Datenbank-Spektrum (2024) 24:53–62
https://doi.org/10.1007/s13222-023-00459-w

FACHBEITRAG

PyPads

Transparent Machine Learning Experiment Tracking

Thomas Weißgerber[1] · Mehdi Ben Amor[1] · Christofer Fellicious[1] · Michael Granitzer[1]

Received: 31 May 2021 / Accepted: 9 October 2023 / Published online: 28 November 2023

Abstract
Despite algorithmic advancements in the field of machine learning, a need for improvement in the infrastructure supporting machine learning development and research has become increasingly apparent. Machine learning experiments usually tend to be more ad-hoc in nature, and results are communicated most often in the form of a publication. Experimental details are often omitted due to size or time constraints, or simply because the complexity in terms of technical setup or parametrization became intractable. Even access to code bases, disregard important properties of the environment and experimental setup, like for example random generators or computing infrastructure. At the same time, tracking and communicating an often inherently exploratory scientific process is a task with considerable effort. We explored different venues to tackle these issues from a data science engineering point of view. The efforts resulted in PyPads, a framework providing an infrastructure to extend experimental setups with logging, communication and analysis features in a mostly non-intrusive way. PyPads can be extended to different Python-based frameworks, utilizing community driven, descriptive metadata in an effort to harmonize library specific logs in an ontology. Meanwhile, we also try to emphasize similarities to practices in software engineering, which have turned out to be essential in practical applications.

Keywords Machine Learning · Reproducibility · Open Science · Automated Logging · Python

1 Introduction

In the last decade, the research and development efforts in Machine Learning and Data Science have increased significantly. This can be seen in the number of new conferences, research tracks, funding and trending searches related to the domain. Another indication can be seen in the quantity of yearly published machine learning papers and Deep Learning models, as shown in Fig. 1 for arXiv.

This growth fosters breakthroughs and may deliver cutting edge AI models that are performing efficiently in many fields of applications. However, there are operational difficulties to overcome when developing a hypothesis, conducting experiments and reporting on findings in many domains of computer science. As shown by Pawlik et al. for data reproducibility [1] and Risch et al. for web science [2], best practices have to be developed to tackle these challenges. This is especially true in special branches of the data science field, like for example in Deep Learning. While Deep Learning provides exceptional results, an inherent trade-off on understandability and robustness exists. Non-obvious changes like the introduction of adversarial examples may for example devolve classification results rapidly [3].

Some of these difficulties can be traced back to gaps in the current infrastructure for data scientists. These gaps mainly manifest when Open Science aspects are to be applied to the machine learning domain. With the great quantity of published machine learning research, a significant number of publications are not meeting one or more of the core scientific principles required for an Open Science environment. The nature of machine learning itself represents a factor for missing compliance to these principles. High data quantities as well as a high number of exper-

✉ Thomas Weißgerber
thomas.weissgerber@uni-passau.de

Mehdi Ben Amor
mehdi.benamor@uni-passau.de

[1] ITZ/IH 161, University of Passau, Innstraße 43, 94032 Passau, Germany

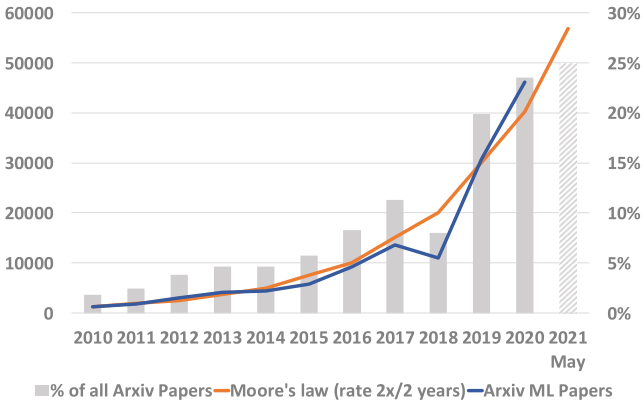

Fig. 1 Statistics for Machine Learning Papers tagged as one of cs.cv, cs.cl, cs.ai, cs.ne or stat.ml released on arXiv.com per year

imental parameters complicate persisting entire pipelines, intermediate steps and results. The resulting systems tend to be opaque and hard to debug. Results mostly are based on correlation, while arguing about causation with plain performance metrics is hard.

The idea of "reproducible research", a term introduced by Jon Claerbout in his paper on his experience creating a reproducible research environment [4], gained support in the majority of research communities ([5, 6]). One of the pressing issues in the scientific domain and related to reproducible research is the reproducibility crisis. Reproducibility crisis is when another researcher cannot reproduce the results from a piece of scientific literature. But how widespread is this reproducibility crisis? How much does it affect researchers? Taking a look at a survey done by Nature magazine in 2016 called "1,500 scientists lift the lid on reproducibility" we find that more than 70% of researchers have tried and failed to reproduce another scientist'sR experiments, and more than half have failed to reproduce their own experiments. These telling figures emerged from the survey's brief online questionnaire on reproducibility in research [7]. This means that researchers sometimes are unable to even verify their own results. If the authors themselves cannot reproduce the results, the same task will be harder for others. This reproducibility crisis has been the focal point of many studies such as [8–10] or [11], justifying the shift to more openness of the scientific community. In this context, Stodden [12] identified a research compendium based on preceding works [13]. This compendium includes the used data, the experiment code, and it's results, the paper itself and auxiliary material like interface and visualizations of the data. In summary, three minimal requirements must be highlighted: Used input data, a comprehensive paper describing the method and results and the software itself have to be easily available. Theoretically, additional high quantity of well-structured metadata about experiments should facilitate reproducibility. To further ease the communication and exploration in a scientific domain, a set of vital features was identified in the last centuries and shown to be essential to react quickly in times of crisis [14]. These properties were summarized with the term Open science [15–17] and need to be implemented in for the research domain suitable ways [18–20]. Open science encompasses practices like open access, open data, open software, which encourage researchers to make their data and source code and publications accessible and available.

In a preceding analysis, the Open Science Process Model for machine learning (OSPMML) [21] describes a non-exhaustive set of requirements for an open science ML research environment. This consists of providing replicability, reproducibility, understandability, comparability and completeness in the context of the scientific process. When navigating the steps of preparation, implementation, execution and distribution, multiple sub steps must be considered. This is needed to not only enable repeating experiments, but also allow for comparison, error analysis and reproducing them in different settings, with different data and environments. Part of these steps include the often-overlooked details about software dependencies, used hardware and drivers. Unfortunately, in the current situation logging and managing this information requires extensive coding effort, while the heterogeneous properties of used environments, libraries and versions hinder an easy reuse of existing solutions.

In this paper, we present an initial tool for an ecosystem that helps fellow researchers in creating reproducible scientific environment by mainly supporting the log production and management process. While taking virtues of software engineering into account, controlling executions and logging the results, setups and environmental information are to be made more accessible. PyPads[1] is a tool which tries to allow for automated tracking and logging of meta information about experiments in different abstraction levels and degrees of fuzziness. This is to be achieved without enforcing any conceptual paradigm to be followed by the machine learning experiment and its data.

The article is structured as follows: The basics of PyPads are discussed in Sect. 2 on the basis of a first mapping file developed for a popular python machine learning library. Sect. 3.1 discusses the concept of community driven mapping files and how we use them to offer automated and unobtrusive logging utilities, while Sect. 3.2 describes logger outputs and their schemas, an extension of an existing ontology model, and an application of PyPads in a Kubernetes cluster. Other tools and approaches for reproducible research are identified in Sect. 4 and shortly set into relation to PyPads. Sect. 5 gives an outlook of what PyPads

[1] https://github.com/padre-lab-eu/pypads.

aims to achieve and presents future plans for improving the framework.

2 Logging Requirements and Code-Injection

In a heterogeneous environment of available machine learning tools, packages and libraries, experiment tracking can be complex. This includes logging aggregated metrics, execution parameters and reproducibility artifacts. Due to the exploratory nature of machine learning experiments, settings and conditions have to be known to be able to interpret results and their origination process consistently. Therefore, extensive logging is needed in order to compare the impact of implementation details of libraries and custom code. This logging has to be applicable in a lot of environments, independent of dependencies of the experimental setup. Getting an overview of the process is important in science, application and learning supervision. Beginners as well as experts can profit from being able to reflect on their last steps.

Another factor for understandability is the scope and quality of logs. Generally, the more logs are available, the more one can argue about the reasoning in the process. To a lesser extent introducing logs with bad quality, i.e. hardly interpret-able interim values, can clutter the output and hinder the overview. The impact of this issue can be reduced with further steps like structural analysis over the logs themselves. Nevertheless, logging everything is simply not feasible. High quantities of data in practice force data scientists to work with aggregated measures. For example, a log of every weight change in a neural network can be impractical. Especially if there are many iterations of the training algorithm to be executed to find suitable hyperparameter settings. Meanwhile, time and effort must be managed conservatively due to a multitude of factors like funding limits and the pressure to publish. In summary, a hands-on approach for logging is required in an infrastructure allowing for verbosity configuration and easy extension, while minimizing the configuration effort.

While algorithms might not change between every version of machine learning libraries, small changes in the architecture might already impact the experiment outcome and automated logging capabilities. In general, a single scientific pipeline is based on a lot of such components and libraries, where a modification of a single part can have a great impact. This is especially critical in sensitive setups, which can already be impacted by small deviations in the computational process resulting in non-determinism[2] [22, 23] or random seeds [24]. Therefore, experimental setups can be quite fragile. In these cases flexible, but structured, logging needs to be implemented to understand why results are different in between runs. Logging must be possible over a wide range of different environments, while providing well-defined formats and metadata about the experiment and data itself. Properties for such an infrastructure include:

Adaptability Due to the high pace of progress in the field of machine learning and frequent changes in existing libraries, a logging infrastructure has to be able to cope with modifications of the environment and dependencies. Additional differing scopes and scales of experiments drive the need for customizable abstraction levels in logging itself. In large scale experiments, tracking every weight change of a neural network for example is not feasible, while in other settings one might want to consider it.

Extensibility To facilitate community driven logging and integrate newly developed algorithms, the infrastructure has to be easily extendable. In its base form, the infrastructure should be framework-agnostic as far as possible.

Non-intrusiveness In day to day work, the effort for logging must be reduced. This includes plumbing code needed for managing logs themselves and to the experiment itself interfacing log logic. In practice, ad-hoc experiments otherwise often only log aggregated metrics. Logging logic itself should be removed from the experiment as far as possible to modularize it and to further reusability. This follows the principles of aspect-oriented coding.

Robustness and Completeness The integration of logging must be robust on a set of issues. In general, logging something, even if incomplete, is mostly better than logging nothing. Nevertheless, defining tracking logic for every version of every available library is a considerable task, which can't be solved easily. While community driven logging may distribute the workload, as much of the existing setups need to be made log-able to provide comparability and value quickly. Generating value is essential to uphold the incentive for community-based development. In this context one must make do with available logging functionality even if not developed for the specific combination of dependencies, hardware, language version etc. This is error-prone and results in a fuzziness of logged data. Logs can not always be considered complete. The infrastructure additionally needs defensively programmed logging modules to not impact the experiment itself.

Quality Subsequently, log quality varies not only depending on which logging functionality was used, but also, on which setup it was used. To allow for veracity assessment, provenance must be ensured for every log. Furthermore, the log functionality itself should provide schemata of expected output to ease interpretation and facilitate error detection in the logs themselves.

A possibility to extend experiment code with the needed functionality, is to inject it into scripting languages on the fly. Python as a dynamically typed language with live in-

[2] https://pytorch.org/docs/stable/notes/randomness.html.

terpreted source code is predestined for such approaches. The MLflow[3] library (see Sect. 4) has shown that injection in general can be considered a powerful means of extracting meta information about experiments. To investigate the limits and identify potential alternatives to hard coded injections, we examined the evolution of the source code for one of the vastly used machine learning libraries, Scikit-learn[4].

As a representative of a core library in academics and practice, Scikit-learn provides a considerable number of 116 tagged versions on GitHub. Analyzing its 51 main releases on code changes shows that library development is to be considered non-linear. Old code might be added again or refactored naming rolled back. The overlap of classes, functions and variable assignments via full path can be seen in Fig. 2. One of such rollbacks is represented by the changes between versions 0.19.x and 0.20.x. Version 0.20.1 has more similarities to the 0.19.x family of the library than its chronologically closer related predecessor 0.20.0. A similar but less distinct effect can be observed between versions 0.13.x and version 0.14.1. Major versions tend to stay rather stable in their structure, resulting in a block pattern in the given heat-map. This allows for a robust live injection of logging functionalities, while changes indicate that the need for an update of the logging extension is more likely. Data like this enable the selection of candidates clusters for respective logging injection. Extending the analysis to include levenshtein distance between versioned files on aligned abstract syntax trees in a further step allow for tracking fine-grained changes in the potential clusters.

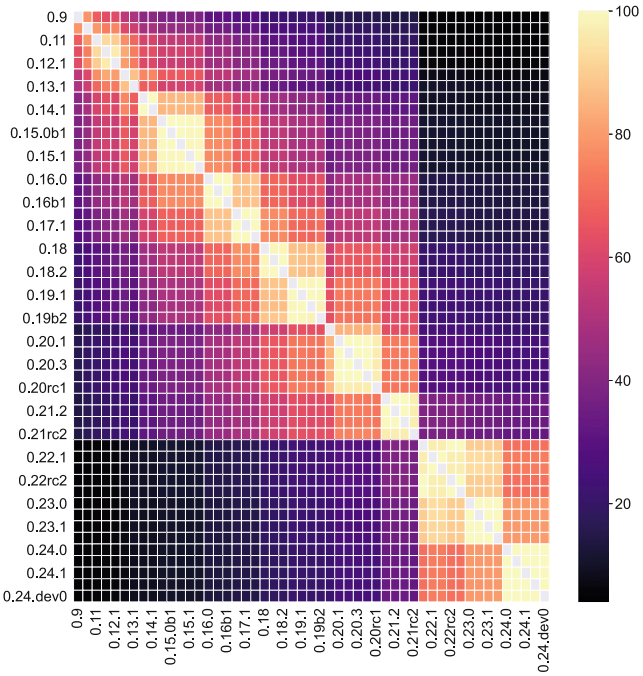

Fig. 2 Heatmap of Shared Function and Class references from version 0.9 to 0.24.dev0 of Scikit-learn

3 PyPads: A community-Driven logging infrastructure for ML

To overcome the challenges introduced by the evolution of software libraries and make use of the identified phases of code-stability, methods to inject logging code in a flexible non-intrusive way have to be found. Approaches like hard coded logging extensions result in a multitude of logging packages which must be chosen manually to suite the current setup and requirements. Investing too much time into logging is generally not accepted, therefore leading to sparsely documented conclusions. The need for different log levels, configurations and heterogeneous environments calls for a management of the logging itself. We developed a small library to provide a first iteration of such an infrastructure. PyPads' main feature provides users with automated, non-intrusive, community-driven, and semantically structured logging. The automated tracking is based on a wrapping mechanism/system that can also be referred to as a duck-punching system for providing hooks into the libraries used by the tracked experiment.

In dynamic programming languages like python, a common method for modifying or extending code is duck-punching, which is also known as monkey-patching [25]. The general concept does not restrict how underlying methods, classes attributes or functions are modified. They can add additional logic or in the case of dynamically typed languages even completely change up signatures. Patches are applied on memory only and don't have to be included in the source code. To secure the requirements qualities of Open Science, the configurations and logging infrastructure itself has to be made available in a structured manner. PyPads employs versioned mapping files to define, which functions need to be patched on the libraries. It introduces the tracking functionality using dynamically configurable loggers and injects the entry points for them when importing the libraries, omitting the need for manual activation. An abstract depiction of the architecture of PyPads can be found in Fig. 3.

When wrapping functions, PyPads uses a similar structure as the *alias_method_chain* or *prepend* of Rails[5], while preserving the original functionality of libraries. Extending the importlib and therefore modifying the outcome of the dynamic loading process of python, allows for a modular extension of user-imported objects.

[3] https://www.mlflow.org/docs/latest/tracking.html.

[4] https://pypi.org/project/scikit-learn/.

[5] https://littlelines.com/blog/2018/01/31/replace-alias-method-chain.

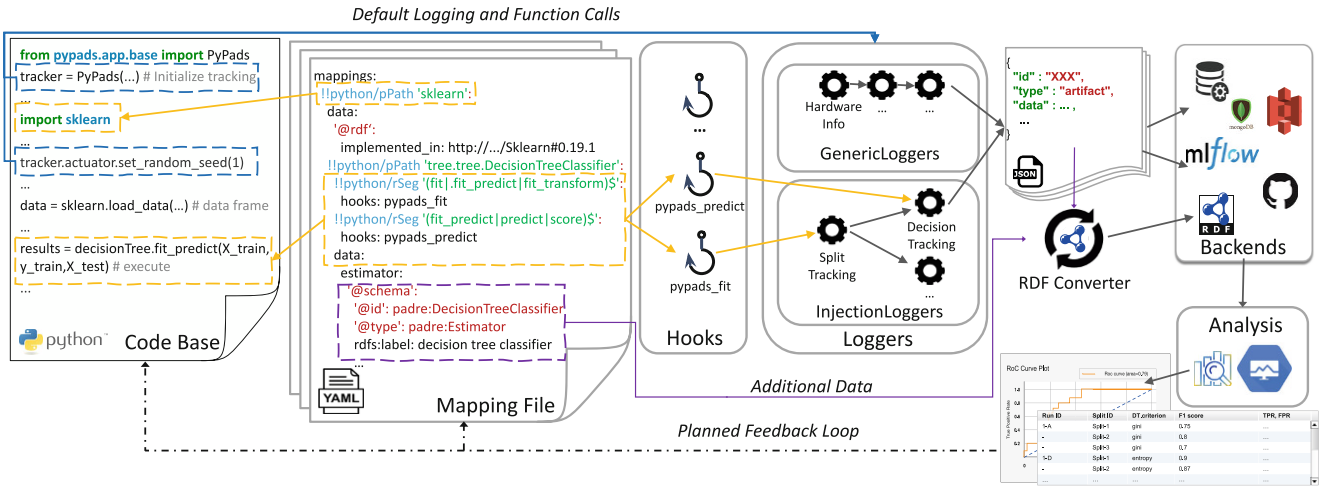

Fig. 3 Composition of PyPads [26]

Extending Python's Importlib PyPads's duck punching consists of modifying the import process of modules, classes, and functions by wrapping them if they are referenced in PyPads' **algorithm mappings**. Algorithm mappings are items in the *mapping registry*[6]. This is done by adding the above-mentioned custom import logic to the python **importlib** extension. Depending on the type of the imported object, PyPads takes the object, the context defining it and the matched mapping found in the mapping file and feeds it into the suitable wrapper to inject with logging functionalities.

3.1 Mapping Files

'YAML' based mapping files are a structured and human-readable medium for linking loggers with code (via hooks), including extra metadata and domain concepts.

Similar to classic XML mapping files in the software engineering domain, these files can be considered as a structural configuration including descriptive metadata [27]. The decoupled nature of mapping files makes them easily shared and extended-on by users, hence their community-driven aspect.

Static monkey-patching, like i.e. the MLflow auto-logging functionality, can become obsolete due to constant changes and updates in a library's code life cycle. Versioned mapping files circumvent this problem since it makes monkey-patching dynamic in the sense of specifying the path of module, class, or function to patch within a single adaptable file. Application of the duck punching itself can be decided on import time, taking environmental data as well as metadata of the mappings into account. A mapping entry represents the object that we wish to inject with logging functionalities. Each entry consists of three parts: a relative path to the object in the library, a set of hooks that maps to the corresponding loggers defined as events, and a data field for all the additional metadata and domain concepts. PyPads also allows for the use of regular expressions (or **Regex**) for the object's relative path, which offers a degree of flexibility to the mapping file that makes it resistant to minor and major re-factors in the source code of a library. Listing 4 shows an example of how a mapping file is structured. Each mapping file would contain a list of *mappings*.

In the current iteration of PyPads, the system attempts to execute loggers for every found valid mapping, regardless if the mapping file was directly compiled for the installed version of the target library. In case the logging process fails due to a version mismatch between the logger and library implementation, the system falls back to the original function call. In these cases, a log failure flag is included in the Logger-call objects, which are stored themselves for

```
mappings:
  !!python/pPath 'sklearn':
    !!python/pPath __init__:
      import_hooks: [ "pypads_import" ]
    data: "..."
    !!python/pPath 'base.BaseEstimator':
      !!python/pPath '__init__':
        hooks: ["pypads_init","pypads_estimator"]
      !!python/rSeg '(fit|.fit_predict)$':
        hooks: "pypads_fit"
      data: "..."
metadata:
  author: "Thomas Weissgerber"
  library:
    name: sklearn
    version: '>= 0.19.0'
  version: 0.1.0
```

Fig. 4 Shortened example of a minimal YAML mapping file for the Scikit-learn library

[6] A collection of hash maps created by compiling all the available mapping files with corresponding packages' names as keys.

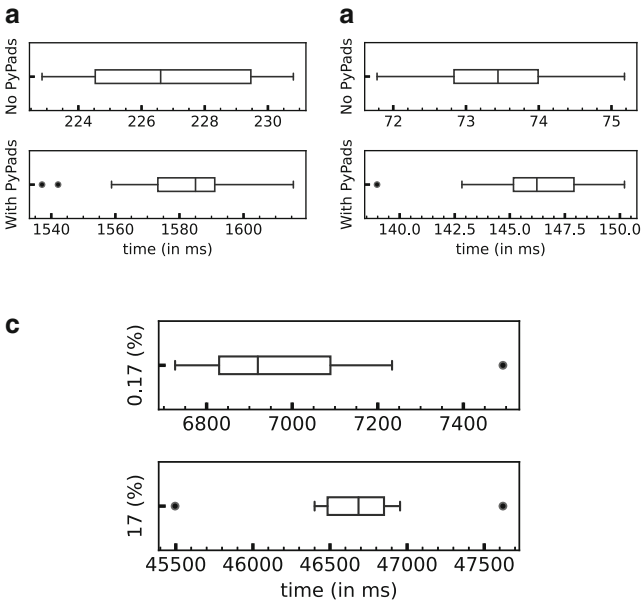

Fig. 5 Distribution of execution times of the *import* statement, metric function *f1-score*, and CoverTypes loading function *fetch_covtype* in sub-figures (a), (b), and (c) respectively. **a** On import of scikit-learn. **b** On f1-score execution. **c** On dataset loading (*fetch_covtype function*) for a size of 1000(0.17%) and 100 000(17%) samples

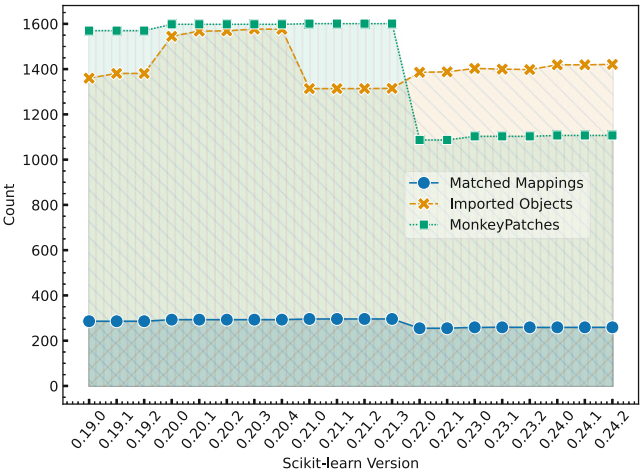

Fig. 6 Compatibility of a mapping file across 20 different releases of Scikit-learn

data provenance reasons. This allows for a more fuzzy, but traceable way of meta information storage.

3.1.1 Performance Impact

To show the performance impact introduced by PyPads both on import and on execution level of patched objects (or more specifically patched functions), we ran a minimal example script 10 times when we imported all modules of sklearn, load the CoverType dataset[7], fit a decision tree classifier and compute the f1-score on the predicted labels. An empty dummy *Logger* is defined to highlight the introduced constant overhead, it was used for duck-punching any matched mapping for the library metrics i.e, f1_score (with hook:"*pypads_metric*").

Additionally, loggers could also introduce an overhead that scales with the size of the experiment, i.e, the size of the dataset. Therefore, we defined a custom logger that trains a Random Forest Regressor and logs the feature importance weights when loading the data (with hook:"*pypads_dataset*").

Fig. 5a shows that the overhead in execution time when using PyPads' Importlib extension is around 1200 ms on average, which is not of much significance given the size of the mapping file with over 20k lines. The Importlib extension overhead mostly depends on the number and size of mapping files used during the lookup, as well as the number of imported objects. Fig. 5b highlights the overhead introduced by the monkey-patching and execution of our dummy logger for the metric function f1-score. There is no significant performance impact ($\leq$ 100 ms on average) introduced when executing the patched f1-score function. The other kind of introduced overhead can be seen in Fig. 5c, where our custom logger was executed when loading 0.17%, and 17% of the Covertype dataset (1000 and 100 000 samples respectively). Fig. 5c simply shows an example of the potential scaling of overhead in execution time when using loggers that deals with input data.

3.1.2 Compatibility and Log-Coverage

Similarly to the study in Sect. 2 where we looked into the changes in the source code across the life cycle of a library i.e, the Scikit-learn library, we wanted to show the compatibility of our current mapping file created for Scikit-learn (version 0.21.3) by comparing the number of matched mappings, monkey-patches, and imported objects when importing all modules. Fig. 6 shows the values for 20 different releases of the library from 0.19.0 to 0.24.2. It can be observed that matched mappings are stable across all versions, except for the transition from 0.21.3 to 0.22.0. The decrease in the number of matched mappings and duck-punched functions is also seen in the heatmap of Fig. 2. The overlap in shared functions, classes dropped drastically from above 90% to below 45%. This is most likely related to a major change in the naming scheme of classes and functions. Looking into it, the reason behind this change can be linked to the clear definition of private and public

[7] https://scikit-learn.org/stable/datasets/real_world.html#forest-covertypes.

APIs which was introduced in the 0.22 milestone[8] of the library's life cycle.

3.2 Output and Formats

Every logger defined in PyPads needs to adhere to a basic interface enforcing provenance and structure information. Loggers have to define JSON-Schema compatible output model for their produced logs, while also defining a set of meta-information about themselves. The content of the logs can furthermore be grouped in different tracked object classes and arbitrary metadata. In the logging context, loggers have access to the mapping files and their corresponding mappings relevant for the current execution. This information is summarized with run-time statistics and additional metadata provided by the mapping itself, as well as programmatic logic of the logger into the log. Listing 7 depicts the reduced output schema of a feature importance extraction logger.

The output of loggers is intended to be integrated into an ontology. For this purpose, the PyPads extension PyPads-Onto was developed. As base ontology for the conversion, ML-Schema [28] was chosen due to its easy structure and focus on machine learning structures. PyPads-Onto extends ML-Schema Ontology from its initial 21 Classes to 211 Classes including relations to SUMO and additional details like logger, libraries and their meta-information as objects themselves. Parts of the extensions to ML-Schema can be seen in Fig. 8.

To enable the mapping of output logs into ontologies, two main procedures can be conducted. Either logs have to be transcribed into RDF on the fly or, after their storage as JSON, via a cronjob. The second approach is to be considered less detailed than the first due to the converter not having access to the full context information anymore. Only the logged metadata itself can be used for the conversion, instead of analyzing the environment of the live system. To control and enrich available metadata mapping files can be used, adding information like RDF types on hooks. We expect this enriching information to represent most of a full-fledged mapping file, due to the compressed syntax of the mappings themselves. A current version of sklearn mapping consists of 95.89% semantic context mappings [26].

This mapping data can not only be used to embed the produced logs into the correct ontological environment but could in further steps also be used to supplement a mining-based mapping approach. The injection infrastructure may be able to analyze the executed code base on the fly to identify suitable mapping candidates.

Currently, PyPads finds application in a small scale setup, delivering computation containers as a service. Students and data scientist get access to computation containers in a Kubernetes[9] cluster via a JupyterHub[10] deployment. Logs are written with Kubernetes native logging systems to an Elasticsearch backend. The data flow diagram in Fig. 9 gives an insight into the communication process. Logs are gathered in the cluster via custom fluentd filters in sidecar containers. References to artifacts like pickled models are identified by a fluentd filter calling an upload script to store the binary data in corresponding s3 buckets. A trial run to transfer logs into the ontology was conducted. In further steps, it has to be revisited if the benefits of RDF storage out-weight the significant overhead registered on the live data.

```
{"title": "FeatureImportanceILFOutput",
"description": "This model represents the output
    of a singular
feature importance logger being executed on
    dataset import.
A logger might be able to produce multiple
    complex outputs.",
"type": "object",
"properties": {
  "abstraction_type": { "$ref": "#/definitions/
      AbstractionType"},
  "created_at": {"title": "Created At", "type": "
      number"},
  "experiment": {"$ref": "#/definitions/
      ExperimentReference"},
  "category": {"title": "Category", "default": "
      LoggerOutput",
      "type": "string"},
  "produced_by": {"$ref": "#/definitions/
      IdReference"},
  ...
  "tags": {"title": "Tags", "default": [], "type
      ": "array",
    "items": { "$ref": "#/definitions/
        IdReference"}
  },
  "tracked_objects": {"title": "Tracked Objects",
      "default": [],
      ...},
  "feature_importance": {"$ref": "#/definitions/
      IdReference"}},
...}
```

Fig. 7 Logger output schema without require and definitions

4 Related Work

Due to the increased interest on structuring ML research and ensuring its openness, a set of frameworks were developed and released. The goal was to provide necessary features to secure an open, shareable and reproducible environment for ML research experiments. In the following section, a selection of tools that focuses on managing, tracking and deploying re-usable machine learning workflows are described.

[8] https://github.com/scikit-learn/scikit-learn/issues/9250.

[9] https://kubernetes.io.

[10] https://jupyter.org/hub.

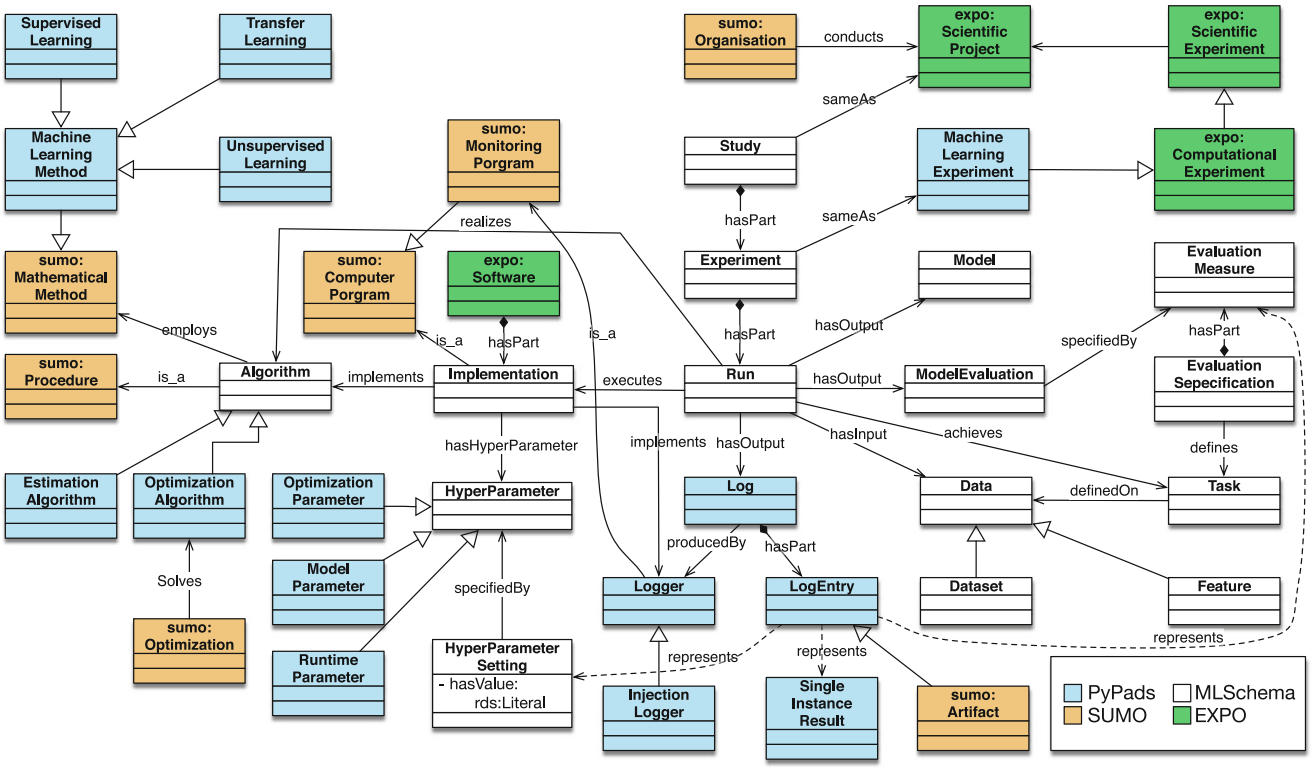

Fig. 8 Excerpt of the in PyPads-Onto used 221 Classes. Blue is the PyPads namespace, orange SUMO, green Expo and grey ML-Schema

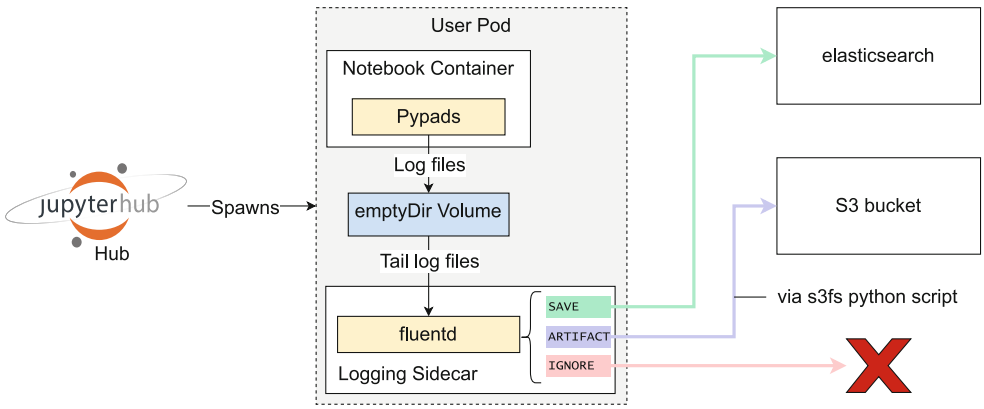

Fig. 9 Dataflow of pypads in the kubernetes cluster

MLflow [29] is a well-known platform for the machine learning life cycle by Databricks. The open source platform aims to manage the end-to-end machine learning life cycle. Its main function is to track experiments for recording and comparing results, packaging the machine learning code in a reusable and reproducible form and managing deployable models from a variety of model serving and inference platforms. MLflow's key principle is an open interface design, where data scientists and engineers can bring their own training code, metrics, and inference logic while benefiting from a structured development process [30]. However, since MLflow does not enforce **standardized structure** on the written logs, it becomes hard to understand and make use of an experiment's logs in some cases. Logging with MLflow is mainly done manually by adding additional lines into the existing code base, which could present an unnecessary overhead for researchers. While their auto-logging feature aims to solve this problem, it is based on **static monkey-patching** and will require constant updates in case of huge revisions and reworks in the source code of the target library.

Sacred [31] is an open source project which allows for annotation and tracking of experiments. Sacred facilitates easy configuration of experiments and uses annotations as a way to offer non-intrusive logging, from which PyPads takes inspiration to remedy the limitations presented

by MLflow. Similarly, Weights and Biases [32] provides experiment tracking and dataset versioning via manual code integration. PyPads improves on MLflow by enforcing a schema model for each produced logs to guarantee a defined structure for every logger. This helps standardize logs and makes them easily shareable among scientists. It also extends upon MLflow's auto-logging feature by using mapping files to dynamically match and inject target functions with the corresponding loggers based on defined hooks. Hence, the introduction of dynamic monkey-patching. The latter reduces efforts in manual code configuration and annotation required by Sacred [31] and Weights and Biases [32].

Reusable Analysis(REANA) [33], Data Version Control(DVC) [34], OpenML [35], Sumatra [36], and Kubeflow [37] to add a few more to a non-exhaustive list of tools and frameworks that were developed to help with the tracking of the research workflow by providing ways to ease logging for developers.

5 Conclusion

The paper presents the workings of a prototypical logging framework for machine learning, making use of dynamic code injection, community driven mappings and cluster native logging applications. With an ontology and resource description framework export, we hope to facilitate semantic harmonization between libraries and further the understandability of experiment outcomes. The paper presets key concepts and features of PyPads, while shining the light on needed extensions, i.e, PyPads-Onto. The modular nature of the stack and the code-less tracking capabilities aim to facilitate an ecosystem sustained by community efforts. Existing Loggers support the automated bookkeeping of the used hardware, parameter set, metrics and setups of mapped libraries. To show the capabilities for not only providing logging but also supporting the development of experiments, PyPads implements a simple TensorFlow determinism warning validator.

On top of existing state-of-the-art tools, PyPads introduces a novel approach of non-intrusive logging using mapping files that ensure the flexibility and extensibility of our autologging functionality and offer a community-driven support for the evolving ML algorithms and frameworks. Moreover, PyPads provides end-to-end provenance history of every log by enforcing a JSON schema model on the producing logger, which also defines metadata about the logger itself that can be packaged and referenced with a versioning system i.e, Git. The last main contribution of PyPads consists of extending ML-Schema Ontology with logs enriched with a semantic context that can be added via the aforementioned mapping files.

Future versions of PyPads aim to incorporate additional tools to secure replicability of experimental runs. One such venture could be built on the logger employing *strace* to track process calls similarly to ReproZip [38], CARE [39], PTU [40], Sumatra [36], and CDE [41] or by directly using these tools. These environment capturing tools "solve" replicability by persisting all libraries and data files accessed in a run on the operating system layer. A current implementation for reproducibility semi-automatically compiles the experiment code into a docker image. While PyPads can currently use the hooks to define a simple call graph, more sophisticated approaches could be introduced by extending the PyPads stack with tools such as ProvenanceCurious [42] or noWorkflow [43], which apply tracing routines and code inspection to include additional data flow information on source code basis. A taxonomic overview of tracing tools [44] may in the future allow for a structured tracing with differing granularity depending on the use case.

Funding This work has been partially funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy by means of the funding program "Internetkompetenzzentrum Ostbayern" as well as by the German Federal Ministry of Education and Research in the project "Provenance Analytics" with grant agreement number 03PSIPT5C.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Pawlik M, Hütter T, Kocher D, Mann W, Augsten N (2019) Datenbank Spektrum 19(2):107. https://doi.org/10.1007/s13222-019-00317-8
2. Risch J, Krestel R (2019) Datenbank Spektrum 19(2):117. https://doi.org/10.1007/s13222-019-00316-9
3. Kurakin A, Goodfellow I, Bengio S et al (2016) Adversarial examples in the physical world
4. Schwab M, Karrenbach N, Claerbout J (2000) Comput Sci Eng 2(6):61
5. Vanschoren J, Blockeel H, Pfahringer B, Holmes G (2012) Mach Learn 87(2):127
6. Sonnenburg S, Braun ML, Ong CS, Bengio S, Bottou L, Holmes G, LeCun Y, MÃžller KR, Pereira F, Rasmussen CE et al (2007) J Mach Learn Res 8(Oct):2443
7. Baker M (2016) Nat News 533(7604):452

8. Olorisade BK, Brereton P, Andras P (2017) ICML 2017 RML Workshop: Reproducibility in Machine Learning
9. Begley CG (2013) Nature 497(7450):433
10. Hutson M (2018) Artificial intelligence faces reproducibility crisis
11. Wendlinger L, Stier J, Granitzer M (2021) Genetic Programming. EuroGP 2021. In: Hu T, Lourenço N, Medvet E (eds) Genetic Programming. EuroGP 2021. Lecture Notes in Computer Science, Springer, Cham, vol 12691, pp 162–178
12. Stodden V (2009) Ann Intern Med. https://doi.org/10.1109/MCSE.2009.19
13. Gentleman R (2007) D. Temple Lang. J Comput Graph Stat. https://doi.org/10.1198/106186007X178663
14. Homolak J, Kodvanj I, Virag D (2020) Scientometrics 124(3):2687. https://doi.org/10.1007/s11192-020-03587-2
15. Fecher B, Friesike S (2014) Opening science, pp 17–47
16. Nosek BA, Alter G, Banks GC, Borsboom D, Bowman SD, Breckler SJ, Buck S, Chambers CD, Chin G, Christensen G, Contestabile M, Dafoe A, Eich E, Freese J, Glennerster R, Goroff D, Green DP, Hesse B, Humphreys M, Ishiyama J, Karlan D, Kraut A, Lupia A, Mabry P, Madon T, Malhotra N, Mayo-Wilson E, McNutt M, Miguel E, Paluck EL, Simonsohn U, Soderberg C, Spellman BA, Turitto J, VandenBos G, Vazire S, Wagenmakers EJ, Wilson R, Yarkoni T (2015) https://science.sciencemag.org/content/348/6242/1422. Science 348(6242):1422. https://doi.org/10.1126/science.aab2374
17. McNutt M (2014) Science 343(6168):229. https://doi.org/10.1126/science.1250475 (https://science.sciencemag.org/content/343/6168/229)
18. Braun ML, Ong CS (2018) Implementing reproducible research. Chapman and Hall/CRC, pp 343–365
19. Marwick B, d'Alpoim Guedes J, Barton CM, Bates LA, Baxter M, Bevan A, Bollwerk EA, Bocinsky RK, Brughmans T, Carter AK et al (2017) SAA archaeological. Record 17(4):8
20. Dienlin T, Johannes N, Bowman ND, Masur PK, Engesser S, Kümpel AS, Lukito J, Bier LM, Zhang R, Johnson BK, Huskey R, Schneider FM, Breuer J, Parry DA, Vermeulen I, Fisher JT, Banks J, Weber R, Ellis DA, Smits T, Ivory JD, Trepte S, McEwan B, Rinke EM, Neubaum G, Winter S, Carpenter CJ, Krämer N, Utz S, Unkel J, Wang X, Davidson BI, Kim N, Won AS, Domahidi E, Lewis NA, de Vreese C (2020) J Commun 71(1):1. https://doi.org/10.1093/joc/jqz052
21. Weißgerber T, Fellicious C, Granitzer M (2019) PADRE: Platform for mAchine learning and Data science REproducibility. Open Sci Process Model Mach Learn (OSPMML). https://doi.org/10.5281/zenodo.4870627
22. Heumos L, Ehmele P, Menden K, Cuellar LK, Miller E, Lemke S, Gabernet G, Nahnsen S (2021) CoRR abs/2104.07651. https://arxiv.org/abs/2104.07651. Accessed 19.4.2021
23. Nagarajan P, Warnell G, Stone P (2018) CoRR abs/1809.05676. http://arxiv.org/abs/1809.05676. Accessed 5.10.2018
24. Fellicious C, Weißgerber T, Granitzer M (2020) LOD
25. Hunt J (2019) A beginners guide to python 3 programming. Springer, pp 325–336
26. Weißgerber T, BenAmor M, Fellicious C, Granitzer M (2021) Pypads: bootstrapping community-driven open. Science for Machine Learning https://doi.org/10.5281/zenodo.4697245
27. Keith M, Schincariol M, Nardone M (2018) XML Mapping Files. Apress, Berkeley, pp 593–654 https://doi.org/10.1007/978-1-4842-3420-4_13
28. Publio GC, Esteves D, Lawrynowicz A, Panov P, Soldatova LN, Soru T, Vanschoren J, Zafar H (2018) CoRR abs/1807.05351. http://arxiv.org/abs/1807.05351. Accessed 2018-8-13
29. MLFlow. Mlflow – a platform for the machine learning lifecycle | mlflow. https://mlflow.org/. Accessed 2019-12-17
30. Zaharia M, Chen A, Davidson A, Ghodsi A, Hong SA, Konwinski A, Murching S, Nykodym T, Ogilvie P, Parkhe M et al (2018) IEEE Data Eng Bull 41(4):39
31. Greff K (2015) J. Schmidhuber, proceedings of the AutoML. International machine learning society
32. W&B. Weigths & biases. https://docs.wandb.ai/. Accessed 2021-08-02
33. Šimko T, Heinrich L, Hirvonsalo H, Kousidis D, Rodríguez D (2019) EPJ web of conferences. EDP Sci 214:6034
34. dvc. Data version control · dvc. https://dvc.org/. Accessed 2020-03-16
35. Vanschoren J, Van Rijn JN, Bischl B, Torgo L (2014) ACM SIGKDD. Explor Newsl 15(2):49
36. Davison AP, Mattioni M, Samarkanov D, Teleńczuk B (2018) Sumatra: A Toolkit for Reproducible Research. Implementing Reproducible Research [Internet] 2018:57–78. https://doi.org/10.1201/9781315373461-3
37. Google. Kubeflow|kubeflow. https://www.kubeflow.org/docs/about/kubeflow/. Accessed 2019-11-16
38. Chirigati F, Rampin R, Shasha D, Freire J (2016) *SIGMOD 2016 - proceedings of the 2016 international conference on management of data* (association for computing machinery, 2016), proceedings of the ACM SIGMOD international conference on management of data. In: ACM SIGMOD International Conference on Management of Data, SIGMOD 2016, pp 2085–2088 https://doi.org/10.1145/2882903.2899401
39. Janin Y, Vincent C, Duraffort R (2014) Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering. Association for Computing Machinery, New York, NY, USA https://doi.org/10.1145/2618137.2618138
40. Pham Q, Malik T, Foster I. 2013. Using provenance for repeatability. In Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP '13). USENIX Association, USA, Article 2, 1–4.
41. Guo P (2012) Comput Sci Eng 14(4):32
42. Huq M, Apers P, Wombacher A (2013) in *proceedings of the 16th international conference on extending database technology*. EDBT, vol 2013. Association for Computing Machinery (ACM), United States, pp 765–768 https://doi.org/10.1145/2452376.2452475
43. Murta L, Braganholo V, Chirigati F, Koop D, Freire J (2015) Provenance and annotation of data and processes. In: Ludäscher B, Plale B (eds) Provenance and Annotation of Data and Processes, Springer, Cham, pp 71–83
44. Pimentel JF, Freire J, Murta L, Braganholo V (2019) ACM Comput Surv. https://doi.org/10.1145/3311955