JunoBench: A Benchmark Dataset of Crashes in Python Machine Learning Jupyter Notebooks

Yiran Wang
yiran.wang@liu.se
Linköping University
Linköping, Sweden

José Antonio Hernández López
joseantonio.hernandez6@um.es
University of Murcia
Murcia, Spain

Ulf Nilsson
ulf.nilsson@liu.se
Linköping University
Linköping, Sweden

Dániel Varró
daniel.varro@liu.se
Linköping University
Linköping, Sweden

Abstract

Jupyter notebooks are widely used for machine learning (ML) prototyping. Yet, few debugging tools are designed for ML code in notebooks, partly, due to the lack of benchmarks. We introduce JunoBench, the first benchmark dataset of real-world crashes in Python-based ML notebooks. JunoBench includes 111 curated and reproducible crashes with verified fixes from public Kaggle notebooks, covering popular ML libraries (e.g., TensorFlow/Keras, PyTorch, Scikit-learn) and notebook-specific out-of-order execution errors. JunoBench ensures reproducibility and ease of use through a unified environment that reliably reproduces all crashes. By providing realistic crashes, their resolutions, richly annotated labels of crash characteristics, and natural-language diagnostic annotations, JunoBench facilitates research on bug detection, localization, diagnosis, and repair in notebook-based ML development.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Benchmark, software bugs, machine learning, Jupyter notebooks

ACM Reference Format:
Yiran Wang, José Antonio Hernández López, Ulf Nilsson, and Dániel Varró. 2025. JunoBench: A Benchmark Dataset of Crashes in Python Machine Learning Jupyter Notebooks. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Machine learning (ML) and deep learning (DL) are increasingly used in modern software programs, with Jupyter notebooks emerging as a dominant development environment for Python-based ML prototyping [2, 10] (referred to as *ML notebooks*). Notebooks offer a flexible, interactive interface that supports incremental coding, intermediate feedback, and custom execution order for notebook cells. These features make notebooks well suited for exploratory ML development, but also introduce unique debugging challenges [2]. When combined with the inherent complexity of data manipulation and the intensive use of advanced ML and DL libraries, these factors further increase the likelihood of errors during ML prototyping [3].

Existing tools offer limited support for identifying ML bugs, especially in the notebook environment [1]. One critical missing component are benchmarks. Benchmarks are crucial for advancing automatic debugging research by providing standardized datasets of real-world bugs for reproducible evaluation [18]. While prior work [7–9, 23, 25] focus on ML bugs in Python scripts (`.py` files), they do not account for the unique semantics of notebooks (`.ipynb`). Furthermore, existing ML bug benchmarks (e.g., [9]) often mix bug symptoms such as poor performance, incorrect functionality, and crashes. Among these, crashes represent the most severe symptom, as they terminate program execution, and are the most common in ML programs [1, 3, 9]. They are particularly important in the notebook setting, as a crashing cell can alter the notebook's execution state and lead to unexpected behavior in subsequent cells.

To fill this gap, we present *JunoBench*, the first benchmark of crashes in real-world Python ML notebooks. JunoBench supports the evaluation and development of debugging tools for notebook-based ML development by providing:

- 111 curated and reproducible real-world crashes from Python ML notebooks, each paired with a verifiable fix;
- Balanced coverage of major ML libraries (e.g., *TensorFlow/Keras*, *PyTorch*, *Scikit-learn*, *Pandas*, *NumPy*, *Matplotlib*, *Seaborn*) and notebook-specific failures such as out-of-order execution;
- Crash categorizations along four dimensions: library cause, crash type, root cause, and ML pipeline stage;
- Ground-truth labels for crash detection and diagnosis;
- A unified execution environment that reproduces all notebooks.

JunoBench is available on HuggingFace [21], and all artifacts used to construct the benchmark are provided on GitHub [20].

2 Methodology

This section outlines our methodology for constructing JunoBench (see Figure 1), including data collection, benchmark inclusion, and how we ensure reproducibility and usability.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Data collection

Data sources. The crashing notebooks in JunoBench are derived from the dataset released in our prior empirical study on public Python ML notebook crashes [22]. That study analyzed 64,031 notebooks containing crashes and manually labeled 746 of them with detailed annotations, including *root cause* and the ML library used at the point of failure (i.e., the *library cause* label).

The original pool comprises notebooks collected from GitHub and Kaggle. This paper focuses specifically on *Kaggle* notebooks (2,869 notebooks with 3,875 crashes, of which 356 are labeled), due to their stronger support for reproducibility. Kaggle notebooks typically reference publicly accessible datasets, facilitating input retrieval and consistent reproduction.

Crash filtering. We started from the 356 labeled crashes and selected those that (1) involved ML libraries (based on *library cause*) or (2) were due to notebook-specific issues, such as out-of-order execution (based on *root cause*). Moreover, we excluded crashes that could not be reproduced or fixed within the notebook context, such as those whose root causes were: (1) insufficient resource (hardware-dependent), (2) incompatible library versions (requiring complex environment setups), (3) bugs inside third-party libraries. After filtering, 174 candidate crashes remained for benchmark inclusion.

Resampling. The filtered labeled crashes are unevenly distributed across ML libraries. To achieve balanced coverage, we resampled 500 additional crashes from the unlabeled Kaggle pool, selecting notebooks using at least one of the ML libraries used in the filtered set (i.e., TensorFlow/Keras, PyTorch, Scikit-learn, Pandas, NumPy, Matplotlib, Seaborn, Statsmodels, Torchvision, and LightGBM).

Due to the non-linear execution order of notebooks, identifying the relevant ML libraries involved in the crashing code cell is challenging without manual inspection. To assist this process, we developed a tool that automatically estimated the *library cause* of each crash by assigning scores to libraries based on their presence in the crashing cell and traceback in the cell output, with higher weights for those appearing in the crashing line. The heuristic achieved 80% accuracy when evaluated against labeled crashes. It was then used to guide our prioritization of selecting resampled notebooks for manual benchmark inclusion.

2.2 Benchmark inclusion

Input dataset retrieval. For each candidate notebook, we first retrieved the metadata of the required input datasets using KGTorrent [13] with the Meta Kaggle [15] dataset, including dataset titles, downloading links, and licenses.

We attempted to download the datasets using the retrieved links. Because our focus is on crash reproduction rather than performance of ML models, the use of full training data is not relevant. For datasets exceeding 100MB, we evaluated whether downsampling preserved the crash behavior without altering code logic. For instance, a 2.5GB image dataset was reduced to 94MB by retaining just 4% of images per class while still reproducing the crash.

Manual reproducing and fixing. In our previous empirical study, the dataset was organized at the crash level, where multiple crashes could originate from the same notebook. In JunoBench, we instead design each benchmark instance as an independent notebook containing a single, isolated, and reproducible crash. This ensures that each case is self-contained, simplifying evaluation and comparison across automated debugging tools.

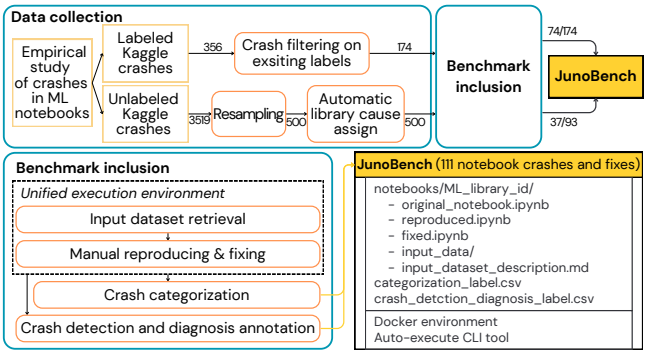


Figure 1: Overview of the benchmark construction process.

For each notebook, we maintain three versions: (1) the *original* notebook as collected, (2) a *reproduced* version containing minimal adjustments (e.g., the sequence of execution order for crash reproduction, dataset paths, training settings), and (3) a *fixed* version where the crash is repaired.

For each crash, we executed only the minimal subset of code cells required to trigger the crash, which serves as one valid execution sequence of the crash, although alternatives may exist. This sequence is reflected in the cell execution counts, which are preserved in the fixed version for comparability. A crash is considered fixed when it no longer occurs and the notebook's original intent remains intact.

We aim to construct a balanced benchmark across ML libraries with inclusion of notebook-specific issues. For the 174 labeled candidates, we successfully reproduced and fixed 74 crashes. We then continued with the 500 unlabeled candidates, prioritizing reproduction based on the automatically inferred *library cause*. To maintain a balanced coverage of ML libraries, we monitored library distribution throughout the process and halted reproduction once sufficient coverage was achieved. From the unlabeled set, 93 crashes were examined, and 37 were successfully reproduced and fixed, while others were excluded primarily due to missing or inaccessible datasets.

In total, JunoBench includes 111 reproducible crashes and corresponding fixes, covering both notebook-specific execution issues and a balanced set of diverse ML libraries.

Crash categorization. We adopted the crash categorization scheme from our prior study [22] to annotate all crashes in JunoBench, ensuring consistency and facilitating future analysis. Each crash is labeled along four dimensions: (1) *library cause*, (2) *crash type* (e.g., tensor shape mismatch, unsupported broadcast), (3) *root cause* (e.g., API misuse, data confusion), and (4) *ML pipeline stage* (i.e., where in the general ML workflow the crash occurs, see Figure 2d).

For the 74 labeled crashes, we revalidated and refined the annotations, especially the *root cause* labels, based on our verified fixes. We identified seven mislabels. For example, a crash initially labeled as data confusion was found to stem from an incorrect value passed to the `plot_acf` API in `statsmodels`, indicating API misuse. All corrections are documented in our GitHub repository [20].

For the 37 unlabeled crashes, we followed the same annotation protocol as in our prior study [22]. Specifically, we applied grounded theory methods [16, 17], using first-cycle coding by one annotator and second-cycle review by another. Disagreements were resolved through discussion. A total of five disagreements were discussed, and full consensus was reached on all final labels.

Crash detection and diagnosis annotation. To support automated crash identification and diagnosis, we provide ground-truth annotations at two levels: (1) a *crash detection label*, assigned at the notebook level (*true* for reproduced, *false* for fixed), and (2) a *crash diagnosis label*, describing the cause and localization of each crash with a short natural language explanation. Each diagnosis label was created by one annotator using evidence from the crashing code cells, the error outputs, and the fixed versions, and then independently validated by a second annotator. In cases of disagreement, the two annotators conducted discussions until full consensus was achieved. Six such cases were resolved in total.

Input dataset license. Most datasets in JunoBench are publicly available and licensed for research use. In four cases where the original datasets (all numeric) are under restrictive license, we replaced them with synthetic equivalents by replicating the schema and missing-value structure while generating random values. Finally, each dataset is accompanied by a metadata file documenting its title, source URL, license, and modifications (e.g., downsampling or synthetic substitution).

2.3 Benchmark usability

To improve usability, we provide a unified execution environment. We built a Docker image based on the official Kaggle notebook platform [6] and extended it with the dependencies required by all JunoBench notebooks. This one-time setup guarantees consistent execution across all cases.

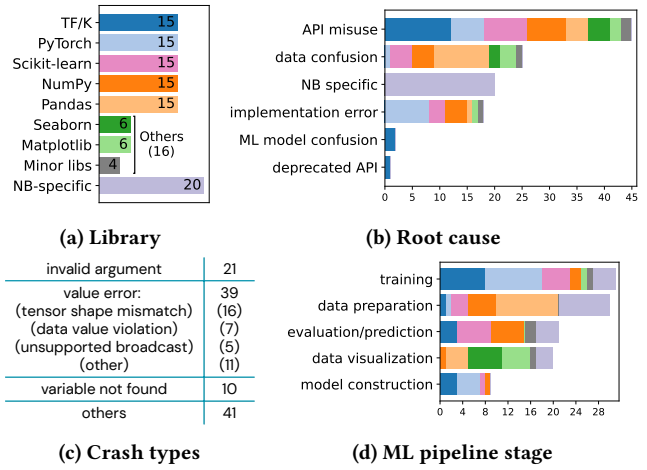
We also provide an execution-level interface [26], implemented as a command-line tool. It automatically runs both buggy and fixed versions with provided cell execution order, and verifies that the buggy version crashes while the fixed version executes successfully. This interface enables users to reproduce and validate all benchmark cases with a single command.

3 JunoBench

We present the characteristics of JunoBench in four dimensions: *library cause*, *root cause*, *crash type*, and *ML pipeline stage*.

Library cause. JunoBench comprises 111 notebook crashes with balanced coverage across popular ML libraries, as well as notebook-specific issues. The distribution is shown in Figure 2a. Specifically, the benchmark includes:

- 15 crashes each involving the DL libraries *TensorFlow/Keras* and *PyTorch*, the classical ML library *Scikit-learn*, and the data processing libraries *NumPy* and *Pandas*;
- 16 crashes involving a mix of ML libraries: 12 related to visualization (6 from *Seaborn* and 6 from *Matplotlib*) and individual cases from *Statsmodels*, *TorchVision*, and *LightGBM*;
- 20 notebook-specific (i.e., out-of-order cell execution) crashes.



(a) Library (b) Root cause

(c) Crash types (d) ML pipeline stage

Figure 2: Characteristics of JunoBench. Each bar in (b) and (d) is segmented by libraries (a). "TF/K" stands for "TensorFlow/Keras". "Minor libs" include Statsmodels(2), TorchVision(1), and LightGBM(1).

This distribution highlights JunoBench's diverse coverage of challenges in ML notebook development, spanning DL, classical ML, data processing, and visualization libraries, as well as execution order issues unique to interactive notebook environments [22].

Root cause. The root causes of the 111 crashes are shown in Figure 2b. In addition to the 20 *notebook-specific* out-of-order execution failures (e.g., re-running a code cell that deletes a column, resulting in an error when the column is referenced again), the majority of remaining crashes fall into three main categories. The most frequent is *API misuse* (45 cases), where developers call library APIs with invalid arguments or unsupported usage patterns. *Data confusion* (25 cases) follows, stemming from misunderstandings of data shape, type, or structure. *Implementation errors* (18 cases) account for logic mistakes such as incorrect variable references or flawed algorithm design. JunoBench also includes less common but representative real-world ML issues. For example, *ML model confusion* (2 cases) occurs when attempting to load a model with an incompatible architecture or from a different framework. The rest are caused by *deprecated APIs* that were removed in newer library versions.

Crash type. Crashes in JunoBench span a variety of types, as summarized in Figure 2c. The most frequent is *invalid argument* (21 cases), where function calls violate API constraints due to incorrect or missing parameters. Tensor-related issues such as *tensor shape mismatch* (16) and *unsupported broadcast* (5) are also common, particularly in TensorFlow/Keras and PyTorch notebooks, where tensor alignment is often complex. *Data value violation* (7) occurs when input values violate API or code assumptions. Another frequent category is *variable not found* (10), all caused by out-of-order cell execution, where variables defined in unexecuted cells are referenced later. These crash types capture how developers frequently struggle with runtime dynamics, data integrity, and ML library usage, reflecting ML-specific challenges in notebook development.

ML pipeline stage. JunoBench captures crashes across key stages of a typical ML pipeline (Figure 2d). Most crashes occur during *model training* (31 cases), primarily involving TensorFlow/Keras

and PyTorch, reflecting the complexity of using DL libraries. *Data preparation* follows closely (30), often affected by out-of-order execution and data frame handling errors involving Pandas. Model *evaluation and prediction* account for 21 crashes, typically involving incorrect metric usage or incompatible model inputs related to Tensorflow/Keras, Scikit-learn, and NumPy. *Data visualization* introduces 20 crashes, largely tied to Matplotlib and Seaborn, but also reflecting upstream data processing issues in Pandas and NumPy. Finally, *model construction* contributes 9 crashes, mostly due to misconfigured models with DL libraries.

4 Potential research opportunities

Evaluating crash detection and repair tools. JunoBench provides reproducible crashes with verifiable fixes, a unified execution environment, and detailed labels along four dimensions: *library cause*, *crash type*, *root cause*, and *ML pipeline stage*. Together, these allow development and benchmarking of crash detection and repair tools designed for Python-based ML notebook environments, supporting comprehensive and replicable studies. The structured labels further enable comparative evaluation across crash types, causes, and ML libraries, and encourage research on ML pipeline stage-aware debugging and reliability improvement.

Evaluating large language models (LLMs). Given the growing use of LLMs for software engineering, JunoBench provides *crash detection* and *crash diagnosis* labels, providing ground-truth for evaluating LLM-based crash analysis. These labels support tasks beyond binary crash detection, such as crash diagnosis and repair, bridging the gap between detection and deeper crash understanding.

Initial experiments on LLM-based crash detection. To demonstrate the utility of JunoBench and verify that it comprises challenging, non-trivial cases, we conducted an initial evaluation of two LLMs (i.e., *llama3.3:70b* and *mistral-small3.1*) as crash detectors. Given executed code cells and a target cell, each model predicted whether the target cell (the crashing cell in the reproduced version and the corresponding cell in the fixed version) would crash, framing the task as binary classification against the *crash detection labels*.

We adopted a low temperature of 0.1 to encourage deterministic predictions. During the experiment, we excluded 14 notebook pairs that exceeded token limits (i.e., 11 for Llama, 3 for Mistral), resulting in 200 and 216 total predictions, respectively. Llama correctly classified 115 out of 200 cases (reproduced: 58/100, fixed: 57/100), while Mistral achieved 128/216 (reproduced: 63/108, fixed: 65/108). The results suggest that JunoBench presents a meaningful and non-trivial benchmark for LLM-based crash detection. Consequently, more complex tasks beyond detection, such as crash diagnosis and repair, are expected to pose even greater challenges. Full experiment details and outputs are available in our GitHub repository [20].

5 Related work

Several studies have introduced bug benchmarks in Python-based ML applications. Morovati et al. [9] proposed Defect4ML, a benchmark of 100 reproducible bugs in ML systems using TensorFlow/Keras. The bugs were collected from GitHub, Stack Overflow, and prior studies [4, 11, 23, 25]. Liang et al. [8] presented gDefects4DL, a curated set of 64 DL bugs from TensorFlow/Keras and PyTorch projects. However, the download links for their buggy programs and replication environments are no longer active. Kim et al. [7] introduced Denchmark, a large-scale dataset comprising 4,577 bug reports from DL projects. While offering broad coverage, it lacks reproduction-ready artifacts, making it unsuitable for studies requiring program execution or tool evaluation [26].

Several other ML studies have released bug datasets as byproducts of empirical analyses. Zhang et al. [25] reproduced 151 TensorFlow bugs in their empirical study. Wu et al. [24] introduced Tensfa, a dataset specialized for tensor shape mismatches in TensorFlow/Keras applications. While valuable, these datasets rely on outdated Python and TensorFlow versions without detailed dependency specifications, which limits their relevance and reproducibility to modern ML development [5].

All of the above studies target ML applications written as Python scripts (`.py` file) rather than Jupyter notebooks (`.ipynb`). While several notebook datasets exist, they focus on general notebook usage [2, 13], reproducibility [19], or quality analysis [12, 14], and none include bug benchmarks. One notable exception is the empirical study by De Santana et al. [1] on bugs in Jupyter notebook projects. However, their released dataset does not include reproducible artifacts, such as the buggy and fixed notebooks, execution environments, or input data.

In contrast, JunoBench targets crashes in ML notebooks, providing reproducible buggy-fixed notebook pairs together with the necessary input datasets. To our best knowledge, it is the first benchmark dataset of ML notebook crashes. While prior benchmarks [8, 9, 24, 25] focus on TensorFlow/Keras and PyTorch, JunoBench spans a wider set of ML libraries, capturing popular libraries used in real-world ML development. Moreover, while many existing benchmarks [8, 9, 23] require setting up separate environments for each case, JunoBench provides a unified execution environment that ensures consistent reproduction of all cases with a single setup.

6 Conclusion and future work

This paper presents JunoBench, the first benchmark of real-world crashes in Python-based ML notebooks. JunoBench comprises 111 curated and reproducible crashes with verifiable fixes, spanning diverse ML libraries, crash types, root causes, and ML pipeline stages. It provides diagnosis labels and a unified environment to ensure consistent reproduction and facilitate standardized evaluation of debugging tools. By providing this curated and reproducible dataset, JunoBench lays the groundwork for advancing reliability and automated debugging in notebook-based ML development.

While JunoBench provides reproducible crashes in modern ML workflows, its coverage is limited to commonly used libraries and current released versions, as well as pipeline stages during development, leaving some specialized libraries, older library versions, and deployment-specific issues unrepresented. Future extensions could broaden library coverage, maintain library version-specific behaviors, and capture crashes during deployment.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the Software Center Project 61.

References

[1] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–34. https://doi.org/10.1145/3641539

[2] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. 2022. A large-scale comparison of Python code in Jupyter notebooks and scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 353–364. https://doi.org/10.1145/3524842.3528447

[3] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[4] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1135–1146. https://doi.org/10.1145/3377811.3380378

[5] Gunel Jahangirova, Nargiz Humbatova, Jinhan Kim, Shin Yoo, and Paolo Tonella. 2024. Real Faults in Deep Learning Fault Benchmarks: How Real Are They? arXiv:2412.16336

[6] Kaggle. 2025. Kaggle Docker Image GitHub Repository. https://github.com/Kaggle/docker-python.

[7] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2021. Denchmark: A Bug Benchmark of Deep Learning-Related Software. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (Madrid, Spain, 2021-05). IEEE Press, New York, NY, USA, 540–544. https://doi.org/10.1109/MSR52588.2021.00070

[8] Yunkai Liang. 2022. gDefect4DL- A Dataset of General Real-World Deep Learning Program Defects. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (Pittsburgh, Pennsylvania, 2022) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 90–94. https://doi.org/10.1145/3510454.3516826

[9] Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming Jiang. 2023. Bugs in Machine Learning-Based Systems: A Faultload Benchmark. *Empirical Software Engineering* 28, 3 (2023), 62. https://doi.org/10.1007/s10664-023-10291-1

[10] NASA high-end computing capability. 23 Apr, 2025. *Using Jupyter Notebook for Machine Learning Development on NAS Systems.* NASA. https://www.nas.nasa.gov/hecc/support/kb/using-jupyter-notebook-for-machine-learning-development-on-nas-systems_576.html Accessed: 2025-05-15.

[11] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 14 (Sept. 2021), 27 pages. https://doi.org/10.1145/3470006

[12] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (Montreal, QC, Canada, 2019-05). IEEE Press, New York, NY, USA, 507–517. https://doi.org/10.1109/MSR.2019.00077

[13] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2021. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (2021-05). IEEE Press, New York, NY, USA, 550–554. https://doi.org/10.1109/MSR52588.2021.00072

[14] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proceedings of the ACM on Human-Computer Interaction* 6 (2022), 1–41. Issue CSCW1. https://doi.org/10.1145/3512934

[15] Megan Risdal and Timo Bozsolik. 2022. Meta Kaggle. https://doi.org/10.34740/KAGGLE/DS/9

[16] J. Saldana. 2015. *The Coding Manual for Qualitative Researchers.* SAGE Publications, London, England. https://books.google.se/books?id=jh1iCgAAQBAJ

[17] C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[18] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[19] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (Madrid, Spain) *(ICSE '21)*. IEEE Press, New York, NY, USA, 1622–1633. https://doi.org/10.1109/ICSE43902.2021.00144

[20] Yiran Wang, José Antonio Hernández López, Ulf Nilsson, and Daniel Varro. 2025. *Source code repository of paper "JunoBench: A Benchmark Dataset of Crashes in Python Machine Learning Jupyter Notebooks"*. Linkoping University. https://github.com/PELAB-LiU/JunoBench_construct

[21] Yiran Wang, José Antonio Hernández López, Ulf Nilsson, and Dániel Varró. 2025. JunoBench (Revision ba4fb60). https://doi.org/10.57967/hf/6876

[22] Yiran Wang, Willem Meijer, Jose Antonio Hernandez Lopez, Ulf Nilsson, and Daniel Varro. 2025. Why Do Machine Learning Notebooks Crash? An Empirical Study on Public Python Jupyter Notebooks . , 2181-2196 pages. https://doi.org/10.1109/TSE.2025.3574500

[23] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (Madrid, ES, 2021-05) *(ICSE '21)*. IEEE Press, New York, NY, USA, 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[24] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2021. Tensfa: Detecting and Repairing Tensor Shape Faults in Deep Learning Systems. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)* (Wuhan, China, 2021-10). IEEE Press, New York, NY, USA, 11–21. https://doi.org/10.1109/issre52982.2021.00014

[25] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam Netherlands, 2018-07-12). Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[26] Hao-Nan Zhu, Robert M. Furth, Michael Pradel, and Cindy Rubio-González. 2025. From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets. arXiv:2504.17977