# Code and Data are not all you need for reproducibility

Bohui Lyu[1], John Bonini[2], Danel Wines[2], Kangming Li[1]*
[1][Division of Physical Sciences and Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal 23955-6900, Saudi Arabia] [2][Material Measurement Laboratory, National Institute of Standards and Technology, Gaithersburg, Maryland 20899, United States]

*Correspondence to: [Kangming Li]* kangming.li@kaust.edu.sa

## 1. Introduction

Machine learning has become an integral tool in materials science, making the reproducibility of computational workflows and predictive results increasingly critical, as emphasized by the FAIR principles. To support transparent comparison and reuse, the community has developed benchmark platforms, [1-3] which standardize datasets and evaluation pipelines and typically require the source code along with software requirements in accompanying metadata to facilitate re-execution. Such benchmarks implicitly assume that these artifacts are sufficient for reliable reproduction of reported results.

In practice, however, the reproducibility of benchmarked results depends not only on the availability of code and data, but also on the executability of the provided software environments and workflows—an aspect that has received limited systematic scrutiny. As a case study, we assess practical reproducibility on Matbench by evaluating 28 algorithms using the provided source code and documented environment specifications. For each algorithm, we attempted to reproduce one representative task selected from the authors' reported results by re-executing the original evaluation pipeline.

Notably, only 7% of the evaluated algorithms could be successfully re-executed using the provided code and environment specifications without modification. After systematic efforts to repair the documented software environments, we find that only 13 of the 28 algorithms regenerate outputs that match those reported by the platform. Similar reproducibility issues are observed on the JARVIS-Leaderboard, with a complete accounting in progress. Our findings highlight the practical difficulty of reproducing reported results using the currently provided artifacts. Accordingly, we identify common obstacles to re-execution and propose platform-level recommendations for improving benchmark reproducibility.

## 2. Results and discussion

### *2.1 Set up Python environment*

For the 28 algorithms uploaded to Matbench, we followed a three-stage procedure for environment provisioning and, grouped them into four categories by three stages of operation. Details of the classification results are provided in Appendix A. In stage 1, using the dependency specifications provided on the platforms, we attempted to install each environment and verified that all packages/functions required by the execution scripts could be imported. Among 28 distinct algorithms, only 2 of them could pass this test. The failures mainly fall into two categories: (i) unavailability of specific pinned package versions. (ii) version conflicts among dependencies. In stage 2 we extracted the error logs and attempted to fix the environment setup process. Following these error-guided fixes, 15 of the remaining 26 algorithms installed successfully. For the remaining 11 algorithms, we observed three principal issues. (i) several packages invoked by the code were not fully enumerated in the metadata file files, resulting in import errors. (ii) the information provided in metadata file was not machine-readable in practice. (iii) even after removing all version pins, some environments still exhibited dependency conflicts that prevented installation. Therefore, in stage 3, we manually addressed the remaining environment issues. In this stage, we successfully set up environments for 9 algorithms; the remaining 2 still failed to install.

### *2.2 Benchmark result generation*

After provisioning these environments, we used the execution scripts provided by the authors to reproduce the benchmark results. Of the 26 provisioned environments, 13 produced benchmark outputs, while the remainder failed at runtime primarily due to API/entry-point changes and module refactoring. For the algorithms deemed reproducible, our reproduced metrics were nearly identical to the reported values, with the difference in mean RMSE smaller than 8%. (Fig. 1) Our results further indicate that, given executable code and a correctly provisioned environment, the benchmark outcomes are highly reproducible, with only minimal deviations.

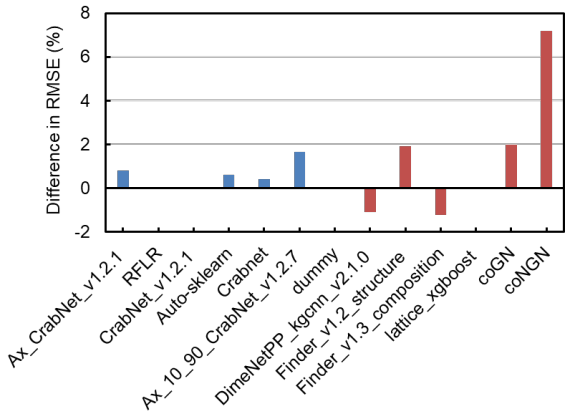


Fig. 1 Difference in mean RMSE of the algorithms on different benchmark tasks. The composition-based algorithms are in colour blue, the structure-based algorithms are in colour red.

### *2.3 Recommendations for better reproducibility*

Based on our observations, we propose several recommendations for benchmark platform maintainers to improve the reproducibility of submitted algorithms. (1) When a new algorithm is submitted to the GitHub repository, GitHub Actions–based CI workflow can be used to automatically provision and validate the Python environment; submissions should be accepted only if they pass environment setup and import checks. (2) It would be better if the authors of the algorithms could provide an `environment.yml` file, which is more user-friendly and facilitates environment reconstruction. (3) As package ecosystems evolve, an `environment.yml` may become insufficiently robust to resolve the intended dependency set. In such algorithms, tools such as `conda-lock` can strengthen reproducibility by generating a fully resolved lockfile that explicitly specifies the concrete dependency versions to be installed. (4) For some algorithms, the target package is no longer available, making it impossible for even a lockfile to retrieve the required artifacts. In such situations, another practical option is 'conda-pack', which packages and exports the entire environment into a relocatable archive that users can activate and run directly. Notably, all four steps above can be automated within GitHub Actions.

## 3. Conclusion

Our case study shows that reproducing widely used materials-ML benchmark results can be challenging in practice, even when source code and environment specifications are provided. We suggest that platform-level enforcement, such as automated continuous integration (CI) checks (e.g., GitHub Actions)—could be used to validate submissions and generate reusable environment artifacts, thereby substantially improving the reliability and usability of benchmark platforms.

## References

[1] Dunn, A. *et al.* Benchmarking materials property prediction methods: the Matbench test set and Automatminer reference algorithm. *npj Comput. Mater.,* 6, 138 (2020).
[2] Choudhary, K. *et al.* JARVIS-Leaderboard: a large scale benchmark of materials design methods. *npj Comput. Mater.,* 10, 93 (2024).
[3] Riebesell, J. *et al.* A framework to evaluate machine learning crystal stability predictions. *Nat. Mach. Intell.,* 7, 836–847 (2025).

## Appendix A. [Categories of Matbench algorithms based on reproducibility of the python environment.]

| Categories | Algorithms |
|---|---|
| Cat 1: Can setup the environment directly (7%) | Auto-sklearn, Ax_10_90_CrabNet_v1.2.7 |
| Cat 2: Can setup the environment by auto-fixing (54%) | alignn, Ax_CrabNet_v1.2.1, Ax_SAASBO_CrabNet_v1.2.7, DimeNetPP_kgcnn_v2.1.0, dummy, Finder_v1.2_composition, Finder_v1.2_structure, GN-OA, gptchem, MegNet_kgcnn_v2.1.0, modnet_v0.1.10, modnet_v0.1.12, RFLR, SchNet_kgcnn_v2.1.0, TPOT |
| Cat 3: Can setup the environment with human effort (32%) | automatminer_expressv2020, coGN, coNGN, cgcnnv2019, CrabNet, CrabNet_v1.2.1, darwin, lattice_xgboost, matformer |
| Cat 4: Not reproducible despite substantial effort (7%) | DeeperGATGNN, rf |