Towards a High Level Linter for Data Science

Greta Dolcetti
greta.dolcetti@unive.it
Ca' Foscari University of Venice
Venice, Italy

Agostino Cortesi
cortesi@unive.it
Ca' Foscari University of Venice
Venice, Italy

Caterina Urban
caterina.urban@inria.fr
Inria & ENS | PSL
Paris, France

Enea Zaffanella
enea.zaffanella@unipr.it
University of Parma
Parma, Italy

Abstract

Due to its interdisciplinary nature, the development of data science code is subject to a wide range of potential mistakes that can easily compromise the final results. Several tools have been proposed that can help the data scientist in identifying the most common, low level programming issues. We discuss the steps needed to implement a tool that is rather meant to focus on higher level errors that are specific of the data science pipeline. To this end, we propose a static analysis assigning *ad hoc* abstract datatypes to the program variables, which are then checked for consistency when calling functions defined in data science libraries. By adopting a descriptive (rather than prescriptive) abstract type system, we obtain a linter tool reporting data science related code smells. While being still work in progress, the current prototype is able to identify and report the code smells contained in several examples of questionable data science code.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → **Program analysis**; **Abstraction**.

Keywords: Static Analysis, Abstract Interpretation, Data Science.

ACM Reference Format:
Greta Dolcetti, Agostino Cortesi, Caterina Urban, and Enea Zaffanella. 2024. Towards a High Level Linter for Data Science. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3689609.3689996

NSAD '24, October 22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1217-3/24/10
https://doi.org/10.1145/3689609.3689996

1 Introduction

In recent years the amount of data available to organizations and to the general public has become massive: this fast growth has been identified with the term *data explosion* or *data deluge*. These beneficial circumstances have stimulated a corresponding explosive growth of the *data science* field, which can be defined informally as an interdisciplinary field that synthesizes and builds on statistics, informatics, computing, communication, management, and sociology to study data and its environments (including domains and other contextual aspects, such as organizational and social aspects); the end goal is to transform data into insights and decisions by following a data-to-knowledge-to-wisdom thinking and methodology [2]. Data science has been adopted and employed in a pervasive way in various applications in a wide range of fields such as healthcare, retail, manufacturing and finance. For this reason, data science tools and libraries, like `scikit-learn` [17], `seaborn` [26], and `Jupyter Notebooks` [10], are now remarkably popular.

Independently from their low level implementation details, these tools and libraries are often accessed by using programming languages such as R or Python: these are dynamically typed languages that, by their nature, perform their type correctness checks at runtime and do not offer native support for a more systematic, static control of the operations that are allowed on the values of variables. In contrast, statically typed languages perform most (sometimes all) of the type checks before running the program, so that they can eagerly spot the most common errors even before running a single dynamic test.

It is well known that the mere adoption of a statically typed language is no silver bullet: even after the programmer has corrected all of the typing errors spotted by the type checking tool (quite often, the compiler for the language), the program may still contain logic errors, so that a user might unknowingly execute code that performs operations which are either not meaningful or, worse, could lead to unexpected or incorrect results. Experience has shown that a significant portion of these logic errors is somehow still related to the types of the variables, provided one adopts a type system that differs from the standard one built into

the considered programming language. Ad hoc type systems have been designed that are able to prevent some classes of programming errors: for instance, *session types* have been developed to help in checking that a concurrent program fulfills the requirements of a given communication protocol [6]; in safety critical contexts, the MISRA-C coding standard [12] defines the *essential type system* (which among other things forbids some of the implicit type conversions that are legal for C code) and requires that the program is well typed according to its rules. The examples above have in common the fact that these non-standard type systems have a *prescriptive* nature: a deviation from the typing rules is considered an error which should be corrected.

In many cases a clear distinction between correct code and wrong code cannot be made: the tool maybe identifies a *smell* in the code and, being in doubt, simply reports it to the programmer. For instance, almost all compilers can issue a rich set of warnings: when clear and to the point, this feedback is arguably useful and greatly appreciated by the programmer. This is also the reason for the development of *linter* tools, i.e., lightweight tools that assist the programmer in improving code quality by spotting questionable code. Available linter tools differ in two main dimensions: the considered programming language and the kind of issues they focus on. The latter ranges from low level issues (e.g., respecting variable naming conventions or software metric thresholds) to higher level issues, which often take into account the intended semantics of a portion of code.

In this paper, we argue for the development of a linter tool for data science code (in particular, Python code) that is focused on detecting high level, data science related code smells. The tool works by collecting information about the possible runtime values of variables into an *abstract type system*, featuring high level datatypes that are specific for data science code; calls to data science library functions are checked for consistency with the computed abstract datatypes. Note that the type system differs from the ones mentioned above in that it is *descriptive*, rather than prescriptive: namely, it is not meant to be an oracular tool dictating to the user, with absolute certainty, what they should do; rather, the aim is to inform the user about possible unwanted behaviors, encouraging critical thinking about the adopted solution. Furthermore, due to the widespread adoption of these tools and their interdisciplinary nature, data scientists are not necessarily software engineers or professional developers, so a tool with the purpose of our prototype would offer additional support during development.

While the design of our linter tool is still work in progress, we are already working on a prototype implementation which is based on `LYRA` [24], a static analyzer for Python. Note that, by their intrinsic nature, linter tools should focus on identifying the most frequent logic errors; hence, having a working prototype is of great help in guiding its own design and development. Moreover, it is our opinion that the Abstract Interpretation framework [3], being able to model the concepts of approximation and abstract domain refinement, turns out to be adequate for the incremental development of a descriptive (i.e., permissive) type system.

Paper Structure. Section 2, after briefly describing related work, shows some examples of code smells that inspired this paper. Section 3 provides a high level view of the linter tool we are proposing, focusing on the description of the abstract type system used by the analysis and giving examples of the corresponding type rules. We conclude in Section 4 by discussing possible future work and plans for a thorough experimental evaluation.

2 Background

2.1 Data Science

Data science is an interdisciplinary field, bringing together different forms of knowledge, skills and expertise cooperating together to deliver a final, valuable result in terms of application or decisional strategy. The most valuable asset in data science are data themselves, from which all the subsequent steps eventually proceed. Therefore, data should be managed and manipulated carefully, in order to avoid misinterpretation that could lead to fallacies.

Before starting to use data to build models or extract wisdom, a first, almost mandatory, processing step is Exploratory Data Analysis (EDA) [21], whose aim is to examine and summarize the data, extracting meaningful characteristics, patterns, and relationships, employing descriptive statistics and quite often also data visualization techniques. Errors and inaccuracies in this preliminary step can easily lead to a completely different interpretation of the data and, possibly, to different strategical decisions. As a consequence, many software libraries have been developed to simplify and streamline this EDA phase, so as to try and decrease the chances to incur into errors.

The most prominent library for this task is `pandas` [16], which allows the user to perform rather complex operations without writing a lot of code and hence is primarily adopted for data analysis, data manipulation and data cleaning. `Pandas` allows reading the data directly from many sources, such as CSV or JSON files, and handles them using Dataframes, a two-dimensional data structure holding data like a two-dimension array or a table with rows and columns, and Series, a one-dimensional labeled array; in both cases, the data elements can be of any type (integers, strings, Python objects, etc.).

In addition to the classic EDA which employs descriptive statistics, `pandas` allows the user to produce plots using underlying Python modules such as `matplotlib` [9], providing a rich set of data visualization tools. Plots are one of the most direct and easy way to summarize and understand the data, but they should be chosen wisely because different kinds of

plots can highlight or hide part of the available information more than others.

In this context, the goal of our prototype is to guide the user to achieve the expected results, avoiding unexpected behavior by reporting possible code smells.

2.2 Related Work

Due to the importance and pervasiveness of data science, the need to analyze Jupyter Notebooks has been highlighted [25], and many techniques to analyze data sciences code have been proposed accordingly. For example, [14, 22, 23] propose a framework based on Abstract Interpretation [3] to infer necessary conditions on the structure and values of the data read by a data-processing program or to automatically detect unused input data [24]. Other static analysis frameworks focus on detecting data leakage [5, 19, 20] or studying the impact of code changes across code cells in notebooks. On the other end, open-source tools like `pandera` [1] and `pynblint` [18] have been released with the aim to perform data validation using schemas, and reveal potential notebook defects, recommending corrective actions that promote best practices such as using version control and putting `import` statements at the beginning of the notebook.

Regarding static type analysis and inference, many tools based on Abstract Interpretation, such as [11, 13], or relying on Z3 [4] or other SMT solvers, such as [8], have been proposed. However, these tools typically focus on inferring Python type hints [7] and detecting potential errors. They usually target the standard Python language and some standard libraries (e.g., `os`, `json`), aiming to infer concrete type hints and errors.

In contrast, our goal is to infer and reason about more abstract datatypes, potentially capturing a broader and less conventional set of errors and code smells. Our work is inspired by these projects but aims at finding more subtle code smells and proposing an easily extensible framework to help developers achieve correct results.

2.3 Plain Errors, Logical Mistakes and Code Smells

In this section we provide a few examples of the kind of coding issues that, in our opinion, should be reported by a data science linter tool; while doing it, we will also indirectly describe those kinds of issues that are not really meant to be the target of such a tool and hence should be disregarded.

`Pandas` handles a wide range of internal data types such as integers, floats, booleans, strings, timestamps, periods, and categorical data. It offers detailed documentation and it enforces some form of runtime type checking that does not allow the execution of certain operations on certain data types: when the user tries to execute a forbidden operation, an exception is raised. An example showing this mechanism is provided in Figure 1, where a `ValueError` exception is raised when the user tries to compute the mean on

```
In [1]:
import pandas as pd
x = ["Apple", "Orange", "Apple", "Apple",
    "Orange", "Apple"]
df = pd.DataFrame(x, columns=["Fruit"])
mean = df["Fruit"].mean()
```

```
Out [1]:
ValueError: could not convert string to
float: 'AppleOrangeAppleAppleOrangeApple'
```

Figure 1. An attempt to compute the mean of a string-type DataFrame column resulting in a ValueError exception.

a DataFrame whose elements' type is `str`. Since the execution is interrupted,[1] no unexpected behavior can emerge: as a consequence, this kind of *plain errors* can be safely ignored by our prototype tool.

Unfortunately, the previous example of an error causing program interruption can be considered the unlikely case: due to the intrinsic flexibility of the data science pipeline, a more frequent scenario is one where the computation does not end in an error and simply proceeds by computing and/or visualizing data that are not really meaningful, leading to *logical mistakes*. Detecting all forms of logical mistakes is clearly impossible, so that in general neither this kind of problems can be considered a meaningful target. However there might be cases where, by applying some form of simple approximate reasoning, a logical mistake can be traced back to the misuse of a specific data science library function, which appears to be called with inappropriate arguments. These (data science) *code smells* are the target of our analysis.

Misleading data visualization. As a first example, we consider the visualization component. `pandas` allows the user to plot data in many ways: in principle, the user should carefully select the kind of plot so as to provide a meaningful description of the data at hand; however, the available runtime type checks can do very little, if anything, in this respect. Consider the code shown in Figure 2 and the generated line plot shown below, on the left of the figure: here, a string data type (probably, the labels of some categorical data) on the x-axis is related to a numeric datatype on the y-axis. Even though at a first sight this plot looks reasonable, the specific choice of a *line* plot is questionable: a line plot hints at a continuous function modeling the relation between domain and codomain values, so that the user is implicitly encouraged to reason about, e.g., function monotonicity, local minima and maxima, or even to approximate missing values by linear interpolation. Clearly, all of the above makes little sense if the x-axis is representing nominal-scale (i.e., unordered) categorical data; in such a context, a bar chart,

[1] Exception handling code is seldomly used in data science code.

In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd

x = ["Apple", "Banana", "Orange", "Grape",
    "Strawberry"]
y = [10, 15, 20, 12, 18]

df = pd.DataFrame({"Fruit": x,
                   "Amount": y})
```

In [2]:
```
# code smell: line plot
plt.plot(df["Fruit"], df["Amount"])
```

In [3]:
```
# correct code
plt.bar(df["Fruit"], df["Amount"])
```

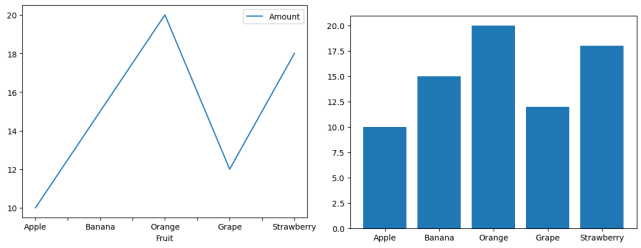

Figure 2. On the left, a line plot relating a string-type column and an integer-type column of a DataFrame. No exception is raised, although this plot can be deemed inadequate. On the right, a bar plot providing an appropriate visualization.

shown in the right hand side of Figure 2, would have been more appropriate.

The previous example impacts how the data can be interpreted; nevertheless some errors can be even more insidious because they can go completely unnoticed.

Misleading central tendency measures. As an example, consider the code shown in Figure 3. Here the user wants to compute the average for a set of speedup values, which in turn are computed as the ratio between timing data; this is a rather frequent computation, especially in benchmarking. We can note, though, that in the 2nd code block the user is computing the arithmetic mean of the speedups; in general, this may be significantly different from the geometric mean (whose computation is shown in the 3rd code block). In the context above, the geometric mean should be preferred since it provides a more faithful and accurate account for the multiplicative effects of performance improvements, whereas the arithmetic mean can be skewed by extreme values, probably yielding an optimistic over-approximation of the speedups.

A similar example could be shown where a user willing to obtain a central tendency measure of some *ordinal* categorical data could inadvertently compute the arithmetic or geometric mean of the corresponding ranking indices: in this

In [1]:
```
import pandas as pd
from scipy.stats import gmean
t1 = [1.4, 5.5, 4.9, 3.9]
t2 = [3.2, 9.8, 1.3, 1.2]

df = pd.DataFrame({'t1': t1, 't2': t2})
df['speedup'] = df['t1'] / df['t2']
```

In [2]:
```
# code smell: arithmetic average
avg_spdup = df['speedup'].mean()
print(avg_spdup)
```

```
2.0044888147566717
```

In [3]:
```
# corrected code
avg_spdup = gmean(df['speedup'])
print(avg_spdup)
```

```
1.3169299965028327
```

Figure 3. Jupyter notebook code that shows how arithmetic mean and geometric mean can lead to different results. Since the mean is computed on speedup values, which are computed as ratios, the geometric mean is more appropriate.

case, common wisdom holds that the median value should be preferred.

Central tendency measure of scaled data. As another example, the code in Figure 4 shows an instance of a data manipulation processing step occurring frequently in the data science pipeline, which is data normalization and scaling. The goal is to transform and remap data to a new range of values according to some criteria, such as removing the mean and scaling to unit variance or scaling the values to a given range (usually the interval $[0, 1]$). For instance, when considering data that is going to be used in a machine learning model, data normalization and scaling are necessary to apply feature scaling, so as to prevent features with large ranges from dominating the model and to reduce dimensionality; as positive side effects, feature scaling also comes handy to improve data visualization and to reduce the effects of outliers. In the 1st code block of Figure 4 the user scales a Series of numeric values using a `StandardScaler`, resulting in new standardized values, with mean equal to 0 and standard deviation equal to one. Since the scaled values are still numeric values, in principle it is possible to compute their mean value, as done by the user in the 2nd code block. However, the mean of the scaled values is different from the mean of the original values, computed in the 3rd code block. This code smell can be very subtle to detect because no warning or issue of any kind is raised, but the impact on the data can be huge in terms of difference of the results.

In [1]:
```
import pandas as pd
from sklearn.preprocessing import
    StandardScaler
x = [1, 2, 3, 4, 5]
df = pd.DataFrame(x, columns=['x'])
sc = StandardScaler()
df['x_norm'] = sc.fit_transform(df[['x']])
```

In [2]:
```
# code smell: mean of normalized data
x_mean = df['x_norm'].mean()
print(x_mean)
```

```
0
```

In [3]:
```
# correct code
x_mean = df['x'].mean()
print(x_mean)
```

```
3.0
```

Figure 4. Jupyter notebook code that contains one example of the captured code smells.

Data leakage. As for the errors regarding the central tendency measure of scaled data, in other scenarios, the order in which some operations are performed can lead to mistakes. This is the case for a common error in machine learning: splitting the train and test sets after having already normalized the data. Normalization is an important step that can improve the performance of the model that is being trained; splitting the data into train and test sets is a standard procedure, often performed using the `train_test_split` method, for the validation of the model to check how well it has learned to generalize after the training phase. In this scenario, it is important to normalize the data *after* it has been split to avoid data leakage: the undesired mechanism through which some information from the train set flows into the test set and provides distorted validation results that do not reflect the real generalization capabilities of the trained model. The reason behind the data leakage in this context can be found in the parameters used during the normalization phase: if they are computed on the whole dataset, they can also be influenced by values that are present in the test set and that the model should not have bias on.

Code smells can be subtle yet dangerous, for this reason, in our abstract domain and analyzer, we plan to target `pandas` operations that could lead to this kind of errors. However, since `pandas` is mostly involved in the EDA phase, which is just the initial step of the data science pipeline, we plan to extend our approach to other data science libraries, such as `scikit-learn`, in order to check that the machine learning models are created by using the algorithms that mostly adapt to the available data.

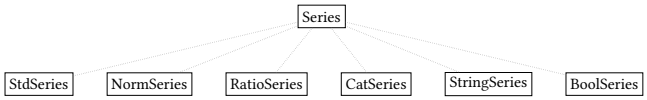

Figure 5. Diagram of the abstract domain specific to Series.

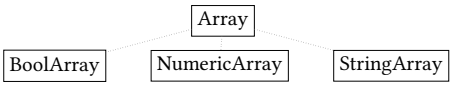

Figure 6. Diagram of the abstract domain specific to arrays.

3 Domain and Implementation

In this section we describe the design a linter tool for data science, which is paired by the corresponding development of a prototype implementation. Strictly speaking, our current prototype can only handle Python code. However, since many data science applications are developed in `Jupyter Notebooks`, we have extended the applicability of the prototype by implementing a simple preprocessing phase, which collects all Python code from the cells of the notebook, also removing specific *magic commands* related to the environment. Note that, for the scope of this project, we do not consider the case of arbitrary code cell execution, even if this would be allowed in `Jupyter Notebook` (and a possible source of programming errors in itself [20]). Rather, we focus on the sequential execution of all the Python code extracted from the cells of the notebook (i.e., the code interpreted from the first to the last cell).

After the code extraction phase, the prototype invokes the analysis phase: currently, this is structured as a classical static analysis, separating the fixpoint computation engine from the abstract domain. As a matter of fact, the implementation of the prototype has been obtained by extending the Lyra abstract interpreter [24]. Most of the design and implementation work has been dedicated to:

- the definition of an abstract domain expressing *abstract datatypes* (to be discussed in the following);
- the writing of abstract datatype rules for the most widely used data science library functions;
- the implementation of corresponding type checkers, triggering the generation of suitable reports when identifying the targeted code smells.

The tool implements a classical forward static analysis, maintaining a non-relational abstract state that maps each program variable to the corresponding abstract datatype. When processing a function call, the analyzer checks for a corresponding abstract datatype rule and, if one can be found, it applies it and updates the abstract state accordingly; otherwise, if no rule can be found, it falls back to providing a safe, but typically imprecise, over-approximation.

It is worth stressing that the development and implementation of our prototype tool are inherently incremental; in particular, during the tool development process, the fallback case can be modified so as to notify the tool developers of those library function calls that are still missing a dedicated abstract datatype rule; this allows to obtain a simple prioritization of the rule creation process. Also note that the addition of a rule for a specific function sometimes triggers a corresponding change in the set of abstract values that can be tracked by the abstract domain.

Abstract datatypes

We now informally describe the elements of the abstract datatype domain used by the current prototype; as said before, this domain is still subject to be extended, if this allows for significant more precision when modeling specific data science library calls.

- Several abstract datatypes are in direct correspondence with concrete datatypes that are built-in in the language; for instance, the scalar types `Bool` and `String` and the collection datatypes `Array`, `List`, `Dict`, `Set`.
- Other abstract datatypes are in direct correspondence with those defined in specific data science libraries, such as `DataFrame` and `Series`.
- A few abstract datatypes are introduced to intuitively model the join of several concrete datatypes, when there seems to be no gain in keeping a fined grained differentiation; for instance, datatype `Numeric` is for variables storing a numeric scalar value, no matter if integral or floating point.
- Some abstract datatypes are introduced to model specific library functions: *encoders* (e.g., `LabelEncoder`, `OneHotEncoder` and `OrdinalEncoder`) are used to model `scikit-learn` transformers mapping the representation of categorical variables into numeric variables, so as to allow further processing; and *scalers* (e.g., `StdScaler`, `MinMaxScaler` and `MaxAbsScaler`), which can be used as explained when discussing Figure 4 to remap data ranges according to different criteria.
- When deemed useful, new datatypes have been introduced to refine the concrete ones, so as to keep track of relevant properties such as the way a value has been computed. In Figure 5 we show the refinements available for the `Series` datatype: for instance, datatype `NormSeries` indicates that the values in the series have been subject to normalization. Similarly, in Figure 6 we show the refinements for the array collections (similar refinements have been defined for list collections); the reason why the `Array` datatype happens to have fewer refinements with respect to the `Series` datatype is that they are less frequently used in calls to the relevant data science library functions.

As usual, the domain includes top ($\top$) and bottom ($\bot$) elements and the lattice partial order ($\sqsubseteq$) encodes the precision of each abstract datatype, i.e., the subset relation for the corresponding set of concrete values. In our prototype, each variable is assigned a single abstract type, although extending the analysis to a disjunctive form, where each variable is mapped to a finite set of possible types, is a possible future direction.

Type rules and Code Smells

The static analysis computes and propagates type information in a usual way, maintaining the abstract type environment Γ. Newly encountered variables are added to Γ and mapped to the top element $\top$, meaning that nothing is known about their abstract datatype; optional Python (concrete) type annotations are taken into account. As an example, the following type rule for the division operator, which comes into play when analyzing the code in Figure 3, tracks the generation of a `RatioSeries` value:

$$\frac{\Gamma \vdash x : t_x \qquad \Gamma \vdash y : t_y \qquad t_x, t_y \sqsubseteq \texttt{Series}}{\Gamma \vdash x/y : \texttt{RatioSeries}}$$

Clearly, a more interesting case is the one of data science library calls, which are the main source of refined abstract datatype information. The analyzer disregards the actual implementation of these library functions and simply relies on their abstract datatype signature, which is encoded in corresponding type rules. For example, the following type rule models the reading of a `DataFrame` from a CSV file:

$$\frac{}{\Gamma \vdash \texttt{read\_csv}(\textit{filename}) : \texttt{DataFrame}}$$

Note that the lack of premises in the rule above is on purpose: no requirement at all is placed on argument *filename*, because our type system is not meant to target low level errors, mistakes or code smells that are not strictly related to the high level usage of data science libraries.

As another example, the following type rule comes into play when analyzing the code in Figure 4:

$$\frac{\Gamma \vdash x : t_x \qquad t_x \sqsubseteq \texttt{Series} \qquad \Gamma \vdash s : \texttt{StdScaler}}{\Gamma \vdash s.\texttt{fit\_transform}(x) : \texttt{StdSeries}}$$

As said before, when no type rule exists or when merging two control flows where variables are typed with incomparable types, the type of the result is the top element $\top$.

The rule specification process also needs to take into proper account a few peculiarities of the considered data science libraries; in particular, most of the `pandas` library functions allow for using the boolean flag `inplace` to switch from the default functional-style specification, which returns a newly computed value without changing the input arguments, to an imperative-style variant specification, where the operation directly modifies its input arguments without

Gmean Warning

GmeanWarning: Warning [definite]: in mean(df["speedup"]) @ line 7 -> df["speedup"] is a RatioSeries, gmean should be used.

Figure 7. Definite warning raised during the analysis of the code shown in Fig 3.

returning a value. As an example, a function call such as

```
result = x.fillna(val, inplace=True)     (1)
```

replaces all the null values occurring in series `x` by the scalar value `val`; in particular, the function call does *not* return a series value, so that the assignment to variable `result` is likely a code smell. Our abstract domain provides the abstract element `NoneRet` which is used, for instance, in the type rule schema

$$\frac{\Gamma \vdash x : t_x \qquad t_x \sqsubseteq \texttt{Series}}{\Gamma \vdash x.\mathit{funcname}(\texttt{inplace=True}) : \texttt{NoneRet}}$$

where parameter *funcname* is instantiated using the relevant library function names. A similar rule schema is defined where t_x is a subtype of `Dataframe`.

As said before, the creation of the abstract domain is guided by the identification of code smells like the ones discussed before; specifically, our current prototype is able to identify and report instances of code smells corresponding to the patterns examined in Figures 2, 3 and 4, as well as code smells matching the pattern in Equation (1). As an example, in Figure 7 we show the warning generated when analyzing the code shown in Figure 3, hinting the user that the arithmetic mean is inappropriate for `RatioSeries` and the geometric mean should be used instead.

Since we focus on executions that do not cause runtime errors or trigger exceptional behaviors, our prototype does not raise warnings for code such as the one shown in Figure 1. Also note that the datatype analysis is meant to provide an overapproximation of the computed values and hence, in principle, it is correct; however, when actually using this information to perform the type checks, we deliberately select and report only those code smells that are likely to be true positives. Hence, when considered as a whole, our prototype behaves as other linter tools, allowing for both false positives and false negatives, in an attempt to obtain a reasonable signal-to-noise ratio.

Discussion

The incremental design and implementation of our linter tool has highlighted several directions for further development. First, we believe that the overall usefulness of the tool would be greatly enhanced by extending it to support code annotations. These would be used for several purposes:

- annotations provided by the data science programmer can be used to silence those reports that happen to be false positives;
- the user may also want to annotate the input variables, which otherwise are assumed to hold arbitrary values, so as to improve the precision of the analysis;
- annotations could also be used by the developers of the linter tool, so as to simplify the addition of type rules when integrating a new data science library (currently, the type rules are more or less hard-coded into the prototype implementation); in this respect, we could adapt the approach put forward in [15] for the standard C library.

The annotation language should be as simple as possible, in particular when considering the first two cases above, which require a direct interaction from a data scientist. In this respect, a possibility that is worth investigating is for the tool itself to provide annotation hints. For instance, whenever reporting a code smell, the tool could also suggest the annotation required to suppress the report, if the user thinks it is a false positive. Similarly, the linter tool could systematically use the results of the static analysis to provide *tentative* abstract type annotations for the variables in the program: these can later be checked by the end user and maybe validated into proper annotations, with a significantly reduced programming effort. Clearly, the development of a suitable, simple annotation syntax would benefit from a close interaction with the data scientists.

Another clear direction for extension is the identification of other data science code smells that could be detected and reported by the linter tool. As an example, it is rather common to compute the Pearson correlation coefficient between a pair of variables: since this implicitly assumes that data is continuous and linearly related, it is not really adequate for ordinal data (where the Spearman coefficient is a more suitable choice). Once again we believe that a close interaction with data scientist would be beneficial, as often code smells are in direct correspondence with deviations from rule-of-thumbs and methodological antipatterns that are more or less well-known by experienced users.

4 Conclusions and Future Work

In this paper we have described the ideas underlying our ongoing work on the design and development of a high level linter tool for data science code. The main characteristic of our proposal is that it is meant to be incrementally extended in three directions: the abstract domain for the underlying static analysis, the specification of the type rules for data science library functions, and the set of warnings that can be reported. This fits well within the Abstract Interpretation framework, which allows for adapting the abstract domain and operators so as to fine tune the precision of the analysis.

We believe that the use of a high level linter tool can create a positive interaction with the user, which can interpret the warnings raised as some sort of automated code review. To make the adoption and development of this tool more beneficial, some interaction with expert data scientists is desirable, so as to understand the most subtle yet common mistakes that this prototype could help avoid.

Regarding the experimental evaluation, we plan to run the analyzer on a broad benchmark of Jupyter Notebooks published on Kaggle[2].

Acknowledgments

This work has been supported by SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU. The authors would like to thank Francesco Bernini and Daniele Molinari for their help with the implementation of the prototype.

References

[1] Niels Bantilan. 2020. pandera: Statistical Data Validation of Pandas Dataframes. In *Proceedings of the 19th Python in Science Conference 2020 (SciPy 2020), July 6 - 12, 2020*, Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe (Eds.). scipy.org, 116–124. https://doi.org/10.25080/MAJORA-342D178E-010

[2] Longbing Cao. 2017. Data Science: A Comprehensive Overview. *ACM Comput. Surv.* 50, 3 (2017), 43:1–43:42. https://doi.org/10.1145/3076253

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[4] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[5] Filip Drobnjakovic, Pavle Subotic, and Caterina Urban. 2024. An Abstract Interpretation-Based Data Leakage Static Analysis. In *Theoretical Aspects of Software Engineering - 18th International Symposium, TASE 2024, Guiyang, China, July 29 - August 1, 2024, Proceedings*. Springer.

[6] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

[7] Łukasz Langa Guido van Rossum, Jukka Lehtosalo. 2014. PEP 484 – Type Hints. https://peps.python.org/pep-0484/

[8] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, 2018, Proceedings (LNCS, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 12–19. https://doi.org/10.1007/978-3-319-96142-2_2

[9] John D. Hunter. 2007. Matplotlib: A 2D Graphics Environment. *Comput. Sci. Eng.* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[10] Thomas Kluyver et al. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[11] Teddy Sudol Martin Demello Alvaro Caceres Dave Baum Arthur Peters Peter Ludemann Paul Swartz Ned Batchelder Allison Kaptur Laura Lindzey Matthias Kramm, Rebecca Chen. 2019. Pytype: A static type analyzer for Python code. https://github.com/google/pytype

[12] MISRA. 2013. *MISRA-C:2012 - Guidelines for the use of the C language in critical systems*. MIRA Limited, Warwickshire CV10 0TU, UK.

[13] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPICS.ECOOP.2020.17

[14] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023*, Pietro Ferrara and Liana Hadarean (Eds.). ACM, 8–13. https://doi.org/10.1145/3589250.3596145

[15] Abdelraouf Ouadjaout and Antoine Miné. 2020. A Library Modeling Language for the Static Analysis of C Programs. In *Static Analysis - 27th International Symposium, SAS 2020, November 18-20, 2020, Proceedings (LNCS, Vol. 12389)*, David Pichardie and Mihaela Sighireanu (Eds.). Springer, 223–247. https://doi.org/10.1007/978-3-030-65474-0_11

[16] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[18] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Pynblint: a static analyzer for Python Jupyter notebooks. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN 2022, Pittsburgh, Pennsylvania, May 16-17, 2022*, Ivica Crnkovic (Ed.). ACM, 48–49. https://doi.org/10.1145/3522664.3528612

[19] Pavle Subotic, Uros Bojanic, and Milan Stojic. 2022. Statically Detecting Data Leakages in Data Science Code. In *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, Laure Gonnord and Laura Titolo (Eds.). ACM, 16–22. https://doi.org/10.1145/3520313.3534657

[20] Pavle Subotic, Lazar Milikic, and Milan Stojic. 2022. A Static Analysis Framework for Data Science Notebooks. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 13–22. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794067

[21] John Wilder Tukey et al. 1977. *Exploratory data analysis*. Vol. 2. Springer.

[22] Caterina Urban. 2020. What Programs Want: Automatic Inference of Input Data Specifications. (2020). https://arxiv.org/abs/2007.10688

[23] Caterina Urban. 2023. Static Analysis for Data Scientists. In *Challenges of Software Verification*. Springer, 77–91.

[24] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (LNCS, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 683–710. https://doi.org/10.1007/978-3-319-89884-1_24

[25] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better code, better sharing: on the need of analyzing jupyter notebooks. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas*

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

and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 53–56. https://doi.org/10.1145/3377816.3381724

[26] Michael L. Waskom. 2021. seaborn: statistical data visualization. *Journal of Open Source Software* 6, 60 (2021), 3021. https://doi.org/10.21105/joss.03021