

Detailed Implementation of a Reproducible Machine Learning-Enabled Workflow

[Kenneth E. Schackart III](#)^{1,2}, [Heidi J. Imker](#)^{1,3}, [Charles E. Cook](#)¹

¹ Global Biodata Coalition, 12 quai Saint-Jean 67080, Strasbourg, France

² Department of Biosystems Engineering, The University of Arizona, Tucson, Arizona 85721, USA

³ University Library, University of Illinois at Urbana-Champaign, Urbana, Illinois 31821, USA

Standfirst

Machine learning (ML) and advanced computational methods are powerful tools for processing and deriving value from large data volumes. These methods are being developed and deployed rapidly, but best practices are still evolving regarding code and data standards leading to irreproducibility of ML-enabled research. In this Comment we describe a ML enabled research project to create a global inventory of biodata resources, using it as a case study to discuss the efforts and decisions we made in our efforts to make this study reproducible.



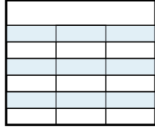
	Objectives	Tools & Methods
Reproducibility 	Data, model, and code availability Single-command dependency installation Ability to rerun entire analysis	GitHub, Zenodo Pip, conda, renv, Make Snakemake
Code Quality 	In-code documentation Readability and community compliance Bug prevention	docstrings, type hints pylint, flake8, lintr Unit tests, mypy
Data 	Findability Accessibility Interoperability Reusability	DataCite metadata GitHub, Zenodo PMIDs, ISO 3166 CSV formatting

Figure 1: Graphical abstract depicting the objectives of the study and the tools and methods used to address them regarding reproducibility, code quality, and data standards.

1. Introduction

There is broad concern over the lack of reproducibility in science^{1,2}, with many believing there is a crisis³. While the extent is contested^{4,5}, concerns about scientific reproducibility are ongoing, and flawed study designs and irreproducible analyses play a role. There have been efforts to encourage better practices, such as pre-publication of study protocols, analysis plans, and all code⁶. However, as argued in Haring 2018, while the different biases in production and reporting of research are largely identifiable and modifiable, continued methodological training for early career researchers is also crucial.

Use of machine learning (ML) in biosciences has proliferated so rapidly that it is difficult for adoption of good practices and proper training to keep pace. Open Science practices, such as public release of code and data, aim to remedy this⁷. While access to code and data are necessary for reproduction of computational results, such access does not guarantee that results can be reproduced. Indeed, the recent Ten Years Reproducibility Challenge investigated the ability to rerun code and reproduce results from projects ten years or older, and the issues involved resulted in a useful “reproducibility checklist”⁸. Additionally, efforts have been made to set standards for reproducible code, including for ML, and they serve as rubrics for assessing reproducibility⁹. What seems lacking, however, are detailed examples of practical implementations. This work provides such an example by explaining how a ML enabled study was planned and executed with reproducibility as an explicit goal from the onset of the project.

In our example, the study is a ML enabled inventory of biodata resources identified from the scientific literature. Biodata resources are biological, life sciences, and biomedical databases that archive research data generated by scientists, serving as the repositories of record for particular data types; as well as knowledge bases that add value by aggregation, processing, and expert curation. These resources are connected through extensive exchanges of data and form a distributed global infrastructure. They are crucial for the entire life science research endeavor and are used ubiquitously.

However, the infrastructure is not well-described: neither the number of resources nor their location has been systematically explored. A better understanding of the scale of the infrastructure, as provided by this inventory, will aid funders and other stakeholders in addressing challenges to sustainability faced by the infrastructure. The results of the project will be fully described elsewhere (manuscript in preparation¹⁰). Here we describe how we implemented a workflow that ensures reproducibility in future.

The reproducibility case study we describe here first utilizes the API of Europe PMC (Europe PMC.org)¹¹, which is a data resource that archives a large corpus of medical and life sciences publications¹². Europe PMC provides both individual (browser-based) and automated (API-based) queries. Our workflow starts with a targeted query to the Europe PMC API to retrieve the titles and abstracts of publications for which both a URL and the words “data,”

“database,” or “resource” are present in the title and/or abstract. The results of the query represent publications that might describe a biological (biodata) resource. A 10% random subset of publications from this initial result was manually classified as describing or not describing a biodata resource. Those that did describe a biodata resource were curated to label the resource’s common name (*e.g.*, PDB) and full name (*e.g.*, Protein Data Bank)¹³. Recently, BERT (Bidirectional Encoder Representations from Transformers) performed well on NLP tasks¹⁴. Several BERT models pre-trained on biomedical corpora were selected from huggingface.co and fine-tuned for the classification (predicting if the article describes a biodata resource) and named-entity recognition (predicting common and full name) tasks. Further downstream processing was performed, including URL extraction and http status checking. A full article describing the study is in preparation and during its preparation we realized that there are many, many additional details to share about how we have attempted to address reproducibility—details we wish we had found in the literature ourselves. Here we provide an in-depth overview with such details as an example of how to design and implement a reproducible workflow.

During the study, a strong emphasis was placed on Open Science, reproducibility, and robustness of the codebase and documentation for both philosophical reasons (in support of Open Science) and practical reasons (enabling future updating of the inventory). The entire process, from data splitting, model training and selection, to all downstream processing, is encapsulated in a Snakemake workflow¹⁵. This allows reproduction of the entire analysis with a single command. Strong standards of code quality were developed and are enforced through the use of static code checking and automated testing. Additionally, significant efforts were made to make all data products findable, accessible, interoperable, and reusable (FAIR)¹⁶.

When we began the project we turned to the literature for robust examples of reproducibility. However, we were unable to locate implementation details that mapped to our project and goals, so we developed our own workflow. We recognize that there are other ways of doing all of these tasks, and we offer this Comment as just one example of how to make a computationally-heavy study reproducible and open. We provide the reasoning behind the various considerations, which may be applicable to other research projects. We also provide specific examples of how those were realized in this study.

2. Have a Plan

“A goal without a plan is just a wish,” wrote Antoine de Saint-Exupéry in *The Little Prince*¹⁷. As with any other part of a research project, planning ahead makes the path to achieving reproducibility as smooth as possible. To this end, early in the project we developed an Open Science Implementation Plan¹⁸. In this document, we outlined the goals for reproducibility and how we planned to achieve them. These goals were organized into four

groups: reproducibility of methods, code standards, data standards, and external review/validation.

By considering these topics early in the project, we explicitly defined what expectations we had for our Open Science goals. Keeping these goals in mind helped ensure that the effort and resources required to obtain them was anticipated and considered a core aspect of the project. This minimized accumulation of technical debt that would have been time consuming and difficult to address near the end of the project.

3. Reproducibility of Methods

We found the reproducibility standards defined by Heil et al.⁹ useful for ranking reproducibility levels. In our case, bronze alone was not acceptable (data published and downloadable, models published and downloadable, source code published and downloadable). Obtaining silver was acceptable (bronze + dependencies set up in a single command, key analysis details recorded, analysis components set to deterministic) but the gold standard was our goal (silver + entire analysis reproducible with a single command).

3.1. Meeting the Bronze Standard

The bronze standard of reproducibility is characterized by having the following published and downloadable: all data necessary for reproduction, trained models, and source code.

Data availability and, more broadly, FAIRness (findability, accessibility, interoperability, and reusability) will be further discussed in a later section. To address the minimum set out in the bronze standard, all data will be available for download from the project's Github and Zenodo repositories.

Model availability is addressed in a few ways. All of the models used in this project were pre-trained by other groups and made available on HuggingFaceHub (HFHub, <https://huggingface.co/>). As part of model training, these pretrained models were fine-tuned to various tasks (sequence classification and token classification). We will make these fine-tuned models available in a Zenodo archive, as well as on HFHub.

All source code is stored in two places. First, GitHub serves as a living repository. An important aspect of Open Science is providing a place for open discussion (and criticism) of methods. The GitHub Issues system permits and encourages free and open commentary of computational methods. However, GitHub repositories are not immutable. It is important to have the methods, as described in the original publication, preserved and available, so the source code used to obtain the results in the associated full publication mentioned above will be deposited into the Zenodo archive unmodified.

3.2. Meeting the Silver Standard

The silver standard requires, in addition to those aspects listed in the bronze standard, that all dependencies can be installed and set up with a single command, key analysis details are recorded, and all analysis components are deterministic (not random).

A common challenge for reproducibility is having simple installation procedures. To reach the silver standard in this regard we wanted it to be possible to install all dependencies with a single command. For Python-based projects that is often possible with the command “`pip install -r requirements.txt`”¹⁹. However, sometimes other dependencies not covered by pip need to be installed. To simplify this step, we utilized Make (GNU Make v42.1)²⁰. While Make is a powerful tool intended for the control of executable files, we use it only for effectively creating aliases for shell commands. In the case of installation, we provide a Make target called “`setup`”. By doing so, the user can simply type “`make setup`” and shell commands are executed to install all dependencies, including running pip (v21.1.2) for installing Python dependencies¹⁹ and renv (v0.14.0) for installing R dependencies²¹.

In addition to providing a simple pip install procedure we created a conda installation procedure²². While using pip to install dependencies at the user level is sufficient in isolated environments, such as Google Colab (<https://colab.research.google.com/>), it can lead to conflicts on other systems if a virtual environment is not used. Conda (v22.9.0) provides an isolated environment in which the project-specific dependencies are installed. By providing a conda environment description (yaml) file, it is possible to recreate the conda environment in a single command.

Beyond virtual environments, containers such as Docker²³ are often used for documenting and sharing computational environments directly. However, containers can be challenging to use in certain environments. We wanted this project to be reusable for people with a wide range of technical skills, including those who may not have ready access to a robust computational infrastructure. This is especially important when thinking of potential users on a global scale, whose access to resources will be highly variable. This dependence on access to computational resources has been noted as an important part of data democratization²⁴. Here, we designed this project to be run on Google Colab for its low barrier to entry and its provision of graphics processing units (GPUs) for free use. Unfortunately, Colab does not natively support common container services such as Docker. However, by providing several options for dependency installation we hope that future users can find one to suit their needs.

Sufficient documentation of “key analysis details” is subjective. To satisfy this requirement, in addition to an overview README that describes the entire repository, we provide README files in every directory within the repository. These explain what the various files/scripts are and how they relate to each other. Since 2021 GitHub supports the use of Mermaid, a JavaScript-based diagramming and charting tool²⁵, in markdown files, which we leverage to create informative flowcharts illustrating workflow logic.

An often overlooked key to reproducibility in computational methods, particularly ML methods, is seeding pseudo-random processes such that they are deterministic^{9,26}. The random numbers generated by pseudo-random number generators can have significant effects on the trained model and model performance²⁶. So, to make the process reproducible, we added options to use seeding to make the processes deterministic.

3.3. Meeting the Gold Standard

The gold standard implies that the entire analysis can be run with a single command⁹. Such single-command analyses require the use of a workflow manager, of which there are several options. We utilize Snakemake (v7.1.1), which facilitates automation through the definition of “rules” or steps that take inputs and generate outputs. By stating what outputs are desired Snakemake creates a directed acyclic graph of which rules must be executed to create the specified output. For instance, in this project we specify that we would like the final output file to contain the classified articles along with extracted metadata. If the final output is not present, Snakemake executes all necessary steps in the pipeline including data splitting, model training and comparison, classification and Named Entity Recognition (NER), and all downstream processing. With the help of a Make alias, the Snakemake workflow for reproducing all results can be run with the single command “make train_and_predict”.

It is important to be able to reproduce all results from the raw data to final results, including model training. However, model training is resource intensive, and may require the use of specialized hardware such as a GPU for training to be performed in a reasonable amount of time. Requiring that all models be trained to reproduce results may be a practical challenge to reproducibility. To minimize the computational resources necessary for reproduction all fine-tuned models will also be available in the Zenodo repository and HFHub. If the fine-tuned models are downloaded and present when Snakemake is run then Snakemake will not execute model training.

4. Beyond Reproducibility

The goal of reproducibility is to allow anyone to reproduce the results of published research. We have provided, as described above, a system that allows the results of the inventory of global biodata resources to be reproduced. However, this project was also designed to allow the entire analysis to be rerun periodically. Strictly speaking, this goes beyond reproduction since the underlying data is expected to change as more publications are added to the corpus of literature archived in Europe PMC, so the methods developed need to be generalizable. Generalizability benefits from the same considerations as reproducibility but tends to include additional challenges.

We approached generalizability with the same standards as reproducibility and wanted to make updating the inventory possible with a single command. To this end we have designed a second Snakemake workflow for periodically updating the inventory. For this process the trained models can be automatically obtained from Zenodo using the setup command. The previously best performing models for each task are used, which eliminates the need for retraining and evaluation.

5. Code Standards

We've taken the philosophy that the results of a computational research project are no more trustworthy than the code used to produce them. Trustworthiness of code is dependent on code quality, including considerations such as readability and robustness. In this section we will describe the measures taken to ensure code quality such as code formatting, static code checking, and automated testing.

5.1. Code Formatting

To accomplish Open Science, accessibility of code should not be limited to code being publicly available. True accessibility requires that code also be readable and well documented. A good first step is to utilize a code formatter, which all modern programming languages have. We used yapf v0.31.0 to format all of the Python code in this project²⁷. Similarly, Snakemake files were formatted with snakefmt v0.6.0, and R files were formatted with styler v1.7.0^{28,29}. These steps are meant to ensure that all components of the project are readably formatted and documented to maximize their ease of use for others.

5.2. Static Code Checking

Another measure taken to increase code robustness is static code checking. Again, the code checking tools available will depend on the language. We utilize the linters pylint v2.8.2 and flake8 v3.9.2 to check all Python code to ensure that community code standards are upheld and to detect code smell (patterns indicative of potential problems)^{30,31}. Many of the items that these linters consider can greatly improve code quality and readability. Some examples of considerations of the linters are: line lengths must be limited to predefined thresholds, within any context (e.g., a function) there should not be too many variables, and all functions should have docstrings. These, and many other requirements, encourage developers to write cleaner, more readable code.

Additionally, while type annotations are not required in the Python community, we implemented them as they provide a number of benefits. Type annotations provide built-in documentation by defining the data types of all inputs and outputs of functions. A lesser discussed benefit of type annotations is that they provide an enhanced integrated development

environment (IDE) experience since the IDE has more knowledge of the variables and can give better help messages, syntax highlighting, and autocompletion. The final benefit of type annotations is prevention of unforeseen bugs when they are used in conjunction with a static type checker. We used mypy v0.812 to check type compatibility within all our Python code³². This can significantly reduce the chances of encountering bugs that occur not at compile time (since Python is interpreted and dynamically typed), but instead at runtime, which can be more difficult to resolve and may not show up until running the code at a later time.

While static code checking has many benefits, programmers need not strictly adhere to all suggestions made by the code checkers. Luckily, most tools are configurable. Importantly, the user can disable certain warnings. For Pylint, this is done with a resource configuration (rc) file. When an rc file is used to modify code checkers, it is important to include them in the repository, so that when someone else runs the code checkers on published code they see the same results.

5.3. Testing

A crucial software engineering practice that is often absent from research code is testing. Testing in all of its forms: unit, integration, and end-to-end, defines the specifications of a piece of software and ensures that the software meets those specifications when the tests pass. This has numerous benefits that cannot be understated.

One of the primary benefits is that tests serve as a contract, which is a form of documentation. A unit test of a function explicitly states what kinds of input are expected and what kinds of outputs will be produced. For documentation, the only thing better than telling what a function does (through comments and docstrings) is showing through tests (asserting that when certain inputs are provided, the expected output is returned). While the descriptions provided in docstrings and comments are what the developer intends the software to do, a passing test demonstrates that it indeed does what was intended. Conversely, anything not covered in the test cases is where the contract ends. Tests ensure that the code can do what it says.

From an Open Science perspective testing is particularly valuable. Not only does testing provide more detailed documentation than could ever be provided in an article's methods section, but it facilitates community feedback and contributions. Making changes to software always poses the risk of disrupting previous functionality. When considering applying community feedback or contributions this is problematic. However, with strong test coverage, developers can have more confidence that updates do not introduce breaking changes, as long as all previously passed tests still pass. Indeed, they provide a clear avenue for addressing bugs which may be caught by the community. Developers can add another test case that exposes the bug, then modify the code such that the new test and all previous ones pass. This is effectively amending the contract provided by the tests so that it is more comprehensive. Without tests in

place developers would have to check that the code still behaves as described manually. Such checking is so error prone that many researchers may be hesitant to implement changes suggested by others.

Of course, adding strong test coverage does require more work than, for instance, implementing static code checks or formatting. Without tests, though, code must be manually assessed to ensure that a given piece of software is able to perform its intended task, and there is a barrier to implementing community feedback. Further, a lack of tests is a form of technical debt, and the price is paid when trying to refactor or fix bugs.

Pytest v6.2.4 was used as a testing framework for all Python code in this project³³. Pytest plugins for flake8, pylint, and mypy are used to include static code checks of each file as part of the test suite (pytest-flake8 v1.0.7, pytest-pylint v0.18.0, pytest-mypy v0.8.1)^{34–36}. This makes it such that the test suite cannot pass without all static checks passing. Additionally, most functions have associated tests, and most scripts also have end-to-end tests that ensure that they properly reject bad inputs and produce correct output when given good input. While we aim to have good test coverage, some functions and scripts are not comprehensively tested. This is generally the case for functions/scripts that take a very long time to run, such as the actual process of model training. Additionally, the Snakemake workflows developed are not formally tested using an automated testing framework, although it would be best to do so and we may implement this at a later time.

5.4. Configurability

Our aim was that the users of code, whether for reproducibility, generalization, or separate implementation, would not need to edit source code to change its behavior within the intended use cases. Parameters that may change could be supplied as inputs/arguments instead. Often, this means that paths to input files should not be hard-coded but rather passed in when calling a script. In terms of ML projects, this also often applies to hyperparameters.

One solution to this is to use parameterization extensively and, in order to make the analyses reproducible, to store the parameters used in configuration (config) files. By doing so, others can see what parameters were used to generate the results. This process additionally gives future users a clear indication of what parameters are likely okay to change, all without them having to edit any source code.

We store a large number of parameters in config files such as input/output directories, training parameters, and locations of fine-tuned models. To train a new model and compare their performance to existing models, a new row need simply be added to a tab-separated config file. The README file in the config/ directory describes the acceptable ranges of values allowed in the config files, such as a description of what kind of models are compatible with the existing workflow.

Snakemake also makes extensive use of config files, and the config files described here are formatted such that Snakemake can utilize them when executing the workflow. So, to change the behavior of the workflow (again, within the expected range of uses), only config files need to be edited.

6. Data Standards

6.1. Source Selection

Both code and data are integral components of this project and both require consideration for reproducible outcomes. To create an open inventory as a product we aimed to reuse and create data that aligned with the FAIR (Findable, Accessible, Interoperable, and Reusable) Guiding Principles¹⁶. The primary data source needed was bibliographic metadata. There are several commercial sources of bibliographic metadata such as Dimensions (Digital Science), Scopus (Elsevier), and Web of Science (Clarivate Analytics). However, these resources require a subscription which would limit others' ability to reproduce and reuse our workflow and neither are they openly licensed. Therefore, we opted to use the open metadata available from Europe PMC as the data source for creating the inventory. Although not as exhaustive as the commercial options mentioned, Europe PMC covers a large swath of the life sciences; as of October 2022, high quality, interoperable metadata, including titles and abstracts, was available for over 40 million articles. Additionally, Europe PMC offers robust and well-documented APIs that facilitate access and are especially useful for a reproducible pipeline. Although we know that some biodata resources will be missed due to articles being published outside of the ~4000 journals available in Europe PMC, we felt that this tradeoff was justified in order to optimize openness and reproducibility.

6.2. Addressing Data Findability and Accessibility

Depending on context, anyone interested in reusing the data from this project might wish to start at different points. We therefore offer multiple options. The exact query string we used can be rerun to obtain results from Europe PMC. Additionally, since bibliographic databases may change slightly over time (e.g. records added, removed, or corrected), query results themselves (PMID, title, abstract) may be of use to reproduce our results using the exact same data. There is also the labeled training data that was used to train the various models, a preliminary inventory that is subjected to selective review by a curator, and, finally, the primary data product for this project is the final inventory itself. The query string, query results, training data, preliminary inventory, and the final inventory are all available within the project's GitHub repository and will be archived for long-term preservation and persistent reference in an associated Zenodo deposit when the preprint of the forthcoming full article is posted. Zenodo

provides a DOI and relies on the DataCite metadata schema, which allows the dataset to be found within Zenodo's search interface, DataCite's central metadata store, and via internet search engines such as Google.

6.3. Addressing Data Interoperability

For the final inventory, we retained unique article identifiers (PMIDs) to allow easy extraction of additional metadata or for access to the full text, when available, from either Europe PMC or PubMed Central. Additionally, we logged URL status codes per specification RFC 9110³⁷, extracted countries from author affiliations following ISO 3166³⁸, and retained geo coordinates for IP address look-ups, when available. While it would have been ideal to include a persistent identifier for the biodata resources located (e.g. ROR ID or DOI), most resources do not have an identifier, which perfectly illustrates the challenge of trying to locate these resources in the first place.

6.4. Addressing Data Reusability

In addition to the efforts towards interoperability described above, we also maintained a structured format throughout and used the CSV format for preservability and to ensure ease of reuse. These files are accompanied by a plaintext README file that includes a description of each variable as well as data collection details and licensing. By using open data from Europe PMC we are able to release the data with CC0 licensing, thus allowing the broadest reuse possible. Together, this documentation, the repository's Github history, and Zenodo's commitment to long-term archiving all provide provenance.

Finally, to further extend the potential for reuse, we plan to provide identified biodata resources to Europe PMC as community annotations. This will allow easy bulk access to the identified resources as well as their associated articles. The annotations can be used for several purposes, for example, mining articles with full text available or analysis of the intersection between these annotations and the many other annotation types available within Europe PMC.

7. External Review/Validation

In the Open Science Implementation Plan that we drafted (see Section 2 above), we also included a desire to have a party external to the team review the products of the study. Working within a team inherently provides a mechanism for internal feedback, but review by another person outside of the project helps reveal implicit knowledge developed during the project that would otherwise remain hidden to potential reusers. For example, team members may, without realizing it, adopt terms or abbreviations that are not well-known outside of the project.

This section of the Open Science Implementation Plan was not particularly well-developed beyond acknowledging that such a review would be ideal, as noted by others^{9,39}

, and that this role is included in the CRediT taxonomy⁴⁰. As we moved closer to having products finalized, we had a better sense of what sorts of reviews will be most valuable. We recruited an individual who reviewed the code and documentation in detail and ran nearly all the code in the forms available to assess if everything is well-described and consistent. We budgeted for 40 hours for this work, which was easily consumed given the amount of materials which required detailed review. Others may wish to plan to devote even more of their resources to this activity, which we found extremely helpful in finding errors and pointing out gaps in our documentation. We formally acknowledge this effort in this comment as well as the forthcoming full article.

8. Discussion

Here we have described the efforts that were taken to develop a methodology for obtaining and updating a biodata resource inventory with Heil's gold standard reproducibility, a robust codebase, and complying with FAIR data standards. Installation of dependencies and reproduction of the entire analysis can be performed with a single command each, and analysis steps are fully documented. All code passes static code checks for formatting, linting, and type compatibility. Much of the code is formally tested with unit and integration tests. The core data products, such as the labeled training data and preliminary inventory, are currently present in GitHub and will be archived in Zenodo, with accompanying documentation. While we put a considerable amount of effort into something we believe is reproducible, time will tell and time will tell for how long. We anticipate updating the inventory in spring 2024, and that will be the first true test. By then most of the research team will have moved on and the remaining members will not be those who directly wrote any of the code or are not programmers themselves. Community interest has been strong so far but projected reuse is unknown at this time.

The methodologies used in this work are not novel on their own. The automation employed to make reproduction simple relies on the widely used Snakemake workflow manager. It is also common practice in software engineering disciplines to leverage static code checking and testing as we have done. Regarding data standards, we looked to the FAIR principles. The purpose of this work is to provide an example of how a research project that utilizes computational methods, particularly ML, can be implemented to maintain robustness and strive for a high level of reproducibility. However, we recognize that there are numerous ways to accomplish this and do not mean to claim our implementation is failproof.

Certain improvements could be made, such as using a more robust package manager like poetry, and using git hooks to automatically run tests upon committing to git. Importantly, test coverage is lacking in some areas, especially for portions that involve heavy computation such as model training. Still, the current test coverage is enough to increase confidence in the code's

behavior. As Peng noted, "Given the barriers to reproducible research, it is tempting to wait for a comprehensive solution to arrive."⁴¹, thus we thought our experiences may be helpful to share.

Beyond the technical details, it is likely quite apparent that this work required a substantial amount of time to think through and implement. We were able to do this only because of our team's collective belief that these efforts were worth the resources invested. However, not only did it require allocation of those resources, it required trust among team members and a willingness to engage in the process. Additionally, many tradeoffs were made. One example is the tradeoff of using only open data versus a more extensive commercial data source, which would likely have yielded a larger, but in our estimation a less useful, inventory. There are also ambitions that we had at the start of the study that are now future directions because we chose to devote time developing a robust workflow instead. This required principled project management and caused, even as we write this, some amount of wistfulness. In the end, we could not do it "all", and we fully appreciate that others must decide for themselves where to place their efforts.

We hope that this work can provide an example—hopefully a helpful one—to others interested in attempting greater reproducibility in their research. We would be especially gratified to learn of others who suggest better solutions than those presented here.

9. References

1. Baggerly, K. A. & Coombes, K. R. Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. *The Annals of Applied Statistics* **3**, 1309–1334 (2009).
2. Peng, R. D. & Hicks, S. C. Reproducible Research: A Retrospective. *Annual Review of Public Health* **42**, 79–93 (2021).
3. Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature* **533**, 452–454 (2016).
4. Fanelli, D. Is science really facing a reproducibility crisis, and do we need it to? *Proceedings of the National Academy of Sciences* **115**, 2628–2631 (2018).
5. Leek, J. T. & Jager, L. R. Is Most Published Research Really False? *Annual Review of Statistics and Its Application* **4**, 109–122 (2017).
6. Haring, R. & Bell, R. J. Lack of research reproducibility, the rise of open science and the

- need for continuing education in research methods. *Climacteric* **21**, 413–414 (2018).
7. Walters, W. P. Code Sharing in the Open Science Era. *J. Chem. Inf. Model.* **60**, 4417–4420 (2020).
 8. Perkel, J. M. Challenge to scientists: does your ten-year-old code still run? *Nature* **584**, 656–658 (2020).
 9. Heil, B. J. *et al.* Reproducibility standards for machine learning in the life sciences. *Nat Methods* **18**, 1132–1135 (2021).
 10. Imker, H. J., Schackart, K. E., Istrate, A.-M. & Cook, C. E. A Machine-Learning Enabled Open Biodata Resource Inventory from the Scientific Literature. *forthcoming* (forthcoming).
 11. The Europe PMC Consortium. Europe PMC: a full-text literature database for the life sciences and platform for innovation. *Nucleic Acids Research* **43**, D1042–D1048 (2015).
 12. Ferguson, C. *et al.* Europe PMC in 2020. *Nucleic Acids Research* **49**, D1507–D1514 (2021).
 13. Berman, H. M. *et al.* The Protein Data Bank. *Nucleic Acids Research* **28**, 235–242 (2000).
 14. Wolf, T. *et al.* Transformers: State-of-the-Art Natural Language Processing. in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* 38–45 (Association for Computational Linguistics, 2020).
doi:10.18653/v1/2020.emnlp-demos.6.
 15. Köster, J. & Rahmann, S. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* **28**, 2520–2522 (2012).
 16. Wilkinson, M. D. *et al.* The FAIR Guiding Principles for scientific data management and stewardship. *Sci Data* **3**, 160018 (2016).
 17. de Saint-Exupéry, A. Le petit prince [The little prince]. *Verenigde State van Amerika: Reynal & Hitchcock (US), Gallimard (FR)* (1943).

18. Imker, H. J. & Schackart, K. E. Open Science Implementation Plan for the Biodata Resource Inventory. (2022) doi:10.5281/zenodo.7392518.
19. pypi. Python Package Index - PyPI.
20. GNU Make. (1988).
21. Ushey, K. *renv: Project Environments*. (2022).
22. Conda. (2017).
23. Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux j* **239**, 2 (2014).
24. Hook, D. W. & Porter, S. J. Scaling Scientometrics: Dimensions on Google BigQuery as an Infrastructure for Large-Scale Analysis. *Frontiers in Research Metrics and Analytics* **6**, (2021).
25. Sveidqvist, K. Mermaid: Generation of diagrams like flowcharts or sequence diagrams from text in a similar manner as markdown. (2014).
26. Ahmed, H. & Lofstead, J. Managing Randomness to Enable Reproducible Machine Learning. in *Proceedings of the 5th International Workshop on Practical Reproducible Evaluation of Computer Systems* 15–20 (Association for Computing Machinery, 2022). doi:10.1145/3526062.3536353.
27. yapf: A formatter for Python files. (2004).
28. Hall, M. & Letcher, B. Snakefmt: The uncompromising Snakemake code formatter. (2020).
29. Müller, K., Walthert, L. & Patil, I. styler: Non-Invasive Pretty Printing of R Code. (2021).
30. Thénault, S. Pylint: It's not just a linter that annoys you! (2001).
31. Ziade, T. & Cordasco, I. Flake8: Your Tool For Style Guide Enforcement. (2011).
32. Lehtosalo, J. mypy: Optional static typing for Python. (2012).

33. Krekel, H. pytest: The pytest framework makes it easy to write small tests, yet scales to support complex functional testing. (2004).
34. Gee, C. pytest-pylint: pytest plugin for running pylint against your codebase. (2015).
35. Bader, D. pytest-mypy: Mypy static type checker plugin for Pyest. (2016).
36. Lockhert, T. pytest-flake8: pytest plugin to run flake8. (2015).
37. Fielding, R., Nottingham, M. & Reschke, J. *RFC 9910 HTTP Semantics*.
<https://www.doi.org/10.17487/RFC9110> (2022).
38. Country Codes - ISO 3166.
39. Coburn, E. & Johnston, L. Testing Our Assumptions: Preliminary Results from the Data Curation Network. *Journal of eScience Librarianship* **9**, (2020).
40. Allen, L., O’Connell, A. & Kiermer, V. How can we ensure visibility and diversity in research contributions? How the Contributor Role Taxonomy (CRediT) is helping the shift from authorship to contributorship. *Learned Publishing* **32**, 71–74 (2019).
41. Peng, R. D. Reproducible Research in Computational Science. *Science* **334**, 1226–1227 (2011).

Glossary of Software

Name	Description	Reference
conda	Package and environment management system	22
flake8	Python linter (static code checking)	31
Make	Build automation tool, used here for creating shell command aliases	20
Mermaid	Diagram generator for Markdown	25
mypy	Static type checker for Python	32

pip	Package manager for Python	19
pylint	Python linter (static code checking)	30
pytest	Python testing framework	33
pytest-flake8	Pytest plugin to run flake8	36
pytest-mypy	Pytest plugin to run mypy	35
pytest-pylint	Pytest plugin to run pylint	34
renv	Dependency manager for R	21
snakefmt	Code formatter for Snakemake	28
Snakemake	General-purpose workflow manager	15
styler	Code formatter for R	29
yapf	Code formatter for Python	27

Acknowledgements

The authors would like to thank Ana-Maria Istrate with the Chan Zuckerberg Initiative for her contributions to developing the machine learning methods used in the project as well as CZI colleagues Dario Taraborelli, Donghui Li, and Gully Burns for their support and feedback on early versions of the study. We also thank Ken Youens-Clark formerly at The University of Arizona, Alise Ponsero at The University of Helsinki, and Bonnie Hurwitz at The University of Arizona for their mentorship of Kenneth Schackart. Additionally, we would like to acknowledge Jodie Forbes for detailed review of the associated code and documentation.

Conflict of Interest Declaration

The authors declare no conflict of interest.

Funding

This work was supported by the Chan Zuckerberg Initiative, which is a member of the Global Biodata Coalition. This work was also funded by the Global Biodata Coalition (globalbiodata.org), a coalition of research funding organizations working towards sustainability of biodata resources worldwide.

Code and Data Availability

Code and data to date are the associated Github repository, which is available at https://github.com/globalbiodata/inventory_2022/tree/inventory_2022_dev. Additionally, all code and data will be archived in Zenodo when the forthcoming full article describing the study is published.