Analyze historical data with Perl and R

Hindsight

The Perl project's Git repository contains all the commits since Larry Wall published the first version back in 1987. The R statistics tool retrieves some surprising snippets of information from this historic data and visualizes it for posterity. *By Michael Schilli*

It's a great feeling to be sitting on a plane with a US$ 200 laptop, but no Internet connection, and have the complete history of the Perl core in front of your very eyes. What files did Larry Wall check in back in 1987? Who submitted the first patch? What did it contain?

The `git log` command shown in Figure 1 immediately returns initial results and takes just a couple of seconds to flash back to the beginnings of the project, even on an underpowered netbook.

Or, you might be interested in what happened last Monday. All of this information is contained in a 120MB repository, which `git` updates with `git://perl5.git.perl.org/perl.git` effectively as soon as you have your Internet connection back. Only the uninitiated are surprised that Git points the way for traditional versioning systems, like Subversion.

Compact Information

This compact bundle of information is interesting not only for programmers wanting to follow the development of the project. State-of-the-art statistics tools can extract and visualize trends from it, but even with shell tools, you can discover that precisely 38,206 commits have occurred since 1987:

```
$ git log --oneline | wc -l
38206
```

The `--oneline` option reduces the output format to a single line per commit. This is not much use for in-

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.

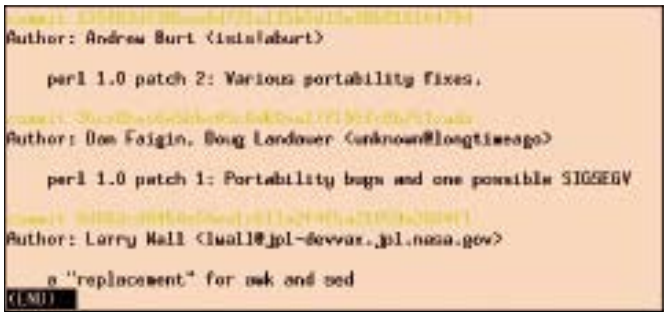

Figure 1: The Perl project's Git repository contains all the commits since Larry Wall published the initial version back in 1987.

Figure 2: With additional options, `git-log` gives more detailed information in a parser-friendly format.

depth analysis because it omits valuable information such as modified files, the precise commit date, the author of a patch, or the email address of the committer. In contrast, the call

```
git log --name-status --date=raw \
        --pretty= \
          'format:commit,%ae,%at,%ce' \
          >perl-git-log.txt
```

shows more detailed information, formatted in an easy-to-parse format, as shown in Figure 2.

Each commit in this format contains one or multiple files that Git lists, line-by-line, below the commit header with a change flag (M = modified, A = added, R = removed). Headers start with the text string `commit, ...`, so the parser I will be building later on can easily distinguish them from data lines.

After the command has completed, the `perl-git-log.txt` file has a bunch of interesting data. Listing 1 [1] picks up the data and converts them into CSV (comma-separated values) format, which the statistics tool R supports as input.

Perl as Transformer

Even in this Perl column, I have to turn to other languages from time to time because I like using the best tool for the job at hand. The R language [2] is the king of the hill in the statistics field [3], with its speed-optimized data transformations, a huge selection of graphics libraries, and a CPAN-style developer network that goes by the name of CRAN. Scripts written in R are amazingly compact, although it can take a while for newcomers to understand the new paradigms and data structures. In contrast, Perl is a master of data format conversions, which is why I am using it (log2csv, Listing 1) as a feeder that converts the Git repository logfiles to the comma-separated entries shown in Figure 3.

The script `log2csv` looks through the `perl-git-log.txt.bz2` logfile, which I previously compressed from 5MB to 0.5MB, line by line. If line 40 of the script discovers a commit header, it stores its details such as the patch author, the Unix timestamp, and the email address of the executing committer in three variables declared outside of the `while` loop.

If line 45 then discovers a line with a file change (e.g., `M filename`), it uses a regular expression to cut off the change marker and stores the name of the modified file in `$file`. The CPAN Text::CSV module's `print` method in line 47 then appends and prints out the variables to `perl-git-log.csv`.

LISTING 1: log2csv

```
01 #!/usr/local/bin/perl -w
02 ############################
03 # log2csv - Convert git logs
04 #           to CSV format
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 ############################
08 use strict;
09 use local::lib;
10 use Text::CSV;
11
12 my $logfile =
13   "perl-git-log.txt.bz2";
14 my $csvfile =
15   "perl-git-log.csv";
16
17 my $csv = Text::CSV_XS->new(
18  { binary => 1, eol => $/ })
19   or die "Cannot use CSV: ",
20   Text::CSV->error_diag();
21
22 open my $logfh,
23   "bzip2 -dc $logfile |"
24   or die "$logfile: $!";
25 open my $csvfh, ">$csvfile"
26   or die "$csvfile: $!";
27
28 my ( $dummy, $author,
29  $time,  $committer );
30
31 $csv->print(
32  $csvfh,
33  [
34   "time",   "file",
35   "author", "committer"
36  ]
37 );
38
39 while (<$logfh>) {
40  if (/^commit/) {
41   chomp;
42   ( $dummy, $author,
43    $time,  $committer
44   ) = split /,/, $_;
45  } elsif (/^(\w)\s+(.*)/) {
46   my $file = $2;
47   $csv->print(
48    $csvfh,
49    [
50     $time,   $file,
51     $author, $committer
52    ]
53     )
54     or die "print failed: ",
55     Text::CSV->error_diag();
56  }
57 }
58
59 close $logfh
60   or die "$logfile: $!";
61 close $csvfh
62   or die "$csvfile: $!";
```

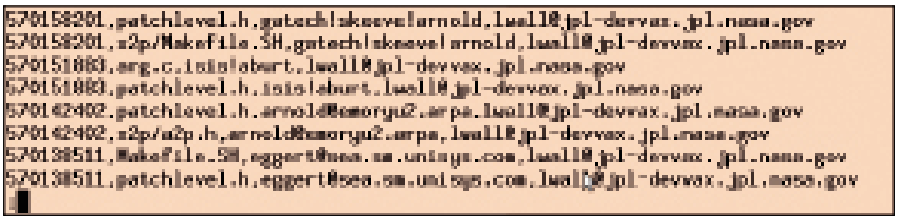

Figure 3: The resulting CSV file splits commits into separate files and provides the raw material for statistical analysis in R.

This creates a new line in the output file that includes the author, committer, and timestamp fields for each modified file in the repository. The CPAN Text::CSV (or the speed-optimized Text::CSV_XS version) masks any special characters that occur in the line and puts strings containing blanks in double quotes to comply with the requirements of the comma-separated output format.

The call to the constructor in line 17 sets the "binary" flag to allow a wide range of characters. The `eol` option defines the line separator in the output format and is assigned a value of `$/` – that is, the line break character configured for the active Perl installation.

Taking R for a Test Drive

The 8.5MB CSV file can be parsed and processed with the R statistics language and interpreter once you have installed it following the commands that are described in the "Installation" box. The trial run with R shown in Figure 4 illustrates how the tool starts up at the Unix command line (the interpreter's name really is *R*).

Following some introductory information, such as the version number, the interpreter then stops and waits for user input at the `>` prompt.

INSTALLATION

Ubuntu installs the R interpreter with the command

```
sudo apt-get install r-base-core
```

and the Perl modules you need to prepare the data are also available as `libtext-csv-perl` and `libtext-csv-xs-perl`. I hope you enjoy exploring treasure troves of information [4].

The obligatory `print("hello r")` calls R's print function, which hands the string passed to it to standard output. Note that R insists on parentheses for function calls and doesn't want a semicolon at the end of a command. The `print` output consists of a single line, which R precedes with `[1]`. This confirms the nasty suspicion that arrays, or "vectors" as they're called in R-speak, start at index 1 and not at 0. A command of `source("testprog.r")` in the R interpreter runs another R program, as I'll show you in one of the scripts later in this article. As you can see in Figure 4, R can't find the script specified.

> "R insists on parentheses for function calls and doesn't want a semicolon at the end of a command."

Alternatively, the R distribution includes an `Rscript` program you can add to the header of an executable script, as in `#!/usr/bin/Rscript`, to tell the kernel to call the R interpreter automatically when the script launches and pass the program code to it for execution. The `q()` function quits an interactive interpreter session. R then asks whether you want to dump the current interpreter status to disk. If yes, R writes the values of all known variables to the `.RData` directory. After relaunching, R restores these data so you can carry on exactly where you left off.

R the Data Juggler

The `read.csv()` command in Figure 5 parses the recently created CSV file with the git repository data into R. Note that dots in function or variable names in R simply serve to improve readability and don't have any deeper syntactical significance. If successful, the function returns a data structure of the R-specific type "Dataframe" that looks like a database table; the columns provide the internal structure and each line is a record. The assignment of the data structure to the `commits` variable is handled by the operator `<-`, which R purists prefer to the functionally identical `=` to avoid confusion with comparisons (`==`).

After storing the complete data from 23 years of Perl development in the `commits` variable, which takes a couple of seconds, a call to `head(commits)` in Figure 5 shows the first six rows of data. The corresponding `tail(commits)` call would produce the last few lines instead. To convert the `files` column in the dataframe to a separate vector, the first command in Figure 6 assigns the expression `commits$file` to the `files` variable, which contains precisely 139,442 file names, as the call to `length(files)` shows. This vector includes a large number of duplicate entries, and the `levels()` function extracts the unique values. A call to `length()` shows that 16,429 different files were created, modified, or deleted since Perl development began.

Frequency Counter

The R `table()` function accepts a vector and returns a data structure in which it assigns a counter to each unique element, holding the number of times the element occurs in the vector:

```
> data=c("one", "two", "three",
         "two", "one", "two")
```

Figure 4: R on trial: print command; executing the `testprog.r` program; exiting R.

Figure 5: In R, you can easily parse the `.csv` file and convert it into a data structure.

```
> table(data)
data
  one three   two
    2     1     3
```

The code snippet above also shows how R uses the `c()` function (for "concatenate") to create a vector from individual elements. The second command line in Figure 6 puts together everything learned here in a single line and shows which files in the repository were changed the most frequently. To do so, `table(files)` classifies the files listed in the commits and creates a counter for each to accumulate the number of occurrences. The `sort()` function then sorts the `table()` counters in ascending order and `tail`, with an option of `n = 20L`, returns the last 20 entries – that is, the entries with the highest counters. As is the wont of the interactive R interpreter, the results are neatly structured if the return value is not assigned to a variable. R also has a useful help function; when you type a question mark followed by a function name, as in `?tail`), it pulls up neatly written manual pages describing the function.

LISTING 2: file-plot.r

```
01 #!/usr/bin/Rscript
02 ##############################
03 # file-plot.r
04 # 2010, Mike Schilli
05 #       <m@perlmeister.com>
06 ##############################
07 commits <- read.csv(
08         "perl-git-log.csv")
09 files <- commits$file
10 data=tail(sort(table(files)),
11           n = 20L)
12 data=rev(data)
13 png(file="files.png")
14 plot(data, type="h",
15   main="File Commits in Perl Git
           Repo",
16   xlab="Most modified files",
17   ylab="Number of commits per
           file")
```

Pictures Say More than Words

This short introduction to R should suffice to start plotting some interesting charts that shed light on the activities within the Perl repository. Listing 2 creates a PNG image file from the data structure with the 20 most frequently modified files. Line 13 prepares the output of the following plot function, as `png(file="files.png")`, which redirects all drawing activities to the file `files.png`. Without this line, R would pop up a new window and display the chart there. The `plot` function call from the standard R repertoire in lines 14-17 illustrates that R doesn't need a confusingly long list of parameters to paint a professional-looking graph. Axis labels, maximum and minimum values, you name it – everything has meaningful defaults. But this doesn't mean that R is inflexible. On the contrary, you can modify every single detail of a chart from the shape, position and number of axes, number of values in the scale, colors, fonts, and so on to match your own taste.

R functions expect parameters as comma-separated `par=value` pairs. Figure 7 displays what the plot of the dataframe `data` in a histogram format looks like.

The Ravages of Time

To display the activity from the past 23 years, Listing 3 takes the `commits$time` timestamp column, which uses the Unix seconds format, and adds a new column, `commits$year`, which displays the corresponding four-digit year, to the dataframe. For this to happen, the internal R function `as.POSIXlt()` converts the Unix timestamp, by reference to the date `1970-01-01`, to the native `POSIXlt` (POSIX "local time") date type.

The `format()` function then uses the `"%Y"` placeholder to extract the four-digit year from the date, which results in a new column with 140,000 year numbers,

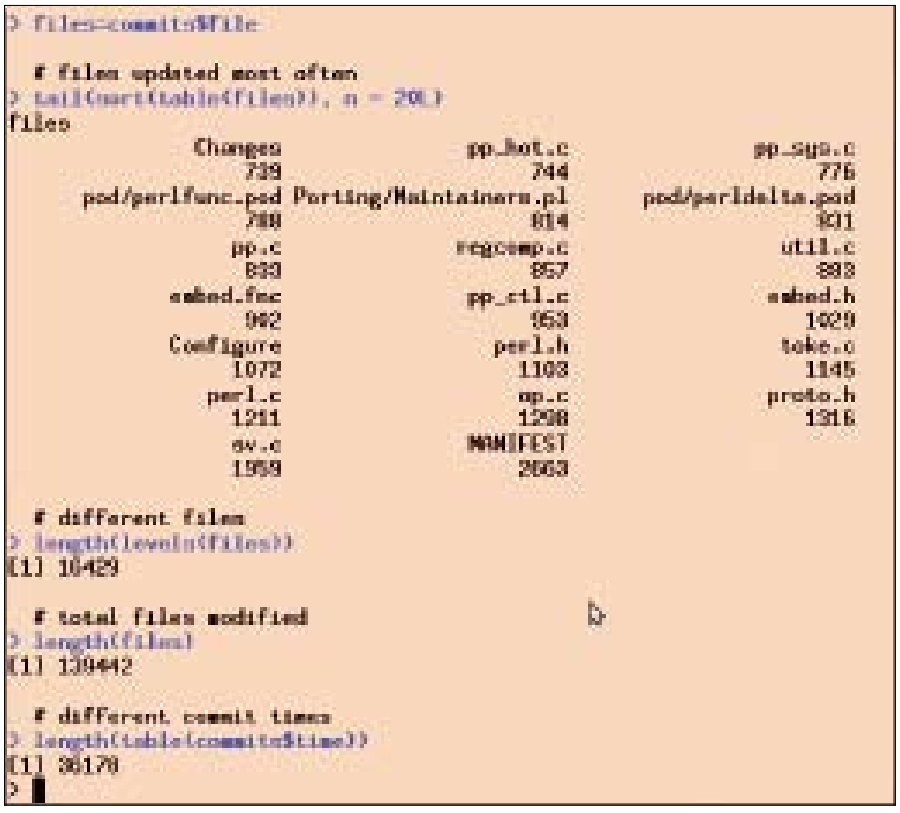

Figure 6: First steps with R and the CSV data imported from the Perl Git repository log.

Mike: I changed the first sentence of this section (pls refer to your original) because it called out a Fig 10 (before Fig 9) that didn't appear to agree with the figure supplied, and it talked about the "10 hardest working authors" rather than the five, as later. OK as set?? -rls

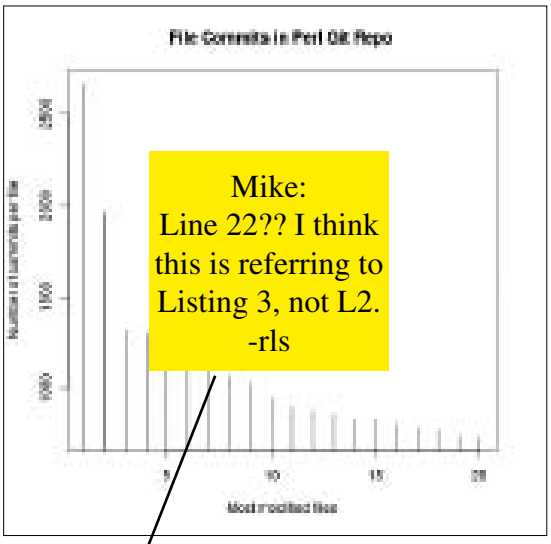

Figure 7: The most frequently modified files as a chart.

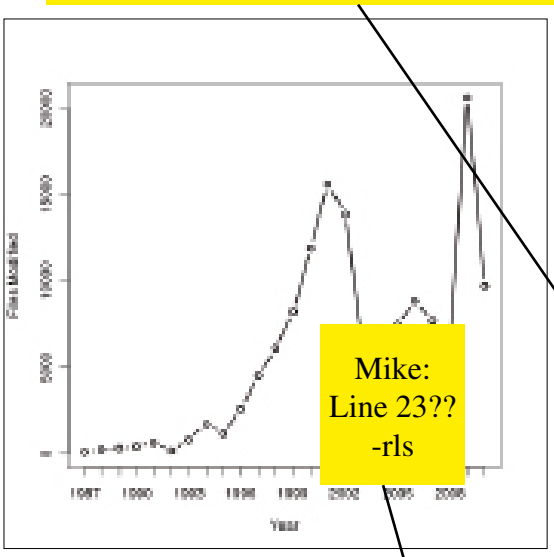

Figure 8: Number of files modified per year.

which the `table()` function then converts into a data structure that contains a counter for each year.

The `plot()` function called in line 14 is smart enough to convert the data structure into the chart shown in Figure 8 without further instructions from the developer. Listing 3 simply specifies the axis labels `"Year"` and `"Files Modified"`.

If you look closely, you will notice that the lines in the graph do not cross through the data point circles, but fade out shortly before and restart shortly after them. This feature comes courtesy of the `type="b"` option; if you prefer uninterrupted lines, use `"o"` instead. For more options, check out the man page that appears when you type `?plot` in the R interpreter. This page also tells you that `plot()` doesn't just process `table()` output but can also work with two vectors for the *x* and *y* values of the graph. R generally tries to guess what the user means and very often takes a huge amount of work off the user's plate.

The man pages for the lattice graphics library reveals useful details of the amazing number of parameters the xyplot() function supports.

A Detailed Picture Tells Even More

The R code in Listing 4 grabs from the Perl repository the hardest working Perl authors in the last 23 years, along with their activities. Because Git didn't exist in 1987, the data were imported retrospectively from the versioning system used before its introduction. The R code that creates the multigraph diagram first looks for the hardest working authors and only remembers those who have produced more than 5,000 file commits total. The `subset()` function filter out all other commits in line 14 by calling `subset(au,au > 5000)`.

The `au table`-type variable carries all authors of all commits. The condition passed in as the second parameter filters out any entries that don't match. One of R's specialties is vector operations [5], such as `au > 5000`, which are short and to the point and also capable of performing mass operations in a highly efficient way.

Line 27 calls `table()` with two parameters, `commits$author` and `commits$year`, and thus creates a data structure that assigns all combinations of author and year to a counter. For ease of plotting, line 30 uses R utility function `as.data.frame()` to create a dataframe from the table, and line 32 then names its previ-

LISTING 3: files-per-year.r

```
01 #!/usr/bin/Rscript
02 ############################
03 # files-per-year.r
04 # 2010, Mike Schilli
05 #       <m@perlmeister.com>
06 ############################
07 commits <- read.csv(
08           "perl-git-log.csv")
09
10 commits$year <- format(
11   as.POSIXlt(
12     commits$time,
13     origin="1970-01-01"),
14   "%Y"
15 )
16
17 files.per.year <-
18         table( commits$year )
19
20 png(file=
21     "files-per-year.png")
22 plot( files.per.year,
23   xlab="Year",
24   ylab="Files Modified",
25   type = "b" )
```

```
> tail (files.by.auth.year)
                        author year files
25200 rgarciasuarez@gmail.com 2009  1068
25808           gsar@cpan.org 2010     0
25908             ju@iki.fi 2010     0
26170          nick@ccl4.org 2010  1437
26177       nik@tiuk.ti.com 2010     0
26306 rgarciasuarez@gmail.com 2010     0
>
```

Figure 9: The tail of the dataframe shortly before plotting.

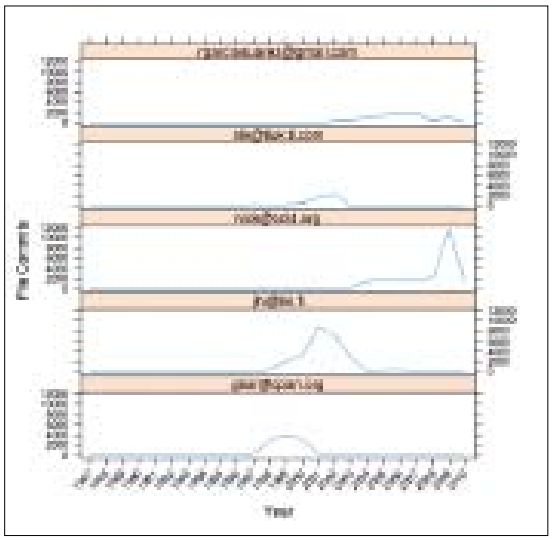

Figure 10: Committers with more than 5,000 file commits and their most active years.

ously unnamed columns with a somewhat surprising left-hand-side call to `names()` on the dataframe, which gets the column names `"author"`, `"year"`, and `"files"` assigned from the right side of the assignment.

At this point in time, the variable `files.by.auth.year` still contains the data of all authors, but line 36 extracts the subset of the five hardest working authors previously identified by `au` before then assigning the results back to `files.by.auth.year`. Figure 9 shows the tail of the intermediate results.

The more complex chart is not plotted by `plot()`, but by the `xyplot()` function from the `lattice` graphics library, which was included previously in line 9 with the command `library("lattice")`. Incidentally, this also imports the man pages for the library, which is included with the R distribution. Typing `?xyplot` reveals hugely useful details of the amazing number of parameters this function supports.

The most important parameter is the first, which takes the form

```
y ~ x | g
```

where *x* and *y* are vectors with the *x* and *y* values, respectively, and *g* defines the various groups for which you want a separate graph. In this case, all three variables reference different columns of the `files.by.auth.year` dataframe, which is passed in as the `data` parameter to `xyplot()`. The layout of the graph, which is defined with `c(1,5)` to the `layout` parameter, lets R draw five diagrams per display with one each per row.

Because the year labels are closely packed at the bottom end of the chart and would interfere with each other because of their length, the `scales` parameter rotates them 45 degrees in line 49. The `type=l` parameter defines the line type for the charts, and `xlab` and `ylab` specify the axis legends.

Each panel in Figure 10 represents one of the five hardest working authors. The graphs show the number of files modified by the committer each year, clearly revealing the active development times of legendary Perl authors, such as Gurusamy Sarathy, who have since retired from core development. ■■■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[2] The R Project for Statistical Computing: *http://www.r-project.org/*

[3] Jones, Owen, Robert Maillardet, and Andrew Robinson. *Introduction to Scientific Programming and Simulation Using R*. Chapman and Hall/CRC, 2009

[4] Talk: Fun with numbers: R and Perl (and IMDB data): *http://blog.moertel.com/articles/2007/06/21/talk-fun-with-numbers-r-and-perl-and-imdb-data*

[5] Sarkar, Deepayan. *Lattice: Multivariate Data Visualization with R*, Use R! series. Springer, 2008

LISTING 4: author-by-year.r

```
01 #!/usr/bin/Rscript
02 ##############################
03 # author-by-year.r
04 # Lattice multivariate data
05 # visualization
06 # 2010, Mike Schilli
07 #       <m@perlmeister.com>
08 ##############################
09 library("lattice")
10
11 commits <- read.csv(
12          "perl-git-log.csv")
13
14 commits$year <- format(
15   as.POSIXlt(
16     commits$time,
17     origin="1970-01-01"),
18   "%Y"
19 )
20   # Authors with more than
21   # 5000 # file commits
22 au=table(commits$author)
23 au = sort(subset(au,
24                  au > 5000 ))
25
26 files.by.auth.year =
27       table( commits$author,
28              commits$year )
29 files.by.auth.year =
30       as.data.frame(
31          files.by.auth.year )
32 names( files.by.auth.year ) =
33       c("author", "year",
34         "files")
35
36 files.by.auth.year = subset(
37   files.by.auth.year,
38   files.by.auth.year$author
39     %in% names(au)
40 )
41
42 png(file=
43     "authors-by-year.png")
44 xyplot(
45   files ~ year | author,
46   data = files.by.auth.year,
47   layout = c(1, 5),
48   scales =
49     list(x = list(rot = 45)),
50   type = "l",
51   xlab = "Year",
52   ylab = "File Commits",
53   title =
54 "Authors with > 5000 Commits"
55 )
```