A Comparison of Machine Learning Code Quality in Python Scripts and Jupyter Notebooks*

Kyle Adams[1] and Aleksei Vilkomir[2] and Mark Hills[3]
[1]Moravian University
Bethlehem, PA
adamsk04@moravian.edu
[2]Department of Computer Science
East Carolina University
Greenville, NC
vilkomira21@ecu.edu
[3]Department of Computer Science
Appalachian State University
Boone, NC
hillsma@appstate.edu

Abstract

Jupyter notebooks are currently one of the most popular environments for Python development, especially in domains such as data science. Existing studies have shown that notebooks may promote bad coding habits, leading to poor code quality and challenges with replicating notebook results. In this paper, we compare the code quality of Python machine learning code found in Jupyter notebooks to that found in regular Python scripts. The goal of this work is to better understand how the machine learning code created in Jupyter notebooks differs both from machine learning code provided in scripts and from the larger body of Python code, with the aim of creating tools to better support both data science students and practitioners.

*Copyright ©2023 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

1 Introduction

According to a study in 2018 [4], Python is the most used language for machine learning on GitHub, as well as the third most used programming language on GitHub overall. One of the most popular environments for Python programming is Jupyter notebooks [16]. Jupyter notebooks provide a literate programming environment, where Python code can be combined with descriptive text and computational results. These results can be in text, but formats such as images and even video are also supported. While Jupyter notebooks can be used for a wide variety of different application domains, they were created with a focus on supporting applications in data science and scientific computing.

With the growth of interest in data science, millions of Jupyter notebooks have been created and shared on sites such as GitHub and Kaggle, a website that hosts data science competitions. Recent work (see Section 2) has explored issues with the quality of the code in these notebooks. For instance, the work of Grotov et al. [5] compared the quality of the code found in Jupyter notebooks to that of Python scripts. This comparison was done across all domains, while noting that results may differ if focused on a specific domain. In this paper, we narrow this prior comparison to focus specifically on machine learning code, while also comparing our results to those found in this earlier work.

The rest of this paper is organized as follows. First, Section 2 discusses related work. Section 3 then details the corpus of Python scripts and Jupyter notebooks used to conduct this study. Section 4 describes the process used to extract metrics and style errors from the code and to compute the results reported in Section 5. Section 5 evaluates our initial results, including a comparison to results found in earlier work. Lastly, Section 6 concludes the paper and describes possible future applications of our results. All code and data sets used in this paper are available in a replication package on Zenodo [29].

2 Related Work

Some of the prior work on Jupyter notebooks focuses on how notebooks are used in practice, including for collaboration and reproducibility. Kery et al. [11] interviewed 21 professional data scientists to learn more about how they use Jupyter notebooks, and also conducted a survey of 45 data scientists to learn more about potential tool support for allowing data scientists to interact with a history of notebook changes. Chattopadhyay et al. [3] used a combination of a survey and interviews to identify "pain points" encountered by users of computational notebooks, including Jupyter notebooks. Pimentel at al. [14] studied the use of both good and bad coding practices in Jupyter notebooks, with a focus on the impact of these practices on reproducibility. They used the results of this study to recommend best practices for developing notebooks. Pimintel et al. [15] then extended this work with additional analyses and the introduction of a tool, Julynter, that suggests notebook modifications for improving reproducibility. Quaranta et al. [17] also focused on best practices, specifically for collaboration. Wang et al. [25] looked at notebook documentation, evaluating a system named Themisto that automatically generates notebook documentation based on best practices gleaned from a collection of 80 top-rated notebooks from Kaggle. Additional work on collaboration with notebooks includes that of Zhang et al. [30], which explored collaborative roles and practices; and Rule et al. [20], which evaluated the impact of a cell folding extension to Jupyter notebooks on notebook understandability and reuse. Beyond this, papers by Rule et al. [19] and Amershi et al. [1] focused on how notebooks are used for different purposes and on how teams collaborate on machine learning tasks.

Other prior work focuses specifically on tools and analyses for understanding and improving the use of notebooks. Kery and Meyers [10] described and evaluated Verdant, a system for versioning notebook artifacts. Head at al. [6] described an analysis that uses program slicing over execution logs to compute notebook dependencies, along with *code gathering* tools that track notebook history. Wenskovitch et al. [27] designed a visualization tool, Albireo, that allows for multiple levels of visualization as well as exploration of cell dependencies. Wang et al. [26] focused on the need for improved tools for analyzing Jupyter notebooks, showing that notebooks include substantial amounts of poor-quality code. Yang et al. [28] used code summarization techniques to show the impact of *data wrangling* code by showing the effect of the code on the data being analyzed by the notebook. Venkatesh and Bodden [23] presented an analysis and tool that generates cell headers for notebook cells based on the position of the cell in a broader machine learning workflow. Titov et al. [21] described an analysis and tool, ReSplit, that refactors existing notebook cells. Jiang et al. [8] presented an algorithm for generating a dependency graph between cells in a Jupyter notebook and labeling these cells with the relevant machine learning state, based on the API calls used in the cell. Quaranta et al. [18] introduced Pynblint, a linter for Jupyter notebooks that checks for violations of best practices and provides fix recommendations.

Grotov et al. [5] conducted a comparison of Python code in Python scripts and in Jupyter notebooks. Scripts and notebooks were compared using a combination of structural metrics (e.g., SLOC, cyclomatic complexity) and code style violations. The corpus included all Jupyter notebooks publicly available on GitHub, as well as all Python scripts from the 10,000 most starred Python repositories on GitHub, narrowed to use only those notebooks and scripts released under a permissive software license. This resulted in 847,881 notebooks and 465,776 scripts. To detect style errors, the authors used Hyperstyle [2], a tool for detecting style violations in student programming assignments. A second tool, named Matroskin, was created to work with notebooks, including to compute structural metrics. One specific style issue, that of excessive imports, was also noted by Vilkomir [24]. Our work builds upon, and is compared with, the earlier work of Grotov et al. and Vilkomir, focusing on differences in code quality between scripts and notebooks used specifically for machine learning.

3 Corpus

To carry out our comparison, a corpus of Python scripts and Jupyter notebooks, focused specifically on machine learning, was needed. Kaggle [9] is a website dedicated to machine learning and data science. It holds data science competitions, some with cash prizes, and makes some of the solutions (those the authors allow to be released publicly) available for public inspection and download. Some of the published solutions are created using Jupyter notebooks, while some are created using Python scripts. Some are also created using other languages (e.g., R), but those are ignored here.

To get the initial corpus of data, we used Meta Kaggle [13]. Meta Kaggle is a public dataset, published by Kaggle, that contains information about competitions and the public solutions posted as part of these competitions. Using this data set, a Python script was used to extract the information needed to download each solution. The projects were sorted by the user rating (stars) they received on Kaggle in descending order. We decided to limit our corpus to the top 100,000 projects. Next, the Kaggle API was used to download the project code corresponding to each project. This download process was executed as part of a Bash script for repeatability. The result of executing this script was a collection of Python script files, Jupyter notebooks, and potentially other files that may have been included with the solution.

Using the Kaggle API, a total of 69,858 machine learning files were obtained. Only files written in Python (either notebooks or scripts) were processed further. These files were split into two datasets, one for Python scripts, and one for notebooks. In total, there are 12,136 Python scripts (with a .py extension), and 57,722 Jupyter notebooks (with a .ipynb extension) in these two datasets. The number of files in the final corpus is different from the initial size we planned to obtain because some files were unavailable for download.

4 Methodology

We have used a methodology that mirrors that found in the paper by Grotov et al. [5] to allow for a comparison of our results, focused just on machine learning code, with their results, which looked at notebooks and scripts in general (including machine learning code). This includes using and reporting the same structural metrics they used over their dataset.

Matroskin [12, 5] is a library designed for analyzing Juptyer notebooks. It focuses on computing structural metrics for the Python code included in the notebooks. Hyperstyle [2, 7] is a tool that is able to analyze code for style errors. It works for code written in Python, Java, or Kotlin, and can follow different style guides, such as PEP-8 [22]. To compute metrics over the code in our corpus, several scripts were developed to automate the use of Hyperstyle and Matroskin. These scripts extract the results of running these tools and aid in the analysis of the metrics and style errors. Other scripts were developed to convert between notebook and script formats, which allowed the scripts (after conversion to notebook format) to be processed using Matroskin and the notebooks (after conversion to script format) to by analyzed using Hyperstyle.

4.1 Matroskin Methodology

Matroskin was developed to handle a large number of notebooks at once, therefore we were able to run the tool on our entire corpus of notebooks by running it on the directory containing our dataset of downloaded notebooks. The script that runs the tool is a variant of a script provided with Matroskin, edited in order to output the results in a format more amenable to further processing. Results of running the script were stored in a single JSON file for each processed notebook. To avoid problems with name collisions in cases where multiple notebooks used the same file name, the results were stored in files that were each given a unique number (e.g., 0.json, 1.json), with a separate CSV file mapping from the original file name of the processed notebook to the relevant JSON file containing the analysis results.

This process was then repeated on the dataset of Python scripts. The analysis first converted each script into a notebook with a single cell containing the script code. The same process described previously was then used to process these converted files: information on each notebook was extracted and saved into a JSON file, with information to link these files back to the related notebook (and thus, the original script).

A separate script was created to analyze the results stored in these JSON files. This script reads the results saved in the JSON files, computing the results shown in Section 5. During this process, we could not process 385 of the JSON files generated for the notebooks and 154 of the JSON files generated for the scripts due to an error in the result format. This error appears to be in the information generated by Matroskin, but further work is needed to determine the actual source of the error.

4.2 Hyperstyle Methodology

Hyperstyle was run individually on each Python script in the corpus, with results output to individual files for later processing. A bug in Hyperstyle, which we have not yet isolated, would cause it to get stuck at random points during the analysis if run repeatedly on a sequence of input scripts. This required us to process these files outside of the script we developed to automate this process. In this way, it was possible to process all scripts in the dataset. While Hyperstyle also includes an option to process an entire directory, the information returned by Hyperstyle in that case differs from the per-file results, generally reporting many fewer style issues. The process used for the scripts was then repeated on the dataset of notebooks. A script was used to first convert each notebook into a script file, which was then processed by Hyperstyle.

A separate analysis script then processes the Hyperstyle result files, generating the results seen in Section 5. During processing, for the results from the collection of scripts, one result file was unable to be processed due to a UnicodeDecoder Error and thirteen results files were unable to be processed due to syntax errors. These are based on characteristics of the original analyzed files. For the results from the collection of notebooks, three results files were unable to be processed due to syntax errors.

5 Evaluation

In this section, we present the results of both the structural and style analyses described in Section 4. For the structural results, the mean (written as M) and standard deviation (written as STD) are shown. Using the mean of the results allows a better comparison of the two datasets due to their difference in size.

5.1 Evaluation of Structural Metrics

The results of comparing the structural metrics between scripts and notebooks can be found in Table 1. We found that the notebooks (M = 174.48, STD = 216.45), on average, contained more source lines of code than the scripts (M = 105.02, STD = 180.11). Notebooks (M = 27.93, STD = 48.51) and scripts (M = 26.96, STD = 57.13) have a similar number of commented lines unless markdown cells ("extended comments", which may be actual comments on the code or descriptive text to be shown as part of the notebook) are included, in which case notebooks (M = 75.40, STD = 113.45) are shown to have almost three times as many comment lines as scripts.

While they tend to have fewer lines of code, the scripts (M = 12.00, STD = 23.71) are shown to have a higher cyclomatic complexity as compared to notebooks (M = 6.18, STD = 9.28). Also, scripts (M = 1.00, STD = 0.24) have,

Table 1: Structural Metrics, Notebooks and Scripts

Metric	Notebook Mean (STD)	Script Mean (STD)
SLOC	174.48 (216.45)	105.02 (180.11)
Comments LOC	27.93 (48.51)	26.96 (57.13)
Extended Comments LOC	75.40 (113.45)	N/A (N/A)
Blank Lines Count	30.42 (46.97)	32.53 (46.69)
Cyclomatic Complexity	6.18 (9.28)	12.00 (23.71)
NPAVG	0.71 (0.27)	1.00 (0.24)
API Functions Count	6.95 (6.69)	10.20 (10.06)
API Functions Uses	13.17 (20.24)	23.29 (81.93)
Defined Functions Count	3.45 (6.00)	3.39 (6.86)
Defined Functions Uses	5.83 (13.31)	4.72 (12.91)
Built in Functions Count	4.83 (3.30)	3.80 (3.20)
Built in Functions Uses	22.51 (35.88)	14.78 (25.26)
Other Functions Uses	86.02 (99.11)	28.73 (54.03)
Cell Coupling	48.44 (358.84)	N/A (N/A)
Function Coupling	10.49 (101.02)	12.59 (105.80)

on average, a higher number of inputs per function (NPAVG) than notebooks ($\mathsf{M} = 0.71$, $\mathsf{STD} = 0.27$). When it comes to function usage, scripts ($\mathsf{M} = 10.20$, $\mathsf{STD} = 10.06$) include more unique API function imports than notebooks ($\mathsf{M} = 6.95$, $\mathsf{STD} = 6.69$). Scripts ($\mathsf{M} = 23.29$, $\mathsf{STD} = 81.93$) also use those API functions more than the notebooks ($\mathsf{M} = 13.17$, $\mathsf{STD} = 20.24$). For built-in functions, notebooks ($\mathsf{M} = 4.83$, $\mathsf{STD} = 3.30$) take advantage of a larger number of different functions than the scripts ($\mathsf{M} = 3.80$, $\mathsf{STD} = 3.20$). The notebooks ($\mathsf{M} = 22.51$, $\mathsf{STD} = 35.88$) also call the built-in functions much more than scripts ($\mathsf{M} = 14.78$, $\mathsf{STD} = 25.26$). When it comes to user-defined functions, notebooks ($\mathsf{M} = 3.45$, $\mathsf{STD} = 6.00$) have about the same amount as scripts ($\mathsf{M} = 3.39$, $\mathsf{STD} = 6.86$). For number of uses of the user-defined functions, notebooks ($\mathsf{M} = 5.83$, $\mathsf{STD} = 13.31$) again have more uses than scripts ($\mathsf{M} = 4.72$, $\mathsf{STD} = 12.91$). For any other functions, the number of uses in notebooks ($\mathsf{M} = 86.02$, $\mathsf{STD} = 99.11$) is significantly greater than in scripts ($\mathsf{M} = 28.73$, $\mathsf{STD} = 54.03$). Lastly, coupling (use of shared/common elements) between functions is greater in scripts ($\mathsf{M} = 12.59$, $\mathsf{STD} = 105.80$) than notebooks ($\mathsf{M} = 10.49$, $\mathsf{STD} = 101.02$) unless we consider cells to be functions (cell coupling), in which case coupling is much higher in notebooks ($\mathsf{M} = 48.44$, $\mathsf{STD} = 358.84$), showing how connected code in notebooks is, even though it is less complex.

Comparing with Prior Results: While most of the metrics computed by Grotov et al. [5] (referenced just as "prior results" below) have been nor-

Table 2: Top Style Errors, Notebooks and Scripts.

Error Code	Error Description	Category	Notebooks %	Scripts %
W0611	Import module or variable is not used	Best Practices	41	64
W0621	Redefining name from outer scope	Best Practices	22	27
W0404	Re-imported module	Best Practices	16	13
W0612	Unused variable name	Best Practices	8	14
W0613	Unused argument	Best Practices	4	9
C0411	Import order not followed	Code Style	38	57
C0412	Imports not grouped by package	Code Style	21	19
C0305	Trailing newlines	Code Style	20	15
W0301	Unnecessary semicolon	Code Style	7	4
W0311	Bad indentation	Code Style	5	11
E1101	Variable accessed for nonexistent member	Error Proneness	47	59
E0001	Syntax error	Error Proneness	47	3
W0104	Statement seems to have no effect	Error Proneness	33	6
E0611	No name in module	Error Proneness	29	54
W0106	Expression is assigned to nothing	Error Proneness	9	1

malized to the source lines of code, we can still compare certain metrics as well as general trends. We found that both notebooks and scripts in our data contain more SLOC on average (64 for notebooks, 23 for scripts) than those from the prior results. The results for functions are very similar except for built-in function uses. In our corpus, notebooks have a higher mean number of uses compared to scripts, while the prior results show both having a similar number. Interestingly, our results show both notebooks and scripts having higher cyclomatic complexity than the prior results, although in both cases scripts are more complex than notebooks.

5.2 Evaluation of Style Metrics

Next we compare the style metrics gathered using Hyperstyle. Although the notebooks had a much higher number of source lines of code, they have a much lower average of warnings per file—for notebooks, a mean of 14.21, versus a mean of 23.77 for scripts. In order to evaluate the warnings, we have taken the ones that appear most frequently from three different categories and found the percentage of scripts and notebooks in which they appear in each dataset. The results of this are found in Table 2. Note that the error codes are based on the codes used in Pylint, a popular linting tool for Python.

Of the top five issues in Best Practices, two are related to imports and two are related to variables. For these four, there is a higher frequency of errors in scripts than in notebooks. For both issues with imports, W0611 (import module or variable is not used) and W0404 (re-imported module), it is possible they are caused because of how the code is created, with developers copying code from existing samples and pasting it (including imports) in to their own code. Error code W0621 (redefining name from outer scope) is the issue of duplicate variable names that refer to values in different scopes, while errors W0612 (unused variable name) and W0613 (unused argument) are related to unused variables and parameters. Again, these issues may be created when code is copied in to a program and then edited, keeping declarations of variables and/or parameters but removing lines of code including the uses.

Of the top five issues in Code Style, this time three are more frequent in notebooks while two are more common in scripts. Also, two of them again have to do with imports, while the other two deal with spacing. Both C0411 (import order not followed) and C0412 (imports not grouped by package) are problems with the ordering of imports. PEP8 has a specific import standard which expects imports to be ordered in a certain way and grouped together. This can be an issue if imports are not added until they are needed, and therefore get added to the bottom of the import list. For spacing, issue C0305 (trailing newlines) and issue W0311 (bad indentation) do not impact correctness, although W0311 could lead to maintenance issues later since bad or irregular indenting can lead to confusion about the expected behavior of the code. The last code style issue is W0301 (unnecessary semicolon). Semicolons are rarely used in Python code, but this could imply that the writer of the file has experience in writing in other languages where semicolons are commonly used.

In the last category, Error Proneness, we once again see that three of the issues are more frequent in notebooks while the other two are more frequent in scripts. The most frequent issue for both sets of files is E1101 (variable accessed for nonexistent member). This error can indicate a bug in the code, but can also indicate that the dependencies that provide this member are not available. This may also account for the high frequency of code E0611 (no name in module). E0001 (syntax error) is the code given for syntax errors. This issue is significantly more frequent in notebooks than scripts. This could be due to how code is executed in specific parts of a notebook, meaning that some cells could contain syntax errors. The following issue, W0104 (statement seems to have no effect) could have a similar problem, where the statement does have an effect but only for the user using the notebook (e.g., to display a value). The last issue is W0106 (expression is assigned to nothing). This only occurs in 1% of the scripts as opposed to 9% of the notebooks.

Comparing with Prior Results: Comparing with the prior results, we share 1 style error in the top 5 errors for the Error-Proneness category, 0 style errors in the Code Style category, and 2 style errors for the Best Practices category. Of the 15 style errors from the prior results, 13 of them appear in a higher percentage of notebooks than scripts, while this is only true for 7 out of the 15 of our style errors. Of the 3 shared errors, the prior results show all 3 appearing in a higher percentage of notebooks, however, in our results, only 1 of them appears in a higher percentage of notebooks. Studying the reason for these differences is part of our future work.

6 Conclusions and Future Work

In this paper, we presented an analysis of Python scripts and Jupyter notebooks, comparing structural and style metrics extracted from both. The results show that Jupyter notebooks are generally larger, but less complex. They tend to use built-in functions much more than scripts and have a higher coupling rate. Scripts averaged more style issues per file than notebooks. For the fifteen issues chosen, the scripts had the issues appear more frequently for eight, while the notebooks had the issues appear more frequently for seven.

For future work, we would like to expand the machine learning corpus to include code from other sources beyond Kaggle (e.g., through detecting use of ML APIs in code on GitHub). We would also like to explore what aspects of the code are leading to the differences between ML code (specifically) and Python code (in general) that we reported in Section 5. Beyond this, we also believe it is important to create new analysis tools, including linters/style checkers, that are specifically aimed at notebooks created for machine learning tasks. These tools could better support both students and practitioners, giving targeted advice that would make sense given the medium (notebooks), the domain, and the domain's structural and style characteristics. Given the results, tools that would help to integrate existing code examples (such as snippets of code from other samples, from API documentation, or from Stack Overflow) seem particularly important. These could help to keep a consistent set of properly-ordered imports while avoiding the re-declaration of existing variables or the inclusion of unused variables and/or code, and could ensure consistent spacing based on current style conventions. Navigation and code inspection tools that can help developers navigate the many API calls they use in notebooks could also be important, especially for newer developers learning how to use machine learning APIs. Finally, tools that can detect unused code in notebooks, and that can illustrate dependencies between code blocks, could be valuable for students trying to understand existing notebooks or developing their own.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant CNS-2050883.

References

[1] Saleema Amershi et al. “Software Engineering for Machine Learning: A Case Study”. In: *Proceedings of ICSE-SEIP 2019.* 2019, pp. 291–300.

[2] Anastasiia Birillo et al. “Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments”. In: *Proceedings of SIGCSE 2022.* ACM, 2022, pp. 307–313.

[3] Souti Chattopadhyay et al. “What’s Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities”. In: *Proceedings of CHI 2020.* ACM, 2020, pp. 1–12.

[4] Thomas Elliott. *The State of the Octoverse: Machine Learning.* Jan. 2019. URL: `https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/`.

[5] Konstantin Grotov et al. “A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts”. In: *Proceedings of MSR 2022.* ACM, 2022, pp. 353–364.

[6] Andrew Head et al. “Managing Messes in Computational Notebooks”. In: *Proceedings of CHI 2019.* ACM, 2019, pp. 1–12.

[7] *Hyperstyle.* URL: `https://github.com/hyperskill/hyperstyle`.

[8] Yuan Jiang, Christian Kästner, and Shurui Zhou. “Elevating Jupyter Notebook Maintenance Tooling by Identifying and Extracting Notebook Structures”. In: *Proceedings of ICSME 2022.* IEEE, 2022.

[9] *Kaggle.* URL: `https://www.kaggle.com/`.

[10] Mary Beth Kery and Brad A. Myers. “Interactions for Untangling Messy History in a Computational Notebook”. In: *Proceedings of VL/HCC 2018.* 2018, pp. 147–155.

[11] Mary Beth Kery et al. “The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool”. In: *Proceedings of CHI 2018.* ACM, 2018, pp. 1–11.

[12] *Matroskin: A library for the large scale analysis of Jupyter notebooks.* URL: `https://github.com/JetBrains-Research/Matroskin`.

[13] *Meta Kaggle.* URL: `https://www.kaggle.com/datasets/kaggle/meta-kaggle`.

[14] João Felipe Pimentel et al. “A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks”. In: *Proceedings of MSR 2019.* 2019, pp. 507–517.

[15] João Felipe Pimentel et al. "Understanding and improving the quality and reproducibility of Jupyter notebooks". In: *Empirical Software Engineering* 26.4 (2021), pp. 1–55.

[16] *Project Jupyter.* URL: `https://jupyter.org/`.

[17] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. "Eliciting Best Practices for Collaboration with Computational Notebooks". In: *Proc. ACM Hum.-Comput. Interact.* 6.CSCW1 (Apr. 2022).

[18] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. "Pynblint: a Static Analyzer for Python Jupyter Notebooks". In: *Proceedings of CAIN 2022.* ACM, 2022, pp. 48–49.

[19] Adam Rule, Aurélien Tabard, and James D. Hollan. "Exploration and Explanation in Computational Notebooks". In: *Proceedings of CHI 2018.* ACM, 2018, pp. 1–12.

[20] Adam Rule et al. "Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018).

[21] Sergey Titov, Yaroslav Golubev, and Timofey Bryksin. "ReSplit: Improving the Structure of Jupyter Notebooks by Re-Splitting Their Cells". In: *Proceedings of SANER 2022.* 2022, pp. 492–496.

[22] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8—Style Guide for Python Code.* 2001. URL: `https://peps.python.org/pep-0008/`.

[23] Ashwin Prasad Shivarpatna Venkatesh and Eric Bodden. "Automated Cell Header Generator for Jupyter Notebooks". In: *Proceedings of AISTA 2021.* ACM, 2021, pp. 17–20.

[24] Aleksei Vilkomir. *An Empirical Exploration of Python Machine Learning API Usage.* East Carolina University, 2020.

[25] April Yi Wang et al. "Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks". In: *ACM Trans. Comput.-Hum. Interact.* 29.2 (Jan. 2022).

[26] Jiawei Wang, Li Li, and Andreas Zeller. "Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks". In: *Proceedings of ICSE-NEIR 2020.* ACM, 2020, pp. 53–56.

[27] John Wenskovitch et al. "Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure". In: *Proceedings of VDS 2019.* 2019, pp. 1–10.

[28] Chenyang Yang et al. “Subtle Bugs Everywhere: Generating Documentation for Data Wrangling Code”. In: *Proceedings of ASE 2021.* 2021, pp. 304–316.

[29] *Zenodo Artifact for A Comparison of Machine Learning Code Quality in Python Scripts and Jupyter Notebooks.* URL: `https://dx.doi.org/10.5281/zenodo.8122385`.

[30] Amy X. Zhang, Michael Muller, and Dakuo Wang. “How Do Data Science Workers Collaborate? Roles, Workflows, and Tools”. In: *Proc. ACM Hum.-Comput. Interact.* 4.CSCW1 (May 2020).