Module 5

PROBABILITY DISTRIBUTIONS

Being a statistical programming language, R easily handles all the basic necessities of statistics, including drawing random numbers and calculating distribution values (the focus of this chapter), means, variances, maxmima and minima, correlation and t-tests. Probability distributions lie at the heart of statistics, so naturally R provides numerous functions for making use of them. These include functions for generating random numbers and calculating the distribution and quantile.

Normal Distribution

Perhaps the most famous, and most used, statistical distribution is the normal distribution, sometimes referred to as the Gaussian distribution, which is defined as

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{(-(x-\mu)^2)/(2\sigma^2)}$$

where μ is the mean and σ the standard deviation. This is the famous bell curve that describes so many phenomena in life. To draw random numbers from the normal distribution use the rnorm function, which optionally allows the specification of the mean and standard deviation.

> # 10 draws from the standard 0-1 normal distribution

> rnorm(n=10)

[1] 0.4385627 1.1969098 1.0130680 0.0053413 -0.6086422 -1.5829601

[7] 0.9106169 -1.9663997 1.0108341 0.1931879

> # 10 draws from the 100-20 distribution

> rnorm(n=10, mean=100, sd=20)

[1] 114.99418 121.15465 95.35524 95.73121 86.45346 106.73548

[7] 104.05061 113.61679 101.40346 61.48190

The density (the probability of a particular value) for the normal distribution is calculated using dnorm.

> randNorm10 <- rnorm(10)

> randNorm10

[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883

[7] -0.1721544 1.8832073 0.5361347 -1.2932703

> dnorm(randNorm10)

[1] 0.06406161 0.33673288 0.34194033 0.02472905 0.29799802 0.21037889

[7] 0.39307411 0.06773357 0.34553589 0.17287050

> dnorm(c(-1, 0, 1))

[1] 0.2419707 0.3989423 0.2419707

dnorm returns the probability of a specific number occuring. While it is technically mathematically impossible to find the exact probability of a number from a continuous distribution, this is an estimate of the probability. Like with rnorm, a mean and standard deviation can be specified for dnorm. To see this visually we generate a number of normal random variables, calculate their distributions and then plot them. This should result in a nicely shaped bell curve, as seen in Figure 17.1.

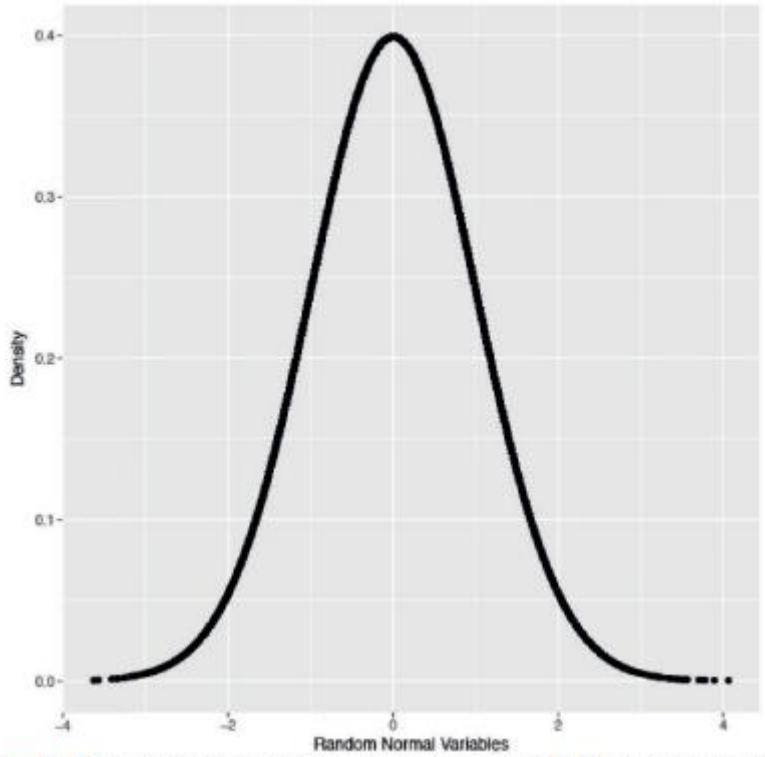


Figure 17.1 Plot of random normal variables and their densities, which results in a bell curve.

```
> # generate the normal variables
> randNorm <- rnorm(30000)
> # calcualte their distributions
> randDensity <- dnorm(randNorm)
> # load ggplot2
> library(ggplot2)
> # plot them
> ggplot(data.frame(x=randNorm, y=randDensity)) + aes(x=x, y=y) +
+ geom_point() + labs(x="Random Normal Variables", y="Density")
```

Similarly, pnorm calculates the distribution of the normal distribution, that is, the cumulative probability that a given number, or smaller number, occurs. This is defined as

$$\Phi(a) = P\{X <= a\} = \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \, \mathrm{d}x \qquad (17.2)$$

> pnorm(randNorm10)

[1] 0.972098753 0.280188016 0.710656152 0.009178915 0.777520317

[6] 0.871033114 0.431658071 0.970163858 0.704067283 0.097958799

> pnorm(c(-3, 0, 3))

[1] 0.001349898 0.500000000 0.998650102

> pnorm(-1)

[1] 0.1586553

By default this is left-tailed. To find the probability that the variable falls between two points, we must calculate the two probabilities and subtract them from each other.

> pnorm(1) - pnorm(0)

[1] 0.3413447

> pnorm(1) - pnorm(-1)

[1] 0.6826895

This probability is represented by the area under the curve and illustrated in Figure 17.2, which is drawn by the following code.

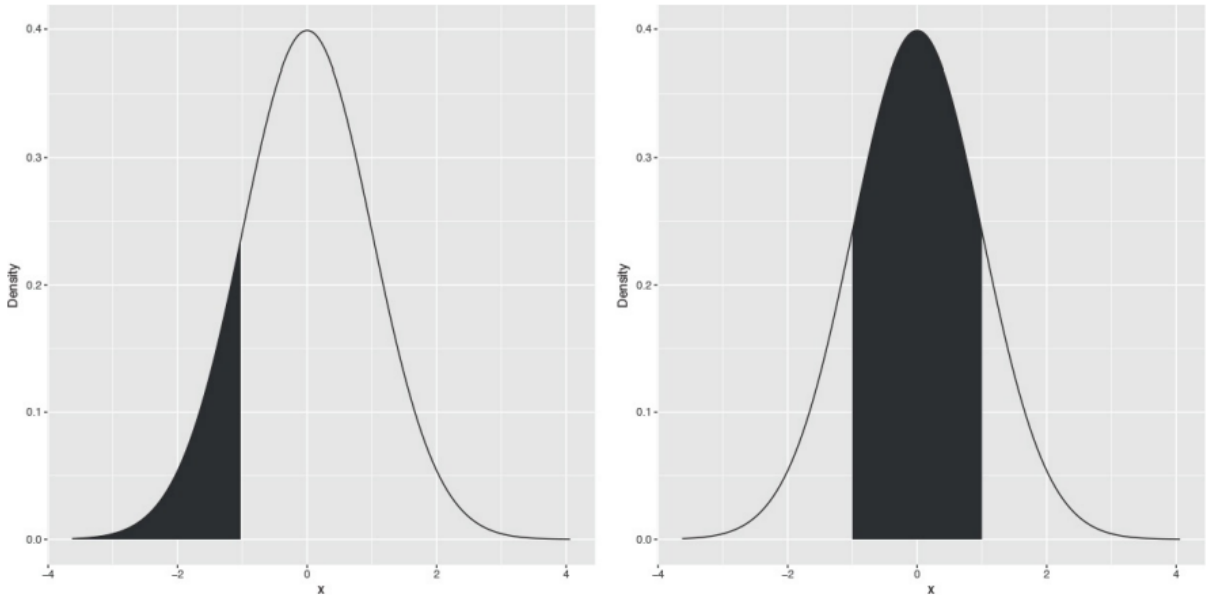


Figure 17.2 Area under a normal curve. The plot on the left shows the area to the left of –1, while the plot on the right shows the area between –1 and 1.

> # a few things happen with this first line of code

> # the idea is to build a ggplot2 object that we can build upon later

> # that is why it is saved to p

> # we take randNorm and randDensity and put them into a data.frame

> # we declare the x and y aes outside of any other function

```
> # this just gives more flexibility
> # we add lines with geom_line()
> # x- and y-axis labels with labs(x="x", y="Density")
> p <- ggplot(data.frame(x=randNorm, y=randDensity)) + aes(x=x, y=y) +
+ geom_line() + labs(x="x", y="Density")
>
> # plotting p will print a nice distribution
> # to create a shaded area under the curve we first calculate that area
> # generate a sequence of numbers going from the far left to -1
> neg1Seq <- seq(from=min(randNorm), to=-1, by=.1)
>
> # build a data.frame of that sequence as x
> # the distribution values for that sequence as y
> lessThanNeg1 <- data.frame(x=neg1Seq, y=dnorm(neg1Seq))
>
> head(lessThanNeg1)
x y
1 -4.164144 6.847894e-05
2 -4.064144 1.033313e-04
3 -3.964144 1.543704e-04
4 -3.864144 2.283248e-04
5 -3.764144 3.343484e-04
6 -3.664144 4.847329e-04
> # combine this with endpoints at the far left and far right
> # the height is 0
> lessThanNeg1 <- rbind(c(min(randNorm), 0),
+ lessThanNeg1,
+ c(max(lessThanNeg1$x), 0))
>
> # use that shaded region as a polygon
> p + geom_polygon(data=lessThanNeg1, aes(x=x, y=y))
```

```
>
> # create a similar sequence going from -1 to 1
> neg1Pos1Seq <- seq(from=-1, to=1, by=.1)
>
> # build a data.frame of that sequence as x
> # the distribution values for that sequence as y
> neg1To1 <- data.frame(x=neg1Pos1Seq, y=dnorm(neg1Pos1Seq))
>
> head(neg1To1)
x y
1 -1.0 0.2419707
2 -0.9 0.2660852
3 -0.8 0.2896916
4 -0.7 0.3122539
5 -0.6 0.3332246
6 -0.5 0.3520653
> # combine this with endpoints at the far left and far right
> # the height is 0
> neg1To1 <- rbind(c(min(neg1To1$x), 0),
+ neg1To1,
+ c(max(neg1To1$x), 0))
>
> # use that shaded region as a polygon
> p + geom_polygon(data=neg1To1, aes(x=x, y=y))
```

The distribution has a non-decreasing shape, as shown in Figure 17.3. The information displayed here is the same as in Figure 17.2 but it is shown differently. Instead of the cumulative probability being shown as a shaded region, it is displayed as a single point along the y-axis.

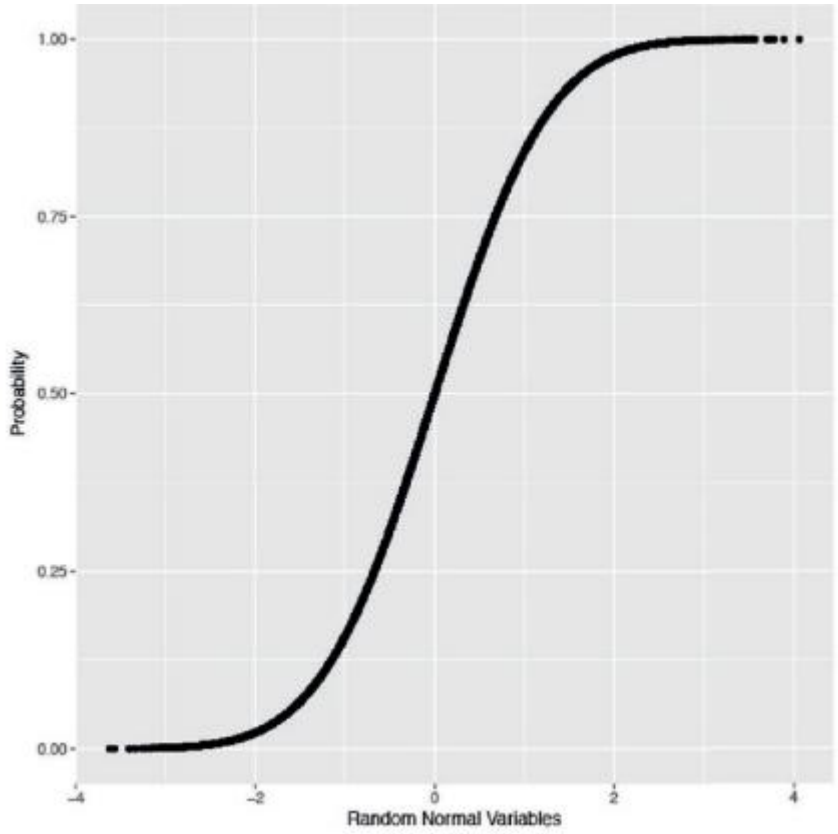


Figure 17.3 Normal distribution function.

```
> randProb <- pnorm(randNorm)
> ggplot(data.frame(x=randNorm, y=randProb)) + aes(x=x, y=y) +
+ geom_point() + labs(x="Random Normal Variables", y="Probability")
```

The opposite of pnorm is qnorm. Given a cumulative probability it returns the quantile.

```
> randNorm10
[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883
[7] -0.1721544 1.8832073 0.5361347 -1.2932703
> qnorm(pnorm(randNorm10))
[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883
[7] -0.1721544 1.8832073 0.5361347 -1.2932703
> all.equal(randNorm10, qnorm(pnorm(randNorm10)))
[1] TRUE
```

Binomial Distribution

Like the normal distribution, the binomial distribution is well represented in R. Its probability mass function is

$$p(x;n,p) = \binom{n}{x} p^x (1-p)^{n-x} \quad (17.3)$$

Where

$$\binom{n}{x} = \frac{n!}{x!(n-x)!} \quad (17.4)$$

and n is the number of trials and p is the probability of success of a trial. The mean is np and the variance is np(1 − p). When n = 1 this reduces to the Bernoulli distribution. Generating random numbers from the binomial distribution is not simply generating random numbers but rather generating the number of successes of independent trials. To simulate the number of successes out of ten trials with probability 0.4 of success, we run rbinom with n=1 (only one run of the trials), size=10 (trial size of 10) and prob=0.4 (probability of success is 0.4).

> rbinom(n=1, size=10, prob=.4)

[1] 1

That is to say that ten trials were conducted, each with 0.4 probability of success, and the number generated is the number that succeeded. As this is random, different numbers will be generated each time. By setting n to anything greater than 1, R will generate the number of successes for each of the n sets of size trials.

> rbinom(n=1, size=10, prob=.4)

[1] 3

> rbinom(n=5, size=10, prob=.4)

[1] 4 3 5 2 5

> rbinom(n=10, size=10, prob=.4)

[1] 5 2 7 4 7 3 2 3 3 3

Setting size to 1 turns the numbers into a Bernoulli random variable, which can take on only the value 1 (success) or 0 (failure).

> rbinom(n=1, size=1, prob=.4)

[1] 0

> rbinom(n=5, size=1, prob=.4)

[1] 1 1 0 0 0

> rbinom(n=10, size=1, prob=.4)

[1] 0 0 0 0 0 0 0 1 0 0

To visualize the binomial distribution, we randomly generate 10,000 experiments, each with 10 trials and 0.3 probability of success. This is seen in Figure 17.4, which shows that the most common number of successes is 3, as expected.

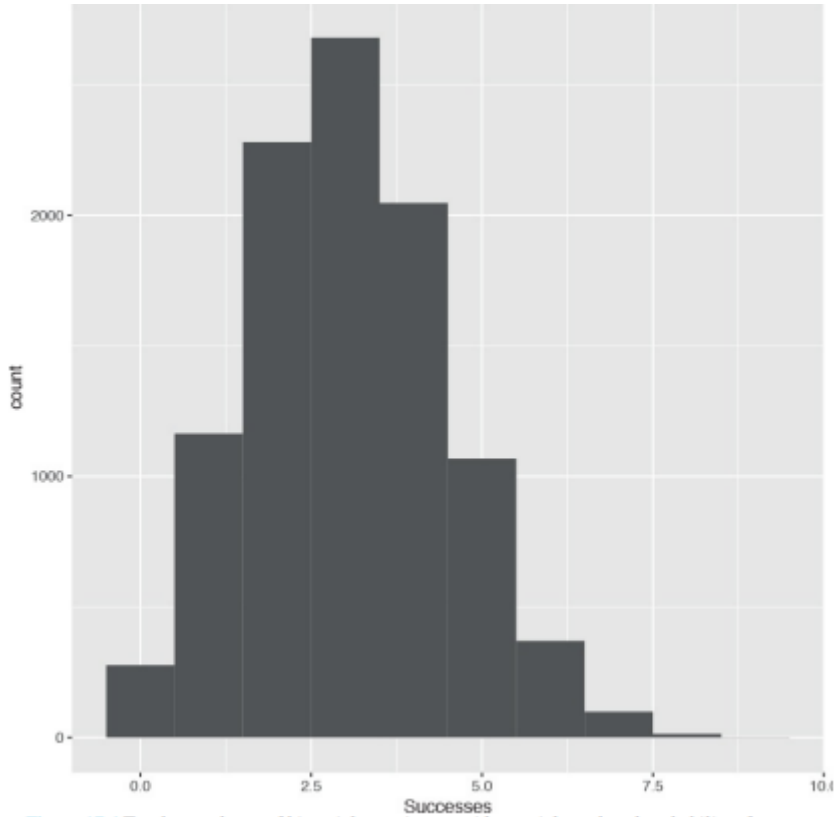


Figure 17.4 Ten thousand runs of binomial experiments with ten trials each and probability of success of 0.3.

```
> binomData <- data.frame(Successes=rbinom(n=10000, size=10, prob=.3))
```

```
> ggplot(binomData, aes(x=Successes)) + geom_histogram(binwidth=1)
```

To see how the binomial distribution is well approximated by the normal distribution as the number of trials grows large, we run similar experiments with differing numbers of trials and graph the results, as shown in Figure 17.5.

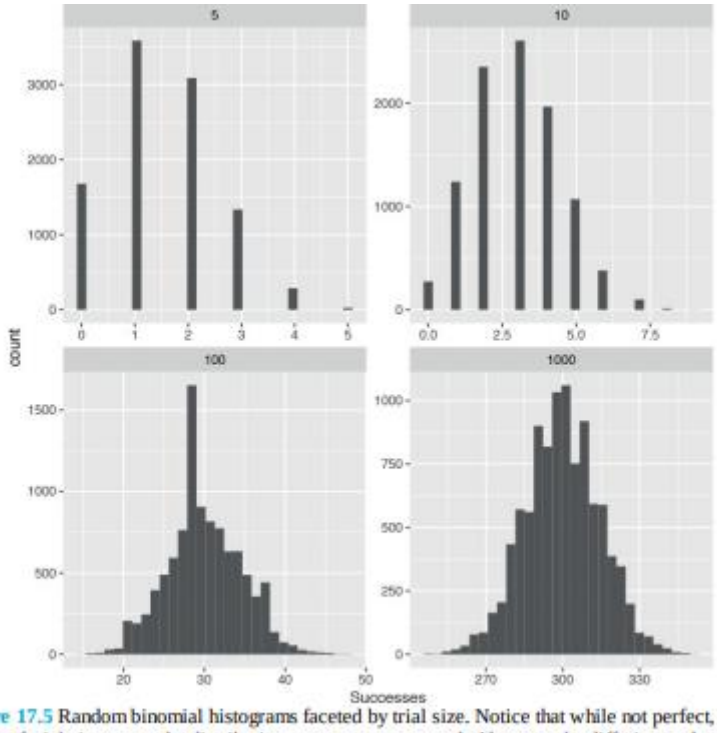


Figure 17.5 Random binomial histograms faceted by trial size. Notice that while not perfect, as the number of trials increases the distribution appears more normal. Also note the differing scales in each pane.

```
> # create a data.frame with Successes being the 10,000 random draws
> # Size equals 5 for all 10,000 rows
> binom5 <- data.frame(Successes=rbinom(n=10000, size=5, prob=.3), Size=5)
> dim(binom5)
[1] 10000 2
> head(binom5)
Successes Size
1 2 5
2 1 5
3 2 5
4 1 5
5 2 5
6 2 5
> # similar as before, still 10,000 rows
> # numbers are drawn from a distribution with a different size
> # Size now equals 10 for all 10,000 rows
> binom10 <- data.frame(Successes=rbinom(n=10000, size=10, prob=.3), Size=10)
> dim(binom10)
```

```
[1] 10000 2
> head(binom10)
Successes Size
1 2 10
2 2 10
3 1 10
4 2 10
5 4 10
6 1 10
> binom100 <- data.frame(Successes=rbinom(n=10000, size=100, prob=.3), Size=100)
>
> binom1000 <- data.frame(Successes=rbinom(n=10000, size=1000, prob=.3), Size=1000)
>
> # combine them all into one data.frame
> binomAll <- rbind(binom5, binom10, binom100, binom1000)
> dim(binomAll)
[1] 40000 2
> head(binomAll, 10)
Successes Size
1 2 5
2 1 5
3 2 5
4 1 5
5 2 5
6 2 5
7 1 5
8 1 5
9 2 5
10 1 5
> tail(binomAll, 10)
Successes Size
```

39991 288 1000

39992 289 1000

39993 297 1000

39994 327 1000

39995 336 1000

39996 290 1000

39997 310 1000

39998 328 1000

39999 281 1000

40000 307 1000

> # build the plot

> # histograms only need an x aesthetic

> # it is faceted (broken up) based on the values of Size

> # these are 5, 10, 100, 1000

> ggplot(binomAll, aes(x=Successes)) + geom_histogram() +

+ facet_wrap(~ Size, scales="free")

The cumulative distribution function is

$$F(a; n, p) = P\{X <= a\} = \sum_{i=0}^{a} \binom{n}{i} p^i (1-p)^{n-i} \qquad (17.5)$$

where n and p are the number of trials and the probability of success, respectively, as before. Similar to the normal distribution functions, dbinom and pbinom provide the density (probability of an exact value) and distribution (cumulative probability), respectively, for the binomial distribution.

> # probability of 3 successes out of 10

> dbinom(x=3, size=10, prob=.3)

[1] 0.2668279

> # probability of 3 or fewer successes out of 10

> pbinom(q=3, size=10, prob=.3)

[1] 0.6496107

> # both functions can be vectorized

> dbinom(x=1:10, size=10, prob=.3)

[1] 0.1210608210 0.2334744405 0.2668279320 0.2001209490 0.1029193452

[6] 0.0367569090 0.0090016920 0.0014467005 0.0001377810 0.0000059049

> pbinom(q=1:10, size=10, prob=.3)

[1] 0.1493083 0.3827828 0.6496107 0.8497317 0.9526510 0.9894079

[7] 0.9984096 0.9998563 0.9999941 1.0000000

Given a certain probability, qbinom returns the quantile, which for this distribution is the number of successes.

> qbinom(p=.3, size=10, prob=.3)

[1] 2

> qbinom(p=c(.3, .35, .4, .5, .6), size=10, prob=.3)

[1] 2 2 3 3 3

Poisson Distribution

Another popular distribution is the Poisson distribution, which is for count data. Its probability mass function is

$$p(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!} \tag{17.6}$$

$$F(a;\lambda) = P\{X <= a\} = \sum_{i=0}^{a} \frac{\lambda^i e^{-\lambda}}{i!} \tag{17.7}$$

and the cumulative distribution is where λ is both the mean and variance. To generate random counts, the density, the distribution and quantiles use rpois, dpois, ppois and qpois, respectively. As λ grows large the Poisson distribution begins to resemble the normal distribution. To see this we will simulate 10,000 draws from the Poisson distribution and plot their histograms to see the shape.

> # generate 10,000 random counts from 5 different Poisson distributions

> pois1 <- rpois(n=10000, lambda=1)

> pois2 <- rpois(n=10000, lambda=2)

> pois5 <- rpois(n=10000, lambda=5)

> pois10 <- rpois(n=10000, lambda=10)

> pois20 <- rpois(n=10000, lambda=20)

> pois <- data.frame(Lambda.1=pois1, Lambda.2=pois2,

+ Lambda.5=pois5, Lambda.10=pois10, Lambda.20=pois20)

> # load reshape2 package to melt the data to make it easier to plot

> library(reshape2)

```
> # melt the data into a long format
> pois <- melt(data=pois, variable.name="Lambda", value.name="x")
> # load the stringr package to help clean up the new column name
> library(stringr)
> # clean up the Lambda to just show the value for that lambda
> pois$Lambda <- as.factor(as.numeric(str_extract(string=pois$Lambda,
+ pattern="\\d+")))
> head(pois)
Lambda x
1 1 1
2 1 1
3 1 1
4 1 2
5 1 2
6 1 0
> tail(pois)
Lambda x
49995 20 22
49996 20 15
49997 20 24
49998 20 23
49999 20 20
50000 20 23
```

Now we will plot a separate histogram for each value of λ, as shown in Figure 17.6.

```
> library(ggplot2)
> ggplot(pois, aes(x=x)) + geom_histogram(binwidth=1) +
+ facet_wrap(~ Lambda) + ggtitle("Probability Mass Function")
```

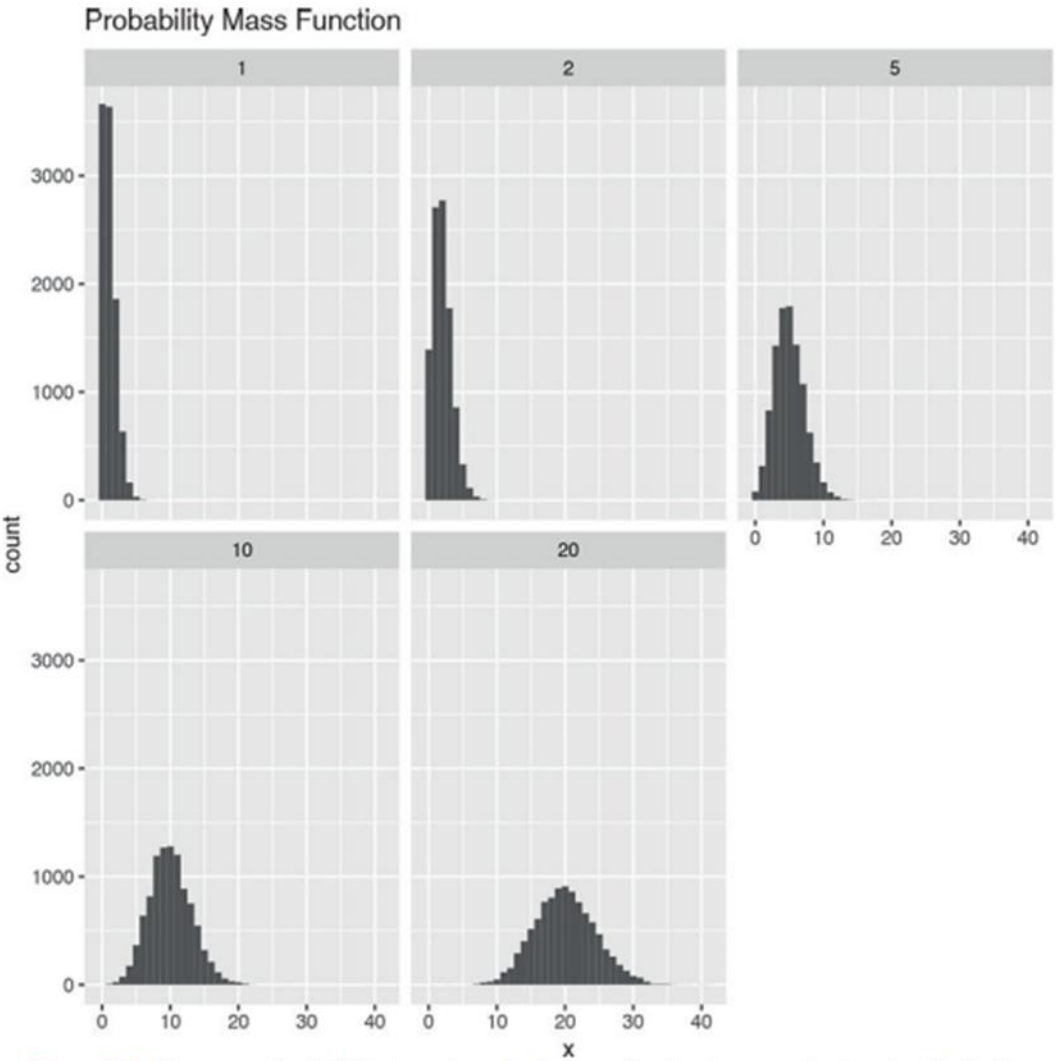


Figure 17.6 Histograms for 10,000 draws from the Poisson distribution at varying levels of λ. Notice how the histograms become more like the normal distribution.

```
> ggplot(pois, aes(x=x)) +
+ geom_density(aes(group=Lambda, color=Lambda, fill=Lambda),
+ adjust=4, alpha=1/2) +
+ scale_color_discrete() + scale_fill_discrete() +
+ ggtitle("Probability Mass Function")
```

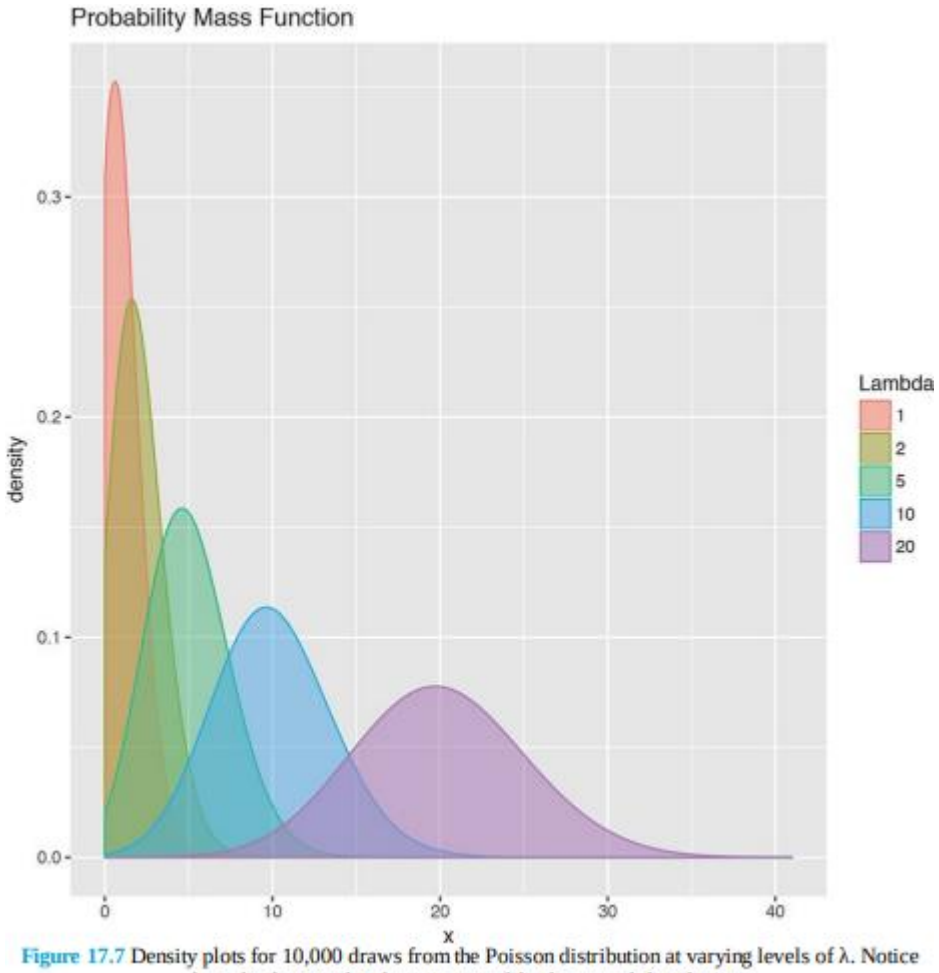


Figure 17.7 Density plots for 10,000 draws from the Poisson distribution at varying levels of λ. Notice how the density plots become more like the normal distribution.

Other Distributions

R supports many distributions, some of which are very common, while others are quite obscure. They are listed in Table 17.1; the mathematical formulas, means and variances are in Table 17.2.

Distribution	Random Number	Density	Distribution	Quantile
Normal	rnorm	dnorm	pnorm	qnorm
Binomial	rbinom	dbinom	pbinom	qbinom
Poisson	rpois	dpois	ppois	qpois
t	rt	dt	pt	qt
F	rf	df	pf	qf
Chi-Squared	rchisq	dchisq	pchisq	qchisq
Gamma	rgamma	dgamma	pgamma	qgamma
Geometric	rgeom	dgeom	pgeom	qgeom
Negative Binomial	rnbinom	dnbinom	pnbinom	qnbinom
Exponential	rexp	dexp	pexp	qexp
Weibull	rweibull	dweibull	pweibull	qweibull
Uniform (Continuous)	runif	dunif	punif	qunif
Beta	rbeta	dbeta	pbeta	qbeta
Cauchy	rcauchy	dcauchy	pcauchy	qcauchy
Multinomial	rmultinom	dmultinom	pmultinom	qmultinom
Hypergeometric	rhyper	dhyper	phyper	qhyper
Log-normal	rlnorm	dlnorm	plnorm	qlnorm
Logistic	rlogis	dlogis	plogis	qlogis

Table 17.1 Statistical distributions and their functions

Distribution	Formula	Mean	Variance
Normal	$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$	μ	σ^2
Binomial	$p(x;n,p) = \binom{n}{x} p^x (1-p)^{n-x}$	np	$np(1-p)$
Poisson	$p(i) = \binom{n}{i} p^i (1-p)^{n-i}$	λ	λ
t	$f(x;n) = \frac{\Gamma(\frac{n+1}{2})}{\sqrt{n\pi}\Gamma(\frac{n}{2})} \left(1 + \frac{x^2}{n}\right)^{-\frac{n+1}{2}}$	0	$\frac{n}{n-2}$
F	$f(x;\lambda,s) = \frac{\sqrt{\frac{(n_1 x)^{n_1} n_2^{n_2}}{(n_1 x + n_2)^{n_1+n_2}}}}{x B(\frac{n_1}{2}, \frac{n_2}{2})}$	$\frac{n_2}{n_2-2}$	$\frac{2n_2^2(n_1+n_2-2)}{n_1(n_2-2)^2(n_2-4)}$
Chi-Squared	$f(x;n) = \frac{e^{-\frac{y}{2}} y^{(\frac{n}{2})-1}}{2^{\frac{n}{2}}\Gamma(\frac{n}{2})}$	n	$2n$
Gamma	$f(x;\lambda,s) = \frac{\lambda e^{-\lambda x}(\lambda x)^{s-1}}{\Gamma(s)}$	$\frac{s}{\lambda}$	$\frac{s}{\lambda^2}$
Geometric	$p(x;p) = p(1-p)^{x-1}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
Negative Binomial	$p(x;r,p) = \binom{x-1}{r-1} p^r (1-p)^{x-r}$	$\frac{r}{p}$	$\frac{r(1-p)}{p^2}$
Exponential	$f(x;\lambda) = \lambda e^{-\lambda x}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
Weibull	$f(x;\lambda,k) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(x/\lambda)^k}$	$\lambda\Gamma(1+\frac{1}{k})$	$\lambda^2\Gamma(1+\frac{2}{k}) - \mu^2$
Uniform	$f(x;a,b) = \frac{1}{b-a}$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
Beta	$f(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
Cauchy	$f(x;s,t) = \frac{s}{\pi\left(s^2 + (x-t)^2\right)}$	Undefined	Undefined
Multinomial	$p(x_1,\ldots,x_k;n,p_1,\ldots,p_k) = \frac{n!}{x_1!\cdots x_k!} p_1^{x_1}\cdots p_k^{x_k}$	np_i	$np_i(1-p_i)$
Hypergeometric	$p(x;N,n,m) = \frac{\binom{m}{x}\binom{N-m}{n-x}}{\binom{N}{n}}$	$\frac{nm}{N}$	$\frac{nm}{N}\left[\frac{(n-1)(m-1)}{N-1} + 1 - \frac{nm}{N}\right]$
Log-normal	$f(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$	$e^{\mu+\frac{\sigma^2}{2}}$	$\left(e^{\sigma^2}-1\right)e^{2\mu+\sigma^2}$
Logistic	$f(x;\mu,s) = \frac{e^{-\frac{x-\mu}{s}}}{s\left(1+e^{-\frac{x-\mu}{s}}\right)^2}$	μ	$\frac{1}{3}s^2\pi^2$

Table 17.2 Formulas, means and variances for various statistical distributions. The *B* in the F distribution is the Beta function, $B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\mathrm{d}t$.

Basic Statistics

Some of the most common tools used in statistics are means, variances, correlations and t-tests. These are all well represented in R with easy-to-use functions such as mean, var, cor and t.test.

Summary Statistics

The first thing many people think of in relation to statistics is the average, or mean, as it is properly called. We start by looking at some simple numbers and later in the chapter play with bigger datasets. First we generate a random sampling of 100 numbers between 1 and 100.

```
> x <- sample(x=1:100, size=100, replace=TRUE)

> x

[1] 53 89 28 97 35 51 21 55 47 3 46 35 86 66 51 20 41 15 10 22 31

[22] 86 19 13 10 59 60 58 90 11 54 79 45 49 23 91 80 30 83 69 20 76

[43] 2 42 35 51 76 77 90 84 12 36 79 38 68 87 72 17 20 57 61 83 23

[64] 61 64 41 31 74 35 20 85 89 64 73 11 36 12 81 10 64 39 4 69 42

[85] 41 85 84 66 76 23 47 56 50 82 21 67 89 57 6 13
```

sample uniformly draws size entries from x. Setting replace=TRUE means that the same number can be drawn multiple times.

Now that we have a vector of data we can calculate the mean.

```
> mean(x)

[1] 49.85
```

This is the simple arithmetic mean.

$$E[X] = \frac{\sum_{i=1}^{N} x_i}{N} \qquad (18.1)$$

Simple enough. Because this is statistics, we need to consider cases where some data is missing. To create this we take x and randomly set 20 percent of the elements to NA.

```
> # copy x

> y <- x

> # choose a random 20 elements, using sample, to set to NA

> y[sample(x=1:100, size=20, replace=FALSE)] <- NA

> y

[1] 53 89 28 97 35 51 21 55 47 NA 46 35 86 NA NA NA 41 15 10 22 31

[22] NA 19 13 NA 59 60 NA 90 11 NA 79 45 NA 23 91 80 30 83 69 20 76

[43] 2 42 35 51 76 77 NA 84 NA 36 79 38 NA 87 72 17 20 57 61 83 NA

[64] 61 64 41 31 74 NA 20 NA 89 64 73 NA 36 12 NA 10 64 39 4 NA 42
```

[85] 41 85 84 66 76 23 47 56 50 82 21 67 NA NA 6 13

Using mean on y will return NA. This is because, by default, if mean encounters even one element that is NA it will return NA. This is to avoid providing misleading information.

> mean(y, na.rm=TRUE)

[1] 49.6

To calculate the weighted mean of a set of numbers, the function weighted.mean takes a vector of numbers and a vector of weights. It also has an optional argument, na.rm, to remove NAs before calculating; otherwise, a vector with NA values will return NA.

> grades <- c(95, 72, 87, 66)

> weights <- c(1/2, 1/4, 1/8, 1/8)

> mean(grades)

[1] 80

> weighted.mean(x=grades, w=weights)

[1] 84.625

The formula for weighted.mean is in Equation 18.2, which is the same as the expected value of a random variable.

$$E[X] = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i} = \sum_{i=1}^{N} p_i x_i \qquad (18.2)$$

Another vitally important metric is the variance, which is calculated with var.

> var(x)

[1] 724.5328

This calculates variance as

$$Var(x) = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N - 1} \qquad (18.3)$$

which can be verified in R.

> var(x)

[1] 724.5328

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

[1] 724.5328

Standard deviation is the square root of variance and is calculated with sd. Like mean and var, sd has the na.rm argument to remove NAs before computation; otherwise, any NAs will cause the answer to be NA.

```
> sqrt(var(x))

[1] 26.91715

> sd(x)

[1] 26.91715

> sd(y)

[1] NA

> sd(y, na.rm=TRUE)

[1] 26.48506
```

Other commonly used functions for summary statistics are min, max and median. Of course, all of these also have na.rm arguments.

```
> min(x)

[1] 2

> max(x)

[1] 97

> median(x)

[1] 51

> min(y)

[1] NA

> min(y, na.rm=TRUE)

[1] 2
```

The median, as calculated before, is the middle of an ordered set of numbers. For instance, the median of 5, 2, 1, 8 and 6 is 5. In the case when there are an even amount of numbers, the median is the mean of the middle two numbers. For 5, 1, 7, 4, 3, 8, 6 and 2, the median is 4.5. A helpful function that computes the mean, minimum, maximum and median is summary. There is no need to specify na.rm because if there are NAs, they are automatically removed and their count is included in the results.

```
> summary(x)

Min. 1st Qu. Median Mean 3rd Qu. Max.

2.00 23.00 51.00 49.85 74.50 97.00

> summary(y)

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

2.00 26.75 48.50 49.60 74.50 97.00 20
```

This summary also displayed the first and third quantiles. These can be computed using quantile.

```
> # calculate the 25th and 75th quantile
```

```
> quantile(x, probs=c(.25, .75))

25% 75%

23.0 74.5

> # try the same on y

> quantile(y, probs=c(.25, .75))

Error in quantile.default(y, probs = c(0.25, 0.75)): missing values and NaN's

not allowed if 'na.rm' is FALSE

> # this time use na.rm=TRUE

> quantile(y, probs=c(.25, .75), na.rm=TRUE)

25% 75%

26.75 74.50

> # compute other quantiles

> quantile(x, probs=c(.1, .25, .5, .75, .99))

10% 25% 50% 75% 99%

12.00 23.00 51.00 74.50 91.06
```

Quantiles are numbers in a set where a certain percentage of the numbers are smaller than that quantile. For instance, of the numbers one through 200, the 75th quantile—the number that is larger than 75 percent of the numbers—is 150.25.

Correlation and Covariance

When dealing with more than one variable, we need to test their relationship with each other. Two simple, straightforward methods are correlation and covariance. To examine these concepts we look at the economics data from ggplot2.

```
> library(ggplot2)

> head(economics)

# A tibble: 6 × 8

date pce pop psavert uempmed unemploy year month

<date> <dbl> <int> <dbl> <dbl> <int> <dbl> <ord>

1 1967-07-01 507.4 198712 12.5 4.5 2944 1967 Jul

2 1967-08-01 510.5 198911 12.5 4.7 2945 1967 Aug

3 1967-09-01 516.3 199113 11.7 4.6 2958 1967 Sep

4 1967-10-01 512.9 199311 12.5 4.9 3143 1967 Oct

5 1967-11-01 518.1 199498 12.5 4.7 3066 1967 Nov
```

6 1967-12-01 525.8 199657 12.1 4.8 3018 1967 Dec

In the economics dataset, pce is personal consumption expenditures and psavert is the personal savings rate. We calculate their correlation using cor.

> cor(economics$pce, economics$psavert)

[1] -0.837069

This very low correlation makes sense because spending and saving are opposites of each other.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y} \qquad (18.4)$$

Correlation is defined as where and are the means of x and y, and sx and sy are the standard deviations of x and y. It can range between -1 and 1, with higher positive numbers meaning a closer relationship between the two variables, lower negative numbers meaning an inverse relationship and numbers near zero meaning no relationship. This can be easily checked by computing Equation 18.4.

> # use cor to calculate correlation

> cor(economics$pce, economics$psavert)

[1] -0.837069

> ## calculate each part of correlation

> xPart <- economics$pce - mean(economics$pce)

> yPart <- economics$psavert - mean(economics$psavert)

> nMinusOne <- (nrow(economics) - 1)

> xSD <- sd(economics$pce)

> ySD <- sd(economics$psavert)

> # use correlation formula

> sum(xPart * yPart) / (nMinusOne * xSD * ySD)

[1] -0.837069

To compare multiple variables at once, use cor on a matrix (only for numeric variables).

> cor(economics[, c(2, 4:6)])

pce psavert uempmed unemploy

pce 1.0000000 -0.8370690 0.7273492 0.6139997

psavert -0.8370690 1.0000000 -0.3874159 -0.3540073

uempmed 0.7273492 -0.3874159 1.0000000 0.8694063

unemploy 0.6139997 -0.3540073 0.8694063 1.0000000

Because this is just a table of numbers, it would be helpful to also visualize the information using a plot. For this we use the ggpairs function from the GGally package (a collection of helpful plots built on ggplot2) shown in Figure 18.1. This shows a scatterplot of every variable in the data against every other variable. Loading GGally also loads the reshape package, which causes namespace issues with the newer reshape2 package. So rather than load GGally, we call its function using the :: operator, which allows access to functions within a package without loading it.

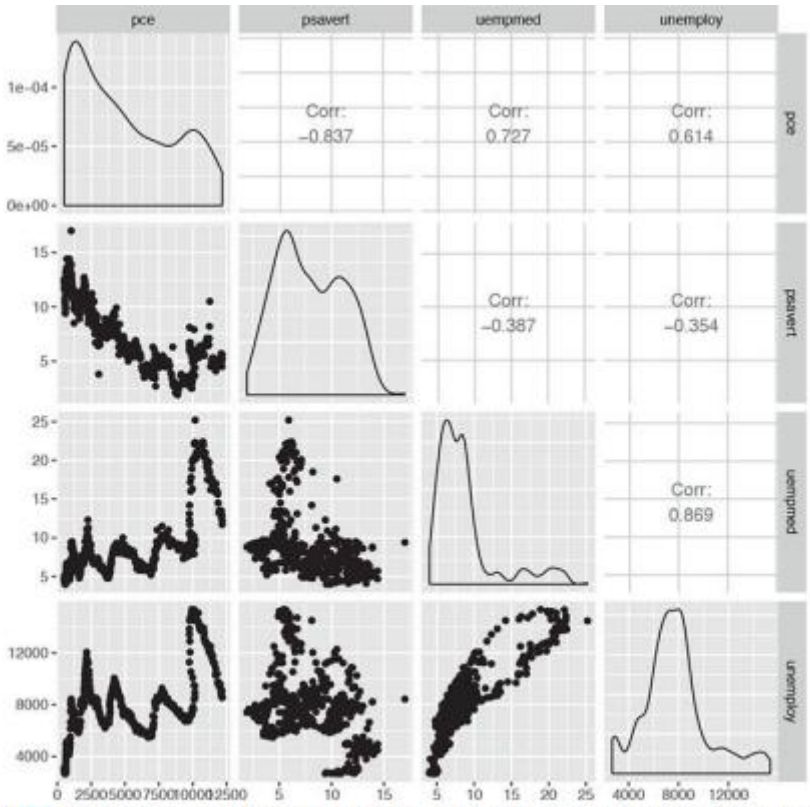


Figure 18.1 Pairs plot of economics data showing the relationship between each pair of variables as a scatterplot with the correlations printed as numbers.

```
> GGally::ggpairs(economics[, c(2, 4:6)])
```

This is similar to a small multiples plot except that each pane has different x- and y-axes. While this shows the original data, it does not actually show the correlation. To show that we build a heatmap of the correlation numbers, as shown in Figure 18.2. High positive correlation indicates a positive relationship between the variables, high negative correlation indicates a negative relationship between the variables and near zero correlation indicates no strong relationship.

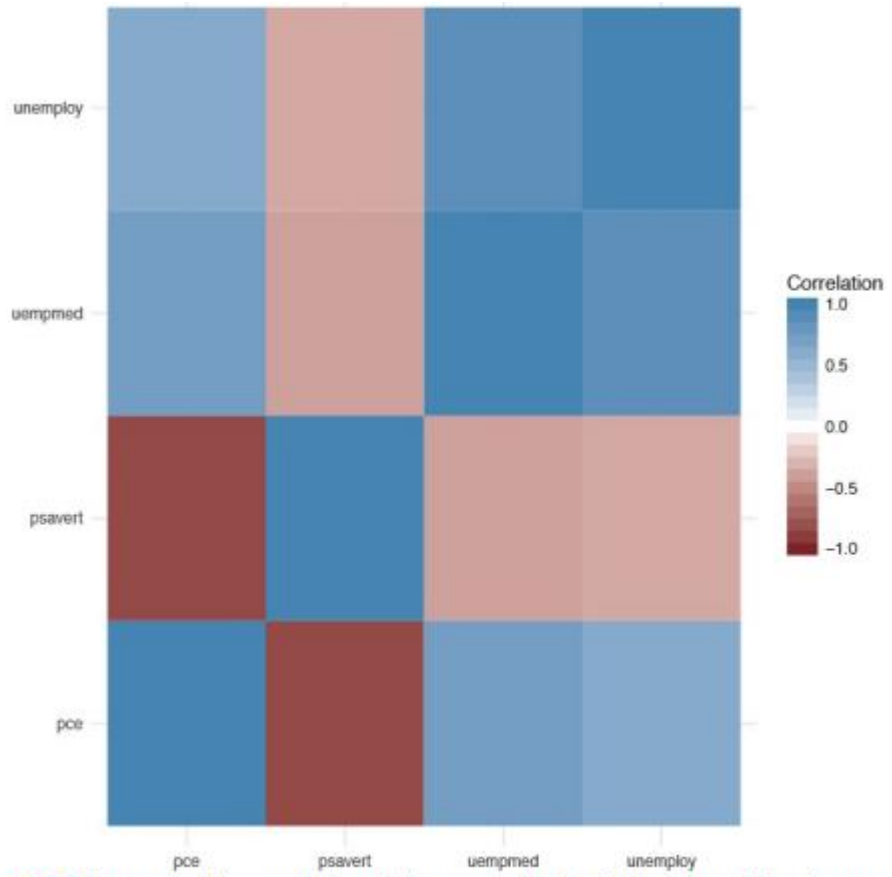


Figure 18.2 Heatmap of the correlation of the economics data. The diagonal has elements with correlation 1 because every element is perfectly correlated with itself. Red indicates highly negative correlation, blue indicates highly positive correlation and white is no correlation.

```
> # load the reshape package for melting the data
> library(reshape2)
> # load the scales package for some extra plotting features
> library(scales)
> # build the correlation matrix
> econCor <- cor(economics[, c(2, 4:6)])
> # melt it into the long format
> econMelt <- melt(econCor, varnames=c("x", "y"), value.name="Correlation")
> # order it according to the correlation
> econMelt <- econMelt[order(econMelt$Correlation), ]
> # display the melted data
> econMelt
x y Correlation
2 psavert pce -0.8370690
```

```
5 pce psavert -0.8370690
7 uempmed psavert -0.3874159
10 psavert uempmed -0.3874159
8 unemploy psavert -0.3540073
14 psavert unemploy -0.3540073
4 unemploy pce 0.6139997
13 pce unemploy 0.6139997
3 uempmed pce 0.7273492
9 pce uempmed 0.7273492
12 unemploy uempmed 0.8694063
15 uempmed unemploy 0.8694063
1 pce pce 1.0000000
6 psavert psavert 1.0000000
11 uempmed uempmed 1.0000000
16 unemploy unemploy 1.0000000
> ## plot it with ggplot
> # initialize the plot with x and y on the x and y axes
> ggplot(econMelt, aes(x=x, y=y)) +
+ # draw tiles filling the color based on Correlation
+ geom_tile(aes(fill=Correlation)) +
+ # make the fill (color) scale a three color gradient with muted
+ # red for the low point, white for the middle and steel blue
+ # for the high point
+ # the guide should be a colorbar with no ticks, whose height is
+ # 10 lines
+ # limits indicates the scale should be filled from -1 to 1
+ scale_fill_gradient2(low=muted("red"), mid="white",
+ high="steelblue",
+ guide=guide_colorbar(ticks=FALSE, barheight=10),
+ limits=c(-1, 1)) +
+ # use the minimal theme so there are no extras in the plot
```

+ theme_minimal() +

+ # make the x and y labels blank

+ labs(x=NULL, y=NULL)

Missing data is just as much a problem with cor as it is with mean and var, but is dealt with differently because multiple columns are being considered simultaneously. Instead of specifying na.rm=TRUE to remove NA entries, one of “all.obs”, “complete.obs”, “pairwise.complete.obs”, “everything” or “na.or.complete” is used. To illustrate this we first make a five-column matrix where only the fourth and fifth columns have no NA values; the other columns have one or two NAs.

> m <- c(9, 9, NA, 3, NA, 5, 8, 1, 10, 4)

> n <- c(2, NA, 1, 6, 6, 4, 1, 1, 6, 7)

> p <- c(8, 4, 3, 9, 10, NA, 3, NA, 9, 9)

> q <- c(10, 10, 7, 8, 4, 2, 8, 5, 5, 2)

> r <- c(1, 9, 7, 6, 5, 6, 2, 7, 9, 10)

> # combine them together

> theMat <- cbind(m, n, p, q, r)

The first option for use is “everything”, which means that the entirety of all columns must be free of NAs; otherwise the result is NA. Running this should generate a matrix of all NAs except ones on the diagonal—because a vector is always perfectly correlated with itself—and between q and r. With the second option —“all.obs”—even a single NA in any column will cause an error.

> cor(theMat, use="everything")

m n p q r

m 1 NA NA NA NA

n NA 1 NA NA NA

p NA NA 1 NA NA

q NA NA NA 1.0000000 -0.4242958

r NA NA NA -0.4242958 1.0000000

> cor(theMat, use="all.obs")

Error in cor(theMat, use = "all.obs"): missing observations in cov/cor

The third and fourth options—“complete.obs” and “na.or.complete”—work similarly to each other in that they keep only rows where every entry is not NA. That means our matrix will be reduced to rows 1, 4, 7, 9 and 10, and then have its correlation computed. The difference is that “complete.obs” will return an error if not a single complete row can be found, while “na.or.complete” will return NA in that case.

> cor(theMat, use="complete.obs")

m n p q r

```
m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470
n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718
p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470
q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384
r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000
> cor(theMat, use="na.or.complete")
m n p q r
m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470
n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718
p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470
q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384
r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000
> # calculate the correlation just on complete rows
> cor(theMat[c(1, 4, 7, 9, 10), ])
m n p q r
m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470
n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718
p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470
q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384
r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000
> # compare "complete.obs" and computing on select rows
> # should give the same result
> identical(cor(theMat, use="complete.obs"),
+ cor(theMat[c(1, 4, 7, 9, 10), ]))
[1] TRUE
```

The final option is “pairwise.complete”, which is much more inclusive. It compares two columns at a time and keeps rows—for those two columns—where neither entry is NA. This is essentially the same as computing the correlation between every combination of two columns with use set to “complete.obs”.

```
> # the entire correlation matrix
> cor(theMat, use="pairwise.complete.obs")
m n p q r
```

m 1.00000000 -0.02511812 -0.3965859 0.4622943 -0.2001722

n -0.02511812 1.00000000 0.8717389 -0.5070416 0.5332259

p -0.39658588 0.87173889 1.0000000 -0.5197292 0.1312506

q 0.46229434 -0.50704163 -0.5197292 1.0000000 -0.4242958

r -0.20017222 0.53322585 0.1312506 -0.4242958 1.0000000

> # compare the entries for m vs n to this matrix

> cor(theMat[, c("m", "n")], use="complete.obs")

m n

m 1.00000000 -0.02511812

n -0.02511812 1.00000000

> # compare the entries for m vs p to this matrix

> cor(theMat[, c("m", "p")], use="complete.obs")

m p

m 1.0000000 -0.3965859

p -0.3965859 1.0000000

To see ggpairs in all its glory, look at tips data from the reshape2 package in Figure 18.3. This shows every pair of variables in relation to each other building either histograms, boxplots or scatterplots depending on the combination of continuous and discrete variables. While a data dump like this looks really nice, it is not always the most informative form of exploratory data analysis.

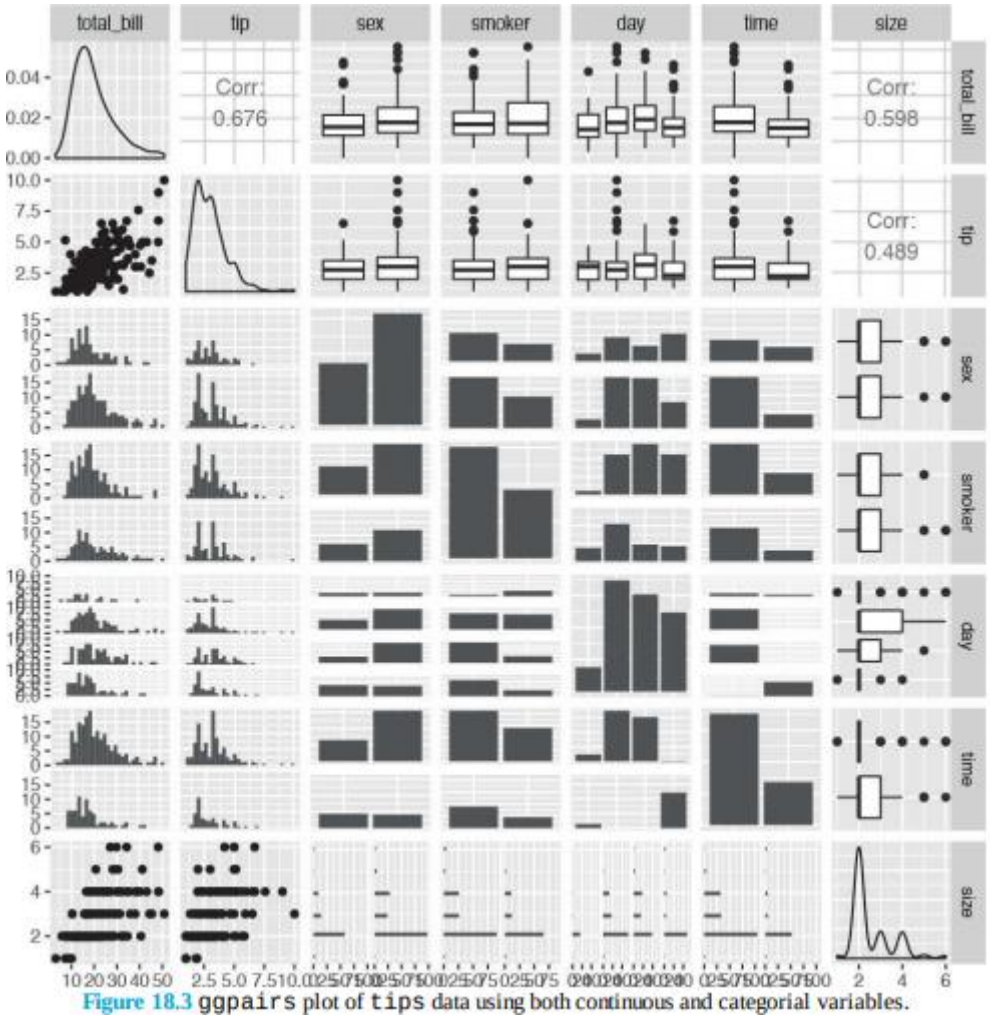


Figure 18.3 ggpairs plot of tips data using both continuous and categorial variables.

> data(tips, package="reshape2")

> head(tips)

total_bill tip sex smoker day time size

1 16.99 1.01 Female No Sun Dinner 2

2 10.34 1.66 Male No Sun Dinner 3

3 21.01 3.50 Male No Sun Dinner 3

4 23.68 3.31 Male No Sun Dinner 2

5 24.59 3.61 Female No Sun Dinner 4

6 25.29 4.71 Male No Sun Dinner 4

> GGally::ggpairs(tips)

No discussion of correlation would be complete without the old refrain, “Correlation does not mean causation.” In other words, just because two variables are correlated does not mean they have an effect on each other. This is exemplified in xkcd 1 comic number 552. There is even an R package, RXKCD, for downloading individual comics. Running the following code should generate a pleasant surprise.

> library(RXKCD) > getXKCD(which="552") Similar to correlation is covariance, which is like a variance between variables, its formula is in Equation 18.5. Notice the similarity to correlation in Equation 18.4 and variance in Equation 18.3.

$$cov(X,Y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \quad (18.5)$$

The cov function works similarly to the cor function, with the same arguments for dealing with missing data. In fact, ?cor and ?cov pull up the same help menu.

> cov(economics$pce, economics$psavert)

[1] -9361.028

> cov(economics[, c(2, 4:6)])

pce psavert uempmed unemploy

pce 12811296.900 -9361.028324 10695.023873 5806187.162

psavert -9361.028 9.761835 -4.972622 -2922.162

uempmed 10695.024 -4.972622 16.876582 9436.074

unemploy 5806187.162 -2922.161618 9436.074287 6979955.661

> # check that cov and cor*sd*sd are the same

> identical(cov(economics$pce, economics$psavert),

+ cor(economics$pce, economics$psavert) *

+ sd(economics$pce) * sd(economics$psavert))

[1] TRUE

T-Tests

In traditional statistics classes, the t-test—invented by William Gosset while working at the Guinness brewery—is taught for conducting tests on the mean of data or for comparing two sets of data. To illustrate this we continue to use the tips data from Section 18.2.

> head(tips)

total_bill tip sex smoker day time size

1 16.99 1.01 Female No Sun Dinner 2

2 10.34 1.66 Male No Sun Dinner 3

3 21.01 3.50 Male No Sun Dinner 3

4 23.68 3.31 Male No Sun Dinner 2

5 24.59 3.61 Female No Sun Dinner 4

6 25.29 4.71 Male No Sun Dinner 4

> # sex of the bill payer

> unique(tips$sex)

[1] Female Male

Levels: Female Male

> # day of the week

> unique(tips$day)

[1] Sun Sat Thur Fri

Levels: Fri Sat Sun Thur

One-Sample T-Test

First we conduct a one-sample t-test on whether the average tip is equal to $2.50. This test essentially calculates the mean of data and builds a confidence interval. If the value we are testing falls within that confidence interval, then we can conclude that it is the true value for the mean of the data; otherwise, we conclude that it is not the true mean.

> t.test(tips$tip, alternative="two.sided", mu=2.50)

One Sample t-test

data: tips$tip

t = 5.6253, df = 243, p-value = 5.08e-08

alternative hypothesis: true mean is not equal to 2.5

95 percent confidence interval:

2.823799 3.172758

sample estimates:

mean of x

2.998279

The output very nicely displays the setup and results of the hypothesis test of whether the mean is equal to $2.50. It prints the t-statistic, the degrees of freedom and p-value. It also provides the 95 percent confidence interval and mean for the variable of interest. The p-value indicates that the null hypothesis 2 should be rejected, and we conclude that the mean is not equal to $2.50. 2. The null hypothesis is what is considered to be true; in this case that the mean is equal to $2.50. We encountered a few new concepts here. The t-statistic is the ratio where the numerator is the difference between the estimated mean and the hypothesized mean and the denominator is the standard error of the estimated mean. It is defined in Equation 18.6.

$$\text{t-statistic} = \frac{(\bar{x} - \mu_0)}{s_{\bar{x}}/\sqrt{n}} \qquad (18.6)$$

Here, is the estimated mean, μ0 is the hypothesized mean and is the standard error of . 3 3. s is the standard deviation of the data and n is the number of observations. If the hypothesized mean is correct, then we expect the t-statistic to fall somewhere in the middle— about two standard deviations from the mean—of the t distribution. In Figure 18.4 we see that the thick black line, which represents the estimated mean, falls so far outside the distribution that we must conclude that the mean is not equal to $2.50.

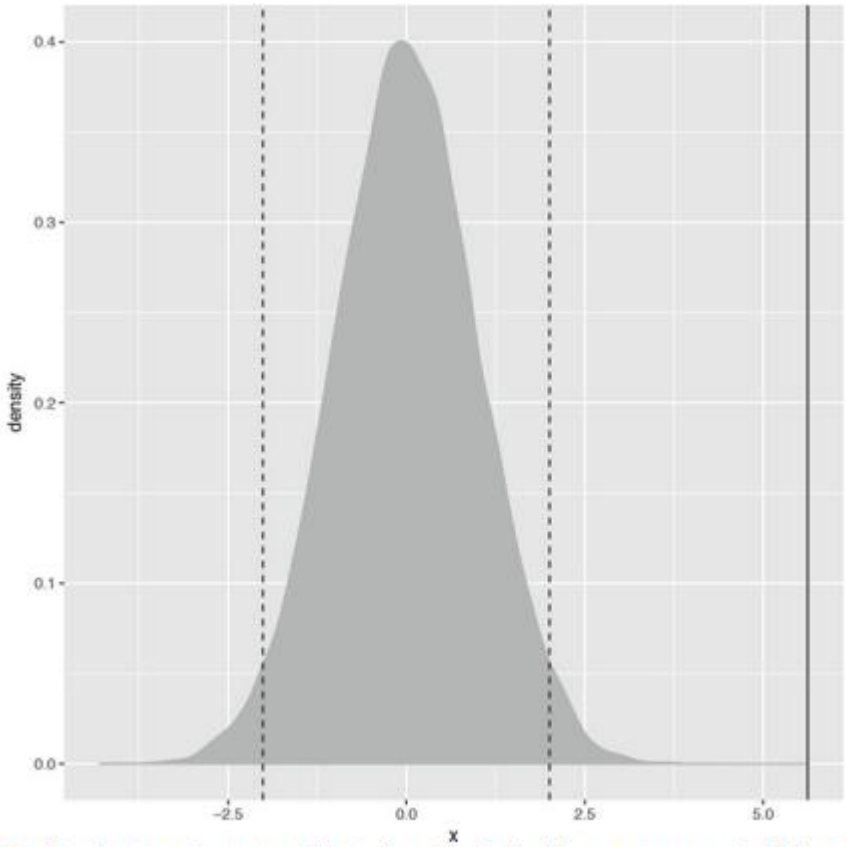


Figure 18.4 t-distribution and t-statistic for tip data. The dashed lines are two standard deviations from the mean in either direction. The thick black line, the t-statistic, is so far outside the distribution that we must reject the null hypothesis and conclude that the true mean is not $2.50.

```
> ## build a t distribution
> randT <- rt(30000, df=NROW(tips)-1)
>
> # get t-statistic and other information
> tipTTest <- t.test(tips$tip, alternative="two.sided", mu=2.50)
>
> # plot it
> ggplot(data.frame(x=randT)) +
+ geom_density(aes(x=x), fill="grey", color="grey") +
+ geom_vline(xintercept=tipTTest$statistic) +
+ geom_vline(xintercept=mean(randT) + c(-2, 2)*sd(randT), linetype=2)
```

The p-value is an often misunderstood concept. Despite all the misinterpretations, a p-value is the probability, if the null hypothesis were correct, of getting as extreme, or more extreme, a result. It is a measure of how extreme the statistic—in this case, the estimated mean— is. If the statistic is too extreme, we conclude that the null hypothesis should be rejected. The main problem with p-values, however, is determining what should be considered too extreme. Ronald A. Fisher, the father of modern statistics, decided we should consider a p-value that is smaller than 0.10, 0.05 or 0.01 to be too extreme. While those p-values have been the standard for decades, they were arbitrarily chosen, leading some modern data scientists to question their usefulness. In this example, the p-value is 5.0799885×10^{-8}; this is smaller than 0.01, so we reject the null hypothesis. Degrees of freedom is another difficult concept to grasp but is pervasive throughout statistics. It represents the effective number of observations. Generally, the degrees of freedom for some statistic or distribution is the number of observations minus the number of parameters being estimated. In the case of the t distribution, one parameter, the standard error, is being estimated. In this example, there are nrow(tips)-1=243 degrees of freedom. Next we conduct a one-sided t-test to see if the mean is greater than $2.50.

```
> t.test(tips$tip, alternative="greater", mu=2.50)

One Sample t-test

data: tips$tip

t = 5.6253, df = 243, p-value = 2.54e-08

alternative hypothesis: true mean is greater than 2.5

95 percent confidence interval:

2.852023 Inf

sample estimates:

mean of x

2.998279
```

Once again, the p-value indicates that we should reject the null hypothesis and conclude that the mean is greater than $2.50, which coincides nicely with the confidence interval.

Two-Sample T-Test

More often than not the t-test is used for comparing two samples. Continuing with the tips data, we compare how female and male diners tip. Before running the t-test, however, we first need to check the variance of each sample. A traditional t-test requires both groups to have the same variance, whereas the Welch two-sample t-test can handle groups with differing variances. We explore this both numerically and visually in Figure 18.5.

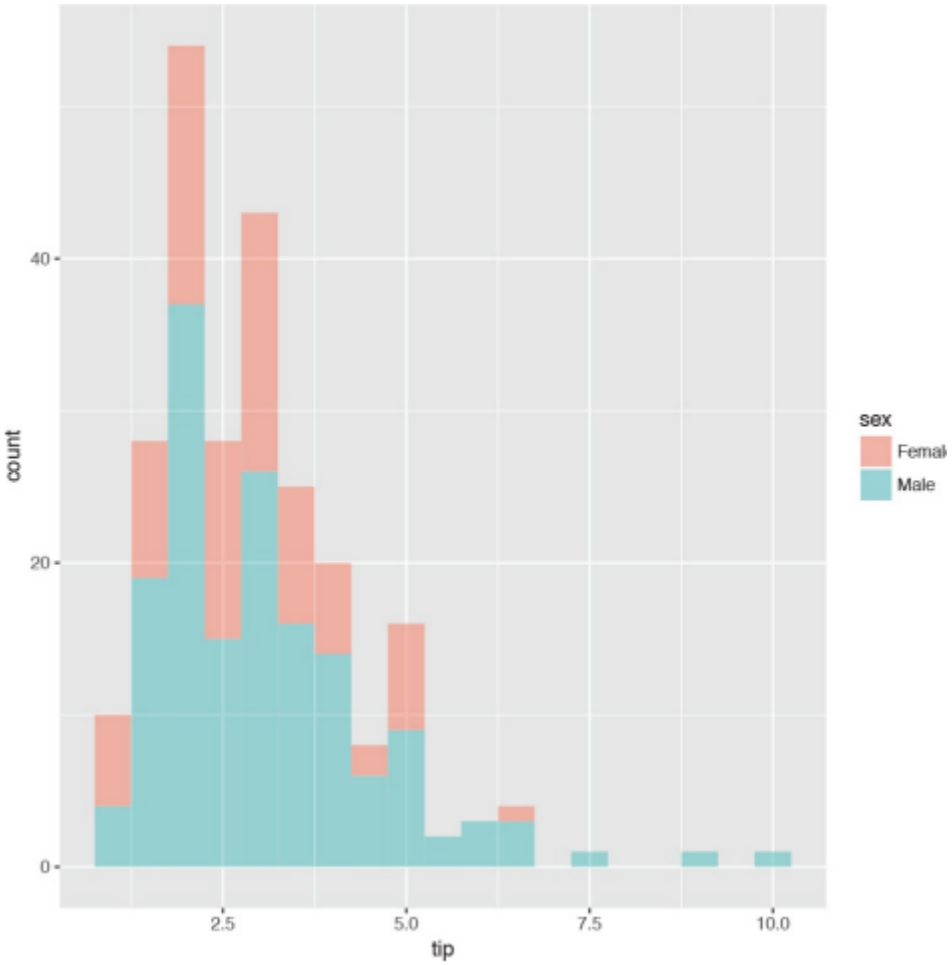


Figure 18.5 Histogram of tip amount by sex. Note that neither distribution appears to be normal.

```
> # first just compute the variance for each group
> # using the the formula interface
> # calculate the variance of tip for each level of sex
> aggregate(tip ~ sex, data=tips, var)
sex tip
1 Female 1.344428
2 Male 2.217424
> # now test for normality of tip distribution
> shapiro.test(tips$tip)
Shapiro-Wilk normality test
data: tips$tip
W = 0.89781, p-value = 8.2e-12
> shapiro.test(tips$tip[tips$sex == "Female"])
Shapiro-Wilk normality test
```

data: tips$tip[tips$sex == "Female"]

W = 0.95678, p-value = 0.005448

> shapiro.test(tips$tip[tips$sex == "Male"])

Shapiro-Wilk normality test

data: tips$tip[tips$sex == "Male"]

W = 0.87587, p-value = 3.708e-10

> # all the tests fail so inspect visually

> ggplot(tips, aes(x=tip, fill=sex)) +

+ geom_histogram(binwidth=.5, alpha=1/2)

Since the data do not appear to be normally distributed, neither the standard F-test (via the var.test function) nor the Bartlett test (via the bartlett.test function) will suffice. So we use the nonparametric Ansari-Bradley test to examine the equality of variances.

> ansari.test(tip ~ sex, tips)

Ansari-Bradley test

data: tip by sex

AB = 5582.5, p-value = 0.376

alternative hypothesis: true ratio of scales is not equal to 1

This test indicates that the variances are equal, meaning we can use the standard two-sample t-test.

> # setting var.equal=TRUE runs a standard two sample t-test

> # var.equal=FALSE (the default) would run the Welch test

> t.test(tip ~ sex, data=tips, var.equal=TRUE)

Two Sample t-test

data: tip by sex

t = -1.3879, df = 242, p-value = 0.1665

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-0.6197558 0.1074167

sample estimates:

mean in group Female mean in group Male

2.833448 3.089618

According to this test, the results were not significant, and we should conclude that female and male diners tip roughly equally. While all this statistical rigor is nice, a simple rule of thumb would be to see if the two means are within two standard deviations of each other.

```
> library(plyr)
> tipSummary <- ddply(tips, "sex", summarize,
+ tip.mean=mean(tip), tip.sd=sd(tip),
+ Lower=tip.mean - 2*tip.sd/sqrt(NROW(tip)),
+ Upper=tip.mean + 2*tip.sd/sqrt(NROW(tip)))
> tipSummary
sex tip.mean tip.sd Lower Upper
1 Female 2.833448 1.159495 2.584827 3.082070
2 Male 3.089618 1.489102 2.851931 3.327304
```

A lot happened in that code. First, ddply was used to split the data according to the levels of sex. It then applied the summarize function to each subset of the data. This function applied the indicated functions to the data, creating a new data.frame. As usual, we prefer visualizing the results rather than comparing numerical values. This requires reshaping the data a bit. The results, in Figure 18.6, clearly show the confidence intervals overlapping, suggesting that the means for the two sexes are roughly equivalent.

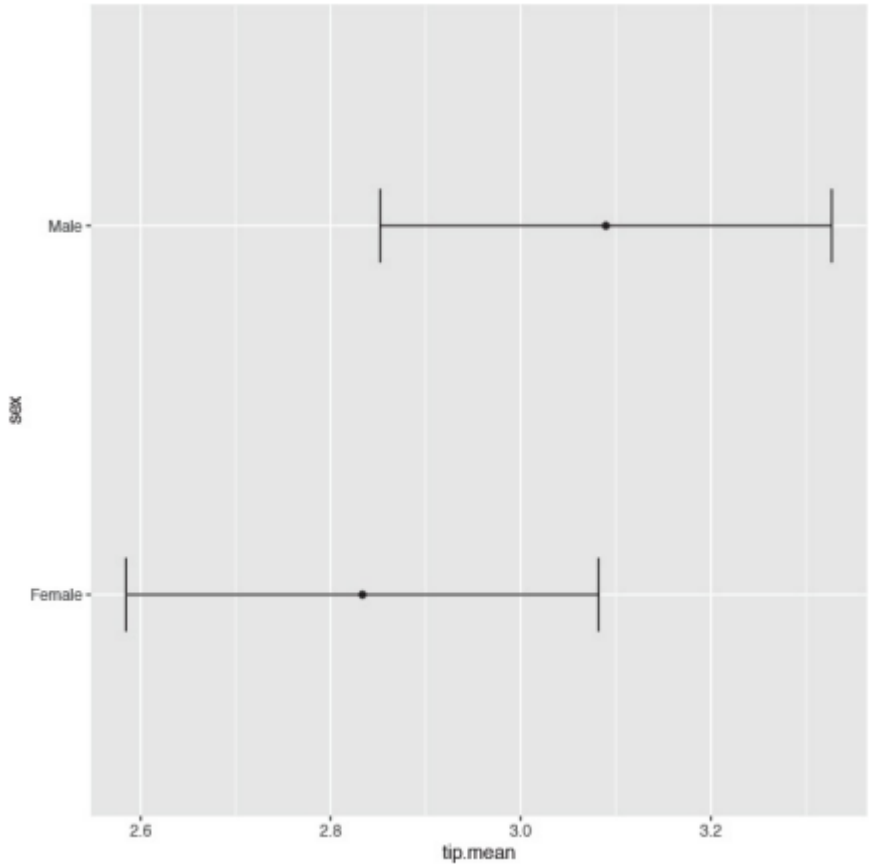


Figure 18.6 Plot showing the mean and two standard errors of tips broken down by the sex of the diner.

```
> ggplot(tipSummary, aes(x=tip.mean, y=sex)) + geom_point() +
+ geom_errorbarh(aes(xmin=Lower, xmax=Upper), height=.2)
```

ANOVA

After comparing two groups, the natural next step is comparing multiple groups. Every year, far too many students in introductory statistics classes are forced to learn the ANOVA (analysis of variance) test and memorize its formula, which is

$$F = \frac{\sum_i n_i(\overline{Y}_i - \overline{Y})^2/(K-1)}{\sum_{ij}(Y_{ij} - \overline{Y}_i)^2/(N-K)} \qquad (18.7)$$

where ni is the number of observations in group i, i is the mean of group i, is the overall mean, Yij is observation j in group i, N is the total number of observations and K is the number of groups. Not only is this a laborious formula that often turns off a lot of students from statistics; it is also a bit of an old-fashioned way of comparing groups. Even so, there is an R function—albeit rarely used—to conduct the ANOVA test. This also uses the formula interface where the left side is the variable of interest and the right side contains the variables that control grouping. To see this, we compare tips by day of the week, with levels Fri, Sat, Sun, Thur.

> tipAnova <- aov(tip ~ day - 1, tips)

In the formula the right side was day - 1. This might seem odd at first but will make more sense when comparing it to a call without -1.

> tipIntercept <- aov(tip ~ day, tips)

> tipAnova$coefficients

dayFri daySat daySun dayThur

2.734737 2.993103 3.255132 2.771452

> tipIntercept$coefficients

(Intercept) daySat daySun dayThur

2.73473684 0.25836661 0.52039474 0.03671477

Here we see that just using tip ~ day includes only Saturday, Sunday and Thursday, along with an intercept, while tip ~ day - 1 compares Friday, Saturday, Sunday and Thursday with no intercept. The importance of the intercept is made clear in Chapter 19, but for now it suffices that having no intercept makes the analysis more straightforward. The ANOVA tests whether any group is different from any other group but it does not specify which group is different. So printing a summary of the test just returns a single p-value.

> summary(tipAnova)

Df Sum Sq Mean Sq F value Pr(>F)

day 4 2203.0 550.8 290.1 <2e-16 ***

Residuals 240 455.7 1.9

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Since the test had a significant p-value, we would like to see which group differed from the others. The simplest way is to make a plot of the group means and confidence intervals and see which overlap. Figure 18.8 shows that tips on Sunday differ (just barely, at the 90 percent confidence level) from both Thursday and Friday

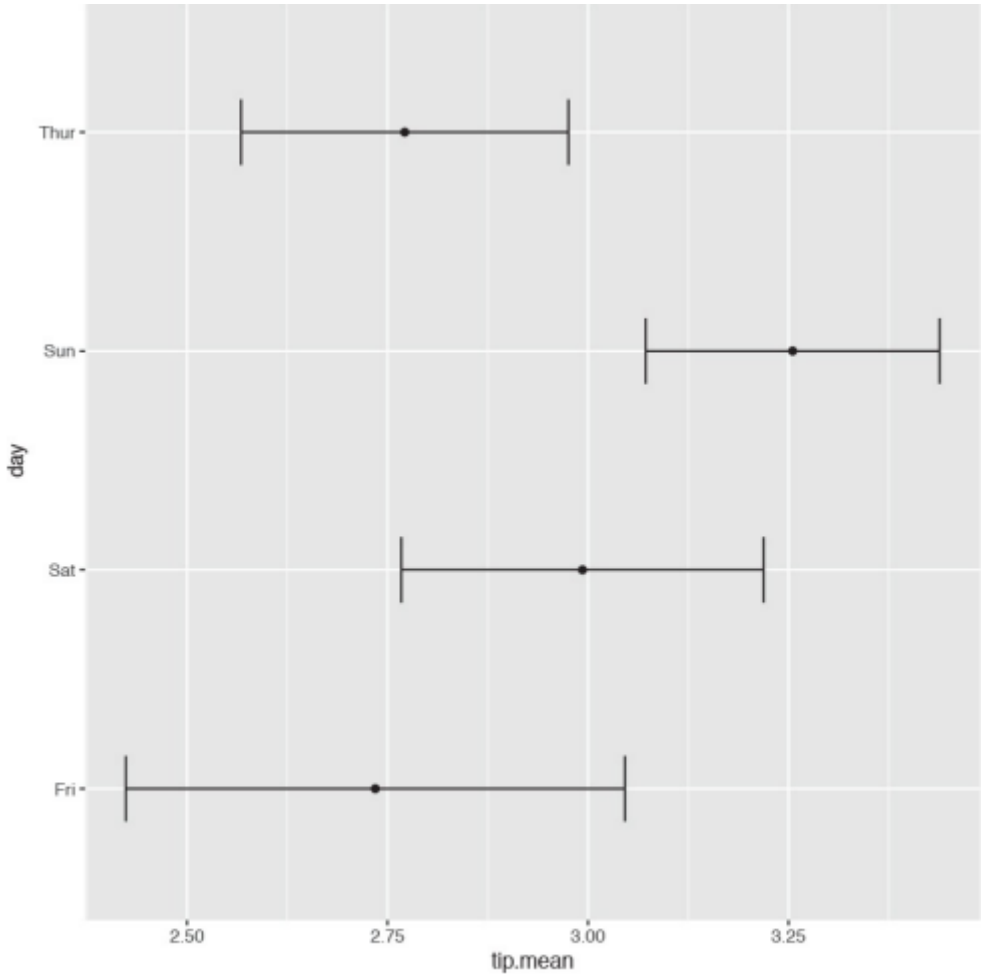


Figure 18.8 Means and confidence intervals of tips by day. This shows that Sunday tips differ from Thursday and Friday tips.

```
> tipsByDay <- ddply(tips, "day", plyr::summarize,
+ tip.mean=mean(tip), tip.sd=sd(tip),
+ Length=NROW(tip),
+ tfrac=qt(p=.90, df=Length-1),
+ Lower=tip.mean - tfrac*tip.sd/sqrt(Length),
+ Upper=tip.mean + tfrac*tip.sd/sqrt(Length)
+ )
>
> ggplot(tipsByDay, aes(x=tip.mean, y=day)) + geom_point() +
```

+ geom_errorbarh(aes(xmin=Lower, xmax=Upper), height=.3)

The use of NROW instead of nrow is to guarantee computation. Where nrow works only on data.frames and matrices, NROW returns the length of objects that have only one dimension.

```
> nrow(tips)

[1] 244

> NROW(tips)

[1] 244

> nrow(tips$tip)

NULL

> NROW(tips$tip)

[1] 244
```

To confirm the results from the ANOVA, individual t-tests could be run on each pair of groups. Traditional texts encourage adjusting the p-value to accommodate the multiple comparisons. However, some professors, including Andrew Gelman, suggest not worrying about adjustments for multiple comparisons. An alternative to the ANOVA is to fit a linear regression with one categorical variable and no intercept.

Linear Models

The workhorse of statistical analysis is the linear model, particularly regression. Originally invented by Francis Galton to study the relationships between parents and children, which he described as regressing to the mean, it has become one of the most widely used modelling techniques and has spawned other models such as generalized linear models, regression trees, penalized regression and many others. In this chapter we focus on simple and multiple regression.

Simple Linear Regression

In its simplest form regression is used to determine the relationship between two variables. That is, given one variable, it tells us what we can expect from the other variable. This powerful tool, which is frequently taught and can accomplish a great deal of analysis with minimal effort, is called simple linear regression. Before we go any further, we clarify some terminology. The outcome variable (what we are trying to predict) is called the response, and the input variable (what we are using to predict) is the predictor. Fields outside of statistics use other terms, such as measured variable, outcome variable and experimental variable for response, and covariate, feature and explanatory variable for predictor. Worst of all are the terms dependent (response) and independent (predictor) variables. These very names are misnomers. According to probability theory, if variable y is dependent on variable x, then variable x cannot be independent of variable y. So we stick with the terms response and predictor exclusively. The general idea behind simple linear regression is using the predictor to come up with some average value of the response. The relationship is defined as

$$y = a + bx + \epsilon \qquad (19.1)$$

Where

$$b = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \quad (19.2)$$

$$a = \bar{y} - b \quad (19.3)$$

And

$$\epsilon \sim \mathcal{N}(0, \sigma^2) \quad (19.4)$$

which is to say that there are normally distributed errors.

Equation 19.1 is essentially describing a straight line that goes through the data where a is the yintercept and b is the slope. This is illustrated using fathers' and sons' height data, which are plotted in Figure 19.1. In this case we are using the fathers' heights as the predictor and the sons' heights as the response. The blue line running through the points is the regression line and the gray band around it represents the uncertainty in the fit.

> data(father.son, package='UsingR')

> library(ggplot2)

> head(father.son)

fheight sheight

1 65.04851 59.77827

2 63.25094 63.21404

3 64.95532 63.34242

4 65.75250 62.79238

5 61.13723 64.28113

6 63.02254 64.24221

> ggplot(father.son, aes(x=fheight, y=sheight)) + geom_point() + + geom_smooth(method="lm") + labs(x="Fathers", y="Sons")

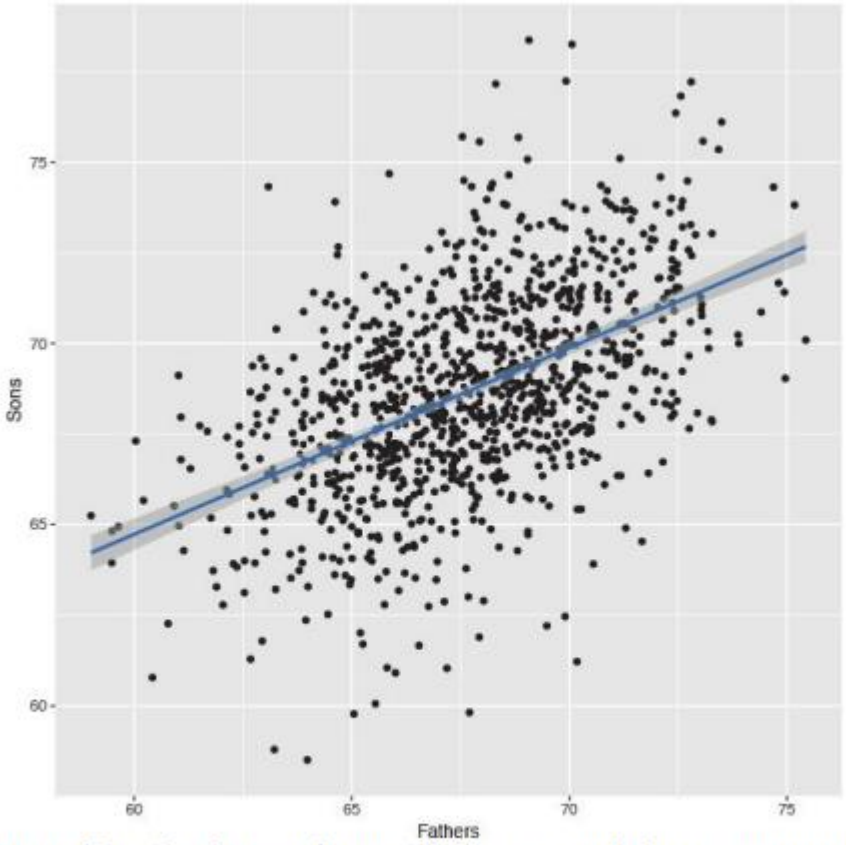


Figure 19.1 Using fathers' heights to predict sons' heights using simple linear regression. The fathers' heights are the predictors and the sons' heights are the responses. The blue line running through the points is the regression line and the gray band around it represents the uncertainty in the fit.

While that code generated a nice graph showing the results of the regression (generated with geom_smooth(method="lm")), it did not actually make those results available to us. To actually calculate a regression, use the lm function.

> heightsLM <- lm(sheight ~ fheight, data=father.son)

> heightsLM

Call:

lm(formula = sheight ~ fheight, data = father.son)

Coefficients:

(Intercept) fheight

33.8866 0.5141

Here we once again see the formula notation that specifies to regress sheight (the response) on fheight (the predictor), using the father.son data, and adds the intercept term automatically. The results show coefficients for (Intercept) and fheight which is the slope for the fheight, predictor. The interpretation of this is that, for every extra inch of height in a father, we expect an extra half inch in height for his son. The intercept in this case does not make much sense because it represents the height of a son whose father had zero height, which obviously cannot exist in reality. While the point estimates for the coefficients are nice, they are not very helpful without the standard errors, which give the sense of

uncertainty about the estimate and are similar to standard deviations. To quickly see a full report on the model, use summary.

```
> summary(heightsLM)

Call:

lm(formula = sheight ~ fheight, data = father.son)

Residuals:

Min 1Q Median 3Q Max

-8.8772 -1.5144 -0.0079 1.6285 8.9685

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 33.88660 1.83235 18.49 <2e-16 ***

fheight 0.51409 0.02705 19.01 <2e-16 ***

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.437 on 1076 degrees of freedom

Multiple R-squared: 0.2513,Adjusted R-squared: 0.2506

F-statistic: 361.2 on 1 and 1076 DF, p-value: < 2.2e-16
```

This prints out a lot more information about the model, including the standard errors, t-test values and p-values for the coefficients, the degrees of freedom, residual summary statistics (seen in more detail in Section 21.1) and the results of an F-test. This is all diagnostic information to check the fit of the model, and is covered in more detail in Section 19.2 about multiple regression.

Multiple Regression

The logical extension of simple linear regression is multiple regression, which allows for multiple predictors. The idea is still the same; we are still making predictions or inferences 1 on the response, but we now have more information in the form of multiple predictors. The math requires some matrix algebra but fortunately the lm function is used with very little extra effort.

In this case the relationship between the response and the p predictors (p − 1 predictors and the intercept) is modeled as

$$Y = X\beta + \epsilon \qquad (19.5)$$

$$\boldsymbol{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_n \end{bmatrix} \tag{19.6}$$

where Y is the nx1 response vector

X is the nxp matrix (n rows and p − 1 predictors plus the intercept)

$$\boldsymbol{X} = \begin{bmatrix} 1 & X_{11} & X_{12} & \dots & X_{1,p-1} \\ 1 & X_{21} & X_{22} & \dots & X_{2,p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & X_{n1} & X_{n2} & \dots & X_{n,p-1} \end{bmatrix} \tag{19.7}$$

β is the px1 vector of coefficients (one for each predictor and intercept)

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{bmatrix} \tag{19.8}$$

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \vdots \\ \epsilon_n \end{bmatrix} \tag{19.9}$$

and ε is the nx1 vector of normally distributed errors with

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I}) \tag{19.10}$$

which seems more complicated than simple regression but the algebra actually gets easier. The solution for the coefficients is simply written as in Equation 19.11.

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{Y} \tag{19.11}$$

To see this in action we use New York City condo evaluations for fiscal year 2011-2012, obtained through NYC Open Data. NYC Open Data is an initiative by New York City to make government more transparent and work better. It provides data on all manner of city services to the public for analysis, scrutiny and app building (through http://nycbigapps.com/). It has been surprisingly popular, spawning hundreds of mobile apps and being copied in other cities such as Chicago and Washington, DC.

Its Web site is at https://data.cityofnewyork.us/.

The original data were separated by borough with one file each for Manhattan, Brooklyn, Queens,

the Bronx and Staten Island, and contained extra information we will not be using. So we combined the five files into one, cleaned up the column names and posted it at

http://www.jaredlander.com/data/housing.csv. To access the data, either download it from that URLand use read.table on the now local file, or read it directly from the URL.

2.https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Manhattan/dvzp-h4k9

3. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Brooklyn-/bss9-579f

4. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Queens-FY/jcih-dj9q

5. https://data.cityofnewyork.us/Property/DOF-Condominium-Comparable-Rental-Income-Bronx-FY-/3qfc-4tta

6. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Staten-Is/tkdy-59zg

```
> housing <- read.table("http://www.jaredlander.com/data/housing.csv", + sep = ",", header = TRUE,
+ stringsAsFactors = FALSE)
```

A few reminders about what that code does: sep specifies that commas were used to separate columns; header means the first row contains the column names; and stringsAsFactors leaves character columns as they are and does not convert them to factors, which speeds up loading time and also makes them easier to work with. Looking at the data, we see that we have a lot of columns and some bad names, so we should rename those.

```
> names(housing) <- c("Neighborhood", "Class", "Units", "YearBuilt",
+ "SqFt", "Income", "IncomePerSqFt", "Expense",
+ "ExpensePerSqFt", "NetIncome", "Value",
+ "ValuePerSqFt", "Boro")
> head(housing)
```

Neighborhood Class Units YearBuilt SqFt Income

1 FINANCIAL R9-CONDOMINIUM 42 1920 36500 1332615

2 FINANCIAL R4-CONDOMINIUM 78 1985 126420 6633257

3 FINANCIAL RR-CONDOMINIUM 500 NA 554174 17310000

4 FINANCIAL R4-CONDOMINIUM 282 1930 249076 11776313

5 TRIBECA R4-CONDOMINIUM 239 1985 219495 10004582

6 TRIBECA R4-CONDOMINIUM 133 1986 139719 5127687

IncomePerSqFt Expense ExpensePerSqFt NetIncome Value

1 36.51 342005 9.37 990610 7300000

2 52.47 1762295 13.94 4870962 30690000

3 31.24 3543000 6.39 13767000 90970000

4 47.28 2784670 11.18 8991643 67556006

5 45.58 2783197 12.68 7221385 54320996

6 36.70 1497788 10.72 3629899 26737996

ValuePerSqFt Boro

1 200.00 Manhattan

2 242.76 Manhattan

3 164.15 Manhattan

4 271.23 Manhattan

5 247.48 Manhattan

6 191.37 Manhattan

For these data the response is the value per square foot and the predictors are everything else. However, we ignore the income and expense variables, as they are actually just estimates based on an arcane requirement that condos be compared to rentals for valuation purposes. The first step is to visualize the data in some exploratory data analysis. The natural place to start is with a histogram of ValuePerSqFt, which is shown in Figure 19.3.

> ggplot(housing, aes(x=ValuePerSqFt)) + + geom_histogram(binwidth=10) + labs(x="Value per Square Foot")

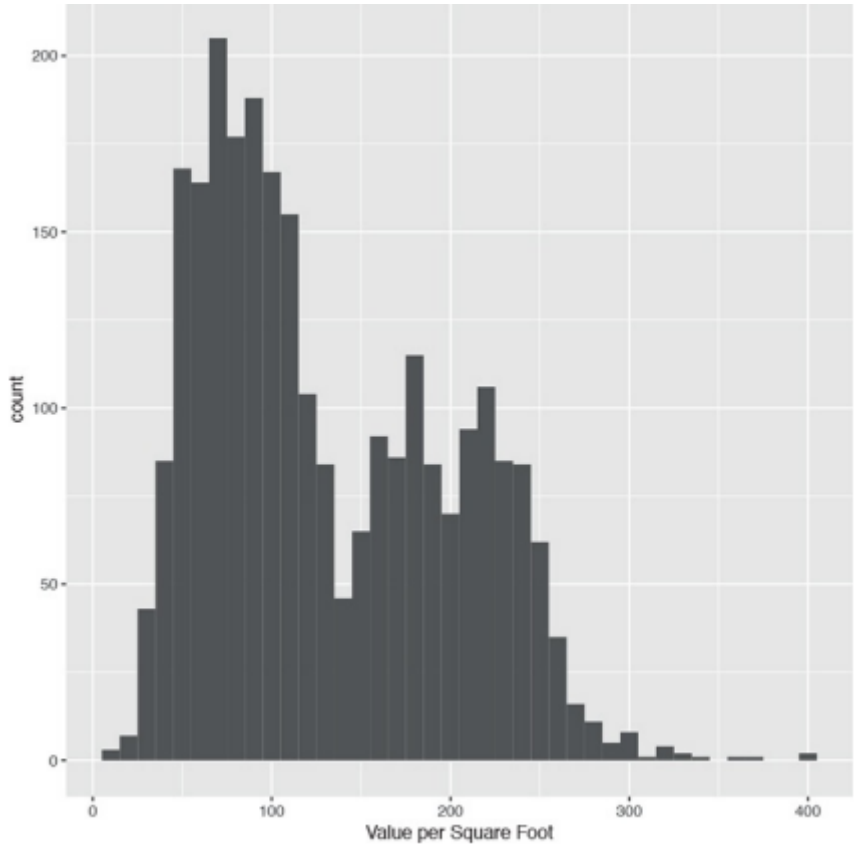
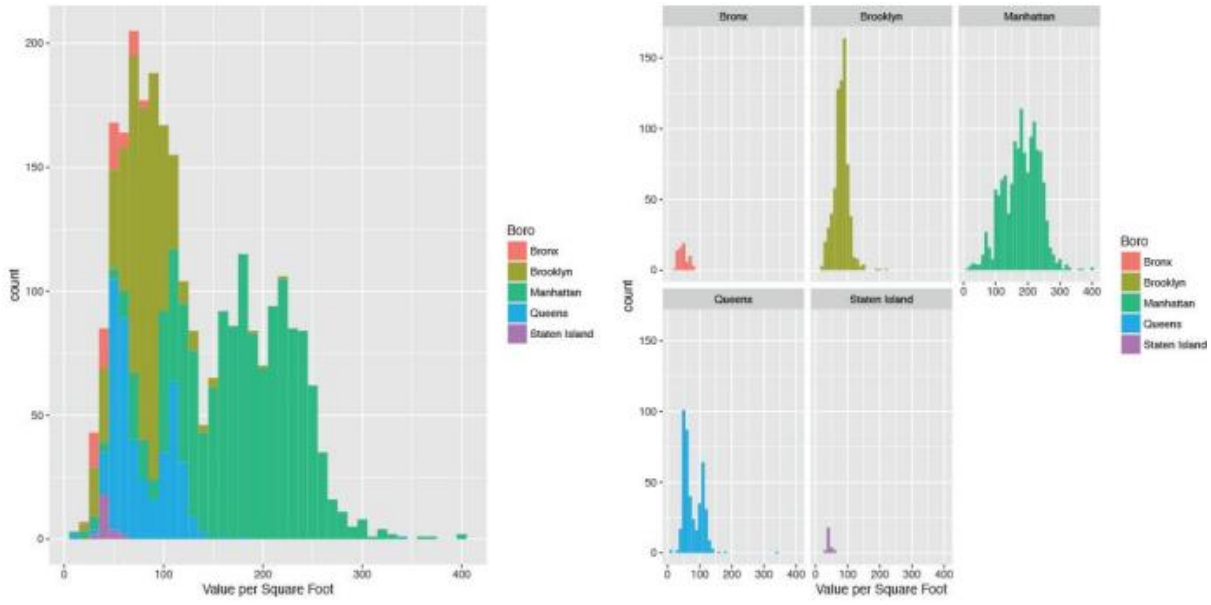


Figure 19.3 Histogram of value per square foot for NYC condos. It appears to be bimodal.

The bimodal nature of the histogram means there is something left to be explored. Mapping color to Boro in Figure 19.4a and faceting on Boro in Figure 19.4b reveal that Brooklyn and Queens make up one mode and Manhattan makes up the other, while there is not much data on the Bronx and Staten Island.

```
> ggplot(housing, aes(x=ValuePerSqFt, fill=Boro)) +
+ geom_histogram(binwidth=10) + labs
(x="Value per Square Foot")
> ggplot(housing, aes(x=ValuePerSqFt, fill=Boro)) +
+ geom_histogram(binwidth=10) + labs
(x="Value per Square Foot") +
+ facet_wrap(~Boro)
```



(a) Boros seperated by color (b) Boros separated by color and facet

Figure 19.4 Histograms of value per square foot. These illustrate structure in the data, revealing that Brooklyn and Queens make up one mode and Manhattan makes up the other, while there is not much data on the Bronx and Staten Island.

Next we should look at histograms for square footage and the number of units.

```
> ggplot(housing, aes(x=SqFt)) + geom_histogram()

> ggplot(housing, aes(x=Units)) + geom_histogram()

> ggplot(housing[housing$Units < 1000, ], aes(x=SqFt)) +

+ geom_histogram()

> ggplot(housing[housing$Units < 1000, ], aes(x=Units)) +

+ geom_histogram()
```

Figure 19.5 shows that there are quite a few buildings with an incredible number of units. Plotting scatterplots in Figure 19.6 of the value per square foot versus both number of units and square footage, with and without those outlying buildings, gives us an idea whether we can remove them from the analysis.

```
> ggplot(housing, aes(x=SqFt, y=ValuePerSqFt)) + geom_point()

> ggplot(housing, aes(x=Units, y=ValuePerSqFt)) + geom_point()

> ggplot(housing[housing$Units < 1000, ], aes(x=SqFt, y=ValuePerSqFt)) +

+ geom_point()

> ggplot(housing[housing$Units < 1000, ], aes(x=Units, y=ValuePerSqFt)) +

+ geom_point()
```

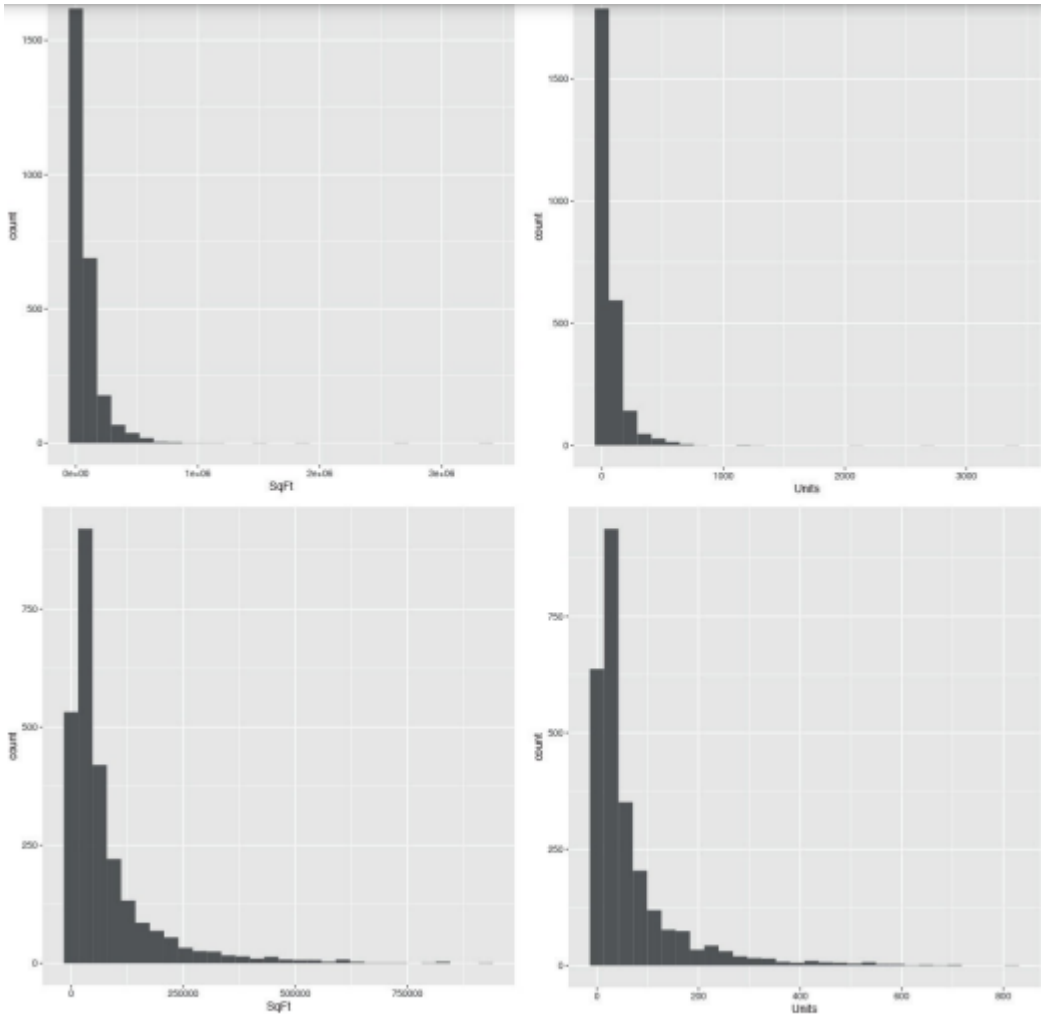


Figure 19.5 Histograms for total square feet and number of units. The distributions are highly right skewed in the top two graphs, so they were repeated after removing buildings with more than 1,000 units.

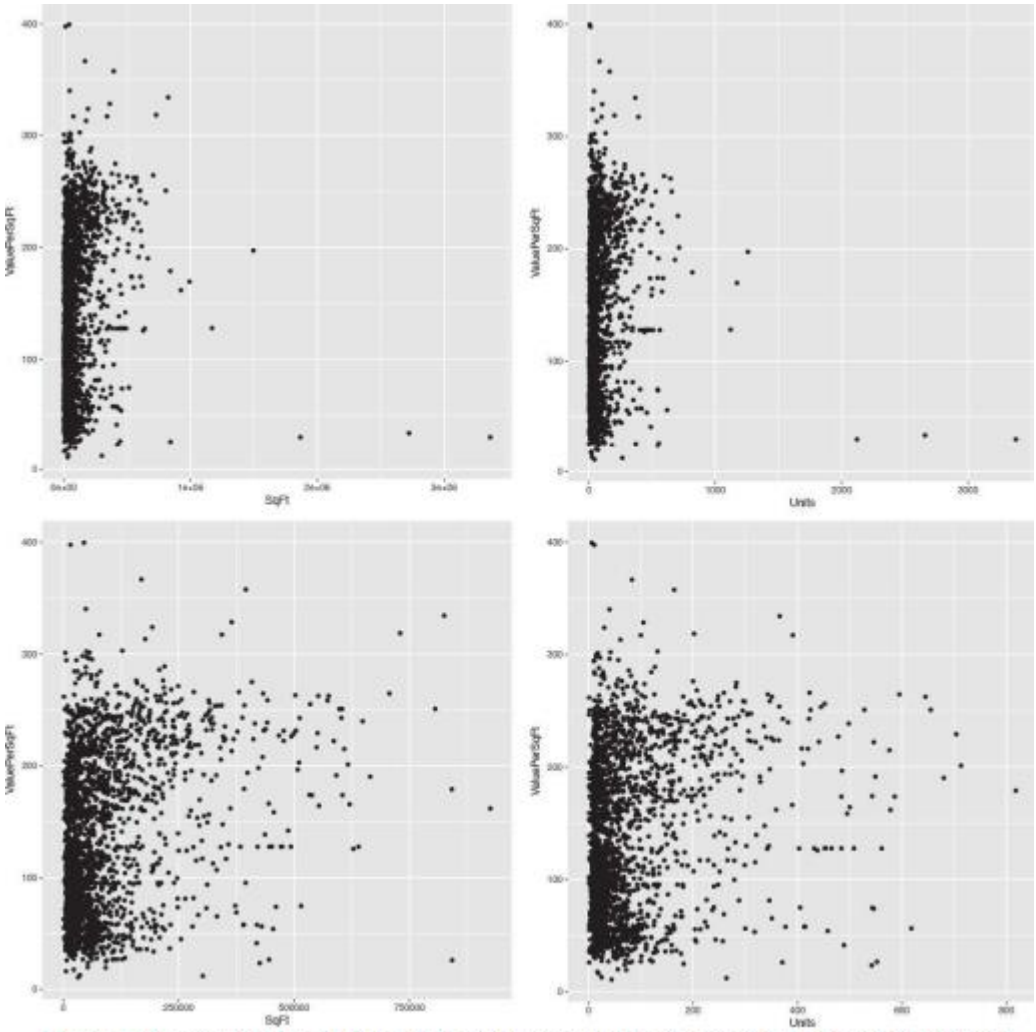


Figure 19.6 Scatterplots of value per square foot versus square footage and value versus number of units, both with and without the buildings that have over 1,000 units.

```
> # how many need to be removed?
> sum(housing$Units >= 1000)
[1] 6
> # remove them
> housing <- housing[housing$Units < 1000, ]
```

Even after we remove the outliers, it still seems like a log transformation of some data could be helpful. Figures 19.7 and 19.8 show that taking the log of square footage and number of units might prove helpful. It also shows what happens when taking the log of value.

```
> # plot ValuePerSqFt against SqFt
> ggplot(housing, aes(x=SqFt, y=ValuePerSqFt)) + geom_point()
> ggplot(housing, aes(x=log(SqFt), y=ValuePerSqFt)) + geom_point()
> ggplot(housing, aes(x=SqFt, y=log(ValuePerSqFt))) + geom_point()
> ggplot(housing, aes(x=log(SqFt), y=log(ValuePerSqFt))) +
```

+ geom_point()

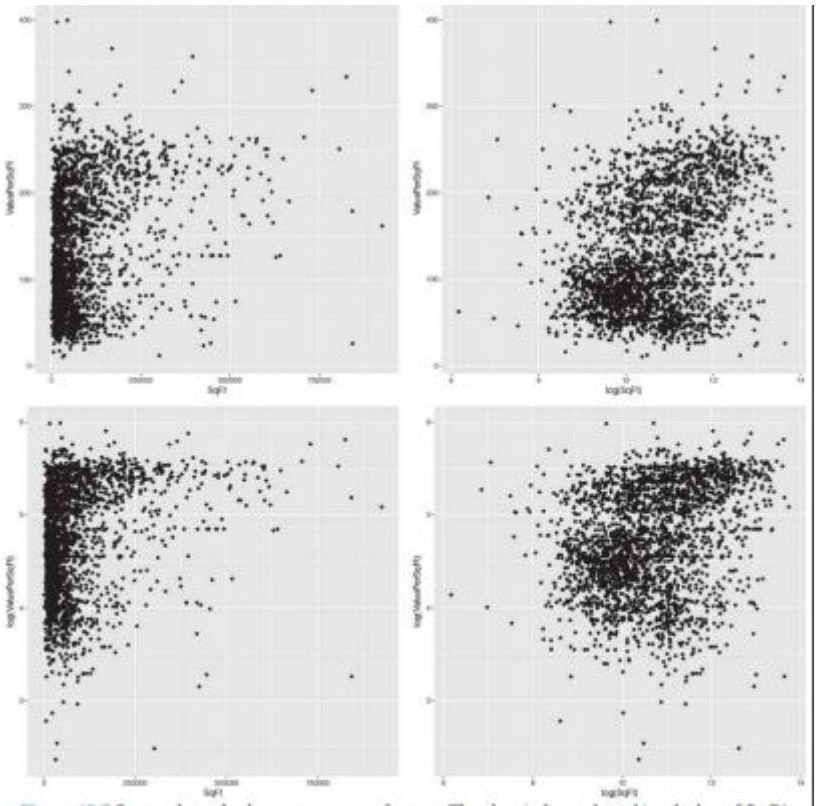


Figure 19.7 Scatterplots of value versus sqaure footage. The plots indicate that taking the log of SqFt might be useful in modelling.

> # plot ValuePerSqFt against Units

> ggplot(housing, aes(x=Units, y=ValuePerSqFt)) + geom_point()

> ggplot(housing, aes(x=log(Units), y=ValuePerSqFt)) + geom_point()

> ggplot(housing, aes(x=Units, y=log(ValuePerSqFt))) + geom_point()

> ggplot(housing, aes(x=log(Units), y=log(ValuePerSqFt))) +

+ geom_point()

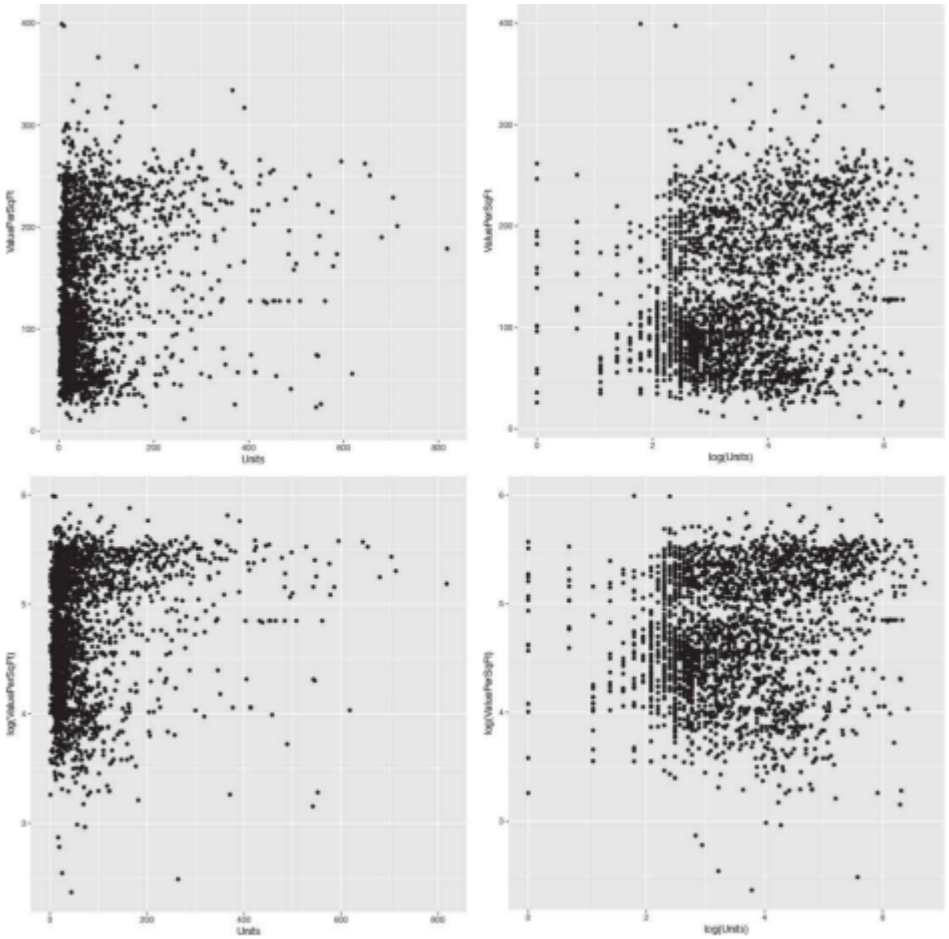


Figure 19.8 Scatterplots of value versus number of units. It is not yet certain whether taking logs will be useful in modelling.

Now that we have viewed our data a few different ways, it is time to start modelling. We already saw from Figure 19.4 that accounting for the different boroughs will be important and the various scatterplots indicated that Units and SqFt will be important as well. Fitting the model uses the formula interface in lm. Now that there are multiple predictors, we separate them on the right side of the formula using plus signs (+).

> house1 <- lm(ValuePerSqFt ~ Units + SqFt + Boro, data=housing)

> summary(house1)

Call:

lm(formula = ValuePerSqFt ~ Units + SqFt + Boro, data = housing)

Residuals:

Min 1Q Median 3Q Max

-168.458 -22.680 1.493 26.290 261.761

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 4.430e+01 5.342e+00 8.293 < 2e-16 ***

Units -1.532e-01 2.421e-02 -6.330 2.88e-10 ***

SqFt 2.070e-04 2.129e-05 9.723 < 2e-16 ***

BoroBrooklyn 3.258e+01 5.561e+00 5.858 5.28e-09 ***

BoroManhattan 1.274e+02 5.459e+00 23.343 < 2e-16 ***

BoroQueens 3.011e+01 5.711e+00 5.272 1.46e-07 ***

BoroStaten Island -7.114e+00 1.001e+01 -0.711 0.477

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 43.2 on 2613 degrees of freedom

Multiple R-squared: 0.6034,Adjusted R-squared: 0.6025

F-statistic: 662.6 on 6 and 2613 DF, p-value: < 2.2e-16

The first thing to notice is that in some versions of R there is a message warning us that Boro was converted to a factor. This is because Boro was stored as a character, and for modelling purposes character data must be represented using indicator variables, which is how factors are treated inside modelling functions, as seen in Section 5.1.

The summary function prints out information about the model, including how the function was called, quantiles for the residuals, coefficient estimates, standard errors and p-values for each variable, and the degrees of freedom, p-value and F-statistic for the model. There is no coefficient for the Bronx because that is the baseline level of Boro, and all the other Boro coefficients are relative to that baseline. The coefficients represent the effect of the predictors on the response and the standard errors are the uncertainty in the estimation of the coefficients. The t value (t-statistic) and p-value for the coefficients are numerical measures of statistical significance, though these should be viewed with caution, as most modern data scientists do not like to look at the statistical significance of individual coefficients but rather judge the model as a whole as covered in Chapter 21. The model p-value and F-statistic are measures of its goodness of fit. The degrees of freedom for a regression are calculated as the number of observations minus the number of coefficients. In this example, there are nrow(housing) − length(coef(house1)) = 2613 degrees of freedom. A quick way to grab the coefficients from a model is to either use the coef function or get them from the model using the $ operator on the model object.

> house1$coefficients

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00

> coef(house1)

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00

> # works the same as coef

> coefficients(house1)

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00

As a repeated theme, we prefer visualizations over tables of information, and a great way of visualizing regression results is a coefficient plot, like the one shown in Figure 19.2. Rather than build it from scratch, we use the convenient coefplot package that we wrote. Figure 19.9 shows the result, where each coefficient is plotted as a point with a thick line representing the one standard error confidence interval and a thin line representing the two standard error confidence interval. There is a vertical line indicating 0. In general, a good rule of thumb is that if the two standard error confidence interval does not contain 0, it is statistically significant.

> library(coefplot)

> coefplot(house1)

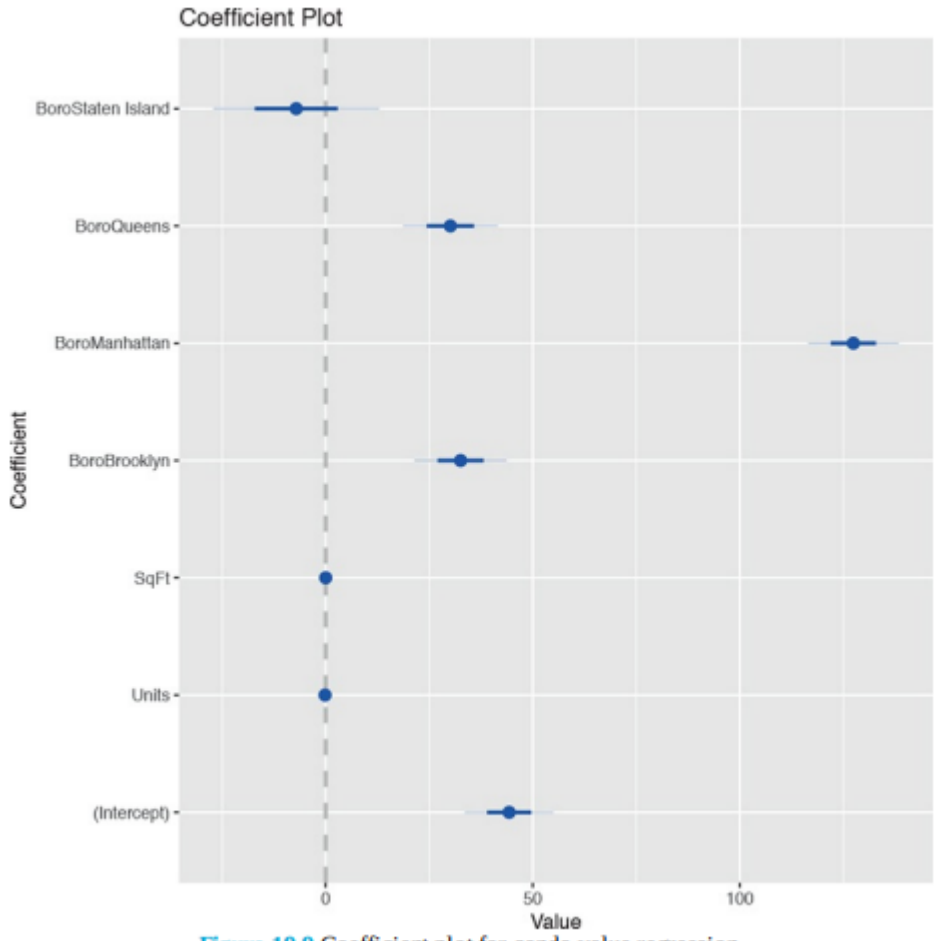


Figure 19.9 Coefficient plot for condo value regression.

Figure 19.9 shows that, as expected, being located in Manhattan has the largest effect on value per square foot. Surprisingly, the number of units or square feet in a building has little effect on value. This is a model with purely additive terms. Interactions between variables can be equally powerful. To enter them in a formula, separate the desired variables with a * instead of +. Doing so results in the individual variables plus the interaction term being included in the model. To include just the interaction term, and not the individual variables, use : instead. The results of interacting Units and SqFt are shown in Figure 19.10.

> house2 <- lm(ValuePerSqFt ~ Units * SqFt + Boro, data=housing)

> house3 <- lm(ValuePerSqFt ~ Units : SqFt + Boro, data=housing)

> house2$coefficients

(Intercept) Units SqFt

4.093685e+01 -1.024579e-01 2.362293e-04

BoroBrooklyn BoroManhattan BoroQueens

3.394544e+01 1.272102e+02 3.040115e+01

BoroStaten Island Units:SqFt

-8.419682e+00 -1.809587e-07

> house3$coefficients

(Intercept) BoroBrooklyn BoroManhattan

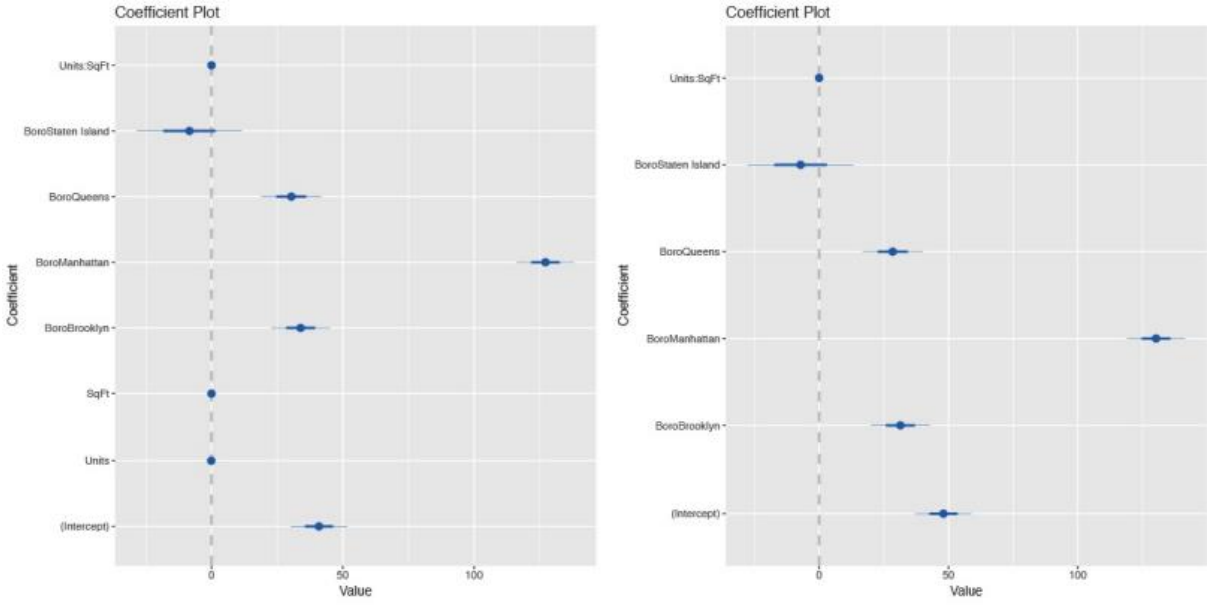
4.804972e+01 3.141208e+01 1.302084e+02

BoroQueens BoroStaten Island Units:SqFt

2.841669e+01 -7.199902e+00 1.088059e-07

> coefplot(house2)

> coefplot(house3)



(a) Individual variables plus the interaction term (b) Just the interaction term

Figure 19.10 Coefficient plots for models with interaction terms. The figure on the left includes individual variables and the interaction term, while the figure on the right only includes the interaction term.

If three variables all interact together, the resulting coefficients will be the three individual terms, three two-way interactions and one three-way interaction.

> house4 <- lm(ValuePerSqFt ~ SqFt*Units*Income, housing)

> house4$coefficients

(Intercept) SqFt Units

1.116433e+02 -1.694688e-03 7.142611e-03

Income SqFt:Units SqFt:Income

7.250830e-05 3.158094e-06 -5.129522e-11

Units:Income SqFt:Units:Income

-1.279236e-07 9.107312e-14

Interacting (from now on, unless otherwise specified, interacting will refer to the * operator) a continuous variable like SqFt with a factor like Boro results in individual terms for the continuous variable and each non-baseline level of the factor plus an interaction term between the continuous

variable and each non-baseline level of the factor. Interacting two (or more) factors yields terms for all the individual non-baseline levels in both factors and an interaction term for every combination of non-baseline levels of the factors.

> house5 <- lm(ValuePerSqFt ~ Class*Boro, housing)

> house5$coefficients

(Intercept)

47.041481

ClassR4-CONDOMINIUM

4.023852

ClassR9-CONDOMINIUM

-2.838624

ClassRR-CONDOMINIUM

3.688519

BoroBrooklyn

27.627141

BoroManhattan

89.598397

BoroQueens

19.144780

BoroStaten Island

-9.203410

ClassR4-CONDOMINIUM:BoroBrooklyn

4.117977

ClassR9-CONDOMINIUM:BoroBrooklyn

2.660419

ClassRR-CONDOMINIUM:BoroBrooklyn

-25.607141

ClassR4-CONDOMINIUM:BoroManhattan

47.198900

ClassR9-CONDOMINIUM:BoroManhattan

33.479718

ClassRR-CONDOMINIUM:BoroManhattan

10.619231

ClassR4-CONDOMINIUM:BoroQueens

13.588293

ClassR9-CONDOMINIUM:BoroQueens

-9.830637

ClassRR-CONDOMINIUM:BoroQueens

34.675220

ClassR4-CONDOMINIUM:BoroStaten Island

NA

ClassR9-CONDOMINIUM:BoroStaten Island

NA

ClassRR-CONDOMINIUM:BoroStaten Island

NA

Neither SqFt nor Units appear to be significant in any model when viewed in a coefficient plot. However, zooming in on the plot shows that the coefficients for Units and SqFt are non-zero as seen in Figure 19.11.

> coefplot(house1, sort='mag') + scale_x_continuous(limits=c(-.25, .1))

> coefplot(house1, sort='mag') + scale_x_continuous(limits=c(-.0005, .0005))

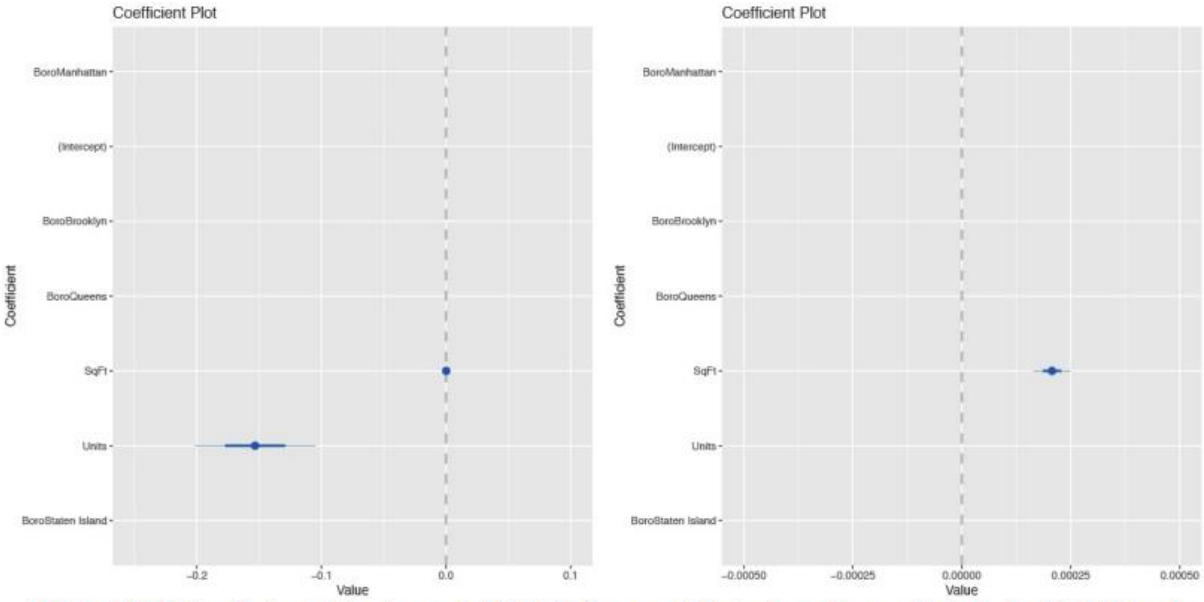


Figure 19.11 Coefficient plots for model `house1` zoomed in to show the coefficients for `Units` and `SqFt`.

This is likely a scaling issue, as the indicator variables for Boro are on the scale of 0 and 1 while the range for Units is between 1 and 818 and SqFt is between 478 and 925,645. This can be resolved by standardizing, or scaling, the variables. This subtracts the mean and divides by the standard deviation. While the results of the model will mathematically be the same, the coefficients will have different

values and different interpretations. Whereas before a coefficient was the change in the response corresponding to a one-unit increase in the predictor, the coefficientis now the change in the response corresponding to a one-standard-deviation increase in the predictor. Standardizing can be performed within the formula interface with the scale function.

> house1.b <- lm(ValuePerSqFt ~ scale(Units) + scale(SqFt) + Boro, + data=housing)

> coefplot(house1.b, sort='mag')

The coefficient plot in Figure 19.12 shows that for each change in the standard deviation of SqFt there is a change of 21.95 in ValuePerSqFt. We also see that Units has a negative impact. This implies that having fewer, but larger, units is beneficial to the value of a building.

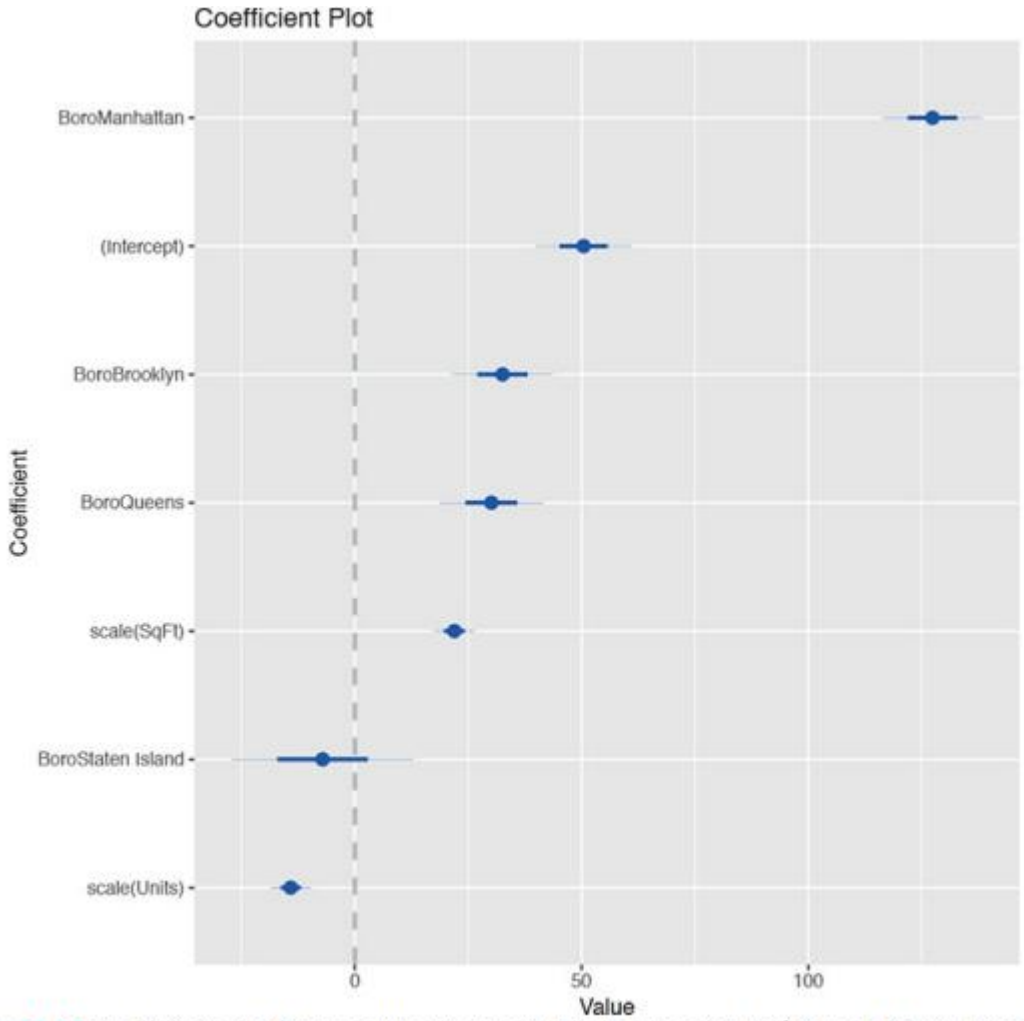


Figure 19.12 Coefficient plot for a model with standardized values for `Units` and `SqFt`. This shows that having fewer, but larger, units is beneficial to the value of a building.

Another good test is to include the ratio of Units and SqFt as a single variable. To simply divide one variable by another in a formula, the division must be wrapped in the I function.

> house6 <- lm(ValuePerSqFt ~ I(SqFt/Units) + Boro, housing)

> house6$coefficients

(Intercept) I(SqFt/Units) BoroBrooklyn

43.754838763 0.004017039 30.774343209

BoroManhattan BoroQueens BoroStaten Island

130.769502685 29.767922792 -6.134446417

The I function is used to preserve a mathematical relationship in a formula and prevent it from being interpreted according to formula rules. For instance, using (Units + SqFt)^2 in a formula is the same as using Units * SqFt, whereas I(Units + SqFt)^2 will include the square of the sum of the two variables as a term in the formula.

> house7 <- lm(ValuePerSqFt ~ (Units + SqFt)^2, housing)

> house7$coefficients

(Intercept) Units SqFt Units:SqFt

1.070301e+02 -1.125194e-01 4.964623e-04 -5.159669e-07

> house8 <- lm(ValuePerSqFt ~ Units * SqFt, housing)

> identical(house7$coefficients, house8$coefficients)

[1] TRUE

> house9 <- lm(ValuePerSqFt ~ I(Units + SqFt)^2, housing)

> house9$coefficients

(Intercept) I(Units + SqFt)

1.147034e+02 2.107231e-04

We have fit numerous models from which we need to pick the “best” one. Model selection is discussed in Section 21.2. In the meantime, visualizing the coefficients from multiple models is a handy tool. Figure 19.13 shows a coefficient plot for models house1, house2 and house3.

> # also from the coefplot package

> multiplot(house1, house2, house3)

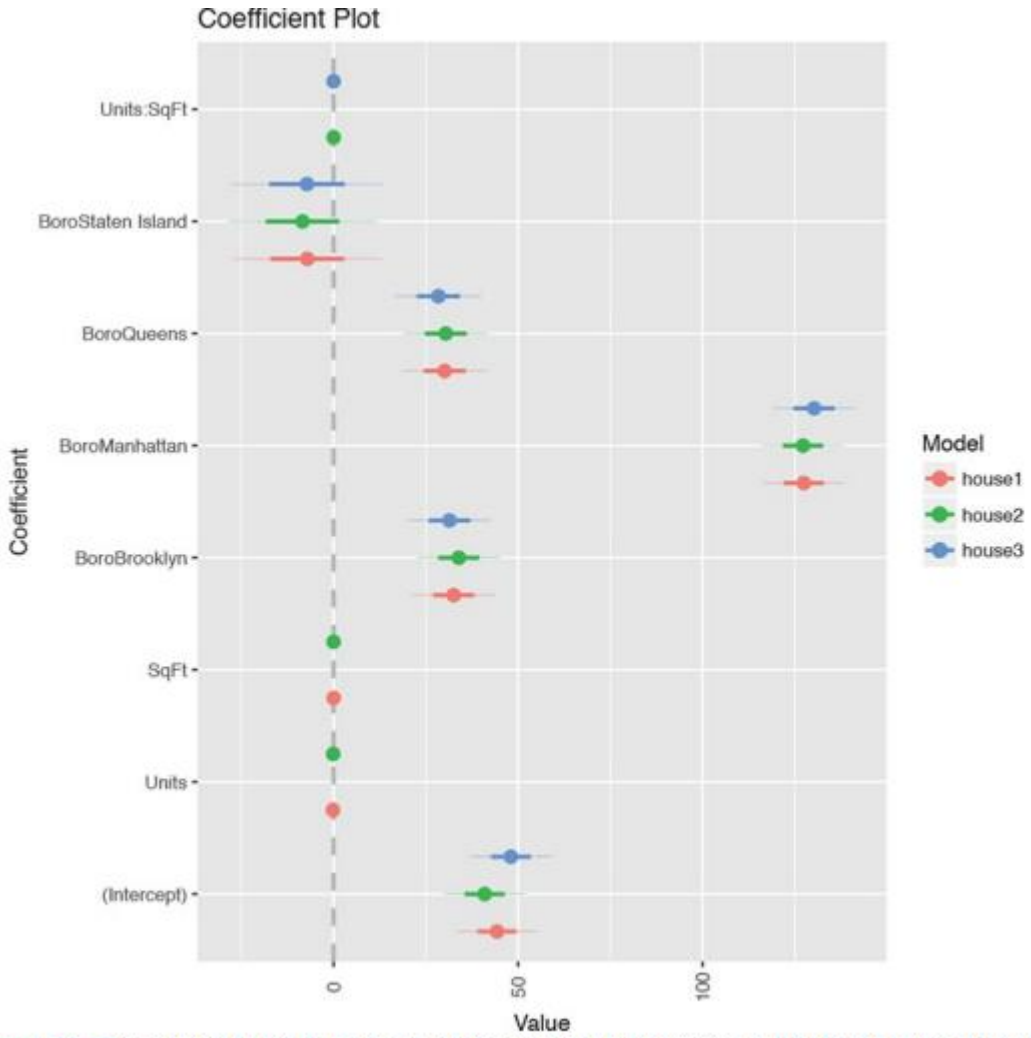


Figure 19.13 Coefficient plot for multiple condo models. The coefficients are plotted in the same spot on the *y*-axis for each model. If a model does not contain a particular coefficient, it is simply not plotted.

Regression is often used for prediction, which in R is enabled by the predict function. For this example, new data are available at http://www.jaredlander.com/data/housingNew.csv.

> housingNew <- read.table("http://www.jaredlander.com/data/housingNew.csv", + sep=",", header=TRUE, stringsAsFactors=FALSE)

Making the prediction can be as simple as calling predict, although caution must be used when dealing with factor predictors to ensure that they have the same levels as those used in building the model.

> # make prediction with new data and 95% confidence bounds

> housePredict <- predict(house1, newdata=housingNew, se.fit=TRUE,

+ interval="prediction", level=.95)

> # view predictions with upper and lower bounds based on standard errors

> head(housePredict$fit)

fit lwr upr

1 74.00645 -10.813887 158.8268

2 82.04988 -2.728506 166.8283

3 166.65975 81.808078 251.5114

4 169.00970 84.222648 253.7968

5 80.00129 -4.777303 164.7799

6 47.87795 -37.480170 133.2361

> # view the standard errors for the prediction

> head(housePredict$se.fit)

1 2 3 4 5 6

2.118509 1.624063 2.423006 1.737799 1.626923 5.318813

Generalized Linear Models

Not all data can be appropriately modeled with linear regression, because they are binomial (TRUE/FALSE), count data or some other form. To model these types of data, generalized linear models were developed. They are still modeled using a linear predictor, Xβ, but they are transformed using some link function. To the R user, fitting a generalized linear model requires barely any more effort than running a linear regression.

Logistic Regression

A very powerful and common model—especially in fields such as marketing and medicine—is logistic regression. The examples in this section will use the a subset of data from the 2010 American Community Survey (ACS) for New York State. 1 ACS data contain a lot of information, so we have made a subset of it with 22,745 rows and 18 columns available at http://jaredlander.com/data/acs_ny.csv.

> acs <- read.table("http://jaredlander.com/data/acs_ny.csv", + sep="," , header=TRUE, stringsAsFactors=FALSE)

Logistic regression models are formulated as

$$p(y_i = 1) = \text{logit}^{-1}(X_i\beta) \qquad (20.1)$$

where yi is the ith response and Xiβ is the linear predictor. The inverse logit function

$$\text{logit}^{-1}(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}} \qquad (20.2)$$

transforms the continuous output from the linear predictor to fall between 0 and 1. This is the inverse of the link function. We now formulate a question that asks whether a household has an income greater than $150,000. To do this we need to create a new binary variable with TRUE for income above that mark and FALSE for income below.

> acs$Income <- with(acs, FamilyIncome >= 150000)

> library(ggplot2)

> library(useful)

```
> ggplot(acs, aes(x=FamilyIncome)) +
+ geom_density(fill="grey", color="grey") +
+ geom_vline(xintercept=150000) +
+ scale_x_continuous(label=multiple.dollar, limits=c(0, 1000000))
```

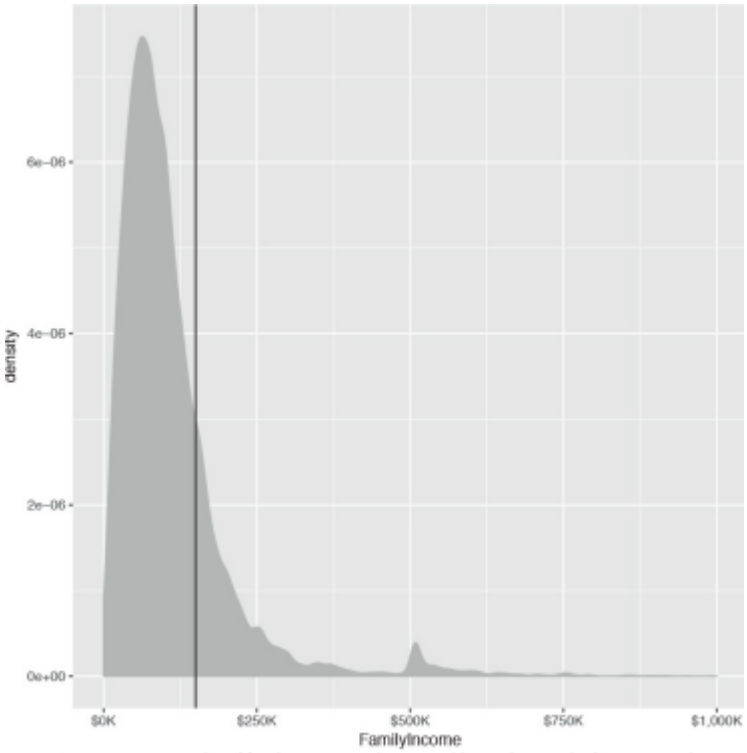


Figure 20.1 Density plot of family income, with a vertical line indicating the $150,000 mark.

```
> head(acs)
Acres FamilyIncome FamilyType NumBedrooms NumChildren NumPeople
1 1-10 150 Married 4 1 3
2 1-10 180 Female Head 3 2 4
3 1-10 280 Female Head 4 0 2
4 1-10 330 Female Head 2 1 2
5 1-10 330 Male Head 3 1 2
6 1-10 480 Male Head 0 3 4
NumRooms NumUnits NumVehicles NumWorkers OwnRent
1 9 Single detached 1 0 Mortgage
2 6 Single detached 2 0 Rented
3 8 Single detached 3 1 Mortgage
4 4 Single detached 1 0 Rented
5 5 Single attached 1 0 Mortgage
```

6 1 Single detached 0 0 Rented

YearBuilt HouseCosts ElectricBill FoodStamp HeatingFuel Insurance

1 1950-1959 1800 90 No Gas 2500

2 Before 1939 850 90 No Oil 0

3 2000-2004 2600 260 No Oil 6600

4 1950-1959 1800 140 No Oil 0

5 Before 1939 860 150 No Gas 660

6 Before 1939 700 140 No Gas 0

Language Income

1 English FALSE

2 English FALSE

3 Other European FALSE

4 English FALSE

5 Spanish FALSE

6 English FALSE

Running a logistic regression is done very similarly to running a linear regression. It still uses the formula interface but the function is glm rather than lm (glm can actually fit linear regressions as well), and a few more options need to be set.

> income1 <- glm(Income ~ HouseCosts + NumWorkers + OwnRent +

+ NumBedrooms + FamilyType,

+ data=acs, family=binomial(link="logit"))

> summary(income1)

Call:

glm(formula = Income ~ HouseCosts + NumWorkers + OwnRent + NumBedrooms +

FamilyType, family = binomial(link = "logit"), data = acs)

Deviance Residuals:

Min 1Q Median 3Q Max

-2.8452 -0.6246 -0.4231 -0.1743 2.9503

Coefficients:

Estimate Std. Error z value Pr(>|z|)

(Intercept) -5.738e+00 1.185e-01 -48.421 <2e-16 ***

HouseCosts 7.398e-04 1.724e-05 42.908 <2e-16 ***

NumWorkers 5.611e-01 2.588e-02 21.684 <2e-16 ***

OwnRentOutright 1.772e+00 2.075e-01 8.541 <2e-16 ***

OwnRentRented -8.886e-01 1.002e-01 -8.872 <2e-16 ***

NumBedrooms 2.339e-01 1.683e-02 13.895 <2e-16 ***

FamilyTypeMale Head 3.336e-01 1.472e-01 2.266 0.0235 *

FamilyTypeMarried 1.405e+00 8.704e-02 16.143 <2e-16 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 22808 on 22744 degrees of freedom

Residual deviance: 18073 on 22737 degrees of freedom

AIC: 18089

Number of Fisher Scoring iterations: 6

> library(coefplot)

> coefplot(income1)

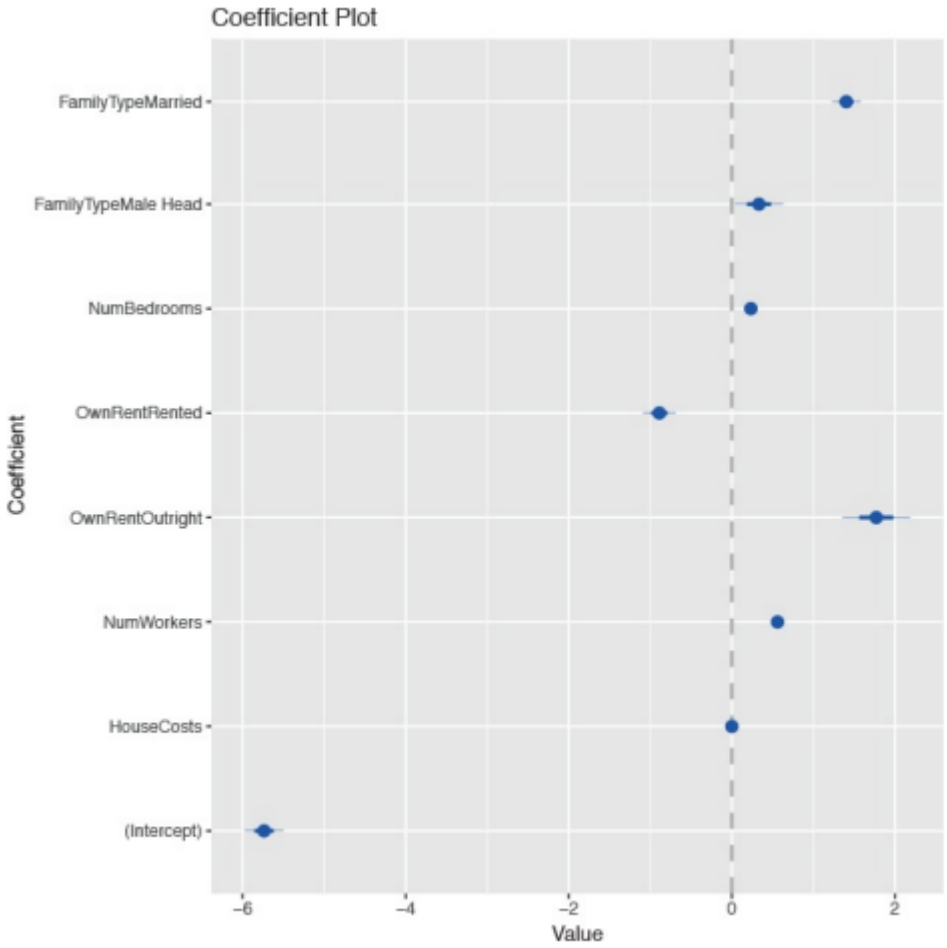


Figure 20.2 Coefficient plot for logistic regression on family income greater than $150,000, based on the American Community Survey.

The output from summary and coefplot for glm is similar to that of lm. There are coefficient estimates, standard errors, p-values—both overall and for the coefficients—and a measure of correctness, which in this case is the deviance and AIC. A general rule of thumb is that adding a variable (or a level of a factor) to a model should result in a drop in deviance of two; otherwise, the variable is not useful in the model. Interactions and all the other formula concepts work the same. Interpreting the coefficients from a logistic regression necessitates taking the inverse logit.

```
> invlogit <- function(x)
+ {
+ 1 / (1 + exp(-x))
+ }
> invlogit(income1$coefficients)
(Intercept) HouseCosts NumWorkers
0.003211572 0.500184950 0.636702036
OwnRentOutright OwnRentRented NumBedrooms
0.854753527 0.291408659 0.558200010
FamilyTypeMale Head FamilyTypeMarried
```

0.582624773 0.802983719

Poisson Regression

Another popular member of the generalized linear models is Poisson regression, which, much like the Poisson distribution, is used for count data. Like all other generalized linear models, it is called using glm. To illustrate we continue using the ACS data with the number of children (NumChildren) as the response. The formulation for Poisson regression is

$$y_i \sim pois(\theta_i) \qquad (20.3)$$

where yi is the ith response and

$$\theta_i = e^{X_i \beta} \qquad (20.4)$$

is the mean of the distribution for the ith observation. Before fitting a model, we look at the histogram of the number of children in each household.

> ggplot(acs, aes(x=NumChildren)) + geom_histogram(binwidth=1)

While Figure 20.3 does not show data that have a perfect Poisson distribution, it is close enough to fit a good model. The coeficient plot is shown in Figure 20.4.

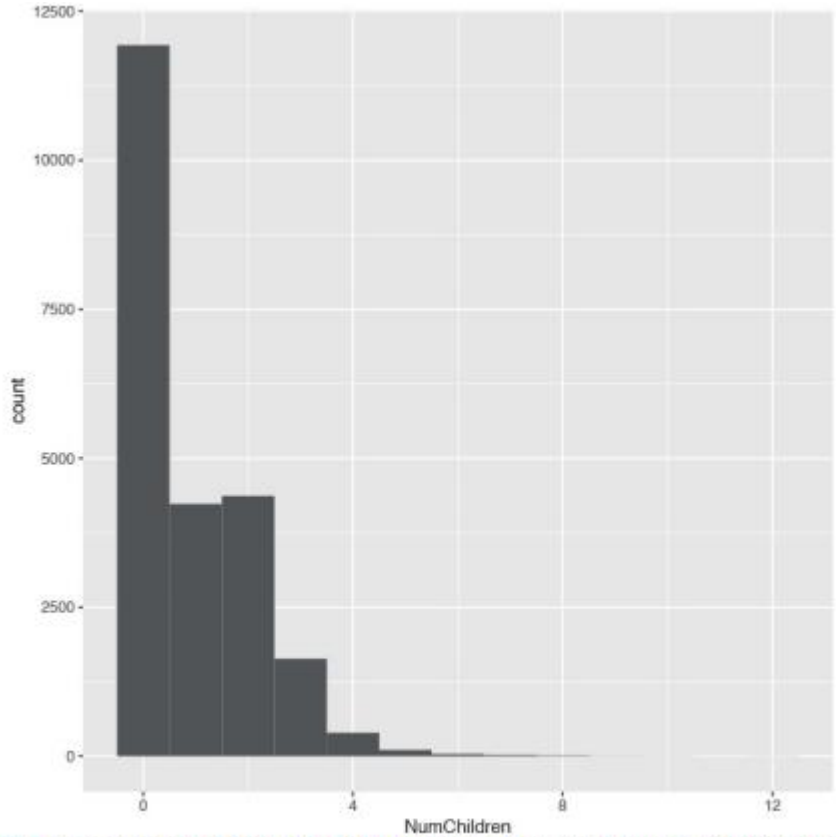


Figure 20.3 Histogram of the number of children per household from the American Community Survey. The distribution is not perfectly Poisson but it is sufficiently so for modelling with Poisson regression.

> children1 <- glm(NumChildren ~ FamilyIncome + FamilyType + OwnRent,

+ data=acs, family=poisson(link="log"))

> summary(children1)

Call:

glm(formula = NumChildren ~ FamilyIncome + FamilyType + OwnRent,

family = poisson(link = "log"), data = acs)

Deviance Residuals:

Min 1Q Median 3Q Max

-1.9950 -1.3235 -1.2045 0.9464 6.3781

Coefficients:

Estimate Std. Error z value Pr(>|z|)

(Intercept) -3.257e-01 2.103e-02 -15.491 < 2e-16 ***

FamilyIncome 5.420e-07 6.572e-08 8.247 < 2e-16 ***

FamilyTypeMale Head -6.298e-02 3.847e-02 -1.637 0.102

FamilyTypeMarried 1.440e-01 2.147e-02 6.707 1.98e-11 ***

OwnRentOutright -1.974e+00 2.292e-01 -8.611 < 2e-16 ***

OwnRentRented 4.086e-01 2.067e-02 19.773 < 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: 35240 on 22744 degrees of freedom

Residual deviance: 34643 on 22739 degrees of freedom

AIC: 61370

Number of Fisher Scoring iterations: 5

> coefplot(children1)

The output here is similar to that for logistic regression, and the same rule of thumb for deviance applies. A particular concern with Poisson regression is overdispersion, which means that the variability seen in the data is greater than is theorized by the Poisson distribution where the mean and variance are the same.

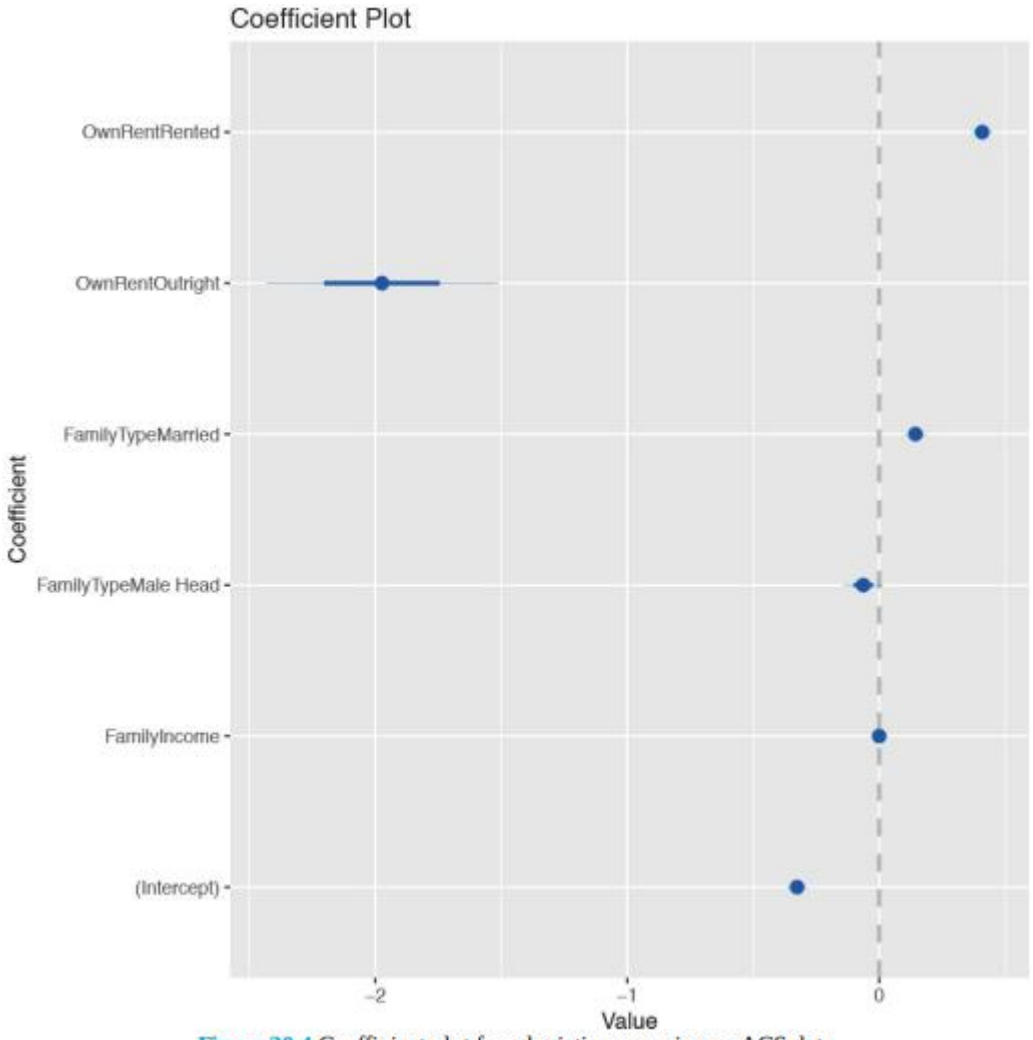


Figure 20.4 Coefficient plot for a logistic regression on ACS data.

$$OD = \frac{1}{n-p}\sum_{i=1}^{n} z_i^2 \qquad (20.5)$$

Overdispersion is defined as

Where

$$z_i = \frac{y_i - \hat{y}_i}{sd(\hat{y}_i)} = \frac{y_i - u_i\hat{\theta}_i}{\sqrt{u_i\hat{\theta}_i}} \qquad (20.6)$$

are the studentized residuals. Calculating overdispersion in R is as follows.

```
> # the standardized residuals
> z <- (acs$NumChildren - children1$fitted.values) /
+ sqrt(children1$fitted.values)
> # Overdispersion Factor
> sum(z^2) / children1$df.residual
```

[1] 1.469747

> # Overdispersion p-value

> pchisq(sum(z^2), children1$df.residual)

[1] 1

Generally an overdispersion ratio of 2 or greater indicates overdispersion. While this overdispersion ratio is less than 2, the p-value is 1, meaning that there is a statistically significant overdispersion. So we refit the model to account for the overdispersion using the quasipoisson family, which actually uses the negative binomial distribution.

> children2 <- glm(NumChildren ~ FamilyIncome + FamilyType + OwnRent, + data=acs, family=quasipoisson(link="log"))

> multiplot(children1, children2)

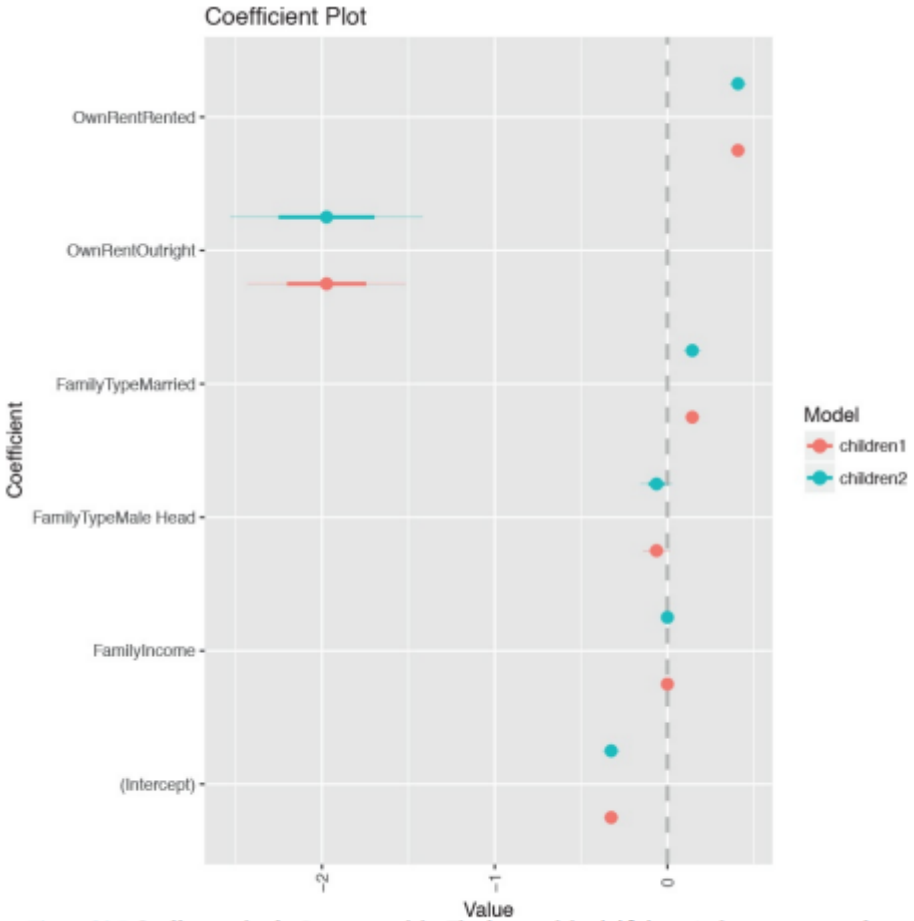


Figure 20.5 Coefficient plot for Poisson models. The first model, `children1`, does not account for overdispersion while `children2` does. Because the overdispersion was not too big, the coefficient estimates in the second model have just a bit more uncertainty.

Figure 20.5 shows a coefficient plot for a model with that accounts for overdispersion and one that does not. Since the overdispersion was not very large, the second model adds just a little uncertainty to the coefficient estimates.

Other Generalized Linear Models

Other common generalized linear models supported by the glm function are gamma, inverse gaussian and quasibinomial. Different link functions can be supplied, such as the following: logit, probit, cauchit, log and cloglog for binomial; inverse, identity and log for gamma; log, identity and sqrt for Poisson; and 1/mu^2, inverse, identity and log for inverse gaussian. Multinomial regression, for classifying multiple categories, requires either running multiple logistic regressions (a tactic well supported in statistical literature) or using the polr function or the multinom function from the nnet package.

Survival Analysis

While not technically part of the family of generalized linear models, survival analysis is another important extension to regression. It has many applications, such as clinical medical trials, server failure times, number of accidents and time to death after a treatment or disease. Data used for survival analysis are different from most other data in that they are censored, meaning there is unknown information, typically about what happens to a subject after a given amount of time. For an example, we look at the bladder data from the survival package.

```
> library(survival)

> head(bladder)

id rx number size stop event enum

1 1 1 1 3 1 0 1

2 1 1 1 3 1 0 2

3 1 1 1 3 1 0 3

4 1 1 1 3 1 0 4

5 2 1 2 1 4 0 1

6 2 1 2 1 4 0 2
```

The columns of note are stop (when an event occurs or the patient leaves the study) and event (whether an event occurred at the time). Even if event is 0, we do not know if an event could have occurred later; this is why it is called censored. Making use of that structure requires the Surv function.

```
> # first look at a piece of the data

> bladder[100:105, ]

id rx number size stop event enum

100 25 1 2 1 12 1 4

101 26 1 1 3 12 1 1

102 26 1 1 3 15 1 2

103 26 1 1 3 24 1 3

104 26 1 1 3 31 0 4

105 27 1 1 2 32 0 1

> # now look at the response variable built by build.y
```

```
> survObject <- with(bladder[100:105, ], Surv(stop, event))
> # nicely printed form
> survObject
[1] 12 12 15 24 31+ 32+
> # see its matrix form
> survObject[, 1:2]
time status
[1,] 12 1
[2,] 12 1
[3,] 15 1
[4,] 24 1
[5,] 31 0
[6,] 32 0
```

This shows that for the first three rows where an event occurred, the time is known to be 12, whereas the bottom two rows had no event, so the time is censored because an event could have occurred afterward. Perhaps the most common modelling technique in survival analysis is using a Cox proportional hazards model, which in R is done with coxph. The model is fitted using the familiar formula interface supplied to coxph. The survfit function builds the survival curve that can then be plotted as shown in Figure 20.6. The survival curve shows the percentage of participants surviving at a given time. The summary is similar to other summaries but tailored to survival analysis.

```
> cox1 <- coxph(Surv(stop, event) ~ rx + number + size + enum,
+ data=bladder)
> summary(cox1)
Call:
coxph(formula = Surv(stop, event) ~ rx + number + size + enum,
data = bladder)
n= 340, number of events= 112
coef exp(coef) se(coef) z Pr(>|z|)
rx -0.59739 0.55024 0.20088 -2.974 0.00294 **
number 0.21754 1.24301 0.04653 4.675 2.93e-06 ***
size -0.05677 0.94481 0.07091 -0.801 0.42333
enum -0.60385 0.54670 0.09401 -6.423 1.34e-10 ***
---
```

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

exp(coef) exp(-coef) lower .95 upper .95

rx 0.5502 1.8174 0.3712 0.8157

number 1.2430 0.8045 1.1347 1.3617

size 0.9448 1.0584 0.8222 1.0857

enum 0.5467 1.8291 0.4547 0.6573

Concordance= 0.753 (se = 0.029)

Rsquare= 0.179 (max possible= 0.971)

Likelihood ratio test= 67.21 on 4 df, p=8.804e-14

Wald test = 64.73 on 4 df, p=2.932e-13

Score (logrank) test = 69.42 on 4 df, p=2.998e-14

> plot(survfit(cox1), xlab="Days", ylab="Survival Rate", conf.int=TRUE)

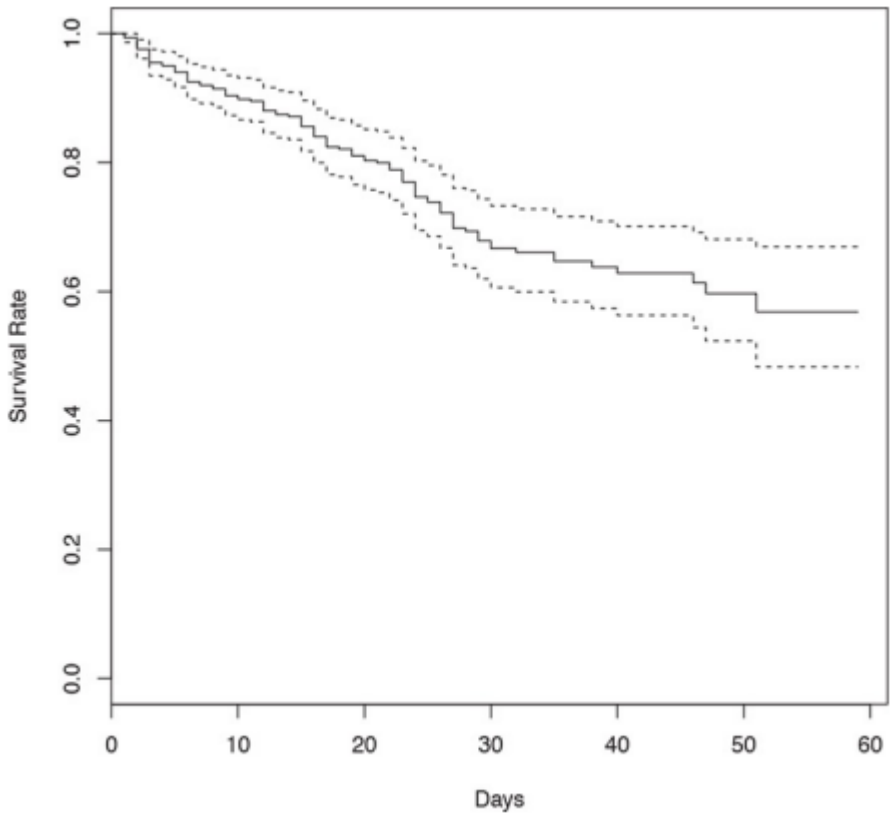


Figure 20.6 Survival curve for Cox proportional hazards model fitted on bladder data.

In this data, the rx variable indicates placebo versus treatment, which is a natural stratification of the patients. Passing rx to strata in the formula splits the data into two for analysis and will result in two survival curves like those in Figure 20.7.

> cox2 <- coxph(Surv(stop, event) ~ strata(rx) + number

+ + size + enum, data=bladder)

```
> summary(cox2)
Call:
coxph(formula = Surv(stop, event) ~ strata(rx) + number + size +
enum, data = bladder)
n= 340, number of events= 112
coef exp(coef) se(coef) z Pr(>|z|)
number 0.21371 1.23826 0.04648 4.598 4.27e-06 ***
size -0.05485 0.94662 0.07097 -0.773 0.44
enum -0.60695 0.54501 0.09408 -6.451 1.11e-10 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
exp(coef) exp(-coef) lower .95 upper .95
number 1.2383 0.8076 1.1304 1.3564
size 0.9466 1.0564 0.8237 1.0879
enum 0.5450 1.8348 0.4532 0.6554
Concordance= 0.74 (se = 0.04 )
Rsquare= 0.166 (max possible= 0.954 )
Likelihood ratio test= 61.84 on 3 df, p=2.379e-13
Wald test = 60.04 on 3 df, p=5.751e-13
Score (logrank) test = 65.05 on 3 df, p=4.896e-14
> plot(survfit(cox2), xlab="Days", ylab="Survival Rate",
+ conf.int=TRUE, col=1:2)
> legend("bottomleft", legend=c(1, 2), lty=1, col=1:2,
+ text.col=1:2, title="rx")
```

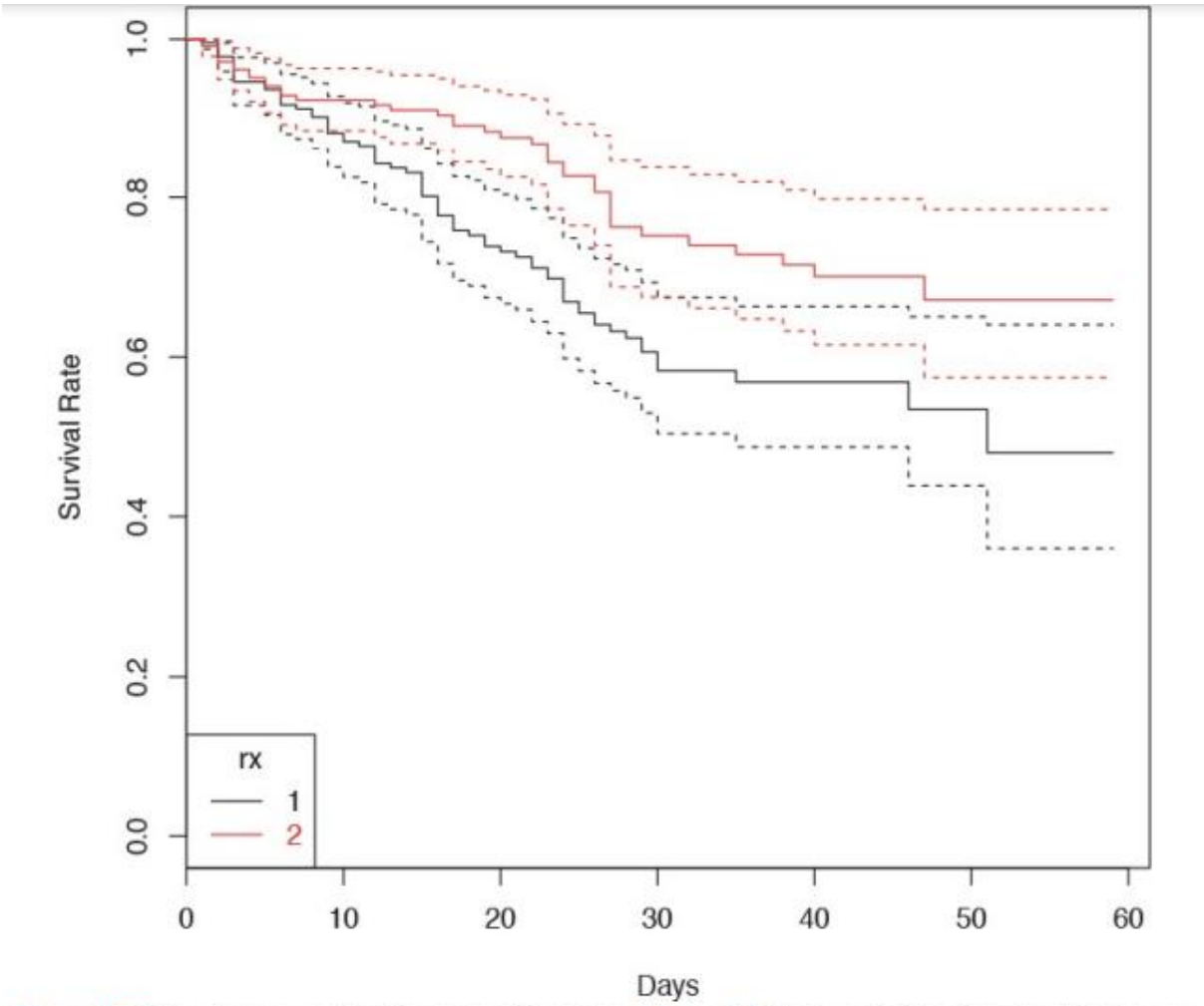


Figure 20.7 Survival curve for Cox proportional hazards model fitted on bladder data stratified on rx.

As an aside, this was a relatively simple legend to produce but it took a lot more effort than it would with ggplot2. Testing the assumption of proportional hazards is done with cox.zph.

> cox.zph(cox1)

rho chisq p

rx 0.0299 0.0957 7.57e-01

number 0.0900 0.6945 4.05e-01

size -0.1383 2.3825 1.23e-01

enum 0.4934 27.2087 1.83e-07

GLOBAL NA 32.2101 1.73e-06

> cox.zph(cox2)

rho chisq p

number 0.0966 0.785 3.76e-01

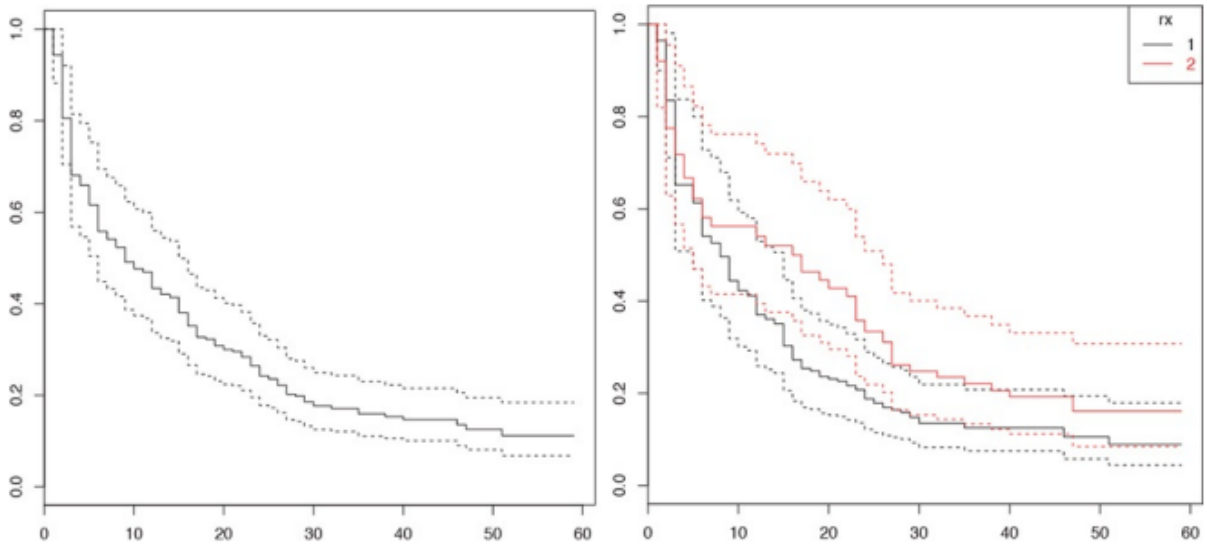
size -0.1331 2.197 1.38e-01

enum 0.4972 27.237 1.80e-0

GLOBAL NA 32.101 4.98e-07

An Andersen-Gill analysis is similar to survival analysis, except it takes intervalized data and can handle multiple events, such as counting the number of emergency room visits as opposed to whether or not there is an emergency room visit. It is also performed using coxph, except an additional variable is passed to Surv, and the data must be clustered on an identification column (id) to keep track of multiple events. The corresponding survival curves are seen in Figure 20.8.

```
> head(bladder2)

id rx number size start stop event enum

1 1 1 1 3 0 1 0 1

2 2 1 2 1 0 4 0 1

3 3 1 1 1 0 7 0 1

4 4 1 5 1 0 10 0 1

5 5 1 4 1 0 6 1 1

6 5 1 4 1 6 10 0 2

> ag1 <- coxph(Surv(start, stop, event) ~ rx + number + size + enum +

+ cluster(id), data=bladder2)

> ag2 <- coxph(Surv(start, stop, event) ~ strata(rx) + number + size +

+ enum + cluster(id), data=bladder2)

> plot(survfit(ag1), conf.int=TRUE)

> plot(survfit(ag2), conf.int=TRUE, col=1:2)

> legend("topright", legend=c(1, 2), lty=1, col=1:2,

+ text.col=1:2, title="rx")
```



(a) No stratification (b) Stratification on rx

Figure 20.8 Andersen-Gill survival curves for bladder2 data.

Nonlinear Models

A key tenet of linear models is a linear relationship, which is actually reflected in the coefficients, not the predictors. While this is a nice simplifying assumption, in reality nonlinearity often holds. Fortunately, modern computing makes fitting nonlinear models not much more difficult than fitting linear models. Typical implementations are nonlinear least squares, splines, decision trees and random forests and generalized additive models (GAMs).

Splines

A smoothing spline can be used to fit a smooth to data that exhibit nonlinear behavior and even make predictions on new data. A spline is a function f that is a linear combination of N functions (one for each unique data point) that are transformations of the variable x.

$$f(x) = \sum_{j=1}^{N} N_J(x)\theta_j \qquad (23.3)$$

The goal is to find the function f that minimizes

$$RSS(f,\lambda) = \sum_{i=1}^{N}\{y_i - f(x_i)\}^2 + \lambda \int \{f''(t)\}^2 \, dt \qquad (23.4)$$

where λ is the smoothing parameter. Small λs make for a rough smooth and large λs make for a smooth smooth. This is accomplished in R using smooth.spline. It returns a list of items where x holds the unique values of the data, y are the corresponding fitted values and df is the degrees of freedom used. We demonstrate with the diamonds data.

```
> data(diamonds)

> # fit with a few different degrees of freedom

> # the degrees of freedom must be greater than 1

> # but less than the number of unique x values in the data
```

```
> diaSpline1 <- smooth.spline(x=diamonds$carat, y=diamonds$price)
> diaSpline2 <- smooth.spline(x=diamonds$carat, y=diamonds$price,
+ df=2)
> diaSpline3 <- smooth.spline(x=diamonds$carat, y=diamonds$price,
+ df=10)
> diaSpline4 <- smooth.spline(x=diamonds$carat, y=diamonds$price,
+ df=20)
> diaSpline5 <- smooth.spline(x=diamonds$carat, y=diamonds$price,
+ df=50)
> diaSpline6 <- smooth.spline(x=diamonds$carat, y=diamonds$price,
+ df=100)
```

To plot these we extract the information from the objects, build a data.frame, and then add a new layer on top of the standard scatterplot of the diamonds data. Figure 23.3 shows this. Fewer degrees of freedom leads to straighter fits while higher degrees of freedom leads to more interpolating lines.

```
> get.spline.info <- function(object)
+ {
+ data.frame(x=object$x, y=object$y, df=object$df)
+ }
>
> library(plyr)
> # combine results into one data.frame
> splineDF <- ldply(list(diaSpline1, diaSpline2, diaSpline3, diaSpline4,
+ diaSpline5, diaSpline6), get.spline.info)
> head(splineDF)
x y df
1 0.20 361.9112 101.9053
2 0.21 397.1761 101.9053
3 0.22 437.9095 101.9053
4 0.23 479.9756 101.9053
5 0.24 517.0467 101.9053
6 0.25 542.2470 101.9053
> g <- ggplot(diamonds, aes(x=carat, y=price)) + geom_point()
```

> g + geom_line(data=splineDF,

+ aes(x=x, y=y, color=factor(round(df, 0)), group=df)) +

+ scale_color_discrete("Degrees of \nFreedom")

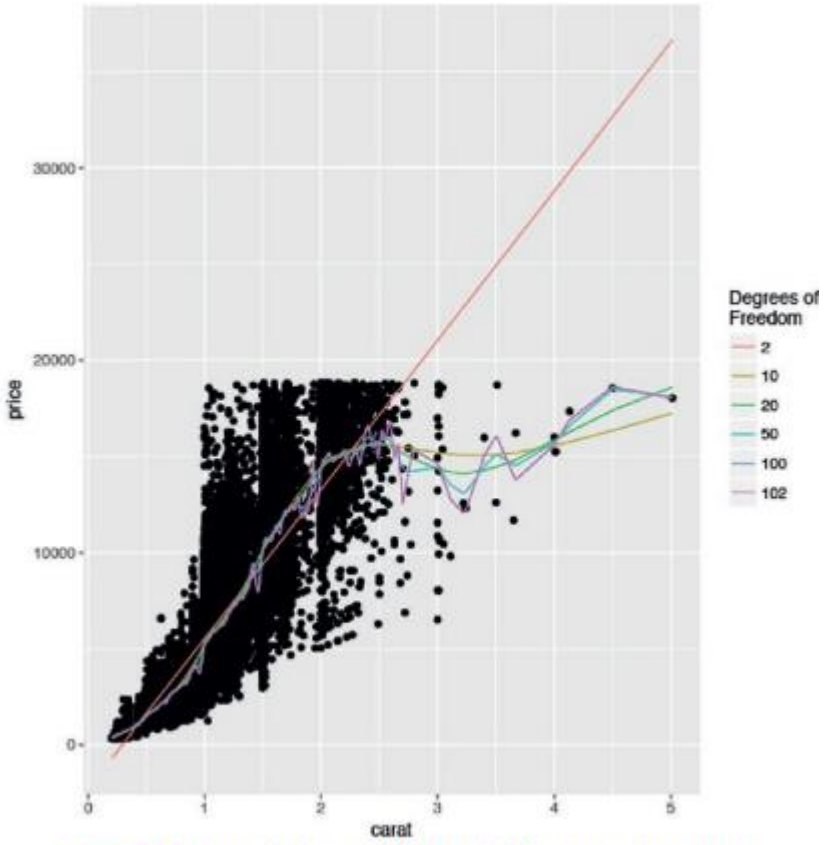


Figure 23.3 Diamonds data with a number of different smoothing splines.

Making predictions on new data is done, as usual, with predict. Another type of spline is the basis spline, which creates new predictors based on transformations of the original predictors. The best basis spline is the natural cubic spline because it creates smooth transitions at interior breakpoints and forces linear behavior beyond the endpoints of the input data. A natural cubic spline with K breakpoints (knots) is made of K basis functions

$$N_1(X) = 1, N_2(X) = X, N_{k+2} = d_k(X) - d_{K-1}(X) \qquad (23.5)$$

Where

$$d_k(X) = \frac{(X - \xi_k)_+^3 - (X - \xi_K)_+^3}{\xi_K - \xi_k} \qquad (23.6)$$

and ξ is the location of a knot and t+ denotes the positive part of t.

While the math may seem complicated, natural cubic splines are easily fitted using ns from the splines package. It takes a predictor variable and the number of new variables to return.

> library(splines)

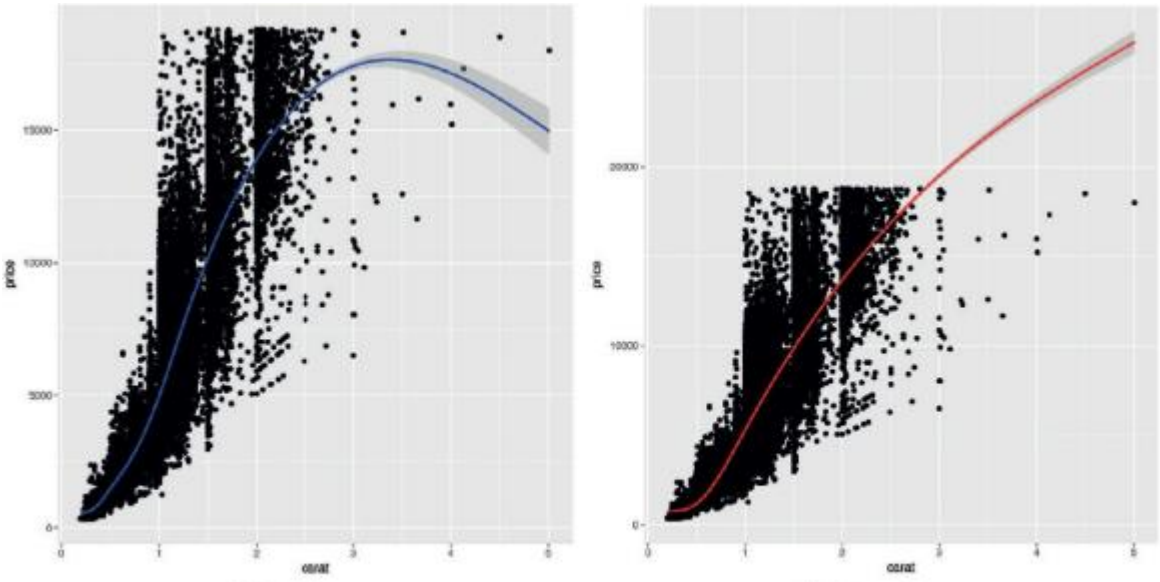
> head(ns(diamonds$carat, df=1))

```
1
[1,] 0.00500073
[2,] 0.00166691
[3,] 0.00500073
[4,] 0.01500219
[5,] 0.01833601
[6,] 0.00666764
> head(ns(diamonds$carat, df=2))
1 2
[1,] 0.013777685 -0.007265289
[2,] 0.004593275 -0.002422504
[3,] 0.013777685 -0.007265289
[4,] 0.041275287 -0.021735857
[5,] 0.050408348 -0.026525299
[6,] 0.018367750 -0.009684459
> head(ns(diamonds$carat, df=3))
1 2 3
[1,] -0.03025012 0.06432178 -0.03404826
[2,] -0.01010308 0.02146773 -0.01136379
[3,] -0.03025012 0.06432178 -0.03404826
[4,] -0.08915435 0.19076693 -0.10098109
[5,] -0.10788271 0.23166685 -0.12263116
[6,] -0.04026453 0.08566738 -0.04534740
> head(ns(diamonds$carat, df=4))
1 2 3 4
[1,] 3.214286e-04 -0.04811737 0.10035562 -0.05223825
[2,] 1.190476e-05 -0.01611797 0.03361632 -0.01749835
[3,] 3.214286e-04 -0.04811737 0.10035562 -0.05223825
[4,] 8.678571e-03 -0.13796549 0.28774667 -0.14978118
[5,] 1.584524e-02 -0.16428790 0.34264579 -0.17835789
[6,] 7.619048e-04 -0.06388053 0.13323194 -0.06935141
```

These new predictors can then be used in any model just like any other predictor. More knots means a more interpolating fit. Plotting the result of a natural cubic spline overlaid on data is easy with ggplot2. Figure 23.4a shows this for the diamonds data and six knots, and Figure 23.4b shows it with three knots. Notice that having six knots fits the data more smoothly.

> g + stat_smooth(method="lm", formula=y ~ ns(x, 6), color="blue")

> g + stat_smooth(method="lm", formula=y ~ ns(x, 3), color="red")



(a) Six Knots (b) Three Knots

Figure 23.4 Scatterplot of price versus carat with a regression fitted on a natural cubic spline.

Decision Trees

A relatively modern technique for fitting nonlinear models is the decision tree. Decision trees work for both regression and classification by performing binary splits on the recursive predictors.

For regression trees, the predictors are partitioned into M regions R1 , R2 ,..., RM and the response y is

$$\hat{f}(x) = \sum_{m=1}^{M} \hat{c}_m I(x \in R_m) \qquad (23.8)$$

modeled as the average for a region with where

$$\hat{c}_m = \text{avg}(y_i | x_i \in R_m) \qquad (23.9)$$

is the average y value for the region. The method for classification trees is similar. The predictors are partitioned into M regions and the proportion of each class in each of the regions, mk , is calculated as

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \qquad (23.10)$$

where Nm is the number of items in region m and the summation counts the number of observations of class k in region m. Trees can be calculated with the rpart function in rpart. Like other modelling functions, it uses the formula interface but does not work with interactions.

```
> library(rpart)

> creditTree <- rpart(Credit ~ CreditAmount + Age +

+ CreditHistory + Employment, data=credit) Printing the object displays the tree in text

form.

> creditTree

n= 1000

node), split, n, loss, yval, (yprob)

* denotes terminal node

1) root 1000 300 Good (0.7000000 0.3000000)

2) CreditHistory=Critical Account,Late Payment,Up To

Date 911 247 Good (0.7288694 0.2711306)

4) CreditAmount< 7760.5 846 211 Good (0.7505910 0.2494090) *

5) CreditAmount>=7760.5 65 29 Bad (0.4461538 0.5538462)

10) Age>=29.5 40 17 Good (0.5750000 0.4250000)

20) Age< 38.5 19 4 Good (0.7894737 0.2105263) *

21) Age>=38.5 21 8 Bad (0.3809524 0.6190476) *

11) Age< 29.5 25 6 Bad (0.2400000 0.7600000) *

3) CreditHistory=All Paid,All Paid This Bank 89 36

Bad (0.4044944 0.5955056) *
```

The printed tree has one line per node. The first node is the root for all the data and shows that there are 1,000 observations of which 300 are considered “Bad.” The next level of indentation is the first split, which is on CreditHistory. One direction—where CreditHistory equals either “Critical Account,” “Late Payment” or “Up To Date”—contains 911 observations, of which 247 are considered “Bad.” This has a 73% probability of having good credit. The other direction—where CreditHistory equals either “All Paid” or “All Paid This Bank”—has a 60% probability of having bad credit. The next level of indentation represents the next split. Continuing to read the results this way could be laborious; plotting will be easier. Figure 23.8 shows the splits. Nodes split to the left meet the criteria while nodes to the right do not. Each terminal node is labelled by the predicted class, either “Good” or “Bad.” The percentage is read from left to right, with the probability of being “Good” on the left.

```
> library(rpart.plot)

> rpart.plot(creditTree, extra=4)
```

While trees are easy to interpret and fit data nicely, they tend to be unstable with high variance due to overfitting. A slight change in the training data can cause a significant difference in the model.

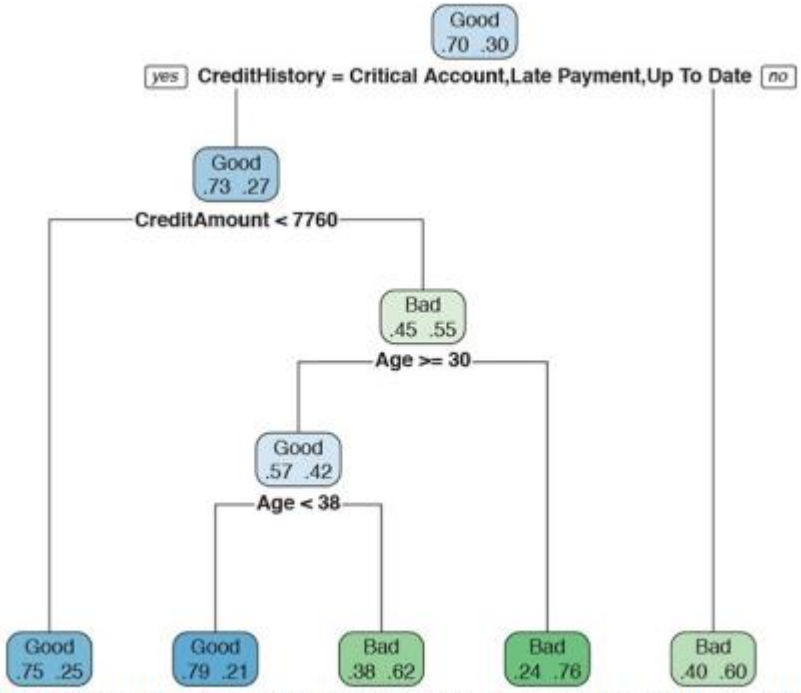


Figure 23.8 Display of decision tree based on credit data. Nodes split to the left meet the criteria while nodes to the right do not. Each terminal node is labelled by the predicted class, either "Good" or "Bad." The percentage is read from left to right, with the probability of being "Good" on the left.

Random Forests

Random forests are a type of ensemble method. An ensemble method is a process in which numerous models are fitted, and the results are combined for stronger predictions. While this provides great predictions, inference and explainability are often limited. Random forests are composed of a number of decision trees where the included predictors and observations are chosen at random. The name comes from randomly building trees to make a forest. In the case of the credit data we will use CreditHistory, Purpose, Employment, Duration, Age and CreditAmount. Some trees will have just CreditHistory and Employment, another will have Purpose, Employment and Age, while another will have CreditHistory, Purpose, Employment and Age. All of these different trees cover all the bases and make for a random forest that should have strong predictive power. Fitting the random forest is done with randomForest from the randomForest package. Normally, randomForest can be used with a formula, but categorical variables must be stored as factors. To avoid having to convert the variables, we provide individual predictor and response matrices. This requirement for factor variables is due to the author's (Andy Liaw) frustration with the formula interface. He even warned users "I will take the formula interface away." We have seen, for this function, that using matrices is generally faster than formulas.

```
> library(randomForest)

> creditFormula <- Credit ~ CreditHistory + Purpose + Employment +

+ Duration + Age + CreditAmount - 1

> # we use all levels of the categorical variables since it is a tree

> creditX <- build.x(creditFormula, data=credit, contrasts=FALSE)

> creditY <- build.y(creditFormula, data=credit)

>
```

```
> # fit the random forest
> creditForest <- randomForest(x=creditX, y=creditY)
>
> creditForest
Call:
randomForest(x = creditX, y = creditY)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 4
OOB estimate of error rate: 27.4%
Confusion matrix:
Good Bad class.error
Good 644 56 0.0800000
Bad 218 82 0.7266667
```

The displayed information shows that 500 trees were built and four variables were assessed at each split; the confusion matrix shows that this is not exactly the best fit, and that there is room for improvement. Due to the similarity between boosted trees and random forests, it is possible to use xgboost to build a random forest by tweaking a few arguments. We fit 1000 trees in parallel (num_parallel_tree=1000) and set the row (subsample=0.5) and column (colsample_bytree=0.5) sampling to be done at random.

```
> # build the response matrix
> creditY2 <- as.integer(relevel(creditY, ref='Bad')) - 1
> # Fit the random forest
> boostedForest <- xgboost(data=creditX, label=creditY2, max_depth=4,
+ num_parallel_tree=1000,
+ subsample=0.5, colsample_bytree=0.5,
+ nrounds=3, objective="binary:logistic")
[1] train-error:0.282000
[2] train-error:0.283000
[3] train-error:0.279000
```

In this case the error rate for the boosted-derived random forest is about the same as for the one fit by randomForest. Increasing the nrounds argument will improve the error rate, although it could also lead to overfitting. A nice benefit of using xgboost is that we can visualize the resulting random forest as a single tree as shown in Figure 23.11.

> xgb.plot.multi.trees(boostedForest, feature_names=colnames(creditX))

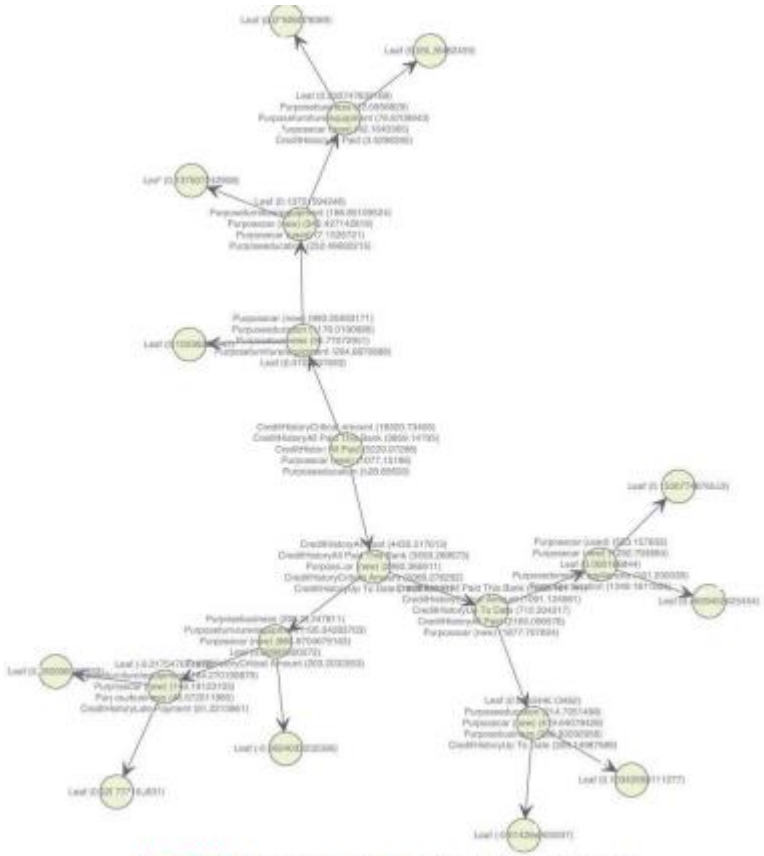


Figure 23.11 Projection of boosted random forest trees onto one tree.