An Introduction to Categorical Data Analysis *Using R*

Brett Presnell

March 28, 2000

Abstract

This document attempts to reproduce the examples and some of the exercises in *An Introduction to Categorical Data Analysis* [1] using the R statistical programming environment.

Chapter 0

About This Document

This document attempts to reproduce the examples and some of the exercises in *An Introduction to Categorical Data Analysis* [1] using the R statistical programming environment. Numbering and titles of chapters will follow that of Agresti's text, so if a particular example/analysis is of interest, it should not be hard to find, assuming that it is here.

Since R is particularly versatile, there are often a number of different ways to accomplish a task, and naturally this document can only demonstrate a limited number of possibilities. The reader is urged to explore other approaches on their own. In this regard it can be very helpful to read the online documentation for the various functions of R, as well as other tutorials. The help files for many of the R functions used here are also included in the appendix for easy reference, but the online help system is definitely the preferred way to access this information.

It is also worth noting that as of this writing (early 2000), R is still very much under development. Thus new functionality is likely to become available that might be more convenient to use than some of the approaches taken here. Of course any user can also write their own R functions to automate any task, so the possibilities are endless. Do not be intimidated though, for this is really the fun of using R and its best feature: you can teach it to do whatever is neede, instead of being constrained only to what is "built in."

A Note on the Datasets

Often in this document I will show how to enter the data into R as a part the example. However, most of the datasets are avaiable already in R format in the R package for the course, sta4504, available from the course web site. After installing the library on your computer and starting R, you can list the functions and data files available in the package by typing

```
> library(help = sta4504)
> data(package = sta4504)
```

You can make the files in the package to your R session by typing

```
> library(sta4504)
```

and you can read one of the package's datasets into your R session simply by typing, e.g.,

```
> data(deathpen)
```

Chapter 1

Introduction

1.3 Inference for a (Single) Proportion

The function prop.test (appendix A.1.3) will carry out test of hypotheses and produce confidence intervals in problems involving one or several proportions. In the example concerning opinion on abortion, there were 424 "yes" responses out of 950 subjects. Here is one way to use prop.test to analyze these data:

```
> prop.test(424,950)

        1-sample proportions test with continuity correction

data:  424 out of 950, null probability 0.5
X-squared = 10.7379, df = 1, p-value = 0.001050
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.4144634 0.4786078
sample estimates:
       p
0.4463158
```

Note that by default:

- the null hypothesis $\pi = .5$ is tested against the two-sided alternative $\pi \neq .5$;
- a 95% confidence interval for π is calculated; and
- both the test and the CI incorporate a continuity correction.

Any of these defaults can be changed. The call above is equivalent to

```
prop.test(424,950,p=.5,alternative="two.sided",conf.level=0.95,correct=TRUE)
```

Thus, for example, to test the null hypothesis that $\pi = .4$ versus the one-sided alternative $\pi > .4$ and a 99% (*one-sided*) CI for π, all without continuity correction, just type

```
prop.test(424,950,p=.4,alternative="greater",conf.level=0.99,correct=FALSE)
```

Chapter 2

Two-Way Contingency Tables

Entering and Manipulating Data

There are a number of ways to enter counts for a two-way table into R. For a simple concrete example, we consider three different ways of entering the "belief in afterlife" data. Other methods and tools will be introduced as we go along.

Entering Two-Way Tables as a Matrix

One way is to simply enter the data using the matrix function (this is similar to using the array function which we will encounter later). For the "belief in afterlife" example, we might type:

```
> afterlife <- matrix(c(435,147,375,134),nrow=2,byrow=TRUE)
> afterlife
     [,1] [,2]
[1,]  435  147
[2,]  375  134
```

Things are somewhat easier to read if we name the rows and columns:

```
> dimnames(afterlife) <- list(c("Female","Male"),c("Yes","No"))
> afterlife
       Yes  No
Female 435 147
Male   375 134
```

We can dress things even more by providing names for the row and column variables:

```
> names(dimnames(afterlife)) <- c("Gender","Believer")
> afterlife
        Believer
Gender   Yes  No
  Female 435 147
  Male   375 134
```

Calculating the total sample size, n, and the overall proportions, $\{p_{ij}\}$ is easy:

```
> tot <- sum(afterlife)
> tot
[1] 1091
```

```
> afterlife/tot
        Believer
Gender         Yes        No
  Female 0.3987168 0.1347388
  Male   0.3437214 0.1228231
```

To calculate the row and column totals, n_{i+} and n_{+j} and the row and column proportions, p_{i+} and p_{+j}, one can use the apply (appendix A.1.1) and sweep (appendix A.1.4) functions:

```
> rowtot <- apply(afterlife,1,sum)
> coltot <- apply(afterlife,2,sum)
> rowtot
Female   Male
   582    509
> coltot
Yes  No
810 281
> rowpct <- sweep(afterlife,1,rowtot,"/")
> rowpct
        Believer
Gender         Yes        No
  Female 0.7474227 0.2525773
  Male   0.7367387 0.2632613
> round(rowpct,3)
        Believer
Gender     Yes    No
  Female 0.747 0.253
  Male   0.737 0.263
> sweep(afterlife,2,coltot,"/")
        Believer
Gender        Yes        No
  Female 0.537037 0.5231317
  Male   0.462963 0.4768683
```

Entering Two-Way Tables as a Data Frame

One might also put the data into a data frame, treating the row and column variables as factor variables. This approach is actually be more convenient when the data is stored in a separate file to be read into R, but we will consider it now anyway.

```
> Gender <- c("Female","Female","Male","Male")
> Believer <- c("Yes","No","Yes","No")
> Count <- c(435,147,375,134)
> afterlife <- data.frame(Gender,Believer,Count)
> afterlife
  Gender Believer Count
1 Female      Yes   435
2 Female       No   147
3   Male      Yes   375
4   Male       No   134
> rm(Gender, Believer, Count) # No longer needed
```

As mentioned above, you can also just enter the data into a text file to be read into R using the read.table command. For example, if the file `afterlife.dat` contained the lines

```
Gender Believer Count
Female Yes      435
Female No       147
Male   Yes      375
Male   No       134
```

then the command

```
> read.table("afterlife.dat",header=TRUE)
```

would get you to the same point as above.[1]

To extract a contingency table (a matrix in this case) for these data, you can use the tapply (appendix A.1.5) function in the following way:

```
> attach(afterlife) # attach the data frame
> beliefs <- tapply(Count,list(Gender,Believer),c)
> beliefs
        No Yes
Female 147 435
Male   134 375
> detach(afterlife) # can detach the data when longer needed
> names(dimnames(beliefs)) <- c("Gender","Believer")
> beliefs
        Believer
Gender    No Yes
  Female 147 435
  Male   134 375
> beliefs <- beliefs[,c(2,1)]  # reverse the columns?
> beliefs
        Believer
Gender   Yes  No
  Female 435 147
  Male   375 134
```

At this stage, `beliefs` can be manipulated as in the previous subsection.

2.3 Comparing Proportions in Two-by-Two Tables

As explained by the documentation for prop.test (appendix A.1.3), the data may be represented in several different ways for use in prop.test. We will use the matrix representation of the last section in examining the Physician's Health Study example.

```
> phs <- matrix(c(189,10845,104,10933),byrow=TRUE,ncol=2)
> phs
     [,1]  [,2]
[1,]  189 10845
[2,]  104 10933
> dimnames(phs) <-
+ list(Group=c("Placebo","Aspirin"),MI=c("Yes","No"))
> phs
```

[1] Actually, there is one small difference: the levels of the factor "Believer" will be ordered alphabetically, and this will make a small difference in how some things are presented. If you want to make sure that the levels of the factors are ordered as they appear in the data file, you can use the read.table2 function provided in the sta4504 package for R. Or use the relevel command.

```
         MI
Group     Yes    No
  Placebo 189 10845
  Aspirin 104 10933
> prop.test(phs)

   2-sample test for equality of proportions
   with continuity correction

data:  phs
X-squared = 24.4291, df = 1, p-value = 7.71e-07
alternative hypothesis: two.sided
95 percent confidence interval:
 0.004597134 0.010814914
sample estimates:
    prop 1     prop 2
0.01712887 0.00942285
```

A continuity correction is used by default, but it makes very little difference in this example:

```
> prop.test(phs,correct=F)

   2-sample test for equality of proportions
   without continuity correction

data:  phs
X-squared = 25.0139, df = 1, p-value = 5.692e-07
alternative hypothesis: two.sided
95 percent confidence interval:
 0.004687751 0.010724297
sample estimates:
    prop 1     prop 2
0.01712887 0.00942285
```

You can also save the output of the test and manipulate it in various ways:

```
> phs.test <- prop.test(phs)
> names(phs.test)
[1] "statistic"   "parameter"   "p.value"     "estimate"
[5] "null.value"  "conf.int"    "alternative" "method"
[9] "data.name"
> phs.test$estimate
    prop 1     prop 2
0.01712887 0.00942285
> phs.test$conf.int
[1] 0.004597134 0.010814914
attr(,"conf.level")
[1] 0.95
> round(phs.test$conf.int,3)
[1] 0.005 0.011
attr(,"conf.level")
[1] 0.95
> phs.test$estimate[1]/phs.test$estimate[2] % relative risk
```

```
  prop 1
1.817802
```

2.4 The Odds Ratio

Relative risk and the odds ratio are easy to calculate (you can do it in lots of ways of course):

```
> phs.test$estimate
    prop 1     prop 2
0.01712887 0.00942285
> odds <- phs.test$estimate/(1-phs.test$estimate)
> odds
     prop 1      prop 2
0.017427386 0.009512485
> odds[1]/odds[2]
  prop 1
1.832054
> (phs[1,1]*phs[2,2])/(phs[2,1]*phs[1,2]) # as cross-prod ratio
[1] 1.832054
```

Here's one way to calculate the CI for the odds ratio:

```
> theta <- odds[1]/odds[2]
> ASE <- sqrt(sum(1/phs))
> ASE
[1] 0.1228416
> logtheta.CI <- log(theta) + c(-1,1)*1.96*ASE
> logtheta.CI
[1] 0.3646681 0.8462073
> exp(logtheta.CI)
[1] 1.440036 2.330790
```

It is easy to write a quick and dirty function to do these calculations for a 2×2 table.

```
odds.ratio <-
  function(x, pad.zeros=FALSE, conf.level=0.95) {
    if (pad.zeros) {
      if (any(x==0)) x <- x + 0.5
    }
    theta <- x[1,1] * x[2,2] / ( x[2,1] * x[1,2] )
    ASE <- sqrt(sum(1/x))
    CI <- exp(log(theta)
              + c(-1,1) * qnorm(0.5*(1+conf.level)) *ASE )
    list(estimator=theta,
         ASE=ASE,
         conf.interval=CI,
         conf.level=conf.level)
  }
```

This has been added to the sta4504 package. For the example above:

```
> odds.ratio(phs)
$estimator
```

```
[1] 1.832054

$ASE
[1] 0.1228416

$conf.interval
[1] 1.440042 2.330780

$conf.level
[1] 0.95
```

2.5 Chi-Squared Tests of Independence

Gender Gap Example The chisq.test function will compute Pearson's chi-squared test statistic (X^2) and the corresponding P-value. Here it is applied to the gender gap example:

```
> gendergap <- matrix(c(279,73,225,165,47,191),byrow=TRUE,nrow=2)
> dimnames(gendergap) <- list(Gender=c("Females","Males"),
+ PartyID=c("Democrat","Independent","Republican"))
> gendergap
         PartyID
Gender    Democrat Independent Republican
  Females      279          73        225
  Males        165          47        191
> chisq.test(gendergap)

         Pearson's Chi-square test

data:  gendergap
X-squared = 7.0095, df = 2, p-value = 0.03005
```

In case you are worried about the chi-squared approximation to the sampling distribution of the statistic, you can use simulation to compute an approximate P-value (or use an exact test; see below). The argument B (default is 2000) controls how many simulated tables are used to compute this value. More is better, but eventually you will run out of either compute memory or time, so don't get carried away. It is interesting to do it a few times though to see how stable the simulated P-value is (does it change much from run to run). In this case the simulated P-values agree closely with the chi-squared approximation, suggesting that the chi-squared approximation is good in this example.

```
> chisq.test(gendergap,simulate.p.value=TRUE,B=10000)

         Pearson's Chi-square test with simulated p-value
         (based on 10000 replicates)

data:  gendergap
X-squared = 7.0095, df = NA, p-value = 0.032

> chisq.test(gendergap,simulate.p.value=TRUE,B=10000)

         Pearson's Chi-square test with simulated p-value
         (based on 10000 replicates)
```

```
data:  gendergap
X-squared = 7.0095, df = NA, p-value = 0.0294
```

An exact test of independence in $I \times J$ tables is implemented in the function fisher.test of the ctest (classical tests) package (this package is now part of the base distribution of R and is loaded automatically when any of its functions are called). This test is just a generalization of Fisher's exact test for 2×2 tables. Note that the P-value here is in pretty good agreement with the simulated values and the chi-squared approximation.

```
> library(ctest)  # this is not needed with R versions >= 0.99
> fisher.test(gendergap)

         Fisher's Exact Test for Count Data

data:  gendergap
p-value = 0.03115
alternative hypothesis: two.sided
```

Job Satisfaction Example For the job satisfaction example given in class, there was some worry about the chi-squared approximation to the null distribution of the test statistic. However the P-value again agrees closely with the simulated P-values and P-value for the the exact test:

```
> jobsatis <- c(2,4,13,3, 2,6,22,4, 0,1,15,8, 0,3,13,8)
> jobsatis <- matrix(jobsatis,byrow=TRUE,nrow=4)
> dimnames(jobsatis) <- list(
+ Income=c("<5","5-15","15-25",">25"),
+ Satisfac=c("VD","LS","MS","VS"))
> jobsatis
       Satisfac
Income  VD LS MS VS
  <5     2  4 13  3
  5-15   2  6 22  4
  15-25  0  1 15  8
  >25    0  3 13  8
> chisq.test(jobsatis)

         Pearson's Chi-square test

data:  jobsatis
X-squared = 11.5243, df = 9, p-value = 0.2415

Warning message:
Chi-square approximation may be incorrect in: chisq.test(jobsatis)
> chisq.test(jobsatis,simulate.p.value=TRUE,B=10000)

         Pearson's Chi-square test with simulated p-value
         (based on 10000 replicates)

data:  jobsatis
X-squared = 11.5243, df = NA, p-value = 0.2408

> fisher.test(jobsatis)
```

```
        Fisher's Exact Test for Count Data

data:  jobsatis
p-value = 0.2315
alternative hypothesis: two.sided
```

A "PROC FREQ" for R Here is a little R function to do some of the calculations that SAS's PROC FREQ does. There are other ways to get all of this information, so the main idea is simply to illustrate how you can write R functions to do the sorts of calculations that you might find yourself doing repeatedly. Also, you can always go back later and modify your function add capabilities that you need. Note that this is just supposed to serve as a simple utility function. If I wanted it to be really nice, I would write a general method function and a print method for the output (you can also find source for this function on the course web page).

```
"procfreq" <-
  function(x, digits=4) {
    total <- sum(x)
    rowsum <- apply(x,1,sum)
    colsum <- apply(x,2,sum)
    prop <- x/total
    rowprop <- sweep(x,1,rowsum,"/")
    colprop <- sweep(x,2,colsum,"/")
    expected <- (matrix(rowsum) %*% t(matrix(colsum))) / total
    dimnames(expected) <- dimnames(x)
    resid <- (x-expected)/sqrt(expected)
    adj.resid <- resid /
      sqrt((1-matrix(rowsum)/total) %*% t(1-matrix(colsum)/total))
    df <- prod(dim(x)-1)
    X2 <- sum(resid^2)
    attr(X2,"P-value") <- 1-pchisq(X2,df)
    ## Must be careful about zero freqencies.  Want 0*log(0) = 0.
    tmp <- x*log(x/expected)
    tmp[x==0] <- 0
    G2 <- 2 * sum(tmp)
    attr(G2,"P-value") <- 1-pchisq(G2,df)
    list(sample.size=total,
         row.totals=rowsum,
         col.totals=colsum,
         overall.proportions=prop,
         row.proportions=rowprop,
         col.proportions=colprop,
         expected.freqs=expected,
         residuals=resid,
         adjusted.residuals=adj.resid,
         chi.square=X2,
         likelihood.ratio.stat=G2,
         df=df)
  }
```

If you save this function definition in a file called "procfreq.R" and then "source" it into R, you can use it just like any built-in function. Here is procfreq in action on the income data:

```
> source("procfreq.R")
> jobsat.freq <- procfreq(jobsatis)
> names(jobsat.freq)
 [1] "sample.size"           "row.totals"
 [3] "col.totals"            "overall.proportions"
 [5] "row.proportions"       "col.proportions"
 [7] "expected.freqs"        "residuals"
 [9] "adjusted.residuals"    "chi.square"
[11] "likelihood.ratio.stat" "df"
> jobsat.freq$expected
       Satisfac
Income         VD       LS       MS       VS
  <5    0.8461538 2.961538 13.32692 4.865385
  5-15  1.3076923 4.576923 20.59615 7.519231
  15-25 0.9230769 3.230769 14.53846 5.307692
  >25   0.9230769 3.230769 14.53846 5.307692
> round(jobsat.freq$adjusted.residuals,2)
       Satisfac
Income     VD    LS    MS    VS
  <5     1.44  0.73 -0.16 -1.08
  5-15   0.75  0.87  0.60 -1.77
  15-25 -1.12 -1.52  0.22  1.51
  >25   -1.12 -0.16 -0.73  1.51
> jobsat.freq$chi.square
[1] 11.52426
attr(,"P-value")
[1] 0.2414764
> jobsat.freq$likelihood.ratio.stat
[1] 13.46730
attr(,"P-value")
[1] 0.1425759
```

Fisher's Exact Test As mentioned above, Fisher's exact test is implemented as fisher.test in the ctest (classical tests) package. Here is the tea tasting example in R. Note that the default is to test the two-sided alternative.

```
> library(ctest) # not needed with R versions >= 0.99
> tea <- matrix(c(3,1,1,3),ncol=2)
> dimnames(tea) <-
+ list( Poured=c("milk","tea"), Guess=c("milk","tea"))
> tea
      Guess
Poured milk tea
  milk    3   1
  tea     1   3
> fisher.test(tea)

         Fisher's Exact Test for Count Data

data:  tea
p-value = 0.4857
```

```
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
   0.2117329 621.9337505
sample estimates:
odds ratio
  6.408309

> fisher.test(tea,alternative="greater")

         Fisher’s Exact Test for Count Data

data:  tea
p-value = 0.2429
alternative hypothesis: true odds ratio is greater than 1
95 percent confidence interval:
 0.3135693       Inf
sample estimates:
odds ratio
  6.408309
```

Chapter 3

Three-Way Contingency Tables

The Cochran-Mantel-Haenszel test is implemented in the mantelhaen.test function of the ctest library.

The Death Penalty Example Here we illustrate the use of mantelhaen.test as well as the ftable function to present a multiway contigency table in a "flat" format. Both of these are included in base R as of version 0.99. Note that by default mantelhaen.test applies a continuity correction in doing the test.

```
> dp <- c(53, 414, 11, 37, 0, 16, 4, 139)
> dp <- array(dp, dim=c(2,2,2))
> dimnames(dp) <- list(DeathPen=c("yes","no"),
+ Defendant=c("white","black"), Victim=c("white","black"))
> dp
, , Victim = white

        Defendant
DeathPen white black
     yes    53    11
     no    414    37

, , Victim = black

        Defendant
DeathPen white black
     yes     0     4
     no     16   139

> ftable(dp, row.vars=c("Victim","Defendant"), col.vars="DeathPen")
                 DeathPen yes  no
Victim Defendant
white  white               53 414
       black               11  37
black  white                0  16
       black                4 139
> mantelhaen.test(dp)

      Mantel-Haenszel chi-square test with continuity correction
```

```
data:  dp
Mantel-Haenszel X-square = 4.779, df = 1, p-value = 0.02881

> mantelhaen.test(dp,correct=FALSE)

      Mantel-Haenszel chi-square test without continuity correction

data:  dp
Mantel-Haenszel X-square = 5.7959, df = 1, p-value = 0.01606
```

Smoking and Lung Cancer in China Example This is a bigger example that uses the Cochran-Mantel-Haenszel test. First we will enter the data as a "data frame" instead of as an array as in the previous example. This is mostly just to demonstrate another way to do things.

```
> cities <- c("Beijing","Shanghai","Shenyang","Nanjing","Harbin",
+ "Zhengzhou","Taiyuan","Nanchang")
> City <- factor(rep(cities,rep(4,length(cities))),levels=cities)
> Smoker <-
+ factor(rep(rep(c("Yes","No"),c(2,2)),8),levels=c("Yes","No"))
> Cancer <- factor(rep(c("Yes","No"),16),levels=c("Yes","No"))
> Count <- c(126,100,35,61,908,688,497,807,913,747,336,598,235,
+ 172,58,121,402,308,121,215,182,156,72,98,60,99,11,43,104,89,21,36)
> chismoke <- data.frame(City,Smoker,Cancer,Count)
> chismoke
        City Smoker Cancer Count
1    Beijing    Yes    Yes   126
2    Beijing    Yes     No   100
3    Beijing     No    Yes    35
4    Beijing     No     No    61
5   Shanghai    Yes    Yes   908
6   Shanghai    Yes     No   688
7   Shanghai     No    Yes   497
8   Shanghai     No     No   807
9   Shenyang    Yes    Yes   913
10  Shenyang    Yes     No   747
11  Shenyang     No    Yes   336
12  Shenyang     No     No   598
13   Nanjing    Yes    Yes   235
14   Nanjing    Yes     No   172
15   Nanjing     No    Yes    58
16   Nanjing     No     No   121
17    Harbin    Yes    Yes   402
18    Harbin    Yes     No   308
19    Harbin     No    Yes   121
20    Harbin     No     No   215
21 Zhengzhou    Yes    Yes   182
22 Zhengzhou    Yes     No   156
23 Zhengzhou     No    Yes    72
24 Zhengzhou     No     No    98
25   Taiyuan    Yes    Yes    60
26   Taiyuan    Yes     No    99
```

```
27   Taiyuan     No    Yes    11
28   Taiyuan     No     No    43
29  Nanchang    Yes    Yes   104
30  Nanchang    Yes     No    89
31  Nanchang     No    Yes    21
32  Nanchang     No     No    36
> rm(cities,City,Smoker,Cancer,Count) # Cleaning up
```

Alternatively, we can read the data directly from the file `chismoke.dat`. Note that if we want "Yes" before "No" we have to relevel the factors, because read.table puts the levels in alphabetical order.

```
> chismoke <- read.table("chismoke.dat",header=TRUE)
> names(chismoke)
[1] "City"   "Smoker" "Cancer" "Count"
> levels(chismoke$Smoker)
[1] "No"  "Yes"
> chismoke$Smoker <- relevel(chismoke$Smoker,c("Yes","No"))
> levels(chismoke$Smoker)
[1] "Yes" "No"
> levels(chismoke$Cancer)
[1] "No"  "Yes"
> chismoke$Cancer <- relevel(chismoke$Cancer,c("Yes","No"))
> levels(chismoke$Cancer)
[1] "Yes" "No"
```

If you use the function read.table2 in the sta4504 package, you will not have to relevel the factors. Of course if you have the package, then

Now, returning to the example:

```
> attach(chismoke)
> x <- tapply(Count,list(Smoker,Cancer,City),c)
> detach(chismoke)
> names(dimnames(x)) <- c("Smoker","Cancer","City")
> # ftable will be in the next release of R
> ftable(x,row.vars=c("City","Smoker"),col.vars="Cancer")
                Cancer Yes  No
City      Smoker
Beijing   Yes           126 100
          No             35  61
Shanghai  Yes           908 688
          No            497 807
Shenyang  Yes           913 747
          No            336 598
Nanjing   Yes           235 172
          No             58 121
Harbin    Yes           402 308
          No            121 215
Zhengzhou Yes           182 156
          No             72  98
Taiyuan   Yes            60  99
```

```
          No             11  43
Nanchang  Yes           104  89
          No             21  36
> ni.k <- apply(x,c(1,3),sum)
> ni.k
      City
Smoker Beijing Shanghai Shenyang Nanjing Harbin Zhengzhou
   Yes     226     1596     1660     407    710       338
   No       96     1304      934     179    336       170
      City
Smoker Taiyuan Nanchang
   Yes     159      193
   No       54       57
> n.jk <- apply(x,c(2,3),sum)
> n.jk
      City
Cancer Beijing Shanghai Shenyang Nanjing Harbin Zhengzhou
   Yes     161     1405     1249     293    523       254
   No      161     1495     1345     293    523       254
      City
Cancer Taiyuan Nanchang
   Yes      71      125
   No      142      125
> n..k <- apply(x,3,sum)
> mu11k <- ni.k[1,] * n.jk[1,] / n..k
> mu11k
  Beijing  Shanghai  Shenyang   Nanjing    Harbin Zhengzhou
 113.0000  773.2345  799.2830  203.5000  355.0000  169.0000
  Taiyuan  Nanchang
  53.0000   96.5000
> sum(mu11k)
[1] 2562.517
> sum(x[1,1,])
[1] 2930
> varn11k <- ni.k[1,]*ni.k[2,]*n.jk[1,]*n.jk[2,] / (n..k^2 * (n..k-1))
> sum(varn11k)
[1] 482.0612
>
> MH <- (sum(x[1,1,]-mu11k))^2/sum(varn11k)
> MH
[1] 280.1375
```

Chapter 4

Chapter 4: Generalized Linear Models

Snoring and Heart Disease This covers the example in Section 4.2.2 and also Exercise 4.2. There are several ways to fit a logistic regression in R using the glm function (more on this in Chapter 5). In the method illustrated here, the response in the model formula (e.g., `snoring` in `snoring ~ scores.a`) is a matrix whose first column is the number of "successes" and whose second column is the number of "failures" for each observed binomial.

```
> snoring <-
+ matrix(c(24,1355,35,603,21,192,30,224), ncol=2, byrow=TRUE)
> dimnames(snoring) <-
+ list(snore=c("never","sometimes","often","always"),
+      heartdisease=c("yes","no"))
> snoring
           heartdisease
snore       yes   no
  never      24 1355
  sometimes  35  603
  often      21  192
  always     30  224
> scores.a <- c(0,2,4,5)
> scores.b <- c(0,2,4,6)
> scores.c <- 0:3
> scores.d <- 1:4
> # Fitting and comparing logistic regression models
> snoring.lg.a <- glm( snoring ~ scores.a, family=binomial() )
> snoring.lg.b <- glm( snoring ~ scores.b, family=binomial() )
> snoring.lg.c <- glm( snoring ~ scores.c, family=binomial() )
> snoring.lg.d <- glm( snoring ~ scores.d, family=binomial() )
> coef(snoring.lg.a)
(Intercept)    scores.a
 -3.8662481   0.3973366
> coef(snoring.lg.b)
(Intercept)    scores.b
 -3.7773755   0.3272648
> coef(snoring.lg.c)
(Intercept)    scores.c
 -3.7773755   0.6545295
> coef(snoring.lg.d)
```

```
(Intercept)    scores.d
 -4.4319050   0.6545295
> predict(snoring.lg.a, type="response")  # compare to table 4.1
[1] 0.02050742 0.04429511 0.09305411 0.13243885
> predict(snoring.lg.b, type="response")
[1] 0.02237077 0.04217466 0.07810938 0.14018107
> predict(snoring.lg.c, type="response")
[1] 0.02237077 0.04217466 0.07810938 0.14018107
> predict(snoring.lg.d, type="response")
[1] 0.02237077 0.04217466 0.07810938 0.14018107
```

Note that the default link function with the binomial family is the logit link. To do a probit analysis, say using the original scores used in Table 4.1:

```
> snoring.probit <-
+ glm( snoring ~ scores.a, family=binomial(link="probit") )
> summary(snoring.probit)

Call:
glm(formula = snoring ~ scores.a, family = binomial(link = "probit"))

Deviance Residuals:
[1]  -0.6188   1.0388   0.1684  -0.6175

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.06055    0.07017 -29.367  < 2e-16 ***
scores.a     0.18777    0.02348   7.997 1.28e-15 ***
---
Signif. codes:
  0  ‘***’  0.001  ‘**’  0.01  ‘*’  0.05  ‘.’  0.1  ‘ ’  1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 65.9045  on 3  degrees of freedom
Residual deviance:  1.8716  on 2  degrees of freedom
AIC: 26.124

Number of Fisher Scoring iterations: 3

> predict(snoring.probit, type="response") # compare with Table 4.1
[1] 0.01967292 0.04599325 0.09518762 0.13099512
```

There is no identity link provided for the binomial family, so we cannot reproduce the third fit given in Table 4.1. This is not such a great loss of course, since linear probability models are rarely used.

Grouped Crabs Data This is the example done in class (slightly different from that done in the text. The data are in the file “crabs.dat” (available on the course web site) and can be read into R using the read.table function:

```
> crabs <- read.table("crabs.dat",header=TRUE)
```

Alternatively, these data can accessed directly from the sta4504 package by typing

```
> library(sta4504)
> data(crabs)
```

By the way, you will notice that these data look slightly different from those given in Table 4.2 (pp. 82–3) of the text. This is because here the color codes go from 2 to 5 (they are one more than the codes used in the text), and the weights are recorded in grams, not kilograms.

For this analysis we want to create a grouped version of the data by dividing the female crabs into weight categories. Again the grouped data is avaible already from the sta4504 package (data(crabsgp)), but here is how they were created from the raw data, in case you are interested.

```
> attach(crabs)
> table(cut(weight,
+ breaks= c(0,1775,2025,2275,2525,2775,3025,Inf),
+ dig.lab=4))

   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
         16          31          30          22          24
(2775,3025]  (3025,Inf]
         21          29
> grp <- cut(weight,
+ breaks= c(0,1775,2025,2275,2525,2775,3025,Inf),
+ dig.lab=4)
> cases <- table(grp)
> avgwt <- tapply(weight,grp,mean)
> avgwt
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
   1526.562    1917.484    2174.167    2372.727    2642.708
(2775,3025]  (3025,Inf]
   2900.810    3310.345
> avgwt <- round(avgwt/1000,2)
> avgwt
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
       1.53        1.92        2.17        2.37        2.64
(2775,3025]  (3025,Inf]
       2.90        3.31
> totsat <- tapply(satell,grp,sum)
> totsat
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
         13          55          72          78          61
(2775,3025]  (3025,Inf]
         86         140
> crabsgp <-
+ data.frame(cbind(weight=avgwt,cases=cases,satell=totsat))
> crabsgp
            weight cases satell
(0,1775]      1.53    16     13
(1775,2025]   1.92    31     55
(2025,2275]   2.17    30     72
(2275,2525]   2.37    22     78
(2525,2775]   2.64    24     61
(2775,3025]   2.90    21     86
(3025,Inf]    3.31    29    140
```

```
> rm(cases,avgwt,totsat) # cleaning up
> detach(crabs)
```

To fit the loglinear model on these data, we just use the glm command. Here the first argument is the model formula for the fit, the second is the offset, and the third specifies the random component for the GLM. Note the log-link is the default for the Poisson family. Finally "data=crabsgp" tells the function that the data will be taken from the data frame crabsgp.

```
> crabsgp.glm1 <-
+ glm(satell ~ weight,offset=log(cases),family=poisson,
+ data=crabsgp)
```

The results can be viewed in a number of ways. Perhaps most importantly there is the summary function:

```
> summary(crabsgp.glm1)

Call:
glm(formula = satell ~ weight, family = poisson, data = crabsgp,
    offset = log(cases))

Deviance Residuals:
   (0,1775]  (1775,2025]  (2025,2275]  (2275,2525]
    -2.1285      -0.3081       0.6564       2.7036
(2525,2775]  (2775,3025]   (3025,Inf]
    -1.6776       0.7307      -0.6560

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.7870     0.2229   -3.53 0.000415 ***
weight        0.7300     0.0823    8.87  < 2e-16 ***
---
Signif. codes:
  0  '***'  0.001  '**'  0.01  '*'  0.05  '.'  0.1  ' '  1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 96.312  on 6  degrees of freedom
Residual deviance: 16.144  on 5  degrees of freedom
AIC: 61.748

Number of Fisher Scoring iterations: 3
```

Note that summary includes the Wald test statistic, $z = 8.87$ and the associated P-value for the two-tailed test, 2×10^{-16} (this is close to the limits of machine accuracy, so let's just say the P-value is "very close to zero"). It also shows the deviance (labelled residual deviance) and its associated degrees of freedom. The "null deviance" is the deviance for the "intercept only" model, so the likelihood ratio test can be carried out just by differencing these two deviances and their degrees of freedom and referring to a chi-square distribution:

```
> 96.312 - 16.144
[1] 80.168
> 1 - pchisq(96.312 - 16.144, 1)
[1] 0
```

Yes, that's a very small P-value again. Finally, AIC is the Akaike Information Criterion, which is often used for model selection (the smaller the better for nested models).

A better way to get the results of the likelihood ratio test is to use the anova function, which prints a sort of ANOVA table:

```
> anova(crabsgp.glm1,test="Chisq")
Analysis of Deviance Table

Model: poisson, link: log

Response: satell

Terms added sequentially (first to last)


       Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                       6     96.312
weight  1   80.168         5     16.144     0.000
```

There is another approach to getting the results of the likelihood ratio test that generalizes better to more complicated model comparisons. This involves fitting the null model and then comparing models using the anova function. The null model can either be fit as before using the glm function or using the update function. We will show the latter here. Note that the intercept only model is denoted by `satell ~ 1`.

```
> crabsgp.glm0 <- update(crabsgp.glm1, satell ~ 1)
> crabsgp.glm0

Call:  glm(formula = satell ~ 1,
           family = poisson, data = crabsgp, offset = log(cases))

Coefficients:
(Intercept)
      1.071

Degrees of Freedom: 6 Total (i.e. Null);  6 Residual
Null Deviance:      96.31
Residual Deviance: 96.31        AIC: 139.9
> summary(crabsgp.glm1)

Call:
glm(formula = satell ~ weight, family = poisson, data = crabsgp,
    offset = log(cases))

Deviance Residuals:
   (0,1775]  (1775,2025]  (2025,2275]  (2275,2525]  (2525,2775]
    -2.1285      -0.3081       0.6564       2.7036      -1.6776
(2775,3025]   (3025,Inf]
     0.7307      -0.6560

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.7870     0.2229   -3.53 0.000415 ***
```

```
weight        0.7300     0.0823    8.87  < 2e-16 ***
---
Signif. codes:  0  ‘***’  0.001  ‘**’  0.01  ‘*’  0.05  ‘.’  0.1  ‘ ’  1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 96.312  on 6  degrees of freedom
Residual deviance: 16.144  on 5  degrees of freedom
AIC: 61.748

Number of Fisher Scoring iterations: 3

> anova(crabsgp.glm0,crabsgp.glm1)
Analysis of Deviance Table

Response: satell


       Resid. Df Resid. Dev Df Deviance
1              6     96.312
weight         5     16.144  1   80.168
```

The residuals function will extract the unadjusted Pearson (or chi-square) and deviance residuals (the default) from a fitted glm object. There seems to be no built in function for computing the adjusted, or standardized residuals, but lm.influence will extract the diagonal of the so called “hat matrix”, and this is enough to construct the adjusted residuals:

```
> round(residuals(crabsgp.glm1),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -2.13       -0.31        0.66        2.70       -1.68
(2775,3025]  (3025,Inf]
       0.73       -0.66
> round(residuals(crabsgp.glm1, type="pearson"),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -1.96       -0.31        0.67        2.86       -1.62
(2775,3025]  (3025,Inf]
       0.74       -0.65
> h <- lm.influence(crabsgp.glm1)$hat
> round(residuals(crabsgp.glm1, type="deviance")/sqrt(1-h),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -2.43       -0.37        0.75        2.92       -1.82
(2775,3025]  (3025,Inf]
       0.81       -1.25
> round(residuals(crabsgp.glm1, type="pearson")/sqrt(1-h),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -2.24       -0.37        0.76        3.09       -1.76
(2775,3025]  (3025,Inf]
       0.82       -1.24
```

Of course you could easily write a simple function to automate this process:

```
adj.residuals <-
  function(fit, ...) {
```

```
    residuals(fit, ...) / sqrt(1 - lm.influence(fit)$hat)
  }
```

Now the above computations become

```
> round(adj.residuals(crabsgp.glm1),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -2.43       -0.37        0.75        2.92       -1.82
(2775,3025]  (3025,Inf]
n       0.81       -1.25
> round(adj.residuals(crabsgp.glm1,type="pearson"),2)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
      -2.24       -0.37        0.76        3.09       -1.76
(2775,3025]  (3025,Inf]
       0.82       -1.24
```

Finally, here's how to calculate the group means and variances discussed in lecture when examining overdispersion. Here I use the tapply function, which applies a function (here the mean and var functions) to a variable (`satell`) after grouping according to some factor(s) (`grp`). Recall that the variable `grp` was created ealier. The predict function gives the actual fitted values from the model, and here we need to divide this by the number of cases in each group to get the fitted "rates". Note that I attach the `crabs` data frame to make its variables local (otherwise I would have to type `crabs$satell` for example).

```
> attach(crabs) # Makes the variables in the data frame local
> tapply(satell,grp,mean)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
   0.812500    1.774194    2.400000    3.545455    2.541667
(2775,3025]  (3025,Inf]
   4.095238    4.827586
> tapply(satell,grp,var)
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
   1.895833    7.313978    7.972414   12.545455    6.432971
(2775,3025]  (3025,Inf]
  12.490476   10.647783
> predict(crabsgp.glm1,type="response")
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
   22.25331    57.31630    66.57256    56.49405    75.05670
(2775,3025]  (3025,Inf]
   79.40089   147.90622
> predict(crabsgp.glm1,type="response")/crabsgp$cases
   (0,1775] (1775,2025] (2025,2275] (2275,2525] (2525,2775]
   1.390832    1.848913    2.219085    2.567911    3.127362
(2775,3025]  (3025,Inf]
   3.780995    5.100214
```

So here's how to get R to print the table from class:

```
> round(cbind(
+ weight=crabsgp$weight,
+ fitted.mean=predict(crabsgp.glm1,type="response")/crabsgp$cases,
+ sample.mean=tapply(satell,grp,mean),
+ sample.variance=tapply(satell,grp,var)),
+ 2)
```

```
            weight fitted.mean sample.mean sample.variance
(0,1775]      1.53        1.39        0.81            1.90
(1775,2025]   1.92        1.85        1.77            7.31
(2025,2275]   2.17        2.22        2.40            7.97
(2275,2525]   2.37        2.57        3.55           12.55
(2525,2775]   2.64        3.13        2.54            6.43
(2775,3025]   2.90        3.78        4.10           12.49
(3025,Inf]    3.31        5.10        4.83           10.65
> detach(crabs)
```

Ungrouped Analysis of Crab Data Here's the single predictor Poisson loglinear model fit to the ungrouped crab data.

```
> crabs.glm1 <- glm(satell ~ weight, family=poisson(), data=crabs)
> summary(crabs.glm1)

Call:
glm(formula = satell ~ weight, family = poisson(), data = crabs)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.9307  -1.9981  -0.5627   0.9298   4.9992

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -4.284e-01  1.789e-01  -2.394   0.0167 *
weight       5.893e-04  6.502e-05   9.064   <2e-16 ***
---
Signif. codes:  0  '***'  0.001  '**'  0.01  '*'  0.05  '.'  0.1  ' '  1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 632.79  on 172  degrees of freedom
Residual deviance: 560.87  on 171  degrees of freedom
AIC: 920.16

Number of Fisher Scoring iterations: 5

> anova(crabs.glm1,test="Chisq")
Analysis of Deviance Table

Model: poisson, link: log

Response: satell

Terms added sequentially (first to last)


        Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                      172     632.79
weight   1    71.93       171     560.87     0.00
```

A Plot Here's how I produced the plot given in class. The main thing going on is the use of the predict function to get the predicted mean for the model on a grid of x values. Note that I have to be careful about the fact that weight is measured in kilograms in one data set and in grams in the other.

```
> x <- seq(1.2,5.2,length=100)
> gp.muhat <-
+ predict(crabsgp.glm1,
+ newdata=data.frame(weight=x,cases=1),type="response")
> muhat <-
+ predict(crabs.glm1,
+ newdata=data.frame(weight=1000*x),type="response")
> grp.means <- tapply(satell,grp,mean)
> plot(crabsgp$weight,grp.means,
+ xlab="Weight", ylab="Number of Satellites")
> lines(x,gp.muhat)
> lines(x,muhat,lty=2) # $
```

Chapter 5

Chapter 5: Logistic Regression

There are three ways to fit a binomial regression in R using glm(). Following *An Introduction to R* [2] and Venables and Ripley [3]:

- The response may be a vector of binary (0/1) or logical responses. In this case it is easily handled similarly to fitting any other GLM.
- If the response is a a numeric vector representing proportions of successes, then the number of trials for the proportions must be given as a vector of weights using the `weights` argument.
- If the response is a two column matrix it is assumed that the first column holds the number of successes for the trials and the second holds the number of failures. In this case no weights argument is required.

Three link functions (`logit`, `probit`, and `cloglog` (complementary log-log, an asymmetric link function) are provided. Of course in this chapter we are concerned with the logit link.

Snoring and Heart Disease Revisited Here's another look at the snoring example from the last chapter, just to illustrate two of the three ways of carrying out a logistic regression with such data. Here we initially enter the data as a data frame instead of a matrix, with scores (0,2,4,5) representing the levels of snoring:

```
> snoring <- data.frame(
+ snore = c(0,2,4,5),
+ heartdisyes = c(24,35,21,30),
+ n = c(1379,638,213,254) )
> snoring
  snore heartdisyes    n
1     0          24 1379
2     2          35  638
3     4          21  213
4     5          30  254
```

Here is how we might fit a logistic regression model using the proportion of cases with heart disease as the response and the number of cases as the weights:

```
> snoring.lg <-
+ glm(heartdisyes/n ~ snore, weights=n, family=binomial(), data=snoring)
> snoring.lg

Call:  glm(formula = heartdisyes/n ~ snore, family = binomial(),
           data = snoring, weights = n)
```

```
Coefficients:
(Intercept)        snore
    -3.8662       0.3973

Degrees of Freedom: 3 Total (i.e. Null);  2 Residual
Null Deviance:      65.9
Residual Deviance: 2.809        AIC: 841.7
```

To fit the model with a matrix of success and failure counts as the response, we first add this matrix to the data frame:

```
> snoring$YN <- cbind(snoring$heartdisyes,snoring$n-snoring$heartdisyes)
> snoring
  snore heartdisyes    n YN.1 YN.2
1     0          24 1379   24 1355
2     2          35  638   35  603
3     4          21  213   21  192
4     5          30  254   30  224
```

Now exactly the same fit is achieved as before by treating this matrix as the response (with no `weights` argument):

```
> snoring.lg <-
+ glm(YN ~ snore, family=binomial(), data=snoring)
> snoring.lg

Call:  glm(formula = YN ~ snore, family = binomial(), data = snoring)

Coefficients:
(Intercept)        snore
    -3.8662       0.3973

Degrees of Freedom: 3 Total (i.e. Null);  2 Residual
Null Deviance:      65.9
Residual Deviance: 2.809        AIC: 27.06
```

Crabs Here we will fit a few logistic regression models to the crabs data. You may already have these data available in R from working with them in Chapter 4, or you can read them in as described there. I will load them from the sta4504 package here and add a logical variable `psat` indicating the presence of absence of satellites. This will be used as the binary response when fitting logistic regression models to the ungrouped data.

```
> library(sta4504)
> data(crabs)
> names(crabs)
[1] "color"  "spine"  "width"  "satell" "weight"
> crabs$psat <- crabs$satell > 0
```

We first fit a simple logistic regression with crab weight as predictor.

```
> crabs.lg.1 <- glm(psat ~ weight, family=binomial(), data=crabs)
> summary(crabs.lg.1)
```

```
Call:
glm(formula = psat ~ weight, family = binomial(), data = crabs)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.1108  -1.0749   0.5426   0.9122   1.6285

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.6946338  0.8779167  -4.208 2.57e-05 ***
weight       0.0018151  0.0003755   4.833 1.34e-06 ***
---
Signif. codes:
       0  ‘***’  0.001  ‘**’  0.01  ‘*’  0.05  ‘.’  0.1  ‘ ’  1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 225.76  on 172  degrees of freedom
Residual deviance: 195.74  on 171  degrees of freedom
AIC: 199.74
```

To compare this model with a null model having no predictors we can either use the Wald test above, with $z = 4.833$ and P-value $< .0001$, or a the likelihood ratio test.

```
> crabs.lg.0 <- glm(psat ~ 1, family=binomial(), data=crabs)
> anova(crabs.lg.0,crabs.lg.1,test="Chisq")
Analysis of Deviance Table

Response: psat


       Resid. Df Resid. Dev  Df Deviance P(>|Chi|)
1            172    225.759
weight       171    195.737   1   30.021 4.273e-08
```

Note that you don’t actually need to fit the null model here:

```
> anova(crabs.lg.1,test="Chisq")
Analysis of Deviance Table

Model: binomial, link: logit

Response: psat

Terms added sequentially (first to last)


        Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                     172    225.759
weight   1   30.021       171    195.737 4.273e-08
```

Bibliography

[1] Alan Agresti. *An Introduction to Categorical Data Analysis*. John Wiley & Sons, Inc., New York, 1996.

[2] R Core Development Team. *An Introduction to R*, 2000.

[3] W. N. Venables and B. D. Ripley. *Modern Applied Statistics With S-Plus*. Springer-Verlag, New York, second edition, 1997.

Appendix A

Appendix

A.1 Help Files for Some R Functions

A.1.1 `apply`

`apply` *Apply Functions Over Array Margins* `apply`

Usage

```
apply(x, MARGIN, FUN, ...)
```

Arguments

`x`	the array to be used.
`MARGIN`	a vector giving the subscripts which the function will be applied over. `1` indicates rows, `2` indicates columns, `c(1,2)` indicates rows and columns.
`FUN`	the function to be applied. In the case of functions like `+`, `%*%`, etc., the function name must be quoted.
`...`	optional arguments to `FUN`.

Value

If each call to `FUN` returns a vector of length `n`, then `apply` returns an array of dimension `c(n,dim(x)[MARGIN])` if `n > 1`. If `n` equals `1`, `apply` returns a vector if `MARGIN` has length 1 and an array of dimension `dim(x)[MARGIN]` otherwise.

If the calls to `FUN` return vectors of different lengths, `apply` returns a list of length `dim(x)[MARGIN]`.

See Also

`lapply`, `tapply`, and conveniency functions `sweep` and `aggregate`.

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

all( apply(x,2, is.vector)) # TRUE [was not in R <= 0.63.2]

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1,c2) c(mean(x[c1]),mean(x[c2]))
apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nr = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, quantile)# 5 x n matrix with rownames

all(dim(ma) == dim(apply(ma, 1:2, sum)))## wasn't ok before R 0.63.1
```

A.1.2 `ftable`

`ftable` *Flat Contingency Tables* `ftable`

Description

Create and manipulate "flat" contingency tables.

Usage

```
ftable(..., exclude = c(NA, NaN), row.vars = NULL, col.vars = NULL)
ftable2table(x)
```

Arguments

`...`	R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, or a contingency table object of class `"table"` or `"ftable"`.
`exclude`	values to use in the exclude argument of `factor` when interpreting non-factor objects.
`row.vars`	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the rows of the flat contingency table.

`col.vars` a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the columns of the flat contingency table.

`x` an arbitrary R object.

Details

`ftable` creates "flat" contingency tables. Similar to the usual contingency tables, these contain the counts of each combination of the levels of the variables (factors) involved. This information is then re-arranged as a matrix whose rows and columns correspond to unique combinations of the levels of the row and column variables (as specified by `row.vars` and `col.vars`, respectively). The combinations are created by looping over the variables in reverse order (so that the levels of the "left-most" variable vary the slowest). Displaying a contingency table in this flat matrix form (via `print.ftable`, the print method for objects of class `"ftable"`) is often preferable to showing it as a higher-dimensional array.

`ftable` is a generic function. Its default method, `ftable.default`, first creates a contingency table in array form from all arguments except `row.vars` and `col.vars`. If the first argument is of class `"table"`, it represents a contingency table and is used as is; if it is a flat table of class `"ftable"`, the information it contains is converted to the usual array representation using `ftable2table`. Otherwise, the arguments should be R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, which are cross-tabulated using `table`. Then, the arguments `row.vars` and `col.vars` are used to collapse the contingency table into flat form. If neither of these two is given, the last variable is used for the columns. If both are given and their union is a proper subset of all variables involved, the other variables are summed out.

Function `ftable.formula` provides a formula method for creating flat contingency tables.

`ftable2table` converts a contingency table in flat matrix form to one in standard array form.

Value

`ftable` returns an object of class `"ftable"`, which is a matrix with counts of each combination of the levels of variables with information on the names and levels of the (row and columns) variables stored as attributes `"row.vars"` and `"col.vars"`.

See Also

`ftable.formula` for the formula interface; `table` for "ordinary" cross-tabulation.

Examples

```
## Start with a contingency table.
data(Titanic)
ftable(Titanic, row.vars = 1:3)
ftable(Titanic, row.vars = 1:2, col.vars = "Survived")
ftable(Titanic, row.vars = 2:1, col.vars = "Survived")

## Start with a data frame.
data(mtcars)
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))
```

A.1.3 `prop.test`

Description

`prop.test` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

Usage

```
prop.test(x, n = NULL, p = NULL, alternative = "two.sided",
          conf.level = 0.95, correct = TRUE)
```

Arguments

`x`	a vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
`n`	a vector of counts of trials; ignored if `x` is a matrix.
`p`	a vector of probabilities of success. The length of `p` must be the same as the number of groups specified by `x`, and its elements must be greater than 0 and less than 1.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used for testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`conf.level`	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Only used when testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`correct`	a logical indicating whether Yates' continuity correction should be applied.

Details

Only groups with finite numbers of successes and failures are used. Counts of successes and failures must be nonnegative and hence not greater than the corresponding numbers of trials which must be positive. All finite counts should be integers.

If `p` is `NULL` and there is more than one group, the null tested is that the proportions in each group are the same. If there are two groups, the alternatives are that the probability of success in the first group is less than, not equal to, or greater than the probability of success in the second group, as specified by `alternative`. A confidence interval for the difference of proportions with confidence level as specified by `conf.level` and clipped to $[-1, 1]$ is returned. Continuity correction is used only if it does not exceed the difference of the sample proportions in absolute value. Otherwise, if there are more than 2 groups, the alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

If there is only one group, then the null tested is that the underlying probability of success is `p`, or .5 if `p` is not given. The alternative is that the probability of success if less than, not equal to, or greater than `p` or 0.5, respectively, as specified by `alternative`. A confidence interval for the underlying proportion with confidence level as specified by `conf.level` and clipped to $[0, 1]$ is returned. Continuity correction is used only if it does not exceed the difference between sample and null proportions in absolute value.

Finally, if p is given and there are more than 2 groups, the null tested is that the underlying probabilities of success are those given by p. The alternative is always "two.sided", the returned confidence interval is NULL, and continuity correction is never used.

Value

A list with class "htest" containing the following components:

statistic	the value of Pearson's chi-square test statistic.
parameter	the degrees of freedom of the approximate chi-square distribution of the test statistic.
p.value	the p-value of the test.
estimate	a vector with the sample proportions x/n.
conf.int	a confidence interval for the true proportion if there is one group, or for the difference in proportions if there are 2 groups and p is not given, or NULL otherwise. In the cases where it is not NULL, the returned confidence interval has an asymptotic confidence level as specified by conf.level, and is appropriate to the specified alternative hypothesis.
null.value	the value of p if specified by the null, or NULL otherwise.
alternative	a character string describing the alternative.
method	a character string indicating the method used, and whether Yates' continuity correction was applied.
data.name	a character string giving the names of the data.

Examples

```
heads <- rbinom(1, size=100, pr = .5)
prop.test(heads, 100)          # continuity correction TRUE by default
prop.test(heads, 100, correct = FALSE)

## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
```

A.1.4 sweep

sweep	*Sweep out Array Summaries*	sweep

Usage

```
sweep(x, MARGIN, STATS, FUN="-", ...)
```

Arguments

x	an array.
MARGIN	a giving the extents of x which correspond to STATS.
STATS	the summary statistic which is to be swept out.
FUN	the function to be used to carry out the sweep. In the case of binary operators such as "/" etc., the function name must be quoted.
...	optional arguments to FUN.

Value

An array with the same shape as x, but with the summary statistics swept out.

See Also

apply on which sweep is based; scale for centering and scaling.

Examples

```
data(attitude)
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)# subtract the column medians
```

A.1.5 tapply

tapply	*Apply a Function Over a "Ragged" Array*	tapply

Usage

```
tapply(X, INDEX, FUN = NULL, simplify = TRUE, ...)
```

Arguments

X	an atomic object, typically a vector.
INDEX	list of factors, each of same length as X.
FUN	the function to be applied. In the case of functions like +, %*%, etc., the function name must be quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces.
simplify	If FALSE, tapply always returns an array of mode "list". If TRUE (the default), then if FUN always returns a scalar, tapply returns an array with the mode of the scalar.
...	optional arguments to FUN.

Value

When FUN is present, tapply calls FUN for each cell that has any data in it. If FUN returns a single atomic value for each cell (e.g., functions mean or var) and when simplify is true, tapply returns a multi-way array containing the values. The array has the same number of dimensions as INDEX has components; the number of levels in a dimension is the number of levels (nlevels(.)) in the corresponding component of INDEX.

Note that contrary to S, simplify = TRUE always returns an array, possibly 1-dimensional.

If FUN does not return a single atomic value, tapply returns an array of mode list whose components are the values of the individual calls to FUN, i.e., the result is a list with a dim attribute.

See Also

the convenience function aggregate (using tapply); apply, lapply with its version sapply.

Examples

```
groups <- as.factor(rbinom(32, n = 5, p = .4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

data(warpbreaks)
## contingency table from data.frame : array with named dimnames
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
tapply(warpbreaks$breaks, warpbreaks[,3,drop=F], sum)

n <- 17; fac <- factor(rep(1:3, len = n), levels = 1:5)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
tapply(1:3, ind) #-> the split vector
tapply(1:3, ind, sum)
```