Analysing Python Machine Learning Notebooks with Moose

Marius Mignard[1], Steven Costiou[1], Nicolas Anquetil[1] and Anne Etien[1]

[1]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

Machine Learning (ML) code, particularly within notebooks, often exhibits lower quality compared to traditional software. Bad practices arise at three distinct levels: general Python coding conventions, the organizational structure of the notebook itself, and ML-specific aspects such as reproducibility and correct API usage. However, existing analysis tools typically focus on only one of these levels and struggle to capture ML-specific semantics, limiting their ability to detect issues. This paper introduces Vespucci Linter, a static analysis tool with multi-level capabilities, built on Moose and designed to address this challenge. Leveraging a metamodeling approach that unifies the notebook's structural elements with Python code entities, our linter enables a more contextualized analysis to identify issues across all three levels. We implemented 22 linting rules derived from the literature and applied our tool to a corpus of 5,000 notebooks from the Kaggle platform. The results reveal violations at all levels, validating the relevance of our multi-level approach and demonstrating Vespucci Linter's potential to improve the quality and reliability of ML development in notebook environments.

Keywords

Software Analysis, Software Quality, Linter, Machine Learning Code

1. Introduction

Despite the exponential growth of software development in Machine Learning (ML) and related fields, such as data science, recent studies indicate that ML code often shows lower quality than traditional software [1, 2]. One reason is that many ML developers come from non-software backgrounds, making them unfamiliar with established best practices in Software Engineering (SE) or prone to adopting alternative coding conventions [1]. Additionally, much of ML development follows the literate computing paradigm [3, 4], which combines text, code, and outputs—most often in notebooks such as *Jupyter Notebook*[1] [5]. While notebooks facilitate rapid prototyping, ease of documentation, and sharing, they often lack rigorous enforcement of coding standards due to limited integration with code quality tools such as linters or code-smell detectors.

Previous studies have worked on formalizing coding rules, smells, and bad practices in both notebooks and ML applications [6, 7, 8]. These practices can be divided into three levels. The first is the Python level, which involves following PEP8-style[2] rules, creating well-structured functions, and writing tests. The second is the notebook level, such as including *Markdown* cells to explain the code, placing imports at the beginning, and recording library versions. Finally, there are ML-specific rules, like setting random seeds for reproducibility and using recommended methods with the correct parameters.

Single-level linters work well for conventional Python projects, but ML notebooks introduce challenges that current tools struggle with: fragmented context across cells and ML-specific semantics that go beyond style or AST-only checks and often span multiple cells. Vespucci Linter addresses these challenges by unifying notebook structure and Python code entities into a single model, so rules can be expressed at the Python, notebook, and ML levels within one analysis. While most current rules are single-level, the unified model already supports cross-level reasoning.

The notebook quality issue is critical as notebooks increasingly serve as vehicles for sharing analytical insights, prototypes, and educational materials [9, 10], often targeting audiences with limited programming expertise (and prone to reuse code [11]). Without tools capable of enforcing good practices across

[1]https://jupyter.org/

[2]https://peps.python.org/pep-0008/

all three rule levels, poor coding habits are propagated, potentially resulting in production-level systems built on suboptimal ML models [12]. This can create a negative feedback loop, perpetuating poor quality code and models[3] through successive generations of learners and developers.

Although various analysis tools are available, including some dedicated to data science (data validation or correct use of data-related functions) [13], only one tool has recently started to incorporate aspects of the machine learning *workflow*. This highlights the need for specialized analysis tools tailored to notebook-based ML development, where code is fragmented across multiple cells interleaved with text and outputs. Furthermore, such tools must incorporate ML-specific semantic analysis, addressing issues like appropriate method calls, correct parameter usage, and proper sequencing of operations. Existing tools typically focus either on general Python coding conventions, notebook-level structural recommendations, or data-level validations, often neglecting the deeper semantic analysis related to the machine learning workflow itself [10, 1, 13].

Detecting issues at this ML-specific level poses significant challenges, as it necessitates extracting and interpreting semantic information from Python code to accurately identify the intended logic of ML workflows. Misuse or misconfiguration at this level can significantly alter the meaning and validity of the results, thus requiring sophisticated domain-specific analysis.

In response to these challenges, this paper introduces *Vespucci linter*[4], a tool built upon the *Moose*[5] software analysis platform [14]. Our tool leverages Moose's modeling capabilities to detect bad practices in ML notebooks across all three defined levels. *Vespucci linter* aims to promote best practices within the literate computing paradigm.

This paper is structured as follows. Section 2 reviews related work. Section 3 describes our tool and methodological approach. Section 4 illustrates the application of *Vespucci linter* through a real-world dataset sourced from the Kaggle[6] platform. Finally, Section 5 concludes the paper and outlines potential directions for future research.

2. Related Work

2.1. Python, Notebook And Machine Learning Code Smells

Code smells are recognized as bad practices, poor design choices or anti-patterns that negatively impact software quality, readability, and maintainability. Unlike bugs, code smells do not necessarily cause direct faults but can lead to other negative consequences [15]. Empirical research consistently highlights the negative impacts of code smells, linking them to reduced code quality, higher defect rates, and increased long-term maintenance challenges [16, 17]. Several studies have identified and formalized code smells across the three levels mentioned in the introduction (Python, Notebook, and ML).

At the Python level, many guidelines and best practices addressing code smells have already been formalized. A well-known example is the widely adopted linter *PyLint*, which enforces a set of coding conventions derived from established SE standards. Typical rules include naming conventions, proper structuring of modules and functions, PEP8 style, and the identification of unused variables [18].

Previous works have analyzed Python-based ML code using *PyLint* [19, 2], and identified that the most common rule violations are related to imports, excessive numbers of function arguments or local variables, pointless statements, and unused variables.

At the notebook level, studies have proposed specific recommendations to improve notebook usage. For instance, Quaranta et al. identified 17 best practices through a systematic literature review [20], emphasizing practices such as placing imports at the beginning, using clear notebook names, and including Markdown cells for documentation. Similarly, Rule et al. outlined 10 rules to effectively share

[3]It is important to note that here, we are referring to ML models that enable prediction. In the remainder of the text, the term *model* refers to its software engineering meaning—an abstract representation of a system conforming to a metamodel.
[4]https://github.com/JMLF/FamixNotebook
[5]https://modularmoose.org/
[6]https://www.kaggle.com/

computational analyses in notebooks, highlighting the importance of splitting cells logically, explicitly recording dependencies, and providing thorough documentation [21].

At the ML-specific level, Nikanjam et al. presented a catalog of 8 design smells, identified through a review of related literature and an analysis of 659 deep learning programs. These smells highlight poor architectural or configuration decisions that negatively impact model performance and maintainability [22]. Complementarily, Zhang et al. collected 22 ML-specific code smells from academic literature, online forums, and open-source repositories, and proposed corresponding solutions. Their work underlines recurrent issues in ML application development, such as uncontrolled randomness and improper hyperparameter configurations [8].

However, existing research mainly addresses code smells independently at one level—either Python, notebook, or ML-specific—without considering how these issues interact or compound across multiple levels. Thus, a significant gap remains in understanding the cumulative effects of code smells when combined across different abstraction levels, how smells identified at one level can propagate and impact others and how to globally address them.

2.2. Existing Quality Control Tools

Quality control tools currently available primarily cover the Python and notebook-specific levels, along with data validation in data science, but lack dedicated analysis capabilities on ML workflows.

At the Python level, widely adopted tools include *PyLint*[7], *PyFlakes*[8], and *Ruff*[9]. These tools typically operate by parsing Python code into an Abstract Syntax Tree (AST) and analyzing it against predefined rules. They primarily address general coding issues.

On the notebook level, specialized tools have been proposed. *Pynblint*[10], developed by Quaranta et al., implements best practices specifically identified for notebooks. It analyzes notebooks based on criteria like proper cell structure, correct usage of Markdown for documentation, and dependency management, providing feedback for improving notebook quality [23]. Similarly, *NBLyzer*[11], proposed by Subotić et al., incorporates the unique cell execution semantics—the possibility of out-of-order cell execution—of notebooks into its analysis. It provides warnings and recommendations on cell ordering, dependencies, and potential data leakage due to improper execution order [24].

Specialized tools also exist for data validation and correct data handling in data science workflows, such as *Pandera*[12] and the prototype linter proposed by Dolcetti et al. These tools specifically target data-related logic errors, ensuring the correctness of data manipulation and transformation steps [13].

In addition, a dedicated tool named *ML Lint* was proposed by van Oort et al. [25]. It introduces the novel concept of *project smells* aiming for a more holistic analysis of ML projects. *ML Lint* performs static analysis on Python ML projects, reviewing source code, data, and configurations. Its linting rules cover five major categories: version control (code and data), dependency management, continuous integration, code quality, and testing practices. *ML Lint* checks for the appropriate use of tools like Git and Data Version Control (DVC), proper separation of development and runtime dependencies, presence of tests, and produces reports highlighting detected project smells. For code issues, *ML Lint* relies entirely on existing Python linters and does not define or enforce any notebook-specific or ML–oriented rules.

Another relevant tool is *SonarQube*[13], a static analysis platform originally developed for general-purpose software quality assurance. Recently, it has started to incorporate rules and practices related to the ML-specific level. However, these analyses remain primarily focused on traditional software engineering aspects and do not fully extend to the structure or semantics of ML workflows. Moreover,

[7]https://www.pylint.org/
[8]https://github.com/PyCQA/pyflakes
[9]https://docs.astral.sh/ruff/
[10]https://github.com/collab-uniba/pynblint
[11]https://github.com/microsoft/NBLyzer
[12]https://pandera.readthedocs.io/
[13]https://www.sonarsource.com/products/sonarqube/

defining additional, domain-specific rules is not straightforward, and the tool does not support analysis at the notebook level.

Existing tools remain mostly focused on single-level analyses, addressing either general Python coding conventions, notebook structure, or data validation. While some tools, such as *SonarQube* have begun to incorporate elements related to ML, they do not yet fully address the ML workflows-level. This gap is problematic, as ML workflows involve domain-specific patterns that are not captured by existing tools. There is a need for solutions that extend quality analysis to include the ML workflows-level alongside existing checks.

3. Vespucci Linter

We designed Vespucci to detect best-practice violations at the *Python*, *notebook*, and *machine-learning* levels within a single analysis tool. It is based on top of the *Moose* analysis platform, which offers a flexible meta-platform centered around the *Famix* meta-modeling framework. *Famix* provides basic modeling elements that can be combined to define custom metamodels [14]. Leveraging this flexibility allows us to define and interconnect metamodels: one for Python source code and another for notebooks. Additionally, *Moose* gives us a dedicated model query language and the *Critics*[14] rule engine, enabling a rapid implementation of new quality rules with just a few lines of Pharo code. It also allows users to benefit from the broader Moose ecosystem — including existing filtering and grouping functionalities, and visualization tools. Detected violations can be tagged at the level of a cell, function, or any model entity, and then explored visually to reveal structural hotspots.

3.1. Notebook Meta-model

Existing linters typically operate on Python's Abstract Syntax Trees (AST). Relying solely on AST requires transforming notebooks into plain Python scripts, discarding positional context information specific to notebook structures, such as cell boundaries. Consequently, these linters lose the granularity required for precise, notebook-specific quality analysis. Similarly, when analyzing notebooks from their JSON structure on a per-cell basis, dependencies across cells may be lost, as individual cells might refer to definitions located in preceding ones that are not part of the current analysis scope.

To address these limitations, we propose a *metamodel* (Figure 1) to combine notebook structural information with Python elements provided by *FamixPython*. Both Figure 1 and the accompanying explanation present an abstracted view, focusing on the key entities and relationships relevant to our analysis, while deliberately omitting less central elements present in the full *FamixPython* metamodel.

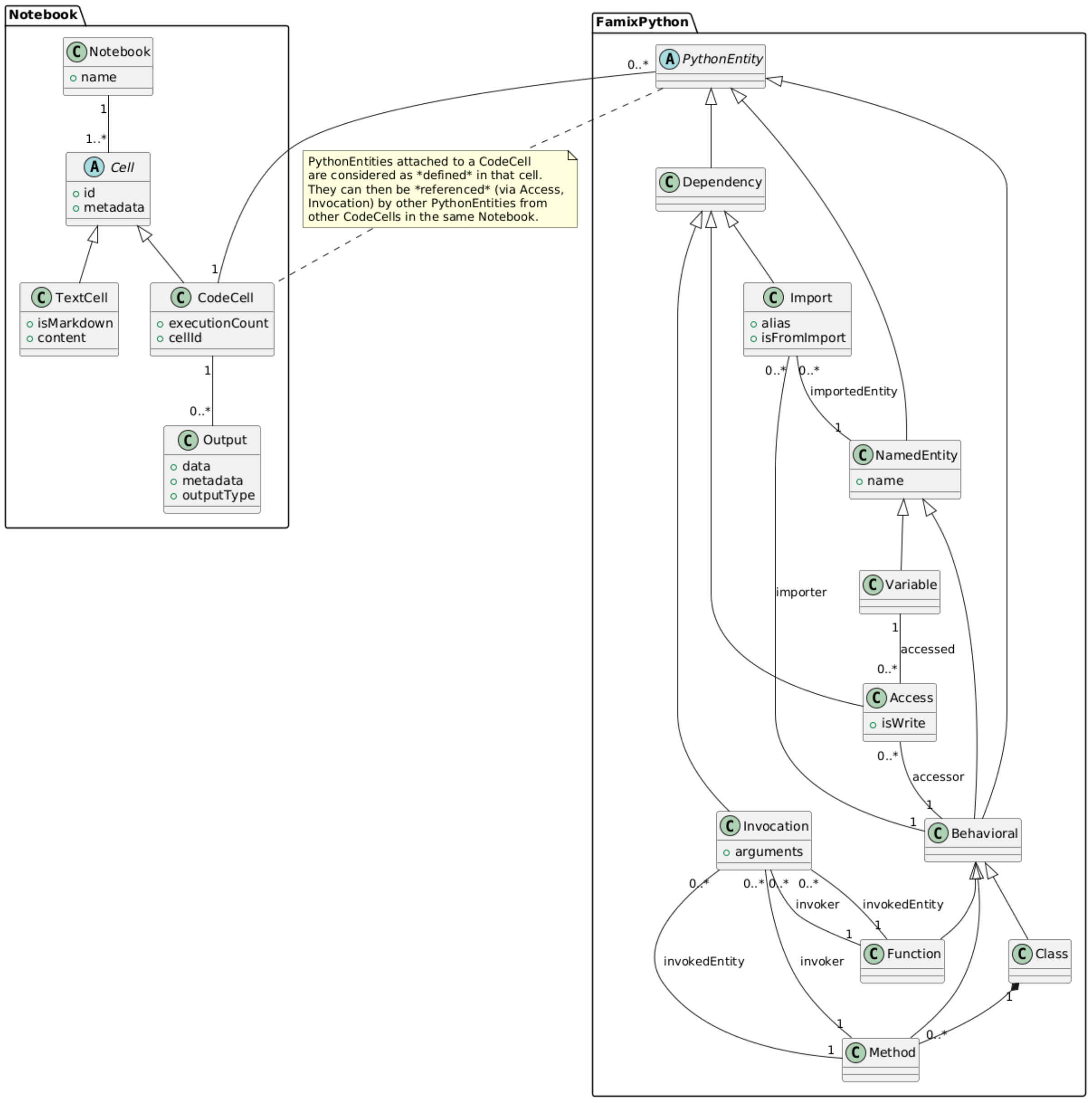
Our metamodel preserves the mapping between Python entities and notebook cells, ensuring that no contextual information is lost. In our metamodels, a *Notebook* consists of multiple *Cells*, categorized into *CodeCells* and *TextCells*. *CodeCells* hold executable Python code, whereas *TextCells* typically provide markdown documentation. Each *CodeCell* can yield multiple *Outputs*, capturing results from execution, visual outputs, and textual outputs, preserving the full execution context.

The Python aspect of our metamodel leverages the *FamixPython* entities. Each *CodeCell* is directly linked to associated Python entities, including variables, functions, classes, imports, and invocations. This linkage ensures a detailed mapping, enabling queries to precisely trace Python entities back to the cells where they were defined. Our Python metamodel includes entities containing executable instructions, such as *Behavioral* entities (functions, methods, and classes), and *Association* relationships linking entities. These relationships include *Access* entities capturing read/write operations on variables, *Invocation* entities connecting the invoker to the invoked entity and *Import*.

This granularity does not stop at cell boundaries but instead provides a notebook-wide view, allowing analyses to span multiple cells, accurately track dependencies, and identify potentially problematic patterns across the entire notebook. By combining these two metamodels, our approach offers rich

[14]https://modularmoose.org/users/moose-ide/moose-critics/

querying capabilities such as identifying Python imports that do not appear in the first *CodeCell*, or detecting potentially problematic invocation sequences across different notebook cells.



PythonEntities attached to a CodeCell are considered as *defined* in that cell. They can then be *referenced* (via Access, Invocation) by other PythonEntities from other CodeCells in the same Notebook.

Figure 1: Simplified representation of our notebook meta-model.

3.2. Process Overview

The implementation of *Vespucci Linter* follows a four-step pipeline (summarised in Figure 2) that transforms raw Jupyter notebooks into analyzable models which enable quality analysis without requiring prior code extraction or manual preprocessing. It applies a set of predefined linting rules and reports any detected violations. The following steps describe our approach, from notebook parsing to rule evaluation and result export

1. **Notebook parsing.** The notebook (`*.ipynb`) file is parsed; cells are extracted while metadata (execution counter, id, type) are preserved. We then append all Python code from code cell in a temporary file.

2. **Notebook modelling.** A *model* is automatically built that merges the *FamixPython* model (from the parse of the previously built temporary file)—capturing imports, functions, invocations…—with a custom *Notebook* model—representing *Markdown* and code cells, execution order, and raw metadata.
3. **Lint-rule application.** On the generated *model* we evaluate all defined rules. Rules are detailed in Section 3.3. Each rule returns a *Violation* object with severity, precise location (file/cell), and a suggestion for remediation.
4. **Result export.** All violations are serialized to JSON for analysis or manual inspection.

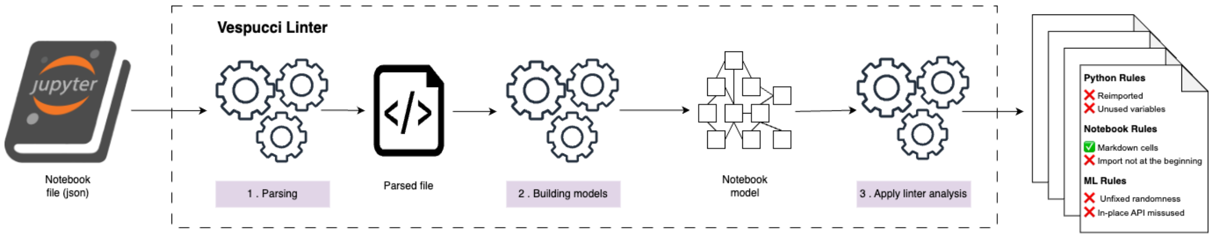


Figure 2: Overview of our process to detect rule violations in notebooks.

3.3. Proposed Multi-level Rules

We selected 22 specific linting rules tailored to address frequent bad practices observed in Python notebooks used for machine learning. These rules originate from the literature and are organized into general Python practices, notebook-specific guidelines, and ML-specific usage patterns. The thresholds and conventions for these rules are extracted from literature but can easily be adapted to specific contexts and libraries. For clarity, we assign an identifier to each rule, consisting of a letter indicating its level—*P* for Python-level, *N* for notebook-level, and *M* for ML-specific—followed by a number. For example, the first Python-level rule is labeled *P1*.

3.3.1. General Python Rules

Due to its wide usage and acceptance within the Python community, we base our Python-level rules primarily on those defined and validated by *PyLint*. However, we exclude style-related issues — which do not affect code correctness — as well as runtime errors such as undefined variables, which would prevent the code from executing properly regardless of any linting effort. Instead, we focus on code issues categorized as warnings and refactoring opportunities, as they are more relevant to maintainability and can help uncover potential logical flaws or unintended behaviors.

We selected some of the most frequently violated rules observed in machine learning notebooks [19, 2], while excluding those that are not relevant in this context. For example, rules like `pointless-statement`—which detect statements with no effect—are not appropriate for notebooks, where placing a variable or expression at the end of a cell to display its output is a common and intentional practice. The selected rules and their descriptions are presented in Table 1. Threshold limits used for each rule match those defined by *Pylint*, but they can be easily adjusted according to specific requirements.

3.3.2. Notebook-specific Rules

The following linting rules have been defined specifically for notebook-level quality, sourced from literature [20, 21]. These rules address common issues in code organization, naming, and execution behavior that affect readability, reproducibility, and maintainability. Naming conventions and threshold limits used for each rule match those defined by *Pynblint*, and can be easily modified if needed. Unlike

Table 1
General Python Linting Rules Implemented in Vespucci

Rule Name	Description	Motivation
P1 - Unused variables	Identifies variables that are assigned but never used.	Helps remove dead code and clarify the intent.
P2 - Variables reassignments	Detects repeated use of the same variable name after initial assignment.	Prevents confusion and potential bugs in notebooks.
P3 - Variables naming	Enforces a minimum variable name length (more than 3 characters, except for ML conventions like `X`, `y`).	Promotes readability and avoids ambiguous identifiers.
P4 - Too many parameters	Detects functions with more than 5 parameters.	Encourages simpler, modular function design.
P5 - Consider using from import	Recommends using `from module import x` instead of importing the full module.	Improves clarity and avoid namespace pollution
P6 - Unused import	Identifies unused imports with an exception for the star import (`from module import *`) where we can not statically determine which symbols are actually used in the code.	Improves clarity.
P7 - Reimported	Identifies redundant imports of the same module.	Reduces code clutter and redundancy.
P8 - Too many local	Detects functions that declare more than 11 local variables.	Indicates high complexity.
P9 - Global variables in function	Warns against the use of global variables within functions.	Prevents hidden dependencies and side effects.

Pynblint, which includes checks requiring external context (e.g., presence of git repositories), we focus exclusively on rules verifiable using only the notebook file itself. Selected rules and their descriptions are presented in Table 2.

3.3.3. Machine Learning-specific Rules

Machine learning-specific rules are derived from literature [22, 8] and focus mostly on correct usage patterns of ML-related libraries, particularly `pandas` (version 2.2) and `scikit-learn` (version 1.6.1). Our framework is extensible, allowing easy integration of additional libraries and checks. Selected rules and their descriptions are presented in Table 3.

4. Exploratory Analysis on Real-World Python Notebooks

To assess the practical relevance of our linter, we applied it to a representative sample of 5,000 real-world Jupyter notebooks from the Kaggle platform. This empirical evaluation aims to answer whether real notebooks exhibit rule violations detectable by our tool and, if so, which types of violations are most common, and what do they reveal about prevalent coding and notebook practices.

By aggregating and analyzing the rule violations detected across the dataset, we were able to uncover usage patterns and recurring issues—from common structural inconsistencies (e.g., misplaced imports) to ML-specific issues such as lack of control over randomness or missing parameters in data loading.

As we will see, certain rules are violated in a majority of notebooks, sometimes repeatedly within the same notebook, while others are rarely or never triggered. This analysis validates the utility of our tool and also offers insights into current practices in notebook development and potential areas for improvement.

Table 2
Notebook-Level Linting Rules Implemented in Vespucci

Rule Name	Description	Motivation
N1 - Notebook naming	Enforces the use of filenames longer than 2 characters (*N1.1*) with only alphanumeric characters, hyphens, or underscores (*N1.2*). Detect default names containing `Untitled` (*N1.3*).	Improves clarity, navigation, and compatibility across systems. Prevents confusion during sharing or version control.
N2 - Version control	Requires displaying library versioning information with `package.__version__` or using `watermark`[15].	Ensures reproducibility and helps diagnose variances across executions or environments.
N3 - Imports at the top	Enforces grouping all import statements in the first code cell.	Enhances readability and makes dependencies explicit for reproducibility.
N4 - Long code cells	Detect cells exceeding 30 lines of code.	Encourages modular design, improves readability.
N5 - Empty code cells	Detects code cells that contain no code.	Avoids unnecessary visual clutter and potential confusion.
N6 - Missing text cells	Checks for absence of markdown cells providing context or explanation.	Promotes documentation and guides readers through the workflow.
N7 - Notebook too long	Detect notebooks with more than 50 code cells.	Notebooks should remain reasonably short and focused.
N8 - Non-linear execution	Detects out-of-order execution based on cell execution counts (if present).	Ensuring a linear execution order helps maintain a clear, predictable workflow and supports reliable re-execution of the notebook from top to bottom.

4.1. Dataset

Our empirical analysis is based on the open-source **KGTorrent** dataset [26], which contains 248,761 Python notebooks collected from Kaggle between November 2015 and October 2020. Each notebook is named following the pattern `<kaggle_username>+<notebook_name>` and is organized alphabetically. For computational feasibility, we randomly selected and analysed 5,000 notebooks from the dataset, without applying any thematic or quality-based filtering.

4.2. Experimental Setup

We processed the selected notebooks using our analysis tool, which generates a report for each notebook, listing the violated rules. We then aggregated these individual reports into a unified dataset where each entry corresponds to a rule violation, identified by the notebook name and the rule violated. To analyze the aggregated results, we grouped the data by rules and computed the following statistics: the total number of violations per rule (num_violations), the number of distinct notebooks in which each rule was violated (num_notebooks), and the percentage of our analysed notebooks affected by each rule (pct_notebooks). Additionally, we calculated the average number of violations per affected notebook (violations_per_affected_nb). Each rule was also mapped to the corresponding level (*Python*, *Notebook*, or *ML*). Finally, we organized and sorted the results based on the frequency of violations, and applied a visual style to highlight the most frequent issues.

Table 3
ML-Specific Linting Rules Implemented in Vespucci

Rule Name	Description	Motivation
M1 - Uncontrolled randomness	Detects missing random seed parameters in ML-related functions (e.g., train/test split, model initialization).	Ensures reproducibility of results and consistency across runs.
M2 - In-place API misuse	Warns when functions like `dropna()` are used without assigning the result.	Prevents logic errors due to misunderstanding between in-place modification and returning copies.
M3 - Implicit hyperparameters	Detects ML model instantiations where key hyperparameters are not explicitly set.	Enhances transparency, reproducibility, and adaptability to future library changes.
M4 - Columns and dtypes not set	Detects when columns (*M4.1*) and datatypes (*M4.2*) are not explicitly specified during data loading.	Improves data integrity, parsing performance, and avoids unintended type inference.
M5 - Merge without explicit parameters	Detects DataFrame merges where parameters `on`, `how` (*M5.1*), or `validate` (*M5.2*) are not specified.	Reduces risk of ambiguous or incorrect merges and improves code clarity.

4.3. Analysis

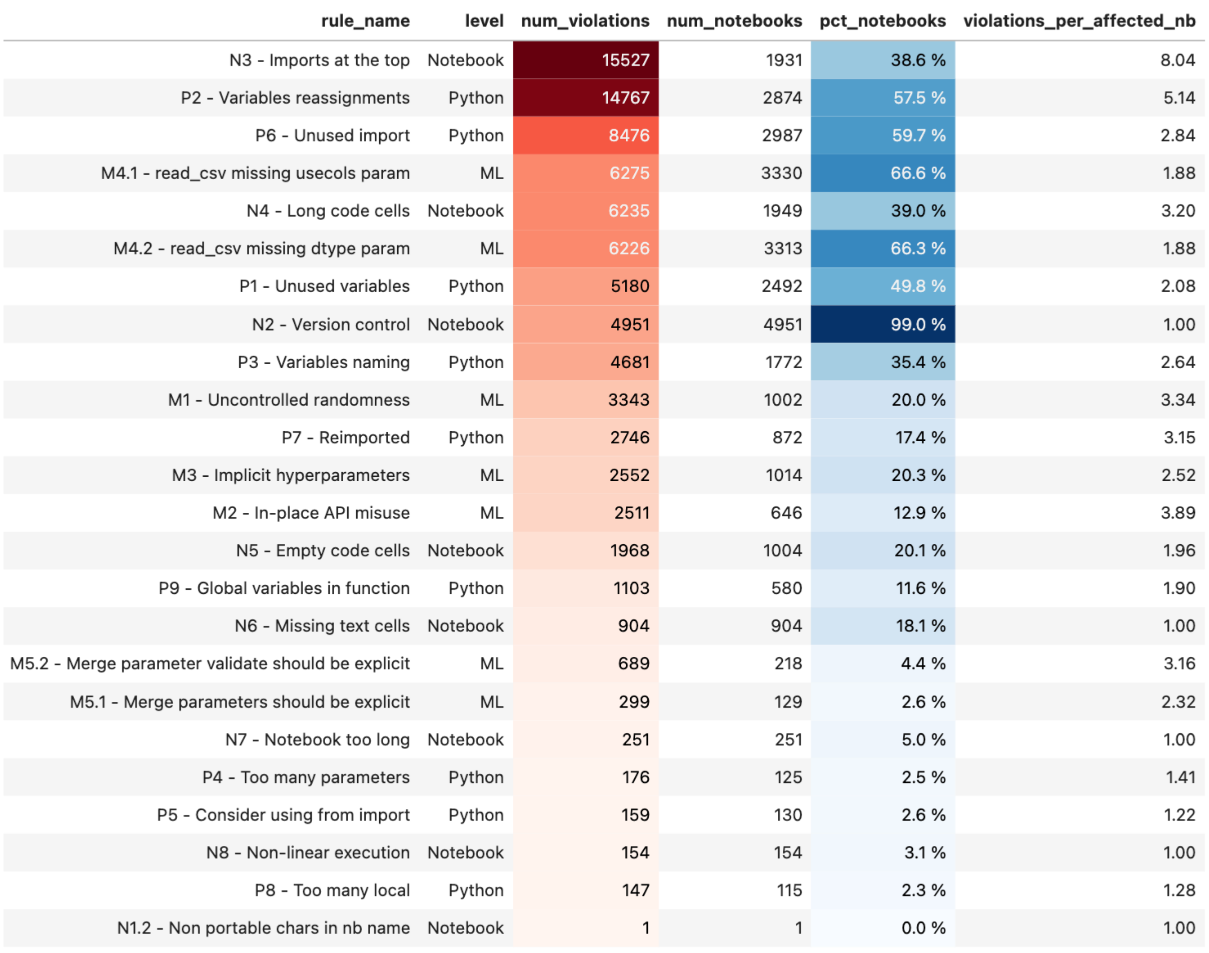
Figure 3 displays the rules that we found have at least one violation on our dataset of 5,000 notebooks. During the experiment, we conducted a manual verification to assess the accuracy of the detected violations. We selected 15 notebooks and reviewed the violated rules. This inspection revealed that most rules were correctly triggered. However, we identified consistent false positives for the rule *Unused Variable*. These errors are due to technical limitations in our current Python parsing mechanism. This limitation will be addressed in future work to improve detection precision. The analysis of the 5,000 notebooks from our subsample reveal several patterns in the rule violations detected by our tool.

Frequent multiple violations per notebook. Some rules are frequently violated and tend to appear multiple times within the same notebook. The rule *N3 - Imports at the top* was violated 15,527 times across 1931 notebooks—38.6% of the sample—with an average of 8.04 violations per affected notebook. Similarly, the rule *P2 - Variable reassignments* appear in 2874 notebooks (57.5%) with a total of 14,767 violations (5.14 per notebook). These findings could suggest that once a certain practice is adopted in a notebook, it tends to be consistently repeated throughout.

Widespread but single-instance violations. Some rules are structurally limited to a single occurrence per notebook. This is the case for *N2 - Version control*, which can only be violated once per notebook. As we can see, it is violated in 4951 notebooks—99.0% of the dataset. This high prevalence may reflect different possibilities: our rule definition might be too narrow, and alternative detection strategies could be more appropriate, environment-level indicators (e.g., `requirements.txt`, Git) should be included in the analysis, or a widespread lack of version control usage on the studied notebooks (which is the least likely). This contrasts with rules like *N3 - Imports at the top* , which, despite a higher absolute number of violations, only affects 38.6% of notebooks.

Lack of textual documentation. The rule *N6 - Missing text cells* is violated in 904 notebooks, representing 18.1% of our sample. Each violation occurs once per notebook by design. This could mean those notebooks might be used for rapid prototyping or personal experimentation rather than structured communication or result sharing.

Machine Learning rules. Rules specific to ML highlight some common issues. The omission of `usecols` and `dtype` parameters in `read_csv` calls is very common, with 6275 and 6226 violations respectively, occurring in over 65% of notebooks. Reproducibility-related rules like *M1* -



rule_name	level	num_violations	num_notebooks	pct_notebooks	violations_per_affected_nb
N3 - Imports at the top	Notebook	15527	1931	38.6 %	8.04
P2 - Variables reassignments	Python	14767	2874	57.5 %	5.14
P6 - Unused import	Python	8476	2987	59.7 %	2.84
M4.1 - read_csv missing usecols param	ML	6275	3330	66.6 %	1.88
N4 - Long code cells	Notebook	6235	1949	39.0 %	3.20
M4.2 - read_csv missing dtype param	ML	6226	3313	66.3 %	1.88
P1 - Unused variables	Python	5180	2492	49.8 %	2.08
N2 - Version control	Notebook	4951	4951	99.0 %	1.00
P3 - Variables naming	Python	4681	1772	35.4 %	2.64
M1 - Uncontrolled randomness	ML	3343	1002	20.0 %	3.34
P7 - Reimported	Python	2746	872	17.4 %	3.15
M3 - Implicit hyperparameters	ML	2552	1014	20.3 %	2.52
M2 - In-place API misuse	ML	2511	646	12.9 %	3.89
N5 - Empty code cells	Notebook	1968	1004	20.1 %	1.96
P9 - Global variables in function	Python	1103	580	11.6 %	1.90
N6 - Missing text cells	Notebook	904	904	18.1 %	1.00
M5.2 - Merge parameter validate should be explicit	ML	689	218	4.4 %	3.16
M5.1 - Merge parameters should be explicit	ML	299	129	2.6 %	2.32
N7 - Notebook too long	Notebook	251	251	5.0 %	1.00
P4 - Too many parameters	Python	176	125	2.5 %	1.41
P5 - Consider using from import	Python	159	130	2.6 %	1.22
N8 - Non-linear execution	Notebook	154	154	3.1 %	1.00
P8 - Too many local	Python	147	115	2.3 %	1.28
N1.2 - Non portable chars in nb name	Notebook	1	1	0.0 %	1.00

Figure 3: Frequency and distribution of rule violations detected across 5,000 notebooks.

Uncontrolled randomness (3343 violations in 20% of notebooks) and *M3 - Implicit hyperparameters* (2552 violations in 20.3%) are also prevalent. This could indicate a lack of attention to experimental reproducibility and parameter tracking.

Rarely violated rules Some rules are rarely triggered. For instance, *P4 - Too many parameters*, *P8 - Too many locals*, and *P5 - Consider using from-import* each appear in fewer than 3% of notebooks. For P4 and P8, this might reflect moderate function complexity across the dataset or a tendency toward simple, single-purpose functions. It is also possible that the thresholds defined for these rules are too high or that very few functions were defined.

Marginal or non-appearing rules. Finally, we can see the almost absence of violations of the rules *N1 - Notebook naming* with a single violation. As described in Section 4.1, all notebook filenames were normalized in the original dataset, which prevents this rule from being triggered.

4.4. Threats To Validity

Several threats to validity should be considered when interpreting our findings. During manual inspection, we identified false positives associated with the *Unused Variable* rule. These are due to current limitations in our tool, which will be addressed in future work. For all other rules, however, no systematic false positives were observed. Also, our static analysis tool is intentionally designed to favor false negatives over false positives. This approach can help avoid overreporting violations, but it also implies that our results may underestimate the true number of violations present in the notebooks. Finally, our analysis is based on a relatively small sample of 5270 notebooks, all sourced from the Kaggle platform. This limits the generalizability of our findings, as these notebooks may not be representative of broader notebook usage patterns across other platforms.

5. Conclusion and Future Work

In this paper, we introduced *Vespucci Linter*, an analysis tool designed to detect bad practices in machine learning notebooks across three levels: general Python code, notebook structure, and ML-specific workflows. Unlike existing tools that focus on a single level, our tool leverages metamodels to provide richer and more contextualized analysis. We implemented 22 linting rules based on state-of-the-art literature and validated their effectiveness through an exploratory analysis of 5,000 real-world notebooks collected from the Kaggle platform. The results highlight widespread violations on the three levels which confirms the relevance of our multi-level approach.

Diverse directions remain to be explored in future work. First, we plan to integrate our tool more directly into notebook environments, such as JupyterLab or VSCode to provide real-time overlays or inline annotations. This would allow developers to receive immediate feedback as they write or modify notebooks. Additionally, we aim to develop more semantically rich linting rules that take full advantage of our tool's capabilities. Future rules could focus on deeper semantic aspects such as pipeline structure validation or data leakage detection—analyses that are difficult to implement with traditional linters. Finally, a more extensive empirical study is needed to explore the early results introduced in the exploratory analysis.

Acknowledgments

This work was supported by the French National Research Agency (ANR) under project ANR-AAPG2024, PROFIL.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, R. Vasa, A large-scale comparative analysis of coding standard conformance in open-source data science projects, in: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20, Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3382494.3410680. doi:10.1145/3382494.3410680.

[2] M. S. Siddik, C.-P. Bezemer, Do code quality and style issues differ across (non-)machine learning notebooks? yes!, in: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023, pp. 72–83. doi:10.1109/SCAM59687.2023.00018.

[3] D. E. Knuth, Literate programming, The Computer Journal 27 (1984) 97–111. URL: https://doi.org/10.1093/comjnl/27.2.97. doi:10.1093/comjnl/27.2.97. arXiv:https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf.

[4] V. Stodden, F. Leisch, R. D. Peng (Eds.), Implementing Reproducible Research, 1st ed., Chapman and Hall/CRC, 2014. URL: https://doi.org/10.1201/9781315373461. doi:10.1201/9781315373461.

[5] H. Shen, Interactive notebooks: Sharing the code, TNature 515 (2014).

[6] Z. Chen, L. Chen, W. Ma, B. Xu, Detecting code smells in python programs, in: 2016 International Conference on Software Analysis, Testing and Evolution (SATE), 2016, pp. 18–23. doi:10.1109/SATE.2016.10.

[7] L. Quaranta, F. Calefato, F. Lanubile, Eliciting best practices for collaboration with computational notebooks, Proc. ACM Hum.-Comput. Interact. 6 (2022). URL: https://doi.org/10.1145/3512934. doi:10.1145/3512934.

[8] H. Zhang, L. Cruz, A. van Deursen, Code smells for machine learning applications, in: Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 217–228. URL: https://doi.org/10.1145/3522664.3528620. doi:10.1145/3522664.3528620.

[9] A. Rule, A. Tabard, J. D. Hollan, Exploration and explanation in computational notebooks, in: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1–12. URL: https://doi.org/10.1145/3173574.3173606. doi:10.1145/3173574.3173606.

[10] J. Wang, L. Li, A. Zeller, Better code, better sharing: on the need of analyzing jupyter notebooks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 53–56. URL: https://doi.org/10.1145/3377816.3381724. doi:10.1145/3377816.3381724.

[11] A. P. Koenzen, N. A. Ernst, M.-A. D. Storey, Code duplication and reuse in jupyter notebooks, in: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020, pp. 1–9. doi:10.1109/VL/HCC50065.2020.9127202.

[12] T. Menzies, The five laws of se for ai, IEEE Software 37 (2020) 81–85.

[13] G. Dolcetti, A. Cortesi, C. Urban, E. Zaffanella, Towards a high level linter for data science, in: Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains, NSAD '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 18–25. URL: https://doi.org/10.1145/3689609.3689996. doi:10.1145/3689609.3689996.

[14] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, M. Derras, Modular moose: A new generation of software reverse engineering platform, in: S. Ben Sassi, S. Ducasse, H. Mili (Eds.), Reuse in Emerging Software Engineering Practices, Springer International Publishing, Cham, 2020, pp. 119–134.

[15] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software 167 (2020) 110610. URL: https://www.sciencedirect.com/science/article/pii/S0164121220300881. doi:https://doi.org/10.1016/j.jss.2020.110610.

[16] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 482. URL: https://doi.org/10.1145/3180155.3182532. doi:10.1145/3180155.3182532.

[17] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering 39 (2013) 1144–1156. doi:10.1109/TSE.2012.89.

[18] PyCQA, Pylint Messages Overview, https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html, 2025. URL: https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html, accessed: 2025-05-03.

[19] B. van Oort, L. Cruz, M. Aniche, A. van Deursen, The prevalence of code smells in machine learning projects, in: 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN), 2021, pp. 1–8. doi:10.1109/WAIN52551.2021.00011.

[20] L. Quaranta, F. Calefato, F. Lanubile, Eliciting best practices for collaboration with computational notebooks, Proceedings of the ACM on Human-Computer Interaction 6 (2022) 1–41. doi:10.1145/3512934.

[21] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez, P. W. Rose, Ten simple rules for writing and sharing computational analyses in jupyter notebooks, PLOS Computational Biology 15 (2019) 1–8. URL: https://doi.org/10.1371/journal.pcbi.1007007. doi:10.1371/journal.pcbi.1007007.

[22] A. Nikanjam, F. Khomh, Design smells in deep learning programs: An empirical study, in: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 332–342. doi:10.1109/ICSME52107.2021.00036.

[23] L. Quaranta, F. Calefato, F. Lanubile, Pynblint: a static analyzer for python jupyter notebooks, in: Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 48–49. URL: https://doi.org/10.1145/3522664.3528612. doi:10.1145/3522664.3528612.

[24] P. Subotić, L. Milikić, M. Stojić, A static analysis framework for data science notebooks, in: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 13–22. URL: https://doi.org/10.1145/3510457.3513032. doi:10.1145/3510457.3513032.

[25] B. van Oort, L. Cruz, B. Loni, A. van Deursen, "project smells": experiences in analysing the software quality of ml projects with mllint, in: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 211–220. URL: https://doi.org/10.1145/3510457.3513041. doi:10.1145/3510457.3513041.

[26] L. Quaranta, F. Calefato, F. Lanubile, Kgtorrent: A dataset of python jupyter notebooks from kaggle, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 550–554. doi:10.1109/MSR52588.2021.00072.