Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis

Ashwin Prasad Shivarpatna Venkatesh[§]
Heinz Nixdorf Institut
Paderborn University, Paderborn, Germany

Jiawei Wang[†]
Faculty of Information Technology
Monash University, Melbourne, Australia

Li Li[‡]
School of Software
Beihang University, Beijing, China

Eric Bodden[*]
Heinz Nixdorf Institut
Paderborn University & Fraunhofer IEM, Paderborn, Germany

Email: [§]ashwin.prasad@upb.de, [†]jiawei.wang1@monash.edu, [‡]lilicoding@ieee.org, [*]eric.bodden@upb.de

***Abstract*—Jupyter notebooks enable developers to interleave code snippets with rich-text and in-line visualizations. Data scientists use Jupyter notebook as the de-facto standard for creating and sharing machine-learning based solutions, primarily written in Python. Recent studies have demonstrated, however, that a large portion of Jupyter notebooks available on public platforms are undocumented and lacks a narrative structure. This reduces the readability of these notebooks. To address this shortcoming, this paper presents HeaderGen, a novel tool-based approach that automatically annotates code cells with categorical markdown headers based on a taxonomy of machine-learning operations, and classifies and displays function calls according to this taxonomy. For this functionality to be realized, HeaderGen enhances an existing call graph analysis in *PyCG*. To improve precision, HeaderGen extends *PyCG*'s analysis with support for handling external library code and flow-sensitivity. The former is realized by facilitating the resolution of function return-types. Furthermore, HeaderGen uses type information to perform pattern matching on code syntax to annotate code cells.**

The evaluation on 15 real-world Jupyter notebooks from Kaggle shows that HeaderGen's underlying call graph analysis yields high accuracy (96.4% precision and 95.9% recall). This is because HeaderGen can resolve return-types of external libraries where existing type inference tools such as *pytype* (by Google), *pyright* (by Microsoft), and *Jedi* fall short. The header generation has a precision of 82.2% and a recall rate of 96.8% with regard to headers created manually by experts. In a user study, HeaderGen helps participants finish comprehension and navigation tasks faster. All participants clearly perceive HeaderGen as useful to their task.

***Index Terms*—static analysis, python, code comprehension, annotation, literate programming, jupyter notebook**

I. Introduction

Machine learning (ML) and data-science are evolving as a multi-disciplinary field, comprising of software engineering on one end and domain-specific knowledge on the other. The ML community has largely adopted Jupyter notebooks as the de-facto standard for developing ML solutions. Notebooks are based on the principle of *literate programming* [1] that advocates the combination of code, documentation and visualization as a single document. The central idea of literate programming is to enhance comprehension and sharing of solutions to complex problems. This can be achieved by following literate programming principles such as: (1) enriching code with rich descriptive texts and figures, (2) creating a narrative structure in the program by adding headers to code snippets, and (3) logically dividing and labeling reusable sections of the program. In notebooks, executable code is written in code cells and documentation is written in markdown cells. An example notebook showing Python code and markdown cells can be seen in Figure 2. Note that the most used language for developing ML-based solutions in notebooks is Python [2].

Enriching code snippets with explanatory text enhances the overall comprehensibility of notebooks and further promotes collaboration [3]. Furthermore, Wagemann et al. [3] suggests that a markdown/code cell ratio of 2, i.e., twice the number of markdown cells compared to code cells, is an indication of good literate programming practice. In addition, Samuel and Mietchen [4] also report that notebooks with higher markdown/code cell ratio are expected to have better reproducibility, which is a critical indicator in scientific studies.

While Jupyter notebooks enable the easy creation of computational narratives according to literate programming principles, this is often not practiced in real-world notebooks [5]. Instead, studies have shown that code-smells and bad practices are common in publicly available notebooks [6]. According to a study by Rule et al. [2], interviewees defined their notebooks as personal scratch-pads and "messy", in other words, that their notebooks lack a narrative structure. The authors also highlighted that data scientists often do not annotate their notebooks, citing either lack of time or being "too lazy". In a later study, Pimentel et al. [7] found that 30.93% of the 1.4 million real-world notebooks they studied had no markdown cells. This finding is consistent with the latest study by Quaranta et al. [8]. On assessing the extent to which data scientists are familiar with, and follow best practices, the authors note that there is lack of effort in annotating notebooks with markdown cells. Yet, striving to adhere to literate programming principles becomes crucial in educational and sharing communities, for instance, in platforms such as Kaggle [9], as bad coding practices can lead to mistakes being carried on to the next generation of developers. Therefore, we argue that there is a strong need for the software engineering research community to develop tools for notebook users.

To this end, this paper proposes HeaderGen, a tool-based approach to enhance the comprehension and navigation of undocumented Python based Jupyter notebooks by automatically creating a narrative structure in the notebook.

Figure 1 shows a taxonomy of ML operations inspired by the work of Wang et al. [10]. Data scientists build an ML-based solution notebook by first preparing the data, then extracting key features, and then creating and training the model. HeaderGen leverages this implicit narrative structure of an ML notebook to add structural headers as annotations to the notebook. HeaderGen works by precisely detecting every function call in the notebook, classifying it according to the ML operations taxonomy, and then uses this classification information to create a structural map of the notebook. This map is displayed as an *"index of ML operations"* at the top of the notebook, giving the notebook a narrative structure. Additionally, each code cell is annotated with a markdown header indicating the ML operations being performed. (see example in page 6)

To yield useful results, HeaderGen requires a fast and accurate program analysis that can precisely identify all function calls in the notebook. However, we found that none of the existing techniques were able to statically identify all function calls in a notebook with acceptable precision, recall and, run time. This is attributed to the complex features of Python, such as duck typing, dynamic code execution, reflection, etc, that are challenging to static analyzers [11], [12]. Moreover, unlike other programming languages like Java, Python lacks a lot of tool-support for state-of-the-art static analysis (SA) techniques. Instead, most tools available for Python today are based on a makeshift analysis of abstract syntax trees (AST) of Python source code [13]. Furthermore, due to the dynamically typed nature of Python, concrete static type-inference of variables is required for precise static analysis. A recently published call-graph generation technique called PyCG [11] is based on an intermediate representation of the AST and handles several complex Python features. Yet, PyCG fails to analyze function calls to external libraries and its analysis is flow-insensitive, making it impossible to precisely identify function calls in real-world programs. HeaderGen rectifies these limitations.

To summarize, the challenge that HeaderGen addresses is two-fold: (1) Inaccurate static analysis: the absence of a static program analysis technique that can precisely identify function calls in a Python program. To mitigate this, HeaderGen extends the call-graph analysis in PyCG with the ability to resolve function calls to external libraries and adds flow-sensitivity. (2) Undocumented notebooks: many publicly available notebooks are undocumented, which hampers comprehension and goes against the principle of literate programming. HeaderGen uses precise SA to automatically annotate the notebook with structural headers and creates a narrative structure to aid comprehension of undocumented notebooks.

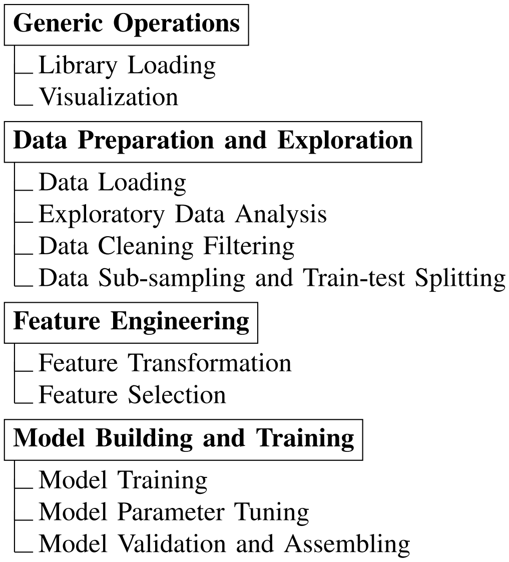


Fig. 1. Taxonomy of machine learning operations based on [10].

To assess HeaderGen's static function-call analyzer, we use an extended version of PyCG's micro-benchmark, and in addition, a real-world benchmark with 15 notebooks from Kaggle. On the real-world benchmark, HeaderGen achieved 96.4% precision and 95.9% recall, outperforming PyCG and other function call analyzers based on off-the-shelf tools such as *pyright* [14] and *Jedi* [15]. On the same benchmark we also evaluated HeaderGen for header annotation and achieved 82.2% precision and 96.8% recall. Furthermore, we conducted a user-study with eight data-science practitioners and found clear evidence that HeaderGen improves the speed of navigation and comprehension. The contributions of this work are summarized as follows:

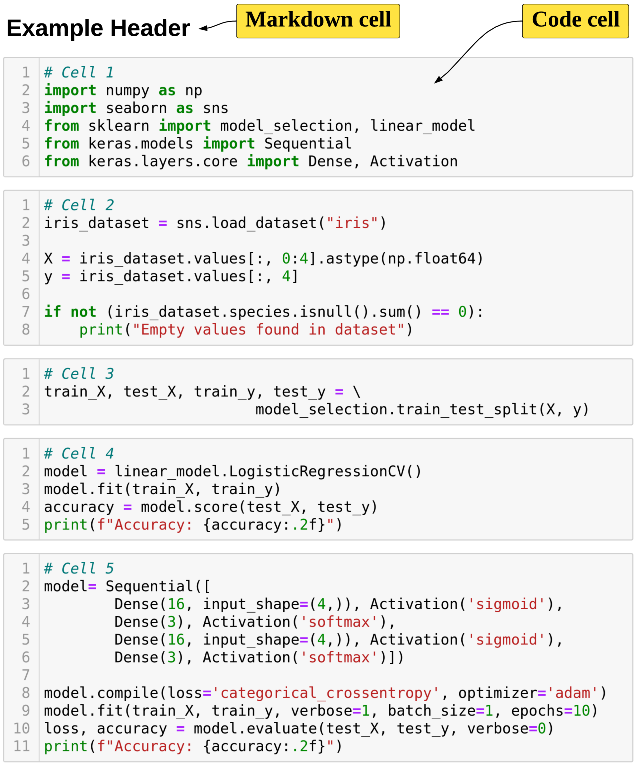
- We propose a novel static analysis based approach for Python Jupyter notebooks that can automatically annotate them with structural, commentary, and navigational text, aiming to facilitate the literal programming practice.
- We implement a static function call extraction technique for Python with 96.4% precision and 95.9% recall on our real-world benchmark.
- We give an evaluation of our approach based on extensive experimental results.
- We implement the prototype named HeaderGen and make it publicly available for our community to reuse.

The remainder of this paper is organized as follows: we present challenges in supporting computational notebooks with static header generation in the Section II followed by detailing our design in Section III. We then present the evaluation from Section IV to Section VIII, and discuss existing research in Section IX. The limitations of HeaderGen is discussed in Section X and finally the paper is concluded in Section XI.

Availability. HeaderGen is published on GitHub as open-source software under Apache 2.0 license: https://github.com/ashwinprasadme/headergen

II. Motivating Example

As a motivating example, consider the notebook in Figure 2. It consists of one markdown cell which is rendered as an



```
 1 # Cell 1
 2 import numpy as np
 3 import seaborn as sns
 4 from sklearn import model_selection, linear_model
 5 from keras.models import Sequential
 6 from keras.layers.core import Dense, Activation
```

```
 1 # Cell 2
 2 iris_dataset = sns.load_dataset("iris")
 3 
 4 X = iris_dataset.values[:, 0:4].astype(np.float64)
 5 y = iris_dataset.values[:, 4]
 6 
 7 if not (iris_dataset.species.isnull().sum() == 0):
 8     print("Empty values found in dataset")
```

```
 1 # Cell 3
 2 train_X, test_X, train_y, test_y = \
 3                         model_selection.train_test_split(X, y)
```

```
 1 # Cell 4
 2 model = linear_model.LogisticRegressionCV()
 3 model.fit(train_X, train_y)
 4 accuracy = model.score(test_X, test_y)
 5 print(f"Accuracy: {accuracy:.2f}")
```

```
 1 # Cell 5
 2 model= Sequential([
 3         Dense(16, input_shape=(4,)), Activation('sigmoid'),
 4         Dense(3), Activation('softmax'),
 5         Dense(16, input_shape=(4,)), Activation('sigmoid'),
 6         Dense(3), Activation('softmax')])
 7 
 8 model.compile(loss='categorical_crossentropy', optimizer='adam')
 9 model.fit(train_X, train_y, verbose=1, batch_size=1, epochs=10)
10 loss, accuracy = model.evaluate(test_X, test_y, verbose=0)
11 print(f"Accuracy: {accuracy:.2f}")
```

Fig. 2. Machine learning Jupyter notebook example.

HTML header, and five code cells that can be identified by the comments in the first line of each code cell. The example notebook in Figure 2 is a concise version of a real-world notebook containing a machine learning (ML) based solution.

In cell 1, various ML libraries are imported. In cell 2, a sample dataset called “iris” from the seaborn library is loaded, and further feature selection operations are performed to retain only the essential columns from the dataset. Values are type-cast to numpy based float64 type. Finally, the dataset is checked for null values. In cell 3, the dataset is split into training and test datasets. In cells 4 and 5, with the processed dataset, two different ML models are defined, trained, and their accuracies are reported. In cell 4, a basic linear model based on logistic regression is used. In cell 5, a deep learning based sequential model is used.

Note that this notebook is undocumented and does not contain any explanatory text or structural headers as markdown cells, violating the literate programming principle. One in three notebooks found in the wild does not contain any markdown cells [7]. In absence of explanatory text or structural headers, ML practitioners, especially beginners, must spend more time to navigate and comprehend different aspects of the notebook. Particularly considering that nearly a third of all notebooks in the real-world contain at least 50 cells [7].

On the other hand, the example notebook poses several challenges to SA, including:

- **Import aliasing:** different ways of importing libraries, and importing libraries with aliases.
- **Dynamic typing:** in cell 2, the type of the variable `iris_dataset` is not known statically, i.e., the return-type of the function `load_dataset()` is not known statically. As a result, subsequent statements that involve the variable `iris_dataset` cannot be resolved, i.e., in cell 2 lines 4–7.
- **Chained function calls:** consider the function call in cell 2 line 4, `iris_dataset.values[].astype()`, here, the variable `iris_dataset` is of type *Dataframe* from the *Pandas* library. `iris_dataset.values` refers to an attribute of the class *Dataframe*, which is in-turn defined as a *Numpy* array. Furthermore, `astype()` refers to a function from the *Numpy* library. Existing SA tools fail to resolve all this information statically.
- **Variable reuse:** the same variable `model` is reused in cells 4 and 5, for different model objects, i.e., `Sequential` and `LogisticRegressionCV` objects. Reuses of the same variable names are common in notebooks. Therefore, for precise annotation of code cells, the analyzer should know the type of an object at a specific location in the notebook. In other words, the analysis should be flow-sensitive.

In summary, for HeaderGen to accurately classify code cells based on function calls, the static analyzer needs to: (1) handle complex Python features, (2) statically resolve return-types of external library calls, and (3) be flow-sensitive.

III. Approach

Figure 3 gives a high-level overview of HeaderGen. First, it converts a notebook into a native Python script for analysis. This strips metadata from the notebook that are irrelevant for analysis. HeaderGen then analyzes the Python script to create an extended assignment graph (EAG). Further, HeaderGen extracts flow-sensitive callsite information based on the EAG, and finally annotates the notebook with headers based on the identified callsites and adds the index of ML operations based on the library-to-taxonomy mapping database.

We discuss the details of constructing an EAG and extracting flow-sensitive callsite information in Sections III-A and III-B. Then, in Section III-C, we discuss the process of annotating the notebook based on the output of the analyzer.

A. *Extended Assignment Graph*

To extract all possible callsites in the program, we add flow-sensitivity and the ability to analyze external libraries to the existing state-of-the-art context-insensitive and inter-procedural call-graph (CG) generation technique, PyCG [11]. PyCG works on a custom intermediate representation of a Python AST and generates an assignment graph (AG) that represents assignment relations between program identifiers. The CG is then generated based on the AG by resolving all function calls that a program variable might point-to. Figure 4a shows the AG generated by PyCG for the variable `model` in our motivating example. Since PyCG cannot analyze calls to external libraries, it does not add any edges to the `model` node. However, callsite information from real-world notebooks cannot be extracted with high accuracy without analyzing external library functions. To wit, in our motivating example, without analyzing the function `load_dataset()`

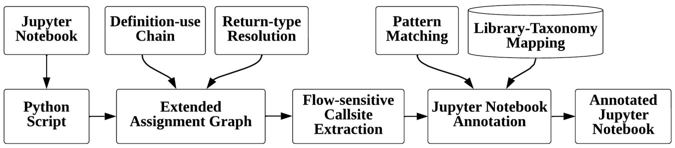

Jupyter Notebook
Definition-use Chain
Return-type Resolution
Pattern Matching
Library-Taxonomy Mapping
Python Script
Extended Assignment Graph
Flow-sensitive Callsite Extraction
Jupyter Notebook Annotation
Annotated Jupyter Notebook

Fig. 3. High-level overview of HeaderGen.

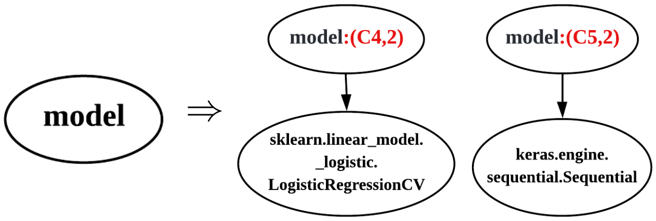
from the seaborn library, further references to the variable `iris_dataset` cannot be resolved. Moreover, PyCG's analysis is flow-*insensitive*, therefore the generated AG fails to distinguish between different assignments to the same variable. For instance, in our motivating example, `model` is redefined in cell 5 (cf. Figure 2), however, the generated AG shown in Figure 4a maintains only a single node for the `model` variable. PyCG over-approximates `model` with weak-updates to the AG, thereby, compromising on precision.

Therefore, we extend PyCG's AG by an extended assignment graph (EAG) based on an additional helper analysis to enable flow-sensitive callsite recognition and further add a return-type approximation technique to resolve calls to external libraries.

Definition-use Chain for Flow-sensitivity. A definition-use chain [16] (DUC) is a data structure that represents a definition, or assignment, of a program variable and all the subsequent uses without any re-definitions in between. DUCs are generated by analyzing all assignment statements in the program with consideration of variable scopes.

We use an existing tool, Beniget [17], a DUC generation tool that works by analyzing the AST of Python programs. While a tool exists for Python to compute the DUC, no existing implementation makes use of DUC to construct flow-sensitive call-graphs for Python. HeaderGen first uses the DUC generated by Beniget to create a location map that gives information about what variables are used at particular locations of a notebook. Then, this map is used to create the EAG that can differentiate variables based on the location of its definition. For instance, the EAG shown in Figure 4b captures multiple definitions of the `model` variable separately.

Return-type Resolution of Machine Learning Libraries. Consider the variable `iris_dataset` assigned to the return of function `load_dataset()` at location cell 2 line 2, represented as (C2,2) in the motivating example (Figure 2). Within the seaborn library, the call to `load_dataset()` is resolved to `seaborn.utils.load_dataset`, which returns an object of type `pandas.Dataframe`. For HeaderGen, this type information is crucial: only if HeaderGen knows `iris_dataset`'s type can it statically analyze calls on this variable. For instance `iris_dataset` is used at (C2,4), (C2,5), and (C2,7) all of which cannot be resolved without knowing `iris_dataset` is of type `pandas.Dataframe`. Yet, Python is a dynamically typed language, return-type information is not readily available for most library code. Although a set of Python Enhancement Proposals (PEPs) such as PEP484 [18] are placed for Python language to support type annotations directly in source code, recent work has suggested that such user-demanding knowledge is still missing [19].


model
⇒
model:(C4,2)
sklearn.linear_model._logistic.LogisticRegressionCV
model:(C5,2)
keras.engine.sequential.Sequential

(a) Assignment Graph (b) Extended Assignment Graph

Fig. 4. Generated assignment graphs for the variable "model" in the motivating example, left in PyCG (empty), right in HeaderGen (flow-sensitive).

Though it still remains an open challenge, researchers have given type inference for Python a lot of attention. While leading tech giants like Google, Meta, and Microsoft rely on static tools (e.g. *pytype* [20]) to ensure the quality of their codebase, the majority of current efforts employ the deep learning technique. Unfortunately, none of the available tools can accomplish what we need. This is mainly because external function calls frequently create dataflow disruptions in notebook programs. Existing learning-based approaches such as Typilus [21] often only leverage the source code's contextual information to generate the probabilistic type candidates. Static tools such as *pytype* and *pyre* often ship with tailored type stubs, providing no support for user type stubs. The two tools also do not infer types for local variables, leaving class method calls hard to obtain. *pyright* [14], a type checking tool, enables support for using custom type stubs of external libraries, but, does not model library specific behavior leading to loss of recall. Moreover, *pyright* needs to be further re-engineered to obtain inferred type hints as it is designed for type checking [22]. Furthermore, the well-known open-source project *Jedi* [15] cannot analyze complex Python features, and suffers from performance issues.

PyCG is of no help here: it does not analyze calls to external libraries, instead ignores them. We attempted to force *PyCG* to analyze ML libraries such as *Numpy* and *Pandas*. Yet, we failed to obtain results due to crashes and out-of-memory exceptions. External libraries, especially ML libraries, can contain millions of lines of code and PyCG's fixed-point algorithm does not terminate within reasonable time and memory. Even after (unsoundly) limiting the number of iterations of PyCG's fixed-point algorithm, the resulting AG was unsuitable for real-world application because of the low precision and recall. An analysis of the ML libraries' code thus seems out of reach with current tooling. We further explore these limitations with a quantitative comparison of *PyCG*, *Jedi*, and *pyright* with HeaderGen in the evaluation Section VIII.

We thus instead designed a tool-assisted approximative technique for resolving return-types of function calls to external libraries. Figure 5 shows HeaderGen's approach for return-type approximation. First, we created a database of stub files for popular ML libraries such as *Keras*, *Numpy*, *Pandas*, etc. Stub files contain type hints defined relative to the original Python source code and stored as *.pyi* file. To

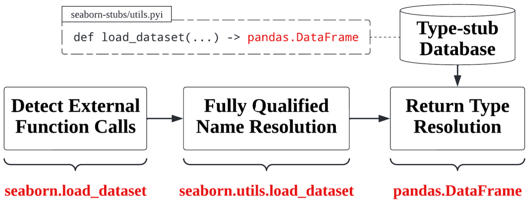

seaborn-stubs/utils.pyi
def load_dataset(...) -> pandas.DataFrame
Type-stub Database
Detect External Function Calls
Fully Qualified Name Resolution
Return Type Resolution
seaborn.load_dataset
seaborn.utils.load_dataset
pandas.DataFrame

Fig. 5. Workflow of imported library function return-type resolution.

build the database, we first created scaffolding .pyi files for all ML libraries we selected. This was followed by a manual inspection of function documentation and in some instances, confirmation by manual function execution to create type annotations for individual function calls. We note that this is still a work in progress and does not yet cover the entire source code of all the ML libraries that we selected. We intend to fully automate type-stub generation in the future utilizing type-inference systems such as pytype [20] which are currently under development. However, no accurate and maintained type-inference implementations for Python currently exists.

As shown on the bottom left of Figure 5, additional steps are required to make return-type resolution work for Python. We took for granted that `sns.load_dataset()` resolves to `seaborn.utils.load_dataset`. But looking at the example in Figure 2 at (C2,2), this fully qualified function name is not at all apparent. HeaderGen thus must implement two additional steps that resolve application-side function calls to their fully qualified names. First, the external function call in the notebook is resolved based on the import information and EAG. For instance, consider location (C1,2) in our motivating example. Here, `seaborn` is imported with alias `sns` and therefore the function call is resolved as `seaborn.load_dataset`, as shown in in red text at the bottom left of Figure 5. But in Python, top-level modules can access function definitions in submodules by transitive imports, mapping full path API names to shorter names. For instance, `seaborn` exports functions from submodules (*utils.py* here) that actually implement the function. Fortunately, given the fact, that the module `seaborn` has now been determined, HeaderGen can next perform a *dynamic* fully qualified name resolution using the builtin Python reflection mechanism `inspect`, and dynamic execution using the function `eval` on that module. During startup time, HeaderGen imports a selected set of popular ML libraries into memory. Then, during analysis, the `eval` function is used to dynamically evaluate strings as Python expressions. In our motivating example, a reference to the function returned by: `eval('seaborn.load_dataset')` is evaluated and stored. Note that the function `load_dataset` is not called—only a reference to the function is dynamically created. Further, this reference is examined using the builtin `inspect` module, which can retrieve information about live Python objects. HeaderGen uses it to fetch the location of the function's definition in the source code, i.e., the fully qualified name.

B. Flow-sensitive Callsite Extraction

The EAG generated in the previous step is used to construct a flow-sensitive CG using PyCG's CG construction algorithm. Wherein, the intermediate representation of the program is iterated while looking for callable objects based on the EAG and adding it to the CG. Then, the callsites are mapped according to the location of their definition in the notebook. This is achieved by mapping the line numbers of the Python script with the notebook during conversion.

In addition, note that when a user-defined function that is defined elsewhere in the notebook, say `x()`, is called from a code cell, any other function called from inside `x()` is also added as originating from that particular location in the notebook, i.e., the transitive closure of the CG. This step is needed to ensure that HeaderGen can annotate code cells that are only calling functions defined in some other code cell.

C. Jupyter Notebook Annotation

The goal of HeaderGen is to aid data scientists in easily navigating and comprehending undocumented notebooks. To this end, the callsite information output by HeaderGen's analyzer is used to add helpful information to the notebook. First, function calls found by the analysis are classified based on the ML operations in Figure 1. The classification is based on a manually curated database that maps individual API calls of popular ML libraries to ML operations. The ML operation mapping was created by manually inspecting the official function documentation and cross-referencing usages in the real-world. Some functions can be easily mapped to one of the operation categories. For instance, `pyplot.plot` is classified as *Visualization*. However, calls such as `numpy.reshape` can belong to both *Data Cleaning Filtering* and *Feature Transformation*, and therefore classified into both categories.

Pattern Matching. Notebooks can contain code cells that perform ML operations without explicit function calls, but rather, use other Python constructs that alter objects. For instance, consider the first pattern in Table I that represents a *Feature Engineering* operation, i.e., `df['xy'] = df.x * df.y`. Here, a new column *xy* is being created in the Dataframe object `df` by multiplying columns *x* and *y*. In absence of a function call, HeaderGen resorts to AST based pattern matching to identify ML operations. In this specific case, HeaderGen first consults the EAG to infer that the type of the variable `df` is a Dataframe. Then, both sides of the binary operator '*', i.e., `df.x` and `df.y`, are checked if they are indeed Dataframe accesses. From this, HeaderGen concludes that this statement is a *Feature Engineering* operation. Table I further lists some of the Dataframe access patterns that HeaderGen currently supports.

Text Annotation Generation. Based on this classification and pattern matching, the following annotations are added to the notebook: (1) Index of ML Operations, (2) Code cell headers, and (3) Table of contents.

1) Index of ML Operations: The index provides a clickable and nested list of all function calls in the notebook classified according to the taxonomy of ML operations shown in

TABLE I
DATAFRAME USAGE PATTERN MAPPED TO ML OPERATIONS

ID	Pattern	ML Operation
1	`df['xy'] = df.x * df.y`	Feature Engineering
2	`df.x = 1`	Feature Transformation Data Preparation
3	`df.x[df.x == 1] = 1`	Feature Transformation Data Preparation
4	`x = df.x[['f1', 'f2']]`	Feature Selection
5	`print(df[0:20])`	Exploratory Data Analysis

Figure 1. Figure III-C shows the index of ML operations generated for our motivating example. The index is displayed on top of the notebook using HeaderGen's notebook plugin. If no functions are found for a particular ML operation category, the category is displayed struck out. Each ML operation category and cell list can be expanded or collapsed as required. Function calls are organized based on the library as seen in the figure. Additionally, different areas of the notebook are hyperlinked, this makes it easy for the user to explore the notebook back-and-forth. For instance, cell 5 can be quickly visited by pressing *"goto cell # 5"* and back to the index again by pressing *"back to top"*.

2) Code cell headers: High-level ML operation categories from the taxonomy are added as headers for individual code cells. Note that when code cells contain ML operations from more than one category, all of these are added to the header. The headers can be further extended to see all the functions used in the following code cell, along with the docstrings that were fetched during analysis time from the source code.

3) Table of contents: Code cell headers are attached with anchors that allow in-page navigation. Using this information, the table of contents combines the headers of all code cells and adds an anchor-link to each entry. This simplifies access to relevant sections of the notebook based on the taxonomy.

IV. Evaluation

We evaluated HeaderGen to answer the following four research questions:

RQ1: *Does HeaderGen improve comprehension and navigation of undocumented Jupyter Notebooks?*

RQ2: *How accurate is HeaderGen's callsite recognition?*

RQ3: *How accurately can HeaderGen classify code cells using callsites?*

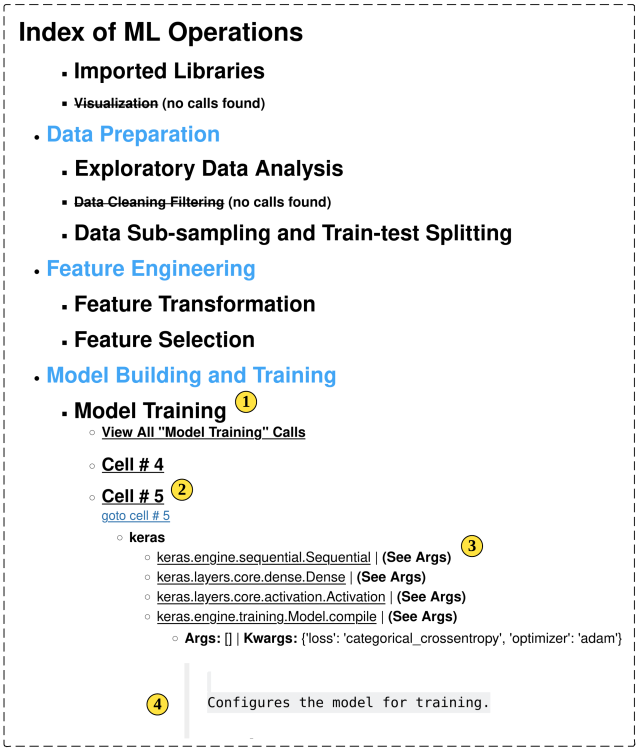
RQ4: *How does HeaderGen compare to other tools?*

We first describe the benchmarks we developed for evaluating HeaderGen, and then examine the research questions.

A. Benchmarks

We evaluate HeaderGen by building two benchmarks: (1) a micro-benchmark containing 121 notebooks, and (2) a real-world benchmark containing 15 notebooks from Kaggle.

Jupyter Notebook Micro-benchmark. We evaluate HeaderGen by adopting the benchmark created by Salis et. al [11] as part of PyCG. PyCG's benchmark does not have specific challenges targeting flow-sensitive analysis, and the benchmark contains ground truth only for flow-insensitive call-graphs. Yet, to evaluate HeaderGen's analysis, flow-sensitive callsite information is required, i.e., information about function calls associated with line numbers. To address this, we first converted Python scripts from PyCG's benchmark into notebooks, and then created ground truth by manually mapping callsites to line numbers. Furthermore, we created eight new test cases that have specific challenges to flow-sensitivity.



Index of ML Operations

- **Imported Libraries**
- ~~Visualization~~ (no calls found)
- Data Preparation
  - **Exploratory Data Analysis**
  - ~~Data Cleaning Filtering~~ (no calls found)
  - **Data Sub-sampling and Train-test Splitting**
- Feature Engineering
  - **Feature Transformation**
  - **Feature Selection**
- Model Building and Training
  - **Model Training**
    - View All "Model Training" Calls
    - Cell # 4
    - Cell # 5
      goto cell # 5
      - keras
        - keras.engine.sequential.Sequential | (See Args)
        - keras.layers.core.dense.Dense | (See Args)
        - keras.layers.core.activation.Activation | (See Args)
        - keras.engine.training.Model.compile | (See Args)
          - **Args:** [] | **Kwargs:** {'loss': 'categorical_crossentropy', 'optimizer': 'adam'}

```
Configures the model for training.
```

Fig. 6. Index of ML operations for our motivational example. ① ML operation category "Model Training" is expanded to view all code cells that are performing model training operations. ② Cell # 5 is expanded to view all function calls in the cell. ③ Fully qualified function names are displayed. ④ Expanded view showing the arguments used and its docstring.

Real-world Benchmark. To assess HeaderGen in real-world scenarios, we tested for precision and recall on 15 notebooks from Kaggle, a community where data-science practitioners come together to create and share ML based solutions written in notebooks. The platform hosts open competitions where data scientists around the world compete against each other to build the best solution. Kaggle encourages beginners to learn from experts in the field by making their submissions public. However, these notebooks often lack documentation. We found that 99 of the top 500 notebooks submitted to the most popular competition on Kaggle contained no markdown cell. Therefore, we base our real-world benchmark on these undocumented notebooks which are still being viewed.

We selected notebooks from three different and most popular competitions on Kaggle based on the number of

TABLE II
Evaluation of HeaderGen on our real-world benchmark for callsite recognition and header annotation

Competition	Name	Votes	Views	Callsite Recognition		Header Annotation	
				Precision	Recall	Precision	Recall
Titanic - Machine Learning from Disaster	bulentsiyah/keras-deep-learning-to-solve-titanic	65	1,926	100	90	71.4	100
	hongdnghuy/relu-sigmoid	13	693	100	100	80	100
	vaidicjain/titanic-easy-deeplearning-acc-78	9	449	95.8	95.8	100	87.5
	tanvikurade/complete-analysis-of-titanic	18	277	100	100	72.7	98
	alexanderbader/mytitanic	10	97	94.7	93.5	83.3	90.9
Predict Future Sales	econdata/predicting-future-sales-with-lstm	7	2,935	88.4	100	100	100
	lhavanya/predict-future-sales	3	457	94.3	91.7	85	100
	elvinagammed/stacked-lstm-top-5-4-mae	9	419	100	100	91.3	100
	ashishkapasiya/prediction-future-sales-with-keras	3	494	90.9	97.2	80.9	100
	the0electronic0guy/keras-begineer-friendly	12	264	100	100	82.2	98.7
Santander Customer Transaction Prediction	higepon/starter-keras-simple-nn-kfold-cv	20	4,145	100	87.5	61.1	100
	vishesh17/keras-nn-with-scaling-and-regularization	32	3,052	100	100	85.7	94.7
	christofhenkel/nn-with-magic-augmentation	19	1,408	94.2	94.3	100	100
	naivelamb/multibranch-nn-baseline-magic	10	569	96.6	94.6	64.5	87
	miklgr500/nn-embedding	10	502	91.2	93.9	74.2	95.8
		240	**17,687**	**96.4**	**95.9**	**82.2**	**96.8**
		Total		**Average**		**Average**	

submissions to encourage variation in the benchmark: (1) Titanic - Machine Learning from Disaster, (2) Predict Future Sales, and (3) Santander Customer Transaction Prediction. We downloaded the top 30 notebooks according to votes for each competition with the search term “Keras”. Since Keras [23] is a popular ML library among novices. We used the Kaggle API to search and download notebooks. All 30 notebooks from each competition were further filtered to target those without any markdown cells. Finally, we selected the top five most viewed notebooks from each competition. The selected notebooks in our benchmark are listed in Table II. These notebooks have a median of 20 code cells, compared to 13 cells that are found in real-world notebooks as reported by Pimentel et al. [7]. Note that these undocumented notebooks still have 240 upvotes and 17,687 views as of Octorber 2022.

The ground truth is then created manually by inspecting code cells in each notebook, and listing the fully qualified names of all function calls. Notebooks were executed cell-by-cell and dynamically analyzed using Python’s reflection module `inspect` to gather the fully qualified names. Multiple iterations were carried out to avoid errors in the ground truth.

V. RQ1: Comprehension and Navigation Study

The goal of HeaderGen is to increase comprehension and navigation in undocumented notebooks. We therefore conducted a user-study to quantitatively measure the improvements of HeaderGen over undocumented notebooks.

A. Study Design

The study is aimed at recreating the exploration of notebooks that data scientists routinely do. The study is designed as a within-subject study where the participants were given two notebooks from our real-world benchmark and asked to complete five comprehension tasks on each notebook one after the other. To minimize learning effects, we chose a latin-square design: participants were divided into two groups. While participants in group-1 were given the undocumented notebook first, followed by the HeaderGen annotated version, participants in group-2 saw the annotated notebook first. Each study was conducted in a one-on-one online session lasting about one hour using a video-conferencing tool. First, an overview of the study-protocol was presented to the participant including a walk-through of HeaderGen. Next, participants were provided access to the remote Jupyter instance along with a questionnaire containing step-wise instructions on how to proceed. Before proceeding to the study, participants were instructed to examine an example notebook annotated with HeaderGen in order to get them comfortable with the features. The entire session was recorded with the consent of the participant for further analysis. Upon completion of the comprehension tasks, participants were asked to fill a likert-scale questionnaire to understand the participant’s perception of improvements provided by HeaderGen. Finally, participants were asked if they had any general comments about the tool.

Comprehension Tasks. We created a set of tasks to simulate typical questions that arise when a data scientist is exploring an unseen notebook. The tasks were finalized after discussions with a data-science expert. For each task, participants were expected to select the right answers from all the choices given to them. Overall, six comprehension tasks were created, as listed in Table III. For each notebook given to the participant, five tasks from the table were assigned to them based on the relevance to the notebook.

Likert-scale Questionnaire. Following the completion of the session, participants were asked to rate the level of agreement to statements about the usefulness of HeaderGen. The level of agreement was based on a 5-point Likert scale, where “1” is *Strongly disagree* and “5” is *Strongly agree*. The statements given to the participants are listed in Table IV.

TABLE III
COMPREHENSION TASKS

Id	Question
Q1	What are the deep learning layers used in the model?
Q2	What are the different data cleaning & data preparation operations?
Q3	Which of the following cells are used for model building and model training?
Q4	Select ML and visualization libraries that are used in the notebook
Q5	What are the different visualizations used in the notebook?
Q6	How is the dataset split into test and train subsets?

TABLE IV
STATEMENTS ABOUT PERCEPTION OF USEFULNESS

Id	Statement
S1	The classification of cells according to ML phases and headers helped me navigate the undocumented notebook.
S2	The generated list of functions used in the notebook helped me understand the notebook better.
S3	The header annotations added to the notebook is rather hindering the understanding of the notebook.
S4	I would install HeaderGen if it is made available as a plugin.

B. Participants

The study comprised of eight participants. Three of them were master students from the computer science department, three of them were full-time employees working in the data-science domain, and two of them were computer science researchers. Students were recruited by contacting the group leaders in the data-science research department. Professional employees were contacted using Linkedin [24] based on their job titles. The researchers were contacted based on their publications in common research topics. Due to privacy concerns, information of the participants are omitted. Participation was voluntary and did not involve monetary incentives.

C. Metrics

(1) Time: Time taken to complete all five tasks per notebook.
(2) Accuracy: Inspired by a similar comprehension study by Adeli et. al. [25], the accuracy is measured using F1-score that takes into account both precision and recall.
(3) Navigability: The perceived navigability based on responses to Likert scale questions.
(4) Usefulness: The perceived usefulness based on responses to Likert scale questions.

D. Results

The study resulted in 80 $(8 * 5 * 2)$ measurements for accuracy, from eight participants performing five tasks on two treatments (undocumented and annotated), and 16 $(8 * 2)$ measurements for time, from two treatments. We compare accuracy and time measurements between treatments using the non-parametric two-sided Wilcoxon Signed Rank (WSR) test as the measurements between treatments are paired and the sample size is small. In addition, all measurements are analyzed based on descriptive statistics. Figure 7 shows the boxplot of accuracy scores, time measurements, and perception ratings.

Time. Both mean and median values of time taken for the annotated treatment (*mean=336.6s, median=328.5s*) are lower than the undocumented variants (*mean=486.4s, median=464.5s*). Moreover, WSR test on time measurements showed statistical significance (*p-value=0.025, statistic=34.0*). The large difference in completion time for the undocumented variant is associated with the back-and-forth navigation in the notebook trying to find relevant areas. This shows that participants took significantly more time to complete comprehension tasks when given an undocumented notebook.

Accuracy. The mean accuracy of all comprehension tasks was greater for the annotated treatment, except for task T6, where it was equal. The variance of accuracy across the tasks was three times higher for the undocumented treatment, showing that it is more likely to yield better accuracy with annotated notebooks. However, the median is greater for the annotated treatment only in T4 and T5. In addition, WSR test showed that the accuracy scores from the study are not statistically significant between two treatments (*p-value=0.106, statistic=55.0*). Nonetheless, note that the study was not time-boxed. Participants thus took significantly longer to solve the tasks correctly for undocumented notebooks.

Navigability and Usefulness. The perceived ratings for statements in Table IV showed that the participants find HeaderGen considerably helpful in completing the tasks. None of the participants disagreed to statements S1, S2 and S4, and none of them agreed to statement S3. All participants showed interest in actually installing the tool when it is published.

Qualitative Results. Participants noted that HeaderGen would be especially useful when dealing with very large undocumented notebooks as it provides a "map" of the notebook. Participants also found the function documentation to be useful, given that the libraries are continuously evolving and that they would often come across methods that they have not seen before. Furthermore, minor recommendations to improve the taxonomy categories were noted and added to the final version. Recommendations to change the layout of the plugin were also noted and will be considered in future versions.

Threats to Validity. The study we conducted is prone to some common limitations of conducting user studies. Due to the small number of participants, it may not be representative of a larger population. However, participants were selected from all fields: students, professionals, and academics to get inputs from different perspectives. Furthermore, since the study follows a within-subject design, the order of tasks and treatments can have an effect on the outcome. Therefore, to limit the learning effect, we use latin-square design to randomize the order of treatments, tasks, and multiple choices. However, using notebooks that only use the Keras API might have had a learning effect as the study progressed. Although the participants were experienced working with the default notebook environment, HeaderGen adds additional interfaces that might seem confusing at first. As a result, some participants did not make full use of HeaderGen's capabilities.

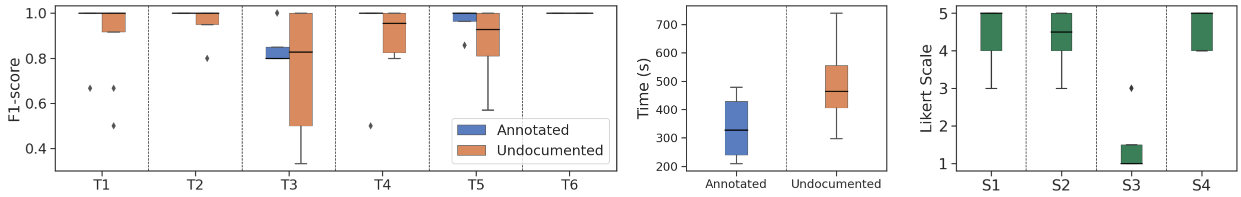


Fig. 7. **Left:** Box plots of accuracy for participant responses grouped by treatment. **Center:** Box plots of time measurements for two treatments. **Right:** Box plots of responses to likert-questions about perception.

VI. RQ2: Accuracy of Callsite Recognition

Micro-benchmark Results. We evaluate HeaderGen for complete and sound recognition of callsites. The analysis is complete when there are no false positives, and sound when there are no false negatives. In total, the analysis is sound in 113 of 121 cases, and complete in 113 of 121 test cases. Lack of soundness in eight of 121 test cases are due to the lack of implementation for analyzing challenging Python features such as decorators. On the other hand, out of the eight test cases that are incomplete, only three of them are due to missing implementation of challenging features. The remaining five test cases are not complete because our analysis is context-insensitive. As a result, it over-approximates the solution in certain scenarios.

Note that we do not perform a direct comparison of HeaderGen with PyCG because the micro-benchmark does not pose specific challenges to flow-sensitivity, except for the new *flow_sensitive* category with eight test cases that we added. When compared to PyCG for this category, as expected, PyCG is incomplete for all eight test cases. Furthermore, note that this micro-benchmark contains no challenges associated with handling external library function calls.

Real-world Benchmark Results. Table II lists the precision and recall values of HeaderGen for real-world notebooks.

HeaderGen achieves an average of 96.4% precision and 95.9% recall. Note that in four instances, the analysis achieves 100% precision and recall.

The precision loss is due to our type-stub database's over-approximation of return-types. For instance, a call `x.isnull()` can be either `Series.isnull` or `DataFrame.isnull`, depending on whether `x` is a *Series* or *Dataframe*, which is determined based on the underlying structure of the data. However, this is not straight forward to infer and needs advanced data-flow analysis.

Where recall is lost, it is because our analysis lacks supports for some complex Python features.

VII. RQ3: Accuracy of Generated Headers

HeaderGen uses identified function calls in code cells to automatically add relevant headers based on the taxonomy of ML operations. We evaluated the headers generated by HeaderGen for precision and recall against manually annotated headers. Again, we use our real-world benchmark as a basis. 15 notebooks from the benchmark were divided and assigned to four data scientists working in the industry for manual annotation of each code cell. Notebooks were distributed such that each notebook was seen by at least two reviewers. Based on the taxonomy of ML operations, each annotator inspected and classified each code cell into relevant categories. The inter-rater reliability score, as measured by Cohen's kappa coefficient [26], was improved by conducting follow-up interviews with all four reviewers. Finally, a score of 0.89 was achieved, which signals an almost perfect agreement.

TABLE V
Comparison with existing tools on our real-world benchmark

Tool	Callsite Recognition		Header Annotation	
	Precision	Recall	Precision	Recall
HeaderGen	96.4	95.9	82.2	96.8
Pyright	96.7	87.2	83.8	82.7
Jedi	84.6	65.8	85.1	69.8
PyCG	41.7	23.3	84.6	26.2

Results. The resulting precision and recall are listed on the right side of Table II. The headers that are generated by HeaderGen are matched on the high-level categories of the taxonomy listed in Figure 1. HeaderGen achieves a precision of 82.2% and recall of 96.8%. Precision is lost because some functions can be mapped to more than one ML operation.

VIII. RQ4: Comparison with Existing Tools

We compare HeaderGen in terms of callsite recognition and header annotation with *PyCG*, *pyright*, and *Jedi* using our real-world benchmark. Since both *pyright* and *Jedi* are designed for type checking and auto-completion, we added helper functions to output type information and callsite information as required by HeaderGen. Furthermore, note that our type-stub database of ML libraries was provided to *pyright* and *Jedi* for analysis.

Results. The precision and recall values are listed in Table V. Since header annotation is based on identified callsites, it is evident that higher recall of callsite recognition leads to higher recall in header annotation. HeaderGen achieves the highest recall of 95.9% which leads to a 96.8% recall in header annotation of code cells. However, *pyright* is the closest with 87.2% recall for callsite recognition which leads to 82.7% recall for header annotation. Note that without our type-stub database, these tools would perform even worse.

The loss of precision is attributed to the over-approximation of return-types in our type-stub database as discussed earlier.

Modeling of Pandas Behavior. Listing 1 shows simplified data manipulation methods of the Pandas library based on our real-world benchmark. Furthermore, Table VI lists the type of each variable used in Listing 1 as inferred by the tools being compared. It can be seen that both *pyright* and *Jedi* fail to infer return-types of variables `x1` through `x6`. This is because HeaderGen can model complex pandas accesses while the other two tools fail. For instance, in line 6, a dot notation access `df.a` is ignored by other tools while HeaderGen models it as a `Series`.

```
1 import pandas as pd
2
3 df = pd.read_csv("./input.csv")
4 x1 = df["a"].map(lambda x: x + 1.0)
5 x2 = df.iloc[[False]].reset_index().copy()
6 x3 = df.a.fillna(0)
7 x4 = df.groupby(["a"])[["b"]].agg({"b": ["min"]})
8 x5 = df[["b", "c"]]
9 x6 = df.c.values.astype(int)
```

Listing 1. Common uses of Pandas DataFrame that existing tools fail to infer.

TABLE VI
COMPARISON OF TYPE INFERENCE BY EXISTING TOOLS FOR LISTING 1

Var	Actual	HeaderGen	Pyright	Jedi
df	DataFrame	DataFrame	DataFrame	DataFrame
x1	Series	Series	Any	Any
x2	DataFrame	DataFrame	Any	Any
x3	Series	Series	Any	Any
x4	DataFrame	DataFrame	Any	Any
x5	DataFrame	DataFrame	Any	Any
x6	Ndarray	Ndarray	Any	Any

IX. Related Work

Tool-support for Jupyter Notebooks. In recent years, many publications [2], [5]–[8], [27], [28] have experimentally analyzed notebooks to gather insights on coding patterns and highlight that notebook quality is poor and needs attention from the software engineering community. However, there has been little research into developing tools to help with the highlighted issues. To this end, Wang et al. [10] propose *Themisto*, a tool that encourages data scientists to write documentation for code cells by first applying a deep learning based approach to automatically generate documentation in natural language and then recommending to the user whether to adopt it or use it directly. *Themisto* directly uses AST of the Python code to train its model and does not explore using SA based approaches to extract contextual information from source code. To this end, we expect that analysis results from HeaderGen can aid deep learning based approaches to achieve better results. In another study, Pimentel et al. [7] studied 1.4 million notebooks for features that affect reproducibility and suggested a set of best practices. Following this, Wang et al. [29] propose *Osiris*, a tool-based approach to restore reproducibility in notebooks by using AST parsing for data-flow analysis to find dependencies of variables between code cells. Furthermore, Yang et al. [22] design a SA approach to detect data leakage in notebooks. Our work automatically annotates code cells and provides tool-support for literal programming.

Static Analysis for Python. Although Python is one of the most popular programming languages, there is still a shortage of SA infrastructure for Python as noted by Yang et al's [13] empirical investigation of Python's features. Yang et al. further argue that analysis for Python cannot adopt algorithms developed over past decades of scientific research due to its unique language features. Dynamic features such as duck typing in Python that make it stand out for fast-prototyping result in difficulties for its analysis. Call graph construction, which is the foundational technique in SA, remained an open problem until 2021 when a practical call graph generation approach named PyCG was offered [11]. However, the call graph generator does not consider flow of values and has no support for Jupyter notebooks. Moreover, there is still no general-purpose SA framework for Python that can provide data flow IRs. The closest one is the Scalpel project [30]. Nevertheless, Scalpel does not infer return-types for external function calls and does not take notebook cells into no consideration. In this work, we supplement existing SA work for real-world application by offering return-type resolution of external library API and flow-sensitive function callsite extraction using def-use relations.

X. Limitations & Future work

To obtain sound function name resolution, our approach uses the reflection mechanism from Python runtime, which somewhat reduces the coverage of APIs depending on the actual library version installed. We will explore static API mapping techniques to solve transitive imports in Python to address this. Second, our approach currently relies on manual classification of library function calls to ML operations. To address this, we are currently investigating natural language processing techniques to automatically classify library functions to ML operations based on function docstrings. Lastly, our analysis is limited to the scope of machine learning applications. However, the framework we designed is not limited to a particular scope. HeaderGen can annotate notebooks with domain-specific return-type stubs and library taxonomy.

Additionally, input from HeaderGen can be used to automatically restructure code cells in notebooks for better readability. For instance, by splitting up complex code cells performing multiple ML operations into sequential code cells. Furthermore, fast and precise function call analysis of HeaderGen can facilitate large-scale mining studies of Python code base.

XI. Conclusion

Many notebooks encountered in the wild are undocumented, making program comprehension and navigation difficult. To this end, HeaderGen utilizes precise static analysis to automatically annotate notebooks with structural headers based on a taxonomy of machine learning operations. HeaderGen achieved high precision and recall on both of our micro and real-world benchmarks. We further showed that HeaderGen can annotate headers with adequate precision and high recall when evaluated against ground truth manually curated by experts. Finally, we conducted a user-study to demonstrate that data scientists found HeaderGen to be helpful in improving program comprehension and navigation.

References

[1] D. E. Knuth, “Literate Programming,” *The Computer Journal*, vol. 27, no. 2, pp. 97–111, Jan. 1984.

[2] A. Rule, A. Tabard, and J. D. Hollan, “Exploration and Explanation in Computational Notebooks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–12.

[3] J. Wagemann, F. Fierli, S. Mantovani, S. Siemen, B. Seeger, and J. Bendix, “Five guiding principles to make jupyter notebooks fit for earth observation data education,” *Remote Sensing*, vol. 14, no. 14, p. 3359, 2022.

[4] S. Samuel and D. Mietchen, “Computational reproducibility of jupyter notebooks from biomedical publications,” *arXiv preprint arXiv:2209.04308*, 2022.

[5] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, “The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–11.

[6] J. Wang, L. Li, and A. Zeller, “Better code, better sharing: On the need of analyzing jupyter notebooks,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER ’20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 53–56.

[7] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, “A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, May 2019, pp. 507–517.

[8] L. Quaranta, F. Calefato, and F. Lanubile, “Eliciting Best Practices for Collaboration with Computational Notebooks,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 6, no. CSCW1, pp. 87:1–87:41, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3512934

[9] “Kaggle: Your Machine Learning and Data Science Community,” https://www.kaggle.com/, (accessed 2022-06-10).

[10] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, “Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks,” *ACM Transactions on Computer-Human Interaction*, vol. 29, no. 2, pp. 17:1–17:33, Jan. 2022.

[11] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, “PyCG: Practical Call Graph Generation in Python,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE, May 2021, pp. 1646–1657.

[12] S. Kummita, G. Piskachev, J. Späth, and E. Bodden, “Qualitative and Quantitative Analysis of Callgraph Algorithms for Python,” in *2021 International Conference on Code Quality (ICCQ)*, Mar. 2021, pp. 1–15.

[13] Y. Yang, A. Milanova, and M. Hirzel, “Complex Python Features in the Wild,” 2022.

[14] “Static type checker for python.” [Online]. Available: https://github.com/microsoft/pyright

[15] D. Halter, “Jedi - an awesome autocompletion, static analysis and refactoring library for Python,” Sep. 2022.

[16] K. Kennedy, “Use-definition chains with applications,” *Computer Languages*, vol. 3, no. 3, pp. 163–179, Jan. 1978.

[17] serge-sans-paille, “Gast, Beniget!” Apr. 2022.

[18] “PEP 484 – Type Hints — peps.python.org,” https://peps.python.org/pep-0484/#stub-files, (accessed 2022-05-26).

[19] L. Di Grazia and M. Pradel, “The evolution of type annotations in python: An empirical study,” in *Proceedings of the 30th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2022.

[20] “pytype.” [Online]. Available: https://github.com/google/pytype

[21] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, “Typilus: Neural type hints,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, pp. 91–105. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3385997

[22] C. Yang, R. A. Brower-Sinning, G. A. Lewis, and C. Kästner, “Data leakage in notebooks: Static detection and better processes,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[23] “Keras: The Python deep learning API,” https://keras.io/, (accessed 2022-06-22).

[24] “LinkedIn,” https://www.linkedin.com, (accessed 2022-06-22).

[25] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma, “Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Aug. 2020, pp. 1–10.

[26] J. Cohen, “A Coefficient of Agreement for Nominal Scales,” *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, Apr. 1960.

[27] A. P. Koenzen, N. A. Ernst, and M.-A. D. Storey, “Code Duplication and Reuse in Jupyter Notebooks,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Aug. 2020, pp. 1–9.

[28] W. Epperson, A. Wang, R. DeLIne, and S. Drucker, “Strategies for Reuse and Sharing among Data Scientists in Software Teams,” in *ICSE 2022*, May 2022.

[29] J. Wang, T.-y. Kuo, L. Li, and A. Zeller, “Assessing and restoring reproducibility of Jupyter notebooks,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Virtual Event Australia: ACM, Dec. 2020, pp. 138–149.

[30] L. Li, J. Wang, and H. Quan, “Scalpel: The Python Static Analysis Framework,” *Li Li*.

Acknowledgment

Funding for this study was provided by the Ministry of Culture and Science of the State of North Rhine-Westphalia under the SAIL project with the grand no NW21-059D.

Ministry of Culture and Science
of the State of
North Rhine-Westphalia

PADERBORN
UNIVERSITY