Elevating Jupyter Notebook Maintenance Tooling by Identifying and Extracting Notebook Structures

Yuan Jiang

CMU-CS-22-123

July 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christian Kästner, Carnegie Mellon University, Institute for Software Research
Eunsuk Kang, Carnegie Mellon University, Institute for Software Research
Shurui Zhou, University of Toronto, Department of Electrical & Computer Engineering

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Abstract

Data analysis is an exploratory, interactive, and often collaborative process. Computational notebooks have become a popular tool to support this process, among others because of their ability to interleave code, narrative text, and results. The exploratory nature of computational notebooks allows their users to edit and execute parts of their program in any order. However, notebooks in practice are often criticized as hard to maintain and being of low code quality, including problems such as unused or duplicated code and out-of-order code execution. Data scientists can benefit from better tool support when maintaining and evolving notebooks. We argue that central to such tool support is identifying the structure of notebooks. We present a lightweight and accurate approach to extract notebook structure and outline several ways such structure can be used to improve maintenance tooling for notebooks, including navigation and finding common structural patterns. In addition, we investigate the history of notebooks and extend our approach to visualize how notebooks evolve over multiple revisions. We measure statistics of changed, added, and removed cells in Kaggle notebooks with history versions. Our formative study shows our visualizations can be useful for tracing and understanding changes in notebook evolution and identifying alternatives explored in specific stages of a data analysis pipeline over notebook histories.

July 31, 2022
DRAFT

Acknowledgments

I would like to thank my advisor Professor Christian Kästner for his invaluable guidance and support for this project and my thesis. Thank you for introducing me to this project in 2020, encouraging me to move forward with my research these past two years, and challenging me to become a better researcher.

I would like to thank Professor Shurui Zhou for her continuing help and encouragement throughout this project. Thank you for always being there when I have concerns or questions.

I would like to thank Professor Eunsuk Kang for joining my thesis committee and providing valuable feedback for my thesis.

I would also like to thank everyone who has given me help or encouraged me over the course of this project in any possible way.

Last but not least, I would like to thank my parents and friends for their enormous support during the pursuit of my master degree at Carnegie Mellon University.

Contents

List of Figures

July 31, 2022
DRAFT

List of Tables

July 31, 2022

DRAFT

Chapter 1

Introduction

Data science is a field that extracts insights from data and applies these insights across a broad range of applications. With the abundance of available data and the success of machine learning, the community of data science practitioners is ever-growing [14]. Data science work is usually exploratory and iterative, and often collaborative [7, 10, 11]. *Computational notebooks* enable their users to interleave code, visualizations, and narrative texts in a single document [11]. The exploratory nature of *computational notebooks* allows data scientists to write and refine code easily with the ability to execute parts of their code in any order and view the computational results immediately [4]. *Computational notebooks* have become the primary coding environment for data scientists, with thousands of papers and millions of data science notebooks shared publicly each year [14]. Despite that *computational notebooks* are a widely-used tool for data scientists, many problems still occur, including poor code quality [20], which undermines understanding and collaboration among notebook users. Our project aims to provide better tool support for notebook users to maintain and evolve notebooks. We implement an efficient and accurate approach to identify and extract notebook structures and extend our work to investigate notebook history. For each part, we outline how our work can be useful for improving maintenance tooling for notebooks.

1.1   Motivation

While computational notebooks are very popular among data scientists, many practitioners and researchers report problems [2, 4]. Previous work examining millions of notebooks and dozens of interviews has shown that many notebooks are "messy" and most contain minimal to no documentation and structuring (in markdown cells) that could facilitate easy understanding [2, 13]. Understanding is essential for collaboration, reuse, and maintenance though. Poor quality code in public notebooks makes them unreliable for inexperienced learners who might use these notebooks as tutorials [20]. Common problems manifest in dead-ends, duplicated code, and tangled or scattered code [19]. For example, one study shows that many notebooks contain unused variables – those defined but never reused [20], while another research finds that most notebook cells are presented in a different order than they are executed and code related to a result is often scattered across many distant cells [4]. These issues make it difficult for notebook users to or-

ganize or refactor their code in a reusable fashion [20]. Not only are "messy" notebooks hard to understand, they are also difficult to reproduce. Because notebook users can execute any part of their code at any time, notebooks are criticized for unexpected execution order and bad practices in modularizing code [13]. Our goal is to make it easier to build tooling that helps notebook practitioners understand, navigate, modularize, and maintain notebook code, even across a series of history revisions.

1.2 Thesis Statement

I develop an efficient and light-weight approach to identify and extract Jupyter notebook structures as labeled dependency graphs using static analysis to find data dependencies between cells and a heuristics-based approach to label code cells with machine learning pipeline stages to improve maintenance tooling for notebooks. I extend this algorithm to merge structures of different revisions of a notebook into one labeled dependency graph to help understand changes and alternatives in notebook histories.

1.3 Our Contribution

We lay the foundation for maintenance tooling with an efficient algorithm to identify and extract Jupyter notebook structures as labeled dependency graphs, as summarized in Figure 1.1. We automatically label each notebook cell with machine learning (ML) stages (e.g., data collection, training, evaluation) and extract data dependency relations among the cells. Each labeled node in the output graph represents a code cell, and every directed edge represents a def-use relation between a pair of cells. In our evaluation, we show that our approach is accurate and very lightweight, outperforming prior approaches in terms of lower complexity, higher accuracy, and lower execution time. We discuss potential tooling based on our labeled dependency graph by sketching a navigation tool and reporting structural patterns commonly found in notebooks.

We extend our algorithm above to visualize how notebooks evolve over time, by merging structures of a series of notebook versions as one directed, labeled dependency graph, as an example shown in Figure 1.2. In this graph, each node represents a set of cells with the same content from consecutive notebook versions, each cluster represents a set of nodes which represent the same cell with slightly different contents, and each edge represents a data dependency relation between a pair of nodes. Nodes are also labeled by the ML stages inherited from the cells they represent. Our visualizations can be useful for understanding and navigating through notebook histories, and learning how notebook users explore alternatives in specific stages of an ML pipeline.

The **contributions** of this thesis include the following:
- An efficient and light-weight algorithm to identify and extract Jupyter notebook structures as directed, labeled dependency graphs. Data dependencies between cells are identified using static analysis, and notebook cells are automatically labeled with ML stages that fit best on their source code. Evaluation shows 75% accuracy for cell labelings and practical

July 31, 2022
DRAFT
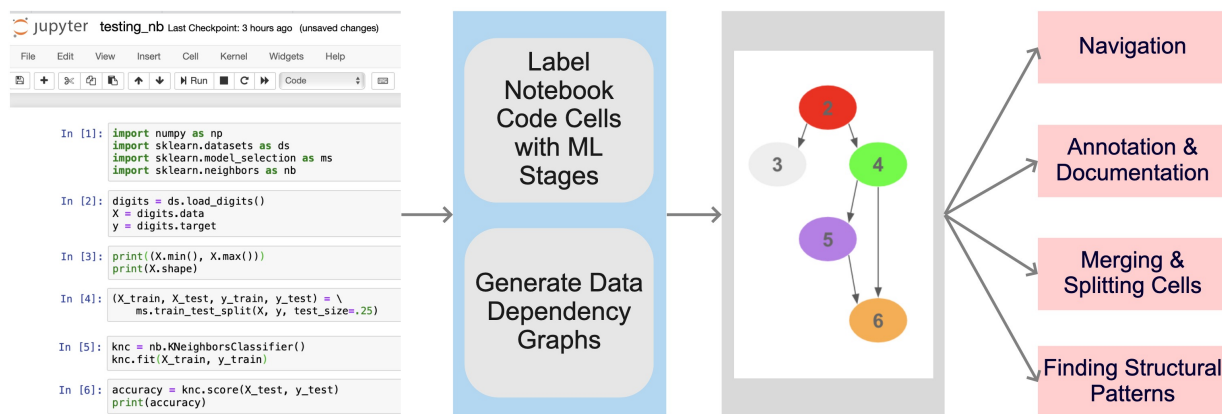
Figure 1.1: Summary of Our Workflow: a Jupyter notebook is passed to our algorithm, which labels notebook cells with ML stages and generates a data dependency graph. The labeled dependency graph can be useful for applications such as navigation, annotation and documentation generation, merging and splitting cells, and finding structural patterns in a notebook dataset.

performance to run our methods in the background of notebook maintenance tooling. We discuss potential applications of our approach.

- Manual labelings of 50 notebooks with ML stages that are considered ground truth.

- Comparison with two prior approaches about the accuracy and runtime of cell labelings. Results show that our methods are not only more light-weight, but also more accurate.

- An approach to merge structures of different notebook revisions as a directed, labeled dependency graph. Our analysis generates statistics of changed, added, and removed cells over a notebook's history of many versions, while our formative study shows our methods could be useful for understanding and navigating through changes in notebook evolution and identifying alternatives explored in specific ML stages.

We make all implementation code and manually labeled data publicly available. [1]

1.4 Thesis Outline

This thesis is outlined as follows. In Chapter 2, we discuss the background of Jupyter notebooks and related work of our project. In Chapter 3, we present our approach and evaluation of identifying and extracting Jupyter notebook structures as labeled data dependency graphs, and outline several examples of envisioned tooling using our algorithm. We describe our solution of merging notebook structures and visualizing notebook histories in Chapter 4, and discuss some of its anticipated applications in notebook maintenance tooling. We discuss future work in Chapter 5 and give our conclusion in Chapter 6.

---

[1]https://github.com/cindyyuanjiang/Jupyter-Notebook-Project

Version 1

```
import sklearn
import sklearn.datasets as ds
import sklearn.model_selection as ms
import sklearn.neighbors as nb

digits = ds.load_digits()
X = digits.data
y = digits.target

(X_train, X_test, y_train, y_test) = \
    ms.train_test_split(X, y, test_size=.25)

knc = nb.KNeighborsClassifier()
knc.fit(X_train, y_train)

y_predicted = knc.predict(X_test)
print(y_predicted[:10])
print(y_test[:10])
```

Version 2

```
import sklearn
import sklearn.datasets as ds
import sklearn.model_selection as ms
import sklearn.neighbors as nb

digits = ds.load_digits()
X = digits.data
y = digits.target

(X_train, X_test, y_train, y_test) = \
    ms.train_test_split(X, y, test_size=.25)

knc = nb.KNeighborsClassifier()
knc.fit(X_train, y_train)

y_predicted = knc.predict(X_test)
print(y_predicted[:10])
print(y_test[:10])
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X_train, y_train)

y_predicted = clf.predict(X_test)
print(y_predicted[:10])
print(y_test[:10])
```

Version 3

```
import sklearn
import sklearn.datasets as ds
import sklearn.model_selection as ms
import sklearn.neighbors as nb

digits = ds.load_digits()
X = digits.data
y = digits.target

(X_train, X_test, y_train, y_test) = \
    ms.train_test_split(X, y, test_size=.25)
print(X_train[0:10])

from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X_train, y_train)

y_predicted = clf.predict(X_test)
print(y_predicted[:10])
print(y_test[:10])
```

Version 4

```
import sklearn
import sklearn.datasets as ds
import sklearn.model_selection as ms
import sklearn.neighbors as nb

digits = ds.load_digits()
X = digits.data
y = digits.target

(X_train, X_test, y_train, y_test) = \
    ms.train_test_split(X, y, test_size=.25)
print(X_train[0:10])

from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X_train, y_train)

params = clf.best_params_
print(params)
```

Legend:
- Data Collection
- N/A
- Data Wrangling
- Training
- Evaluation

Figure 1.2: This is a graph which merges 4 versions of a notebook. Nodes in the graph are labeled with ML stages and colored accordingly. The clusters in the graph are outlined by boxes. Every node contains a starting and ending version number and a cell number (e.g., "V1-2[3]" represents Cell 3 from Version 1 to 2), representing the cells with the same content across all versions in between the starting and ending versions inclusively. Edges represent data dependency relations among nodes.

July 31, 2022
DRAFT

Chapter 2

Background & Related Work

We present the background of Jupyter notebook and discuss related work of our project in this chapter. We study previous analysis and tooling which aim to improve Jupyter notebooks, and focus on prior work addressing problems related to documentation, managing variants and revisions, and structuring notebook code in a more orderly fashion. Understanding which ML pipeline stages each notebook cell is working on is an important part of our methods. We find and discuss two prior approaches of labeling notebook cells using different techniques.

2.1 Jupyter Notebook

A computational notebook is an interactive literate programming document which is executed in the computational environment; Python notebooks in the Jupyter environment are the most popular of these [13]. Literate programming refers to the concept of combining code and natural language which allows programmers to express their thoughts behind the logic of a program [7]. An interactive computational notebook environment allows code parts, known as cells, to be executed incrementally to produce immediate results and visualizations.

A Jupyter notebook consists of a sequence of cells, which can be code cells or markdown cells. Code cells contain executable Python source code while markdown cells explain the programmer's ideas behind the code logic [21]. Figure 2.1 illustrates an example of a notebook excerpt, which contains one markdown cell followed by two code cells with immediate results and visualization. The execution marks indicate the execution orders of code cells. Because notebook users are free to execute any cell at any time, the execution marks may not be in chronological order from top down, as shown in Figure 2.2.

2.2 Previous Analysis or Tools to Improve Jupyter Notebooks

Coding practices in notebooks and the popular computational notebook environments like Jupyter themselves have been studied extensively (e.g., [2, 7, 13, 16, 19]), revealing many poor practices and pain points that hamper understanding and maintenance. Many researchers have subsequently tried to address various problems through improved tooling.

Learning to recognize handwritten digits

with a K-nearest neighbors classifier

Markdown Cell

Data loading and visualization

Code Cell

```
In [9]:   from sklearn.datasets import load_digits
          digits = ds.load_digits()
          print(digits.data.shape)

          (1797, 64)
```

```
In [11]:  import matplotlib.pyplot as plt
          plt.gray()
          plt.matshow(digits.images[0])
          plt.show()

          <Figure size 432x288 with 0 Axes>
```
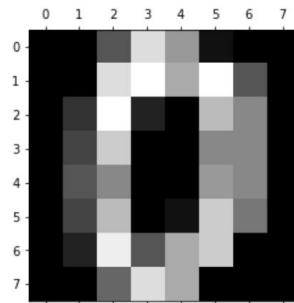
Figure 2.1: Jupyter Notebook Example

```
In [3]:   (X_train, X_test, y_train, y_test) = \
              ms.train_test_split(X, y, test_size=.25)
```

```
In [6]:   knc = nb.KNeighborsClassifier()
          knc.fit(X_train, y_train)
```

```
In [5]:   y_predicted = knc.predict(X_test)
          print(y_predicted[:10])
          print(y_test[:10])
```

Figure 2.2: Non-chronological Notebook Execution Marks Example

A common theme are attempts to improve documentation. Previous research examining one million open-source computational notebooks on Github has shown that one in four does not contain any sort of written documentation [16]. Documentation is critical for collaboration and understanding among data scientists though. Wang et al. [19] implemented a deep-learning-based automated documentation generation system with three distinct approaches: creating new documentation for source code, retrieving online API documentation for external libraries and

packages, and nudging users to write documentation. Yang et al. [26] used program synthesis techniques and dynamic program analysis to generate documentation for data wrangling code by summarizing data transformations on representative examples to help users with program understanding. Rule et al. [15] designed a Jupyter notebook extension for annotated cell folding which aids navigation and comprehension.

Other tooling focuses on managing variants and revisions: For example, Kery et al. [5] designed a local versioning plugin for Jupyter notebooks using algorithmic and visualization techniques to help data science practitioners better forage and understand their past analysis choices. Their tool provides light-weight support to quickly retrieve, trace or reproduce versions of specific artifacts, and to compare multiple versions of different artifacts. Head et al. [4] introduced code gathering tools to help data scientists clean and recover different versions of code in cluttered notebooks using software slicing. The tools highlight dependencies used to compute selected results and provide ordered, minimal, and complete code slices for them. In addition, the tools store past results and code slices which produce them so that users can find, explore, and compare different versions of their code. Weinman et al. [22] introduced a tool to support *forking* and *backtracking*, which allow users to create new interpreter sessions and navigate through previous execution states for exploring alternatives. The tool's design presents users multiple programming states side by side for the users to compare these results easily.

Finally, several papers focus on supporting data scientists with structuring their code into ordered cells: Titov et al. [17] proposed a heuristics-based algorithm for automatically resplitting (merging and splitting) cells into more semantically cohesive units. Wenskovitch et al. [23] designed a visualization tool to support communication and exploration by summarizing and displaying the relationships and dependencies between the cells of a notebook, using dynamic analysis.

Each of these tools developed custom infrastructure from scratch. We aim to encourage more maintenance tooling for notebooks by providing a common underlying analysis infrastructure that can extract the structure in a notebook effectively and efficiently.

2.3   Labeling Notebook Cells

In addition to dependencies between cells, it is often useful to understand what different parts of a notebook are doing. Data science code is often structured according to a conceptual *data science pipeline*, which starting from model requirements, considers data collection, data cleaning, data labeling, feature engineering, model training, model evaluation, and deployment [1], as shown in Figure 2.3. In notebooks, particularly data cleaning and feature engineering (collectively called data wrangling [26, 27]), model training, and model evaluation are common.

Understanding which pipeline stages correspond to which notebook cells can be helpful for various comprehension and maintenance tasks and is a core of our approach. Past approaches to identify stages either relied on very simple heuristics or relied heavily on expensive ML classification. On one end, Venkatesh et al. [18] simply labeled cells by API calls contained in them;
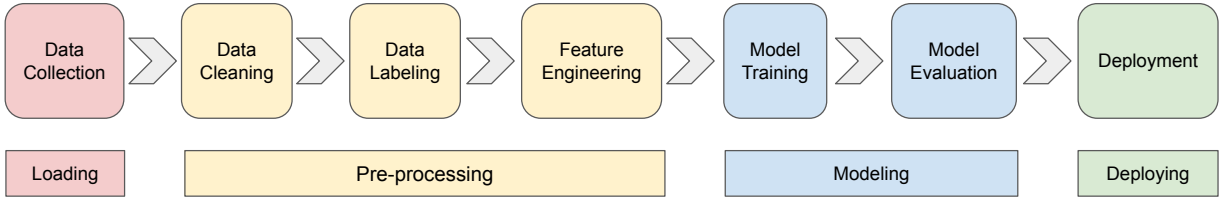
Figure 2.3: Data Science Pipeline

on the other end, Zhang et al. [27] used a weakly supervised transformer architecture to classify code snippets which jointly models data science code and natural language annotations. Our proposed work outperforms both of these approaches, providing labels accurately and fast.

Chapter 3

Generation of Labeled Dependency Graphs

To provide the foundation for more maintenance tooling for notebooks, to address their common "messy" and undocumented nature in an exploratory and iterative workflow, we develop an algorithm to identify and extract structures from Jupyter notebooks as directed, labeled dependency graphs where every node represents a code block (usually a notebook cell), every edge represents a data dependency relation between a pair of nodes, and nodes are labeled corresponding to their stages in the ML pipeline. In Figure 3.3, we illustrate the resulting output graph for an excerpt of a notebook. By default, we use notebook cells as the granularity for graph nodes because an ideal notebook cell can be viewed as a proto-function and reused to do one dedicated action [17]. To build the labeled dependency graph, we proceed in two steps: identifying dependencies between cells and mapping cells to ML stages.

3.1 Data Dependency

We used standard data flow analysis to identify def-use chains in a notebook's code. We then group these dependencies by code blocks (cells), representing dependencies among cells as directed edges in our graph.

We and others [13] found that most notebook code is fairly simple, hence even fairly simple and fast data-flow analysis provides accurate results (e.g., context sensitivity and pointer analysis add little benefit). We largely reused the static data-flow analysis from the *python-program-analysis* package developed by Microsoft [19] and modified it as follows: First, we did not track dependencies from import statements because they obfuscate the dependency graph without adding value for maintenance tasks. Second, we made the tool conservative with regards to dependencies resulting from function side effects, assuming that a function call might modify its arguments, therefore treating the function as a definition site of its arguments. We made the latter change in preferring occasional false positive dependencies between cells over missing edges or high analysis costs from inter-procedural analysis of library code.

3.2 Identifying ML Stages

We label each node in our dependency graph with a corresponding stage of the ML pipeline. Commonly, an ML pipeline consists of data collection, data cleaning, labeling, feature engineering, model training, and model evaluation [1]. As in prior work [26, 27], we combine data cleaning, labeling, and feature engineering collectively as data wrangling to avoid potential overlapping of their meanings. We define the most relevant stages – *Data Collection*, *Data Wrangling*, *Training*, *Evaluation*, and *Exploration* – with corresponding examples in Figure 3.1. Our labels are similar to prior classification by Zhang et al. [27] because they are all based on standard stages, but we do not include the *Import* stage because it is not very important from a maintenance perspective.

Human developers can map most cells clearly to one or multiple of these stages (as we will show in our evaluation). While investigating notebooks, we found that some cells may correspond to multiple stages, for example, both perform feature engineering and data exploration in the same cell. Usually though one stage is clearly the dominant purpose of a cell. To avoid the complexity of having multiple labels, we agreed on a priority order, assigning always the stage with the highest priority if multiple stages may apply. As sole exception, we introduce a dedicated label for cells that perform both training and evaluation, as they often co-occur and neither stage should be considered as subsumed by the other. Our final priority order is: *Training+Evaluation > Training* or *Evaluation > Data Collection > Data Wrangling > Exploration*. If a cell does not correspond to any stage above, we label it as *"N/A"*.

While there are several different strategies to identify stages for code fragments, we develop a simple but accurate heuristics-based approach that does not rely on textual documentation in the notebook and avoids computationally expensive and brittle ML techniques.

As recognized in prior work [14, 23], data science code often uses a small set of popular libraries for typical ML activities. We use knowledge about such APIs as the seed for our labels. We build an API-to-ML-stage mapping for commonly used ML libraries (currently *scikit-learn*, *Keras*, and *pandas*). We map API calls to specific stages by inspecting their functionalities in the respective official API references. To correctly distinguish API calls with the same name (e.g., *'fit'* used for *Training* in *KNeighborsClassifier* or used for *Data Wrangling* in *PCA* in *scikit-learn*), we use a type inference tool *pyright* [9] to identify which library class makes the API call and label it accordingly.

We use known APIs as seeds to identify stages for a cell and propagate information from there along data-flow edges. We found that identifying a cell's stage solely by API calls contained in the cell is insufficient, but that notebook users often put logically related statements in the same cell or structurally close to one another. We hence propagate information as follows: Every time we analyze a statement, we consider two scenarios based on the number of *child statements* it has according to the data-flow analysis.

- *One child statement*: if the current statement and its child statement are in the same code cell or the current statement is the closest parent to the child statement with regard to

their location in the source code, we propagate the current statement's labels to the child statement.

- *Multiple child statements*: for every pair consisting of the current statement and one of its child statements, we follow the same mechanism in the previous case.

After the algorithm traverses every data-flow edge between a pair of statements in the source code and propagates information along the edge using heuristics described above, it labels each cell with the highest-priority label existing in that cell. If none of the labels exist, the cell is labeled as *"N/A"*.

3.3 Evaluation

To be useful in tooling for maintenance and evolution, our labeled dependency graph needs to be *accurate* and *fast to compute*. Accuracy ensures our results provide valuable insights into Jupyter notebook structures, whereas the latter is important both when analyzing many notebooks (e.g., when indexing reusable structures for search) and when computing analyses in the background (e.g., within a notebook plugin).

First, we evaluate the accuracy of our algorithm, especially cell labelings. Second, we measure the performance of each step in our algorithm.

3.3.1 Dataset

We assemble a dataset of *all* public notebooks scraped from GitHub repositories created on two specific days, January 1 (704 notebooks) and January 6 (1629 notebooks), 2021. Both are weekdays, though January 1 is a holiday in many countries. We expect that January 1 skews more toward hobbyists whereas January 6 represents a more typical workday, together covering a comprehensive representative of notebooks on GitHub. We sample by release days rather than popularity to get a full cross section of notebooks typically published on GitHub; we evaluate on recent notebooks that represent the state of practice now, rather than performing longitudinal analysis of historic data.

3.3.2 Accuracy

To evaluate the correctness of the output data dependency graphs, we need to measure the accuracy of both the node labelings and the dependencies between cells.

Accuracy of Node Labelings. To evaluate accuracy, we need to establish ground truth of the correct label for each cell. We establish ground truth manually.

To assure reliability of manual labeling, we first created explicit labeling instructions and evaluated inter-rater agreement among three labelers. Specifically, two authors independently labeled 102 and 153 cells from two different sets of 6 notebooks randomly selected from our dataset, and

Stage Name	Definition/Description	Example
Data Collection	Data Collection cells load data that will often be passed to other stages in the notebook.	`import sklearn.datasets as ds` `digits = ds.load_digits()`

Data Wrangling	Data wrangling cells clean, transform, or filter data loaded from the data collection stage. These cells also perform feature engineering on the collected data.	`from sklearn.preprocessing import OneHotEncoder`
		`enc = OneHotEncoder(handle_unknown='ignore')`
		`enc.fit(data)`
		
		`from sklearn.decomposition import PCA`
		`pca = PCA(n_components=2)`
		`pca.fit(X)`

Training	Training cells define and fit supervised-learning ML models to the collected data to predict on some feature of the data.	`logreg = lm.LogisticRegression()` `logreg.fit(X_train, y_train)`

Evaluation	Evaluation cells predict a feature of some dataset using ML model(s) or measure/compute the accuracy or explanatory power of the model(s).	`y_predicted = logreg.predict(X_test)`
		`import sklearn.model_selection as ms`
		`ms.cross_val_score(logreg, X, y)`

Training+Evaluation	Training+Evaluation cells include both training and evaluation stages, as described above.	`logreg = lm.LogisticRegression()`
		`logreg.fit(X_train, y_train)`
		
		`y_predicted = logreg.predict(X_test)`
		`import sklearn.model_selection as ms`
		`ms.cross_val_score(logreg, X, y)`

Exploration	Exploration cells visualize, print or plot data or data's relevant information (e.g. shape) and plot or print results related to the training or evaluation process. Unsupervised learning is also classified as exploration.	`fig, ax = plt.subplots(1, 1, figsize=(8, 3))`
		`ax.imshow(np.vstack((y_test, y_predicted)),`
		`        interpolation='none', cmap='bone')`
		`ax.set_axis_off()`
		`ax.set_title("Actual and predicted survival outcomes "`
		`        "on the test set")`
		
		`from sklearn.cluster import KMeans`
		`import numpy as np`
		`X = np.array([[1, 2], [1, 4], [1, 0],`
		`        [10, 2], [10, 4], [10, 0]])`
		`kmeans = KMeans(n_clusters=2, random_state=0).fit(X)`

Figure 3.1: ML Stage Definitions and Examples

a third author independently labeled all 255 cells of these 12 notebooks, until each cell had two independent labels. We computed agreement with Cohen's kappa and discussed disagreements between raters. We then refined the instructions and repeated the process for 434 cells of another 11 randomly selected notebooks. After the second round, we reached a kappa score of 0.83, which is generally interpreted as almost perfect agreement [8], suggesting that manual labeling is indeed reliable.

After establishing reliability, we then manually label 1208 cells from 50 notebooks as ground

Table 3.1: Average runtime per notebook for each step in the methods.

Step in Methods	Avg. Runtime
Type Inference File Generation	3720 ms
Seed Function Identification & Data Flow Analysis	144 ms
Information Propagation & Labeled Graph Generation	187 ms

truth, 25 fresh notebooks randomly selected from each of the January 1 and January 6 datasets respectively. We run our algorithm on these notebooks and compare the results against experts' manual labels. Automated labels match our ground truth in 903 out of 1208 cells, for 75% accuracy (compared to 38% accuracy for a simple baseline predictor that always predicts the majority label *Data Wrangling*).

To better understand the sources of inaccuracy, we explored the confusion matrix, shown in Figure 3.2. Almost half of the errors (44%) come from mislabeling *Exploration* and *Data Wrangling* cells as *"N/A"*, and almost a quarter of the errors (23%) come from mislabeling *Exploration* as *Data Wrangling* or vice versa. Distinguishing among *Exploration*, *Data Wrangling*, and *"N/A"* could be difficult in some scenarios. For instance, some *Data Wrangling* processes are unidentified because they do not call any *Data Wrangling* APIs nor are they near them, thus require deeper understanding of the code's context to identify the stage. Another problem is that some cells make *Data Wrangling* API calls, but only intend to explore the data. We have not yet found a way to better distinguish these cases heuristically.

We compare our accuracy result against two previously discussed approaches on identifying ML stages for notebook cells. Venkatesh et al. [18] labeled cells solely by API calls contained in them, but they did not evaluate accuracy or release their implementation. We approximate their approach by running our implementation without type inference or information propagation along data-flow edges. This results in an accuracy of 69%, showing how our improvements correspond to a 19% reduction in error over merely identifying API calls within a cell. Zhang et al. [27] used ML techniques to classify code snippets based on content and context. We were unable to reproduce their results (we only achieved 12% accuracy replicating their methods on their dataset), but the paper reports 70% accuracy for a very similar task. This indicates that our *much simpler* approach can achieve a reduction in error of 17% over their reported numbers.

Accuracy of Cell Dependencies. Establishing ground truth for cell dependencies is tedious. We opted to not perform a systematic evaluation, but instead rely on manual inspection of analysis results in the sampled notebooks. We found occasional spurious edges from conservative assumptions in our analysis, but no substantial problems.
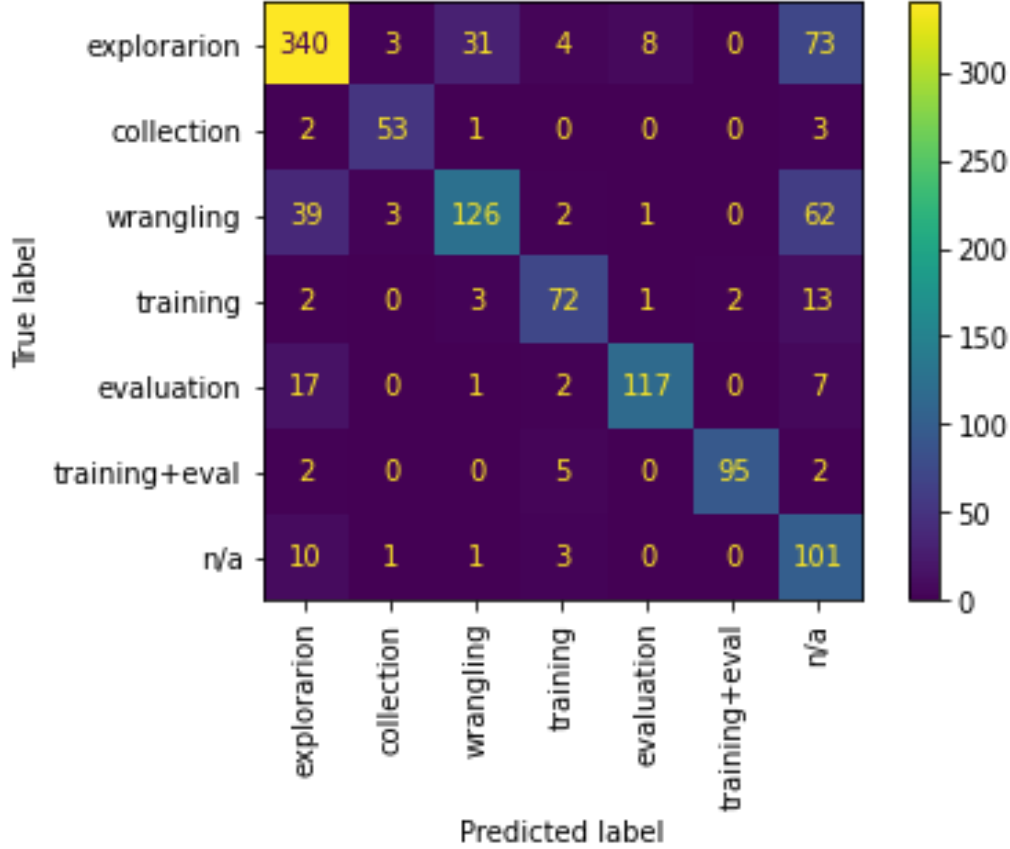
Figure 3.2: Confusion Matrix Between Manual and Algorithm's Labels

3.3.3 Performance

We measure execution times of our analysis using a commodity laptop (2.8 GHz Quad-Core Intel Core i7, Intel Iris Plus Graphics 655 1536MB, 16GB memory) for all 2333 notebooks in our dataset and report times separately for the three steps. The slowest component is *Type Inference File Generation* using the off-the-shelf tool *pyright* [9], which is used to disambiguate API calls with the same name. All other components can be executed in much under one second for almost all notebooks, see Table 3.1. Assuming type inference information can be cached (or improved with a different tool), the entire analysis for a full notebook can be performed in 331ms on average, fast enough to run in the background during interactive use.

Previous work by Zhang et al. [27] used an ML architecture to predict stages for notebook cells. The paper did not report any performance numbers. While we were not able to exactly replicate their approach due to limitation of our GPU support and did not receive access to their pretrained models, we could train a smaller model using the paper's script (at significant one time training cost). Even with the smaller model, label inference took 2007ms per notebook in the provided test dataset. That is, our much simpler (and more accurate) approach seems more feasible for interactive settings.
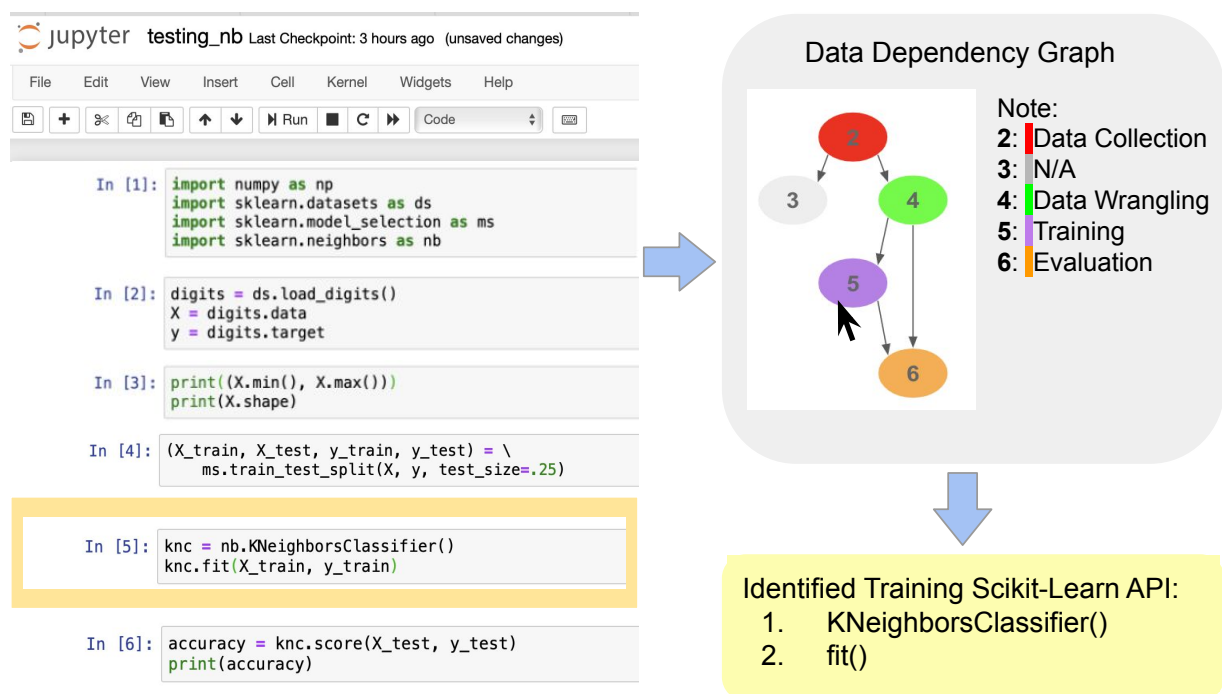
Figure 3.3: Sketch of Navigation Tool Prototype

3.4 Examples of Anticipated Applications

We believe the labeled dependency graph is a *useful* foundation for many tools by providing support for maintenance and evolution for notebooks specifically and data-science pipelines more generally. Here we outline examples of envisioned tooling.

3.4.1 Navigation

Most obviously, we expect that visualizations of the graph will be useful for navigating in a notebook along dependency edges (e.g., jumping over deadends or cells that perform exploration) or navigating directly to code of specific stages. We sketch a simple visualization in Figure 3.3. A plugin could link nodes in the graph with cells in the notebook in both directions. It could further highlight through which variables or cells are dependent or why cells are identified as belonging to specific stages. Highlighting the cells helps the users to track where the cells are and how the dependencies are reflected in the notebook.

3.4.2 Notebook Patterns

Extracting structures from a large set of notebooks allows us to find patterns among them, useful for a variety of tasks. Users can search over code structures of public notebooks. A plugin may highlight alternative cells to the one currently edited, if they exist. Analysis tools might indicate when a user's notebook has an unusual structure. Researchers and tool builders can learn about

July 31, 2022

DRAFT

common or uncommon patterns and use this information to develop tools that are useful for a large number of notebook users. Our graph provides a good abstraction for analyzing patterns.

As an example, we identify (1) when notebooks train multiple models in parallel (models trained independently in different cells on shared or separate input data), (2) when they compare the results of multiple training models, and (3) when they contain deadends. We record the number of these pattern occurrences over all 2333 notebooks from the January 1 and January 6 datasets.

Parallel training processes happen when users explore multiple ML training models on a shared or separate datasets. In such settings, developer tools could help to prune no longer needed branches, merge branches, or even make manually explored differences accessible to AutoML tools. Among all 2333 notebooks in our dataset, 169 notebooks contain parallel training processes on a shared dataset and 575 notebooks contain parallel training processes on separate datasets. In total, 32% of the notebooks explore alternatives in training processes.

In contrast, explicit comparison between different evaluation processes is rare. We found only 83 notebooks among the ones analyzed, which accounts for less than 4% of all notebooks. It seems more common to simply print accuracy numbers and to compare them manually than to compare them in code.

Finally, deadends – data wrangling or exploration cells with no children in the data dependency graphs – occur in almost every notebook analyzed (94%). Tooling could suggest cleanup mechanisms, manual or automated.

3.4.3 Documentation Generation

Our methods can be useful for documentation generation tools for Jupyter notebooks. Well-documented notebooks make it easier for notebook users to understand each other's work. Understanding is important for collaboration among data scientists. One core of our approach is to understand which notebook code cells correspond to which ML pipeline stages. Our methods also provide information of how the cells are classified. These information could be useful for documentation purposes by showing the data science pipelines in notebooks.

3.4.4 Restructuring Notebooks

Restructuring notebooks by merging or splitting cells is another possible application of our work. Our tool labels each code cell with an ML stage using a priority rule to avoid the complexity of multiple labels. Ideally, a notebook cell can be reused to do one dedicated action [17]. Multiple labels might indicate that the cell contains content from different ML stages. Therefore, it could be split up into multiple cells so that each cell only corresponds to one specific ML stage. It is also possible that consecutive cells labeled by the same ML stage should be merged into a larger cell. Our methods provide a fast and accurate way to generate labels for each cell, which can be useful for tools that need to restructure notebooks by merging or splitting cells.

July 31, 2022
DRAFT

Chapter 4

Merging Notebook Structures

Data science is a fundamentally iterative and exploratory process. There are many different approaches to solve a data science problem. Data scientists usually explore many alternatives for different stages in an ML pipeline (e.g., data wrangling, model training and evaluation) to achieve better performance or prediction results or overcome a dead-end [12]. Computational notebooks provide little support for their users to compare or visualize alternatives explored over the histories of notebooks. While notebooks can easily contain over 10 history versions, manually identifying these alternatives is a difficult and time-consuming task. Developers could benefit from better tool support when studying and comparing different notebook versions. We argue central to such support is visualization or navigation tooling which summarizes notebook structures of history versions. We extend previous methods to merge structures of different notebook versions into one labeled dependency graph. Our goal is to use this graph as a road-map to navigate through and understand notebook evolution and identify alternatives explored in notebook histories.

4.1 Methods

To provide more effective support for comprehension of notebook evolution and comparing and visualizing alternatives explored in specific ML stages, we present an algorithm to merge structures of different notebook versions as one directed, labeled dependency graph where every node represents one or more cells with the same content from multiple consecutive notebook versions, every cluster represents one or more connected nodes, and every edge represents a data dependency relation between a pair of nodes in different clusters. In other words, cells identified as the same across a number of consecutive notebook versions belong to one cluster. Within a cluster, multiple nodes occur when a cell's content is modified at some point(s) over these versions. In Figure 1.2, we illustrate the output graph of a notebook with 4 versions.

4.1.1 Matching Notebook Cells

To cluster cells from different notebook versions by their content, we need to find out which cells are related. Since all notebook versions follow a chronological order, we can first deter-
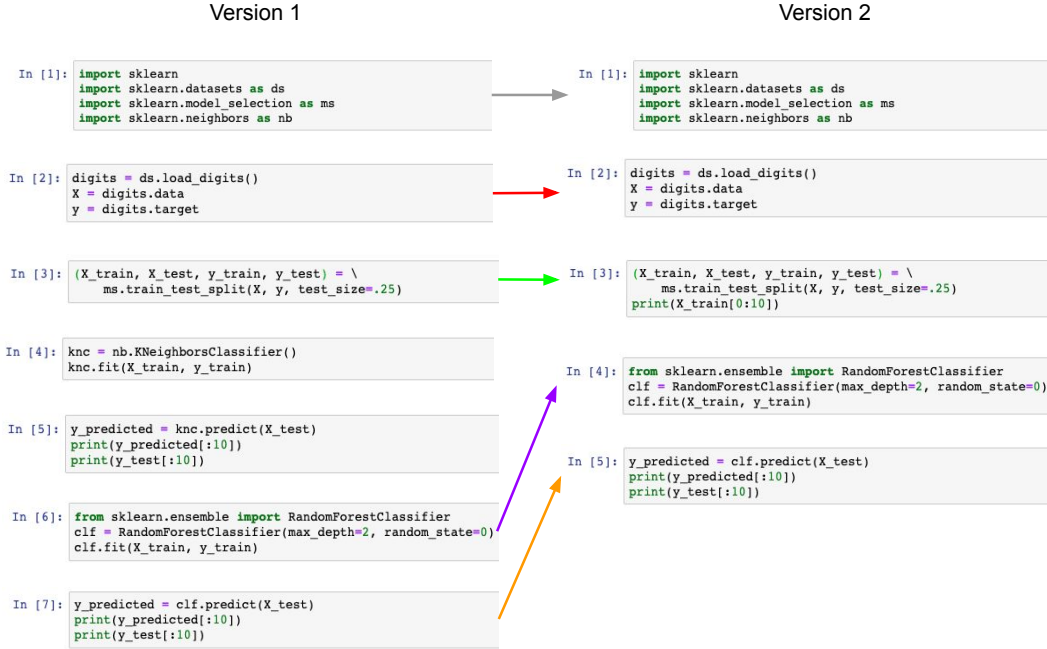
July 31, 2022

DRAFT

Figure 4.1: Result of Matching Algorithm Between Two Notebook Versions

mine which cells are considered the same or related (modified from previous version) between all pairs of consecutive versions and use the results to learn which cells are related over multiple consecutive versions. To solve the fundamental problem, we use the Needleman-Wunsch Algorithm [24] to match cells in two consecutive notebook versions and store a one-to-one mapping of these matching cells. This algorithm produces an optimal global alignment between two sets of notebook cells. We use this alignment to identify related cells in two notebook versions. This alignment works because most notebooks are linear and notebook users are unlikely to change the order of cells in a notebook. Therefore, notebook cells usually remain in relatively the same order as those in the previous version.

To achieve the most optimal matching, we use the Needleman-Wunsch Algorithm to minimize the total Levenshtein distances [25] of all matched pairs of cells. The Levenshtein distance measures the difference between two cells (represented as strings in our program). We argue that a lower Levenshtein distance indicates that the cells have more similar contents – a better match. Therefore, the minimal total Levenshtein distances of all matched cells produces an optimal solution for our problem. An example of our matching result of two notebook versions is shown in Figure 4.1.

4.1.2 Constructing Clusters of Notebook Cells

In a notebook revision, the user might modify existing cells, add new cells, or remove unwanted cells. The cells in the new version which result from modification of existing cells are considered

Figure 4.2: Example of chains and four types of cells: ①, ② and ③ represent the beginning of three chains. ④ represents a cell in the middle of a chain, whereas ⑤ represents the end of a chain. ⑥ and ⑦ represent two isolated cells.
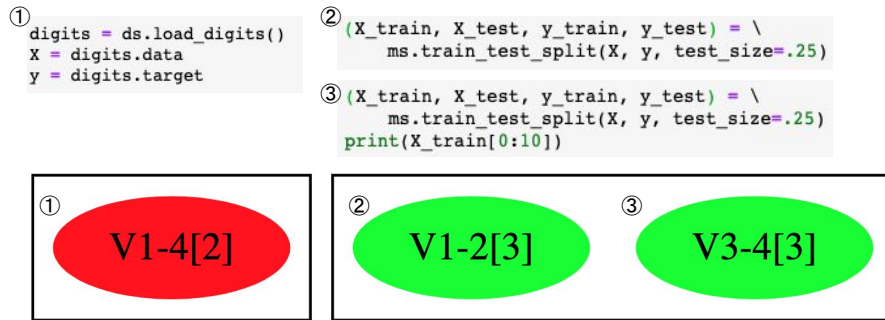

Figure 4.3: Cluster examples: On the left, the cluster contains a single node that is unchanged over 4 versions, labeled by ①. On the right, the cluster contains two nodes, ② and ③, which reflects the modification of the cell in version 3 (an additional line of print statement). ② represents the cell from version 1 to 2, whereas ③ represents the cell from version 3 to 4.

as the same cell in our methods. In a notebook's history, a cell could stay the same for a series of versions, then being continuously modified in the next few versions, and stay unchanged for a number of versions. We expect to represent these cells in consecutive versions by a cluster in our graph to show their connections over time. To form these clusters, we need to find groups of cells related to each other that are identified as the same across a number of consecutive notebook versions using our matching algorithm. Each notebook version is associated with a unique version number to represent the chronological order among all notebook versions.

To form these clusters, we iterate through all notebook versions chronologically (i.e. from the lowest version number to the highest) and find all pairs of matching cells between every pair of consecutive versions. In each notebook version, we classify every cell into four types: the middle, beginning, or end of a chain, or an isolated cell. A chain refers to a series of consecutively matched cells across multiple versions, and an isolated cell is one that does not belong to any chain, as shown in Figure 4.2.

Every match we found between two cells is represented by an edge from the cell with the lower version number to the one with the higher number. To classify these cells, we consider their

July 31, 2022

DRAFT

number of incoming and outgoing edges as the following cases:

- No incoming or no outgoing edges: this cell is an isolated cell.
- No incoming edge but an outgoing edge: this cell is the beginning of a chain.
- An incoming edge but no outgoing edge: this cell is the end of a chain.
- Both an incoming and an outgoing edge: this cell is in the middle of a chain.

We construct a cluster for every chain and every isolated cell in our output graph. For every isolated cell, the cluster has one node representing the cell. For every chain, the cluster can have one or multiple nodes. Every time the content of the cells changes in a chain, one node is added to reflect this modification, as shown in Figure 4.3.

4.1.3 Constructing Edges

Edges represent data dependency relations between nodes in different clusters. To identify these edges, we iterate through data dependency relations in all notebook versions. For each data dependency relation between two cells in a notebook version, we find their corresponding nodes and add an edge between them. We keep a dictionary to record existing edges in the graph to avoid multiple edges between the same pair of nodes. We also maintain a mapping from a notebook version number and a cell number to its corresponding node in the graph so that we can look up nodes easily.

4.1.4 Identifying ML Stages

Our goal is to label each node in our graphs of merged notebook structures to learn which ML pipeline stage it belongs to. We first generate labeled dependency graphs of all notebook versions using our previous algorithm, then inherit the ML stage labelings from these results. Since every cell in the notebooks corresponds to a unique node in graph after merging, we can use the cell number and its notebook version number to find the node using our mapping from before and label the node accordingly.

4.2 Evaluation

To be useful for maintenance and evolution tooling, our algorithm of merging structures of notebook versions into labeled, dependency graphs needs to be accurate. This ensures our results provide valuable understanding of notebook histories. To show our methods is useful for learning and extracting insights from notebook evolution, we measure how often notebook users modify, add, or remove cells in different ML stages. We also discuss feedback from a formative evaluation on the usefulness of our methods for learning alternatives explored in specific ML stages over notebook histories.

We evaluate our methods in three ways. First, we measure the accuracy of our generated graph.

Table 4.1: Total and Average Number (per Version) of Changed, Added, and Removed Cells in 3 Kaggle Notebooks with 32 Versions in Total

Type of Cell	Total Number	Average Number
Changed	64	2
Added	83	2.59
Removed	38	1.19

Table 4.2: Total Number of Changed, Added, and Removed Cells in 3 Kaggle Notebooks Categorized by ML Stages

ML Stage	Changed	Added	Removed
Data Collection	0	1	1
Data Wrangling	14	9	2
Training	8	4	0
Evaluation	1	0	0
Training+Evaluation	0	0	0
Exploration	22	17	10
N/A	19	52	25

Second, we present statistics measured for a sample of 3 Kaggle notebooks. Third, we present feedback collected from our formative evaluation.

4.2.1 Accuracy

To evaluate the accuracy of our graph, we manually checked all nodes, clusters, and edges for a selected notebook with 4 versions, and confirmed matched cells between all pairs of consecutive notebook versions. We opted not to perform a systematic evaluation here because establishing ground truth is both tedious and time-consuming. Since we have evaluated cell labelings in the last chapter, we decided not to perform new evaluation here.

4.2.2 Analysis for Notebook Evolution

We expect our algorithm to be useful for extracting insights in notebook evolution by learning how often notebook cells are modified, added, or removed in specific ML stages or in general across a number of notebook versions. We generated a sample of 3 Kaggle notebooks. Kaggle is a platform where data scientists upload datasets or solutions to Kaggle competitions [26]. We decided to use Kaggle notebooks because they are high-quality and they provide all history versions of the notebooks automatically.

We measure the number of cells modified, added, and removed across all versions in all three notebooks, and categorize them by ML stages. For cells modified, we output the Levenshtein distances of these cells between the previous and current versions, and whether their ML stage

labels become different. Our observations are shown in Table 4.1 and Table 4.2. The total Levenshtein distances is 3816, averaged to 59.63 per modified cell. Out of all 64 modified cells, only 3 ML stage labels change.

4.2.3 A Formative Evaluation

Formative evaluation is a systematic process which gathers information and data to revise and improve the product under development, which usually happens in the early stages of the design [3]. We expect a formative evaluation to be helpful for us to understand how users are thinking about the usefulness of our methods.

We performed a simple formative evaluation with one participant, who is currently working on a project that requires manually identifying alternatives explored in different ML pipeline stages over a notebook's history. We aim to select participants whose work that can potentially benefit from our methods, specifically in identifying alternatives explored in notebook versions. Due to limited time, this participant is the one we can find who fits best to our goal.

In the evaluation, we discuss our methods and results with our participant and collect feedback on the usefulness of our algorithm for identifying alternatives in different ML stages. We first present our participant a small graph example of a notebook with 4 versions, and explain how different parts of the graph structure work. Once the participant gets a better understand of our methods, we present our result of a notebook with 14 versions which the participant has worked on before, and discuss how we can use the graph to trace potential explored alternatives in specific ML stages in the notebook versions. We show the participant a cluster in the graph corresponding to three alternatives in the training process, which confirms with the results found manually by the participant. The participant agreed that the graph is useful for navigating through notebook history and visualizing the notebook structures and ML stages: "The graph can be used like a road-map to trace the machine learning pipeline workflows in different notebook versions". In addition, we discussed whether our tool could be helpful for finding alternatives explored in the ML pipeline, and concluded that the graph is useful for showing potential sites of alternatives explored in specific ML stages that can be confirmed easily after checking the source code. This visualization could make manually identifying alternatives in specific ML stages a much less stressful and more efficient process.

While our formative evaluation provides insights into the usefulness of our work, it is limited by the number of participants. As future work, we expect to reach out to more developers and structure our evaluation in a more formal way.

4.3 Examples of Anticipated Applications

We believe that merging notebook structures as one graph is a *useful* foundation for tools that provide support for visualizing and navigating through notebook evolution and identifying alternatives explored in specific ML stages over notebook histories. We discuss examples of potential
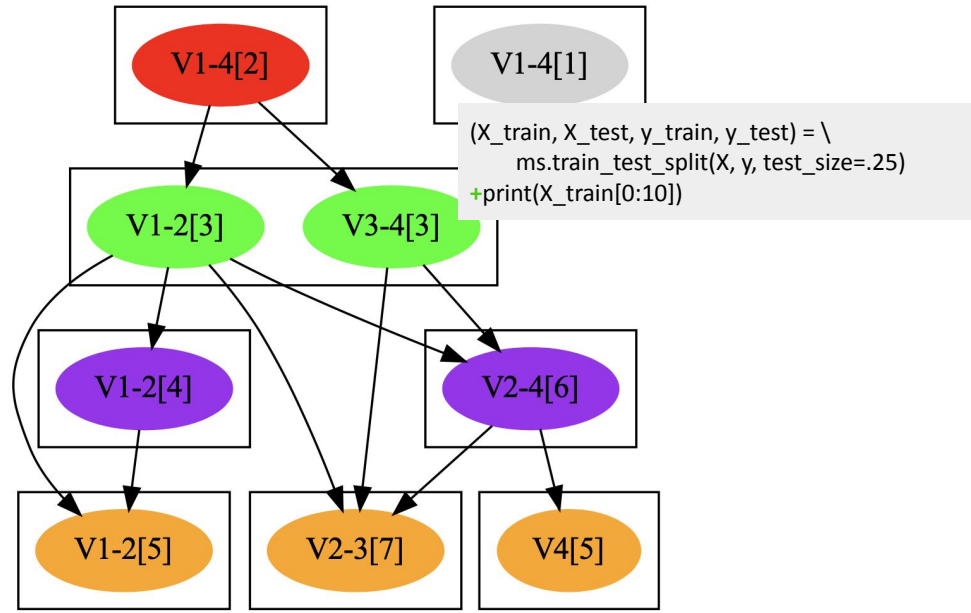
Figure 4.4: Visualization of Merged Notebook Structures and Highlight of Node Content

applications as follows.

4.3.1 A Navigation Prototype

We expect that merging notebook structures as a labeled dependency graph will be useful for visualizing and navigating through notebook histories. The output graph shows how notebook cells change over time and which cells are identified as the same over a series of versions. The visualizations can be helpful for navigating along dependency edges (e.g., tracing a ML pipeline) or jumping directly to code of specific versions or stages. A plugin could link nodes in the graph with cells represented by the nodes in both directions. It could further highlight how the source code of a node is changed compared with the previous one in a cluster (e.g., which lines are changed, added or removed). Highlighting the cells helps the users to track where the cells are in the notebook versions and highlighting the changes shows the users how the cells change over time. The tools could also mark which variables or cells are dependent to show how the dependencies are reflected in the notebooks. We sketch a simple visualization in Figure 4.4.

4.3.2 Finding Alternatives

Data scientists explore alternatives in different ML pipeline stages to analyze data and build high-performance ML models. Studies show that data scientists frequently reuse previous ideas to develop new analysis and explore a non-linear path to reach their final results (e.g., copying and reusing code, keeping old code and analysis in case they will be useful later) [6]. Relying
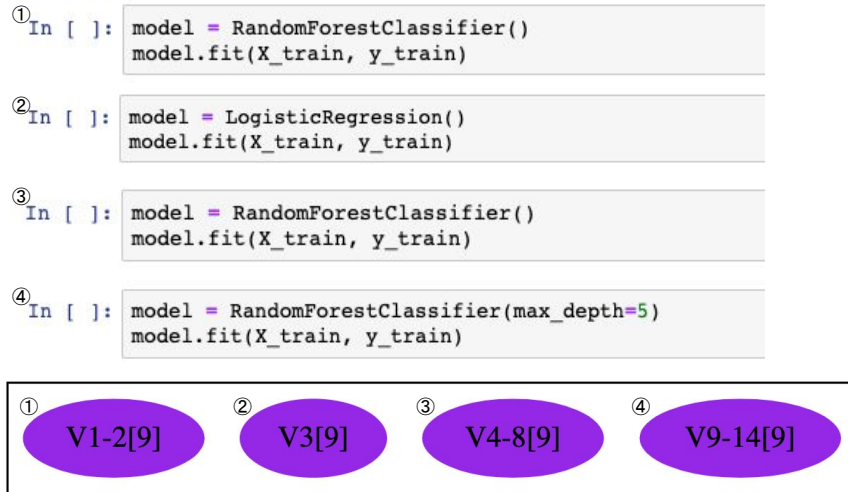
Figure 4.5: Example of a cluster with 4 nodes: this cluster shows that the cell content has changed three times over the notebook history. Examining the source code, we find three training alternatives explored in this cluster: *RandomForestClassifier()* in ① and ③, *LogisticRegression()* in ②, and *RandomForestClassifier(maxdepth=5)* in ④.

only on memory of their code or informal practices like copying or commenting out code is insufficient for data scientists to find all explored alternatives in past notebook versions.

Our visualization of merging structures of different notebook versions is useful for finding alternatives explored in a notebook's history. Users can search for clusters on a graph in which the cell's content changes multiple times. These could be potential sites of explored alternatives and the users can check the notebooks' source code to confirm their hypotheses easily. Finding these clusters in our graph and checking their source code would be much more time-efficient than manually looking for alternatives in a large number of notebook versions. We generated our visualization for a notebook with 14 versions, and show one cluster of nodes in the graph corresponding to explored training alternatives in Figure 4.5.

Chapter 5

Conclusion

Computational notebooks enable their users to interleave code, visualizations, and narrative texts in a single document [11], and have become a very popular coding environment among data scientists. While their exploratory nature allows data scientists to write and refine code easily and to execute parts of their code in any order [4], many practitioners and researchers still report problems [2, 4]. Common complaints include dead-ends, duplicated code, and tangled or scatter code [19]. Studies have shown that most notebook cells are presented in a different order than they are executed, and code that produces a result is often scatter across many cells [4]. "Messy" notebooks are also criticized for bad practices in modularizing code [13]. Poor code quality in notebooks undermines understanding, reuse, and collaboration among data scientists [20]. These problems motivate us to improve tool support for notebook maintenance and evolution so that data scientists can better understand, navigate, modularize, and maintain notebook code, even across many notebook revisions. We believe central to such tool support is identifying the structure of notebooks.

We implemented an efficient and lightweight algorithm to identify and extract Jupyter notebook structures as directed, labeled data dependency graphs, where nodes represent cells and directed edges represent data dependency relations among cells. The algorithm involves generating data dependency information of cells and labeling cells with ML stages. Our evaluation shows that our methods achieve high accuracy (75%) for labeling cells and fast runtime performance. We compare our evaluation results with two prior works and find our methods are better in terms of both accuracy and performance. We sketch a navigation tool prototype using our labeled data dependency graphs and discuss a number of patterns found in our notebook dataset. Given the efficient runtime, tool builders can run our analysis in the browser background and use our labeled dependency graphs for various purposes like navigation, documentation generation, or learning about notebook structures in general. We believe our methods would encourage more notebook maintenance tooling by providing a common underlying analysis infrastructure to identify and extract notebook structures effectively and efficiently.

In addition, we extend our algorithm to merge notebook structures as one directed, labeled data dependency graph for multiple versions of a notebook to visualize notebook evolution and revisions. In this graph, nodes represent cells with the same content from consecutive notebook

versions, clusters represent a set of nodes which represent the same cell with slightly different content, and edges represent data dependency relations among nodes. For evaluation, we manually verify the accuracy of a graph, and present how often are cells changed, added, or removed in a sample of 3 Kaggle notebooks, which shows our methods allow us to analyze and measure the changes between consecutive notebook versions. We discuss examples of possible applications of this extension, which include navigating through a notebook's history for better understanding and finding alternatives explored in specific ML stages over notebook revisions.

We discuss future work for our methods of identifying and merging notebook structures in Chapter 3.4 and 4.3 respectively, including building a navigation tool and finding alternatives explored in specific ML stages.

Appendix A

Example Keywords for Identifying ML Stages

Stage Name	Class Name/API Calls
Data Collection	fetch_20newsgroups fetch_california_housing load_digits load_iris load_sample_images …
Data Wrangling	PCA DictVectorizer MinMaxScaler OneHotEncoder FunctionTransformer …
Training	GaussianProcessClassifier LogisticRegression Perceptron RidgeClassifier LinearRegression …
Evaluation	accuracy_score average_precision_score confusion_matrix f1_score log_loss …
Training+Evaluation	Training and Evaluation API calls as above
Exploration	ConfusionMatrixDisplay PrecisionRecallDisplay summary histogram KMeans …

Figure A.1: Example Keywords for Identifying ML Stages

Bibliography

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *Proc. ICSE-SEIP*, 2019. 2.3, 3.2

[2] Souti Chattopadhyay, I. Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proc. CHI*, 2020. 1.1, 2.2, 5

[3] Walter Dick and Lou M Carey. Formative evaluation. *Instructional design: Principles and applications*, pages 311–333, 1977. 4.2.3

[4] Andrew Head, Fred Hohman, Titus Barik, Steven Mark Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Proc. CHI*, 2019. 1, 1.1, 2.2, 5

[5] Mary Beth Kery and Brad A. Myers. Interactions for untangling messy history in a computational notebook. In *Proc. Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 147–155, 2018. 2.2

[6] Mary Beth Kery, Amber Horvath, and Brad A. Myers. Variolite: Supporting exploratory programming by data scientists. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017. 4.3.2

[7] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proc. CHI*, 2018. 1, 2.1, 2.2

[8] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012. 3.3.2

[9] Microsoft. Pyright. `https://github.com/microsoft/pyright`, 2020. 3.2, 3.3.3

[10] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. In *Proc. ICSE*, 2022. 1

[11] Kayur Patel, James Fogarty, James A Landay, and Beverly Harrison. Investigating statistical machine learning as a tool for software development. In *Proc. CHI*, pages 667–676, 2008. 1, 5

[12] Kayur Patel, James Fogarty, James A. Landay, and Beverly L. Harrison. Investigating statistical machine learning as a tool for software development. In *CHI*, 2008. 4

[13] João Felipe Pimentel, Leonardo Gresta Paulino Murta, Vanessa Braganholo, and Juliana

Freire. A large-scale study about quality and reproducibility of Jupyter notebooks. In *Proc. Conf. Mining Software Repositories (MSR)*, pages 507–517, 2019. 1.1, 2.1, 2.2, 3.1, 5

[14] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. Data science through the looking glass and what we found there. *ArXiv*, abs/1912.09536, 2019. 1, 3.2

[15] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. Aiding collaborative reuse of computational notebooks with annotated cell folding. In *Proc. CHI*, 2018. 2.2

[16] Adam Rule, Aurélien Tabard, and James D. Hollan. Exploration and explanation in computational notebooks. *Proc. CHI*, 2018. 2.2

[17] Sergey D. Titov, Yaroslav Golubev, and Timofey Bryksin. Resplit: Improving the structure of Jupyter notebooks by re-splitting their cells. *ArXiv*, abs/2112.14825, 2021. 2.2, 3, 3.4.4

[18] Ashwin Prasad Shivarpatna Venkatesh and Eric Bodden. Automated cell header generator for Jupyter notebooks. In *Proc. International Workshop on AI and Software Testing/Analysis*, page 17–20, 2021. ISBN 9781450385411. doi: 10.1145/3464968.3468410. URL https://doi.org/10.1145/3464968.3468410. 2.3, 3.3.2

[19] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael J. Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. Documentation matters: Human-centered AI system to assist data science code documentation in computational notebooks. *ACM Transactions on Computer-Human Interaction*, 29, 2022. 1.1, 2.2, 3.1, 5

[20] Jiawei Wang, Li Li, and Andreas Zeller. Better code, better sharing: On the need of analyzing Jupyter notebooks. In *Proc. ICSE-NIER*, page 53–56, 2020. ISBN 9781450371261. doi: 10.1145/3377816.3381724. URL https://doi.org/10.1145/3377816.3381724. 1, 1.1, 5

[21] Jiawei Wang, Tzu yang Kuo, Li Li, and Andreas Zeller. Assessing and restoring reproducibility of Jupyter notebooks. In *Proc. ASE*, pages 138–149, 2020. 2.1

[22] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. Fork it: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445527. URL https://doi.org/10.1145/3411764.3445527. 2.2

[23] John E. Wenskovitch, Jian Zhao, Scott A. Carter, Matthew L. Cooper, and Chris North. Albireo: An interactive tool for visually summarizing computational notebook structure. In *Proc. Visualization in Data Science (VDS)*, pages 1–10, 2019. 2.2, 3.2

[24] Wikipedia contributors. Needleman–wunsch algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Needleman%E2%80%93Wunsch_algorithm&oldid=1090847497, 2022. [Online; accessed 19-July-2022]. 4.1.1

[25] Wikipedia contributors. Levenshtein distance — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Levenshtein_

July 31, 2022

DRAFT

`distance&oldid=1096620738`, 2022. [Online; accessed 19-July-2022]. 4.1.1

[26] Chenyang Yang, Shurui Zhou, Jin L. C. Guo, and Christian Kästner. Subtle bugs everywhere: Generating documentation for data wrangling code. In *Proc. ASE*, pages 304–316, 2021. 2.2, 2.3, 3.2, 4.2.2

[27] Ge Zhang, Michael Merrill, Yang Liu, Jeffrey Heer, and Tim Althoff. Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis. *EPJ Data Science*, 11, 2022. 2.3, 3.2, 3.3.2, 3.3.3

July 31, 2022

DRAFT