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: the effect of style-guides, the effect of introducing new features, and the 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. [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. 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), are available (Table 1) [2].

Among the 3 style-guides, the major differences are the suggested naming convention and indentation, as highlighted in Table 1. 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. contributors 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

All exported functions 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 ($7 \times 2^{10} = 7,168$).

Methodology of community-specific variations analysis

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 submission history of every package from 1998 to 2019.

By applying the Walktrap Community Detection on the 2019 data, we have identified 931 communities in this CRAN dependency graph. The purpose of this analysis is to show the PSV in different communities. We selected the largest 20 communities for further analysis. The choice of 20 is deemed enough to show these community-specific variations. Roughly 88% of the total 14,491 packages are covered in these 20 identified communities, which show that the coverage of our analysis is comprehensive. Readers could explore other choices themselves using our openly shared data.

Methodology of within-package variations analysis

Maintaining a consistent style in source code can enable efficient reading by multiple readers (Gillespie and Lovelace, 2016) [3]. In addition to community-level analysis, we extend our work to the package-level, in which we investigate the consistency of different style elements within a package. In this analysis, we studied 12 style elements, including fx_assign, fx_commas, fx_integer, fx_semi, fx_t_f, fx_closecurly, fx_infix, fx_opencurly, fx_singleq, fx_tab, and fx_name. In other words, 11 binary variables (the first 11) and 1 multinomial variable (fx_name) could be assigned to each function within a package.

We quantified the package-level consistency of each style element within by computing the entropy for each style element. Given a style element S of an R package R_i, with possible n choices $s_1, \ldots s_n$ (e.g. n = 2 for binary; n = 7 for fx_names), the entropy $H(S)$ is calculated as:

$$H(S) = -\sum_{i=1}^{n} P(s_i) \log P(s_i) \tag{1}$$

$P(s_i)$ is calculated as the proportion of all functions in R_i with the style element s_i. For example, if a package has 4 functions and the S of these 4 functions are 0,0,1,2. The entropy $H(S)$ is $-((0.5 \times log(0.5)) + (0.25 \times log(0.25)) + (0.25 \times log(0.25))) = 0.45$.

As the value of $H(S)$ is not comparable across different S with a different number of n, we normalize the value of $H(S)$ into $H'(S)$ by dividing $H(S)$ with the theoretical maximum. The maximum values of $H(S)$ for n = 2 and n = 7 are 0.693 and 1.946, respectively.

Finally, we calculate the $H'\bar{(}S)$ of all CRAN packages (i.e. $R_1 \ldots R_n$, where n equals the number of all CRAN packages) by averaging the package-level entropy we computed previously.

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 an illustrative example of the effect of introducing a new language feature 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.

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 [4]. 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, while the growth of lower_snake is stronger, with almost a 20% growth in the share of all functions in contrast with the 1-2% growth of other naming conventions. 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.

The evolution of line lengths is tricky to be visualized on a 2-D surface. We have prepared a Shiny app (`https://chung-hong-chan.shinyapps.io/shiny/`) 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

[3]Maintaining a consistent style is even thought to be a quality of a successful R programmer in Gillespie and Lovelace (2016).

[4]'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.

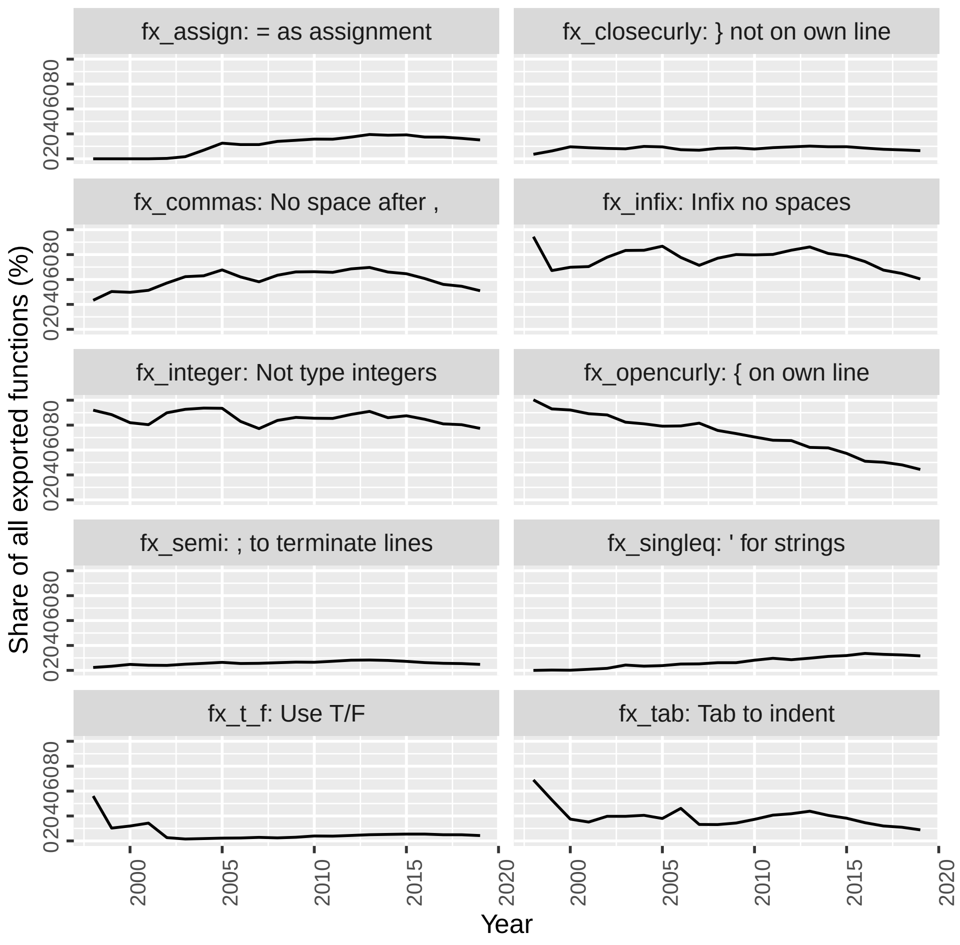

Figure 1: Evolution in popularity of 10 style-elements from 1998 to 2019.

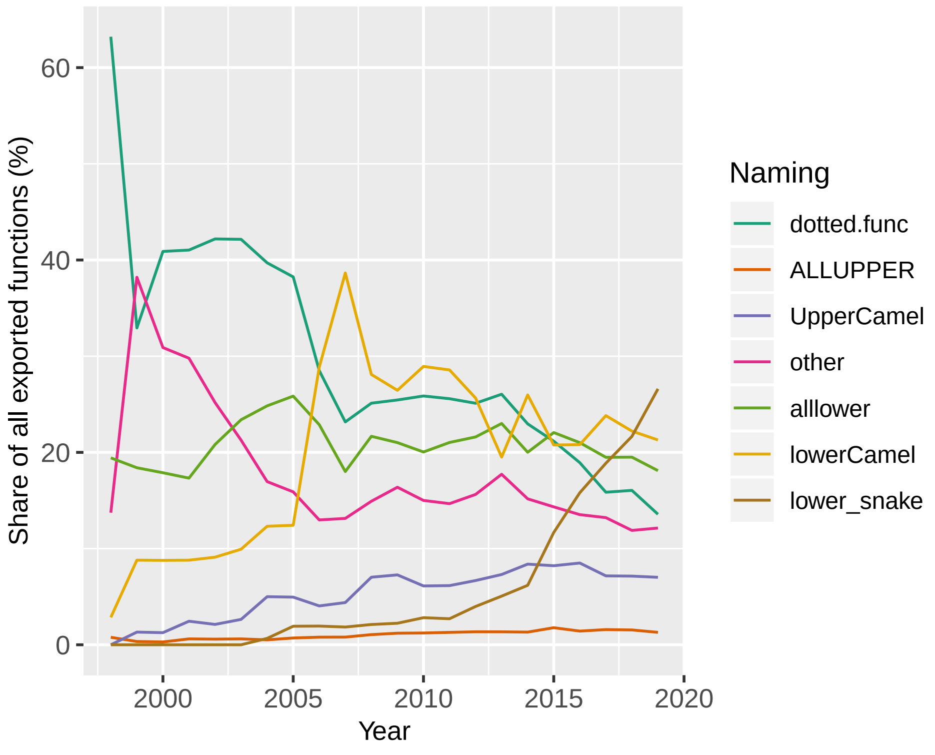

Figure 2: Evolution in popularity of 7 naming conventions from 1998 to 2019.

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.

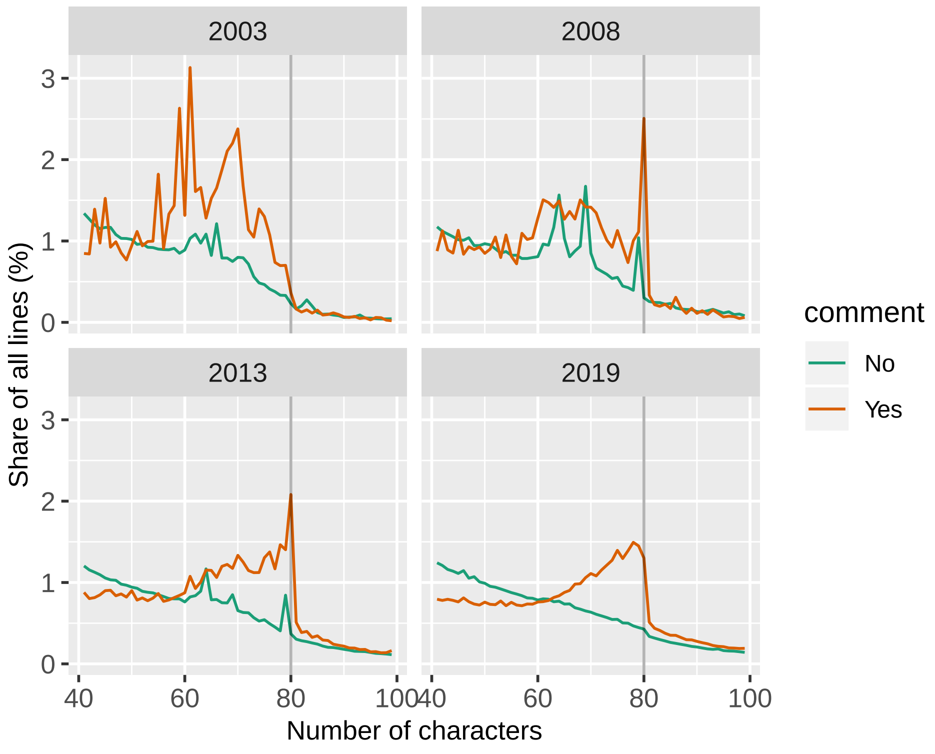

Figure 3: Change in line length distribution: 2003, 2008, 2013 and 2019.

Community-based variations

Using the aforementioned community detection algorithm of the dependency graph, the largest 20 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" community as expected because it is the naming convention endorsed by the Tidyverse Style-guide. However, only a few functions exported by the packages from "GUI: Gtk" community uses such convention.

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

Within-package variations

The result shows that the consistency of style elements within a package varies (Figure 6). For example, style elements like fx_integer, fx_commas, fx_infix, fx_opencurly, and fx_name have less consistency within a package than fx_tab, fx_semi, fx_t_f, fx_closecurly, fx_singleq, and fx_assign. This finding echoes previous concerns e.g. Oman and Cook (1991); Elish and Offutt (2002); Wang and Hahn (2017); Gillespie and Lovelace (2016) that these inconsistent style variations within a software project (e.g. in an R package) might make open source collaboration difficult.

In Figure 7, we contextualize the above analysis by showing the distribution of fx_name in 20 R packages with the highest PageRank (Page et al., 1999) in the CRAN dependency graph. Many of these packages have only 1 dominant naming convention (e.g. lower_snake or lowerCamel), but not always. For instance, functions with 6 different naming conventions can be found in the package **Rcpp**.

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

Community	Number_of_Packages	Top_3_Packages
base	5157	methods, stats, MASS
RStudio	4758	testthat, knitr, rmarkdown
Rcpp	826	Rcpp, tinytest, pinp
Statistical Analysis	463	survival, Formula, sandwich
Machine Learning	447	nnet, rpart, randomForest
Geospatial	367	sp, rgdal, maptools
GNU gsl	131	gsl, expint, mnormt
Graph	103	graph, Rgraphviz, bnlearn
Text Analysis	79	tm, SnowballC, NLP
GUI: Tcl/Tk	55	tcltk, tkrplot, tcltk2
Infrastructure	54	rsp, listenv, globals
Numerical Optimization	51	polynom, magic, numbers
Genomics	43	Biostrings, IRanges, S4Vectors
RUnit	38	RUnit, ADGofTest, fAsianOptions
Survival Analysis	33	kinship2, CompQuadForm, coxme
Sparse Matrix	32	slam, ROI, registry
GUI: Gtk	31	RGtk2, gWidgetstcltk, gWidgetsRGtk2
Bioinformatics	29	limma, affy, marray
IO	28	RJSONIO, Rook, base64
rJava	27	rJava, xlsxjars, openNLP

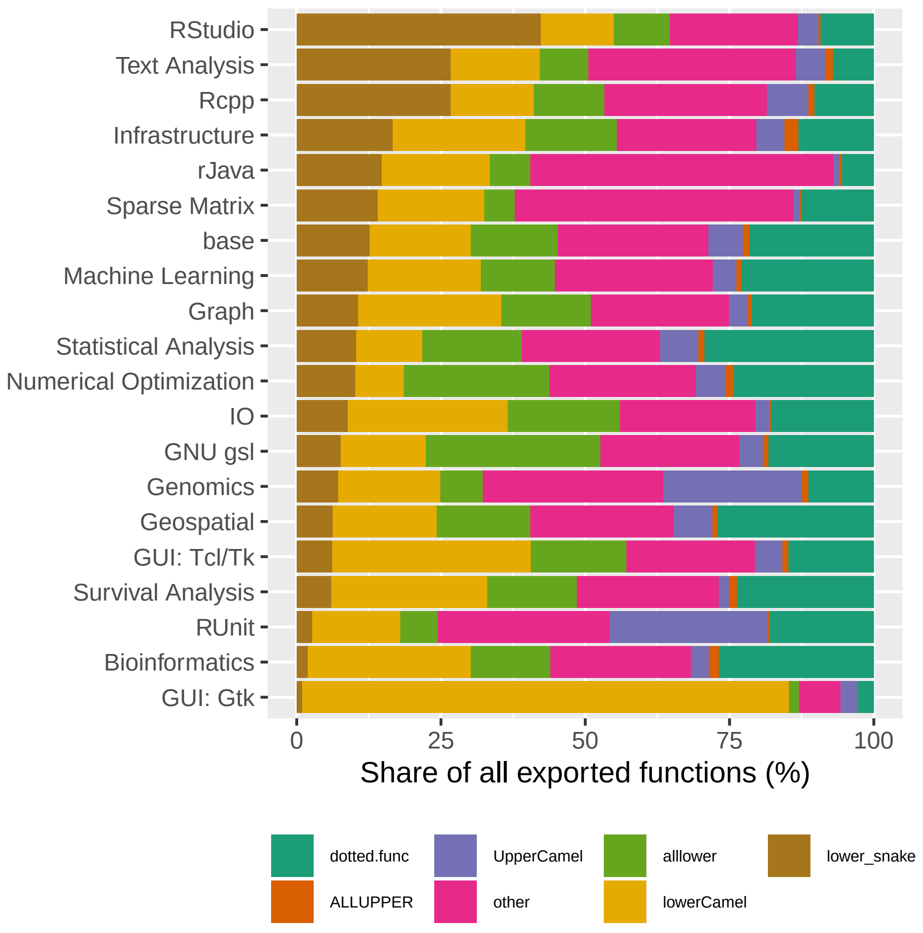

Figure 4: Community-specific distribution of naming conventions among 20 large communities.

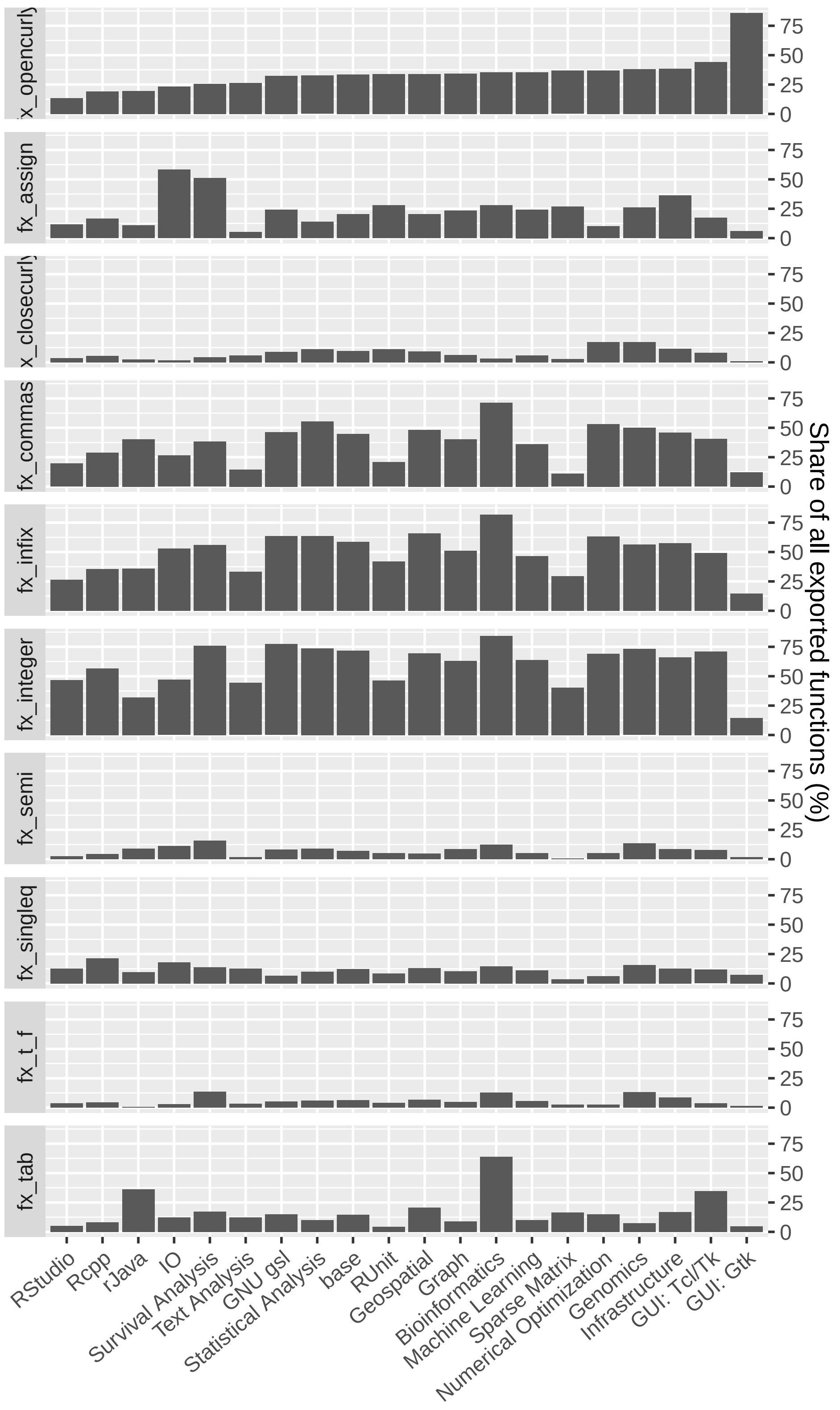

Figure 5: Community-specific distribution of style-elements among 20 large communities

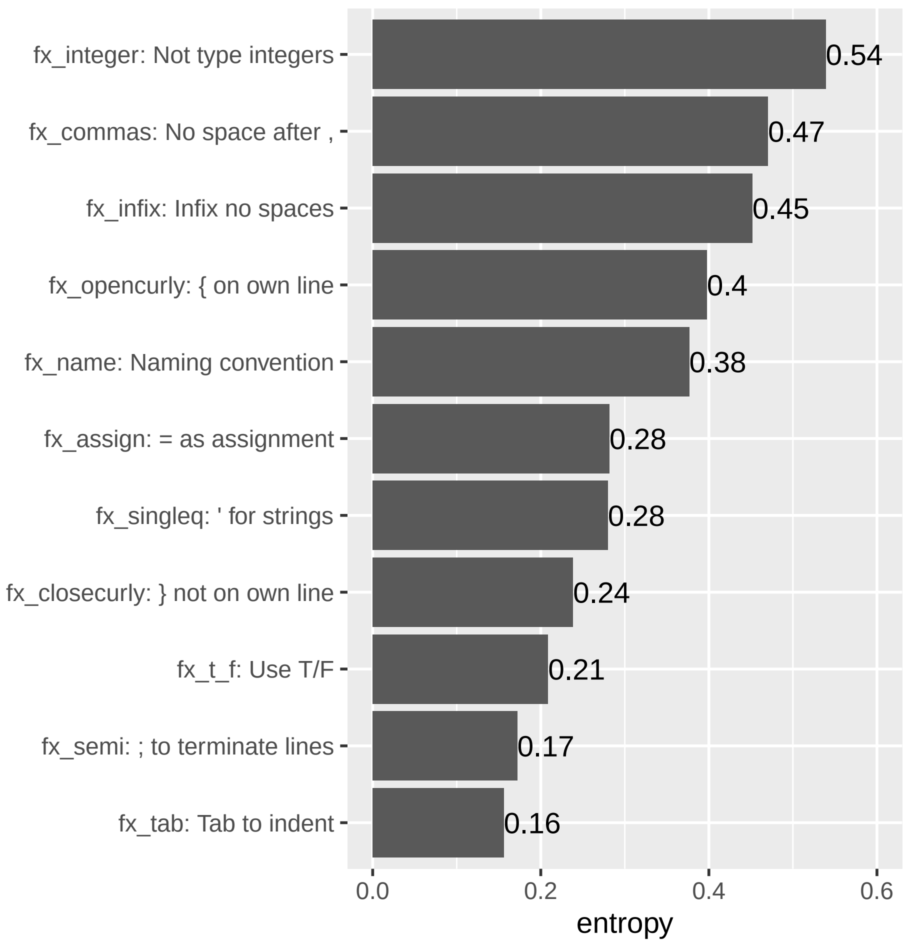

Figure 6: Average package-level entropy of 12 style elements

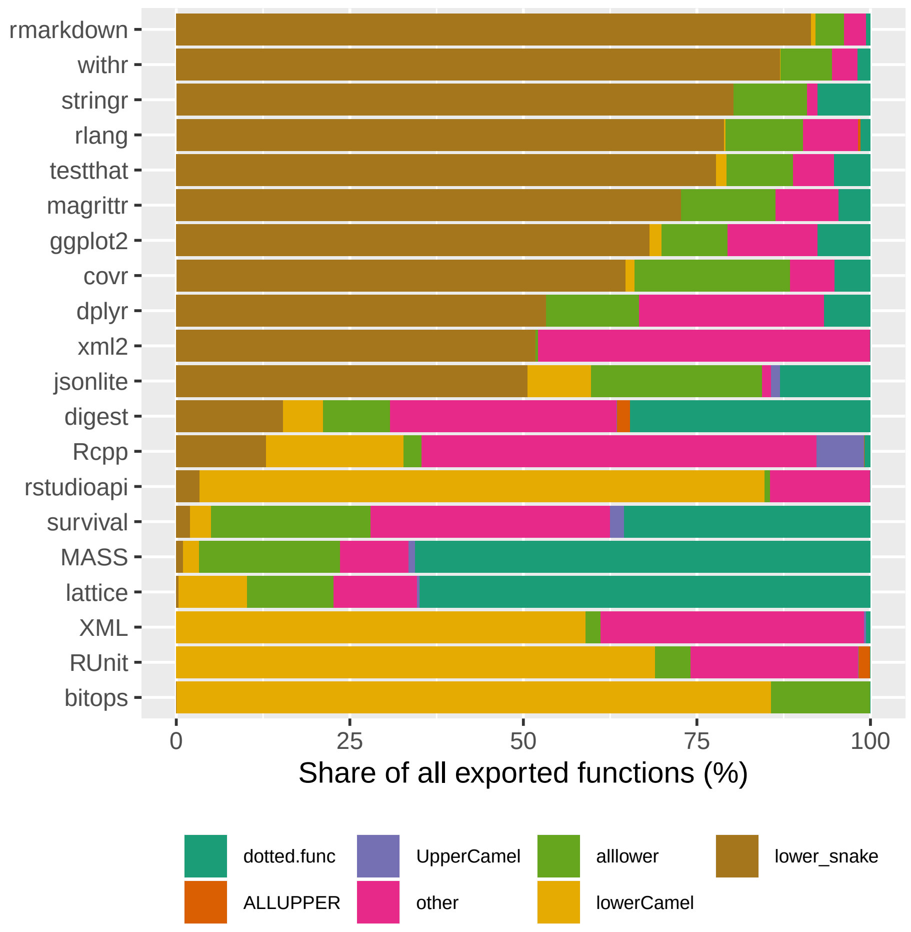

Figure 7: Package-specific distribution of naming conventions among the most important packages

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, which contains all submissions from 1998 to 2019. From our analysis, we identify three factors that possibly drive PSV: the 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 the 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. First, the introduction of a new language feature 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 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.

Second, 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 [5], we infer their effect by study the time trend (Figure 1). Even with these standardizers, programming styles are slow to change. As indicated by the local PSV patterns, we found in some communities, some package developers have their own style. 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. The authors of the style guides 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 essence,

- 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

[5]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. In this study, similar to other studies, e.g. Bafatakis et al. (2019), we use style compliance as a proxy to usage of a particular style guide, linter or editor.

- 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) [6]. 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 inaccessible 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. An archived version is available at this DOI: http://doi.org/10.5281/zenodo.4026589.

Acknowledgment

The authors would like to thank Wush Wu and Liang-Do Wang for their valuable comments that greatly improved this paper. Taiwan R User group and R-Ladies Taipei provided feedback for this research.

Bibliography

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

N. Bafatakis, N. Boecker, W. Boon, M. Cabello Salazar, J. Krinke, G. Oznacar, and R. White. Python coding style compliance on stack overflow. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019. doi: 10.1109/msr.2019.00042. URL http://dx.doi.org/10.1109/MSR.2019.00042. [p12]

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, 12]

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/. [p12]

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, 7]

C. Gillespie and R. Lovelace. *Efficient R programming: a practical guide to smarter programming*. "O'Reilly Media, Inc.", 2016. [p5, 7]

Google. *Google's R Style Guide*, 2019. URL https://google.github.io/styleguide/Rguide.html. [p1, 12]

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/. [p13]

[6]In 2019, 5.35% of all functions are in this *Zeitgeist* style. Using electoral 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)

P. W. Oman and C. R. Cook. A programming style taxonomy. *Journal of Systems and Software*, 15(3): 287–301, 1991. [p1, 7]

L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. [p7]

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. [p12]

J. Vanderplas. *Exploring Line Lengths in Python Packages*, 2017. URL `https://jakevdp.github.io/blog/2017/11/09/exploring-line-lengths-in-python-packages/`. [p7]

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, 7, 13]

H. Wickham. *The tidyverse style guide*, 2017. URL `https://style.tidyverse.org/`. [p1, 12]

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