2

Making research code useful paradata

Richèl J.C. Bilderbeek

Abstract Paradata is data about the data collection process, that allows use and reuse of data. Within the context of computational research, computer code is the paradata of an experiment, allowing the study to be reproduced. A recent study recommended how to make paradata (more) useful, for paradata in general. This study applies those recommendations to computer code, using the field of genetic epidemiology as an example. The chapter concludes by some rules how to better code to serve as paradata, and hence allowing computational research to be more reproducible.

Keywords: paradata, reproducible research, code, software, computational research, Open Science, best practices, genetic epidemiology

2.1 Introduction

> Talk is cheap. Show me the code.
>
> Linus Torvalds, 2000-08-25

Two different researchers in genetic epidemiology (more on that field later), write two equally good manuscripts that describe an experiment with computational steps. Both manuscripts are accepted by an equally prestigious journal after peer review. One researcher, however, does not supply the computer code (from now on: 'code') that was used to generate the results, where the other does. Are the conclusion of these papers to be trusted equally? Is this difference relevant and worth the effort? How common is it to share code, and, if shared, how can it be preserved? This chapter discusses why code, a type of paradata, should be supplied and what features it needs to have for it to be useful.

The concept of paradata (although not named as such yet) was introduced by Couper and colleagues, who developed a computer-assisted interview program that, among others, records all key strokes, measures the time used to answer each question and even the time the monitor is turned off (Couper [1998]). The goal in that context was to assess and improve survey quality.

There exists no standard definition of paradata (Nicolaas [2011], Sköld et al. [2022], Huvila [2022]). Without declaring it a formal definition, however, paradata can be understood in general sense as 'data about processes' (with e.g. survey paradata termed alternatively as records and audit trails) Nicolaas [2011], or in more specific sense in relation to data collection, as 'data about the data collection process' (Choumert-Nkolo et al. [2019]). This chapter uses the latter as a working definition.

The code used in computational experiments is just that, 'data about the data collection process', as it commonly downloads data, selects relevant subsets in those

data, performs statistical tests and generates figures. In each case, code answers the question: 'Where is it (i.e. the result) coming from?'. Code, hence, is paradata and this chapter explores and illustrates the consequence of that premise.

A recent paper explores the use of paradata to increase the impact of data, stating that lack of paradata can be seen as 'a drastic constraint' in the use of data and offer some suggestions to make paradata useful for data re(use): paradata should be comprehensive, documented in a useful way, the documentation and data should have co-evolved, and the paradata should be computer-friendly (Huvila [2022]). At first glance, these suggestions appear to work well for code and will be discussed in detail below.

There are multiple reasons why useful paradata matter. Most obvious is that having useful paradata gives an understanding how data is produced. This knowledge helps researchers from different fields to understand each other and collaborate. Additionally, useful open data is needed for Open Science to convincingly show its benefits. Finally, in computational fields, it can help understand how scholarly knowledge is produced (Huvila [2022]).

This chapter discusses these general reasons applied to code and its implications for knowledge management, including recommendations how to make code useful paradata in section 2.6.

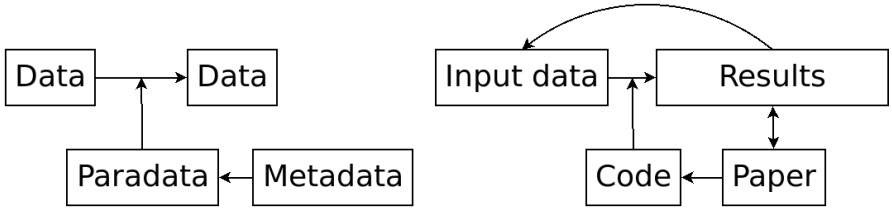

Fig. 2.1 **Left**: general relation between data, paradata and metadata. **Right**: the same relations specified for genetic epidemiology. Input data: genetic data, phenotypic data and associations. found in earlier studies. Results: associations between genetic data and phenotypic data. Code: the computer code used in a computational experiment. Paper: the scholarly paper describing the (code of the) experiment.

2.2 Code availability

To determine how useful code is, it needs to be available. However, it is not common to publish the code of an experiment or analysis (Stodden [2011], Read et al. [2015]) (with a pleasant exception being Conesa and Beck [2019]). For example, in computer graphics, a field intimately familiar with computer code, 42% of 454 SIGGRAPH

papers supply computer code (Bonneel et al. [2020]). Another study analysed the reproducibility of registered reports in the field of psychology, where 60% of 62 studies supplied the code to redo the analysis (Obels et al. [2020]). For articles in Science magazine, 12% of 204 studies published the code (Stodden et al. [2018]) (note that these were years 2011-2012). Unpublished code has been the cause of some saddening examples, such as an algorithm that detects breast cancer from images better than a human expert, yet failed to ever be reproduced (Haibe-Kains et al. [2020]).

When code of an academic paper is not published, one could contact the corresponding author and request it. However, the response rate of corresponding authors with a request for data in computational fields is around 50% (Manca et al. [2018], Stodden et al. [2018], Teunis et al. [2015]) (the field of emergency medicine seems to be a pleasant outlier, with 73% of 118 emails being replied (O'Leary [2003])). When getting an answer of the corresponding author, 48% out of 134 replied emails will actually result in a sharing of the code (Stodden et al. [2018]). The responses of unwilling authors (see Stodden et al. [2018] for some real examples) can come across as so caustic, that one may be excused from not contacting an a corresponding author.

2.3 Genetic epidemiology

This chapter uses genetic epidemiology as a specific example, to illustrate in which way code is paradata, and why this type of paradata is relevant. However, any field that uses computation and sensitive data in its experiments, could be used as an example.

Genetic epidemiology is a field within biology that tries to determine the spread of heritable traits and their underlying biological mechanism. For example, we know that lactose intolerance in adults is caused by a decline in the production of lactose-degrading enzymes, and is most commonly found in south-east Asia and south Africa (Storhaug et al. [2017]). The trait is caused by the genetic make-up, or 'genotype' , of a person. The trait, also called 'phenotype', in this example is lactose intolerance at adult age, yet any human property, such as weight or height can be studied. When an association between genotype and phenotype is found, these associations, when relevant enough, are used to create so-called 'gene panels', where the location of the gene causing an association is measured specifically, to detect people at risk for the associated phenotype. The rest of this section describes a genetic epidemiology study in more detail, with special focus on the computational experiment.

The example study followed is a pseudorandomly selected paper from Ahsan et al. [2017]. The primary data used by that paper is from a population study called the

Northern Swedish Population Health Study (NSPHS) that started in 2010 (Igl et al. [2010]). The approximately 1000 participants were initially mostly surveyed about lifestyle (Igl et al. [2010]) and follow-up studies provided the type of data relevant for this paper, which are (1) the genotypes (Johansson et al. [2013]), (2) the phenotypes, in this case, concentrations of certain proteins in the blood (Enroth et al. [2014, 2015]).

The first type of primary data, the genotypes, consists out of single nucleotide polymorphisms (SNPs, pronounced 'snips'). A SNP has a name and a location within the genome. At the SNP's location in the DNA, there will be two nucleotides. One of these nucleotides is inherited from the mother, the other from the father. The DNA consists out of billions of nucleotides. There are four types of nucleotides, called adenosine, cytosine, guanine and thyrosine, commonly abbreviated as A, C, G and T respectively.

One SNP example is `rs12133641`, which is a SNP located at position 154,428,283 (that is, at the 154 millionth nucleotide), where 67 percent of the people within this study have an A, and 33 percent have a G (also from Ahsan et al. [2017], Table S3). From this it follow (assuming the nucleotides are inherited independently) that 45% of subjects have the genotype AA, 44% have AG and 11% have GG.

The second type of data, the phenotypes, are concentrations of proteins in the blood. The nucleotides of the DNA contain the code for building proteins , as well as the rate at which a protein is created (for sake of simplicity it is assumed that such a rate is constant, yet, in practice there are complex regulation mechanisms). Some proteins end up in the blood and their presence can be used to assess the health of an individual. IL6RA is one such protein and its concentration may (and will, see below) be associated with the SNP mentioned earlier.

The field of genetic epidemiology looks -among others- for correlations between genetic data and biological traits. For example, Ahsan and colleagues show that SNP `rs12133641` is highly correlated (p-value is 3.0^{-73} , for n = 961 individuals) with protein IL6RA (Ahsan et al. [2017]). The direction of the association is also concluded: the more guanines are present at that SNPs location, the higher concentration of IL6RA can be found in a human's blood . The amount of variance that can be explained by an association (i.e. the R^2) is rarely 100%, which means that a trait (in this case, the concentration of IL6RA) cannot be perfectly explained from the genotype (in this case, SNP `rs12133641`) alone. In this example, as much as 43% of the variance can be attributed to an individuals' genotype . Additional factors, such as the effect of the environment (e.g. geographic location, time of day the measurement was done), lifestyle (e.g. smoking yes/no) or having a disease (e.g. diabetes) are needed to explain the additional variation.

The conclusions drawn from this paper, may end up in the clinic. For the sake of having a clear (yet fictitious) example, let's assume that a high level of IL6RA is associated with a disease that develops later in life, yet is preventable by lifestyle

changes (see Pope et al. [2003] for an example in Alzheimer's disease). Would this be the case, we can create a tailored experiment, called a gene panel, that specifically measures SNP `rs12133641`. If the gene panel shows an individual has two guanines, we know that this person is likelier to develop higher levels of IL6RA and is likelier to benefit from the lifestyle changes.

From this simple example, it will be easier to measure the level of IL6RA in the blood, than using a gene panel, as blood tests are easier and cheaper. However, there are associations published for many diseases, in which one SNP (e.g. phenylketonuria) or many SNPs (e.g. Bruce and Byrne [2009]) contribute to being more likely to develop a disease in the future. Here, the phenotype (having a disease in the future) is impossible to detect at the present and associations found in earlier studies are used to create a gene panel. As creating a gene panel is costly, those associations better be correct.

Additionally, there is an interdependency of scholarly findings here: the SNP has received its name based on a computational experiment. This earlier experiment that concluded the usefulness of that SNP, is based on some DNA sequences. This experiment is based on assumed DNA sequences. DNA sequences, however, are (nowadays) not simply read, yet are the result of a complex computational analysis instead, with its own dedicated field of research. Both studies assumed a correctly calculated DNA sequence. This means that if the DNA sequence analysis contained a software bug, this study may be invalidated. Additionally, the result of this study may be used in follow-up studies, that assume the result to be correctly calculated: one paper's conclusion is the next paper's assumption.

2.4 Why code is useful paradata

The experiment described above is run by code. It was code that detected the relationship between the genotype (in this case, SNP `rs12133641`) and the phenotype (in this case, the concentration of IL6RA). To be more precise, it was code that read the data, subsetted the data, removed outliers, performed the statistics and generated the plots. For the rest of the discussion, we assume that the code is available to us (if not, see section 2.2 for a glum estimate of the chance of obtaining the code).

There are multiple reasons why (useful) code matters, and these are the same reasons as why useful paradata matters: code gives an understanding how the raw results and the subsequent scholarly knowledge is obtained from an experiment. Additionally, code helps researchers from different fields to understand each other and collaborate. Additionally, code helps Open Science reach its goals of openness and transparency. The core of these reasons it to achieve reproducible science: that any person in any field can redo a computational experiment and see exactly what happened.

For computational science, it may appear to be relatively easy to reproduce an experiment, as all it takes is a computer, electricity, an optional internet connection, the code and the data. In practice, however, only 18% of 180 computational studies are easily reproducible (Stodden et al. [2018]). To some, it appears that the academic culture to reproduce results has been lost over time Peng [2011], with labs that embrace reproducibility (for example, Barba [2016]) being the exception. One suggested way forward is to make the reproduction of research a minimal requirement for publication (Peng [2011]).

A genetic epidemiologist works with sensitive data as well: the genetic sequences of participants are private (Clayton et al. [2019]). For research to be reproducible one needs both the code and the data to reproduce the (hopefully) same results. This problem is discussed in section 2.7.

Code holds the ground truth of an experiment; it does the actual work. The more complex the computation pipeline is, the easier it is to have a mismatch between the article (that describes what the code does) and the code (that actually does the work). The moment these two disagree, it is the code that is true.

2.5 Preserving code

Code is rarely preserved (Barnes [2010]). This section discusses the preservation of code for a short, medium and long term.

2.5.1 Code hosting

To preserve code for a short term, a code hosting website is a good first step. A code hosting website is a website where its users can create dedicated pages (called 'repositories') for a project, upload code and interact with that code. There are multiple code hosting websites, with GitHub being the most popular one (Cosentino et al. [2017]). The use of code hosting websites has increased strongly (Russell et al. [2018]), accommodates collaboration (Perez-Riverol et al. [2016]) and improves transparency (Gorgolewski and Poldrack [2016]), due to its inherent computer-friendliness. See Cosentino et al. [2017] for an extensive overview of research conducted on GitHub.

Hosted code commonly keeps a history of file changes, This means that when a change is made to the code, a new version is created. In the case that the change

was harmful, one can go back to an earlier version and continue again from there. The version control system keeps track of who-did-what transparently. It is a general recommendation to put version control on all human-produced data (Wilson et al. [2014]), as well as openly working on the code from the start (Jiménez et al. [2017]). Half of the published code has such a version control system (Stodden et al. [2018]).

The degradation of software is a known feature for nearly four decades. This is called 'bit rot' by Steele Jr et al. [1983], or 'software collapse' by Hinsen [2019], in which software fails due to dependencies on other software. Using a code hosting website, that only passively stores code, ignores this problem.

2.5.2 Continuous integration

To preserve code for a longer timespan, it needs to be embraced that software degrades (Beck [2000]). Continuous integration ('CI') allows one to verify if code still works and, if not, to be notified.

Some code hosts, allow a user to trigger specialised code upon uploading a change, called a CI script. Such a CI script typically builds and tests the code. This practice is known to significantly increase the number of bugs exposed (Vasilescu et al. [2015]) and increases the speed at which new features are added (Vasilescu et al. [2015]). CI can be scheduled to run on a regular basis and notify the user directly when the code has broken down.

2.5.3 Containerisation

To preserve code for the longest time, both code and its dependencies can be put into a so-called container. The most reproducible way of submitting the code of an experiment, is to put all code with all the (software) dependencies in a file that acts as a virtual computational environment, called a 'virtual container' (from now on: 'container'). Such a container is close to the golden standard of reproducible research as suggested by Peng [2011] .

2.6 Making code useful paradata

Useful paradata, in general, are (1) comprehensive, (2) documented appropriately, (3) documented in co-evolution with the data, and (4) friendly to computers (Huvila [2022]). In this section, these ideal properties of paradata are applied to code. However, code has multiple uses, as code can be used to (1) reproduce, (2) replicate, or (3) extend a computational experiment (Benureau and Rougier [2018]). Depending on the intended use of the code, there are different requirements for code being useful paradata.

2.6.1 Code must be usefully documented

For code to be ideal paradata, it must be usefully documented (Huvila [2022]). For purposes of reproducing code, it should at least be documented how to run the code and what it ought to do. Although this may be obvious, only 57% out of 56 Science papers with obtainable code (in total there were 180 papers) do so (Stodden et al. [2018]).

When it can be found out how how an experiment is run, it is possible (even ideal!) that no code is read at all. Within that context, it could argued that no (further) documentation is needed. However, all code in general should be documented 'adequately' (Peng et al. [2006]), ideally writing code in such a way that it becomes self-explanatory (Wilson et al. [2014]) and for the remaining code to document the reasons behind it, its design and its purpose (Wilson et al. [2014]).

For purposes of extending a study and its code, documentation becomes even more important, as the code will be read and modified. The extent of investing time in documenting code is recommended to be proportional to the intended reuse (Pianosi et al. [2020]) and there exists a clear relationship between the reuse of code and its documentation effort (Cosentino et al. [2017], Hata et al. [2015]).

2.6.2 Code and documentation must align

For code to be ideal paradata, its creation and documentation need to align, *not least because they shape each other* (Huvila [2022]). This emphasised part of the quote resonates strongly with the idea of literate programming (Knuth [1984]), in which documentation and code are developed hand-in-hand. Literate programming is the

practice of writing code and documentation in the same file. Contemporary examples of this idea are, among others, vignettes (Wickham [2015]) for the R programming language (R Core Team [2021]) and Jupyter notebooks (Wang et al. [2020]) for the Julia (Bezanson et al. [2017]), Python (Van Rossum and Drake Jr [1995]) and R (R Core Team [2021]) programming languages. rOpenSci, a community that, among others, reviews programming code (Ram [2013], Ram et al. [2018]), is one example of where extensive documentation is mandatory and all code must have examples (that are actually run) as part of the documentation (rOpenSci et al. [2021]). In general, when developing software, it is recommended to to write documentation while writing software, as well as to include many examples (Lee [2018]), as this leads to both better code and documentation (Reenskaug and Skaar [1989]).

2.6.3 Code must be extensive

For code to be ideal paradata, it must be extensive. As code has many properties, there are many recommendations on this aspect.

Code should be distributed in standard ways (Peng et al. [2006]), as is done by using a code hosting website (see subsection 2.5.1). Additionally, code must be more extensive when it is (intended to be) used on different data, as then 'code must act as a teacher for future developers' (Sadowski et al. [2018]). Error handling is one of the mechanisms to do so. In genetic epidemiology, it is common to have incomplete or missing data, so analyses should take this into account with clear error messages.

Coding errors are extremely common (Baggerly and Coombes [2009], Vable et al. [2021]) and contribute to the reproducibility crisis in science (Vable et al. [2021]). Testing, in general, is an important mechanism to ensure the correctness of code. One clear example is Rahman and Farhana [2020], showing bugs in scientific software on the COVID-19 pandemic.

Testing is so important that it is at the heart of a software development methodology called 'Test-Driven Development' ('TDD'), in which tests are written before the 'real' code. TDD improves code quality (Alkaoud and Walcott [2018], Janzen and Saiedian [2006]) and it is easy to integrate the writing of documentation as part of the TDD cycle (Shmerlin et al. [2015]).

The percentage of (lines of) code tested is called the code coverage. Code coverage correlates with code quality (Horgan et al. [1994], Del Frate et al. [1995]). and, due to this, having a code coverage of (around) 100% is mandatory to pass a code peer-review by rOpenSci (Ram et al. [2018]). When CI is activated, the code coverage of a project can be shown on the repository's website .

It is considered good practice to add a software license (Jiménez et al. [2017]), so that it is clear that the software can be reused. Although this may seem trivial, only two-thirds of 56 computational experiments supply a software license (Stodden et al. [2018]).

Code reviews are recommended by software development best-practices (Wilson et al. [2014]). However, more than half of 315 scientists have their code rarely or never reviewed (Vable et al. [2021]), although code reviews are known to accelerate learning of the developers, improve the quality of the code and resulting to an experiment that is likelier to be reproducible (Vable et al. [2021]).

2.6.4 Code must be computer friendly

The most reproducible way of submitting the code of an experiment, is by providing the code with all its (software) dependencies in a container. Containers allow a computation experiment to be highly reproducible: given the same data, an experiment put into a container will give the same results on different platforms, at least in theory. In practice, differences may be observed when peripheral factors are different, such as the random numbers as generated by an operating system, or data that are downloaded from online (and hence, probably changing) sources.

For paradata to be useful, it has to be computer-friendly, yet 'the best paradata does not necessarily look like 'data' at all for its human users' Huvila [2022]. There are features of code that humans find useful, without directly being able to measure these. In the end, code is just 'another kind of data' and should be designed as such, for example, by using tools to work on it (Wilson [2022]).

A first example is to use a tool to enforce a coding style (e.g. the Tidyverse style guide (Wickham [2019]) for R, or PEP 8 (Van Rossum et al. [2001]) for Python), as following a consistent coding style improves software quality (Fang [2001]). A second example is to use a tool to enforce a low cyclomatic complexity. The cyclomatic complexity is approximately defined as the number of independent paths that the code can be executed. The cyclomatic complexity correlates with code complexity, where more complex code is likelier to contain or give rise to bugs (Abd Jader and Mahmood [2018], Chen [2019], Zimmermann et al. [2008]).

2.7 Sensitive data

Next to the code, it is the data used in an experiment that must be made available for an experiment to be called 'reproducible' (Peng et al. [2006]). In some fields, such as genetic epidemiology, the data are sensitive, hence cannot be released, thus one cannot reproduce an experiment. To solve this problem in the future, there are some interesting methods being developed to run code on sensitive data with assured privacy (Zhang et al. [2016], Azencott [2018]).

To alleviate the problem today, a developer should supply a simulated (also called 'analytical' (Peng et al. [2006])) dataset together with the code. This simulated data is needed to run tests, as is part of the TDD methodology. In the case of genetic epidemiology this would mean simulated genotypes and associated phenotypes, as can be done with the `plinkr` R package (Bilderbeek [2022]). One extra benefit of simulated data is that these can be used as a benchmark, as slightly different analyses should give similar conclusions.

2.8 Discussion

In a perfect world, all code has the characteristics of ideal paradata and is written from software development best practices. This section discusses the problems that arise by doing so.

To know these best practices, one needs to be trained. Articles that suggest these best practices (such as this one), claim that this initial investment pays off. Code reviews are a good way to accelerate the learning of team members (Vable et al. [2021]).

Code needs maintenance, as code that will stand the test of time perfectly is deemed 'impossible' (Benureau and Rougier [2018]). CI can help a maintainer to be notified when the code breaks, where the use of containers may slow down time, as an entire computational environment is preserved.

Uploading code, preferably to a code hosting website, may feel like a risk, as all code can be seen and scrutinised. However, not publishing code may put oneself in the focus of attention and -after much effect by others reproducing an incorrect result- at the cost of a scientific career (Baggerly and Coombes [2009]).

When the author of code can be contacted, there will be users asking for technical support. One solution for the author is to ignore such emails, as is done in a third of 357 cases (Teunis et al. [2015]): it can be argued that no energy should be wasted

on published code and work on something new instead. However, see Barnes [2010] for a better way to deal with this problem.

When the author of code can be contacted, users will send in bug reports. If the bug is severe enough, the question arises if all the research that use that code still results in the same conclusions. One such bug is described in Eklund et al. [2016], with 40,000 studies using that incorrect code. These reports could be ignored to work on something new.

Containers do have problems. First, they themselves require software to run, with the same software decay being possible. Second, one needs to install that software and have the computer access rights (i.e. admin rights under Windows, or root rights under Linux) to do so. Third, one needs to learn how to build and use containers. Lastly, containers can be several gigabytes big files, which makes their distribution harder. Ideally, containers are stored online and distributed in standardised ways. Although progress is being made, there is no way to do so for all container types. Additionally, probably due to their novelty, container hosting sites lack metadata.

2.9 Conclusions

This chapter started with some suggestions to make paradata useful for data re(use): paradata should be extensive, comprehensively documented, with the creation of documentation and code going hand-in-hand, as well as friendly to computers (Huvila [2022]).

Before applying these features, the first step is to publish the code. When applying these general recommendations to code, this list can be phrased more precisely:

1. Code should be comprehensive in supplying automatically generated metadata (such as commit history and code coverage).
2. The documentation should be as extensive as recommended by the software development literature.
3. The documentation should have co-evolved with the code following the best practices in literate programming.
4. Code should be made machine-readable by, at least, being uploaded to a code hosting website. Ideally, the code is checked by CI and is put in a container.

For the preservation of code, these recommendations are made:

1. Uploading code to a code hosting website is better than not publishing code at all.

2. Adding CI to code allows one to detect the day when that code does not run anymore.

3. Putting the code in a container is the best way to preserve code.

When research truly needs to be reproducible, putting the code of an experiment into a container is today's best solution, as containers are the best solution to keep code running for the longest amount of time. Creating such a container, however, requires more skill that -as of today- is not rewarded, although an experiment put into a container can be considered the pinnacle of reproducible research.

The simplest and most impactful step to make code more useful paradata is, however, to publish it on a code hosting website along a publication. From then on, the next steps can be taken gradually as the skills of the author(s) progress. To quote Barnes [2010]: Publish your code, it is good enough.

The world of science would be a more open, humble, trustworthy, truthful and helpful would the code that accompanies a scientific paper be treated like a first class citizen. Doing so, however, is yet to be rewarded and still both of the two scientists at the start of this paper can provide a good rationale for their behaviour. This will change when reward incentives are put into place that reward making paradata useful. For code specifically, in any computational field, the rewards are even higher, as reproducibility should again be a cornerstone in science.

2.10 Data Accessibility

This article and its metadata can be found at `https://github.com/richelbilderbeek/chapter_paradata`.

References

Mick Couper. Measuring survey quality in a casic environment. *Proceedings of the Survey Research Methods Section of the ASA at JSM1998*, pages 41–49, 1998.

Gerry Nicolaas. Survey paradata: a review. 2011.

Olle Sköld, Lisa Börjesson, and Isto Huvila. Interrogating paradata. 2022.

Isto Huvila. Improving the usefulness of research data with better paradata. *Open Information Science*, 6(1):28–48, 2022.

Johanna Choumert-Nkolo, Henry Cust, and Callum Taylor. Using paradata to collect better survey data: evidence from a household survey in tanzania. *Review of Development Economics*, 23(2):598–618, 2019.

Victoria C Stodden. Trust your science? Open your data and code. 2011.

Kevin B Read, Jerry R Sheehan, Michael F Huerta, Lou S Knecht, James G Mork, Betsy L Humphreys, and NIH Big Data Annotator Group. Sizing the problem of improving discovery and access to NIH-funded data: a preliminary study. *PLoS One*, 10(7):e0132735, 2015.

Ana Conesa and Stephan Beck. Making multi-omics data accessible to researchers. *Scientific data*, 6(1):1–4, 2019.

Nicolas Bonneel, David Coeurjolly, Julie Digne, and Nicolas Mellado. Code replicability in computer graphics. *ACM Transactions on Graphics (TOG)*, 39(4):93–1, 2020.

Pepijn Obels, Daniel Lakens, Nicholas A Coles, Jaroslav Gottfried, and Seth A Green. Analysis of open data and computational reproducibility in registered reports in psychology. *Advances in Methods and Practices in Psychological Science*, 3(2): 229–237, 2020.

Victoria Stodden, Jennifer Seiler, and Zhaokun Ma. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, 115(11):2584–2589, 2018.

Benjamin Haibe-Kains, GA Adam, Ahmed Hosny, Farnoosh Khodakarami, MS Board, Levi Waldron, Bo Wang, Chris Mcintosh, Anshul Kundaje, Casey Greene, et al. The importance of transparency and reproducibility in artificial intelligence research. 2020, 2020.

Andrea Manca, Lucia Cugusi, Zeevi Dvir, and Franca Deriu. Non-corresponding authors in the era of meta-analyses. *Journal of clinical epidemiology*, 98:159–161, 2018.

Teun Teunis, Sjoerd PFT Nota, and Joseph H Schwab. Do corresponding authors take responsibility for their work? A covert survey. *Clinical Orthopaedics and Related Research®*, 473:729–735, 2015.

F O'Leary. Is email a reliable means of contacting authors of previously published papers? A study of the Emergency Medicine Journal for 2001. *Emergency*

medicine journal, 20(4):352–353, 2003.

Christian Løvold Storhaug, Svein Kjetil Fosse, and Lars T Fadnes. Country, regional, and global estimates for lactose malabsorption in adults: a systematic review and meta-analysis. *The Lancet Gastroenterology & Hepatology*, 2(10):738–746, 2017.

Muhammad Ahsan, Weronica E Ek, Mathias Rask-Andersen, Torgny Karlsson, Allan Lind-Thomsen, Stefan Enroth, Ulf Gyllensten, and Åsa Johansson. The relative contribution of dna methylation and genetic variants on protein biomarkers for human diseases. *PLoS genetics*, 13(9):e1007005, 2017.

Wilmar Igl, Åsa Johansson, and Ulf Gyllensten. The Northern Swedish Population Health Study (NSPHS)–a paradigmatic study in a rural population combining community health and basic research. *Rural and remote health*, 10(2):198–215, 2010.

Åsa Johansson, Stefan Enroth, Magnus Palmblad, André M Deelder, Jonas Bergquist, and Ulf Gyllensten. Identification of genetic variants influencing the human plasma proteome. *Proceedings of the National Academy of Sciences*, 110(12): 4673–4678, 2013.

Stefan Enroth, Åsa Johansson, Sofia Bosdotter Enroth, and Ulf Gyllensten. Strong effects of genetic and lifestyle factors on biomarker variation and use of personalized cutoffs. *Nature communications*, 5(1):1–11, 2014.

Stefan Enroth, Sofia Bosdotter Enroth, Åsa Johansson, and Ulf Gyllensten. Effect of genetic and environmental factors on protein biomarkers for common non-communicable disease and use of personally normalized plasma protein profiles (PNPPP). *Biomarkers*, 20(6-7):355–364, 2015.

Sandra K Pope, Valorie M Shue, and Cornelia Beck. Will a healthy lifestyle help prevent Alzheimer's disease? *Annual review of public health*, 24(1):111–132, 2003.

KD Bruce and CD Byrne. The metabolic syndrome: common origins of a multifactorial disorder. *Postgraduate medical journal*, 85(1009):614–621, 2009.

Roger D Peng. Reproducible research in computational science. *Science*, 334(6060): 1226–1227, 2011.

Lorena A Barba. The hard road to reproducibility. *Science*, 354(6308):142–142, 2016.

Ellen Wright Clayton, Barbara J Evans, James W Hazel, and Mark A Rothstein. The law of genetic privacy: applications, implications, and limitations. *Journal of Law and the Biosciences*, 6(1):1–36, 2019.

Nick Barnes. Publish your computer code: it is good enough. *Nature*, 467(7317): 753–753, 2010.

Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with GitHub. *IEEE Access*, 5:7173–7192, 2017.

Pamela H Russell, Rachel L Johnson, Shreyas Ananthan, Benjamin Harnke, and Nichole E Carlson. A large-scale analysis of bioinformatics code on GitHub. *PLoS One*, 13(10):e0205898, 2018.

Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe Leprevost, Christian Fufezan, Tobias Ternent, Stephen J Eglen, Daniel SS Katz, et al. Ten simple rules for taking advantage of git and GitHub. *bioRxiv*, page 048744, 2016.

Krzysztof J Gorgolewski and Russell Poldrack. A practical guide for improving transparency and reproducibility in neuroimaging research. *bioRxiv*, page 039354, 2016.

Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1): e1001745, 2014.

Rafael C Jiménez, Mateusz Kuzak, Monther Alhamdoosh, Michelle Barker, Bérénice Batut, Mikael Borg, Salvador Capella-Gutierrez, Neil Chue Hong, Martin Cook, Manuel Corpas, et al. Four simple recommendations to encourage best practices in research software. *F1000Research*, 6, 2017.

Guy L Steele Jr, Donald R Woods, Raphael R Finkel, Richard M Stallman, and Geoffrey S Goodfellow. *The hacker's dictionary: a guide to the world of computer wizards*. Harper & Row Publishers, Inc., 1983.

Konrad Hinsen. Dealing with software collapse. *Computing in Science & Engineering*, 21(3):104–108, 2019.

Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

Fabien CY Benureau and Nicolas P Rougier. Re-run, repeat, reproduce, reuse, replicate: transforming code into scientific contributions. *Frontiers in neuroinfor-*

matics, 11:69, 2018.

Roger D Peng, Francesca Dominici, and Scott L Zeger. Reproducible epidemiologic research. *American journal of epidemiology*, 163(9):783–789, 2006.

Francesca Pianosi, Fanny Sarrazin, and Thorsten Wagener. How successfully is open-source research software adopted? Results and implications of surveying the users of a sensitivity analysis toolbox. *Environmental Modelling & Software*, 124:104579, 2020.

Hideaki Hata, Taiki Todo, Saya Onoue, and Kenichi Matsumoto. Characteristics of sustainable oss projects: A theoretical and empirical study. In *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 15–21. IEEE, 2015.

Donald Ervin Knuth. Literate programming. *The computer journal*, 27(2):97–111, 1984.

Hadley Wickham. *R packages: organize, test, document, and share your code*. " O'Reilly Media, Inc.", 2015.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. URL `https://www.R-project.org/`.

Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 138–149, 2020.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671. URL `https://epubs.siam.org/doi/10.1137/141000671`.

Guido Van Rossum and Fred L Drake Jr. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

K Ram. rOpenSci-open tools for open science. In *AGU Fall Meeting Abstracts*, volume 2013, pages ED43E–04, 2013.

Karthik Ram, Carl Boettiger, Scott Chamberlain, Noam Ross, Maëlle Salmon, and Stefanie Butland. A community of practice around peer review for long-term research software sustainability. *Computing in Science & Engineering*, 21(2): 59–65, 2018.

rOpenSci, Brooke Anderson, Scott Chamberlain, Laura DeCicco, Julia Gustavsen, Anna Krystalli, Mauro Lepore, Lincoln Mullen, Karthik Ram, Noam Ross, Maëlle

Salmon, Melina Vidoni, Emily Riederer, Adam Sparks, and Jeff Hollister. rOpenSci Packages: Development, Maintenance, and Peer Review, November 2021. URL `https://doi.org/10.5281/zenodo.6619350`.

Benjamin D Lee. Ten simple rules for documenting scientific software, 2018.

Trygve Reenskaug and Anna Lise Skaar. An environment for literate Smalltalk programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 337–345, 1989.

Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

Keith A Baggerly and Kevin R Coombes. Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. *The Annals of Applied Statistics*, pages 1309–1334, 2009.

Anusha M Vable, Scott F Diehl, and M Maria Glymour. Code review as a simple trick to enhance reproducibility, accelerate learning, and improve the quality of your team's research. *American Journal of Epidemiology*, 190(10):2172–2177, 2021.

Akond Rahman and Effat Farhana. An exploratory characterization of bugs in covid-19 software projects. *arXiv preprint arXiv:2006.00586*, 2020.

Hessah Alkaoud and Kristen R Walcott. Quality metrics of test suites in test-driven designed applications. *International Journal of Software Engineering Applications (IJSEA)*, 2018, 2018.

David S Janzen and Hossein Saiedian. Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. *Acm Sigcse Bulletin*, 38(1):254–258, 2006.

Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. To document or not to document? an exploratory study on developers' motivation to document code. In *Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27*, pages 100–106. Springer, 2015.

Joseph R. Horgan, Saul London, and Michael R Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994.

Fabio Del Frate, Praerit Garg, Aditya P Mathur, and Alberto Pasquini. On the correlation between code coverage and software reliability. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 124–132. IEEE, 1995.

Greg Wilson. Twelve quick tips for software design. *PLoS Computational Biology*, 18(2):e1009809, 2022.

Hadley Wickham. *Advanced R*. CRC press, 2019.

Guido Van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8–style guide for Python code. *Python. org*, 1565, 2001.

Xuefen Fang. Using a coding standard to improve program quality. In *Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on*, pages 73–78. IEEE, 2001.

Marwa Najm Abd Jader and Riyadh Zaghlool Mahmood. Calculating McCabe's cyclomatic complexity metric and its effect on the quality aspects of software. 2018.

Changqi Chen. An empirical investigation of correlation between code complexity and bugs. *arXiv preprint arXiv:1912.01142*, 2019.

Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. Predicting bugs from history. In *Software evolution*, pages 69–88. Springer, 2008.

Lifang Zhang, Yan Zheng, and Raimo Kantoa. A review of homomorphic encryption and its applications. In *Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications*, pages 97–106, 2016.

C-A Azencott. Machine learning and genomics: precision medicine versus patient privacy. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376(2128):20170350, 2018.

Richèl J C Bilderbeek. , 2022. `https://github.com/richelbilderbeek/plinkr` [Accessed: 2022-08-25].

Anders Eklund, Thomas E Nichols, and Hans Knutsson. Cluster failure: Why fmri inferences for spatial extent have inflated false-positive rates. *Proceedings of the national academy of sciences*, 113(28):7900–7905, 2016.