Stat 3701 Lecture Notes: Data

Charles J. Geyer

November 12, 2022

The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.

— John W. Tukey (the first of six "basics" against statistician's hubrises) in "Sunset Salvo", *American Statistician*, 40, 72–76 (1986).

quoted by the `fortunes` package

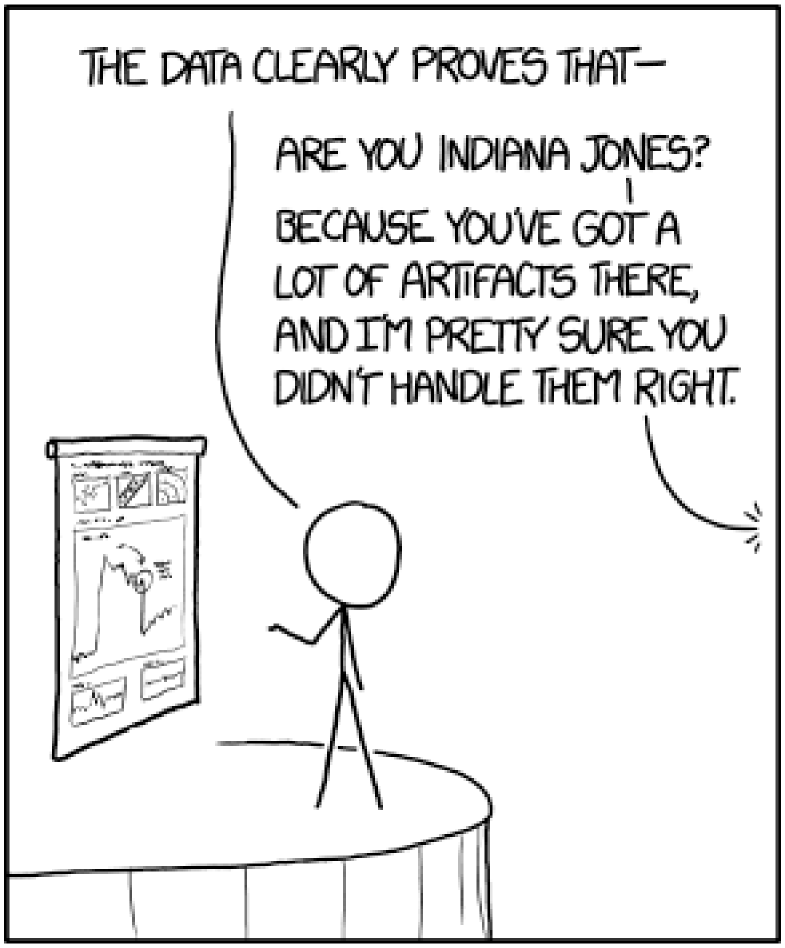

THE DATA CLEARLY PROVES THAT—
ARE YOU INDIANA JONES?
BECAUSE YOU'VE GOT A LOT OF ARTIFACTS THERE, AND I'M PRETTY SURE YOU DIDN'T HANDLE THEM RIGHT.

Figure 1: xkcd:1781 Artifacts

1 License

2 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.17.
- The version of the `MASS` package used to make this document is 7.3.58.1.
- The version of the `quantreg` package used to make this document is 5.94.
- The version of the `rvest` package used to make this document is 1.0.3.
- The version of the `jsonlite` package used to make this document is 1.8.3.
- The version of the `pkgsearch` package used to make this document is 3.1.2.
- The version of the `RSQLite` package used to make this document is 2.2.18.
- The version of the `DBI` package used to make this document is 1.1.3.
- The version of the `dplyr` package used to make this document is 1.0.10.

```r
library(MASS)
library(quantreg)
```

```
## Loading required package: SparseM

##
## Attaching package: 'SparseM'

## The following object is masked from 'package:base':
##
##     backsolve
```

```r
library(rvest)
library(jsonlite)
library(pkgsearch)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following object is masked from 'package:MASS':
##
##     select

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

3 Data

3.1 Data Frames

Statistics or “data science” starts with data. Data can come in many forms but it often comes in a data frame — or at least something R can turn into a data frame. As we saw at the end of the handout about matrices, arrays, and data frames, the two R functions most commonly used to input data into data frames are `read.table` and `read.csv`. We will see some more later.

3.2 Data Artifacts, Errors, Problems

3.2.1 Introduction

> Anything which uses science as part of its name isn’t: political science, creation science, computer science.
>
> — Hal Abelson

Presumably he would have added “data science” if the term had existed when he said that. Statistics doesn’t get off lightly either because of the journal *Statistical Science*.

It is the dirty little secret of science, business, statistics, and “data science” is that there are a lot of errors in almost all data when they get to a data analyst, whatever his or her job title may be.

Hence, *the most important skill* of a data analyst (whatever his or her job title may be) is IMHO knowing how to find errors in data. If you can’t do that, anything else you may be able to do has no point. GIGO.

But this is a secret because businesses don’t like to admit they make errors, and neither do scientists or anyone else who generates data. So the data clean up always takes place behind the scenes. Any publicly available data set has already been cleaned up.

Thus we look at a made-up data set (I took an old data set, the R dataset `growth` in the CRAN package `fda` and introduced errors of the kind I have seen in real data sets).

As we shall see. There are no R functions or packages that help with data errors. You just have to think hard and use logic (both logic in your thinking and R logical operators).

3.2.2 Overview

```
growth <- read.csv("https://www.stat.umn.edu/geyer/3701/data/growth.csv",
    stringsAsFactors = FALSE)
class(growth)
```

```
## [1] "data.frame"
```

```
names(growth)
```

```
##  [1] "HT1"    "HT1.25" "HT1.5"  "HT1.75" "HT2"    "HT3"    "HT4"    "HT5"
##  [9] "HT6"    "HT7"    "HT8"    "HT8.5"  "HT9"    "HT9.5"  "HT10"   "HT10.5"
## [17] "HT11"   "HT11.5" "HT12"   "HT12.5" "HT13"   "HT13.5" "HT14"   "HT14.5"
## [25] "HT15"   "HT15.5" "HT16"   "HT16.5" "HT17"   "HT17.5" "HT18"   "SITE"
## [33] "SEX"
```

```
sapply(growth, class)
```

```
##         HT1      HT1.25       HT1.5      HT1.75         HT2         HT3
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##         HT4         HT5         HT6         HT7         HT8       HT8.5
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##         HT9       HT9.5        HT10      HT10.5        HT11      HT11.5
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##        HT12      HT12.5        HT13      HT13.5        HT14      HT14.5
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##        HT15      HT15.5        HT16      HT16.5        HT17      HT17.5
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##        HT18        SITE         SEX
##   "numeric" "character"   "integer"
```

The variables whose names start with `HT` are heights at the indicated age in years. `SITE` and `SEX` are

```
sort(unique(growth$SITE))
```

```
## [1] "A" "B" "C" "D"
```

```
sort(unique(growth$SEX))
```

```
## [1] 1 2
```

both categorical despite the latter being coded with integers. We will have to figure all that out.

3.2.3 Missing Data

The first tool we use is `grep`. This command has a weird name inherited from unix. It (and its relatives documented on the same help page) is the R command for matching text strings (and for doing search and replace).

```
is.ht <- grep("HT", names(growth))
foo <- growth[is.ht]
foo <- as.matrix(foo)
apply(foo, 2, range)
```

```
##      HT1 HT1.25 HT1.5 HT1.75  HT2 HT3 HT4 HT5 HT6 HT7   HT8 HT8.5 HT9 HT9.5
## [1,]  NA     NA   0.0     NA  0.0  NA  NA  NA  NA  NA 110.9    NA  NA 103.3
## [2,]  NA     NA  95.4     NA 97.3  NA  NA  NA  NA  NA 313.8    NA  NA 318.6
##       HT10 HT10.5 HT11 HT11.5 HT12 HT12.5 HT13 HT13.5 HT14 HT14.5 HT15 HT15.5
## [1,] 126.8     NA   NA    0.0   NA -999.0   NA     NA   NA     NA   NA     NA
## [2,] 611.5     NA   NA  519.7   NA  512.5   NA     NA   NA     NA   NA     NA
##      HT16 HT16.5 HT17 HT17.5 HT18
## [1,]   NA  152.6   NA  107.0   NA
## [2,]   NA  255.6   NA  617.8   NA
```

Try again.

```
apply(foo, 2, range, na.rm = TRUE)
```

```
##         HT1 HT1.25 HT1.5 HT1.75  HT2   HT3    HT4 HT5  HT6    HT7   HT8 HT8.5
## [1,] -999.0 -999.0   0.0    9.4  0.0  39.6 -999.0   0 -999 -999.0 110.9 119.2
## [2,]   84.1   90.4  95.4   95.0 97.3 201.3  120.6 210  134  218.8 313.8 236.6
##        HT9 HT9.5  HT10 HT10.5   HT11 HT11.5 HT12 HT12.5  HT13 HT13.5  HT14
## [1,] 121.4 103.3 126.8 -999.0 -999.0    0.0 -999 -999.0 139.5 -999.0 145.0
## [2,] 312.8 318.6 611.5  166.1  410.5  519.7  176  512.5 257.0  714.9 268.8
##      HT14.5  HT15 HT15.5   HT16 HT16.5  HT17 HT17.5 HT18
## [1,] -999.0 150.8  152.1 -999.0  152.6   0.0  107.0    0
## [2,]  614.9 285.6  192.2  715.5  255.6 616.9  617.8  819
```

Clearly −999 and 0 are not valid heights. What's going on?

Many statistical computing systems — I don't want to say "languages" because many competitors to R are not real computer languages — do not have a built-in way to handle missing data, like R's predecessor S has had since its beginning. So users pick an impossible value to indicate missing data. But why some NA, some −999, and some 0?

```
hasNA <- apply(foo, 1, anyNA)
has999 <- apply(foo == (-999), 1, any)
has0 <- apply(foo == 0, 1, any)
sort(unique(growth$SITE[hasNA]))
```

```
## [1] "C" "D"
```

```
sort(unique(growth$SITE[has999]))
```

```
## [1] "A"
```

```
sort(unique(growth$SITE[has0]))
```

```
## [1] "B"
```

So clearly the different "sites" coded missing data differently. Now that we understand that we can

- fix the different missing data codes, and
- be on the lookout for what else the different "sites" may have done differently.

Fix.

```
foo[foo == -999] <- NA
foo[foo == 0] <- NA
min(foo, na.rm = TRUE)
```

```
## [1] 9.4
```

3.2.4 Impossible Data

3.2.4.1 Finding the Problems

Clearly, people don't decrease in height as they age (at least when they are young). So that is another sanity check.

```
bar <- apply(foo, 1, diff)
sum(bar < 0)
```

```
## [1] NA
```

Hmmmmm. Didn't work because of the missing data. Try again.

```
bar <- apply(foo, 1, function(x) {
    x <- x[! is.na(x)]
    diff(x)
})
class(bar)
```

```
## [1] "list"
```

according to the documentation to `apply` it returns a list when the function returns vectors of different lengths for different "margins" (here rows).

```
any(sapply(bar, function(x) any(x < 0)))
```

```
## [1] TRUE
```

So we do have some impossible data. Is it just slightly impossible or very impossible?

```
baz <- sort(unlist(lapply(bar, function(x) x[x < 0])))
length(baz)
```

```
## [1] 186
```

```
range(baz)
```

```
## [1] -539.4   -0.1
```

The -0.1 may be a negligible error, but the -539.4 is highly impossible. We've got work to do on this issue.

3.2.4.2 Fixing the Problems

At this point anyone (even me) would be tempted to just give up using R or any other computer language to work on this issue. It is just too messy. Suck the data into a spreadsheet or other editor, fix it by hand, and be done with it.

But this is not reproducible and not scalable. There is no way anyone (even the person doing the data editing) can reproduce exactly what they did or explain what they did. So why should we trust that? We shouldn't. Moreover, for "big data" it is a nonstarter. A human can fix a little bit of the data, but we need an automatic process if we are going to fix all the data. Hence we keep plugging away with R.

Any negative increment between heights may be because of either of the heights being subtracted being wrong. And just looking at those two numbers, we cannot tell which is wrong. So let us also look at he two numbers to either side (if there are such).

This job is so messy I think we need loops.

```
qux <- NULL
for (i in 1:nrow(foo))
    for (j in seq(1, ncol(foo) - 1))
        if (foo[i, j + 1] < foo[i, j]) {
            below <- if (j - 1 >= 1) foo[i, j - 1] else NA
            above <- if (j + 2 <= ncol(foo)) foo[i, j + 2] else NA
            qux <- rbind(qux, c(below, foo[i, j], foo[i, j + 1], above))
        }
```

```
## Error in if (foo[i, j + 1] < foo[i, j]) {: missing value where TRUE/FALSE needed
```

```
qux
```

```
##       HT1 HT1.25 HT1.5 HT1.75
## [1,] 73.8   78.7    38   85.8
```

That didn't work. Forgot about the `NA`'s. Try again.

```
qux <- NULL
for (i in 1:nrow(foo)) {
    x <- foo[i, ]
    x <- x[! is.na(x)]
    d <- diff(x)
    jj <- which(d < 0)
    for (j in jj) {
        below <- if (j - 1 >= 1) x[j - 1] else NA
        above <- if (j + 2 <= length(x)) x[j + 2] else NA
        qux <- rbind(qux, c(below, x[j], x[j + 1], above))
    }
}
qux
```

```
##          HT1 HT1.25 HT1.5 HT1.75
##    [1,]  73.8   78.7  38.0   85.8
##    [2,] 171.1  185.2 180.8  182.2
##    [3,] 104.1  121.5 119.0  128.0
##    [4,] 136.6  149.5 141.9  144.8
##    [5,] 152.3  156.3 116.2  165.9
##    [6,] 177.3  177.6 177.3  177.0
##    [7,] 177.6  177.3 177.0  177.0
##    [8,] 131.4  144.3 136.8  139.7
##    [9,] 154.0  518.0 162.1  166.5
##   [10,] 153.3  157.7 152.3  166.1
##   [11,] 171.1  714.9 177.9  180.1
##   [12,] 165.5  196.7 173.1  175.6
##   [13,] 179.3  189.8 180.3  180.7
##   [14,] 134.1  137.1 104.2  413.0
##   [15,] 104.2  413.0 145.7  148.5
##   [16,] 158.4  161.2 155.2  166.0
##   [17,] 119.4  221.9 124.2  126.3
##   [18,] 140.5  243.6 146.8  149.0
##   [19,] 164.2  164.6 163.9  165.0
##   [20,] 137.1  139.5 132.9  145.0
##   [21,] 161.8  262.8 163.5  164.0
##   [22,] 161.4  262.4 163.4  154.0
##   [23,] 262.4  163.4 154.0  164.3
##   [24,] 142.7  146.1 140.9  254.1
##   [25,] 140.9  254.1 158.8  160.4
##   [26,] 163.0  163.1 163.0  163.1
##   [27,] 163.7  163.8 163.7     NA
##   [28,]  78.3   92.4  87.2   91.4
##   [29,] 170.4  171.3 127.0  162.7
##   [30,] 129.8  312.8 125.2  137.6
##   [31,]  75.9   79.4  72.1   85.7
##   [32,] 115.3  124.0 103.6  137.6
##   [33,] 144.8  248.0 151.8  155.3
##   [34,] 175.8  176.1 176.0     NA
##   [35,] 143.0  164.7 150.5  153.7
##   [36,] 218.8  313.8 136.9  139.9
##   [37,] 172.2  172.3 171.2  172.5
##   [38,] 110.0  116.1 110.9  123.6
##   [39,] 150.3  161.0 151.4  151.7
##   [40,]  86.2   96.5  14.1  110.8
##   [41,] 125.2  310.9 133.4  135.9
##   [42,] 173.8  175.0 174.7  176.1
##   [43,] 176.5  616.9 177.2  177.5
##   [44,] 168.1  168.4 158.9  169.4
##   [45,] 116.2  134.0 131.1  136.8
##   [46,] 148.1  161.3 154.8  158.5
##   [47,] 158.4  171.7 164.2  165.6
##   [48,] 179.6  189.2 181.6     NA
##   [49,] 149.6  162.2 155.3  159.2
##   [50,] 176.1  178.3 177.9  178.2
##   [51,]  81.3   85.1  39.6  101.9
##   [52,] 130.9  143.1 137.8  139.3
##   [53,] 151.7  155.0 148.9  163.3
##  [54,]  91.4  201.3 108.0  121.5
##  [55,] 163.6  166.6 160.9  175.0
##  [56,]  78.0   80.6  48.4   90.2
##  [57,] 175.5  176.0 166.4     NA
##  [58,] 165.9  166.1 166.0  176.0
##  [59,]  81.8   95.4  77.9   89.6
##  [60,] 161.8  161.8 161.7  161.9
##  [61,] 161.9  161.9 161.7  161.9
##  [62,] 106.7  123.3 118.4  124.2
##  [63,] 164.4  165.8 164.9     NA
##  [64,] 154.2  257.0 159.0  160.5
##  [65,] 163.0  163.4 163.1  163.0
##  [66,] 163.4  163.1 163.0     NA
##  [67,] 153.6  154.1 145.4  154.6
##  [68,] 135.8  318.6 141.4  144.2
##  [69,] 167.8  168.6 159.2  169.8
##  [70,] 169.8  169.8 107.0  170.3
##  [71,] 141.8  154.8 150.5  155.2
##  [72,] 168.6  617.8 169.2     NA
##  [73,]  85.1   87.5  87.3  105.7
##  [74,] 142.0  415.0 148.2  152.0
##  [75,] 167.6  268.8 169.5  169.9
##  [76,]  86.4   89.7  69.6  105.0
##  [77,] 105.0  112.3 110.6  124.7
##  [78,] 133.6  145.8 138.8  141.2
##  [79,] 167.3  167.3 167.2     NA
##  [80,] 137.4  410.5 134.3  145.9
##  [81,] 172.4  182.5 172.5  127.5
##  [82,] 182.5  172.5 127.5     NA
##  [83,]  79.1   91.1  84.4   87.4
##  [84,] 121.9  217.8 129.8  132.4
##  [85,] 172.6  273.2 173.8     NA
##  [86,]  76.0   79.4  28.0   84.2
##  [87,] 171.1  271.8 127.6     NA
##  [88,] 153.6  156.8 150.5  164.6
##  [89,] 184.2  285.6 186.6  187.1
##  [90,] 187.4  287.8 188.2  188.4
##  [91,]  83.7   86.3  79.8   92.2
##  [92,] 145.6  148.7 142.3  157.0
##  [93,] 162.0  176.4 169.9  172.1
##  [94,] 184.7  715.5 176.1  176.4
##  [95,] 122.7  129.9 123.1  136.2
##  [96,] 178.3  191.0 183.1  184.6
##  [97,]  78.3   82.3  68.0   89.1
##  [98,] 179.8  189.4 180.8  180.9
##  [99,] 180.8  180.9 180.8  181.3
## [100,] 155.9  611.5 166.2  169.6
## [101,] 174.2  174.2 174.1     NA
## [102,]  76.2   90.4  83.3   85.7
## [103,] 134.2  139.6 132.0  138.5
## [104,] 132.0  138.5 131.6  144.9
## [105,] 163.6  165.3 164.5  164.4
## [106,] 165.3  164.5 164.4  164.3
## [107,] 164.5  164.4 164.3  164.1
## [108,] 164.4  164.3 164.1  164.0
## [109,] 164.3  164.1 164.0  163.8
## [110,] 164.1  164.0 163.8     NA
## [111,]  84.4   87.1  49.6  103.4
## [112,] 128.1  230.4 133.2  135.8
## [113,] 145.2  184.5 151.3  154.6
## [114,]  81.3   95.0  87.6   94.6
## [115,] 122.7  139.4 132.4  135.6
## [116,] 159.4  159.7 150.9  160.2
## [117,] 160.4  160.4 160.3  160.3
## [118,]  77.0   87.5  19.1   94.0
## [119,] 169.3  196.6 169.8  170.1
## [120,] 170.3  180.5 170.6  170.8
## [121,] 133.2  236.6 139.8  142.7
## [122,] 171.5  171.5 171.4  171.2
## [123,] 171.5  171.4 171.2     NA
## [124,]    NA   74.9  67.8   81.2
## [125,] 163.6  163.7 153.8     NA
## [126,] 132.6  236.1 139.4  142.4
## [127,] 162.1  166.0 160.3  174.3
## [128,] 180.4  180.4 180.2  180.5
## [129,] 163.8  614.9 175.8  166.8
## [130,] 614.9  175.8 166.8  168.9
## [131,] 166.8  168.9 168.2  168.2
## [132,] 168.2  168.2 168.1     NA
## [133,] 142.4  154.0 148.1  151.8
## [134,] 155.5  185.3 160.1  161.3
## [135,] 162.3  173.0 163.3  163.6
## [136,] 163.3  163.6 153.8  164.2
## [137,] 152.9  153.2 143.6  154.1
## [138,] 154.4  154.6 154.5     NA
## [139,]  84.2   96.4  94.0  101.8
## [140,] 121.9  126.8 119.3  121.9
## [141,] 163.6  163.9 154.3  164.2
## [142,] 154.3  164.2 164.0     NA
## [143,] 161.0  161.5 161.0  161.8
## [144,] 161.0  161.8 161.6     NA
## [145,] 125.5  128.0 103.3  132.5
## [146,] 142.1  244.2 146.4  148.9
## [147,] 144.4  158.4 152.5  156.6
## [148,] 156.6  611.5 166.1  170.2
## [149,] 182.6  182.8 182.7  183.2
## [150,] 172.6  172.9 137.1  173.3
## [151,]  78.2   84.2  78.0   88.4
## [152,] 168.7  169.1 160.5  169.9
## [153,] 170.1  179.3 170.3  170.6
## [154,] 100.2  180.0 114.9  121.2
## [155,] 131.3  143.9 136.7  140.1
## [156,] 156.9  157.0 156.9     NA
## [157,] 152.2  163.6 154.9  155.9
## [158,] 157.5  175.7 158.0  518.4
## [159,] 155.1  255.6 156.0  157.1
## [160,] 156.0  157.1 156.5     NA
## [161,] 133.1  136.0 128.8  142.9
## [162,] 156.2  519.7 163.8  168.8
## [163,] 162.3  177.7 171.5  174.3
## [164,] 174.3  176.1 167.4     NA
## [165,]    NA   84.1  78.4   82.6
## [166,] 123.2  129.9 123.0  136.0
## [167,] 154.5  157.7 152.2  167.4
## [168,]  85.0   86.4  77.1   96.2
## [169,] 117.3  214.1 130.0  133.6
## [170,] 148.7  252.3 154.3  158.1
## [171,] 167.5  181.5 175.1  178.0
## [172,]  95.6  120.1 109.2  117.5
## [173,] 173.0  276.3 178.5  180.0
## [174,] 180.0  181.0 171.8  182.2
## [175,]  95.6  120.6 109.7  126.8
## [176,] 109.7  126.8 121.7  127.2
## [177,] 178.0  179.1 108.2  181.0
## [178,] 118.6  133.7 126.4  129.1
## [179,] 102.0  207.5 114.0  119.4
## [180,] 131.6  143.0 136.5  138.8
## [181,] 104.2  210.0 116.0  123.8
## [182,] 150.1  512.5 155.1  158.4
## [183,]  83.3   87.0   9.4   93.1
## [184,] 153.4  156.1 149.3  163.7
## [185,] 189.6  199.7 191.3  191.7
## [186,] 136.8  239.8 142.6  145.1
```

In line 1 it looks like the data enterer transposed digits. These data would make sense if the 38.0 was actually 83.0. In line 2 it looks like the data enterer had an off-by-one error. These data would make sense if the 185.2 was actually 175.2. In fact, those are the two kinds of errors I put in the data. But in real life, we wouldn't know about the kinds of all of the errors in the data. There might be other kinds. Or our guess about these kinds might be wrong.

At this point and perhaps long before, we would have gone back to the data source and asked if the data are correctable at the source. Perhaps the data were entered correctly and corrupted later and we can get the original version. But the kinds of errors we think we found are apparently data entry errors. So there may be no correct data available.

In lines 26 and 27 we notice that the errors are negligible (only 0.1 in size). Perhaps those we can ignore. They might just be different rounding before data entry.

In catching these errors — it is pretty clear that there is no way we can "correct" these errors if correct data are unavailable — we don't want to be clumsy and introduce more error. We want to use the best methods we can. We're statisticians, so perhaps we should use statistics. We need to use the whole data for an individual to identify the errors for that individual.

So let's go back and find which individuals have erroneous data. And while we are at it, let's skip errors less than 0.3 in size.

```
qux <- NULL
for (i in 1:nrow(foo)) {
    x <- foo[i, ]
    x <- x[! is.na(x)]
    d <- diff(x)
    jj <- which(d <= -0.2)
    for (j in jj) {
        below <- if (j - 1 >= 1) x[j - 1] else NA
        above <- if (j + 2 <= length(x)) x[j + 2] else NA
```

```r
        qux <- rbind(qux, c(i, below, x[j], x[j + 1], above))
    }
}
qux
```

```
##             HT1 HT1.25 HT1.5 HT1.75
##   [1,]  1  73.8   78.7  38.0   85.8
##   [2,]  1 171.1  185.2 180.8  182.2
##   [3,]  2 104.1  121.5 119.0  128.0
##   [4,]  2 136.6  149.5 141.9  144.8
##   [5,]  2 152.3  156.3 116.2  165.9
##   [6,]  2 177.3  177.6 177.3  177.0
##   [7,]  2 177.6  177.3 177.0  177.0
##   [8,]  3 131.4  144.3 136.8  139.7
##   [9,]  3 154.0  518.0 162.1  166.5
##  [10,]  4 153.3  157.7 152.3  166.1
##  [11,]  6 171.1  714.9 177.9  180.1
##  [12,]  7 165.5  196.7 173.1  175.6
##  [13,]  7 179.3  189.8 180.3  180.7
##  [14,]  8 134.1  137.1 104.2  413.0
##  [15,]  8 104.2  413.0 145.7  148.5
##  [16,]  8 158.4  161.2 155.2  166.0
##  [17,]  9 119.4  221.9 124.2  126.3
##  [18,]  9 140.5  243.6 146.8  149.0
##  [19,] 10 164.2  164.6 163.9  165.0
##  [20,] 11 137.1  139.5 132.9  145.0
##  [21,] 11 161.8  262.8 163.5  164.0
##  [22,] 12 161.4  262.4 163.4  154.0
##  [23,] 12 262.4  163.4 154.0  164.3
##  [24,] 13 142.7  146.1 140.9  254.1
##  [25,] 13 140.9  254.1 158.8  160.4
##  [26,] 14  78.3   92.4  87.2   91.4
##  [27,] 14 170.4  171.3 127.0  162.7
##  [28,] 16 129.8  312.8 125.2  137.6
##  [29,] 17  75.9   79.4  72.1   85.7
##  [30,] 17 115.3  124.0 103.6  137.6
##  [31,] 17 144.8  248.0 151.8  155.3
##  [32,] 18 143.0  164.7 150.5  153.7
##  [33,] 19 218.8  313.8 136.9  139.9
##  [34,] 19 172.2  172.3 171.2  172.5
##  [35,] 20 110.0  116.1 110.9  123.6
##  [36,] 20 150.3  161.0 151.4  151.7
##  [37,] 21  86.2   96.5  14.1  110.8
##  [38,] 21 125.2  310.9 133.4  135.9
##  [39,] 21 173.8  175.0 174.7  176.1
##  [40,] 21 176.5  616.9 177.2  177.5
##  [41,] 22 168.1  168.4 158.9  169.4
##  [42,] 23 116.2  134.0 131.1  136.8
##  [43,] 23 148.1  161.3 154.8  158.5
##  [44,] 24 158.4  171.7 164.2  165.6
##  [45,] 26 179.6  189.2 181.6     NA
##  [46,] 27 149.6  162.2 155.3  159.2
##  [47,] 28 176.1  178.3 177.9  178.2
##  [48,] 29  81.3   85.1  39.6  101.9
##  [49,] 29 130.9  143.1 137.8  139.3
##  [50,] 29 151.7  155.0 148.9  163.3
##  [51,] 30  91.4  201.3 108.0  121.5
##  [52,] 30 163.6  166.6 160.9  175.0
##  [53,] 31  78.0   80.6  48.4   90.2
##  [54,] 32 175.5  176.0 166.4     NA
##  [55,] 34  81.8   95.4  77.9   89.6
##  [56,] 34 161.9  161.9 161.7  161.9
##  [57,] 35 106.7  123.3 118.4  124.2
##  [58,] 35 164.4  165.8 164.9     NA
##  [59,] 36 154.2  257.0 159.0  160.5
##  [60,] 36 163.0  163.4 163.1  163.0
##  [61,] 37 153.6  154.1 145.4  154.6
##  [62,] 38 135.8  318.6 141.4  144.2
##  [63,] 38 167.8  168.6 159.2  169.8
##  [64,] 38 169.8  169.8 107.0  170.3
##  [65,] 40 141.8  154.8 150.5  155.2
##  [66,] 41 168.6  617.8 169.2     NA
##  [67,] 42  85.1   87.5  87.3  105.7
##  [68,] 42 142.0  415.0 148.2  152.0
##  [69,] 42 167.6  268.8 169.5  169.9
##  [70,] 43  86.4   89.7  69.6  105.0
##  [71,] 43 105.0  112.3 110.6  124.7
##  [72,] 43 133.6  145.8 138.8  141.2
##  [73,] 45 137.4  410.5 134.3  145.9
##  [74,] 45 172.4  182.5 172.5  127.5
##  [75,] 45 182.5  172.5 127.5     NA
##  [76,] 46  79.1   91.1  84.4   87.4
##  [77,] 46 121.9  217.8 129.8  132.4
##  [78,] 46 172.6  273.2 173.8     NA
##  [79,] 47  76.0   79.4  28.0   84.2
##  [80,] 47 171.1  271.8 127.6     NA
##  [81,] 49 153.6  156.8 150.5  164.6
##  [82,] 49 184.2  285.6 186.6  187.1
##  [83,] 49 187.4  287.8 188.2  188.4
##  [84,] 50  83.7   86.3  79.8   92.2
##  [85,] 51 145.6  148.7 142.3  157.0
##  [86,] 51 162.0  176.4 169.9  172.1
##  [87,] 51 184.7  715.5 176.1  176.4
##  [88,] 52 122.7  129.9 123.1  136.2
##  [89,] 52 178.3  191.0 183.1  184.6
##  [90,] 53  78.3   82.3  68.0   89.1
##  [91,] 53 179.8  189.4 180.8  180.9
##  [92,] 54 155.9  611.5 166.2  169.6
##  [93,] 55  76.2   90.4  83.3   85.7
##  [94,] 56 134.2  139.6 132.0  138.5
##  [95,] 56 132.0  138.5 131.6  144.9
##  [96,] 56 163.6  165.3 164.5  164.4
##  [97,] 56 164.4  164.3 164.1  164.0
##  [98,] 57  84.4   87.1  49.6  103.4
##  [99,] 57 128.1  230.4 133.2  135.8
## [100,] 57 145.2  184.5 151.3  154.6
## [101,] 58  81.3   95.0  87.6   94.6
## [102,] 58 122.7  139.4 132.4  135.6
## [103,] 58 159.4  159.7 150.9  160.2
## [104,] 59  77.0   87.5  19.1   94.0
## [105,] 59 169.3  196.6 169.8  170.1
## [106,] 59 170.3  180.5 170.6  170.8
## [107,] 60 133.2  236.6 139.8  142.7
## [108,] 60 171.5  171.4 171.2     NA
## [109,] 61    NA   74.9  67.8   81.2
## [110,] 61 163.6  163.7 153.8     NA
## [111,] 62 132.6  236.1 139.4  142.4
## [112,] 62 162.1  166.0 160.3  174.3
## [113,] 62 180.4  180.4 180.2  180.5
## [114,] 63 163.8  614.9 175.8  166.8
## [115,] 63 614.9  175.8 166.8  168.9
## [116,] 63 166.8  168.9 168.2  168.2
## [117,] 64 142.4  154.0 148.1  151.8
## [118,] 64 155.5  185.3 160.1  161.3
## [119,] 64 162.3  173.0 163.3  163.6
## [120,] 64 163.3  163.6 153.8  164.2
## [121,] 66 152.9  153.2 143.6  154.1
## [122,] 67  84.2   96.4  94.0  101.8
## [123,] 67 121.9  126.8 119.3  121.9
## [124,] 67 163.6  163.9 154.3  164.2
## [125,] 68 161.0  161.5 161.0  161.8
## [126,] 68 161.0  161.8 161.6     NA
## [127,] 71 125.5  128.0 103.3  132.5
## [128,] 71 142.1  244.2 146.4  148.9
## [129,] 72 144.4  158.4 152.5  156.6
## [130,] 72 156.6  611.5 166.1  170.2
## [131,] 73 172.6  172.9 137.1  173.3
## [132,] 74  78.2   84.2  78.0   88.4
## [133,] 74 168.7  169.1 160.5  169.9
## [134,] 74 170.1  179.3 170.3  170.6
## [135,] 75 100.2  180.0 114.9  121.2
## [136,] 76 131.3  143.9 136.7  140.1
## [137,] 77 152.2  163.6 154.9  155.9
## [138,] 77 157.5  175.7 158.0  518.4
## [139,] 78 155.1  255.6 156.0  157.1
## [140,] 78 156.0  157.1 156.5     NA
## [141,] 79 133.1  136.0 128.8  142.9
## [142,] 80 156.2  519.7 163.8  168.8
## [143,] 81 162.3  177.7 171.5  174.3
## [144,] 81 174.3  176.1 167.4     NA
## [145,] 82    NA   84.1  78.4   82.6
## [146,] 82 123.2  129.9 123.0  136.0
## [147,] 82 154.5  157.7 152.2  167.4
## [148,] 83  85.0   86.4  77.1   96.2
## [149,] 83 117.3  214.1 130.0  133.6
## [150,] 84 148.7  252.3 154.3  158.1
## [151,] 84 167.5  181.5 175.1  178.0
## [152,] 85  95.6  120.1 109.2  117.5
## [153,] 85 173.0  276.3 178.5  180.0
## [154,] 85 180.0  181.0 171.8  182.2
## [155,] 86  95.6  120.6 109.7  126.8
## [156,] 86 109.7  126.8 121.7  127.2
## [157,] 87 178.0  179.1 108.2  181.0
## [158,] 88 118.6  133.7 126.4  129.1
## [159,] 89 102.0  207.5 114.0  119.4
## [160,] 89 131.6  143.0 136.5  138.8
## [161,] 90 104.2  210.0 116.0  123.8
## [162,] 90 150.1  512.5 155.1  158.4
## [163,] 91  83.3   87.0   9.4   93.1
## [164,] 91 153.4  156.1 149.3  163.7
## [165,] 91 189.6  199.7 191.3  191.7
## [166,] 93 136.8  239.8 142.6  145.1
```

So let's try a bit of statistical modeling. We know there is a problem with individual 1, so lets work on him or her (we still don't know what the codes are for `SEX`).

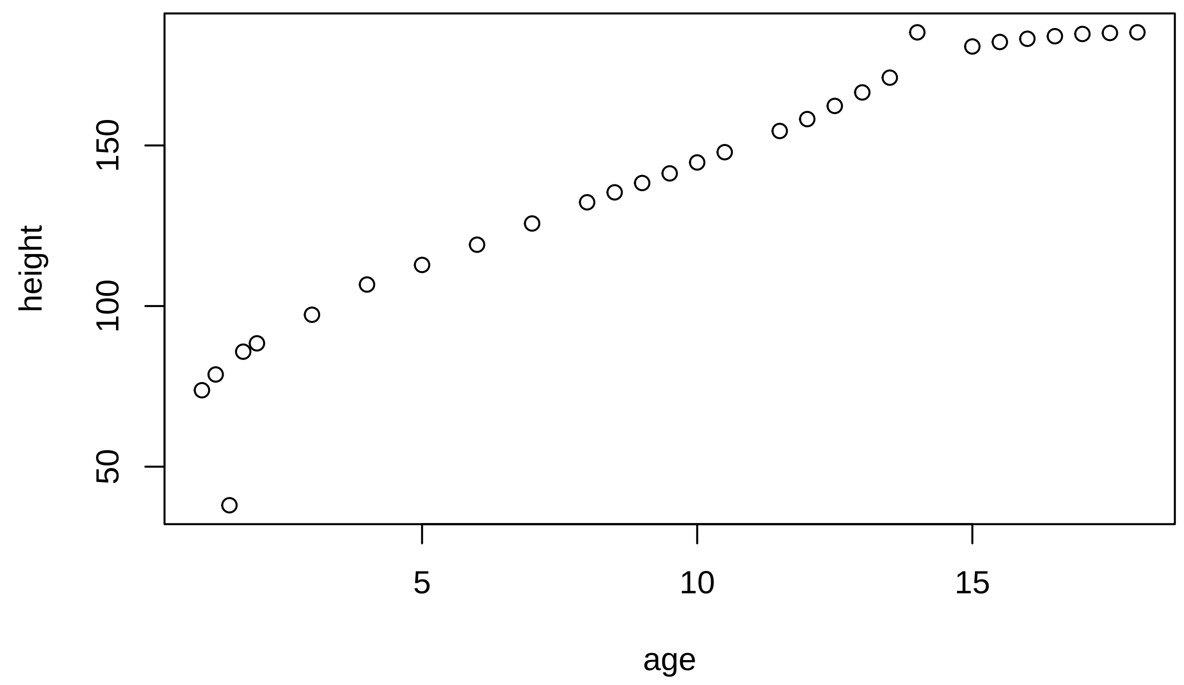
This is always a good idea. Focus on getting one thing right before moving on. I could tell many stories about people coming to me for help with data analysis, and the only problem they had was trying to do too much at once so there was no way to tell what was wrong with what they were doing. At the end, you need to have processed all of the data and done it automatically. But you don't have to start that way.

So individual 1 data.

```
age <- as.numeric(sub("HT", "", colnames(foo)))
age
```

```
##  [1]  1.00  1.25  1.50  1.75  2.00  3.00  4.00  5.00  6.00  7.00  8.00  8.50
## [13]  9.00  9.50 10.00 10.50 11.00 11.50 12.00 12.50 13.00 13.50 14.00 14.50
## [25] 15.00 15.50 16.00 16.50 17.00 17.50 18.00
```

```
plot(age, foo[1, ], ylab = "height")
```

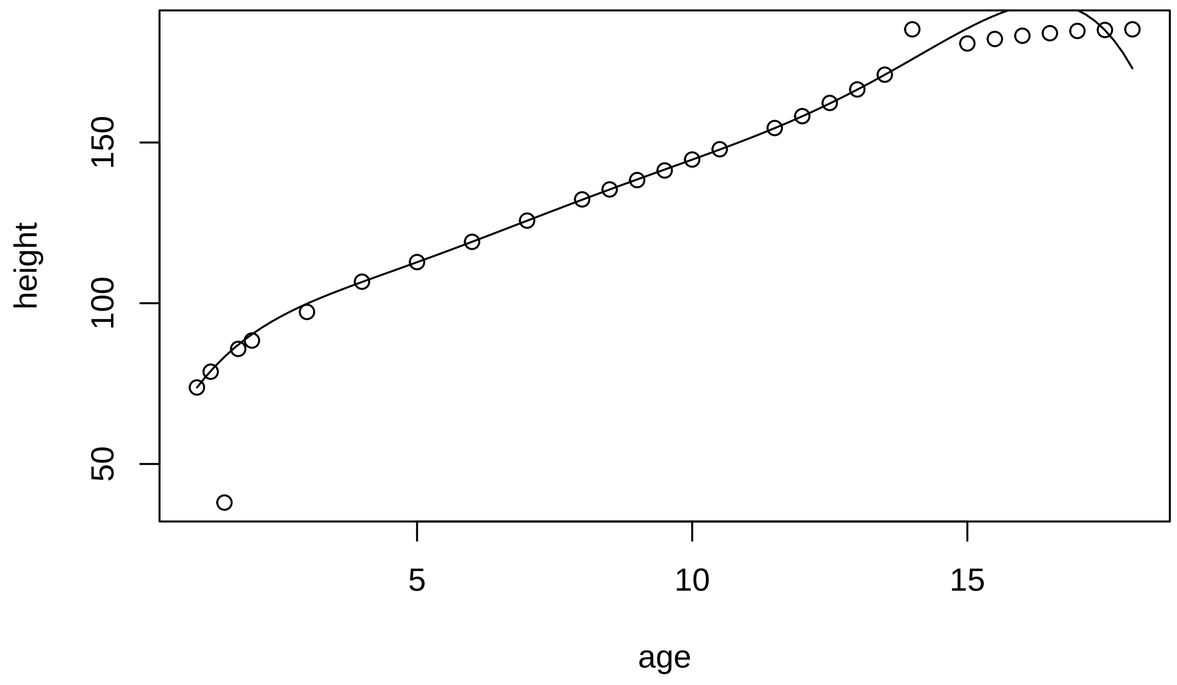


It is pretty clear looking at the picture which points are the gross errors. But can we get statistics to tell us that?

The one thing we know we don't want to use is the usual sort of linear models (those fit by `lm`) because the "errors" are not normal. We want what is called "robust" or "resistant" regression.

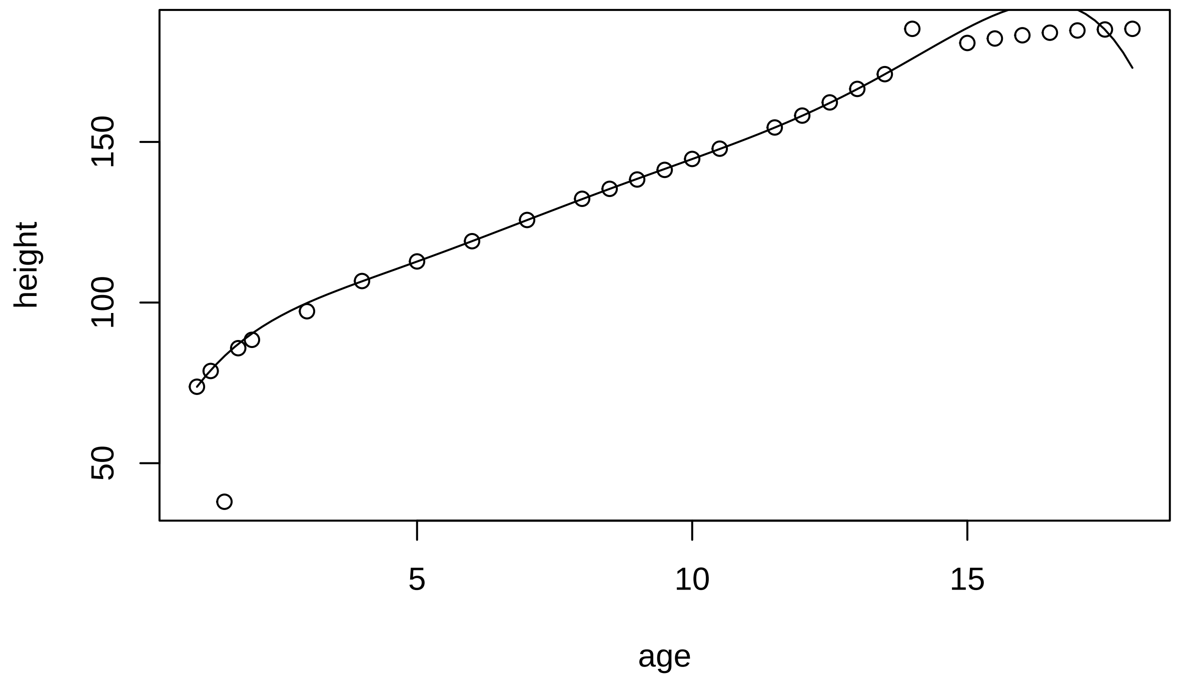
The R command `??robust` turns up the commands `lqs` and `rlm` in the `MASS` (a "recommended" package that comes with every R installation) package and the command `line` in the `stats` package (a core package that comes with every R installation). The `line` function is not going to be helpful because clearly the growth curves curve. So we want to use either `lqs` or `rlm`. Both are complicated. Let us just try `lqs` because it comes first in alphabetical order.

```
plot(age, foo[1, ], ylab = "height")
# R function lqs requires library(MASS) unless already done
lout <- lqs(foo[1, ] ~ poly(age, degree = 6))
curve(predict(lout, newdata = data.frame(age = x)), add = TRUE)
```



Humph! Doesn't seem to fit these data well. Try `rlm`.

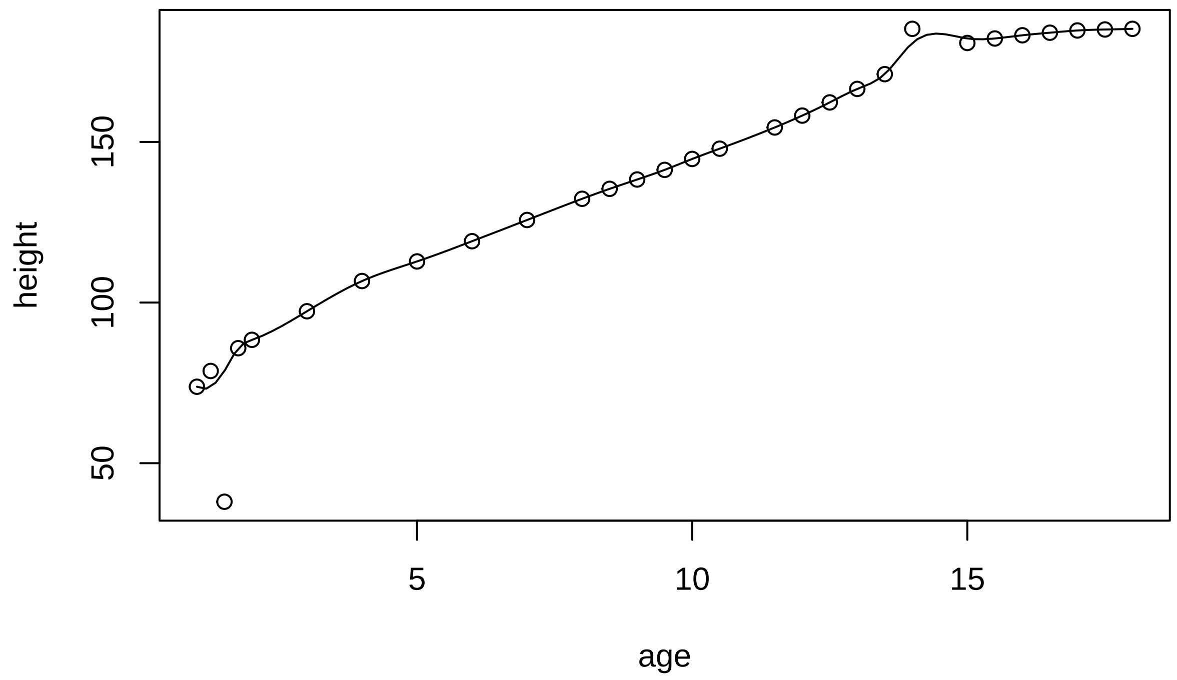
```
plot(age, foo[1, ], ylab = "height")
# R function rlm requires library(MASS) unless already done
rout <- rlm(foo[1, ] ~ poly(age, degree = 6))
curve(predict(lout, newdata = data.frame(age = x)), add = TRUE)
```



Neither of these work because polynomials don't asymptote. Polynomial regression is a horrible tool for curves that asymptote.

Some googling suggested the function `smooth` in the `stats` package. On reading the documentation for that, it is much more primitive and harder to use. But it may work, so let's try it.

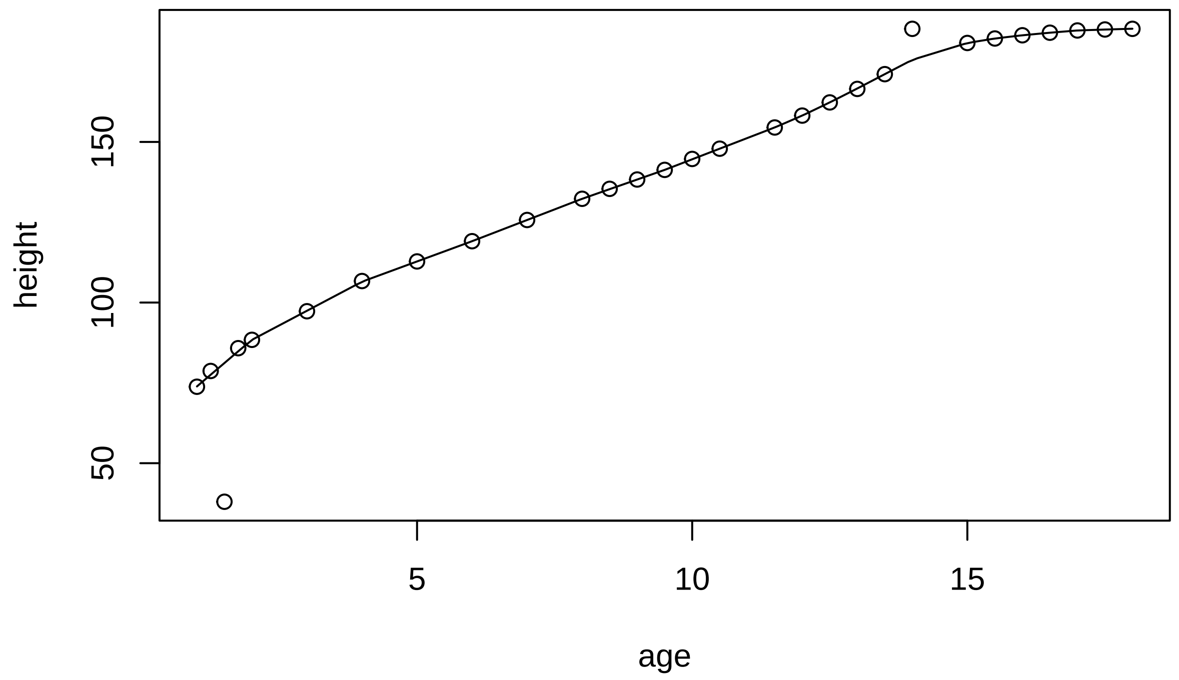
```r
plot(age, foo[1, ], ylab = "height")
y <- foo[1, ]
x <- age[! is.na(y)]
y <- y[! is.na(y)]
sout <- smooth(y)
sally <- splinefun(x, sout)
curve(sally, add = TRUE)
```



Not robust enough.

More googling discovers the CRAN Task View for Robust Statistical Methods in which the only mention of splines is the CRAN package `quantreg`. So we try that.

```
plot(age, foo[1, ], ylab = "height")
y <- foo[1, ]
x <- age[! is.na(y)]
y <- y[! is.na(y)]
lambda <- 0.5 # don't repeat yourself (DRY rule)
# R function rqss  requires library(quantreg) unless already done
qout <- rqss(y ~ qss(x, constraint = "I", lambda = lambda))
curve(predict(qout, newdata = data.frame(x = x)),
    from = min(x), to = max(x), add = TRUE)
```



The model fitting function `rqss` and its method for the generic function `predict` were a lot fussier than those for `lqs` and `rlm`. Like with using `smooth` we had to remove the `NA` values by hand rather than just let the model fitting function take care of them (because `rqss` couldn't take care of them and gave a completely incomprehensible error message). And we had to add optional arguments `from` and `to` to the `curve` function because `predict.rqss` refused to extrapolate beyond the range of the data (this time giving a comprehensible error message).

Anyway, we seem to have got what we want. Now we can compute robust residuals.

```
rresid <- foo[1, ] - as.numeric(predict(qout, newdata = data.frame(x = age)))
rbind(height = foo[1, ], residual = rresid)
```

```
##                  HT1    HT1.25    HT1.5     HT1.75          HT2   HT3   HT4
## height   73.80000000 78.700000  38.0000 85.8000000 8.840000e+01 97.30 106.7
## residual -0.09275655  1.162221 -43.1828  0.9721773 9.355006e-11 -0.15   0.2
##                    HT5           HT6          HT7          HT8 HT8.5
## height    1.128000e+02  1.191000e+02 1.257000e+02 1.323000e+02 135.4
## residual -2.584244e-10 -2.917716e-09 3.795719e-11 3.125223e-09   0.1
##                    HT9         HT9.5  HT10        HT10.5 HT11        HT11.5
## height    1.383000e+02  1.413000e+02 144.7  1.479000e+02   NA  1.545000e+02
## residual -2.804541e-09 -5.930048e-09   0.1 -8.265033e-11   NA -1.577405e-10
##                  HT12        HT12.5        HT13        HT13.5       HT14 HT14.5
## height    1.58200e+02  1.623000e+02 166.5000000  1.711000e+02 185.200000     NA
## residual -1.25624e-11 -1.904255e-12  -0.1302751 -2.606271e-11   9.630275     NA
##                  HT15       HT15.5          HT16        HT16.5          HT17
## height   1.808000e+02  1.82200e+02  1.832000e+02  1.840000e+02  1.847000e+02
## residual 4.263256e-12 -1.29603e-11 -3.728928e-11 -5.570655e-12 -4.279173e-10
##                 HT17.5        HT18
## height   185.00000000 185.20000000
## residual   0.01642436  -0.03749881
```

The robust residuals calculated this way are all small except for the two obvious gross errors. The only one large in absolute value (except for the gross errors) is at the left end of the data, and this is not surprising. All smoothers have trouble at the ends where there is only data on one side to help.

In the fitting we had to choose the `lambda` argument to the `qss` function by hand (because that is what the help page `?qss` says to do), and it did not even tell us whether large `lambda` means more smooth or less smooth. But with some help from what `lambda` does to the residuals, we got a reasonable choice (and perhaps a lot smaller would also do, but we won't bother with more experimentation).

So we want to apply this operation to all the data.

```r
resid <- function(y) {
    idx <- (! is.na(y))
    x <- age[idx]
    y <- y[idx]
    stopifnot(length(x) == length(y))
    qout <- rqss(y ~ qss(x, constraint = "I", lambda = lambda))
    r <- y - as.numeric(predict(qout, newdata = data.frame(x = x)))
    result <- rep(NA_real_, length(idx))
    result[idx] <- r
    result
}
bar <- apply(foo, 1, resid)
```

```
## Error in cbind(deparse.level, ...): args have differing numbers of rows
```

Didn't work. Here we have hit a bizarre bug that I have not been able to isolate. It does occur in R function `rqss`. The fact that the error message is incomprehensible means it is not checking its arguments well enough or that it has an outright bug. But it still may be that the bug is in my code. But if I use a loop instead of `apply`

```r
resids <- matrix(NA_real_, nrow = nrow(foo), ncol = ncol(foo))
for (i in 1:nrow(foo)) {
    y <- foo[i, ]
    idx <- (! is.na(y))
    x <- age[idx]
    y <- y[idx]
    stopifnot(length(x) == length(y))
    qout <- rqss(y ~ qss(x, constraint = "I", lambda = lambda))
    r <- y - as.numeric(predict(qout, newdata = data.frame(x = x)))
    result <- rep(NA_real_, length(idx))
    result[idx] <- r
    resids[i, ] <- result
}
```

```
## Warning in rq.fit.sfnc(x, y, tau = tau, rhs = rhs, control = control, ...): tiny diagonals replaced
```

it works. We do get a different warning (rather than an error), but we also get residuals. The fact that the very same code works when we don't use `apply` and doesn't work when we do use `apply` suggests that R function `rqss` is buggy. But it does not *prove* that. (To *prove* it we would actually have to find the bug and show that fixing the bug fixes the problem.)

Maybe if we change `lambda` we can get rid of the warning.

```r
lambda <- 0.4
resids <- matrix(NA_real_, nrow = nrow(foo), ncol = ncol(foo))
for (i in 1:nrow(foo)) {
    y <- foo[i, ]
    idx <- (! is.na(y))
    x <- age[idx]
    y <- y[idx]
    stopifnot(length(x) == length(y))
    qout <- rqss(y ~ qss(x, constraint = "I", lambda = lambda))
    r <- y - as.numeric(predict(qout, newdata = data.frame(x = x)))
    result <- rep(NA_real_, length(idx))
    result[idx] <- r
    resids[i, ] <- result
}
```

Bingo! That's enough of that. Let us declare that we have robust residuals.

```r
all(is.na(resids) == is.na(foo))
```

```
## [1] TRUE
```

Now we need to select a cutoff

```r
range(resids, na.rm = TRUE)
```

```
## [1] -89.0 539.2
```

```r
stem(resids)
```

```
##
##   The decimal point is 2 digit(s) to the right of the |
##
##   -0 | 987765555
##   -0 | 44444444333321111111111111111111111111111111111111111111000000000+1284
##    0 | 00000000000000000000000000000000000000000000000000000000000000000+1327
##    0 | 7899
##    1 | 00000000000000000000000
##    1 | 7888
##    2 |
##    2 | 777
##    3 |
##    3 | 666
##    4 | 4
##    4 | 5555
##    5 | 44
```

That didn't show much.

```r
bigresid <- abs(as.vector(resids))
bigresid <- bigresid[bigresid > 1]
stem(log10(bigresid))
```

```
##
##   The decimal point is 1 digit(s) to the left of the |
##
##    0 | 000000001224445566666788889990011223355555
##    2 | 0034666623469
##    4 | 012458117
##     6 | 2836
##     8 | 25890233344444445555555566666666677777777778888888888888999999999999
##    10 | 000000000111111225588
##    12 | 245
##    14 | 223333355567
##    16 | 45555633
##    18 | 0556015569999
##    20 | 0000000000000000000
##    22 | 4666
##    24 | 333666
##    26 | 4555533
```

That is still confusing. I had hoped there would be an obvious separation between small OK residuals (less than 1, which is what we have already removed) and the big bad residuals. But it seems to be a continuum. Let us decide that all of the residuals greater than 0.8 on the log scale, which is $10^{0.8} = 6.3095734$ without logs are bad.

```
outies <- log10(abs(resids)) > 0.8
outies[is.na(outies)] <- FALSE
foo[outies] <- NA
```

And now we should redo our whole analysis above and see how big our problems still are.

```
qux <- NULL
for (i in 1:nrow(foo)) {
    x <- foo[i, ]
    x <- x[! is.na(x)]
    d <- diff(x)
    jj <- which(d <= -0.2)
    for (j in jj) {
        below <- if (j - 1 >= 1) x[j - 1] else NA
        above <- if (j + 2 <= length(x)) x[j + 2] else NA
        qux <- rbind(qux, c(i, below, x[j], x[j + 1], above))
    }
}
qux
```

```
##             HT4   HT5   HT6   HT7
##  [1,]  2 104.1 121.5 119.0 128.0
##  [2,]  2 177.3 177.6 177.3 177.0
##  [3,]  2 177.6 177.3 177.0 177.0
##  [4,] 10 164.2 164.6 163.9 165.0
##  [5,] 19 172.2 172.3 171.2 172.5
##  [6,] 21 173.8 175.0 174.7 176.1
##  [7,] 28 176.1 178.3 177.9 178.2
##  [8,] 34 161.9 161.9 161.7 161.9
##  [9,] 35 164.4 165.8 164.9    NA
## [10,] 36 163.0 163.4 163.1 163.0
## [11,] 56 134.2 139.6 132.0 138.5
## [12,] 56 163.6 165.3 164.5 164.4
## [13,] 56 164.4 164.3 164.1 164.0
## [14,] 60 171.5 171.4 171.2    NA
## [15,] 62 180.4 180.4 180.2 180.5
## [16,] 63 166.8 168.9 168.2 168.2
## [17,] 68 161.0 161.5 161.0 161.8
## [18,] 68 161.0 161.8 161.6    NA
## [19,] 78 156.0 157.1 156.5    NA
```

Some of these look quite confusing. It is not clear what is going on. Let's stop here, even though we are not completely satisfied. This is enough time spent on this one issue on a completely made up example.

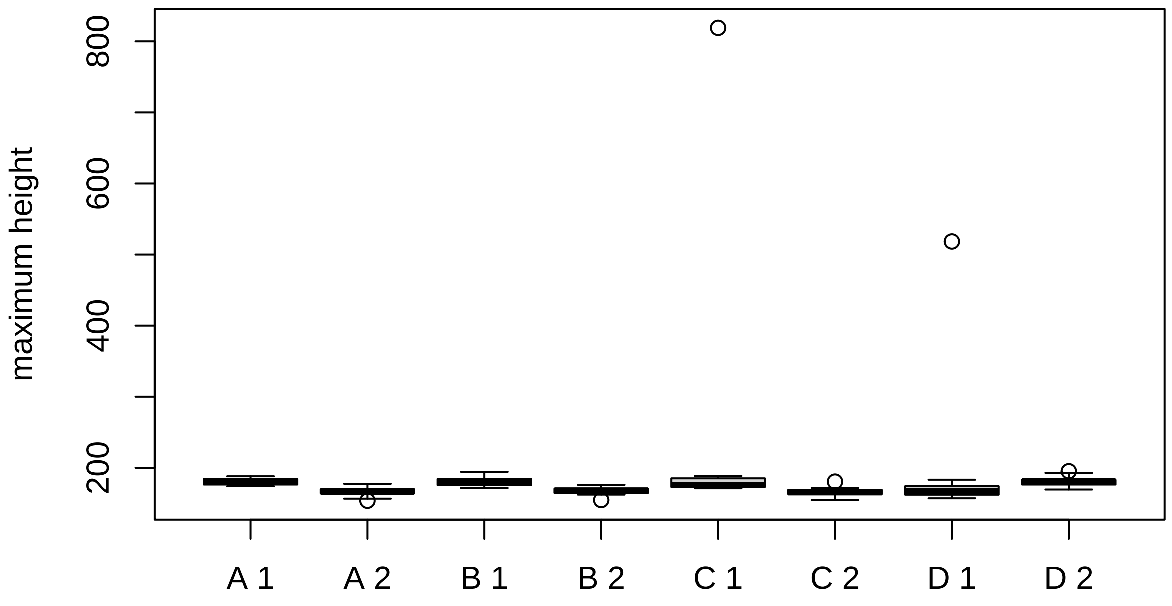
3.2.5 Codes

We still have to figure out what `SEX == 1` and `SEX == 2` mean. And, wary of different sites doing different things, let us look at this per site. There should be height differences at the largest age recorded.

```
maxht <- apply(foo, 1, max, na.rm = TRUE)
sitesex <- with(growth, paste(SITE, SEX))
unique(sitesex)
```

```
## [1] "A 1" "A 2" "B 1" "B 2" "C 1" "C 2" "D 1" "D 2"
```

```
boxplot(split(maxht, sitesex), ylab = "maximum height")
```

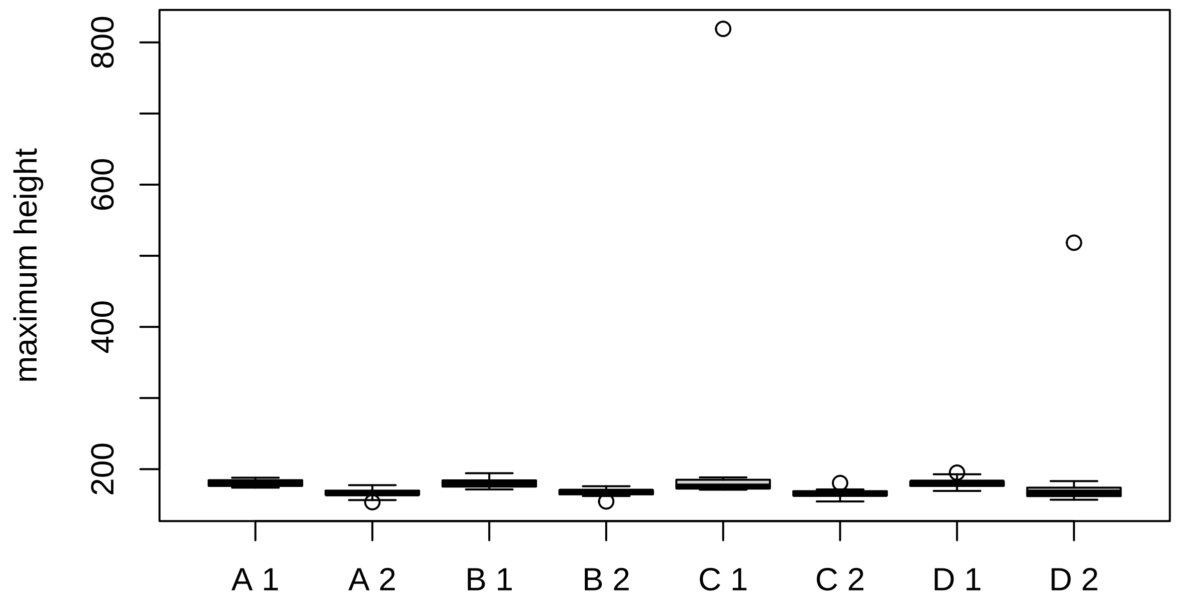


So we see another problem. Sites A, B, and C coded the taller sex (male, presumably) as 1, but site D coded them as 2. So we have to fix that.

```
growth$SEX[growth$SITE == "D"] <- 3 - growth$SEX[growth$SITE == "D"]
sort(unique(growth$SEX))
```

```
## [1] 1 2
```

```
sitesex <- with(growth, paste(SITE, SEX))
boxplot(split(maxht, sitesex), ylab = "maximum height")
```



Looks OK now.

3.2.6 Summary

Finding errors in data is really, really hard. But it is essential.

Our example was hard, we still aren't sure we fixed all the errors. And I cheated. I put the errors in the data in the first place, and then I went and found them (or most of them — it is just impossible to tell whether some of the data entry errors that result in small changes are errors or not).

This example would have been much harder if I did not know what kinds of errors were in the data.

4 Data Scraping

One source of data is the World Wide Web. In general, it is very hard to read data off of web pages. But there is a lot of data there, so it is very useful.

The language HTML in which web pages are coded, is not that easy to parse automatically. There is a CRAN package `xml2` that reads and parses XML data including HTML data. But it is so hard to use that it is beyond the scope of this course. For an example from a more advanced course, see these notes.

Reading data from the web is much easier when the data

- are in an HTML table (surrounded by HTML elements <TABLE> and </TABLE>) and
- the table does not use any stupid tricks for visual look — it is a pure data table. Another way to say this is that all of the presentation is in CSS; the HTML is pure data structure.

In the original version of these notes we used R function `htmltab` in CRAN package `htmltab` but it got kicked of of CRAN for "policy violations". So we are going to switch to using R package `rvest` which is part of the tidyverse, so better supported. This package actually uses package `xml2` under the hood, but it makes it easier to use.

We are still going to stick to just extracting data from tables. Here's how `rvest` does that.

```
u <- "https://www.ncaa.com/rankings/volleyball-women/d1/ncaa-womens-volleyball-rpi/"
# all three R functions in the following command require library(rvest)
# unless already done
foo <- read_html(u) |> html_element("table") |> html_table()
class(foo)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(foo)
```

```
## [1] 344   8
```

```
head(foo)
```

```
## # A tibble: 6 x 8
##    Rank School     Conference Record Road  Neutral Home  `Non Div I`
##   <int> <chr>      <chr>      <chr>  <chr> <chr>   <chr> <chr>
## 1     1 Texas      Big 12     17-1   7-1   0-0     10-0  0-0
## 2     2 Louisville ACC        22-2   9-1   2-0     11-1  0-0
## 3     3 Stanford   Pac-12     18-4   12-1  0-1     6-2   0-0
## 4     4 Nebraska   Big Ten    21-2   9-1   0-0     12-1  0-0
## 5     5 Ohio St.   Big Ten    17-5   9-2   2-1     6-2   0-0
## 6     6 Pittsburgh ACC        24-2   11-0  3-1     10-1  0-0
```

It just grabs the data and puts it in an R dataframe (actually a tibble because it is a tidyverse function, but there isn't much difference).

This uses a pipeline discussed in a section below.

The pipeline is a bit more complicated than using R function `htmltab` (which is no longer available). Now we need three functions in our pipeline. The first reads the web page, the second finds the table in the web page, and the third converts it to a data frame.

Because there was only one HTML table in the document, we got one tibble. Otherwise we could have used a more complicated selection to get the table we wanted (beyond the scope of this course, read about it in the package vignette for `rvest`) or we would have gotten an R list containing tibbles for each table in the document.

5 Web Services

Some web sites have API's that allow computers to talk to them (not just people look at them).

Here is an example using the GitHub public API. It is copied from the vignette `json-apis` for the CRAN package `jsonlite` (for those who don't know JSON is the most popular data interchange format (for computers to talk to computers)).

```
# R function fromJSON requires library(jsonlite) if not already done
foo <- fromJSON("https://api.github.com/users/cjgeyer/repos")
names(foo)
```

```
##  [1] "id"                             "node_id"
##  [3] "name"                           "full_name"
##  [5] "private"                        "owner"
##  [7] "html_url"                    "description"
##  [9] "fork"                        "url"
## [11] "forks_url"                   "keys_url"
## [13] "collaborators_url"           "teams_url"
## [15] "hooks_url"                   "issue_events_url"
## [17] "events_url"                  "assignees_url"
## [19] "branches_url"                "tags_url"
## [21] "blobs_url"                   "git_tags_url"
## [23] "git_refs_url"                "trees_url"
## [25] "statuses_url"                "languages_url"
## [27] "stargazers_url"              "contributors_url"
## [29] "subscribers_url"             "subscription_url"
## [31] "commits_url"                 "git_commits_url"
## [33] "comments_url"                "issue_comment_url"
## [35] "contents_url"                "compare_url"
## [37] "merges_url"                  "archive_url"
## [39] "downloads_url"               "issues_url"
## [41] "pulls_url"                   "milestones_url"
## [43] "notifications_url"           "labels_url"
## [45] "releases_url"                "deployments_url"
## [47] "created_at"                  "updated_at"
## [49] "pushed_at"                   "git_url"
## [51] "ssh_url"                     "clone_url"
## [53] "svn_url"                     "homepage"
## [55] "size"                        "stargazers_count"
## [57] "watchers_count"              "language"
## [59] "has_issues"                  "has_projects"
## [61] "has_downloads"               "has_wiki"
## [63] "has_pages"                   "has_discussions"
## [65] "forks_count"                 "mirror_url"
## [67] "archived"                    "disabled"
## [69] "open_issues_count"           "license"
## [71] "allow_forking"               "is_template"
## [73] "web_commit_signoff_required" "topics"
## [75] "visibility"                  "forks"
## [77] "open_issues"                 "watchers"
## [79] "default_branch"
```

Hmmmmm. An incredible amount of stuff there, most of which I don't understand even though I am a long time github user.

But

```r
foo$name
```

```
##  [1] "AsterPhilo"            "AsterTheory"           "bar"
##  [4] "bernor"                "CatDataAnalysis"       "Evolution-correction"
##  [7] "foo"                   "gdor"                  "glmbb"
## [10] "glmdr"                 "grape"                 "linkingTo"
## [13] "mat"                   "mcmc"                  "nice"
## [16] "Orientation2018"       "polyapost"             "pooh"
## [19] "potts"                 "qux"                   "rcdd"
## [22] "sped"                  "Test"                  "trust"
## [25] "TSHRC"                 "ump"
```

are the names of my github public repos.

Of course, to do anything nontrivial you have to understand the web service. Read up on their API, and so forth.

The only point we are making here is that CRAN has the packages to support this stuff.

6 Searching CRAN

There used to be a JSON API for searching CRAN. This course covered it the last time it was taught, but it is now defunct. It says it has been replaced by R function `advanced_search` in R package `pkgsearch`. So we illustrate that.

```
# function advanced_search requires library(pkgsearch) unless already done
foo <- advanced_search("Geyer", size = 100)
class(foo)
```

```
## [1] "pkg_search_result" "tbl"               "data.frame"
```

```
dim(foo)
```

```
## [1] 21 14
```

```
names(foo)
```

```
##  [1] "score"              "package"             "version"
##  [4] "title"              "description"         "date"
##  [7] "maintainer_name"    "maintainer_email"    "revdeps"
## [10] "downloads_last_month" "license"             "url"
## [13] "bugreports"         "package_data"
```

Woof! 21 CRAN packages, but maybe not all of them are by me.

```
foo
```

```
## - "advanced search" ---------------------------- 21 packages in 0.007 seconds -
##   #     package              version    by                   @ title
##   1 100 glmm                 1.4.4      Christina Knudson   1M Generalized Lin...
##   2  93 fuzzyRankTests       0.4        Charles J. Geyer    1y Fuzzy Rank Test...
##   3  93 trust                0.1.8      Charles J. Geyer    3y Trust Region Op...
##   4  93 aster                1.1.2      Charles J. Geyer    1y Aster Models
##   5  80 aster2               0.3        Charles J. Geyer    6y Aster Models
##   6  80 nice                 0.4.1      Charles J. Geyer    6y Get or Set UNIX...
##   7  79 spark.sas7bdat       1.4        Jan Wijffels        2y Read in 'SAS' D...
##   8  76 pooh                 0.3.2      Charles J. Geyer    6y Partial Orders ...
##   9  76 glmbb                0.5.1      Charles J. Geyer    2y All Hierarchica...
##  10  73 ump                  0.5.8      Charles J. Geyer    6y Uniformly Most ...
##  11  72 partitionComparison 0.2.5      Fabian Ball         4y Implements Meas...
##  12  70 polyapost            1.7        Glen Meeden         1y Simulating from...
##  13  69 mcmc                 0.9.7      Charles J. Geyer    3y Markov Chain Mo...
##  14  68 potts                0.5.11     Charles J. Geyer    3M Markov Chain Mo...
##  15  61 TSHRC                0.1.6      Charles J. Geyer    4y Two Stage Hazar...
##  16  61 audit                0.1.2      Glen Meeden         1y Bounds for Acco...
##  17  60 rcdd                 1.5        Charles J. Geyer    1y Computational G...
##  18  60 CatDataAnalysis      0.1.5      Charles J. Geyer    5M Datasets for Ca...
##  19  60 krm                  2022.10.17 Youyi Fong         26d Kernel Based Re...
##  20  26 sfsmisc              1.1.13     Martin Maechler     7M Utilities from ...
##  21  23 glmmTMB              1.1.4      Mollie Brooks       4M Generalized Lin...
```

Some of these are false positives (authored by other people named Geyer). Let's eliminate those.

```
class(foo$package_data)
```

```
## [1] "AsIs"
```

```
is.list(foo$package_data)
```

```
## [1] TRUE
```

```
length(foo$package_data)
```

```
## [1] 21
```

```
names(foo$package_data[[1]])
```

```
##  [1] "Package"          "Type"             "Maintainer"       "Date"
##  [5] "Title"            "Imports"          "Version"          "License"
##  [9] "Enhances"         "Depends"          "Date/Publication" "VignetteBuilder"
## [13] "RoxygenNote"      "crandb_file_date" "MD5sum"           "Author"
## [17] "LinkingTo"        "NeedsCompilation" "Description"      "Authors@R"
## [21] "Repository"       "ByteCompile"      "revdeps"          "downloads"
## [25] "date"             "Packaged"         "Suggests"
```

```
is.maintainer <- sapply(foo$package_data,
     function(x) grepl("Charles J. Geyer", x$Maintainer))
is.author <- sapply(foo$package_data,
     function(x) grepl("Charles J. Geyer|Charles Geyer", x$Author))
bar <- subset(foo, is.maintainer | is.author)
class(bar)
```

```
## [1] "tbl"        "data.frame"
```

```
bar$package
```

```
##  [1] "glmm"            "fuzzyRankTests"  "trust"           "aster"
##  [5] "aster2"          "nice"            "pooh"            "glmbb"
##  [9] "ump"             "polyapost"       "mcmc"            "potts"
## [13] "TSHRC"           "rcdd"            "CatDataAnalysis" "sfsmisc"
## [17] "glmmTMB"
```

7 Databases

7.1 History

I am not an expert on the history of databases, but at least know there are phases to this history. Most of this relies on the Wikipedia article but has some different emphases.

The history is divided into four eras, or generations, which overlap:

- the dinosaur era (1960's) where there were large databases but they were clumsy to use and relied on highly complicated and usually buggy code written by highly trained programmers,
- the relational and SQL database era (1970's through 1990's) had the following features (not all of which arrived at the same time in the same products):
  - relational databases (Wikipedia article), which to users look like real math: stored tables act like mathematical relations that are "programmed" using mathematical logic via
  - SQL (acronym for *Structured Query Language* but pronounced like the English word "sequel"; Wikipedia article), a standardized computer language for relational database operations, a language just like R or C++ except just for database operations,

    - ACID (acronym for *Atomicity, Consistency, Isolation, Durability*, pronounced like the English word “acid”; Wikipedia article) which describes the highly reliable transactions that are found in modern so-called SQL databases, like Oracle (and many other products),

- the noSQL era (2000’s) in which all of the great ideas of the relational database era were dropped, putting programmers back in the dinosaur era or worse, all in the name of scaling to internet scale (Wikipedia article), leading examples of which are Amazon’s Dynamo, Apache Cassandra, CouchDB, MongoDB, Redis, HBase, and MemcacheDB,
- the newSQL era (now, Wikipedia article) has the best of both worlds, relational, SQL, ACID, and highly scalable, a leading example is Google Spanner.

So while in the 2000’s it looked like SQL was old hat and all “data scientists” needed to learn about noSQL that is now looking dated, although a lot of web services run on noSQL databases.

A word about pronunciation: sometimes SQL is “sequel” and sometimes S-Q-L (sounding the letters). In “Microsoft SQL server”, the SQL is always “sequel”. In Oracle MySQL server, the SQL is always S-Q-L so this is pronounced “my-S-Q-L”. This was originally open source software before acquired by Oracle; its free software successor (fork) is MariaDB.

7.2 SQLite

For learning SQL the greatest thing since sliced bread is SQLite, a relational database with full SQL support that runs as a user application. It is just a software library backed by a file on disk. So you can do little database applications with no expensive database. And you can learn on it.

The author of SQLite pronounces it S-Q-L-ite “like a mineral” but does not object to other pronunciations.

7.3 R and SQL and SQLite

The R package that talks to all SQL databases is CRAN package `DBI` (for database interface). The R package that makes SQLite available to R is CRAN package `RSQLite`.

7.4 Dplyr

R package `dplyr` (which is part of the “tidyverse”) does the amazing trick of making it unnecessary (almost) to use SQL to talk to SQL databases. The best introduction to this package is the main package vignette or the chapter on it in the book *R for Data Science*.

In particular, the main functions in the package and all the jobs they do are summarized in the list in the section Single Table Verbs in the vignette. We are not going to illustrate all of those functions, so if you want to do more with `dplyr` you need to read the vignette.

This is a package for data munging. Originally, it worked on data in R data frames and did not do anything that base R cannot do but did it in different ways that were considered by the package author, Hadley Wickham, to be simpler, more elegant, and easier to use than the base R methods.

But over time R package `dplyr` got a lot more sophisticated. Now it can do all of its operations not only on R data frames but also on *tables in SQL databases!* It can write SQL so you don’t have to!

Moreover, when it is working with SQL databases, it actually does all of the work in the database so R does not have to deal with big data, only with the final results. This is really quite tricky, so we won’t even try to explain it, just leave this by saying it is really cool.

Nevertheless. R package `dplyr` does not know everything there is to know about SQL. This web page says

> As well as working with local in-memory data stored in data frames, `dplyr` also works with remote on-disk data stored in databases. This is particularly useful in two scenarios:
>
> - Your data is already in a database.

> - You have so much data that it does not all fit into memory simultaneously and you need to use some external storage engine.
>
> (If your data fits in memory, there is no advantage to putting it in a database; it will only be slower and more frustrating.)

and also says

> To interact with a database you usually use SQL, the Structured Query Language. SQL is over 40 years old, and is used by pretty much every database in existence. The goal of dbplyr is to automatically generate SQL for you so that you're not forced to use it. However, SQL is a very large language, and dbplyr doesn't do everything.

and

> However, in the long run, I highly recommend you at least learn the basics of SQL. It's a valuable skill for any data scientist, and it will help you debug problems if you run into problems with `dplyr`'s automatic translation.

But we won't try to teach SQL here (there are lots of books and videos and whatnot that do this). We will just show you some `dplyr` working on a database.

7.5 SQL Database Example

7.5.1 The SQLite Database

First we go get a SQLite database.

```
fs <- "https:/www.stat.umn.edu/geyer/8054/data/cran-info.sqlite"
ft <- "cran-info.sqlite"
if (! file.exists(ft)) download.file(fs, ft)
```

Then we start up a connection to this database.

```
# the following R command requires that R packages DBI and RSQLite be
# installed but because we use the :: syntax they do not need to be loaded,
# that is, we do not need library(DBI) or library(RSQLite)
mydb <- DBI::dbConnect(RSQLite::SQLite(), dbname = ft)
```

This database has four tables

```
DBI::dbListTables(mydb)
```

```
## [1] "depends"  "imports"  "linking"  "suggests"
```

7.5.2 Starting to Use Dplyr in the Example

We turn tables in the database into `dplyr` thingummies as follows.

```
# R function tbl requires library(dplyr) unless already done
# many of the R functions in this section are also in this library
depends <- tbl(mydb, "depends")
imports <- tbl(mydb, "imports")
linking <- tbl(mydb, "linking")
suggest <- tbl(mydb, "suggests")
depends
```

```
## # Source:   table<depends> [?? x 2]
## # Database: sqlite 3.39.4 [/home/geyer/ClassNotes/3701/Handouts/data/cran-info.sqlite]
##    packfrom packto
##    <chr>    <chr>
##   1 A3       xtable
##   2 A3       pbapply
##   3 abc      abc.data
##   4 abc      quantreg
##   5 abc      locfit
##   6 abcdeFBA Rglpk
##   7 abcdeFBA rgl
##   8 abcdeFBA corrplot
##   9 abctools abc
##  10 abctools abind
## # ... with more rows
```

And all four tables look the same. Each has two columns named `packfrom` and `packto` both of which are CRAN packages, one of which needs the other (the `packfrom` needs the `packto`). The reason for the four separate tables is that these dependencies are of four different types, listed in four different fields of the DESCRIPTION file of the `packfrom` package. These are

- The `Depends` gives packages that will be attached before the current package (the `packfrom`) when R function `library` or R function `require` is called.
- The `Imports` gives packages whose namespaces are imported from (as specified in the NAMESPACE file) but which do not need to be attached — that is, R functions in the `packto` package are called by R functions in the `packfrom` package, but the `packfrom` package is not attached.
- The `LinkingTo` gives R packages that have header files needed to compile its C/C++ code (this is very specialized, few CRAN packages do this).
- The `Suggests` gives R packages that are not necessarily needed but rather are used only in examples, tests, or vignettes (not in normal usage of the package).

Since these are all different, we should perhaps treat them all differently, but for this example we are going to treat them all the same. The problem we want to do is count all references to a package (all appearances of a package in the `packto` column of any of the four tables).

A reasonable question is why are the data in this form? The reason is that SQL databases have only one data structure: tables, which are equivalent to what R calls data frames. If we had these data in R, we could think of a lot of other ways to structure the data. In an SQL database, we cannot. (Well there is one obvious other structure: just one table with an extra column that says which type of dependence.)

Here goes

```r
foo <- union_all(depends, imports) |>
    union_all(linking) |>
    union_all(suggest) |>
    count(packto, sort = TRUE)
foo
```

```
## # Source:     SQL [?? x 2]
## # Database:   sqlite 3.39.4 [/home/geyer/ClassNotes/3701/Handouts/data/cran-info.sqlite]
## # Ordered by: desc(n)
##    packto         n
##    <chr>      <int>
##  1 knitr       6735
##  2 testthat    6487
##  3 rmarkdown   6113
##  4 Rcpp        4858
##  5 ggplot2     3802
##  6 dplyr       3125
##  7 magrittr    1877
##  8 covr       1807
##  9 rlang      1491
## 10 stringr    1489
## # ... with more rows
```

R functions `union_all` and `count` here are from R package `dplyr`. The former works just like R function `cbind` in base R, but it works on the tables in the database (pasting them together *in the database* not in R). The latter issues a lot of complicated SQL that also works *in the database* to do what we want.

Unlike R function `cbind` R function `union_all` only takes two arguments. Hence the complicated syntax above.

Unlike R function `union`, R function `union_all` does not remove duplicates. It is not clear (this being a toy problem) whether we should use `union_all` or `union`. The tables `depends`, `imports`, and `suggests` are not supposed to have any duplicates (and I think CRAN enforces this). But `linking` is radically different and could have rows duplicating rows in any of the others. So `union_all` says we are counting the duplicate rows.

When we check whether CRAN actually enforces this, we see they don't.

```r
intersect(depends, imports)
```

```
## # Source:   SQL [6 x 2]
## # Database: sqlite 3.39.4 [/home/geyer/ClassNotes/3701/Handouts/data/cran-info.sqlite]
##   packfrom       packto
##   <chr>          <chr>
## 1 FitAR          leaps
## 2 FitAR          ltsa
## 3 cvTools        robustbase
## 4 filehashSQLite DBI
## 5 filehashSQLite filehash
## 6 psychometric   multilevel
```

So we should switch to `union` rather than `union_all` for the first three. Perhaps we still want `union_all` for `linking`.

7.5.3 The R Pipe Operator

The R operator `|>` is the pipe operator. It does function composition.

```r
quote(x |> f() |> g() |> h(y))
```

```
## h(g(f(x)), y)
```

Each left-hand side of `|>` is made the hidden first argument of the function call on the right-hand side of the pipe.

- so `x |> f()` gets turned into `f(x)`
- and when that gets piped into `g()` it gets turned into `g(f(x))`
- and when that gets piped into `h(y)` it gets turned into `h(g(f(x)), y)`

R package `dplyr` is especially written to make maximal use of pipelines. Almost every function in the package has first argument that is a dataframe or the equivalent (here tables in the database) and returns an object of the same kind, so we just move these objects through the pipeline transforming them as they go.

Pipelines are considered easier to read because they read left to right rather than inside out. You do not have to use them, but all of the `dplyr` literature does use them. (A lot of `dplyr` examples use the older pipe operator `%>%` from R package `magrittr` which has the coolest name of any R package ever but has been made mostly obsolete by the addition of a pipe operator to base R.)

Pipelines in computer languages do the same thing as the notion of composition of mathematical functions The mathematical composition operator is $\circ$. The only difference is that it goes in the other direction of the pipeline notation. In our toy example, the math notation for this function is

$$h(g(f(\,\cdot\,)), y)$$

or — when we use composition notation —

$$h(\,\cdot\,, y) \circ g \circ f$$

(so the order of processing is right to left). The point isn't that you want to switch back and forth between R notation and mathematics notation. The point is that this is *real math*.

We should note that the pipe operator does not have to stuff the left-hand side into the first argument of the right-hand side function call. This allows use of functions not designed for pipelining in pipelines. For example, copied from the examples in `help("|>")`

```
mtcars |> subset(cyl == 4) |> lm(mpg ~ disp, data = _)
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = subset(mtcars, cyl == 4))
##
## Coefficients:
## (Intercept)         disp
##     40.8720      -0.1351
```

The underscore is the placeholder, the left-hand side is piped into there.

Of course, this example is silly because it is more easily and more clearly done without pipelines.

```
lm(mpg ~ disp, data = mtcars, subset = cyl == 4)
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars, subset = cyl == 4)
##
## Coefficients:
## (Intercept)         disp
##     40.8720      -0.1351
```

But there are real applications where we have done something complicated to data in a pipeline and then we want to pipe it into a function but not into the first argument of a function. That's what the placeholder is for.

7.5.4 Looking at the SQL

Let's redo this and look at the SQL (even though you are not supposed to understand this)

```
union(depends, imports) |> union(suggest) |> union_all(linking) |>
    count(packto, sort = TRUE) |>
    show_query()
```

```
## <SQL>
## SELECT `packto`, COUNT(*) AS `n`
## FROM (
##   SELECT *
##   FROM `depends`
##   UNION
##   SELECT *
##   FROM `imports`
##   UNION
##   SELECT *
##   FROM `suggests`
##   UNION ALL
##   SELECT *
##   FROM `linking`
## )
## GROUP BY `packto`
## ORDER BY `n` DESC
```

If we knew enough SQL to write all of that, we wouldn't need `dplyr` to work with the database. But we usually don't work with databases. If the data fits in the computer we are using, we just use R. R package `dplyr` makes it easy to use the same code for both cases, so long as we use all `dplyr` (not other function from other packages, not even functions from core R) in our pipelines.

7.5.5 Moving from the Database to R

Now we can get a reasonable amount of data. Let's get the packages that have 100 or more other packages depending on them.

```
filter(foo, n >= 100)
```

```
## Warning: ORDER BY is ignored in subqueries without LIMIT
## i Do you need to move arrange() later in the pipeline or use window_order() instead?
```

```
## # Source:     SQL [?? x 2]
## # Database:   sqlite 3.39.4 [/home/geyer/ClassNotes/3701/Handouts/data/cran-info.sqlite]
## # Ordered by: desc(n)
##    packto           n
##    <chr>        <int>
##  1 BH             278
##  2 DBI            243
##  3 DT             301
##  4 Formula        172
##  5 GGally         103
##  6 Hmisc          301
##  7 R.rsp          308
##  8 R.utils        158
##  9 R6             403
## 10 RColorBrewer   559
## # ... with more rows
```

Humpf! We already had it in order, but let's try what it suggests: move the sort from earlier in the pipeline (in the `count`) to latter (in an `arrange`)

```
bar <- union(depends, imports) |>
    union(suggest) |>
    union_all(linking) |>
    count(packto) |>
    filter(n >= 100) |>
    arrange(desc(n))
bar
```

```
## # Source:     SQL [?? x 2]
## # Database:   sqlite 3.39.4 [/home/geyer/ClassNotes/3701/Handouts/data/cran-info.sqlite]
## # Ordered by: desc(n)
##    packto        n
##    <chr>     <int>
##  1 knitr      6735
##  2 testthat   6487
##  3 rmarkdown  6113
##  4 Rcpp       4858
##  5 ggplot2    3801
##  6 dplyr      3125
##  7 magrittr   1877
##  8 covr       1807
##  9 rlang      1491
## 10 stringr    1489
## # ... with more rows
```

Now no warning. But everything is still in the database. We cannot do anything with it in R (except via R packages `DBI` and `dplyr`). At some point we want to move it to R (when our results are small enough).

```
qux <- collect(bar)
qux
```

```
## # A tibble: 149 x 2
##    packto        n
##    <chr>     <int>
##  1 knitr      6735
##  2 testthat   6487
##  3 rmarkdown  6113
##  4 Rcpp       4858
##  5 ggplot2    3801
##  6 dplyr      3125
##  7 magrittr   1877
##  8 covr       1807
##  9 rlang      1491
## 10 stringr    1489
## # ... with 139 more rows
```

Now R object `qux` is a tibble, which is a tidyverse modification of base R data frames. One difference we notice is that it prints differently. It doesn't show all the rows by default (R function `print.data.frame` does show all of the rows; we would have to combine it with R function `head` to behave like R function `print.tibble`).

But tibble or whatever, `qux` is a regular old R object (not something in the database) so we can use any R functions to operate on it.

Of course, we did not have to do the `collect` in a separate statement. We could have put it in the pipeline too.

```
qux <- union(depends, imports) |>
    union(suggest) |>
    union_all(linking) |>
    count(packto) |>
    filter(n >= 100) |>
    arrange(desc(n)) |>
    collect()
```

7.5.6 Disconnecting from the Database

It may not matter with SQLite, but if one were working with a database on another computer, like Oracle or MySQL or PostgreSQL, one would need to disconnect (in order not to use up the connections allowed).

```
DBI::dbDisconnect(mydb)
```

Now that we have disconnected, nothing in the database is any longer available. But R object `qux` still works, since that was taken out of the database.

```
bar
```

```
## Error: Invalid or closed connection
```

```
qux
```

```
## # A tibble: 149 x 2
##    packto        n
##    <chr>     <int>
##  1 knitr      6735
##  2 testthat   6487
##  3 rmarkdown  6113
##  4 Rcpp       4858
##  5 ggplot2    3801
##  6 dplyr      3125
##  7 magrittr   1877
##  8 covr       1807
##  9 rlang      1491
## 10 stringr    1489
## # ... with 139 more rows
```