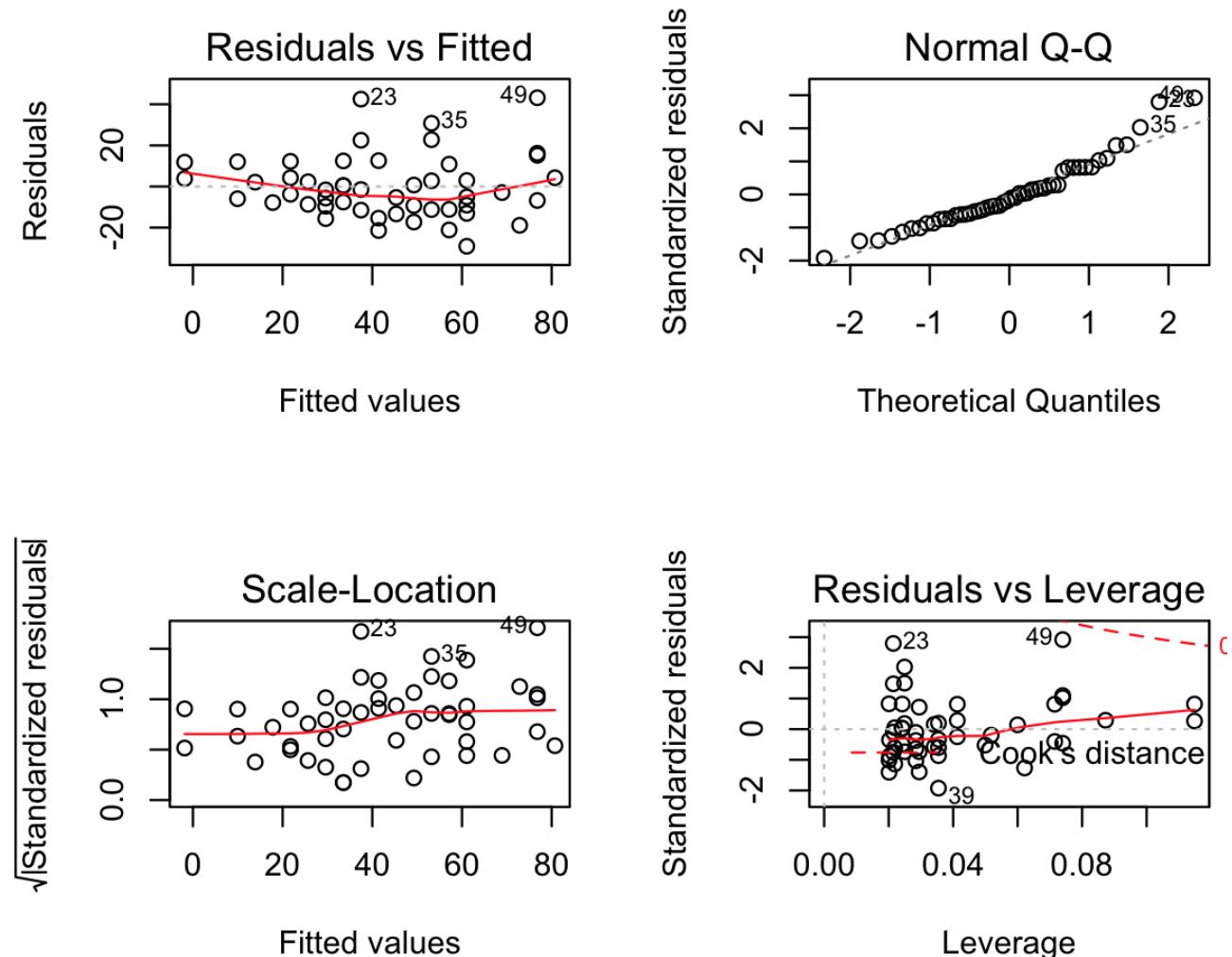


그림 9.1: 선형 모델의 평가

첫번째 차트 ‘Residuals vs Fitted’는 X축에 선형회귀로 적합된 값, Y축에 적합된 값과 실제 데이터의 차이인 잔차를 보여준다. 선형 회귀에서 오차는 평균이 0이고 분산이 일정한 정규분포임을 가정하였으므로 이 그래프에서는 기울기 0인 직선이 관측되는 것이 이상적이다.

두번째 차트인 ‘Noraml Q-Q’는 잔차가 정규분포를 따르는지 확인하기 위한 목적이다. Q-Q plot에 대해서는 [Q-Q Plot](#) (페이지 241)에서 살펴본 바 있다.

세번째 차트인 ‘Scale-Location’은 X축에 선형회귀로 적합된 값, Y축에 표준화 잔차를 보여준다. 이 경우도 기울기가 0인 직선이 이상적이다.

네번째 차트인 'Residuals vs Leverage'는 X축에 Leverage, Y축에 표준화 잔차를 보여준다. 표준화 잔차를 보면 어떤 점들이 이상치(Outlier)에 해당하는지 알 수 있는데, Standardized residual이 지나치게 크거나 작은 점들은 이상치일 가능성이 크다고 본다. Leverage는 설명변수가 얼마나 극단에 치우쳐있는지를 뜻한다. 예를 들어 다른 데이터의 X 값은 모두 1 ~ 10 사이의 값인데 특정 데이터만 99999의 값을 갖고 있다면 해당 데이터의 leverage는 큰 값이 된다. 이런 데이터는 입력이 잘못되었다거나, 해당 범위의 설명변수 값을 갖는 데이터를 더 보충해야 한다거나 하는 등 유심히 살펴볼 필요가 있다.

네번째 차트의 우상단과 우하단에는 빨간 선으로 Cook's Distance가 표시되어 있는데 Cook's distance는 Leverage와 Residual에 비례하므로 두 값이 큰 우상단과 우하단에 Cook's Distance 가 큰 값이 위치하게 된다. Cook's Distance가 크면 영향력 있는 관측치로 고려할 수 있다. 이 차트에 더 관심이 있는 독자는 인터넷에 공개된 책자인 Advanced Statistical Modelling[35]를 참고하거나 또는 그림과 예제로 매우 잘 설명한 같은 강의의 PPT 슬라이드 [Diagnosis \(Outlier\)](#)를 참고하기 바란다<sup>2)</sup>.

`plot()`은 generic function이므로 인자로 선형 회귀 모델을 주면 `plot.lm()`이 호출된다. `plot.lm()`이 그리는 차트는 위에 보인 4가지 차트 외에도 2가지 종류가 더 있다. 이를 차트를 보려면 `which`에 차트 번호 `c(4, 6)`을 다음과 같이 지정한다<sup>3)</sup>.

```
> par(mfrow=c(1,2))
> plot(m, which=c(4, 6))
```

다음은 위 코드의 실행 결과이다.

<sup>2)</sup>강의 사이트 주소는 <http://www.stat.auckland.ac.nz/~lee/330/>이다.

<sup>3)</sup>`which`의 기본값은 `c(1:3, 5)`이므로 `plot.lm()`은 1 ~ 3번째 및 5번째 차트를 기본으로 보여준다

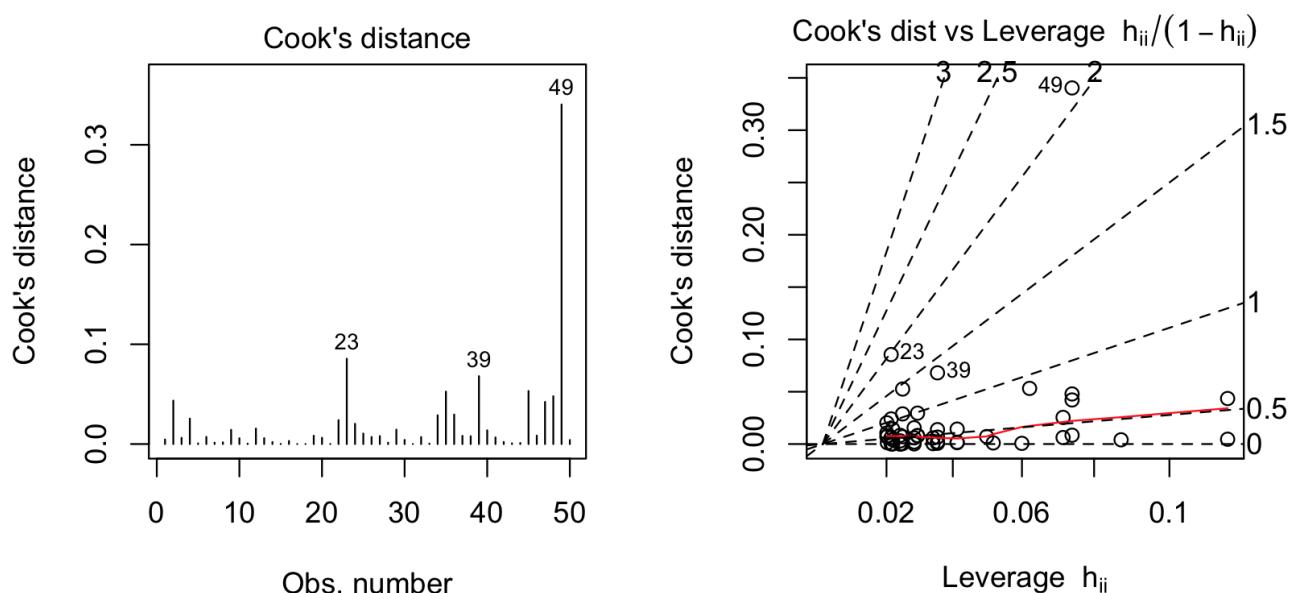


그림 9.2: Cook's Distance와 Leverage

첫번째 차트는 관찰값별 Cook's Distance, 두번째 차트는 각 관찰값에 대한 Leverage와 Cook's Distance를 보여준다.

## 1.7 회귀 직선의 시각화

데이터의 산점도와 회귀 직선은 다음과 같이 그릴 수 있다. 다음 코드에서 `coef()`는 선형 회귀 모델의 절편과 기울기를 추출하는 함수이며, `abline()`은 주어진 절편과 기울기로 그래프를 그리는 함수임을 기억하기 바란다.

```
> plot(cars$speed, cars$dist)
> abline(coef(m))
```

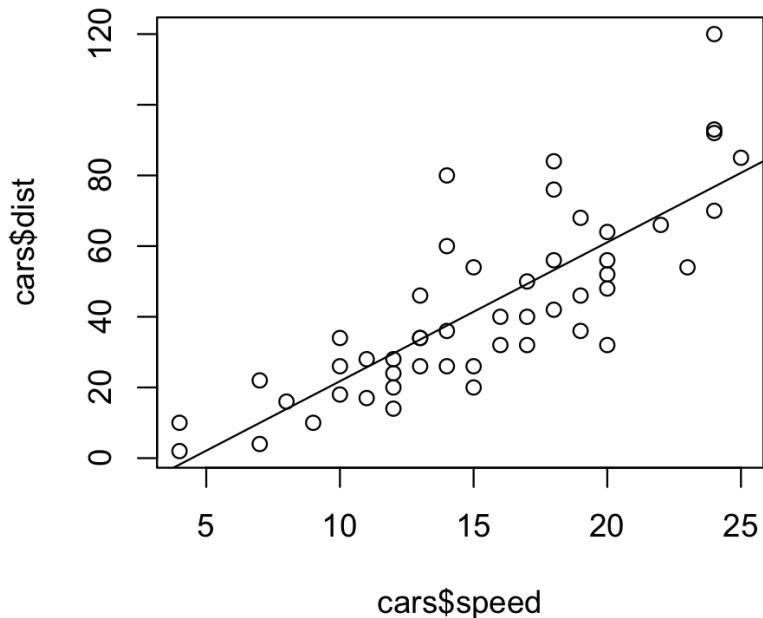


그림 9.3: Cars 데이터와 회귀직선

추정값의 신뢰구간을 포함하여 그리는 방법은 다음과 같다. 먼저 speed의 최소, 최대 값을 찾는다.

```
> summary(cars$speed)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
4.0       12.0    15.0    15.4    19.0    25.0
```

이 값의 범위에 대해 신뢰 구간을 구한다.

```
> predict(m,
+         newdata=data.frame(speed=seq(4.0, 25.0, .2)),
+         interval="confidence")
            fit        lwr        upr
1 -1.8494599 -12.329543  8.630624
2 -1.0629781 -11.391450  9.265494
3 -0.2764964 -10.453842  9.900849
4  0.5099854 -9.516740 10.536711
5  1.2964672 -8.580168 11.173102
6  2.0829489 -7.644150 11.810048
...
```

마지막으로 행렬에 저장된 데이터 그리기(matplot, matlines, matpoints) (페이지 198)을 사용해 차트를 그린다. 다음은 이 모든 단계를 한번에 보인 예이다.

```
> speed <- seq(min(cars$speed), max(cars$speed), .1)
> ys <- predict(m, newdata=data.frame(speed=speed),
+                 interval="confidence")
> matplot(speed, ys, type='n')
> matlines(speed, ys)
```

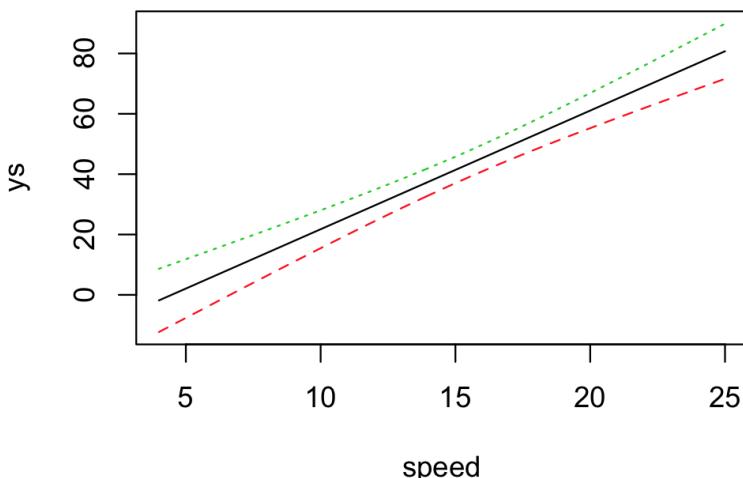


그림 9.4: 회귀직선의 신뢰대(Confidence Band)

## 2 중선형회귀(Multiple Linear Regression)

중회귀는 하나 이상의 설명변수가 사용된 선형 회귀이다. 즉  $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n + \epsilon$  형태의 모델을 말한다.

### 2.1 모델 생성 및 평가

원하는 설명변수를 + 로 연결해 나열함으로써 중선형회귀 모형을 생성할 수 있다. 다음 코드는 iris 데이터의 Sepal.Length를 Sepal.Width, Petal.Length, Petal.Width를 사용해 예측하는 모델을 만든다.

```
> m <- lm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width,
+           data=iris)
```

```

> m
Call:
lm(formula = Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width,
    data = iris)

Coefficients:
(Intercept) Sepal.Width Petal.Length Petal.Width
           1.8560        0.6508       0.7091      -0.5565

> summary(m)
Call:
lm(formula = Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width,
    data = iris)

Residuals:
Min       1Q   Median       3Q      Max
-0.82816 -0.21989  0.01875  0.19709  0.84570

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.85600   0.25078   7.401 9.85e-12 ***
Sepal.Width 0.65084   0.06665   9.765 < 2e-16 ***
Petal.Length 0.70913   0.05672  12.502 < 2e-16 ***
Petal.Width -0.55648   0.12755  -4.363 2.41e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3145 on 146 degrees of freedom
Multiple R-squared:  0.8586, Adjusted R-squared:  0.8557
F-statistic: 295.5 on 3 and 146 DF,  p-value: < 2.2e-16

```

모델의 계수를 읽는 방법이나 `summary()`의 결과를 읽는 방법은 앞서 살펴본 단순선형회귀와 동일하다. 위 결과에서는 Sepal.Width, Petal.Length, Petal.Width의 p 값이 모두 0.05보다 작아 모두 중요한 설명변수이다.

F-statistic는 단순 선형 회귀와 귀무가설에 차이가 있는데, 단순 선형 회귀의 귀무가설은  $\beta_0 + \beta_1 \times X + \epsilon$ 에서  $H_0: \beta_1 = 0$ 인 반면 중선형회귀에서의 귀무가설은 ‘ $H_0: \text{모든 계수가 } 0\text{이다}(\text{즉 } \beta_0 = \beta_1 = \dots = 0)$ ’라는 점이다. 따라서 하나의 설명변수라도 0이 아닌 계수를 갖게되면 모델이 유의한것으로 판단된다.

## 2.2 범주형 변수

앞 절에서는 범주형 변수인 Species를 설명 변수에서 생략하였다. Species를 포함하려면 ‘+ Species’를 formula에 추가해 설명변수를 추가하거나 또는 단순히 ‘Sepal.Length ~ .’로 formula를 적는다. 여기서 ‘.’는 종속변수를 제외한 모든 변수를 의미한다. 다음은 이를 구현한 예이다.

```
> m <- lm(Sepal.Length ~ ., data=iris)
> m
Call:
lm(formula = Sepal.Length ~ ., data = iris)

Coefficients:
              (Intercept)          Sepal.Width          Petal.Length
                  2.1713                 0.4959                 0.8292
             Petal.Width  Speciesversicolor  Speciesvirginica
                 -0.3152                -0.7236                -1.0235

> summary(m)
Call:
lm(formula = Sepal.Length ~ ., data = iris)

Residuals:
    Min      1Q   Median      3Q      Max 
-0.79424 -0.21874  0.00899  0.20255  0.73103 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept)  2.17127   0.27979   7.760 1.43e-12 ***
Sepal.Width  0.49589   0.08607   5.761 4.87e-08 ***
Petal.Length 0.82924   0.06853  12.101 < 2e-16 ***

```

```

Petal.Width      -0.31516    0.15120   -2.084   0.03889  *
Speciesversicolor -0.72356    0.24017   -3.013   0.00306  **
Speciesvirginica  -1.02350    0.33373   -3.067   0.00258  **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3068 on 144 degrees of freedom
Multiple R-squared:  0.8673 , Adjusted R-squared:  0.8627
F-statistic: 188.3 on 5 and 144 DF,  p-value: < 2.2e-16

```

Species를 추가한 모델에서는 Speciesversicolor와 Speciesvirginica의 두개 계수가 더 보인다. Species는 이 외에도 setosa가 있는데 위 결과에는 Speciessetosa 계수가 없다. 그 이유는 범주형 변수 Species를 표 9.1에 보인 바와 같이 2개의 가변수(dummy variable)를 사용해 표현했기 때문이다.

표현하고자 하는 Species	Speciesversicolor	Speciesvirginica
setosa	0	0
versicolor	1	0
virginica	0	1

표 9.1: 가변수를 사용한 범주형 변수의 표현

데이터가 어떻게 코딩되는지를 살펴보고 싶다면 model.matrix() 함수를 사용한다. 다음은 iris데이터의 1행, 51행, 101행 데이터가 어떻게 코딩되어 모델에 사용되는지를 보여준다. iris에서 1행은 setosa, 51행은 versicolor, 101행은 virginica 종에 대한 데이터임을 상기하기 바란다.

```

> model.matrix(m)[c(1, 51, 101),]
  (Intercept) Sepal.Width Petal.Length Petal.Width Speciesversicolor
1             1        3.5       1.4        0.2            0
51            1        3.2       4.7        1.4            1
101           1        3.3       6.0        2.5            0
  Speciesvirginica
1                 0
51                0
101               1

```

이렇게 만들어진 모델은 Species 별로 Sepal.Length에 대해 다음과 같은 세가지 모델을 만든 셈이다<sup>4)</sup>.

$$setosa : 2.17 + Sepal.Width \times 0.49 + Petal.Length \times 0.82 + Petal.Width \times -0.31$$

$$versicolor : 2.17 - 0.72 + Sepal.Width \times 0.49 + Petal.Length \times 0.82 + Petal.Width \times -0.31$$

$$virginica : 2.17 - 1.02 + Sepal.Width \times 0.49 + Petal.Length \times 0.82 + Petal.Width \times -0.31$$

이 모델은 Species별로 절편만 다르게 설정하고 있다. 절편뿐만 아니라 만약 Species 별로 다른 설명변수의 계수까지 다르게 설정하고 싶다면 Species와 다른 열의 상호 작용을 모델링해야한다. 설명 변수간 상호작용은 [상호 작용](#) (페이지 287)에서 다루기로 하자. 상호 작용을 고려한 다양한 모델 설명에 대해서는 Faraway의 Linear Models in R[26]에서 Analysis of Covariance절을 참고하기 바란다<sup>5)</sup>.

`anova()`를 사용해 분산분석 결과를 보자.

```
> anova(m)
Analysis of Variance Table

Response: Sepal.Length
           Df  Sum Sq Mean Sq F value    Pr(>F)
Sepal.Width     1 1.412   1.412 15.0011 0.0001625 ***
Petal.Length    1 84.427  84.427 896.8059 < 2.2e-16 ***
Petal.Width     1 1.883   1.883  20.0055 1.556e-05 ***
Species         2  0.889   0.444   4.7212 0.0103288 *
Residuals      144 13.556   0.094
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

`summary()`로 살펴볼 때와는 달리 `anova()`의 결과에서는 Species가 하나의 설명변수로 묶여서 표시된다. 이 표를 보면 Species의 p 값은 0.0103288이므로 유의미한 설명변수임을 알 수 있다.

<sup>4)</sup>편의를 위해 소수점 3자리 이하를 버리고 오차항은 생략함

<sup>5)</sup>ANOVA와 linear regression은 밀접한 관계가 있다. 이 관계가 이해되지 않는다면 [Why ANOVA and Linear Regression are the Same Analysis](#)를 읽어보기 바란다

### 2.3 중선형회귀모형의 시각화

앞서 ‘Sepal.Length ~ .’로부터 만든 모델은 다수의 설명변수를 사용했다. 그러나 2차원 이상의 데이터는 쉽게 표시할 수 없으므로 여기서는 Species와 Sepal.Width만 사용해 시각화를 해보자. 다음은 iris 데이터를 Species별로 Sepal.Width, Sepal.Length 차원에 산점도로 그린 예이다.

```
> with(iris, plot(Sepal.Width, Sepal.Length,  
+                  cex=.7,  
+                  pch=as.numeric(Species)))
```

위 코드에서 `cex`는 점의 크기를 지정하는 옵션이며 `pch`는 점의 형태를 정하기 위함이다. `as.numeric()`을 `Species`인 범주형 변수에 적용하였으므로 `Species`의 종별로 1, 2, 3의 값을 갖게 된다. 즉 `pch`는 다음과 같이 지정된다.

알아보기 쉽게 legend()를 사용해 범례를 다음과 같이 그려보자

```
> legend("topright", levels(iris$Species), pch=1:3, bg="white")
```

위 코드에서 `bg="white"`는 배경을 흰색으로 칠해준다. 이렇게 하면 범주가 표시된 사각형 안의 데이터는 가려지므로 범주를 읽기 편해진다. `pch`는 앞서 `plot()`에서 사용한 1, 2, 3을 차례로 지정한 것이다. `levels()`는 범주형 변수의 각 수준의 이름을 반환한다. `levels()`의 결과는 다음과 같다.

```
> levels(iris$Species)
[1] "setosa"      "versicolor" "virginica"
```

다음은 회귀 직선을 그릴 차례이다. Sepal.Width와 Species만 사용하기로 하였으므로 모델을 새로 적합하고 계수를 구한다.

```
> m <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris)
> coef(m)
(Intercept) Sepal.Width Speciesversicolor Speciesvirginica
2.2513932    0.8035609     1.4587431      1.9468166
```

회귀 직선은 abline()을 사용해 그린다. abline()의 첫번째 인자는 절편, 두번째 인자는 기울기이며 lty는 선의 유형을 지정한다.

```
> abline(2.25, 0.80, lty=1)
> abline(2.25 + 1.45, 0.80, lty=2)
> abline(2.25 + 1.94, 0.80, lty=3)
```

코드를 한번에 정리하면 다음과 같다.

```
> with(iris, plot(Sepal.Width, Sepal.Length,
+                   cex=.7,
+                   pch=as.numeric(Species)))
> m <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris)
> coef(m)
(Intercept) Sepal.Width Speciesversicolor Speciesvirginica
2.2513932    0.8035609     1.4587431      1.9468166
> abline(2.25, 0.80, lty=1)
> abline(2.25 + 1.45, 0.80, lty=2)
> abline(2.25 + 1.94, 0.80, lty=3)
> legend("topright", levels(iris$Species), pch=1:3, bg="white")
```

결과는 다음과 같다.

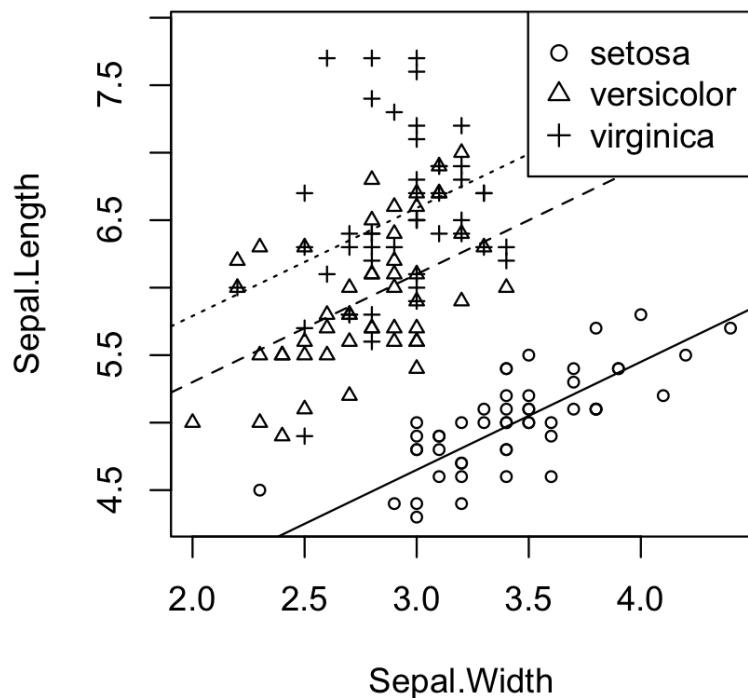


그림 9.5: iris 데이터에 대한 Species별 선형회귀모형의 시각화

## 2.4 표현식을 위한 I()의 사용

지금까지 본 중선형회귀의 formula는 설명변수와 종속변수의 관계가 1차식으로 표현가능했다. 그러나 경우에 따라서는 종속 변수가 독립 변수의 이차 이상에 비례하는  $Y = X^2 + 3X + 5 + \epsilon$ 와 같은 경우가 있을 수 있다. 또는  $Y = 3 \times (X_1 + X_2) + \epsilon$ 과 같이 두 변수의 합을 하나의 변수로 보고 회귀 분석을 하고자 할 때도 있다.

이러한 경우에는 I()안에 원하는 수식을 표현한다. 다음은  $Y = X^2 + 3X + 5 + \epsilon$ 의 예이다.

```
> x <- 1:1000
> y <- x^2 + 3 * x + 5 + rnorm(1000)
> lm(y ~ I(x^2) + x)
```

Call:

```
lm(formula = y ~ I(x^2) + x)
```

Coefficients:

(Intercept)	I(x^2)	x
5.109	1.000	2.999

위 코드에서는 1 ~ 1000 까지의 값을 가진 x로 부터 종속변수 y를 만들었다. 그 뒤 formula에  $y \sim I(x^2) + x$  를 지정해 계수를 찾았다.

$I(x^2)$  대신  $x^2$ 를 사용하면 에러는 발생하지 않지만 전혀 다른 결과를 얻게 된다.

```
> lm(y ~ x^2)

Call:
lm(formula = y ~ x^2)

Coefficients:
(Intercept)          x
-167162            1004
```

그 이유는 lm에서 formula의  $x^2$ 는 [상호 작용](#) (페이지 287)으로 해석되기 때문이다. 이에 대해서는 해당 절에서 다시 다룬다.

다음은  $Y = 3 \times (X_1 + X_2) + \epsilon$ 의 예이다.

```
> x1 <- 1:1000
> x2 <- 3 * x1
> y <- 3 * (x1 + x2) + rnorm(1000)
> lm(y ~ I(x1 + x2))

Call:
lm(formula = y ~ I(x1 + x2))

Coefficients:
(Intercept)    I(x1 + x2)
-0.02753      3.00002
```

보다시피 계수 3과 0에 가까운 절편이 구해졌다.

같은 데이터에 대해  $y \sim x_1 + x_2$ 를 formula로 지정하면 다음과 같이  $x_1$ 과  $x_2$ 를 별개로 취급한 모델을 구하게 된다.

```
> lm(y ~ x1 + x2)

Call:
lm(formula = y ~ x1 + x2)
```

Coefficients:

(Intercept)	x1	x2
-0.02753	12.00007	NA

## 2.5 변수의 변환

종속변수에  $\log$ 를 취하거나 설명변수에 제곱근을 취하는 등의 변환은 formula에 곧바로 수식을 적으면 된다.

다음은  $Y = e^{X+\epsilon}$ 의 관계가 있는  $X, Y$ 가 있을 때  $\log(Y)$ 를 종속변수로 하고  $X$ 를 설명변수로 하여 회귀직선을 구한 예이다.

```
> x <- 101:200
> y <- exp(3 * x + rnorm(100))
> lm(log(y) ~ x)
```

Call:

```
lm(formula = log(y) ~ x)
```

Coefficients:

(Intercept)	x
-0.05137	2.99944

반대로 설명변수에  $\log()$ 를 취하는 예를 보자. 다음은  $Y = \log(x) + \epsilon$ 의 경우에 대해 회귀직선을 구하는 예이다.

```
> x <- 101:200
> y <- log(x) + rnorm(100)
> lm(y ~ log(x))
```

Call:

```
lm(formula = y ~ log(x))
```

Coefficients:

(Intercept)	log(x)
3.7496	0.2615

이 외에도  $\exp()$ ,  $\sqrt()$  등의 함수를 formula에 사용할 수 있다. 결국 [표현식을 위한 I\(\)](#)의 사용 ([페이지 284](#))에서 설명한 바와 같이  $X^2$ 나  $X^3$  등의 표현만 피하면 되는 것이다. 그 이

유에 대해서는 [상호 작용](#) (페이지 287)에서 설명한다.

## 2.6 상호 작용

이 절에서는 변수의 상호작용에 대해 설명하고자 한다. 지금까지 계속 예로 살펴보았던 자동차의 주행속도와 제동거리의 예를 생각해보자. 이 데이터는 (주행속도, 제동거리)의 순서쌍 데이터로 구성되어 있다. 만약 이 데이터에 자동차의 크기(소형, 중형, 대형의 세가지 범주형 변수)가 추가된다면 상호 작용을 어떻게 고려하는가에 따라 모델을 그림 2.6에 보인 세가지 경우로 나누어 생각할 수 있다[26, 33].

- 그림 (a)는 차량의 크기를 고려할 필요가 없다고 가정한 모델로, 제동거리는 주행속도에만 비례한다.
- 그림 (b)는 차량의 크기는 상수항에만 영향을 미칠뿐 주행속도에 따른 제동거리의 기울기에는 영향을 미치지 않는 경우이다.
- 그림 (c)는 차량의 크기가 상수항과 주행속도의 기울기 모두에 영향을 미치는 경우이다.

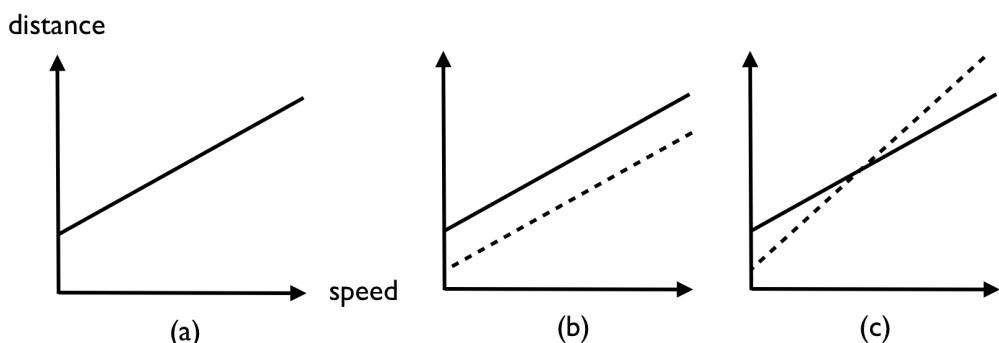


그림 9.6: 변수간의 상호작용과 그에 따른 선형 회귀 모형

이들 각각을 lm()의 formula로 표현하면 다음과 같다. (이 중 (b)의 경우에 대해서는 범주형 변수 (페이지 279)에서 살펴본 바 있다.)

- (a)  $\text{dist} \sim \text{speed}$
- (b)  $\text{dist} \sim \text{speed} + \text{size}$
- (c)  $\text{dist} \sim \text{speed} + \text{size} + \text{speed:size}$  또는  $\text{dist} \sim \text{speed} * \text{size}$

(c)의 경우에 보인 것처럼, 상호작용은 두 변수 speed와 dist를 ‘:’로 연결해 speed:dist로 표현한다. 그리고 개별 변수와 상호 작용을 모두 formula에 표현하는 speed + dist + speed:dist는 speed \* dist로 축약하여 표현할 수 있다.

만약 3개 이상의 변수 A, B, C가 있다면 어떻게 될까? 모든 상호작용을 표현한다면 A + B + C + A:B + A:C + B:C + A:B:C가 적절한 formula가 될 것이다. 그리고 이는 A\*B\*C로 축약할 수 있다.

그러나 A:B:C처럼 세개 변수가 동시에 상호작용할 수는 없고 최대 2개까지 상호작용한다면 A + B + C + A:B + A:C + B:C로 표현해야한다. 그리고 이를 축약해 표현하는 방법이 바로  $(A + B + C)^2$ 이다. 마찬가지로 설명변수 A, B, C, D가 있고 이들중 최대 2개 변수가 상호작용하며 개별 변수 역시 formula에 표현된다면  $(A + B + C + D)^2$ 가 된다. 또 최대 3개 변수가 상호작용한다면  $(A + B + C + D)^3$ 으로 표현할 수 있다.

이런 까닭에 설명변수 X의 제곱을 formula에 사용하고 싶다면  $I(X^2)$ 처럼  $I()$ 를 사용해한다. 그렇지 않으면 이는 설명변수 X와 그의 상호작용으로 해석되어버리기 때문이다. 마찬가지로  $(X + Y)^2$ 는  $X + Y + X:Y$ 를 뜻하지만  $I((X+Y)^2)$ 는 X와 Y의 합의 제곱을 뜻한다.

좀 더 다양한 포맷의 예는 [29]을 참고하기 바란다.

지금까지 살펴본 내용을 Orange 데이터 셋에 적용해보자. Orange 데이터는 다음과 같은 모양을 하고 있다.

```
> data(Orange)
> Orange
  Tree   age circumference
1     1    118              30
2     1    484              58
3     1    664              87
4     1   1004             115
5     1   1231             120
6     1   1372             142
7     1   1582             145
8     2    118              33
9     2    484              69
10    2    664             111
11    2   1004             156
12    2   1231             172
13    2   1372             203
14    2   1582             203
15    3    118              30
```

16	3	484	51
17	3	664	75
18	3	1004	108
19	3	1231	115
20	3	1372	139
21	3	1582	140
22	4	118	32
23	4	484	62
24	4	664	112
25	4	1004	167
26	4	1231	179
27	4	1372	209
28	4	1582	214
29	5	118	30
30	5	484	49
31	5	664	81
32	5	1004	125
33	5	1231	142
34	5	1372	174
35	5	1582	177

Orange 데이터에서 Tree는 서로 다른 나무를 뜻하고, age는 나무의 수령, circumference는 나무의 둘레를 뜻한다. 특히 age는 Tree별로 모두 동일한 나이인 118, 484, 664, ..., 1582 때 측정되었다. 따라서 모델을 만들기에 앞서 Tree와 circumference간 상호 연관관계를 다음처럼 손쉽게 살펴볼 수 있다<sup>6)</sup>.

```
> with(Orange,
+       plot(Tree, circumference, xlab="tree", ylab="circumference"))
```

<sup>6)</sup>plot(circumference ~ Tree) 와 같이 formula를 사용해도 된다

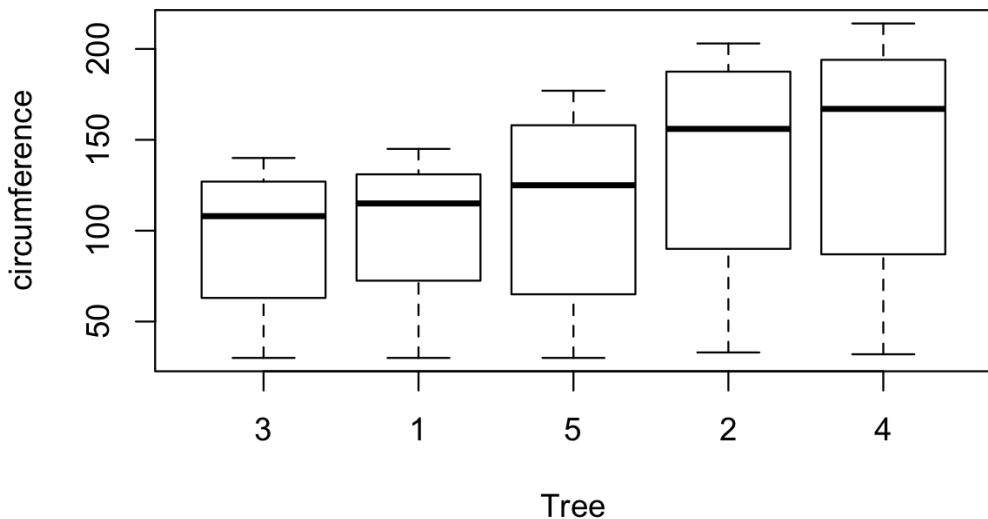


그림 9.7: Tree에 따른 circumference의 상자 그림

보다 일반적으로 자료의 상호작용을 살펴보는 그림은 상호 작용 그림(interaction plot)이며, 다음과 같이 그릴 수 있다.

```
> with(Orange, interaction.plot(age, Tree, circumference))
```

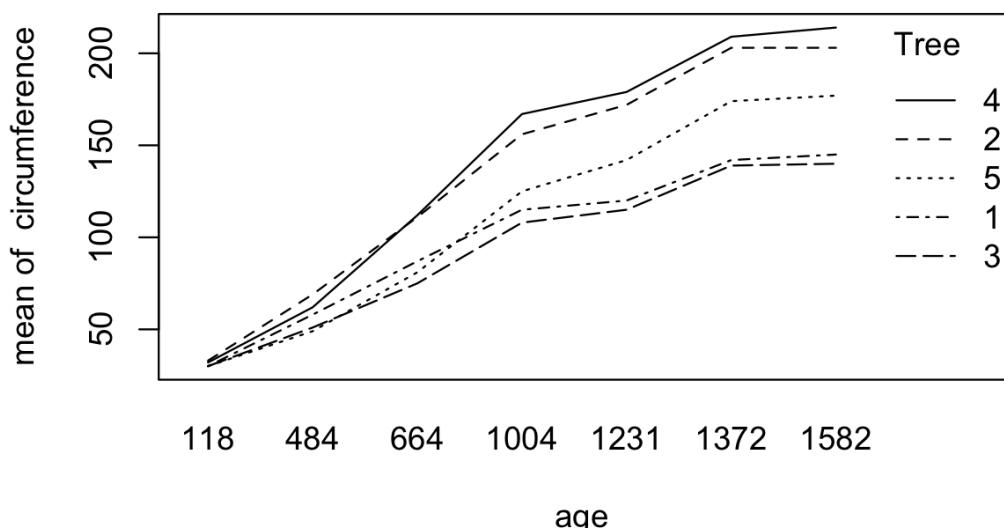


그림 9.8: Tree, age, circumference의 상호 작용 그림(interaction plot)

그림에서 볼 수 있듯이 나무의 수령이 높아짐에 따라 둘레가 길어진다. 그리고 어떤 나무인가에 따라 수령과 둘레의 길이의 관계는 서로 다른 것으로 보인다.

`lm()`을 사용해 선형 회귀를 수행하기에 앞서 Tree 열을 수정할 필요가 있다. 이 절에서 다루고자 하는 내용은 범주형 변수와 연속형 변수의 상호작용이다. 반면 `Orange$Tree`는 순서가 있는 범주형 변수이므로, 이를 순서가 없는 명목형 변수로 바꿔준다.

```
> Orange[, "fTree"] <- factor(Orange[, "Tree"], ordered=FALSE)
```

다음은 fTree, age를 설명변수로 한 선형 회귀를 수행하는 코드이다.

```
> m <- lm(circumference ~ fTree * age, data=Orange)
> anova(m)

Analysis of Variance Table

Response: circumference
          Df  Sum Sq Mean Sq F value    Pr(>F)
fTree       4   11841   2960  27.2983 8.428e-09 ***
age         1   93772   93772 864.7348 < 2.2e-16 ***
fTree:age   4    4043    1011   9.3206 9.402e-05 ***
Residuals 25    2711     108
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

보다시피 fTree, age 두개의 설명변수가 모두 p 값이 0.05보다 작아 유의한 것으로 나타났다. 둘의 상호작용인 fTree:age 역시 유의한 것으로 보인다.<sup>7)</sup>

앞서 [범주형 변수](#) (페이지 279)에서 선형 회귀를 수행할 때 범주형 변수는 가변수로 바뀐다고 설명했다. 상호작용이 있을 때 `model.matrix`는 상호 작용을 고려한 열들을 추가로 담고 있게 된다.

```
> head(model.matrix(m))
  (Intercept) fTree1 fTree5 fTree2 fTree4   age fTree1:age fTree5:age
                                         fTree2:age fTree4:age
1           1       1     0     0       0    118      118             0
                                         0       0
2           1       1     0     0       0    484      484             0
                                         0       0
3           1       1     0     0       0    664      664             0
                                         0       0
```

<sup>7)</sup> 만약 fTree:age가 유의하지 않게 나타났다면 상호작용을 제외하고 fTree + age의 formula로 새로 회귀 분석을 수행해야 할 것이다.

4	1	1	0	0	0	1004	1004	0
	0		0					
5	1	1	0	0	0	1231	1231	0
	0		0					
6	1	1	0	0	0	1372	1372	0
	0		0					

위 결과를 보면, fTree는 fTree1, fTree5, fTree2, fTree4의 4개 가변수로 표현되었음을 알 수 있다. 또 age는 fTree와의 상호작용을 고려해 fTree1:age, fTree5:age, fTree2:age, fTree4:age로 표현되었다.

좀 더 보기쉽게 age와 연관이 있는 열들만 뽑아보자.

```
> mm[,grep("age", colnames(mm))]
   age fTree1:age fTree5:age fTree2:age fTree4:age
1  118      118        0        0        0
2  484      484        0        0        0
3  664      664        0        0        0
4 1004     1004        0        0        0
5 1231     1231        0        0        0
6 1372     1372        0        0        0
7 1582     1582        0        0        0
8  118       0        0      118        0
9  484       0        0      484        0
10 664       0        0      664        0
11 1004      0        0      1004       0
12 1231      0        0      1231       0
13 1372      0        0      1372       0
14 1582      0        0      1582       0
15  118       0        0        0        0
16  484       0        0        0        0
17  664       0        0        0        0
18 1004      0        0        0        0
19 1231      0        0        0        0
20 1372      0        0        0        0
21 1582      0        0        0        0
22  118       0        0        0      118
23  484       0        0        0      484
```

24	664	0	0	0	664
25	1004	0	0	0	1004
26	1231	0	0	0	1231
27	1372	0	0	0	1372
28	1582	0	0	0	1582
29	118	0	118	0	0
30	484	0	484	0	0
31	664	0	664	0	0
32	1004	0	1004	0	0
33	1231	0	1231	0	0
34	1372	0	1372	0	0
35	1582	0	1582	0	0

위 코드에서 grep()은 패턴과 문자열을 인자로 받아 패턴이 문자열 내에 존재하면 1, 존재하지 않으면 0으로 반환하는 함수이다. 따라서 grep("age", colnames(mm))은 age라는 문자열이 열 이름에 존재하면 1, 존재하지 않으면 0을 반환하므로 age가 포함된 열의 선택을 쉽게 해준다.

결과를 보면 age 열은 각 데이터의 age를 그대로 담고 있지만 상호작용을 뜻하는 fTree1:age는 1 ~ 7행에 대해서만 age를 갖고 있고 8 ~ 35행에 대해서는 age값을 갖고 있지 않다. fTree5:age, fTree2:age, fTree4:age에도 마찬가지로 일부 행에만 age 값이 들어있다. 그 이유는 이들 상호작용을 뜻하는 열들은 formula에 주어진 수식 fTree \* age 처럼 가변수 fTreeN에 들어있는 1 또는 0의 값과 age를 곱한 값을 담고 있기 때문이다.

따라서 가변수가 1인 열에서는 age값이 상호작용 열에 나타나지만, 가변수가 0인 열에 대해서는 상호작용 열의 값이 0으로 나타난다. 이러한 방식으로 행렬을 만듦으로써 상호작용을 고려한 선형회귀를 수행하는 것이다.

순서형 범주형 변수 Tree를 사용한 선형 회귀와 더 다양한 시각화를 수행한 내용은 [Analysis of Covariance – Extending Simple Linear Regression](#)를 참고하기 바란다.

### 3 이상치(outlier)

이상치는 주어진 회귀모형에 의해 잘 설명되지 않는 자료점을 뜻한다. 이상치 검출에서는 잔차, 특히 외면 스튜던트화 잔차(externally studentized residual)을 사용한다. 외면 스튜던트화 잔차는 rstudent()를 사용해 구한다.

```
> data(Orange)
> m <- lm(circumference ~ age + I(age), data=Orange)
> rstudent(m)
```

1	2	3	4	5
6.372161e-05	-4.735583e-01	-5.474343e-02	-4.051405e-01	-1.250387e+00
6	7	8	9	10
-9.461338e-01	-1.884773e+00	1.318100e-01	-3.258711e-03	9.737615e-01
11	12	13	14	15
1.359966e+00	9.960118e-01	1.744963e+00	7.283842e-01	6.372161e-05
16	17	18	19	20
-7.773767e-01	-5.647282e-01	-7.042510e-01	-1.480911e+00	-1.080244e+00
21	22	23	24	25
-2.143664e+00	8.788132e-02	-3.019190e-01	1.018008e+00	1.881656e+00
26	27	28	29	30
1.311188e+00	2.045052e+00	1.226404e+00	6.372161e-05	-8.652966e-01
31	32	33	34	35
-3.087970e-01	1.696690e-02	-2.897485e-01	4.323381e-01	-4.040268e-01

외면 스튜던트화 잔차는 t 분포를 따르므로 t-test를 사용해 `rstudent()` 값이 너무 크거나 작은 점을 찾으면 된다. 다행히 R에는 이를 간단하게 할 수 있는 라이브러리 `car::outlierTest()`가 있다. 다음은 위 모델 `m`에 대해 `outlier`를 검출하기 위한 테스트를 진행한 결과이다.

```
> library(car)
> outlierTest(m)

No Studentized residuals with Bonferroni p < 0.05
Largest |rstudent|:
      rstudent unadjusted p-value Bonferroni p
21 -2.143664           0.039761          NA
```

21번 데이터가 가장 큰 `rstudent` 값을 가지고 있기는 하지만 이상치는 발견되지 않았다.

다음과 같이 이상치를 직접 추가해서 어떻게 결과가 달라지는지 살펴보자.

```
> data(Orange)
> Orange <- rbind(Orange,
+                   data.frame(Tree=as.factor(c(6, 6, 6)),
+                             age=c(118, 484, 664),
+                             circumference=c(177, 50, 30)))
> tail(Orange)
   Tree  age circumference
33     5    1231            142
```

34	5	1372	174
35	5	1582	177
36	6	118	177
37	6	484	50
38	6	664	30

```
> m <- lm(circumference ~ age + I(age^2), data=Orange)
> outlierTest(m)

    rstudent unadjusted p-value Bonferroni p
36 5.538438          3.429e-06     0.0001303
```

보다시피 36번째 데이터가 이상치로 검출되었다.

## 4 변수 선택

### 4.1 단계적 변수 선택

중선형 회귀 모형에서의 설명변수를 선택하는 방법 중 한 가지는 특정 기준(예를 들어 F-statistics나 AIC)을 사용해 변수를 하나씩 택하거나 제거하는 것이다[26, 33]. 이 방법을 좀 더 세부적으로 구분하면 다음과 같다.

- 전진 선택법(forward selection)은 절편만 있는 모델에서 기준 통계치를 가장 많이 개선시키는 변수를 차례로 추가하는 방법이다.
- 변수 소거법(backward elimination)은 모든 변수가 포함된 모델에서 기준 통계치에 가장 도움이 되지 않는 변수를 하나씩 제거하는 방법이다.
- 단계적 방법(stepwise selection)은 모든 변수가 포함된 모델에서 출발한다<sup>8)</sup>. 그리고 기준 통계치에 가장 도움이 되지 않는 변수를 삭제하거나 모델에서 빠져있는 변수 중에서 기준 통계치를 가장 개선시키는 변수를 추가한다. 그리고 이러한 변수의 추가 또는 제거를 반복한다.

단계적 변수 선택을 위해 BostonHousing 데이터를 사용해보자. 이 데이터는 보스턴 집 가격 medv를 종속 변수로 하고, 범죄율, 방의 수 등을 설명변수로하여 회귀 분석을 할 수 있는 자료이다. 다음 코드에서는 선형 모델을 lm()을 통해 구한 뒤 step(모델, direction="both")를 사용해 단계적 방법에 의한 변수 선택을 수행한다. 만약 전진 선택법을 사용하고 싶다면 direction에 forward를, 변수 소거법을 사용하고 싶다면 direction에 backward를 지정하면 된다.

<sup>8)</sup> 또는 절편만 포함된 모델에서 출발해 변수의 추가 삭제를 반복할 수도 있다.

```

> library(mlbench)
> data(BostonHousing)
> m <- lm(medv ~ ., data=BostonHousing)
> m2 <- step(m, direction="both")
Start: AIC=1589.64
medv ~ crim + zn + indus + chas + nox + rm + age + dis + rad +
      tax + ptratio + b + lstat

          Df Sum of Sq    RSS     AIC
- age      1     0.06 11079 1587.7
- indus    1     2.52 11081 1587.8
<none>            11079 1589.6
- chas     1    218.97 11298 1597.5
- tax      1    242.26 11321 1598.6
- crim    1    243.22 11322 1598.6
- zn       1    257.49 11336 1599.3
- b        1    270.63 11349 1599.8
- rad      1    479.15 11558 1609.1
- nox     1    487.16 11566 1609.4
- ptratio   1   1194.23 12273 1639.4
- dis      1   1232.41 12311 1641.0
- rm       1   1871.32 12950 1666.6
- lstat    1   2410.84 13490 1687.3

Step: AIC=1587.65
medv ~ crim + zn + indus + chas + nox + rm + dis + rad + tax +
      ptratio + b + lstat

          Df Sum of Sq    RSS     AIC
- indus    1     2.52 11081 1585.8
<none>            11079 1587.7
+ age      1     0.06 11079 1589.6
- chas     1    219.91 11299 1595.6
- tax      1    242.24 11321 1596.6
- crim    1    243.20 11322 1596.6
- zn       1    260.32 11339 1597.4

```

```

- b          1    272.26 11351 1597.9
- rad        1    481.09 11560 1607.2
- nox        1    520.87 11600 1608.9
- ptratio    1   1200.23 12279 1637.7
- dis        1   1352.26 12431 1643.9
- rm         1   1959.55 13038 1668.0
- lstat      1   2718.88 13798 1696.7

```

Step: AIC=1585.76

```

medv ~ crim + zn + chas + nox + rm + dis + rad + tax + ptratio +
     b + lstat

```

	Df	Sum of Sq	RSS	AIC
<none>		11081	1585.8	
+ indus	1	2.52	11079	1587.7
+ age	1	0.06	11081	1587.8
- chas	1	227.21	11309	1594.0
- crim	1	245.37	11327	1594.8
- zn	1	257.82	11339	1595.4
- b	1	270.82	11352	1596.0
- tax	1	273.62	11355	1596.1
- rad	1	500.92	11582	1606.1
- nox	1	541.91	11623	1607.9
- ptratio	1	1206.45	12288	1636.0
- dis	1	1448.94	12530	1645.9
- rm	1	1963.66	13045	1666.3
- lstat	1	2723.48	13805	1695.0

Call:

```

lm(formula = medv ~ crim + zn + chas + nox + rm + dis + rad +
    tax + ptratio + b + lstat, data = BostonHousing)

```

Coefficients:

(Intercept)	crim	zn	chas1	nox
36.341145	-0.108413	0.045845	2.718716	-17.376023
rm	dis	rad	tax	ptratio
3.801579	-1.492711	0.299608	-0.011778	-0.946525

```

          b           lstat
 0.009291     -0.522553

> formula(m2)
medv ~ crim + zn + chas + nox + rm + dis + rad + tax + ptratio +
    b + lstat

```

이제 위 결과를 하나씩 나눠서 살펴보자. step()의 첫번째 출력결과는 다음과 같다.

```

> step(m, direction="both")
Start:  AIC=1589.64
medv ~ crim + zn + indus + chas + nox + rm + age + dis + rad +
      tax + ptratio + b + lstat

          Df  Sum of Sq    RSS      AIC
- age      1      0.06  11079  1587.7
- indus    1      2.52  11081  1587.8
<none>                 11079  1589.6
- chas     1     218.97  11298  1597.5
- tax      1     242.26  11321  1598.6
- crim     1     243.22  11322  1598.6
- zn       1     257.49  11336  1599.3
- b        1     270.63  11349  1599.8
- rad      1     479.15  11558  1609.1
- nox      1     487.16  11566  1609.4
- ptratio   1    1194.23  12273  1639.4
- dis      1    1232.41  12311  1641.0
- rm       1    1871.32  12950  1666.6
- lstat    1    2410.84  13490  1687.3

```

가장 첫줄을 살펴보면 인자로 주어진 모델 m에서는 crim, zn, indus, ..., lstat의 총 13개 변수가 사용되고 있음을 알 수 있다. 그리고 그 때 AIC 값은 1589.64였다.

그 뒤에 나열된 결과는 각 변수를 삭제했을때 (따라서 '- 설명변수명'의 형식으로 각 행이 표시되어있다) AIC의 변화를 표현하고 있다. 예를들어 age 변수가 제거된 경우 AIC는 1587.7이 고, indus 변수가 제거되었을때 AIC는 1587.8이었다. AIC는 작을수록 더 좋은 모델을 뜻하므로 AIC를 가장 작게 만드는 age 변수가 이 단계에서 제거된다.

age가 삭제 된뒤 step() 함수의 출력은 다음과 같았다.

```

Step: AIC=1587.65
medv ~ crim + zn + indus + chas + nox + rm + dis + rad + tax +
      ptratio + b + lstat

          Df  Sum of Sq    RSS     AIC
- indus     1       2.52 11081  1585.8
<none>                 11079  1587.7
+ age       1       0.06 11079  1589.6
- chas      1     219.91 11299  1595.6
- tax       1     242.24 11321  1596.6
- crim      1     243.20 11322  1596.6
- zn        1     260.32 11339  1597.4
- b         1     272.26 11351  1597.9
- rad       1     481.09 11560  1607.2
- nox      1     520.87 11600  1608.9
- ptratio   1    1200.23 12279  1637.7
- dis       1    1352.26 12431  1643.9
- rm        1    1959.55 13038  1668.0
- lstat     1    2718.88 13798  1696.7

```

위 결과에서는 age가 전 단계에서 제거되었으므로 crim, zn, indus, ..., lstat의 총 12개 변수로 출발한다. 이 때 AIC는 1587.65이다.

이번에는 각 설명변수를 제거하는 테스트 뿐만 아니라 제거되었던 변수들을 추가하는 검토까지 한다. 제거되었던 변수는 age 밖에 없으므로 변수 추가는 age에 대해서만 수행하며 '+ age'로 표시된 행이 이에 해당한다. 변수의 추가 및 삭제에 따른 결과를 보면 indus 변수를 제거한 경우 AIC가 1585.8이 되어 가장 많은 개선이 있으므로 해당 변수를 제거한다.

이와 같은 방법을 반복하여 step()은 다음과 같은 최종 모델을 결정 지은 것이다.

```

> m2 <- step(m, direction="both")
...
> formula(m2)
medv ~ crim + zn + chas + nox + rm + dis + rad + tax + ptratio +
      b + lstat

```

새로운 데이터에 대한 예측 등은 predict(m2, newdata=...)과 같이 수행할 수 있다.

## 4.2 모든 경우에 대한 비교

앞 절의 내용은 적절한 회귀 모형을 찾기 위한 탐색 범위를 줄이는 방법을 사용했다. 반면  $N$  개의 설명변수가 있다면 각 변수를 추가하거나 뺀 총  $2^N$  개의 회귀 모델을 만들고 이를 모두를 비교해 볼 수도 있다. 데이터나 설명 변수가 많다면 이런 방법이 부적절하겠지만 그 반대의 경우라면 적용해 볼만한 방법이다.

leaps 패키지의 `regsubsets()`을 사용해 이러한 모든 경우에 대한 비교를 수행할 수 있다<sup>9)</sup>.

`BostonHousing`에 대한 모든 회귀 분석을 비교한 결과를 보려면 `regsubsets()`에 모든 변수를 포함한 `formula` 를 기재하고 그 결과를 살펴보면 된다<sup>10)</sup>.

```
> install.packages("leaps")
> library(leaps)
> m <- regsubsets(medv ~ ., data=BostonHousing)
> summary(m)

Subset selection object

Call: regsubsets.formula(medv ~ ., data = BostonHousing)
13 Variables (and intercept)

      Forced in Forced out

crim          FALSE        FALSE
zn            FALSE        FALSE
indus         FALSE        FALSE
chas1         FALSE        FALSE
nox           FALSE        FALSE
rm             FALSE        FALSE
age            FALSE        FALSE
dis            FALSE        FALSE
rad            FALSE        FALSE
tax            FALSE        FALSE
ptratio       FALSE        FALSE
b              FALSE        FALSE
lstat          FALSE        FALSE

1 subsets of each size up to 8

Selection Algorithm: exhaustive

      crim  zn  indus  chas1  nox  rm  age  dis  rad  tax  ptratio  b  lstat
```

<sup>9)</sup> <http://stat.wharton.upenn.edu/~khyuns/stat431/ModelSelection.pdf> 와 <http://www.stat.columbia.edu/~martin/W2024/R10.pdf> 를 참고.

<sup>10)</sup> 수행 결과 내용을 읽기 쉽게 하기 위해 약간의 편집을 가했다

1	" "	" "	" "	" "	" "	" "	" "	" "	" "	" "	" "	" "	" "	" *
2	" "	" "	" "	" "	" "	" *	" "	" "	" "	" "	" "	" "	" "	" *
3	" "	" "	" "	" "	" "	" *	" "	" "	" "	" "	" *	" "	" "	" *
4	" "	" "	" "	" "	" "	" *	" "	" *	" "	" "	" *	" "	" "	" *
5	" "	" "	" "	" "	" *	" *	" "	" *	" "	" "	" *	" "	" "	" *
6	" "	" "	" "	" *	" *	" *	" "	" *	" "	" "	" *	" "	" "	" *
7	" "	" "	" "	" *	" *	" *	" "	" *	" "	" "	" *	" *	" "	" *
8	" "	" *	" "	" *	" *	" *	" "	" *	" "	" "	" *	" *	" "	" *

결과의 하단에서 제일 좌측의 1, 2, 3, ..., 8로 표시된 부분은 변수의 갯수, 즉 모델의 크기를 뜻한다. 그리고 우측의 “\*”은 변수가 해당 갯수만큼 사용되었을 때 최적의 모델을 뜻한다.

예를 들어 첫줄만 따로 떼서 살펴보자. 다음 결과는 변수를 하나만 포함하겠다면 lstat를 포함한 모델이 가장 좋다는 의미이다.

crim	zn	indus	chas1	nox	rm	age	dis	rad	tax	ptratio	b	lstat
1	"	"	"	"	"	"	"	"	"	"	"	" *

변수가 2개인 행을 살펴 보면 rm, lstat을 포함한 모델이 가장 우수하다는 것을 알 수 있다.

crim	zn	indus	chas1	nox	rm	age	dis	rad	tax	ptratio	b	lstat
2	"	"	"	"	"	"	" *	"	"	"	"	" *

regsubsets()의 결과에 summary()를 적용하면 BIC, Adjusted R squared 등의 값을 쉽게 얻을 수 있다.

```
> summary(m)$bic
[1] -385.0521 -496.2582 -549.4767 -561.9884 -585.6823 -592.9553
[7] -598.2295 -600.1663
> summary(m)$adjr2
[1] 0.5432418 0.6371245 0.6767036 0.6878351 0.7051702 0.7123567
[7] 0.7182560 0.7222072
```

plot()을 사용해 다음과 같이 Adjusted R squared를 그려볼 수 있다. 그림을 통해 각 변수들이 선택되었을 때의 Adjusted R squared를 좀 더 쉽게 알 수 있다. 예를 들어 절편과 lstat이 선택된 경우에는 Adjusted R squared값이 0.54이며 절편, rm, lstat이 선택된 경우 Adjusted R squared 값이 0.64였다.

더 자세한 옵션은 ?plot.regsubsets를 참고하기 바란다.

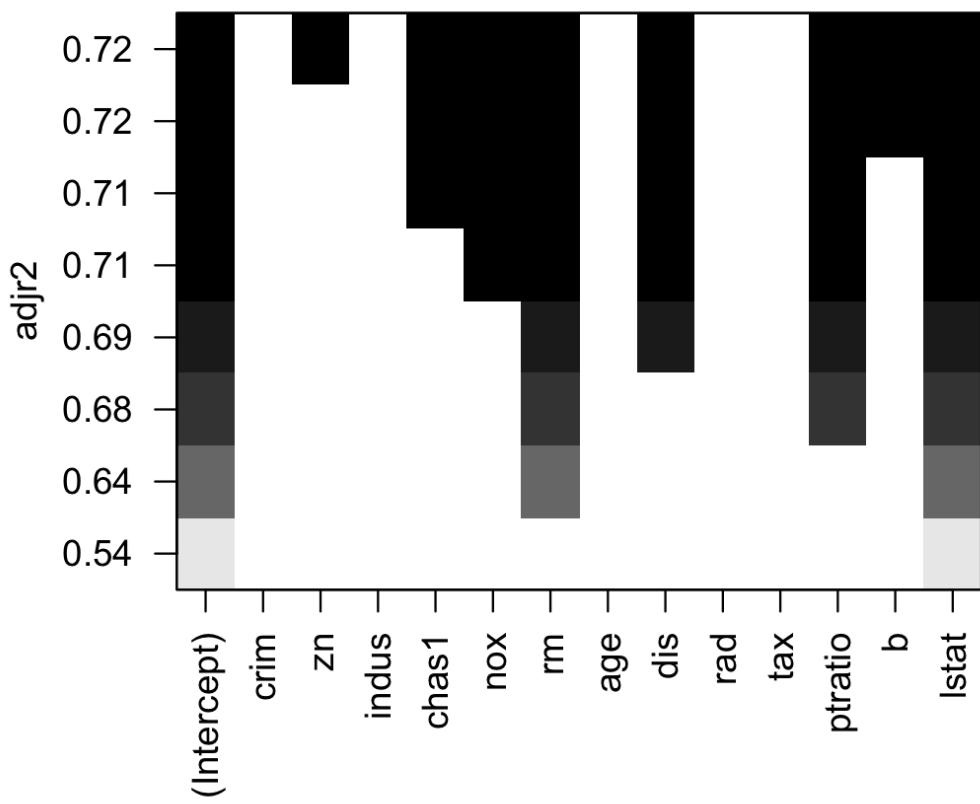


그림 9.9: regsubsets() 결과로부터 구한 Adjusted R squared

# 10

## 분류 알고리즘(Classification Algorithms)

어떤 분류(또는 레이블. Category 또는 Label)에 속하는지 알려진 훈련 데이터를 사용해 모델을 훈련시키고, 이 모델을 사용해 새로운 관찰값의 분류를 예측하는 방법을 분류 알고리즘이라고 말한다.<sup>1)</sup> 이 책에서 지금까지 언급되온 붓꽃(iris) 데이터의 Petal.Width, Petal.Length, Sepal.Width, Sepal.Length로부터 Species(붓꽃의 종류)를 예측하는 문제가 이에 속한다. 분류 알고리즘은 의학 데이터로부터 환자의 종양이 양성인지 악성인지지를 판단하거나, 어떤 금융 거래가 사기인지 아닌지를 판단하거나, 어떤 네트워크 트래픽이 해킹인지 아닌지 판단하는 등 그 사용처가 매우 많은 분야이다.

분류 알고리즘에 사용되는 훈련 데이터에는 각 데이터가 어떤 분류에 속하는지가 모두 명시되어있을 수도 있고, 또는 데이터의 일부에만 분류가 명시되어있을 수 있다. 훈련 데이터의 전부에 분류가 명시된 경우를 Supervised Learning(교사학습, 지도학습, 또는 감독학습으로 번역됨)이라고 하고, 일부에만 분류가 명시된 경우를 Semi-Supervised Learning(준교사학습 또는 준지도학습으로 번역됨)이라고 한다. 본 장에서는 이를 중 데이터의 전부에 분류가 명시된 경우인 Supervised Learning에 대해서만 살펴본다.

이 장에서는 R을 사용한 머신 러닝 방법을 설명하지만, 머신 러닝 알고리즘 자체는 자세히 설명하지 않는다. 이 책의 내용으로 부족한 부분은 참고 도서[36, 37, 38, 39]와 강의[40]를 참조하기 바란다. 또한 여기서 설명하는 패키지나 알고리즘이 R이 제공하는 전부가 아니다. CRAN Task View: Machine Learning Statistical Learning에 Machine Learning 과 관련한 다양한 알고리즘이 정리되어있다. 대표적으로 the caret package와 DMwR에는 유용한 기능이 많이 포함되어 있으므로 꼭 확인해보기 바란다.

### 1 데이터 탐색

분류 알고리즘의 적용에 앞서 데이터의 모양을 살펴보고 데이터에 대한 직관적인 이해를 높힐

<sup>1)</sup>[http://en.wikipedia.org/wiki/Statistical\\_classification](http://en.wikipedia.org/wiki/Statistical_classification)

필요가 있다. 이러한 용도로 값의 최소, 최대, 평균 등 분포에 대한 기술 통계를 구해보거나 데이터 시각화기법을 사용할 수 있다.

## 1.1 기술 통계

summary()는 doBy 패키지 (페이지 99)에서 살펴본 함수로 데이터에 대한 간략한 분포 정보를 알려준다. 다음은 iris 데이터에 대해 summary()를 적용한 예이다.

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

Species
setosa     :50
versicolor:50
virginica :50
```

데이터의 분포를 살펴보는 또 다른 함수는 Hmisc 패키지의 describe()이다. 특히 데이터내 결측치(NA)의 존재 및 서로다른 값(unique)의 수를 알려주는 점이 편리하다.

```
> library(Hmisc)
> describe(iris)
iris

5 Variables      150 Observations
-----
Sepal.Length
  n missing unique   Mean      .05      .10      .25      .50
  150      0      35  5.843    4.600    4.800    5.100    5.800
                .75      .90      .95
  6.400    6.900    7.255
```

```
lowest : 4.3 4.4 4.5 4.6 4.7, highest: 7.3 7.4 7.6 7.7 7.9
```

Sepal.Width

	n	missing	unique	Mean	.05	.10	.25	.50
150		0	23	3.057	2.345	2.500	2.800	3.000
	.75	.90	.95					
	3.300	3.610	3.800					

```
lowest : 2.0 2.2 2.3 2.4 2.5, highest: 3.9 4.0 4.1 4.2 4.4
```

Petal.Length

	n	missing	unique	Mean	.05	.10	.25	.50
150		0	43	3.758	1.30	1.40	1.60	4.35
	.75	.90	.95					
	5.10	5.80	6.10					

```
lowest : 1.0 1.1 1.2 1.3 1.4, highest: 6.3 6.4 6.6 6.7 6.9
```

Petal.Width

	n	missing	unique	Mean	.05	.10	.25	.50
150		0	22	1.199	0.2	0.2	0.3	1.3
	.75	.90	.95					
	1.8	2.2	2.3					

```
lowest : 0.1 0.2 0.3 0.4 0.5, highest: 2.1 2.2 2.3 2.4 2.5
```

Species

	n	missing	unique
150		0	3

```
setosa (50, 33%), versicolor (50, 33%), virginica (50, 33%)
```

## 1.2 데이터 시각화

가장 간단한 시각화 방법은 `plot()`을 이용하는 것이다. `plot()`은 의외로 데이터를 별 생각없이 넘겨주어도 자동으로 적절한 형태의 그래프를 찾아서 그려준다. 다음은 `iris` 데이터를 `plot()` 해본 예이다.

```
> plot(iris)
> plot(iris$Sepal.Length)
> plot(iris$Species)
```

위 코드의 실행 결과는 다음과 같다.

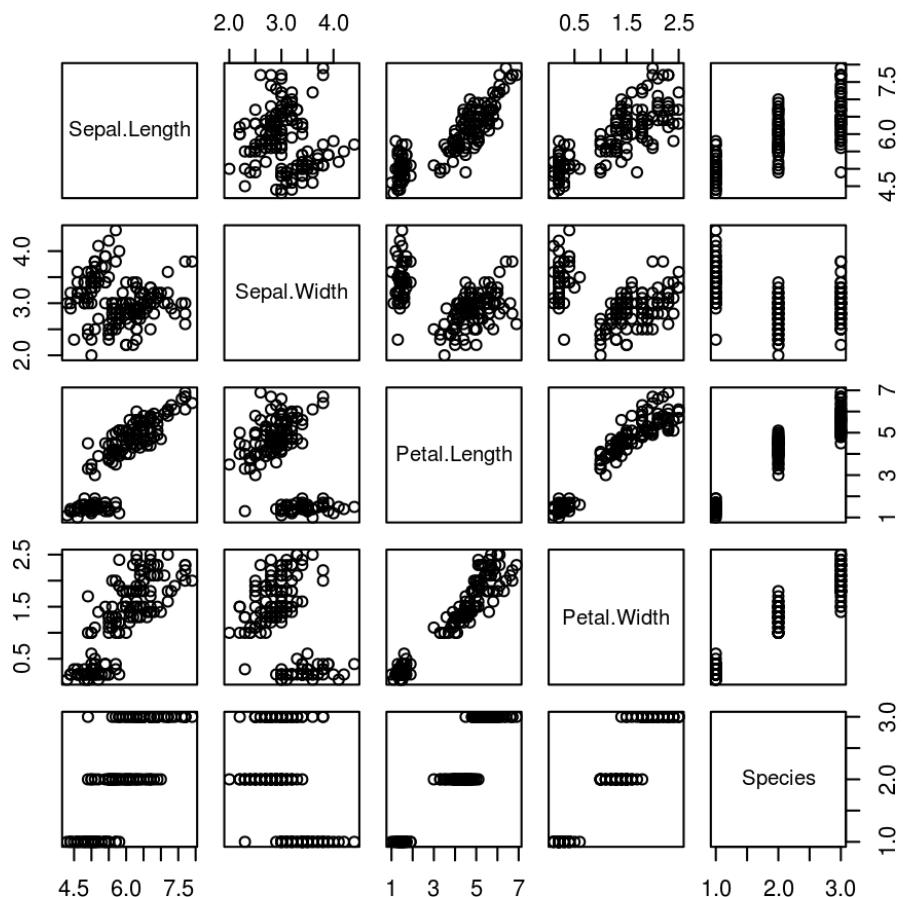


그림 10.1: `plot(iris)`

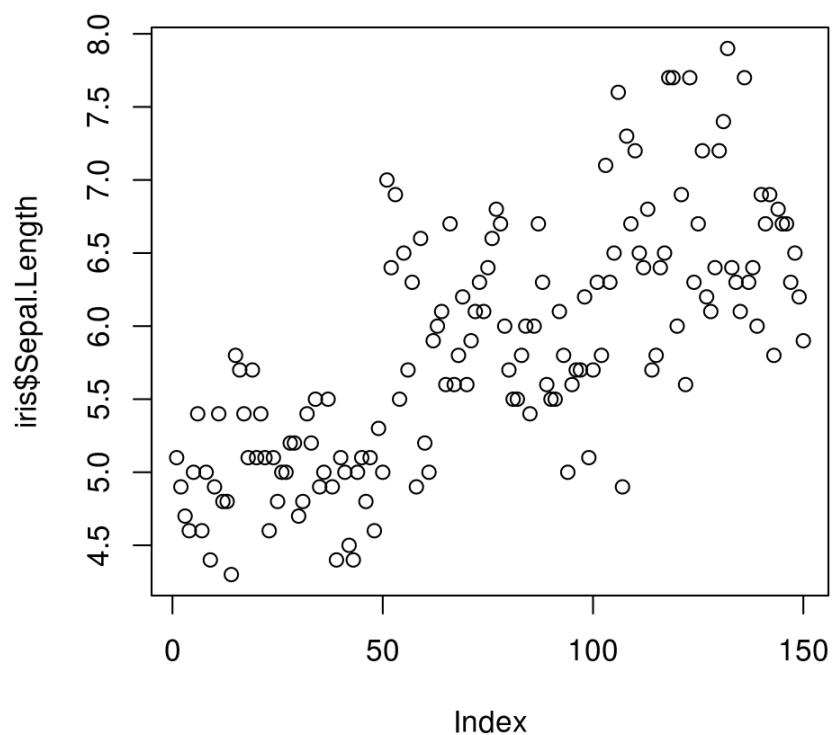


그림 10.2: `plot(iris$Sepal.Length)`

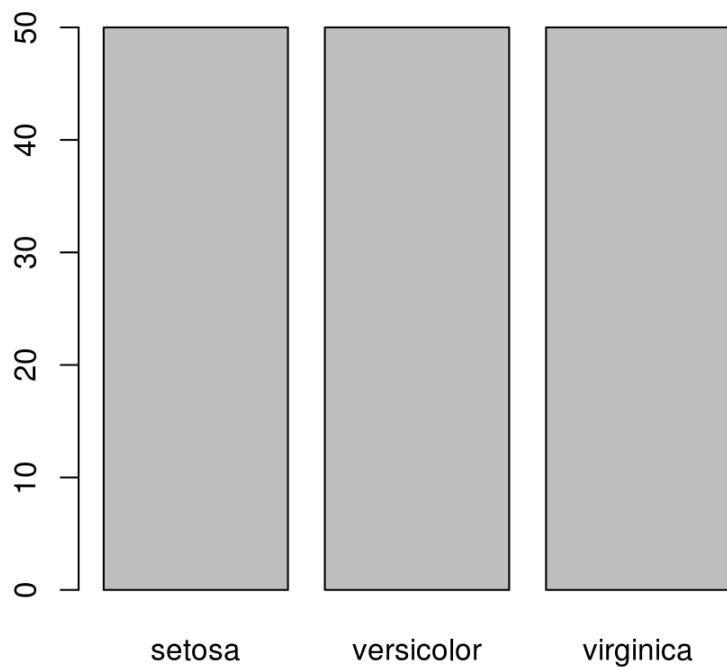


그림 10.3: plot(iris\$Species)

그래프에 표시된 데이터가 붓꽃의 어느 종별인지를 표현하기 위해 formula를 사용해 데이터를 붓꽃 종별로 분리해 그릴 수도 있고, 산점도 상의 점 색상을 붓꽃 종에 따라 다르게 지정할 수도 있다. 먼저 formula를 사용한 예를보자.

```
> plot(iris$Species ~ iris$Sepal.Length, data=iris)
```

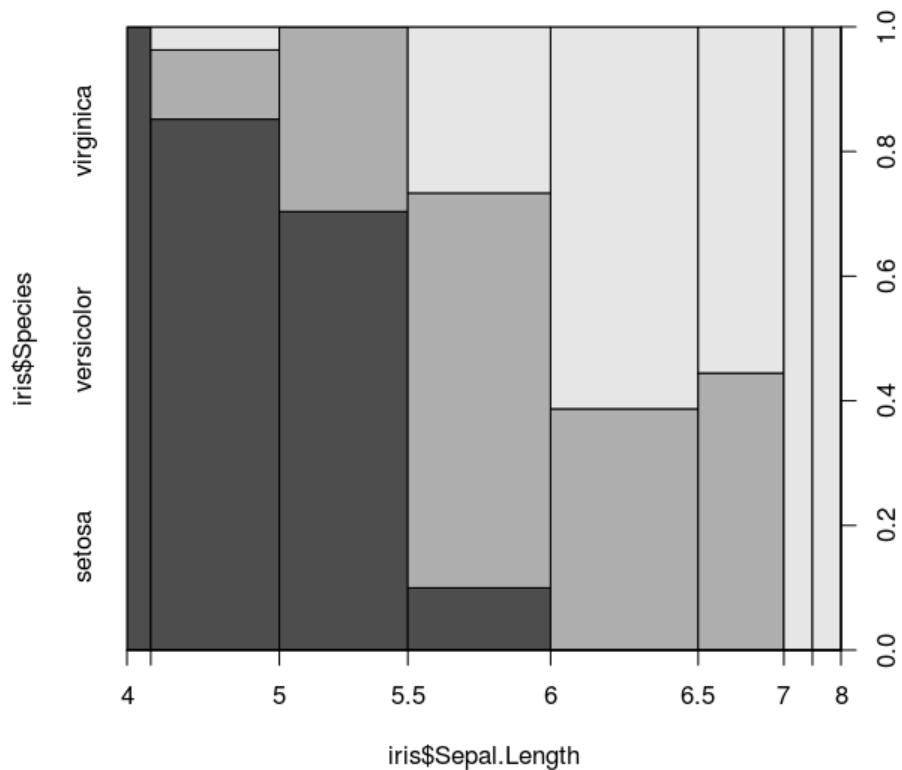


그림 10.4: plot()과 formula의 사용

다음 코드에서는 Factor 타입의 Species를 숫자로 변환해 점의 색상(col)에 지정하였다.

```
> plot(iris$Sepal.Length, col=as.numeric(iris$Species))
```

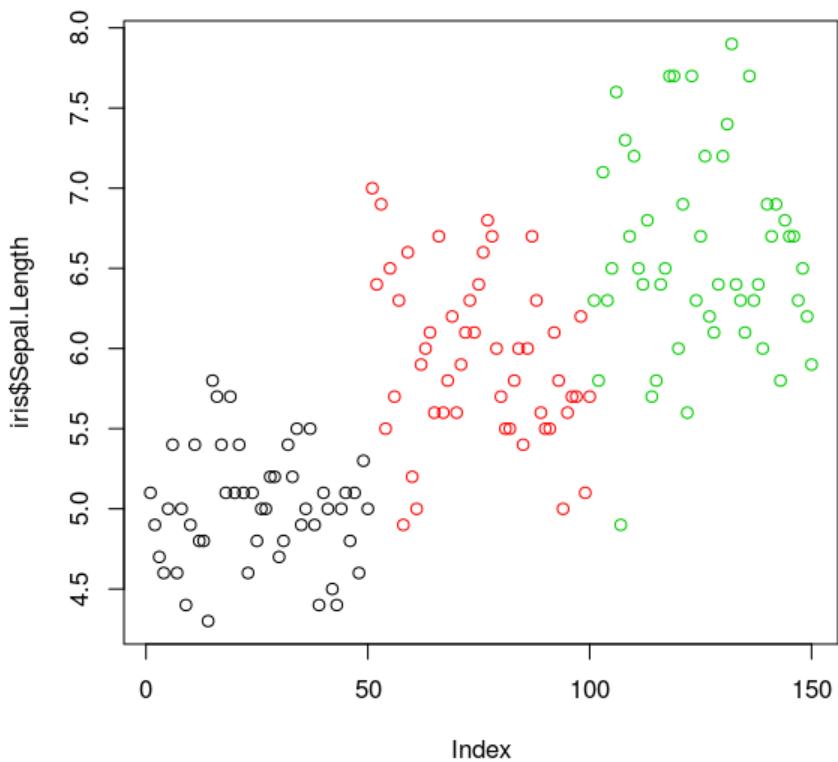


그림 10.5: plot()과 col의 사용

caret 패키지에는 이러한 작업을 편하게 대신해주는 featurePlot() 함수가 있다. featurePlot()은 인자로 X, Y를 받아 X 점들을 Y에 따라 분리해 표현해준다. featurePlot()이 지원하는 그림의 유형에는 ellipse, strip, box, pairs 등이 있다. iris 데이터에 ellipse 유형의 그래프를 그린 예를 아래에 보였다.

```
> library(caret)
> featurePlot(iris[, 1:4], iris$Species, "ellipse")
```

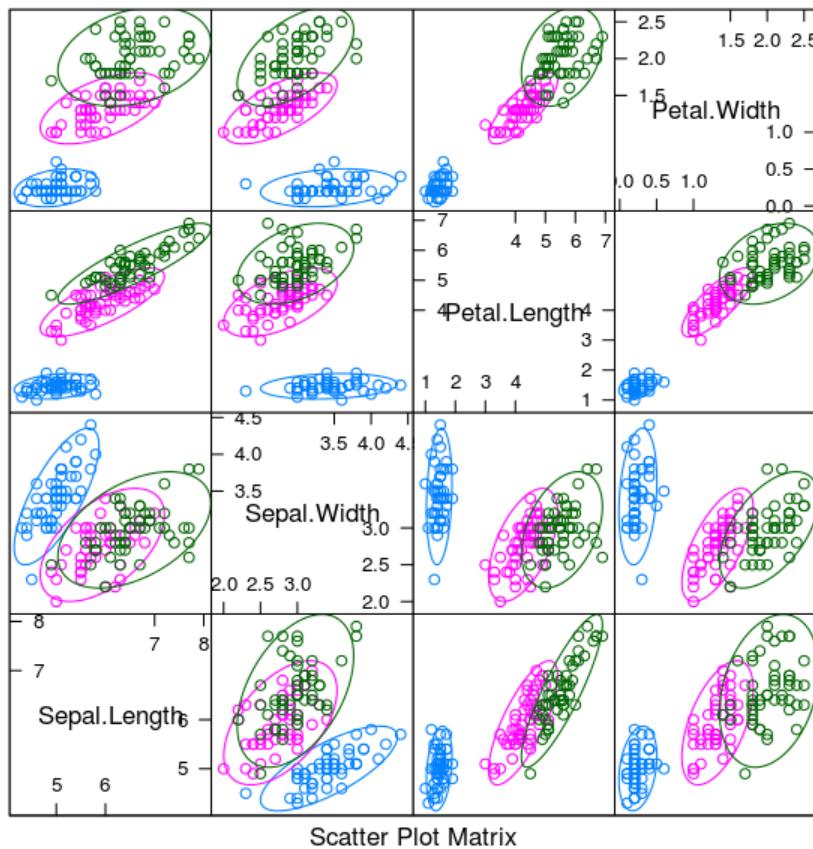


그림 10.6: featurePlot()을 iris에 적용한 예

그림에서 보다시피 붓꽃 종별로 데이터의 분포를 한눈에 파악할 수 있어 모델을 연구하는데 좋은 출발점이 된다.

## 2 전처리(Preprocessing)

분류 알고리즘을 적용하기에 앞서 모델링에 알맞은 형태로 데이터를 처리할 필요가 있다. 이러한 전처리의 예에는 데이터를 정규화하거나, 다른 형태로 재표현하거나, 또는 NA 값을 채워주는 작업 등이 있다.

### 2.1 데이터 변환

#### 데이터 정규화(Feature Scaling)

kNN(k-Nearest Neighbor), SVM, Neural Net 등 많은 분류 알고리즘들에서 좋은 성능을 얻으려면 알고리즘 사용에 앞서 Feature Scaling의 적용이 필요하다. R에서 데이터를 정규화 하는

함수는 scale()이며, 인자로 center, scale를 지정할 수 있다. 이 중 center는 값에서 평균을 뺀지의 여부를, scale은 값을 표준 편차로 나눌지의 여부를 정한다.

다음은 iris 데이터의 값을 정규화한 예이다. scale()이 행렬을 반환하므로 이를 다시 데이터 프레임으로 변환하기 위해 as.data.frame()이 사용되었으며 Species는 정규화에서 제외했다가 후에 cbind()로 합쳤다.

```
> cbind(as.data.frame(scale(iris[1:4])), iris$Species)
   Sepal.Length Sepal.Width Petal.Length Petal.Width iris$Species
1    -0.89767388  1.01560199  -1.33575163 -1.3110521482      setosa
2    -1.13920048 -0.13153881  -1.33575163 -1.3110521482      setosa
3    -1.38072709  0.32731751  -1.39239929 -1.3110521482      setosa
4    -1.50149039  0.09788935  -1.27910398 -1.3110521482      setosa
5    -1.01843718  1.24503015  -1.33575163 -1.3110521482      setosa
6    -0.53538397  1.93331463  -1.16580868 -1.0486667950      setosa
...
...
```

## PCA(Principal Componenet Analysis)

PCA는 차원 감소(Dimensionality Reduction) 또는 데이터를 독립된 차원으로 재표현하기 위하여 사용하는 방법이다. princomp()를 사용해 상관계수행렬에 대한 PCA를 수행해보자.

```
> p <- princomp(iris[, 1:4], cor=TRUE)
```

PCA의 수행후 summary()를 통해 주성분들이 데이터의 분산 중 얼마만큼을 설명해주는지를 알 수 있다. Proportion of Variance 행을 보면 첫번째 주성분(PC1)은 데이터의 분산중 72.96%를 설명해주며 두번째 주성분(PC2)는 데이터의 분산중 22.85%를 설명함을 알 수 있다. 가장 마지막 행의 Cumulative Proportion은 Proportion of Variance의 누적값이다.

```
> summary(p)
Importance of components:
              Comp.1        Comp.2        Comp.3        Comp.4
Standard deviation     1.7083611  0.9560494  0.38308860  0.143926497
Proportion of Variance 0.7296245  0.2285076  0.03668922  0.005178709
Cumulative Proportion  0.7296245  0.9581321  0.99482129  1.000000000
```

좀 더 보기쉽게 Scree Plot을 그려보자.

```
> plot(p, type="l")
```

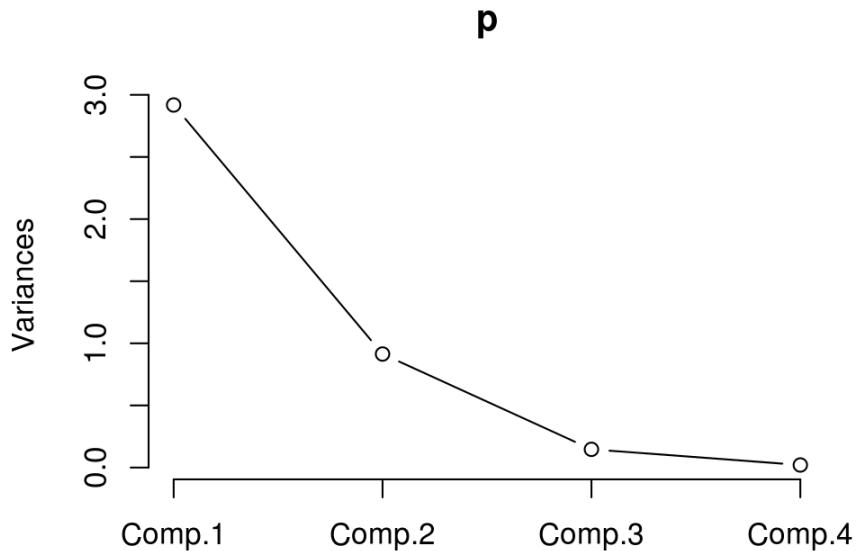


그림 10.7: iris데이터의 PCA후 Scree Plot

iris 데이터를 PCA 변환한 결과는 predict()를 사용해 구한다.

```
> predict(p, iris[, 1:4])
      Comp.1       Comp.2       Comp.3       Comp.4
[1,] -2.26470281 -0.480026597  0.127706022  0.024168204
[2,] -2.08096115  0.674133557  0.234608854  0.103006775
[3,] -2.36422905  0.341908024 -0.044201485  0.028377053
[4,] -2.29938422  0.597394508 -0.091290106 -0.065955560
[5,] -2.38984217 -0.646835383 -0.015738196 -0.035922813
...
```

### 범주형 변수의 재표현

지금까지 하나의 범주형 데이터는 하나의 요인(Factor)형 데이터 변수로만 표현해왔다. 범주형 변수의 수준(level)의 수가 적다면 하나의 Factor 변수로 여러 수준(level)을 표현해도 문제될 게 없으나, 범주 수준의 수가 많다면 경우에 따라 표현방식을 달리해야 할 필요가 있다.

예를들어 Random Forest (페이지 348)의 경우 범주형 변수의 수준의 수를 32개로 제한하고 있다. 이를 확인해보기위해 많은 수의 수준을 가진 범주형 변수를 포함한 데이터를 작성해보자.

아래 코드에서는 all에 총 52개 level을 가진 범주형 변수를 저장하고, value에는 임의의 수를 저장하였다.

```
> (all <- factor(c(paste0(LETTERS, "0"), paste0(LETTERS, "1"))))
[1] A0 B0 C0 D0 E0 F0 G0 H0 I0 J0 K0 L0 M0 N0 O0 P0 Q0 R0 S0 T0
[21] U0 V0 W0 X0 Y0 Z0 A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 K1 L1 M1 N1
[41] O1 P1 Q1 R1 S1 T1 U1 V1 W1 X1 Y1 Z1
52 Levels: A0 A1 B0 B1 C0 C1 D0 D1 E0 E1 F0 F1 G0 G1 H0 ... Z1

> (data <- data.frame(lvl=all, value=rnorm(length(all))))
   lvl      value
1   A0 -2.03998512
2   B0 -0.39505084
3   C0  0.06381953
4   D0 -0.94488257
5   E0  0.49949404
...
...
```

이 데이터를 Random Forest에 입력으로 주면 다음과 같이 32개 이상의 수준은 처리할 수 없다는 에러 메시지가 출력된다. 그 이유는 32개 이상의 수준을 한번의 가지치기로 나누는 경우의 수가 약  $2^{32}$ 에 달하기 때문에 지나치게 계산량이 많다고 본 때문이다.

```
> library(randomForest)
> m <- randomForest(value ~ lvl, data=data)
Error in randomForest.default(m, y, ...) :
  Can not handle categorical predictors with more than 32 categories.
```

이를 해결하기 위해 발생 빈도가 적은 수준들을 하나로 묶거나, 범주형 변수의 수준을 숫자로 취급할 수 있다. 또 다른 방법은 여러개의 가변수(dummy variables)를 사용해 범주형 변수를 재표현 하는 것으로 One Hot Encoding이라고도 불린다.

One Hot Encoding은 model.matrix()를 사용해 구할 수 있다. 다음은 'A', 'B', 'C'의 3개 수준을 저장한 lvl이라는 Factor를 3개의 컬럼으로 재표현한 예이다. model.matrix()의 결과를 보면 A는 (0, 0), B는 (1, 0), C는 (0, 1)로 변환된 것을 알 수 있다.

```
> (x <- data.frame(lvl=factor(c("A", "B", "A", "A", "C")),
+                     value=c(1, 3, 2, 4, 5)))
   lvl value
1     A     1
2     B     3
3     A     2
4     A     4
5     C     5
```

```

2     B      3
3     A      2
4     A      4
5     C      5

> model.matrix(~ lvl, data=x) [, -1]
   lvlB lvlC
1     0    0
2     1    0
3     0    0
4     0    0
5     0    1

```

## 2.2 결측값(NA)의 처리

데이터에 결측치(NA)가 있는 경우 [rpart](#) (페이지 343)는 surrogate 변수를 사용한 유연한 모델링을 제공하지만 [Random Forest](#) (페이지 348)와 같은 모델은 곧바로 에러를 발생시킨다. 이 문제를 해결하기 위해 Random Forest 알고리즘을 제공하는 `randomForest` 패키지는 `rFImpute()` 함수를 제공한다. 이처럼 데이터에 결측치가 있는 경우 해당 분류 모형에서 제공하는 알고리즘을 사용하거나 결측치를 다른 값으로 대치해주는 별도의 패키지를 활용하게 된다.

결측치의 존재를 확인하려면 `complete.cases()`를 사용한다. `complete.cases()`는 데이터 프레임의 각 행마다 적용하며, 각 행에 저장된 모든 값이 NA가 아닐 때에만 TRUE를 반환하므로 NA값이 하나라도 존재하는 행을 찾는데 편리하다. 다음은 `iris`에 일부러 NA값을 입력하고 해당 행들을 다시 찾아내는 예이다.

```

> iris_na <- iris
> iris_na[c(10, 20, 25, 40, 32), 3] <- NA
> iris_na[c(33, 100, 123), 1] <- NA
> iris_na[!complete.cases(iris_na),]

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
10        4.9       3.1         NA       0.1 setosa
20        5.1       3.8         NA       0.3 setosa
25        4.8       3.4         NA       0.2 setosa
32        5.4       3.4         NA       0.4 setosa
33        NA        4.1       1.5       0.1 setosa
40        5.1       3.4         NA       0.2 setosa

```

100	NA	2.8	4.1	1.3	versicolor
123	NA	2.8	6.7	2.0	virginica

만약 한 열에 대해서만 조사를 하고자하면 `is.na()`를 사용한다.

> iris_na[is.na(iris_na\$Sepal.Length), ]						
		Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
33	NA	4.1		1.5	0.1	setosa
100	NA	2.8		4.1	1.3	versicolor
123	NA	2.8		6.7	2.0	virginica

NA 값의 처리를 분류 알고리즘에서 지원하지 않는다면 해당 데이터를 제외하고 모델링 및 예측을 수행하거나, NA 값을 다른 값으로 대치해야 한다. 값을 대치하는 가장 간단한 방법은 데이터의 평균이나 중앙값을 취하는 것이다.

다음 코드는 `iris_na`에서 중앙값을 구한 예이다. `median()` 호출시 `na.rm=TRUE`는 NA를 제외하고 중앙값을 계산하기 위해 사용되었다. 만약 `na.rm=TRUE`를 지정하지 않으면 NA 값이 포함된 연산의 결과는 NA이므로 제대로 된 중앙값을 얻을 수 없게 된다. `na.rm=TRUE`는 `mapply()`의 인자로 주어지지만 `mapply()`내에서 직접 사용되는 것은 아니고, `median()`을 호출할 때 `median()`의 인자로 넘겨지게 된다. `mapply()`는 앞서 [mapply\(\)](#) (페이지 99)에서 살펴보았다.

> mapply(median, iris_na[1:4], na.rm=TRUE)				
Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	
5.8	3.0	4.4	1.3	

이 값을 실제로 `iris_na`에서 NA 값이 위치한 곳에 대치하는 것이 번거롭기 때문에 이러한 일을 대신해주는 DMwR 패키지를 사용해 NA를 중앙값으로 대치해보자.

> library(DMwR)					
> iris_na[!complete.cases(iris_na), ]					
	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
10	4.9	3.1	NA	0.1	setosa
20	5.1	3.8	NA	0.3	setosa
25	4.8	3.4	NA	0.2	setosa
32	5.4	3.4	NA	0.4	setosa
33	NA	4.1	1.5	0.1	setosa
40	5.1	3.4	NA	0.2	setosa
100	NA	2.8	4.1	1.3	versicolor
123	NA	2.8	6.7	2.0	virginica

```
> centralImputation(iris_na[1:4]) [
+   c(10, 20, 25, 32, 33, 40, 100, 123), ]
  Sepal.Length Sepal.Width Petal.Length Petal.Width
10          4.9         3.1        4.4       0.1
20          5.1         3.8        4.4       0.3
25          4.8         3.4        4.4       0.2
32          5.4         3.4        4.4       0.4
33          5.8         4.1        1.5       0.1
40          5.1         3.4        4.4       0.2
100         5.8         2.8        4.1       1.3
123         5.8         2.8        6.7       2.0
```

단순한 중앙값 대신 모델을 만들어 NA를 대체할 값을 구할 수도 있다. 다음은 NA 값을 kNN 을 사용해 k개 근접 이웃의 가중 평균으로 대치한 예이다. 가중치는 NA값이 있는 데이터 와의 거리로부터 계산되며, knnImputation()이 자동으로 데이터 정규화를 수행하기에 scale()을 별도로 수행하지는 않았다.

```
> knnImputation(iris_na[1:4]) [c(10, 20, 25, 32, 33, 40, 100, 123), ]
  Sepal.Length Sepal.Width Petal.Length Petal.Width
10      4.900000     3.1      1.452250     0.1
20      5.100000     3.8      1.539881     0.3
25      4.800000     3.4      1.457144     0.2
32      5.400000     3.4      1.483821     0.4
33      5.462532     4.1      1.500000     0.1
40      5.100000     3.4      1.475718     0.2
100     5.891169     2.8      4.100000     1.3
123     7.077197     2.8      6.700000     2.0
```

패키지 [DMwR](#)외에도 [imputation](#) 등의 패키지를 사용해 NA값의 대치를 수행할 수 있으며, 이를 위해 선형 모델이나 bagged tree 등의 모델을 사용할 수 있다.

위에 보인 방법들은 NA를 대체할 값을 구할 때 Species를 활용하지 않음에 주목하기 바란다. 예를들어 Sepal.Length를 중앙값으로 대체할 때 단순히 Sepal.Length 전체의 중앙값을 사용했으며, 붓꽃 종별로 중앙값을 계산해 사용하지 않았다. 만약 각 행의 Species 값을 참고해 Species.Length의 NA를 해당 붓꽃 종의 중앙값으로 대체한다면 NA가 있던 자리에 붓꽃 종별에 대한 직접적인 정보가 흘러들어갈 수 있다. 이로인해 예측 모델을 만들었을 때 Sepal.Length의 예측력이 실제보다 크게 나타날 수 있다.

## 2.3 변수 선택(Feature Selection)

주어진 데이터의 전부를 사용하기보다는 그 중 모델링에 가장 적합한 변수만 택하는 과정을 Feature Selection이라 한다.

변수 선택 방법은 특정 모델링 기법에 의존하지 않고 데이터의 통계적 특성<sup>2)</sup>으로부터 변수를 택하는 Filter Method와 변수의 일부만을 모델링에 사용하고 그 결과를 확인하는 작업을 반복하면서 변수를 택해나가는 Wrapper Method, 모델 자체에 변수 선택이 포함된 Embedded methods<sup>3)</sup>로 분류된다[41].

또한 Feature Selection은 예측대상이 되는 분류를 보지 않고 변수들만으로 수행하는 비지도(Unsupervised) 방식과 분류를 참고하여 변수를 선택하는 지도(Supervised) 방식으로도 분류할 수 있다.

이 절에서는 이들 방법중 대표적인 몇가지 기법들을 살펴본다.

### 0에 가까운 분산(Near Zero Variance)

변수를 선택하는 기법중 가장 단순한 방법은 변수값의 분산을 보는 것이다. 예를들어 어떤 변수의 값이 0인 경우가 9990개, 1인 경우가 10개 존재하는 총 1000개의 표본으로 구성된 데이터가 있다면 해당 변수는 데이터 모델링시에 별 효과가 없음을 쉽게 짐작할 수 있다.

caret 패키지의 nearZeroVar()는 이와같이 분산이 작은 변수를 식별하는데 사용하는 함수이다. mlbench 패키지의 콩(Soybean) 질병 데이터에 nearZeroVar()를 적용해보자.

```
> library(caret)
> library(mlbench)
> data(Soybean)
> nearZeroVar(Soybean, saveMetrics=TRUE)

      freqRatio percentUnique zeroVar     nzv
Class          1.010989      2.7818448 FALSE FALSE
date           1.137405      1.0248902 FALSE FALSE
plant.stand    1.208191      0.2928258 FALSE FALSE
precip         4.098214      0.4392387 FALSE FALSE
temp           1.879397      0.4392387 FALSE FALSE
hail            3.425197      0.2928258 FALSE FALSE
crop.hist       1.004587      0.5856515 FALSE FALSE
area.dam        1.213904      0.5856515 FALSE FALSE
sever           1.651282      0.4392387 FALSE FALSE
```

<sup>2)</sup>예를들어 Mutual Information이나 피어슨 상관계수(Pearson Correlation Coefficient) (페이지 245)

<sup>3)</sup>예를들어 LASSO

seed.tmt	1.373874	0.4392387	FALSE	FALSE
germ	1.103627	0.4392387	FALSE	FALSE
plant.growth	1.951327	0.2928258	FALSE	FALSE
leaves	7.870130	0.2928258	FALSE	FALSE
leaf.halo	1.547511	0.4392387	FALSE	FALSE
leaf.marg	1.615385	0.4392387	FALSE	FALSE
leaf.size	1.479638	0.4392387	FALSE	FALSE
leaf.shread	5.072917	0.2928258	FALSE	FALSE
leaf.malf	12.311111	0.2928258	FALSE	FALSE
leaf.mild	26.750000	0.4392387	FALSE	TRUE
stem	1.253378	0.2928258	FALSE	FALSE
lodging	12.380952	0.2928258	FALSE	FALSE
stem.cankers	1.984293	0.5856515	FALSE	FALSE
canker.lesion	1.807910	0.5856515	FALSE	FALSE
fruiting.bodies	4.548077	0.2928258	FALSE	FALSE
ext.decay	3.681481	0.4392387	FALSE	FALSE
mycelium	106.500000	0.2928258	FALSE	TRUE
int.discolor	13.204545	0.4392387	FALSE	FALSE
sclerotia	31.250000	0.2928258	FALSE	TRUE
fruit.pods	3.130769	0.5856515	FALSE	FALSE
fruit.spots	3.450000	0.5856515	FALSE	FALSE
seed	4.139130	0.2928258	FALSE	FALSE
mold.growth	7.820896	0.2928258	FALSE	FALSE
seed.discolor	8.015625	0.2928258	FALSE	FALSE
seed.size	9.016949	0.2928258	FALSE	FALSE
shriveling	14.184211	0.2928258	FALSE	FALSE
roots	6.406977	0.4392387	FALSE	FALSE

위 결과에서 보듯이 `nearZeroVar()` 호출시 `saveMetrics=TRUE`를 지정하면 분석 결과의 표가 출력된다. 이 표에서 nzv컬럼은 Near Zero Variance를 뜻하므로 nzv 컬럼에 TRUE로 표시된 변수들을 제거할 수 있다.

`nearZeroVar()` 호출시 `saveMetrics`을 지정하지 않으면 분산이 0에 가까운 변수에 해당하는 컬럼 번호를 곧바로 출력해준다. 따라서 이를 사용해 손쉽게 분산이 0에 가까운 컬럼들을 제거 할 수 있다.

```
> nearZeroVar(Soybean)
[1] 19 26 28
```

```
> mySoybean <- Soybean[, -nearZeroVar(Soybean)]
```

## 상관 계수(Correlation)

변수간 높은 상관계수가 존재할 경우 모델을 불안정하게 만들수 있다. 따라서 상관관계가 높은 변수들이 있다면 이들을 PCA(Principal Componenet Analysis) (페이지 312)와 같은 방법을 사용해 서로 독립된 차원으로 변환하거나, 상관계수가 큰 변수들을 제거할 수 있다.

caret의 findCorrelation()은 상관계수 행렬을 입력으로 받아 임의의 변수와 다른 모든 변수 간의 상관계수의 평균을 계산한 뒤, 이 값이 주어진 threshold를 넘을 경우 해당 변수를 제거 가능한 변수로 나열한다. findCorrelation()의 threshold 기본값은 0.90이다.

4가지 종류의 자동차에 대한 속성을 나열한 데이터인 mlbench의 Vehicle에 대해 findCorrelation()을 적용해 보자. 다음 코드에서 subset()은 Class 열을 제거하는 목적으로 사용되었고, cor()은 상관계수 행렬을 계산하는 함수이다.

```
> library(mlbench)
> data(Vehicle)
> findCorrelation(cor(subset(Vehicle, select=-c(Class))))
[1] 3 8 11 7 9 2
```

실행 결과 3, 8, 11, 7, 9, 2 번째 컬럼의 상관계수가 높은 것으로 나타났으므로 이를 제거할 수 있다.

```
> myVehicle <- Vehicle[, -c(3, 8, 11, 7, 9, 2)]
```

반대로 상관계수는 변수 선택에 사용할 수도 있다. FSelector[42] 패키지의 linear.correlation()과 rank.correlation()은 상관계수로부터 변수의 중요도를 구하는 함수로서 각각 피어슨 상관계수(Pearson Correlation)와 스피어만 상관계수(Spearman's Correlation)를 사용 한다. 다음은 mlbench의 Ozone 데이터에서 예측 대상이되는 변수인 V4와 나머지 변수들간의 Pearson Correlation을 구하고 이로부터 변수의 중요도를 평가한 예이다. V1, V2, V3은 Factor 형 변수이므로 계산에서 제외되었다.

```
> library(mlbench)
> data(Ozone)
> (v <- linear.correlation(V4 ~ .,
+                           data=subset(Ozone, select=-c(V1, V2, V3))))
attr_importance
V5          0.58414447
V6          0.00468138
```

V7	0.44356639
V8	0.76986408
V9	0.72317299
V10	0.58026757
V11	0.22990285
V12	0.73194978
V13	0.41471463

실행결과 V18, V12, V9 컬럼 등이 V4와 상관계수가 큰 것으로 보인다.

v 는 attr\_importance 컬럼 하나만을 갖는 데이터 프레임이다. 따라서 이를 다음과 같이 정렬해 볼 수 있다.

> v[order(-v), , drop=FALSE]	attr_importance
V8	0.76986408
V12	0.73194978
V9	0.72317299
V5	0.58414447
V10	0.58026757
V7	0.44356639
V13	0.41471463
V11	0.22990285
V6	0.00468138

데이터 프레임 v는 order(-v)를 사용해 내림차순으로 정렬되었다. 데이터 정렬시 v[order(-v), ] 명령을 사용하면 데이터 프레임에 컬럼이 하나밖에 없어 결과가 벡터로 변환되어 버린다. 따라서 drop=FALSE를 사용해 데이터가 벡터로 변환되는것을 막았다.

## Chi Square

독립성 검정(Independence Test) (페이지 233)에서 변수간 독립의 여부를 Chi-squared test를 사용해 살펴보았다. 마찬가지 방법을 예측대상이 되는 분류와 변수간에 수행하여 변수와 분류간의 독립성을 검정해 볼 수 있다. 만약 둘간의 관계가 독립이라면 해당 변수는 모델링에 적합하지 않은 것으로 볼 수 있다. 반대로 둘간의 관계가 독립이아니라면 해당 변수는 모델링에 중요한 변수로 볼 수 있다.

다음은 mlbench의 Vehicle 데이터에 대해 FSelector의 chi.squared()를 사용해 데이터의 중요도를 평가한 결과이다.

```
> chi.squared(Class ~., data=Vehicle)
      attr_importance
Comp          0.3043172
Circ          0.2974762
D.Circ        0.3587826
Rad.Ra         0.3509038
Pr.Axis.Ra    0.2264652
Max.L.Ra      0.3234535
Scat.Ra        0.4653985
Elong          0.4556748
Pr.Axis.Rect   0.4475087
Max.L.Rect     0.3059760
Sc.Var.Maxis   0.4338378
Sc.Var.maxis   0.4921648
Ra.Gyr         0.2940064
Skew.Maxis     0.3087694
Skew.maxis     0.2470216
Kurt.maxis     0.3338930
Kurt.Maxis     0.2732117
Holl.Ra         0.3886266
```

### 모델을 사용한 변수 중요도 평가

caret의 [varImp\(\)](#)는 다양한 모델로부터 변수 중요도를 측정할 수 있게 해준다. 뒤에서 살펴볼 rpart (페이지 343)를 사용해 mlbench의 유방암데이터에 대한 나무 모형을 만들고 이로부터 변수 중요도를 찾아보자.

```
> library(mlbench)
> library(rpart)
> library(caret)
> data(BreastCancer)
> m <- rpart(Class ~., data=BreastCancer)
> varImp(m)
      Overall
Bare.nuclei    203.7284
Bl.cromatin    197.9057
Cell.shape     216.3834
```

Cell.size	222.9401
Id	307.8953
Cl.thickness	0.0000
Marg.adhesion	0.0000
Epith.c.size	0.0000
Normal.nucleoli	0.0000
Mitoses	0.0000

부족하지만 이 정도로 변수 선택을 하는 방법들이 어떻게 사용되는지 감을 잡았으리라 생각한다. 추가적인 부분은 caret의 [feature selection](#) 메뉴얼과 FSelector의 [Reference manual](#) 등을 참고하기 바란다.

### 3 모델 평가 방법

이 절에서는 예측 모델의 성능을 평가하는 방법들을 살펴본다.

#### 3.1 평가 메트릭(metric)

모델의 성능을 평가하는 기준에는 precision, recall, kappa, lift 등이 있으며 ROC curve 등으로 시각화해 볼 수 있다. 참고문헌[43, 2]에 이들의 정의에 대해서 설명되어 있으므로 여기서는 계산 방법에 대해서만 알아본다.

예측결과가 담긴 predicted 벡터와 이들의 실제 분류가 담긴 actual 벡터를 정의해보자.

```
> predicted <- c(1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1)
> actual     <- c(1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1)
```

예측결과와 실제 분류가 주어졌을 때 가장 간단하게 상태를 볼 수 있는 방법은 [분할표](#) (Contingency Table) (페이지 231)를 작성하는 것이다.

```
> xtabs(~ predicted + actual)
      actual
predicted 0 1
      0 3 2
      1 1 6
```

이 표를 통해서 예측 결과와 실제 결과가 일치하는 경우와 그렇지 않은 경우를 쉽게 알 수 있다. 앞서 설명했던대로 prop.table()을 사용하면 비율역시 쉽게 계산가능하다.

Accuracy는 예측값중 올바른 값의 비율로 다음과 같이 계산한다.

```
> sum(predicted == actual) / NROW(actual)
[1] 0.75
```

이렇게 매 경우를 하나하나 코드를 작성해 계산할 수도 있겠지만 caret의 confusionMatrix()를 사용하면 손쉽게 정리된 결과를 얻을 수 있다.

```
> confusionMatrix(predicted, actual)
Confusion Matrix and Statistics

Reference
Prediction 0 1
      0 3 2
      1 1 6

Accuracy : 0.75
95% CI : (0.4281, 0.9451)
No Information Rate : 0.6667
P-Value [Acc > NIR] : 0.3931

Kappa : 0.4706
McNemar's Test P-Value : 1.0000

Sensitivity : 0.7500
Specificity : 0.7500
Pos Pred Value : 0.6000
Neg Pred Value : 0.8571
Prevalence : 0.3333
Detection Rate : 0.2500
Detection Prevalence : 0.4167

'Positive' Class : 0
```

만약 세부 결과를 바로 얻고자한다면 str()을 사용해 구조를 살펴보고 개별 메트릭을 가져올 수 있다.

```
> cm <- confusionMatrix(predicted, actual)
> str(cm)
```

```
List of 5
$ positive: chr "0"
$ table    : 'table' int [1:2, 1:2] 3 1 2 6
.. - attr(*, "dimnames")=List of 2
.. .. $ Prediction: chr [1:2] "0" "1"
.. .. $ Reference : chr [1:2] "0" "1"
$ overall  : Named num [1:7] 0.75 0.471 0.428 0.945 0.667 ...
.. - attr(*, "names")= chr [1:7] "Accuracy" "Kappa" ...
$ byClass : Named num [1:7] 0.75 0.75 0.6 0.857 0.333 ...
.. - attr(*, "names")= chr [1:7] "Sensitivity" "Specificity" ...
$ dots     : list()
- attr(*, "class")= chr "confusionMatrix"
> cm$overall["Accuracy"]
Accuracy
0.75
```

## 3.2 ROC 커브

ROCR 패키지[44]는 ROC 커브, Recall/Precision 차트, Lift 차트 등 다양한 성능 평가 시각화 기능을 제공한다.

다음과 같이 예측값과 실제 분류가 주어졌다고 가정해보자. probs는 분류 알고리즘이 예측한 점수이고 labels는 정답에 해당하는 분류(true class)가 저장된 벡터이다. labels 내의 ifelse는 약간의 분류 실패를 시뮬레이션 해 본 것이다.

```
> probs <- runif(100)
> labels <- as.factor(ifelse(probs > .5 & runif(100) < .4, "A", "B"))
```

ROCR을 사용하기 위해 prediction 객체를 만든다.

```
> library(ROCR)
> pred <- prediction(probs, labels)
```

prediction 객체를 performance() 함수에 넘겨 분류 알고리즘의 다양한 성능을 얻을 수 있다. performance()가 제공하는 성능 메트릭에는 tpr(True Positive Rate), fpr(False Positive Rate), acc(Accuracy), rec(Recall) 등 많은 지표가 있다. help(performance)를 사용해 자세한 목록을 확인해보도록하고, 아래 예에서는 tpr, fpr을 사용해 ROC 커브를 그려보도록 하자.

```
> plot(performance(prediction(probs, labels), "tpr", "fpr"))
```

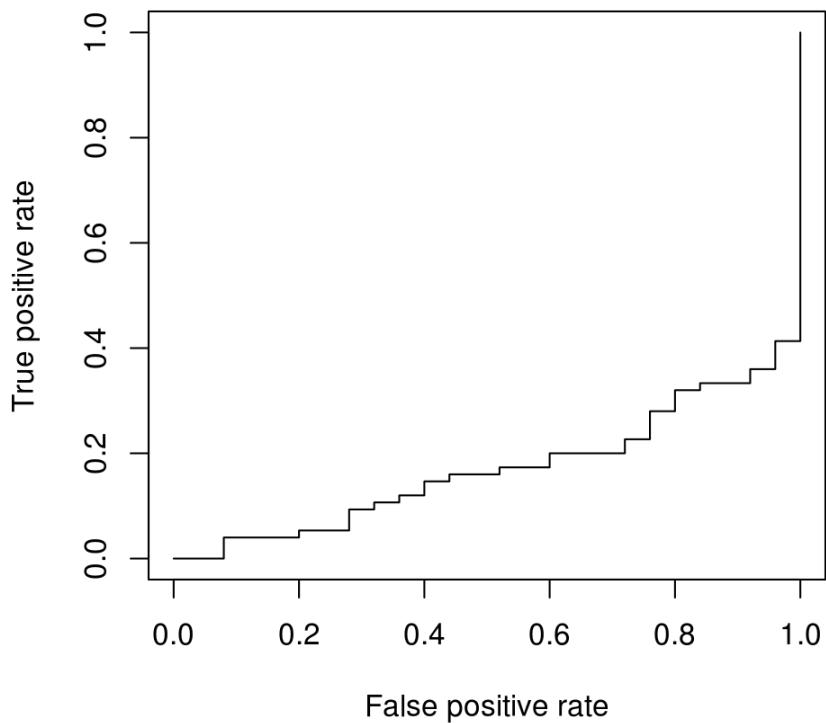


그림 10.8: ROCR 패키지를 사용한 ROC 커브

마찬가지로 acc, cutoff 를 인자로 지정하면 cutoff 값에 따른 Accuracy의 변화를 볼 수 있다.

```
> plot(performance(prediction(probs, labels), "acc", "cutoff"))
```

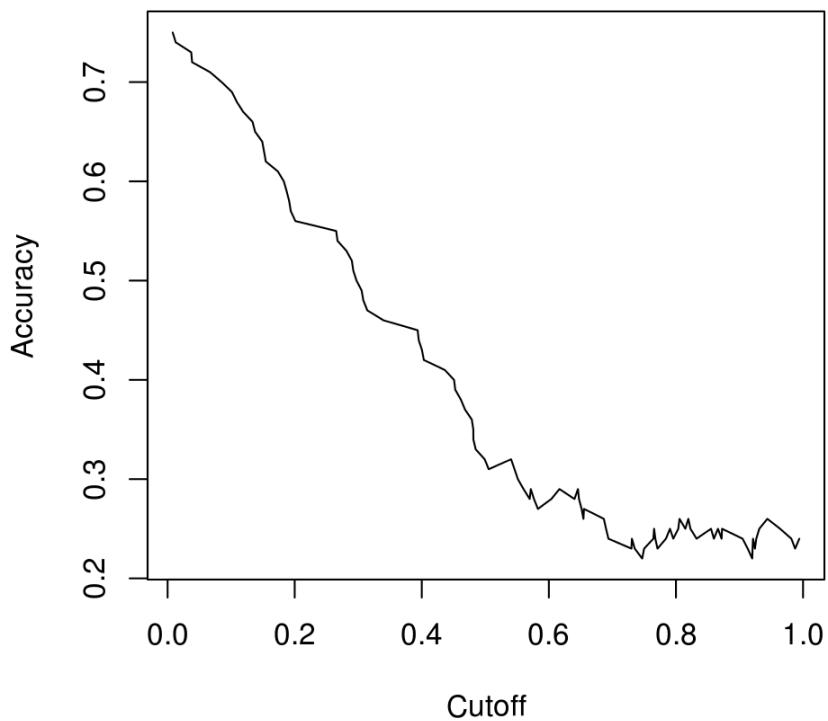


그림 10.9: ROCR 패키지를 사용한 Accuracy/Cutoff 차트

AUC(Area Under the Curve)는 auc 인자를 지정하고 y.values를 보면 된다.

```
> performance(pred, 'auc')
An object of class "performance"
Slot "x.name":
[1] "None"

Slot "y.name":
[1] "Area under the ROC curve"

Slot "alpha.name":
[1] "none"

Slot "x.values":
list()
```

```
Slot "y.values":  
[[1]]  
[1] 0.8341875
```

```
Slot "alpha.values":  
list()
```

위 코드의 실행결과 AUC는 0.8341875로 나타났다.

### 3.3 교차 검증(cross validation)

주어진 데이터 전체를 사용해 모델을 만들 경우, 해당 데이터에는 잘 동작하지만 새로운 데이터에는 좋지 않은 성능을 보이는 모델을 만들 가능성 있다. 이러한 사례 중 과적합(overfitting)에 대해서 살펴보자.

#### 과적합(Overfitting)

과적합은 주어진 데이터로부터 보장되는 것 이상으로 모델을 만들 때 발생한다[45]. 다음에 보인 그림 10.10은 위키피디아 Overfitting 항목의 예를 단순화한 그림이다. 좌표평면 위의 점들은 주어진 데이터를 뜻하며, 직선과 점선은 해당 점들을 설명하는 모델을 의미한다.

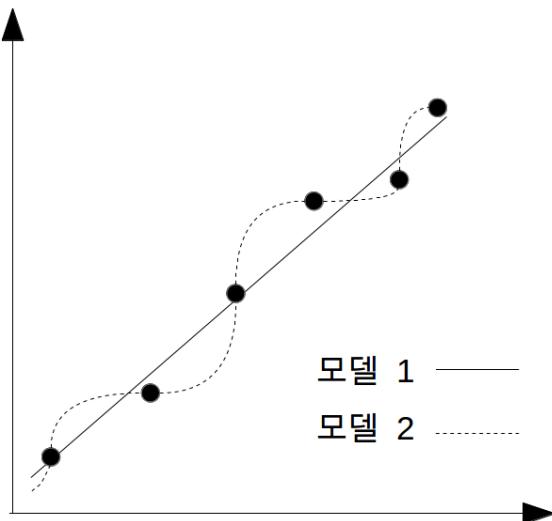


그림 10.10: 과적합의 예

그림 10.10에서 직선으로 표현된 모델 1은 단순한 직선으로 데이터를 표현한 모델이다. 반면 점선으로 표현한 모델 2는 높은 차수의 다항식으로 데이터를 모델링한 예이다. 이 데이터로만

보면 모델1이 모델2에 비해 Accuracy 등의 지표가 낮다. 그러나 데이터의 분포를 보면 복잡한 곡선으로부터 나온 데이터가 아니라 단순 선형 관계로부터 나온 데이터일 가능성이 높아보인다. 즉 모델1이 모델2에 비해 보다 더 일반적인 모델일 가능성이 있고, 따라서 새로운 데이터에 대한 예측을 더 잘 수행하는 모델일 수 있다.

### 검증 데이터(Validation Data)

직선을 선택할 것인지 또는 고차의 다항식을 선택할 것인지의 문제처럼 여러가지 모델을 놓고 어떤 모델이 새로운 데이터가 주어졌을 때 더 잘 동작할지를 추정해야 할 때가 있다. 뿐만 아니라 한가지 모델을 놓고 어떤 파라미터를 어떻게 설정하는 것이 더 나은지 결정해야 할 상황도 있을 수 있다. 마지막으로 특정 모델이 더 잘 동작할 것으로 보인다면, 과연 새로운 데이터가 주어졌을 때 얼마나 잘 동작할 것인지 그 성능을 추정할 필요도 있다.

데이터가 새로운 데이터에 얼마나 잘 동작할 것인지를 판단하는 방법 중 한 가지는 데이터의 일부를 따로 검증(validation) 데이터로 떼어놓고 모델 평가에 사용하는 것이다. 이 방법의 수행 단계는 다음과 같다.

1. 데이터의 일부를 훈련 데이터(training data), 나머지를 검증 데이터(validation data)로 분리한다.
2. 훈련 데이터로부터 모델을 만든다.
3. 만들어진 모델을 검증 데이터에 대해 적용해 그 성능을 평가한다. 성능이 만족스럽지 않다면 2단계로 돌아간다.
4. 전체 데이터로부터 모델을 만들고 이를 최종 모델로 정한다.

위에 설명한 절차를 그림 10.11에 보였다.

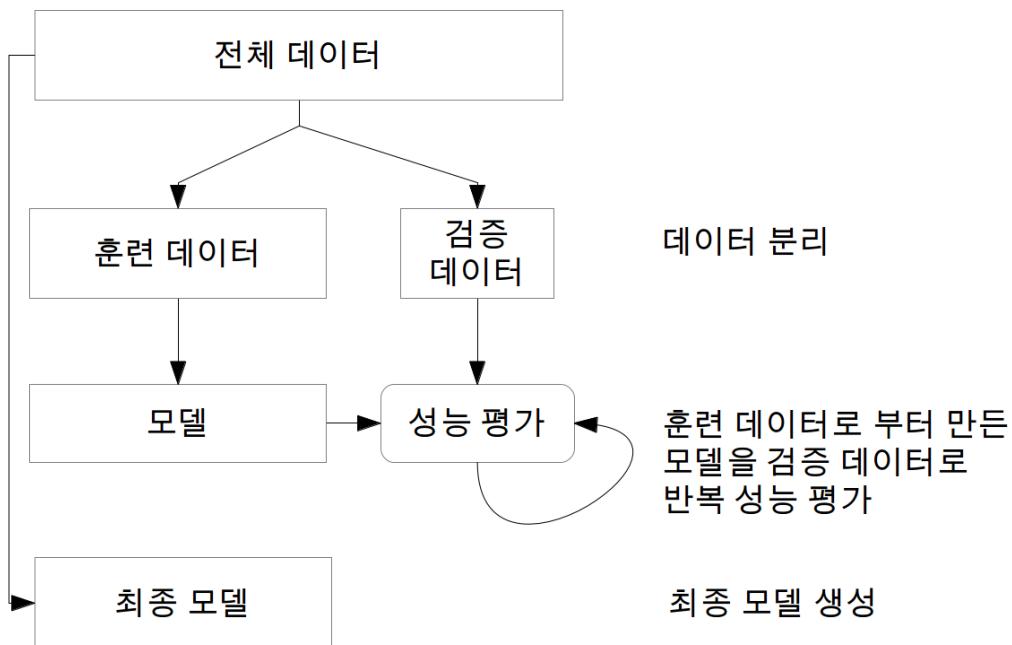


그림 10.11: 훈련데이터와 검증 데이터를 사용한 모델링 과정

### 교차 검증(Cross Validation)

데이터의 어느정도를 훈련 데이터로 하고 또 어느정도를 검증 데이터로 할지의 결정은 쉽지 않다. 만약 훈련 데이터의 크기를 너무 작게한다면 모델링에 사용할 데이터가 적어 실제 도달 가능한 성능보다 낮은 성능의 모델이 만들어 질 것이다. 이 경우 검증 데이터로부터 추정한 성능은 실제보다 낮은 값이 된다. 반대로 훈련 데이터의 크기를 너무 크게한다면 검증 데이터의 크기가 작아진다. 이 경우 적은 수의 데이터로만 검증을 수행하게 되어 계산한 성능의 신뢰도가 낮아진다.

이 문제를 개선하는 한가지 방법은 교차 검증이다. 교차 검증은 검증 데이터와 훈련데이터를 분리하여 모델링 및 평가하는 작업을 K회 반복하는 것으로 이를 K겹 교차검증(K-fold Cross Validation)이라 한다. 보통 K값은 10으로 지정한다. 다음은 10겹 교차검증(10-fold Cross Validation)의 수행 단계이다.

1. 데이터를 10등분하여  $D_1, D_2, \dots, D_{10}$ 으로 분할한다.
2. K값을 1로 초기화한다.
3.  $D_K$ 를 검증 데이터, 그외의 데이터를 훈련데이터로하여 모델을 생성한다.
4. 검증 데이터  $D_K$ 를 사용해 모델의 성능을 평가한다. 평가된 모델의 성능을  $P_K$ 라 한다.
5. K가 9이하의 값이면  $K=K+1$ 을 하고 3단계로 간다. 만약 K=10이면 종료한다.

예를 들어  $K=1$ 이라면  $D_1$ 이 검증 데이터,  $D_2, D_3, \dots, D_{10}$ 이 훈련데이터로 사용된다. 만약  $K=2$ 라면  $D_2$ 이 검증데이터,  $D_1, D_3, D_4, \dots, D_{10}$ 이 훈련데이터로 사용된다. 이처럼 각  $K$ 마다  $D_K$ 를 검증데이터로 사용해나간다. 최종적으로  $K=10$ 일 경우  $D_{10}$ 이 검증데이터,  $D_1, D_2, \dots, D_9$ 가 훈련데이터로 사용된다.

이 단계를 거치면 전체 데이터의 성능은  $P_1, P_2, \dots, P_{10}$ 으로 구해지며 최종 성능은 단순히 이를 값의 산술 평균으로 정할 수 있다.

그림 10.12에 8겹 교차 검증의 예를 보였다. 그림으로부터  $K=4, 5, 6, 7, 8$  일 때의 검증 데이터 역시 쉽게 짐작할 수 있을 것이다.

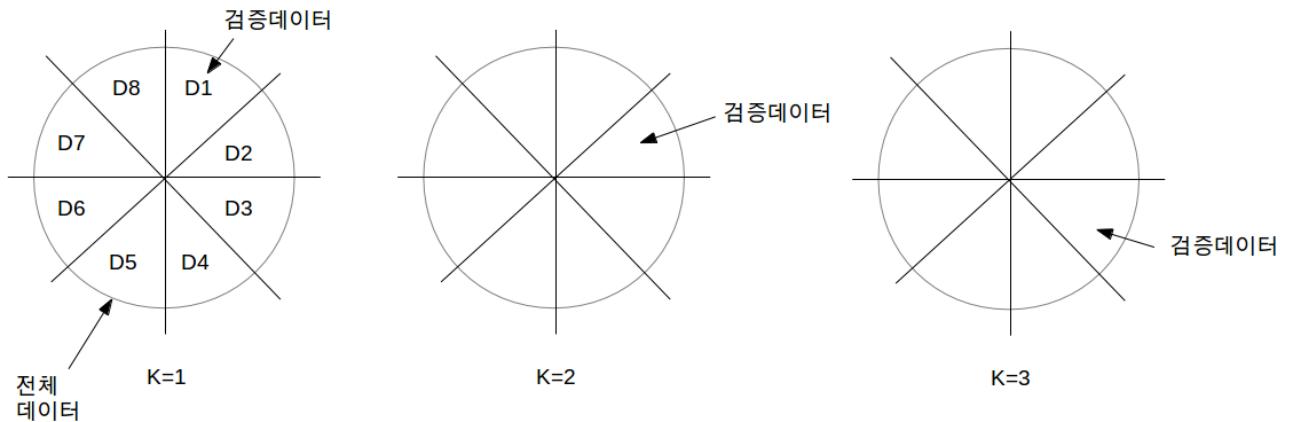


그림 10.12: 8겹 교차검증에서  $K=1, 2, 3$ 일 때 검증 데이터

경우에 따라서는 평가의 정확도를 더 높이기 위해  $K$ 겹 교차 검증을  $R$ 회 반복할 수 있다. 예를들면 10겹 교차검증을 3회 반복하는 식이다.

### 테스트 데이터(Test Data)

교차 검증도 검증 데이터를 반복사용한다는데 문제가 있다. 모델을 개선해나가는 과정에서 검증 데이터가 계속해서 사용된다면 결국 검증 데이터 역시 또 다른 훈련데이터에 지나지 않게 되기 때문이다. 그러므로 검증 데이터를 사용해 계산한 성능(예를들어 Accuracy 등)은 실제 모델의 성능과는 다르게 나올 가능성이 있다. 따라서 테스트 데이터(test data)라는 하나의 단계를 더 두게 된다.

테스트 데이터는 모델링이 별 문제 없이 진행되었는지를 검토하는 목적으로도 사용된다. 만약 검증 데이터로부터 예상한 성능과 테스트 데이터로부터 계산한 성능의 차이가 너무 크다면 무언가 모델링 과정에 잘못이 있음을 짐작할 수 있다.

테스트 데이터는 최초 분리 후 모델링 과정에서 이용되지 않다가 최종적으로 모델을 확정지었을 때 그 모델의 성능을 평가하는 목적으로 단 한차례만 이용된다. 테스트 데이터를 사용한 모델링 단계는 다음과 같다.

1. 데이터의 일부를 테스트 데이터로 떼어놓는다.
2. K겹 교차 검증을 수행하여 최종 모델을 선택한다.
3. 테스트 데이터에 최종 모델을 적용해 성능을 평가하고, 그 결과를 최종 모델과 함께 제출 한다.

## cvTools

cvTools[46]는 교차검증을 위한 패키지로서 cvFolds() 함수를 사용해 데이터의 분할(fold)을 만들 수 있다. 다음은 iris 데이터에 대해 10겹 교차 검증을 3회 반복 수행하기 위해 cvFolds()를 사용한 예이다. cvFolds() 실행전에 호출한 set.seed()는 난수를 생성하는 초기값(seed)를 지정하기 위해 사용하였다. cvFolds()는 난수를 사용하여 데이터를 분리하므로 매 호출시마다 서로 다른 folds를 결과로 내놓는다. 하지만 seed를 지정해주면 매번 같은 folds를 결과로 내놓게되어 교차 검증을 수회 반복하더라도 같은 folds를 사용해 안정적으로 모델을 개선할 수 있다.

```
> set.seed(719)
> (cv <- cvFolds(NROW(iris), K=10, R=3))
```

Repeated 10-fold CV with 3 replications:

Fold	1	2	3
1	92	4	86
2	52	3	144
3	17	75	5
4	13	98	61
5	61	30	16
6	9	129	148
7	8	121	49
8	31	89	37
9	136	140	82
10	37	102	141
1	90	51	78
2	119	80	132
3	116	70	97
4	11	29	93
5	39	72	45
6	94	125	114
7	68	84	25

```

8      75 123 69
9      131 73 87
10     100 132 63
...
6      35 137 139
7      73 133 50
8      133 64 7
9      111 74 134
10     66 27 138

```

위 결과에서 각 행은 매 Fold를 의미하고 각 열은 매 반복을 의미한다. 따라서 1행 1열의 92는 첫번째 반복에서 첫번째 Fold의 Validation Data에 iris의 92번째 데이터를 사용하라는 의미이다. 마찬가지로 Fold가 1인 또다른 행을 찾아보면 90, 51, 78이 있다. 이들역시 각각 첫 번째, 두번째, 세번째 반복에서 68, 8, 102를 첫번째 fold의 Validation Data로 각각 사용하라는 뜻이다.

위 표의 Fold에 해당하는 부분은 cv\$which에, 실제 선택할 행을 저장한 부분은 cv\$subset에 다음과 같이 저장되어있다.

```

> head(cv$which, 20)
[1] 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
> head(cv$subset)
[,1] [,2] [,3]
[1,] 92   4   86
[2,] 52   3   144
[3,] 17   75  5
[4,] 13   98  61
[5,] 61   30  16
[6,] 9   129 148

```

따라서 첫번째 반복의 첫번째 fold에서 Validation Data로 사용해야할 행의 번호는 다음과 같이 구할 수 있다. 아래 코드에서 which() 함수는 주어진 조건을 만족하는 행의 번호를 얻기 위해 사용하였다.

```

> validation_idx <- cv$subset[which(cv$which == 1), 1]
> validation_idx
[1] 92 90 14 105 85 95 128 56 121 112 144 5 65 146 4

```

따라서 첫번째 반복의 첫번째 fold에서 Training, Validation Data는 다음과 같이 구한다.

```
> train <- iris[-validation_idx, ]
> validation <- iris[validation_idx, ]
```

이를 사용해 K겹 교차검증을 반복하는 전체 코드의 모습을 그려보면 다음과 같다.

```
> library(foreach)
> set.seed(719)
> R = 3
> K = 10
> cv <- cvFolds(NROW(iris), K=K, R=R)
>
> foreach(r=1:R) %do% {
+   foreach(k=1:K, .combine=c) %do% {
+     validation_idx <- cv$subsets[which(cv$which == k), r]
+     train <- iris[-validation_idx, ]
+     validation <- iris[validation_idx, ]
+     # preprocessing
+
+     # training
+
+     # prediction
+
+     # estimating performance
+     # used runif for demonstration purpose
+     return(runif(1))
+   }
+ }
```

[[1]]

```
[1] 0.3841235 0.9322875 0.4758566 0.9419178 0.2495917 0.7490405 0
    .6129799 0.6606287 0.4115543 0.6133585
```

[[2]]

```
[1] 0.34082047 0.11616376 0.08345950 0.62234106 0.32939378 0.97048141
    0.09819358 0.02692820 0.70704409 0.78077527
```

[[3]]

```
[1] 0.86445690 0.81376815 0.16194006 0.28633551 0.13754093 0.47817326
    0.18470658 0.51545379 0.58434991 0.01900469
```

위 코드에서 주목할만한 점은 for 대신 [foreach](#) (페이지 152)를 사용한 점이다. [foreach\(\)](#)는 값을 반환할 수 있어 모델을 평가한 결과를 한번에 모으는데 유용하다. 또, [foreach\(\)](#) 사용시 내부의 [foreach\(\)](#)에서는 .combine에 'c'를 지정하여 결과가 리스트가 아닌 벡터로 되게하였다. 이렇게하면 결과가 리스트의 리스트가 아니라 벡터의 리스트가되어 조작이 용이할 것이다.

### **caret::createDataPartition()**

`cvTools`를 사용한 교차검증은 데이터의 속성에 대한 고려없이 무작위로 데이터를 나눴다. 그러나 좋은 모델 성능 평가가 되려면 예측하고자하는 분류(Y), 그리고 예측에 사용하는 설명 변수(X)에 대한 고려가 필요하다. 예를들어 검증데이터의 Species에 `setosa`는 너무 많고 `versicolor`, `virginica`는 너무 적다면 그 평가가 공정하지 않을 것이기 때문이다.

`caret`의 `createDataPartition()`, `createResample()`, `createFolds()`, `createMultiFolds()`, `createTimeSlices()`는 Y값을 고려한 훈련 데이터(training data)와 검증 데이터(validation data)의 분리를 지원하며, 이를 함수를 사용해 분리한 데이터는 Y값의 비율이 원본 데이터와 같게 유지된다. 다음은 `createDataPartition()`을 사용해 `iris` 데이터의 80%를 훈련 데이터, 나머지 20%를 검증 데이터로 분리한 예이다.

```
> library(caret)
> (parts <- createDataPartition(iris$Species, p=0.8))
$Resample1
 [1] 1 2 4 6 7 8 9 11 12 14 15 16
[13] 17 18 20 21 22 23 24 25 26 27 29 30
[25] 31 32 34 35 37 38 39 40 41 42 43 46
[37] 47 48 49 50 51 53 54 55 56 57 58 59
[49] 61 62 63 64 65 66 67 68 69 71 72 73
[61] 75 76 77 78 81 82 84 86 87 88 89 90
[73] 91 92 93 95 97 98 99 100 101 102 104 105
[85] 106 107 108 109 110 111 112 113 114 117 118 120
[97] 121 122 123 124 126 128 130 131 132 135 136 137
[109] 138 139 140 141 142 143 144 145 146 147 149 150

> table(iris(parts$Resample1, "Species"))
  setosa versicolor virginica
        40          40          40
```

위 결과에서 `createDataPartition()`은 Species를 고려하여 데이터를 분리하고, 각 Species마다 40개씩을 훈련 데이터로 추출했다. `parts$Resample1`에 포함되지 않은 행들은 검증 데이터로 사용하면 되며, 다음에 보였듯이 검증 데이터에서는 Species마다 각 10개씩 데이터가 할당된다.

```
> table(iris[-parts$Resample1, "Species"])
  setosa versicolor virginica
    10        10        10
```

## 4 로지스틱 회귀모형(Logistic Regression)

이제 여러가지 모델링 알고리즘에 대해 살펴보자. 로지스틱 회귀모형은 주어진 데이터  $X$ 의 분류가 1일 확률을  $p$  라 할 때 다음과 같은 선형 모형을 가정한다.

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X \quad (10.1)$$

로지스틱 회귀 모형을 처음 접하는 독자를 위해 이 모델에 대한 단순한 설명이 Speech and Language Processing[47]에서 언급되어 있어 이를 간단히 소개한다.  $\beta_0 + \beta_1 X$ 는  $(-\infty, \infty)$ 까지의 값을 가질 수 있다. 그러나 우리가 예측하고자 하는 값은 0 또는 1의 값을 갖는다. 따라서 예측값의 범위가  $(-\infty, \infty)$ 가 되도록 해줄 필요가 있다. 이를 위한 첫번째 단계는  $p$ 를 예측하는 것이 아니라  $p$ 와  $1 - p$ 의 비(오즈. odds)를 예측하도록 하는 것이다.

$$\frac{p}{1-p} = \beta_0 + \beta_1 X$$

이 식에서 좌변의 값은  $(0, \infty)$ 까지의 값만 가질 수 있다. 좌변이  $(-\infty, \infty)$ 까지의 값을 갖게하기 위해  $\log$  함수를 취한다. 그 결과 얻어진  $\log(\frac{p}{1-p})$ 를 로짓 함수(logit function)라고 하고 이를 이용한 결과는 식 10.1이 된다. 보다 본격적인 설명은 참고문헌[27]을 보기 바란다.

`iris` 데이터로부터 로지스틱 회귀모형을 작성해 보자. 예측값이 0 또는 1의 두개 분류이어야 하므로 `setosa`, `versicolor`의 두개 분류만 남긴다.

```
> data(iris)
> d <- subset(iris, Species == "setosa" | Species == "versicolor")
> str(d)
'data.frame': 100 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 ...
```

```
$ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 ...
```

위 결과에서 보듯이 subset()을 적용하고나면 데이터는 setosa, versicolor만 남도록 잘 걸려지지만 Species 데이터의 범주 수준은 여전히 3개 레벨이 남게된다. 따라서 Species 열에 새로 범주 수준이 정해지도록 다시 한번 factor() 함수를 거치게 해야한다.

```
> d$Species <- factor(d$Species)
> str(d)

'data.frame': 100 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/ 2 levels "setosa","versicolor": 1 1 ...
```

그 결과 Species가 2개의 레벨로 잘 정리되었다. 모델은 glm() 함수에 family=binomial을 지정해 회귀분석하듯이 만들면 된다.

```
> (m <- glm(Species ~ ., data=d, family=binomial))

Call: glm(formula = Species ~ ., family = binomial, data = d)

Coefficients:
(Intercept) Sepal.Length Sepal.Width Petal.Length Petal.Width
              6.556       -9.879        -7.418       19.054       25.033

Degrees of Freedom: 99 Total (i.e. Null); 95 Residual
Null Deviance: 138.6
Residual Deviance: 1.317e-09 AIC: 10
Warning messages:
1: glm.fit: algorithm did not converge
2: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

모델이 적합된 값은 fitted()를 사용해 알 수 있다.

```
> fitted(m)[c(1:5, 51:55)]
           1          2          3          4
2.220446e-16 2.220446e-16 2.220446e-16 5.151938e-13
           5          51         52         53
```

```
2.220446e-16 1.000000e+00 1.000000e+00 1.000000e+00
      54          55
1.000000e+00 1.000000e+00
```

로지스틱 회귀 모형은 0 또는 1의 값을 예측하는 모델이므로 setosa에 해당하는 1부터 5행 까지는 0, versicolor에 해당하는 51부터 55행은 1로 잘 예측된 것을 알 수 있다.

예측값이 0.5이하인 경우 setosa, 0.5보다 큰 경우 versicolor라고 하고 이를 실제 데이터와 비교해보자. 아래 코드에서 `as.numeric()`은 요인(Factor)을 숫자를 저장한 벡터로 변환한다. R에서 Factor의 수준은 1, 2, 3, ...처럼 1부터 값이 부여되기 시작한다. 따라서 `as.numeric()`으로 Factor를 변환한 뒤 1을 빼주어야 로지스틱 회귀분석의 결과에 맞게 0 또는 1의 값을 갖게된다.

```
> f <- fitted(m)
> as.numeric(d$Species)
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
[34] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
[67] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...
[100] 2
> ifelse(f > .5, 1, 0) == as.numeric(d$Species) - 1
 1   2   3   4   5   6   7   8   9   10  11  12  13  14
TRUE TRUE
 15  16  17  18  19  20  21  22  23  24  25  26  27  28
TRUE TRUE
 29  30  31  32  33  34  35  36  37  38  39  40  41  42
TRUE TRUE
 43  44  45  46  47  48  49  50  51  52  53  54  55  56
TRUE TRUE
 57  58  59  60  61  62  63  64  65  66  67  68  69  70
TRUE TRUE
 71  72  73  74  75  76  77  78  79  80  81  82  83  84
TRUE TRUE
 85  86  87  88  89  90  91  92  93  94  95  96  97  98
TRUE TRUE
 99 100
TRUE TRUE
```

예측된 분류와 실제 분류가 일치한 경우 TRUE, 일치하지 않은 경우 FALSE가 표시된다. 눈으로 확인하기에 불편한 점이 있으므로 위 결과의 TRUE 갯수를 코드를 사용해 세보자.

```
> is_correct <- (ifelse(f > .5, 1, 0) == as.numeric(d$Species) - 1)
> sum(is_correct)
[1] 100
> sum(is_correct) / NROW(is_correct)
[1] 1
```

sum() 함수는 TRUE를 1, FALSE를 0으로 취급하므로 sum()은 전체에서 TRUE의 갯수를 반환하고 NROW()은 데이터의 갯수를 반환한다. 따라서 sum(is\_correct) / NROW(is\_correct)는 분류가 일치한 비율이 된다. 이 경우에는 전체 훈련 데이터에 대해 올바른 예측값이 구해졌다.

새로운 데이터에 대한 예측은 predict() 함수를 사용한다. 이 예에서는 미리 테스트할 데이터를 제외시켜놓지 않았기 때문에 모델을 만들때 사용한 데이터를 재사용한 예를 보였다<sup>4)</sup>. type을 response로 지정하고 예측을 수행하면 0에서 1사이의 결과값을 구해준다.

```
> predict(m, newdata=d[c(1, 10, 55),], type="response")
      1          10          55
2.220446e-16 2.220446e-16 1.000000e+00
```

predict()는 일반 함수(Generic Function)이므로 주어진 인자에 따라 다른 메소드를 호출한다. predict() 메소드가 호출하는 메소드의 목록은 다음과 같이 볼 수 있다.

```
> methods("predict")
[1] predict.Arima*           predict.HoltWinters*
[3] predict.StructTS*        predict.ar*
[5] predict.arima0*          predict.glm
[7] predict.lm               predict.loess*
[9] predict.mlm              predict.nls*
[11] predict.poly             predict.ppr*
[13] predict.prcomp*         predict.princomp*
[15] predict.smooth.spline*  predict.smooth.spline.fit*

Non-visible functions are asterisked
```

이 절에서 만든 모델은 glm() 함수를 호출해 작성하였다. 따라서 predict()에 모델을 넘기면 내부적으로 predict.glm()이 사용되고, 이때 지정할 수 있는 인자들은 help(predict.glm) 또는 ?predict.glm 명령으로 알아볼 수 있다.

<sup>4)</sup>물론 테스트, 검증 데이터를 미리 떼어놓는것이 올바른 방법이다.

로지스틱 회귀모형에는 이외에도 선형 회귀(페이지 264)에서 살펴본 다양한 함수가 적용 가능하니 확인해보기 바란다.

## 5 다향 로지스틱 회귀분석(Multinomial Logistic Regression)

예측 하고자하는 분류가 0, 1의 두개 경우가 아니라 여러개가 될 수 있는 경우 Multinomial Logistic Regression을 사용한다.

이 방법의 기본아이디어는 로지스틱 회귀분석을 확장하는 것이다. 위키피디아의 [Multinomial Logistic Regression](#)에 설명된 독립 바이너리 회귀의 집합에 따라 모델을 설명하면 다음과 같다.

분류  $K$ 를 기준으로 하여 각 분류의 확률을  $\beta_i X_i$ 로 놓는다.

$$\begin{aligned} \ln \frac{P(Y = 1)}{P(Y = K)} &= \beta_1 * X \\ \ln \frac{P(Y = 2)}{P(Y = K)} &= \beta_2 * X \\ \ln \frac{P(Y = 3)}{P(Y = K)} &= \beta_3 * X \\ &\dots \\ \ln \frac{P(Y = K - 1)}{P(Y = K)} &= \beta_{K-1} * X \end{aligned}$$

양변을 e의 지수로 하고 정리하면 다음과 같다.

$$\begin{aligned} P(Y = 1) &= P(Y = K) e^{\beta_1 X} \\ P(Y = 2) &= P(Y = K) e^{\beta_2 X} \\ P(Y = 3) &= P(Y = K) e^{\beta_3 X} \\ &\dots \\ P(Y = K - 1) &= P(Y = K) e^{\beta_{K-1} X} \end{aligned}$$

확률의 합은 1이므로  $P(Y = K)$  를 다음과 같이 정한다.

$$P(Y = K) = \frac{1}{1 + \sum_{k=1}^{K-1} e^{\beta_k X}}$$

최종적으로  $P(Y = i)$ 는 다음과 같다( $i < K$ 인 경우).

$$P(Y_i) = \frac{e^{\beta_i X}}{1 + \sum_{k=1}^{K-1} e^{\beta_k X}} \quad (10.2)$$

R에서는 multinom()을 사용해 모델을 작성할 수 있다. iris의 Species에 대해 모델을 작성해본다.

```
> library(nnet)

> (m <- multinom(Species ~ ., data=iris))
# weights: 18 (10 variable)
initial value 164.791843
iter 10 value 16.177348
iter 20 value 7.111438
iter 30 value 6.182999
iter 40 value 5.984028
iter 50 value 5.961278
iter 60 value 5.954900
iter 70 value 5.951851
iter 80 value 5.950343
iter 90 value 5.949904
iter 100 value 5.949867
final value 5.949867
stopped after 100 iterations
Call:
multinom(formula = Species ~ ., data = iris)

Coefficients:
              (Intercept) Sepal.Length Sepal.Width Petal.Length
versicolor     18.69037    -5.458424   -8.707401    14.24477
virginica     -23.83628    -7.923634   -15.370769    23.65978

                  Petal.Width
versicolor      -3.097684
virginica       15.135301

Residual Deviance: 11.89973
```

AIC: 31.89973

작성한 모델이 주어진 훈련데이터를 어떻게 분류하고 있는지는 fitted()를 사용해 구할 수 있다.

```
> head(fitted(m))
      setosa    versicolor    virginica
1 1.0000000 1.526406e-09 2.716417e-36
2 0.9999996 3.536476e-07 2.883729e-32
3 1.0000000 4.443506e-08 6.103424e-34
4 0.9999968 3.163905e-06 7.117010e-31
5 1.0000000 1.102983e-09 1.289946e-36
6 1.0000000 3.521573e-10 1.344907e-35
```

fitted()의 결과는 각 행의 데이터가 각 분류에 속할 확률을 뜻한다. 어떤 분류로 예측되었는지를 알아내기 위해 매 행마다 가장 큰 값이 속하는 열을 뽑을 수도 있겠지만, 더 간단하게 predict()를 사용해도 된다. 특히 predict()에는 newdata에 새로운 데이터를 지정할 수 있으므로 새로운 관찰값에 대한 예측을 수행하려면 predict()를 사용해야 한다.

setosa, versicolor, virginica에서 한행씩 뽑아 predict()를 적용해보자. 분류를 얻을 때는 type="class"를 지정해야 하지만 type의 기본값이 "class" 이므로 생략해도 된다.

```
> predict(m, newdata=iris[c(1, 51, 101), ], type="class")
[1] setosa      versicolor  virginica
Levels: setosa versicolor virginica
```

각 분류에 속할 확률을 예측하고자 한다면 predict() 사용 시 type="probs"를 지정한다.

```
> predict(m, newdata=iris, type="probs")
      setosa    versicolor    virginica
1 1.000000e+00 1.526406e-09 2.716417e-36
2 9.99996e-01 3.536476e-07 2.883729e-32
3 1.000000e+00 4.443506e-08 6.103424e-34
4 9.999968e-01 3.163905e-06 7.117010e-31
5 1.000000e+00 1.102983e-09 1.289946e-36
...
```

모델의 정확도는 예측된 Species와 실제 Species를 비교하여 알 수 있다. 예측값을 predicted에 저장하고 이 중 iris\$Species와 같은 경우의 비율을 세서 정확도를 계산해보자.

```
> predicted <- predict(m, newdata=iris)
> sum(predicted == iris$Species) / NROW(predicted)
[1] 0.9866667
```

iris처럼 대상으로 하는 분류의 갯수가 2개 이상인 경우에는 [분할표\(Contingency Table\)](#) (페이지 231)를 사용해 세부적인 예측 정확도를 분석할 수 있다.

```
> xtabs(~ predicted + iris$Species)
            iris$Species
predicted    setosa versicolor virginica
setosa          50         0         0
versicolor       0        49         1
virginica        0         1        49
```

표를 통해 총 2개의 예측이 잘못되었고, versicolor를 virginica로 예측한 경우가 한건, virginica를 versicolor로 예측한 경우가 한건 있었음을 알 수 있다. 그러나 여기서 구한 정확도는 훈련 데이터에 대해서 직접 계산한 것이므로, 새로운 데이터에 대한 예측 성능으로 활용할 수는 없음을 기억하기 바란다. 평가 데이터를 분리하여 모델을 평가하는 방법에 대해서는 [모델 평가 방법](#) (페이지 323)을 참고하기 바란다.

## 6 나무 모형(Tree Models)

[의사 결정 나무](#)는 Gini 불순도(Impurity) 또는 Information Gain을 사용하여 노드를 재귀적으로 분할해 나가는 방법이다. 이와 같은 방식들을 일컬어 나무 모형이라하는데, 나무 모형은 if - else 의 조건문과 같은 형식이어서 이해하기 쉽고, 속도가 빠르며, 여러가지 feature의 상호작용을 잘 표현해주고, 다양한 데이터 유형에 사용할 수 있는 장점이 있다. 나무 모델 중 Random Forest의 경우 꽤 괜찮은 성능을 보여주어 머신 러닝 대회가 열리는 [Kaggle](#)에서도 가장 기본이 되는 알고리즘으로 자주 제시된다.

### 6.1 rpart

의사 결정 나무를 만드는 패키지중 이 책에서는 rpart를 사용하도록 한다. rpart는 잘 알려진 CART(Classification and Regression Trees)의 아이디어를 구현한 패키지이다.

다음은 iris 데이터에 대해 rpart()를 사용해 의사 결정 나무를 작성한 예이다.

```
> (m <- rpart(Species ~., data=iris))
n= 150
```

```

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 150 100 setosa (0.33 0.33 0.33)
2) Petal.Length < 2.45 50 0 setosa (1.00 0.00 0.00) *
3) Petal.Length >= 2.45 100 50 versicolor (0.00 0.50 0.50)
6) Petal.Width < 1.75 54 5 versicolor (0.00 0.90 0.09) *
7) Petal.Width >= 1.75 46 1 virginica (0.00 0.02 0.97) *

```

결과화면에서 첫줄의  $n=150$ 은 150개의 데이터가 있었음을 의미한다. 하단에는 트리가 표현되어있는데, 들여쓰기는 가지가 갈라지는 모양을 뜻한다. '\*'은 잎사귀 노드(leaf node)를 의미한다. 트리의 최상단은 root노드로 위 결과에는 '1)'로 표시되어있다. 팔호안은 iris의 Species별 비율을 의미한다.

root노드 하단의 '2)'는 root 노드 바로 밑의 좌측 가지를 뜻한다. 이 가지로 가는 기준은 Petal.Length < 2.45 이다. 이 기준을 만족하는 경우는 setosa 였고 이 때 setosa 50개가 모두 분류되었다.

좀 더 아래쪽의 '6)'은 잎사귀 노드로서 Petal.Width < 1.75인 경우 3번 노드에서 좌측으로 갈라지는 가지이다. 이 때 54개 데이터가 versicolor로 분류되었다.

모델을 좀 더 쉽게보기위해 plot()을 사용해 트리를 그려보자. 아래 코드의 다양한 인자들은 트리가 좀 더 잘 보이게 조절하기 위해 사용한 것이다. compress는 나무를 좀 더 조밀하게 그린 것이고, margin은 여백, cex는 글자의 크기를 뜻한다.

```

> plot(m, compress=TRUE, margin=.2)
> text(m, cex=1.5)

```

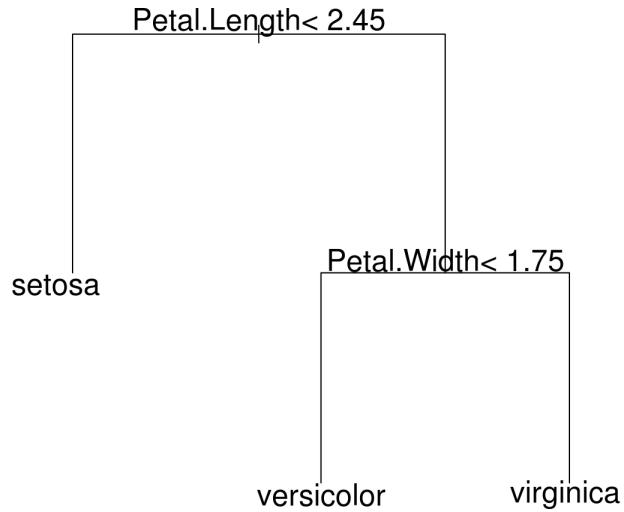


그림 10.13: iris 데이터에 대한 rpart() 수행 결과

그림을 통해 만들어진 트리의 의미를 볼 수 있지만 무언가 부족한 느낌이다. 이를 해결해주는 패키지가 rpart.plot이며, prp() 함수를 통해 다양한 시각화를 제공한다. 아래 코드는 prp()를 사용해 rpart를 시각화한 예이다. type이나 extra의 의미에 대해서는 help(prp)를 참고하기 바란다.

```

> library(rpart.plot)
> prp(m, type=4, extra=2)

```

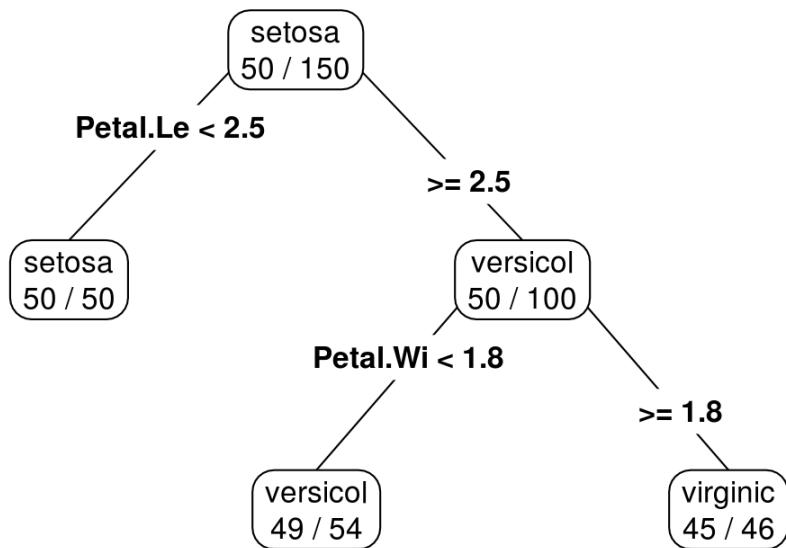


그림 10.14: prp()를 사용한 rpart 시각화

위 그림을 살펴보면 Petal.Length < 2.5인 경우에 붓꽃의 종별이 setosa로 예측되며, 이에 해당하는 50개 데이터중 50개 모두가 실제로 setosa임을 알 수 있다. 마찬가지로 Petal.Length  $\geq 2.5$ , Petal.Width < 1.8 인 경우 versicolor로 예측되는데 이 조건에 해당하는 54개 데이터중 49개가 실제로 versicolor였다.

rpart()를 사용한 예측 역시 predict()를 통해 쉽게 구할 수 있다.

```

> head(predict(m, newdata=iris, type="class"))
 1   2   3   4   5   6 
setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica

```

그외에도 rpart에는 가지치기(pruning)를 위한 prune.rpart() 함수, 트리의 크기를 제한하기 위해 사용하는 CP(Complexity Parameter)와 NA값 처리를 위한 Surrogate Split 를 지정하는 rpart.control() 함수 등 다양한 성능 튜닝을 위한 함수들이 있다. 보다 자세한 내용은 참고문헌[48]를 참고하기 바란다.

## 6.2 party::ctree

ctree(Conditional Inference Tree)[49]는 rpart의 의사 결정 나무가 가진 1) 과적합 문제, 2) 통계적 유의성을 보지 않는 문제를 해결하기 위한 방식이다. 따라서 rpart가 과적합 등의 이유로 인해 성능이 잘 나오지 않는 경우 ctree를 사용해 효과를 볼 수 있다. 또 ctree는 rpart보다 훨씬 더 이해하기 쉬운 트리 그림을 제공하는 장점이 있다.

party 라이브러리를 불러온 후 ctree()를 사용해 모델을 만든다.

```
> library(party)
> (m <- ctree(Species ~., data=iris))

Conditional inference tree with 4 terminal nodes

Response: Species
Inputs: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width
Number of observations: 150

1) Petal.Length <= 1.9; criterion = 1, statistic = 140.264
   2)* weights = 50
1) Petal.Length > 1.9
   3) Petal.Width <= 1.7; criterion = 1, statistic = 67.894
      4) Petal.Length <= 4.8; criterion = 0.999, statistic = 13.865
         5)* weights = 46
      4) Petal.Length > 4.8
         6)* weights = 8
   3) Petal.Width > 1.7
      7)* weights = 46
```

만들어진 모형은 plot()을 사용해 보기 좋은 결과물을 얻을 수 있다.

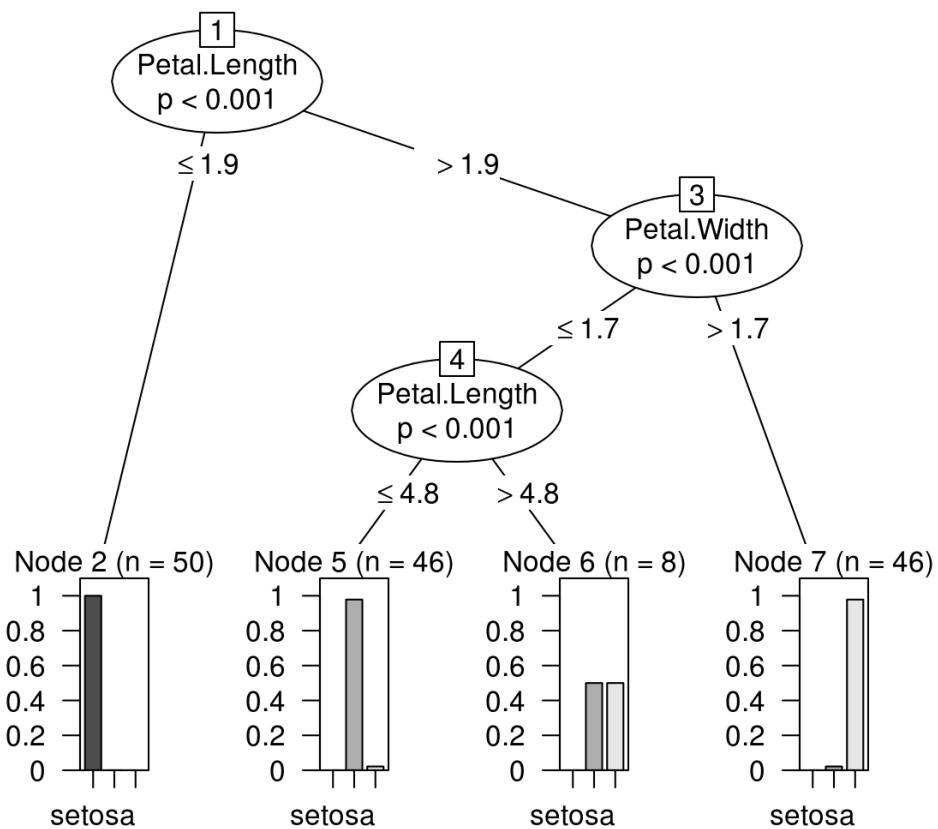


그림 10.15: iris 데이터에 대한 ctree() 수행 결과

ctree()의 결과 그림은 잎사귀 노드의 최종 분류 결과가 무엇이었는지, 또 만약 잘못 분류된 경우가 있다면 그 정도가 어느 정도인지를 알려주어 모델을 개선하는데 도움을 준다. 예를 들어 위 그림에서는 Node 6 ( $\text{Petal.Length} \geq 1.9$ ,  $\text{Petal.Width} \leq 1.7$ ,  $\text{Petal.Length} > 4.8$ 인 경우)에 총 8개의 결과가 있는데 이 때 두개 Species가 거의 동일한 숫자로 나타난 것을 알 수 있다. 따라서 이 경우를 개선하기 위한 feature나 모델을 개발한다면 성능을 더 향상시킬 수 있다.

### 6.3 Random Forest

Random Forest는 양상블(Ensemble) 모델 중 하나이다. 데이터의 일부를 추출하여 의사 결정 나무를 만드는 작업을 반복하고, 이렇게 만들어진 다수의 의사 결정 나무들의 투표(voting)로 최종 결과를 출력한다. 또 각 가지를 나누는 변수를 선택할 때 전체 변수를 매번 모두 고려하는 대신 변수의 일부를 임의로 선택하는 특징을 갖는다.

Random Forest는 randomForest::randomForest()를 사용해 모델을 만든다.

```
> library(randomForest)
> m <- randomForest(Species ~., data=iris)
```

모델을 출력하면 모델 훈련에 사용되지 않았던 데이터를 사용한 에러 추정치(OOB estimate of error rate[50])를 볼 수 있다.

```
> m
Call:
randomForest(x = iris[, 1:4], y = iris[, 5])
                         Type of random forest: classification
                         Number of trees: 500
No. of variables tried at each split: 2

OOB estimate of error rate: 4%
Confusion matrix:

            setosa versicolor virginica class.error
setosa      50          0          0        0.00
versicolor    0         47          3        0.06
virginica     0          3         47        0.06
```

모델을 사용한 예측은 predict.randomForest를 Generic Function인 predict()를 사용해 수행한다.

```
> head(predict(m, newdata=iris))
   1     2     3     4     5     6
setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica
```

## X와 Y의 직접 지정

randomForest()를 비롯한 몇몇 모델링 함수들에는 설명변수(또는 feature)에 해당하는 X와 예측 대상이 되는 분류(Y)를 인자로 직접 지정할 수도 있다. formula를 사용한 표현이 보기에는 편리해보이지만 (X, Y) 형태로 변수를 지정하는 경우에 비해 더 많은 메모리를 필요로하고 속도가 더 느리다는 것이 알려져 있다. 따라서 훈련 데이터의 용량이 크고 모델링 함수가 지원하는 경우라면 (X, Y)를 직접 지정하는 형식을 고려하기 바란다. 다음에 iris의 독립변수와 종속변수를 randomForest()에 직접 지정한 예를 보였다.

```
> m <- randomForest(iris[,1:4], iris[,5])
```

## 변수 중요도 평가

randomForest()는 설명 변수의 중요도를 평가하는데 사용할 수 있다. 이 방법은 각 변수들이 Gini 또는 정확도(Accuracy)에 얼마만큼 기여하는지를 통해 변수의 중요도를 판별한다. randomForest로 구한 변수의 중요도 결과는 연이어 다른 모델, 예를 들어 선형 회귀에 사용할 변수를 선택하는데 사용할 수 있다. 즉 이 방법은 변수 선택(Feature Selection) 방법 중 [Filter Method](#)로 활용할 수 있다.

변수의 중요도를 알아보려면 randomForest() 호출시 importance=TRUE를 지정한다. 그 뒤 importance(), varImpPlot()를 사용해 결과를 출력한다. 다음 결과를 살펴보면 Accuracy 측면에서는 Petal.Length, Petal.Width, Sepal.Length, Sepal.Width 순으로 변수가 중요함을 알 수 있다. Gini 측면에서는 Petal.Width, Petal.Length, Sepal.Length, Sepal.Width 순으로 중요했다.

```
> m <- randomForest(Species ~., data=iris, importance=TRUE)
> importance(m)

      setosa versicolor virginica MeanDecreaseAccuracy
Sepal.Length 7.152771  6.6933858  9.337384          11.682065
Sepal.Width   4.355985 -0.7440227  4.855949          4.095249
Petal.Length 22.223321 33.9951048 30.376644         35.694348
Petal.Width   22.549154 33.7757691 33.673222         35.634290

      MeanDecreaseGini
Sepal.Length        10.693481
Sepal.Width         2.324579
Petal.Length        43.008320
Petal.Width         43.259537
```

varImpPlot()을 사용해 이를 그림으로 좀 더 쉽게 알아 볼 수 있다.

```
> varImpPlot(m, main="varImpPlot of iris")
```

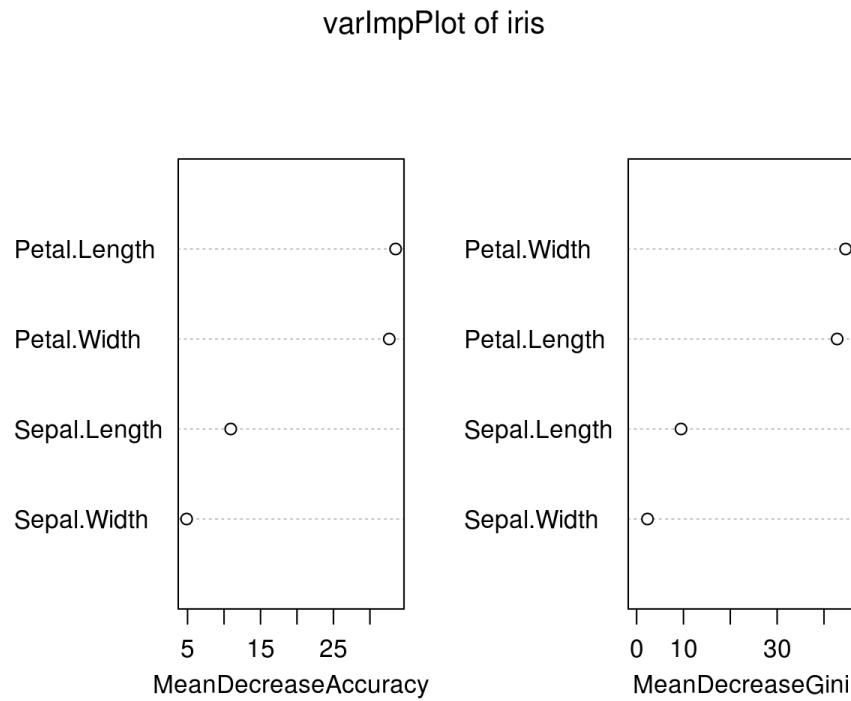


그림 10.16: randomForest를 사용한 변수 중요도 평가

RRF(Regularized Random Forest)[51, 52]는 Random Forest를 개선한 Feature Selection 방법을 제공한다. [RRF 패키지](#)를 참고하기 바란다.

## 파라미터 튜닝

randomForest()에는 트리 개수(ntree), 각 노드에서 가지를 칠 때 고려할 변수의 갯수(mtry) 등의 파라미터가 있다. 이 숫자를 적절히 지정하는 방법 중 하나는 [교차 검증\(Cross Validation\)](#)(페이지 330)을 사용하는 것이다.

ntree를 10, 100, 200의 3개 값, mtry를 3, 4의 2개 값으로 바꿔보면서 모델의 성능을 평가해보자. 이를 위해 가능한 조합의 목록은 expand.grid()를 사용해 다음과 같이 만들 수 있다.

```
> (grid <- expand.grid(ntree=c(10, 100, 200), mtry=c(3, 4)))
   ntree mtry
1     10    3
2    100    3
3    200    3
4     10    4
5    100    4
6    200    4
```

이러한 파라미터 조합을 10개로 분할한 데이터에 적용하여 모델의 성능을 평가하는 일을 R회 반복하면 교차 검증을 사용한 파라미터를 찾을 수 있게 된다. 이를 위한 코드는 다음과 같다.

```

library(cvTools)
library(foreach)
library(randomForest)
set.seed(719)

K = 10
R = 3
cv <- cvFolds(NROW(iris), K=K, R=R)

grid <- expand.grid(ntree=c(10, 100, 200), mtry=c(3, 4))

result <- foreach(g=1:NROW(grid), .combine=rbind) %do% {
  foreach(r=1:R, .combine=rbind) %do% {
    foreach(k=1:K, .combine=rbind) %do% {
      validation_idx <- cv$subsets[which(cv$which == k), r]
      train <- iris[-validation_idx, ]
      validation <- iris[validation_idx, ]
      # training
      m <- randomForest(Species ~.,
                          data=train,
                          ntree=grid[g, "ntree"],
                          mtry=grid[g, "mtry"])

      # prediction
      predicted <- predict(m, newdata=validation)
      # estimating performance
      precision <- sum(predicted == validation$Species) / NROW(
        predicted)
      return(data.frame(g=g, precision=precision))
    }
  }
}

```

위 코드에서 `foreach`문의 `.combine`에 `rbind` 가 사용되었음을 유의해서 보기 바란다. `foreach()`에 `.combine`을 지정하지 않으면 반환값이 리스트형태이고, 여기서는 데이터를 데이터

프레임으로 모으기 위해 rbind를 사용하였다. 코드의 수행결과 result에는 다음과 같은 값이 저장된다.

```
> result
  g precision
1 1 1.0000000
2 1 1.0000000
3 1 0.9333333
4 1 0.9333333
5 1 1.0000000
...
176 6 0.9333333
177 6 0.9333333
178 6 0.9333333
179 6 0.9333333
180 6 1.0000000
```

이를 g값마다 묶어 평균을 구하기 위해 [ddply\(\)](#) (페이지 124)를 사용한다.

```
> library(plyr)
> ddply(result, .(g), summarize, mean_precision=mean(precision))
      g mean_precision
1 1     0.9444444
2 2     0.9533333
3 3     0.9533333
4 4     0.9555556
5 5     0.9533333
6 6     0.9555556
> grid[c(4, 6), ]
      ntree mtry
4     10     4
6    200     4
```

[ddply\(\)](#) 수행결과 가장 높은 성능을 보인 조합은 ntree=10, mtry=4와 ntree=200, mtry=4의 두가지 경우였다.

## 7 신경망(Neural Networks)

신경망은 다층의 뉴런을 구성해 모델을 구성하는 형태로, 보통 입력/은닉/출력의 3개 층으로 구성된다. 분류 알고리즘으로 사용할 때 은닉층(hidden layer)과 출력층(output layer)은 sigmoid 함수를 사용한다.

nnet 패키지에서 신경망의 파라미터는 엔트로피(entropy) 또는 SSE(Sum of Squared Error)를 고려해 최적화되며, 출력결과는 선택적으로 softmax를 사용해 확률과 같은 형태로 변환할 수 있다. 과적합(overfitting)을 막기위한 방법으로 weight decay를 제공하며 decay에 원하는 값을 지정할 수 있다.

nnet은 노드의 수가 어느 정도를 넘어서면 weight의 수가 많다면 에러를 발생시킨다. 이 경우 MaxNWts 파라미터를 지정해 weight 갯수 제한을 높일 수 있으므로 참고하기 바란다.

### 7.1 Formula를 사용한 모델 생성

신경망을 잘 사용하려면 데이터 정규화(Feature Scaling) (페이지 311)를 적용한 뒤 신경망 모델을 만들어야하지만, 아래 예에서는 편의를 위해 iris 데이터를 직접 사용하였다. nnet() 함수에 지정한 size 파라미터는 은닉층 노드의 수를 의미한다.

```
> library(nnet)
> m <- nnet(Species ~., data=iris, size=3)
# weights:  27
initial  value 209.776713
iter    10  value 68.839766
iter    20  value 68.325309
iter    30  value 48.408527
iter    40  value 11.403077
iter    50  value 7.095460
iter    60  value 6.106387
iter    70  value 5.988069
iter    80  value 5.965337
iter    90  value 5.959421
iter   100  value 5.958498
final  value 5.958498
stopped after 100 iterations

> predict(m, newdata=iris)
      setosa    versicolor    virginica
```

```

1  9.999874e-01 1.263331e-05 3.959453e-28
2  9.999588e-01 4.118988e-05 7.729874e-27
3  9.999766e-01 2.335103e-05 1.855355e-27
4  9.999421e-01 5.786730e-05 1.817189e-26
5  9.999888e-01 1.119066e-05 2.918959e-28
...
146 8.955349e-19 1.350138e-05 9.999865e-01
147 1.310257e-15 1.087811e-03 9.989122e-01
148 1.559853e-15 1.208206e-03 9.987918e-01
149 4.507296e-19 8.928967e-06 9.999911e-01
150 2.071929e-13 2.276016e-02 9.772398e-01

```

만약 모델로부터 예측된 분류를 바로 얻고자한다면 다음과 같이 type에 class를 지정한다<sup>5)</sup>.

```
> predict(m, newdata=iris, type="class")
```

여기서 보인 방법은 Formula 를 지정한 형태였다. Formula를 지정할 경우 nnet()은 다음과 같이 동작하도록 되어있다.

- linout의 기본값이 FALSE이므로 출력층에서 linear 함수가 아니라 sigmoid 함수가 사용된다.
- 만약 예측대상이 되는 분류(즉, Y의 레벨)의 수가 2개라면 entropy를 사용해 파라미터가 추정된다.
- 분류의 수가 3개이상이라면 SSE(Sum of Squared Errors)로 파라미터가 추정되며, softmax가 적용된다.

## 7.2 X와 Y의 직접 지정

좀 더 빠른 속도를 원하거나 또는 각종 파라미터가 nnet()에서 자동으로 지정되는 것을 원치 않는 경우 X, Y를 직접 지정하는 형태로 nnet()을 호출할 수도 있다. 이를 위해 가장 먼저 할 일은 Y(iris의 경우 Species)를 가변수(dummy variables)로 변환하는 것이다.

가변수 변환은 nnet의 class.ind()를 사용한다.

```
> class.ind(iris$Species)
      setosa  versicolor  virginica
[1,]       1           0           0
```

<sup>5)</sup>help(predict.nnet) 참조

[2 ,]	1	0	0
[3 ,]	1	0	0
[4 ,]	1	0	0
[5 ,]	1	0	0
...			
[51 ,]	0	1	0
[52 ,]	0	1	0
[53 ,]	0	1	0
[54 ,]	0	1	0
[55 ,]	0	1	0
...			
[101 ,]	0	0	1
[102 ,]	0	0	1
[103 ,]	0	0	1
[104 ,]	0	0	1
[105 ,]	0	0	1
...			

X에는 iris에서 Species를 제외한 나머지 변수들을 지정하면 된다.

```
> m2 <- nnet(iris[, 1:4], class.ind(iris$Species), size=3,
+             softmax=TRUE)
# weights:  27
initial  value 179.822518
iter    10  value 69.476514
iter    20  value 65.407837
iter    30  value 13.239150
iter    40  value 6.182978
iter    50  value 5.988953
iter    60  value 5.964934
iter    70  value 5.960658
iter    80  value 5.960084
iter    90  value 5.958269
iter   100  value 5.956502
final   value 5.956502
stopped after 100 iterations
```

```
> predict(m2, newdata=iris[, 1:4], type="class")
[1] "setosa"      "setosa"      "setosa"      "setosa"      "setosa"      ...

```

## 8 SVM(Support Vector Machine)

SVM 모델을 위한 패키지에는 e1071, kernlab 등이 있다. e1071은 svm()을, kernlab은 ksvm() 함수를 각각 제공한다.

이들 중 ksvm()은 formula를 사용한 형식과 X, Y를 각각 지정하는 형식을 모두 지원한다. iris에 Formula를 사용한 모델을 만들어보자.

```
> (m <- ksvm(Species ~ ., data=iris))
Using automatic sigma estimation (sigest) for RBF or laplace kernel
Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 1

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.982467042345241

Number of Support Vectors : 63

Objective Function Value : -5.0512 -5.6974 -19.859
Training error : 0.013333
```

ksvm() 함수는 자동으로 데이터의 정규화를 수행하기에(scaled 파라미터의 기본값이 TRUE임), 함수 호출 전 데이터를 정규화할 필요는 없다.

만들어진 모델로부터의 예측은 predict()를 사용한다.

```
> head(predict(m, newdata=iris))
[1] setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica
```

앞서 ksvm() 함수는 Radial Basis Kernel Function을 사용한 모델을 자동으로 선택했다. 만약 kernel함수를 바꾸고 싶다면 kernel 파라미터에 원하는 kernel 함수를 지정하면 된다. 지원하는 kernel의 목록은 help(kernlab::dots)를 참고하기 바란다.

```
> ksvm(Species ~., data=iris, kernel="vanilladot")
Setting default kernel parameters
Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 29

Objective Function Value : -0.9818 -0.322 -17.0644
Training error : 0.033333
```

커널에 사용하는 파라미터는 kpar에 리스트 형태로 값을 지정한다.

```
> (m <- ksvm(Species ~., data=iris, kernel="polydot",
+   kpar=list(degree=3)))
Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 1

Polynomial kernel function.

Hyperparameters : degree = 3 scale = 1 offset = 1

Number of Support Vectors : 22

Objective Function Value : -0.0252 -0.0225 -6.3396
Training error : 0.013333
```

SVM을 잘 사용하기 위해서는 각종 파라미터 값을 잘 찾아야 한다. 이를 위한 첫번째 방법은 [교차 검증\(Cross Validation\)](#) (페이지 330)을 서로 다른 파라미터 값에 대해 수행하는 것이다. 두번째 방법은 SVM 패키지가 제공하는 파라미터 튜닝을 사용하는 것이다.

e1071에서는 tune() 함수를 사용해 모델을 튜닝할 수 있다. help(tune)을 입력해 자세한 사용법을 알아보기 바란다. 다음에 example(tune)의 일부를 보였다.

```
> library(e1071)
```

```
> tune.svm(Species ~., data=iris, gamma=2^(-1:1), cost=2^(2:4))
Parameter tuning of `svm':
- sampling method: 10-fold cross validation
- best parameters:
  gamma  cost
  0.5     4
- best performance: 0.05333333
```

tune.svm의 반환값은 객체이며, 객체의 속성은 attributes()로 살펴볼 수 있다. 다음에 최적 파라미터 값을 빼내는 코드의 예를 적었다.

```
> attributes(result)
$names
[1] "best.parameters" "best.performance" "method"           "nparcomb"
[5] "train.ind"        "sampling"          "performances"    "best.model"

$class
[1] "tune"

> result$best.parameters
  gamma  cost
1 0.5     4
> result$best.parameters["gamma"]
  gamma
1 0.5
> result$best.parameters["cost"]
  cost
1 4
```

## 9 클래스 불균형(Class Imbalance)

클래스내 각 분류에 해당하는 데이터의 비율이 50%:50%이 아닌 경우 훈련데이터내 비율이 많은 분류쪽으로 예측결과가 쏠리는 현상이 발생할 수 있다. 예를들어 다음의 유방암 데이터를 보자.

```
> library(mlbench)
> data(BreastCancer)
```

```
> table(BreastCancer$Class)
benign malignant
 458       241
```

유방암 데이터의 양성(benign)의 수는 458, 악성(malignant)의 수는 241이었다. 이 데이터로 모델링을 수행하면 해당 모델은 주어진 데이터에 대해 benign을 결과로 예측할 확률이 malignant을 결과로 예측할 확률에 비해 높아지게된다. 왜냐하면 주어진 데이터를 무조건 benign으로만 예측해도 65.5%<sup>6)</sup>의 정확도를 확보하게 되기 때문이다. 이로인해 malignant가 갖고 있는 특성이 무엇인지를 모델이 잘 배우지 않게될 가능성이 높아진다.

이를 해결하는 방법에는 up sampling, down sampling, SMOTE(Synthetic Minority Over-sampling Technique)[53]가 있다.

### upSample, downSample

caret 패키지의 upSample(), downSample()은 각각 분류중 적은 쪽의 레이블의 데이터를 더 많이 추출하거나 또는 많은 쪽의 레이블의 데이터를 적게 추출하는 기능을 한다. 기본적인 사용법은 이들 두 함수가 유사하므로 여기서는 upSample()의 예만 살펴보자. upSample()은 인자로 설명변수(X)와 예측 대상이 되는 분류(Y)를 인자로 받아 변환된 데이터를 결과로 출력한다.

다음은 BreastCancer 데이터를 upSample() 수행한 예이다.

```
> x <- upSample(subset(BreastCancer, select=-Class),
+                  BreastCancer$Class)
> table(BreastCancer$Class)
benign malignant
 458       241
> table(x$Class)
benign malignant
 458       458
```

교차테이블을 살펴보면 up sampling 이전에는 458:241의 비율이었던 데이터가 458:458로 균형잡히게 변한 것을 볼 수 있다. upSample은 이를 위해 단순히 적은 쪽에 해당하는 분류의 데이터를 중복하여 추출하는 방식을 사용한다. 이는 다음을 통해 확인할 수 있다.

```
> NROW(x)
[1] 916
> NROW(unique(x))
```

<sup>6)</sup>458/(458+241)=0.6552217

```
[1] 691
```

x 내 행의 수는 916이지만 이를 중 상당수는 중복된 것이다.

다음은 upSample을 훈련데이터에 적용한 경우와 그렇지 않은 경우 각각 모델의 성능을 비교한 코드이다.

```
> library(party)
> data <- subset(BreastCancer, select=-Id)
> parts <- createDataPartition(data$Class, p=.8)
> data.train <- data[parts$Resample1,]
> data.validation <- data[-parts$Resample1, ]
> m <- rpart(Class ~., data=data.train)
> confusionMatrix(data.validation$Class,
+                   predict(m, newdata=data.validation, type="class"))
Confusion Matrix and Statistics

Reference
Prediction   benign malignant
benign          86        5
malignant        3       45

Accuracy : 0.9424
95% CI : (0.8897, 0.9748)
No Information Rate : 0.6403
P-Value [Acc > NIR] : <2e-16

Kappa : 0.874
McNemar's Test P-Value : 0.7237

Sensitivity : 0.9663
Specificity : 0.9000
Pos Pred Value : 0.9451
Neg Pred Value : 0.9375
Prevalence : 0.6403
Detection Rate : 0.6187
Detection Prevalence : 0.6547
```

```

`Positive' Class : benign

> data.up.train <- upSample(subset(data.train, select=-Class),
+                               data.train$Class)
> m <- rpart(Class ~., data=data.up.train)
> confusionMatrix(data.validation$Class,
+                   predict(m, newdata=data.validation, type="class"))
Confusion Matrix and Statistics

                    Reference
Prediction    benign malignant
benign          83        8
malignant       2        46

Accuracy : 0.9281
95% CI : (0.8717, 0.965)
No Information Rate : 0.6115
P-Value [Acc > NIR] : <2e-16

Kappa : 0.8455
McNemar's Test P-Value : 0.1138

Sensitivity : 0.9765
Specificity : 0.8519
Pos Pred Value : 0.9121
Neg Pred Value : 0.9583
Prevalence : 0.6115
Detection Rate : 0.5971
Detection Prevalence : 0.6547

`Positive' Class : benign

```

교차표에서 볼 수 있듯이 원본 데이터를 사용한 경우 malignant로 예측한 수는 45개였으나 upSample 이후에는 46개로 늘었다. 이로인해 Sensitivity도 0.9663에서 0.9765로 올랐으며, 대신 Specificity는 0.9000에서 0.8519로 내렸다.

코드에서 `subset()` (페이지 105)문은 특정 컬럼을 선택하거나 또는 제외하는 목적으로 사용되었으며 Id는 각 데이터의 식별자로 모델링에는 적합하지 않기에 가장 처음 단계에서 제거되었다.

`createDataPartition()`은 교차 검증(Cross Validation) (페이지 330)에서 살펴본 함수로 훈련 데이터와 검증 데이터의 분리를 위해 사용하였으며 이 예에서는 80%의 데이터를 훈련데이터, 나머지 20%를 검증 데이터로 사용하기 위해  $p=0.8$ 을 지정하였다.

사용된 모델은 나무 모형이며 `rpart` (페이지 343)를 사용하였다.

## SMOTE

SMOTE는 비율이 적은 분류의 데이터를 생성하는 방법이다. SMOTE의 정확한 내용은 참고 자료[53]를 보기바라며, 여기서는 기본 개념만 살펴보자. SMOTE는 먼저 분류 갯수가 적은 쪽의 데이터 샘플을 취한 뒤 이 샘플의  $k$  최근접 이웃(k nearest neighbor)를 찾는다. 그리고 현재 샘플과 이를  $k$ 개 이웃간의 차(difference)를 구한뒤 이 차이에  $0 \sim 1$  사이의 임의의 값을 곱하여 원래 샘플에 더한다. 이렇게 만든 새로운 샘플을 훈련 데이터에 추가한다. 결과적으로 SMOTE는 기존의 샘플을 주변의 이웃을 고려해 약간씩 이동시킨 점들을 추가하는 방식으로 동작한다.

DMwR 패키지의 `SMOTE()` 함수는 이 알고리즘의 구현으로 비율이 적은 분류의 데이터를 생성하는 기능과 비율이 큰쪽 데이터를 under sampling하는 기능을 제공한다. SMOTE는 numeric 데이터를 기본으로하여 작성된 알고리즘이기에 `BreastCancer`를 예로 사용하기 곤란하다. 따라서 여기서는 `example(SMOTE)`의 예를 기반으로 한 설명을 하도록 하겠다.

다음은 예시를 위해 사용할 데이터를 만드는 코드이다. `iris`에서 `setosa` 종은 `rare`로, 그외의 `versicolor`, `virginica`는 `common`으로 바꿨다. 그 결과 `common`에는 100개의 데이터가, `rare`에는 50개의 데이터가 생성되었다.

```
> data(iris)
> data <- iris[, c(1, 2, 5)]
> data$Species <- factor(ifelse(data$Species == "setosa", "rare", "common"))
> table(data$Species)
common     rare
      100      50
```

이 데이터에 SMOTE를 사용하여 `common`과 `rare`의 갯수를 대략 맞춘 결과는 다음과 같다.

```
> newData <- SMOTE(Species ~ ., data, perc.over = 600, perc.under=100)
> table(newData$Species)
```

common	rare
300	350

perc.over는 갯수가 적은 분류로부터 얼마나 많은 데이터를 생성해낼지(즉 over sampling)를 조정하는 변수이며, perc.under는 갯수가 많은 분류의 데이터에서의 under sampling을 조정하는 변수이다. 이들 파라미터를 지정하는 정확한 방법은 다소 복잡한 면이 있기에 help(SMOTE)를 참고하기 바란다.

## 10 문서 분류(Document Classification)

이 절에서는 텍스트 마이닝(Text Mining) 패키지인 tm[54, 55]을 사용한 문서 분류 방법에 대해 설명한다.

### 10.1 코퍼스와 문서

tm에서 문서의 집합은 Corpus로, 각 문서는 TextDocument로 표현된다. Reuter 뉴스 중 ‘crude’ 토픽에 대한 문서 20개를 포함하고 있는 crude 데이터를 살펴보자.

```
> library(tm)
> data(crude)
> summary(crude)
A corpus with 20 text documents

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID
```

문서의 본문은 inspect() 함수로 볼 수 있다. inspect(crude)를 호출하면 모든 문서에 대한 내용을 보여주며 특정 문서를 지정해서 보려면 crude[start:end] 형태로 범위를 지정하거나 crude[index] 형태로 색인을 지정한다. 다음은 crude의 첫번째 문서를 살펴보는 예이다.

```
> inspect(crude[1])
A corpus with 1 text document

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
```

```

create_date creator

Available variables in the data frame are:

MetaID

$`reut-00001.xml` 

Diamond Shamrock Corp said that
effective today it had cut its contract prices for crude oil by
1.50 dlr s a barrel.

The reduction brings its posted price for West Texas
Intermediate to 16.00 dlr s a barrel, the copany said.

"The price reduction today was made in the light of falling
oil product prices and a weak crude oil market," a company
spokeswoman said.

Diamond is the latest in a line of U.S. oil companies that
have cut its contract, or posted, prices over the last two days
citing weak oil markets.

Reuter

```

## 10.2 문서 변환

문서 분류에 앞서 글에서 문장 부호를 제거하거나, 문자를 모두 소문자로 바꾸거나, [Stemming](#)을 적용하는 등 문서를 변환할 필요가 있을 수 있다. 이 때 사용하는 함수가 tm\_map()이다. tm\_map()은 인자로 Corpus와 문서 변환 함수(FUN)을 받는다. tm 은 문서 변환을 위한 다양한 함수를 제공하는데, 이 함수들의 목록은 getTransformations()로 볼 수 있다.

다음은 crude 문서들의 글자들을 모두 소문자로 바꾸고 문장 부호를 제거하는 예이다.

```

> inspect(tm_map(tm_map(crude, tolower), removePunctuation)[1])
A corpus with 1 text document

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator

Available variables in the data frame are:
  MetaID

$`reut-00001.xml` 

diamond shamrock corp said that

```

```

effective today it had cut its contract prices for crude oil by
150 dlr s a barrel

the reduction brings its posted price for west texas
intermediate to 1600 dlr s a barrel the company said

the price reduction today was made in the light of falling
oil product prices and a weak crude oil market a company
spokeswoman said

diamond is the latest in a line of us oil companies that
have cut its contract or posted prices over the last two days
citing weak oil markets

reuter

```

### 10.3 문서의 행렬 표현

**TermDocumentMatrix(), DocumentTermMatrix()**

term과 document(문서)의 행렬로 Corpus를 표현하려면 TermDocumentMatrix() 또는 DocumentTermMatrix()를 사용한다. 이들 중 TermDocumentMatrix()는 주어진 문서들로부터 term을 행, document(문서)를 열로하는 행렬을 만든다. 반대로 DocumentTermMatrix()는 문서를 행, term을 열로 표현한다. 다음은 crude를 term × document의 행렬로 표현한 예이다.

```

> (x <- TermDocumentMatrix(crude))
A term-document matrix (1266 terms, 20 documents)

Non-/sparse entries: 2255/23065
Sparsity : 91%
Maximal term length: 17
Weighting : term frequency (tf)

```

위 결과를 보면 term × document 행렬의 차원이  $1266 \times 20$ 이고 행렬의 값은 tf (term frequency), 즉 각 term의 출현빈도임을 알 수 있다.

행렬의 내부는 inspect()를 사용해 볼 수 있다.

```

> inspect(x[1:10, 1:10])
A term-document matrix (10 terms, 10 documents)

Non-/sparse entries: 4/96
Sparsity : 96%

```

```
Maximal term length: 7
Weighting : term frequency (tf)
```

Terms	Docs									
	127	144	191	194	211	236	237	242	246	248
...	0	0	0	0	0	0	0	0	0	1
100,000	0	0	0	0	0	0	0	0	0	0
10.8	0	0	0	0	0	0	0	0	0	0
1.1	0	0	0	0	0	0	0	0	0	0
1.11	0	0	0	0	0	0	0	0	0	0
1.15	0	0	0	0	0	0	0	0	0	0
1.2	0	0	0	0	0	1	0	0	0	0
12.	0	0	0	1	0	0	0	0	0	0
12.217	0	0	0	0	0	0	0	0	1	0
12.32	0	0	0	0	0	0	0	0	0	0

만약 다른 Weighting을 사용하고 싶다면 TermDocumentMatrix()의 control 인자에 weighting을 지정한다. 다음은 TF×IDF 를 사용한 예이다.

> x <- TermDocumentMatrix(crude, control=list(weighting=weightTfIdf))					
> inspect(x[1:10, 1:5])					
A term-document matrix (10 terms, 5 documents)					
Non-/sparse entries: 1/49					
Sparsity : 98%					
Maximal term length: 7					
Weighting :					
term frequency - inverse document frequency (normalized) (tf-idf)					
Docs					
Terms 127 144 191 194 211					
...	0	0	0	0.000000	0
100,000	0	0	0	0.000000	0
10.8	0	0	0	0.000000	0
1.1	0	0	0	0.000000	0
1.11	0	0	0	0.000000	0
1.15	0	0	0	0.000000	0

1.2	0	0	0	0.000000	0
12.	0	0	0	0.074516	0
12.217	0	0	0	0.000000	0
12.32	0	0	0	0.000000	0

help(TermDocumentMatrix)에 보다 다양한 사례가 예로 설명되어 있으므로 참고하기 바란다.

### findFreqTerms(), findAssocs()

findFreqTerms()는 term × document 행렬로부터 자주 출현하는 term을 추출해준다. 다음은 전체 20개 문서로 구성된 crude에서 10회 이상 출현한 단어를 찾은 예이다.

```
> findFreqTerms(TermDocumentMatrix(crude), lowfreq=10)
[1] "about"      "and"        "are"        "bpd"        "but"
[6] "crude"      "dlrs"       "for"        "from"       "government"
[11] "has"        "its"        "kuwait"     "last"       "market"
[16] "mln"        "new"        "not"        "official"   "oil"
[21] "one"        "opec"       "pct"        "price"      "prices"
[26] "reuter"     "said"       "said."      "saudi"      "sheikh"
[31] "that"        "the"        "they"       "u.s."       "was"
[36] "were"       "will"       "with"      "would"
```

참고로 전체 단어와 문서의 목록은 rownames(), colnames()로 볼 수 있다.

```
> x <- TermDocumentMatrix(crude)
> head(rownames(x))
[1] "..."      "100,000"  "10.8"     "1.1"      "1.11"     "1.15"
> head(colnames(x))
[1] "127"      "144"      "191"      "194"      "211"      "236"
```

findAssocs()는 term × document 행렬과 특정 term이 주어졌을 때 그 term과 상관계수가 높은 term들을 찾아준다. 상관관계가 높은 term들은 주어진 term과 연관이 있다고 볼 수 있다.

다음은 “oil”과 상관계수가 0.7 이상인 term들을 찾은 예이다. opec, winter, market, prices 등의 단어를 보면 직관적으로도 oil과 함께 출현하는 빈도가 높을 만한 단어들임을 쉽게 예상할 수 있다.

```
> findAssocs(TermDocumentMatrix(crude), "oil", 0.7)
15.8      opec      clearly      late      trying      who      winter
```

0.87	0.87	0.80	0.80	0.80	0.80	0.80
analysts	said	meeting	above	emergency	market	fixed
0.79	0.78	0.77	0.76	0.75	0.75	0.73
that	prices	agreement	buyers			
0.73	0.72	0.71	0.70			

## 10.4 문서 분류

이제 본격적으로 tm 을 사용한 문서 분류의 예를 살펴보자. 이 절에서는 Reuter 기사 중 crude 토픽과 acq 토픽에 대한 문서들을 훈련 데이터로 하여, 주어진 문서가 crude와 acq중 어느 토픽에 속하는지 분류해주는 모델을 만들어본다.

### 데이터 준비

crude 토픽의 문서와 acq 토픽의 문서는 각각 같은 이름의 Corpus에 저장되어있다. 따라서 이를 DocumentTermMatrix()를 사용하여 document × term 로 만든뒤 합친다. 그 다음 이를 행렬(matrix), 데이터 프레임으로 변환해나가면서 LABEL 컬럼에 crude, acq 레이블을 붙이면 모델링에 적합한 형태가 된다.

다음은 이 과정을 보인 코드이다.

```
> data(crude)
> data(acq)
> to_dtm <- function(corpus, label) {
+   x <- tm_map(corpus, tolower)
+   x <- tm_map(corpus, removePunctuation)
+   return(DocumentTermMatrix(x))
+ }
> crude_acq <- c(to_dtm(crude), to_dtm(acq))
> crude_acq_df <- cbind(
+   as.data.frame(as.matrix(crude_acq)),
+   LABEL=c(rep("crude", 20), rep("acq", 50)))
```

위 코드에서 DocumentTermMatrix()를 만들기 전에 tolower를 적용하였으므로 “LABEL”이라는 컬럼이름은 문서내 존재하던 term들과 중복될 염려가 없다.

적절한 데이터 타입이 잘 부여되었는지를 확인하기 위해 str()로 crude\_acq\_df를 살펴보자.

```
> str(crude_acq_df)
'data.frame': 70 obs. of 2373 variables:
```

```
$ 150           : num  1 0 0 0 0 0 0 0 0 0 ...
$ 1600          : num  1 0 0 0 0 0 0 0 0 0 ...
$ and           : num  1 9 0 1 2 7 11 3 9 6 ...
$ barrel         : num  2 0 1 1 0 3 0 0 0 2 ...
$ brings          : num  1 0 1 1 0 0 0 0 0 0 ...
$ citing          : num  1 0 0 0 0 0 0 0 0 0 ...
> str(crude_acq_df$LABEL)
Factor w/ 2 levels "acq","crude": 2 2 2 2 2 2 2 2 2 2 ...
> str(crude_acq_df$LABEL[21:30])
Factor w/ 2 levels "acq","crude": 1 1 1 1 1 1 1 1 1 1
```

이 결과를 보면 term에는 숫자가 저장되어있고 각 문서의 분류를 나타내는 LABEL은 Factor로 잘 저장되어있는 것을 알 수 있다.

## 모델링

분류 알고리즘을 적용하기에 앞서 훈련 데이터와 검증 데이터를 나눠보자. 여기서는 설명의 편의를 위해 단순히 80%의 데이터를 훈련 데이터, 20%의 데이터를 검증 데이터로 나누었으며 [교차 검증\(Cross Validation\)](#) (페이지 330)은 적용하지 않는다.

```
> library(caret)
> train_idx <- createDataPartition(
+   crude_acq_df$LABEL, p=0.8)$Resample1
> crude_acq.train <- crude_acq_df[train_idx, ]
> crude_acq.validation <- crude_acq_df[-train_idx, ]
```

[rpart](#) (페이지 343)를 사용하여 모델을 만든다.

```
> library(rpart)
> m <- rpart(LABEL ~ ., data=crude_acq.train)
```

결과는 [평가 메트릭\(metric\)](#) (페이지 323)에서 설명한 [confusionMatrix\(\)](#)를 사용해 분석해 보자.

```
> confusionMatrix(
+   predict(m, newdata=crude_acq.validation, type="class"),
+   crude_acq.validation$LABEL)
Confusion Matrix and Statistics
```

```

    Reference

Prediction acq crude
  acq      9      0
  crude     1      4

Accuracy : 0.9286
95% CI : (0.6613, 0.9982)
No Information Rate : 0.7143
P-Value [Acc > NIR] : 0.0594

Kappa : 0.8372
McNemar's Test P-Value : 1.0000

Sensitivity : 0.9000
Specificity : 1.0000
Pos Pred Value : 1.0000
Neg Pred Value : 0.8000
Prevalence : 0.7143
Detection Rate : 0.6429
Detection Prevalence : 0.6429

'Positive' Class : acq

```

분석결과 정확도가 92.86%로 들인 노력에 비해 상당히 괜찮은 모델을 구할 수 있었다. 다만 데이터의 크기가 작아 정확도의 95% CI(Confidence Interval)는 66.13%에서 99.82%로 구해졌다.

## 10.5 파일로부터 Corpus 생성

지금까지의 보인 문서 분류의 예에서는 tm 패키지내에 이미 잘 정리된 crude 데이터 등을 사용하였다. 그러나 문서 분류를 실제로 수행하게 되면 직접 문서 파일을 읽어들여야한다.

문서를 읽는 소스는 디렉토리, 데이터 프레임 등을 지정할 수 있다. tm이 지원하는 소스의 목록은 getSources()로 볼 수 있다.

```

> getSources()
[1] "DataframeSource" "DirSource"          "GmaneSource"
[4] "ReutersSource"   "URISource"         "VectorSource"

```

이 절에서는 DataframeSource를 사용하는 방법을 살펴보자. 다음과 같이 문서 파일이 저장되어 있다고 가정하자.

```
$ cat > docs.csv
Title,Body,Label
"Hello World","This is the body of hello world document","notspam"
"Linear algebra","This book is about linear algebra","nonspam"
"Online Casino","$50,000 is waiting for you","spam"
"Poker","Poker game players","spam"
```

이 데이터는 Title, Body, Label로 구성되어 있으며 각 문서가 스팸인지의 여부가 가장 마지막 열인 Label에 저장되어 있다. 이 데이터는 CSV 포맷으로 되어 있어 read.csv()를 사용해 데이터 프레임으로 읽어들일 수 있다.

```
> docs <- read.csv("docs.csv", stringsAsFactors=FALSE)
> docs
      Title                      Body      Label
1 Hello World This is the body of hello world document notspam
2 Linear algebra          This book is about linear algebra nonspam
3 Online Casino           $50,000 is waiting for you      spam
4        Poker            Poker game players      spam
> str(docs)
'data.frame': 4 obs. of 3 variables:
 $ Title: chr "Hello World" "Linear algebra" "Online Casino" "Poker"
 $ Body : chr "This is the body of hello world document" "This book
   is about linear algebra" "$50,000 is waiting for you" "Poker game
   players"
 $ Label: chr "notspam" "nonspam" "spam" "spam"
```

read.csv()에서 stringsAsFactors=FALSE를 지정했음을 기억하기 바란다. 만약 이 파라미터가 지정되지 않으면 문자열이 모두 Factor형 변수로 저장되어버린다.

읽어들인 데이터 프레임은 Corpus()에 DataframeSource()를 지정해 Corpus로 변환할 수 있다.

```
> corpus <- Corpus(DataframeSource(docs[,1:2]))
> inspect(corpus)
A corpus with 4 text documents
```

```

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID

$`1`
Hello World
This is the body of hello world document

$`2`
Linear algebra
This book is about linear algebra

$`3`
Online Casino
$50,000 is waiting for you

$`4`
Poker
Poker game players

```

이 예에서는 아직 문서의 분류(Label)은 저장하지 않았다. 문서의 분류를 저장하는 방법은 [메타 데이터](#) (페이지 373)에서 설명한다.

## 10.6 메타 데이터

문서의 집합인 Corpus 또는 각 문서에는 메타 데이터<sup>7)</sup>를 붙일 수 있다. 메타데이터는 다음에 보인 meta() 함수를 사용해 접근한다.

tm에서 메타 데이터를 붙이는 수준에는 corpus, local, indexed의 세가지 유형이 있다. 이중 corpus 는 문서 집합 전체에 대해 붙이는 메타데이터이며, local은 개별 문서에 직접 저장되는 메타 데이터이다. indexed는 local과 유사하게 개별 문서와 연관된다. 그러나 local은 각 문서와 함께 저장되어 각 문서를 꺼내어 볼때에만 그 메타데이터를 볼 수 있는 반면, indexed는 그 메타데이터가 독립적으로 존재하되 각 문서와 연관해서 볼 수 있게 되어있다. 다음 예를 통해 이들을 구분해보자.

---

<sup>7)</sup>데이터를 설명하는 데이터로 이해하면 된다. 예를들어 문서의 생성 일시, 저자, ID, 언어, 출처, 분류 레이블 등이 있다.

다음 코드는 crude 문서집합 전체의 메타 데이터를 출력하는 예이다. Corpus 생성일자와 생성한 사람의 이름이 기록되어있음을 볼 수 있다.

```
> data(crude)
> meta(crude, type="corpus")
$create_date
[1] "2010-06-17 07:32:26 GMT"

$creator
LOGNAME
"feinerer"
```

다음은 local 메타 데이터를 살펴보자.

```
> meta(crude, type="local")
... 생략 ...
$People
character(0)

$Orgs
character(0)

$Exchanges
[1] "nymex"

Available meta data pairs are:
Author      :
DateTimeStamp: 1987-03-02 14:49:06
Description  :
Heading      : ARGENTINE OIL PRODUCTION DOWN IN JANUARY 1987
ID          : 708
Language    : en
Origin      : Reuters-21578 XML

User-defined local meta data pairs are:
$TOPICS
[1] "YES"

$LEWISSPLIT
```

```
[1] "TRAIN"

$CGISPLIT
[1] "TRAINING-SET"

$OLDID
[1] "12891"

$Topics
[1] "crude"    "nat-gas"

$Places
[1] "argentina"

$People
character(0)

$Orgs
character(0)

$Exchanges
character(0)
```

local 메타데이터는 각 문서를 하나씩 꺼내어 살펴보는 것이므로 위와같이 코드를 입력하면 전체 문서에 대한 데이터가 한꺼번에 나와버려 읽기가 쉽지 않다. 대신 각 문서를 직접 지정해보는 것이 편리하다. 다음은 첫번째 문서의 메타데이터만 출력한 예이다.

```
> meta(crude[1], type="local")
Available meta data pairs are:
  Author      :
  DateTimeStamp: 1987-02-26 17:00:56
  Description  :
  Heading     : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
  ID          : 127
  Language    : en
  Origin      : Reuters-21578 XML
User-defined local meta data pairs are:
```

```
$TOPICS
[1] "YES"

$LEWISSPLIT
[1] "TRAIN"

$CGISPLIT
[1] "TRAINING-SET"

$OLDID
[1] "5670"

$Topics
[1] "crude"

$Places
[1] "usa"

$People
character(0)

$Orgs
character(0)

$Exchanges
character(0)
```

local 메타데이터안에 저자, 날짜 등의 정보뿐만 아니라 TOPICS 등의 정보도 저장되어 있음을 볼 수 있다.

indexed는 type의 기본값이므로 단순히 meta(corpus) 명령으로 볼 수 있다. 그러나 다음 결과에서 보듯이 crude에는 별다른 indexed 메타 데이터가 없다.

```
> meta(crude)
  MetaID
 1      0
 2      0
 3      0
```

4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0

위 예를 통해 알 수 있는 점은 local 메타데이터는 각 문서의 정보와 함께 저장되어있어 각 문서별로 볼 수 있고, indexed는 indexed 메타 데이터만 따로 떼어놓고 한번에 볼 수 있다는 차이가 있다는 것이다.

앞서 [파일로부터 Corpus 생성](#) (페이지 371)에서 보인 DataframeSource()를 사용한 파일 읽기 방법의 예에서는 문서의 분류(spam, nonspam)를 저장하지 않았다. 이 정보를 문서의 indexed 메타 데이터로 저장해보자.

```
> meta(corpus, "Label") <- docs[, "Label"]
> meta(corpus)
  MetaID    Label
1      0  notspam
2      0  nonspam
3      0      spam
4      0      spam
```

메타데이터에 분류가 잘 저장되었다. 이제 이 Corpus를 사용해 예측 모형을 만들어 볼 수 있을 것이다.

# 11

## 타이타닉 데이터를 사용한 머신 러닝 연습

이 장에서는 타이타닉호의 생존자 데이터로부터 각 탑승자의 생존 여부를 예측하는 모델을 만드는 사례를 살펴본다. 이를 통해 이 책에서 설명했던 내용을 복습하고 좀 더 복잡한 R 코드를 작성하는 방법을 연습해 볼 것이다.

타이타닉 데이터는 <http://biostat.mc.vanderbilt.edu/wiki/Main/DataSets>에서 다운로드 받을 수 있다. 해당 페이지에서 “Data for Titanic passengers” 섹션을 찾아 titanic3.csv를 다운로드 받기 바란다.

### 1 타이타닉 데이터 형식

타이타닉 데이터는 CSV 파일 형식으로 저장되어 있으며, 이 장에서는 CSV 파일의 일부 컬럼만 사용할 것이다. 표 11.1은 이 장에서 사용할 컬럼에 대한 설명이다.

컬럼명	내용
pclass	1, 2, 3등석여부를 1, 2, 3으로 저장.
survived	생존여부. survived(생존), dead(사망)
name	이름
sex	성별. female(여성) 또는 male(남성)
age	나이
sibsp	함께 탑승한 형제 또는 배우자의 수
parch	함께 탑승한 부모 또는 자녀의 수
ticket	티켓 번호
fare	티켓 요금
cabin	선실번호
embarked	탑승한 곳. C(Cherbourg), Q(Queenstown), S(Southampton).

표 11.1: 타이타닉 데이터

## 2 데이터 불러오기

파일을 R로 읽어들인 후 올바른 포맷으로 지정하는 작업이 필요하기 때문에 데이터를 불러들이는 작업은 생각보다 까다롭다. `read.csv()`를 사용해 파일을 읽어들이고, 불필요한 컬럼들을 삭제한 뒤 데이터를 살펴보자.

```
> titanic = read.csv("titanic3.csv")
> titanic <- titanic[, !names(titanic) %in%
+                         c("home.dest", "boat", "body")]
> str(titanic)
'data.frame': 1309 obs. of 11 variables:
 $ pclass   : int 1 1 1 1 1 1 1 1 1 ...
 $ survived : int 1 1 0 0 0 1 1 0 1 0 ...
 $ name     : Factor w/ 1307 levels "Abbing, Mr. Anthony",...: 22 24 ...
 $ sex      : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 ...
 $ age      : num 29 0.92 2 30 25 48 63 39 53 71 ...
 $ sibsp    : int 0 1 1 1 0 1 0 2 0 ...
 $ parch    : int 0 2 2 2 0 0 0 0 ...
 $ ticket   : Factor w/ 929 levels "110152","110413",...: 188 50 50 ...
 $ fare     : num 211 152 152 152 152 ...
 $ cabin    : Factor w/ 187 levels "", "A10", "A11", ...: 45 81 81 81 ...
 $ embarked: Factor w/ 4 levels "", "C", "Q", "S": 4 4 4 4 4 4 4 4 ...
```

데이터의 일부 또는 전체를 살펴보려면 `head(변수명)` 또는 `View(변수명)`을 사용한다. 특히 RStudio를 사용중이라면 `View()` 함수를 사용할 경우 그림 11.1처럼 상당히 편리한 화면을 제공한다.

1309 observations of 11 variables

	pclass	survived	name	sex
1	1	1	Allen, Miss. Elisabeth Walton	female
2	1	1	Allison, Master. Hudson Trevor	male
3	1	0	Allison, Miss. Helen Loraine	female
4	1	0	Allison, Mr. Hudson Joshua Creighton	male
5	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female
6	1	1	Anderson, Mr. Harry	male
7	1	1	Andrews, Miss. Kornelia Theodosia	female
8	1	0	Andrews, Mr. Thomas Jr	male
9	1	1	Appleton, Mrs. Edward Dale (Charlotte Lamson)	female
10	1	0	Artagaveytia, Mr. Ramon	male
11	1	0	Astor, Col. John Jacob	male
12	1	1	Astor, Mrs. John Jacob (Madeleine Talmadge Force)	female
13	1	1	Aubart, Mme. Leontine Pauline	female
14	1	1	Barber, Miss. Ellen "Nellie"	female
15	1	1	Barkworth, Mr. Algernon Henry Wilson	male
16	1	0	Baumann, Mr. John D	male
17	1	0	Baxter, Mr. Quigg Edmond	male
18	1	1	Baxter, Mrs. James (Helene DeLaudeniere Chaput)	female
19	1	1	Bazzani, Miss. Albina	female

그림 11.1: View(titanic)의 실행화면

## 2.1 데이터 타입 지정

str(titanic)을 수행해보면 pclass는 1등석이 1, 2등석이 2, 3등석이 3인 int(정수)로 표현되어 있다. 그러나 이들은 서로간에 사적 연산 관계가 존재하는 것은 아니므로 숫자로 취급하기보다는 범주형 변수인 Factor로 표현하는 것이 낫다.

survived는 생존여부를 뜻하므로 Factor 타입으로 설정 해야 R의 머신 러닝 함수를 호출했을 때 분류(classification) 알고리즘이 수행된다. 만약 현재처럼 int(정수)로 둘 경우 회귀분석(regression)을 수행해버리게 된다. 따라서 Factor로 변경할 필요가 있다.

마지막으로 name, ticket, cabin 은 수준(level)의 수가 너무 많으므로 Factor로 나타내기보다는 단순 문자열(character)로 취급하도록 하자.

```
> titanic$pclass <- as.factor(titanic$pclass)
> titanic$name <- as.character(titanic$name)
> titanic$ticket <- as.character(titanic$ticket)
> titanic$cabin <- as.character(titanic$cabin)
> titanic$survived <- factor(titanic$survived, levels=c(0, 1),
+                                labels=c("dead", "survived"))
```

```
> str(titanic)
'data.frame': 1309 obs. of 11 variables:
 $ pclass   : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
 $ survived : Factor w/ 2 levels "dead","survived": 2 2 1 1 1 2 2 1 ...
 $ name     : chr "Allen, Miss. Elisabeth Walton" ...
 $ sex      : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 ...
 $ age      : num 29 0.92 2 30 25 48 63 39 53 71 ...
 $ sibsp    : int 0 1 1 1 0 1 0 2 0 ...
 $ parch    : int 0 2 2 2 0 0 0 0 0 ...
 $ ticket   : chr "24160" "113781" "113781" "113781" ...
 $ fare     : num 211 152 152 152 152 ...
 $ cabin    : chr "B5" "C22 C26" "C22 C26" "C22 C26" ...
 $ embarked: Factor w/ 4 levels "", "C", "Q", "S": 4 4 4 4 4 4 4 4 4 2
 ...

```

`str()`을 확인해본 결과 값이 올바른 데이터 타입으로 저장되었음을 볼 수 있다.

그런데 `embarked`에 조금 이상한 값이 보인다. Factor 수준(level)에 “” 값이 있다. 이 값이 어떻게 쓰이고 있는지 확인해보자.

```
> levels(titanic$embarked)
[1] "" "C" "Q" "S"

> table(titanic$embarked)
   C   Q   S
2 270 123 914
```

“” 값이 2개 행에 존재하고 있음을 볼 수 있다. `read.csv()`가 CSV 파일을 그대로 읽어들임을 생각해보면 빈 문자열 “”는 NA를 뜻함을 쉽게 알 수 있다. “”를 NA로 수정해보자.

```
> levels(titanic$embarked)[1] <- NA
> table(titanic$embarked, useNA='always')
   C   Q   S <NA>
270 123 914     2
```

수정 결과 “”가 NA로 잘 치환되었다. `table()` 함수는 NA 값을 제외하고 값을 출력시키기에 `useNA`에 `always`를 지정해 NA에 대한 갯수도 출력하도록 하여 빈도를 확인하였다.

마찬가지로 `cabin` 컬럼에도 빈 문자열이 저장되어있다. 이 값 역시 NA로 바꿔보자.

```
> titanic$cabin <- ifelse(titanic$cabin == "", NA, titanic$cabin)
```

이처럼 Factor의 level을 바꾸는 경우와 달리 문자열은 직접적으로 값을 수정하면 된다. 마지막으로 str(), head(), View() 등의 명령어를 사용해 데이터가 잘 변경되었는지 확인해본다.

```
> str(titanic)

'data.frame': 1309 obs. of 11 variables:
 $ pclass    : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
 $ survived   : Factor w/ 2 levels "dead","survived": 2 2 1 1 1 2 2 ...
 $ name      : chr  "Allen, Miss. Elisabeth Walton" ...
 $ sex       : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 ...
 $ age        : num  29 0.92 2 30 25 48 63 39 53 71 ...
 $ sibsp      : int  0 1 1 1 0 1 0 2 0 ...
 $ parch      : int  0 2 2 2 0 0 0 0 0 ...
 $ ticket     : chr  "24160" "113781" "113781" "113781" ...
 $ fare       : num  211 152 152 152 152 ...
 $ cabin      : chr  "B5" "C22 C26" "C22 C26" "C22 C26" ...
 $ embarked   : Factor w/ 3 levels "C","Q","S": 3 3 3 3 3 3 3 3 3 1 ...
```

## 2.2 테스트 데이터(Test Data)의 분리

데이터를 불러오는 작업이 끝나고나면 곧바로 테스트 데이터를 전체 데이터에서 분리해야 한다<sup>1)</sup>. 만약 테스트 데이터를 데이터의 형태를 살펴보는 [데이터 탐색](#) (페이지 386)에 사용해 버리거나 모델링 단계에서 사용해버리면 테스트 데이터에 대한 정보를 모델러가 미리 알게 되어버릴 가능성이 있다. 그리고 그렇게되면 미지의 새로운 데이터라는 테스트 데이터로서의 의미가 퇴색된다.

데이터 분리는 [교차 검증\(Cross Validation\)](#) (페이지 330)에서 설명한 createDataPartition()을 사용한다. createDataPartition()은 Y 값을 고려한 데이터의 분할을 지원하므로 생존자(survived)와 사망자(dead)의 수가 훈련데이터와 테스트 데이터간에 일정하게 유지된다.

아래 코드에서 set.seed(137)은 난수 생성시 seed를 지정한 것이다. 이 값을 지정하는 이유는 데이터 로드 부터 테스트 데이터 분할까지의 과정을 혹시라도 수회 반복할 일이 있을 때 항상 같은 데이터가 훈련 데이터와 테스트 데이터로 분리되게 하기 위함이다. 만약 seed를 지정하지 않으면 데이터 분리시마다 매번 다른 데이터가 훈련 데이터로 나오게되고, 모델링 과정에 훈련데이터가 계속 바뀌면 모델 성능이 계속 변하는 등 혼란이 있을 수 있기 때문이다.

---

<sup>1)</sup> 이 책에서는 데이터 정제를 먼저 하고나서 테스트 데이터를 분리했지만, 데이터를 분리한다음 정제하는 방법을 택할 수도 있다.

```

> library(caret)
> set.seed(137)
> test_idx <- createDataPartition(titanic$survived, p=0.1)$Resample1
> titanic.test <- titanic[test_idx, ]
> titanic.train <- titanic[-test_idx, ]
> NROW(titanic.test)
[1] 131
> prop.table(table(titanic.test$survived))
    dead   survived
0.6183206 0.3816794
> NROW(titanic.train)
[1] 1178
> prop.table(table(titanic.train$survived))
    dead   survived
0.6179966 0.3820034

```

코드 수행결과를 보면 총 131행의 테스트 데이터와 1178행의 훈련 데이터가 구해졌음을 알 수 있다. 사망자(dead)와 생존자(survived) 비율 역시 약 61% : 38%로 일정하게 유지되고 있다.

데이터 분리가 끝나면 이후 단계에서 사용이 편리하도록 저장시켜 놓는다. 다음 코드는 titanic, titanic.test, titanic.train 데이터를 titanic.RData 라는 파일에 저장한다.

```
> save(titanic, titanic.test, titanic.train, file="titanic.RData")
```

## 2.3 교차 검증 준비

테스트 데이터를 분리한 뒤 남은 데이터로부터 교차 검증을 수행하기로 한다. 10겹 교차 검증 (10-fold Cross Validation)을 수행하기로 하고 caret 패키지의 createFolds()를 사용해 데이터를 분리해보자.

```

> createFolds(titanic.train$survived, k=10)
$Fold01
 [1] 2 34 39 41 49 50 66 67 72 86 89 92 108
[14] 116 143 148 151 172 201 211 214 216 224 245 247 250
[27] 259 268 270 277 289 304 308 321 351 370 375 407 420
[40] 422 434 441 446 447 448 457 460 462 467 490 501 502
[53] 515 522 538 551 554 562 589 592 601 602 618 624 628
[66] 632 654 676 708 710 713 716 717 719 720 744 751 755

```

[79]	759	770	774	782	807	814	827	847	862	864	899	916	922
[92]	937	944	947	952	958	959	960	978	989	993	1003	1007	1010
[105]	1020	1024	1026	1043	1045	1063	1082	1084	1089	1117	1142	1157	1160
[118]	1165												
<b>\$Fold02</b>													
[1]	15	21	35	38	40	57	60	85	101	103	113	128	141
[14]	142	145	156	161	177	182	193	199	209	230	252	263	309
[27]	324	328	359	361	372	391	402	412	414	435	458	472	483
[40]	485	492	498	506	511	521	525	527	539	545	552	561	567
[53]	573	578	585	631	649	656	660	666	669	673	677	682	684
[66]	687	688	690	701	702	703	730	740	742	762	764	767	773
[79]	799	810	839	840	849	876	897	911	914	920	933	936	942
[92]	954	975	981	992	994	1001	1004	1009	1019	1031	1061	1076	1078
[105]	1095	1102	1107	1108	1111	1114	1118	1120	1132	1138	1143	1150	1152
[118]	1176												
...													
<b>\$Fold10</b>													
[1]	5	9	27	33	51	54	65	69	76	77	91	97	100
[14]	105	109	125	131	132	139	146	154	174	218	226	238	253
[27]	265	281	282	288	293	300	307	323	330	332	334	336	344
[40]	363	366	378	379	396	404	405	408	418	423	439	442	463
[53]	468	478	523	530	532	537	563	564	565	576	577	583	597
[66]	663	674	691	694	725	728	733	736	746	775	777	787	792
[79]	795	802	821	831	838	848	855	857	866	880	883	889	896
[92]	905	926	946	961	967	995	1013	1017	1025	1039	1040	1055	1057
[105]	1066	1079	1086	1105	1116	1125	1130	1131	1156	1158	1166	1171	1174
[118]	1177												

코드 수행결과 10개의 Fold가 만들어져 리스트에 Fold01, Fold02, ..., Fold10의 이름으로 저장됨을 알 수 있다. 각 Fold에는 검증 데이터로 사용할 데이터 번호가 저장되어있다. 따라서 데이터 번호를 보고 훈련 데이터와 검증 데이터를 분리하면 된다. 다음은 10겹 교차 검증 데이터를 만드는 함수이다. 앞서 설명한 바 있듯이 `set.seed()`는 해당 함수를 반복해서 호출해도 매번 같은 결과가 나오게 하기 위해 사용되었다.

```
create_ten_fold_cv <- function() {
```

```

set.seed(137)
lapply(createFolds(titanic.train$survived, k=10), function(idx) {
  return(list(train=titanic.train[-idx, ],
              validation=titanic.train[idx, ]))
})
}

```

이 함수는 Fold01, Fold02, ..., Fold10을 가진 리스트를 반환하며 각 Fold에는 train과 validation이라는 이름에 훈련 데이터와 검증 데이터 저장된다. 실제 내용을 살펴보자.

```

> x <- create_ten_fold_cv()
> str(x)
List of 10
$ Fold01:List of 2
..$ train      :'data.frame': 1061 obs. of 11 variables:
.. ..$ pclass    : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 ...
.. ..$ survived: Factor w/ 2 levels "dead","survived": 2 1 1 2 2 ...
.. ..$ name     : chr [1:1061] "Allen, Miss. Elisabeth Walton" ...
.. ..$ sex      : Factor w/ 2 levels "female","male": 1 2 1 2 1 ...
.. ..$ age      : num [1:1061] 29 30 25 48 63 39 53 71 47 18 ...
.. ..$ sibsp    : int [1:1061] 0 1 1 0 1 0 2 0 1 1 ...
.. ..$ parch   : int [1:1061] 0 2 2 0 0 0 0 0 0 0 ...
.. ..$ ticket  : chr [1:1061] "24160" "113781" "113781" "19952" ...
.. ..$ fare    : num [1:1061] 211.3 151.6 151.6 26.6 78 ...
.. ..$ cabin   : chr [1:1061] "B5" "C22 C26" "C22 C26" "E12" ...
.. ..$ embarked: Factor w/ 3 levels "C","Q","S": 3 3 3 3 3 3 ...
..$ validation:'data.frame': 117 obs. of 11 variables:
.. ..$ pclass    : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 ...
.. ..$ survived: Factor w/ 2 levels "dead","survived": 1 1 2 ...
.. ..$ name     : chr [1:117] "Allison, Miss. Helen Loraine" ...
.. ..$ sex      : Factor w/ 2 levels "female","male": 1 2 1 1 ...
.. ..$ age      : num [1:117] 2 24 47 60 41 39 38 38 NA 30 ...
.. ..$ sibsp    : int [1:117] 1 0 1 0 0 1 1 0 0 0 ...
.. ..$ parch   : int [1:117] 2 1 1 0 0 0 0 0 0 0 ...
.. ..$ ticket  : chr [1:117] "113781" "PC 17558" "11751" "11813" ...
.. ..$ fare    : num [1:117] 151.6 247.5 52.6 76.3 134.5 ...
.. ..$ cabin   : chr [1:117] "C22 C26" "B58 B60" "D35" "D15" ...

```

```
... . . . $ embarked: Factor w/ 3 levels "C","Q","S": 3 1 3 1 1 1 ...  
...
```

결과가 길어 위의 코드에는 Fold01의 내용만 보였다. Fold01에서 훈련데이터를 가져오려면 다음과 같이 하면 된다.

```
> head(x$Fold01$train)

  pclass survived                               name
1      1    survived          Allen, Miss. Elisabeth Walton
4      1     dead           Allison, Mr. Hudson Joshua Creighton
5      1     dead Allison, Mrs. Hudson J C (Bessie Waldo Daniels)
6      1    survived          Anderson, Mr. Harry
7      1    survived        Andrews, Miss. Kornelia Theodosia
8      1     dead           Andrews, Mr. Thomas Jr

  sex age sibsp parch ticket      fare cabin embarked
1 female 29     0     0   24160 211.3375      B5      S
4 male  30     1     2  113781 151.5500  C22 C26      S
5 female 25     1     2  113781 151.5500  C22 C26      S
6 male  48     0     0   19952  26.5500      E12      S
7 female 63     1     0   13502  77.9583      D7      S
8 male  39     0     0  112050  0.0000      A36      S
```

또는 `x[[1]]$train` 으로 첫번째 Fold를 가져올 수도 있다.

### 3 데이터 탐색

모델을 작성하기 전 데이터가 어떤 모습을 띠고 있는지 살펴보면 어떻게 모델을 세울 것인지에 대한 많은 힌트를 얻을 수 있다. 또 데이터를 불러들일 때 혹시 오류가 있지는 않았는지의 여부도 이 단계에서 알 수 있게된다. 이 단계에서는 간단하게는 `summary()`를 사용해 데이터의 분포를 살펴볼 수도 있고, 데이터를 그림으로 그려서 표현해 볼 수도 있다.

Hmisc 패키지에는 `summary()`에 `formula`를 지정해 데이터의 요약정보를 얻을 수 있는 기능이 있다. `survived`가 `pclass`, `sex`, `age` 등의 값에 따라 어떻게 달라지는지 요약 자료를 살펴보자.

```
> library(Hmisc)
> summary(survived ~ pclass + sex + age + sibsp + parch + fare +
embarked, data=data)
```

## Descriptive Statistics by survived

	N	dead (N=656)	survived (N=405)
pclass : 1	1061	15% ( 99)	39% (159)
2		20% (130)	24% ( 96)
3		65% (427)	37% (150)
sex : male	1061	85% (555)	32% (131)
age	850	21/28/39	20/27/39
sibsp : 0	1061	72% (473)	62% (253)
1		19% (123)	32% (128)
2		3% ( 20)	4% ( 16)
3		2% ( 10)	1% ( 6)
4		2% ( 16)	0% ( 2)
5		1% ( 6)	0% ( 0)
8		1% ( 8)	0% ( 0)
parch : 0	1061	83% (547)	67% (270)
1		7% ( 48)	21% ( 84)
2		7% ( 47)	11% ( 46)

	3			0% ( 3)		1% ( 4)	
+-----+-----+-----+-----+							
	4			1% ( 4)		0% ( 0)	
+-----+-----+-----+-----+							
	5			0% ( 3)		0% ( 1)	
+-----+-----+-----+-----+							
	6			0% ( 2)		0% ( 0)	
+-----+-----+-----+-----+							
	9			0% ( 2)		0% ( 0)	
+-----+-----+-----+-----+							
fare		1061   7.8542/10.5000/26.0000   10.5000/26.0000/57.0000					
+-----+-----+-----+-----+							
embarked : C   1059   16% (103)   30% (120)							
+-----+-----+-----+-----+							
Q     9% ( 58)   9% ( 37)							
+-----+-----+-----+-----+							
S     75% (495)   61% (246)							
+-----+-----+-----+-----+							

표를 통해 pclass, sex 등 각 변수 값에 따라 생존률의 분포가 어떻게 달라지는지 볼 수 있다. Hmisc의 사용에 대한 자세한 내용은 `help(library="Hmisc")` 또는 `help("summary.formula")`를 통해 살펴보기 바란다.

이번에는 `caret::featurePlot()`을 사용해 데이터를 시각화 해보자. 예측 대상이 되는 생존/사망 여부를 Y, 나머지 독립 변수(또는 feature라고 함)를 X로 해서 `featurePlot()`을 호출하는 것이 목표이다. `featurePlot()` 사용시 NA 값이 있으면 차트가 제대로 그려지지 않으므로 NA부터 제거해야한다. 이 때 데이터 프레임의 각 행에 NA값이 하나도 없는지 여부를 테스트해주는 `complete.cases()`를 사용한다. 그 뒤, `featurePlot()`의 X에는 Factor를 지정할 수 없으므로 숫자(numeric) 컬럼만 선택하여 X에 지정한다.

```
> data.complete <- data[complete.cases(data), ]
> featurePlot(
+   data.complete[, 
+     sapply(names(data.complete), 
+           function(n) { is.numeric(data.complete [, n]) })],
+   data.complete [, c("survived")],
+   "ellipse")
```

sapply() 부분은 다음과 같이 동작한다. names(data.complete)는 data.complete의 컬럼명들(pclass, survived, name, sex, age, ..., embarked 등)을 반환한다. sapply()는 이 각각의 이름을 function()에 n이라는 이름으로 넘긴다. 호출된 함수는 data.complete에서 해당 열이 숫자형 데이터인지를 is.numeric()으로 테스트해 반환한다. 따라서 sapply()의 최종결과는 TRUE 또는 FALSE가 저장된 벡터이다. 이 결과가 data.complete[, sapply()] 형태로 호출되므로 TRUE, FALSE 값에따라 각 열의 선택여부가 결정된다.

그림 11.2에 featurePlot()의 결과를 보였다.

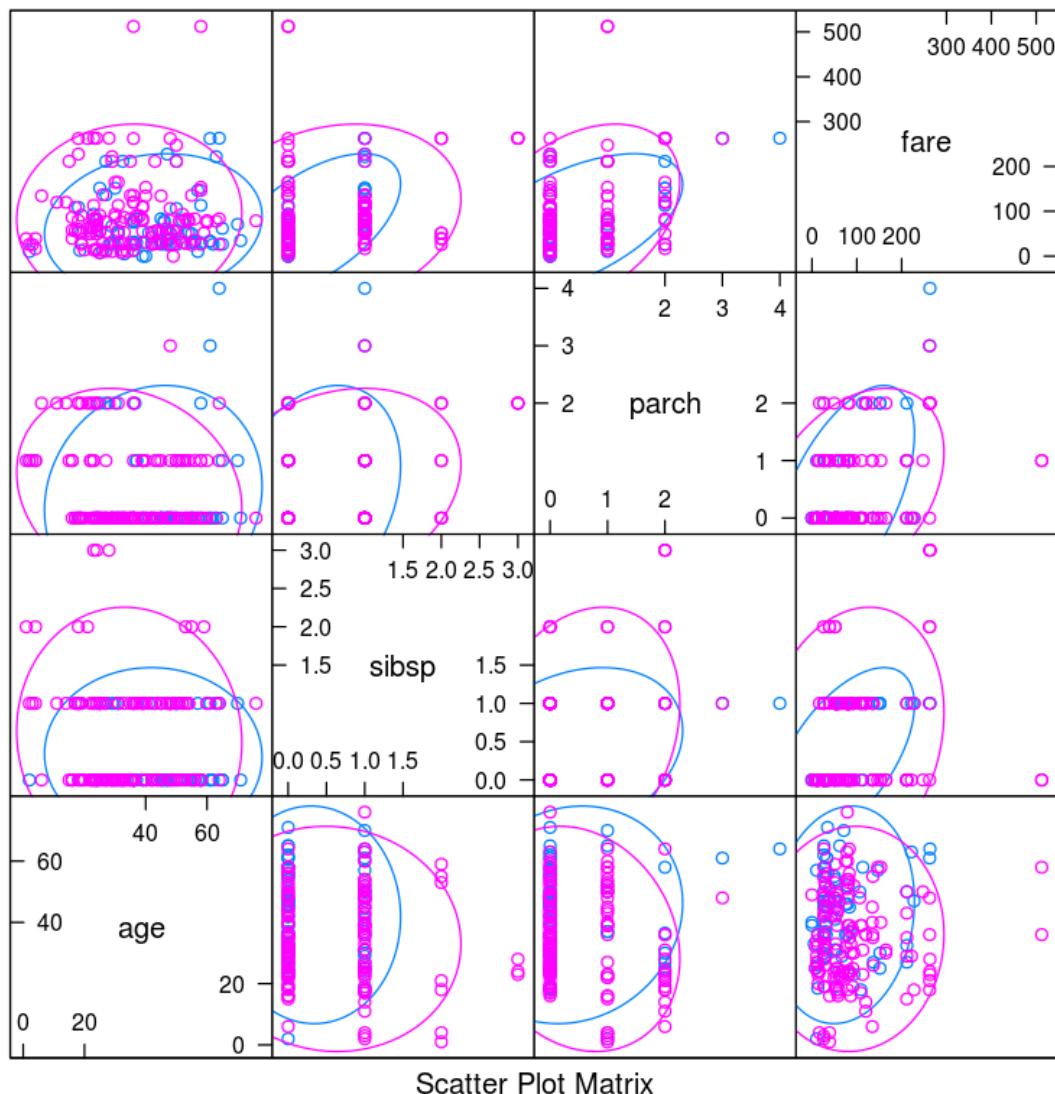


그림 11.2: featurePlot()의 실행 결과

Factor 데이터 탑입의 차트에는 [모자이크 플롯\(mosaicplot\)](#) (페이지 211)을 사용할 수 있다. pclass, sex 에 따라 생존 여부가 어떻게 달라지는지 살펴보자.

```
> mosaicplot(survived ~ pclass + sex, data=data, color=TRUE,
+             main="pclass and sex")
```

그림 11.3에 실행 결과를 보였다. pclass가 3, 성별이 male인 경우(즉 3등석 남성) 사망자가 가장 많고 pclass 1, female(1등석 여성)에서 사망자가 가장 적은 것을 볼 수 있다. 그러나 이 값은 탑승자 대비 사망률이 아니라 단순한 사망 또는 생존자 수임에 유의하기 바란다.

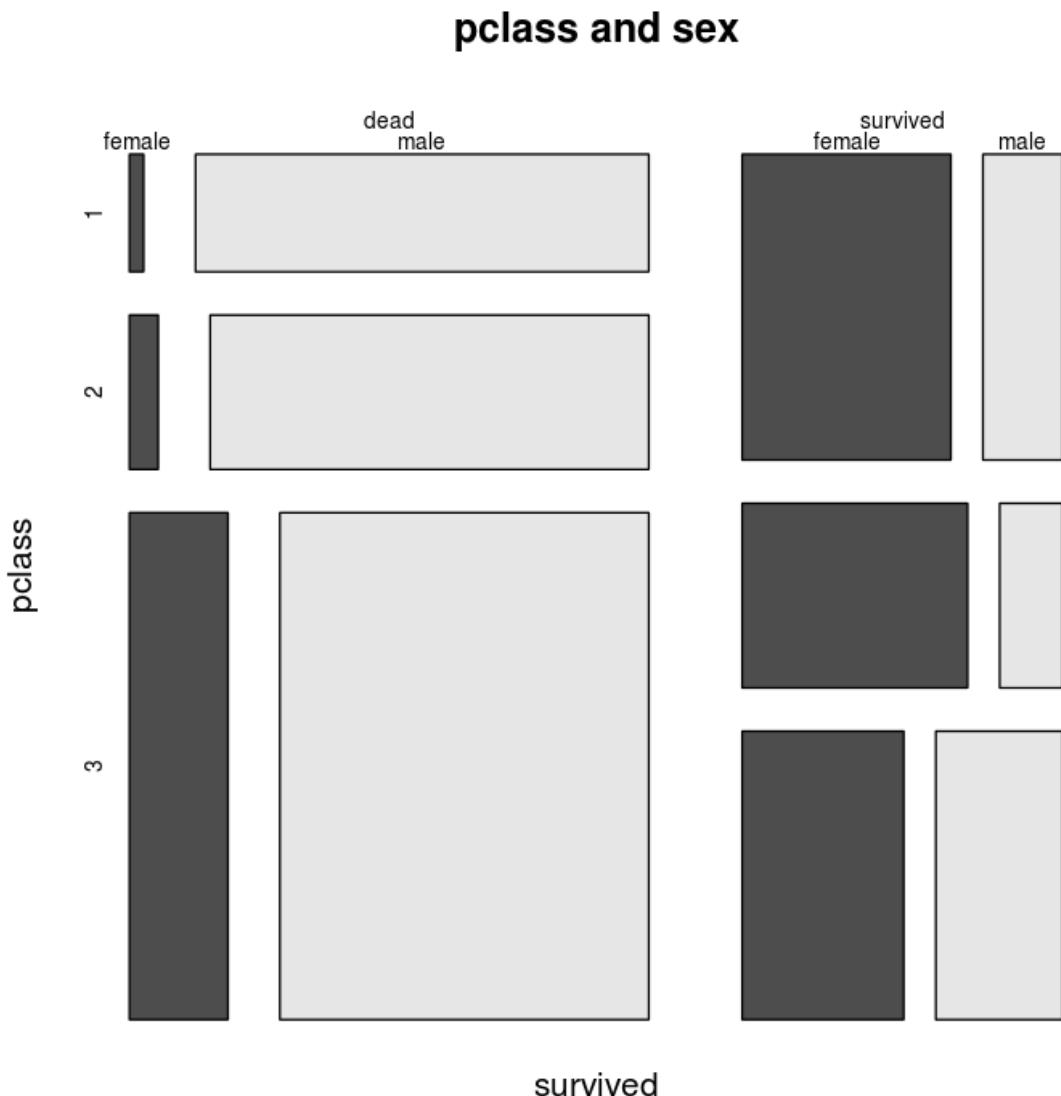


그림 11.3: mosaicplot()의 실행 결과

만약 pclass, sex 별 생존률과 사망률을 보고 싶다면 [분할표\(Contingency Table\)](#) (페이지 231)를 사용해 볼 수 있다. 탑승자 수는 다음과 같이 구한다.

```
> xtabs(~ sex + pclass, data=data)
      pclass
sex       1   2   3
female 118 83 174
male    140 143 403
```

생존자수는 survived 컬럼의 값이 “survived”인 행의 수이다.

```
> xtabs(survived == "survived" ~ sex + pclass, data=data)
      pclass
sex       1   2   3
female 115 75 84
male    44 21 66
```

편리하게도 xtabs()은 연산자를 지원한다. 따라서 두 결과를 조합해 생존율을 구할 수 있다.

```
> xtabs(survived == "survived" ~ sex + pclass, data=data) /
+     xtabs(~ sex + pclass, data=data)
      pclass
sex       1   2   3
female 0.975 0.904 0.483
male    0.314 0.147 0.164
```

지금까지 살펴본 다양한 방법들을 활용해 데이터가 어떤 모습을 띠고 있는지, 주어진 데이터로부터 어떻게 하면 탑승객의 생존 여부를 잘 추정할 수 있을지를 살펴보기 바란다.

## 4 평가 메트릭

탑승객 생존 여부 예측 모델의 성능은 Accuracy(정확도)로 하기로 한다. [모델 평가 방법](#) (페이지 323)에서 설명한 바 있는 Accuracy는 예측한 값 중 정확히 예측한 값의 비율을 뜻한다. 예를 들어 예측값이 predicted, 실제 값이 actual일 때 Accuracy는 다음과 같이 계산한다.

```
> predicted <- c(1, 0, 0, 1, 1)
> actual <- c(1, 0, 0, 0, 0)
> sum(predicted == actual) / NROW(predicted)
[1] 0.6
```

## 5 rpart 모델

rpart (페이지 343)는 나무 모형 중 하나로 다양한 변수의 상호작용을 잘 표현해준다. 타이타닉 데이터에는 NA값이 많은데 rpart는 이를 대리 변수(surrogate variable)로 처리해준다. 대리변수란 노드에서 가지치기를 할 때 사용된 변수를 대신할 수 있는 다른 변수를 뜻한다. 예를 들어 age < 10과 age ≥ 10의 기준으로 가지치기되는 노드를 가정해보자. 만약 age값이 NA인 데이터가 주어지면 age값이 없기 때문에 해당 노드에서 어느쪽 가지를 택해야할지 정할 수 없다. 이 때 rpart는 age와 10간의 비교를 대신할 수 있는 또 다른 가지치기 변수를 찾아 해당 변수를 사용한다. 예를 들어 height(키)를 생각해보자. height < 140이면 age < 10이고 height ≥ 140이면 age ≥ 10일 경우가 많다면 rpart는 age가 NA인 데이터에 대해서는 age대신 height을 대신하여 사용한다[48].

titanic.train에서 모델에 사용하기에 적합해보이지 않는 name, ticket, cabin을 제외한 나머지 변수들로 rpart 모델을 만들어보자.

```
> library(rpart)
> m <- rpart(
+   survived ~ pclass + sex + age + sibsp + parch + fare + embarked ,
+   data=titanic.train)
> p <- predict(m, newdata=titanic.train, type="class")
> head(p)
  1       3       4       5       6       7
survived survived     dead survived     dead survived
Levels: dead survived
```

### 5.1 rpart의 교차 검증

같은 방법으로 교차 검증 데이터에 대해 예측값을 구해보자. [교차 검증 준비](#) (페이지 383)에서 설명한 create\_ten\_fold\_cv()는 train에 훈련 데이터를, validation에 검증 데이터를 담은 리스트의 리스트를 반환함을 기억하기 바란다. 다음에 데이터의 대략적인 모습을 보였다.

```
List of 10
$ Fold01: List of 2
..$ train
...
..$ validation
...
$ Fold02: List of 2
```

```

.. $ train
...
.. $ validation
...
...
$ Fold10: List of 2
.. $ train
...
.. $ validation
...

```

10개 Fold에 대한 예측값과 실제값 데이터는 다음과 같이 구할 수 있다.

```

> folds <- create_ten_fold_cv()
> rpart_result <- foreach(f=folds) %do% {
+   model_rpart <- rpart(
+     survived ~ pclass + sex + age + sibsp + parch + fare + embarked,
+     data=f$train)
> predicted <- predict(model_rpart, newdata=f$validation,
+                         type="class")
> return(list(actual=f$validation$survived, predicted=predicted))
> }

```

위 코드에서 `foreach`는 리스트의 Fold01, Fold02 등을 `f`라는 변수로 받는다. 그리고 `f$train`과 `f$validation`을 사용해 `rpart()`, `predict()`를 수행한다. 결과는 `actual`에 생존 여부의 실제값을, `predicted`에 생존 여부의 예측값을 저장한 리스트로 반환되며 `foreach`는 `folds` 전체에 대한 결과를 또 다시 리스트로 묶는다.

이해를 돋기 위해 `rpart_result`의 일부를 다음에 옮겼다.

```

> head(rpart_result)
[[1]]
[[1]]$actual
[1] dead      dead      survived survived survived dead
[7] survived survived dead      dead      survived survived
...
Levels: dead survived

[[1]]$predicted

```

```

      3       17       22       44       45       85
survived    dead survived survived survived     dead
      86       104      107      111      117      128
...
Levels: dead survived

[[2]]
[[2]]$actual
...

```

## 5.2 Accuracy 평가

반환값이 여러 Fold에 대한 결과를 저장한 리스트이고 각 Fold의 결과는 actual과 predicted에 저장된 리스트임을 감안해 Accuracy 계산함수를 재작성 해보자. 여기서 만드는 함수를 뒤에서 만들 다른 모델에서도 계속 사용할 것이다.

```

> evaluation <- function(lst) {
+   accuracy <- sapply(lst, function(one_result) {
+     return(sum(one_result$predicted == one_result$actual)
+           / NROW(one_result$actual)))
+   })
+   print(sprintf("MEAN +/- SD: %.3f +/- %.3f",
+                 mean(accuracy), sd(accuracy)))
+   return(accuracy)
+ }
> evaluation(rpart_result)
[1] "MEAN +/- SD: 0.808 +/- 0.019"
[1] 0.7948718 0.8034188 0.8050847 0.8305085 0.7966102 0.8050847 0
     .8220339 0.8135593 0.8389831 0.7711864

```

evaluation() 함수는 rpart\_result를 입력으로 받아 sapply()를 수행한다. sapply는 각 fold에 대한 결과에 대해 Accuracy를 계산하며 이를 벡터로 묶는다. 마지막으로 편의를 위해 평균과 표준편차를 계산한 뒤 Accuracy의 벡터를 결과로 반환했다.

Accuracy 계산결과 rpart 모델의 성능은 80.8%로 나타났다.

## 6 ctree 모델

rpart를 통해 교차 검증 수행과 교차 검증 결과로부터 Accuracy를 계산하는 방법까지 살펴보았고, 코드가 잘 동작함을 확인했다. 이 다음 단계는 계속적으로 모델을 향상시키는 것으로 이에는 크게 1) 다른 모델링 기법을 적용하거나, 2) 데이터내에 숨겨진 쓸만한 특징값(feature)을 찾는 방법이 있다. 이 절에서는 이 중 첫번째 방법인 다른 모델링 기법의 적용을 위해 `party::ctree` (페이지 347)를 사용한다. 다음에 `ctree()`를 사용한 교차 검증을 보였다.

```
> library(party)
> ctree_result <- foreach(f=folds) %do% {
+   model_ctree <- ctree(
+     survived ~ pclass + sex + age + sibsp + parch + fare + embarked,
+     data=f$train)
+   predicted <- predict(model_ctree, newdata=f$validation,
+                         type="response")
+   return(list(actual=f$validation$survived, predicted=predicted))
+ }
> (ctree_accuracy <- evaluation(ctree_result))
[1] "MEAN +/- SD: 0.812 +/- 0.030"
[1] 0.8119658 0.8034188 0.8135593 0.8305085 0.7881356 0.7966102 0
    .8050847 0.8220339 0.8813559 0.7711864
```

전체적인 코드의 모양은 rpart와 유사하지만 ctree는 rpart와 달리 `type`에 `response`를 지정해야 `class(dead, survived)`가 반환된다는 차이가 있다. 이런 내용은 `?ctree`, `help(library="party")` 등의 명령을 통해 찾아볼 수 있다.

`ctree()` 수행결과 성능이 0.812로 나타나 `rpart()`의 0.808을 근소하게 앞서는 것을 볼 수 있다. 이처럼 성능이 유사한 경우 표준편차(sd)값을 눈여겨 볼 필요가 있다. 만약 Accuracy 평균의 차이가 sd 대비 작은 값이라면 유의미한 성능 향상이 아닐 수도 있기 때문이다.

또는 다음과 같이 Accuracy 벡터에서 밀도 그림(density) (페이지 206)을 그려 정확도의 분포를 살펴볼 수도 있다.

```
> plot(density(rpart_accuracy), main="rpart VS ctree")
> lines(density(ctree_accuracy), col="red", lty="dashed")
```

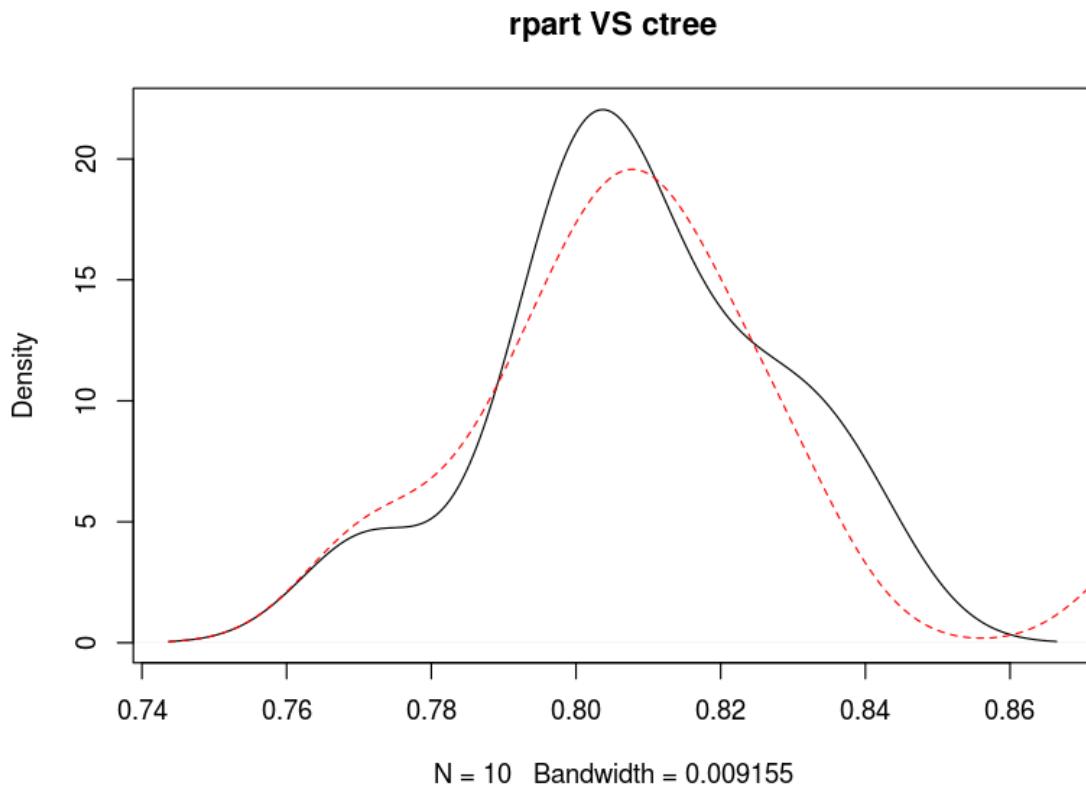


그림 11.4: rpart()와 ctree()의 Accuracy 비교

## 7 또 다른 특징(Feature)의 발견

지금까지 rpart 대신 ctree를 사용한 결과 survived에 대한 예측 정확도가 0.808에서 0.812로 향상되는 것을 관찰하였다. 그리고 분류 Y를 예측하기 위한 변수인 데이터의 특징(feature)으로는 pclass, sex, age, sibsp, parch, fare, embarked처럼 데이터에 직접적으로 주어진 속성을 사용해왔다.

이제 생각해 볼 수 있는 성능 향상 기법은 데이터에 숨겨진 또 다른 특징(feature)을 발견하는 것이다. 이 절에서 생각해 볼 아이디어는 ‘가족’이다. 혹시 부모가 사망한 경우 보호자가 없어진 자식의 사망률이 더 낮아지지는 않았을까? 또는 자식이 있는 부모의 경우 자식을 살리기 위해 부모가 희생하지는 않았을까? 반대로 젊은 자녀가 나이든 부모를 구하기 위해 희생하지는 않았을까? 이런 점들을 생각해보면 가족 단위로 데이터를 묶어서 모델링 하는 것이 의미있어 보인다.

이러한 연관 관계를 표현하기 위해 데이터에 가족단위 식별번호인 family\_id를 부여하고 한 사람의 생존 가능성을 평가할 때 다른 가족 구성원의 생사 여부를 고려하는 모델을 만들어보자.

## 7.1 ticket을 사용한 가족 식별

타이타닉 데이터에서 sibsp 속성은 함께 탑승한 형제 또는 배우자의 수를 나타내며, parch는 함께 탑승한 부모 또는 자녀의 수를 나타낸다. 따라서 parch가 0이 아니면 함께 탑승한 부모 또는 자녀가 있음을 알 수 있다. 그러나 parch 값이 0이 아니라고 해도 함께 탑승한 부모나 자녀가 누구인지까지는 알 수 없다. 따라서 추측에 의해서 가족 관계를 찾아내야한다.

타이타닉 데이터를 살펴보면 다음 속성들이 한 가족을 찾는 단서가 될 수 있음을 알 수 있다.

- cabin(선실번호): 같은 선실에 탑승한 사람들은 한 가족일 가능성이 높다.
- embarked(탑승한 곳)과 name(이름): 같은 곳에서 탑승했고 이름 중 성이 같다면 한 가족 일 가능성이 높다.
- ticket(티켓 번호): 한가족이 표를 한번에 샀다면 티켓번호가 같을 것이다.

세가지 방법 중 이 절에서는 ticket을 사용해 가족을 찾아볼 것이다. 또한 ticket이 같은 값을 갖는 데이터를 훈련 데이터와 검증 데이터에 모두로부터 찾을 것이다. 그 이유는 훈련데이터에 있는 탑승객 중 일부는 검증 데이터 내 탑승객과 가족 관계일 수 있기 때문이다.

훈련 데이터와 검증 데이터를 모두 사용하는 방법은 훈련 데이터만 사용해 모델을 만들고, 검증 데이터로만 성능을 평가하는 일반적인 그림에는 맞지 않는다. 그러나 검증 데이터가 주어질 때마다 모델링을 새로 해야한다는 점을 제외하면 잘못된 것은 아니다. 검증 데이터가 새로 주어질 때마다 모델을 만드므로 속도가 느리지 않을까 생각해 볼 수는 있지만, 타이타닉 탑승자 명단은 이미 정해진 것이어서 매번 새로운 데이터에 대한 예측을 새로 할 필요 자체가 없다.

ticket이 가족을 찾는데 얼마나 유용한지 확인해보자. 다음은 검증 데이터를 분리하지 않은 titanic.train 을 ticket에 따라 정렬해 표시한 예이다.

```
> View(titanic.train[order(titanic.train$ticket),
+   c("ticket", "parch", "name", "cabin", "embarked")])
ticket  parch                               name cabin embarked
...
110413      2                           Taussig, Miss. Ruth    E68      S
110413      1                           Taussig, Mr. Emil    E67      S
110413      1 Taussig, Mrs. Emil (Tillie Mandelbaum)    E67      S
...
111361      1                         Hippach, Miss. Gertrude    B18      C
111361      1                         Hippach, Mrs. Louis Albert    B18      C
```

...

결과를 보면 ticket 번호가 110413이고 성이 Taussig인 세명의 탑승자가 보인다. 이들의 선실은 E68, E67, E67로 근처이며 탑승지도 S(Southampton)으로 모두 동일하다. 따라서 이들이 한 가족임을 쉽게 짐작할 수 있다. 마찬가지로 ticket 번호 111361이고 성이 Hippach인 가족 역시 쉽게 찾을 수 있다. Hippach 가족 역시 cabin이 B18로 동일하고 embarked가 C(Cherbourg)로 동일하다.

Taussig 가족을 고려하면 인접 선실에도 한 가족이 있을 수 있어 cabin이 ticket에 비해 불완전한 속성임을 알 수 있다. name에서 성을 추출하고 embarked를 고려하여 가족을 찾는 경우도 생각해볼 수 있지만 embarked에는 NA값이 저장된 행이 있어 ticket에 비해 온전치 못하다. 다음은 ticket, embarked, cabin에서 값이 NA인 행의 수를 확인하는 코드이다.

```
> sum(is.na(titanic.train$ticket))
[1] 0
> sum(is.na(titanic.train$embarked))
[1] 2
> sum(is.na(titanic.train$cabin))
[1] 912
```

확인 결과 ticket에는 NA 값이 저장된 행이 없으나 embarked에는 2개 행에 NA가 저장되어있고 cabin에는 912개 행에 NA가 저장되어있다. NA 값이 없는 ticket 속성이 더 정확하고 사용에 편리함을 알 수 있다.

## 7.2 생존 확률 예측

이제 탑승자들을 가족단위로 묶을 수 있음이 확실해졌으므로 각 가족 구성원의 생존 확률을 구해보자. 이 생존 확률은 후에 가족단위로 묶일 것이다.

`rpart`, `ctree`에서와 마찬가지로 교차 검증을 위해 다음 형태의 `foreach` 반복문을 만든다.

```
family_result <- foreach(f=folds) %do% {
  ...
}
```

이후 내용은 이 `foreach` 블럭에 들어가는 코드를 설명할 것이며 한개 fold를 뜻하는 `f`를 사용할 것이다.

가족 정보를 훈련 데이터와 검증 데이터를 모두 사용해 구하기로 하였으므로 데이터를 합하자. 그 뒤, `party::ctree` (페이지 347)에서 보인 모델을 적용해 생존 여부를 예측하는 확률 prob 컬럼을 추가한다.

```
f$train$type <- "T"
f$validation$type <- "V"
all <- rbind(f$train, f$validation)
ctree_model <- ctree(
  survived ~ pclass + sex + age + sibsp + parch + fare + embarked,
  data=f$train)
all$prob <- sapply(
  predict(ctree_model, type="prob", newdata=all),
  function(result) { result[1] })
```

이 코드는 한 fold내 훈련데이터(train)과 검증데이터(validation)를 합쳐 하나의 데이터 프레임 all 을 만든다. 이 때 type 컬럼을 추가해 훈련 데이터와 검증 데이터를 후에 분리할 수 있도록 해둔다. ctree()는 훈련데이터만 사용해 만들되 predict()는 훈련데이터와 검증데이터 모두에 수행한다. (당연한 이야기이지만 validation은 ctree()를 사용한 모델링에서는 사용하지 않는다. validation\$survived를 훈련 과정에서 사용해버리면 validation을 사용한 성능 평가가 무의미해져버리기 때문이다.) 최종 결과로 생존 확률만 뽑기 위해 sapply()를 사용해 predict() 결과의 첫번째 컬럼만 선택해 사망확률이 저장된 두번째 컬럼을 제외한다. 코드 수행결과 all 에는 type 컬럼과 prob 컬럼이 추가 된다.

### 7.3 가족 ID 부여

생존 확률의 예측값을 가족단위로 모으기 위해 어떤 탑승객이 누구와 가족인지를 알아보자. ticket이 이에 적당한 속성임을 앞서 확인하였으므로 티켓 번호별로 가족 ID를 부여할 것이다. 이런 작업에는 [plyr 패키지](#) (페이지 121)이 매우 유용하다. 혹시라도 잘 기억이나지 않는다면 책의 전반부를 복습하기 바란다.

```
family_idx <- 0
ticket_based_family_id <- ddply(all, .(ticket), function(rows) {
  family_idx <- family_idx + 1
  return(data.frame(family_id=paste0("TICKET_", family_idx)))
})
```

이 코드에서 가장 핵심적인 부분은 ddply()이다. ddply()에서 all(훈련데이터와 검증 데이터를 모두 모은 데이터 프레임)은 .(ticket)에 의해 같은 ticket 값을 갖는 행끼리 그룹지어지며 같은 티켓번호를 갖는 행들은 뒤에 따르는 함수에 rows란 이름으로 넘겨진다. ddply()에 인자로 주

어진 함수에서는 family\_idx 를 1 증가시킨 뒤<sup>2)</sup> 이를 family\_id 라는 컬럼에 TICKET\_family\_id 형태로 저장한다. 문자열을 합치는데는 paste0()를 사용하였다. paste0()는 인자로 주어진 문자열들을 공백없이 합친다. 반면 paste()는 주어진 인자들을 합칠 때 사이에 공백을 삽입한다.

ddply()에 인자로 넘어간 rows의 값을 보고 싶거나 인자로 주어진 함수의 동작을 한줄씩 확인해보고 싶다면 [디버깅](#) (페이지 161)에서 설명한 browser()를 사용하거나 [testthat](#) (페이지 157)과 같은 유닛 테스팅을 사용해 검증해보면 좀 더 코딩을 쉽게 할 수 있으니 참고하기 바란다.

첫번째 fold에 대해 위 코드를 수행한 결과 ticket\_based\_family\_id 에는 다음과 같은 데이터 프레임이 저장된다.

```
> str(ticket_based_family_id)
'data.frame': 861 obs. of 2 variables:
$ ticket    : chr "110152" "110413" "110465" "110469" ...
$ family_id: Factor w/ 861 levels "TICKET_1","TICKET_2",...: 1 2 3 4 5
   6 7 8 9 10 ...
> head(ticket_based_family_id)
  ticket family_id
1 110152 TICKET_1
2 110413 TICKET_2
3 110465 TICKET_3
4 110469 TICKET_4
5 110489 TICKET_5
6 110564 TICKET_6
```

이제 all 데이터 프레임에 ticket 값에 따라 family\_id 를 추가할 차례이다. all\$ticket을 보면 서 ticket\_based\_family\_id로부터 family\_id를 찾아 이를 all\$family\_id에 저장하면 된다.

```
all <- adply(all,
             1,
             function(row) {
               family_id <- NA
               if (!is.na(row$ticket)) {
                 family_id <- subset(ticket_based_family_id,
                                       ticket == row$ticket)$family_id
               }
               return(data.frame(family_id=family_id))
             })
```

<sup>2)</sup>family\_idx 가 함수 외부에 선언된 변수이므로 <<- 를 사용한 할당문이 사용되었다.

})

adply()를 all 데이터에 .margin=1 인자와 함께 호출하였다. .margin=1은 매행마다 호출됨을 뜻하고 .margin=2는 매 열마다 호출된다. 따라서 뒤따라 나온 함수의 인자 row에는 all의 각 행이 넘겨진다. adply()에 인자로 주어진 함수는 ticket\_based\_family\_id에서 row\$ticket과 같은 ticket 값을 갖는 행을 찾고, 그 행의 family\_id를 family\_id 변수에 저장한다. 최종적으로 family\_id는 함수에서 데이터 프레임으로 반환된다. adply()는 이 반환값을 all 데이터 프레임에 새로운 컬럼으로 추가해 반환하므로 이를 all에 저장하면 family\_id가 추가된 all을 얻을 수 있다.

다음은 첫번째 fold에 위 코드를 수행한 결과 구해진 all 데이터 프레임의 구조이다.

```
> str(all)
'data.frame': 1178 obs. of 14 variables:
 $ pclass      : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
 $ survived    : Factor w/ 2 levels "dead","survived": 2 1 1 2 2 1 2 1 1
   2 ...
 $ name        : chr  "Allen, Miss. Elisabeth Walton" "Allison, Mr.
 Hudson Joshua Creighton" "Allison, Mrs. Hudson J C (Bessie Waldo
 Daniels)" "Anderson, Mr. Harry" ...
 $ sex         : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 2 1
   ...
 $ age          : num  29 30 25 48 63 39 53 71 47 18 ...
 $ sibsp        : int  0 1 1 0 1 0 2 0 1 1 ...
 $ parch        : int  0 2 2 0 0 0 0 0 0 0 ...
 $ ticket       : chr  "24160" "113781" "113781" "19952" ...
 $ fare         : num  211.3 151.6 151.6 26.6 78 ...
 $ cabin        : chr  "B5" "C22 C26" "C22 C26" "E12" ...
 $ embarked     : Factor w/ 3 levels "C","Q","S": 3 3 3 3 3 3 3 1 1 1 ...
 $ type         : chr  "T" "T" "T" "T" ...
 $ prob         : num  0.0547 0.6857 0.0547 0.6857 0.0547 ...
 $ family_id: Factor w/ 861 levels "TICKET_1","TICKET_2",...: 176 47 47
   118 86 16 71 766 774 774 ...
```

## 7.4 가족 구성원 생존 확률의 병합

all 데이터 프레임에 각 탑승객의 생존확률 prob과 어느 가족에 속하는지를 의미하는 family\_id가 준비되었으므로 생존확률 prob을 좀더 다양하게 모을 준비가 되었다. 이 절에서는 다음에

보인 정보를 all 데이터 프레임에 추가할 것이다.

- avg\_prob: 가족 구성원들의 평균 생존확률
- maybe\_parent, maybe\_child: 특정 탑승객이 부모인지 또는 자녀인지의 여부
- parent\_prob, child\_prob: 부모의 평균 생존률과 자녀의 평균 생존률

가족의 평균 생존률 avg\_prob은 ddply()로 family\_id가 같은 행들을 모아 prob의 평균을 avg\_prob에 저장하는 것으로 간단하게 구할 수 있다.

```
all <- ddply(all,
             .(family_id),
             function(rows) {
               rows$avg_prob <- mean(rows$prob)
               return(rows)
             })
```

다음은 각 탑승객이 부모 또는 자녀 중 어느쪽에 속하는지를 알아볼 차례이다. 부모인지 자녀인지의 여부는 maybe\_parent, maybe\_child에 저장될 것이며, 부모 자녀를 판단하는 기준으로는 나이(age) 속성을 사용한다.

```
all <- ddply(all, .(family_id), function(rows) {
  rows$maybe_parent <- FALSE
  rows$maybe_child <- FALSE
  if (NROW(rows) == 1 ||
      sum(rows$parch) == 0 ||
      NROW(rows) == sum(is.na(rows$age))) {
    return(rows)
  }
  max_age <- max(rows$age, na.rm=TRUE)
  min_age <- min(rows$age, na.rm=TRUE)
  return(adply(rows, 1, function(row) {
    if (!is.na(row$age) && !is.na(row$sex)) {
      row$maybe_parent <- (max_age - row$age) < 10
      row$maybe_child <- (row$age - min_age) < 10
    }
    return(row)
  })))
})
```

```
})
```

이 ddply()는 .(family\_id)마다 행을 묶고 있으므로 한 가족에 해당하는 행들이 인자로 주어진 함수에 rows로 넘겨진다. 함수내 if 문에서는 세가지 조건을 검사하여 이 중 하나라도 해당될 경우 maybe\_parent와 maybe\_child 열을 FALSE로 지정한다. 다음에 그 세가지 조건을 보였다.

- NROW(rows) == 1: 가족 구성원의 수가 한명이라면 부모도 자녀도 아니다.
- sum(rows\$parch) == 0: parch는 부모 또는 자녀의 수를 뜻한다. 만약 이 값이 모든 행에서 0이라면 가족 구성원 중 어느 누구도 부모 또는 자녀가 없다는 의미이므로 모든 구성원이 부모도 자녀도 아니다.
- NROW(rows) == sum(is.na(rows\$age)): 만약 모든 행에 나이가 저장되어 있지 않다면 누가 부모이고 자녀인지를 알 수 없다. 따라서 어느 누구에게도 부모 또는 자녀인지의 여부를 지정하지 않는다.

조건을 통과하면 나이를 사용해 부모, 자녀 여부를 판단한다. 가족 구성원의 최소와 최대 나이를 각각 min\_age, max\_age에 저장한 다음 만약 탑승객의 나이가 max\_age - 10 이상이라면 그 탑승객을 부모로 간주하여 maybe\_parent에 TRUE를 저장한다. 마찬가지로 나이가 min\_age + 10 이하라면 자녀로 간주하여 maybe\_child에 TRUE를 저장했다. 이런 방법을 사용한 부모, 자녀 여부 판단이 완벽하다고하는 할 수 없지만 어느 정도 쓸만한 결과는 얻을 수 있다.

부모 자녀 여부를 판단하고나면 부모의 평균 생존확률(avg\_parent\_prob)와 자녀의 평균 생존확률(avg\_child\_prob)를 다음과 같이 구할 수 있다.

```
all <- ddply(all, .(family_id), function(rows) {
  rows$avg_parent_prob <- rows$avg_prob
  rows$avg_child_prob <- rows$avg_prob
  if (NROW(rows) == 1 || sum(rows$parch) == 0) {
    return(rows)
  }
  parent_prob <- subset(rows, maybe_parent == TRUE)[, "prob"]
  if (NROW(parent_prob) > 0) {
    rows$avg_parent_prob <- mean(parent_prob)
  }
  child_prob <- c(subset(rows, maybe_child == TRUE)[, "prob"])
  if (NROW(child_prob) > 0) {
    rows$avg_child_prob <- mean(child_prob)
  }
})
```

```

    return(rows)
}

```

이 코드에서 가장 눈여겨볼만한 점은 avg\_parent\_prob와 avg\_child\_prob의 기본값이 가족의 평균 생존 확률인 avg\_prob라는 것이다. 만약 NA를 기본값으로 부여한다면 부모 또는 자녀가 없는 많은 탑승객들의 avg\_parent\_prob, avg\_child\_prob에 NA가 저장되게 된다. 이 경우 NA를 저장한 행의 수가 많아 모델링 알고리즘 적용시 힘들게 찾아낸 부모와 자녀의 평균 생존률 정보를 무시하게 되버릴 수 있다. 따라서 적당한 적절한 기본값을 부여하였다.

부모의 평균 생존확률과 자녀의 평균 생존확률은 가족내에서 maybe\_parent 또는 maybe\_child가 TRUE인 행들을 subset()으로 찾아 평균을 구해 avg\_parent\_prob, avg\_child\_prob에 저장하는 방식으로 구현했다.

## 7.5 가족 정보를 사용한 ctree() 모델링

이제 all 데이터를 사용해 모델을 만들고 그 성능을 평가해보자. 지금까지 추가한 컬럼 중 type, avg\_prob, maybe\_parent, maybe\_child, avg\_parent\_prob, avg\_child\_prob를 사용할 것이다.

첫번째 fold에 지금까지 설명한 내용을 적용한 all 데이터 프레임의 모습은 다음과 같다.

```

> str(all)
'data.frame': 1178 obs. of 19 variables:
 $ pclass          : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
 $ survived        : Factor w/ 2 levels "dead","survived": 2 2 2 2 1 2
   1 1 1 1 ...
 $ name            : chr  "Cherry, Miss. Gladys" "Maioni, Miss. Roberta
   "Rothes, the Countess. of (Lucy Noel Martha Dyer-Edwards)" "
   Taussig, Miss. Ruth" ...
 $ sex             : Factor w/ 2 levels "female","male": 1 1 1 1 2 1 2
   2 2 2 ...
 $ age              : num  30 16 33 18 52 39 NA 47 30 42 ...
 $ sibsp           : int  0 0 0 0 1 1 0 0 0 0 ...
 $ parch           : int  0 0 0 2 1 1 0 0 0 0 ...
 $ ticket          : chr  "110152" "110152" "110152" "110413" ...
 $ fare             : num  86.5 86.5 86.5 79.7 79.7 ...
 $ cabin           : chr  "B77" "B79" "B77" "E68" ...
 $ embarked         : Factor w/ 3 levels "C","Q","S": 3 3 3 3 3 3 3 3 3 3
   3 ...

```

```
$ type           : chr  "T" "T" "T" "T" ...
$ prob           : num  0.0591 0.0591 0.0591 0.0591 0.6364 ...
$ family_id      : Factor w/ 861 levels "TICKET_1","TICKET_2",...
1 2 2 2 3 3 4 5 ...
$ avg_prob       : num  0.0591 0.0591 0.0591 0.2515 0.2515 ...
$ maybe_parent   : logi  FALSE FALSE FALSE FALSE TRUE FALSE ...
$ maybe_child    : logi  FALSE FALSE FALSE TRUE FALSE FALSE ...
$ avg_parent_prob: num  0.0591 0.0591 0.0591 0.6364 0.6364 ...
$ avg_child_prob: num  0.0591 0.0591 0.0591 0.0591 0.0591 ...
```

all에 있는 정보 중 훈련 데이터를 사용해 ctree()를 수행하고 이를 검증 데이터에 적용해보자.

```
f$train <- subset(all, type == "T")
f$validation <- subset(all, type == "V")

(m <- ctree(survived ~ pclass + sex + age + sibsp + parch + fare +
embarked + maybe_parent + maybe_child + age + sex + avg_prob + avg-
parent_prob + avg_child_prob,
data=f$train))

print(m)
predicted <- predict(m, newdata=f$validation)
```

그 결과 첫번째 folds로부터 만들어진 ctree()는 다음과 같다.

```
> print(m)

Conditional inference tree with 7 terminal nodes

Response: survived
Inputs: pclass, sex, age, sibsp, parch, fare, embarked, maybe_parent,
maybe_child, avg_prob, avg_parent_prob, avg_child_prob
Number of observations: 1060

1) avg_prob <= 0.5751314; criterion = 1, statistic = 385.146
2) sex == {male}; criterion = 1, statistic = 65.12
3) age <= 11; criterion = 1, statistic = 20.899
```

```

 4)*  weights = 21
 3) age > 11
 5)*  weights = 80
2) sex == {female}
6) pclass == {3}; criterion = 1, statistic = 63.64
7) avg_parent_prob <= 0.4238411; criterion = 0.965, statistic =
   8.81
8)*  weights = 116
7) avg_parent_prob > 0.4238411
9)*  weights = 9
6) pclass == {1, 2}
10)* weights = 203
1) avg_prob > 0.5751314
11) avg_child_prob <= 0.6507765; criterion = 1, statistic = 47.28
12)* weights = 126
11) avg_child_prob > 0.6507765
13)* weights = 505

```

확인 결과 avg\_prob, avg\_child\_prob 등이 유용하게 사용되고 있다.

## 7.6 성능 평가

가족 정보를 사용한 모델의 성능을 알아보자. 이 모델을 위해 설명한 내용이 길어 이해를 돋기 위해 전체 코드를 아래에 보였다.

```

family_result <- foreach(f=folds) %do% {
  f$train$type <- "T"
  f$validation$type <- "V"
  all <- rbind(f$train, f$validation)
  ctree_model <- ctree(
    survived ~ pclass + sex + age + sibsp + parch + fare + embarked ,
    data=f$train)
  all$prob <- sapply(predict(ctree_model, type="prob", newdata=all),
                      function(result) { result[1] })

  # Assign family ID by ticket
  family_idx <- 0
  ticket_based_family_id <- ddply(all, .(ticket), function(rows) {

```

```

family_idx <- family_idx + 1
return(data.frame(family_id=paste0("TICKET_", family_idx)))
})

all <- adply(all, 1,
            function(row) {
              family_id <- NA
              if (!is.na(row$ticket)) {
                family_id <- subset(ticket_based_family_id,
                                      ticket == row$ticket)$family_id
              }
              return(data.frame(family_id=family_id))
            })

# avg_prob
all <- ddply(all,
             .(family_id),
             function(rows) {
               rows$avg_prob <- mean(rows$prob)
               return(rows)
             })

# maybe_{parent, child}
all <- ddply(all, .(family_id), function(rows) {
  rows$maybe_parent <- FALSE
  rows$maybe_child <- FALSE
  if (NROW(rows) == 1 ||
      sum(rows$parch) == 0 ||
      NROW(rows) == sum(is.na(rows$age))) {
    return(rows)
  }
  max_age <- max(rows$age, na.rm=TRUE)
  min_age <- min(rows$age, na.rm=TRUE)
  return(adply(rows, 1, function(row) {
    if (!is.na(row$age) && !is.na(row$sex)) {
      row$maybe_parent <- (max_age - row$age) < 10
      row$maybe_child <- (row$age - min_age) < 10
    }
  }))
})

```

```

    }
    return(row)
})))
})

# avg_{parent,child}_prob.

all <- ddply(all, .(family_id), function(rows) {
  rows$avg_parent_prob <- rows$avg_prob
  rows$avg_child_prob <- rows$avg_prob
  if (NROW(rows) == 1 || sum(rows$parch) == 0) {
    return(rows)
  }
  parent_prob <- subset(rows, maybe_parent == TRUE)[, "prob"]
  if (NROW(parent_prob) > 0) {
    rows$avg_parent_prob <- mean(parent_prob)
  }
  child_prob <- c(subset(rows, maybe_child == TRUE)[, "prob"])
  if (NROW(child_prob) > 0) {
    rows$avg_child_prob <- mean(child_prob)
  }
  return(rows)
})

f$train <- subset(all, type == "T")
f$validation <- subset(all, type == "V")
(m <- ctree(survived ~ pclass + sex + age + sibsp + parch + fare +
  embarked + maybe_parent + maybe_child + age + sex + avg_prob +
  avg_parent_prob + avg_child_prob,
  data=f$train))
print(m)
predicted <- predict(m, newdata=f$validation)
return(list(actual=f$validation$survived, predicted=predicted))
}

```

성능 평가 결과는 다음과 같다.

```
> family_accuracy <- evaluation(family_result)
```

```
[1] "MEAN +/- SD: 0.815 +/- 0.028"
```

cmtree()만 사용했던 이전 모델의 경우 성능이  $0.812 \pm 0.030$  이었으므로 가족 정보를 사용해 성능이 개선되었음을 알 수 있다.

## 8 교차 검증의 병렬화(Parallelization)

10겹 교차 검증은 데이터를 10개로 쪼개 모델링과 성능 평가를 10회 반복하는 방법이므로 모델링과 성능 평가를 한번 수행하는 경우에 비해 수행시간이 10배로 길어진다. 만약 성능 평가의 정확성을 높히기 위해 10겹 교차 검증을 3회 반복한다면 수행시간은 30배가 된다.

하지만 여러 Fold에 대한 모델 생성과 평가를 동시에 수행한다면 수행 시간을 줄일 수 있다. 이 절에서는 병렬화를 통해 교차 검증을 더 빠르게 수행하는 방법을 알아본다.

### 8.1 10겹 교차 검증의 3회 반복 수행

10겹 교차 검증의 수행에는 caret::createFolds()를 사용했다. caret::createMultiFolds()는 k 겹 교차 검증을 times회 반복수행하는데 사용하는 함수로 createMultiFolds(분류 대상이 되는 변수, k, times) 형태로 호출한다. createMultiFolds()의 반환값은 리스트로서 i번째 반복의 j 번째 fold를 \$Foldj.Rep1 를 색인으로하는 리스트로 반환한다.

다음에 createMultiFolds(titanic.train\$survived, k=10, times=3) 결과의 일부를 보였다.

```
> createMultiFolds(titanic.train$survived, k=10, times=3)
$Fold01.Rep1
[1]    1    2    3    4    5    6    7
[8]    9   10   11   14   16   18   19
...
$Fold02.Rep1
[1]    1    2    3    5    6    8    9
[8]   10   11   12   13   14   15   17
...
$Fold10.Rep1
...
$Fold01.Rep2
...
$Fold02.Rep2
...
$Fold10.Rep3
```

예를 들어 위 결과화면에서 Fold01.Rep1는 1번째 반복의 1번째 Fold를 의미하며, 이에 저장된 값은 해당 Fold에서 훈련데이터로 사용할 데이터의 색인 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 14, 16, 18, 19가 저장되어 있다.

따라서 createMultiFolds()가 반환한 리스트를 순서대로 따라가면서 주어진 색인에 해당하는 행을 훈련 데이터로, 주어진 색인에 없는 행을 검증 데이터로 만들면 10겹 교차검증의 3회 반복을 위한 데이터가 준비된다.

다음은 이를 구현한 함수이다.

```
create_three_ten_fold_cv <- function() {
  set.seed(137)
  lapply(createMultiFolds(titanic.train$survived, k=10, times=3),
    function(idx) {
      return(list(train=titanic.train[idx, ],
                  validation=titanic.train[-idx, ]))
    })
}
```

create\_three\_ten\_fold\_cv()는 [ctree 모델](#) (페이지 395)에서 설명한 모델등에 어렵지 않게 적용할 수 있다. create\_three\_ten\_fold\_cv()의 결과를 folds에 저장하고 이를 실행하면 된다. 다음에 ctree()를 사용한 예를 보였다.

```
> folds <- create_three_ten_fold_cv()
> ctree_result <- foreach(f=folds) %do% {
+   model_ctree <- ctree(survived ~ pclass + sex + age + sibsp + parch
+   + fare + embarked,
+   data=f$train)
+   predicted <- predict(model_ctree, newdata=f$validation, type="
+ response")
+   return(list(actual=f$validation$survived, predicted=predicted))
+ }

> (ctree_accuracy <- evaluation(ctree_result))
[1] "MEAN +/- SD: 0.812 +/- 0.035"
[1] 0.8119658 0.8034188 0.8135593 0.8305085 0.7881356 0.7966102
[7] 0.8050847 0.8220339 0.8813559 0.7711864 0.8135593 0.8050847
[13] 0.8050847 0.8632479 0.8119658 0.7881356 0.8135593 0.8644068
[19] 0.8389831 0.7457627 0.8559322 0.8119658 0.8461538 0.7966102
```

```
[25] 0.8135593 0.8644068 0.8220339 0.7288136 0.7881356 0.7627119
```

실행결과 30개의 결과가 잘 구해졌다.

## 8.2 foreach()와 %dopar%를 사용한 병렬화

foreach()에서 %do%를 사용해 순차적으로 각 fold를 처리하는 코드의 수행 시간은 다음과 같이 system.time()을 사용해 측정할 수 있다.

```
> system.time(ctree_result <- foreach(f=folds) %do% {
+   model_ctree <- ctree(
+     survived ~ pclass + sex + age + sibsp + parch + fare + embarked,
+     data=f$train)
+   predicted <- predict(model_ctree, newdata=f$validation,
+                         type="response")
+   return(list(actual=f$validation$survived, predicted=predicted))
+ }
  user    system elapsed
3.132    0.000   3.145
```

doMC (페이지 154) 패키지를 사용해 4개 작업을 동시에 실행하도록 하기 위해 registerDoMC(cores=4)를 수행한 뒤 %do% 대신 %dopar%를 지정하면 4개 Fold를 동시에 처리하는 병렬 처리가 가능하다.

```
> library(doMC)
> registerDoMC(cores=4)
> system.time(ctree_result <- foreach(f=folds) %dopar% {
+   model_ctree <- ctree(
+     survived ~ pclass + sex + age + sibsp + parch + fare + embarked,
+     data=f$train)
+   predicted <- predict(model_ctree, newdata=f$validation,
+                         type="response")
+   return(list(actual=f$validation$survived, predicted=predicted))
+ }
  user    system elapsed
3.804    0.108   1.032
```

병렬 처리를 사용하면 수행 성능은 빨라지지만 doMC (페이지 154)에서 설명한 바와 같이 다음 사항들에 주의해야 한다.

- CPU 코어 갯수를 고려해 동시에 수행할 작업의 갯수를 설정해야한다. 특히 입출력이 적고 CPU 연산이 많은 함수의 경우 CPU 코어 갯수보다 너무 많은 작업을 동시에 실행시키면 실행 성능이 오히려 낮아질 수 있다. 예를들어 코어가 4개인 머신에서 registerDoMC(cores=16)을 지정하면 registerDoMC(cores=8) 보다 더 느리게 수행 될 수 있다.
- 메모리 사용량에따라 동시에 작업의 수를 조절해야한다. 머신 러닝 알고리즘의 수행에는 때때로 많은 메모리가 필요하며 동시에 수행되는 작업의 수에 비례해 메모리 소요량이 늘어난다. 만약 R 코드가 필요로 하는 메모리의 양이 시스템에 부착된 물리적 메모리를 초과하게되면 디스크를 메모리로 사용하기 시작하게 된다. 그러면 메모리와 디스크간에 데이터 이동에만 긴 시간을 허비할 수 있고 이에따라 코드 수행의 전체 시간이 길어질 수 있다.

## 참고 문헌

- [1] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.
- [2] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.
- [3] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, 2012.
- [4] D. M. Smith W. N. Venables and the R Core Team. *An Introduction to R*. Version 2.15.1 edition, 2012.
- [5] Steven Buechler. R language fundamentals. <http://nd.edu/~steve/Rcourse/Lecture1v2.pdf>, 2007.
- [6] L. Chihara. R guide. <http://people.carleton.edu/~lchihara/Splus/RVectors.pdf>, Mar 2010.
- [7] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, and Allan Engelhardt. caret: Classification and regression training. <http://CRAN.R-project.org/package=caret>, 2012. R package version 5.15-044.
- [8] John Chambers. *Software for Data Analysis*. Springer, 2010.
- [9] G. Grothendieck. sqldf: Perform sql selects on r data frames. <http://CRAN.R-project.org/package=sqldf>, 2012. R package version 0.4-6.4.
- [10] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 4 2011. <http://www.jstatsoft.org/v40/i01>.

- [11] Hadley Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21, 2007. <http://www.jstatsoft.org/v21/i12>.
- [12] M Dowle, T Short, and S Lianoglou. data.table: Extension of data.frame for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns. <http://CRAN.R-project.org/package=data.table>, 2013. R package version 1.8.8.
- [13] Revolution Analytics and Steve Weston. foreach: Foreach looping construct for r. <http://CRAN.R-project.org/package=foreach>, 2013. R package version 1.4.1.
- [14] Revolution Analytics. doMC: Foreach parallel adaptor for the multicore package. <http://CRAN.R-project.org/package=doMC>, 2013. R package version 1.3.0.
- [15] Matthias Burger, Klaus Juenemann, and Thomas Koenig. Runit: R unit test framework. <http://CRAN.R-project.org/package=RUnit>, 2010. R package version 0.4.26.
- [16] Hadley Wickham. testthat: Testthat code. tools to make testing fun :). <http://CRAN.R-project.org/package=testthat>, 2013. R package version 0.7.1.
- [17] Hadley Wickham. testthat: Get started with testing. *The R Journal*, 3(1):5–10, June 2011. [http://journal.r-project.org/archive/2011-1/RJournal\\_2011-1\\_Wickham.pdf](http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf).
- [18] Paul E. Johnson. Splines and loess and so forth. <http://pj.freefaculty.org/stat/lectures/18-Nonparametric-1-lecture.pdf>, 2012.
- [19] Nist/sematech e-handbook of statistical methods. <http://www.itl.nist.gov/div898/handbook/>.
- [20] 배도선 외. 통계학 이론과 응용. 청문각, 2003.
- [21] 김성수 이기재 이긍희 이태림, 이정진. 통계학개론. 한국방송통신대학교출판부, 2003.
- [22] 박병욱 김우철, 손중권. 수리통계학. 한국방송통신대학교출판부, 2002.
- [23] 김재희. R을 이용한 통계 프로그래밍 기초. 자유아카데미, 2008.
- [24] 안재형. R을 이용한 누구나하는 통계분석. 한나래, 2011.
- [25] 임동훈. R을 이용한 비모수 통계학. 자유아카데미, 2010.
- [26] Julian J. Faraway. *Linear Models with R*. Chapman Hall/CRC, 2004.

- 
- [27] Adrian G. Barnett Annette J. Dobson. *An Introduction to Generalized Linear Model*. Chapman Hall/CRC, 3rd edition, 2008.
- [28] Julian J. Faraway. *Extending the Linear Model with R*. Chapman Hall/CRC, 2006.
- [29] Paul Teator. *R Cookbook*. O'REILLY, 2011.
- [30] 이기재 이계오, 박진우. 표본조사론. 한국방송통신대학교출판부, 1999.
- [31] 박미라 이태림, 이재원. 생명과학 자료분석. 한국방송통신대학교출판부, 2005.
- [32] 문승호 이태림, 허명회. 탐색적 자료분석. 한국방송통신대학교출판부, 2002.
- [33] 강명욱 박성현, 김성수. 회귀분석입문. 한국방송통신대학교출판부, 2008.
- [34] 박성현. 회귀 분석. 민영사, 3판 edition, 1998.
- [35] Ross Ihaka Alan Lee and Chris Triggs. *Advanced Statistical Modelling*. <http://www.stat.auckland.ac.nz/~lee/330/coursebook.pdf>, 2012.
- [36] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.
- [37] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.
- [38] Max Kuhn and Kjell Johnson. *Applied Predictive Modelling*. Springer, 2013.
- [39] Luis Torgo. *Data Mining with R*. Chapman and Hall/CRC, 2010.
- [40] Andrew Ng. Machine learning at coursera. <https://www.coursera.org/course/ml>. [Online; accessed 2013. 10. 1].
- [41] Wikipedia. Feature selection — Wikipedia, the free encyclopedia. [http://en.wikipedia.org/wiki/Feature\\_selection](http://en.wikipedia.org/wiki/Feature_selection), 2013. [Online; accessed 2013. 10. 13].
- [42] Piotr Romanski. Fselector: Selecting attributes. <http://CRAN.R-project.org/package=FSelector>, 2013. R package version 0.19.
- [43] Tom Fawcett. Roc graphs: Notes and practical considerations for researchers. <http://CRAN.R-project.org/package=testthat>, 2004.

- [44] T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer. Rocr: visualizing classifier performance in r. <http://rocr.bioinf.mpi-sb.mpg.de>, 2005.
- [45] Hsuan-Tien Lin Yaser S. Abu-Mostafa, Malik Magdon-Ismail. AMLbook.com, 2012.
- [46] Andreas Alfons. *cvTools: Cross-validation tools for regression models*, 2012. R package version 0.3.2.
- [47] James. H. Martin Daniel Jurafsky. *Speech and Language Processing*. Pearson Education, 2009.
- [48] Elizabeth J. Atkinson Terry M. Therneau and Elizabeth J. Atkinson. An introduction to recursive partitioning using the rpart routines. <http://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf>.
- [49] Torsten Hothorn et al. party: A laboratory for recursive partytioning. <http://cran.r-project.org/web/packages/party/index.html>.
- [50] Leo Breiman and Adele Cutler. Random forests. [http://www.stat.berkeley.edu/~breiman/RandomForests/cc\\_home.htm#ooberr](http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#ooberr).
- [51] Houtao Deng and George C. Runger. Feature selection via regularized trees. *CoRR*, abs/1201.1587, 2012. <http://arxiv.org/abs/1201.1587>.
- [52] Wikipedia. Regularized trees — Wikipedia, the free encyclopedia. [http://en.wikipedia.org/wiki/Feature\\_selection#Regularized\\_trees](http://en.wikipedia.org/wiki/Feature_selection#Regularized_trees), 2013. [Online; accessed 2013. 9. 21].
- [53] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.
- [54] Ingo Feinerer, Kurt Hornik, and David Meyer. Text mining infrastructure in r. *Journal of Statistical Software*, 25(5):1–54, 3 2008.
- [55] Ingo Feinerer. Introduction to tm package. text mining in r. <http://cran.r-project.org/web/packages/tm/vignettes/tm.pdf>.