Journal of Data Science 19 (1), 111–125
January 2021

DOI: 10.6339/21-JDS998
Computing in Data Science

Reproducible Science with LaTeX

Haim Bar[1,*] and HaiYing Wang[1]

[1]*Department of Statistics, University of Connecticut, Storrs, CT, USA*

Abstract

This paper proposes a procedure to execute external source codes from a LaTeX document and include the calculation outputs in the resulting Portable Document Format (pdf) file automatically. It integrates programming tools into the LaTeX writing tool to facilitate the production of reproducible research. In our proposed approach to a LaTeX-based scientific notebook the user can easily invoke any programming language or a command-line program when compiling the LaTeX document, while using their favorite LaTeX editor in the writing process. The required LaTeX setup, a new Python package, and the defined preamble are discussed in detail, and working examples using R, Julia, and MatLab to reproduce existing research are provided to illustrate the proposed procedure. We also demonstrate how to include system setting information in a paper by invoking shell scripts when compiling the document.

Keywords *literate programming; reproducibility; scientific notebook;* `--shell-escape`

1 Introduction

Reproducing research results has always been key to good science, but in recent years the topic rose to the forefront in the scientific community and the media. Famously, in 2005, John Ioannidis wrote a paper titled 'Why Most Published Research Findings Are False' (Ioannidis, 2005) and ignited a lively discussion and prompted many ideas how to make things better. The expectation is, and should be, that given a careful recipe which describes how an experiment was done or a data set has been analyzed, others will achieve the same results if they follow the same procedure. In the context of statistical analysis this means that authors are required to provide the necessary data and programs, as well as to specify dependencies on external resources (e.g., packages, data repositories, etc.) and settings (including random seeds, for example).

The notion of a 'scientific notebook software' has evolved greatly over the last couple of decades. A scientific notebook makes it possible for computational and statistical results to be generated by the same tool which is used to write the paper, thus ensuring that results reported in the paper match the most up to date analysis performed by the authors. Proprietary software such as MatLab and Mathematica have built-in notebooks, and open-source languages like R and Python have similar implementations, namely, the *Sweave* package (Leisch, 2002), the *knitr* package (Xie, 2020, 2015, 2014) and the *rmarkdown* package (Allaire et al., 2020; Xie et al., 2018) for R, and the *Pweave* package and *Jupyter* for Python. The aforementioned tools use a sequential procedure. They insert prose into programming code to create a special file (e.g., a Rnw file with *knitr* or a .Rmd file with *rmarkdown*), use the programming language to process this special file to generate the tex file, and then use LaTeX to generate the pdf file. For Emacs users, the org-mode major mode (Schulte et al., 2012) has the functionality of incorporating source codes, with a similar markup syntax to markdown. Similar to markdown, it uses pandoc

*Corresponding author. Email: haim.bar@uconn.edu.

Received August 1, 2020; Accepted November 1, 2020

to process an org file and call other programming languages to run the source codes to create a tex file.

In this paper we take a different approach and incorporate the code in a LaTeX document, rather than inserting prose into a program. When the document is compiled, the code is executed and the results are embedded in the pdf file. Our approach has a couple of advantages over the aforementioned programming language-based notebooks. First, any programming language can be easily used. In fact, any command-line executable can be invoked when preparing the pdf file from the tex file. While some integrated development environments (IDEs) support multiple languages, using more than one in the same document is, at best, cumbersome. Second, as far as text editing goes, programming language editors are less convenient than LaTeX editing tools such as TeXworks, Kile, TeXshop, or Emacs, just to name a few. This is obviously a subjective preference, but in our experience the available LaTeX editing tools have a lot of features that (currently) most programming language editors do not have. For example, most LaTeX editors have auto completion and cross-reference support for tex writing. Most LaTeX editors also have the forward search and inverse search to link the tex file and the pdf file, which are not available in most language-specific editors. Each TeX editor has its own convenient user-interface features which users get easily accustomed to. For example, Kile (and other editors) have a DVI preview which does not require full compilation of the document and it is updated automatically when the document is saved. Scientific WorkPlace has short-cut buttons to common constructs (Greek letters, formulas, etc.) Many editors (like TeXShop) have menu items to add LaTeX tables, figures, and other objects to the document. In typical manuscripts more space is ultimately used for prose than for the computational results, so it makes sense to use the tool which is most convenient for the majority of the writing process. Furthermore, the programming language-based notebook approach requires learning the markdown syntax, which is not as rich as what LaTeX already offers. We want to emphasize that we by no means claim that the proposed approach aims to replace a programming language editor or the markdown language. In the stage of developing code for analysis, the specific editors for the used languages are preferable because they often provide rich support for coding and debugging that a LaTeX editor does not have. However, when the code is relative stable and the focus becomes writing a report or a paper, the proposed procedures is recommended.

Our proposed approach achieves the same purpose of reproducibility as the well-known notebooks, but it is a lot more flexible. It is in the spirit of Donald Knuth's literate programming (Knuth, 1984) which considers computer programs 'to be works of literature'. It is also a realization of the concept of a compendium introduced by Gentleman and Temple Lang (2007). The method uses basic features in LaTeX, and the key is to enable the 'shell-escape' option when compiling a LaTeX document. Our LaTeX-driven approach is very simple to implement. Since 'shell-escape' is part of the LaTeX core functionality, only a handful of external packages have to be included in the preamble.

In Section 2 we introduce the components of our proposed method. In Sections 3 and 4 we demonstrate the LaTeX-based notebook approach using R, Julia, and MatLab. Reproducibility requires authors to provide not only code and data, but also system configuration at the time of the compilation. Each operating system has its own functions to report configuration settings, and in the Appendix we show an example on a Mac. This also serves as an example of how to invoke a shell script.

2 Software Setup

We present the LaTeX setup, server configuration, and some short commands specific to certain programming languages for ease of use. The *runcode* package can be used in two modes – 'batch' and 'server'. When the LaTeX document is compiled in the batch mode, pdflatex invokes the command-line statistical tool each time it executes a code segment and when that execution is complete, the statistical software exits. For example, with R batch-mode execution is done by *runcode* like this:

```
Rscript programName.R > outputFile
```

In the 'server-mode' *runcode* maintains a continuous bi-directional communication channel to the statistical software. The machinery enabling the server-mode is described in Section 2.2. In general, the batch-mode is simpler to use (and implement) since it does not require any additional packages, whereas server-mode requires the Python package *talk2stat*. However, we recommend the server-mode since it maintains a 'live session' which is more efficient – both because the statistical software executable does not have to be invoked multiple times, and also because the continuous session makes all previous computations and environment variables readily available for subsequent operations, whereas in batch-mode it is necessary to save and restore sessions between invocations for this purpose. Note that both modes can be used in the same LaTeX document.

In the following it is assumed that the executables for the statistical software and for Python are in the user's path. If that is not the case or if the user wishes to specify a different path, this can be done by modifying the runcode.sty file, and adding the complete path to the executable file name (e.g., Rscript).

2.1 LaTeX Setup

Our proposed approach requires to enable the 'shell-escape' option when compiling a LaTeX document. From the command line, it is done like this:

```
latex --shell-escape myFile.tex
```

Most people use text editors that have LaTeX compilation capability, which can be configured to enable this option. Here are examples using some commonly used text editors.

- With TeXnicCenter: under "Build > Define Output Profile > Latex ⇒ PDF", insert the option `-shell-escape` into "Command line arguments to pass to the compiler".
- With TeXshop: under "Preferences > Engine > pdfTeX > LaTeX Input Field", change
  `pdflatex --file-line-error --synctex=1`
  to
  `pdflatex --file-line-error --synctex=1 --shell-escape`
- With TeXStudio: under "Options > Configure TeXStudio > Commands", change
  `pdflatex -synctex=1 -interaction=nonstopmode %.tex`
  into
  `pdflatex -synctex=1 -interaction=nonstopmode --shell-escape %.tex`
- With TeXworks: in "Preferences > Typesetting > pdflatex", add `--shell-escape` as an argument and move it before the argument called '$fullname'.

For other text editors, see their specific documentation to see how it is done.

It is also worth mentioning the popular web-based editor and collaboration tool for LaTeX, Overleaf. It is possible to use knitr on Overleaf, which allows authors to include R in their manuscripts. Because the 'shell-escape' option is enabled by default on the Overleaf server, our method can also be used on Overleaf by simply installing the runcode.sty file in the project directory. Currently, both R and Python are installed on Overleaf, but Julia and MatLab are not. Whenever a language other than R and Python becomes available on Overleaf, it will be immediately possible for Overleaf users of *runcode* to take advantage of it. Note that currently runcode can only be used in the batch-mode on OverLeaf, but we hope that in the future Overleaf will enable the server-mode functionality.

Our LaTeX-driven approach requires a few external packages in the preamble. The packages *minted*, *fontenc*, and *textgreek*, are used in order to ensure that code and output are displayed correctly and aesthetically, even when they contain non-ASCII characters. Note that the package *minted* requires Python and the Python package *pygments* in order to run. Without *pygments*, one can use the package *fvextra* instead of *minted*. This package does not write temporary files on disk and it is faster to produce the pdf file. However, *fvextra* does not provide syntax highlights[1]. To embed code in LaTeX source file directly, we use the *filecontents* package. The *filecontents* package is only explicitly loaded if an older version of LaTeX is used. It has been part of the LaTeX engine since 2019. The packages *tcolorbox* and *xcolor* are used to enhance the output, and the *xifthen*, *xparse*, and *xstring* packages are used in the implementation of our new commands to make them clearer and more efficient. We put all the LaTeX setups in a new package called *runcode*, and the style file runcode.sty can be loaded by `\usepackage[options]{runcode}`, where possible options[2] are:

- 'run': run source code by setting the global Boolean variable `runcode` to be true.
- 'cache': use cached results by setting the global Boolean variable `runcode` to be false.
- 'R': start server for R (this requires the Python *talk2stat* package).
- 'julia': start server for Julia (this requires the Python *talk2stat* package).
- 'matlab': start server for MatLab (this requires the Python *talk2stat* package).
- 'nominted': use the *fvextra* package instead of the *minted* package to show code (this does not require the Python *pygments* package, but it does not provide syntax highlights).
- 'stopserver': stop the *talk2stat* server(s) when the pdf compilation is done.

We have implemented four basic commands in the *runcode* package:

- `\runExtCode{Arg1}{Arg2}{Arg3}[Arg4]` runs an external code. It takes four arguments: `Arg1` is the executable program, `Arg2` is the source file name, `Arg3` is the output file name (with an empty value, the counter `codeOutput` is used), and `Arg4` controls when to run the code. `Arg4` is an optional argument with three possible values: if skipped or with empty value, the value of the global Boolean variable `runcode` is used; if the value is set to 'run', the code will be executed; if set to 'cache' (or anything else), use cached results (see more about the cache below).
- `\showCode{Arg1}{Arg2}[Arg3][Arg4]` shows the source code, using *minted* for a pretty layout or *fvextra* (if Python or *pygments* are not installed). It takes four arguments: `Arg1` is the programming language, `Arg2` is the source file name, `Arg3` and `Arg4` are the first and

[1]The reason we prefer to use minted and fvextra over the listings package is that the former support the extended Unicode character set. In addition, the minted package uses the pygmentize parsing library which is more extensive and expressive for programming languages compared with the keyword detection in listings.

[2]The *runcode* package also works for arxiv.org. However, one needs to enable the 'cache' and 'nominted' options, because arxiv.org does not support *pygments*.

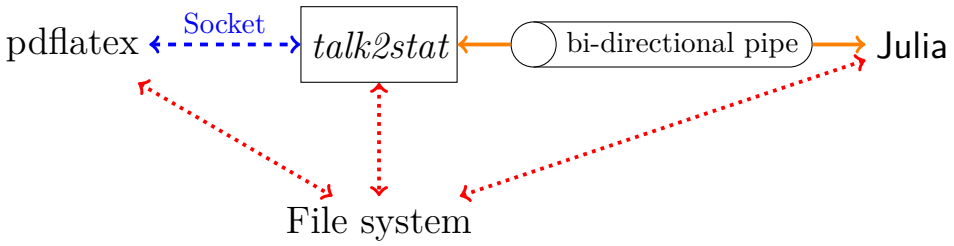

Figure 1: The functionality of *talk2stat* Python package.

last line to show (optional - if skipped or with empty values, all the lines in the source file will be printed).

- `\includeOutput{Arg1}[Arg2]` is used to embed the output from executed code. It takes two arguments: `Arg1` is the output file name, and it needs to have the same value as that of `Arg3` in `\runExtCode`. If an empty value is given to `Arg1`, the counter `codeOutput` is used. `Arg2` is optional and it controls the type output with a default value 'vbox' (skipped or 'vbox' = verbatim in a box, 'tex' = pure latex, or 'inline' = embed result in text).
- `\inln{Arg1}{Arg2}[Arg3]` is designed for simple calculations; it runs one command (or a short batch) and displays the output within the text. `Arg 1` is the executable program or programming language. `Arg2` is the source code. `Arg 3` is the output type (if skipped or with empty value, the default type is 'inline'; 'vbox' = verbatim in a box).

Typically, at least in our experience, the writing of the text portion of a manuscript begins when the code is relatively stable, at which point most of the edits are applied to the text. In this situation, one may want to disable the execution of the embedded code and use the cached results from the last compilation, so that subsequent LaTeX compilations are faster. We define a Boolean variable `runcode` to allow this functionality, and one can disable the execution of the embedded code by declaring the 'cache' option when loading the *runcode* package (this sets `runcode` to false). When code execution is disabled, output from the last time the code was run is used when compiling the tex file. As discussed above, one can override the global setting of `runcode` in order to enable or disable the execution of selected chunks of code. All calculation results that can be included in LaTeX output are stored in the folder "tmp" in the directory of the working LaTeX file.

In the following sections we demonstrate the LaTeX-based notebook approach using R, Julia, MatLab/Octave, and shell scripts.

2.2 *talk2stat* Python Package

The functionality of a new Python package *talk2stat* is described in Figure 1. In this example, we show a situation in which Julia is used as the statistical engine.

When a tex file is being compiled, the *runcode* package first checks if the *talk2stat* server is running, and if not, it starts it. The server listens to a user-defined port number, and expects connections via a socket interface (using the Internet Protocol v4 addresses) which is represented by the dashed blue line. The default port numbers in the *runcode* package for R, Julia, and MatLab are 65432, 65431, and 65430, respectively.

During the compilation of the tex file, the following steps occur each time an embedded code is encountered (using `\runExtCode` or `\inln`).

1. *runcode* reads the code, which can be specified either a file name containing the code, or a short in-line code segment.
2. The code or input file name is sent to *talk2stat*'s client function, which uses the socket interface to accept requests.
3. *talk2stat*'s server function sends the code to the statistical engine via a 'bi-directional pipe'[3]. This is represented by the solid orange line in the figure.
4. The statistical engine runs the code, and prints the output to the standard output via the pipe. The code may contain instructions to print output to the file system (e.g., figures.)
5. *talk2stat*'s server function writes the output from the statistical engine to the output file which is specified in `\runExtCode` or `\inln`.
6. pdflatex recognizes that the operation invoked by `\runExtCode` or `\inln` is complete, embeds the output in the pdf file (via `\include` or `\includegraphics`) and moves on to the next line in the tex file.

All three components in the system, namely, pdflatex, *talk2stat*, and the statistical engine, may use the file system to read data or scripts or to write their output (text, tables, figures, etc.). This is represented by the dotted red lines in the diagram.

2.3 Short Commands for R, Julia, and MatLab

When writing in LaTeX, many people define some custom shortcuts to facilitate the writing. For the same purpose, we define some short commands specifically for R, Julia, and MatLab. These short commands are based on the `\runExtCode` and `\inln` commands presented in Section 2.1, and they are customizable. In this paper, the main purpose of defining the short commands is for illustration, and a user can easily use the similar style to define short commands for other programming languages. All the short commands use the running *talk2stat* server to execute source codes if the executable program is not specified.

The short commands defined for R, Julia, and MatLab are listed below.

- R
  - `\runR[Arg1]{Arg2}{Arg3}[Arg4]` runs an external R code file. `Arg1` is optional and uses *talk2stat*'s R server by default (rather than invoking `Rscript`). `Arg2`, `Arg3`, and `Arg4` have the same effects as those of `\runExtCode` in Section 2.1.
  - `\inlnR[Arg1]{Arg2}[Arg3]` runs R source code (`Arg2`) and displays the output in line. `Arg1` is optional and uses the R server by default. `Arg2` is the R source code to run. If the R source code is wrapped between ``` on both sides (as in the markdown grammar), then it will be implemented directly; otherwise the code will be written to a file on the disk and then be called. `Arg3` has the same effect as that in the definition of `\inln` in Section 2.1.
- Julia
  - `\runJulia[Arg1]{Arg2}{Arg3}[Arg4]` runs an external Julia code file. `Arg1` is optional and uses *talk2stat*'s Julia server by default. `Arg2`, `Arg3`, and `Arg4` have the same effects as those of `\runR`.
  - `\inlnJulia[Arg1]{Arg2}[Arg3]` runs Julia source code and displays the output in line. `Arg1` is optional and uses *talk2stat*'s Julia server by default. `Arg2` and `Arg3` have the same effects as those of `\inlnR`.

[3] A bi-directional pipe allows two programs to communicate in a synchronous fashion by sending one program's output to the other's input

- MatLab Similarly, we define
  - `\runMatLab[Arg1]{Arg2}{Arg3}[Arg4]`
  - `\inlnMatLab[Arg1]{Arg2}[Arg3]`

3 An Example Using R

We demonstrate our approach using data and code provided with the paper *Coauthorship and citation networks for statisticians* (Ji and Jin, 2016; Jin, 2015). In their famous paper, Pengsheng Ji and Jiashun Jin analyze trends and patterns in statistical research, as well as author productivity and centrality, and community structures. In order to do that, they look at papers published in four highly ranked journals in statistics (Annals of Statistics, Biometrika, JASA and JRSS-B) from 2003 to the middle of 2012. After careful cleaning and resolving author names in citations, their dataset contains 3,248 papers and 3,607 authors. They perform many analyses, of which we use only a few to show our approach.

Listing 1 shows the code used by Ji and Jin (2016) to read in the data and compute the adjacency matrix. We have changed the code slightly to make it more efficient, by coercing `authorPaperBiadj` to be a sparse matrix (using the *Matrix* package (Bates and Maechler, 2019); see line 7). To include this listing, we put the following in the tex file:

```
\begin{codelisting}{Reading and preparing the coauthorship data
\citep{ji2016,jin2015}}
\showCode{R}{Code/JiJin2016.R}
\end{codelisting}
```

Listing 1. Reading and preparing the coauthorship data (Ji and Jin, 2016; Jin, 2015).

```
1  # read the data
2  library(Matrix)
3  library(ineq)
4  library(kableExtra)
5  library(igraph)
6
7  authorPaperBiadj = Matrix(as.matrix(read.table(file="Data/authorPaperBiadj.txt", sep="\t",
   ↪  header=F)))
8  authorList = as.matrix(read.table(file="Data/authorList.txt", sep="\t", header=F,
   ↪  stringsAsFactors=F))
9  paperCitAdj = as.matrix(read.table("Data/paperCitAdj.txt", header=F))
10 paperList = read.table("Data/paperList.txt", sep=",", stringsAsFactors=F, header=T)
11
12 # compute adjacency matrix of the coauthorship network
13 coauthorAdjWeighted = authorPaperBiadj %*% t(authorPaperBiadj)
14 coauthorAdj = (coauthorAdjWeighted >= 1) -0
15 diag(coauthorAdj) = 0
16
17 authorN = length(authorList)
18 paperN = dim(paperList)[1]
19 paperYear = paperList$year
```

If we only want to show certain lines, we can use the optional arguments. For example `\showCode{R}{Code/JiJin2016.R}[17][19]`, `\showCode{R}{Code/JiJin2016.R}[17][]`, or `\showCode{R}{Code/JiJin2016.R}[17]` only shows the lines 17-19 as below.

```
authorN = length(authorList)
paperN = dim(paperList)[1]
paperYear = paperList$year
```

Note that using `\showCode` does not cause the code to be executed. To execute it we use `\runR`. Also note that the code in Listing 1 does not produce any output. To run the code using *talk2stat*'s server, we put the following in the tex file:

```
\runR{Code/JiJin2016.R}{initprog}
```

In order to use the server, a user has to use the `R` option when loading the *runcode* package, i.e., to use `\usepackage[R]{runcode}`. All the codes are set to run by default globally. To disable code execution globally, use `\usepackage[cache, R]{runcode}` to enable the `cache` option. We can also force the code to run or to use the cache by using the optional `Arg4`. For example, we can use `\runR{Code/JiJin2016.R}{initprog}[run]` to force the code to run, and use `\runR{Code/JiJin2016.R}{initprog}[cache]` to prohibit the code from running.

If we want to start a new R session each time a code segment is executed instead of using the R server, we use

```
\runR[Rscript --save --restore]{Code/JiJin2016.R}{initprog}
```

Here, the `--save` option saves all variables created in this program so that they can be used by future program, and the `--restore` option restores all variables created in the previous session. If we use *talk2stat*'s server to communicate with a statistical software (R, in this section) there is no need to save any variable or environment setting, since the R session runs continuously and all variable are accessible at all times. Thus, the server option is recommended as it is more efficient than the 'batch mode', in which `Rscript` is invoked for each code segment. We will use the server option by default for the rest of the paper except for the appendix, where we show how to embed system configuration in the paper.

If we want to put R code in the tex file rather than in an external file, we do it by using the *filecontents** environment. For example, we can get a table of the number of papers by year by writing the following:

```
\begin{filecontents*}{tmp/temp00.R}
print(table(paperYear))
\end{filecontents*}
\runR{tmp/temp00.R}{paperYear}
\includeOutput{paperYear}
```

This produces the following:

```
paperYear
2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
 296  320  328  354  350  370  409  355  325  141
```

Table 1: Reproducing Table 2 from Ji and Jin, 2016

# of papers	# of coauthors	# of citers	Closeness	Betweenness
Peter Hall	Peter Hall	Jianqing Fan	Raymond J Carroll	Raymond J Carroll
Jianqing Fan	Raymond J Carroll	Hui Zou	Peter Hall	Peter Hall
Raymond J Carroll	Joseph G Ibrahim	Peter Hall	Jianqing Fan	Jianqing Fan

Note that the second mandatory argument in `\runR` matches the mandatory argument in `\includeOutput`, and it is the name of the file under the tmp folder which is used to save the output from the executable (in this case, an R program called tmp/temp00.R). The default value for the optional argument of `\includeOutput` is `vbox`, so `\includeOutput{paperYear}` is equivalent to `\includeOutput{paperYear}[vbox]`. The `vbox` option is used to print the output verbatim (no LaTeX formatting) and enclose it in a box. The same result can also be obtained by the `\inlnR` command as `\inlnR{```table(paperYear)```}[vbox]`.

The `\inlnR` command can be used when the output should be embedded in the text. For example, in their paper, Ji and Jin report the Gini coefficient based on the paper-adjacency matrix, which they compute using the *ineq* package (Zeileis, 2014). To execute their code and include its output in-line, we put the following in the tex document:

```
The Gini coefficient is \inlnR{```cat(format(ineq(rowSums(paperCitAdj)),
digits=2))```}, suggesting that the in-degree is highly dispersed.
```

This produces the following: 'The Gini coefficient is 0.77, suggesting that the in-degree is highly dispersed.'

The `\inlnR` command is designed to work for very simple calculations, and currently it does not allow " sign in the code.

To demonstrate other output options, we use the code provided by Ji and Jin (2016) to produce their Table 2 and Figure 4. For Table 2, the code is saved in a file called Code/JiJin2016Table2.R, which requires the *kableExtra* library to format the output as a LaTeX table, and the package *igraph* (Csardi et al., 2006) to calculate the closeness and betweenness metrics. Since the code produces a LaTeX-formatted table, we can enclose it in a *table* environment and achieve the same layout as in Ji and Jin (2016) (Table 1).

```
\runR{Code/JiJin2016Table2.R}{table2}
\includeOutput{table2}[tex]
```

The code to produce Figure 4 in Ji and Jin (2016) is saved in Code/JiJin2016Plot4.R. The program creates two pdf files called tmp/LC-citations.pdf and tmp/propCitation.pdf. Using these pdf files and the *figure* environment in LaTeX we can achieve the same layout as in Ji and Jin (2016) (Figure 2).

```
\runR{Code/JiJin2016Plot4.R}{}
\begin{figure}[b!]
\begin{minipage}[b]{0.45\linewidth}
\centering
\includegraphics[width=\textwidth]{tmp/LC-citations.pdf}
\end{minipage}
\hspace{0.5cm}
```

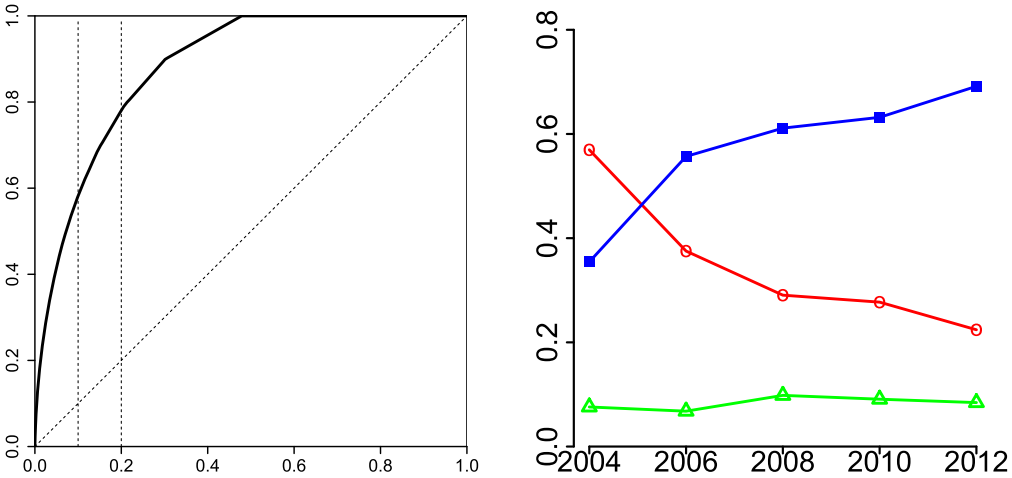

Figure 2: Reproducing Figure 4 from Ji and Jin (2016).

```
\begin{minipage}[b]{0.45\linewidth}
\centering
\includegraphics[width=\textwidth]{tmp/propCitation.pdf}
\end{minipage}
\caption{Reproducing Figure 4 from \cite{ji2016}}\label{jj2016figure4}
\end{figure}
```

4 Using Julia, MatLab, and R

It is common to use multiple programming languages in one project. For example, Ji and Jin (2016) use R and MatLab to generate Figure 6 in that paper. The proposed approach is particularly suitable for this scenario as one can incorporate all the external codes for multiple programming languages in LaTeX. Here, as an illustration, we 1) use Julia to process the data and perform matrix operations; 2) use R and MatLab functions provided by the authors of Ji and Jin (2016) to find author clusters; and 3) use R to generate the figure. Since we use three different programming languages in this example, we use 'R', 'julia', and 'matlab' options when loading the *runcode* package, i.e., use `\usepackage[R,julia,matlab]{runcode}`.

To run the Julia code in file "Code/JiJin2016Julia.jl", we put the following in the tex file:

```
\runJulia{Code/JiJin2016Julia.jl}{initjulia}
```

The Julia code processes the data and finds the 15 authors' names for Figure 6 of Ji and Jin (2016). The 15 authors' names are stored in a variable called `top15`. We can use the `\inlnJulia` option to reproduce the second paragraph in Section 4.2 of Ji and Jin (2016) by writing the following in the tex file:

```
The giant component (236 nodes) is seen to be the ``High-Dimensional
Data Analysis [Coauthorship (A)]'' community (HDDA-Coau-A), including
(sorted descendingly by the degree) \includeOutput{initjulia}[inline],
etc. It seems that the giant component has substructures.
```

This produces the following:
'The giant component (236 nodes) is seen to be the "High-Dimensional Data Analysis [Coauthorship (A)]" community (HDDA-Coau-A), including (sorted descendingly by the degree) Peter Hall, Raymond J Carroll, Jianqing Fan, Joseph G Ibrahim, T Tony Cai, David Dunson, Hua Liang, Jing Qin, Donglin Zeng, Hans-Georg Muller, Hongtu Zhu, Enno Mammen, Jian Huang, Runze Li, Song Xi Chen, etc. It seems that the giant component has substructures.'

Now we run the MatLab code provided in Ji and Jin (2016). The main file is "MatlabCode.m", and we run it from LaTeX by including the following in the LaTeX source file:

```
\runMatlab{Code/MatlabCode.m}{matlabinterim}
```

We do not need to include any direct results from the MatLab code in the paper. Now we run the R code in "JiJin2016Plot6.R" to create Figure 6 of Ji and Jin (2016) by

```
\runR{Code/JiJin2016Plot6.R}{JiJin2016fig6}
```

The resulting plots are shown in Figure 3. Due to the updated `igraph` package, the resulting figures are not identical to Figure 6 of Ji and Jin (2016), but the clusters and connections are the same.

5 Conclusion

To easily reproduce results of software-based scientific analyses we have developed a LaTeX-based approach which automatically runs the code associated with the research and embeds its output when the document is compiled into a pdf file. Our approach is very general, in that any programming language or command-line tool can be used, while allowing LaTeX users to continue to write their paper with their favorite text editor.

In this paper we explained in detail how to use our method, and described its important features, which include the ability to run external code, as well as to embed code directly in the tex file; the flexibility to choose the output format in the final document (inline, pure LaTeX, or verbatim, as the case warrants); and how to use it efficiently, by using cached results when needed.

The method is based on the so-called 'shell-escape' built-in functionality in the LaTeX engine. In doing so, a user has to be mindful about the security implications, since shell-escape does not perform any code validation. However, this is a non-issue *unless* the user who runs the LaTeX compiler has a highly privileged account, *and* if the source of the code is dubious and unverified by the user. This is an unlikely combination of circumstances, and can be easily avoided by using best practices. Furthermore, the 'shell-escape' option is required by many widely used LaTeX packages such as the *TikZ* and *PGF* packages for creating graphics, and the *minted* package we adopted in this paper for syntax highlighting.

Finally, we point out that the method presented in this paper is broadly applicable, and not limited to just compiling LaTeX documents. For example, using the same infrastructure, namely the *talk2stat* package, one can build a web-based graphical user interface or dashboard to perform statistical analysis. To demonstrate this capability, we have implemented a very simple web application where a user can perform a binary classification of a given dataset, using one of seven different classifiers (4 in R and 3 in Julia). The source code for the demo is available on github (see the Supplementary Material section below). In this example, Node.JS is used for the front-end, and connects to R and Julia in the same way that was demonstrated in this paper

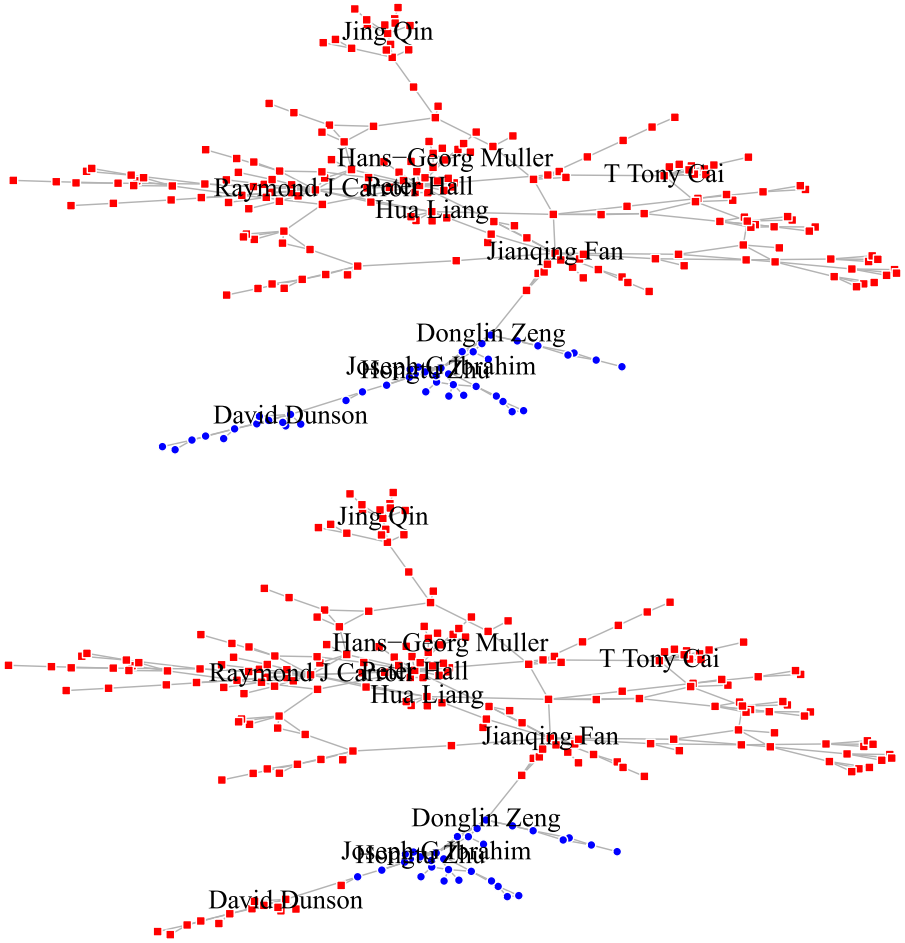

Figure 3: Reproducing Figure 6 from Ji and Jin (2016).

with pdflatex. The *shiny* package in R (Chang et al., 2020) offers a similar functionality, but our approach is more flexible, in that it allows the developers to use their preferred tools, both for the front-end (e.g., Node, React, etc.) and the back-end (R, Julia, MatLab, Python). As such, it is possible to develop smart-phone apps which can perform heavy computational tasks, using server-side statistical software.

Supplementary Material

Detailed examples and documentation are available in a GitHub repository at https://github.com/Ossifragus/runcode. The runcode LaTeX package is available via CTAN, and the *talk2stat* Python package for the server-mode is available at https://pypi.org/project/talk2stat/.

Appendix A Computational Details

The following script includes general system information, software versions (Mac), R, MatLab, and Julia versions, as well as package versions (*Matrix*, *ineq*, and *igraph*, which we used earlier.)

```
date
echo

# get system configuration for the latex paper, Bar and Wang
# Mac version

echo "System Information"
echo "=================="
system_profiler  SPHardwareDataType
echo

echo "Software versions"
echo "================="
sw_vers
echo

echo "R session information"
echo "====================="
Rscript -restore -e 'library("Matrix"); library("ineq"); library("igraph"); sessionInfo()'
echo

echo "Matlab version"
echo "=============="
matlab -n | grep ' MATLAB ' | cut -d= -f2
echo

echo "Julia version"
echo "============="
julia -v
echo

echo "pdfTex version"
echo "=============="
pdflatex -version
```

```
Thu Dec 17 17:11:21 EST 2020

System Information
==================
Hardware:

    Hardware Overview:

      Model Name: MacBook Pro
      Model Identifier: MacBookPro12,1
      Processor Name: Intel Core i5
      Processor Speed: 2.9 GHz
      Number of Processors: 1
      Total Number of Cores: 2
      L2 Cache (per Core): 256 KB
      L3 Cache: 3 MB
      Hyper-Threading Technology: Enabled
```

```
      Memory: 8 GB
      Boot ROM Version: 425.0.0.0.0
      SMC Version (system): 2.28f7
      Serial Number (system): C02QV69SFVH7
      Hardware UUID: 1E16C4B3-9D5B-55BA-B1BE-24DBDCEC5315


Software versions
=================
ProductName:        Mac OS X
ProductVersion:        10.14.6
BuildVersion:        18G7016

R session information
=====================
R version 4.0.1 (2020-06-06)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Mojave 10.14.6

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib

locale:
[1] C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] igraph_1.2.5  ineq_0.2-13   Matrix_1.2-18

loaded via a namespace (and not attached):
[1] compiler_4.0.1  magrittr_1.5    grid_4.0.1      pkgconfig_2.0.3
[5] lattice_0.20-41

Matlab version
==============
 /Applications/MATLAB_R2020a.app

Julia version
=============
julia version 1.4.0

pdfTex version
==============
pdfTeX 3.14159265-2.6-1.40.21 (TeX Live 2020)
kpathsea version 6.3.2
Copyright 2020 Han The Thanh (pdfTeX) et al.
There is NO warranty.  Redistribution of this software is
covered by the terms of both the pdfTeX copyright and
the Lesser GNU General Public License.
For more information about these matters, see the file
named COPYING and the pdfTeX source.
Primary author of pdfTeX: Han The Thanh (pdfTeX) et al.
Compiled with libpng 1.6.37; using libpng 1.6.37
```

```
Compiled with zlib 1.2.11; using zlib 1.2.11
Compiled with xpdf version 4.02
```

References

Allaire J, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, et al. (2020). *rmarkdown: Dynamic Documents for R*. package version 2.1.

Bates D, Maechler M (2019). *Matrix: Sparse and Dense Matrix Classes and Methods*. R package version 1.2-16.

Chang W, Cheng J, Allaire J, Xie Y, McPherson J (2020). *shiny: Web Application Framework for R*. R package version 1.5.0.

Csardi G, Nepusz T, et al. (2006). The igraph software package for complex network research. *InterJournal, complex systems*, 1695(5): 1–9.

Gentleman R, Temple Lang D (2007). Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1): 1–23.

Ioannidis JPA (2005). Why most published research findings are false. *PLOS Medicine*, 2(8).

Ji P, Jin J (2016). Coauthorship and citation networks for statisticians. *Annals of Applied Statistics*, 10(4): 1779–1812.

Jin J (2015). Fast community detection by SCORE. *Annals of Statistics*, 43(1): 57–89.

Knuth DE (1984). Literate programming. *The Computer Journal*, 27(2): 97–111.

Leisch F (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In: *Compstat*, 575–580. Springer.

Schulte E, Davison D, Dye T, Dominik C (2012). A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3): 1–24.

Xie Y (2014). knitr: A comprehensive tool for reproducible research in R. In: *Implementing Reproducible Computational Research* (V Stodden, F Leisch, RD Peng, eds.). Chapman and Hall/CRC. ISBN 978-1466561595.

Xie Y (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition edition. ISBN 978-1498716963.

Xie Y (2020). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.28.

Xie Y, Allaire J, Grolemund G (2018). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 9781138359338.

Zeileis A (2014). *ineq: Measuring Inequality, Concentration, and Poverty*. R package version 0.2-13.