Computational Reproducibility: What to expect from a Replication Package Repository?

Martin Blum[1,*], Ralf Schenkel[1]

[1]University of Trier, Universitätsring 15, 54296 Trier, Germany

Abstract

The release of replication packages with source code has become a valuable step in ensuring the reproducibility of the results promulgated in scholarly articles. Many of the recent works in the field of reproducibility focus on the deviation of reproduced outcomes from their corresponding published results and often simply ignore replication packages which could not be executed and evaluated at all. This study evaluates replication packages hosted on GitHub for more than 200 000 scientific publications and attempts to algorithmically decide if the files provided in the repositories potentially allow reconstituting the original execution environments and, therefore, are more likely to have their published findings reproduced. We find that less than half of the analyzed repositories contain dependency information files to successfully curate runtime environments. Additionally, we discover that these files frequently are of poor quality and only install the newest versions of dependencies instead of those used for the original research; this happens especially often with packages for numerical computation when compared to regular utility packages.

Keywords

Reproducibility, Replication Package, PapersWithCode, GitHub, Text Processing

1. Introduction

The feasibility of reusing existing scientific findings to facilitate more advanced research depends on several factors. An important aspect is the need to trust the results of the original work [1]. This trust is facilitated if an independent researcher is able to apply the same methods on the same data and thereby ”*reproduce*” the original authors' results and conclusions [2]. In contrast, if the attempt to reproduce leads to deviated data and divergent inference, the examined scientific research would be considered less trustful [3].

Attaching replication packages with source code to scholarly works simplifies the process of verifying the *computational reproducibility* of the respective research. A different researcher using the same code and data as in the original work should obtain the same results [4].

Validating reproducibility could be assumed to be a simple task: If both data and source code exist in digital form, it should be easy to just run the code and receive the same results as before. However, 50% of the researchers have reported failing to reproduce their own results and even 70% were unable to perform this task on someone else's work, a situation labeled ”reproducibility crisis” in science [5]. In the context of Machine Learning and Natural Language Processing, these tendencies are reported only slightly better, with 56% of researchers not being able to execute someone else's code at all or obtaining significantly different results compared to the original work [6].

This led to the creation of various ”reproducibility checklists” that authors should fill out when submitting scientific work [7, 8]. A study confirms that scholarly works that satisfy more of the items on such checklists are more likely to be accepted for publication at NLP conferences [9], as they provide less reason for doubt in the validity of their results. A different study concludes that *source code* is *not enough* for reproducibility [10].

36th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 29 - October 01, 2025, Regensburg, Germany

*Corresponding author.

blumma@uni-trier.de (M. Blum); schenkel@uni-trier.de (R. Schenkel)
0000-0003-0005-6162 (M. Blum); 0000-0001-5379-5191 (R. Schenkel)

Although the ability to obtain the exact same results can depend on various factors, such as access to code and data, algorithm documentation, dataset versions, parameter settings, initialization values, random seeds, hardware details, and runtime environment [11], we focus only on the latter in this work: **Do replication packages supplied with scientific publications contain enough information to reproduce their respective original runtime environments?**

2. Background

Modern programming languages (e. g., *Python*) already include standard runtime libraries that offer built-in support for many common tasks. For additional requirements, algorithms have to be either implemented from scratch by programmers or imported from third-party libraries which provide ready-to-use versions of the required functionality.

Since these libraries are maintained independently, code updates can possibly break existing source code relying on them completely (e. g., interface changes) or result in divergent runtime behavior (e. g., arithmetic rounding deviations if internal calculations are performed in a different order).

For example, the ”*numpy*” library – which we later observe as the most referenced third-party Python library in our study – maintains a list of supported Python releases in combination with ”*numpy*” versions [12]. However, even this precaution was not enough when the transition to *pairwise summation* changed the internal computation of the dot product[1] and for $\vec{a} = (a_1, a_2)$ and $\vec{b} = (b_1, b_2)$ the equation $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2$ was no longer true[2].

Therefore, *reproducibility* often depends on the availability of exact versions of third-party libraries. A replication package for source code using ”*numpy*” might:

- Just use ”`import numpy`” in its source code
- Ask to install ”*numpy*”
- Ask to install ”*numpy*” version ”x.y”
- Ask to install ”*numpy*” version ”x.y” in combination with Python release ”a.b.c”

Only the last option would be a good candidate to reproduce outputs comparable to the original results, while the other three possibilities might compute a deviating output or not run at all. As a consequence, this could cast doubt on the validity of the research output.

For this work, we focus only on source code and replication packages using *Python* as programming language.

3. Methodology

This section describes the pipeline we use to select and retrieve the data on which our study is based and the methods we apply to it.

3.1. Data Selection

We use the data provided by *Papers With Code*[3] as the initial seed for our research. The website maintains a large corpus of scientific publications concerning Machine Learning in various fields of science. It also provides information about *methods introduced*, *datasets created*, *evaluation results achieved* and *source code released* for many of the individual articles. This data is available in daily updated JSON dumps [13]. We use the version downloaded on November 12th, 2024 for our research.

[1]https://github.com/numpy/numpy/pull/3685, accessed 2025-07-18
[2]https://stackoverflow.com/questions/32952941/numpy-floating-point-rounding-errors, accessed 2025-07-18
[3]https://paperswithcode.com/, accessed 2025-07-18; offline at the time of the camera-ready version (redirects to https://huggingface.co/)

3.1.1. Papers With Code

The file ”*links-between-papers-and-code.json.gz*” contains mappings of scientific publications to one or more source code repositories. In our downloaded version, approximately 265 000 mappings pointing to 210 000 unique repositories are included. These references exist if

- the publication explicitly links to the source code in its full text,
- the repository states in its *README* file that it contains the replication package for the publication, or
- the publication's authors create an account on the website and manually add the mapping.

The publication's metadata only includes its title and two URLs to its landing page and PDF file download location, but no additional information such as year of publication, field of science, or a digital object identifier (DOI).

Since this would prevent researching the changes in replication packages over time and also complicate deciding if one specific publication known only by its DOI, title and authors matches one of those analyzed in this work, we attempt to determine each publication's DOI and to match it to an OpenAlex[4] metadata profile.

First, we look at the URLs of a publication's landing page and PDF location and generate a list of possible alternative URLs which still lead to the same resource, e. g., if the URL format changed over time. We apply the following heuristics:

- Scanning for embedded DOIs or alternative resolvers (e. g., ”*https://dl.acm.org/doi/*”) and then using ”*https://doi.org/*” and the DOI to create the URL
- Removing file extensions (”*.pdf*”) or path suffixes (e. g., ”*/download*”, ”*/full*”, or ”*/xml*”)
- Replacing known domain names by alternative ones (e. g., ”*https://aclanthology.org/*” instead of ”*https://www.aclweb.org/anthology/*”)
- Removing embedded version numbers (e. g., ”*/pdf/1601.02063**v2**.pdf*” → ”*/pdf/1601.02063.pdf*”)
- Removing or adding leading zeros for files hosted on arXiv[5] (e. g., ”*/1601.**0**2063*” and ”*/1601.2063*”)

Then, we collect all these permutations in a list of candidate URLs and try to match them with a known DOI. For this, we build a reverse lookup table with the public data available from the *Crossref* [14] and *Datacite* [15] DOI registration agencies. If a DOI resolving to one of the URLs is found, we add it to the candidate list.

Finally, we search the OpenAlex [16] metadata dump for any URLs and DOIs contained in our candidate list.

3.1.2. Crossref and Datacite

Crossref and *Datacite* are large DOI registration agencies. They assign DOIs to publishers that can be updated by them to resolve to the most recent URL for the given publication. A DOI may link to various alternative URLs that provide other data formats based on an additional application requested (e. g., ”*text/xml*” for text-mining).

Although the Web-APIs of Crossref and Datacite only allow resolving DOIs to URLs and enforce query limits, they also provide a downloadable dump of their data [14, 15], which we use to create an inverse index to match our candidate URLs to known DOIs. Collectively, both dumps provide metadata for approximately 240 million DOIs.

[4]https://openalex.org/
[5]https://arxiv.org/

3.1.3. OpenAlex

OpenAlex is an open and free bibliographic catalog of scientific publications and their related metadata. Besides its own ”Alex-IDs”, it also includes several other unique identifiers, such as DOIs. In addition, it contains extensive metadata, such as the date of publication, fields of science with rated scores, open access status, all known landing pages, and download locations for each indexed entry.

We use the OpenAlex data dump[6] published on May 29th, 2025 for our research, which contains metadata for 267 million publications.

We then try to match all available URLs to the publications from our *Papers With Code* candidate list and filter out all entries found where OpenAlex lists the *concept* ”Computer Science” with a score ≤ 0.5 or not at all. This way, we attempt to focus only on the computer science domain and ignore source code repositories for Machine Learning publications in fields such as ”Psychology”, ”Chemistry”, or ”Economics”, which might have divergent expectations on reproducibility and replication packages.

3.1.4. unarXive

The *unarXive* [17] dataset contains metadata and full text for all articles uploaded to the arXiv website up to December 2022. Since many URL permutations of arXiv URLs are already included in the candidate list, matching is uncomplicated and gives us additional information about the field of science by returning the *categories* the authors have selected for their uploaded publications.

3.1.5. Metadata Multiplicities

Multiple entries in the OpenAlex and unarXive datasets might be matched to the same publication listed in *Papers With Code*. This happens if several versions of a scientific paper exist (e. g., a few pre-prints and a journal publication). Since we mainly use the metadata to decide if we include a publication in our research, this problem can be neglected.

When processing the *year of publication* in our work, we use the most recent of the multiple available dates since we assume the contents of the source code repository to be up-to-date with the latest version of a publication.

3.1.6. Final Research Data Set Selection

Based on our previous steps, we now have a curated list of computer science publications from the *Papers With Code* file *”links-between-papers-and-code.json.gz”*, including the matching metadata available from the OpenAlex corpus.

Since almost all (99%) linked source code repositories are located on GitHub[7], we only focus on retrieving the repository metadata from there and ignore other hosting solutions.

3.2. Data Retrieval

Since the valid format of a GitHub repository is known to be ”*https://github.com/<username>/<repository>*”, we lowercase the URLs and remove any additional data (e. g., pagination arguments, or directly linking to a branch or file).

For all of the approximately 210 000 distinct GitHub repositories linked in our *Papers With Code* data, we downloaded the repository metadata between December 24th and 29th, 2024 using the GitHub Web-API[8].

We also retrieved the content metadata (e. g., directory structure, filenames, and sizes) within two months after that. Since these downloads were based on the *tree-hash* stored in the repository metadata, all structural data downloaded represents the state it had between December 24th and 29th, regardless

[6]https://docs.openalex.org/download-all-data/download-to-your-machine
[7]https://github.com/
[8]https://docs.github.com/en/rest?apiVersion=2022-11-28

of whether the repository was updated afterward. The same applies to individual files downloaded later during our research.

After removing inaccessible repositories and applying the data selection filters described above, approximately 151 000 GitHub repositories are left for our research.

3.3. Reproducibility Heuristics

The assessment of how likely it is to reproduce a different researcher's outcomes depends not only on the question if source code is provided but also on whether the environment in which the code was originally executed can be replicated as closely as possible. For Python source code, we assume that this depends mainly on the Python version used and the dependencies installed in its environment.

Other external factors, such as the availability of a specific C++ compiler in the operating system or the presence of a given GPU architecture, are ignored in our work.

3.3.1. Environment File Selection

We download individual files from the chosen repositories that are typically connected to the creation of a runtime environment with specific dependency versions. This includes file types such as:

- ”*requirements.txt*”: This file is conventionally associated with the Python Package Index[9] and the ”PIP” module used to install dependencies in an existing Python environment. The text file lists the package names and may include additional restrictions on the versions to be installed (e. g., $=, <, >, \geq, \leq, \neq$). If no version is specified, the most recent compatible package would be installed. It is not possible to change the Python version itself using PIP.
- ”*conda.txt*”, ”*environment.yaml*”: These files are used by Anaconda[10], an alternative Python distribution that uses its own package index, which is additionally able to install different Python releases. Package version restrictions similar to PIP can be applied.
- ”*setup.py*”, ”*pyproject.toml*”, ”*meta.yaml*”: These files are used when building a Python package and specify which version ranges they expect for other packages. While not directly related to setting up a Python environment, they influence which other package versions can be installed at the same time.
- ”*Dockerfile*”, ”*docker-compose.yml*”: These files are used by Docker[11] containers and specify how an operating system environment should be set up. Their existence could mean that the publication's authors are aware of possible problems with setting up the execution environment for their source code and therefore provide a preconfigured container, presumably the same one used for the original research.

Although researchers often follow these file naming conventions, they might also violate them (e. g., ”*conda_env.yml*”, ”*additional_requirements.txt*, ”*python-package-conda.yml*”, ”*test.dockerfile.18.04*”). When scanning the GitHub content metadata for candidate files, we also include similar file names to be downloaded and determine the specific file type during the parsing process.

3.3.2. File Encoding

When retrieving files using the GitHub API, it returns the file contents in *Base64* format. We detect the proper character encoding (e. g., UTF-8 or UTF-16) and remove any existing *Byte Order Marks*. This step is necessary because the parsing libraries used in our research do not support such markers inside character sequences.

[9]https://pypi.org/
[10]https://anaconda.org/anaconda/python
[11]https://www.docker.com/

3.3.3. Parsing

After downloading all candidate files, we parse them based on their contents, since their filenames might be unusual or misleading. For file types that support specifying Python dependency versions, we analyze the extent to which they make use of this possibility.

PIP File Format PIP supports loading a list of dependencies from a plain text file, one package per line. Valid names may contain only ASCII characters, digits, underscores, and hyphens. They are followed by zero or more pairs of a comparator and a version number, e. g., ”*transformers>=4.31.0,<4.35.0*”. White space, blank lines, and everything after a ”#” character is ignored.

Entries may specify to load packages from local files, URLs, or GitHub repositories. PIP also supports commands to add additional package indices or cryptographic hashes.

Anaconda File Format Anaconda supports two types of file formats. One works similarly to PIP, but uses a different line format, e. g., ”*boltons=23.0.0=py310h06a4308_0*”. The other one uses the YAML[12] syntax to specify dependencies. Depending on the position and node type in the YAML structure, packages may be imported using the Anaconda or PIP syntax.

Packaging Tools and Docker Files We detect the TOML- and YAML-based file types by comparing the nodes of their extracted tree structure with a list of nodes we expect for that file type (e. g., we separate Anaconda *environment* files (node: ”dependencies”) from Anaconda *packaging* files (node: ”requirements” or ”build”) by testing for the (non)existence of these nodes). ”*setup.py*” files are detected by checking if they import the ”distutils” or ”setuptools” packages.

4. Results

This section shows the results of our work and the statistics generated during it.

4.1. Matching and Filtering

We present the results of the intermediate steps used in our data selection process. Figure 1 shows the number of repositories in the individual stages.

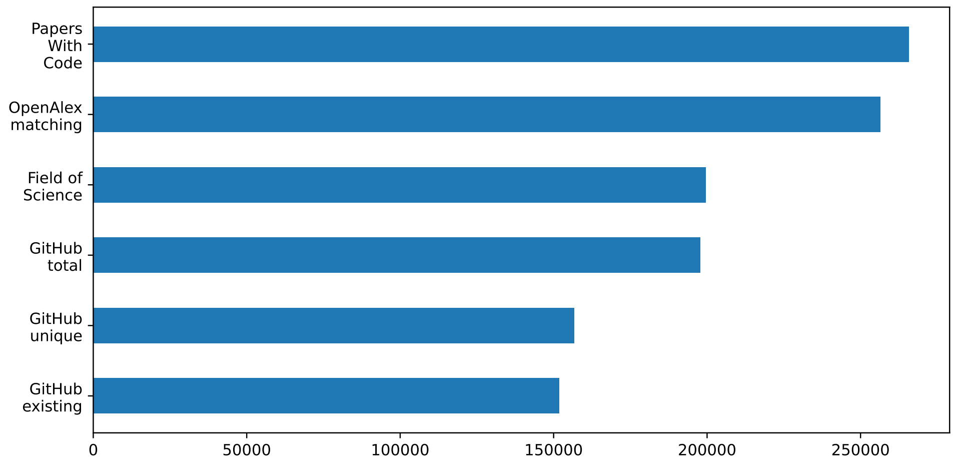

Figure 1: Number of repositories in the individual steps of our data selection process.

[12]https://yaml.org/

4.1.1. OpenAlex Matching

Our *Papers With Code* input file ”*links-between-papers-and-code.json.gz*” contains 265 735 mappings between articles and source code repositories. For 256 560 (96.5%), we are able to match them with an OpenAlex metadata record by using URL permutations and the inverse index generated from the Crossref and Datacite dumps.

129 025 (50.3%) of the articles are matched with more than one and 102 (0.04%) with more than four publication-IDs from the OpenAlex dataset. An inspection of all ”two-match” results (103 582) shows that 103 418 (99.8%) of them have one of the publication-IDs referring to an arXiv article and the other to a non-arXiv URL. This validates our assumption that many publications have a pre-print version in addition to a conference or journal publication.

We also observe 21 outliers with more than 10 000 publication-IDs each. A manual inspection shows that the publishers used a specific URL format that was reduced to the same prefix for all of their publications. This was caused by our code trying to extract DOIs from regular URLs. Since only 21 publications are affected in total, we keep the data in our research set.

4.1.2. Field of Science Filtering

After filtering the 256 600 remaining publications using the OpenAlex ”*concept*” and unarXive ”*category*” information, we reduce our list to 199 547 (77.7%) Computer Science publications.

4.1.3. GitHub Repository Filtering and Retrieval

When only GitHub repository links are considered for further research, 197 878 publications remain. They link to 156 743 *unique* GitHub repositories. We were unable to download metadata on 4984 (3.2%) of them since the GitHub API responded with ”*HTTP 404 - Not Found*” errors. The remaining 151 759 (96.8%) links could be retrieved without problems.

4.2. General File Type Statistics

The largest repository contains 3 787 028 files, while the average counts (647) and the median (39) are much lower. For the total size, the maximum is 85 GB with average 63 MB and median 3 MB.

Table 1 shows the Top-10 distribution of file extensions in the content metadata downloaded for the repository list. As expected, Python source code (”*.py*” and ”*.ipynb*”) accounts for the largest number of files in repositories related to Machine Learning. C / C++ would get the second place (”.cpp/.h/.c/.hpp” ranked on 16/17/34/40), followed by Matlab (”.m/.mat” ranked on 26/38).

Python source code is found in 132 130 repositories (87.0%) with 126 704 (83.4%) containing at least one ”*.py*” file and 36 903 (24.3%) at least one ”*.ipynb*” file. Jupyter Notebooks may be used for programming languages other than Python. Since they account for a total size of 158 GB, we refrained from downloading and analyzing them individually to determine the language used. C / C++ source code is only found in 20 507 (15.5%) repositories.

4.3. Parsing

In the 151 759 repositories we consider for our research, we observe 70 741 (46.6%) that include at least one of the file types we assume to contain instructions for reproducing the original runtime environment. The total number of such candidate files is 189 110, including 2189 with zero-byte length.

Table 1
Ranking of file extensions by size and count.

Rank	Extension	Size	Extension	Count
1	.csv	1054 GB	.md	149201
2	.json	721 GB	.py	126704
3	.txt	632 GB	—	92524
4	.png	568 GB	.txt	84616
5	.npy	347 GB	.gitignore	81166
6	—	308 GB	.png	66383
7	.pt	293 GB	.sh	50519
8	.jpg	268 GB	.ipynb	36903
9	.pkl	245 GB	.json	31276
10	.pth	235 GB	.yml	28824
15	.ipynb	158 GB		
28	.py	59 GB		

4.3.1. PIP Package Installer

Our retriever downloaded 82 225 files with names similar to "*requirements.txt*" located in 51 212 repositories. Of these, 903 were files meant to be used with the Anaconda Package Installer instead of PIP and 683 contained at least one invalid instruction, so they would not be usable by PIP without manual user intervention.

- 35 275 files (42.9%) import all dependencies with specified versions.
- 15 128 files (18.4%) do not specify any versions.
- 26 128 files (31.8%) have versions set for some of their imports.
- 3029 files (3.7%) do not import any dependencies directly.

Files in the last category may import their dependencies from different sources than the Python Package Index. This is suboptimal, as that resource is not necessarily available to a researcher trying to reproduce the results of the publication. These "bad patterns" are found in all four categories listed above and account for:

- 3807 files (4.6%) use instructions to add alternative URLs to lookup packages.
- 6122 files (7.4%) install packages directly from a GitHub repository or URL.
- 1696 files (2.0%) install dependencies from a file path on the local system.

Figure 2 shows the number of replication packages uploaded to GitHub based on the date of the publication to which it belongs. It also gives the absolute number of PIP environment files, split by the way dependencies are imported. The data from 2024 and 2025 appears to be still incomplete in the *Papers With Code* dataset that we used for our research. Figure 3 shows the relative distribution of the same data.

When we ignore the possible data error in 2014, we can observe that the ratio of PIP environment files in replication packages has been constantly increasing from 20% in 2015 to above 50% in recent years. This could indicate a greater awareness of the reproducibility crisis in the academic world. The ratio of PIP environment files specifying versions for *some* of their dependencies has almost doubled over time, which might also be an indicator that version constraints are decided on a per-package basis more frequently.

4.3.2. Anaconda Package Installer

Our retriever downloaded 15 320 files with file extensions "*.yaml*" or "*.yml*". Of these, 14 910 (97.3%) have valid YAML syntax, and 284 (1.9%) are *YAML templates*, which means that they would need to be preprocessed first by an additional template engine.

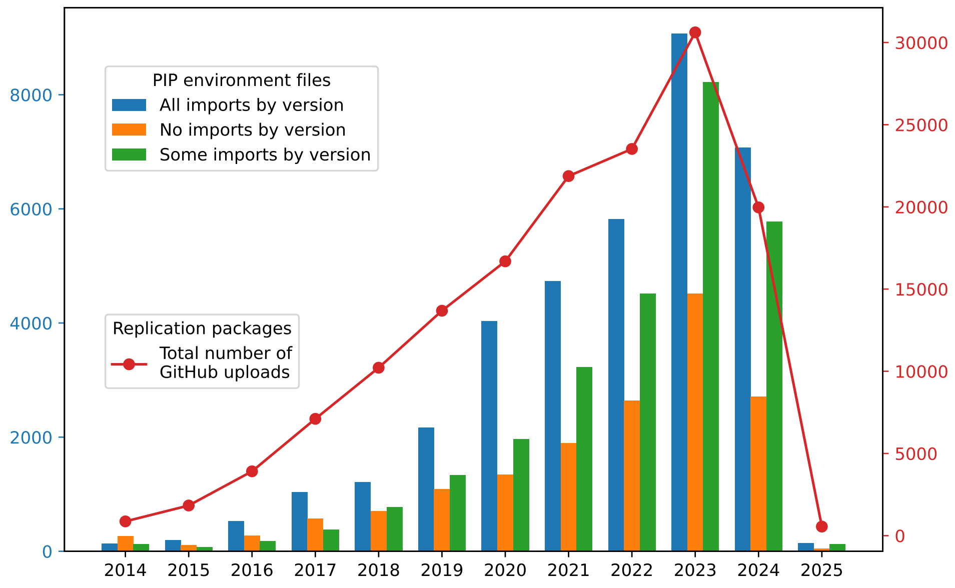

Figure 2: Replication package and PIP environment file counts per year.

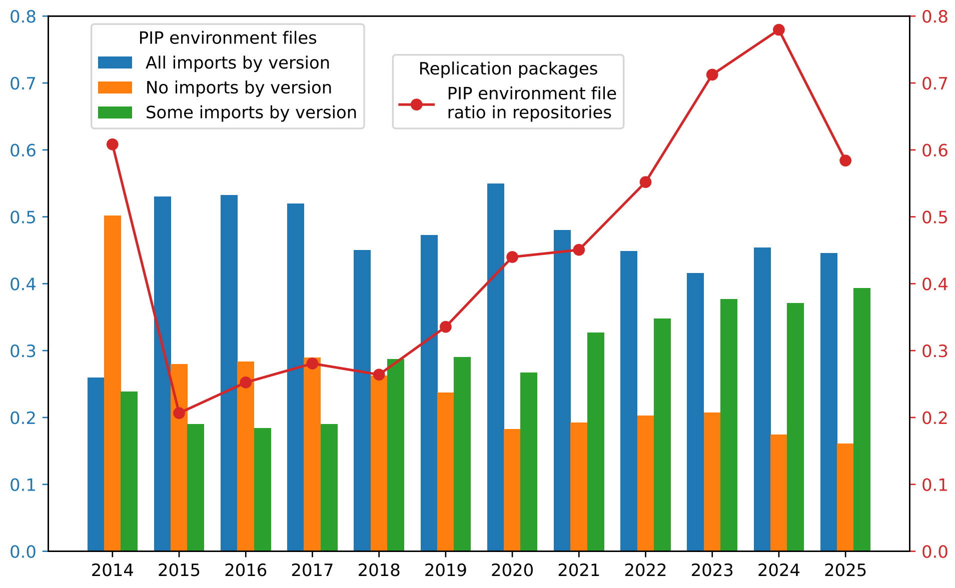

Figure 3: Distribution of replication packages and PIP environment files per year.

The number of Anaconda dependency files is 11 523 distributed across 9452 repositories. The package versioning usage is split up as follows:

- 6876 files (59.7%) import all dependencies with specified versions.
- 324 files (2.8%) do not specify any versions.
- 4027 files (34.9%) have versions set for some of their imports.
- The remaining files are invalid, empty, or do not list any dependencies.

4.3.3. Packaging Tools and Docker Files

We observe the following file types used in the build process of Python packages or Docker containers:

- 27 965 (18.4%) repositories contain at least one file related to *packaging* (total: 45 863 files)
- 10 898 (7.1%) repositories contain at least one file related to *Docker* containers (total: 43 001 files)

Rank	Package	With Version	No Version
1	numpy	30387	13323
2	torch	26986	5419
3	scipy	20007	8026
4	tqdm	17777	10050
5	matplotlib	17889	9629
6	pandas	16471	7400
7	torchvision	14274	4510
8	scikit-learn	12159	5289
9	pillow	11146	2518
10	transformers	10257	2706
11	six	11746	1097
12	requests	10543	2060
13	opencv-python	6888	4817
14	protobuf	9103	2489
15	tensorboard	7436	3663
16	python-dateutil	10237	222
17	datasets	7447	2143
18	pyparsing	9414	133
19	pyyaml	8627	799
20	h5py	7377	1965
21	certifi	9113	141
22	urllib3	8962	177
23	idna	8702	86
24	pytz	8522	166
25	tensorflow	7327	1225

(a) PIP

Rank	Package	With Version	No Version
1	python	10279	201
2	pip	7161	2209
3	numpy	7305	1200
4	scipy	6072	1037
5	matplotlib	5384	1219
6	tqdm	5484	1005
7	setuptools	6089	131
8	wheel	5996	71
9	torchvision	5392	665
10	pandas	5138	854
11	certifi	5878	54
12	openssl	5856	55
13	pillow	5621	282
14	six	5815	62
15	ca-certificates	5804	59
16	xz	5819	23
17	tk	5799	29
18	zlib	5738	32
19	libffi	5704	22
20	sqlite	5629	26
21	readline	5601	22
22	ncurses	5600	21
23	pytorch	5036	540
24	python-dateutil	5459	34
25	scikit-learn	4605	668

(b) Anaconda

Table 2
Top-25 imported packages using PIP and Anaconda

4.3.4. Package Distribution

Table 2a shows the Top-25 packages imported in the PIP file format. As expected, many well-known Machine Learning dependencies appear on top of that list. The proportion of entries with and without version information is not distributed evenly and partially unexpected:

- We assumed that code directly related to numeric computation (e. g., ”numpy”, ”scipy”, ”pandas”) would rarely appear without version information, since the computational results may change even with small code updates. However, they appear to be imported rather frequently (30.1%) without version information.
- ”certifi” contains a list of currently trusted TLS root certificates. Since they become invalid once they reach their expiration date, we would assume that nobody intentionally enforces using old certificates because of the security implications. However, almost all imports (98.5%) are locked to specific versions.
- ”urllib3” has far less unversioned appearances than ”requests”, even though both packages are HTTP clients and fulfill very similar tasks.

Table 2b shows the Top-25 packages imported in the Anaconda YAML file format. The package ”python” specifies which release version should be used to setup the Python runtime environment. ”pip” is a meta-package and allows importing packages from the Python Package Index which are otherwise unavailable in Anaconda itself.

Similarly to the PIP overview, the typical computational Machine Learning packages rank next with moderate amounts of non-versioned imports. We would assume that packages such as ”zlib”, ”readline”, ”xz” or ”ca-certificates” almost never introduce breaking changes with updated library versions; however, these types of packages have a very low number of unversioned uses.

5. Conclusion

In this study, we have analyzed the availability of files in replication packages which facilitate reconstituting the original runtime environments and, therefore, foster *reproducibility* for scientific publications. We focused our research on replication packages from the field of Machine Learning which are hosted on GitHub. Our results show that most repositories contain source code for the Python programming language and are generally small both in file count and size.

We observed that less than half of the repositories include file types which potentially facilitate reconstituting the Python runtime environment used for the respective original scientific work with the appropriate software and dependency versions. Additionally, these files are often of poor quality and might link to arbitrary URLs or local files instead of using official package indices. Although we observed a significant growth in the number of replication packages that include dependency information files in the last decade, there is still much work to be done.

A future goal might be the automated setup and execution of a replication package. For this to work, we would need to know which of the possibly multiple environment files to apply and what commands to execute after that. Such information might be buried somewhere deep in the documentation – if it exists and we are able to locate it.

The use of Docker containers – which we only looked at from the sidelines in this study – or similar solutions could greatly improve reproducibility. Conferences or journals could require the authors to use provided containers for their submissions, similar to the prerequisite of using a given LaTeX template. This could include using fixed directory structures and commands to execute and would allow the automated setup and execution of replication packages.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] S. Siebert, L. M. Machesky, R. H. Insall, Overflow in science and its implications for trust, eLife 4 (2015). doi:10.7554/elife.10825.

[2] T. Hsieh, M. H. Vaickus, D. G. Remick, Enhancing scientific foundations to ensure reproducibility, The American Journal of Pathology 188 (2018) 6–10. doi:10.1016/j.ajpath.2017.08.028.

[3] X.-L. Meng, Reproducibility, replicability, and reliability, Harvard Data Science Review 2 (2020). doi:10.1162/99608f92.dbfce7f9.

[4] C. Willis, V. Stodden, Trust but verify: How to leverage policies, workflows, and infrastructure to ensure computational reproducibility in publication, Harvard Data Science Review 2 (2020). doi:10.1162/99608f92.25982dcf.

[5] M. Baker, 1,500 scientists lift the lid on reproducibility, Nature 533 (2016) 452–454. doi:10.1038/533452a.

[6] M. Mieskes, K. Fort, A. Névéol, C. Grouin, K. Cohen, Nlp community perspectives on replicability, in: Proceedings - Natural Language Processing in a Deep Learning World, RANLP 2019, Incoma Ltd., Shoumen, Bulgaria, 2019, pp. 768–775. doi:10.26615/978-954-452-056-4_089.

[7] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Lariviere, A. Beygelzimer, F. d'Alche Buc, E. Fox, H. Larochelle, Improving reproducibility in machine learning research(a report from the neurips 2019 reproducibility program), Journal of Machine Learning Research 22 (2021) 1–20. URL: http://jmlr.org/papers/v22/20-303.html.

[8] R. Stojnic, Ml code completeness checklist, 2022. URL: https://github.com/paperswithcode/releasing-research-code, accessed 2025-07-16.

[9] I. Magnusson, N. A. Smith, J. Dodge, Reproducibility in nlp: What have we learned from the

checklist?, in: Findings of the Association for Computational Linguistics: ACL 2023, Association for Computational Linguistics, 2023, pp. 12789–12811. doi:10.18653/v1/2023.findings-acl.809.

[10] M. Arvan, L. Pina, N. Parde, Reproducibility in computational linguistics: Is source code enough?, in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2022, pp. 2350–2361. doi:10.18653/v1/2022.emnlp-main.150.

[11] A. Belz, S. Agarwal, A. Shimorina, E. Reiter, A systematic review of reproducibility research in natural language processing, in: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, Association for Computational Linguistics, 2021. doi:10.18653/v1/2021.eacl-main.29.

[12] Nep 29 — recommend python and numpy version support as a community policy standard, 2019. URL: https://numpy.org/neps/nep-0029-deprecation_policy.html, accessed 2025-07-16.

[13] PapersWithCode, Papers with code datasets, 2024. URL: https://github.com/paperswithcode/paperswithcode-data, accessed 2024-11-12.

[14] March 2025 public data file from crossref, 2025. doi:10.13003/87bfgcee6g.

[15] DataCite, Datacite public data file 2024, 2025. doi:10.14454/TJPC-9M93.

[16] J. Priem, H. Piwowar, R. Orr, Openalex: A fully-open index of scholarly works, authors, venues, institutions, and concepts, 2022. doi:10.48550/ARXIV.2205.01833.

[17] T. Saier, J. Krause, M. Färber, unarxive: All arxiv publications pre-processed for nlp, including structured full-text and citation network (full), 2023. doi:10.5281/ZENODO.7752754.