# Static analysis driven enhancements for comprehension in machine learning notebooks

Ashwin Prasad Shivarpatna Venkatesh[1] · Samkutty Sabu[1] · Mouli Chekkapalli[1] · Jiawei Wang[2] · Li Li[3] · Eric Bodden[4]

## Abstract

Jupyter notebooks have emerged as the predominant tool for data scientists to develop and share machine learning solutions, primarily using Python as the programming language. Despite their widespread adoption, a significant fraction of these notebooks, when shared on public repositories, suffer from insufficient documentation and a lack of coherent narrative. Such shortcomings compromise the readability and understandability of the notebook. Addressing this shortcoming, this paper introduces HEADERGEN, a tool-based approach that automatically augments code cells in these notebooks with descriptive markdown headers, derived from a predefined taxonomy of machine learning operations. Additionally, it systematically classifies and displays function calls in line with this taxonomy. The mechanism that powers HEADERGEN is an enhanced call graph analysis technique, building upon the foundational analysis available in *PyCG*. To improve precision, HEADERGEN extends PyCG's analysis with return-type resolution of external function calls, type inference, and flow-sensitivity. Furthermore, leveraging type information, HEADERGEN employs pattern matching techniques on the code syntax to annotate code cells. We conducted an empirical evaluation on 15 real-world Jupyter notebooks sourced from Kaggle. The results indicate a high accuracy in call graph analysis, with precision at 95.6% and recall at 95.3%. The header generation has a precision of 85.7% and a recall rate of 92.8% with regard to headers created manually by experts. A user study corroborated the practical utility of HEADERGEN, revealing that users found HEADERGEN useful in tasks related to comprehension and navigation. To further evaluate the type inference capability of static analysis tools, we introduce TYPEEVALPY, a framework for evaluating type inference tools for Python with an in-built micro-benchmark containing 154 code snippets and 845 type annotations in the ground truth. Our comparative analysis on four tools revealed that HEADERGEN outperforms other tools in exact matches with the ground truth.

**Keywords** Static analysis · Python · Comprehension · Type inference · Benchmark

🖄 Springer

# 1 Introduction

In the evolving landscape of machine learning (ML) and data science, Jupyter Notebooks have emerged as the predominant platform for creating ML solutions within the ML community. These notebooks resonate with the paradigm of *literate programming* postulated by Knuth (1984). This approach advocates the integration of code, comprehensive documentation, and visual representations within a unified document to foster understanding and facilitate the sharing of intricate solutions. The underlying principles of literate programming include: (1) Augmenting code with descriptive text and illustrative diagrams. (2) Imposing a coherent narrative by separating code cells with pertinent headers. (3) Logically segmenting and labeling the program's reusable modules.

Within notebooks, code is written in executable *code cells* while accompanying documentation is written in *markdown cells*. An illustrative example of this configuration, showcasing Python code cells interleaved with markdown cells, is depicted in Fig. 2. Note that Python has emerged as the predominant language of choice for articulating ML-based solutions in notebooks (Rule et al. 2018).
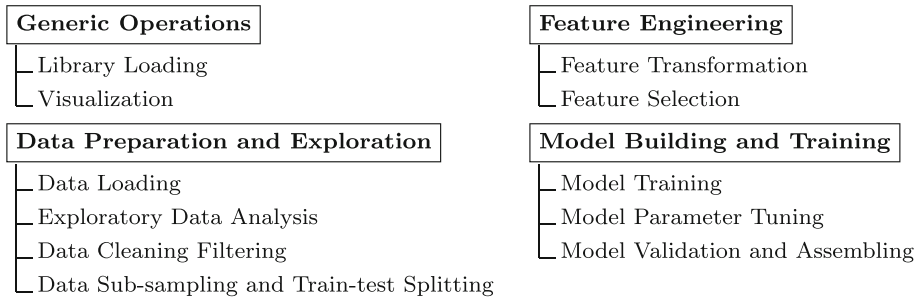
Augmenting code segments with explanatory textual content enhances the overall comprehensibility of notebooks and further promotes collaboration (Wagemann et al. 2022). Moreover, a markdown-to-code cell ratio of 2, as posited by Wagemann et al. (2022), serves as an indication of commendable literate programming adherence. This assertion finds further support in the work of Samuel and Mietchen (2022), who report that a higher markdown-code cell proportion correlates with superior reproducibility, a vital metric in scientific studies.

However, despite the inherent capabilities of Jupyter notebooks resonant with literate programming principles, real-world adoption often diverges from this ideal (Kery et al. 2018). Empirical studies reveal that code-smells and suboptimal practices are common in publicly distributed notebooks (Wang et al. 2020b). Interviews conducted by Rule et al. (2018) with ML practitioners showed that they often defined their notebooks as personal unstructured scratch-pads and "messy". The authors found that the reluctance in annotating notebooks has been attributed to time constraints or being "too lazy". Subsequent large-scale research by Pimentel et al. (2019) underscored that 30.93% of 1.4 million notebooks they examined lack markdown cells, a finding corroborated by Quaranta et al. (2022). This scarcity of annotations reveals the prevailing negligence towards best practices. Such omissions are especially detrimental in platforms like Kaggle[1], where subpar practices risk propagating to the next generation of ML practitioners.

Therefore, there exists an imperative for the software engineering research community to develop tools that address this problem. To this end, this paper introduces HEADERGEN, a tool-based approach to augment the comprehension and navigation of undocumented Python based Jupyter notebooks by automatically creating a narrative structure in the notebook.

The work of Wang et al. (2022) has shed light on the structured taxonomy of ML operations. As demonstrated in Fig. 1, the process undertaken by data scientists when developing an ML-based notebook typically begins with data preparation, followed by feature extraction, and continues with the creation and training of the model. Within this structured process lies an implicit narrative structure that is key to the operation of HEADERGEN. We specifically designed HEADERGEN to capture this narrative by accurately detecting each function call in the ML notebook. Subsequently, it classifies each call according to the ML operations taxonomy. Utilizing this classified data, HEADERGEN creates a structural map of the entire notebook which is then presented as an *"index of ML operations"* at the top of the notebook.

---

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

| Generic Operations | Feature Engineering |
|---|---|
| └ Library Loading | └ Feature Transformation |
| └ Visualization | └ Feature Selection |

| Data Preparation and Exploration | Model Building and Training |
|---|---|
| └ Data Loading | └ Model Training |
| └ Exploratory Data Analysis | └ Model Parameter Tuning |
| └ Data Cleaning Filtering | └ Model Validation and Assembling |
| └ Data Sub-sampling and Train-test Splitting | |

**Fig. 1** Taxonomy of machine learning operations based on (Wang et al. 2022)

This is further complemented by annotating every code cell with relevant markdown headers that highlight the ML operations being performed (see example in page 18).

To yield useful results, HEADERGEN requires a fast and accurate program analysis that can precisely identify all function calls in the notebook. An exploration into existing methodologies revealed an absence of a technique that could statically identify all function calls in a notebook with acceptable precision, recall, and execution time. This can be attributed to Python's inherent complexities such as duck typing, dynamic code execution, reflection, among others that are challenging to static analysis (Salis et al. 2021; Kummita et al. 2021). Moreover, in contrast to other programming languages like Java, Python lacks a lot of tool-support for state-of-the-art static analysis (SA) techniques (Yang et al. 2022b). The prevailing tools for Python predominantly rely on rudimentary analyses of its abstract syntax trees (AST). Compounding this issue is Python's dynamically-typed nature, necessitating a precise static type-inference for variables to facilitate accurate static analysis. A recent advancement in the form of a call-graph generation technique, PyCG (Salis et al. 2021), which operates on an intermediate representation of the AST and addresses many complex Python features, also falls short. Specifically, PyCG ignores analyzing function calls to external libraries and presents a flow-*insensitive* analysis. Flow sensitivity in SA refers to the ability of the analyzer to consider the order in which statements occur in the program. A flow-sensitive analysis tracks the flow of control through the program. This means the analysis can differentiate between the different states of variables at different points in the program. Flow-insensitivity of PyCG limits its capability to accurately extract function calls in real-world notebooks. HEADERGEN rectifies these limitations.

To summarize, the challenge that HEADERGEN addresses is two-fold:

1. **Inadequacies in 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 analysis in PyCG with the ability to resolve function calls to external libraries, infer types of variables, and adds flow-sensitivity.
2. **Lack of Documentation in Notebooks:** a considerable portion of publicly accessible notebooks lack adequate documentation. This absence not only impedes comprehension but also violates the principles of literate programming. HEADERGEN employs precise static analysis to automatically augment these notebooks with structural headers. This addition creates a narrative structure to aid comprehension of undocumented notebooks.

To evaluate the performance of HEADERGEN's static function call analyzer, we employed an enhanced version of PyCG's micro-benchmark, complemented by a real-world benchmark consisting of 15 notebooks sourced from Kaggle. On the real-world benchmark, HEADERGEN

achieved 95.6% precision and 95.3% recall, outperforming PyCG and other function call analyzers based on off-the-shelf tools such as *pyright*[2] and *Jedi*[3]. On the same benchmark we also evaluated HEADERGEN for header annotation and achieved 85.7% precision and 92.8% recall. Additionally, through a user study involving eight data science practitioners, we gathered evidence indicating that HEADERGEN significantly enhances navigation speed and improves comprehension.

Static function call analysis is dependent on the analyzer's capability to statically infer types of variables. Therefore, we systematically evaluated the type inference capabilities of these tools. To facilitate this, we created, TYPEEVALPY, the first micro-benchmarking framework specifically designed for type inference evaluation in Python. TYPEEVALPY contains 154 code snippets, organized into 18 categories, each focusing on distinct language-specific features of Python, summing up to a comprehensive 845 type annotations. Notably, HEADERGEN registered 564 exact matches against the 845 annotations in the ground truth, a performance that notably surpasses other tools.

The primary contributions of our work are as follows:

– We propose a novel static analysis based approach for Python Jupyter notebooks that can automatically enhance them with structural, explanatory, and navigational annotations to augment literal programming practice.
– We implement a static function call extraction technique for Python with 95.6% precision and 95.3% 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.

This manuscript builds upon the findings presented in our initial publication (Venkatesh et al. 2023a). Here, we detail the incremental advancements made since our initial publication:

– We introduce TYPEEVALPY, the first micro-benchmarking framework for type inference evaluation in Python containing 154 code snippets and 845 manually annotated types.
– A detailed evaluation of several type inference tools using TYPEEVALPY, including HEADERGEN. This assessment demonstrates how improved type inference enhances call-site recognition and header generation in Jupyter notebooks.
– The original dataset is expanded to include 15 new Jupyter notebooks from real-world projects, involving detailed manual annotations to identify headers and function callsites, and to extract fully qualified names. This validates HEADERGEN's performance on a broader and more diverse dataset.
– To overcome the limitation in the original HEADERGEN approach concerning the manual curation of a database that maps ML API calls to an ML taxonomy, we developed an ML model that classifies function calls using their documentation strings. Trained on a dataset curated by four data science experts, this model enhances HEADERGEN's adaptability to evolving ML libraries and functions.

The remainder of this paper is organized as follows: we present challenges in statically analyzing Python code with a motivating example in the Section 2, followed by detailing our design in Section 3. We then present the research questions in Section 4. We discuss the details of our micro-benchmark, real-world benchmark, and TYPEEVALPY in Section 5.

---

[2] https://github.com/microsoft/pyright

[3] https://github.com/davidhalter/jedi

We address the research questions from Sections 6 to 10, and discuss existing research in Section 11. The limitations of HEADERGEN is discussed in Section 12 and finally the paper is concluded in Section 13.

**Availability**  HEADERGEN is published on GitHub as open-source software under Apache 2.0 license:
https://github.com/secure-software-engineering/HeaderGen.

TYPEEVALPY is published on GitHub as open-source software under Apache 2.0 license: https://github.com/secure-software-engineering/TypeEvalPy.

## 2 Motivating example

As a motivating example, consider the notebook in Fig. 2. It consists of one markdown cell which is rendered as an HTML header, and five code cells that can be identified by the

Example Header  ← Markdown cell                                      Code cell

```
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** Example of a Jupyter notebook containing a machine learning solution

comments in the first line of each code cell. The example notebook in Fig. 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 (Pimentel et al. 2019). 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 (Pimentel et al. 2019).

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. This is further complicated with wild card imports of the form "`from MODULE import *`".
- **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 unless the developer manually annotates the function definition. Unfortunately, widespread adoption of type annotation is still lacking in practice (Di Grazia and Pradel 2022a). 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. While the syntactic chaining requires careful modeling by the analyzer, the main challenge is that the imprecision of analysis is propagated through these call chains. 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*.

Note that in general, analyzing dynamic programming languages such as Python poses several other challenges not discussed in our example. Features such as dynamic evaluation where a string can be evaluated as a code fragment at runtime and complex control-flow structures with generators also pose challenges to SA.

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.
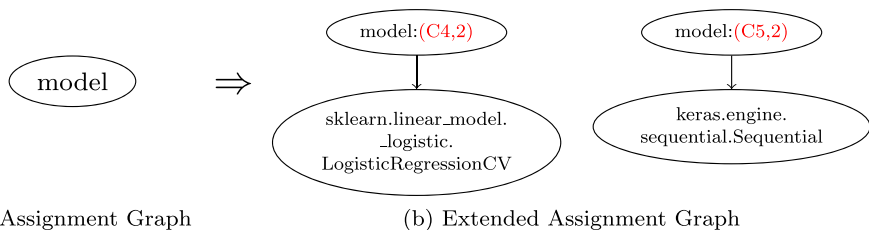
**Fig. 3** High-level overview of HEADERGEN

# 3 Approach

A comprehensive overview of HEADERGEN is shown in Fig. 3. The initial step involves transforming a notebook into a standard Python script. This process aids in eliminating metadata from the notebook, ensuring that only pertinent information remains for subsequent analysis. Post this conversion, HEADERGEN proceeds to analyze the resultant Python script to create an extended assignment graph (EAG). Leveraging the EAG, HEADERGEN extracts flow-sensitive callsite information. The final phase involves enriching the notebook with relevant headers corresponding to the callsites by using an ML-based taxonomy classifier. Additionally, an index of ML operations is generated.

The details of the EAG construction and the methodology for extracting flow-sensitive callsite information are discussed in Sections 3.1 and 3.2. The ML-based taxonomy classifier is elaborated in Section 3.3. Subsequently, Section 3.4 outlines the procedure employed for annotating the notebook, utilizing the outputs from the analyzer.

## 3.1 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 (Salis et al. 2021). 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



(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)

example, without analyzing the function `load_dataset()` 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 first defined in cell 4 and then redefined in cell 5 (cf. Fig. 2), however, the generated AG shown in Fig. 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.

### 3.1.1 Definition-use chain for flow-sensitivity

A definition-use chain (DUC) (Kennedy 1978) 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 (serge-sans-paille 2022), 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 Fig. 4b captures multiple definitions of the `model` variable separately.

### 3.1.2 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 (Fig. 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[4] 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 (Di Grazia and Pradel 2022b).

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* Pyt 2022) 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 (Allamanis et al. 2020) and Type4Py (Mir et al. 2022) often only leverage the
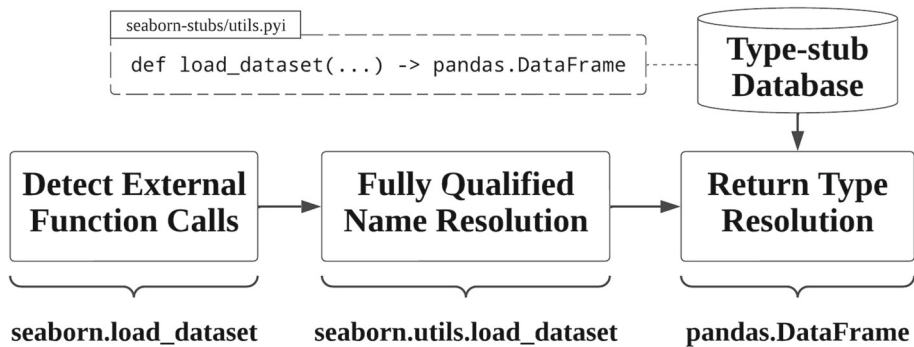
---

[4] https://peps.python.org/pep-0484/

**Table 1** Overview of cross-language usage in machine learning libraries

| Library | Python | C |
|---|---|---|
| Pandas | 90.4 | 1.6 |
| Sklearn | 92.2 | 0.3 |
| Matplotlib | 92.6 | 0 |
| Numpy | 60 | 35.7 |
| Seaborn | 100 | 0 |
| Keras | 99.9 | 0 |
| LightGBM | 20.4 | 3.2 |
| XGBoost | 20.6 | 0 |
| TensorFlow | 26.6 | 0 |
| Plotly | 98.1 | 0 |
| Torch | 50 | 2 |
| Statsmodels | 92.8 | 0 |

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* (Sta 2022), 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 (Yang et al. 2022a). The well-known open-source project *Jedi* (Halter 2022) cannot analyze complex Python features, and suffers from performance issues. Furthermore, *MOPSA* (MOP 2024) is a generic static analysis framework with a focus on analyzing Python programs that use external libraries written in C. It is focused on finding type and value errors. MOPSA uses formal methods and proposed Python-C semantics to model program states in C and Python side. However, C Code is rather not dominant (e.g. XGBoost and LightGBM) in our context. MOPSA, however, focuses on analyzing Python-C code only. We summarize the usage of cross-language usages in Table 1. The percentage of each language usage is retrieved from their official GitHub page.

*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*, *pyright*, and *HiTyper* (Peng et al. 2022) with HEADERGEN in Sections 9 and 10.

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 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

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

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-checking systems such as pytype which currently output conservative results that are not usable for our use-case (Guo et al. 2024). However, no accurate and maintained type-inference implementations for Python currently exists.

As shown on the bottom left of Fig. 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 Fig. 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 at the bottom left of Fig. 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.

### 3.1.3 Type information

Additionally, HEADERGEN gathers type data for variables found within the program and incorporates this information into the EAG. This process involves considering types from

both external sources, identified via return-type resolution and local variables. To facilitate this, we augment PyCG's fixed point algorithm to consider types of variables.

When a variable holds a literal value (such as `str`, `int`), Python's built-in reflection mechanism is used to infer the type of the variable. On the other hand, when the variable is an instance of a user-defined class in the program, the EAG is referred to determine the type of the variable. This is especially useful when HEADERGEN needs information about variable types to perform heuristic pattern matching on the source code (see Section 3.4). Moreover, we posit that having access to flow-sensitive type information of variables can be useful for a broader analysis use case.

## 3.2 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.

## 3.3 Taxonomy classifier

Within the Python environment, there exists a vast array of APIs tailored for diverse ML tasks. These APIs can be classified based on the specific ML operation they perform. In HEADERGEN, the callsite information obtained from the earlier stage assists in categorizing code cells by the API calls they contain. In our previous work (Venkatesh et al., 2023b), HEADERGEN relied on a manually curated database of ML APIs to ML operations. However, this is not maintainable considering the fast development cycles of ML libraries. Therefore, to aid this process, we developed a machine learning-based classifier for ML taxonomies. This classifier categorizes ML APIs into distinct ML operations using their documentation strings (doc strings). In this section, we discuss the process we followed for training this classifier model.

### 3.3.1 Dataset preparation

In the process of constructing a supervised machine learning model, the initial requisite is a well-labeled dataset. Given the absence of any publicly accessible datasets that maps API calls with ML operations, our first step was to establish such a dataset. To accomplish this, we used Kaggle, a platform that offers a wide range of resources such as datasets and notebooks. The ML notebooks on Kaggle serve as a valuable resource for practical code implementations with API calls from diverse libraries. We retrieved 6,698 notebooks from Kaggle's top six competitions, leveraging its API. The rationale behind selecting these specific competitions was their widespread popularity based on the number of submissions and participating teams. A breakdown of these competitions and notebooks is listed in Table 2.

**Table 2** Overview of Kaggle competitions and notebooks

| Kaggle Competition | Teams | Notebooks |
|---|---|---|
| Predict Future Sales | 15,656 | 833 |
| Titanic - Machine Learning From Disaster | 14,560 | 1,089 |
| Santander Customer Transaction Prediction | 8,751 | 907 |
| Home Credit Default Risk | 7,176 | 900 |
| IEEE-CIS Fraud Detection | 6,351 | 820 |
| M5 Forecasting - Accuracy | 5,558 | 608 |
| **Total Notebooks** | | 6,698 |

We used HEADERGEN to analyze the selected notebooks. Our primary objective was to extract the fully qualified names associated with all API calls, as well as their corresponding doc strings. This data forms the foundational basis for our dataset.

**Empirical analysis** HEADERGEN identified a total of 141,657 API calls from 12 different ML libraries. After removing the duplicates, we were left with 2,553 unique API calls. The breakdown of this is listed in Table 3.

**Data annotation** To ensure the accuracy and reliability of our annotations, we engaged six experts specializing in machine learning. These experts were identified through our professional network on LinkedIn. Given the vastness of the dataset, and in consideration of the potential for annotator fatigue, we opted to select a subset comprising 400 APIs from various libraries. This strategic selection was made to ensure a comprehensive coverage across different functionalities while at the same time optimizing the use of our resources.

The chosen APIs are representative of diverse use cases, ensuring a holistic understanding of the domain. For example, libraries such as *Matplotlib*, *Seaborn*, and *Plotly* provide an extensive array of APIs dedicated to plotting functionalities. On the other hand, *Pandas* and *NumPy* are predominantly known for their data manipulation capabilities. Similarly, the *Sklearn*, *Keras*, and *Torch* libraries are renowned for their APIs that facilitate model building in ML. This structured approach in API selection ensured that our annotations captured the broad spectrum of functionalities inherent in the ML domain.

**Table 3** Library API utilization overview

| Library | Total API calls | Unique API calls |
|---|---|---|
| Pandas | 61,949 | 393 |
| Sklearn | 27,314 | 571 |
| Matplotlib | 16,762 | 224 |
| Numpy | 12,355 | 273 |
| Seaborn | 6,531 | 64 |
| Keras | 6,136 | 239 |
| LightGBM | 3,100 | 34 |
| XGBoost | 2,321 | 37 |
| Tensorflow | 1,664 | 345 |
| Plotly | 1,482 | 98 |
| Torch | 1,449 | 188 |
| Statsmodels | 594 | 87 |
| **Total** | 141,657 | 2,553 |

We developed an annotation tool designed to display the API together with its associated doc string. This feature was implemented to aid annotators in efficiently identifying and selecting the relevant machine learning operation, subsequently facilitating the exportation of results. It is important to note that every API was reviewed by multiple annotators to ensure accuracy. To assess the consistency of these annotations, we calculated the inter-annotator agreement and obtained a Cohen's kappa coefficient (Cohen, 1960) of 0.80. This score indicates a substantial level of agreement among the annotators.

### 3.3.2 Data preprocessing

Doc strings often contain special characters, additional reference texts, and example content that can make ML pattern recognition challenging. To tackle this, we use Natural Language Processing (NLP) methods to preprocess these strings. Note that, during the annotation process, the unmodified doc strings were presented to the annotators.

The steps involved are outlined as follows:

– **Data Cleaning:** removal of LaTeX and markdown formatting strings. Further, we elim-inate version numbers, statements indicating deprecation, URLs, punctuation, and other special characters to ensure a cleaner dataset.
– **Stop Word Removal:** commonly used words such as "the", "and", and "is", are removed.
– **Lemmatization:** applied to reduce words to their base form.

### 3.3.3 Model training

We used the finalized dataset with a train-test split of 80%-20% to train and test a series of multi-label classification techniques. The nature of our classification problem is multi-label because a single API can correspond to multiple ML Operations. For instance, the API `numpy.ndarray.reshape` can be simultaneously categorized under both "Data Preparation and Exploration" and "Feature Engineering".

As ML Models operate on numerical data, we first convert doc strings into a numerical format, in a process termed as text vectorization. In this research, we explored three prominent text vectorization methodologies: Word2Vec, TF-IDF, and CountVectorizer. With data from these techniques, we subsequently trained a range of ML models listed as follows: Logistic Regression, Random Forest, Decision Tree, GaussianNB, Support Vector Machines (SVM), and Gradient Boosting. Note that, driven by the constraints posed by our small dataset size, we abstained from incorporating neural networks in our experimentation.

**Model selection**  The performance of various classifier models is shown in Table 4. In com-parison with other classifiers and across vectorization methods, SVM stands out particularly with TF-IDF, showcasing a balance of accuracy, precision, and recall. Therefore, within HEADERGEN we integrated the SVM classifier with TF-IDF vectorization for classifying API calls.

### 3.4 Jupyter notebook annotation

The goal of HEADERGEN is to aid ML practitioners 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.

**Table 4** Performance of classifiers for different text vectorization techniques

A: Accuracy, P: Precision, R: Recall

| Classifier | TF-IDF | | | Word2Vec | | | CountVec | | |
|---|---|---|---|---|---|---|---|---|---|
| | A. | P. | R. | A. | P. | R. | A. | P. | R. |
| LogisticRegression | 0.86 | 1.0 | 0.86 | 0.60 | 0.95 | 0.59 | 0.89 | 0.98 | 0.89 |
| RandomForest | 0.84 | 0.99 | 0.86 | 0.80 | 0.98 | 0.80 | 0.88 | 0.99 | 0.88 |
| DecisionTree | 0.76 | 0.89 | 0.82 | 0.57 | 0.74 | 0.68 | 0.69 | 0.81 | 0.88 |
| GaussianNB | 0.83 | 0.92 | 0.86 | 0.72 | 0.81 | 0.83 | 0.75 | 0.84 | 0.88 |
| **SVM** | **0.94** | **1.0** | **0.94** | 0.83 | 0.89 | 0.91 | 0.77 | 0.93 | 0.80 |
| GradientBoosting | 0.81 | 0.96 | 0.82 | 0.79 | 0.92 | 0.81 | 0.85 | 0.95 | 0.89 |

### 3.4.1 Pattern matching

Notebooks can contain code cells that perform ML operations without explicit function calls, but rather, use other Python constructs that alter objects. In absence of a function call, HEADERGEN resorts to AST based pattern matching to identify ML operations.

The process is as follows: Initially, HEADERGEN identifies code cells devoid of any function calls by analyzing the flow-sensitive CG constructed in the preceding phase of the analysis. Subsequently, it traverses the AST of these code cells to detect the presence of specific patterns outlined in Table 5. However, given that ASTs lack type information, HEADERGEN consults the EAG using line number and node identifier from the AST node, `Name`, to retrieve type information for AST elements identified within the code cell. Finally, if a pattern aligns with one of the supported patterns by HEADERGEN, the corresponding code cell is annotated accordingly.

For instance, consider the first pattern in Table 5 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 this specific case, HEADERGEN checks if both sides of the `BinOp` AST node for the binary operator '`*`' are indeed Dataframe accesses. That is, if `df.x` and `df.y`, are Dataframe accesses. Just by looking at the AST, HEADERGEN cannot ascertain the type of the variable `df`, consequently, it retrieves the type information from the EAG to check if `df` is a Dataframe. Then, From this, HEADERGEN

**Table 5** Mapping of Dataframe usage patterns to ML operations

| ID | Pattern | ML Operation |
|---|---|---|
| 1 | `df['xy'] = df.x * df.y` | Feature Engineering |
| 2 | `df.x = 1` | Feature TransformationData Preparation |
| 3 | `df.x[df.x == 1] = 1` | Feature TransformationData Preparation |
| 4 | `x = df.x[['f1', 'f2']]` | Feature Selection |
| 5 | `print(df[0:20])` | Exploratory Data Analysis |
| 6 | `for i in df:` | Data Preparation |
| 7 | `len(df)` | Exploratory Data Analysis |
| 8 | `df.shape` | Exploratory Data Analysis |

concludes that this statement is a *Feature Engineering* operation. Table 5 lists the Dataframe access patterns that HEADERGEN currently supports.

### 3.4.2 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 Fig. 1. Figure 6a shows the index of ML operations generated for our motivating example. The index



(a) 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.



(b) Annotated version of cell # 3 in our motivational example.

**Fig. 6** Snapshots of the output notebook generated by HEADERGEN for our motivational example

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. Figure 6b shows the annotated version of code cell 3 from our motivating example (Fig. 2). 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.

# 4 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: To what extent does type information contribute to the accuracy of HeaderGen?*
*RQ5: How does HeaderGen compare to other tools?*

We first describe the benchmarks we developed for evaluating HEADERGEN in Section 5, and then examine the research questions.

# 5 Benchmarks

We evaluate HEADERGEN by building four benchmarks: (1) a micro-benchmark containing 121 notebooks, (2) a real-world benchmark containing 15 notebooks from Kaggle, (3) an extended real-world benchmark containing 15 notebooks from the wild, and (4) TYPEEVALPY, a framework for benchmarking type inference in Python. We discuss these benchmarks in the subsequent sections.

## 5.1 Jupyter notebook micro-benchmark

We evaluate HEADERGEN by adopting the benchmark created by Salis et al. (2021) 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

Table 6  Details of notebooks included in the real-world benchmark evaluation

| ID | Name | Votes | Views |
|---|---|---|---|
| 1 | bulentsiyah/keras-deep-learning-to-solve-titanic | 69 | 2,244 |
| 2 | hongdnghuy/relu-sigmoid | 13 | 743 |
| 3 | vaidicjain/titanic-easy-deeplearning-acc-78 | 9 | 489 |
| 4 | tanvikurade/complete-analysis-of-titanic | 19 | 326 |
| 5 | alexanderbader/mytitanic | 10 | 113 |
| 6 | econdata/predicting-future-sales-with-lstm | 7 | 3,363 |
| 7 | lhavanya/predict-future-sales | 3 | 500 |
| 8 | elvinagammed/stacked-lstm-top-5-4-mae | 9 | 523 |
| 9 | ashishkapasiya/prediction-future-sales-with-keras | 4 | 525 |
| 10 | the0electronic0guy/keras-begineer-friendly | 12 | 290 |
| 11 | higepon/starter-keras-simple-nn-kfold-cv | 20 | 4,387 |
| 12 | vishesh17/keras-nn-with-scaling-and-regularization | 32 | 3,205 |
| 13 | christofhenkel/nn-with-magic-augmentation | 21 | 1,573 |
| 14 | naivelamb/multibranch-nn-baseline-magic | 10 | 652 |
| 15 | miklgr500/nn-embedding | 10 | 585 |
| | **Total** | **248** | **19,518** |

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.

## 5.2 Real-world benchmark

To assess HEADERGEN in real-world scenarios, we tested for precision and recall on 15 notebooks from Kaggle, a community where ML 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 by many (cf. Table 6).

We selected notebooks from three different and most popular competitions on Kaggle based on the number of 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[5] 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

---

[5] https://keras.io/

in our benchmark are listed in Table 9. 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. (2019). Note that these undocumented notebooks still have 240 upvotes and 17,687 views as of Octorber 2023.

The callsites ground truth was 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.

Further, the ground truth for headers were created using experts. 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 (Cohen 1960), 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.

### 5.3 Extended real-world benchmark

Building upon our initial publication (Venkatesh et al. 2023b), we have extended our benchmark suite by incorporating an additional 15 Jupyter notebooks. This extension aims to further assess the utility of HEADERGEN, across a broader spectrum of ML notebooks found in real-world contexts. The process of annotating notebooks with metadata pertaining to callsites and headers presents significant resource demands. Fortunately, Ramasamy et al. (2022) recently introduced the DASWOW dataset. This dataset comprises 470 notebooks annotated by domain experts, albeit using a different classification system that does not directly correlate with the taxonomy used by HEADERGEN.

Moreover, to evaluate HEADERGEN's accuracy, we also require ground truth of the callsites within these notebooks. Given the resource requirements of generating ground truth for the entirety of the notebooks, particularly regarding callsites, we narrowed our focus to a set of 15 notebooks. To this end, we first selected notebooks devoid of markdown cells from the DASWOW dataset, resulting in 124 notebooks. Subsequently, we prioritized notebooks with more number of code cells, selecting the top 15 notebooks. This collection contains a median of 27 code cells.

The creation of ground truth for headers involved aligning the DASWOW taxonomy with our own, leveraging the partial mappings provided by Ramasamy et al. (2022) in their work. This was followed by a manual verification of header annotations by the first author, to ensure accuracy and to add any missing labels. Furthermore, the callsites for notebooks were determined through a manual review process, adhering to the methodology described in the preceding Section 5.2.

### 5.4 TypeEvalPy: framework for type inference benchmarking

In recent years, the development and publication of type inference tools for Python have experienced a surge in both academic research and the open-source community. However, to date, no attempt has been made to construct a comprehensive benchmark dedicated to evaluating type inference tools for Python. The majority of existing research evaluates the accuracy of type inference tools using real-world benchmark datasets, for instance *Type4Py* (Mir et al.

2022) and *HiTyper* (Peng et al. 2022). On the other hand, open-source tools are evaluated using specifically-designed test cases.

Such evaluation methods, however, come with their set of challenges:

– Different studies may use different datasets, making it difficult to directly compare and understand the strengths and weaknesses of each tool.
– Real-world datasets occasionally contain inaccurate type annotations.
– Many evaluations give a general score, missing detailed insights into specific challenges, such as handling different language features.

To address this gap, we propose TYPEEVALPY, a comprehensive micro-benchmark suite aimed at evaluating type inference in Python, encompassing the diverse and complex features of Python. Our micro-benchmark comprises 154 code snippets, each focusing on distinct features of the Python language, such as, generators and decorators. The first two authors conducted exhaustive ground-truth type annotations for each snippet, subjecting each to multiple reviews during the developmental stages to ensure accuracy. A foundational objective of TYPEEVALPY is to ensure reproducibility and extensibility to any type inference tool. To realize this, we incorporated Docker and Python boilerplate code, with uniformly implemented functions enabling easy integration and execution of any type inference tool on the micro-benchmark. To ensure reproducibility of results, TYPEEVALPY is highly automated, allowing the evaluation of the benchmark across all tools with the execution of a single command. Additionally, using the provided boilerplate, we have implemented containers compatible with TYPEEVALPY for six renowned Python type inference tools: HEADERGEN, *Jedi*, *Pyright*, *HiTyper*, *Scalpel*, and *Type4Py*.

The workflow of TYPEEVALPY is outlined in Fig. 7. The `runner` performs tests on each type inference tool in TYPEEVALPY using the micro-benchmark. After completion, the `translator` converts results into a standard format defined in the TYPEEVALPY framework. Note that each tool implements its own translation logic and the `translator` orchestrates the conversion. These files are then analyzed to produce detailed evaluation metrics for type inference by the `result analyzer`. Further details about these components are discussed in the following sections.

### 5.4.1 Micro-benchmark

The TYPEEVALPY micro-benchmark encompasses 154 code snippets, systematically organized into 18 categories, reflective of various Python features. These categories are designed, taking inspiration from the PyCG call-graph benchmark and contains the following categories: *args (8)*, *assignments (8)*, *builtins (7)*, *classes (26)*, *decorators (8)*, *dicts (15)*,
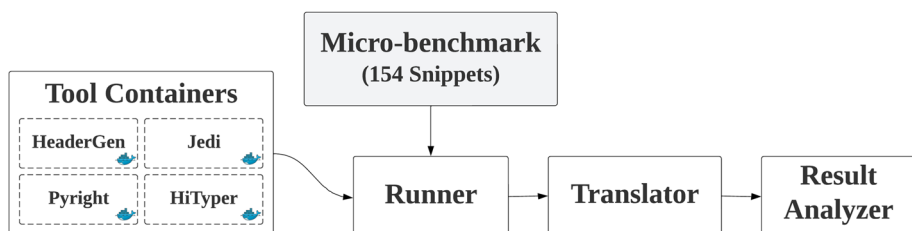


**Fig. 7** Overview of benchmarking workflow in TYPEEVALPY

*direct_calls (6)*, *dynamic (3)*, *exceptions (2)*, *external (7)*, *functions (9)*, *generators (6)*, *imports (14)*, *kwargs (4)*, *lambdas (6)*, *lists (10)*, *mro (7)*, *returns (8)*.

The corresponding ground truth contains a total of 845 type annotations from 154 code snippets, each of which contains the name of the entity, line number, and column offset, supplemented by the category of the type. Type annotations are cataloged into three distinct categories: 1) Function return (FR) type, 2) Function parameter (FP) type, and 3) Local variable (LV) type. The creation of ground truth involved the validation of runtime types through the execution of the program and the use of a debugger.

Several considerations were taken during the design phase of the benchmark:

1. Annotations are flow-sensitive, this means they are connected to the location or line number of the entity.
2. The type of function return, function parameters, and local variables can be different based on the calling context within a function definition. The type annotations are context-*in*sensitive, though, thus enumerate the union of all types in all concrete usage contexts of the application at hand.
3. Annotations use generic types. For example, a list of integers is noted as just a `List`, not `List[int]`. However, individual elements of lists and dicts receive their own annotations.
4. Entities are not labeled with the "Any" type. When a function has the potential to return multiple types, efforts are made to include all possible return types within the calling contexts.
5. If a function does not have a return statement, we specifically annotate it as `Nonetype`.

### 5.4.2 TypeEvalPy runner

In the TYPEEVALPY framework, the role of the `runner` is pivotal. It initiates the container-ized type inference tools and methodically executes type inference on every snippet included in the micro-benchmark. The `runner` component also orchestrates the translation of the results into the TYPEEVALPY format by using the `translator` component. The `runner` gathers all the derived results and consolidates them. After the translation, the `runner` subjects the results to the `result analyzer` to extract meaningful insights.

The `translator` and `result analyzer` components are discussed in the following sections.

### 5.4.3 TypeEvalPy translator

Various type inference tools present their analysis outcomes in diverse formats. Therefore, the TYPEEVALPY framework establishes a common schema to describe types, based on the Scalpel framework (Li et al. 2022). Each tool should implement a `translator` plugin according to the specifications of the TYPEEVALPY framework.

Listing 2 shows all kinds of type annotations within the TYPEEVALPY schema corresponding to the code snippet shown in Listing 1. Below are the categories of type annotations along with their corresponding line numbers as depicted in Listing 2:

– **Function return:** `id_func`: line 2-5
– **Function parameter:** `arg`: line 8-12
– **Local variables:** `x`: line 15-19, `result`: line 22-25 and 28-31

Note that the annotation for the local variable x, at lines 15-19, incorporates a "function" key, serving to signify its definition locality. Conversely, the annotation for the local variable `result`, at lines 22-25 and 28-31, omits this key, attributed to its definition at the module level. Note that the `result` variable is annotated with flow-sensitivity at two program points.

```python
1 def id_func(arg):
2     x = arg
3     return x
4
5 result = id_func("String
    ")
6 result = id_func(1)
```
**Listing 1** simple_code.py

```json
1 [{
2     "file": "simple_code.
      py",
3     "function": "id_func
      ",
4     "line_number": 1,
5     "type": ["int", "str
      "]
6 },
7 {
8     "file": "simple_code.
      py",
9     "function": "id_func
      ",
10    "line_number": 1,
11    "parameter": "arg",
12    "type": ["int", "str
      "]
13 },
14 {
15    "file": "simple_code.
      py",
16    "function": "id_func
      ",
17    "line_number": 2,
18    "type": ["int", "str
      "],
19    "variable": "x"
20 },
21 {
22    "file": "simple_code.
      py",
23    "line_number": 5,
24    "type": ["str"],
25    "variable": "result"
26 },
27 {
28    "file": "simple_code.
      py",
29    "line_number": 6,
30    "type": ["int"],
31    "variable": "result"
32 }]
```
**Listing 2** Types for simple_code.py

### 5.4.4 TypeEvalPy result analyzer

The `result analyzer` component within the TYPEEVALPY framework generates a comprehensive set of comparative statistics to evaluate the performance of various tools. Specifically, it computes the following metrics:

- **Exact matches:** *How many of the inferred types exactly match the ground truth?*
- **Precision:** *Of the types reported, how many are exactly inferred according to the ground truth?*
- **Recall:** *How many of the actual types are exactly reported by the type inference tool?*

- **Soundness:** *Does the type inference tool exactly identify all possible types specified in the Python code to ensure none are omitted?*
- **Completeness:** *Does the tool accurately report only the types that are present, avoiding any incorrect or extraneous types?*
- **Top-n prediction comparison:** *How do the ML-based tools compare in terms of accuracy when considering their top-n inferred types?*
- **Report of missing types:** *Which types, present in the ground truth, are missed by the tools?*
- **Report of mismatched types:** *Which types reported by the tools do not align with the types present in the ground truth?*

# 6 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.

## 6.1 Study design

The study is aimed at recreating the exploration of notebooks that ML practitioners 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,

**Table 7** 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 8** Statements concerning the 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 are rather hindering the understanding of the notebook. |
| S4 | I would install HEADERGEN if it is made available as a plugin. |

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 7. 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 8.

## 6.2 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[6] 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.

## 6.3 Metrics

(1) Time: Time taken to complete all five tasks per notebook.
(2) Accuracy: Inspired by a similar comprehension study by Adeli et al. (2020), 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.
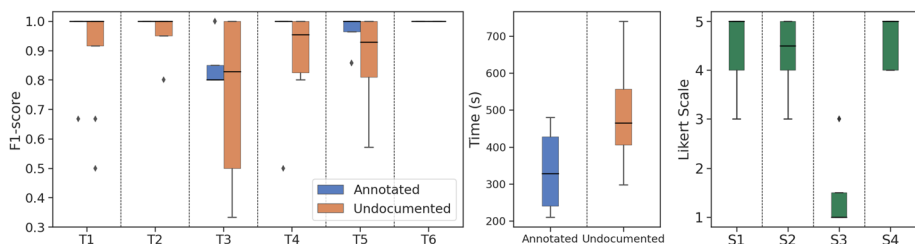
---

[6] https://www.linkedin.com/

## 6.4 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 8 shows the box-plot 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 8 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



**Fig. 8** **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

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.

# 7 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. Furthermore, note that PyCG does not output line numbers in its analysis and therefore a direct automated comparison is not possible. When manually compared to PyCG for the *flow_sensitive* category, as expected, PyCG is incomplete for all eight test cases.

**Real-world benchmark results** Table 9 lists the precision and recall values of HEADERGEN for real-world notebooks.

HEADERGEN achieves an average of 95.6% precision and 95.3% 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 straightforward to infer and needs advanced dataflow analysis.

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

**Table 9** Real-world benchmark evaluation of HEADERGEN for callsite recognition and header annotation

| Id | Callsite Recognition | | Header Annotation | |
|---|---|---|---|---|
| | Precision | Recall | Precision | Recall |
| 1 | 100 | 90.0 | 71.4 | 100 |
| 2 | 100 | 100 | 84.2 | 100 |
| 3 | 95.8 | 95.8 | 100 | 87.5 |
| 4 | 100 | 100 | 75.8 | 95.9 |
| 5 | 93.8 | 97.4 | 83.3 | 90.9 |
| 6 | 86.0 | 97.4 | 100 | 100 |
| 7 | 94.4 | 91.9 | 83.3 | 88.2 |
| 8 | 100 | 100 | 94.1 | 76.2 |
| 9 | 90.9 | 97.2 | 83.6 | 92.7 |
| 10 | 100 | 100 | 85.0 | 90.7 |
| 11 | 97.1 | 85.0 | 68.8 | 100 |
| 12 | 96.0 | 96.0 | 94.7 | 94.7 |
| 13 | 94.3 | 94.4 | 100 | 100 |
| 14 | 94.4 | 93.5 | 74.1 | 87.0 |
| 15 | 91.3 | 91.3 | 87.5 | 87.5 |
| **Average** | **95.6** | **95.3** | **85.7** | **92.8** |

**Extended real-world benchmark results** Table 10 lists the precision and recall values of HEADERGEN for real-world notebooks. HEADERGEN achieves an average of 94.4% precision and 91.6% recall. In two instances, the analysis achieves 100% precision and recall. We observe similar results compared to our evaluation with the real-world benchmark.

However, in the instance of "nb_111547", a notably low recall rate of 66.7% is observed. This issue arises from the authors' frequent use of the `DataFrame.apply` and `Series.apply` methods within the notebook. These methods accept a function reference as an argument and apply it across the DataFrame's contents. While HEADERGEN can partially analyze this functionality, the behavior of these functions is dependent on the data type within the DataFrame, information that is not statically available. This highlights a limitation of our approach.

## 8 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.

**Results** The resulting precision and recall are listed on the right side of Tables 9 and 10. The headers that are generated by HEADERGEN are matched on the high-level categories of the taxonomy listed in Fig. 1. HEADERGEN achieves a precision of 85.7% and recall of 92.8% on the real-world benchmark and 84.8% precision and 91.2% recall on the extended real-world benchmark. HEADERGEN achieves similar results with the extended real-world benchmark. Precision is lost because some functions can be mapped to more than one ML operation.

**Table 10** Extended benchmark evaluation of HEADERGEN for callsite recognition and header annotation

| Id | Callsite Recognition | | Header Annotation | |
| | Precision | Recall | Precision | Recall |
| --- | --- | --- | --- | --- |
| **nb_111547** | 95.4 | 66.7 | 82.5 | 91.7 |
| nb_12952 | 98.1 | 98.7 | 73.4 | 100 |
| nb_13973 | 100 | 100 | 92.0 | 76.7 |
| nb_2004 | 88.9 | 94.1 | 95.8 | 74.2 |
| nb_24325 | 97.2 | 100 | 95.5 | 91.3 |
| nb_27915 | 92.0 | 95.8 | 74.5 | 100 |
| nb_44070 | 85.7 | 88.9 | 81.8 | 93.1 |
| nb_50450 | 89.2 | 87.9 | 79.0 | 89.1 |
| nb_5433 | 97.7 | 80.8 | 92.5 | 100 |
| nb_6005 | 84.4 | 85.7 | 56.4 | 91.7 |
| nb_62235 | 95.8 | 90.7 | 93.9 | 91.2 |
| nb_62337 | 100 | 100 | 100 | 97.5 |
| nb_79900 | 97.2 | 89.7 | 92.9 | 96.3 |
| nb_87231 | 99.1 | 99.1 | 85.7 | 81.1 |
| nb_96079 | 95.6 | 95.6 | 76.3 | 93.5 |
| **Average** | **94.4** | **91.6** | **84.8** | **91.2** |

**Impact of pattern matching** To study the impact of pattern matching on the accuracy of header annotations, we employed HEADERGEN with pattern matching deactivated and compiled the resultant data. Observations indicate that, within the context of a real-world benchmark, activating pattern matching enhances the recall of header annotation by 10.6%, and for the extended real-world benchmark, recall improves by 14.8%. Conversely, the precision of header annotation diminishes across both benchmarks due to the over-approximation of patterns that are mapped to multiple ML operations. Specifically, for the real-world benchmark, precision decreased by 2.3%, and for the extended real-world benchmark, it decreased by 4.6%.

# 9 RQ4: dependency of headergen on type information

HeaderGen relies on its analyzer's ability to infer variable types to accurately resolve callsites. To systematically measure the influence of type inference on the accuracy of HeaderGen, we conduct a comprehensive comparison of type inference tools within the TYPEEVALPY framework. We compare HEADERGEN against *PyCG*, *pyright*, *Jedi*, and *HiTyper* using our real-world benchmark. Note that our type-stub database of ML libraries was supplied to *pyright* and *Jedi* to facilitate their analysis.

We employ TYPEEVALPY to generate statistics related to exact type matches using its micro-benchmark. Table 11 displays the exact matches for all the tools being compared. Additionally, we evaluated *HiTyper*, which is a hybrid analysis technique using deep learning for type inference. This method leverages *PyCG* as a static analysis tool within its process.

**Results** HEADERGEN exhibits dominant performance across various categories. Specifically, it showcases superior type inference for function returns in categories such as *args*, *classes*, *decorators*, *dicts*, *direct_calls*, *imports*, *kwargs*, *lists*, and *mro*. Notably, in the *classes* cat-

egory, HEADERGEN outperformed other tools in function returns. HEADERGEN also shows robust performance in inferring local variables and function parameters across multiple categories, highlighting its comprehensiveness.

Other tools have promising results in certain categories: *Jedi* has its strengths in inferring function returns for *decorators* and excels in *lambdas* with better inference of function returns and local variables. *pyright* shows prowess in the builtins category for local variables. *HiTyper* exhibits strength in categories like *assignments* for function returns and *generators* where it excels in recognizing function returns and local variables.

While HEADERGEN performed robustly in most categories, some outliers exist. In the *builtins* category, HEADERGEN failed to infer function returns, making it a notable exception. Similarly, in *dynamic* and *external* categories, its performance wasn't comprehensive.

In addition, it is notable that both *Jedi* and *pyright* consistently fail to infer function parameter types across all categories. This design choice stems from their tendency to label the type annotation of function parameters as "`Any`", reflecting Python's duck typing philosophy. Essentially, this means a non-type annotated function can accept a variable of any type. While this might align with Python's flexibility, it is not beneficial for static analysis of Python code.

**Discussion** The standout performance of *Jedi* and *pyright* in the *builtins* and *external* categories provide insights into their underlying design. Academic tools, such as *PyCG*, overlook the use of user provided type hints, called typestubs, when analyzing the types of Python

**Table 11** Exact matches of type inference tools for micro-benchmark categories

| Category | Total | | | HeaderGen | | | Jedi | | | Pyright | | | HiTyper | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FR | FP | LV | FR | FP | LV | FR | FP | LV | FR | FP | LV | FR | FP | LV |
| args | 17 | 10 | 16 | 17 | 9 | 12 | 12 | 0 | 9 | 8 | 1 | 8 | 12 | 0 | 6 |
| assignments | 22 | 4 | 56 | 15 | 1 | 33 | 20 | 0 | 21 | 20 | 0 | 25 | 21 | 4 | 9 |
| **builtins** | 3 | 4 | 61 | 0 | 0 | 26 | 0 | 0 | 21 | 1 | 0 | 45 | 1 | 2 | 18 |
| classes | 41 | 8 | 76 | 39 | 7 | 67 | 0 | 0 | 57 | 1 | 0 | 46 | 27 | 2 | 41 |
| decorators | 29 | 17 | 12 | 11 | 6 | 2 | 10 | 0 | 8 | 7 | 0 | 3 | 8 | 0 | 3 |
| dicts | 23 | 3 | 82 | 23 | 3 | 60 | 21 | 0 | 34 | 19 | 2 | 50 | 20 | 3 | 22 |
| direct_calls | 10 | 3 | 11 | 10 | 3 | 8 | 6 | 0 | 7 | 3 | 0 | 6 | 3 | 2 | 4 |
| **dynamic** | 1 | 0 | 8 | 1 | 0 | 2 | 1 | 0 | 2 | 1 | 0 | 2 | 1 | 0 | 5 |
| exceptions | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| external | 3 | 1 | 12 | 0 | 0 | 3 | 0 | 0 | 8 | 0 | 0 | 2 | 1 | 0 | 4 |
| functions | 11 | 9 | 17 | 8 | 9 | 12 | 5 | 0 | 14 | 5 | 2 | 13 | 5 | 5 | 5 |
| generators | 13 | 8 | 35 | 9 | 4 | 17 | 5 | 0 | 23 | 4 | 3 | 18 | 10 | 5 | 11 |
| imports | 3 | 0 | 22 | 3 | 0 | 11 | 1 | 0 | 16 | 3 | 0 | 20 | 3 | 0 | 11 |
| kwargs | 8 | 6 | 8 | 8 | 5 | 5 | 7 | 0 | 4 | 4 | 0 | 5 | 7 | 0 | 0 |
| lambdas | 7 | 7 | 20 | 3 | 7 | 4 | 6 | 0 | 11 | 2 | 0 | 1 | 3 | 0 | 7 |
| lists | 17 | 4 | 38 | 14 | 1 | 26 | 17 | 0 | 27 | 13 | 0 | 25 | 16 | 3 | 16 |
| mro | 16 | 0 | 18 | 14 | 0 | 16 | 0 | 0 | 13 | 0 | 0 | 14 | 15 | 0 | 11 |
| returns | 15 | 4 | 24 | 11 | 1 | 16 | 11 | 0 | 17 | 9 | 0 | 13 | 10 | 1 | 5 |
| **Totals** | **239** | **88** | **518** | **186** | **56** | **322** | **122** | **0** | **293** | **100** | **8** | **297** | **163** | **27** | **179** |
| | | 845 | | | 564 | | | 415 | | | 405 | | | 369 | |

*FR:* Function return types, *FP:* Function parameter types, and *LV:* Local variable types, *845:* Total type annotations, *154:* Total test cases

elements. Conversely, both *Jedi* and *pyright* have fine-tuned their processes to incorporate these typestubs effectively. Nevertheless, the design paradigms of both *Jedi* and *pyright* are oriented towards specific use cases, such as auto-completion and Integrated Development Environments (IDEs) integration, influencing the nature of their outputs. Accessing their internal static analysis data, like points-to and type information, is not straightforward for a more generalized source code analysis, as needed for HEADERGEN.

To achieve better performance, HEADERGEN combines *PyCG's* algorithm, EAG, and leverages typestubs, where required, to increase accuracy. However, the support for handling typestubs are more mature in *Jedi* and *pyright* and this is evident by their performance in the highlighted categories where typestubs are required the most.

Another observation is the performance of *HiTyper*. Despite its foundation on *PyCG* and its hybrid analysis approach, its varied performance on different categories are difficult to explain considering its reliance on ML techniques. We posit that incorporating HEADERGEN's analysis might enhance the accuracy of *HiTyper*. We aim to explore this in future research.

**Modeling of pandas behavior** Listing 3 shows simplified data manipulation methods of the Pandas library based on our real-world benchmark. Furthermore, Table 12 lists the type of each variable used in Listing 3 as inferred by the tools being compared. It can be seen that *pyright*, *Jedi*, and *HiTyper* fail to infer return-types of variables x1 through x6. This is because HEADERGEN can model complex pandas accesses while the other 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 3** Common uses of Pandas DataFrame that existing tools fail to infer.

**Table 12** Comparison of type inference by existing tools for Listing 3

| Variable | Actual | HeaderGen | Pyright | Jedi | HiTyper |
|---|---|---|---|---|---|
| df | DataFrame | DataFrame | DataFrame | DataFrame | List |
| x1 | Series | Series | Any | Any | Int |
| x2 | DataFrame | DataFrame | Any | Any | Int |
| x3 | Series | Series | Any | Any | Int |
| x4 | DataFrame | DataFrame | Any | Any | Int |
| x5 | DataFrame | DataFrame | Any | Any | Int |
| x6 | Ndarray | Ndarray | Any | Any | Int |

## 10 RQ5: comparison with existing tools

We conduct a comparison of HEADERGEN against *PyCG*, *pyright*, *Jedi*, and *HiTyper* focusing on callsite recognition and header annotation, using our real-world benchmark. Since *pyright* and *Jedi* are primarily configured for type checking and auto-completion, we incorporated auxiliary functions to gather type information and callsite data for comparison.

**Results.** The precision and recall values for both the real-world benchmarks and extended real-world benchmarks are listed in Table 13. 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 callsite recall of 95.3% which leads to a 92.8% 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. We observe a similar trend on the extended real-world benchmark results with HEADERGEN leading with a callsite recall of 91.6% and 91.2% recall in header annotation of code cells. The loss of precision is attributed to the over-approximation of return-types in our type-stub database as discussed earlier.

Note that incorporating HiTyper into HEADERGEN for an automated evaluation of callsite recognition, based on HiTyper's inferred types, proved challenging. The difficulty originated from the process of translating HiTyper's output back to the original source code and further integrating it with HEADERGEN's expected format. Despite significant effort, we opted to discontinue this integration process. Nonetheless, insights from TYPEEVALPY, as outlined in Section 9, suggests that HiTyper's performance may not surpass that of HEADERGEN. This perspective was further reinforced by our manual review of HiTyper's results when applied to the real-world benchmark notebooks.

## 11 Related work

**Tool-support for jupyter notebooks** Over the past few years, numerous studies (Kery et al. 2018; Rule et al. 2018; Pimentel et al. 2019; Koenzen et al. 2020; Wang et al. 2020b; Epperson et al. 2022; Quaranta et al. 2022; Grotov et al. 2022) have examined coding patterns

**Table 13** Comparative evaluation of existing tools on real-world benchmark for callsite recognition and header annotation

| Tool | Callsite Recognition | | Header Annotation | |
|---|---|---|---|---|
| | Precision | Recall | Precision | Recall |
| **Real-world Benchmark** | | | | |
| HeaderGen | 95.6 | 95.3 | 85.7 | 92.8 |
| 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 |
| **Extended Real-world Benchmark** | | | | |
| HeaderGen | 94.4 | 91.6 | 84.8 | 91.2 |
| Pyright | 95.7 | 82.4 | 87.4 | 73.1 |
| Jedi | 86.6 | 70.9 | 89.2 | 58.8 |
| PyCG | 62.0 | 43.4 | 80.3 | 7.6 |

in notebooks. These studies consistently indicate a deficiency in the quality of notebooks, signaling a need for greater attention from the software engineering community. Despite these findings, there is a noticeable gap in research efforts focusing on tools that can address the identified issues.

In a step towards addressing this gap, Wang et al. (2022) introduced *Themisto*. This tool prompts data scientists to document their code cells. It employs a deep learning method to auto-generate code documentation in natural language and then suggests to users whether to integrate or directly apply this documentation. Notably, *Themisto* relies on the Abstract Syntax Tree (AST) of Python code for model training, without incorporating SA methods to extract more contextual information from the code. We posit that the analytical capabilities of HEADERGEN might enhance deep learning approaches, potentially yielding better outcomes.

In another study, Pimentel et al. (2019) studied 1.4 million notebooks for features that affect reproducibility and suggested a set of best practices. Following this, Wang et al. (2020a) 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. (2022a) design a SA approach to detect data leakage in notebooks. In contrast, our contribution to this domain focuses on the automatic annotation of code cells, offering tool-support for literal programming.

**Static analysis for python** Python has grown to be one of the leading programming languages, but the field still observes a significant lack in SA tools, as highlighted by a study on Python's features by Yang et al. (2022b). Yang et al. emphasize that Python's distinct characteristics make it difficult to use traditional analysis methods developed from existing scientific research. One reason for this is Python's dynamic features, like duck typing, which while beneficial for rapid prototyping, complicate its analysis.

Only as recently as 2021 was a notable breakthrough achieved in the domain of SA for Python: the introduction of *PyCG*, a method for constructing call graphs (Salis et al. 2021). Yet, this method does not account for flow of values and is not tailored for Jupyter notebooks. Furthermore, Python still lacks a comprehensive SA framework to produce data flow intermediate representations. The most related work in this area is the Scalpel project (Li et al. 2022), but it too falls short, particularly in inferring return types for external functions and considering notebook cells.

In addition, Monat et al. (2020a) delve into using static abstract interpretation to aid Python type analysis. This approach covers a broad range of constructs and precisely combines domains, allowing sound knowledge of nominal and structural types and exceptions raised in the program. Building on this foundation, Monat et al. (2020b) developed *MOPSA*, a prototype tool that integrates value analysis. However, MOPSA focuses specifically on analyzing Python code that are combined with C code. In our study, we aim to enhance the current landscape of SA for Python in practical scenarios. We present methods for resolving return types of external library APIs and extracting flow-sensitive function call sites, leveraging def-use relations. However, MOPSA has the potential to provide more reliable type stubs from C modules, which can benefit our work and other analyses. We suggest future research to study this more closely.

**Code classification** Code classification is fundamental for various tasks, such as determining code authorship, identifying the programming language, and understanding source file content. While significant work exists in this domain, studies that specifically focus on Jupyter notebooks are limited. Several machine learning techniques have been instrumental in the progress of this field. Methods like Max-entropy (Zevin and Holzem 2017), decision

tree (Ugurel et al. 2002), K-nearest neighbor (KNN), and Naive Bayes (Barstad et al. 2014) are noteworthy. They are proficient in tasks like predicting the programming language of source code and identifying the topics within a document. Additionally, recent advancements in deep learning have introduced weakly supervised transformer-based architectures for the classification and tagging of source code, as highlighted by Zhang et al. (2022).

In a recent contribution to this domain, Ramasamy et al. (2022) presented a supervised method specifically tailored for data science code classification. They approached the task as a topic classification challenge and explored both single-label (one label per cell) and multi-label classification to account for multiple data science operations within a single code cell. However, there are inherent limitations with their supervised classification approach. As noted by the authors, the classification of notebook cells "particularly those associated with evaluation, prediction, and visualization tasks" can achieve lower F1-scores due to the skewed distribution of their training dataset, where certain labels are underrepresented. In contrast, our method bypasses the need for an extensive pre-training process. By grounding our approach in API usage analysis, we aim to provide a solution that is more resilient and applicable in real-world contexts.

## 12 Limitations & future work

To enhance the accuracy of function name resolution, we leverage Python's reflection mechanism. This, however, may limit the coverage of APIs, contingent on the version of the library installed. To counteract this, we are considering the use of static API mapping techniques to address transitive imports in Python. Our current analysis predominantly focuses on machine learning applications. Nonetheless, the architecture of the framework we designed is not confined to this domain. HEADERGEN is equipped to annotate notebooks, accommodating domain-specific return-type stubs and library classifications. Lastly, the present version of HEADERGEN utilizes an ML taxonomy that comprises three main categories. Recent studies on notebooks have introduced a more detailed taxonomy, and future iterations of HEADERGEN will adopt this refined classification.

The taxonomy classification model has been developed using a small dataset consisting of 400 function calls, which represents a constraint of our methodology. The process of curating a dataset is costly and requires the involvement of several experts. In an attempt to enhance generalization, we are presently investigating methodologies for employing our dataset to fine-tune a large language model for this classification task.

Additionally, input from HEADERGEN can be used to automatically restructure code cells in notebooks for better readability. It can achieve this by reorganizing complex code cells, particularly those encompassing multiple ML operations, into a sequential arrangement. Furthermore, the efficient and accurate function call analysis provided by HEADERGEN paves the way for large-scale mining studies of Python code base.

## 13 Conclusion

In numerous instances, notebooks found in practical settings lack documentation, leading to challenges in understanding and navigating them. Addressing this, HEADERGEN employs accurate static analysis to automatically annotate notebooks with structural headers. These

headers are derived from a categorization of machine learning operations. Notably, HEAD-ERGEN displayed high precision and recall when tested against both micro and real-world benchmarks. Our evaluations against expert-curated ground truth further confirmed HEAD-ERGEN's capability to annotate headers with adequate precision and high recall. Furthermore, when assessing the type inference capabilities of various static analysis tools using the TYPE-EVALPY framework, HEADERGEN consistently surpassed its counterparts. To understand HEADERGEN's real-world impact, we conducted a user study. The results indicated that ML practitioners perceive HEADERGEN as a valuable tool for enhancing both program understanding and navigation.

**Data Availability** Data to reproduce experiments described in Sections 6, 7, 8,9, and 10 along with the source code for HEADERGEN is published on GitHub at: https://github.com/secure-software-engineering/HeaderGen.Data to reproduce Table 11 along with source code for TYPEEVALPY is published on GitHub at: https://github.com/secure-software-engineering/TypeEvalPy

## Declarations

**Conflicts of interest** The authors declare that they have no conflict of interest.

## References

Pyright (2022) static type checker for Python. https://github.com/microsoft/pyright

Pytype (2022) Google, https://github.com/google/pytype

MOPSA/MOPSA (2024) analyzer · GitLab.https://gitlab.com/mopsa/mopsa-analyzer

Adeli M, Nelson N, Chattopadhyay S, Coffey H, Henley A, Sarma A (2020) 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), pp 1–10. https://doi.org/10.1109/VL/HCC50065.2020.9127264

Allamanis M, Barr ET, Ducousso S, Gao Z (2020) Typilus: Neural type hints. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, ACM, London UK, pp 91–105. https://doi.org/10.1145/3385412.3385997

Barstad V, Goodwin M, Gjøsæter T (2014) Predicting source code quality with static analysis and machine learning. In: Norsk IKT-konferanse for forskning og utdanning

Cohen J (1960) A Coefficient of Agreement for Nominal Scales. Educ Psychol Measure 20(1):37–46. https://doi.org/10.1177/001316446002000104

Di Grazia L, Pradel M (2022a) The evolution of type annotations in python: An empirical study. In: Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering, ACM, Singapore Singapore, pp 209–220, https://doi.org/10.1145/3540250.3549114

Di Grazia L, Pradel M (2022b) 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

Epperson W, Wang A, DeLIne R, Drucker S (2022) Strategies for reuse and sharing among data scientists in software teams. In: ICSE 2022

Grotov K, Titov S, Sotnikov V, Golubev Y, Bryksin T (2022) A large-scale comparison of Python code in Jupyter notebooks and scripts. In: Proceedings of the 19th international conference on mining software repositories, association for computing machinery, New York, NY, USA, MSR '22, pp 353–36https://doi.org/10.1145/3524842.3528447

Guo Y, Chen Z, Chen L, Xu W, Li Y, Zhou Y, Xu B (2024) Generating Python Type Annotations from Type Inference: How Far Are We? ACM Trans Softw Eng Methodolo. https://doi.org/10.1145/3652153

Halter D (2022) Jedi - an awesome autocompletion, static analysis and refactoring library for Python

Kennedy K (1978) Use-definition chains with applications. Comput Lang 3(3):163–17. https://doi.org/10.1016/0096-0551(78)90009-7

Kery MB, Radensky M, Arya M, John BE, Myers BA (2018) The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In: Proceedings of the 2018 CHI conference on human factors in computing systems, association for computing machinery, New York, NY, USA, CHI '18, pp 1–1. https://doi.org/10.1145/3173574.3173748

Knuth DE (1984) Literate Programming. Comput J 27(2):97–111. https://doi.org/10.1093/comjnl/27.2.97

Koenzen AP, Ernst NA, Storey MAD (2020) Code Duplication and Reuse in Jupyter Notebooks. In: 2020 IEEE symposium on visual languages and human-centric computing (VL/HCC), pp 1–9. https://doi.org/10.1109/VL-HCC50065.2020.9127202

Kummita S, Piskachev G, Späth J, Bodden E (2021) Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In: 2021 International conference on code quality (ICCQ), pp 1–15. https://doi.org/10.1109/ICCQ51190.2021.9392986

Li L, Wang J, Quan H (2022) Scalpel: The python static analysis framework. https://doi.org/10.48550/ARXIV.2202.11840 arXiv:2202.11840

Mir AM, Latoškinas E, Proksch S, Gousios G (2022) Type4Py: Practical deep similarity learning-based type inference for python. In: Proceedings of the 44th international conference on software engineering, association for computing machinery, New York, NY, USA, ICSE '22, pp 2241–225. https://doi.org/10.1145/3510003.3510124

Monat R, Ouadjaout A, Miné A (2020a) Static type analysis by abstract interpretation of python programs. In: 34th European conference on object-oriented programming (ECOOP 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik

Monat R, Ouadjaout A, Miné A (2020b) Value and allocation sensitivity in static python analyses. In: Proceedings of the 9th ACM SIGPLAN international workshop on the state of the art in program analysis, pp 8–13

Peng Y, Gao C, Li Z, Gao B, Lo D, Zhang Q, Lyu M (2022) Static inference meets deep learning: A hybrid type inference approach for python. In: Proceedings of the 44th international conference on software engineering, association for computing machinery, New York, NY, USA, ICSE '22, pp 2019–203. https://doi.org/10.1145/3510003.3510038

Pimentel JF, Murta L, Braganholo V, Freire J (2019) A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), IEEE, Montreal, QC, Canada, pp 507–517. https://doi.org/10.1109/MSR.2019.00077

Quaranta L, Calefato F, Lanubile F (2022) Eliciting Best Practices for Collaboration with Computational Notebooks. Proc ACM Human-Comput Inter 6(CSCW1):87:1–87:4. https://doi.org/10.1145/3512934

Ramasamy D, Sarasua C, Bacchelli A, Bernstein A (2022) Workflow analysis of data science code in public GitHub repositories. Empir Softw Eng 28(1). https://doi.org/10.1007/s10664-022-10229-z

Rule A, Tabard A, Hollan JD (2018) Exploration and Explanation in Computational Notebooks. In: Proceedings of the 2018 CHI conference on Human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '18, pp 1–12. https://doi.org/10.1145/3173574.3173606

Salis V, Sotiropoulos T, Louridas P, Spinellis D, Mitropoulos D (2021) PyCG: Practical Call Graph Generation in Python. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), IEEE, Madrid, Spain, pp 1646–1657,https://doi.org/10.1109/ICSE43902.2021.00146

Samuel S, Mietchen D (2022) Computational reproducibility of Jupyter notebooks from biomedical publications. arXiv:2209.04308

serge-sans-paille (2022) Gast, Beniget!

Ugurel S, Krovetz R, Giles CL (2002) What's the code? automatic classification of source code archives. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp 632–638

Venkatesh APS, Wang J, Li L, Bodden E (2023a) Enhancing Comprehension and Navigation in Jupyter Note-books with Static Analysis. In: 2023 IEEE international conference on software analysis, evolution and reengineering (SANER), IEEE Computer Society, pp 391–40. https://doi.org/10.1109/SANER56733. 2023.00044

Venkatesh APS, Wang J, Li L, Bodden E (2023b) Enhancing Comprehension and Navigation in Jupyter Note-books with Static Analysis. In: 2023 IEEE international conference on software analysis, evolution and reengineering (SANER), IEEE Computer Society, pp 391–40. https://doi.org/10.1109/SANER56733. 2023.00044

Wagemann J, Fierli F, Mantovani S, Siemen S, Seeger B, Bendix J (2022) Five guiding principles to make jupyter notebooks fit for earth observation data education. Remote Sensing 14(14):3359

Wang AY, Wang D, Drozdal J, Muller M, Park S, Weisz JD, Liu X, Wu L, Dugan C (2022) Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. ACM Trans Comput Human Inter 29(2):17:1-17:33. https://doi.org/10.1145/3489465

Wang J, Kuo Ty, Li L, Zeller A (2020a) Assessing and restoring reproducibility of Jupyter notebooks. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, ACM, Virtual Event Australia, pp 138–149. https://doi.org/10.1145/3324884.3416585

Wang J, Li L, Zeller A (2020b) 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, Association for Computing Machinery, New York, NY, USA, ICSE-NIER '20, pp 53–56. https://doi.org/10.1145/3377816.3381724

Yang C, Brower-Sinning RA, Lewis GA, Kästner C (2022a) Data leakage in notebooks: Static detection and better processes. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering

Yang Y, Milanova A, Hirzel M (2022b) Complex Python Features in the Wild

Zevin S, Holzem C (2017) Machine learning based source code classification using syntax oriented features. arXiv:1703.07638

Zhang G, Merrill MA, Liu Y, Heer J, Althoff T (2022) Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis. EPJ Data Sci 11(1):14

## Authors and Affiliations

**Ashwin Prasad Shivarpatna Venkatesh[1]** ⬤ · **Samkutty Sabu[1]** · **Mouli Chekkapalli[1]** · **Jiawei Wang[2]** · **Li Li[3]** · **Eric Bodden[4]**

✉ Ashwin Prasad Shivarpatna Venkatesh
ashwin.prasad@upb.de

Samkutty Sabu
samkutty@mail.uni-paderborn.de

Mouli Chekkapalli
moulik@mail.uni-paderborn.de

Jiawei Wang
jiawei.wang1@monash.edu

Li Li
lilicoding@ieee.org

Eric Bodden
eric.bodden@upb.de

[1]    Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany

[2]    Faculty of Information Technology, Monash University, Melbourne, Australia

[3]    School of Software, Beihang University, Beijing, China

[4]    Heinz Nixdorf & Institut Fraunhofer IEM, Paderborn University, Paderborn, Germany