PYRA: A High-level Linter for Data Science Software

Greta Dolcetti[a], Vincenzo Arceri[b], Antonella Mensi[c], Enea Zaffanella[b], Caterina Urban[d] and Agostino Cortesi[a]

[a] Ca' Foscari University of Venice, Via Torino, 155, Venice, 30170, Italy
[b] University of Parma, Parco Area delle Scienze, 53/A, Parma, 43124, Italy
[c] University of Verona, Piazzale L. A. Scuro, 10, Verona, 37134, Italy
[d] Inria & École Normale Supérieure | Université PSL, Paris, France

ARTICLE INFO

Keywords:
Static Analysis
Jupyter Notebooks
Data Science

ABSTRACT

Due to its interdisciplinary nature, the development of data science software is particularly prone to a wide range of potential mistakes that can easily and silently compromise the final results. Several tools have been proposed that can help the data scientist in identifying the most common, low-level programming issues. However, these tools often fall short in detecting higher-level, domain-specific issues typical of data science pipelines, where subtle errors may not trigger exceptions but can still lead to incorrect or misleading outcomes, or unexpected behaviors.

In this paper, we present PYRA, a static analysis tool that aims at detecting code smells in data science workflows. PYRA builds upon the Abstract Interpretation framework to infer abstract datatypes, and exploits such information to flag 16 categories of potential code smells concerning misleading visualizations, challenges for reproducibility, as well as misleading, unreliable or unexpected results. Unlike traditional linters, which focus on syntactic or stylistic issues, PYRA reasons over a domain-specific type system to identify data science-specific problems – such as improper data preprocessing steps and procedures' misapplications – that could silently propagate through a data-manipulation pipeline. Beyond static checking, we envision tools like PYRA becoming integral components of the development loop, with analysis reports guiding correction and helping assess the reliability of machine learning pipelines. We evaluate PYRA on a benchmark suite of real-world Jupyter notebooks, showing its effectiveness in detecting practical data science issues, thereby enhancing transparency, correctness, and reproducibility in data science software.

1. Introduction

Data science informally refers to an interdisciplinary field that integrates concepts from statistics, informatics, computing, communication, management, and sociology to analyze data and its environment (including domain-specific, organizational, and societal aspects). The ultimate aim of this discipline is to extract valuable insights from data that can be used for interpretative purposes or to assist in decision-making, following a data-to-knowledge-to-wisdom approach and methodology [3]. Given the widespread adoption of data science-based approaches across various fields – healthcare, retail, manufacturing, finance, etc. – several data science tools and libraries have become widely popular. These include, but are not limited to:

- `scikit-learn` [27], a Python library that allows the development of a complete machine learning pipeline;
- `pandas` [20], a Python library for data manipulation and analysis;
- `seaborn` [48] and `ggplot2` [49], which are data visualization tools designed for Python and R, respectively;
- `Jupyter Notebooks` [16], a web application that, through the use of notebooks, allows to write and execute code, visualize data and add comments within one interface;
- `BioConductor` [11], an R ecosystem that encompasses a wide variety of bioinformatic tools.

This list of tools and libraries also shows that Python and R are the programming languages of choice for data scientists. Both languages are dynamically typed, meaning that they perform their type correctness checks at runtime and do not enforce native support for a more systematic, static control of the operations that are allowed on the values of variables; this means that a typing error in a seldomly executed computational path will only be discovered when running a test that actually triggers the execution of that specific computational path. In contrast, statically typed languages perform most (sometimes all) of the type checks before running the program, checking all its possible execution paths: hence, they can eagerly spot the most common programming errors even before running a single dynamic test.

It is worth stressing that the mere adoption a statically typed language would provide no guarantee on the code being completely correct: the type checking tool (typically run as a step in the compilation phase) will spot all proper typing errors, but logical errors would remain undetected; when present, logical errors can lead to unwanted or misleading results that the user may wrongly accept as correct.

*Corresponding author

greta.dolcetti@unive.it (G. Dolcetti); vincenzo.arceri@unipr.it (V. Arceri); antonella.mensi@univr.it (A. Mensi); enea.zaffanella@unipr.it (E. Zaffanella); caterina.urban@inria.fr (C. Urban); cortesi@unive.it (A. Cortesi)

ORCID(s): 0000-0002-2983-9251 (G. Dolcetti); 0000-0002-5150-0393 (V. Arceri); 0000-0001-9468-5298 (A. Mensi); 0000-0001-6388-2053 (E. Zaffanella); 0000-0002-8127-9642 (C. Urban); 0000-0002-0946-5440 (A. Cortesi)

Experience has shown that a significant percentage of these logical errors can still be related to the "data type" of the program variables, provided the default type system of the considered programming language is replaced by a non-standard, higher level type system, suitably extended so as to detect and propagate the relevant information. For these scenarios, several *ad hoc* type systems have been developed: for instance, *session types* have been developed to help in checking that a concurrent program fulfills the requirements of a given communication protocol [10]; in safety critical contexts, the MISRA-C coding standard [21] defines the *essential type system* (among other things forbidding 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 approaches 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. However, such a clear-cut distinction between correct and wrong code cannot always be made. In the cases where the tool identifies a *smell* in the code the prescriptive approach is better replaced by a *descriptive* approach, where the tool stops pretending to have a complete knowledge and does its best to help the developer in understanding what is going on. For instance, almost all compilers can issue a rich set of warnings: when clear and to the point, this feedback is 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.

A proposal for the development of a linter tool for data science code, focused on the Python language, was put forward in [8]. The tool aims at detecting several data science related code smells by gathering information about the potential runtime values of variables into an *abstract type system*. The latter comprises high-level data types tailored specifically for data science code. Lastly, the tool verifies that calls to data science library functions are consistent with the determined abstract data types. As explained above, the tool adopts a *descriptive* approach: its end goal is to make the user reason about their code by reporting them a list of putative inappropriate behaviors, without obliging them to take a specific action; this fits rather well with the fact that data science code is highly context-dependent. The usefulness of this prototype is further enhanced by the fact that many data scientists are not code specialists, e.g., software engineers or professional developers. Indeed, data science is interdisciplinary, and the tools we have mentioned, such as pandas, are highly user-friendly for anyone with a basic understanding of programming.

In this paper we thoroughly extend [8, 7] and we present PYRA, a working prototype of the linter tool that is easy to use and integrates seamlessly with Python code, without requiring additional annotations or modifications of the code. The abstract datatype domain of PYRA comprises 56 datatypes – ranging from higher-level ones to others that are data science-specific – designed to capture 16 categories of the most common code smells, of various nature and gravity.

The implementation of PYRA is based on LYRA [44], a static analyzer for Python that automatically detects input data that remains unused by a Python program. It is a research prototype and its support for Jupyter notebook is only a proof of concept. It does not support any other detection of domain-specific issues as PYRA. More concretely, [8] lays the foundations for PYRA by motivating the need for a linter for data science code: the notion of code smells specific to data science is introduced using minimal examples, while formally describing the adopted abstract domain and the corresponding type rules. A refined version of the prototype introduced in [8] is informally presented in [7], where its functionalities and its utility are demonstrated adopting a more practical point of view.

Building upon the previous work, in this paper we describe a further improved version of the tool, characterized by additional checkers and a more robust implementation; the contributions also include a more detailed description of the tool's behavior, with an explanation and classification of the warnings produced, as well as an experimental evaluation conducted on real notebooks, resulting in a significant advancement compared to earlier efforts. We argue that the Abstract Interpretation framework [5], due to its ability to formalize approximation and support abstract domain refinement, is particularly well-suited for the incremental development of a descriptive (i.e., permissive) type system.

The rest of the paper is organized as follows. In Section 2 we briefly cover the related work, whereas in Section 3 we provide an overview on the code smells that we aim to detect, categorize them and describe some of them in detail. Section 4 thoroughly describes the proposed tool, PYRA, covering its architecture, its abstract datatype domain, the implemented checkers, and an example of its execution. Lastly, Section 5 is dedicated to the experimental evaluation, Section 6 discusses some limitations and important notes and in Section 7 we draw some conclusions and discuss potential ideas for future research.

2. Related Work

Abstract Interpretation [5] is a mathematical framework that allows to formally derive approximations of the semantics of programming languages. Its most common application is the systematic development of sound static analyzers, i.e., tools that are able to automatically infer some properties of a program without executing it. In particular, [4] shows how type systems and type inference algorithms can be cast as instances of Abstract Interpretation. A gentle introduction to the modeling of simple type information as Abstract Interpretation is the *dimension calculus* of [6, Section 2.2]: here it is shown how concrete unit of measures (e.g., meter, yard, second, hour, kilogram, pound, ...) can be approximated using abstract dimensions (e.g., length, time, mass, surface, speed, ...) and then propagated via abstract rules such as

$$\begin{aligned}
&\text{length} + \text{length} = \text{length},\\
&\text{length} \times \text{length} = surface,\\
&\text{length} \,/\, \text{length} = \text{nodimension},\\
&\text{length} \,/\, \text{time} = \text{speed},\\
&\dots
\end{aligned}$$

This simple idea can be easily generalized to more sophisticated type systems, such as the one we propose in this paper.

Due to the importance and pervasiveness of data science, the need to analyze Jupyter Notebooks has been highlighted [47], and many techniques to analyze data sciences code have been proposed accordingly. For example, [24, 42, 43] propose a framework based on Abstract Interpretation [5] 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 [44]. Other static analysis frameworks focus on detecting data leakage [9, 38, 39] or studying the impact of code changes across code cells in notebooks. On the other end, open-source tools like `pandera` [1] and `pynblint` [28] have been released with the aim to perform data validation using schemas (i.e. the specification of the expected structure, data types and validation rules for the data), 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 [17, 22], or relying on Z3 [23] or other SMT solvers, such as [13], have been proposed. However, these tools typically focus on inferring Python type hints [30] 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.

Even though not strictly related to the analysis of Jupyter notebooks, research on the R programming language, another one of the most popular languages for data and statistical analysis, is also noteworthy. In [35], the authors conducted a large-scale analysis of R programs, considering both scripts submitted with academic publications and those found in CRAN packages, investigating the most popular features, constructs and operations of R. Based on this study, [36] proposed `flowR`, a static dataflow analyzer and program slicer for R programs, which also supports its most challenging features, such as redefinition of primitive constructs. Finally, in [12], the authors propose a large-scale study on the usage of eval in R. They demonstrate that R allows a higher degree of flexibility in using eval compared to JavaScript, and they discuss the challenges associated with analyzing or refactoring code that employs eval while preserving its intended semantics.

To the best of our knowledge, there is not another framework specifically designed to infer and reason about abstract datatypes in Jupyter Notebooks and to capture a variety of data science code smells by also using concrete dataset information, as we do in PYRA. The most similar framework is MLScent [32], even though it focuses on lower level anti-patterns detection (e.g. missing docstring for function, magic numbers, array creation efficiency, etc.) and it only uses a fully static abstract syntax tree analysis. However, as shown in Section 5, on the two issues that can be detected by both tools, PYRA outperforms MLScent. Therefore, we claim that PYRA is the first framework that combines Abstract Interpretation with concrete dataset information to infer abstract datatypes and detect a wide range of data science code smells in Jupyter Notebooks.

3. Code Smells

In this section we provide an informal definition for what we call a *data science code smell*, along with the issues related to them and some minimal examples.

Generally speaking, a code smell is any characteristics of (a portion of) the source code that hints at the existence of a deeper problem, thereby hindering software maintenance and evolution [26]. Even though code smells are not necessarily bugs, they might cause issues and usually denote a weakness in the code design. In the context of data science code, we refine the definition above to mean any code denoting an operation that, while being legal according to the language of choice (i.e., it has a well defined behavior and does not raise an exception), it may be a logical or methodological mistake, potentially leading to computing results that are incorrect in the considered context.

As mentioned in Section 1, PYRA focuses on code smells that are specific to the data science pipeline when using the Python language. The set of 16 categories of code smells analyzed by PYRA was constructed by considering some of the most common and well-known issues that can arise in data science pipelines [50, 31, 18, 15], as well as some other general issues that can lead to misleading results or unexpected behaviors.

In this section, we provide descriptions and examples of the most representative ones, while a brief overview of all the included issues can be found in Table 1. For each code smell, in Table 1 we also provide:

- the classification type: whether the reported code smell is just a *suggestion*, where the choice of adopting a correction depends on context, or it is a more serious issue, posing a significant *problem* for the pipeline and having a widely recognized better approach to avoid its potential negative consequences;
- the detection method: whether the issue can be identified by using a purely *syntactic* analysis or it requires

Table 1
Warning description (alphabetical order).

Name	Description	Type	Method	Severity Level	Severity Explanation
	Misleading visualizations				
CategoricalPlot	A line plot is being used with categorical (nominal-scale) data on the x-axis	Suggestion	Semantic	Medium	This visualization can mislead users into interpreting categorical data as continuous, suggesting inappropriate concepts such as trends, interpolation, or monotonicity. A bar chart or similar categorical plot type should be used instead
PCAVisualization	PCA used to reduce dimensionality and visualize the data	Suggestion	Semantic	Low	PCA is not always the most appropriate technique for visualizing data
	Misleading results				
CategoricalConversionMean	A numerical average is being calculated on categorical data that has been implicitly converted to numerical codes	Problem	Semantic	Medium	Automatic conversion of categories to numeric codes could lead to unexpected or statistically meaningless results, since the numeric codes assigned to categories do not necessarily represent a quantitative relationship between the categories themselves
DataLeakage	Information outside the training set unfairly influences a machine-learning model	Problem	Semantic	High	Data leakage may cause overestimation of performance, poor generalization, and misleading insights
DuplicatesNotDropped	Duplicated rows present in a DataFrame were not removed	Suggestion	Syntactic	Medium	Duplicates may introduce data integrity issues or bias
FixedNComponentsPCA	Principal Component Analysis (PCA) with an a priori fixed number of components	Suggestion	Syntactic	Medium	These assumptions may cause loss of important information, inefficient dimensionality reduction, and failure to identify true patterns
Gmean	The arithmetic mean is computed on ratio-based data (such as speedups), where the geometric mean would provide a more accurate measure	Problem	Semantic	Medium	Arithmetic means can be misleading or overly influenced by extreme values in this context and may result in misleading results
InappropriateMissingValues	Using summary statistics in place of the missing values	Suggestion	Syntactic	Low	This approach may distort the original data distribution, affect the correlation between variables, and introduce bias
MissingData	The DataFrame contains missing values	Suggestions	Syntactic	Medium	Missing values may cause bias, reduce the quality of the analysis, and lead to incorrect conclusions
NotShuffled	The DataFrame has not been shuffled	Suggestion	Syntactic	Low	Unshuffled data may result in biased model training and overfitting
PCAOnCategorical	PCA applied to categorical data	Suggestion	Semantic	Medium	Applying PCA to categorical data may cause suboptimal results
ScaledMean	Mean on scaled data has no direct relationship to the original data	Problem	Semantic	Medium	This may cause misleading results
	Challenges for reproducibility				
Reproducibility	The random state is not set in `train_test_split` or `sample` function calls	Suggestion	Syntactic	Medium	This can cause reproducibility issues leading to inconsistent results
	General issues				
HighDimensionality	A large number of features (columns) relative to the number of observations (rows)	Suggestion	Syntactic	Medium	High-dimensional data may incur the curse of dimensionality
InconsistentType	The inferred abstract type is different from the user-annotated type	Suggestion	Semantic	Low	The user annotations may not be precise
NoneRetAssignment	Assignment to a variable in the lhs where the rhs evaluation returns None	Problem	Semantic	Low	This is most likely a code smell that may result in unexpected behavior or potential runtime errors

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

df = pd.read_csv("data.csv")

# DataFrame df with columns: 'Fruit', 'Amount'
# Values:
# [Apple-10, Banana-15, Orange-20,
#   Grape-12, Strawberry-18]
```

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

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

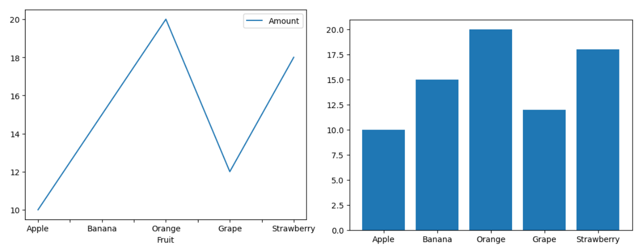

Figure 1: On the left, a line plot relating a string-type column and an integer-type column of a DataFrame. No exception is raised, although this plot can be deemed inadequate. On the right, a bar plot providing an appropriate visualization.

a deeper *semantic* approach, also considering the provenance and content of the data;

- the severity level (*low*, *medium*, *high*) of the issue, based on its potential impact on the pipeline and the influence it may have on the results.

For clarity, we categorize the code smells into four groups: misleading visualizations, misleading results, challenges for reproducibility, and general issues.

3.1. Misleading visualizations

To illustrate a potential issue in data visualization, let us consider a simple yet telling example. The pandas library offers a variety of ways to visualize data. Ideally, users should carefully choose the kind of plot that best fits the nature of the data at hand. However, in practice, runtime type checks provide little to no guidance in this respect. Consider the code shown in Figure 1 and the generated line plot shown below, on the left of the figure: here, a string data type (the labels of some categorical data) on the x-axis is related to a numeric datatype on the y-axis. Even though at first glance 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

In [1]:
```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

digits = datasets.load_digits()
digits_df = pd.DataFrame(data=digits.data)
digits_df['target'] = digits.target
X = digits_df.drop('target', axis=1)
y = digits_df['target']
```

In [2]:
```python
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y,
    cmap='jet', alpha=0.6)
```

In [3]:
```python
tsne = TSNE(n_components=2,
    perplexity=30,
    learning_rate=200,
    n_iter=1000,
    random_state=42)
X_tsne = tsne.fit_transform(X)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y,
    cmap='jet', alpha=0.6)
```

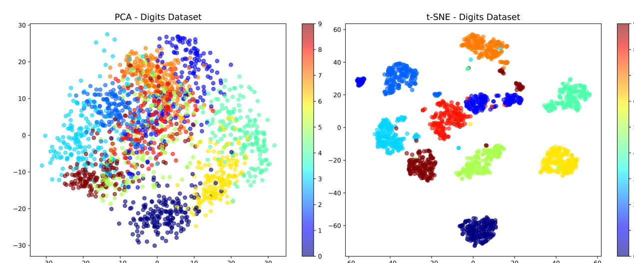

Figure 2: Comparison of PCA and t-SNE visualizations of the digits dataset. On the left, the plot resulting from PCA while on the right, the plot resulting from t-SNE. Redundant parts of the code related to plotting are omitted for clarity.

context, a bar chart, shown in the right hand side of Figure 1, would have been more appropriate.

Another example of a code smell that can lead to misleading visualizations is the use of Principal Component Analysis (PCA), a powerful dimensionality reduction approach, for visualization purposes. In detail, PCA generates a new set of uncorrelated features whose variance is maximized via a linear combination of the original ones. This new variance-based representation may not be the most meaningful for the problem at hand and it may lead to incorrect assumptions about the patterns within the data. An example is shown in the left plot of Figure 2, which illustrates that PCA fails to produce interpretable results, thus making highly difficult the identification of clusters within the data. Thus, quite often PCA is not the best approach for visualizing high-dimensional data, since its linear nature makes it less effective at capturing more complex, non-linear patterns in the data. In contrast, other methods such as t-distributed stochastic neighbor embedding (t-SNE) are designed to manage non-linear relationships, thus making them particularly suitable for visualizing complex datasets [19]. In detail, while PCA solely retains the global structures of the data, t-SNE is able to capture local ones by preserving the relationship between each pair of objects i.e., their similarity, in a lower dimensional space. The latter is particularly evident if we look at the right plot of Figure 2, which, unlike the left one, depicts clear and identifiable clusters.

The two above are examples of code smells leading to data representations being misinterpreted or confusing; the other code smell categories focus on more insidious errors, that in principle could go completely unnoticed.

```
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 3: An attempt to compute the mean of a string-type DataFrame column resulting in a ValueError exception.

3.2. Misleading results

While being tedious for the developer, plain programming errors and/or exceptions, like the one shown in Figure 3, which interrupt the normal execution flow and redirect it to error handling code (or even program termination), are actually beneficial: they force the developer to analyze and correct the issue that has arisen.

However, the highly dynamic nature and inherent flexibility of Python, combined with the vast ecosystem of libraries used in data science pipelines, can result in many code smells or logical mistakes going unnoticed. This happens because the inaccurate action is still syntactically valid and does not raise an exception: this behavior, often considered a feature of the language and its libraries, can lead to unintended consequences, where logical errors remain undetected and produce misleading results.

One of the most infamous and dangerous cases of misleading results is *data leakage*, which is examplified in Figure 4. Data leakage occurs when information contained in the test set is inadvertently used to train the model. This can happen when some pre-processing procedures, such as data scaling, missing data imputation, over or under-sampling, etc., are performed prior to splitting the dataset into training and testing sets. The consequences of data leakage can be severe, as it can result in models with overly optimistic performances on the training set, but poor generalization, i.e., they perform poorly on unseen data, leading to incorrect predictions and potentially harmful decisions.

Another example of a code smell that can lead to misleading results is the use of PCA with a fixed number of components (shown in Figure 5) or on categorical data. Indeed, it is common to set the number of components to 2 or 3, especially if PCA is also used for visualization purposes, or to choose a number based on prior knowledge of the data,

```
In [1]:
import pandas as pd
import numpy as np
from sklearn import StandardScaler,
accuracy_score, train_test_split,
LogisticRegression

df = pd.read_csv("data.csv")

X = df.iloc[:, :-1]
y = df.iloc[:, -1]

s = StandardScaler()
```

```
In [2]:
# Code smell: data leakage
# Test info leaks into training
X_s = s.fit_transform(X)

X_tr, X_ts, y_tr, y_ts = train_test_split(X_s, y)
```

```
In [3]:
# Corrected code
# Split before scaling
X_tr, X_ts, y_tr, y_ts = train_test_split(X, y)

X_tr = s.fit_transform(X_tr)
X_ts = s.transform(X_ts)
```

```
In [4]:
m = LogisticRegression()
m.fit(X_tr, y_tr)
```

Figure 4: A code snippet demonstrating an approach that causes data leakage and the correct way to prevent it. The code is not executable as-is due to shortened imports for improved readability.

```
In [1]:
import pandas as pd
from sklearn.decomposition import PCA

df = pd.read_csv("data.csv")
pca = PCA(n_components=3)
df_pca = pca.fit_transform(df)
print(df_pca)
```

Figure 5: An example of PCA with a fixed number of components.

e.g., the number of classes. However, this approach can lead to overfitting, as the model may capture noise in the data rather than the underlying structure. To address this, it is essential to fine-tune this parameter, which can be achieved by objectively analyzing the results obtained with different number of components using various metrics, e.g., as the cumulative explained variance ratio of the components or the performance of a machine learning model. Similarly, applying PCA on categorical data can lead to misleading results, as it is designed for continuous data and may not capture the underlying structure of categorical data, resulting in sub-optimal performances. In such cases, it is preferable to use Multiple Correspondence Analysis (MCA), if all features are categorical, or mixed PCA, which is a technique combining MCA and PCA.

In [1]:
```python
import pandas as pd
import numpy as np

values = [25, 29, 28, 30, 27, np.nan, 150]
df = pd.DataFrame({'values': values})
# Median:  28.50, std dev: 49.92

df.fillna(df['values'].mean(), inplace=True)
# Median:  29.00, std dev: 45.57
```

Figure 6: An example of inappropriate missing values handling, where the mean is used to impute missing values and this leads to a different distribution of the data.

Moreover, several other issues can lead to misleading results, depending on the data itself or missing procedures. For example, this occurs when duplicates are not removed, the data is not randomly shuffled, or missing data is not handled correctly. In some contexts, failing to remove duplicates can result in biased outcomes, as the model may learn from repeated instances rather than the actual data distribution. For example, a measurement that has been erroneously recorded twice by a sensor does not provide additional information but it only introduces redundancy and unbalances the dataset. Similarly, not shuffling the data can introduce bias, causing the model to learn patterns from the order of the data rather than its underlying distribution.

Missing data can also lead to biased results if not properly addressed. Improper handling of missing values can alter the data distribution, leading to incorrect conclusions. For example, imputing missing values using summary statistics often introduces bias and skews the data distribution, e.g., the mean is highly sensitive to outliers, as shown in Figure 6. In such scenarios, it would be wiser to adopt more complex data imputation techniques, e.g., MissForest [37] or KNNImputer [41], to obtain more reliable estimates. Alternatively, depending on the context and the ratio of missing data, one could remove either the affected sample or feature.

3.3. Challenges for Reproducibility

One of the reasons why data science pipelines are often difficult to reproduce is the lack of proper documentation and version control. This can lead to confusion and misunderstandings about the data, the analysis, and the results. For example, if the data is not properly documented, it may be difficult to understand how it was collected, what it represents, and how it was processed. On the other hand, even if the data is already provided, it may be difficult to reproduce the analysis if some preventive measures are not adopted. For example, some procedures are inherently random by default, therefore difficult to reproduce. In this case, it is important to set a random seed to ensure that the results are reproducible. This is especially important when using machine learning algorithms, as they often rely on randomness to initialize parameters or select subsets of data, i.e., when partitioning the dataset into training and testing sets. The randomness of many of these procedures is governed by a parameter called `random_state`, that works as follows. If `random_state` is set to an integer, the random number generator is seeded with that integer, ensuring that the same results are obtained each time the code is run. If `random_state` is set to `None` (the default value), the random number generator is initialized with a random seed, which means that the results will possibly be different each time the code is run.

3.4. General Issues

Finally, we also include some general issues that can occur in data science pipelines, related to the nature of the data or mistakes made by the developer. The eventuality of having a high dimensional dataset belongs to the first category, and it is a common issue in data science. High dimensionality is caused by the presence of a large number of features relative to a much lower number of samples in the dataset [2]. This not only makes data visualization more complex, but also leads to the curse of dimensionality, which comprises various issues caused by having too many features, ranging from an increased computational complexity to overfitting. A model that overfits accurately recognizes objects used during training, but fails to correctly characterize new, unseen objects, i.e., it is unable to generalize well. Specifically, in a high-dimensional scenario, overfitting is common since as the number of features grows, data become more sparse, making it more difficult to recognize new patterns. In other words, the number of samples required for a machine learning model to generalize well increases exponentially.

Another common issue arises from the use of `inplace` operations, which can lead to unexpected behavior and make the code difficult to understand. In-place operations modify the original data structure rather than creating a new one, therefore the return value of these operations is `None`. Nevertheless, the assignment of the return value to a variable is still possible, which can lead to confusion and unexpected behavior. Even if this is a legal assignment in Python, it is most likely not the intended behavior, and is therefore flagged as a code smell by Pyra.

4. Pyra's Overview

In this section we present our prototype analyzer Pyra, an Abstract Interpretation-based static analyzer for Jupyter notebooks. Pyra extends Lyra [43], a static analyzer originally developed for Python data science applications. Lyra supports input data usage analysis, so as to detect and report unused input data, and interval analysis, to infer the possible ranges of program variables.[1] Pyra builds upon Lyra by integrating several key features: it includes support for the analysis of non-annotated Python programs; it can handle a wider range of specific Python constructs, such as exceptions, `with` statements and `lambda` expressions; and it provides partial support for the libraries `pandas`, `numpy`, and `scikit-learn`, which are frequently used in data science applications. In the following we describe the architecture

[1] Lyra is publicly available at `https://github.com/caterinaurban/Lyra`.

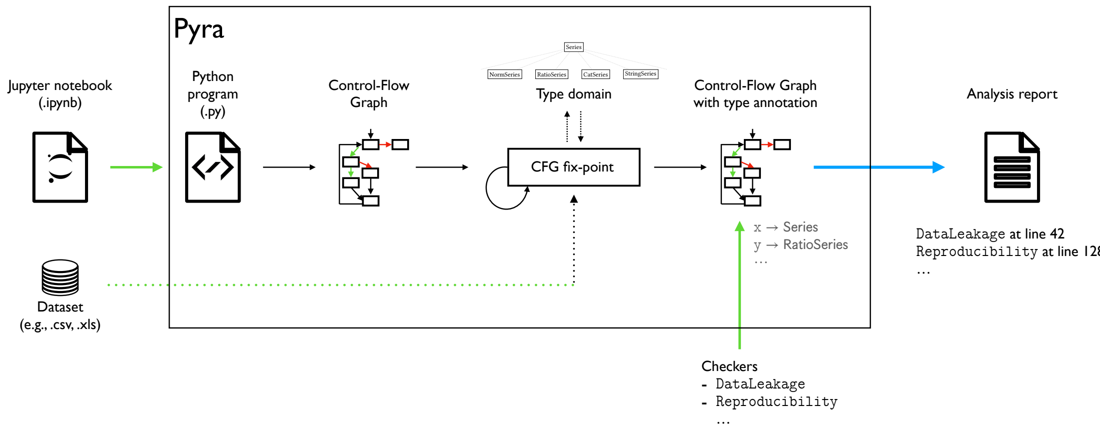

Figure 7: PYRA's overall execution.

of PYRA, the proposed type analysis and the checkers we designed to detect the code smells discussed in Section 3.

4.1. Architecture

Figure 7 provides a high-level view of the architecture of PYRA: taking as input a Jupiter notebook and the confidence of checkers to be activated, PYRA produces as output an analysis report. The pipeline first converts the notebook into a Python program; in order to do this, PYRA implicitly assumes that the code cells contained in the notebook are executed in sequential order. Next, by simply visiting the Abstract Syntax Tree (AST) of the parsed Python code (i.e., the CFG generator is a subclass of the Python `ast.NodeVisitor` class) it constructs the corresponding Control-Flow Graph (CFG), i.e., a graphical and structured representation of all the paths that may be executed by the program.

Then, for each program point and each program variable, PYRA computes the corresponding abstract type information by running an Abstract Intepretation-based static analysis: this is obtained by a generic fixpoint (over-) approximation engine, parameterized with respect to the abstract domain modeling the properties of interest; the specific abstract domain we adopted for our type analysis is described in Section 4.2. Note that, before starting this static analysis phase, it is possible to enrich the input to PYRA by optionally providing the datasets on which the Jupyter notebook operates on (see the dotted line in Figure 7); this additional information, when available, can assist the static analysis in inferring more precise types for some of the variables. As an example, consider the code fragment shown in Figure 8:

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

         df = pd.read_csv("dataset.csv")
         ...
         plt.plot(df['X'], df['Y'])
```

Figure 8: Code fragment showing dataset loading and plotting.

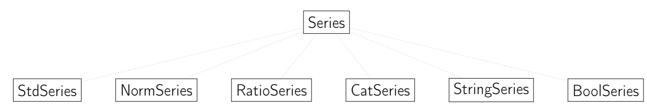

Figure 9: Diagram of the abstract domain specific to Series.

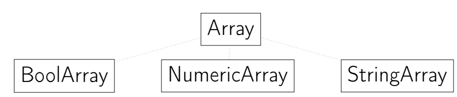

Figure 10: Diagram of the abstract domain specific to arrays.

When adopting a fully static approach, i.e., ignoring the contents of file datatset.csv, no useful type information can be derived for the data contained in df (and hence for the series indexed by X and Y). In contrast, if the user also provides as input the file dataset.csv, PYRA can infer that expression df['X'] has a specific abstract type, e.g., CategoricalSeries; this additional type information can be usefully exploited by the PYRA checkers to issue an appropriate warning when later df['X'] is used as the x-axis in plotting functions, as it happens in the last line of the example above.

When the analysis phase is concluded, its results are used to annotate the CFG with the computed type information. In the next step, PYRA enables the checkers with the confidence specified by the user on this enriched CFG, so as to detect the potential violations and issue the corresponding warnings as output; the checkers available in the current version of PYRA are described in Section 4.4.

4.2. Abstract Datatypes

Our abstract datatype domain is modeled as a finite lattice, where the partial order relation ($\sqsubseteq$) encodes the relative precision of the domain elements: intuitively, if $a \sqsubseteq b$ then abstract element b describes a larger set of possible values and hence it is less precise than abstract element a.[2] As usual, the top element $\top$ ("don't know"), which describes the set of all possible values, is the less precise one; the bottom

[2] In the diagrams smaller elements are depicted below larger ones.

element ⊥, describing an empty set of possible values, is the most precise one and encodes a definite programming error.

We now informally describe the elements of the abstract datatype domain used by PYRA. Currently, the domain contains 56 abstract datatypes.[3]

- 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`, `Tuple` (**7** abstract datatypes).
- Some special abstract datatypes for `None` are used in the abstract datatype domain, filtering weather None is directly assigned or is the result of an `inplace` operation (**2** abstract datatypes).
- Other abstract datatypes are in direct correspondence with those defined in specific data science libraries, such as `DataFrame` and `Series` for `pandas`, or `Tensor` for `torch`.
- 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, and `Scalar` is for scalar values (**2** abstract datatypes).
- 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 (**8** abstract datatypes); and *scalers*, such as `StdScaler`, `MinMaxScaler` and `MaxAbsScaler` (**12** abstract datatypes). Consistently with our previous choices, we also model *Principal Component Analysis (PCA)* (**1** abstract datatype), which is used for linear dimensionality reduction by applying a linear transformation that projects the data into a lower-dimensional space, maximizing variance.
- Some abstract datatypes are introduced to manage specific procedures, such as the division between the training and test sets, which is regularly required when developing a machine learning model (**2** abstract datatypes). These datatypes enable our analyzer to maintain a rather simple but sufficiently clear record of the provenance of the data. Similarly, additional abstract datatypes are introduced to record feature selection, often adopted to refine the data to improve performance and interpretability (**2** abstract datatypes).
- 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 9 we show the refinements available for the `Series` datatype: for instance, datatype `NormSeries` indicates that the values in the series have been subjected to normalization (**8** refined abstract datatypes for `Series`). In Figure 10 we show the refinements for the array collections; the reason why arrays happen to have fewer refinements with respect to series is that they are used less frequently in calls to the relevant data science library functions (**3** refined abstract datatypes for `Array`). We have a similar refinement also for list collections (**3** refined abstract datatypes for `List`), and dataframes (**1** refined abstract datatype for `DataFrame`).

In PYRA, currently, 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. It is also worth highlighting that, while the current implementation of PYRA supports 56 abstract datatypes, the framework is designed to be easily extensible; new datatypes can be integrated into the abstract domain by properly defining the partial order for the newly added datatypes with respect to the already available ones. New abstract datatypes may need to be introduced to support the definition of new checkers, beyond those described in the following sections.

4.3. Abstract Type Evaluation in PYRA

The static analysis computes and propagates type information by maintaining an *abstract type environment* Γ that maps each program variable x to the corresponding element $a_x = \Gamma(x)$ of the abstract datatype domain. Intuitively, newly encountered variables are added to Γ and mapped to the top element ⊤, meaning that nothing is initially known about their abstract datatype; an expression $expr$ is abstractly evaluated to obtain its corresponding datatype, looking up the type environment Γ when evaluating each of the variables occurring in the expression and combining the types of subexpressions using type rules such as

$$\text{Series} \;/\; \text{Series} = \text{RatioSeries},$$

whose intuitive reading is that the division operator, when applied to two expressions having both abstract datatype **Series**, yields a result having abstract datatype **RatioSeries**; when evaluating an assignment statement such as $x = expr$, we first compute the abstract datatype a_{expr} for the right-hand side expression (using Γ) and then update the type environment to $\Gamma[x \mapsto a_{expr}]$, recording that variable x is now mapped to datatype a_{expr}. As an example, given the code fragment reported in Figure 11, PYRA produces the CFG annotated with the abstract type information shown in Figure 12; the final nodes of the CFG contain the final type information about each variable.

When joining two or more control flows, the corresponding type environments are merged by applying the

[3] The full list of the PYRA's abstract datatypes is available at https://github.com/spangea/Pyra/blob/datascience/src/lyra/datascience/datascience_type_domain.py.

```
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']
```

Figure 11: 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.

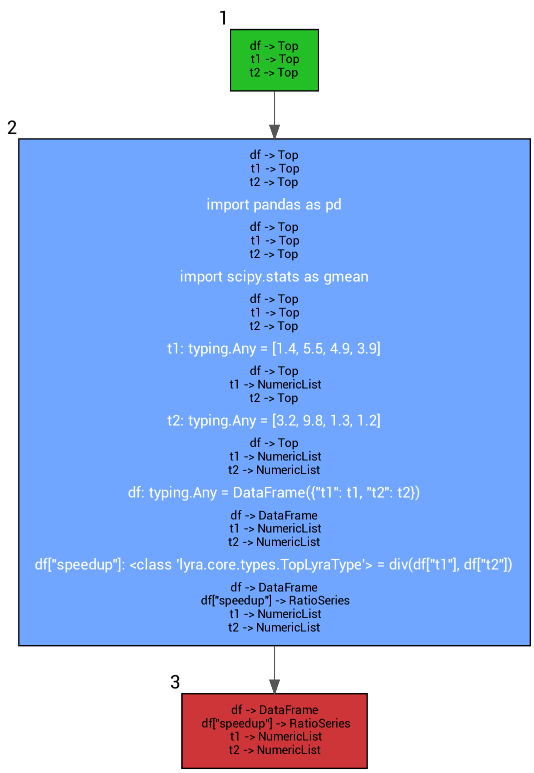

Figure 12: PYRA's abstract type analysis for Jupyter fragment reported in Figure 11.

abstract datatype join (i.e., least upper bound) operator to each variable binding; for instance, if $\Gamma_1(x) = \text{RatioSeries}$ and $\Gamma_2(x) = \text{StdSeries}$ then, after joining Γ_1 and Γ_2 into Γ, we obtain $\Gamma(x) = \text{Series}$.

Concrete Dataset Information. As mentioned before, it is possible to provide PYRA with the external datasets accessed and used by the Jupyter notebook. Even though not strictly necessary, this is useful to improve the precision of the analysis as it allows to compute and propagate more precise datatypes for the content of the datasets.

Algorithm 1 shows the pseudo-code of the procedure implemented in PYRA to extract abstract datatype information from the concrete dataset. The algorithm takes as input the type environment (Γ), the name of the function being called (call) and the the path to the dataset (path). If the call corresponds to `read_csv` (line 2), PYRA reads the CSV into a DataFrame using the `pandas` function (line 3) and performs several checks: whether the DataFrame is small (lines 4–6), high-dimensional (lines 7-9), contains duplicates (lines 10–12), or has missing values (lines 13–15), the information about these attributes is then saved (assignments at lines 5, 8, 11, 14, and 27 are kept during the analysis as further concrete information linked to the DataFrame) along with the abstract datatype information for the dataset in the abstract state. The values adopted for these checks are customizable and given by empirical evaluation of real-world datasets in different contexts. The algorithm also determines the datatypes of each column (lines 18–22) and assigns them to their corresponding abstract datatypes in Γ (lines 19 and 22). Finally, it checks if the DataFrame is shuffled based on the sorting information of its columns (lines 25–29). Note that some procedures like HASDUPLICATES (line 10) and HASNA (line 13) are omitted for brevity, but they correspond to simple checks easily implemented using the `pandas` library.

It is worth highlighting that providing PYRA with the dataset is not mandatory. Even if the dataset is not provided, PYRA can still analyze the code and issue warnings based on the abstract datatypes statically inferred from the code itself. Finally, independently of the dataset being provided or not, the abstract datatype for the variable related to the dataset (the left hand side of the assignment in which the right hand side is the call to `read_csv`) will always be set to `DataFrame`.

While these checks are not strictly required for the analysis to proceed, they help improve the precision and provide more information to the user about the contents of the dataset, which would otherwise remain statically unknown.

4.4. PYRA Checkers

The results of the abstract type analysis are used by the checkers to identify the potential errors and code smells described in Section 3; in the following, we describe how PYRA leverages this analysis to detect them.

Warning Interpretation. In PYRA, warnings are categorized as either *plausible* or *potential* depending on the confidence of the static analysis. A *plausible* warning is emitted when the analysis has sufficient evidence to indicate that a code smell or issue is likely to occur. In contrast, a *potential* warning is issued when the analysis cannot fully determine the nature of the data or operations involved, but there are indications that a problematic pattern might be present. This distinction allows the tool to provide useful and tailored feedback, according to the desired level of confidence that can be set by the user when running PYRA.

`CategoricalConversionMean`, `GMean`, `ScaledMean`. Algorithm 2 reports the pseudo-code of the PYRA checker for identifying `CategoricalConversionMean`, `GMean`, and `ScalerMean` code smells. The checker takes as input the type environment Γ that occurs before the execution of the Python call. If the call corresponds to `mean`, the caller cl is extracted (lines 2–3). Then, the abstract datatype of cl is retrieved from Γ and analyzed to generate potential warnings. Specifically, if the abstract datatype is a Series datatype (line 4), then

Algorithm 1 Pseudo-code of the algorithm that analyzes the concrete dataset information and maps it to the abstract datatypes.

```
1:  function CONCRETE INFO(Γ, call, path)
2:      if call = read_csv then
3:          df ← pd.read_csv(path)
4:          if LEN(df.rows) ≤ 100 then
5:              isSmall ← True
6:          end if
7:          if LEN(df.rows) < 2 * LEN(df.columns) then
8:              isHighDim ← True
9:          end if
10:         if HASDUPLICATES(df) then
11:             hasDuplicates ← True
12:         end if
13:         if HASNA(df) then
14:             hasNa ← True
15:         end if
16:         sortingInfo ← ∅
17:         for col ∈ df.columns do
18:             if col.dtype ∈ {int, float} then
19:                 Γ(col) ← NumericSeries
20:                 sortingInfo[col] ← GETSORTINGINFO(col)
21:             else if col.dtype = object then
22:                 Γ(col) ← CatSeries
23:             end if
24:         end for
25:         isShuffled ← True
26:         for col ∈ sortingInfo do
27:             if sortingInfo[col] ∈ {increasing, decreasing} then
28:                 isShuffled ← False
29:                 break
30:             end if
31:         end for
32:     end if
33: end function
```

Algorithm 2 Pseudo-code of the mean's warning-related checker.

```
1:  function CHECKER(Γ, call)
2:      if call = mean then
3:          cl ← GETCALLER(call)
4:          if Γ(cl) ⊊ Series then
5:              if Γ(cl) = RatioSeries then
6:                  GMEANWARN(call, plausible)
7:              else if Γ(cl) = CatSeries then
8:                  CATCONVMEANWARN(call, plausible)
9:              else if Γ(cl) = ScaledSeries then
10:                 SCALEDMEANWARN(call, plausible)
11:             end if
12:         else if Γ(cl) ∈ {Series, ⊤} then
13:             GMEANWARN(call, potential)
14:             CATCONVMEANWARN(call, potential)
15:             SCALEDMEANWARN(call, potential)
16:         end if
17:     end if
18: end function
```

Algorithm 3 Pseudo-code of the CategoricalPlot checker.

```
1:  function CHECKER(Γ, call)
2:      if call = plot ∧ GETKIND(call) ∉ {bar, barh} then
3:          for ax ∈ ARGS(call) do
4:              if Γ(ax) ∈ {StringList, StringArray, StringSeries} then
5:                  CATPLOTWARN(call, plausible)
6:              else if Γ(ax) = CatSeries then
7:                  CATPLOTWARN(call, plausible)
8:              else if Γ(ax) ∈ {Array, Series, ⊤} ∧ Γ(ax) ∉ {NumericSeries, NumericArray} then
9:                  CATPLOTWARN(call, potential)
10:             end if
11:         end for
12:     end if
13: end function
```

the checker verifies whether cl is a RatioSeries, CatSeries, or ScaledSeries. If so, a plausible related warning is issued on that call (lines 5-9). Otherwise, the static analysis does not have enough information to determine the exact Series's subtype of cl, so three potential warnings are issued (lines 12–16). Except for these cases, no warnings are raised.

Similarly, concerning `CategoricalConversionMean`, we apply the same checker when inspecting the `median` call.

`CategoricalPlot`. When a Jupyter notebook plots something whose one of the axes is nominal-scale data, PYRA uses Algorithm 3 to issue a warning.

When PYRA encounters a plot call which is not a bar plot, it iterates through the axis arguments (line 3) and inspects their abstract datatypes by querying Γ; if the abstract datatype corresponds to `StringList`, `StringArray`, `StringSeries`, or `CatSeries`, a plausible warning is issued for the respective axis (lines 4–7). Otherwise, if Γ identifies the abstract datatype as either an `Array`, `Series`, or the top element (⊤), PYRA issues a potential warning.

`DataLeakage`. This checker is designed to identify potential data leakage issues. As previously explained, data leakage occurs when information from the test set is inadvertently used during the training phase, leading to overly optimistic performance estimates. The checker analyzes the abstract datatypes of the arguments involved in specific function calls and raises warnings if it detects potential data leakage.

Specifically the checker is activated when the functions `train_test_split`, `fit`, and `fit_transform` are called. The checker inspects the arguments of these function calls and checks for specific conditions that may indicate data leakage. The conditions checked by Algorithm 4 are the following:

- If the function call is `train_test_split` (lines 2–7), it checks if any of the arguments are of type `NormSeries`, StdSeries, or CatSeries, or if they are coming from a scaling or feature selection process (line 4). In this case, since the splitting into training and testing data sets is performed after the pre-processing, some training information may have leaked into the test set, therefore a warning is issued (line 5).

- If the function call is `fit` or `fit_transform` (lines 8–14), it checks if the fitting method is called on a test set (line 12) coming from a previous splitting operation. In this case, the warning is raised (line 11).

Algorithm 4 Pseudo-code of the `DataLeakage`'s checker.

```
1:  function CHECKER(Γ, call)
2:      if call = train_test_split then
3:          for ax ∈ ARGS(call) do
4:              if Γ(ax) ∈ {NormSeries, StdSeries, CatSeries} ∨ IS_SCALED(ax) ∨ IS_FEATURE_SELECTED(ax) then
5:                  DATALEAKAGEWARN(call, plausible)
6:              end if
7:          end for
8:      else if call ∈ {fit, fit_transform} then
9:          for ax ∈ ARGS(call) do
10:             if IS_SPLITTED_TEST_DATA(ax) then
11:                 DATALEAKAGEWARN(call, potential)
12:             end if
13:         end for
14:     end if
15: end function
```

`DuplicatesNotDropped`. The checker inspecting for this warning is syntactic, thus it does not rely on the abstract datatype analysis described in Section 4.2. Specifically, during the abstract datatype computation, PYRA tracks whether the `drop_duplicates` method has been called on each DataFrame occurring in the Jupyter notebook. It is important to note that, this warning is always issued as *possible* warning. This is because a dataset may have been pre-processed to remove duplicates outside the notebook, without explicitly invoking methods such as `drop_duplicates` within the notebook source code, or because duplicates in some contexts may be relevant for representing the true data distribution. Consequently, when the `DuplicatesNotDropped` warning is raised, it should be interpreted as a suggestion rather than an actual error in the notebook.

`FixedNComponentsPCA`. The syntactic checker actives when a `PCA` is created. Specifically, PYRA raises a warning if the `n_components` parameter of `PCA` is assigned to a constant value, as shown in Figure 13.

As reported in Table 1, this warning should be interpreted as a *code suggestion*. In particular, if domain knowledge or prior experiments on the dataset, outside the analyzed notebook, suggest that a specific number of principal components captures enough variance, setting `n_components`

```
In [1]:
// FixedNComponentsPCA warning
pca = PCA(n_components=3)
df_reduced = pca.fit_transform(df)
```

Figure 13: Example of fixed number of components in PCA.

may be justified. However, for improved adaptability across different datasets, dynamically determining `n_components`, such as by retaining a target percentage of explained variance, can be a more flexible approach.

`HighDimensionality`. The high-dimensionality checker can be activated only if the user provides PYRA with the datasets, allowing PYRA to extract relevant information about the dataset applied in Algorithm 1. If the algorithm detects high dimensionality, it raises a warning, suggesting that feature selection, feature engineering, or dimensionality reduction may be necessary for that dataset. Note that there is no strict, formal definition of a high-dimensional dataset: generally, they are loosely defined as those datasets having far more features than samples [2]. In practice, the high-dimensionality concept is both context- and technique-dependent; e.g., consider the omics field, where differential expression analyses exploit all available features [29]. Hence, in PYRA we adopt a rule of thumb whereby a dataset is considered high-dimensional when the number of features is at least twice the number of objects. This can be seen as a compromise that avoids raising too many warnings that are false positives; we are aware that this threshold might be too lax in some more classical contexts (e.g., when using a linear regression model).

`InappropriateMissingValues`. PYRA may issue this warning when the code uses the `fillna` method to replace missing values in a DataFrame with summary statistics (e.g., mean or median). This issue becomes more concerning when the DataFrame is small, as it can lead to misleading results. In such cases, PYRA raises a potential warning.

`InconsistentType`. Python allows functions and variables to be annotated with types, even though these annotations are not enforced at runtime. However, if a variable is annotated with a type, but PYRA infers an incompatible type, the annotation is considered incorrect, and PYRA issues a warning. Specifically, let x be a variable and T_x its user-defined type annotation. PYRA raises a warning if $T_x \sqcap \Gamma(x) = \bot$. However, no warning is issued if the inferred type is compatible with the annotation. For example, as shown in Figure 14:

```
In [1]:
x : list = [1, 2, 3, 4]
```

Figure 14: Example of type annotation compatibility.

Here, PYRA infers the type of `x` as `NumericList`, which is compatible with the annotated type `list`, so no warning is generated.

`MissingData`. Similar to the high-dimensionality warning checker, the missing data warning checker can be enabled if the user provides PYRA with the datasets used. This allows PYRA to inspect the dataset and detect any missing values (e.g., `NaN`). If no `dropna` method is applied to the corresponding DataFrame containing the dataset's information, a warning is raised at the end of PYRA's execution.

`NoneRetAssignment`. Given an assignment of the form `lhs = rhs`, if the abstract datatype static analysis infers that Γ(`rhs`) is `None`, PYRA raises a warning for the assignment. While this operation does not inherently indicate an error or a code smell, it may suggest a misunderstanding of the functions or methods used in `rhs`. For example, let us consider the following statement.

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

The `fillna` method does not return a Series when the `inplace=True` parameter is specified. As a result, assigning its output to the variable `result` is likely unintended and could lead to unexpected behavior in subsequent code.

`NotShuffled`. Similar to the `DuplicatesNotDropped` warning, the checker for `NotShuffled` is purely syntactic and does not rely on abstract datatype analysis. During the abstract datatype computation, PYRA tracks whether the `sample` method has been called on each DataFrame in the Jupyter notebook. As with the `DuplicatesNotDropped` warning, this warning is always issued as a *possible* warning and should be interpreted as a suggestion rather than an error. This is because the dataset may have already been shuffled outside the notebook or might be inherently random.

`PCAOnCategorical`. Algorithm 5 checks whether PCA is applied to categorical data. When PYRA encounters a call to `transform`, `fit`, or `fit_transform` (line 2), it retrieves the caller (line 3) and checks whether it is a PCA object (line 4). If so, it retrieves the first argument of the call (line 5) and checks whether it is a DataFrame (line 6). If the argument is a DataFrame, the algorithm iterates through its subscripts (line 7) (i.e. the Series belonging to it) and checks whether any of them are categorical series (line 8). If so, a plausible warning is issued (lines 9). Otherwise, if the analysis has not raised a warning and has not enough information to determine the type of the subscripts (lines 13-16), a potential warning is issued (line 17).

`PCAVisualization`. As mentioned before, using the results of a PCA to visualize the data is a common practice. However, this is not always the best choice, as shown in Figure2. In case this happens, our analyzer issues a warning following the pseudo-code described in Algorithm 6. If the called method is `plot` or `scatter`, the analyzer iterates through the arguments of the call (line 3) and if the argument has abstract datatype `DataFrameFromPCA` (line 4), meaning that is a DataFrame resulting from the application of a PCA, then a plausible warning issued.

Algorithm 5 Pseudo-code of the `PCAOnCategorical` checker.

```
 1: function CHECKER(Γ, call)
 2:     if call ∈ {transform, fit, fit_transform} then
 3:         cl ← GETCALLER(call)
 4:         if Γ(cl) ⊑ PCA then
 5:             arg = GETFIRSTARG(call)
 6:             if Γ(arg) ⊑ DataFrame then
 7:                 for s ∈ SUBSCRIPTS(arg) do
 8:                     if Γ(s) = CatSeries then
 9:                         PCAONCATWARN(call, plausible)
10:                         warning_raised ← True
11:                     end if
12:                 end for
13:                 if ¬ warning_raised then
14:                     no_warning ← True
15:                 end if
16:                 if ¬ warning_raised ∧ ¬ no_warning then
17:                     PCAONCATWARN(call, potential)
18:                 end if
19:             end if
20:         end if
21:     end if
22: end function
```

Algorithm 6 Pseudo-code of the `PCAVisualization` checker.

```
1: function CHECKER(Γ, call)
2:     if call ∈ { plot, scatter} then
3:         for ax ∈ ARGS(call) do
4:             if Γ(ax) = DataFrameFromPCA then
5:                 PCAVISWARN(call, plausible)
6:             end if
7:         end for
8:     end if
9: end function
```

`Reproducibility`. If the `random_state` parameter is not explicitly set when calling a method that allows for its setting, such as the `sample` or `train_test_split` methods, PYRA raises a reproducibility warning for the call.

4.5. Running PYRA

In this section, we provide a running example to illustrate how PYRA works. The example is a simple code that reads a dataset from a CSV file, splits it into training and test sets, and trains a KNeighborsClassifier model. The code is shown in Figure 15.

We can run PYRA on the notebook using the command:

```
pyra -analysis type-datascience code_to_analyze.py,
```

In [1]:
```python
import pandas as pd
from sklearn StandardScaler, train_test_split
KNeighborsClassifier, accuracy_score

df = pd.read_csv("data.csv")
# df.dropna(inplace=True)
# df.drop_duplicates(inplace=True)
# df = df.sample(frac=1, random_state=42)

X = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

In [2]:
```python
sc = StandardScaler()
X_sc = sc.fit_transform(X)

X_tr, X_te, y_tr, y_te =
    train_test_split(X_sc, y, test_size=0.2)
```

In [3]:
```python
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_tr, y_tr)
y_pred = knn.predict(X_te)
acc = accuracy_score(y_te, y_pred)
```

Figure 15: A code snippet containing different issues. Imports are shortened to fit the page and only refer to the library offering them, without the proper module.

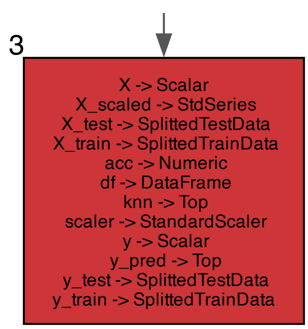

Figure 16: PYRA's results for the abstract type analysis of the code shown in 15 when the dataset is not provided.

this instructs the analyzer to perform a forward analysis on the code, keeping track of the abstract datatypes and issuing both plausible and potential warnings.

The output of the analysis may vary depending on whether or not the user provides the dataset used in the code.

Without Dataset Information. The result of the analysis for this scenario is shown in Figure 16. In this case, PYRA is able to infer the abstract datatypes of all the variables except `KNeighborsClassifier` and `y_pred` because their rules (i.e., the call to the constructor of `KNeighborsClassifier` and the call to the `predict` method) are not implemented in the current version of PYRA since they are not related to specific issues: for this reason their abstract datatypes are set to $\top$.

Nevertheless, the analyzer is able to capture some issues and raise warnings, as shown in Figure 17. The first warning is a reproducibility issue related to the call to the `train_test_split` method without the `random_state` parameter set and it is captured with a syntactic check. This warning

Reproducibility Warning

Warning [plausible]: in `train_test_split(X_sc, y, test_size=0.2)` @ line 16 the random state is not set, the experiment might not be reproducible.

Data Leakage Warning

Warning [plausible]: in `train_test_split(X_sc, y, test_size=0.2)` @ line 16 data should be standardized after the split method.

Figure 17: Warnings raised during the analysis of the code shown in Fig 15 when the dataset is not provided.

can be fixed by setting the `random_state` parameter to a fixed value (for example, `random_state=42`) in the arguments of the call, which is useful for reproducibility purposes. The second warning is related to a data leakage issue, which is captured by the `DataLeakage` checker (Algorithm 4). For this warning, the correct fix is similar to the one shown in Figure 4.

With Dataset Information. The results of the analysis when the dataset (shown in Table 2 and Figure 2) is provided are shown in Figure 18. In this case, PYRA is able to infer the abstract datatypes of all the previously detected variables that were analyzed (keeping the exception of `KNeighborsClassifier` and `y_pred`). Additionally, using the concrete analysis shown in Algorithm 1, the analyzer is able to infer the abstract datatypes of the columns of the dataset, which were not previously known, as shown in Table 2.

Moreover, based on this information and the other attributes inferred by the Algorithm 1, the analyzer is able to raise different warnings from the ones raised in the previous case, as shown in Figure 19. The issues regarding reproducibility and data leakage are still present because they are not linked to the concrete information of the dataset. Using the information retrieved from the concrete dataset the analyzer is able to raise three new warnings. The first one is related to the presence of missing values in the dataset, and it is raised because the analyzer is able to infer that the concrete dataset contains some missing values (i.e., `NaN` values) and that no method has been called to drop them. The solution for this issue is to call the `dropna` method on the DataFrame before splitting it into training and test sets, as shown in the commented code in the snippet. The second one is related to the presence of duplicates in the dataset, which is raised because the analyzer is able to infer that the concrete dataset contains some duplicates (i.e., two rows with the same values) and that no method has been called to drop them. The solution for this issue is to call the `drop_duplicates` method on the DataFrame before splitting it into training and test sets, as shown in the commented code in the snippet. Finally, the analyzer is also able to infer that the dataset is not shuffled because the first column of the dataset is sorted in increasing order. For this reason, the analyzer raises a warning suggesting to shuffle the dataset. As for the previous

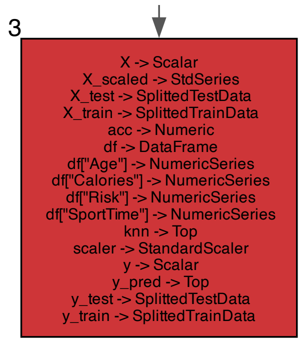

Figure 18: PYRA's results for the abstract type analysis of the code shown in 15 when the dataset is provided.

Reproducibility Warning

Warning [plausible]: in `train_test_split(X_sc, y, test_size=0.2)` @ line 16 the random state is not set, the experiment might not be reproducible.

Data Leakage Warning

Warning [plausible]: in `train_test_split(X_sc, y, test_size=0.2)` @ line 16 data should be standardized after the split method.

Missing Data Warning

Warning [potential]: At the end of the program df might still have NA values, using `dropna()` might be necessary.

Duplicates Not Dropped Warning

Warning [potential]: At the end of the program df might be small and still have duplicates that were not dropped, using `drop_duplicates()` might be necessary.

Not Shuffled Warning

Warning [potential]: At the end of the program df might be not shuffled, using `sample()` might be necessary to guarantee randomness.

Figure 19: Warnings raised during the analysis of the code shown in Fig 15 when the dataset is provided.

case, the solution for this issue is to call the `sample` method on the DataFrame before splitting it into training and test sets, as shown in the commented code in the snippet.

Age	Calories	SportTime	Risk
22	2200	4	1
28	**2100**	**NaN**	**1**
30	2500	5	1
33	*2400*	*4*	*1*
33	*2400*	*4*	*1*
35	2300	2	2
40	2600	2	2
45	**NaN**	**3**	**2**
50	2900	1	3
55	3000	0	3
60	2800	1	3

Table 2
Table representation of the dataset used in the running example reported in Figure 15. The rows in bold are the ones containing missing values, while the rows in italic are duplicated.

	loc	vars	calls
Minimum	21	1	6
Median	90.00	12.00	56.00
Maximum	2872	193	2123
Mean	126.84	16.45	79.58
Standard Deviation	127.33	14.71	83.32
Total	554919	71976	348181

Table 3
Statistics of all the collected notebooks.

5. Experimental Evaluation

5.1. Benchmark suite description and experimental setup

For our experimental evaluation, we created a benchmark by randomly collecting 9259 Jupyter notebooks published in Kaggle[4] and related to popular competitions (e.g., Mayo Clinic - STRIP AI[5]) or popular datasets (e.g., Pima Indians Diabetes Database[6]).

Some information about the collected notebooks is reported in Table 3. The table reports the minimum, median, maximum, mean and standard deviation of: the number of lines of code ('loc'); the number of variables ('vars'); and the number of function calls ('calls') contained in the notebooks.

Starting from this first collection, we filtered the notebooks to exclude those containing features that our analyzer is not designed to handle, e.g., object-oriented constructs such as class or function definitions. This is ensured by simply checking that the Abstract Syntax Tree of the notebook code does not contains any `ast.ClassDef`, `ast.FunctionDef`, and `ast.AsyincFunctionDef` nodes.

Moreover, we kept only notebooks containing at least a variable, since our analyzer specifically annotates program variables, and having more than 20 lines of code (empty lines and comments are not counted), to avoid analyzing files that are too short, such as basic Kaggle templates. This criterion

[4] https://www.kaggle.com/
[5] https://www.kaggle.com/competitions/mayo-clinic-strip-ai
[6] https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

	loc	vars	calls
Minimum	21	1	6
Median	70.00	9.00	43.00
Maximum	1307	140	928
Mean	93.33	11.83	58.91
Standard Deviation	77.18	9.59	52.83
Total	204208	25875	128894

Table 4
Statistics of the filtered benchmark.

and these observations are meant to increase the probability that the code we analyze is somehow meaningful.

After the filtering operation, the resulting number of notebooks is 4375, and this is the benchmark on which our experimental evaluation is run. The content of these notebooks is diverse: some focus on exploratory data analysis (EDA), others build machine learning models for classification or regression tasks, while others generate visualizations or analyze patterns, and so on. The statistics on the filtered benchmark are reported in Table 4.

All the experiments were run on a 2021 MacBook Pro (model MacBookPro18,3) with M1 Pro (10 cores) and 16 GB of RAM, demonstrating how Pyra can be run on a standard laptop without requiring any special hardware or software setup. We provided the 85 projects that we used to build the benchmark, for a total of 66.76 GB of zipped data.

The processing of the entire benchmark took about 110 minutes, with an average of 2.89 seconds per notebook (minimum 1.90, maximum 106.04 seconds). This includes the time needed to analyze the notebook, as well as the time needed to unzip the folder containing the dataset and load the concrete dataset, which can be quite time consuming.

5.2. Qualitative Evaluation

Pyra correctly and automatically analyzes 2286 (i.e., approximately 52%) of the programs contained in the benchmark. Although this success rate may appear limited, the failures primarily arise from the intrinsic flexibility and permissiveness of Python. These features introduce challenges for static analysis tools, especially when handling highly dynamic constructs. In particular, Pyra currently supports a large subset of the core language (e.g., conditional statements, loops, exception handling), but it cannot yet handle more intricate operations such as complex indexing in `pandas`, advanced slicing mechanisms, or comprehension constructs involving nested or dynamic expressions, which result in exceptions. Nevertheless, it is important to highlight that this limitation does not compromise the validity of the proposed type analysis, being instead related to the current prototype implementation, which still lacks support for some advanced Python features. Further work can progressively extend this coverage and improve the robustness of Pyra, without requiring changes to the underlying analysis.

The total number of raised warnings is 4214; it is worth noting that, even though this is a randomly collected benchmark, 15 of the 16 warnings that we defined were raised by the analyzer. These warnings were found in 1661 notebooks, while 625 notebooks were analyzed without raising any warning. In detail, 50 notebooks presented warnings in 3 out of 4 categories, while 451 had warnings in 2 of them. The only warning that was never raised for our benchmark is `InconsistentType`, only raised when the user annotates the type of a variable and the inferred type does not match the user-annotated one. Note that, type annotation is not a common practice in data science and its requirement is usually considered a constraint in the existing tools.

Figure 20 shows the distribution of warnings by name and confidence. The most common warning was the `Reproducibility` warning, which was raised 2019 times with plausible confidence, highlighting a significant concern regarding the deterministic nature of data science workflows in the analyzed notebooks. Another of the most common warning was `CategoricalPlot` warning with a total of 1662 occurrences (89 plausible, 1573 potential), indicating many notebooks potentially misusing categorical data in plots. Related to the misleading visualization issue, our analysis also raised 6 plausible `PCAVisualization` warnings, suggesting that some notebooks may not be using PCA visualizations correctly. Another prevalent issue was the `NotShuffled` warning with 780 potential occurrences, suggesting that many data scientists may not be properly randomizing their datasets.

The `MissingData` warning was detected 547 times with potential confidence, indicating notebooks that might have issues with missing data handling. Similarly, `CategoricalConversionMean` warning (226 occurrences) and `ScaledMean` warning (211 occurrences) were frequently detected, both related to possibly improper results in statistical operations. The `Gmean` warning appeared 211 times with potential confidence.

General data quality issues were also prominent, with `DuplicatesNotDropped` warning (133 occurrences) and `InappropriateMissingValues` warning (134 occurrences) suggesting that many notebooks may not properly handle data preprocessing steps. More critical issues like `DataLeakage` warning were detected 141 times (95 plausible, 46 potential), and it is worth noting that this issue could directly impact the performance of machine learning models.

Less frequent but still significant warnings included `HighDimensionality` warning (43 occurrences), `PCAOnCategorical` warning (13 occurrences), and `FixedNComponentsPCA` warning (20 occurrences: 17 plausible, 3 potential), all related to dimensionality or dimensionality reduction techniques. Two occurrences of the `NoneRetAssignment` warning were also detected.

The wide variety and high frequency of warnings demonstrate the utility of Pyra in automatically detecting potential issues in data science code that might otherwise go unnoticed. The distinction between potential and plausible warnings also provides users with information about the confidence level of the detected issues.

It is important to emphasize that warnings with "potential" confidence can be disabled if the user wants an analysis that raises less warnings. A typical use case might be when

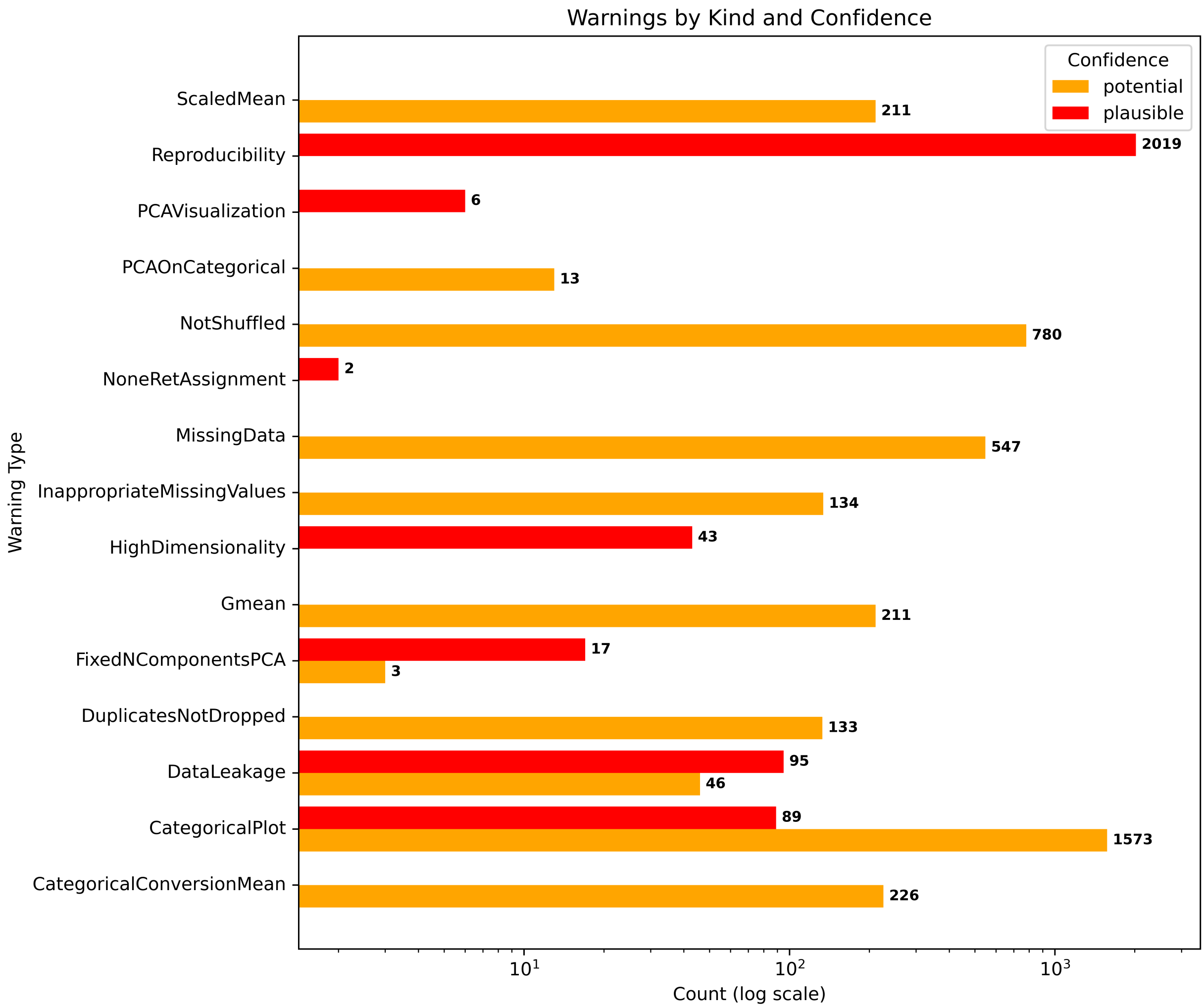

Figure 20: Warning raised in the experimental evaluation grouped by kind and confidence.

the user knows that certain checks are unnecessary in specific notebooks, for example because data quality has already been verified in an earlier phase of the analysis or because some operations were intentionally performed in a certain way for specific purposes related to prior knowledge of the data. Moreover, we want to emphasize that these warnings are not meant to be final sentences, but rather suggestions for the user to consider and incentivate critical thinking about the code they are writing. In fact, sometimes these warnings need to be contextualized. For example, for the `GMean` warning it is important to take into consideration the distribution and scale of the data, since for logarithmic data the arithmetic mean might be a more appropriate choice.

5.2.1. Real-world Code Smells Detected by PYRA

In this section, we show and discuss some examples of code fragments from three different notebooks contained in the selected benchmark suite that have raised plausible warnings, thus demonstrating the effectiveness of PYRA in identifying real-world data science code smells. The first one we analyze is notebook `sales-eda`, in which supermarket sales data are analyzed: first several exploratory plots are generated and then a Decision Tree classifier is used to predict customer ratings on a 1-10 scale. In Figure 21 we

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
train = pd.read_csv('supermarket_sales.csv')
sns.set_theme()
plt.scatter(x = 'Branch', y = 'City',
data = train)
```

In [2]:
```
from sklearn import train_test_split
X = train_dummy.drop('Rating', axis = 1)
y = train_dummy['Rating']
X_train, X_test, y_train, y_test =
        train_test_split(X, y, test_size=0.30)
```

Figure 21: Example from a real notebook showing misuse of a scatter plot and reproducibility issues. Some import and names have been shortened for better readability.

In [1]:
```
import pandas as pd
from sklearn import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import (
MinMaxScaler, StandardScaler)
df = pd.read_csv("glass.csv")
X=df.iloc[:,:-1]
y=df.iloc[:,-1]
```

In [2]:
```
minmax = MinMaxScaler()
x_minscaled = minmax.fit_transform(X)
x_minscaled
sn = []
score = []
model = DecisionTreeClassifier()
for i in range(1,101):
  X_train,X_test,y_train,y_test =
    train_test_split(X,y,stratify=y,test_size=.25)
  model.fit(X_train,y_train)
  sn.append(i)
  score.append(model.score(X_test,y_test))
```

Figure 22: Example from a real notebook showing reproducibility and data leakage issues. Some import and names have been shortened for better readability.

In [1]:
```
import pandas as pd
df_all = pd.read_csv('c19_data.csv')
df_confirmed = pd.read_csv('c19_confirmed.csv')
df_recovered = pd.read_csv('c19_recovered.csv')
df_all['datetime']=df_all['ObservationDate']
df_all['datetime']=df_all['datetime'].apply(
lambda x:datetime.strptime(str(x),'\%m/\%d/\%Y'))
df_all['month']=df_all['datetime'].apply(
lambda x:x.month)
df_all['day']=df_all['datetime'].apply(
lambda x:x.day)
df_all['year']=df_all['datetime'].apply(
lambda x:x.year)
df_all['week']=df_all['datetime'].apply(
lambda x:x.week)
df_all['state']=df_all['Province/State']
df_all['country']=df_all['Country/Region']
df_all.drop(columns=
['ObservationDate','Province/State',
'Country/Region'], inplace=True)
df_all.sample(5)
```

Figure 23: Example from a real notebook showing reproducibility issues. Some import and names have been shortened for better readability.

report two snippets of the notebook, that raise 6 warnings, 4 of which are considered plausible. In detail, in the first snippet, after loading data manipulation and plotting packages, a `DataFrame` is created, followed by a single call to the `scatter` function from the `matplotlib` package. The function is applied to two categorical variables, *Branch* and *City*, making the scatter plot unsuitable: three warnings of the categorical plot type are raised. In the second snippet, after loading the necessary packages, the predictor variables `X` are defined as all columns except `Rating`, which is used as the target variable `y`. Then, the last line splits the original data into training and testing sets. However, the `train_test_split` function is called without setting a random seed, i.e., different runs can produce different partitions, thus producing a reproducibility issue warning.

The second notebook is `classif-using-diff-scaling`, which classifies different glass types using a Decision Tree model. In detail, it compares the performance of the classifier when using no standardization, z-score standardization, and min–max normalization. Figure 22 presents the portion of the code corresponding to the classification pipeline when employing the min–max normalization procedure. In the first code snippet, after importing the required libraries, the dataset is loaded into a `DataFrame` and divided into X, which contains the predictor variables, and the target variable y. The second snippet applies the min–max normalization to X and subsequently executes a loop in which the dataset is split into training and test sets, a `DecisionTreeClassifier` model is fit, and the corresponding accuracy is stored. Equivalent code blocks are executed for the untransformed and z-score–standardized data. Across the entire notebook,eight warnings are raised, 6 of which are classified as plausible. Two of these warnings are related to data leakage: data are normalized before being split into train and test partitions. The remaining four warnings relate to reproducibility issues caused by the random state not being set. Three of these arise from the use of the `train_test_split` function, analogously to the previous notebook, while the last one is caused by the initialization of the `DecisionTreeClassifier`.

The last notebook we consider is `covid-19-data-analysis--and-visualization.py` which presents an exploratory analysis on Covid19 data. As shown in Figure 23, it loads three CSV files into separate `DataFrame` objects, converts date variables into an appropriate datetime format, and extracts different date granularities, e.g., month. It also implicitly renames some columns by creating new ones and then dropping the originals. Lastly, this snippet displays the first five rows of the resulting dataset. This code actually presents 11 warnings, 3 of them plausible. Although, as mentioned, the notebook's primary goal is exploratory, the datasets it relies on suffer from several issues, e.g., missing data, which could affect further analyses. Specifically, among the plausible warnings, two relate to high dimensional datasets: `df_recovered` and `df_confirmed` are variants of the John Hopkins University CSSE COVID-19 datasets, which originally have 468 features but only 261 and 276 samples, respectively. Apart from a few location-related features, the remaining ones represent time points: comparing cities using temporal data would lead to curse of dimensionality issues. The remaining plausible issue, involves the use of the `sample` function without a random seed. However, in this case, the function is used just to inspect the dataset and show the newly generated fields.

5.3. Quantitative Evaluation

To evaluate the effectiveness of PYRA, we also randomly selected 100 notebooks from the files that PYRA correctly analyzed and manually assessed the ground truth for each

Warning Type	Count
CategoricalPlot	6
PCAVisualization	1
CategoricalConversionMean	0
DataLeakage	16
DuplicatesNotDropped	7
FixedNComponentsPCA	2
Gmean	0
InappropriateMissingValues	7
MissingData	13
NotShuffled	16
PCAOnCategorical	0
ScaledMean	0
Reproducibility	116
HighDimensionality	0
InconsistentType	0
NoneRetAssignment	0
Global Statistics	
Total number of warnings	184
Number of analyzed files	100
Files with warnings > 0	66
Files without warnings	34

Table 5
Summary of warnings and global analysis statistics.

file by checking the presence or absence of the issues corresponding to each warning type and cross-checking the results with all the authors. This manual assessment resulted in a total of 184 warnings across the 100 notebooks, as summarized in Table 5. The table also provides a breakdown of the number of warnings per type, along with global statistics such as the total number of warnings, the number of analyzed files, and the number of files with and without warnings. As for the qualitative analysis, also in the manual assessment, the Reproducibility warning is the most frequent one, with 116 occurrences, followed by DataLeakage (16 occurrences), showing how these two issues are particularly relevant in real-world data science code and therefore important to be detected. We then compared the warnings raised by PYRA against this ground truth to compute various performance metrics, including accuracy (Acc.), precision (Prec.), recall (Rec.), F1-score, and specificity (Spec.) for both the combined levels of confidence (plausible and potential warnings) and the plausible-only level of confidence.

The overall metrics for both modes are presented in the last rows of Tables 6 and 7, respectively. These metrics are computed across all warnings raised in the 100 selected notebooks and demonstrate that PYRA performs well in both modes, with accuracy values exceeding 92%, a reasonably high F1 score exceeding 71%, and balanced precision and recall values. As expected, the plausible-only mode achieves higher precision (0.9462) but lower recall (0.6685) compared to the combined mode, which achieves a precision of 0.5942 and recall of 0.8913, reflecting the stricter criteria for raising warnings in the plausible-only mode.

A more detailed analysis is shown in Tables 6 and 7, which present the per-warning type metrics for both modes. These tables provide a detailed breakdown of the performance of PYRA for each specific warning type, allowing for a more granular analysis of its effectiveness across different types of issues.

As expected, for some warning types the results are influenced by false positives, while for others they are affected by false negatives. This is entirely anticipated, as some warnings are inherently more challenging to detect accurately through static analysis due to the complexity of the underlying issues they represent, while others may have ambiguous contexts that require user assessment for validity. For instance, the CategoricalPlot warning often presents difficulties in establishing a clear threshold to differentiate between correct and incorrect usage of categorical data in plots, necessitating a deep understanding of the data and analysis context, which can lead to some false positives.

Data leakage detection is also complex, with false negatives related to domain-specific knowledge (e.g., incorrect usage of time series not linked to data preprocessing) or manual operations (such as manual scaling, e.g., $x = (x\_data - np.min(x\_data))/(np.max(x\_data) - np.min(x\_data)).values$) that are not detected by static analysis. Therefore, considering the complexity of the issues being detected and the fact that some warnings have only potential confidence, the results obtained by PYRA are quite satisfactory overall, especially considering that assessing the ground truth took the authors 15 hours, while the analysis with PYRA was much faster for the entire dataset.

5.4. Tool Comparison

In the quantitative evaluation benchmark, we considered the same 100 notebooks for which we manually assessed the ground truth in the quantitative evaluation and also ran another tool for detecting data science code smells, MLScent [32]. We compared its results with those of PYRA. To the best of our knowledge, there are no other publicly available tools that detect as many data science code smells as PYRA, so we focused our comparison on MLScent, which is the closest tool in terms of the number of detected code smells in common. However, the comparison can only be made between the DataLeakage and Reproducibility warnings, as these are the only two code smells detected by both tools.

Unlike PYRA, MLScent does not provide the exact line for each warning, so we compared the results at the notebook level. Specifically, we checked whether each tool raised a warning of a given type for each notebook, regardless of the exact line where the issue was detected, and then manually validated the results.

As shown in Figure 24, PYRA outperforms MLScent in both warning types. For DataLeakage, PYRA raises this warning in 12 different files (10 with plausible confidence and 2 with potential confidence), while MLScent fails to capture any of them, even though they are all true positives. For Reproducibility, this warning is found in 16 files by both analyzers, in 28 files only by PYRA, and in 5 files only by MLScent. However, upon manually assessing these latter files, we found that they were all false positives (e.g.,

Warning Type	Acc.	Prec.	Rec.	F1	Spec.	TP	FP	TN	FN
CategoricalConversionMean	0.941	0.000	0.000	0.000	0.941	0	6	95	0
CategoricalPlot	0.528	0.062	0.833	0.115	0.516	5	76	81	1
DataLeakage	0.922	0.833	0.625	0.714	0.977	10	2	84	6
DuplicatesNotDropped	0.970	0.833	0.714	0.769	0.989	5	1	92	2
FixedNComponentsPCA	1.000	1.000	1.000	1.000	1.000	2	0	98	0
Gmean	0.941	0.000	0.000	0.000	0.941	0	6	95	0
HighDimensionality	1.000	0.000	0.000	0.000	1.000	0	0	100	0
InappropriateMissingValues	0.970	1.000	0.571	0.727	1.000	4	0	94	3
InconsistentType	1.000	0.000	0.000	0.000	1.000	0	0	100	0
MissingData	0.950	0.722	1.000	0.839	0.943	13	5	82	0
NoneRetAssignment	1.000	0.000	0.000	0.000	1.000	0	0	100	0
NotShuffled	0.950	0.824	0.875	0.848	0.965	14	3	82	2
PCAOnCategorical	0.980	0.000	0.000	0.000	0.980	0	2	99	0
PCAVisualization	0.980	0.333	1.000	0.500	0.980	1	2	99	0
Reproducibility	0.960	0.982	0.957	0.969	0.966	111	2	56	5
ScaledMean	0.941	0.000	0.000	0.000	0.941	0	6	95	0
Overall	0.9256	0.5978	0.8967	0.7174	0.9290	165	111	1452	19

Table 6
Per-warning type metrics for combined mode (plausible + potential).

Warning Type	Acc.	Prec.	Rec.	F1	Spec.	TP	FP	TN	FN
CategoricalConversionMean	1.000	0.000	0.000	0.000	1.000	0	0	100	0
CategoricalPlot	0.922	0.000	0.000	0.000	0.979	0	2	94	6
DataLeakage	0.941	1.000	0.625	0.769	1.000	10	0	86	6
DuplicatesNotDropped	0.930	0.000	0.000	0.000	1.000	0	0	93	7
FixedNComponentsPCA	1.000	1.000	1.000	1.000	1.000	2	0	98	0
Gmean	1.000	0.000	0.000	0.000	1.000	0	0	100	0
HighDimensionality	1.000	0.000	0.000	0.000	1.000	0	0	100	0
InappropriateMissingValues	0.931	0.000	0.000	0.000	1.000	0	0	94	7
InconsistentType	1.000	0.000	0.000	0.000	1.000	0	0	100	0
MissingData	0.870	0.000	0.000	0.000	1.000	0	0	87	13
NoneRetAssignment	1.000	0.000	0.000	0.000	1.000	0	0	100	0
NotShuffled	0.842	0.000	0.000	0.000	1.000	0	0	85	16
PCAOnCategorical	1.000	0.000	0.000	0.000	1.000	0	0	100	0
PCAVisualization	0.980	0.333	1.000	0.500	0.980	1	2	99	0
Reproducibility	0.960	0.982	0.957	0.969	0.966	111	2	56	5
ScaledMean	1.000	0.000	0.000	0.000	1.000	0	0	100	0
Overall	0.9608	0.9538	0.6739	0.7898	0.9960	124	6	1492	60

Table 7
Per-warning type metrics for plausible-only mode.

a warning related to a reproducibility issue for a linear regression was raised, but this operation does not involve randomness).

6. Discussion and Threats to Validity

Our evaluation and the design of PYRA are subject to some threats to validity. A first threat concerns false positives and false negatives. Although our experimental results show that PYRA is effective in detecting real code smells, achieving low false positive and false negative rates, and performing favorably compared with a similar state-of-the-art tool, its precision may degrade when the dataset on which the notebook operates is not available. In such cases, PYRA falls back to a fully static approximation, reducing the precision of the inferred datatypes and potentially lowering the quality of the generated warnings. This can result in missed detections as well as spurious alerts.

Another threat arises from the assumption of sequential execution of notebook cells. While sequential execution is common and typically recommended in data-science workflows, it is not guaranteed in general. Out-of-order execution may therefore introduce discrepancies between the abstract state reconstructed by the analysis and the actual runtime behavior of the notebook.

Furthermore, PYRA currently lacks full support for some advanced Python features, such as some object-oriented programming patterns, which, although relatively uncommon

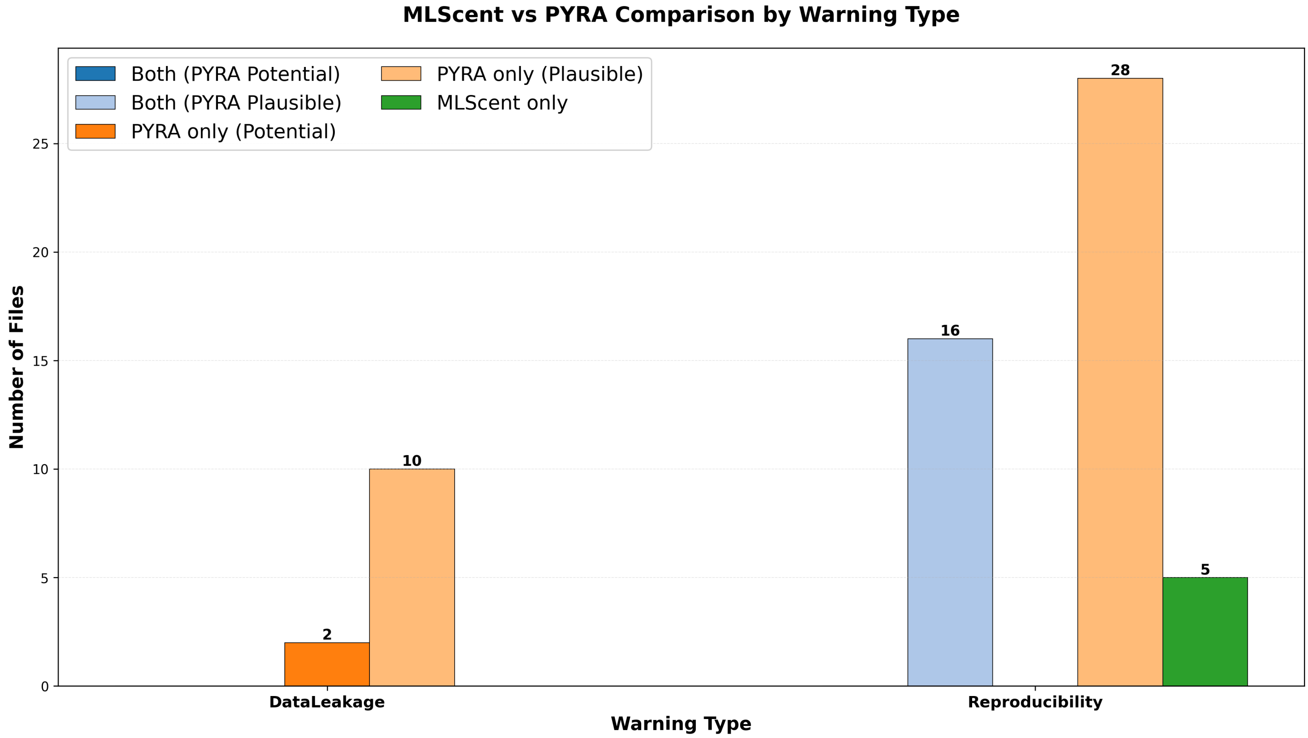

Figure 24: Comparison for DataLeakage and Reproducibility warning with MLScent.

in data science notebooks, may appear in more engineered workflows. As discussed in Section 5.2, this limitation does not undermine the soundness of the proposed type analysis; rather, it reflects the current state of the prototype implementation. Ongoing work is progressively extending feature coverage and improving the robustness and completeness of PYRA.

Finally, we argue that tools like the one proposed in this paper remain valuable in the era of generative AI. Indeed, such tools will be *especially* useful as data analysts increasingly rely on generative models rather than writing code themselves. We envision data analysts using PYRA to validate generated code and leveraging its analysis results and suggestions to repair the code, either manually or with the assistance of LLMs.

7. Conclusion

In this paper, we presented PYRA, a fully automatic static analyzer for Python data science software, aimed at detecting high-level code smells related to typical data science development pipelines rather than low-level programming errors. A key aspect of PYRA is that its warnings are designed to be easily understood not only by static analysis experts, but also, and especially, by data scientists, including early-career ones. We experimentally evaluated PYRA on a set of randomly selected real-world Jupyter notebooks crawled from Kaggle, demonstrating PYRA's ability to detect the high-level data science issues presented and discussed in the paper, despite still being a prototype. Currently, while PYRA supports most of the core features of Python and the most popular data science libraries, some functionalities are still missing (e.g., `nltk` or `statsmodels` libraries). Future work will extend PYRA to broaden the range of Python features and libraries it supports, with the goal of increasing its applicability and usability. In this direction, we also plan to release PYRA as a plug-in for most used IDEs, such as PyCharm and Visual Studio Code.

An interesting direction for future work is to apply PYRA in the medical context, where data science plays a crucial role in tasks such as diagnosis and treatment planning. This would involve investigating domain-specific code smells (e.g., related to data protection and privacy, or associated with the analysis of omics data) and extending PYRA with specific checkers tailored to the unique risks and code smells of medical applications. Such an extension could significantly enhance PYRA's impact and broaden its applicability to critical, high-stakes environments.

Another promising future direction is to integrate PYRA within established quality assessment frameworks. While PYRA effectively detects code smells and potential issues, it does not by itself provide quantitative assessments of quality attributes such as maintainability, security, or reliability. Existing models for post-processing static analysis results, such as the SIG, QUAMOCO, QATCH, and SAM models [14, 25, 46, 45, 33, 34], offer mechanisms to derive actionable quality metrics. Integrating PYRA's output within such frameworks, or developing a similar quality assessment model tailored to data science pipelines, could significantly enhance its practical value for assessing the reliability and maintainability of machine learning systems.

While we target Python, as it is currently the most popular programming language used in data science, the R programming language is also heavily used [35]. We believe that the static analyses described in this paper could be adapted to the R context as well, for instance by integrating them into `flowR` [36], a dataflow static analyzer for R.

Another future relevant direction could be the integration of PYRA within knowledge tracing frameworks for coding tasks, which are aimed at assessing students' capabilities and at predicting their performances. For example in [40], large language models are used to automatically annotate knowledge concepts and PYRA could be used as an additional module to improve concept detection in Python-based data science scenarios.

Finally, at its current stage, PYRA assumes a sequential execution of notebook cells, as this is the recommended way to run a Jupyter notebook. Nevertheless, during the development phase, it is common for users to execute cells in an arbitrary order (e.g., for debugging purposes). To make PYRA applicable in such scenarios as well, a major improvement would be to support the analysis of notebooks under arbitrary execution orders.

8. Data Availability

The source code of PYRA is publicly available at its official Github repository: https://github.com/spangea/Pyra. The materials required to replicate the experimental evaluation presented in this paper are available on Zenodo at https://zenodo.org/records/17895599.

Acknowledgments

This work was supported by Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL_2024_PROGETTI_B_IOTTI - CUP D93C24001250005).

References

[1] Bantilan, N., 2020. pandera: Statistical data validation of pandas dataframes, in: Agarwal, M., Calloway, C., Niederhut, D., Shupe, D. (Eds.), Proceedings of the 19th Python in Science Conference 2020 (SciPy 2020), Virtual Conference, July 6 - July 12, 2020, scipy.org. pp. 116–124. URL: https://doi.org/10.25080/Majora-342d178e-010, doi:10.25080/MAJORA-342D178E-010.

[2] Bühlmann, P., Van De Geer, S., 2011. Statistics for high-dimensional data: methods, theory and applications. Springer Science & Business Media.

[3] Cao, L., 2017. Data science: A comprehensive overview. ACM Comput. Surv. 50, 43:1–43:42. doi:10.1145/3076253.

[4] Cousot, P., 1997. Types as abstract interpretations, in: Lee, P., Henglein, F., Jones, N.D. (Eds.), Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997, ACM Press. pp. 316–331. URL: https://doi.org/10.1145/263699.263744, doi:10.1145/263699.263744.

[5] Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Graham, R.M., Harrison, M.A., Sethi, R. (Eds.), Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, ACM. pp. 238–252. URL: https://doi.org/10.1145/512950.512973, doi:10.1145/512950.512973.

[6] Cousot, P., Cousot, R., 1992. Abstract interpretation and application to logic programs. J. Log. Program. 13, 103–179. URL: https://doi.org/10.1016/0743-1066(92)90030-7, doi:10.1016/0743-1066(92)90030-7.

[7] Dolcetti, G., Arceri, V., Mensi, A., Zaffanella, E., Urban, C., Cortesi, A., 2025. Introducing pyra: A high-level linter for data science software, in: Dutra, I., Pechenizkiy, M., Cortez, P., Pashami, S., Pasquali, A., Moniz, N., Jorge, A.M., Soares, C., Abreu, P.H., Gama, J. (Eds.), Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track and Demo Track - European Conference, ECML PKDD 2025, Porto, Portugal, September 15-19, 2025, Proceedings, Part X, Springer. pp. 449–453. doi:10.1007/978-3-032-06129-4_29.

[8] Dolcetti, G., Cortesi, A., Urban, C., Zaffanella, E., 2024. Towards a high level linter for data science, in: Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains, pp. 18–25.

[9] Drobnjakovic, F., Subotic, P., Urban, C., 2024. An abstract interpretation-based data leakage static analysis, in: Chin, W., Xu, Z. (Eds.), Theoretical Aspects of Software Engineering - 18th International Symposium, TASE 2024, Guiyang, China, July 29 - August 1, 2024, Proceedings, Springer. pp. 109–126. URL: https://doi.org/10.1007/978-3-031-64626-3_7, doi:10.1007/978-3-031-64626-3_7.

[10] Fowler, S., Lindley, S., Morris, J.G., Decova, S., 2019. Exceptional asynchronous session types: session types without tiers. Proc. ACM Program. Lang. 3, 28:1–28:29. doi:10.1145/3290341.

[11] Gentleman, R.C., Carey, V.J., Bates, D.M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., et al., 2004. Bioconductor: open software development for computational biology and bioinformatics. Genome biology 5, 1–16.

[12] Goel, A., Donat-Bouillud, P., Krikava, F., Kirsch, C.M., Vitek, J., 2021. What we eval in the shadows: a large-scale study of eval in R programs. Proc. ACM Program. Lang. 5, 1–23. URL: https://doi.org/10.1145/3485502, doi:10.1145/3485502.

[13] Hassan, M., Urban, C., Eilers, M., Müller, P., 2018. Maxsmt-based type inference for python 3, in: Chockler, H., Weissenbacher, G. (Eds.), Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, Springer. pp. 12–19. URL: https://doi.org/10.1007/978-3-319-96142-2_2, doi:10.1007/978-3-319-96142-2_2.

[14] Heitlager, I., Kuipers, T., Visser, J., 2007. A practical model for measuring maintainability, in: Machado, R.J., e Abreu, F.B., da Cunha, P.R. (Eds.), Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings, IEEE Computer Society. pp. 30–39. URL: https://doi.org/10.1109/QUATIC.2007.8, doi:10.1109/QUATIC.2007.8.

[15] Kapoor, S., Narayanan, A., 2023. Leakage and the reproducibility crisis in machine-learning-based science. Patterns 4, 100804. URL: https://doi.org/10.1016/j.patter.2023.100804, doi:10.1016/J.PATTER.2023.100804.

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

[17] Kramm, M., Chen, R., Sudol, T., Demello, M., Caceres, A., Baum, D., Peters, A., Ludemann, P., Swartz, P., Batchelder, N., Kaptur, A., Lindzey, L., 2019. Pytype: A static type analyzer for python code. URL: https://github.com/google/pytype.

[18] scikit learn.org, . Common pitfalls and recommended practices. URL: https://scikit-learn.org/stable/common_pitfalls.html.

[19] Van der Maaten, L., Hinton, G., 2008. Visualizing data using t-sne. Journal of machine learning research 9.

[20] McKinney, W., et al., 2011. pandas: a foundational python library for data analysis and statistics. Python for high performance and scientific computing 14, 1–9.

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

[22] Monat, R., Ouadjaout, A., Miné, A., 2020. Static type analysis by abstract interpretation of python programs (artifact). Dagstuhl Artifacts Ser. 6, 11:1–11:6. URL: https://doi.org/10.4230/DARTS.6.2.11, doi:10.4230/DARTS.6.2.11.

[23] de Moura, L.M., Bjørner, N.S., 2008. Z3: an efficient SMT solver, in: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, Springer. pp. 337–340. URL: https://doi.org/10.1007/978-3-540-78800-3_24, doi:10.1007/978-3-540-78800-3_24.

[24] Negrini, L., Shabadi, G., Urban, C., 2023. Static analysis of data transformations in jupyter notebooks, in: Ferrara, P., Hadarean, L. (Eds.), 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, ACM. pp. 8–13. URL: https://doi.org/10.1145/3589250.3596145, doi:10.1145/3589250.3596145.

[25] Nugroho, A., Visser, J., Kuipers, T., 2011. An empirical model of technical debt and interest, in: Ozkaya, I., Kruchten, P., Nord, R.L., Brown, N. (Eds.), Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, Waikiki, Honolulu, HI, USA, May 23, 2011, ACM. pp. 1–8. URL: https://doi.org/10.1145/1985362.1985364, doi:10.1145/1985362.1985364.

[26] Paiva, T., Damasceno, A., Figueiredo, E., Sant'Anna, C., 2017. On the evaluation of code smells and detection tools. J. Softw. Eng. Res. Dev. 5, 7. URL: https://doi.org/10.1186/s40411-017-0041-1, doi:10.1186/S40411-017-0041-1.

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

[28] Quaranta, L., Calefato, F., Lanubile, F., 2022. Pynblint: a static analyzer for python jupyter notebooks, in: Crnkovic, I. (Ed.), Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN 2022, Pittsburgh, Pennsylvania, May 16-17, 2022, ACM. pp. 48–49. URL: https://doi.org/10.1145/3522664.3528612, doi:10.1145/3522664.3528612.

[29] Ritchie, M.E., Phipson, B., Wu, D., Hu, Y., Law, C.W., Shi, W., Smyth, G.K., 2015. limma powers differential expression analyses for rna-sequencing and microarray studies. Nucleic acids research 43, e47–e47.

[30] van Rossum, G., Lehtosalo, J., Langa, L., 2014. Pep 484 – type hints. URL: https://peps.python.org/pep-0484/.

[31] Saravanan, N., Sathish, G., Balajee, J.M., 2018. Data wrangling and data leakage in machine learning for healthcare. JETIR- International Journal of Emerging Technologies and Innovative Research 5, 553–557.

[32] Shivashankar, K., Martini, A., 2025. Mlscent: A tool for anti-pattern detection in ML projects, in: 4th IEEE/ACM International Conference on AI Engineering - Software Engineering for AI, CAIN 2025, Ottawa, ON, Canada, April 27-28, 2025, IEEE. pp. 150–160. doi:10.1109/CAIN66642.2025.00026.

[33] Siavvas, M.G., Chatzidimitriou, K.C., Symeonidis, A.L., 2017. QATCH - an adaptive framework for software product quality assessment. Expert Syst. Appl. 86, 350–366. URL: https://doi.org/10.1016/j.eswa.2017.05.060, doi:10.1016/J.ESWA.2017.05.060.

[34] Siavvas, M.G., Kehagias, D.D., Tzovaras, D., Gelenbe, E., 2021. A hierarchical model for quantifying software security based on static analysis alerts and software metrics. Softw. Qual. J. 29, 431–507. URL: https://doi.org/10.1007/s11219-021-09555-0, doi:10.1007/S11219-021-09555-0.

[35] Sihler, F., Pietzschmann, L., Straub, R., Tichy, M., Diera, A., Dahou, A.H., 2025. On the anatomy of real-world R code for static analysis, in: Koziolek, A., Lamprecht, A., Thüm, T., Burger, E. (Eds.), Software Engineering 2025, Fachtagung des GI-Fachbereichs Softwaretechnik, Karlsruhe, Germany, February 24-28, 2025, Gesellschaft für Informatik e.V.. p. 27. URL: https://doi.org/10.18420/se2025-27, doi:10.18420/SE2025-27.

[36] Sihler, F., Tichy, M., 2024. flowr: A static program slicer for R, in: Filkov, V., Ray, B., Zhou, M. (Eds.), Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024, ACM. pp. 2390–2393. URL: https://doi.org/10.1145/3691620.3695359, doi:10.1145/3691620.3695359.

[37] Stekhoven, D.J., Bühlmann, P., 2012. Missforest—non-parametric missing value imputation for mixed-type data. Bioinformatics 28, 112–118.

[38] Subotic, P., Bojanic, U., Stojic, M., 2022a. Statically detecting data leakages in data science code, in: Gonnord, L., Titolo, L. (Eds.), SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022, ACM. pp. 16–22. URL: https://doi.org/10.1145/3520313.3534657, doi:10.1145/3520313.3534657.

[39] Subotic, P., Milikic, L., Stojic, M., 2022b. 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. pp. 13–22. URL: https://doi.org/10.1109/ICSE-SEIP55303.2022.9794067, doi:10.1109/ICSE-SEIP55303.2022.9794067.

[40] Sun, X., Liu, Q., Zhang, K., Shen, S., Yang, L., Li, H., 2025. Harnessing code domain insights: Enhancing programming knowledge tracing with large language models. Knowledge-Based Systems 317, 113396. URL: https://www.sciencedirect.com/science/article/pii/S0950705125004435, doi:https://doi.org/10.1016/j.knosys.2025.113396.

[41] Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., Altman, R.B., 2001. Missing value estimation methods for dna microarrays. Bioinformatics 17, 520–525. URL: https://doi.org/10.1093/bioinformatics/17.6.520, doi:10.1093/bioinformatics/17.6.520.

[42] Urban, C., 2020. What programs want: Automatic inference of input data specifications. CoRR abs/2007.10688. URL: https://arxiv.org/abs/2007.10688, arXiv:2007.10688.

[43] Urban, C., 2023. Static analysis for data scientists, in: Challenges of Software Verification. Springer, pp. 77–91.

[44] Urban, C., Müller, P., 2018. An abstract interpretation framework for input data usage, in: Ahmed, A. (Ed.), Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Springer. pp. 683–710. URL: https://doi.org/10.1007/978-3-319-89884-1_24, doi:10.1007/978-3-319-89884-1_24.

[45] Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., Mayr, A., Plösch, R., Seidl, A., Streit, J., Trendowicz, A., 2015. Operationalised product quality models and assessment: The quamoco approach. Inf. Softw. Technol. 62, 101–123. URL: https://doi.org/10.1016/j.infsof.2015.02.009, doi:10.1016/J.INFSOF.2015.02.009.

[46] Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., Streit, J., 2012. The quamoco product quality modelling and assessment approach, in: Glinz, M., Murphy, G.C., Pezzè, M. (Eds.), 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, IEEE Computer Society. pp. 1133–1142. URL: https://doi.org/10.1109/ICSE.2012.6227106, doi:10.1109/ICSE.2012.6227106.

[47] Wang, J., Li, L., Zeller, A., 2020. Better code, better sharing: on the need of analyzing jupyter notebooks, in: Rothermel, G., Bae, D. (Eds.), ICSE-NIER 2020: 42nd International Conference on Software

Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020, ACM. pp. 53–56. URL: https://doi.org/10.1145/3377816.3381724, doi:10.1145/3377816.3381724.

[48] Waskom, M.L., 2021. seaborn: statistical data visualization. Journal of Open Source Software 6, 3021. doi:10.21105/joss.03021.

[49] Wickham, H., 2011. ggplot2. Wiley interdisciplinary reviews: computational statistics 3, 180–185.

[50] Zhang, H., Cruz, L., van Deursen, A., 2022. Code smells for machine learning applications, in: Crnkovic, I. (Ed.), Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN 2022, Pittsburgh, Pennsylvania, May 16-17, 2022, ACM. pp. 217–228. URL: https://doi.org/10.1145/3522664.3528620, doi:10.1145/3522664.3528620.