A Computational Analysis of the Dynamics of R Style Based on 108 Million Lines of Code from All CRAN Packages in the Past 21 Years

by Chia-Yi Yen, Mia Huai-Wen Chang, Chung-hong Chan

Abstract The flexibility of R and the diversity of the R community leads to a large number of programming styles applied in R packages. We have analyzed 108 million lines of R code from CRAN and quantified the evolution in popularity of 12 style-elements from 1998 to 2019. We attribute 3 main factors that drive changes in programming style: effect of style-guides, effect of introducing new features, and effect of editors. A consensus in programming style is forming and we have summarized it into a style which contains all of the most popular style elements, e.g. snake case, use <- for assignment etc. [1]

Introduction

R is flexible. For example, one can use <- or = as assignment operators. The following two functions can both be correctly evaluated.

```
sum_of_square <- function(x) {
    return(sum(x^2))
}

sum_oF.square=function(x)
{
    sum(x ^ 2)}
```

One area that can highlight this flexibility is naming conventions. According to the previous research by Bååth (2012), there are at least 6 styles and none of the 6 has dominated the scene. Beyond naming conventions investigated by Baath, there are style-elements that R programmers have the freedom to adopt, e.g. whether or not to add spaces around infix operators, use double quotation marks or single quotation marks to denote strings, etc. On one hand, these variations provide programmers with freedom. On the other hand, these variations can confuse new programmers and can have dire effects on program comprehension. Also, incompatibility between programming styles might also affect reusability, maintainability (Elish and Offutt, 2002), and open source collaboration (Wang and Hahn, 2017).

Various efforts to standardize the programming style, e.g. Google's R Style Guide (Google, 2019), the Tidyverse Style Guide (Wickham, 2017), Bioconductor Coding Style (Bioconductor, 2015) etc., are available (Table 1) [2].

Among the 3 style-guides, the major differences are the suggested naming convention and indentation. Other style-elements are essentially the same. These style guides are based on possible improvement in code quality, e.g. style-elements that improve program comprehension (Oman and Cook, 1991). However, we argue that one should first study the current situation, and preferably, the historical development, of programming style variations (PSV) to supplement these standardization efforts. We have undertaken such a task, so that the larger R communities can have a baseline to evaluate the effectiveness of those standardization efforts. Also, we can have a better understanding of the factors driving increase and decrease in PSV historically, such that more effective standardization efforts can be formulated.

[1] We have presented a previous version of this paper as a poster at UseR! 2019 Toulouse. Interested readers can access it with the following link: https://github.com/chainsawriot/rstyle/blob/master/Poster_useR2019_Toulouse.png

[2] Bååth (2012) lists also Colin Gillespie's R style guide. Additional style guides that we found include the style guides by Henrik Bengtsson, Jean Fan, Iegor Rudnytskyi, Roman Pahl, Paul E. Johnson, Joshua Halls, Datanovia, and daqana. We focus only on the 3 style guides of Tidyverse, Google and Bioconductor is because these 3 are arguably most influential. There are groups of developers (e.g. contributers to **tidyverse**, Google employees, and Bioconductor contributors) adhering to these 3 styles.

Table 1: Three major style-guides: Google, Tidyverse and Bioconductor

Feature	Google	Tidyverse	Bioconductor
Function name	UpperCamel	snake_case	lowerCamel
Assignment	Discourage =	Discourage =	Discourage =
Line length	"limit your code to 80 characters per line"	"limit your code to 80 characters per line"	$\leqslant 80$
Space after a comma	Yes	Yes	Yes
Space around infix operators	Yes	Yes	Yes
Indentation	2 spaces	2 spaces	4 spaces
Integer	Not specified	Not specified (Integers are not explicitly typed in included code examples)	Not specified
Quotes	Double	Double	Not specified
Boolean values	Use TRUE / FALSE	Use TRUE / FALSE	Not specified
Terminate a line with a semicolon	No	No	Not specified
Curly braces	{ same line, then a newline, } on its own line	{ same line, then a newline, } on its own line	Not specified

Analysis

Data Source

On July 1, 2020, we cloned a local mirror of CRAN using the rsync method suggested in the CRAN Mirror HOWTO (CRAN, 2019). Our local mirror contains all packages as tarball files (.tar.gz). By all packages, we mean packages actively listed online on the CRAN website as well as orphaned and archived packages. In this analysis, we include all active, orphaned and archived packages.

In order to facilitate the analysis, we have developed the package `baaugwo` (Chan, 2020) to extract all R source code and metadata from these tarballs. In this study, only the source code from the `/R` directory of each tarball file is included. We have also archived the metadata from the `DESCRIPTION` and `NAMESPACE` files from the tarballs.

In order to cancel out the over-representation effect of multiple submissions in a year by a particular package, we have applied the *"one-submission-per-year"* rule to randomly selected only one submission from a year for each package. Unless explicitly notice, we present below the analysis of this *"one-submission-per-year"* sample. Similarly, unless explicitly notice, the unit of the analysis is *exported function*. The study period for this study is from 1998 to 2019.

Quantification of PSV

Every exported function in our sample are parsed into a parse tree using the parser from the **lintr** (Hester and Angly, 2019) package.

These parse trees were then filtered for lines with function definition and then linted them using the linters from the lintr package to detect for various style-elements. Style-elements considered in this study are:

fx_assign Use = as assignment operators

```
softplusFunc = function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L, value, value * 0.01))
    }
```

```
    return(log(1L + exp(value)))
}
```

fx_opencurly An open curly is on its own line

```
softplusFunc <- function(value, leaky = FALSE)
{
    if (leaky)
    {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L, value, value * 0.01))
    }
    return(log(1L + exp(value)))
}
```

fx_infix No spaces are added around infix operators.

```
softplusFunc<-function(value, leaky=FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value>0L, value, value*0.01))
    }
    return(log(1L+exp(value)))
}
```

fx_integer Not explicitly type integers

```
softplusFunc <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0, value, value * 0.01))
    }
    return(log(1 + exp(value)))
}
```

fx_singleq Use single quotation marks for strings

```
softplusFunc <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings('using leaky RELU!')
        return(ifelse(value > 0L, value, value * 0.01))
    }
    return(log(1L + exp(value)))
}
```

fx_commas No space is added after commas

```
softplusFunc <- function(value,leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L,value,value * 0.01))
    }
    return(log(1L + exp(value)))
}
```

fx_semi Use semicolons to terminate lines

```
softplusFunc <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!");
        return(ifelse(value > 0L, value, value * 0.01));
    }
```

```
    return(log(1L + exp(value)));
}
```

fx_t_f Use T/F instead of TRUE / FALSE

```
softplusFunc <- function(value, leaky = F) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L, value, value * 0.01))
    }
    return(log(1L + exp(value)))
}
```

fx_closecurly An close curly is not on its own line.

```
softplusFunc <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L, value, value * 0.01)) }
    return(log(1L + exp(value))) }
```

fx_tab Use tab to indent

```
softplusFunc <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0L, value, value * 0.01))
    }
    return(log(1L + exp(value)))
}
```

We have studied also the naming conventions of all included functions. Using the similar technique of Bååth (2012), we classified function names into the following 7 categories:

- **alllower** softplusfunc
- **ALLUPPER** SOFTPLUSFUNC
- **UpperCamel** SoftPlusFunc
- **lowerCamel** softPlusFunc
- **lower_snake** soft_plus_func
- **dotted.func** soft.plus.func
- **other** sOfTPluSfunc

The last style-element is line-length. For each R file, we counted the distribution of line-length. In this analysis, the unit of analysis is line.

If not considering line-length, the remaining 10 binary and one multinomial leave 7,168 possible combinations of PSVs that a programmer could employ ($7 \times 2^{10} = 7,168$).

Community-specific variations

On top of the overall patterns based on the analysis of all functions, the community-specific variations are also studied. In this part of the study, we ask the question: do local patterns of PSV exist in various programming communities? To this end, we constructed a dependency graph of CRAN packages by defining a package as a node and an import/suggest relationship as a directed edge. Communities in this dependency graph were extracted using the Walktrap Community Detection Algorithm (Pons and Latapy, 2005) provided by the **igraph** package (Csardi and Nepusz, 2006). The step parameter was set at 4 for this analysis. Notably, we analyzed the dependency graph as a snapshot, which is built based on the latest submission of each package in or before 2019.

We selected the largest 20 communities for further analysis. The purpose of this analysis is to show the PSV across communities. The choice of 20 is deemed enough to show these community-specific variations. Readers could explore other choices themselves using our openly shared data.

Results

We studied more than 94 million lines of code from 15,530 unique packages. In total, 1,898,142 exported functions were studied. Figure 1 displays the popularity of the 10 binary style-elements from 1998 to 2008. Some style-elements have very clear trends towards a majority-vs-minority pattern, e.g. fx_closecurly, fx_semi, fx_t_f and fx_tab. Some styles-elements are instead trending towards a divergence from a previous majority-vs-minority pattern, e.g. fx_assign, fx_commas, fx_infix, fx_integer, fx_opencurly and fx_singleq. There are two style-elements that deserve special scrutiny. Firstly, the variation in fx_assign is a clear example illustrating the effect of introducing a new language element by the R Development Core Team. The introduction of the language feature (= as assignment operator) in R 1.4 (Chambers, 2001) has coincided with the taking off in popularity of such style-element since 2001. Up to now, around 20% of exported functions use such style.

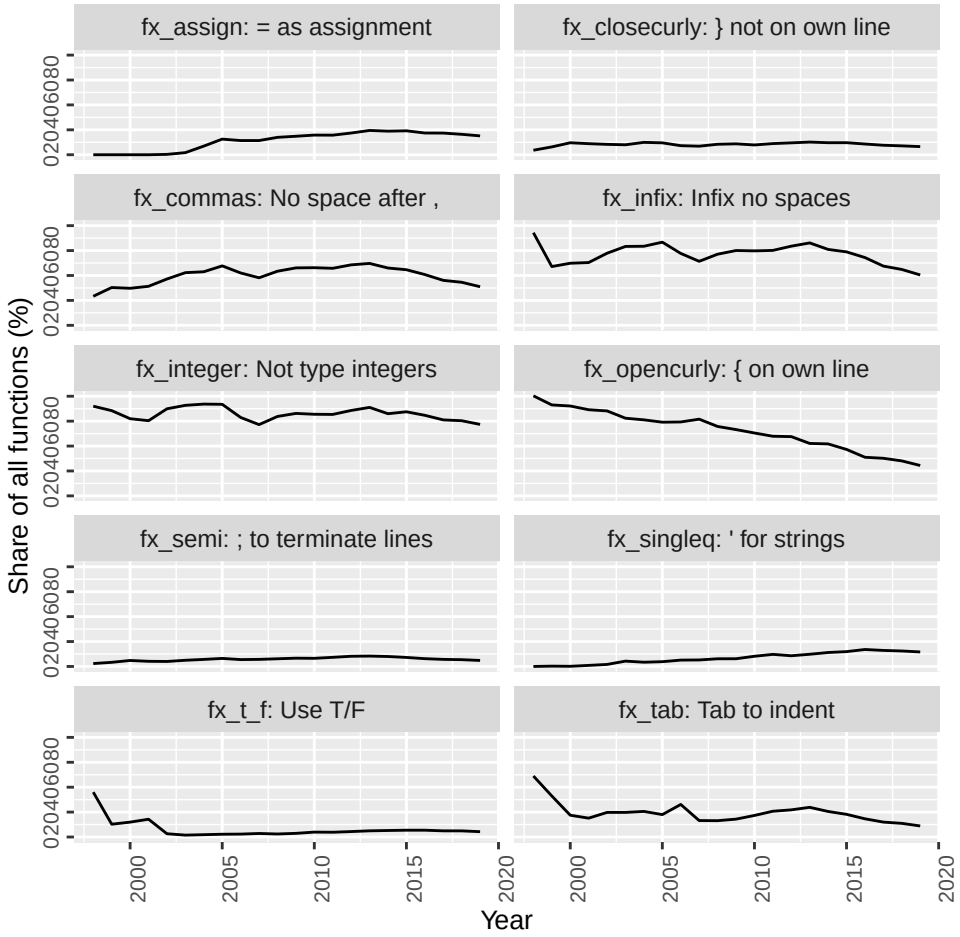

Figure 1: Evolution in popularity of 10 style-elements from 1998 to 2019.

Secondly, the popularity of fx_opencurly shows how a previously established majority style (80% in late 90s) slowly reduced into a minority but still very prominent style (30% in late 10s).

Similarly, the evolution of different naming conventions is shown in Figure 2 [3]. This analysis can best be used to illustrate the effect of style-guides. According to Bååth (2012), dotted.func style is very specific to R programming. This style is the most dominant style in the early days of CRAN. However, multiple style guides advise against the use of dotted.func style and thus a significant declining trend is observed. lower_snake and UpperCamel are the styles endorsed by the Tidyverse Style Guide and the Google's R Style Guide, respectively. These two styles see an increasing trend since the 2010s,

[3] 'Other' is the 4th most popular naming convention. Some examples of function names classified as 'other' are: *Blanc-Sablon*, *returnMessage.maxim*, *table_articles_byAuth*, *mktTime.market*, *smoothed_EM*, *plot.Sncf2D*, *as.igraph.EPOCG*, *TimeMap.new*, *fT.KB*, *IDA_stable*. These functions were classified as 'other' because of the placement of capital letters. For packages using an all capitals object class name (e.g. EPOCG) and S3 generic method names (e.g. as.igraph), their methods are likely to be classified as 'others'. One could also classify these functions as dotted.func. However, we follow both **lintr** and Bååth (2012) to classify a function as dotted.func only when no capital letter is used in its name.

while the growth of lower_snake is stronger. In 2019, lower_snake (a style endorsed by Tidyverse) is the most popular style (26.6%). lowerCamel case, a style endorsed by Bioconductor, is currently the second most popular naming convention (21.3% in 2019). Only 7.0% of functions use UpperCamel, the style endorsed by Google.

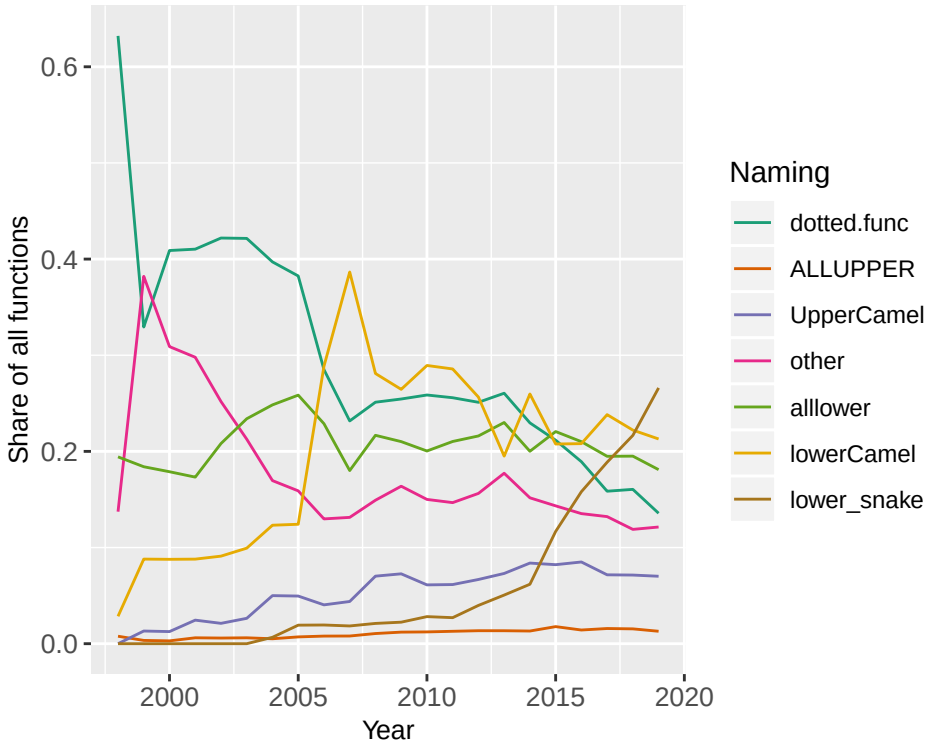

Figure 2: Evolution in popularity of 7 naming conventions from 1998 to 2019.

The evolution of line lengths is tricky to be visualized on a 2-D surface. We have prepared an animation (https://github.com/chainsawriot/rstyle/blob/master/file60f331694ef5.gif) to visualize the change in line distribution over the span of 20 years. In this paper, Figure 3 shows the snapshot of the change in line length distribution in the range of 40 to 100 characters. In general, developers of newer packages write with a lesser number of characters per line. Similar to previous analyses with Python programs (e.g. Vanderplas (2017)), artificial peaks corresponding to recommendations from either style-guides, linters, and editor settings are also observed in our analysis. In 2019, the artificial peak of 80 characters (recommended by most of the style-guides and linters such as **lintr**) is more pronounced for lines with comments but not those with actual code.

Within-package variations

Therefore, 11 binary indicators will be assigned to each function within a package, which refers to style variation it adopts. Second, we compute the entropy of these style indicators among all the functions within a package, and therefore we can analyze the consistency of each style element within a package. Notably, we normalize the entropy of style elements to be [0, 1], such that we can compare the entropy across style elements fairly. Finally, we calculate the average entropy of all CRAN packages by aggregating the package-level entropy we computed in the previous stage.

The result shows that consistency is still an issue for many style elements within a package. For example, style elements like fx_integer, fx_commas, fx_infix, fx_opencurly, and fx_name have rather larger variations, compared to fx_tab, fx_semi, fx_t_f, fx_closecurly, fx_singleq, fx_assign, as shown in the below figure.

This finding echoes our concerns that these inconsistent style variations at package level may hinder developers collaborating in a more efficient way. We provide a more detailed example in the naming convention (fx_name) for a better understanding of the within-package style inconsistency, as shown in the below figure.

We list the most important packages identified by their pagerank in the CRAN dependency graph. As what we expect, lower_snake or lowerCamel is the dominating naming convention in many

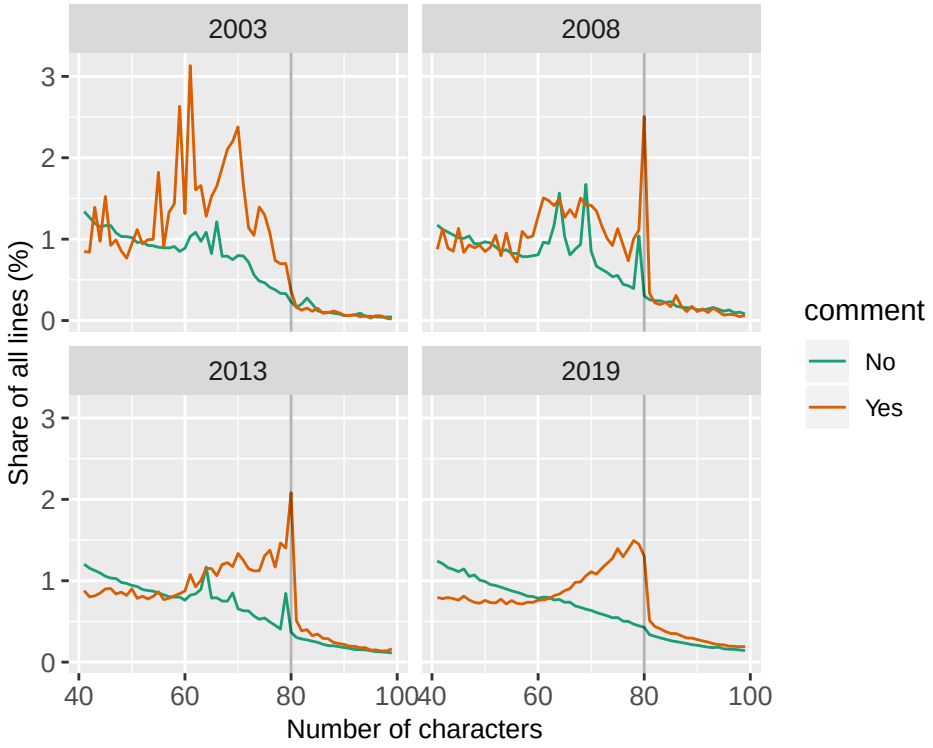

Figure 3: Change in line length distribution: 2003, 2008, 2013 and 2019.

packages. However, for many packages like digest, Rcpp, and survival, there is a lack of consensus on the function names, which may make it harder for developers to read the codes in a more efficient way.

Community-based variations

Using the aforementioned community detection algorithm of the dependency graph, 20 large communities were extracted. These communities are named by their applications. Table 2 lists the details of these communities.

Using the naming convention as an example, there are local patterns in PSV (Figure 4). For example, lower_snake case is the most popular naming convention in the "RStudio-related" community as expected because it is the naming convention endorsed by the Tidyverse Style-guide. However, none of the functions exported by the packages from "Time, Date, and Money" community uses such convention.

For the binary style-elements, local patterns are also observed (Figure 5). The most salient pattern is the "Java" and "Sparse Matrix" communities exceptional high usage of tab indentation, probably due to influences from Java or Matlab. Also, the high level in usage of open curly on its own line for the "Graphics" is also exceptional.

Discussion

In this study, we study the PSV in 21 years of CRAN packages across two dimensions: 1) temporal dimension: the longitudinal changes in popularity of various style-elements over 21 years, and 2) cross-sectional dimension: the variations among communities of the latest snapshot of all packages. From our analysis, we identify three factors that possibly drive PSV: effect of style-guides (trending of naming conventions endorsed by Wickham (2017) and Google (2019)), the effect of introducing a new language feature (trending of = usage as assignments after 2001), and effect of editors (the dominance of 80-character line limit).

From a policy recommendation standpoint, our study provides important insight for the R Development Core Team and other stakeholders to improve the current situation of PSV in R. Firstly, the introduction of a new language can have a very long-lasting effect on PSV. "Assignments with the = operator" is a feature that introduced by the R Development Core Team to “increase compatibility

Table 2: The largest 20 communities and their top 3 packages according to eigenvector centrality

Community	Number of Packages	Top 3 Packages
RStudio-related	3426	knitr, testthat, rmarkdown
base	2618	methods, graphics, lattice
Image Plotting	2228	png, rgl, highr
RCpp	677	Rcpp, inline, pkgKitten
GPS and Geography	530	deldir, sp, maptools
Machine learning	319	rpart, nnet, randomForest
Text Analysis	92	stopwords, NISTunits, ISOcodes
Social Network Analysis	53	texreg, network, ergm
Graphics	49	apt, BioIDMapper, Blaunet
Graph data structure	48	graph, scagnostics, RBGL
Genetics	44	AhoCorasickTrie, BinQuasi, biofiles
Finance	36	RUnit, RcppCCTZ, CFL
Insurance and Actuary	34	rsp, babar, BALD
Numerical Optimization	30	pbivnorm, rgenoud, Matching
Sparse Matrix	30	registry, slam, Rsymphony
Java	28	rJava, RWekajars, AWR
Neuroscience	27	STAR, rstream, extremefit
Time, Date, and Money	27	tis, setRNG, CDNmoney

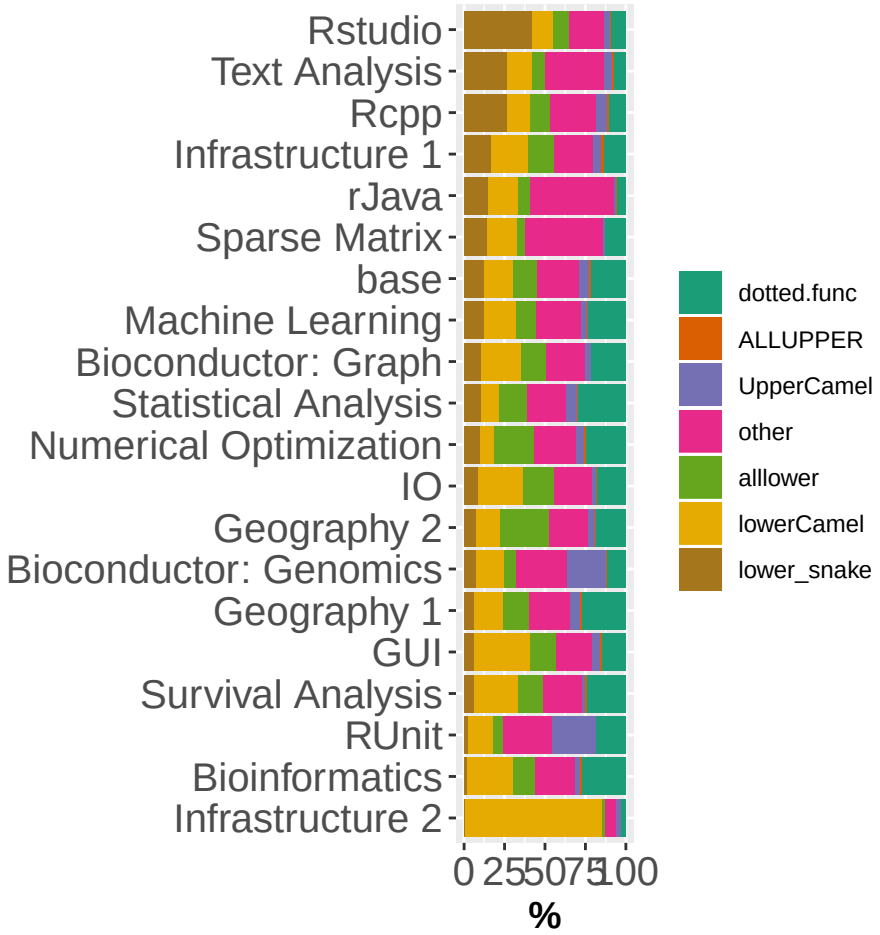

Figure 4: Community-specific distribution of naming conventions among 18 large communities.

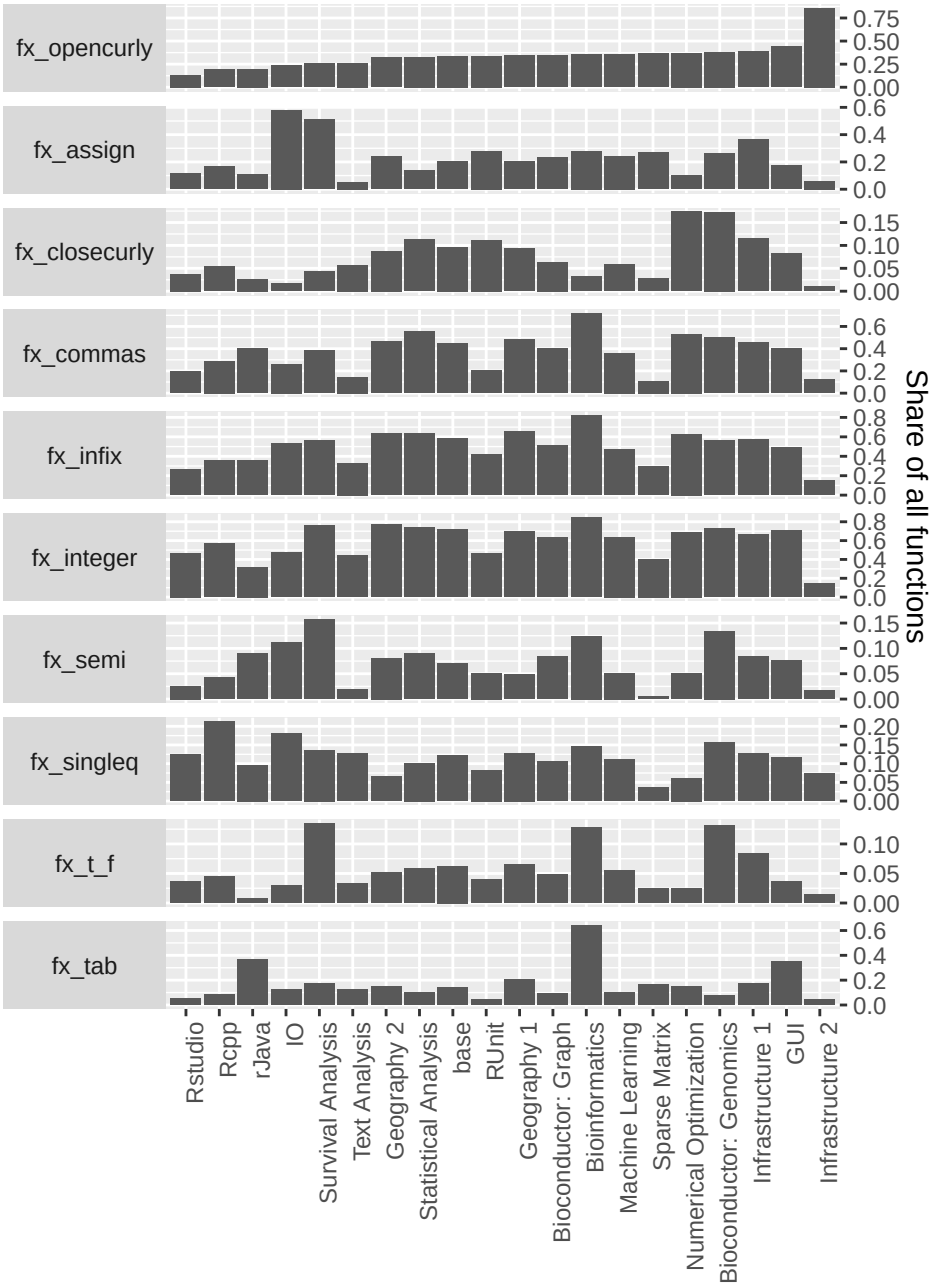

Figure 5: Community-specific distribution of style-elements among 18 large communities

with S-Plus (as well as with C, Java, and many other languages)" (Chambers, 2001). This might be a good intention but it has an unintended consequence of introducing a very persistent PSV that two major style-guides, Wickham (2017) and Google (2019), consider as a bad style.

Secondly, style-guides, linters, and editors are important standardizers of PSV. Although we have not directly measured the use of style-guides, linters, and editors in our analysis [4], we infer their effect by study the time trend (Figure 1). Even with these standizers, programming styles are slow to change. As indicated by the local patterns of PSV we found in some communities, some package developers keep using their own styles. Having said so, we are not accusing those developers of not following the trendy programming styles. Instead, they follow the mantra of "if it ain't broke don't fix it". Again, from a policy recommendation standpoint, the existence of local PSV patterns suggests there are many blind spots to the previous efforts in addressing PSV. Style-guide's authors may consider community outreach to promote their endorsed styles, if they want other communities to adopt their styles.

Our analysis also opens up an open question: should R adopt an official style-guide akin the PEP-8 of the Python Software Foundation (Van Rossum et al., 2001)? There are of course pros and cons of adopting an official style-guide. As written by Christiansen (1998), "style can easily become a religious issue." It is not our intention to meddle in this "religious issue." If such an effort would be undertaken by someone else, we suggest the following consensus-based style. The following is an example of a function written in such style.

```
softplus_func <- function(value, leaky = FALSE) {
    if (leaky) {
        warnings("using leaky RELU!")
        return(ifelse(value > 0, value, value * 0.01))
    }
    return(log(1 + exp(value)))
}
```

In essense,

- Use snake case
- Use <- to assign, don't use =
- Add a space after commas
- Use TRUE / FALSE, don't use T / F
- Put open curly bracket on same line then a newline
- Use double quotation mark for strings
- Add spaces around infix operators
- Don't terminate lines with semicolon
- Don't explicitly type integers (i.e. 1L)
- Put close curly bracket on its own line
- Don't use tab to indent

We must stress here that this *consensus-based* style is only the most popular style based on our analysis, i.e. the *Zeitgeist* (the spirit of the age) [5]. We have no guarantee that this style can improve clarity or comprehensibility. As a final remark: although enforcing a consistent style can improve open source collaboration (Wang and Hahn, 2017), one must also bear in mind that these rules might need to be adjusted sometimes to cater for programmers with special needs. For example, using spaces instead of tabs for indentation can make code not accessible to visually impaired programmers (Mosal, 2019).

Reproducibility

The data and scripts to reproduce the analysis in this paper are available at https://github.com/chainsawriot/rstyle.

[4] The usage of style-guides, linters, and editors cannot be directly measured from the record on CRAN. The maintainers usually do not explicitly state the style-guide they endorsed in their code. Similarly, packages that have been processed with linters do not import or suggest linters such as **lintr**, **styler** or **goodpractice**. It might be possible to infer the use of a specific editor such as RStudio in the development version of a package with signals such as the inclusion of an RStudio Project file. These signals, however, were usually removed in the CRAN submission of the package. Future research should use alternative methods to measure the usage of these 3 things in R packages.

[5] In 2019, 5.35% of all functions are using this *Zeitgeist* style. Using electorial system as an analogy, this style is having the plurality (have the highest number of votes) but not the absolute majority (have over 50% of the votes)

Bibliography

R. Bååth. The state of naming conventions in R. *The R journal*, 4(2):74–75, 2012. [p1, 4, 5]

Bioconductor. *Coding style*, 2015. URL https://www.bioconductor.org/developers/how-to/coding-style/. [p1]

J. Chambers. *Assignments with the = Operator.*, 2001. URL http://developer.r-project.org/equalAssign.html. [p5, 10]

C.-h. Chan. *chainsawriot/baaugwo*, Sept. 2020. URL https://doi.org/10.5281/zenodo.4016596. [p2]

T. Christiansen. *Perl Style: Everyone Has an Opinion*, 1998. URL https://www.perl.com/doc/FMTEYEWTK/style/slide1.html/. [p10]

CRAN. *CRAN Mirror HOWTO/FAQ*, 2019. URL https://cran.r-project.org/mirror-howto.html. [p2]

G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL http://igraph.org. [p4]

M. O. Elish and J. Offutt. The adherence of open source Java programmers to standard coding practices. 2002. [p1]

Google. *Google's R Style Guide*, 2019. URL https://google.github.io/styleguide/Rguide.html. [p1, 7, 10]

J. Hester and F. Angly. *lintr: A 'Linter' for R Code*, 2019. URL https://CRAN.R-project.org/package=lintr. R package version 2.0.0. [p2]

C. Mosal. *Nobody talks about the real reason to use Tabs over Spaces.*, 2019. URL https://www.reddit.com/r/javascript/comments/c8drjo/nobody_talks_about_the_real_reason_to_use_tabs/. [p10]

P. W. Oman and C. R. Cook. A programming style taxonomy. *Journal of Systems and Software*, 15(3): 287–301, 1991. [p1]

P. Pons and M. Latapy. Computing communities in large networks using random walks. In *International symposium on computer and information sciences*, pages 284–293. Springer, 2005. [p4]

G. Van Rossum, B. Warsaw, and N. Coghlan. PEP 8: style guide for Python code. *Python. org*, 1565, 2001. [p10]

J. Vanderplas. *Exploring Line Lengths in Python Packages*, 2017. URL https://jakevdp.github.io/blog/2017/11/09/exploring-line-lengths-in-python-packages/. [p6]

Z. Wang and J. Hahn. The effects of programming style on open source collaboration. 2017. URL https://aisel.aisnet.org/icis2017/DigitalPlatforms/Presentations/22/. [p1, 10]

H. Wickham. *The tidyverse style guide*, 2017. URL https://style.tidyverse.org/. [p1, 7, 10]

Chia-Yi Yen
Mannheim Business School, Universität Mannheim
L 5, 6, 68131 Mannheim
Germany
https://orcid.org/0000-0003-1209-7789
yen.chiayi@gmail.com

Mia Huai-Wen Chang
Akelius Residential Property AB
Erkelenzdamm 11-13, 10999 Berlin
Germany
mia5419@gmail.com

Chung-hong Chan
Mannheimer Zentrum für Europäische Sozialforschung, Universität Mannheim
A5, 6, 68159 Mannheim
Germany
https://orcid.org/0000-0002-6232-7530
chung-hong.chan@mzes.uni-mannheim.de