Chapter 4
Modeling Runners' Times in the Cherry Blossom Race

4.1 Introduction

In this era of 'free and ubiquitous data' there is tremendous potential to seek out data that can bring insight to a problem we are working on professionally or to a topic of personal interest. For example, we are interested in understanding how people's physical performance changes as they age. One source of data about this comes from road races. Hundreds of thousands of people participate in road races each year; the race organizers collect information about the runners' times and often publish individual-level data on the Web. These free data may provide us with insights to our question about performance and age.

One example of the many annual road races is the Cherry Blossom Ten Mile Run held in Washington D.C. in early April when the cherry trees are typically in bloom. The Cherry Blossom started in 1973 as a training run for elite runners who were planning to compete in the Boston Marathon. It has since grown in popularity and in 2012 nearly 17,000 runners ranging in age from 9 to 89 participated. The race has become so popular that entrants are chosen via a lottery or they guarantee a spot by raising $500 for an official race charity. After each year's race, the organizers publish the results at `http://www.cherryblossom.org/` (see Figure 4.1). These data offer a tremendous resource for learning about the relationship between age and performance.

4.1.1 Main Tasks

The publicly available race results from the Cherry Blossom Ten Mile Run can be scraped from the Web and read into *R* for analysis. The currently published results include all years from 1999 to 2012. The task of scraping the Web site and formatting the results in a way that can be analyzed in *R* is a bit challenging because the information reported and the format of this information changes from year to year. Some simple differences in format occur in the size of the table header and the use of footnotes. The tables also include many mistakes, e.g., values that begin in the wrong column, missing headers, and so on. All in all, the acquisition of the data is quite straightforward, but it is an iterative process as we uncover several small errors. We do this statistically, i.e., we examine summary statistics of the data we have read into *R*, find anomalies, such as all the runners in 2003 being under 9, cross check sample observations with the original tables, modify our code to handle these problem cases in a way that is as general as possible, recreate our data, and repeat. This is the story of "messy" data. It is the focus of Section 4.2 and Section 4.3 of this chapter. Additionally, Section 4.7 covers the topic

Figure 4.1: Screen shot of Cherry Blossom Run Website. This page contains links to each year's race results. The year 1999 is the earliest for which they provide data. Men's and women's results are listed separately.

of scraping the Web for the race results, for those who are interested in the entire process of data acquisition.

After the data have been successfully read into *R* and cleaned, we study the relationship between race time and age in Section 4.4. Given the popularity of the race, simple tasks such as visualizing the data present challenges, and we consider how to display tens of thousands of observations in an informative manner.

For any one year of race results, we have a cross-sectional view of the performance-age relationship. That is, we are looking at different groups of people of various ages and their race times; we are not viewing an individual's race performance as he or she gets older. However, we do have race results for 14 years and many runners have participated in multiple races. If we can associate race times with an individual runner, we can examine how performance changes for an individual as he or she ages. The data include the runner's name, age, and hometown so we consider how we might use this information to construct longitudinal views of run times for individuals. This is the subject of Section 4.5

If we study those runners who have competed in multiple years, then we have a longitudinal view of performance. However, we have results for a runner for at most 14 years so we are unable to view performance for an individual over the full range of participant ages from 18 to 89. Can we piece together these longitudinal data to get estimates for performance as a function of age? We look into how to do this in Section 4.6.

4.1.2 Computational Goals of the Case

- Use regular expressions to extract and clean messy data from pre-formatted text tables and to create unique identifiers for matching records that belong to the same individual.
- Employ statistical techniques to identify bad data and to confirm these problems have been corrected.
- Visualize data which have a large number of observations (˜150,000).
- Gain experience with the formula language in plots and model fitting.
- Fit piecewise linear models using least squares and nonparametric curves using local averaging.
- Compare data structures, e.g., data frame and a list of data frames, for holding and working with longitudinal data. This includes the application of 'apply' functions such as tapply(), mapply(), sapply(), and lapply().
- Develop strategies for debugging code with recover() for browsing active function calls after an error.
- Scrape simple Web pages for text content.

4.2 Reading Tables of Race Results into *R*

Our goal in this section is to transform the raw text tables of race results into data that can be analyzed in *R*. These tables have been downloaded from the Web and stored in files, named `1999.txt`, ..., `2012.txt` in a directory called `MenTxt` for men and `WomenTxt` for women. The task of downloading the Web pages and extracting the tables is addressed in Section 4.7. If you want to start this project from the "beginning", then skip ahead to that section and return after you have obtained the text files from the Internet.

Let's examine these text tables to get a sense of their format. After that we should have a few ideas about how we might extract information contained in these tables into variables for statistical analysis. Figure 4.2 and Figure 4.3 provide screenshots of two tables as they appear on the Web. By inspection, we see that a call to read.table() will not properly read the text into a data frame because the information, e.g., place and division, are separated by blanks but blanks also appear in the data values, i.e., blanks also occur where they are not being used as variable separators. For example for the runner's hometown, we see values of `Kenya`, `Tucson AZ`, and `Blowing Rock NC`. The blanks between the different parts of hometown will confuse read.table(). We confirm this when we try to use read.table() to input the data:

```
m2012 = read.table(file="MenTxt/2012.txt", skip = 8)

Error in scan(file, what, nmax, sep, dec, quote,
              skip, nlines, na.strings,   :
  line 1 did not have 12 elements
```

We need a more sophisticated approach. From the figures, it appears that the variables are formatted to occupy particular positions in each line of text. That is, the runner's finishing place occupies the first 5 characters, then comes a blank character, the runner's place in his or her division appears in the next 11 spaces, and so on. While the first two columns of the 2011 and 2012 male results line up, we see that all the columns are not identical across these tables. Given the changes in formats from year to year, we can extract the values from the tables either by programmatically interpreting the format or by using year-dependent fixed-width formats. We will take the first approach here and figure out which column is which by programmatically inspecting the table header. We leave the second approach as an exercise. There you will examine all 28 tables, determine the start and end position of each column of interest, and use read.fwf() to input the data into *R*.

Credit Union Cherry Blossom Ten Mile Run
Washington, DC Sunday, April 1, 2012

Official Male Results (Sorted By Net Time)

Place	Div /Tot	Num	Name	Ag	Hometown	5 Mile	Time	Pace
1	1/347	9	Allan Kiprono	22	Kenya	22:32	45:15	4:32
2	2/347	11	Lani Kiplagat	23	Kenya	22:38	46:28	4:39
3	1/1093	31	John Korir	36	Kenya	23:20	47:33	4:46
4	1/1457	15	Ian Burrell	27	Tucson AZ	23:50	47:34	4:46
5	3/347	19	Jesse Cherry	24	Blowing Rock NC	23:50	47:40	4:46
6	1/1490	37	Ketema Nugusse	31	Ethiopia	23:42	47:50	4:47
7	2/1457	13	Josh Moen	29	Minneapolis MN	24:06	48:38	4:52
8	3/1457	17	Patrick Rizzo	28	Boulder CO	24:24	49:14	4:56
9	4/1457	41	Stephen Hallinan	26	Washington DC	25:01	50:18	5:02
10	2/1490	345	Paolo Natali	31	Washington DC	25:20	50:44	5:05
11	3/1490	346	David McCollam	32	Bridgeport WV	25:33	50:56	5:06
12	4/347	299	Frank Devar	23	Washington DC	25:28	50:57	5:06
13	4/1490	112	Bert Rodriguez	32	Arlington VA	25:31	50:57	5:06
14	1/931	290	Chris Juarez	41	Alexandria VA	25:28	51:10	5:07
15	5/1457	108	Darryl Brown	29	Exton PA	25:28	51:16	5:08
16	6/1457	119	Jay Luna	28	Denver CO	25:22	51:17	5:08
17	7/1457	110	David Burnham	27	Arlington VA	25:27	51:23	5:09
18	8/1457	296	Karl Dusen	29	Rockville MD	25:51	51:27	5:09
19	9/1457	357	Brian Flynn	28	Bridgewater VA	25:34	51:29	5:09
20	10/1457	114	Carlos Renjifo	29	Columbia MD	25:51	51:43	5:11
21	5/1490	358	Dustin Meeker	30	Baltimore MD	25:51	51:53	5:12
22	11/1457	107	Christopher Sloane	28	Rockville MD	25:32	51:57	5:12
23	12/1457	116	Patrick Reaves	27	Durham NC	25:51	52:16	5:14
24	6/1490	111	Jake Klim	31	North Bethesda MD	25:51	52:32	5:16
25	13/1457	298	Will Viviani	29	Alexandria VA	26:30	52:41	5:17
26	14/1457	106	Paul Guevara	25	Alexandria VA	26:01	52:54	5:18
27	7/1490	303	Dickson Mercer	30	Washington DC	26:27	53:04	5:19

Figure 4.2: Screenshot of the 2012 Male Results. This screenshot shows the results, in race order, for men competing in the 2012 Cherry Blossom 10 Mile Run. Notice that both five-mile times and net times are provided. We know that the `"Time"` column is net time because it is so indicated in the header of the table.

Rather than view the Web pages to determine the file format, we can get a better sense of the format if we examine the raw text itself. We use readLines() to read the contents of the file into *R*, where the return value is a character vector with one string per line of text read. We start with reading the 2012 men's file with

```
els = readLines("MenTxt/2012.txt")
```

The first 10 rows of the 2012 Men's table are

```
els[1:10]
```

```
 [1] ""
 [2] "              Credit Union Cherry Blossom Ten Mile Run"
 [3] "              Washington, DC      Sunday, April 1, 2012"
 [4] ""
 [5] "            Official Male Results (Sorted By Net Time)"
 [6] ""
 [7] "Place Div  /Tot   Num    Name      ... Time     Pace  "
```

Credit Union Cherry Blossom Ten Mile Run
Washington, DC   Sunday, April 3, 2011

Official Male Results

Place	Div /Tot	Num	Name	Ag	Hometown	5 Mile	Gun Tim	Net Tim	Pace
1	1/401	3	Lelisa Desisa	21	Ethiopia		45:36	45:36	4:34
2	2/401	13	Allan Kiprono	21	Kenya	23:08	45:41	45:41	4:35
3	1/1471	5	Ridouane Harroufi	29	Morocco	23:10	46:27	46:27	4:39
4	3/401	17	Lani Kiplagat	22	Kenya	23:09	46:30	46:30	4:39
5	2/1471	27	Macdonard Ondara	26	Kenya	21:41	46:52	46:52	4:42
6	3/1471	29	Tesfaye Sendeku	28	Ethiopia	23:15	46:53	46:53	4:42
7	4/1471	21	Stephen Muange	29	Kenya	23:24	47:30	47:30	4:45
8	4/401	23	Simon Cheprot	21	Kenya	23:14	47:32	47:32	4:46
9	5/1471	31	Josphat Boit	27	Kenya	23:24	47:50	47:50	4:47
10	1/1083	25	Girma Tola	35	Ethiopia	23:27	47:56	47:56	4:48
11	5/401	47	Ezkyas Sisay	22	Ethiopia	23:34	47:58	47:58	4:48
12	6/1471	51	Tesfaye Assefa	27	Ethiopia	23:42	48:03	48:03	4:49
13	7/1471	33	Lucas Meyer	27	Ridgefield CT	24:06	48:26	48:26	4:51
14	8/1471	296	David Nightingale	25	Washington DC	24:10	48:39	48:39	4:52
15	9/1471	45	Augustus Maiyo	27	Colorado Springs CA	24:18	49:56	49:56	5:00
16	10/1471	107	Karl Dusen	28	N Bethesda MD	25:13	50:06	50:06	5:01
17	1/1332	105	Bert Rodriguez	31	Arlington VA	25:08	50:25	50:25	5:03
18	6/401	297	Sam Luff	24	Rockville MD	25:22	50:45	50:45	5:05
19	7/401	106	Jerry Greenlaw	23	Alexandria VA	25:19	50:55	50:55	5:06
20	11/1471	112	Brian Flynn	27	Weyers Cave VA	25:24	51:08	51:08	5:07
21	12/1471	49	Birhanu Alemu	28	Ethiopia	25:09	51:10	51:10	5:07
22	2/1083	20510	Michael Wardian	36	Arlington VA	25:20	51:16	51:16	5:08
23	13/1471	304	Joe Wiegner	29	Rockville MD	25:25	51:34	51:34	5:10
24	14/1471	109	Dirk De Heer	29	Silver Spring MD	25:44	51:40	51:40	5:10
25	15/1471	108	David Burnham	26	Arlington VA	25:37	51:49	51:46	5:11
26	1/928	114	Fred Kieser	40	Cleveland OH	25:22	51:48	51:48	5:11
27	16/1471	305	Michael Cassidy	25	Staten Island NY	25:58	52:03	52:03	5:13

Figure 4.3: Screenshot of Men's 2011 Race Results. This screenshot shows the results, in race order, for men competing in the 2011 Cherry Blossom road race. Notice that in 2011 three times are recorded–the time to complete the first five miles and the gun and net times for the full run. In contrast, the results from 2012, shown in [?], do not provide gun time.

```
 [8] "===== =========== ====== =========... ======= ===== "
 [9] "     1      1/347         9 Allan Kip... 45:15   4:32 "
[10] "     2      2/347        11 Lani Kipl... 46:28   4:39 "
```

We also read in and display the first ten rows of the 2011 male results so we have a comparison table. We find the following:

```
els2011 = readLines("MenTxt/2011.txt")
els2011[1:10]
```

```
 [1] ""
 [2] "                 Credit Union Cherry Blossom Ten Mile Run"
 [3] "                 Washington, DC     Sunday, April 3, 2011"
 [4] ""
 [5] "                            Official Male Results"
 [6] ""
 [7] "Place Div  /Tot   Num    Name   ... Gun Tim Net Tim Pace  "
 [8] "===== =========== ====== ======... ======= ======= ===== "
 [9] "    1      1/401        3 Lelisa... 45:36   45:36  4:34 "
[10] "    2      2/401       13 Allan ... 45:41   45:41  4:35 "
```

What do we see by this simple inspection?

- Both of the tables have a header.
- The last line of the header is a row of =s.
- There are blanks inserted in the row of =s that mark the start and end of a column of information, e.g., Place, Name.
- The row above the =s has column names.
- There are two times reported in 2011 (called `Gun Tim` and `Net Tim`) and only one time reported in 2012 (`Time`). The header tells us that this time is net time.

If we examine a few more years of race results, then we find other differences between how the data are organized. Some years have column names that are all capitalized, do not include the time at 5 miles, contain a rightmost column that holds some sort of annotation, have headers with only three lines, etc.

Let's use the 2012 men's results as our test case for developing the code to read in all the files. However, we will try to write the code in a general way so that it can potentially be used for all 28 files. Our first step is to find the row with the equal signs. The rows below it contain the data, the row above it holds the column headers, and the row itself supplies the spacings for the columns. We saw earlier that the equal signs are in the eighth row of the 2012 table. Since the organization of the tables differs a bit from year to year, we use a programmatic search for the equal signs. We use grep() to search through the character strings in `els` for one that begins with three equal signs as follows:

```
eqIndex = grep("^===", els)
eqIndex
```

```
[1] 8
```

Note that an alternative to regular expressions and the grep() function is to use substr() to extract the first three characters from each row and compare them to the string `"==="`. That is,

```
first3 = substr(els, 1, 3)
which(first3 == "===")
```

```
[1] 8
```

The choice of three =s is somewhat arbitrary. We could have used just one as the equal sign does not appear elsewhere in the document.

Now that we have located this key row in the table, we extract it and the row above it and discard the header with

```
spacers = els[eqIndex]
colNames = els[eqIndex - 1]
body = els[ -(1:eqIndex) ]
```

Our next task is to extract the various pieces of information from each string in `body`. How might we extract the runner's age? From inspection, a runner's age appears in the column labeled `Ag` or `AG` so we first convert the column names to lower case so we need not search separately for `Ag` and `AG`. We use tolower() to do this with

```
colNames = tolower(colNames)
```

We can search through `colNames` for this two letter sequence as follows:

```
ageStart = regexpr("ag", colNames)
ageStart
```

```
[1] 49
attr(,"match.length")
[1] 2
attr(,"useBytes")
[1] TRUE
```

The return value from regexpr() tells us a match was found in position 49 of the character string and the length of the match is 2 characters. If no match is found, then regexpr() returns -1. Now we have the information about the location of runner's age: it begins in position 49 and ends at the 50th position in each row of the table. We use this information to extract each runner's age using substr() as follows:

```
age = substr(body, start = ageStart, stop = ageStart + 1)
head(age)

[1] "22" "23" "36" "27" "24" "31"

summary(as.numeric(age))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   9.00   29.00   35.00   37.75   45.00   89.00       1
```

It appears that we have located the runner's age correctly. The youngest male runner in 2012 was 9 and the oldest 89, and there was one runner who did not have an age reported.

We can extract all of our variables in this manner, but the width of a column might change from one year to the next so we generalize our code to search the row of equal signs for the first blank space after the start of `"ag"` and use that position to determine the end of the column. That is, we find the locations of all of the blanks in the line of =s with

```
spaceLocs = gregexpr(" ", spacers)[[1]]
spaceLocs

[1]  6 18 25 48 51 72 80 88 94
attr(,"match.length")
[1] 1 1 1 1 1 1 1 1 1
attr(,"useBytes")
[1] TRUE
```

Here the `g` in gregexpr() stands for "global" which means that the function searches for multiple matches in the string, not just the first match. Blank spaces are found at the 6th, 18th, 25th, 48th, 51st, etc. positions, and we can pick out the ending position for age with

```
ageStop = spaceLocs[spaceLocs > ageStart][1] - 1
ageStop

[1] 50
```

We use `ageStop`, rather than `ageStart + 1` as our ending position in the call to substr(), i.e.,

```
age = substr(body, start = ageStart, stop = ageStop)
summary(as.numeric(age))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   9.00   29.00   35.00   37.75   45.00   89.00       1
```

Our more general extraction matches the earlier one.

In general, we want to write our code so that it does not depend on a variable starting or ending in a particular column. We can do this by searching for pieces of the variable name that are long enough to uniquely determine it but not so long as to require a lot of special cases if the variable name changes a little from year to year. This search gives us the position of the start of the column with the relevant information, and we find the end position using the method above that searches for the first blank after the starting position.

At this point, we need to decide whether we want to keep the union of all variables across the 14 years, or a subset such as name, age, home town, and one of the time measurements. For now, we extract name, age, hometown, and all three times, i.e., gun time, net time, and time, and ignore the rest. When one of these times does not appear in a particular table, then we set the values to `NA`.

We take our beginning column names to be

```
varTips = c("name", "home", "ag", "gun", "net", "time" )
```

We generalize into a function the code that we used above to extract age, and we apply this function to the column names in `varTips`. We need to handle the special case when the variable name is not found. This occurs for example in a file that contains only time and not gun and net times. We also need to handle the special case where we have reached the end of the character string so the blank after the rightmost column may not be found. In this case, we can simply substitute the position of the last character in the string. We do all of this with

```
runCharData = sapply(varTips, function(x) {
  startCol = regexpr(x, colNames)[[1]]
  if (startCol == -1) return(rep(NA, length(body)))
  stopCol = spaceLocs[spaceLocs > startCol][1] - 1
  if (is.na(stopCol) | stopCol <= 0 ) stopCol = nchar(spacers)
  substr(body, start = startCol, stop = stopCol)
})
```

Notice we did not assign this function a name. It is an orphan that disappears after the call to sapply(). Let's examine the return value. First we check the type of the return value with

```
class(runCharData)
```

```
[1] "matrix"
```

The results form a matrix of character strings. (We have not yet converted any values such as age to numeric.) We see that the first few rows of the matrix are

```
head(runCharData)
```

```
     name                 home                  ag   gun net time
[1,] "Allan Kiprono     " "Kenya             " "22" NA  NA  " 45:15"
[2,] "Lani Kiplagat     " "Kenya             " "23" NA  NA  " 46:28"
[3,] "John Korir        " "Kenya             " "36" NA  NA  " 47:33"
[4,] "Ian Burrell       " "Tucson AZ         " "27" NA  NA  " 47:34"
[5,] "Jesse Cherry      " "Blowing Rock NC   " "24" NA  NA  " 47:40"
[6,] "Ketema Nugusse    " "Ethiopia          " "31" NA  NA  " 47:50"
```

The 2012 table had only time and so the gun and net values are `NA`. We also check the last few lines with

```
tail(runCharData)[ , 1:3]
```

```
        name                         home                   ag
[7188,] "Dana Brown               " "Randallstown MD  " "41"
[7189,] "Jurek Grabowski          " "Fairfax VA       " "39"
[7190,] "Larry Hume               " "Arlington VA     " "56"
[7191,] "Sean-Patrick Alexander" "Alexandria VA    " "35"
[7192,] "Joseph White             " "Forestville MD   " "  "
[7193,] "Lee Jordan               " "Herndon VA       " "48"
```

Here we see the one runner who did not report an age. It appears that we have successfully captured the information from the table in `MenTxt/2012.txt`.

We can wrap this process up into a function to apply to each year's data. The function might look as

```
extractVariables = function(els){
  # Take a character vector of rows in a table
  # and extract name, hometown, age, and times

  # Find header, =s, column names, and body
  eqIndex = grep("^===", els)
  spacers = els[eqIndex]
  spaceLocs = gregexpr(" ", spacers)[[1]]

  colNames = els[ eqIndex - 1 ]
  colNames = tolower(colNames)

  body = els[ -(1 : eqIndex) ]

  varTips = c("name", "home", "ag", "gun", "net", "time" )

  runCharData = sapply(varTips, function(x) {
    startCol = regexpr(x, colNames)[[1]]
    if (startCol == -1) return(rep(NA, length(body)))
    stopCol = spaceLocs[spaceLocs > startCol][1] - 1
    if (is.na(stopCol) | stopCol <= 0 ) stopCol = nchar(spacers)
    substr(body, start = startCol, stop = stopCol)
  })

  return(runCharData)
}
```

We are ready to create the character matrices for each table, but this function expects `els` to be a character vector. We first must read the lines of the tables into *R*. We do this with:

```
mfilenames = paste("MenTxt/", 1999:2012, ".txt", sep = "")
menTables = lapply(mfilenames, readLines)
names(menTables) = 1999:2012
```

Similarly, we read the women's results into `WomenTables`. These two objects, `MenTables` and `WomenTables`, are lists where each list contains 14 character vectors, one for each year, and each character vector contains one string for each row in the corresponding table.

We can now apply the extractVariables() function to `menTables` and `womenTables` to obtain a list of character matrices. We do this for the men's list with

```
menResMat = lapply(menTables, extractVariables)
length(menResMat)

[1] 14

sapply(menResMat, dim)

     1999 2000 2001 2002 2003 2004 2005 2006 2007
[1,] 3190 3017 3622 3724 3948 4156 4327 5237 5276
[2,]    6    6    6    6    6    6    6    6    6
     2008 2009 2010 2011 2012
[1,] 5905 6651 6911 7011 7193
[2,]    6    6    6    6    6
```

We see that we get reasonable values for the dimensions for our matrices. Our next task is then to convert these character matrices into a format that we can readily analyze. As we do this, we use statistics to check the results and find that additional data cleaning is necessary. This is the topic of the next section.

4.3 Data Cleaning and Reformatting variables

In this section, we consider how to convert the character matrix, `menResMat`, into an appropriate format for analysis. Currently, the data are in a character matrix, which is not conducive to, e.g., finding the median age of the runners. However, we can easily reformat age into numeric values with the as.numeric() function. Do we want to turn the entire matrix into a numeric matrix? Not really. It doesn't make sense to try to convert the runner's name into a numeric value. For this reason, we want to create a data frame because it allows our variables to be different types. We have six variables: the runner's name, home town, age, and three versions of time. As just mentioned, we will want to convert age to a numeric and leave name as character. What about the other variables? We probably want to also keep hometown as character.

Time is stored as a string in the format: `"hh:mm:ss"`. We want time in a numeric format so it can be more easily summarized and modeled. One possibility would be to convert it to minutes, i.e., `hh * 60 + mm + ss/60`. However, to carry out this computation, we must split the time field up into its constituent pieces and convert each to numeric values. The strsplit() function can be very helpful for splitting strings at, e.g., colons. We also need to reconcile the three different recorded times (gun, net, and plain time). Net time is considered more accurate than gun time so we can simply use net time when available and otherwise use gun time or time, whichever is reported. Of course, we can keep all three versions of time around and let the analyst decide which to use, but we keep things simple for now and just report one time for each runner.

Before we begin converting our character strings into numeric values, we also consider whether there are any new variables we might want to create. If we are to combine all the data from the 14 years of records into one data frame, then we should keep track of the year. Likewise, if we are to combine the men's and women's results then we will want a new variable that indicates the sex of the runner. These can be simply made using rep().

We begin with the task of creating the numeric variable age with as.numeric(), e.g., for the 2012 males,

```
age = as.numeric(menResMat[['2012']][ , 3])
```

Note that we subsetted the list to work with the 2012 matrix and then subsetted this matrix to work with the third column, which we know contains age. We check a few `age` values with

```
tail(age)
```

```
[1] 41 39 56 35 NA 48
```

These values look reasonable, but let's check more thoroughly that our data extraction works as expected by summarizing each year's ages with

```
sapply(menResMat,
       function(x) summary(as.numeric(x[ , 3])))
```

```
$`1999`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  11.00   32.00   40.00   40.34   48.00   80.00       1

$`2000`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  11.00   32.00   40.00   40.41   48.00   79.00       1

$`2001`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00   32.00   39.00   40.28   48.00   80.00      61

$`2002`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   1.00   32.00   39.00   40.29   48.00   79.00       3

$`2003`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  1.000   3.000   3.000   3.585   4.000   8.000       4

$`2004`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   31.00   38.00   39.31   47.00   81.00

$`2005`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  12.00   31.00   38.00   39.56   47.00   82.00      13

$`2006`
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   2.00    3.00   34.00   28.53   46.00   82.00       2

$`2007`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   9.00   30.00   37.00   38.55   46.00   80.00       5

$`2008`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  12.00   29.00   36.00   37.78   45.00   84.00

$`2009`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  10.00   29.00   35.00   37.37   44.00   85.00       2

$`2010`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  11.00   29.00   35.00   36.98   44.00   86.00       6

$`2011`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   8.00   29.00   36.00   37.53   44.00   83.00

$`2012`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   9.00   29.00   35.00   37.75   45.00   89.00       1

Warning messages:
1: In summary(as.numeric(x[, 3])) : NAs introduced by coercion
2: In summary(as.numeric(x[, 3])) : NAs introduced by coercion
3: In summary(as.numeric(x[, 3])) : NAs introduced by coercion
```

A careful inspection of these values shows that all of the runners in 2003 were under 9 and one in four runners in 2006 were under 4! Clearly something has gone wrong.

Let's examine the original text for 2003 and 2006:

```
head(menTables[['2003']])

[1] ""
[2] "Place Div /Tot  Num    Name                         Ag Homet..."
[3] "===== ========= ===== ===================...== == ======..."
[4] "    1    1/1999     6  John Korir                   27  KEN ..."
[5] "    2    2/1999     1  Reuben Cheruiyot             28  KEN ..."
[6] "    3    3/1999     8  Gilbert Okari                24  KEN ..."

menTables[['2006']][2200:2205]

[1] " 2192 1263/2892   1475 Matt Curtis             39 Vienna ..."
[2] " 2193   94/279    1437 Joe McCloskey          59 Columbia..."
```

```
[3] " 2194 257/590    7062 Donald Hofmann         48 Princeto..."
[4] " 2195 1264/2892   7049 Claudio Petruzziello   23 Princet..."
[5] " 2196 339/746    3319 Robert Morrison        40 South Bo..."
[6] " 2197 1265/2892   9345 Larry Cooper           32 Arlingt..."
```

We see that in 2003, the age column is shifted to the right one space so we are picking up only the digit in the tens place, and in 2006 some but not all of the rows have values that are off by one character. We can easily solve both of these problems by including the value in the "blank" space between columns. We can do this by changing the index for the end of each variable when we perform the extraction, i.e.,

```
stopCol = spaceLocs[spaceLocs > startCol][1]
```

When we use this revised calculation of `stopCol`, we pick up the blank character after each field. This shouldn't matter when we convert our text data to numeric and if we don't want trailing blanks in our character-valued variables, we can easily remove them with regular expressions.

In the process of confirming our conversion of age from character to numeric, we uncovered problems with our extraction process. We need to modify extractVariables() from Section 4.2 to address the problem. This process is iterative as we continue to check that our data make sense. When we uncover nonsensical results, we investigate them further, which possibly leads to retracing our steps to clean up messy data.

After we modify this one line of code to locate the end of each substring in our extractVariables() function and reapply this updated version of the function to the tables of race results, we check again the summary statistics. We find

```
sapply(menResMat,
       function(x) summary(as.numeric(x[ , 3])))


...

$`2001`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00   32.00   39.00   40.28   48.00   80.00      61

$`2002`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   1.00   32.00   39.00   40.29   48.00   79.00       3

$`2003`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00   32.00   39.00   40.35   48.00   80.00       2
...

$`2006`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  13.00   30.00   38.00   38.91   46.00   82.00       2
...
```

This change to extractVariables() clears up the problem: we find the lower quartile for all years range between 29 and 32. However, we now notice that there are 61 `NA` values in 2001. Did we introduce this problem with our small change? When we look back at the summary statistics for age *before* our modification to extractVariables(), we see that there were the same number of NAs in 2001, and we were given several warning messages that NAs introduced by coercion.

We have a large number of NAs in 2001 in comparison to other years. We need to investigate. To make our work simpler, let's assign the 2001 ages in MenResMat to a vector called age2001 and convert them to numeric values. We do this with

```
age2001 = menResMat[['2001']][ , 3]
age2001 = as.numeric(age2001)

Warning message:
NAs introduced by coercion
```

Notice the warning message about the NAs. Let's examine the original rows in the table that correspond to NA in age2001. Recall that we dropped the header of the table before extracting the variables so we will need to add an offset to the location of the NAs in age2001 in order to pick out the correct rows in the original table. We do this with

```
menTables[['2001']][ which(is.na(age2001)) + 5 ]

 [1] "               "
 [2] "                                 "
 [3] "                        "
 [4] ""
 [5] "                           "
 [6] ""
...
[60] ""
[61] "# Under USATF OPEN guideline..."
```

With one exception, all of these rows are blank/empty. The one exception is the row that corresponds to the footnote that defines the meaning of the #. Where in the table are these rows located?

```
which(is.na(age2001)) + 5

 [1] 1756 1757 1758 1759 1760 1761 1762 1763 1814 ...
[22] 1877 1878 1879 1930 1931 1932 1933 1934 1935 ...
[43] 2898 2899 2900 2901 2902 2903 2904 2955 2956 ...
```

These blank lines are scattered throughout the table. We can modify the extraction by checking for blank rows and removing them from the table. The regular expression,

```
blanks = grep("^[[:blank:]]*$", body)
```

locates all rows that are entirely blank. The first argument to grep uses several meta characters to specify the pattern to search for. The ^ is an anchor for the start of the string, the $ anchors to the end of the string, the [[:blank:]] denotes the equivalence class of any blank-type character, and * indicates that the blank character can appear 0 or more times. All together the pattern ^[[:blank:]]*$ matches a string that contains any number of blanks from start to end.

A simpler expression locates the footnote rows, i.e., rows that begin with # or *. We leave as an exercise the task of modifying extractVariables() to remove these unwanted rows. After adding this code to carry out the additional cleaning of the tables, the 61 NAs in 2001 are gone as well as many but not all of the other NAs in other the years.

Continued inspection of the summary statistics for age uncovers another problem–the minimum value for age in 2001, 2002 and 2003 remains small, e.g., 0 or 1. That's clearly not possible! Let's find which runners have an age of 0 or 1 and look at their records in the original table. For 2001, we have

```
which(age2001 < 2)

[1] 1377 3063 3112

menTables[['2001']][ which(age2001 < 2) + 5 ]

[1] " 1377  5629 Steve PINKOS           0 Wash..."
[2] " 3003  5033 Jeff LAKE              0 Clar..."
[3] " 3052  5637 Greg RHODE             0 Wash..."
```

Apparently there are runners with an age entered as 0! Since these are the actual values in the table, we leave the decision as to what to do with these runners for later when we analyze the data. At this point, it appears we have successfully taken care of the creation of the age variable.

Next, we turn to the creation of the time variable. As mentioned at the beginning of this section, the time appears as "hh:mm:ss" and we wish to convert it to minutes. However, to carry out this computation, we must split the time field up into its constituent pieces. Also, some runners completed the race in under one hour so their times appear in a slightly different format, i.e., "mm:ss", and we will need to be able to handle both formats in our processing. For simplicity, we again start with converting the time variable for one year, say 2012. We create a vector to develop our code as follows:

```
charTime = menResMat[['2012']][, 6]
head(charTime, 5)

[1] "  45:15 " "  46:28 " "  47:33 " "  47:34 " "  47:40 "

tail(charTime, 5)

[1] "2:27:11 " "2:27:20 " "2:27:30 " "2:28:58 " "2:30:59 "
```

We split each character string up into its parts using strsplit() with

```
timePieces = strsplit(charTime, ":")
```

This call to strsplit() breaks a string up into pieces, where the breaks occur at the locations of the : in the string. The :s are discarded in the process, and we are returned a list of character vectors, one vector for each input string where the elements of the vector contain the pieces of the string before, between, and after the :s. We confirm that the splitting worked properly by examining the first and last times, i.e.,

```
timePieces[[1]]

[1] "  45" "15 "

tail(timePieces, 1)
```

```
[[1]]
[1] "2"   "30"  "59 "
```

We convert the pieces to numeric values and combine them into one value that reports time in minutes with

```
timePieces = sapply(timePieces, as.numeric)
c3 = c(60, 1, 1/60)
c2 = c3[-1]
timeMin = sapply(timePieces,
                 function(x) {
                   if (length(x) == 2) sum(c2 * x)
                   else sum(c3 * x)
                 })
```

We check our conversion with

```
summary(timeMin)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  45.25   77.57   87.47   88.43   97.78  151.00
```

It appears that our time conversion works. We saw earlier that the fastest runner completed the 2012 race in 45 minutes and 15 seconds, which is 45.25 minutes, and the slowest completed it in 2 hours 30 minutes and 59 seconds, which is nearly 151 minutes.

Let's wrap up these conversions into a function to apply to the character matrices in `MenResMat`. We call this function cleanUp(). In addition to the conversion of character strings to numeric, we also create two new variables, `year` and `sex`. To do this, we must have input arguments to tell us which year we are cleaning and whether the results are for men or women. Lastly, we also choose the time from among the three with a preference for net time. The function appears as

```
cleanUp = function(Res, year, sex) {

 # Determine which time to use
  if ( !is.na(Res[1, 5]) ) useTime = Res[ , 5]
  else if ( !is.na(Res[1, 4]) ) useTime = Res[ , 4]
  else useTime = Res[ , 6]

 # Convert time to minutes
  timePieces = strsplit(useTime, ":")
  timePieces = sapply(timePieces, as.numeric)
  c3 = c(60, 1, 1/60)
  c2 = c3[ -1 ]
  timeMin = sapply(timePieces,
                   function(x) {
                     if (length(x) == 2) sum(c2 * x)
                     else sum(c3 * x)
                   })

  Results = data.frame(year = rep(year, nrow(Res)),
                       sex = rep(sex, nrow(Res)),
```

```
                         name = Res[ , 1], home = Res[ , 2],
                         age = as.numeric(Res[, 3]),
                         time = timeMin,
                         stringsAsFactors = FALSE)
  invisible(Results)
}
```

We apply our new function, cleanUp(), to all of the male results and use simple summary statistics to check the values for time. Below is the output:

```
menDF = mapply(cleanUp, menResMat, year = 1999:2012,
               sex = rep("M", 14), SIMPLIFY = FALSE)

There were 50 or more warnings
(use warnings() to see the first 50)

sapply(menDF, function(x) summary(x$time))

$`1999`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.98   74.80   84.28   84.33   93.08  170.80

$`2000`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.10   74.77   83.22   83.61   92.15  155.20

$`2001`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.50   75.32   84.15   84.50   93.15  164.60

$`2002`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  47.20   76.09   84.72   85.68   94.70  150.20

$`2003`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.92   76.00   84.78   85.84   94.73  146.30

$`2004`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  48.20   76.57   85.88   86.44   95.22  166.70

$`2005`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.90   77.42   87.27   88.02   97.20  175.60

$`2006`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  47.40   78.09   87.25   88.17   97.29  167.10
```

```
$`2007`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  49.83   77.65   86.23   87.50   96.58  165.60      83

$`2008`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.23   77.68   87.48   88.26   97.70  139.00

$`2009`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  60.02   78.90   87.98   89.27   98.05  154.40     164

$`2010`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  50.87   78.72   88.00   89.38   98.85  154.10      68

$`2011`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  45.60   78.65   88.47   89.63   99.32  148.70

$`2012`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  45.25   77.57   87.47   88.43   97.78  151.00
```

Do you see any problems or unusual values that should be looked into with greater care?

Two issues appear in these summary statistics. There are a large number of `NA`s in 2007, 2009, and 2010, and the minimum time in 2001 is 1.5 minutes! If we examine the records that have an `NA` in time, we find that these are caused by runners who completed half the race but have no final times and by runners who have a footnote after their time, e.g.,

```
"     1      1/54        13 Tadesse Tola           19
         Ethiopia              46:01#  4:37     28:47 "
"  5273   309/309    16370 Stephen Peterson        57
         Washington DC              #         1:36:29 "
```

We can easily modify cleanUp() to eliminate the `#` and `*` from the times and drop records of runners who do not complete the race. These revisions are

```
 # Remove # and * and blanks from time
  useTime = gsub("#", "", useTime)
  useTime = gsub("\\*", "", useTime)
  useTime = gsub("[[:blank:]]", "", useTime)

 # Drop rows with no time
  Res = Res[ useTime != "", ]
  useTime = useTime[ useTime != "" ]
```

After we apply this revised function to `menResMat` to create our data frame, all missing values in time are gone.

```
           1999   2000   2001   2002   2003   2004   2005   2006
Min.      46.98  46.10   1.50  47.20  46.92  48.20  46.90  47.40
1st Qu.   74.80  74.77  75.32  76.09  76.00  76.57  77.42  78.09
Median    84.28  83.22  84.15  84.72  84.78  85.88  87.27  87.25
Mean      84.33  83.61  84.50  85.68  85.84  86.44  88.02  88.17
3rd Qu.   93.08  92.15  93.15  94.70  94.73  95.22  97.20  97.29
Max.     170.80 155.20 164.60 150.20 146.30 166.70 175.60 167.10

           2007   2008   2009   2010   2011   2012
Min.      46.02  46.23  45.93  45.72  45.60  45.25
1st Qu.   77.42  77.68  78.15  78.47  78.65  77.57
Median    86.08  87.48  87.55  87.85  88.47  87.47
Mean      87.23  88.26  88.56  89.12  89.63  88.43
3rd Qu.   96.42  97.70  97.80  98.75  99.32  97.78
Max.     165.60 139.00 154.40 154.10 148.70 151.00
```

The minimum time of 1.5 comes from a different problem so it remains in the data frame. We leave it as an exercise to ascertain the problem and how it might be fixed. It only occurs for one runner so we leave that observation in our data frame and drop it from our analysis as needed.

Finally, there is one more mess that needs to be taken care of, but for brevity's sake, we leave that problem to the exercises as well. It occurs in 2006. Close inspection of those tables and examination of variables other than age and time reveals the problem.

At last, we combine the race results for all years and men and women into one data frame using the do.call() function to call rbind() with the list of data frames as input. That is,

```
menResDF = do.call(rbind, menDF)
womenResDF = do.call(rbind, womenDF)
cbRes = rbind(menResDF, womenResDF)
save(cbRes, file = "cbDF.rda")
```

The do.call() function is very handy when we want to call a function that takes an arbitrary number of inputs, such as rbind(). The rbind() function's first argument is ..., i.e.,

```
args(rbind)
function (..., deparse.level = 1)
```

The ... argument allows the caller to provide an arbitrary number of arguments that, in the case of rbind(), will be bound together into one object. We could call rbind() as follows:

```
rbind(menDF[[1]], menDF[[2]], menDF[[3]], menDF[[4]],
      menDF[[5]], menDF[[6]], menDF[[7]], menDF[[8]],
      menDF[[9]], menDF[[10]], menDF[[11]], menDF[[12]],
      menDF[[13]], menDF[[14]])
```

That's a lot of typing and it requires us to know that there are 14 data frames in `menDF`. The do.call() function allows us to supply the inputs to rbind() as a list and it puts together the function call for us. This can be very convenient. We check the dimension of our amalgamated data frame with

```
dim(cbRes>
```

```
[1] 146044      6
```

Over these 14 years, nearly 150,000 runners completed the Cherry Blossom race. In the next section we take a closer look at the race results.

4.4 Exploring the Average Performance in a Race

Now that we have completed the extraction of our data from the tables published on the Cherry Blossom Web site, we can begin to study the relationship between age and running performance. Typically, we would first examine our data graphically in a scatter plot with performance on the y-axis and age on the x-axis. We can make such a scatter plot for the male runners with the following call to plot()

```
plot(time ~ age, data = cbRes, subset = sex == "M",
     xlab = "Age (years)", ylab = "Time (minutes)",
     main = menTitle)
```

(Here the variable `menTitle` is `"Cherry Blossom 10-Mile Run\n Male Runners, 1999-2012"`.)

The first argument in this call to plot() is an *R* formula. The formula language is very powerful as it can be used to succinctly express a relationship and a variety of *R* functions can interpret a formula and carry out an analysis appropriate for the data. In our case, the formula is very simple, `time ~ age`, and it indicates that we are interested in how `time` depends on or varies with `age`. The plot() function builds the visual model based on the representation of the data. Since `time` and `age` are both numeric variables, plot() makes a scatter plot with `time` on the y-axis and `age` on the x-axis. Later in this section, we will see other formulas with more variables and with categorical variables, and we will see the formula used with other functions such as lm() and loess().

The resulting plot appears in Figure 4.4. Most of the points appear as a black blob in the scatter plot because so many points have been plotted on top of each other. The shape of the distribution is obscured because we cannot see which regions of the (age, time) space are densely populated. Notice also the vertical stripes in the plot. These are the result of the runners ages being reported to the nearest year, which results in more over plotting. In the next section, we consider a few alterations to this default scatter plot that address the problem.

4.4.1 Making Plots with Many Observations

There are several modifications we can make to the plot in Figure 4.4 to ameliorate the effect of over plotting. We can reduce the size of the plotting symbol, use transparent colors for the plotting symbol, and add a small amount of random noise to the age variable. Alternatively, we can create a plot that reveals a smoothed version of the density of the points. We can also make a series of box plots instead of a scatter plot. We demonstrate each of these approaches in this section.

We first modify the call to plot() to change the plotting symbol from a circle to a disk, and we shrink the size of the disk as well. We also use a transparent blue as the color for the disk. If we use a transparent color for the plotting symbol, then when two symbols overlap, their color appears darker. This way, regions with a higher density of observations appear darker than low density areas.

Colors can be specified in many ways in *R*. The RGBA specification provides a triple of red-blue-green components that combine to make a color. The fourth component in the RGBA specification provides the amount of transparency. For our color, we choose one from Cindy Brewer's color palettes that are available in the `RColorBrewer` package. We load the package and display the objects in the package with

```
library(RColorBrewer)
objects("package:RColorBrewer")
```

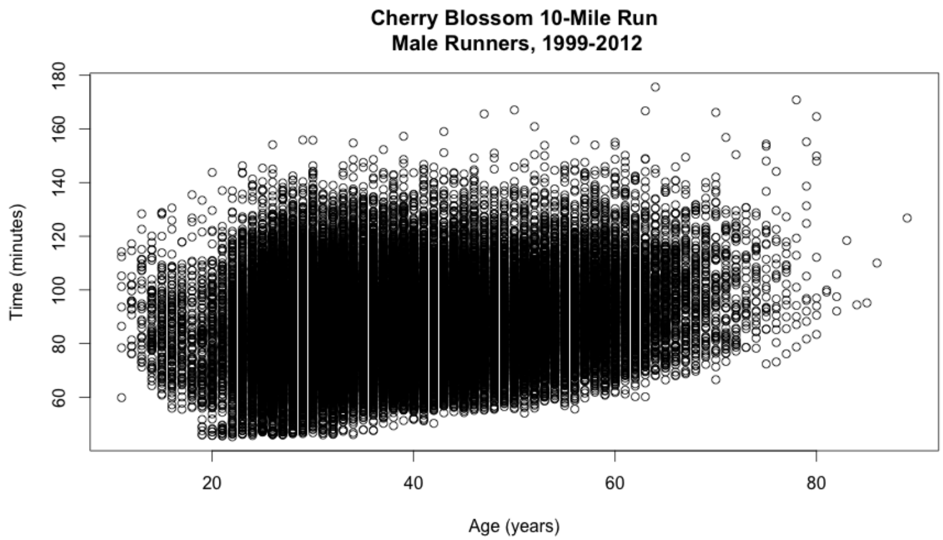


Figure 4.4: Default Scatter Plot for Performance vs. Age for Male Runners. This plot demonstrates that a simple scatter plot of time by age for the 70,000 male runners leads to such severe over plotting that the shape of the data is not discernible. Notice also that the vertical bands are a result of the runners ages being reported to the nearest year.

```
[1] "brewer.pal"          "brewer.pal.info"
[3] "display.brewer.all" "display.brewer.pal"
```

These objects are the four functions available in the package. After reading the help information on display.brewer.all(), we see that it is a good starting place because it displays all of the palettes available in the package. We call it with

```
display.brewer.all()
```

and choose the blue in the `Set3` palette as follows

```
S3Blue = brewer.pal(12, "Set3")[5]
S3Blue
```

```
[1] "#80B1D3"
```

We see that the color is stored in hexadecimal format, where red is 80, blue is B1 and green is D3. This color does not include an alpha transparency, which means that it is an opaque color. However, we can create a transparent version of this color. To do this, we first split the color up into its three components and convert these to decimal values with

```
S3Blue3 = col2rgb(S3Blue)
S3Blue3
```

```
      [,1]
red    128
green  177
blue   211
```

Next we call the rgb() function to create the new color from the red-blue-green combination in `S3Blue3` and an additional transparent value that we provide, i.e.,

```
S3BlueA = rgb(red=S3Blue3[1], green = S3Blue3[2],
              blue = S3Blue3[3], alpha = 20,
              maxColorValue = 255)
S3BlueA
```

```
[1] "#80B1D314"
```

We use this color for our plotting symbol.

Additionally, we change the ages of the runners by a small random amount between -0.5 and 0.5. This operation is called jittering, and we can jitter age with `jitter(age, amount = 0.5)`.

Our resulting plot appears in Figure 4.5. This plot is very much improved from the initial one in Figure 4.4. We can see where the bulk of the runners are, including what appears to be a slight upward curvature in time as age increases. We leave the creation of this plot as an exercise.

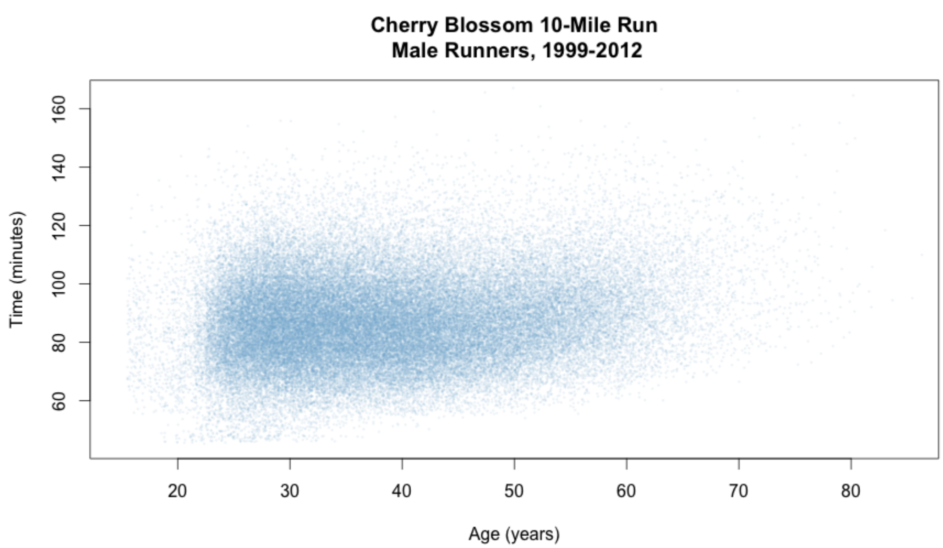


Figure 4.5: Revised Scatter Plot of Male Runners. This plot revises the simple scatter plot of Figure 4.4 by changing the plotting symbol from a circle to a disk, reducing the size of the plotting symbol, using a transparent color for the disk, and adding a small amount of random noise to age. Now we see the shape of the high density region containing most of the runners and the slight upward trend of time with increasing age.

The smoothScatter() function provides a more formal approach to using transparency for visualizing the density of runner's time-age distribution. This function produces a smooth density representation of the scatterplot using color, much like in Figure 4.5, but with a more statistical approach to building regions varying color intensity. With smoothScatter(), the color at an (x, y) location is determined by the density of points in a small region around that point. This averaging process yields a smoother plot with dark shades corresponding to high density regions.

To call smoothScatter(), we first subset the `cbResults` data frame because, unlike plot(),this function does not support the *subset* parameter. We extract all male runners, drop unusually small times, and restrict our attention to males over 15 because we think that the relationship between time and age might be quite different for young teenagers. We create our new data frame as follows:

```
menRes = cbRes[cbRes$sex == "M" & cbRes$time > 20 &
               cbRes$age > 15 & !is.na(cbRes$age) , ]
```

Now we have both `cbRes` and `menRes` in our workspace. With large data sets, this duplication of data may become a problem. In this case we might remove `cbRes` from the workspace while we analyze the male data, or we might create a logical vector to subset `cbRes` in all of our function calls, e.g.,

```
men15plusI =  cbRes$sex == "M" & cbRes$time > 20 &
              cbRes$age > 15 & !is.na(cbRes$age)

plot(time ~ jitter(age, amount = 0.5),
     data = cbRes[ men15plusI, ], col = S3BlueA, ... )
```

We work with `menRes` for simplicity.

We call smoothScatter() with `menRes` as follows:

```
smoothScatter(y = menRes$time, x = menRes$age,
              ylim = c(40, 165), xlim = c(15, 85),
              xlab = "Age (years)", ylab = "Time (minutes)",
              main = menTitle)
```

The resulting plot in Figure 4.6 shows a very similar shape to our plot in Figure 4.5. It has the addition of small black dots to indicate individual points that are far from the main point cloud.

A very different approach to these scatter plots is to plot summary statistics of time for subgroups of runners with roughly the same age. Here, we group the runners into 10-year age intervals and plot the summaries for each subgroup in the form of a boxplot (see Figure 4.7. With these side-by-side boxplots, the size of the data does not obscure the main features, e.g., the quartiles and tails for an age group. To make these boxplots, we categorize age using the cut() function. We do this with

```
ageCat = cut(menRes$age, breaks = c(seq(15, 75, 10), 90))
table(ageCat)

ageCat
(15,25] (25,35] (35,45] (45,55] (55,65] (65,75] (75,90]
   5804   25434   20535   12212    5001     752      69
```

This new variable, `ageCat`, is a factor that categorizes age into 10-year intervals with the exception of all of those over 75 being lumped together into one interval. We use the formula `time ~ ageCat` in the call to plot() as follows:

```
plot(menRes$time ~ ageCat,
     xlab = "Age (years)", ylab = "Time (minutes)",
     main = menTitle)
```

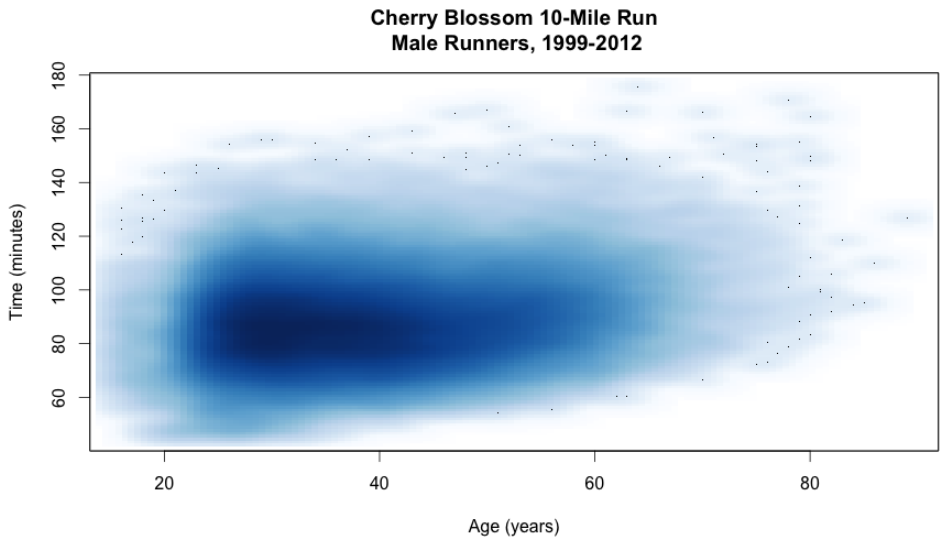


Figure 4.6: Smoothed Scatter Plot of Male Runners Race Times vs. Age. This plot offers an alternative to the scatter plot of Figure 4.5 that uses jittering and transparent color to ameliorate the over plotting. Here there is no need to jitter age because the smoothing action essentially does that for us by spreading an individual runner's (age, time) pair over a small region. The shape of the high density region has a very similar shape to the earlier plot.

We see in Figure 4.7 that the plot() function has created a series of boxplots rather than a scatter plot. The difference between this function call and the earlier one that produced Figure 4.4 is in the formula provided. Since `ageCat` is a factor, the default plot for the formula `time ~ ageCat` is a series of side-by-side boxplots with one boxplot of times per level of the age factor. We observe in this plot that the quartiles of time are flat in the 25-45 range and increase after that with the upper quartile increasing faster than the median and lower quartile. In the next section, we will try summarizing this relationship more formally.

4.4.2 Fitting Models to Average Performance

As seen in Figure 4.7, the average performance looks relatively flat from the twenties through the forties. For this reason, let's begin by fitting a simple linear model to see how well it captures the relationship between performance and age. We do this with

```
lmAge = lm(time ~ age, data = menRes)
```

Here again we use *R*'s formula language to express the relationship we want fitted to the data. Our formula `time ~ age` indicates we want to fit time as a function of age. The lm() function performs linear least squares to find the best fitting line to our data, which we see has the following intercept and slope:

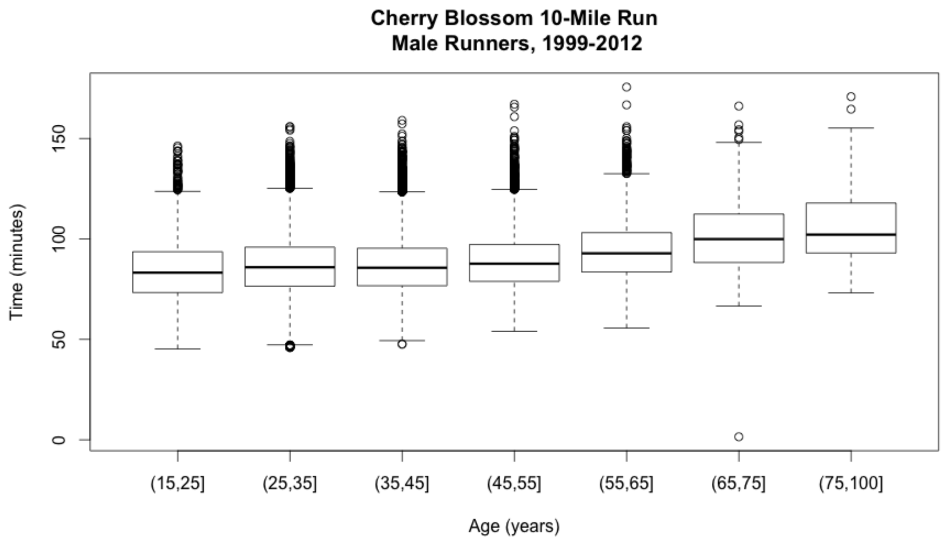


Figure 4.7: Side-by-Side Boxplots of Male Runners' Race Time vs. Age. This sequence of boxplots shows the quartiles of time for men grouped into 10-year age intervals. Those who are in the 25-35 and 35-45 ranges have roughly the same quartiles. As age increases, all the quartiles increase. However, the box becomes asymmetrical with age, which indicates that the upper quartile increases faster than the median and lower quartile.

```
lmAge$coefficients

(Intercept)         age
 78.7567186   0.2252921
```

We have captured the return value from lm() in `lmAge`. This object contains the coefficients from the fit, predicted values, residuals, and other information about the linear least squares fit of time to age. We can retrieve a brief summary of the fit with a call to summary() as follows:

```
summary(lmAge)

Call:
lm(formula = time ~ age, data = menRes)

Residuals:
    Min      1Q  Median      3Q     Max
-40.333 -10.220  -0.952   9.102  82.425

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 78.75672    0.20770  379.18   <2e-16 ***
age          0.22529    0.00517   43.58   <2e-16 ***
```

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

Residual standard error: 14.77 on 69804 degrees of freedom
Multiple R-squared:  0.02649,        Adjusted R-squared:  0.02647
F-statistic:  1899 on 1 and 69804 DF,  p-value: < 2.2e-16
```

Note that the summary() function does not produce the typical quantiles, extreme values, etc. This is because we have passed it an `lm` object, i.e.,

```
class(lmAge)
```

```
[1] "lm"
```

The summary method for class `lm` provides a different set of summary statistics that are more appropriate for a fitted linear model.

To help us assess how well the simple linear model fits the data we plot the residuals against age. As with the original scatterplot of time against age, we need to address the issue of over plotting. We use smoothScatter() to do this. Further, to help us see any curvature in the residuals, we add to the plot a horizontal line at 0. We do this with

```
smoothScatter(x = menRes$age, y = lmAge$residuals,
              xlab = "Age (years)", ylab = "Residuals",
              main = residTitle)
abline(h = 0, col = "purple", lwd = 2)
```

To help us further discern any pattern in the residuals, we augment this residual plot with a smooth curve of local averages of the residuals from the fit. That is, for a particular age, say 37, we take a weighted average of the residuals for those runners with an age in a small neighborhood of 37. Such a locally fitted curve allows us to better see deviations in the pattern of residuals. We fit the curve using loess() with

```
resid.lo = loess(resids ~ age,
                 data = data.frame(resids = lmAge$residuals,
                                   age = menRes$age))
```

Notice that the loess() function also accepts a formula object to describe the relationship to fit to the data. Here we request a fit of the `resids` variable to `age`. The data frame provided via the parameter *data* contains these two variables; it may contain others as well, but we have created this data frame specially so that it has only the residuals from `lmAge` and the ages of runners in `menRes`. Similar to lm(), the return value from loess() is a special object that contains predicted values from the fit as well as other relevant information about the curve fitted to the data.

To add the fitted curve to the smooth scatter of the residuals, we can predict the average residual for each year of age and then use lines()to "connect the dots" between these predictions to form an approximation of the fitted curve. We start by making a vector of ages from 18 to 80 with

```
age20 = 20:80
```

Now, if we have the predicted average residual for each of these ages in a vector called, say, `resid.lo.pr` then we can add the curve to the smooth scatter with

```
lines(x = age120, y = resid.lo.pr, col = "green", lwd = 2)
```

We can obtain these predicted values from the predict.loess() function. This function takes the loess object from a fit, e.g., `resid.lo` and a data frame with variables matching those used in the loess curve fitting, in this case `age`. That is, we create `resid.lo.pr` with

```
resid.lo.pr = predict(resid.lo, newdata = data.frame(age = age20))
```

Notice that we called predict() rather than predict.loess(). The predict() function is a wrapper that allows us to write parallel code that is not dependent on the form of the fit. It takes an object returned from a fit, such as that returned from lm() or loess(), and depending on which class of object it is, predict() calls the relevant function, i.e., predict.lm() for `lm` objects and predict.loess() for `loess` objects.

The augmented smoothed scatter plot appears in Figure 4.8. We see that the simple linear model tends to underestimate the performance for men over 60. This confirms our observations from the boxplot and smooth scatter plot of the upward trend in time for men over 55. We see that this simple linear model is not able to capture the change in performance with age.

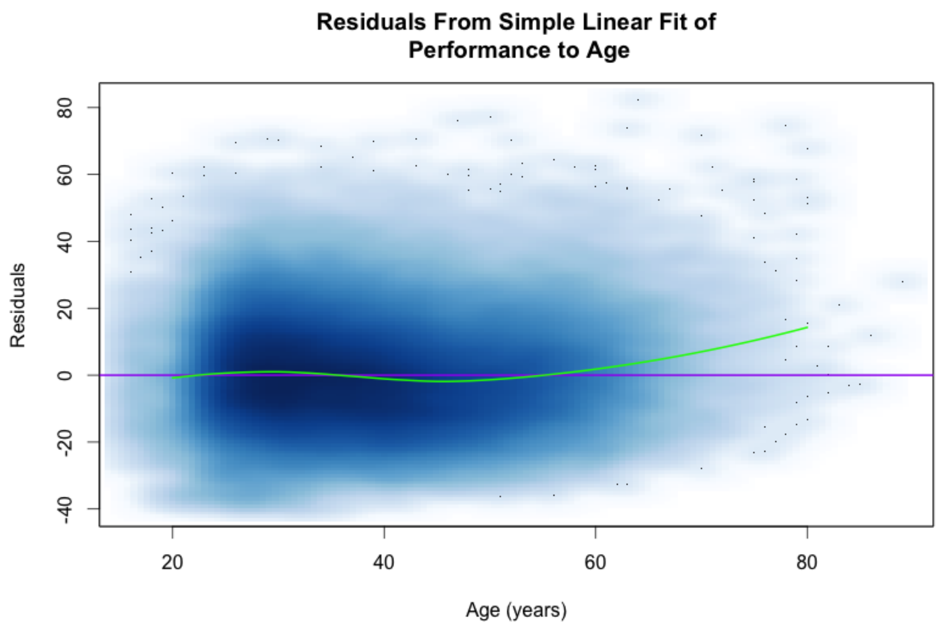


Figure 4.8: Residual Plot From Fitting a Simple Linear Model of Performance to Age. Shown here is a smoothed scatter plot of the residuals from the fit of the simple linear model of time to age for male runners who are 15 to 80 years old. Overlaid on the scatter plot are two curves. The "curve" in purple is a horizontal line at y = 0. The green curve is a local smooth of the residuals. These curves helps us see that there is a pattern in the residuals, i.e., the residuals for the above 55 group are not evenly scattered about the regression line. There are too many positive residuals, indicating the runners are slower than the prediction.

We consider two approaches to a more complex fit: a piecewise linear model and a nonparametric smooth curve. For the latter, we simply take local weighted averages of time as age varies, just as we smoothed the residuals from the linear fit. We use loess() again to do this with

```
menRes.lo = loess(time ~ age, menRes)
```

and we make predictions for all ages ranging from 20 to 80 with

```
menRes.lo.pr = predict(menRes.lo, data.frame(age = age20))
```

The curve appears in Figure 4.9.

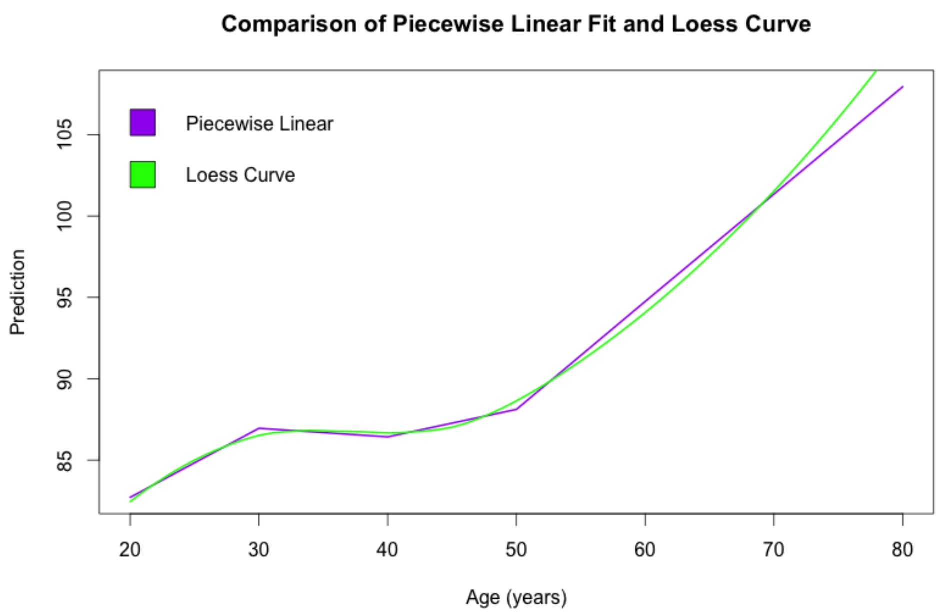


Figure 4.9: Piecewise Linear and Loess Curves Fitted to Time vs. Age. Here we have plotted the fitted curves from loess() and a piecewise linear model with hinges at 30, 40, 50, and 60. These curves follow each other quite closely. However, there appears to be more curvature in the over 50 loess fit that is not captured in the piecewise linear fit.

Next we fit a piecewise linear model which consists of several connected line segments. This is similar to the idea of the locally smoothed curve from loess() in that it allows us to bend the line at certain points to better fit the data. The difference is that the fit must be linear between the hinges. We place hinges at 30, 40, 50, and 60 and thus allow the slope of the line to change at these decade markers. The fitted "curve" appears in Figure 4.9.

How do we fit such a model to our data? Before we fit the full piecewise model, we consider a simpler model with one hinge at 50. We first create an `over50` variable that takes on the value 0 for ages 50 and under and otherwise holds the number of years over 50, e.g., 1 for someone who is 51, 2 for someone who is 52, and so on. If our fit is `a + b*age + c*over50` then for an age below

50 this is simply `a + b*age` and for an age over 50 it is equivalent to `(a - c*50) + (b + c)*age`. We see that the coefficient `c` is the change in the slope from below 50 to above 50, and the intercept makes the line segments connect.

Our first task then is to create this `over50` variable. We use the `pmax()` function, which performs an element-wise or 'parallel' maximum. We find the maximum of each element of `menRes$age - 50` and 0 with

```
over50 = pmax(0, menRes$age - 50)
```

We then fit this augmented model as follows

```
lmOver50 = lm(time ~ age + over50,
              data = cbind(menRes, over50))

summary(lmOver50)

Call:
lm(formula = time ~ age + over50, data = cbind(menRes, over50))

Residuals:
    Min      1Q  Median      3Q     Max
-40.265 -10.098  -0.881   9.060  79.044

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 82.754891   0.265039  312.24   <2e-16 ***
age          0.105693   0.007147   14.79   <2e-16 ***
over50       0.563871   0.023371   24.13   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.71 on 69803 degrees of freedom
Multiple R-squared:  0.03454,       Adjusted R-squared:  0.03451
F-statistic:  1248 on 2 and 69803 DF,  p-value: < 2.2e-16
```

Now the slope of the line for those under 50 is less steep than in our original simple linear model, and for ages over 50, the average man slows by 0.67 minutes, which is an additional 0.56 minutes a year compared to those under fifty.

We can create these `over30`, `over40`, etc. variables as follows:

```
decades = seq(30, 60, by = 10)
overAge = lapply(decades,
                 function(x) pmax(0, (menRes$age - x)))
names(overAge) = paste("over", decades, sep = "")
overAge = as.data.frame(overAge)

tail(overAge)

      over30 over40 over50 over60
69801     36     26     16      6
```

```
69802     11     1     0     0
69803      9     0     0     0
69804     26    16     6     0
69805      5     0     0     0
69806     18     8     0     0
```

Now that we have each of these variables, we find the least squares fit to them with

```
lmPiecewise = lm(time ~ . ,
                 data = cbind(menRes[, c("time", "age")], overAge))
```

Here we have used the . in the formula to indicate that the model should include all of the variables in the data frame (other than `time`) as covariates. This is equivalent to the formula,

```
time ~ age + over30 + over40 + over50 + over60
```

When we call summary() with the `lm` object `lmPiecewise`, we obtain the coefficients, their standard errors, and other summary statistics for the fit:

```
summary(lmPiecewise)

Call:
lm(formula = time ~ .,
   data = cbind(menRes[, c("time", "age")], overAge))

Residuals:
    Min      1Q  Median      3Q     Max
-40.921 -10.119  -0.885   9.023  78.965

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 74.228626   0.915248  81.102  < 2e-16 ***
age          0.424285   0.033207  12.777  < 2e-16 ***
over30      -0.477010   0.047778  -9.984  < 2e-16 ***
over40       0.221574   0.040666   5.449 5.09e-08 ***
over50       0.494432   0.052932   9.341  < 2e-16 ***
over60      -0.003601   0.077654  -0.046    0.963
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.7 on 69800 degrees of freedom
Multiple R-squared:  0.03593,        Adjusted R-squared:  0.03586
F-statistic: 520.3 on 5 and 69800 DF,  p-value: < 2.2e-16
```

Notice that the coefficient for `over60` is essentially 0, meaning that those over 60 do not slow down any faster than those in their fifties, i.e., about 0.494 minutes more per year for each year over 50 for a total of about 0.66 minutes per year.

How do we plot this piecewise linear function that we have fitted? As with the loess curve, if we have the predicted values for every year from 20 to 80 then we can ask predict() to provide fitted values for these ages. However, we need to provide predict() with all of the covariates used in making the fit, i.e., `age`, `over30`, `over40`, `over50`, and `over60`. We can create a data frame of these covariates just as we did for the full data set as follows:

```
overAge20 = lapply(decades, function(x) pmax(0, (age20 - x)))
names(overAge20) = paste("over", decades, sep = "")
overAgeDF = cbind(age = data.frame(age = age20), overAge20)

head(overAgeDF)

   age over30 over40 over50 over60
18  18      0      0      0      0
19  19      0      0      0      0
20  20      0      0      0      0
21  21      0      0      0      0
22  22      0      0      0      0
23  23      0      0      0      0

tail(overAgeDF)

   age over30 over40 over50 over60
75  75     45     35     25     15
76  76     46     36     26     16
77  77     47     37     27     17
78  78     48     38     28     18
79  79     49     39     29     19
80  80     50     40     30     20
```

Then we call predict() passing it the `lm` object, with the details of the fit, i.e., `lmPiecewise`, and the covariates to use to make the predictions, i.e., `overAgeDF`. That is, we call predict() with

```
predPiecewise = predict(lmPiecewise, overAgeDF)
```

We plot this fitted piecewise linear function with

```
plot(predPiecewise ~ age20,
     type = "l", col = "purple", lwd = 2,
     xlab = "Age (years)", ylab = "Prediction",
     main = "Comparison of Piecewise Linear Fit and Loess Curve")
```

And we add the `loess` curve with

```
lines(x = age20, y = menRes.lo.pr, col = "green", lwd = 2)
legend("topleft", fill = c("purple", "green"),
       legend = c("Piecewise Linear", "Loess Curve"), bty = "n")
```

The two fitted curves appear in Figure 4.9. We see that they follow each other quite closely. The main deviation is in the over 70 group. We did not include a hinge at 70 so our fitted model is unable to capture the sharper increase for those over 70. We may want to consider adding this additional hinge to our model to see if it improves the fit. It may see that we have made great progress in modeling the average performance, but we must interpret these results with care.

In any one year, the Cherry Blossom runners form a cross-sectional snapshot of runners of all ages. The participants in the road race are self-selected, and we can imagine that they are representative of the typical runners. We might ask ourselves: how similar is the composition of the participants from year to year? This is the topic of the next section.

4.4.3 Cross-sectional data and Covariates

In our earlier analysis, we examined the average performance for runners of different ages. That is, we looked at average performance for, e.g., 30-39 year olds and 40-49 year olds in the Cherry Blossom road race. However, we have not seen how a runner's performance changes as he or she ages. These two groups (30-39 and 40-49 year olds) are composed of different people and if these groups of people differ from each other in some significant way, e.g., those in their 30s are more likely to be world class runners and those in their 40s are more likely to be local amateur athletes, then we might be misled by comparing these two group's average performances. To further complicate the matter, we have data from 14 different races so we are also averaging across the participants in these different races. We would expect the average performances to be the same across the years. However, each year we have a self-selected group of participants, and we might wonder whether the composition of the participants has changed over the years. If it has, that could further muddy things.

We know that the race has become increasing popular. The number of annual participants is

```
table(cbRes$year)
```

```
 1999  2000  2001  2002  2003  2004  2005  2006  2007
 5548  5182  6533  7057  7488  8055  8657 10670 10854
 2008  2009  2010  2011  2012
12302 14972 15762 16041 16923
```

The size of the race has tripled over the fourteen years so it seems reasonable to question if the demographics of the participants has changed over this time period.

Historically, the race was used as a preparation for the Boston Marathon. The fastest runners in the Cherry Blossom primarily come from Ethiopia, Kenya, and Tanzania. And, their times are within a minute or two of the world record of 44:24 set in 2005 by Haile Gebrselassie from Ethiopia, who was 32 at the time (see *URL*). Professional runners continue to compete in the Cherry Blossom road race.

Let's compare the distribution of performance for the extreme years, i.e., the 1999 and 2012 races. We see below that while the fastest man has gotten faster from 1999 to 2012, the quartiles of the 2012 distribution are each about 3 minutes slower compared to 1999:

```
summary(menRes$time[menRes$year == 1999])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  46.98   74.82   84.29   84.35   93.06  170.80
```

```
summary(menRes$time[menRes$year == 2012])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  45.25   77.57   87.47   88.44   97.78  151.00
```

Could it be that the men competing in 2012 are older and therefore slower than their counterparts in 1999? We can compare the age distributions of the runners in the two races. For simplicity, we make two vectors of age for the 1999 and 2012 runners with

```
age1999 = menRes[ menRes$year == 1999, "age" ]
age2012 = menRes[ menRes$year == 2012, "age" ]
```

We next superpose the density curves for the two sets of ages. We do this as follows:

```
plot(density(age1999, na.rm = TRUE),
     ylim = c(0, 0.05), col = "purple",
     lwd = 3,  xlab = "Age (years)",
     main = "Age Distribution for 1999 and 2012 Male Runners")
lines( density(age2012, na.rm = TRUE),
       lwd = 3, col="green")
legend("topleft", fill = c("purple", "green"),
       legend = c("1999", "2012"), bty = "n")
```

Note that the first time we made this plot we used the default x and y axis limits in the call to plot(). We found that the y axis was not large enough to include the peak of the second density when we added it to the first density plot so we remade this plot and specified `ylim = c(0, 0.05)` to accomodate the higher peak of the 2012 density. Typically the visualization process is iterative like this. We make plots using the default settings for most of the arguments, and as we uncover interesting structure, we remake the plots to fix the scale and to add information, e.g., axis labels, titles, legends, color, line type and thickness, etc.

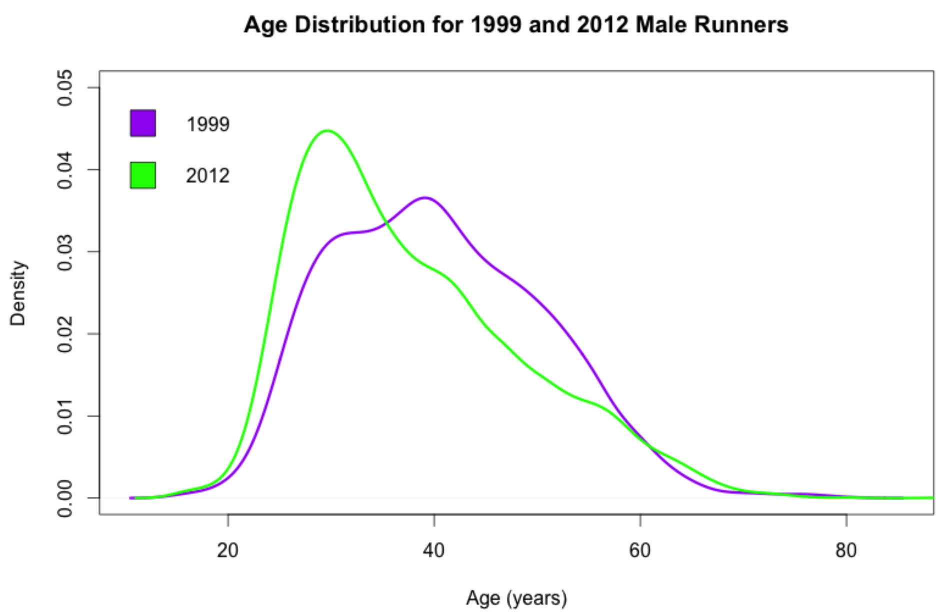


Figure 4.10: Density Curves for the Age of Male Runners in 1999 and 2012. These two density curves have quite different shapes. The 1999 male runners have a broad, nearly flat mode where they are roughly evenly distributed in age from 28 to 45. In contrast, the 2012 runners are younger with a sharper peak just under 30 years and a skew right distribution.

The density curves in Figure 4.10 are surprising. The males in 2012 are not older. In fact, the opposite is the case. There are many more younger men in 2012 in comparison to 1999, as evidenced by the sharp peak in the 2012 distribution at about 30. We can also compare these two distributions with a quantile-quantile plot. We leave this as an exercise. The difference in performance between 1999 and 2012 is subtler than simply having an aging population of runners. We need to control the covariates, age and year, simultaneously when we analyze race performance.

In the previous section, we saw how the average performance was flat for runners in their 30s and rose slightly in the 40s and more sharply in the 50s and 60s. We make separate smooth curves of time vs. age for the 1999 and 2012 runners and plot them together as follows:

```
mR.lo99 = loess(time ~ age, menRes[ menRes$year == 1999, ])
mR.lo.pr99 = predict(mR.lo99, data.frame(age = age20))

mR.lo12 = loess(time ~ age, menRes[ menRes$year == 2012, ])
mR.lo.pr12 = predict(mR.lo12, data.frame(age = age20))

plot(mR.lo.pr99 ~ age20,
     type = "l", col = "purple", lwd = 2,
     xlab = "Age (years)", ylab = "Prediction (minutes)",
     main = "Comparison of Average Performance in 1999 and 2012")

lines(x = age20, y = mR.lo.pr12, col = "green", lwd = 2)

legend("topleft", fill = c("purple", "green"),
       legend = c("1999", "2012"), bty = "n")
```

We see in Figure 4.11 that the two curves are similar in shape but the curve for 2012 sits above the 1999 curve. There appears to be a consistent difference between these two groups of runners. We can find the difference in times for the two curves for each year of age with

```
gap14 = mR.lo.pr12 - mR.lo.pr99
summary(gap14)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.007   2.891   4.732   4.447   5.555   8.524


names(gap14) = age20
print(head(gap14), digits = 3)


  20   21   22   23   24   25
4.54 4.57 4.61 4.65 4.69 4.73


print(tail(gap14), digits = 3)


  75   76   77   78   79   80
6.23 6.65 7.09 7.55 8.03 8.52
```

The typical deviation is 4.7 minutes. This difference narrows to 2 minutes for men in their 50s and gradually widens for men in their 60s, 70s, and 80s from 2.5 to 8.5 minutes.

We mention one last idea for comparing these two distributions of runners, and we leave it to the exercises to carry out this comparison. In track, there is a performance standard called age grading

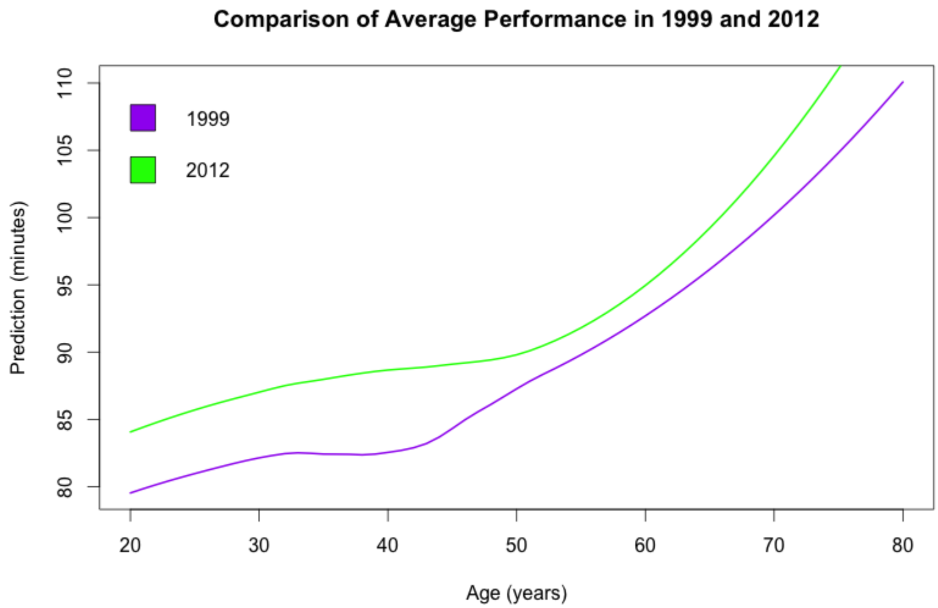


Figure 4.11: Loess Curves Fit to Performance for 1999 and 2012 Male Runners. This loess fit of performance to age for 2012 male runners sits above the fit for 1999 male runners. The gap between these curves is about 5 minutes for most years. The exception is in the late 40s to early 60s where the curves are within 2-3 minutes of each other. Both curves have a similar shape.

that measures an individual's performance based on his or her age. It normalizes the individual's time by the world record for that distance for that age group [**?**]. Since the fastest runners in the Cherry Blossom road race perform close to the world record, we might uses the fastest runner in each age-sex category to normalize the times. To minimize the year-to-year fluctuations, we can smooth the fastest times and use these smoothed times to normalize each runner's time. When we do this, we find the age graded performance for 1999 and 2012 both roughly follow the normal distribution. However, the 1999 runners tend to be better than their 2012 counter parts as evidenced by the peak at 1.4 rather than 1.5 and a smaller IQR.

The distribution of participants appears to have changed over the years, and this points out the main issue with cross-sectional studies. However, there is an advantage to having 14 years worth of race results. It is possible that some runners have participated in the race over several years and we can study how each runner's performance changes as he or she grows older. In order to do this, we will need to connect runners across the years. This is the subject of the next section.

4.5 Constructing a Record for Individual Runner Across Years

We would like to match records from runners who have participated in more than one Cherry Blossom run. The race results do not have unique identifiers for each person so we will need to construct these from the information we have on each race entrant. Ideally we would use all of the information, i.e., the runner's name, hometown, age, race time, and the year of the race. However, if this information is reported inconsistently from one year to the next then this can reduce the number of matches. On the other hand, even using all of this information we may be incorrectly matching records from two different athletes. Whatever approach we devise will not be completely accurate, and the purpose of this section will be to investigate several possibilities and settle on one that we think is reasonable.

We consider the following questions:

- How many entrants are there over the 14 years?
- How many unique names are there among these entrants?
- How many names appear twice, three times, four times, etc. and what name occurs most often?
- How often does a name appear more than once in a year?

Answering these questions will give us a sense of the magnitude of the matching problem. Additionally, we will consider how to improve the matching by cleaning the `name` and `home` values. For example, recall that we picked up some trailing blanks when we parsed the text tables. Now might be a good time to eliminate them. We also noted earlier that capitalization was inconsistent. This can prove problematic for matching records. Other issues with cleaning the character strings will crop up as we begin to examine the records more carefully. Let's begin to answer these questions by examining summary statistics and sets of records.

How many entrants are there over the 14 years? We use nrow() to find out:

```
nrow(menRes)
```

```
[1] 69806
```

Recall, we have dropped those records with time under 20 minutes, and age under 16. How many unique names are there?

```
length(unique(menRes$name))
```

```
[1] 54096
```

How many names appear once, twice, etc.? We can figure this out by calling table() on `table(menRes$name)`, i.e.,

```
table(table(menRes$name))
```

```
    1      2      3      4      5      6      7      8
43947   6979   1850    715    313    184     81     15
    9     10     12     14     17
    6      1      1      3      1
```

We see that over 7000 names appear two times throughout the 14 years, and one name appears 17 times. We know this name must correspond to multiple people because we have only 14 years of race results.

Which name appears 17 times? We can find that with

```
head( sort(table(menRes$name), decreasing = TRUE), 1)

Michael Smith
                        17
```

Let's examine all of the information about these 17 Michael Smith's. We extract them with

```
menRes[menRes$name == "Michael Smith", ]

[1] year sex  name home age  time dob  ID
<0 rows> (or 0-length row.names)
```

How is it that we found no rows with a name of Michael Smith? The issue of blanks enters here. When we examine the first few values for `name`, we see these blanks:

```
head(menRes$name)

[1] "Worku Bikila          " "Lazarus Nyakeraka     "
[3] "James Kariuki         " "William Kiptum         "
[5] "Joseph Kimani         " "Josphat Machuka        "
```

We have not included blanks in Michael Smith's name, i.e., `"Michael Smith          "`. Rather than doing that, let's clean up the names and remove extra blanks.

Any blanks appearing before of after a name can be dropped. Also, if there are multiple blanks between, e.g., the first and last name, we can convert them to one blank. The gsub() function is helpful here. We use gsub() successively as follows:

```
nameClean = gsub("^[[:blank:]]+", "", menRes$name)
nameClean = gsub("[[:blank:]]+$", "", nameClean)
nameClean = gsub("[[:blank:]]+", " ", nameClean)
```

The first substitution eliminates all beginning blanks, the second all trailing blanks, and the third substitutes multiple contiguous blanks with a single blank. Notice that we use the meta character `[:blank:]` so that we find all forms of blank characters.

When we rerun our double call to table() we find even more duplicates:

```
table(table(nameClean))

    1     2     3     4     5     6     7     8
29293  7716  2736  1386   712   417   249   149

    9    10    11    12    13    14    15    17
   92    56    44    19     7     3     1     1

   18    19    30
    1     1     1
```

Now we have a name that appears 30 times. Is it still Michael Smith?

```
head( sort(table(nameClean), decreasing = TRUE), 1)

Michael Smith
           30
```

Indeed, it is.

A natural next question might be whether we can do more cleaning that will improve the matching. We have seen that the column headers have inconsistent capitalization. The same is undoubtedly the case for the name. We can check this, but we can also simply proceed to make all characters lower case letters with

```
nameClean = tolower(nameClean)
```

Additionally, we can remove punctuation such as a period after someone's middle initial, the apostrophe in a name like `O'Reilly`, and any stray commas. We do this in once call to gsub() with

```
nameClean = gsub("['.,]", "", nameClean)
```

This additional cleaning picked up three more Michael Smith's:

```
head( sort(table(nameClean), decreasing = TRUE), 1)

michael smith
           33
```

With so many duplicates, let's figure out how many times a name appears in the same year. We can create a table of year-name combinations with

```
tabNameYr = table(menRes$year, nameClean)
```

and then call max() to find the cell in the table with the greatest count, i.e.,

```
max(tabNameYr)

[1] 5
```

Is this Michael Smith again? It takes a bit of work to find the name associated with this maximum. The table saved in `tabNameYr` is of class `table`, which we see is a numeric vector with three attributes, `dim`, `dimnames` and `class`. Calls to class(), mode(), and attributes(), help us figure this out, i.e.,

```
class(tabNameYr)

[1] "table"

mode(tabNameYr)

[1] "numeric"

names(attributes(tabNameYr))

[1] "dim"      "dimnames" "class"
```

There are several implications of this data structure. First, some matrix functions work on a `table`, e.g., we can call dim() and colnames() and find

```
dim(tabNameYr)

[1]    14 39077
```

```
head(colnames(tabNameYr), 3)

[1] "8illiam maury"   "a gudu memon"    "a miles simmons"
```

Notice we have uncovered another piece of messy data! To find out which cell has a count of five, we can use which(), but to find the row and column location, we need to include the *arr.ind* argument in our call. That is,

```
which( tabNameYr == max(tabNameYr) )

[1] 356034

which( tabNameYr == max(tabNameYr), arr.ind = TRUE )

     row   col
2012  14 25431
```

Finally we locate the name(s) with

```
indMax = which( tabNameYr == max(tabNameYr), arr.ind = TRUE )
colnames(tabNameYr)[indMax[2]]

[1] "michael brown
```

It's Michael Brown, not Michael Smith!

Now that we have a cleaned version of runner's name, we add it to our data frame with

```
menRes$nameClean = nameClean
```

We use this format of the name to create our identifier.

We can also derive an approximation to year of birth because we have the runner's age and the year of the race. The difference between these two is an approximation to age because the race is held on the first Sunday in April every year. Those runners who have a birthday in the first seven days of April may have their age reported inconsistently from one race year to the next. What fraction of the records can we expect to have this problem? We create `yob` and add it to the data frame with

```
menRes$yob = menRes$year - menRes$age
```

Since we know that there is also an issue with blanks and capitalization for hometown. We leave it as an exercise to clean the values for `home` and assign the cleaned version of `home` into `menRes` as `home`.

Let's look closer at the records for Michael Brown in our data frame. We do this with

```
mb = which(nameClean == "michael brown")
birthOrder = order(menRes$yob[mb])
menRes[mb[birthOrder], -(2:3)]
      year          home age     time     nameClean  yob
5696  2000     tucson az  61 96.88333 michael brown 1939
53001 2010 north east md  57 92.26667 michael brown 1953
58669 2011 north east md  58 85.95000 michael brown 1953
66425 2012 north east md  59 88.43333 michael brown 1953
47374 2009     oakton va  52 99.73333 michael brown 1957
```

```
40144 2008     ashburn va  50  93.73333 michael brown 1958
45642 2009     ashburn va  51  88.56667 michael brown 1958
54063 2010     ashburn va  52  99.75000 michael brown 1958
66702 2012      reston va  54  89.95000 michael brown 1958
28434 2006    chevy chase  40  84.56667 michael brown 1966
50671 2010 chevy chase md  44  79.35000 michael brown 1966
67711 2012 chevy chase md  46  95.81667 michael brown 1966
18363 2004  berryville va  26  76.31667 michael brown 1978
38751 2008   arlington va  24  84.68333 michael brown 1984
55055 2010    new york ny  26 110.88333 michael brown 1984
57918 2011   arlington va  27  81.70000 michael brown 1984
63515 2012   arlington va  28  70.93333 michael brown 1984
65710 2012     clifton va  24  84.88333 michael brown 1988
```

What observations can we make about these various michael browns?

- The three entries for michael brown born in 1953 seem to be the same person because all have a hometown of `"north east md"`. Additionally, the three race times are within 7 minutes of each other.
- The four entries for michael brown born in 1958 have race years of 2008, 2009 2010 and 2012. The most recent entry lists Reston VA for a hometown while the other three show Ashburn VA. Do we have 1, 2, 3 or 4 different michael brown's here? The 2010 entrant ran the slowest of the 4 races by about 11 minutes and the other three times are closer. An Internet search reveals that Reston and Ashburn are within 22 km of each other. It is conceivable that these four records belong to the same individual who moved from Ashburn to Reston between Apr 2010 and 2012. We can't know for sure.
- Another three michael brown entries have 1966 for a birth year. All three list Chevy Chase as the hometown, except that for 2006 the state, MD, is not provided. When we examine more locations for other runners in 2006 we find that none of them list a state. These three michael brown records also have a range of 11 minutes for time with the middle year (2010) being the fastest.
- Next, we have four records for michael browns born in 1984, with races in 2008, 2010, 2011, and 2012. Of these, the 2010 record is clearly a different person as his home is listed as New York NY and his race time is 25 to 40 minutes slower than the other three records. These three all have the same hometown of Arlington VA. They also have increasingly better times with a 2008 time of 84 and a 2012 time of 71 minutes. It is not unreasonable to think that these three records belong to is the same person who has been training and running faster as he ages.
- Lastly, notice that the five michael browns registered for the same race competed in 2012, and they have different years of birth (1953, 1958, 1966, 1984, and 1988).

We summarize our various observations to make a first attempt to create an identifier for individuals. We might paste together the cleaned name and the derived year of birth. We do this with

```
menRes$ID = paste(nameClean, menRes$yob, sep = "")
```

We have ignored the information provided by the hometown and the performance times and so have created the least restrictive identifier.

Since our goal is to study how an athlete's time changes with age, let's focus on those IDs that appear in at least eight races. To do this, we first determine how many times each `ID` appears in `menRes` with

```
races = tapply(menRes$year, menRes$ID, length)
```

Then we select those IDs that appear at least 8 times with

```
races8 = names(races)[which(races > 7)]
```

and we subset `menRes` to select the entries belonging to these identifiers with

```
menRes8 = menRes[ menRes$ID %in% races8, ]
```

Finally, we organize the data frame so that entries with the same `ID` are contiguous. This will make it easier to examine records, etc. We can do this with

```
orderByRunner = order(menRes8$ID, menRes8$year)
menRes8 = menRes8[orderByRunner, ]
```

An alternative organization for the data is to store them as a list with an element for each ID in `races8`. In this list, each element is a data frame containing only those results for the records with the same ID. We can create this list with

```
menRes8L = lapply(races8, function(x) menRes[ menRes$ID == x, ])
names(menRes8L) = races8
```

Which data structure is preferable? That will depend on what we want to do with the data. In the following we show how to accomplish a task using both approaches to help make a comparison between the two structures. In the next section, we find it easiest to work with the list of data frames as we often need to apply a function of multiple arguments to each runner's entries.

How many IDs do we have left?

```
length(unique(menRes8$ID))

[1] 481

length(menRes8L)

[1] 481
```

We might also want to discard matches if the performance varies too much from year to year. How large a fluctuation would make us think that we have mistakingly connected two different people? Of course, we don't want to bias our results by eliminating an individual whose run times vary a lot. Let's look at a few records where the year to year difference in time exceeds 20 minutes. We determine which satisfy this constraint with

```
gapTime = tapply(menRes8$time, menRes8$ID,
                 function(t) any( abs(diff(t)) > 20 ) )
```

or with

```
gapTime = sapply(menRes8L,
                 function(df) any( abs(diff(df$time)) > 20 ) )
```

How many of these runners do we have?

```
sum(gapTime)

[1] 49
```

Slightly reformatted displays of the first two of these athletes are

```
head(menRes8L[ gapTime ], 2)

[[1]]
year                 home age      time  yob            ID
1999      gaithersburg md  32  96.51667 1967  abiy zewde1967
2000   montgomery vill md  33  96.63333 1967  abiy zewde1967
2001   montgomery vill md  34  89.10000 1967  abiy zewde1967
2002   montgomery vill md  35 123.00000 1967  abiy zewde1967
2003      gaithersburg md  36  97.68333 1967  abiy zewde1967
2004   montgomery vill md  37 100.36667 1967  abiy zewde1967
2006         gaithersburg  39 108.40000 1967  abiy zewde1967
2008   montgomery vill md  41  98.78333 1967  abiy zewde1967
2009 montgomery villag md  42  98.50000 1967  abiy zewde1967
2010 montgomery villag md  43  99.91667 1967  abiy zewde1967
2011 montgomery villag md  44 113.10000 1967  abiy zewde1967
2012 montgomery villag md  45  84.88333 1967  abiy zewde1967

[[2]]
2005        washington dc  27  80.38333 1978 adam hughes1978
2006           washington  28  85.16667 1978 adam hughes1978
2007        washington dc  29  77.78333 1978 adam hughes1978
2008        washington dc  30  74.23333 1978 adam hughes1978
2009        washington dc  31 108.06667 1978 adam hughes1978
2010        washington dc  32 103.06667 1978 adam hughes1978
2011        washington dc  33  77.11667 1978 adam hughes1978
2012        washington dc  34  77.76667 1978 adam hughes1978
```

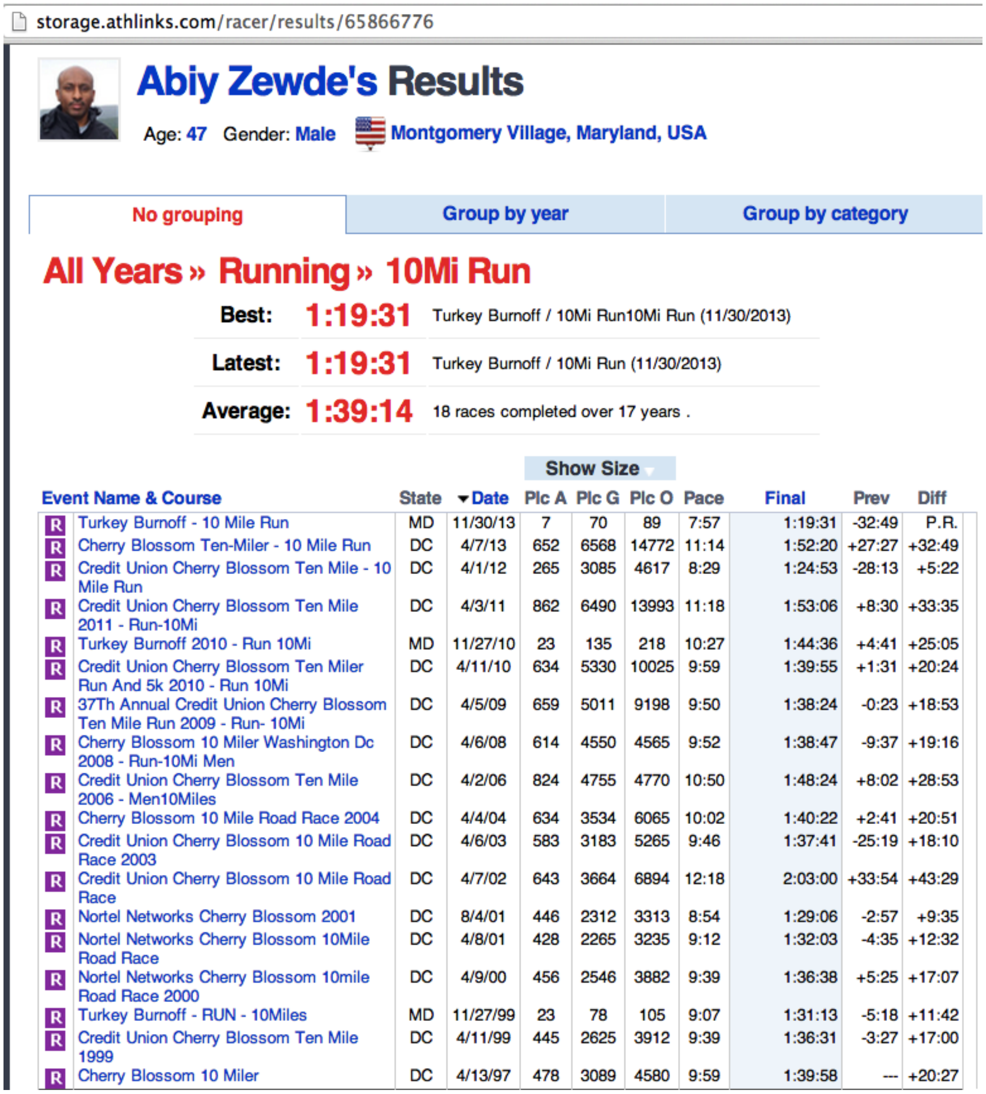
The name `abiy zewde` seems unusual enough to most likely be the same person participating in 12 of the 14 races even though the hometown has changed over the years and the race results differ by nearly 40 minutes with one of the fastest time being the most recent when he was 45 years old. In fact, a Google search locates a Web page at `storage.athlinks.com/racer/results/65866776` with his published race times from several different runs. A screenshot of this page appears in Figure 4.12. Clearly these entries all belong to the same person.

Do we want to further restrict our matching to those with the same hometowns? This would eliminate abiy zewde even though we're quite certain the records all belong to the same individual. We could identify the mismatches and manually examine them for potentially false matches. We need to eliminate the state abbreviation from the end of those records that have one because the 2006 records do not have it. We do this by substituting a blank followed by 2 letters occurring at the end of the string with an empty string, i.e.,

```
gsub("[[:blank:]][a-z]{2}$", "", home)
```

We leave it as an exercise to determine how to limit the matches to those where the entries have the same hometown and to assess whether this additional restriction should be added to the matching process.

Here, we consider a less strict matching where we match only those records that have the same values for the state of residence. To do this, we need to create a new variable that holds the 2 letter abbreviation for the state. We return to work with `menRes` because the data structure is simpler and we will maintain consistency. We pull the last 2 characters from each home string. This will be the


storage.athlinks.com/racer/results/65866776

Abiy Zewde's Results

Age: 47 Gender: Male Montgomery Village, Maryland, USA

No grouping | Group by year | Group by category

All Years » Running » 10Mi Run

Best:	1:19:31	Turkey Burnoff / 10Mi Run10Mi Run (11/30/2013)
Latest:	1:19:31	Turkey Burnoff / 10Mi Run (11/30/2013)
Average:	1:39:14	18 races completed over 17 years .

Show Size

	Event Name & Course	State	Date	Plc A	Plc G	Plc O	Pace	Final	Prev	Diff
R	Turkey Burnoff - 10 Mile Run	MD	11/30/13	7	70	89	7:57	1:19:31	-32:49	P.R.
R	Cherry Blossom Ten-Miler - 10 Mile Run	DC	4/7/13	652	6568	14772	11:14	1:52:20	+27:27	+32:49
R	Credit Union Cherry Blossom Ten Mile - 10 Mile Run	DC	4/1/12	265	3085	4617	8:29	1:24:53	-28:13	+5:22
R	Credit Union Cherry Blossom Ten Mile 2011 - Run-10Mi	DC	4/3/11	862	6490	13993	11:18	1:53:06	+8:30	+33:35
R	Turkey Burnoff 2010 - Run 10Mi	MD	11/27/10	23	135	218	10:27	1:44:36	+4:41	+25:05
R	Credit Union Cherry Blossom Ten Miler Run And 5k 2010 - Run 10Mi	DC	4/11/10	634	5330	10025	9:59	1:39:55	+1:31	+20:24
R	37Th Annual Credit Union Cherry Blossom Ten Mile Run 2009 - Run- 10Mi	DC	4/5/09	659	5011	9198	9:50	1:38:24	-0:23	+18:53
R	Cherry Blossom 10 Miler Washington Dc 2008 - Run-10Mi Men	DC	4/6/08	614	4550	4565	9:52	1:38:47	-9:37	+19:16
R	Credit Union Cherry Blossom Ten Mile 2006 - Men10Miles	DC	4/2/06	824	4755	4770	10:50	1:48:24	+8:02	+28:53
R	Cherry Blossom 10 Mile Road Race 2004	DC	4/4/04	634	3534	6065	10:02	1:40:22	+2:41	+20:51
R	Credit Union Cherry Blossom 10 Mile Road Race 2003	DC	4/6/03	583	3183	5265	9:46	1:37:41	-25:19	+18:10
R	Credit Union Cherry Blossom 10 Mile Road Race	DC	4/7/02	643	3664	6894	12:18	2:03:00	+33:54	+43:29
R	Nortel Networks Cherry Blossom 2001	DC	8/4/01	446	2312	3313	8:54	1:29:06	-2:57	+9:35
R	Nortel Networks Cherry Blossom 10Mile Road Race	DC	4/8/01	428	2265	3235	9:12	1:32:03	-4:35	+12:32
R	Nortel Networks Cherry Blossom 10mile Road Race 2000	DC	4/9/00	456	2546	3882	9:39	1:36:38	+5:25	+17:07
R	Turkey Burnoff - RUN - 10Miles	MD	11/27/99	23	78	105	9:07	1:31:13	-5:18	+11:42
R	Credit Union Cherry Blossom Ten Mile 1999	DC	4/11/99	445	2625	3912	9:39	1:36:31	-3:27	+17:00
R	Cherry Blossom 10 Miler	DC	4/13/97	478	3089	4580	9:59	1:39:58	---	+20:27

Figure 4.12: Screen Shot of One Runner's Web Page of Race Results. This Web page at `storage.athlinks.com` contains the race results of one runner who participated in the Cherry Blossom run for 12 of the 14 years for which we have data. Notice that his fastest time was from his most recent run in 2012 where he completed the race in under 85 minutes. He was 45 at that time. Also, his slowest time was 123 minutes in 2002 at the age of 35.

state, if it is present. We know that in 2006, state was not present so we set these to NA. For those athletes who come from outside the US, we will be picking up the last two letters of either the country or province, but these should not dramatically affect our matches.

We first determine how many characters are in each value for home with

```
strL = nchar(menRes$home)
```

Then we use it to extract the last two characters and add them back to our data frame with

```
menRes$state = substr(menRes$home, start = strL - 1, stop = strL)
```

And we set the 2006 values to NA:

```
menRes$state[menRes$year == 2006] = NA
```

Next, we recreate the ID so that it includes state. We do this with

```
menRes$ID = paste(menRes$nameClean, menRes$yob,
                  menRes$state, sep = "")
```

And we again select those IDs that occur at least 8 times with

```
races = tapply(menRes$year, menRes$ID, length)
races8 = names(races)[which(races > 7)]
menRes8L = lapply(races8, function(x) menRes[ menRes$ID == x, ])
```

We do not create the data frame version of this information, i.e., menRes8, because in the next section we work solely with the list structure.

This addition to the runner id further reduces the number of runners who have completed 8 races, i.e.,

```
length(races8)
```

```
[1] 306
```

We now have 306 athletes who have the same name, year of birth, and state and who have run in 8 of the 14 races. We carry on with this set of matches we have obtained thus far, and in the next section, we examine how each runner's performance changes as he grows older.

4.6 Modeling the Change in Time for Individuals

These data include recordings for athletes from 20 to 80 years old. However, we don't have records for any one person that covers this range of years. That's not possible because we have only 14 years of race results so we can at most observe a 20 year old until he turns 33 and an 80 year old when he was 67. This means when we examine the performance of an individual over time, we will be looking at short time series that are at most 14 years long. To examine performance from 20 to 80 necessarily means that we rely on the cross-sectional aspect of the data, but there is information to be gleaned in these short time series.

It's reasonable to imagine that over a short period of time, say 8 to 10 years, a runner's performance will be roughly linear with age. We can make plots to see if this is the case. We begin by creating a blank canvas with

```
plot( x = 40, y = 60, type = "n",
      xlim = c(20, 80), ylim = c(40, 160),
      xlab = "Age (years)", ylab = "Time (minutes)",
      main = "Individual Runners Performance")
```

To this canvas we add one runner's records with

```
oneRunner = menRes8L[[1]]
lines(oneRunner$time ~ oneRunner$age,
      col = rainbow(12)[1], lwd = 2)
```

We can apply this call to lines() over say the first 40 IDs in our list with

```
invisible( mapply(function(df, i) {
                    lines(df$time ~ df$age,
                          col = rainbow(12)[(i %% 12)], lwd = 2)
                  }, menRes8L[1:40], i = 1:40) )
```

The invisible() function hides the return value from mapply(). Since mapply() adds lines to the canvas, it returns `NULL` for each iteration which we can safely ignore.

In Figure 4.13 we see 40 line plots for 40 athletes. Some are flat, others fluctuate quite a bit, and others for older runners increase. Nonetheless, fitting a line to each individual's performance seems a reasonable approach.

A longitudinal analysis of each individual runner, implicitly controls for covariates that may influence performance, e.g., sex. One exception is the race condition in any given year–some years might be slow and some fast due to changes in the course or weather. However, it seems plausible that such an effect will be uncorrelated with age and so amounts to measurement noise.

Now that we have our list of runners, we wish to fit a line to each runner's performance. If we write a function that carries out this work for one runner, then we can apply it to all of the runners in our list. What do we need this function to do? We can have it fit a line via lm(). What do we want the function to return? We are interested in the coefficient for age, but we need to be able to interpret it in the context of age. Since we have multiple ages for each runner, let's return a middle value for age. And, while we are at it, let's also return a predicted value for the runner's performance at that age. What inputs do we need for our function? Really just the runner's time and age. We can pass these into our function as separate parameters or we can pass in our data frame. If we do the latter, then we need to know the names of the variables to fit. Let's do this. Let's also have our function add the fitted line segment to a plot. We can make this an optional operation by adding a parameter that has a default value of `FALSE` so the function only adds the line when the call specifically specifies this parameter as `TRUE`.

We leave the writing of this function as an exercise. We specify only that the input parameters are called *oneRunner* and *addLine*, the return value is a numeric vector of length three with the coefficient, age, and prediction in that order, and the name of the function is fitOne().

We call fitOne() to add the fitted lines to the 40 line plots for the runners in Figure 4.13.

```
invisible(lapply(menRes8L[1:40], fitOne, addLine = TRUE))
```

See the grey line segments in Figure 4.14. These line segments seem to capture each runner's performance. Next we fit lines to all 306 athletes with the following call to lapply():

```
menRes8LongFit = lapply(menRes8L, fitOne)
```

We can extract the 306 coefficients for `age` and each runner's representative age with

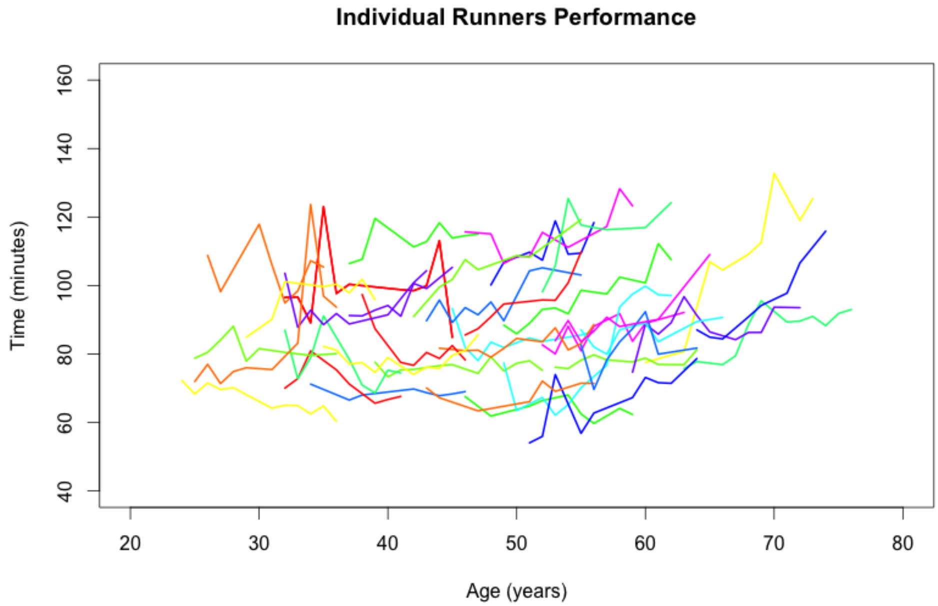


Figure 4.13: Times for Multiple Races 40 Runners. This line plot shows the times for 40 runners who completed at least 8 Cherry Blossom races. Each set of connected segments corresponds to the times for one runner. Looking at all 40 line plots, we see a similar shape to the scatter plot in Figure 4.5, i.e. an upward curve with age. However, we can also see how an individual's performance changes. For example, many middle-aged runners show an increase in time with age but that is not the case for all. Some of them improve and others remain roughly constant.

```
coeffs = sapply(menRes8LongFit, "[", 1)
ages = sapply(menRes8LongFit, "[", 2)
```

Now we have a single coefficient that represents the relationship between performance and age for each runner who ran at least 8 times (and who resided in the same state over those race years). This coefficient has units of minutes per year. A positive coefficient means that the runner is slowing down by that number of minutes a year.

Looking at how these coefficients vary with age, we see a wide distribution. There is plenty of individual variation in performance with a few in their 50s and 60s getting faster and many in their 30s slowing down. However, we also see a relationship beween age and performance in Figure 4.15. There appears to be a positive linear trend in the coefficients. We fit this with

```
longCoeffs = lm(coeffs ~ ages)
```

The summary from the fit appears below

```
summary(longCoeffs)

Call:
lm(formula = coeffs ~ ages)
```

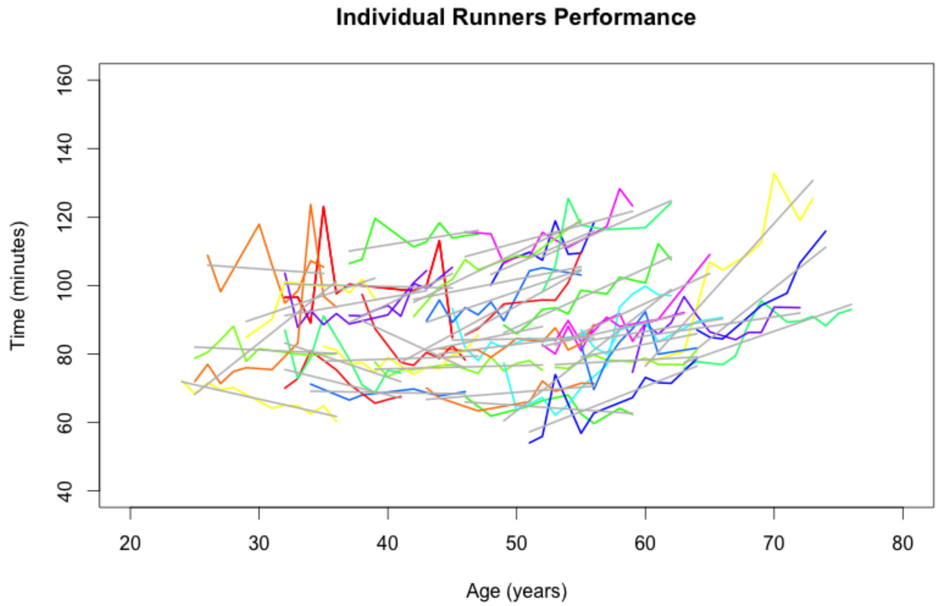


Figure 4.14: Linear Fits of Time to Age for Individual Runners. Here we have augmented the line plot from Figure 4.13 with the least squares fit of performance for each of the 40 runners. These are the 40 grey line segments plotted on each of the individual runner's times series.

```
Residuals:
    Min      1Q  Median      3Q     Max
-4.4018 -0.6382 -0.0231  0.5643  3.3571

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.965571   0.305607  -6.432 4.89e-10 ***
ages         0.055403   0.006177   8.969  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.01 on 304 degrees of freedom
Multiple R-squared:  0.2093,        Adjusted R-squared:  0.2067
F-statistic: 80.45 on 1 and 304 DF,  p-value: < 2.2e-16
```

We have added to the plot the fitted line along with a reference line at 0 and a smooth curve fit to the coefficients using loess(). This graph suggests that, on average, performance improves for people who are younger than about 35. That is, the age coefficient is negative for ages under 35. The hypothetical

Figure 4.15: Coefficients from Longitudinal Analysis of Athletes. This scatter plot displays the slope of the fitted line to each of the 300+ runners who completed in at least 8 Cherry Blossom road races. A negative coefficient indicates the runner is getting faster as he ages. Notice that nearly all of the coefficients for those over 50 are positive. The typical size of this coefficient for a 50-year old is about one minute per year.

"average" runner who is older than 35 will slow down. By age 60, the typical runner will slow by about 1.3 minutes per year, about twice as fast as indicated by the cross-sectional analysis.

4.7 Scraping Data from the Web

The race results for the Cherry Blossom Ten Mile Run are available at `http://www.cherryblossom.org`. Figure 4.1 shows a screen shot of the site's main Web page with links leading to each year's results. The results for, e.g., men in 2012 are displayed in the screen shot in Figure 4.2. We see that the data there are simply formatted in what appears to be a block of plain text arranged in fixed-width columns. We can examine the source code for the Web page to check if this is the case. We do this in, e.g., a Chrome browser by clicking on `View -> Developer -> View Source`. When we do this we see that the table itself contains no *HTML* markup, and it has been inserted into a <*pre*> node within the document. (The *HTML* for the page shown in Figure 4.2 is shown in Figure 4.16.) It should be very easy to extract this "table" from the *HTML* for further processing.

```
<!-- saved from url=(0022)http://internet.e-mail -->
<html>
<pre>
                 Credit Union Cherry Blossom Ten Mile Run
                 Washington, DC     Sunday, April 1, 2012

                Official Male Results (Sorted By Net Time)

Place Div  /Tot   Num    Name                   Ag Hometown             5 Mile  Time    Pace
===== =========== ====== ====================== == ==================== ======= ======= =====
    1     1/347        9 Allan Kiprono          22 Kenya                  22:32   45:15  4:32
    2     2/347       11 Lani Kiplagat          23 Kenya                  22:38   46:28  4:39
    3     1/1093      31 John Korir             36 Kenya                  23:20   47:33  4:46
    4     1/1457      15 Ian Burrell            27 Tucson AZ              23:50   47:34  4:46
and so on
 7190   375/375    18780 Larry Hume             56 Arlington VA         1:07:17 2:27:20 14:44
 7191  1093/1093   19104 Sean-Patrick Alexander 35 Alexandria VA        1:08:44 2:27:30 14:45
 7192              22280 Joseph White              Forestville MD       1:10:04 2:28:58 14:54
 7193   648/648     6555 Lee Jordan             48 Herndon VA           1:09:06 2:30:59 15:06
</pre>
```

Figure 4.16: Screen shot of the Source for Men's 2012 Cherry Blossom Results. This screen shot is of the *HTML* source for the male results for the 2012 Cherry Blossom road race. Notice that times given are for the midpoint of the race (`5 Mile`) and the finish (`Time`). We know the finish time is net time from reading the header. While the format is not quite the same as the female results for 2011 (see Figure 4.17), both are plain text tables within *<pre>* nodes in an *HTML* document.

We examine one more year to ascertain if the format is the same. When we view the source for the page of 2011 women's results, we see that the basic format is the same. A screen shot of the source for 2011 female results appears in Figure 4.17. However, the columns are not identical. In 2011, a net time is reported as well as a time. And, following the pace column there is a column labelled `S`, which has an exclamation mark for the first few runners and nothing for the rest. Our task here is simply to extract the text table so we need only locate the table and extract it as a block of text. Other functions will take care of turning the columns of information into variables.

We use the htmlParse() function in the XML package to scrape the 2012 male's page from the site.

```
library(XML)

ubase = "http://www.cherryblossom.org/"
url = paste(ubase, "results/2012/2012cucb10m-m.htm", sep = "")

doc = htmlParse(url)
```

We saw from the *HTML* source that we want to extract the text content of the *<pre>* node. We can access all *<pre>* nodes in the document with the simple *XPath* expression, `//pre`. We do this with

```
preNode = getNodeSet(doc, "//pre")
```

The getNodeSet() function returns a list where each element corresponds to one of the *<pre>* nodes in the document. In our case, there is only one such node. Next, we use the xmlValue() function to extract the text content from this node as follows

```
txt = xmlValue(preNode[[1]])
```

```
<!-- saved from url=(0022)http://internet.e-mail -->
<html>
<pre>
                 Credit Union Cherry Blossom Ten Mile Run
                 Washington, DC     Sunday, April 3, 2011

               Female Official Results (Sorted By Net Time)

Place Div  /Tot   Num    Name                   Ag Hometown             5 Mile  Time    Net Tim Pace  S
===== =========== ====== ====================== == ==================== ======= ======= ======= ===== =
    1    1/2706     14 Julliah Tinega         25 Kenya                          54:02   54:02  5:25 !
    2    1/937      16 Risper Gesabwa         22 Kenya                  27:17   54:03   54:03  5:25 !
    3    1/1866     48 Tgist Tufa             30 Ethiopia               27:17   54:13   54:13  5:26 !
    4    2/1866     44 Alemtsehay Misganaw    30 Ethiopia               27:17   55:17   55:17  5:32 !
    5    2/2706     24 Claire Hallissey       28 United Kingdom         28:01   56:17   56:17  5:38 !
    6    3/1866     40 Kelly Jaske            34 Portland OR            27:58   57:06   57:06  5:43 !
    7    4/1866    156 Michelle Miller        30 Damascus MD            29:33   59:20   59:20  5:56
    8    1/1265    148 Sharon Lemberger       37 Stamford CT            29:50   59:40   59:40  5:58
    9    3/2706    167 Katie Howery           25 Verona WI              29:38   59:52   59:52  6:00
   10    2/1265    157 Kara Waters            36 Ellicott City MD             1:00:03 1:00:03  6:01
```

Figure 4.17: Screen shot of the Source for Women's 2011 Cherry Blossom Results. This screen shot is of the *HTML* source for the female results for the 2011 Cherry Blossom road race. Notice that times given are for the midpoint of the race (`5 Mile`) and for two finish times (`Time` and `Net Tim`). Also notice the leftmost column labeled `S`. While the format is different than the male results for 2012, both are plain text tables within *<pre>* nodes in an *HTML* document.

Let's examine the contents of `txt`. We first determine how many characters long it is and then examine the first few and last few of them. We do this with

```
nchar(txt)

[1] 690904

substr(txt, 1, 50)

[1] "\r\n                  Credit Union Cherry Blossom Ten "

substr(txt, 690850, 690904)

[1] "       48 Herndon VA           1:09:06 2:30:59 15:06 \r\n"
```

It appears that we have successfully extracted the information from the Web page. We also see that the lines end with `\r\n`. We can use these characters to split up the 690000+ characters into separate strings corresponding to lines in the table. That is,

```
els = strsplit(txt, "\r\n")[[1]]

els[1:3]

[1] ""
[2] "                  Credit Union Cherry Blossom Ten Mile Run"
[3] "                  Washington, DC     Sunday, April 1, 2012"

length(els)
```

```
[1] 7201

els[7201]

[1] " 7193   648/648     6555 Lee Jordan
      48 Herndon VA           1:09:06 2:30:59 15:06 "
```

We have succeeded in extracting the rows of the table as elements of a character vector.

Let's formalize our code into a function that we can apply to each of the 28 Web pages (2 pages for each year from 1999 to 2012). We want our function to take as input the URL for the Web page and return a character vector with one element per row in the table of results. We arrange our previous code into a function as

```
extractResTable =
  # Retrieve data from web site,
  # find the preformatted text,
  # return as a character vector.

  function(url)
  {
    doc = htmlParse(url)
    preNode = getNodeSet(doc, "//pre")
    txt = xmlValue(preNode[[1]])
    els = strsplit(txt, "\r\n")[[1]]

    return(els)
  }
```

Let's try out our function with the 2012 men's results.

```
m2012 = extractResTable(url)

length(m2012)

[1] 7201
```

Our function seems to have extracted the same results as before. Let's now apply it to all of the men's data.

If we have a vector of all the URLs then we can simply apply our function to the vector. We make this vector by pasting together the base URL to the year-specific information as follows:

```
ubase = "http://www.cherryblossom.org/"
urls = paste(ubase, "results/", 1999:2012, "/",
             1999:2012, "cucb10m-m.htm", sep = "")
urls[1:3]

 urls
 [1] "http://www.cherryblossom.org/results/1999/1999cucb10m-m.htm"
 [2] "http://www.cherryblossom.org/results/2000/2000cucb10m-m.htm"
 [3] "http://www.cherryblossom.org/results/2001/2001cucb10m-m.htm"
```

Now we can apply extractResTable() to `urls` with

```
menTables = lapply(urls, extractResTable)

Error in preNode[[1]] : subscript out of bounds
```

We have an error that indicates there is a problem with `preNode`.

To find out more information about what is causing this error, we turn on the error handling by setting the `error` option to recover() so that when an error occurs, the recover() function is called. This function gives us access to the active call frames so that we can examine the objects and see if they are what we expect. We set options() and call the extractResTable() again:

```
options(error = recover)
menTables = lapply(urls, extractResTable)

Error in preNode[[1]] : subscript out of bounds

Enter a frame number, or 0 to exit

1: lapply(urls, extractResTable)
2: FUN(c("http://www.cherryblossom.org/results/1999/...
3: #13: xmlValue(preNode[[1]])

Selection:
```

R offers us three locations to enter the environment of the function call. We choose the second as it is within the function call to extractResTable(). We do this with

```
Selection: 2
Called from: lapply(urls, extractResTable)
Browse[1]>
```

After selecting this frame, we use *R*'s browser capabilities to examine the objects in this environment. We find:

```
Browse[1]> objects()

[1] "doc"     "preNode" "url"

Browse[1]> url

[1] "http://www.cherryblossom.org/results/1999/1999cucb10m-m.htm"

Browse[1]> length(preNode)

[1] 0
```

It appears that there is no <*pre*> node in the 1999 race results Web page.

Let's check this out by visiting the site. When we paste the URL

```
http://www.cherryblossom.org/results/1999/1999cucb10m-m.htm
```

into the Web browser, we find that it takes us to the main page shown in Figure 4.1, not to the page we were expecting. When we use the navigation system on the main Web page to go to the 1999 men's results we see the problem. The URL is not as we expected. Instead, it is

```
http://www.cherryblossom.org/cb99m.htm
```

This is a very different format from what we created based on the 2011 and 2012 *URL*s. It tells us that we need to determine all 28 *URL*s by using the Web site's navigation system. We can do this programmatically, but here we will simply gather the *URL*s for the male results into a text file called `MenURLs.txt`. We save only the portion of the *URL* that follows the base: http://www.cherryblossom.org/. We see that these *URL*s have changed quite a bit over the years, i.e., the contents of `MenURLs.txt` is

```
more MenURLs.txt

cb99m.htm
cb003m.htm
results/2001/oof\_m.html
results/2002/oofm.htm
results/2003/CB03-M.HTM
results/2004/men.htm
results/2005/CB05-M.htm
results/2006/men.htm
results/2007/men.htm
results/2008/men.htm
results/2009/09cucb-M.htm
results/2010/2010cucb10m-m.htm
results/2011/2011cucb10m-m.htm
results/2012/2012cucb10m-m.htm
```

Let's reconstruct the `urls` vector so that it contains the proper Web addresses. If `uDir` contains the location of the file `MenURLs.txt`, then we read these names into *R* with

```
fileNames = read.csv(paste(uDir, "MenURLs.txt", sep = ""),
                     header = FALSE,
                     stringsAsFactors = FALSE)[[1]]

urls = paste(ubase, fileNames, sep = "")

urls[1:3]

[1] "http://www.cherryblossom.org/cb99m.htm"
[2] "http://www.cherryblossom.org/cb003m.htm"
[3] "http://www.cherryblossom.org/results/2001/oof_m.html"
```

Now that we have addressed the problem of the incorrect *URL*s, we again try to read the results into *R* with

```
menTables = lapply(urls, extractResTable)
```

```
Error in preNode[[1]] : subscript out of bounds

Enter a frame number, or 0 to exit

1: lapply(urls, extractResTable)
2: FUN(c("http://www.cherryblossom.org/cb99m...."
3: #13: xmlValue(preNode[[1]])

Selection:
Enter an item from the menu, or 0 to exit
```

Yet again, we have an error related to `preNode`. Let's return to frame #2 and again check the values of the objects there. we do this with

```
Selection: 2

Called from: lapply(urls, extractResTable)

Browse[1]> objects()

[1] "doc"     "preNode" "url"

Browse[1]> url

[1] "http://www.cherryblossom.org/results/2005/CB05-M.htm "
```

It appears that the first six Web pages (1999 through 2004) were processed without an error because the value of `url` in the function's environment is the 2005 *URL*. Notice that it has a blank character at the end of the *URL*.

```
urls[7]

[1] "http://www.cherryblossom.org/results/2005/CB05-M.htm "
```

Could this be causing our problem? In other words, are we going to the wrong URL? It doesn't seem likely, but let's fix it and try again.

That blank must be in the text file that contains the URLs. Rather than change the txt file, let's simply modify the character string in *R*, in case this really isn't our problem. If we find that it is the problem, then we can update `MenURLs.txt` later. To drop the last character from the string, we find out how many characters are in that URL and shorten the string by one. We do this as follows:

```
nchar(urls[7])

[1] 53

urls[7] = substr(urls[7], 1, 52)
urls[7]

[1] "http://www.cherryblossom.org/results/2005/CB05-M.htm"
```

Let's see if that fixes our problem. Rather than apply extractResTable() to all 14 URLs, we just try it on 2005,

```
test05 = extractResTable(urls[7])
```

We received no errors this time, and when we examine the first few rows of `test05` we see the header of the 2005 table:

```
head(test05)

[1] "                                                                              "
[2] "                Credit Union Cherry Blossom 10 Mile Road Race"
[3] "                              Washington, DC"
[4] "                           Sunday, April 3, 2005"
[5] "                          Official Men's Results "
```

We have identified the problem.

After correcting the *URL* in our text file, we reapply extractResTable() to the *URL*s with

```
menTables = lapply(urls, extractResTable)
names(menTables) = 1999:2012
```

At last, we made it through all of the URLs without generating an error! Of course, simply because we didn't run into any errors, does not mean that we have properly extracted the data. We need to check the results to see if they contain the information expected.

Let's first check the length of each of the character vectors. From the Web site we have seen that several thousand runners compete each year so we expect several thousand elements in our vectors.

```
sapply(menTables, length)

1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
3193    1 3627 3727 3951 4164 4335 5245 5283 5913
2009 2010 2011 2012
   1 6919 7019 7201
```

Hmmm, the 2000 and 2009 extractions resulted in one element vectors.

The file names for these two years are correct so this requires digging deeper. We view the source of the 2000 Web page to see if it is formatted as expected. Below are the first few lines of the 2000 document:

```
<html>
<body bgcolor="#CCFFFF">
<p align="center"><font color="#800000" size="4" face="Arial"><strong>
Nortel Networks Cherry Blossom 10mile Road Race<br>
Washington, DC *** April 9, 2000
</strong></font><strong><font color="#800000" face="Arial"><br>
<h3 align="center"><font color="#800000" face="Arial">
Official Results, MEN *** Gun Time Is The Official Time</font></h3>
<BR>
<PRE><Strong>
</font><font color="#800000" face="Courier New">
PLACE DIV /TOT  NUM   NAME                  AG ...
===== ========= ===== ===================== == ...
```

Let's rearrange the *HTML* tags and use indentation to see if there is a problem with the format of the document. Below is the same content displayed in a more readable format:

```
<html>
<body bgcolor="#CCFFFF">
 <p align="center">
  <font color="#800000" size="4" face="Arial">
    <strong>
Nortel Networks Cherry Blossom 10mile Road Race<br>
Washington, DC *** April 9, 2000
    </strong>
  </font>
  <strong>
    <font color="#800000" face="Arial">
      <br>
      <h3 align="center"><font color="#800000" face="Arial">
Official Results, MEN *** Gun Time Is The Official Time
        </font>
      </h3>
    <BR>
    <PRE>
      <Strong>
        </font>
        <font color="#800000" face="Courier New">
PLACE DIV /TOT  NUM   NAME                 AG ...
===== ========= ===== ==================== == ...
```

This document is not well-formed *HTML*. The htmlParse() function can fix many problems with ill-formed documents, e.g., closing a <*br*> tag and matching case for tag names. However, this function can only do so much. Notice that the <*font*> and <*h3*> tags are not properly nested, and similarly the closing </*font*> tag that appears after the <*pre*> tag is problematic. If htmlParse() closes the <*pre*> tag so that the tags in the document are properly nested, then the <*pre*> node will not contain the table of race results.

We could programmatically edit the *HTML* so that it is well formed. Alternatively, we could try another *XPath* expression for locating the content for this particular file. We proceed with the second of these options and leave the first as an exercise.

If we want to handle one of the year's differently than the others, then we will need a way to distinguish between the two approaches. One way to do this might be to add a second argument to the function definition which indicates with which year we are working. Then our code can check the year, and if it is 2000, we can extract the table of results differently. We supply a default value to *year* so that if we don't specify this argument then the function will carry out the default extraction. We provide a modified extractResTable() to do this:

```
extractResTable =
  # Retrieve data from web site,
  # find the preformatted text,
  # and return as a character vector.

  function(url, year = 1999)
  {
```

```
    doc = htmlParse(url)

    if (year == 2000) {
      # Get text from fourth font element
      # File is ill-formed so <pre> search doesn't work.
      ff = getNodeSet(doc, "//font")
      txt = xmlValue(ff[[4]])
    }
    else {
      preNode = getNodeSet(doc, "//pre")
      txt = xmlValue(preNode[[1]])
    }

    els = strsplit(txt, "\r\n")[[1]]
    return(els)
  }
```

Since we now have two arguments to our function, we use mapply() to call extractResTable():

```
years = 1999:2012
menTables = mapply(extractResTable, url = urls, year = years)
names(menTables) = years
sapply(menTables, length)

1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
3193 3019 3627 3727 3951 4164 4335 5245 5283 5913
2009 2010 2011 2012
   1 6919 7019 7201
```

We have cleared up the problem with 2000, but the problem with 2009 remains. We leave it as an exercise to modify extractResTable() to handle this special case. Once modified, we find that there are 6659 rows in the 2009 table.

Now that we have the function working for the Web pages of men's results, we can try it on the women's pages. When we do, we find that all works fine except for the year 2009. As it happens, we don't need any special handling for the women's results for that year. We leave it as an exercise to modify the function again so that the 2009 women's results use the default processing, rather than the special 2009 processing needed for the men's results.

We have two lists, one for women and one for men. We save them for further processing.

```
save(womenTables, menTables,
     file = "CBTextTables.rda")
```

Lastly, an alternative to saving the two lists of character vectors in an *R* data format is to write the character vectors out as plain text files where each element in the vector corresponds to a line in the output file, which in turn corresponds to a row in the results table. We would use writeLines() to do this. In fact, we can modify extractResTable() to accept a *file* argument. If supplied, the function would write the results to a file with that name, and if NULL then the function would return the character vector. Again, we leave this enhancement as an exercise.

4.8 Exercises

- Write a function to read the 28 text tables in `MenTxt/` and `WomenTxt/` into *R*. These are called `1999.txt`, `2000.txt`, etc. and are described in greater detail in Section 4.2. Examine the tables in a plain text editor to determine the start and end position of each column of interest (name, hometown, age, and gun and net time). Use statistics to explore the results and confirm that you have extracted the information from the correct positions in the text.
- Revise the extractVariables() function to remove the rows in `menTables` and `womenTables` that are blank. In addition, eliminate the rows that begin with a `"*"` or a `"#"`. You may find the following regular expression helpful for locating blank rows in a table

  ```
  grep("^[[:blank:]]*$", body)
  ```

  The first argument to `grep` uses several meta characters to specify the pattern to search for. The `^` is an anchor for the start of the string, the `$` anchors to the end of the string, the `[[:blank:]]` denotes the equivalence class of any blank-type character, and `*` indicates that the blank character can appear 0 or more times. All together the pattern `^[[:blank:]]*$` matches a string that contains any number of blanks from start to end. After adding this code to extractVariables(), the 61 `NA`s in 2001 should be eliminated as well as many but not all of the other `NA`s in other the years.
- Find the record where the time is only 1.5. What happened? Determine how to handle the problem and which function should be modified: extractResTable(), extractVariables(), or cleanUp(). In your modification, include code to provide a warning message about the rows that are being dropped for having a time that is too small.
- Examine the head and tail of the 2006 table. Look at both the character matrix in the list called `menResMat` and the character vector in the list called `menTables`. (Recall that the character matrix and the character vector are both called `"2006"`). What is wrong with the hometown? Examine the header closely to figure out how this error came about. Modify the extractVariables() function to fix the problem. You may want to add an additional argument to identify the year of the data to determine when the 2006 table is being processed.
- Modify the call to the plot() function that created Figure 4.4 to create Figure 4.5. To do this, read the documentation for plot() to determine which parameters would be most helpful, i.e., plot.default? contains helpful information about the commonly used graphical parameters.
- Modify the piecewise linear fit from Section 4.4.2 to include a hing at 70. Examine the coefficients from the fit and compare the fitted curve to the loess curve. Does the additional hing improve the fit? Is the piecewise linear model closer to the loess curve?
- We have see that the 1999 runners were typically older than the 2012 runners. Compare the age distribution of the runners across all 14 years of the races. Use quantile-quantile plots, boxplots, and density curves to make your comparisons. How do the distributions change over the years? Was it a gradual change?
- Normalize each runner's time by the fastest time for the runner of the same sex and age. To do this, find the fastest male runner for each year of age from 20 to 80. The tapply() function may be helpful here. Smooth these times using loess(), and find the smoothed time using predict(). Use these smoothed times to normalize each male's time. Use density plots, quantile-quantile plots, and summary statistics to compare the distribution of the age-normalized times for the runners in 1999 and 2012. What do you find. Repeat the process for the women. Compare the women in 1999 to the women in 2012 and to the men in 1999 and 2012.

- Clean the strings in `home` in `menRes` to remove all leading and trailing blanks and multiple contiguous blanks. Also make all letters lower case and remove any punctuation such as . or , or ' from the string. Assign the cleaned version of `home` into `menRes` replacing the existing `home` variable.
- In Section 4.5 we created an id for a runner by pasting together the name, year of birth and state. Consider using the home town instead of the state. What is the impact on the matching? How many runners have competed in at least 8 races using this new id? What if you reduced the number of races to 6? Should this additional restriction be used in the matching process?
- Further refine the set of athletes in the longitudinal analysis by dropping those IDs (from Section 4.5 with a large jump in time in consecutive races and who did not compete for two or more years in a row. How many unique IDs do you have when you include these additional restrictions? Does the longitudinal analysis from [?] change?
- Consider adapting a nonparametric curve fitting approach to the longitudinal analysis. Rice [**?**] suggests modeling an individual's behavior as a combination of an average curve plus an individual curve. That is the predicted performance for an individual comes from the sum of the average curve and the individual's curve: Y_i(t) = mu(t) + f_i(t) + error, where Y_i(t) is the performance of individual i at age t. They suggest a "two-step" process where

  – Take a robust average of all of the smoothed curves for the individuals.
  – Subtract this average smoothed curve from the individual data points and smooth the residuals.

  Rather than using only the individual's run times to produce the individual's curve, they also suggest smoothing over a set of nearest neighbors' times. Here a nearest neighbor is a runner with similar times for similar age.
- In Section 4.7, we discovered that the *HTML* file for the male 2000 results was so poorly formatted that htmlParse() was unable to fix it to allow us to extract the text table from the *<pre>* tag. In this exercise, we programmatically edit this *HTML* file so that we can use htmlParse() as desired. To do this, begin by reading the *HTML* file located at http://www.cherryblossom.org/cb003m.htm using readLines(). Carefully examine the *HTML* displayed in Section 4.7 and come up with a plan for correcting it. Consider whether you want to drop *<font>*s or close them properly. Once you have fixed the problem so that the *<pre>* tag contains the text table, pass your corrected *HTML* to htmlParse(). You may want to use a text connection to do this rather than writing the file to disk and reading it in.
- Revise the extractResTable() function in Section 4.7 so that it can read the male 2009 results. Carefully examine the raw *HTML* to determine how to find the information about the runners. You will want to work with *XPath* to locate *<div>* and *<pre>* tags and extract the text value. The female 2009 results do not need this special handling. Modify the extractResTable() function to accept an additional parameters: *sex*. Give the *sex* parameter a default value of `"male"`, and use it to determine whether to perform the special processing of the *<div>* and *<pre>* tags.
- Revise the extractResTable() function in Section 4.7 so that it takes an additional parameters: *file*. Give the *file* parameter a default value of `NULL`. When `NULL`, the parsed results are returned from extractResTable() as a character vector. If not `NULL`, the results are written to the file named in *file*. The writeLines() function should be helpful here.