Automated Cell Header Generator for Jupyter Notebooks

Ashwin Prasad Shivarpatna Venkatesh
Heinz Nixdorf Institut
Paderborn University
Paderborn, Germany
ashwin.prasad@upb.de

Eric Bodden
Heinz Nixdorf Institut
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

ABSTRACT

Jupyter notebooks are now widely adopted by data analysts as they provide a convenient environment for presenting computational results in a literate-programming document that combines code snippets, rich text, and inline visualizations. Literate-programming documents are intended to be computational narratives that are supplemented with self-explanatory text, but, recent studies have shown that this is lacking in practice. Efforts in the software engineering community to increase code comprehension in literate programming are limited. To address this, as a first step, this paper presents a prototype Jupyter notebook annotator, *HeaderGen*, that automatically creates a narrative structure in notebooks by classifying and annotating code cells based on the machine learning workflow. *HeaderGen* generates a markdown cell header for each code cell by statically analyzing the notebook, and in addition, associates these cell headers with a clickable table of contents for easier navigation. Further, we discuss our vision and opportunities based on this prototype.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation.*

KEYWORDS

static analysis, literate programming, python, jupyter notebook

ACM Reference Format:

Ashwin Prasad Shivarpatna Venkatesh and Eric Bodden. 2021. Automated Cell Header Generator for Jupyter Notebooks. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis (AISTA '21), June 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3464968.3468410

1 INTRODUCTION

Literate programming is a programming paradigm of interleaving code snippets with visualization and explanatory text [4]. Jupyter notebook is an open-source literate-programming tool for developing interactive documents that augment executable code snippets with rich text. Jupyter notebooks are evolving as the de-facto tool for data analysts and machine learning experts [7]. Jupyter notebooks are used as a "detailed lab notebook", where, the exploratory process of extracting insights from the underlying data is detailed with accompanying descriptive text [3]. Literate programming plays an important role in sharing and reproducibility of computational results, especially in platforms such as Kaggle,[1] where the data science community comes together to share and learn.

Jupyter notebooks are composed of a sequence of *cells.* These cells can be of three types: code, markdown, or raw. Code cells are used to write executable code snippets. Markdown cells contain rich text, where, the markdown language can be used to format explanatory text. Raw cells are for rendering different code formats into HTML or LaTeX. Figure 1 presents a simple Jupyter notebook containing three code cells, three markdown cells which are rendered HTML H3 header text, and an inline visualization which is the output of the last code cell. Note that a raw cell is not shown here.

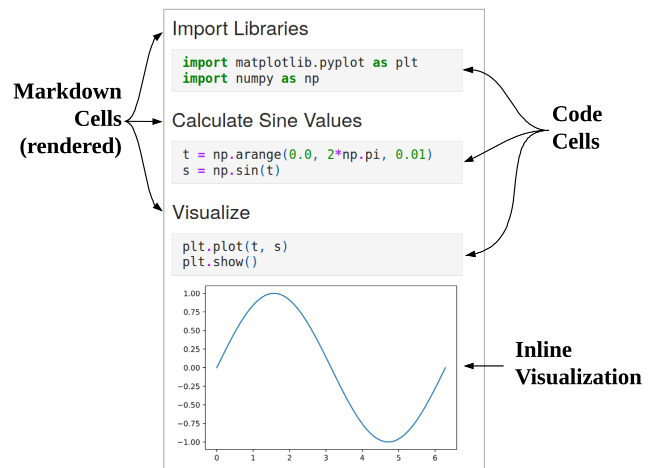

Figure 1: A simple Jupyter notebook

Jupyter notebooks are widely used as walk-through guides for beginners, especially in platforms such as Kaggle. To cultivate best practices in learners it is therefore important to also follow these practices and to maximize code comprehension in such environments.

However, studies have shown that this is not the case and data analysts tend to sway away from literate programming principles:

1) Wang et al. [12], found that even notable Jupyter notebooks (i.e., advocated by the Jupyter team) contain signs of poor coding practices such as unused variables and deprecated APIs.

[1] https://www.kaggle.com/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AISTA '21, June 12, 2021, Virtual, Denmark
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8541-1/21/06…$15.00
https://doi.org/10.1145/3464968.3468410

2) Rule et al. [9], in a series of large scale study and interviews, reported that analysts describe their notebooks to be "messy" and needed further refactoring and annotation to be presentable to others. Furthermore, authors note analysts are using markdown headers to organize the structure of their notebooks. However, a quarter of the notebooks they studied had no markdown cells, showing a lack of narrative structure.

3) Kery et al. [3], interviewed 21 data scientists to study literate programming behavior. Authors found that the notebooks are often structured mainly through cell structure rather than using descriptive markdown cells to create a narrative.

4) Pimentel et al. [8], in a large scale study of 1.4 million notebooks, found that notebooks often contained out-of-order cells which makes reproducibility difficult and hampers the narrative structure of notebooks. In addition, their study found that more than 90% of the markdown cells in notebooks contained headers (H1, H2, and H3). This shows that analysts are using headers in markdown cells for creating a narrative structure.

While these earlier studies highlight the need for tool support in improving literate programming practices, efforts to improve coding practices, code comprehension, and bug detection in Jupyter notebooks are still in the early stages.

To this end, this paper presents a proof-of-concept (POC) Jupyter notebook annotator called *HeaderGen* for automatically generating descriptive markdown header cells by statically analyzing the notebook. Header cells are annotations to the respective code cells that follow and in addition these header cells are associated with a clickable *Table of Contents* at the top of the notebook. Although headers are not a direct replacement for detailed descriptions, they can enable easy navigation and improve code comprehension. Moreover, several other capabilities can be realized based on the analysis information gathered by *HeaderGen*. While Jupyter notebooks support many programming languages, we target Python, considering that it is used in the vast majority of notebooks [9]. We implement a static analyzer for identifying all function calls in a notebook. The analyzer resolves each call to the target function's definition. For instance, *plt.plot* is recognized as *matplotlib.pyplot.plot* and therefore a function of the library *matplotlib*. *HeaderGen* generates an appropriate cell header based on the utility of these function calls.

In this work, we present: 1) an approach to automatically annotate Jupyter notebooks based on static analysis results and 2) our vision for advancing the development of tools to support data scientists, specifically tools that improve code comprehension, bug detection, and enforce best practices in Jupyter notebooks.

2 PROOF OF CONCEPT: ANNOTATOR

Machine learning (ML) based data analysis is carried out in sequential phases, often with feedback loops between phases [1]. Although several workflows exist in the literature, we present a common and simplified workflow for solving ML problems as shown in the Figure 2. Data analysts tend to form narratives in their notebooks based on this workflow. This is evident in platforms such as Kaggle.

A notebook presenting a solution to an ML problem typically consists of three phases.

1) *Data Phase:* where the dataset is ingested, validated, and preprocessed according to the context of the problem.

2) *Model Phase:* where a specific model is trained on a clean training dataset generated in the previous phase. Furthermore, the predictions of the model are typically evaluated using a separate test dataset.

3) *Post Development Phase:* finally, the trained model is prepared for deployment in an external application or simply saved for use in a different context.

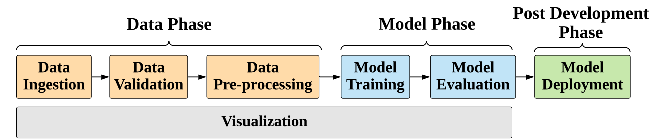

Figure 2: A simplified workflow of solving ML problems

As a first step towards bringing tool support to improve the coding practices of data analysts, *we implement a Jupyter notebook annotator called HeaderGen that associates code cells with the ML phases. HeaderGen* annotates notebooks with markdown cells based on the ML phases and provides an overview of the notebook by generating a clickable *Table of Contents* at the top of the notebook.

Cell headers are generated in two steps. First, function calls in each code cell are identified using static analysis methods. Then, these function calls are matched against a pre-built *function-to-ML-phase* mapping. We briefly discuss this two-step process as follows.

2.1 Identifying Function Calls

Statically identifying function calls in a dynamic programming language like Python is challenging. For Python, precise call graph generation is an evolving topic and existing call graph generation algorithms for Python are not practical for use in our analysis [6]. Fortunately, our requirement is the identification of all call sites (i.e., where a function call exists) and determining the library it belongs to, hence, a complete call graph is not required for our analysis. Yet, currently, no tool exists that can generate call site information for a Jupyter notebook. Therefore, as a starting point for our POC, we implement a static analyzer for identifying call sites in Jupyter notebooks. *HeaderGen* can identify call sites and trace the function calls back to their library of origin. To do this, a Jupyter notebook is taken as an input and first converted into a native Python script. Next, the Abstract Syntax Tree (AST) of the Python script is generated using the built-in AST library. This is followed by an iterative analysis of the AST in two steps.

First, the AST is analyzed to gather information about: 1) Imports, i.e., names of the imported third-party libraries and 2) Variable aliases, i.e., different identifiers referring to the same underlying variable. Then, the import and alias information is used to resolve the function call to the library where it is defined (see Figure 3).

The operational sequence of *HeaderGen* is listed as follows:

(1) **Analyze import statements**
Python allows multiple ways to import libraries and also provides ways to import libraries with an alias. Import nodes in the AST are analyzed to gather the import information.

(2) **Analyze assignment statements**
Assignment nodes in the AST are analyzed to generate alias information for all variables.

(3) **Separate Library Calls**
In the second iteration, every Call node in the AST is visited and the function calls are categorized into built-in, user-defined and library functions. The line number of the function is also stored for mapping it back to its cell.

(4) **Recognize Library**
Using import and alias information, function calls are resolved back to its library.

(5) **Tag ML Phase**
Library calls are matched with the *function-to-ML-phase* mapping to find the appropriate tag.

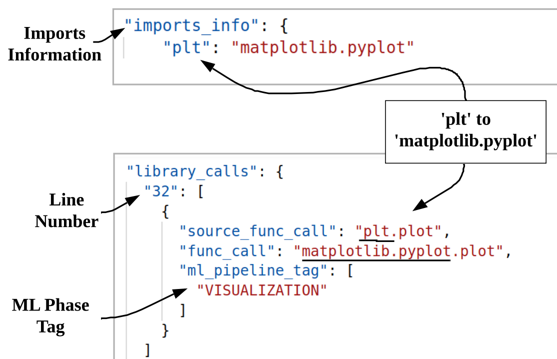

Figure 3: Analysis information of an example notebook

Figure 3 shows a section of the analysis information of a real-world Jupyter notebook. Here, the original call *plt.plot*, at line number 32, is recognized as *matplotlib.pyplot.plot* based on the import information gathered in the first iteration of the analysis.

We note that *HeaderGen* does not perform well with several advanced Python features such as `eval` functions, higher-order functions, and situations where type information is lacking due to dynamic typing in Python. However, our POC is still in the early stages and new research in this direction is promising — PyCG [10], Google's Pytype,[2] and Facebook's Pyre[3].

2.2 Library Mapping

For the current POC we have built a *function-to-ML-phase* mapping for a few popular machine learning libraries. To establish the mapping, we manually inspected the respective official API documentation. In cases where the classification is uncertain, or when a function can be used in multiple phases, such functions are mapped to multiple phases by process of elimination. For instance, a call to the Keras API "*keras.layers.LSTM*" is mapped to a single phase as "`MODEL TRAINING`", whereas, a call to the Numpy API "*numpy.array*" is mapped as "`DATA INGESTION, DATA PREPROCESSING`". All the phases shown in Figure 2 are considered in the mapping.

[2]https://github.com/google/pytype
[3]https://github.com/facebook/pyre-check

```
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

(a) Code cell before analysis

DATA PREPROCESSING (2)

```
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

(b) Code cell annotated with cell header after analysis

Figure 4: Example of an annotated code cell

Table of Contents

- IMPORTS (9) | CONFIGURATION (1)
- FUNCTION DECLARATION (2)
- DATA INGESTION (1)
- DATA PREPROCESSING (2)
- MODEL TRAINING (12)
- MODEL EVALUATION (2)
- VISUALIZATION (7)

Figure 5: Table of contents generated at the top of notebooks

2.3 Notebook Annotation

The analysis is not performed directly on the Jupyter notebook, instead, before the analysis, notebooks are first converted into native Python scripts. Therefore, additional helper functions are implemented to: 1) convert between Jupyter notebook and Python script, 2) map line numbers in Python scripts back to its respective notebook cell, 3) generate table of contents, and 4) add analysis results as markdown cells into the notebook.

Figure 4 shows a section of the same notebook before and after the analysis. Here, a code cell is annotated with a markdown cell as a header. The number inside the brackets represents the number of calls for a particular tag, for instance, "`DATA PREPROCESSING (2)`" means two function calls associated with data pre-processing is identified in the following code cell. Figure 5 shows the table of contents that is generated at the top of a notebook. The contents of the notebook can be navigated using the hyperlinked sections.

3 FUTURE WORK

HeaderGen is only a first demonstration of the possibilities of using static analysis techniques to bring tool support to literate programming. We therefore list some ideas for future direction in this area.

Boilerplate Notebook Generation. Code generators exist for other domains, for instance CogniCrypt [5] for cryptography. However, nothing equivalent exists for ML literate programming. We suggest that a code generator for various ML models is worthy to be explored. Such a generator can support data analysts by preparing a boilerplate notebook that is structured according to the ML workflow with markdown headers and table of contents.

Cell Restructuring. Cells cluttered with functions from multiple ML phases can be flagged for refactoring, such that individual cells are logically separated to aid code comprehension in spirit of literate programming.

Automated API Mapping. ML libraries are continuously evolving and new libraries and APIs are being developed regularly. Therefore, manually mapping each of them to ML phases is not advisable. Natural language processing techniques can be explored to automate *function-to-ML-phase* mapping.

Phase Targeted Feedback. Along with boilerplate notebook generation, the generated notebook can be followed by continuous static analysis. One can use the analysis results to nudge analysts in the right direction by giving real-time targeted feedback based on the ML phase the analyst is currently working on. For instance, if an analyst is writing a custom pre-processing function for a specific model, a feedback can be presented with a list of already available pre-processing functions for that specific model.

Phase Based Notebook Mining. Another potential area that can be explored is code mining for ML notebooks. This can be useful for analysts looking for code examples for a specific model and library. The ML workflow can be exploited here as well, for instance, a query language can be developed to enable listing of all code cells of a specific ML phase. For instance, listing all data pre-processing cells of notebooks implementing a certain ML model.

Teaching Framework. Considering the wide adoption of Jupyter notebooks for teaching, a framework for teaching ML programming by keeping the learner in check using static analysis can be effective. Experts can define a high-level protocol for implementing a specific model (e.g., API patterns). Then, the learner is actively encouraged to follow a specific pattern of API usage, followed up with real-time feedback as the learner tackles individual ML phases.

Bug Detection for Deep Learning. In the last 5 years an array of empirical studies have investigated the programming challenges faced by analysts. The most recent work by Humbatova et al. [2], presents a taxonomy of deep learning faults faced by analysts. Tool support to tackle faults based on this taxonomy can be explored.

Dynamic and Static Analysis. Rule et al. [9] found from a sample of 1 million notebooks that the median notebook contains 85 lines of code. Considering the small size of notebooks, a hybrid approach combining dynamic and static analysis can be explored.

4 RELATED WORK

Research at the intersection of static analysis and literate programming seems to be lacking. The only other relevant work that directly targets Jupyter notebooks is by Wang et al. [11]. The authors implement a tool called *Osiris*, which attempts to improve reproducibility by recognizing and satisfying dependencies between code cells. Osiris provides all possible execution orders that reproduce the original notebook results.

On the other hand, multiple studies have indicated that experts in the ML community lack sufficient knowledge of software engineering and code-quality principles. Wang et al. [12] recently highlighted the need for research in the direction of improving software quality and reliability in the literate-programming world, specifically for Jupyter notebooks. Kery et al. [3] and Rule et al. [9] investigated coding behavior in Jupyter notebooks. Kery et al. [3] found that a narrative structure is often created using cell structure rather than with explanatory markdown cells. Rule et al. [9] found that analysts are willing to spend time documenting their notebooks but note that analysts would benefit from tool support for structuring and annotation.

5 CONCLUSION

While the software engineering community has been investigating the challenges faced by data scientists in literate programming, tool support to mitigate these challenges is lacking. To this end, in this preliminary work, we make an argument for bringing static analysis based tool support to literate programming by building *HeaderGen*, a proof-of-concept Jupyter notebook annotator. *HeaderGen* can automatically annotate notebooks by adding markdown cells to Jupyter notebooks. We further put forward our vision and opportunities for further development of our annotator.

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Montreal, QC, Canada, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[2] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[3] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. 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 (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[4] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Jan. 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[5] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 931–936.

[6] Sriteja Kummita, Goran Piskachev, Johannes Späth, and Eric Bodden. 2021. Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In *2021 International Conference on Code Quality (ICCQ)*. 1–15. https://doi.org/10.1109/ICCQ51190.2021.9392986

[7] Jeffrey M. Perkel. 2018. Why Jupyter Is Data Scientists' Computational Notebook of Choice. *Nature* 563, 7729 (Oct. 2018), 145–146. https://doi.org/10.1038/d41586-018-07196-1

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

[9] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. 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, 1–12. https://doi.org/10.1145/3173574.3173606

[10] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, Spain, 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[11] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 138–149. https://doi.org/10.1145/3324884.3416585

[12] Jiawei Wang, Li Li, and Andreas Zeller. 2020. 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, 53–56. https://doi.org/10.1145/3377816.3381724