Do Code Quality and Style Issues Differ Across (Non-)Machine Learning Notebooks? Yes!

Md Saeed Siddik and Cor-Paul Bezemer
Department of Electrical and Computer Engineering,
University of Alberta, Canada
{msiddik, bezemer}@ualberta.ca

***Abstract*—The popularity of computational notebooks is rapidly increasing because of their interactive code-output visualization and on-demand non-sequential code block execution. These notebook features have made notebooks especially popular with machine learning developers and data scientists. However, as prior work shows, notebooks generally contain low quality code. In this paper, we investigate whether the low quality code is inherent to the programming style in notebooks, or whether it is correlated with the use of machine learning techniques. We present a large-scale empirical analysis of 246,599 open-source notebooks to explore how machine learning code quality in Jupyter Notebooks differs from non-machine learning code, thereby focusing on code style issues. We explored code style issues across the Error, Convention, Warning, and Refactoring categories. We found that machine learning notebooks are of lower quality regarding PEP-8 code standards than non-machine learning notebooks, and their code quality distributions significantly differ with a small effect size. We identified several code style issues with large differences in occurrences between machine learning and non-machine learning notebooks. For example, package and import-related issues are more prevalent in machine learning notebooks. Our study shows that code quality and code style issues differ significantly across machine learning and non-machine learning notebooks.**

***Index Terms*—Jupyter Notebook, Python Code Style, Code Quality, SE4ML**

I. Introduction

Computational notebooks are the contemporary application of Knuth's literate programming paradigm [22], which combines a programming language with a documentation language to increase the program's robustness, portability, and maintainability. Computational notebooks offer unique developer features that are not available in traditional IDEs [17], such as independent code cell execution and integrated output visualization [30], [48]. Notebooks have already been successful in a wide range of applications, i.e., developing machine learning models [30], [41] and building rapid prototypes [35]. However, computational notebooks have become particularly popular in the data science and machine learning domains, as the above unique features facilitate data exploration and analysis [17], [30].

The most popular computational notebook technology by far is Jupyter Notebook [30]. While Jupyter Notebook supports several programming languages, such as R, Julia, and Scala, most notebooks are written in Python [7], [38]. The popularity of Python in notebooks goes hand in hand with its popularity with machine learning (ML) developers.

The sudden increase in the popularity of notebooks has led to software engineering challenges related to their quality [32]. Wang et al. showed that Jupyter notebooks have lower quality code than Python scripts in terms of following PEP-8 guidelines [46]. Additionally, van Oort et al. showed that ML projects contain many code style issues [41]. Since many notebooks contain ML code, it is unclear whether the notebook programming style or ML features cause the lower code quality.

In this paper, we investigate whether the usage of ML functionality is the reason for the low code quality or whether non-ML notebooks suffer from the same problem. Our study will help narrow down the direction of future research efforts on improving notebook code quality.

We conduct a large-scale empirical analysis of the code quality and code style issues in ML and non-ML notebooks. We study a publicly available open-source dataset [33] of 246,599 Jupyter notebooks written in Python. We use the PEP-8 Python code standard [4], [42] to measure the code quality and identify the code style issues in the notebooks. In particular, we address the following research questions (RQs):

RQ1: How do the code quality ratings differ between ML and non-ML notebooks?
Code quality ratings are lower in ML than in non-ML notebooks, and their distributions are significantly different, with a small effect size. Compared to non-ML notebooks, the median code quality rating in ML notebooks is lower (2.2 and 3.3 for ML and non-ML respectively).

RQ2: How do the code style issues differ between ML and non-ML notebooks?
The distributions of all four studied categories of code style issues are significantly different, where the Error, Warning, and Refactoring categories have non-weak effect sizes. Package or library handling-related issues are the most prevalent among the categories, with larger differences between ML and non-ML notebooks distributions.

Based on our findings, we conclude that the quality of ML code in Jupyter notebooks is significantly different from non-ML code. ML code depends more on libraries, which is the primary reason for causing code style issues and low quality code. Our studies suggest a need for different style guidelines for ML code in notebooks. We also recommend developing tools to detect and refactor ML-notebook-specific code issues. The code and results of this study are available publicly.[1]

Paper Organization. The rest of this paper is organized as follows. Section II gives background information about our study. Section III describes related work. Section IV presents our methodology. Sections V and VI present the findings of our two research questions. Section VII discusses the implications of our findings. Section VIII describes the threats to the validity of our work. Finally, Section IX concludes the paper.

II. Background

Python Enhancement Proposal 8, or PEP-8, is a style guide for Python code that lists all possible code style violations that affect the overall code quality [15], [42]. PEP-8 can be used as a reference for measuring Python code quality, i.e., using a tool like *Pylint*. This section gives background information about code style issues and code quality.

A. Python Code Style Issues

We used the *Pylint* tool to statically identify code style issues in notebooks [2], [41]. *Pylint* groups code style issues into six categories, i.e., Convention, Error, Fatal, Information, Refactor, and Warning [6]. Since *Pylint*'s code quality rating (see Section II-B) is measured based on four of these categories (Error, Convention, Warning, and Refactor), we focus on these four in our study as well. Note that *Pylint* stops processing a notebook as soon as it encounters a Fatal issue (such as a syntax error). We do not include such notebooks as this *Pylint* behaviour could bias our results. Below, we explain the four studied categories of code style issues.

- **Errors (E):** These style issues indicate coding problems that are likely to cause the program to behave incorrectly. They indicate severe problems with the code that might prevent it from running too. Examples of Error messages include syntax errors, undefined variables, and unsupported operators. These issues are serious and should be fixed as soon as possible.
- **Conventions (C):** These style issues refer to a set of PEP-8 coding guidelines and best practices recommended for writing readable and maintainable Python code. Issues in this category are less likely to cause actual bugs. For example, this category covers conventions for naming variables, using whitespace, and the length of a single code line.
- **Warnings (W):** These style issues indicate potential problems in the code but do not cause the program to behave incorrectly. Examples of Warning code style issues include using deprecated functions, ignoring exceptions, or using an else clause after a return statement. These issues are less serious compared to Error messages but should still be addressed.
- **Refactoring (R):** These style issues cause the code to be less readable or maintainable, but they do not cause the program to behave incorrectly. Examples of Refactoring code style issues include not using a docstring, a too-large function or nondescriptive variable names.

[1]https://github.com/saeedsiddik/NotebookCodeStyleIssue

B. Code Quality

Pylint computes a code quality rating between 0 and 10, which indicates the quality of the code in the notebook, with a higher rating indicating higher quality code. Equation 1 presents the code quality rating equation used by *Pylint*, which combines the four code style issue categories discussed above to compute the code quality rating of a Python file (or, in our case, a notebook) [5]. Errors (E) are given the most weight (5 times more than the other categories) since they can be potential bugs and affect the functionality of the notebook. The other categories are given equal weight. Furthermore, SLOC denotes the number of lines of code analyzed by *Pylint* [13], [27].

$$\textit{Rating} = max(0,\ 10\ \ -\ \frac{5 \times E + R + W + C}{\text{SLOC}} \times 10) \quad (1)$$

For example, consider a notebook with 50 SLOC in which *Pylint* detects 2 Errors, 0 Refactoring issues, 10 Warnings, and 5 Convention-related issues. Then the rating of that notebook will be:

$$max(0,\ 10\ \ -\ \frac{5 \times 2 + 0 + 10 + 5}{50} \times 10) = 5$$

III. Related Work

Jupyter notebooks are indispensable for data analysis and scientific computing. However, the code quality and style in these notebooks are often neglected. This section discusses prior related work on the code quality and style issues in Jupyter notebooks and on code analysis of notebooks.

A. Notebook Code Quality

Code quality is a critical aspect of software development, especially in the context of Jupyter notebooks. As the popularity of Jupyter notebooks continues to grow in the data science and machine learning communities, it becomes increasingly important to ensure that the code in these notebooks is of high quality. According to Wang et al., the quality of the code in Jupyter notebooks is lower than in Python scripts in terms of adhering to PEP-8 standards [46]. Wang et al. examined notebooks from GitHub and discovered more errors (36.26%) than in Python scripts (13.40%). They also reported that it is hard to reproduce Jupyter notebooks, as 73% of the publicly available notebooks were not reproducible with straightforward approaches [45], [47]. Pimentel et al. demonstrated that more than 75% of valid notebooks could not be re-executed properly and that fewer than 5% would achieve the same outcomes as the original [32].

Several studies have focused on how to improve the code quality in notebooks. Subotic et al. proposed a static analysis framework for data science notebooks to find potential issues inside the code [39]. Their framework can detect data leakage bugs triggered by wrong notebook execution sequences. Wang et al. designed an automated documentation generation system to increase the computational quality of notebooks and make them more user-friendly [43]. Their system learned from 80 highly-voted Kaggle notebooks and was evaluated by 24 data science practitioners. Patra and Pradel presented an automatic name-value inconsistency detection tool for Jupyter Notebooks to improve code quality by resolving misleading variable names and their values [28]. They reported that their approach complements existing techniques for finding coding issues with 80% precision and 76% recall. Dong et al. demonstrated that cleaning activities in Jupyter Notebooks improve their quality [14]. They presented several code-cleaning activities for improving understanding in a notebook, such as adding markdown cells and reordering and splitting code cells. Titov et al. showed that properly splitting notebook code cells can improve the structure and quality of Jupyter notebooks [40]. They proposed an algorithm for automatically splitting cells and reported that in 29.5% of cases, their suggestions were selected as the preferred way of perceiving the code.

In contrast to prior work, our study focuses on analyzing how (not) using machine learning-related functionality is correlated with code quality in Jupyter notebooks.

B. Code Style Issues and Smells

Addressing code style issues and smells is essential in notebooks because they affect the code's readability, maintainability, and overall quality. A consistent and clean code style can make the code more understandable and easier to work with, while code smells can indicate deeper problems or inefficiencies. There have been many studies on detecting code style issues — we refer to recent survey papers for an overview [8], [23]. In this section, we give an overview of empirical studies on code style issues in notebooks, Python code and machine learning systems.

Van Oort et al. found at least one style issue in every studied notebook during their study [41]. They conducted an empirical study on the prevalence of code smells in Python ML notebooks and found that style issues like *unused-import* and *invalid-name* are the most frequent issues. Grotov et al. showed that notebook code differs from Python scripts having 1.4 times more stylistic issues [17]. They presented their empirical results from structural and stylistic points of view and found more issues in every comparison.

Simmons et al. reported that coding style issues are also more frequent in data science projects written in Python than in non-data science projects [37]. ML code involves more complex, and longer expressions than traditional code, and the number of code smells increases across the releases of ML applications [19]. Researchers also studied that Python code tends to violate coding standards, i.e., Bafatakis et al. showed that 93.87% of Python code on Stack Overflow contains style violations, with an average of 0.7 violations per statement [9]. Code clones are a type of code smell that is also frequent in ML code [20]. Nikanjam and Khomh reported that deep learning programs are designed differently and have unique architectural design smells in the models [26]. Zhang et al. reported that machine learning code quality is more challenging to evaluate [49]. They conducted a literature review of academic papers, gray literature, Github commits, and Stack Overflow posts to discover ML-specific code smells to assess the quality of ML code.

Existing literature empirically studied code style issues and smells in notebooks, in Python and in ML code. However, we are the first to combine these three targets by studying the differences in code style issues between ML and non-ML notebooks in Python.

IV. Methodology

This section describes the methodology of our empirical study on examining whether the low quality code is a byproduct of notebook programming or if it is associated with the usage of ML functionality. Figure 1 gives a visual overview of the steps taken in our methodology. This section explains how we collect notebook data, process notebooks, and analyze code style issues.

A. Collecting and Labeling Notebooks

We studied the publicly available dataset of Jupyter Notebooks from Quaranta et al. [33]. The dataset contains $246,599$ notebook files from the Kaggle repository written in Python and includes notebooks that were published between November 2015 and October 2020. We categorized the notebooks based on whether they use at least one machine learning library. If yes, the notebook is labelled as an ML notebook; otherwise, it is labelled as a non-ML notebook. We retrieved a list of machine learning libraries from literature [24], [41] and matched this list with all $import$ and $from$ statements in the notebooks. We ended up with $177,252$ ML notebooks (71.9%) and $69,347$ non-ML notebooks (28.1%). ML notebooks (median SLOC 75) are typically larger than non-ML notebooks (median SLOC 43). The Mann-Whitney U test (see Section IV-C) shows that the distribution of SLOC between ML and non-ML notebooks is significantly different ($p < 0.05$) with a medium effect size (Cliff's Delta $d = 0.41$).

B. Processing and Detecting Code Style Issues

We analyzed the notebooks in the dataset using *Pylint* (through the *nbQA* [16] tool) and stored the output in a separate corresponding text file. *nbQA* is a tool that facilitates the execution of standard Python code style tools on Jupyter notebooks. We used *Pylint* version $2.14.0$ to identify the code style issues and compute the code quality ratings.

We developed a Python script to extract the necessary information (file name, code quality rating, and code style issues) from the *Pylint* output files. We stored the extracted style issues and code quality ratings with the corresponding filename in an SQLite database for further processing. The style issues are categorized by *Pylint* into the Error, Convention, Warning, and Refactoring categories. We extracted a total number of $2,773,937$ issues that occur in our dataset, where

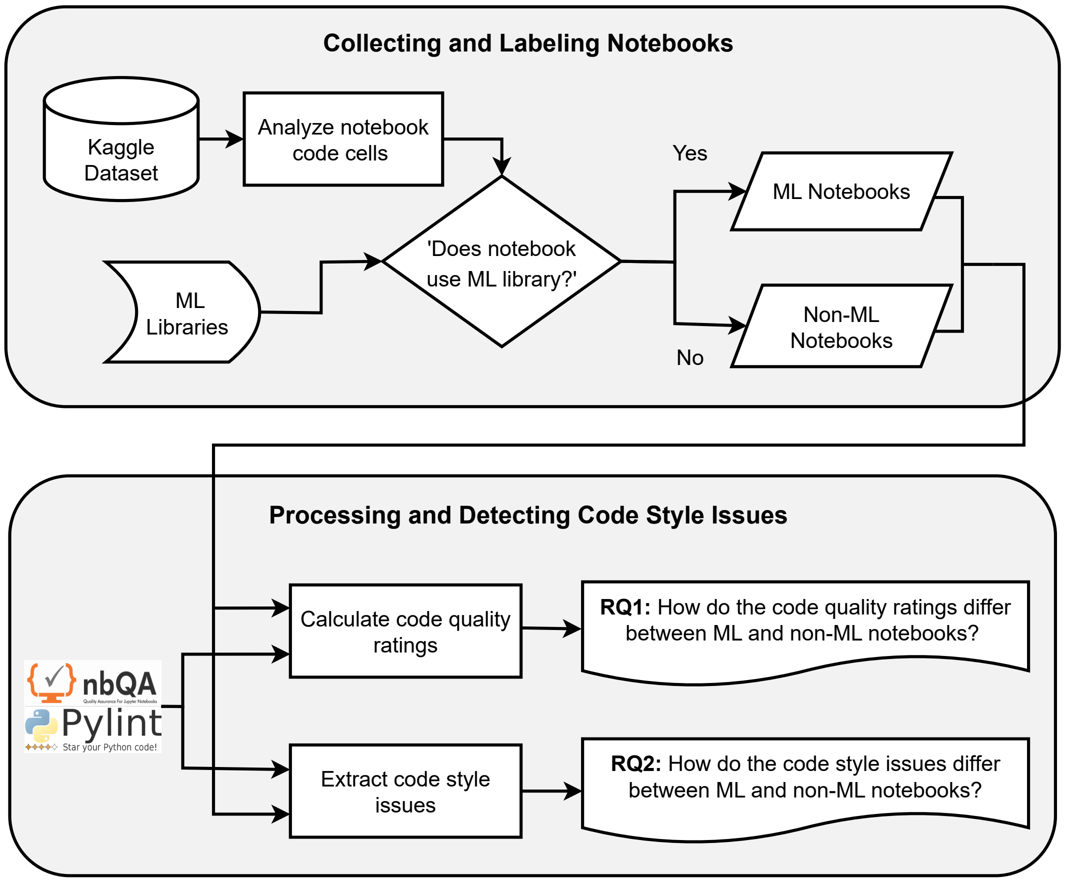

Collecting and Labeling Notebooks
Kaggle Dataset
Analyze notebook code cells
ML Libraries
'Does notebook use ML library?'
Yes
No
ML Notebooks
Non-ML Notebooks
Processing and Detecting Code Style Issues
nbQA
Pylint
Star your Python code!
Calculate code quality ratings
Extract code style issues
RQ1: How do the code quality ratings differ between ML and non-ML notebooks?
RQ2: How do the code style issues differ between ML and non-ML notebooks?

Fig. 1: Overview of the methodology of our empirical study

77.9% $(2,159,994)$ and 22.1% $(613,943)$ come from ML and non-ML notebooks respectively. We also excluded 28 ML and 2 non-ML notebooks from our analysis due to the presence of a Fatal issue in the notebook.

C. Statistical Analysis

We used Pearson's Chi-square (χ^2) test [29] to determine whether the numbers of detected code style issues in a category (see Section VI) differ significantly between ML and non-ML notebooks. We selected the chi-square test since basically, we are comparing the columns of a contingency table in which the rows represent the code style issues and the columns the number of occurrences of those issues in ML and non-ML notebooks. Additionally, we individually explored all four code style issue categories (Error, Convention, Warning, and Refactoring) to report how they differ across ML and non-ML notebooks. Although we consider a 5% level of significance for the chi-square test, we adjusted the p-value using Bonferroni correction to $\frac{0.05}{4}$ to mitigate the effect of making multiple comparisons.

If the chi-square test shows that the distributions within a category differ significantly, we calculate the Cramér's V effect size [12] to quantify the difference:

$$\text{Cramér's V} = \sqrt{\frac{\chi^2}{n(k-1)}}$$

where k is the length of the smaller dimension. We use the following thresholds to interpret Cramér's V effect size [3]:

$$\text{Cramér's } V \text{ Effect size} = \begin{cases} \textit{weak}, & \text{if } |V| < 0.2 \\ \textit{moderate}, & \text{if } 0.2 \le |V| < 0.6 \\ \textit{strong}, & \text{if } 0.6 \le |V| \end{cases}$$

In addition, to identify which of the code style issues cause the distributions to be significantly different, we conducted a posthoc analysis using the residuals of the chi-square test [36]. The code style issues with the largest residuals contribute the most to the chi-square statistic. We used standardized residuals, also called Pearson residuals [18], to make the residuals more comparable. To calculate the standard residual of a code style issue, we measure the difference in its frequency between ML and non-ML notebooks and then divide it by the square root of the frequency in non-ML notebooks. Following Sharpe [36], we define residuals as high when they have a value greater than 3 or less than -3.

We used the Mann-Whitney U test [25] to determine whether code quality rating distributions (see Section V) or line of code distributions are statistically different. Since those distributions are independent and continuous but neither categorical nor normally distributed, we selected the Mann-Whitney U test. If the p-value, produced by the test, falls below a cutoff of 0.05, we consider those distributions as significantly different. Then we calculate Cliff's delta (d) effect size [11] to show the magnitude of the difference. We follow Romano et al.'s [34] thresholds to interpret the value of d:

$$\text{Cliff's } d \text{ Effect size} = \begin{cases} \textit{negligible}, & \text{if } |d| < 0.147 \\ \textit{small}, & \text{if } 0.147 \leq |d| < 0.330 \\ \textit{medium}, & \text{if } 0.330 \leq |d| < 0.474 \\ \textit{large}, & \text{if } 0.474 \leq |d| \end{cases}$$

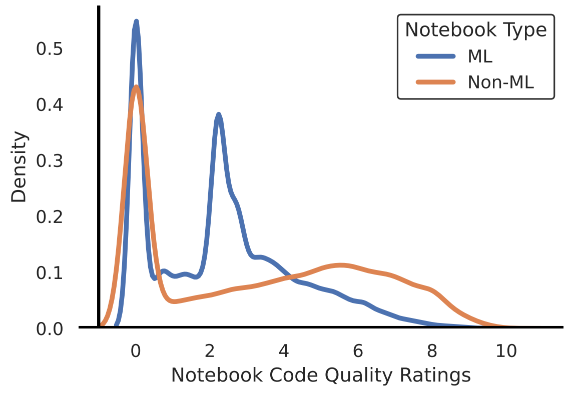


Fig. 2: Distribution of the code quality ratings in ML and non-ML notebooks

V. RQ1: How do the code quality ratings differ between ML and non-ML notebooks?

A. Motivation

This research question aims to determine whether notebook code quality is correlated with the use of ML features or inherent to the programming style in notebooks. Research showed that Python code in notebooks is of lower quality than Python in scripts [17]. Since the majority of notebooks contain ML code, we investigate whether the ML code is the main reason for the low-quality code. Hence, we study the code quality ratings across ML and non-ML notebooks.

B. Approach

We grouped the collected code quality ratings (found by *Pylint*) by the notebook type (ML and non-ML). Code quality ratings range from 0 to 10; the higher the rating, the better the notebook code quality. We used the Mann–Whitney U-test with a 5% significance level to test the differences between the distributions. We computed Cliff's delta (d) effect size to determine how much code quality ratings differ between ML and non-ML notebooks.

We report the skewness of the distributions of ML and non-ML notebooks' code quality ratings to determine the differences in the shapes and symmetry of the distributions. A positive skew (between 0.5 and 1) indicates that the distribution is skewed to the right (mostly low quality ratings), and a negative skew (between -1 and -0.5) indicates that the distribution is skewed to the left (a majority of high quality ratings). We consider two distributions unskewed if the skewness value is between -0.5 and 0.5.

We describe the distributions' kurtosis to find the peak in notebook code quality ratings. The greater the kurtosis value, the higher the peak. The expected value of kurtosis is 3. A distribution with a kurtosis smaller than 3 (negative kurtosis) has a flat peak. In contrast, a distribution with kurtosis of higher than 3 (positive kurtosis) has a sharp peak.

C. Findings

The code quality ratings of ML and non-ML notebooks are significantly different, with a small effect size. Figure 2 shows the distribution of the code quality ratings of ML and non-ML notebooks. The median of ML notebooks' quality ratings is 2.2, lower than non-ML notebooks' quality ratings (3.3), which indicates that ML notebooks are of lower quality regarding PEP-8 coding standards. One possibility is that the advances in ML are attracting the attention of users from other disciplines who do not have a background in software engineering and consequently make mistakes in following code standards. The Mann–Whitney U test shows that the code quality rating distributions of ML and non-ML notebooks are significantly different ($p = 0.0$). Cliff's delta ($d = 0.15$) confirms that those rating distributions are different with a small effect size.

The distribution of ML notebooks' code quality ratings is positively skewed, whereas it is unskewed for non-ML. The skewness values of the distributions of the code quality ratings of ML and non-ML notebooks are 0.603 and 0.228, respectively. Hence, the ML code quality rating distribution is positively skewed, and the non-ML code quality rating distribution is unskewed. That indicates that the rating distribution of ML notebooks is more skewed to the right than that of the non-ML notebooks (i.e., they are generally lower).

The distribution of code quality ratings has a lower kurtosis value for non-ML notebooks than for ML notebooks. Both kurtosis values are negative (-0.214 and -1.367, respectively). The lower value of non-ML distributions indicates that the distribution of code quality ratings of non-ML notebooks has a flatter peak and thinner tails compared to that of ML notebooks. According to Figure 2, there are peaks near the value of 0 in both distributions because *Pylint* has a lower cap of 0 for the code quality rating. In addition, there is a second peak (around 2.3) for the ML notebooks. ML notebooks tend to have at least one Error code style issue (vs. zero in non-ML notebooks – details will be discussed in Section VI). We expect that the peak around 2.3 for ML notebooks is caused by Error type issues decreasing the code quality faster than other types of issues (see Equation 1).

The code quality ratings of non-ML notebooks are generally higher than those of ML notebooks, and their distributions are significantly different, with a small effect size.

VI. RQ2: How do the code style issues differ between ML and non-ML notebooks?

A. Motivation:

This research question investigates whether the types of code style issues correlate with using ML functionality. As prior work showed that there exist machine learning-specific style issues [26], [49], we investigate whether the use of ML functionality also affects the occurrence of code style issues in Jupyter notebooks.

B. Approach:

We explored the code style issue distributions in ML and non-ML notebooks across the four studied categories (Error, Convention, Warning, and Refactoring). We retrieved the issue category from the *Pylint* output, where issues are depicted with their code line. The issue code consists of the first letter of the category and the issue number. For example, *E0611* represents an Error category code style issue with style issue number *0611*.

We used the Pearson Chi-square (χ^2) test as described in Section IV-C to determine how code style issues differ between ML and non-ML notebooks. If the chi-square test indicates that the distributions of a code style category are significantly different, we explored their standard residuals to identify which code style issues are the reason for the significant difference.

As an example of generating the contingency table that serves as input to the chi-square test, suppose we want to investigate whether the distributions of Refactoring code style issues are different between ML and non-ML notebooks. To create the contingency table, we detect the list of all unique code style issues under the Refactoring category and count the frequency of every code style issue for ML and non-ML notebooks (including a 0 for zero occurrences).

Because our dataset contains a much larger number of ML notebooks, we normalized the values in the contingency table by replacing them with the percentage of the total number of occurrences in that type of notebook in a style issue category. Table I presents the top 5 most frequent code style issues in each category in such a contingency table. It demonstrates a sample of a contingency table for both using the absolute frequency before conversion and the percentage values after conversion.

We analyzed the *Total %*, *ML %*, *non-ML %*, and *Standard Residuals* of every individual code style issue. Here *Total %* represents the percentage of a style issue within all found style issues in the respective category. For example, the *Total %* of the code $E1101$ is 7.5, which indicates that 7.5% of all Error code style issues are of type $E1101$. Likewise, the *ML %* and *non-ML %* values represent the percentages for that type of notebook within the respective category. Throughout this section, whenever we present the occurrence percentage of a code style issue, it denotes the value within its respective style issue category.

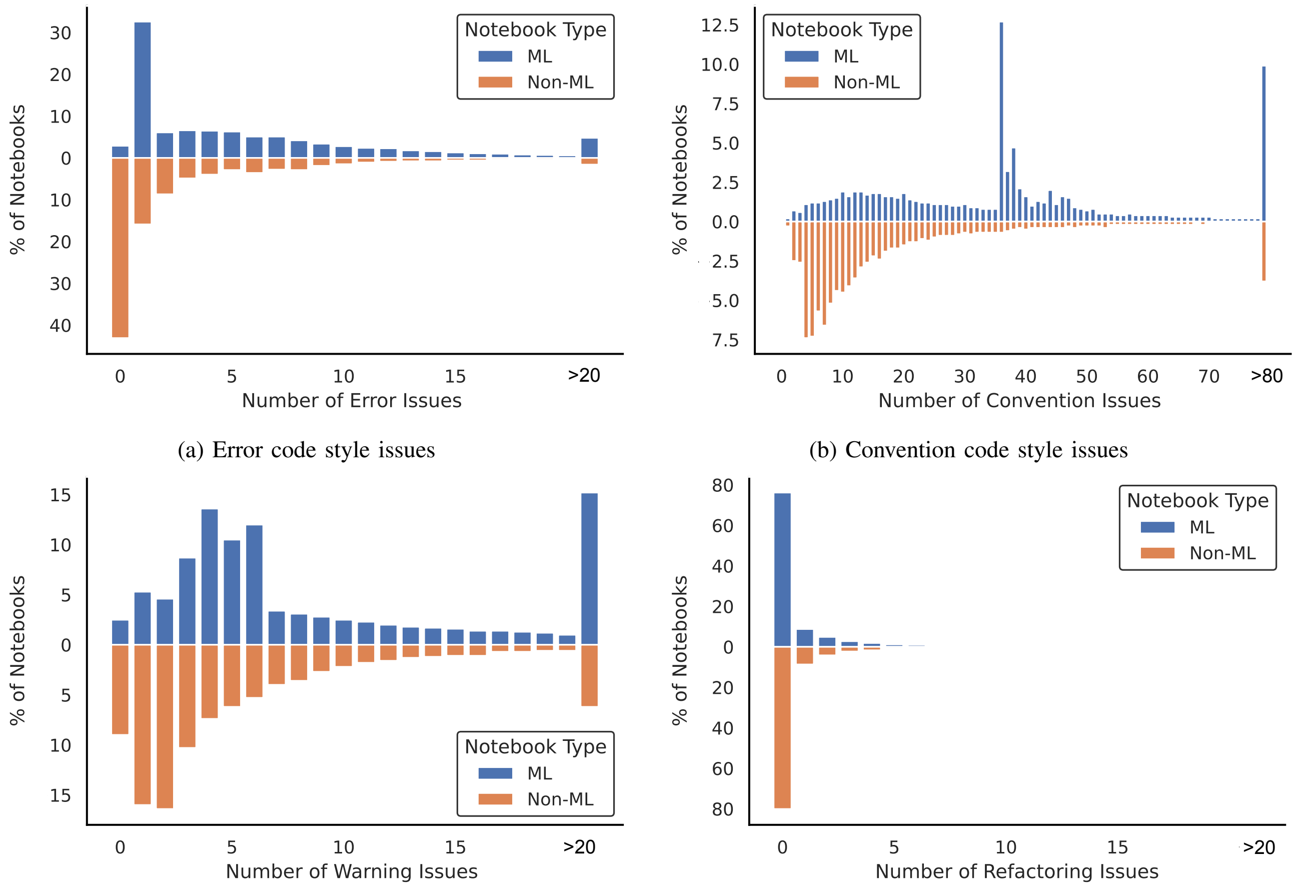
C. Findings

1) Error code style issues: **The prevalence of Error code style issues is higher in ML than in non-ML notebooks; the distributions significantly differ with a moderate effect size**. Figure 3a shows the number of Error code style issues per notebook. The median number of Error code style issues per notebook is higher in ML than in non-ML notebooks having values of 4 and 1, respectively. The chi-square test shows that the error code style issue distributions differ significantly between non-ML and ML notebooks with a moderate effect size ($V = 0.43$). We found that almost all (97.1%) of the ML notebooks contain at least one Error code style issue, in contrast to only 56.9% of the non-ML notebooks. The reason is that 96.1% of ML notebooks have at least one *E0401: Import error* code style issue.

ML notebooks are more sensitive to package or import-related Errors because they deal with more packages than non-ML notebooks. Table II shows that 3 of the top 4 Error-type style issues (ranked by residual) are related to package handling. These package-related Error-type style issues occur almost twice as often in ML notebooks. *E0401: import-error*, occurring when a module fails to import inside the notebook, has the highest residual value (5.28) and occurs in 66.38% of the ML vs. 35.10% of the non-ML notebooks. The package-related Error style issue with the second-largest residual (4.42) is *E1101: No member*. It occurs when a variable is accessed from a nonexistent module or function. This style issue occurs almost four times more frequently in ML (8.61%) than in non-ML notebooks (2.14%). The third package-related Error style issue based on residual is *E0611: No name in module*, which also occurs more frequently in ML (4.86%) than in non-ML notebooks (2.09%). It occurs when a name cannot be found in a package called from the code.

The higher occurrence of package-related Error issues could be due to the fact that machine learning models often require specialized libraries and frameworks, some of which may not be available or compatible with the current Python environment. It may also be because machine learning models are often more complex and require more dependencies than non-machine learning models. We observed that the average usage of packages in ML notebooks is higher (6.05 ± 2.90 packages per notebook) than in non-ML notebooks (3.44 ± 2.28 packages per notebook), which could explain the higher occurrence of package handling-related issues.

2) Convention code style issues: **The distribution of Convention-type code style issues significantly differs between ML and non-ML notebooks with a weak effect size.** The median numbers of Convention-type code style issues are 36 and 11 in ML and non-ML notebooks, respectively. The chi-square test shows that the Convention-type code style



(a) Error code style issues

(b) Convention code style issues

(c) Warning code style issues

(d) Refactoring code style issues

Fig. 3: Category-wise code style issue distributions for ML and non-ML notebooks

issue distributions significantly differ between non-ML and ML notebooks with a weak effect size ($V = 0.16$). We investigated the standard residual values to determine the cause and found that 4 out of 38 issues have standard residuals of more than 1 or less than -1 (shown in Table II). The frequency of Convention code style issues in each ML and non-ML notebook are depicted in Figure 3b. There is a long tail in the Convention code style issues distribution, with a maximum of $9,919$ issues in a single ML notebook. We investigated and found an unusually large notebook with many hardcoded values, which are not written following Python standards causing numerous $(9,900)$ *C0303: trailing-whitespace* issues.

One of the Convention-type code style issues in the top three residuals is related to package handling. Table III shows that no Convention-type code style issues have high residuals (greater than 3 or less than -3), which indicates that no issue really stands out strongly, but there are several that cause a significant difference. The package handling-related style issue (*C0413: wrong-import-position* with a standard residual of 1.29) is also more frequent in ML notebooks (13.9% vs. 9.9% in non-ML notebooks). It occurs when code and imports are mixed. The *C0103: Invalid name* issue has the largest standard residual (1.52) and is also more frequent in ML (28.9%) than in non-ML notebooks (21.8%). It occurs when the name does not conform to naming rules associated with its type, e.g., function names should be lowercase, with words separated by underscores.

3) Warning code style issues: **The prevalence of Warning code style issues is higher in ML than in non-ML notebooks; the distributions significantly differ with a moderate effect size.** Figure 3c shows the number of Warning code style issues per ML and non-ML notebook. The median numbers of occurrences are 6 and 3, respectively. The Pearson chi-square test indicates that the distributions of Warning code style issues significantly differ between ML and non-ML notebooks with a moderate effect size ($V = 0.26$). We explored the standard residuals to check which Warning issues are responsible for the difference. We found one issue with a high standard residual (more than 3), and 7 issues with standard residuals of more than 1 or less than -1 (see Table IV).

One of the top three differences in Warning-type code style issues is package handling-related. Table IV presents the Warning-type code style issues with standard residuals of more than 1 or less than -1. *W0611: Unused import* (third-largest residual, 2.21) is package handling-related, and also the most occurring Warning issue (24.10% of all Warning-type

TABLE I: Part of a contingency table for code style issues for (non-)ML notebooks before and after conversion to percentage

Category	Code	Style Issue	Absolute frequency			Percentage (%)		
			ML	Non-ML	Row Total	ML	Non-ML	Row Total
Error	E0401	import-error	748,983	79,048	828,031	66.38	35.10	61.17
	E0602	undefined-variable	151,569	104,236	255,805	13.43	46.28	18.89
	E1101	no-member	97,152	4,819	101,971	8.61	2.14	7.53
	E1121	too-many-function-args	43,265	16,753	60,018	3.83	7.44	4.34
	E0611	no-name-in-module	54,851	4,713	59,564	4.86	2.09	4.40
		Column Total	1,128,416	225,218	1,353,634	100	100	100
Convention	C0103	invalid-name	2,097,268	323,371	2,420,639	28.78	21.77	27.67
	C0303	trailing-whitespace	1,358,955	380,428	1,739,383	18.71	25.62	19.88
	C0301	line-too-long	1,282,391	313,161	1,595,552	17.66	21.09	18.24
	C0413	wrong-import-position	1,011,630	146,556	1,158,186	13.93	9.87	13.24
	C0116	missing-function-docstring	443,844	71,212	515,056	6.11	4.79	5.89
		Column Total	7,359,795	1,485,122	8,844,917	100	100	100
Warning	W0611	unused-import	571,391	81,964	653,355	25.73	16.70	24.10
	W0621	redefined-outer-name	486,647	85,593	572,240	21.91	17.44	21.10
	W0104	pointless-statement	304,734	132,951	437,685	13.72	27.09	16.14
	W0311	bad-indentation	227,302	40,723	268,025	10.24	8.30	9.88
	W0404	reimported	190,310	31,234	221,544	8.57	6.36	8.17
		Column Total	2,220,708	490,758	2,711,466	100	100	100
Refactor	R1705	no-else-return	17,261	8,234	25,495	11.27	20.22	13.15
	R0913	too-many-arguments	18,402	2,782	21,184	12.01	6.83	10.92
	R0914	too-many-locals	17,081	2,984	20,065	11.15	7.33	10.35
	R0402	consider-using-from-import	15,336	1,727	17,063	10.01	4.24	7.80
	R1732	consider-using-with	10,505	3,293	13,798	6.86	8.09	7.11
		Column Total	153,220	40,726	193,946	100	100	100

TABLE II: Error code style issues having standard residuals greater than 1 or less than -1

Code	Style Issue	Total (%)	ML (%)	Non-ML (%)	Standard Residuals
E0401	import-error	61.17	66.38	35.10	**5.28**
E0602	undefined-variable	18.90	13.43	46.28	**-4.83**
E1101	no-member	7.53	8.61	2.14	**4.42**
E0611	no-name-in-module	4.40	4.86	2.09	1.91
E1121	too-many-function-args	4.43	3.83	7.44	-1.32
E0102	function-redefined	1.24	0.88	3.06	-1.25
E1102	not-callable	0.27	0.32	0.05	1.15
E0107	nonexistent-operator	0.19	0.00	1.16	-1.08
Absolute number of Error code style issues		1,353,634	1,128,416	225,218	

TABLE III: Convention code style issues having standard residuals greater than 1 or less than -1

Code	Style Issue	Total (%)	ML (%)	Non-ML (%)	Standard Residuals
C0103	invalid-name	27.67	28.87	21.77	1.52
C0303	trailing-whitespace	19.88	18.71	25.62	-1.36
C0413	wrong-import-position	13.24	13.93	9.87	1.29
C0114	missing-module-docstring	2.81	2.43	4.65	-1.03
Absolute number of Convention code style issues		8,844,917	7,359,795	1,485,122	

issues). *W0611* occurs when an imported module or variable is not used, which is more prevalent in ML (25.73%) than in non-ML notebooks (16.70%). As we described earlier, since ML notebooks deal with more packages than non-ML notebooks, it is expected to see more unused import issues in ML code. Although the *W1406: Redundant U-String Prefix* issue has a high standard residual (3.62), it is not very frequent in notebooks (2.39% of all Warning issues).

4) Refactoring code style issues: **The distributions of Refactoring code style issues significantly differ across ML and non-ML notebooks with a moderate effect size.** Refactoring code style issues are less prevalent than other categories (only 1.48% of all code style issues). Figure 3d presents the number of Refactoring code style issues per notebook. The chi-square test indicates that the Refactoring code style issue distributions differ significantly between ML and non-ML notebooks with a moderate effect size (V = 0.34). We found 51 unique Refactoring code style issues in the dataset, where a total of 10 issues have standard residuals greater than 1 or less than -1 (shown by Table V).

One of the top three differences in Refactoring-type code

TABLE IV: Warning code style issues having standard residuals greater than 1 or less than -1

Code	Style Issue	Total (%)	ML (%)	Non-ML (%)	Standard Residuals
W1406	redundant-u-string-prefix	2.39	2.82	0.43	**3.64**
W0104	pointless-statement	16.14	13.72	27.09	-2.57
W0611	unused-import	24.10	25.73	16.70	2.21
W0612	unused-variable	3.85	4.22	2.19	1.37
W0301	unnecessary-semicolon	2.34	1.90	4.31	-1.16
W0621	redefined-outer-name	21.10	21.91	17.44	1.07
W0401	wildcard-import	1.02	0.75	2.25	-1.00
Absolute number of Warning code style issues		2,711,466	2,220,708	490,758	

TABLE V: Refactoring code style issues having standard residuals greater than 1 or less than -1

Code	Style Issue	Total (%)	ML (%)	Non-ML (%)	Standard Residuals
R1725	super-with-arguments	4.54	5.70	0.19	**12.77**
R0402	consider-using-from-import	8.80	10.01	4.24	2.80
R0902	too-many-instance-attributes	2.41	2.85	0.72	2.50
R1703	simplifiable-if-statement	1.14	0.20	4.65	-2.06
R1705	no-else-return	13.15	11.27	20.22	-1.99
R0913	too-many-arguments	10.92	12.01	6.83	1.98
R1708	stop-iteration-return	0.14	0.18	0.01	1.70
R0914	too-many-locals	10.35	11.15	7.33	1.41
R0133	comparison-of-constants	0.90	0.54	2.29	-1.16
R1712	consider-swap-variables	0.24	0.01	1.11	-1.04
Absolute number of Refactoring code style issues		193,946	153,220	40,726	

style issues is package handling related. *R0402: Consider using from import* is a package handling-related Warning issue with the second-largest residual (2.80, see Table V). *R0402* occurs when a submodule of a package is imported and aliased with the same name, e.g., use `import os.path as path` instead of using `from os import path`. It is more prevalent in ML (10.01%) than in non-ML notebooks (4.24). *R1725: Super with Arguments* issue has the highest standard residual among refactoring suggestions and is more frequent in ML than in non-ML notebooks. It recommends against using the `super()` function with explicit arguments in a method and instead uses `super()` with no arguments. This is because using explicit arguments can make the code more brittle and harder to maintain, as it requires knowledge of the superclass' implementation details. In ML code, it is common to define custom classes and override superclass methods to implement specific functionality, such as training or predicting with a model. This often requires calling the superclass methods using `super()` and leads the occurrence of *R1725* to be much more frequent in ML notebooks.

The distributions of all four studied categories of code style issues are significantly different across ML and non-ML notebooks. Package or library handling-related style issues are much more prevalent in ML than in non-ML notebooks.

VII. Implications

A. Implications for PEP-8 code style developers

Our results show that all the studied PEP-8 code style issue categories significantly differ between ML and non-ML notebooks. Several individual code style issues have large differences and are more prevalent in ML than non-ML notebooks, e.g., *R1725: super-with-arguments*, *E0401: import-error* and *E1101: no-member*. There are two possible explanations for these differences: (1) ML developers care less about code style than non-ML developers, or (2) not all PEP-8 style guidelines are suitable for ML code in notebooks. Since the first reason is unlikely, we argue that some PEP-8 style guidelines may need to be adapted to better fit ML code in notebooks. A natural consequence of this observation is that we may need different PEP-8 style guidelines for different types of code. We encourage researchers to further investigate which types of code could benefit from updated PEP-8 style guidelines. The following is an example of a guideline that could be improved to better fit ML code in notebooks.

***R1725:super-with-arguments*:** This code style issue has the largest difference in occurrence between ML and non-ML notebooks (see Table V). Our observation indicates that this refactoring-type code style issue is 30 times more prevalent in ML (5.70%) than in non-ML notebooks (0.19%). One possible reason is that, in machine learning code, it is common to use deep learning frameworks such as PyTorch or TensorFlow that involve a large amount of inheritance and multiple levels of subclassing. In such cases, it is important to use `super()` with explicit arguments to avoid ambiguity in method resolution order (MRO) and ensure the correct order of inheritance. When we use `super()` without explicit arguments, it automatically looks for the following method in the MRO, which may not be what we intend and could lead to unexpected behavior and errors in our code. Therefore, using `super()` with explicit arguments in machine learning code is more common to ensure that the intended method is called in the correct order of inheritance. In fact, the official TensorFlow documentation includes explicit arguments when calling `super()`.[2] These examples demonstrate that the circumstances for triggering the *R1725:super-with-arguments* code style issue need to be adapted for ML notebooks.

B. Implications for researchers

We revealed that package handling-related issues such as *E0401: import-error* are more prevalent in ML notebooks (see Section VI-C1 and Table II). As researchers, when we run *Pylint* in a large-scale study, we generally do not restore the execution environment of a project or notebook before analyzing it (since *Pylint* is a static analysis tool). This is not really a problem for non-ML code, but as our results show, ML code is much more affected by such package handling style issues. Therefore, future studies should consider including a tool such as SnifferDog [44] to restore the execution environment of every notebook before analyzing it.

VIII. Threats to Validity

A. Internal Validity

Validity of static analysis: Static analysis tools like *Pylint* analyze code without executing it. However, Wang et al. showed that many Jupyter notebooks are difficult to reproduce due to issues with their dependencies [44]. We observed that the code style issues that differed the most between ML and non-ML notebooks mostly had to do with dependencies as well. Therefore, the choice of not executing the notebooks or at least restoring their execution environments may have impacted the results. Tools such as SnifferDog [44] can help to restore the execution environments of notebooks at a large scale. However, such tools come with their own limitations which may affect the study results as well. Future studies should consider replicating the study in an environment that allows the notebooks to execute.

Validity of *Pylint*: We used *Pylint* for identifying code style issues in the studied notebooks since SE researchers widely use this tool to lint Python code [33], [37], [41]. Other tools for identifying code style issues in Python, such as *Pyflakes* [1], exist. Future studies should investigate whether our findings hold for the results of other code style detection tools.

One might argue that some of the results from *Pylint* are false positives. For example, one might argue that the often-occurring import error (*E0401*) stems from an incorrectly configured execution or *Pylint* environment, rather than actual low quality code. We used the default *Pylint* and notebook configuration for our experiments, as this is the most representative of what developers and users will run into when using notebooks from others (which is one of the most common use cases of notebooks, as explained by Kery et al. [21]).

Validity of ML notebooks identification: We follow the repository filtering strategy of Biswas et al. [10] to identify the ML notebooks in our study. In that paper, they filtered their repositories that use data science libraries in the import statements. Here, we prepared a list of ML libraries[3] and checked whether they are used in the import statements. If yes, we labelled the notebook as ML; otherwise we labelled them as non-ML. This process did not check whether any features from the imported ML packages were called during the notebook execution.

Validity of notebook code quality calculation: An internal threat to the validity of how we compute notebook code quality is that Errors are weighted five times as heavy as other types of issues. The way to compute the notebook code quality score was a design decision by the *Pylint* developers [5], and we decided to use the same weighting to allow future researchers to compare their results with ours without having to make changes to *Pylint*.

B. External Validity

Threats to the external validity of our study concern the generalizability of the results. We focused on Python notebooks. Even though 93% of the computational notebooks are written in Python [31], future studies should investigate whether our results hold for other programming languages. Finally, we focused on notebooks from *Kaggle*, an online community platform for publicly sharing notebook code. As a consequence, our results apply to notebooks on Kaggle only. Future studies should investigate whether our results apply to notebooks from other platforms, such as GitHub.

IX. Conclusion

In this paper, we conducted a large-scale empirical investigation on code style issues in $246,599$ open-source Jupyter notebooks written in Python. In particular, we investigated whether lower code quality in notebooks is correlated with the usage of machine learning functionality. We grouped the dataset into $177,252$ ML notebooks (71.9%) and $69,347$ non-ML notebooks (28.1%) based on the usage of at least one machine learning library in a notebook. We studied the notebook's code quality based on PEP-8 code standards and explored how code style issues differ between ML and non-ML notebooks. The most important findings of our study are:

- ML notebooks tend to have lower code quality in terms of PEP-8 code standards than non-ML notebooks.
- All types of studied code style issue categories occur significantly more frequently in ML notebooks.
- Package or library handling-related style issues are much more prevalent in ML than in non-ML notebooks.

We conclude that the usage of ML functionality is an important reason for low quality code (based on the PEP-8 standard) in notebooks. However, the differences between code style issues in ML and non-ML notebooks also suggest that the PEP-8 standard may not always be the most suitable for ML code in notebooks. Hence, we recommend that future studies investigate how existing style guidelines can be improved to better fit code inside ML notebooks.

[2]https://www.tensorflow.org/guide/keras/custom_layers_and_models#calling_the_super_constructor

[3]https://github.com/saeedsiddik/NotebookCodeStyleIssue/blob/main/data/ml_python_package.txt

References

[1] "Pyflakes: Project description," Nov. 2022. [Online]. Available: https://pypi.org/project/pyflakes/

[2] "Pylint features," Dec. 2022. [Online]. Available: https://pylint.pycqa.org/en/latest/technical_reference/features.html

[3] "Cramẽr's v," Jan. 2023. [Online]. Available: https://www.ibm.com/docs/en/cognos-analytics/11.1.0?topic=terms-cramrs-v

[4] "Pep 8 – style guide for Python code," Jan. 2023. [Online]. Available: https://peps.python.org/pep-0008/

[5] "Pylint code rating," Jan. 2023. [Online]. Available: https://pylint.pycqa.org/en/latest/faq.html#pylint-gave-my-code-a-negative-rating-out-of-ten-that-can-t-be-right

[6] "Pylint message categories," Jan. 2023. [Online]. Available: https://pylint.pycqa.org/en/latest/messages/messages_introduction.html

[7] "Tiobe index for february 2022," Feb 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/

[8] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[9] N. Bafatakis, N. Boecker, W. Boon, M. C. Salazar, J. Krinke, G. Oznacar, and R. White, "Python coding style compliance on stack overflow," in *16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 210–214.

[10] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan, "Boa meets python: A boa dataset of data science software in python language," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 577–581.

[11] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[12] H. Cramér, *Mathematical methods of statistics*. Princeton university press, 1999, vol. 26.

[13] S. Dasgupta and S. Hooshangi, "Code quality: Examining the efficacy of automated tools," in *23rd Americas Conference on Information Systems (AMCIS)*. Association for Information Systems, 2017, pp. 1–6.

[14] H. Dong, S. Zhou, J. L. Guo, and C. Kästner, "Splitting, renaming, removing: A study of common cleaning activities in Jupyter Notebooks," in *9th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 2021, pp. 1–6.

[15] D. Goodger, "Code like a Pythonista: Idiomatic Python," 2007. [Online]. Available: https://david.goodger.org/projects/pycon/2007/idiomatic/handout.html

[16] M. E. Gorelli, "nbqa," https://github.com/nbQA-dev/nbQA, 2022.

[17] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A large-scale comparison of Python code in Jupyter Notebooks and scripts," in *18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022, pp. 1–12.

[18] S. J. Haberman, "The analysis of residuals in cross-classified tables," *Biometrics*, pp. 205–220, 1973.

[19] H. Jebnoun, H. Ben Braiek, M. M. Rahman, and F. Khomh, "The scent of deep learning code: An empirical study," in *17th International Conference on Mining Software Repositories (MSR)*. IEEE, 2020, pp. 420–430.

[20] H. Jebnoun, M. S. Rahman, F. Khomh, and B. A. Muse, "Clones in deep learning code: what, where, and why?" *Empirical Software Engineering*, vol. 27, no. 4, p. 84, 2022.

[21] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–11.

[22] D. E. Knuth, "Literate programming," *The computer journal*, vol. 27, no. 2, pp. 97–111, 1984.

[23] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[24] V. Liermann, "Overview machine learning and deep learning frameworks," *Springer Books*, pp. 187–224, 2021.

[25] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[26] A. Nikanjam and F. Khomh, "Design smells in deep learning programs: an empirical study," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 332–342.

[27] S. Omari and G. Martinez, "Enabling empirical research: A corpus of large-scale Python systems," in *Proceedings of the Future Technologies Conference*. Springer, 2019, pp. 661–669.

[28] J. Patra and M. Pradel, "Nalin: learning from runtime behavior to find name-value inconsistencies in Jupyter Notebooks," in *44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1469–1481.

[29] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[30] J. M. Perkel, "Why Jupyter is data scientists computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

[31] F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "Understanding and improving the quality and reproducibility of Jupyter Notebooks," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–55, 2021.

[32] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter Notebooks," in *16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 507–517.

[33] L. Quaranta, F. Calefato, and F. Lanubile, "Kgtorrent: A dataset of Python Jupyter Notebooks from Kaggle," in *18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 550–554.

[34] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and cohen's d indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.

[35] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[36] D. Sharpe, "Chi-square test is statistically significant: Now what?" *Practical Assessment, Research, and Evaluation*, vol. 20, no. 1, p. 8, 2015.

[37] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, and R. Vasa, "A large-scale comparative analysis of coding standard conformance in open-source data science projects," in *14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[38] C. Stephen, "Top programming languages 2022," Aug 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages/#toggle-gdpr

[39] P. Subotić, L. Milikić, and M. Stojić, "A static analysis framework for data science notebooks," in *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 13–22.

[40] S. Titov, Y. Golubev, and T. Bryksin, "Resplit: improving the structure of Jupyter Notebooks by re-splitting their cells," in *2022 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2022, pp. 492–496.

[41] B. van Oort, L. Cruz, M. Aniche, and A. Deursen, "The prevalence of code smells in machine learning projects," in *1st Workshop on AI Engineering-Software Engineering for AI*. IEEE, 2021, pp. 1–8.

[42] G. van Rossum and B. Warsaw, "Style guide for Python," in *Pro Python*. Springer, 2010, pp. 283–297.

[43] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, "Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks," *ACM Transactions on Computer-Human Interaction*, vol. 29, no. 2, pp. 1–33, 2022.

[44] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter Notebooks," in *43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1622–1633.

[45] J. Wang, K. Tzu-Yang, L. Li, and A. Zeller, "Assessing and restoring reproducibility of Jupyter Notebooks," in *35th International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 138–149.

[46] J. Wang, A. Zeller, and L. Li, "Better code, better sharing: On the need of analyzing Jupyter Notebooks," in *42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 53–56.

[47] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[48] A. X. Zhang, M. Muller, and D. Wang, “How do data science workers collaborate? roles, workflows, and tools,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. CSCW1, pp. 1–23, 2020.

[49] H. Zhang, L. Cruz, and A. van Deursen, “Code smells for machine learning applications,” in *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI (CAIN)*, 2022, pp. 217–228.