The VLDB Journal
https://doi.org/10.1007/s00778-021-00726-w

SPECIAL ISSUE PAPER

Data distribution debugging in machine learning pipelines

Stefan Grafberger[1] · Paul Groth[1] · Julia Stoyanovich[2] · Sebastian Schelter[1]

Received: 27 February 2021 / Revised: 9 September 2021 / Accepted: 3 December 2021

Abstract
Machine learning (ML) is increasingly used to automate impactful decisions, and the risks arising from this widespread use are garnering attention from policy makers, scientists, and the media. ML applications are often brittle with respect to their input data, which leads to concerns about their correctness, reliability, and fairness. In this paper, we describe `mlinspect`, a library that helps diagnose and mitigate technical bias that may arise during preprocessing steps in an ML pipeline. We refer to these problems collectively as *data distribution bugs*. The key idea is to extract a directed acyclic graph representation of the dataflow from a preprocessing pipeline and to use this representation to automatically instrument the code with predefined *inspections*. These inspections are based on a lightweight annotation propagation approach to propagate metadata such as lineage information from operator to operator. In contrast to existing work, `mlinspect` operates on declarative abstractions of popular data science libraries like estimator/transformer pipelines and does not require manual code instrumentation. We discuss the design and implementation of the `mlinspect` library and give a comprehensive end-to-end example that illustrates its functionality.

Keywords Data debugging · Machine learning pipelines · Data preparation for machine learning

1 Introduction

Machine learning (ML) is increasingly used to automate decisions that impact people's lives, in domains as varied as credit and lending, medical diagnosis, and hiring, with the potential to reduce costs, reduce errors, and make outcomes more equitable. Yet, despite their potential, the risks arising from the widespread use of ML-based tools are garnering attention from policy makers, scientists, and the media [52]. In large part this is because the correctness, reliability, and fairness of ML models critically depend on their training data. Preexisting bias, such as under- or over-representation of particular groups in the training data [12], and technical bias, such as skew introduced during data preparation [49], can heavily impact performance. In this work, we focus on helping diagnose and mitigate technical bias that arises during preprocessing steps in an ML pipeline. We refer to these problems collectively as *data distribution bugs*.

Data distribution bugs are often introduced during preprocessing Input data for ML applications come from a variety of data sources, and it has to be preprocessed and encoded as features before it can be used. This preprocessing can introduce skew in the data, and, in particular, it can exacerbate under-representation of historically disadvantaged groups. For example, preprocessing operations that involve filters or joins can heavily change the distribution of different groups represented in the training data [58], and missing value imputation can also introduce skew [47]. Recent ML fairness research, which mostly focuses on the use of learning algorithms on static datasets [14], is therefore insufficient because it cannot address such technical bias originating from the data preparation stage. Furthermore, it is important to detect and mitigate bias as close to its source as possible [52].

Data distribution bugs are difficult to catch In part, this is because different pipeline steps are implemented using different libraries and abstractions, and data representation often

✉ Sebastian Schelter
s.schelter@uva.nl

Stefan Grafberger
s.grafberger@uva.nl

Paul Groth
p.t.groth@uva.nl

Julia Stoyanovich
stoyanovich@nyu.edu

[1] University of Amsterdam, Amsterdam, Netherlands

[2] New York University, New York, USA

changes from relational data to matrices during data preparation. Further, preprocessing in the data science ecosystem [44] often combines relational operations on tabular data with *estimator/transformer pipelines*.[1] These pipelines are composable and nestable abstractions for operations on array data. The approach originates from scikit-learn [37] and has been adopted by libraries like SparkML [28] and TensorFlow Transform.[2] Tracing problematic featurized entries that may be the result of nested function calls back to the pipeline's initial human-readable input is tedious work.

We need automated inspection of ML pipelines Due to the pressures of their day-to-day activities, most data scientists will not invest the necessary time and effort to manually instrument their code or insert logging statements for tracing, as required by model management systems [53,60]. We envision support for data scientists in the form of *automated inspections of their pipelines*, similar to the inspections used by modern IDEs to highlight potentially problematic parts of a program, such as the use of deprecated code. Once data scientists become aware of such issues, they can use data debuggers like Dagger [26] to drill down into the specific intermediate pipeline outputs and explore the root cause of the issue. We furthermore argue that, to be most beneficial, automated inspections need to *work with code natively written with popular ML library abstractions*.

Lightweight pipeline inspection with `mlinspect` We design and implement `mlinspect`, a library that helps data scientists automatically detect data distribution bugs in their ML pipelines. The `mlinspect` library extracts logical query plans, modeled as directed acyclic graphs (DAGs) of preprocessing operators, from pipelines that use popular libraries like pandas and scikit-learn [37], and that combine estimator/transformer pipelines and relational operators. The pipeline code is then automatically instrumented to trace the impact of operators on properties like the distribution of sensitive groups in the data. In this way, `mlinspect` empowers data scientists to automatically and comfortably check their ML pipeline code for data distribution bugs.

Importantly, `mlinspect` provides a library-independent interface to propagate annotations such as the lineage of tuples across operators from different libraries and introduces only constant overhead per tuple flowing through the DAG. Thereby, `mlinspect` offers a general runtime for pipeline inspection and allows for integration of many detection techniques for data distribution bugs that previously required custom code, such as automated model validation of data slices [42], identification of distortions with respect to protected group membership in the training data [58], and automated dataset sanity checking [21].

[1] https://scikit-learn.org/stable/modules/compose.html.

[2] https://github.com/tensorflow/transform.

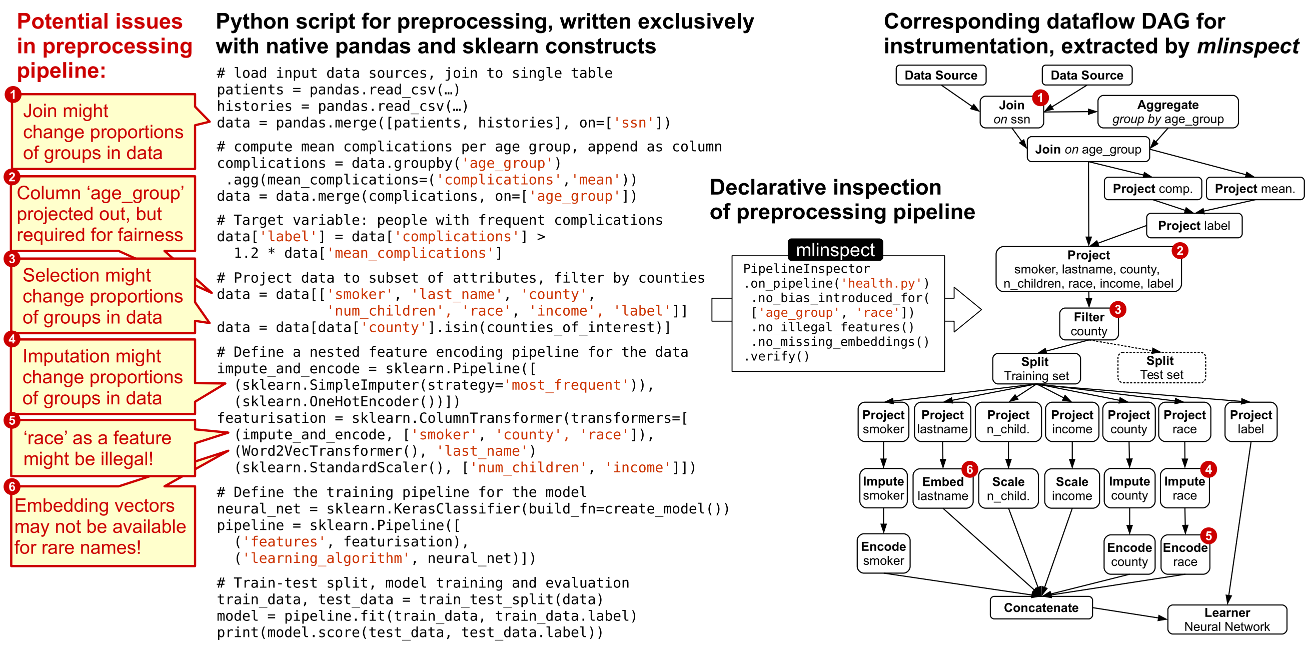
We proposed the initial ideas for our approach in earlier work [17]. In this paper, we give a comprehensive description of the approach and of the corresponding open source library. We explain how to instrument estimator/transformer pipelines (Sect. 3.2), provide implementation details for all our components (Sect. 4), and add an extensive discussion of related work (Sect. 6). We also present quantitative and qualitative experiments to evaluate `mlinspect` with respect to its runtime overhead and usability.
In this paper, we make the following contributions:

– We describe hard-to-identify issues in ML preprocessing pipelines with respect to the fairness and correctness of the resulting models (Sects. 2, 3.3).
– We discuss the design of `mlinspect`, which enables lightweight lineage-based inspection of ML preprocessing pipelines. The `mlinspect` library bases its analysis on declarative abstractions of popular data science libraries and does not require manual code instrumentation (Sect. 3).
– We describe how to efficiently implement the instrumentation and inspections of `mlinspect` and how to enable support for control flow (Sect. 4).
– We experimentally show that the runtime overhead of `mlinspect` is linear in the number of input and output records of instrumented operators and highlight performance trade-offs (Sect. 5).
– We provide a qualitative comparison of our approach to related libraries for experiment tracking and provenance capturing. We also conduct a user study, showing that `mlinspect` is helpful to data scientists in their data distribution debugging tasks (Sect. 5).

2 Data distribution bugs by example

We illustrate the need for assisting data scientists with the inspection of their preprocessing pipelines with an example from the medical domain, shown in Fig. 1. Consider a data scientist who implements a Python pipeline that takes demographic and clinical history data as input, and trains a classifier to identify patients at risk for serious complications. Further, assume that the data scientist is under a legal obligation to ensure that the resulting model works equally well for patients across different age groups and races. This obligation is operationalized as an intersectional fairness criterion, requiring equal false-negative rates for groups of patients identified by a combination of `age_group` and `race`.

The pipeline first reads two CSV files, which contain patient demographics and their clinical histories, respectively. Next, the resulting dataframes are joined on the `ssn` column. This join may introduce a data distribution bug (as indicated by issue ①) if a large percentage of the records of


Potential issues in preprocessing pipeline:
1 Join might change proportions of groups in data
2 Column 'age_group' projected out, but required for fairness
3 Selection might change proportions of groups in data
4 Imputation might change proportions of groups in data
5 'race' as a feature might be illegal!
6 Embedding vectors may not be available for rare names!
Python script for preprocessing, written exclusively with native pandas and sklearn constructs
Declarative inspection of preprocessing pipeline
mlinspect
Corresponding dataflow DAG for instrumentation, extracted by *mlinspect*
Data Source
Data Source
Join on ssn
Aggregate group by age_group
Join on age_group
Project comp.
Project mean.
Project label
Project smoker, lastname, county, n_children, race, income, label
Filter county
Split Training set
Split Test set
Project smoker
Project lastname
Project n_child.
Project income
Project county
Project race
Project label
Impute smoker
Embed lastname
Scale n_child.
Scale income
Impute county
Impute race
Encode smoker
Encode county
Encode race
Concatenate
Learner Neural Network

```python
# load input data sources, join to single table
patients = pandas.read_csv(…)
histories = pandas.read_csv(…)
data = pandas.merge([patients, histories], on=['ssn'])

# compute mean complications per age group, append as column
complications = data.groupby('age_group')
 .agg(mean_complications=('complications','mean'))
data = data.merge(complications, on=['age_group'])

# Target variable: people with frequent complications
data['label'] = data['complications'] >
  1.2 * data['mean_complications']

# Project data to subset of attributes, filter by counties
data = data[['smoker', 'last_name', 'county',
             'num_children', 'race', 'income', 'label']]
data = data[data['county'].isin(counties_of_interest)]

# Define a nested feature encoding pipeline for the data
impute_and_encode = sklearn.Pipeline([
  (sklearn.SimpleImputer(strategy='most_frequent')),
  (sklearn.OneHotEncoder())])
featurisation = sklearn.ColumnTransformer(transformers=[
  (impute_and_encode, ['smoker', 'county', 'race']),
  (Word2VecTransformer(), 'last_name')
  (sklearn.StandardScaler(), ['num_children', 'income']])

# Define the training pipeline for the model
neural_net = sklearn.KerasClassifier(build_fn=create_model())
pipeline = sklearn.Pipeline([
  ('features', featurisation),
  ('learning_algorithm', neural_net)])

# Train-test split, model training and evaluation
train_data, test_data = train_test_split(data)
model = pipeline.fit(train_data, train_data.label)
print(model.score(test_data, test_data.label))
```

```python
PipelineInspector
.on_pipeline('health.py')
 .no_bias_introduced_for(
 ['age_group', 'race'])
 .no_illegal_features()
 .no_missing_embeddings()
.verify()
```

Fig. 1 Example of an ML pipeline that predicts which patients are at a higher risk of serious complications, under the requirement to achieve comparable false-negative rates across intersectional groups by age and race. The pipeline is implemented using native constructs from the popular pandas and scikit-learn libraries. On the left, we highlight potential issues identified by `mlinspect`. On the right, we show the corresponding dataflow graph extracted by `mlinspect` to instrument the code and pinpoint issues. (Operations on the test set are omitted for readability)

some combination of age group and race do not have matching entries in the clinical history dataset.

Next, the pipeline computes the average number of complications per age group and adds the binary target label to the dataset, indicating which patients had a higher than average number of complications compared to their age group. Data is then projected to a subset of the attributes, to be used by the classification model. This leads to the second issue ② in the pipeline: the data scientist needs to ensure that the model achieves comparable accuracy across different age groups, but the age group attribute is projected out here, making it difficult to catch this data distribution bug later in the pipeline. The data scientist additionally filters the data to only contain records from patients within a given set of counties. This may lead to issue ③: a data distribution bug may be introduced if populations of different counties systematically differ in age.

Next, the pipeline creates a feature matrix from the dataset by applying feature encoders with scikit-learn's `ColumnTransformer`, before training a neural network on the features. For the categorical attributes `smoker`, `county`, and `race`, the pipeline imputes missing values with mode imputation (using the most frequent attribute value), and subsequently creates one-hot encoded vectors from the data. The `last_name` attribute is replaced with a corresponding vector from a pretrained word embedding, and we normalize the numerical attributes `num_children` and `income`.

This feature encoding part of the pipeline introduces several potential issues: ④ the imputation of missing values for the categorical attributes may introduce statistical bias by attributing records with a missing value of `race` to the majority race in the dataset; ⑤ depending on the legal context (i.e., if the disparate treatment doctrine is enforced[3]), it may be forbidden to use `race` as an input to the classifier; ⑥ we may not have vectors for rare non-western names in the word embedding, which may in turn lead to lower model accuracy for such records. As illustrated by this example, preprocessing can give rise to subtle data distribution bugs that are difficult to identify manually, motivating the development of our automatic inspection library, `mlinspect`.

3 Design of mlinspect

The analysis of Python code for data science pipelines is difficult because, in contrast to SQL queries, these pipelines are not built on top of an algebraic abstraction. Further, these pipelines operate not only on relational data but also on tensors, when converting input data to feature matrices. However, popular data science libraries expose a set of

[3] https://en.wikipedia.org/wiki/Disparate_treatment.

declarative abstractions with some algebraic properties. For example, pandas and pyspark both operate on dataframes with SQL-like operations, and scikit-learn, SparkML, and TensorFlow Transform[4] rely on (potentially nested) estimator/transformer chains.

This abstraction consists of an *estimator* that conducts an aggregation over its inputs to create a reusable *transformer*. The transformer applies a tuple-at-a-time transformation to the data based on the state computed by its corresponding estimator. This abstraction allows data scientists to build nested pipelines of estimators and transformers that combine common operations like feature transformations (like one-hot encoding of categorical variables) with model training and hyperparameter optimization (like k-fold cross-validation). The estimator/transformer abstraction can be seen as a declarative way to specify ML pipelines and has recently been the subject of database-style research to optimize execution time [50].

3.1 Overview

We propose `mlinspect`, a runtime for lightweight lineage-based inspection of python scripts that uses existing library code and does not require manual code instrumentation. In the current research prototype, we restrict ourselves to scripts that use a combination of SQL-like operations on dataframes and estimator/transformer pipelines, analogously to our example in Sect. 2. This has the potential to cover a wide range of existing ML code: According to results of a recent analysis of several million Jupyter Notebooks, more than 50% of these use pandas, and more than 25% use scikit-learn [44]. The `mlinspect` library focuses on declarative pipeline code, supports control flow, and has fallbacks for when it encounters unsupported code snippets.

The `mlinspect` library extracts a directed acyclic graph (DAG) representing the dataflow from ML pipelines with logical operators like join, selection, projection, column encoders, and missing value imputation. Based on this extracted DAG, `mlinspect` automatically instruments the code with predefined lightweight *inspections* that detect data distribution bugs in the pipeline and give hints to users.

We now give a high-level overview of how `mlinspect` executes and inspects data preprocessing operations based on the architecture shown in Fig. 2. The execution takes place as follows: (1) Users execute their data science pipeline implemented in native pandas/sklearn code via `mlinspect` and define the inspections to apply; (2) `mlinspect` automatically instruments relevant function calls (Sect. 3.2) and executes the instrumented program; (3) during the execution, `mlinspect` delegates instrumented function calls to library-specific backends, which expose the inputs, annotations, and outputs of operators to the configured inspections (Sect. 3.3); (4) `mlinspect` extracts a dataflow representation of the program (Sect. 3.4) and maps the results of the inspection to the corresponding operators. In the remainder of this section, we detail the design of each component. We will discuss implementation decisions in Sect. 4.

[4] Note that TensorFlow Transform refers to estimators and transformers as TensorFlow Transform Analyzers and TensorFlow Ops https://www.tensorflow.org/tfx/tutorials/transform/simple?hl=en.

3.2 Instrumentation and annotation propagation

Instrumentation and DAG extraction at runtime We conduct all instrumentation necessary for inspection before the execution of the pipeline and extract the DAG at runtime during a single execution of the pipeline, as follows. During the execution of each instrumented function call, corresponding operator nodes are added to the DAG. For this, `mlinspect` generates a unique identifier for each DAG node. Whenever a dataframe object is returned from an instrumented function, `mlinspect` adds a new attribute that contains the identifier of the DAG operator that produced the dataframe. For example, when processing the `pd.merge(df_a, df_b)` call, `mlinspect` retrieves the DAG node identifiers for `df_a` and `df_b` and adds a new DAG node, in this case a `JOIN`, with nodes representing `df_a` and `df_b` as parents. There might be cases where a user pipeline contains operators that `mlinspect` cannot recognize (e.g., custom transformers in a scikit-learn pipeline). Such operators are ignored and not represented in the DAG, and execution continues with the remaining known operations. Due to this fallback, the library does not fail for pipelines where it recognizes only a subset of the relevant dataflow operations, but still applies all inspections and checks on a best-effort basis.

Handling control flow Early `mlinspect` versions [17] lacked support for control flow in pipelines; they created the DAG based on the pipeline code after execution, using module information obtained through Python's `inspect` module. This made it difficult to deal with conditional code such as loops, where the number of iterations depends on runtime variables. The current DAG extraction method supports pipelines with control flow by building up the DAG dynamically at runtime based on the actual execution of the program. If there are branches in the user code, only operators from the executed branch are contained in the DAG. As a consequence, `mlinspect` now runs and instruments pipeline code contained in custom functions, which leverage loops and branches. This approach enables easy instrumentation of relevant function calls, even if they happen indirectly (as is the case with nested scikit-learn pipelines). We refer to Sect. 4.3.2 for further details.

Annotation propagation The data flowing through the preprocessing pipeline is further enriched with user-definable "annotations" that propagate through operators and can be

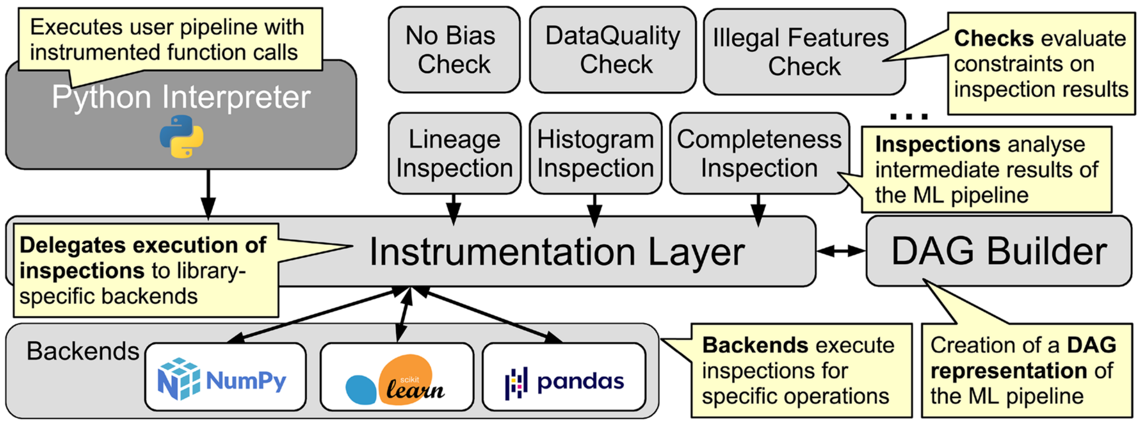

Executes user pipeline with instrumented function calls
Python Interpreter
No Bias Check
DataQuality Check
Illegal Features Check
Checks evaluate constraints on inspection results
...
Lineage Inspection
Histogram Inspection
Completeness Inspection
Inspections analyse intermediate results of the ML pipeline
Delegates execution of inspections to library-specific backends
Instrumentation Layer
DAG Builder
Backends
NumPy
learn
pandas
Backends execute inspections for specific operations
Creation of a DAG representation of the ML pipeline

Fig. 2 Architecture of `mlinspect`. We apply checks and inspections to an instrumented ML pipeline written by the user. The instrumentation layer delegates the execution of the inspections to library-specific backends and creates a DAG representation of the pipeline

created, read, and modified by the inspection code. This annotation propagation mechanism offers a simple library-independent interface to propagate annotations (e.g., for tracking the lineage of tuples) across operators from different libraries. We base the design of our *inspections* on this annotation propagation mechanism. Each inspection retains a fixed-size state that is reset after each operator and is invoked only once for each DAG operator. The inspection has access to the output tuples of the operator and the corresponding annotated inputs. The following listing details the abstract operations performed by such an inspection. At runtime, the `visit_op` method is called for each operator invocation and provided with information about the operator as well as an iterator over the annotated input rows. The inspection then produces the corresponding output annotations and can optionally annotate the logical operator in the DAG with the computed result (such as a histogram of the outputs) via the `op_annotation_after_visit` method.

```
# Abstract base class for all inspections
class Inspection:

  # Inspect intermediate data at a DAG operator, based on
  #  operator information (op_context), and an iterator
  #  over annotated input rows with the corresponding
  #  output rows (row_iterator);
  # Return computed annotations for output rows
  def visit_op(self, op_context,
               row_iterator) -> Iterable

  # Persist inspection result for the current DAG node
  def op_annotation_after_visit(self)
```

Users have to specify the inspections to apply in advance, which allows only the state that is required for the actual inspections configured by the user to be materialized. This avoids materializing arbitrary information from the pipeline.

As long as each row annotation has a fixed size limit, and each inspection only uses a fixed-size state, the overhead of the framework is constant per inspected tuple. This approach does not introduce additional memory overhead, as there is only the constant overhead of a fixed number of additional function calls per user function call.

We maintain a mapping between the input rows of an operator and their corresponding output rows and then expose this mapping along with the corresponding annotated inputs to each inspection. This input/output mapping is constructed differently depending on operator semantics. Operators like projection and transformers are guaranteed to have the same number of input and output elements, listed in the same order. For operators like selection, join, and train–test split, the mapping is maintained by generating an identifier column, which is transparently pushed through the operator and removed immediately afterward to hide it from user code. Note that only one possible source tuple (and not all possible sources) is tracked for aggregation operators and for duplicate elimination, as the performance overhead of detailed provenance tracking using the full provenance semiring framework [18] would be too significant, introducing dependencies between all input–output pairs [3].

Function call capturing To allow inspections to access the output of an operator such as a join, along with the corresponding input rows and their annotations, arguments and return values of function calls must be efficiently captured. For this, the abstract syntax tree (AST) from the Python parser is modified before compiling and executing the code. A function call is added before the user code to "monkey patch" functions from libraries like pandas and scikit-learn that are supported by `mlinspect`. Monkey patching [55] allows `mlinspect` to extend or modify functionality of third-party libraries at runtime by completely replacing the original implementation of a function. These monkey patched functions internally call the original, unpatched version of the function, delegate the execution of the inspections, and create new DAG operator nodes corresponding to the function. `mlinspect` also captures the exact function call location and source code snippet corresponding to each DAG operator. See Sect. 4.3.1 for implementation details.

Backends for popular Python libraries The mlinspect library is designed based on the semantics of preprocessing operations from popular Python frameworks like scikit-learn and pandas. The instrumentation based on captured function calls described so far is independent of the specific library. Importantly, libraries differ in their data representation choices and in what data preprocessing operations they support. So, pandas functions can be directly mapped to DAG operators, and each operation is executed eagerly. In contrast, scikit-learn encourages users to first declaratively define a nested pipeline using components like the ColumnTransformer, which allows passing specific columns to specific transformers like one-hot encoders. Once a pipeline is defined in a declarative way, data is passed to the nested pipeline object in a second, separate step. The function calls that actually process data, such as the fit/transform calls of transformers contained in scikit-learn pipeline objects, may not be directly visible in user code. The user pipeline only calls the fit method once on the final pipeline object, and the pipeline then internally calls the fit and transform functions of the transformers and estimators it contains. We introduce library-specific backends in mlinspect to handle the operations and data representations of popular libraries like scikit-learn.

Execution of inspections Each backend is responsible for hiding library implementation details from the inspections. The pandas backend, for example, is responsible for calling the inspections as necessary whenever it is alerted of a pandas function call. For this, it has access to the arguments and return values as described before. The backend then needs to map operator output rows to operator input rows and their corresponding annotations. It needs to create efficient iterators to expose the input/output rows in a specific format. Afterward, the backend stores the resulting new annotations created by the inspection in an efficient manner (e.g., as attributes of the processed dataframe in the case of pandas).

This annotation propagation functionality is enough to implement a variety of useful inspections. For example, basic fine-grained lineage tracking on the row level can be implemented with a simple inspection on top of the annotation propagation approach as follows: unique identifier annotations are generated for each row after the data source operator and are propagated forward through the DAG. For selections, projections, and transformers, annotations are directly forwarded through the DAG. For joins, combinations of identifier annotations from all join inputs are created and forwarded.

Optimizable inspections based on dataframe operators In addition to the generic interface for inspections written in Python, a second interface for inspections is supported. In this interface, inspections have to be expressed in terms of operations on dataframes. This approach is less general than the standard approach (which allows for arbitrary Python code), but is much more performant, because inspections can be jointly executed with the user code operations, and common optimizations from query processing such as scan sharing and projection pushdowns can be applied. We discuss implementation details in Sect. 4.3.2. Note, this approach is still in an experimental stage and not yet part of the open-source release.

3.3 Automatic inspections and checks

Inspections serve as the basis for detecting data distribution bugs in ML pipelines. They annotate the extracted DAG with information like computed histograms for different DAG nodes. On top of the extracted and annotated DAG, mlinspect provides *checks*, a rule-based approach to verify constraints on the DAG, for example, by comparing the change in a histogram to a threshold. Before execution, mlinspect determines which inspections are required based on the checks specified by the user. It then instruments the pipeline and executes it using a minimal set of inspections, based on what is required by the checks and directly specified by the user. After the execution of the instrumented pipeline and the DAG extraction, each check can access the final result to evaluate its constraint.

In the following, we discuss a set of more complex automatic inspections and checks for ML preprocessing pipelines that are enabled by our lineage-based annotation propagation approach.

Algorithmic fairness In recent years, problems with respect to the fairness of ML-based decision-making systems have been uncovered [52]. Such problems are often difficult to detect and are the focus of mlinspect. As discussed in the example from Sect. 2 and outlined in previous work [58], operations like join and selection can accidentally filter out records from protected groups and thereby *introduce or exacerbate under-representation of historically disadvantaged groups in the data*. The mlinspect library provides an inspection that computes histograms of operator outputs based on protected groups, and alerts the user if group membership proportions change drastically after an operator. A related problem is the low coverage of some population groups identified by a combinations of attributes [7]. For tracing group membership in coverage-related problems, mlinspect forward-propagates annotations identifying the groups of interest and materializes the annotated input and final output of the complete pipeline.

Furthermore, there are *legal restrictions on the usage of demographic features* such as gender, race, or disability status in automated decision making. One can check the operator DAG against a list of sensitive features and alert the user about the places in the code where such features are used.

ML models may also *perform particularly badly for specific demographic groups* in the data (e.g., yielding higher false-positive rates for recidivism predictions for African Americans [6]). The identification of such groups is in the focus of recent research [42]. This identification might be difficult in cases where the attribute required to identify the protected group is projected out early in the pipeline or is only available as a specific dimension of the feature matrix during feature transformation. To address this, `mlinspect` supports inspections that forward-propagate sensitive column annotations and then materialize the minimum amount of information needed for analyzing performance for different groups: rows only containing the predicted label and the sensitive columns.

Methodology and robustness Additionally, inexperienced data scientists may make methodological mistakes, such as fitting featurizers on the whole data instead of the training set only, forgetting to scale numerical features even though the model requires that (as in the case of L2 regularization), or selecting hyperparameters on the test set instead of the validation set. Such issues can impact fairness-related metrics as well [47]. All of these issues can be identified by analyzing the extracted operator DAG. Furthermore, there may be robustness issues in the pipeline. For example, some scikit-learn transformers cannot handle null values. One can identify such cases from the operator DAG and recommend that the user applies a simple imputation technique. Another problem that can be detected by analyzing histograms of operator outputs is *class imbalance*. The DAG can be analyzed to see whether the data scientist already addresses these with resampling or reweighing and alert her otherwise.

Data quality Data quality testing in the form of unit tests for data as offered by libraries like Deequ [48] can also be implemented using `mlinspect`. Data unit tests typically evaluate constraints based on aggregate statistics of the data such as the completeness (ratio of non-NULL values) of a column or the number of distinct values in a column. The `mlinspect` library can compute these data quality statistics over all intermediate results of a pipeline.

3.4 Algebraic definition of the `mlinspect` dataflow graph

Data preparation pipelines that use declarative abstractions such as pandas data slicing, scikit-learn's `ColumnTransformer`, or SparkML pipelines have a natural directed acyclic graph (DAG) representation [46]. Data sources in this DAG are typically comprised of tables or files holding relational data. The data flowing through the DAG is either collections of relational tuples or tensors. The operators are either relational operators like join, selection, and projection (consuming relational data and producing relational data), standard feature encoders like one-hot encoders (consuming relational data and producing vectors), or standard ML preprocessing operations like normalization or concatenation (consuming vectors and producing vectors). In the following, we list the operations supported by the current implementation of `mlinspect` in Table 1, and discuss their formalization. We would like to note that we focus on common operations from pandas and scikit-learn in our current research prototype. That said, the instrumentation approach of `mlinspect` is general, and extending its capabilities to support additional functions can be done with moderate engineering effort.

Dataframe algebra We introduced our operators as a mixture of relational algebra operators with estimator/transformer pipelines. However, relational algebra is insufficient to formalize `mlinspect` operators because it operates on unordered collections, while typical exploratory operations on dataframes (like printing the first or last n rows) assume an ordered data representation [39]. Estimator/transformer pipelines in scikit-learn also fundamentally rely on order: transformers map over a list and transform the data without changing the order (e.g., when converting categorical strings to one-hot vectors). Model training methods also assume that their inputs are ordered, by implicitly associating each featurized datapoint with its corresponding label. Furthermore, support for linear algebra is crucial for typical ML pipelines, because many operations, especially for feature processing, have a natural representation as matrix operations and are internally implemented on numerical array data structures. In addition, dataframes in libraries like pandas offer many specialized methods that do not have an equivalent in relational algebra [39]. Examples include the `TRANSPOSE` operation that interchanges rows and columns, and the `TOLABELS` operation that projects a column out to use it as a row label.

Peterson et al. [39] observed that dataframes combine operations from relational algebra, linear algebra and spreadsheets and proposed a novel dataframe algebra to unify them. We use this algebra as a basis for the abstract representation of ML pipelines, in order to formalize our approach. Because `mlinspect` currently focuses on ML pipelines that use relational operations and estimator/transformer operators, we only require a subset of the dataframe algebra.

Operator formalization Peterson et al. [39] define a *dataframe* as a tuple (A_{mn}, R_m, C_n, D_n), where A_{mn} is an array of entries from the domain Σ^*, R_m is a vector of row labels from Σ^*, C_n is a vector of column labels from Σ^*, and D_n is a vector of n domains from *Dom*, one per column, representing the schema of the dataframe. Each component of the tuple can be left unspecified. Since D_n can be left unspecified, there is a schema induction function $S(\cdot)$ that, when applied to a column of A_{mn}, returns its domain i. Function $p(\cdot)$ can be used to get the values of the column. This definition allows to represent matrices as dataframes with a

Table 1 Functions supported by `mlinspect` and their corresponding operators in the dataflow representation of the pipeline

Function call	Operator
`('pandas.io.parsers', 'read_csv')`	Data Source
`('pandas.core.frame', 'DataFrame')`	Data Source
`('pandas.core.frame', '__getitem__')`, arg type: strings	Projection
`('pandas.core.frame', '__getitem__')`, arg type: series	Selection
`('pandas.core.frame', 'dropna')`	Selection
`('pandas.core.frame', 'replace')`	Projection (Mod)
`('pandas.core.frame', '__setitem__')`	Projection (Mod)
`('pandas.core.frame', 'merge')`	Join
`('pandas.core.groupbygeneric', 'agg')`	Groupby/Agg
`('sklearn.compose._column_transformer', 'ColumnTransformer')`, column selection	Projection
`('sklearn.compose._column_transformer', 'ColumnTransformer')`, concatenation	Concatenation
`('sklearn.preprocessing._encoders', 'OneHotEncoder')`	Transformer
`('sklearn.preprocessing._data', 'StandardScaler')`	Transformer
`('sklearn.impute._base', 'SimpleImputer')`	Transformer
`('sklearn.preprocessing._discretization', 'KBinsDiscretizer')`	Transformer
`('sklearn.tree._classes', 'DecisionTreeClassifier')`, `('tensorflow.python.keras.wrappers.scikit_learn', 'KerasClassifier')`,...	Estimator
`('sklearn.model_selection._split', 'train_test_split')`	Split (Train/Test)
`('sklearn.preprocessing._label', 'label_binarize')`	Projection (Mod)
`('sklearn.pipeline', 'fit')`, arg: train data	Train Data
`('sklearn.pipeline', 'fit')`, arg: train labels	Train Labels

homogeneous numeric schema D_n, with *null* labels R_m and C_n. See Figure 3 in Peterson et al. [39] for an illustration.

We detail the representation of the one-hot encoder operator in this algebra as an example. Given a $DF = (A_{m,1}, R_m, C_1, D_1)$ with a categorical string column, the one-hot encoder is a map operator $MAP(DF, f)$ with the output $(A'_{mn'}, R_m, C'_{n'}, D'_{n'})$, and the function $f : D_n \rightarrow D'_{n'}$, where $A'_{mn'}$ is the result of the function f as applied to each row, $C'_{n'}$ is the resulting column labels, and $D'_{n'}$ is the resulting vector of domains. For a one-hot encoder, f is a function that transforms each categorical string into an n'-dimensional vector, where n' is the domain cardinality of D_1, with only a single nonzero entry in the dimension corresponding to the string value in a given row. The cardinality n' of the string column becomes the number of dimensions of the one-hot vectors and, thus, also the number of columns in the result dataframe. The column labels $C'_{n'}$, in this case, are generated by combining the attribute and string values.

In general, our operators map to this algebra as follows. Our DAGs start with one or multiple `Data Source` operators. In the dataframe algebra, the initial data inputs are not operators, rather, they are modeled as leaf nodes in their DAG. Our operator `Projection` has the same semantics as the PROJECTION operator in the dataframe algebra. The corresponding operator for our `Projection (Mod)` is a MAP because the dataframe algebra does not have extended projections but uses the MAP operator instead to also handle that functionality. Our `Selection` and `Join` operators work exactly like their equivalents in the dataframe algebra, SELECTION and JOIN. Our `Group by Agg` operator works like the GROUPBY operator in the dataframe algebra that can directly apply aggregation functions. Note that the GROUPBY operation in the data frame algebra is more powerful than ours, in that it offers a `collect` aggregation function that can group rows into multiple dataframes, which we do not support. The MAP function in the dataframe algebra applies a function uniformly to every row. Our `Transformers` have the same semantics as these MAPs. Our `Estimator` can also be expressed as a MAP that does not produce an output. The `Split (Train/Test)` and its two outputs can be expressed using a MAP to add a temporary column, a SELECT to filter records using this column, and a PROJECT to remove the temporary column afterward. The `Concatenation` can be used to append the columns of multiple dataframes that have the same number of records. In the dataframe algebra, this can be done using TRANSPOSE to interchange the columns and the rows, followed by a UNION of the two dataframes, and then a TRANSPOSE again.

Additionally, we enrich our DAG representation of ML pipelines with other information inferred from the pipeline code, which is potentially helpful for further analysis. Examples for this are the `Train Data` and `Train Label` DAG nodes that mark the data on which `estimator.fit` was called. Clearly, identifying the exact version of train and test data used to fit the ML model greatly simplifies the implementation of inspections. When formalizing our DAG operators, these operators can be ignored, as they result in *no-op* label nodes that do not change the semantics of the ML pipeline query but they simplify its analysis.

Discussion As we already pointed out, the major difference between the dataframe algebra and the relational algebra is order preservation. Relational algebra operates on sets of tuples, while dataframes are modeled as ordered collections of tuples, and operations on them preserve this order. This property is a fundamental obstacle for the efficient pushdown [23] of the execution of ML pipelines and inspections into relational databases, as we would either need to implement order-preserving variants of common relational operators, or introduce artificial sort columns and always sort query results based on them.

4 Implementation

We now discuss the salient aspects of the implementation of `mlinspect` and revisit the example from Sect. 2. Our research prototype is available at: https://github.com/stefan-grafberger/mlinspect.

4.1 Overview

Our research prototype contains the core operator DAG extraction functionality, and it implements instrumentation, checks, and inspections for pandas and scikit-learn. We offer implementations of representative inspections, including an inspection that materializes the first row output by each operator, an inspection that tracks the detailed lineage of all rows flowing through the DAG, data quality inspections, and an inspection that computes histograms of operator outputs for sensitive groups. In addition, we offer implementations of checks, which evaluate a constraint on the outputs of our inspections, such as a threshold comparison of the magnitude of change in the proportions of certain groups in the data after a filter.

4.2 Inspections

Some checks only require the extracted DAG for analysis. An example for this is the `NoIllegalFeatures` check, which inspects the names of projected attributes used as features to ensure that no illegal features, such as gender or race, are used. Other checks only require simple inspections that investigate an operator in isolation. An example is the `NoMissingEmbeddings` check, which simply

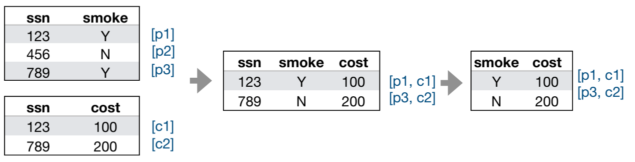


Fig. 3 Lineage tracking by propagating identifier annotations through operators

counts the null values in the outputs of embedding operators. Another example are inspections for data unit testing. Data unit tests typically evaluate constraints based on aggregate statistics of the data such as the completeness (ratio of non-NULL values) of a column or the number of distinct values in a column. Often, these statistics only require a single pass over the data and can therefore be pipelined with the actual execution of an operator. The `Completeness` and `NumDistinctValues` inspections compute these statistics by iterating over the values of a given column and maintaining the counts for NULL/non-NULL values (for completeness) or a hashmap containing the number of occurrences per distinct value.

In general, however, inspections need to work with the data annotations flowing through the operators at runtime, as described in the previous sections. In the following, we discuss two such cases in detail: lineage tracking and change detection for proportions of protected groups.

Lineage tracking It is simple to integrate lineage tracking into `mlinspect` directly using the built-in annotation propagation mechanisms. As part of lineage tracking, unique identifier annotations for all input tuples are generated and forwarded according to operator semantics (e.g., for a join, a combination of the identifier annotations of matching tuples are forwarded).

We implement lineage tracking (Fig. 3) via the lineage inspection. To illustrate our approach, we use a pandas code snippet that joins a table of patient data with a table of cost data, and projects the result to the attributes `smoke` and `cost`.

```
patient = pd.read_csv(...)
cost = pd.read_csv(...)
data = pd.merge([patient, cost], on="ssn")
data = data[["smoke", "cost"]]
```

The `visit_op(self, op_context, row_iterator)` function of the inspection is called first, as `patient` data is loaded on line 1. The inspection then checks the type of the current operator. In our example, operator type, *data source*, is contained in the `op_context`. After checking this, the inspection generates unique identifiers for each row. This process is repeated for the `cost` data source on line 2. The third call to `visit_op` corresponds to the join, which results from the `pd.merge` call on line 3. There, `visit_op` operates on five-tuples comprised of the output row from the join, the corresponding rows from the two dataframes `patient` and `cost`, and the annotations for the two input rows. The two input annotations are then combined to create the output annotation. For projection on `smoke` and `cost` on line 4, we only need to forward-propagate the existing input annotations.

One notable case not shown here is lineage inspection for the groupby operator type, where the aggregation following the groupby is treated as a new data source. We expect that the detailed lineage information from aggregations is not relevant for many ML use cases, which often mostly apply global aggregations (e.g., for normalizing features), where each tuple depends on the whole input anyways. We leave a more fine-grained treatment of aggregations for future work.

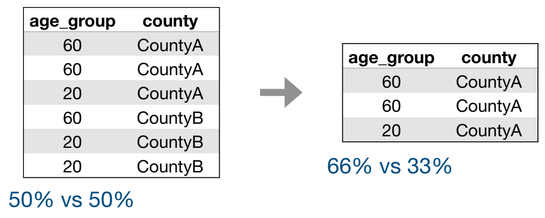
Change detection for proportions of protected groups In our running example (Fig. 1 in Sect. 2), we briefly discussed an inspection to discover the introduction of accidental changes in the proportions of protected groups. This refers to the issues ①, ②, ③ and ④ from the example and requires the histogram inspection to (*i*) trace the group membership variables `age_group` and `race` through the DAG, and handle the fact that `age_group` is projected out early (issue ②). We designed a custom check called `NoBiasIntroducedFor` for such cases. Internally, this check uses the `HistogramForColumns` inspection, which we will now explain. Consider the following selection statement:

```
data = data[data.county == "CountyA"]
```

Figure 4 shows how this selection might affect an example dataset flowing through it. Before the selection, the two `age_groups`, 60 and 20, are distributed evenly. After the selection, the majority of data points is in the `age_group` 60. This is an artifact of the strong correlation between the attribute `county` and the attribute `age_group`. Our simple example illustrates a common real-world trend, namely, that geographic and demographic attributes are often correlated.

To detect such distribution changes, we apply the `HistogramForColumns(['age_group'])` inspection that annotates both the DAG node before the selection and the selection DAG node itself with an `age_group` histogram of the outputs. After inspection execution and DAG extraction, the `NoBiasIntroducedFor` check can then look at these two annotated DAG nodes. For each sensitive attribute, it checks whether there is a significant distribution change of group memberships, and, if so, alerts the user.

We use a simple detection strategy that is easy for users to understand and configure. We start by calculating the group membership ratio compared to the overall number of people in the data. Here, this group membership ratio for people with `age_group=20` is 0.5 before the selection and 0.33 after it. We compute the relative change before and after the selection as $(0.33 - 0.5)/0.5 = -0.34$. We then compare this quotient to a test threshold, set to -0.3. If the change is below that minimal threshold, as is the case in our example, we warn the user. This approach is especially sensitive to changes in the proportion of minority groups.



age_group	county
60	CountyA
60	CountyA
20	CountyA
60	CountyB
20	CountyB
20	CountyB

age_group	county
60	CountyA
60	CountyA
20	CountyA

Fig. 4 Histogram-based change detection for the proportions of protected groups in operators such as selections and joins. Here, in the beginning, the two age groups are distributed evenly, with a drastic change after the operator application

What is not encountered in this example is the removal of a group membership attribute. If projection is used to remove the attribute `age_group`, we annotate each row with its corresponding `age_group` value and propagate these row annotations forward. Subsequent operations like join, selection, and missing value imputation, which may change group proportions in the data, rely on these propagated group membership annotations to compute a histogram of group memberships of all inspected operator outputs, and test them for distribution changes.

We implement additional inspections to compute histograms of intersectional group membership. We also provide a check for calculating the removal probabilities of different demographic groups in the data. This check detects cases where filter-like operations that affect only a small subset of the data disparately impact specific demographic groups.

4.3 Execution of inspections, checks, and DAG extraction

Next, we discuss the detailed execution of inspecting a preprocessing script with `mlinspect`. The execution proceeds according to the following steps (which we detail in the remainder of this section):

1. **Preparation:** Determination of a minimal required set of inspections based on the inspections and checks specified by the user.
2. **Instrumentation:** Instrumentation of function calls in the AST of the user program.
3. **Execution of the instrumented program:** Delegation of the execution of inspections to library-specific backends; joint execution with pipeline operations; creation of the dataflow DAG.
4. **Results:** Evaluation of checks using the DAG and the inspection results.

4.3.1 Preparation

Determining a minimal required set of inspections The first step consists of determining which inspections to execute. Users have two ways to specify inspections: they can either use the check API or specify inspections they are interested in directly. We collect all of the required inspections from these two sources and build a unified set with them.

Capturing relevant function calls As discussed in Sect. 3.2, we instrument the user code via monkey patching and callback functions. It is crucial to only patch relevant function calls, due to the high amount of additional function calls for the callback functions. Determining whether a given function call is relevant for us (e.g., maps to an operator in our DAG) is difficult without executing the code. Monkey patching allows us to create specific patches for function calls relevant for `mlinspect`, while leaving other function calls unaffected. We leverage the Python package `gorilla`[5], which simplifies monkey patching, while also retaining the original unpatched version of the function. When a user executes source code with `mlinspect`, AST nodes corresponding to the following code before and after the original user code are added. The two added function calls only need to be executed once per user script and patch all functions supported by `mlinspect` from libraries like pandas and scikit-learn.

```
from mlinspect.instrumentation
 import monkey_patch, undo_monkey_patch
monkey_patch()
# ...original user code...
undo_monkey_patch()
```

Handling indirect function calls Monkey patching affects all calls to a patched function, even though we only want to execute inspections for calls relevant to the user pipeline. An example for a problematic case is the constructor `pandas.DataFrame(...)`, which is internally used by Pandas as well. As we are only interested in the invocations by our user program, we detect whether a certain operation is directly called by the user program as follows: In the patched code, we call the Python function `sys._getframe` to determine the source code filename of the stack frame of the call and check whether the source file is the root level file executed by `mlinspect`.

Example We present the code for a simplified example of our instrumentation technique, which adds support for the sklearn function `label_binarize` (which creates a binary vector from a categorical column with two distinct values). We initiate the patching of the method `label_binarize` in the package `sklearn.preprocessing` via gorilla's annotations. Next, we implement a patched version of the function, which creates a new DAG operator and retrieves the corresponding DAG parent node and the input annotations required for our inspections. Afterward, we call both the backend responsible for the operation (the `SklearnBackend` in this case), as well as the original function and insert the newly created operator node to our DAG. We would like to note that adding support for a new API function to `mlinspect` only requires a similar patching implementation, which makes it easy to extend our library with moderate engineering efforts.

```
@gorilla.patches(sklearn.preprocessing)
class SklearnPreprocessingPatching:
  @gorilla.name('label_binarize')
  @gorilla.settings(allow_hit=True)
  def execute_label_binarize(*args, **kwargs):
    original = gorilla.get_original_attribute(
      sklearn.preprocessing, 'label_binarize')
    # Patched function
    def patched(...):
      function_info = FunctionInfo(
        'sklearn.preprocessing._label',
        'label_binarize')
      # Operator mapping for DAG
      op_ctx = OperatorContext(
        OperatorType.PROJECTION_MODIFY,
        function_info)
      parent_info = get_parent_node_info(
        args[0], ...)
      # Initiate inspection execution via
      #  backend
      input_df = SklearnBackend.before_call(
        op_ctx, [parent_info])
      # Execute original function
      result = original(input_df,
        *args[1:], **kwargs)
      # Finalize inspection execution via
      #  backend
      backend_result = SklearnBackend\
        .after_call(op_ctx, input_df, result)
      # Append DAG node with inspection result
      add_new_operator_node_to_dag(
        DagNode(...), [parent_info],
        backend_result)
      # Return original result
      return backend_result.updated_result_df
    return execute(original, patched, *args,
      **kwargs)
```

Indirect data processing ML pipelines often contain several functions calls that only lead to data processing indirectly. Scikit-learn's `ColumnTransformer` pipeline step for specifying a set of feature transformations on a dataframe is an example for this. The user code defines a nested pipeline first and then passes the data to it in a second step by calling `fit` on the final pipeline object. The resulting `fit` calls on the contained transformers such as a `OneHotEncoder` or the projections required by the `ColumnTransformer` are only executed indirectly. Our approach identifies and handles these indirect calls by patching the constructors of the pipeline steps and using the source code location retrieved during the constructor invocation to determine that the fit calls originate from the user pipeline code (and must therefore be handled by the system).

Tracking source code locations of operators Python stack frames only contain the line number of the corre-

[5] https://pypi.org/project/gorilla/.

sponding operations. mlinspect can add extra function calls to the AST to track code locations. The AST of the user program, extracted by the Python parser, contains more detailed information: nodes have the attributes lineno and coloffset that indicate the start of the code location, and one can also determine where the snippet corresponding to an operator ends (the end_lineno and end_coloffset). These two attributes are provided by a recent addition to the parser in Python 3.8. Instrumentation is conducted with an ast.NodeTransformer in Python, where the code locations are directly added as arguments to callback functions. This more detailed tracking is configurable, as the additional function calls introduce a minor overhead. We experimentally evaluate the overheads of different instrumentation techniques in Sect. 5.1.4.

4.3.2 Execution of the instrumented program

After instrumenting the user pipeline code, the instrumented AST is compiled and executed, which triggers the execution of the patched functions and the build up of the DAG as described in Sect. 3.2. The execution of each inspection is delegated to the corresponding backend, e.g., inspections for a merge call on a pandas dataframe will be handled by the pandas backend. The API for the different backends comprises of two functions: before_call and after_call, where the before_call function can modify the input before the original function is called. In case of a pandas merge call, for example, an index column is introduced to later associate output rows with the corresponding input rows. The after_call method then executes the inspections and removes metadata such as the index column.

Handling control flow We discuss the implementation details for handling control flow (Sect. 3.2). In order to be able to work with pipelines containing control flow, a DAG is built from the actual execution of the program, instead of just relying on information in the AST (as in previous versions of mlinspect [17]). This prior approach does not allow for the determination of which branches are executed. The current version directly patches function calls, independently of where they occur. Based on these function calls, the DAG is built up dynamically at runtime. During the execution of a patched function, the current stack frame is investigated to determine whether the function call is relevant for the inspections, as described in Sect. 4.3.1. We carefully implemented the corresponding logic to ensure a low overhead for repeated function calls that are not of interest to mlinspect, and experimentally evaluate this overhead in Sect. 5.1.4.

Efficient execution of our Python-based inspections via scan-sharing We implement inspections to both consume and produce iterators, based on for-comprehensions and the yield keyword in Python.

```
def visit_op(self, op_context, rows) -> Iterable
  for row in rows:
    annotation = annotate_and_update_state(self,row)
    yield annotation
```

The inspections are supplied with an iterator over their input rows. To create the iterator, three different arguments are needed: the output of the operator, the corresponding input, and the annotations for the input. They all have the same order and an equal number of rows, so one can scan over those three list-like elements at the same time to create the row_iterator. However, we only want to do a single scan over this even if we have multiple inspections. The only complication is that each inspection has its own separate annotations for each record. The following listing shows how scan-sharing is done with Python iterators and the itertools library[6]. It starts by creating multiple iterators over the input and output rows, one copy per inspection. For each inspection, an iterator is constructed over the inspection's annotations of the input rows. Finally, the functions zip and map are used to create a single iterator that outputs simple data class objects with the current input row, the input row annotation, and the output row. These data class object iterators are the input for the inspections.

```
# Duplicate iterators for each inspection
duplicated_inputs = itertools.tee(input_rows,
   len(inspections))
duplicated_outputs = itertools.tee(output_rows,
   len(inspections))
# Create the inspection_iterator for each inspection
for inspection_index, _ in enumerate(inspections):
  inputs = duplicated_inputs[inspection_index]
  outputs = duplicated_outputs[inspection_index]
  annotations = iterator_for_annotation(
    input_annotations, inspection_index)
row_it_for_inspection = map(
  lambda input_tuple: RowUnaryOperator(*input_tuple),
   zip(inputs, annotations, outputs))
```

The function itertools.tee internally uses one iterator over the input and one over the output and buffers the values until each duplicated iterator processed the value. All inspections consume the iterator elements at the same pace, so only one pass over the data is being made and itertools.tee only needs to buffer the current input and output row. This approach is based on the *banana split law* [20] for loop fusion. When we have multiple functions that we can express using a fold (e.g., computing the count or the sum for a numerical column), we can build a single fold function that combines them to conduct the same computation with a single pass over the data. Here, the visit_op functions of each inspection work similarly to folds. Therefore, we can apply the fusion from the banana split law, to avoid repeated scans over the data.

Handling different types of data Backends also provide a custom function to create datatype-specific iterators for all datatypes that can currently be passed around in the supported ML pipelines. For example, the following listing shows the code to create iterators for pandas dataframes.

[6] https://docs.python.org/3/library/itertools.html.

Table 2 Overview of the internal operator types

Operator(s)	Operator type
Data Source, Group by Agg	Data Source
Projection (Mod), Transformer, Train Data, Train Labels	Unary map
Concatenation	N-ary map
Selection, Train/Test-Split	Unary resampling
Join	Join
Estimator	Sink

```
def get_df_row_iterator(dataframe):
  column_info = ColumnInfo(list(
    dataframe.columns.values))
  arrays = []
  arrays.extend(dataframe.iloc[:, k] for k in
       range(0, len(dataframe.columns)))
  return column_info, map(tuple, zip(*arrays))
```

We provide corresponding implementations for other datatypes like the `ndarray` in numpy, the `Series` in pandas, the sparse matrix `csr_matrix` in scipy, and plain Python `list` objects. Our support for tensors is currently restricted to two-dimensional cases where it is obvious which dimensions correspond to the rows and columns of a dataframe. A prime example for this is feature matrices built from vectorized input samples. We leave support for operations on higher-dimensional tensors (e.g., to represent images, pixels, and channels in three dimensions) for future work.

Instrumentation for different operator types To execute our inspections, we only need to differentiate between a small set of different types of operators, as listed in Table 2. We base the classification on the number of parent operators, whether the operator produces output data, and whether the operator can change the order or number of elements. A `Data Source`-type operator does not get input data from a parent operator and does produce an arbitrary output. A `Unary map` uses the data from one parent operator as input and outputs one output row per input row without changing the existing order of elements. The `N-ary map` has data from multiple parent operators as input, each of them having the same number of elements, and maps n-tuples of input rows to one output row without changing the existing order of elements. `Unary resampling` receives data from one parent operator as input, and can arbitrarily reorder or drop input elements to produce its output. A `Join`-type operator receives input data from multiple parent operators, and combines and reorders them in arbitrary ways to produce its output. A `Sink`-type operator gets input data from a parent operator but does not produce any output data.

The previous examples assumed the operator type of a unary map. In the following, we describe how to handle the remaining types of operators. Data source operator types are simpler because we do not have input data or input annotations we need to consider. The N-ary map works analogously: we can associate row annotations, input, and the corresponding output based on them having the same order and number of elements. The only difference is that we have multiple input dataframes instead of a single one, each with its own annotations. The sink also works analogously; we can associate input and input annotations based solely on the order and number of elements. Functions for operators of the type unary resampling require more complex logic to associate input rows, input annotations and the corresponding output rows. For them, an index column to the input data using the callback functions like `before_call` needs to be added. After execution, this column is removed during the `after_call` function to hide it from the user code. We then utilize these index columns as follows. We start by concatenating the input and the input annotations. Next, we read the index column and join the annotated input with the output. Subsequently, we create iterators over this join result, giving us the required for input, output, and the different annotations. The remaining execution proceeds analogously to the unary map function. In the case of joins, we need to apply the described indexing techniques for both join inputs. In the majority of cases, we use pandas dataframes as data structure to store the actual annotations. They are convenient because we can then leverage joins and concatenation in pandas for the execution of inspections. Once the data is inside a scikit-learn pipeline, we switch to plain Python lists to store the annotations.

Optimizable inspections based on dataframe operators A drawback of our Python-based inspections is the high runtime overhead inherited from Python and a lack of vectorization, which typically requires calling external C code. Due to this, we design an alternative, less general but more efficient method for executing inspections. As outlined in Sect. 3.2, we also support the implementation of inspections based on dataframe operators. The core idea is to model both the inspections and the user program operations as dataframe operators and execute them jointly. This approach is less general than allowing users to write arbitrary python code for inspections, but has a much lower overhead, as we can leverage optimized operator implementations (which apply vectorization) and common techniques from query optimization.

For this approach, inspections are again expressed via two functions, one for computing output annotations for each row and one for computing the final annotations for the current DAG operator. However, instead of relying on the Python generator abstraction, these functions return a partial query plan comprised of dataframe operators. For the annotation propagation, inspections still operate on output rows of the instrumented user operations and the corresponding annotated input rows, but express the computation of the output annotations for each row with dataflow operators.

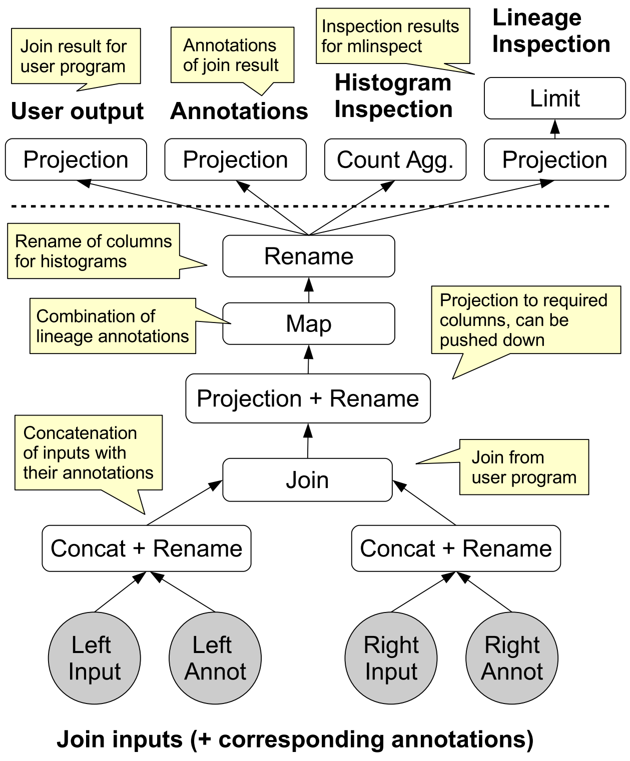

Join result for user program
Annotations of join result
Inspection results for mlinspect
Lineage Inspection
Histogram Inspection
User output
Annotations
Limit
Projection
Projection
Count Agg.
Projection
Rename of columns for histograms
Rename
Combination of lineage annotations
Map
Projection to required columns, can be pushed down
Projection + Rename
Concatenation of inputs with their annotations
Join
Join from user program
Concat + Rename
Concat + Rename
Left Input
Left Annot
Right Input
Right Annot
Join inputs (+ corresponding annotations)

Fig. 5 Example for optimizable inspections: we generate and execute a query plan to apply the histogram and the lineage annotations to a join on two dataframes

Example We discuss how to build up a query plan to apply the histogram and the lineage annotations to a join on two dataframes, illustrated in Fig. 5. As shown in the figure, we start by concatenating each of the two input dataframes with the dataframes holding their input annotations. Next, we apply the original user operation, the join. We use a projection on the joint results to create the result from which the inspections compute the output annotations. This dataframe contains all input columns from both sides and the output columns. This dataframe offers our optimized inspection the same logical view with separated input and output columns as we provide for the Python-based inspections. The histogram inspection forwards the existing annotation column and renames it to follow our naming conventions for inputs and outputs; the lineage inspection combines the two lineage annotation input columns using a map operation. Now, we have a dataframe with the annotated output rows and their corresponding input rows.

In our example, we compute four final outputs from the intermediate dataframe with the annotated output rows and the corresponding inputs. The first output is for the user program: the original result dataframe of the join without annotations and inputs. We use a projection on the intermediate result to remove the annotations and the input columns. Subsequently, we compute the dataframe containing the new output annotations for each row, again using a projection to retrieve only the annotation columns from the intermediate result. The last two outputs correspond to the DAG node annotations from the histogram inspection and the lineage inspection. The histogram inspection uses a groupby operation with a count aggregation, while the lineage inspection applies a limit operation and a projection to materialize the first n output rows and their lineage annotations. Finally, we can optimize and execute the query plan. We experimentally evaluate the performance benefits of this approach in Sect. 5.1.3.

Garbage collecting the annotations Once we obtain the final data structure with the annotations, we need to decide where to store it. One option would be to just save the annotations in the different backends. For example, we could maintain a map from specific function calls to the annotations. However, this would result in unnecessary memory overhead because we do not know when we can free the annotation variables. We only want to remember annotations for a variable as long as that version of the variable exists. For this reason, we store the annotations along with the variables themselves. We achieve this for each data representation relevant to the ML pipelines by either adding the attributes to the original class via monkey patching for pure Python classes, or via a simple wrapper class for classes like numpy arrays that are partially implemented directly in C. These wrapper classes extend the original class and do not change the behavior in any way observable by the original pipeline. Based on this design, the garbage collector of the Python runtime automatically takes care of freeing obsolete annotations.

4.3.3 Extraction of the dataflow graph and evaluation of checks

As discussed in Sect. 3.2, we extract the DAG during the execution of the instrumented user code. As a consequence, the DAG exactly represents the actual dataflow, even if the user code has complex control flow. After obtaining all inspection results and the dataflow graph, we evaluate all user-specified checks on the DAG and the inspection results. Finally, `mlinspect` returns the complete DAG, the inspection results, and the check results.

4.4 Implementation of our example

We provide an executable implementation of our example [7] from Sect. 2, along with a Jupyter Notebook [8] that details

[7] https://github.com/stefan-grafberger/mlinspect/tree/19ca0d6ae8672249891835190c9e2d9d3c14f28f/example_pipelines.

[8] https://github.com/stefan-grafberger/mlinspect/blob/19ca0d6ae8672249891835190c9e2d9d3c14f28f/demo/feature_overview/feature_overview.ipynb.

and visualizes the automatically extracted DAG representation and inspection results for this example. We offer a declarative API for users to state their expectations using the aforementioned checks, which we will then internally convert to constraints on inspection results, e.g.,

```
PipelineInspector
 .on_pipeline_from_py_file('healthcare.py')
 .check(NoBiasIntroducedFor(['age_group',
   'race']))
 .check(NoIllegalFeatures())
 .check(NoMissingEmbeddings())
.execute()
```

The expectation about the lack of the introduction of technical bias refers to the issues ①, ②,③, and ④ from our example and requires the aforementioned change detection inspection from Sect. 4.2 to (i) trace the group membership variables `age_group` and `race` through the DAG, and handle the fact that the former is projected out early (issue ②).

With this in mind, `mlinspect` proceeds as follows: when we visit the projection operator that removes the attribute, we annotate each row with its corresponding `age_group` value and propagate these row annotations forward; (ii) the join, selection, and imputation operators might change the proportions of groups in the data. To handle this, we use the propagated group membership annotations, compute a histogram of group memberships of all inspected operator outputs, and test them for distribution changes afterward. To check whether illegal features have been used (issue ⑤), we simply search the list of projected attributes that are used as features. This information is available as part of our DAG. The check for missing embeddings (issue ⑥) only requires counting the null values in the outputs of the embedding operator.

5 Experimental evaluation

In this section, we present results of an extensive quantitative and qualitative evaluation of `mlinspect`. In Sect. 5.1, we measure the runtime overhead of `mlinspect` for different operators, inspections, and instrumentation techniques. Then, in Sect. 5.2, we present results of an interview-based user study of effectiveness of `mlinspect`. Finally, in Sect. 5.3, we qualitatively compare our library to an experiment tracking and workflow provenance solution.

5.1 Runtime overhead

As `mlinspect` operates on Python scripts and allows for user-defined inspection functions with generic code, it naturally runs in Python, inheriting its overheads. Therefore, our experiments focus on the overhead in terms of the number of input and output rows of the operators. We designed our approach with a constant overhead per tuple and therefore expect the overhead to be linear in the number of input and output rows of an instrumented operator. This is due to the fact that our design requires us to only conduct a single scan over operator inputs and outputs to execute our Python-based inspections and to only materialize intermediate results of interest, which requires a constant overhead per processed row for our discussed inspections. We present a set of experiments to measure the runtime overhead of our `mlinspect` research prototype. We evaluate the overhead of instrumenting operators in Sect. 5.1.1, the overhead of our Python-based inspection execution in Sect. 5.1.2, and we show how we can drastically reduce the inspection overhead with our optimized execution of inspections in Sect. 5.1.3. Additionally, we measure the overhead of instrumenting function calls in the AST in Sect. 5.1.4.

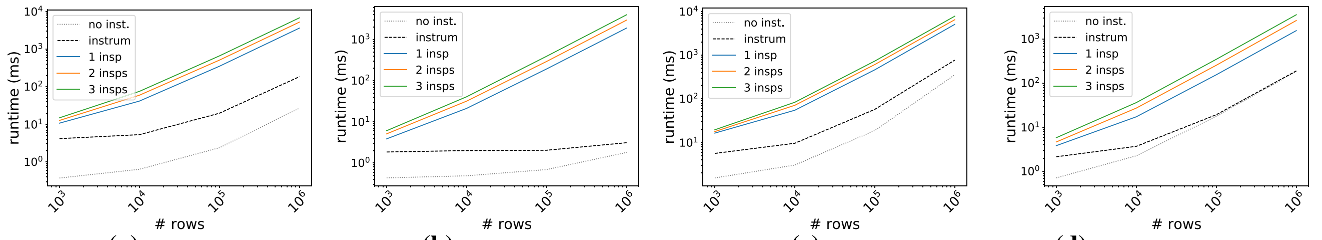
5.1.1 Overhead of python-based operator instrumentation

In our first experiment, we measure the runtime overhead of instrumenting different operators. In particular, we focus on the selection, projection and join operators of pandas, and on an ML-specific operator, the one-hot encoder from scikit-learn, which transforms a categorical string column into a sparse matrix representation. For each operator, we measure the execution time (i) without instrumentation; (ii) with instrumentation without inspections; and (iii) with instrumentation and with one to three empty inspections that read the respective inputs and outputs of operators but do not propagate annotations.

We report the average runtime from 20 repetitions of the experiment for 1000 to 1,000,000 input rows on the logarithmic scale. (For join, we generate the same number of rows for both join inputs.) The results are shown in Fig. 6. We observe the expected increase in the absolute runtime stemming from our usage of Python. However, the overhead per tuple is constant, indicated by the fact that the runtime overhead grows linearly with the number of input and output rows for all operators, as expected. We scale with operator output size for operations like many-to-many joins, where the output is potentially larger than the inputs. This is because inspections need to scan all output rows, along with the corresponding input rows and input annotations. Note that the runtime for projection without instrumentation, and with instrumentation but without inspections, is constant due to the underlying columnar data layout.

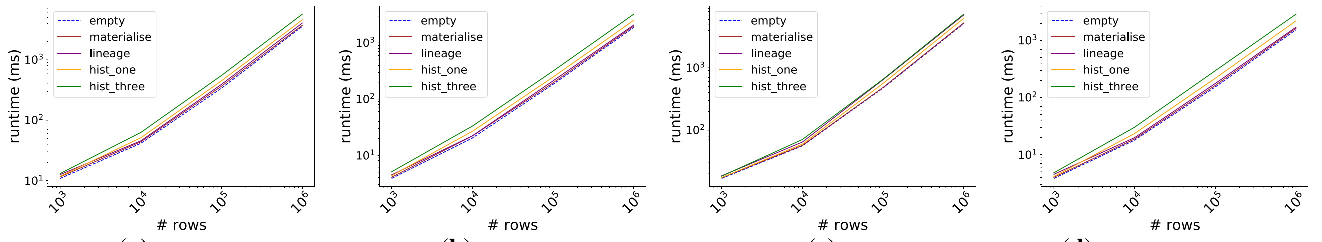
5.1.2 Python-based inspection overhead

We repeat our experiment with the four previously chosen operators and measure the runtime overhead of inspections. For each instrumented operator, we compare the runtime of an empty inspection to the runtime of the following inspections (each of which scans all processed rows): (i) materialize



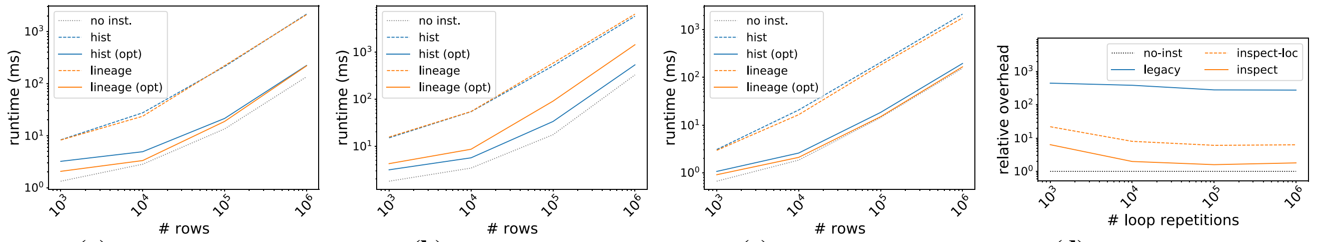
(a) Selection. **(b)** Projection. **(c)** Join. **(d)** One-Hot-Encoder.

Fig. 6 Instrumentation overhead for different operators. We compare the runtime of the execution of a given operator with no instrumentation (`no inst`), instrumentation without inspections (`instrum`), and with one to three empty inspections. We find that the overhead is linear in the number of input and output rows of the operators



(a) Selection. **(b)** Projection. **(c)** Join. **(d)** One-Hot-Encoder.

Fig. 7 Runtime overhead for different inspections in various operators. We compare the runtime of the execution of a given instrumented operator with an "empty" inspection (`empty`) to inspections for materialization (`materialize`), lineage tracking (`lineage`) and histogram computation for one and three columns (`hist_one` and `hist_three`). We find that the overhead is linear in the number of input and output rows of the operators



(a) pandas `dropna`. **(b)** pandas `merge`. **(c)** sklearn `1-hot`. **(d)** AST instrumentation.

Fig. 8 **(a)–(c)** Runtime overhead for executing inspections. Our optimized execution with dataframe operators reduces the overhead by an order of magnitude compared to the Python-based execution and exhibits an overhead of less than 8% compared to non-instrumented execution in some cases; **(d)** AST instrumentation overhead for function calls in a loop. Our patched-based instrumentation approach outperforms the previous approach by up to an order of magnitude and its runtime is within a factor of two of the uninstrumented runtime for a large number of repetitions with disabled code location tracking

a sample of output rows for each operator; (ii) track the lineage via annotation propagation for a sample of output rows for each operator; (iii) compute histograms over one or three columns of the outputs for each operator. We report the average runtime from 20 repetitions of the experiment for 1000, 10,000, 100,000, and 1,000,000 input rows.

The results are shown in Fig. 7. We again observe an overhead for all inspections that is linear in the number of input and output rows. We see that the overhead for the actual inspection logic (e.g., lineage tracking via annotation propagation) is low compared to the empty inspection, which indicates that most of the overhead stems from instrumentation and data access. We also see that the overhead of running additional inspections within one execution is a tiny fraction of the overall instrumentation overhead. This is a validation of the benefits of our loop fusion technique from Sect. 4.3.2. Recall that we implement our inspections with generator-like iterators that yield their elements, and execute the inspections in a way that avoids multiple scans over the data by exposing each record to all inspections during a single scan over the data.

5.1.3 Optimized execution of inspections

We introduced an additional approach to execute inspections in Sect. 4.3.2, based on query plans built from dataframe operations. This approach is less general than Python-based inspections from Sect. 5.1.2 that allow for arbitrary Python code, but has a much lower overhead. In the following, we evaluate both approaches on three operators and two inspections. We implement the optimized execution for our lineage and histogram inspections applied to the pandas functions `dropna` and `merge`, as well as for the `OneHotEncoder` from scikit-learn. We vary the number of randomly generated input rows from 1000 to 1,000,000 on the logarithmic scale and compare the runtime of the original operation without instrumentation `no_inst`, the Python-based `mlinspect` execution from Sect. 5.1 (`hist` and `lineage`), and the optimized execution with dataframe operators (`hist-opt` and `lineage-opt`).

Figure 8a–c shows the results of this experiment. We find that the relative overhead of our optimized inspections is an order of magnitude lower than for the Python-based execution. For the highest number of rows in this experiment, the overhead varies between the factors of only 1.08 and 4.3, compared to the runtime of the operation without instrumentation. This is due to the fact that we can optimize data access during the execution of the query plan corresponding to the inspections, that is, the lineage inspection no longer needs to scan all of the data. For the one-hot encoder, for example, it only needs to forward-propagate the existing lineage annotation column. We only materialize a small row sample from the output dataframe with an additional lineage column, and apply selection pushdown to optimize the computation of the final DAG node annotation. In summary, the optimized execution strategy drastically reduces overhead.

5.1.4 AST instrumentation overhead

In Sect. 4.3.1, we introduced an improved patch-based AST instrumentation mechanism. In the following, we measure the overhead of the instrumentation approach in a worst-case scenario, where it is necessary to instrument a cheap function that is invoked an excessive number of times. The following code snippet is used for the experiment, where list access via an index subscript is executed n-times in a loop.

```
n = ...
test_list = list(range(n))
for index in range(0, n):
  test_list[index] = index
```

We execute this code snippet for different values of n with different instrumentation mechanisms: `inspect` refers to the instrumentation mechanism described in Sect. 4.3.1, `inspect-loc` refers to the instrumentation mechanism with detailed source location tracking enabled, `legacy` refers to the instrumentation used in previous versions of `mlinspect` [17], and `no-inst` refers to the execution of the code without any instrumentation. In this experiment, we exclude the time it takes the library `gorilla` to apply the monkey patches and remove them again after execution of the instrumented user code. This constant cost only needs to be paid once per script and it is independent of the user code. In our measurements, this one-time cost was lower than 7ms.

Figure 8d shows the corresponding execution times. We find that the patch-based instrumentation approach is more than an order of magnitude faster than the earlier `legacy` approach. We also find that the instrumentation overhead diminishes for large values of n, where `inspect` exhibits less than twice the runtime of the uninstrumented execution `no-inst` as soon as the number of loop repetitions is 10,000 or higher. Furthermore, we find that `inspect-loc`, which tracks the exact source code locations (e.g., not only the line number but the character offsets in the line), introduces an overhead proportional to the overhead of `inspect`. Note that these experiments show an extreme worst-case scenario and that code location tracking is optional.

In summary, we find that instrumentation based on monkey patching drastically reduces AST instrumentation overhead.

5.2 Exploratory interview study with experts

We conduct an exploratory interview study with six expert users to qualitatively evaluate `mlinspect` in an ML pipeline debugging task. We provide the materials used in the study[9].

Participants Six participants solicited from our professional networks were interviewed. All participants have several years of experience in domains like data science, data engineering, and algorithmic fairness. The group consists of an expert data scientist from a large European retail company, a research engineer who previously worked on data science topics at an NLP-focused startup, three PhD students in machine learning and data management, and a data science Masters student.

Methodology We first give a fifteen-minute presentation about `mlinspect`, focused on data distribution bugs, to the participants. Next, a demonstration of `mlinspect` was given for ten-to-fifteen minutes, showing the detection of a data distribution bug in an example pipeline. Participants were allowed to ask questions. After this introduction, participants were instructed to individually solve two tasks similar to the demonstration. The first task uses a pipeline on a dataset about recidivism [6] with two artificial data distribution bugs caused by filter operations, which participants

[9] https://github.com/stefan-grafberger/mlinspect-exploratory-user-study/tree/b9546a7ff675af95811d3fe0c517093eb184e8d2.

had to identify. In the second task, a synthetic dataset from the healthcare domain and a pipeline with one data distribution bug were used. The goal of this setting was to find out whether `mlinspect` helps participants to quickly discover data distribution bugs and understand their root cause in complex pipelines with multiple operations, all potentially affecting the data distribution. Once participants completed the tasks, we studied their solutions, asked them a predefined set of questions about their experience with the library and about the technical aspects of its application, and also gathered their unstructured verbal feedback.

Results We briefly summarize the results from the tasks and interview questions.

Feasibility of the tasks All participants were successfully able to perform the two tasks within half an hour, despite not having any previous experience with the library. Most participants solved the second task much faster than the first one, after getting more familiar with the library. One participant stated that she spent most of the time on understanding the task pipeline, not on the usage of `mlinspect`.

Effectiveness for debugging All participants stated during the user interview that they could complete the tasks using `mlinspect` effectively. None of the participants were aware of alternative libraries to `mlinspect` for debugging ML pipelines. When asked how they would handle the tasks without `mlinspect`, all participants stated that they would repeatedly adjust the code to compute histograms of intermediate results and analyze the distribution changes manually. Based on their professional experience, all of them estimated that the alternative approach would have been more time-intensive, tedious, and error-prone than using `mlinspect`.

We highlight one quote from a participant: *The tool [...] can detect bias to the precision of which operator. That is quite impressive. [...] The DAG representation is powerful.*

Real-world applicability All participants thought that `mlinspect` is useful for data scientists; one participant commented that PySpark support is required to work with larger datasets. All but one participant stated that they would use `mlinspect` again when encountering an applicable problem. The remaining participant said they would only use our library again if it included additional functionality for model debugging.

Feature requests Participants named features they would like to see added to `mlinspect`, such as support for PySpark and support for detecting intersectional data distribution bugs. Another suggested feature was the detection of bias that is gradually amplified by multiple operators. The current implementation will not detect an issue if all operator changes are under the detection threshold, despite the overall change being over the threshold. Four users stated that they would have liked a final report by `mlinspect` that directly summarizes all potential issues, and includes detailed information about the issues that triggered alerts. One of the participants wanted `mlinspect` to integrate the detection of data quality issues like duplicate rows. Another suggestion was to test the initial input distribution and not just detect whether user code introduces new issues or amplifies existing issues. We note that the modular design of `mlinspect` allows for the implementation of all of the suggested features in future work. Indeed, we were able to already build an inspection for intersectional group memberships in response to a feature request.

In summary, participants confirmed the need to simplify data distribution debugging and found `mlinspect` helpful and usable.

5.3 Qualitative comparison against experiment tracking and workflow provenance tools

We are not aware of any system that offers the functionality `mlinspect` provides. As a consequence, we compare it against two systems from adjacent use cases: MLFlow[10] and noWorkflow [40]. MLFlow is an open-source experiment tracking solution with a rich feature set; noWorkflow is an open-source workflow provenance system that can handle unmodified programs. We qualitatively evaluate these tools for detecting the issues outlined in our example pipeline from Sect. 2.

5.3.1 MLFlow

MLFlow offers two different ways to log experiment data: (i) users can manually add logging statements to their code to track events and parameters of experiments with statements like `create_experiment()`, `start_run()`, `log_param()`, `log_metric()`, and `log_ar-tifact()`; (ii) the tool offers an auto-logging API, which is still in an experimental state, to log certain parameters and metrics for libraries like scikit-learn and Tensorflow. Auto-logging is implemented by patching all `fit` methods of all estimators. To enable auto-logging, users only need to add a single function call to the beginning of the pipeline, `mlflow.sklearn.autolog()`. MLFlow then logs data like sampled input rows from `the train_data` used as input to `pipeline.fit`, the parameters of all nested estimators, the training score, as well as strings describing the applied transformers. During execution, MLFlow saves all of the captured data to a directory. Afterward, a UI can be started with the command `mlflow ui` in the browser. There, users can get an overview of past runs and experiments and see a summary of important information, including certain metrics. There is also a detailed view for runs. Based on the information presented in the UI, it is easy for users to find a particular version of the experiment code, deploy the trained

10 https://github.com/mlflow/mlflow.

model from that run, and obtain a file containing the initial column names and five example rows.

However, MLFLow does not capture intermediate versions of the data between different transformers in the pipeline. It also does not capture preprocessing operations in pandas. For discovering data distribution bugs like the shown in Fig. 1, users will still have to debug the pipeline on their own; the only help they would get from MLFLow would be artifact logging, to save CSV-versions of dataframes. To add detailed logging to scikit-learn pipelines, users still have to modify the pipeline code, for example, by adding transformers with the sole purpose of logging the data flowing through them[11].

Revisiting our running example in Fig. 1: we could detect issue ① with the help of straightforward artifact logging to the pandas part of the code. However, we would still need to directly load the CSV-files created by MLFlow and manually compute histograms. We could also deal with issue ② and ③ in a similar way, but we would have to build a custom mechanism to track group membership through the selection. For detecting issues ④ and ⑥, we would have to implement scikit-learn debug transformers using CSV logging provided by MLFlow. For issue ⑤, we would have to manually inspect the code to discover columns used as features.

In summary, we find that MLFlow is designed for recording experiment metadata, but it does not provide strong support for debugging data-related issues in the user code. Using MLFlow does not make it significantly more convenient to identify data distribution bugs in our running example. However, for other use cases, the auto-logging approach is very convenient.

5.3.2 noWorkflow

The noWorkflow[12] tool runs unmodified Python files, collects provenance information, and optionally other information such as variable usage and dependencies. It allows users to browse the data of past executions and investigate details such as module dependencies, function activations, and file accesses. Furthermore, it can generate a dataflow graph with fine-grained provenance data for the function call graph. (Figure 9 shows this call graph for our example pipeline.)

How can noWorkflow help us detect the data distribution bugs outlined in Sect. 2? We can list all function activations, including their parameters and return values. For these captured function calls, noworkflow stores and can display all intermediate dataframes and tensors passed around. We could use this to detect issue ①, but we would have to implement custom code to compute histograms of the data before and after the join. Issues ② and ③ are more problematic: once the projection removes important columns, the intermediate results stored by noWorkflow will not help us anymore; we would have to write custom debugging code to trace the group membership attributes. For detecting issues ④ and ⑥, noWorkflow provides no help. Unfortunately, the tool only captures function calls related to user-defined functions. Because of this, noWorkflow cannot capture the intermediate data of nested scikit-learn pipelines: a `pipeline.fit` call lead to many `.fit` calls on child transformers. These indirect calls are not captured. To detect issue ⑤, we would also have to identify it manually, by looking at the code.

In addition to not providing the required support for detecting these issues, noWorkflow also slows down the pipeline's execution: its execution time for our example pipeline is about an order of magnitude longer than `mlinspect`'s execution time. For its detailed tracking, noWorkflow saves all inputs and outputs of captured function calls to disk, leading to a considerable overhead compared to `mlinspect`, which only stores histograms and group membership information in-memory. Overall, modifying the pipeline code directly instead of using noWorkflow would likely be easier for data distribution debugging. This is because working with the original pipeline code is more straightforward in this case than implementing custom code that uses the data captured by noWorkflow.

Internally, noWorkflows captures function calls via the Python profiling API[13], where it registers itself as a listener. During pipeline execution, the Python profiler informs noWorkflow of all function activations. However, even a simple test script provided by the noWorklow authors leads to 156,086 function activations [33]. This is because the profiling API itself also considered function activations that were called indirectly. To avoid overloading users with large volumes of information (and likely to avoid performance problems), the authors decided to let noWorkflow only register function activations related to user-defined functions. This decision, in turn, leads to noWorkflow ignoring indirect scikit-learn calls.

In summary, we find that noWorkflow is designed for provenance tracking at a lower level (function calls) than `mlinspect`, and, as a consequence, it does not appropriately capture the semantics of relational and ML operations in the code, which greatly reduces its utility as a data distribution debugger, the issue of the interest of our work.

[11] https://stackoverflow.com/questions/34802465/sklearn-is-there-any-way-to-debug-pipelines.

[12] https://github.com/gems-uff/noworkflow.

[13] https://github.com/gems-uff/noworkflow/blob/cbb8964eba7d58a5e87f96fb5bb91ac452b80763/capture/noworkflow/now/collection/prov_execution/profiler.py.

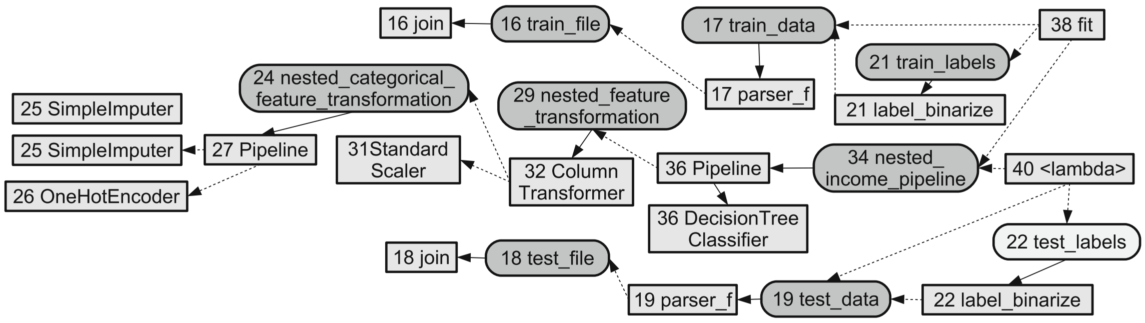


Fig. 9 Simplified illustration of the call graph for our example pipeline produced by noWorkflow. Unfortunately, it is difficult to understand the dataflow of ML pipelines using pandas and scikit-learn. The graph directly reflects each function call in the user code and does not provide an abstract representation of the dataflow of the ML components

6 Related work

The challenges of data management for end-to-end ML pipelines [41] and the Python-based data science ecosystem [44,45] are coming into the focus of the data management community in recent years. Proposed approaches often borrow ideas from provenance for relational workloads, a well-studied subject [13].

6.1 Provenance for relational workloads

There have been different notions of provenance for relational workloads, and there are several surveys of the field [13,19]. In the rest of this section, we will highlight a few important notions and use examples and explanations, mainly taken from the survey by Cheney et al. [13]. For more information, we refer to that survey and other papers cited in this section.

Provenance information is sometimes also called *lineage*. Three forms of provenance we want to discuss here briefly are why-, how-, and where-provenance. However, a lot of existing work does not fall into one of these categories. The idea behind why-provenance is to collect a set of all *witness* tuples that contributed to the existence of a tuple in the output of a query. However, for example, when the *distinct* keyword is used in a query, multiple tuples can result in the same output tuple while not needing to coexist. In contrast, the results of a natural join require multiple tuples to coexist. Why-provenance does not capture these distinctions as precisely as necessary for some purposes. Further, because the number of witnesses for each output tuple can be exponential in the size of the input database, the focus is usually on subsets of witnesses.

The precision issues mentioned just now are addressed by how-provenance, which aims to capture *how* a query output was derived. Important work in this area are provenance semirings [18]. The idea behind this is to use polynomials to capture how a query output was derived. Suppose two identical tuples t_1 and t_2 are present in a dataset, and we use the *distinct* keyword to only get one of the two in the result. In that case, we can represent the provenance information of the output tuple as $t_1 + t_2$: the existence of one of the two is enough to produce that output tuple. If we join t_1 and t_2, we can represent the output's provenance as $t_1 * t_2$, because both tuples need to coexist to produce that output. If a tuple can be the output of joining t_1 with either t_2 or t_3, then we can represent the provenance of the tuple as $t_1 * (t_2 + t_3)$. Extensions of this approach to aggregate queries [4] and linear algebra operators [57] also exist. In practice, however, it is not easy to use this approach due to performance reasons. It requires a lot of metadata to be captured, as the polynomial for one single output tuple can be arbitrarily complex depending on the query and the data.

Where-provenance captures the relationship between source and output locations. In a relation, the location refers to the cell. For example, where-provenance can capture that the $Smith$ cell in the tuple t_1: $(123, Jane, Smith)$ was copied from the *name* cell of some tuple t_2. However, where-provenance would not capture that t_1 is only present in the output because a join partner t_3 existed at some point during query execution.

There are many applications and implementations of the different notions of provenance. When using provenance in practice, paying attention to performance is crucial. Psallidas et al. [43], for example, present many tricks to implement provenance capturing efficiently. The authors implement core database operators with fine-grained lineage support baked-in. They list many optimization techniques that can be used when considering lineage support from the start.

6.2 Workflow provenance

There exist a large number of approaches for tracking provenance more broadly [31] and specifically in general data processing workflows [3,5,22,24,27,35,40,61]. However, none of these approaches can leverage the semantics of ML-specific operators such as the components of estimator/-transformer pipelines. NoWorkflow [40] is such an approach. It extracts provenance from function calls in Python scripts in three different levels: definition, deployment, and execution. It also uses the AST and extracts a dependency graph of the variables and directly handles unmodified programs. However, it considers functions as black boxes and does not capture fine-grained provenance inside called functions. Their system has many technical similarities with ours. However, their focus is on general Python scripts containing arbitrary functions. Because of that, they do not know of, e.g., the semantics of declarative pipeline operators and cannot track finer-grained lineage. For more information, we refer to Sect. 5.3.2. YesWorkflow [27] is a system that aims to bring the advantages of workflow analysis and modeling features to scripts written in languages like Python and R that define workflows. However, they heavily rely on users annotating their code. StarFlow is similar to YesWorkflow, but offers features like automatic parallelization [5]. It combines dynamic runtime analysis, static code analysis, and user annotations. It enables workflow abstraction, and it was implemented in the cloud. Lipstick [3] is a system that marries database-style and workflow-style provenance. While typical workflow provenance systems treat different modules as black box, they expose the functionality of modules using Pig Latin. This way, they can generate a detailed provenance graph with fine-grained provenance information. They use a provenance formalization that is based on the provenance semiring framework. Further, Inspector Gadget [35] is a framework for custom monitoring and debugging of distributed dataflows. They implemented it in Pig and called the implementation Penny. They exploit forward processing only, do not require dataflow engine modifications, and do not rely on injecting paint columns that may be observed by the operators. They allow users to insert monitoring agents that observe edges in the dataflow graph and propagate annotations through the execution. Their system is technically similar to our system in some aspects but does not consider ML-specific operators or applications. Titian [22] is another system using provenance to support users with debugging. It enables fine-grained data provenance capturing in Apache Spark. When implementing Spark support for our system in future work, the implementation described in their paper will likely be a great reference. Logothetis et al. [24] present Newt, a scalable architecture for capturing and using record-level data lineage to discover and resolve errors in analytics. As case studies, Newt is used to instrument two DISC systems, Hadoop and Hyracks. Zhang et al. [61] propose a system to capture lineage for distributed machine learning pipelines. Their focus is on how to efficiently encode the lineage information, especially in scenarios with image features. It records input and output datasets and cell-level mapping between the two. They do this by defining different mapping types for operators, e.g., a geometric mapping that can map regions of pixels to other regions of pixels. They built their system to support KeystoneML, which runs over Spark and HDFS. They expose this mapping interface to users, who need to decide which information they want to capture with it. Users can then ask provenance queries after executing the pipeline with lineage capturing. Not knowing the types of queries before pipeline execution requires a lot of metadata capturing, so they use these mapping types to reduce this overhead.

6.3 Experiment tracking and model management

Capturing high-level provenance, hyperparameters, and evaluation results is in the focus of model management systems such as ModelDB [53], mlflow [60], and ExperimentTracker [46], where the latter proposed the analysis of declarative abstractions like estimator/transformer pipelines. In contrast to our work, these systems only capture basic metadata and mainly require users to instrument their code with system-specific logging statements manually. ModelDB automatically tracks ML models in their native environment [53]. It tracks metadata about models and allows visual exploration of this metadata. To capture this metadata, it requires users to modify their script and add logging statements. ModelHub [29] focuses on deep neural networks and captures used parameters and hyperparameters like neural network weights across different versions of a model. It also logs information like loss values during the training of the model and performance metrics. Then, it allows users to query this captured information. In 2017, ExperimentTracker was proposed, a system for tracking metadata and provenance of ML experiments [46]. It tracks data provenance for SparkML and scikit-learn pipelines. For this, it also relies on abstractions like transformers and estimators. However, it relies on the user to expose certain data structures and integrate their code with their system's API. To our knowledge, this system was the first to use logical abstractions of SparkML and scikit-learn pipelines. ProvDB [30] stores metadata and some provenance information as well. It focuses on collaborative model development and offers a command-line interface for users to commit their changes. It uses a graph-model internally to store this provenance information. Node types in this graph are agents (e.g., team members or system components), activities (train, git commit, cron), and entities (project artifacts like files, datasets, and scripts). It then allows users to query this information. As a lot of information is being produced, they carefully consider how to store and efficiently query it. Overall, the tool requires users to organize their whole workflow around this system and use their command-line interface tools. Another system from 2018 is MLFlow that also aims to address challenges like experimentation and reproducibility [60]. They offer an API to support experiment tracking, reproducible runs, and model packaging and deployment. They again rely on users to provide additional metadata and integrate their pipelines with MLFlow. They then help in tasks like production deployment and reproducing, e.g., parameter settings of previous experiment runs. As MLFlow is currently one of the most successful tools in this area, we decided to try it out in practice and discovered that they recently added a still experimental option to log certain predefined metadata for libraries like scikit-learn automatically. For this, MLFlow requires users to add an auto-logging statement to their code. For more information, we refer to Sect. 5.3.1.

While systems like MLFlow rely on users to explicitly mark operations in their ML pipeline that should be saved in their metadata store, Ormenisan et al. [36] try to move from explicit provenance capturing to implicit provenance capturing. To achieve this, the authors rely on change capture APIs that capture events such as the usage or creation of files. In addition to this, they rely on file naming conventions and tagging of files. This way, they can capture the relation between different ML artifacts. However, only capturing events like the creation of files is not fine-grained enough for many use-cases. While these experiment-tracking tools mostly focus on particular experiments by particular teams, there also is the need to communicate information like how a particular dataset or model was created across different teams. For datasets, Gebru et al. [16,32] propose manually curated information in the form of *datasheets* and *model cards* to accompany them. The FAIR data principles [56] also propose guidelines to improve the findability, accessibility, interoperability, and reuse of digital assets but emphasize machine-actionability. Stoyanovich et al. [51] go one step further and propose nutritional labels for data and models, analogous to nutritional labels for the food industry. The goal is to provide simple, standard labels to evaluate the "fitness for use" of a model or dataset. The authors discuss these labels' desired properties and describe Ranking Facts [59], a system that can automatically derive labels for rankings.

6.4 Debugging for ML pipelines and data

Dagger [26] is a data-centric debugger that allows users to set data-breakpoints and store and query intermediate results from Python-based data pipelines. It requires users to mark code blocks in their Python pipelines, becoming nodes in their Dagger pipeline. It logs the data and provides its own query language for users to post queries through a command-line interface. Data breakpoints allow users to write assertions for the data between the different user-defined blocks. We see our system, `mlinspect`, as a complementary solution to Dagger: `mlinspect` can point users to hard-to-identify issues in their pipeline; Dagger will then enable them to drill-down and explore the data and identify the root causes of the problems. Vamsa [34] is a provenance-based analysis approach for data science scripts in Python that is technically close to ours. Like, `mlinspect`, Vamsa does not require changes to user code and uses a knowledge base about different ML libraries. However, Vamsa has a much narrower focus, as it only aims to identify which columns of the input contributed to a particular feature used for an ML model. Their system also aims to work for general Python code using various libraries and leverages the AST and intermediate representations. Vizier [9] is a notebook environment integrating Python, SQL, and data debugging and exploration techniques. It requires a tight integration into the user's development process and offers support for fine-grained provenance capture for SQL queries only.

Deequ [48] is another approach for the validation of ML data. It enables users to write "unit tests for data" using a declarative API. Breck et al. propose another data validation system [10]. It was integrated into TensorFlow Extended (TFX) to detect anomalies specifically in data fed into machine learning pipelines. However, these tools mostly focus on detecting data issues, not debugging them. There is also MISTIQUE, a system from 2018 to store and query model intermediates from ML pipelines and hidden representations from deep learning [54]. BugDoc [25] is a framework that implements and combines methods to select pipeline instances to try out to find root causes of problems in pipelines. However, it can only identify the root causes of problems related to the input parameter space, which has to be manually specified by the user.

There has been a large-scale study of the usage of different data science tools [44]. Many of their findings support our research direction, despite our restriction to specific libraries. Besides confirming assumptions that Python is by far the most used language for these types of problems, they also find that most data science code is linear and a mere orchestration of different libraries. This makes projects like ours feasible. They also confirm that most work relies on a handful of core libraries, such as scikit-learn, numpy, matplotlib and pandas. Another important finding is that in the last few years, declarative specification of data science logic is becoming increasingly common. Polyzotis et al. [41] wrote a survey of data lifecycle challenges in production ML. They identify data-related open challenges in areas such as data understanding, data validation and cleaning, and data preparation. An interesting tool inspired by various best practices in ML data preparation is DataLinter [21]. They propose data linting for deep neural networks, based on predefined linting rules applied to the training data and the outputs of the model, but they cannot inspect pipeline code. DeepXplore [38] is a system for automated white-box testing of ML models. It can find corner cases in application areas like self-driving cars.

They measure neuron coverage, which they describe as measuring the part of the neurons that are exercised in test inputs. Then they try to generate test cases that produce errors. Their test inputs can also be used to train the model to improve its performance.

6.5 Fairness-specific analysis of ML pipelines and predictions

In recent years, a set of specialized analysis tools with respect to the fairness and accountability of ML-based decision-making systems has been developed. Examples include SliceFinder [42], Coverage [7], and fairDags [58]. `mlinspect` provide a general runtime for implementing and integrating these and similar approaches into a common inspection platform. In our work on FairDags [58], we initially proposed extracting a DAG from ML pipelines to check for data distribution issues that result in bad model performance for sensitive demographic groups. Asudeh et al. [7] propose techniques to assess the coverage of a dataset over multiple categorical variables. The authors present an efficient strategy for traversing the combinatorial explosion of value combinations to identify problematic regions of the attribute space. Even with their optimized approach, the number of attributes to consider has a high impact on the performance. Slice finder [42] is a system to assist with finding slices of data an ML model performs particularly bad on. AI Fairness 360 (AIF360) [8] is a Python toolkit to calculate many fairness metrics and different algorithms to mitigate bias in datasets and models. Fairlearn [1] is another Python package to assess the fairness of AI systems and mitigate observed unfairness issues. Fairlearn also contains different mitigation algorithms and a Jupyter widget for model assessment.

Fairness issues in software are not just limited to issues specific to ML pipelines. Brun et al. [11] discuss how software engineering as a discipline needs to consider fairness from the start when building software systems (e.g., with fairness annotations like in Fairness-Aware Programming [2]), and Galhotra et al. [15] propose to test software for discrimination issues based on a schema of valid system inputs.

7 Conclusion and future work

We discussed several hard-to-identify data issues in ML pipelines that have the potential to impact correctness, reliability, and fairness. We proposed `mlinspect`, a library that enables lightweight lineage-based inspection of ML preprocessing pipelines. The `mlinspect` library extracts a directed acyclic graph representation of the dataflow from a pipeline and automatically instruments the code with predefined *inspections* based on a lightweight annotation propagation approach. We describe several custom inspections that data scientists can use to detect data distribution bugs in their pipelines. In contrast to existing work, `mlinspect` operates on declarative abstractions of popular data science libraries like estimator/transformer pipelines and does not require manual code instrumentation. We discuss the design and implementation of `mlinspect` and give a comprehensive end-to-end example that illustrates its functionality.

A future challenge is to assist data scientists in the analysis of the outputs of `mlinspect`. Complex pipelines can produce a variety of inspection results, and it may be helpful to explore anomaly detection techniques to point data scientists to potentially problematic cases or to suggest thresholds for checks. We also plan to incorporate additional backends for popular ML libraries into `mlinspect`, including Tensorflow Transform and Apache SparkML [28]. For these libraries, it will be challenging to find efficient ways to include inspections during the distributed execution of Beam and Spark operators. As discussed in Sect. 4.3.2, a future challenge is to support complex ML pipelines on high-dimensional tensors; it is still unclear whether such tensor operations are sufficiently captured by the dataframe algebra (Sect. 3.4) onto which `mlinspect` is built. As also outlined in Sect. 4.3.2, we intend to explore query optimization techniques for more efficient execution of inspections based on dataframe operations as a means to reduce the runtime overhead induced by Python.

Acknowledgements This work was supported in part by Ahold Delhaize, and by NSF Awards No. 1934464, 1916505 and 1922658. All contents represent the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

References

1. Agarwal, A., Beygelzimer, A., Dudik, M., Langford, J., Wallach, H.: A reductions approach to fair classification. In: FAT* (2017)
2. Albarghouthi, A., Vinitsky, S: Fairness-aware programming. In: FAT* (2019)
3. Amsterdamer, Y., Davidson, S.B., Deutch, D., Milo, T, Stoyanovich, J. Tannen, V. Enabling database-style workflow provenance. In: PVLDB, Putting Lipstick on Pig (2011)
4. Amsterdamer, Y., Deutch, D., Tannen, V: Provenance for aggregate queries. In: PODS (2011)
5. Angelino, E., Yamins, D., Seltzer, M.: Starflow: a script-centric data analysis environment. In: Provenance and Annotation of Data and Processes (2010)
6. Angwin, J., Larson, J., Mattu, S., Kirchner, L.: Machine bias. (propublica) (2016)
7. Asudeh, A., Jin, Z., Jagadish, H.V.: Assessing and remedying coverage for a given dataset. In: ICDE (2019)
8. Bellamy, R.K.E., Dey, K., Hind, M., Hoffman, S.C., Houde, S., et al.: AI fairness 360: an extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias (2018)
9. Brachmann, M., Bautista, C., Castelo, S., Feng, S., Freire, J., et al.: Data debugging and exploration with vizier. In: SIGMOD, Su Feng (2019)

10. Breck, E., Zinkevich, M., Whang, S., Roy, S.: Data validation for machine learning. In: SysML, Neoklis Polyzotis (2019)
11. Brun, Y., Meliou, A.: Software fairness. In: ESEC/FSE (2018)
12. Chen, I., Johansson, F.D., Sontag, D.: Why is my classifier discriminatory? In: NeurIPS (2018)
13. Cheney, J., Chiticariu, L., Tan, W.C: Provenance in Databases: Why, How, and Where. Found. Trends Databases, vol. 1, no. 4 (2009)
14. Chouldechova, A., Roth, A.: A snapshot of the frontiers of fairness in machine learning. In: CACM, vol 63, no. 5 (2020)
15. Galhotra, S., Brun, Y., Meliou, A: Testing software for discrimination. In: ESEC/FSE, Fairness Testing (2017)
16. Gebru, T., Morgenstern, J., Vecchione, B. et al.: Datasheets for datasets (2018)
17. Grafberger, S., Stoyanovich, J., Schelter, S.: Lightweight inspection of data preprocessing in native machine learning pipelines. In: Conference on Innovative Data Systems Research (CIDR) (2021)
18. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS (2007)
19. Herschel, M., Diestelkämper, R., Lahmar, H.B.: A survey on provenance: What for? What form? What from? VLDBJ **26**(6) (2017)
20. Hutton, G.: A tutorial on the universality and expressiveness of fold. J. Funct. Program, 8 (1999)
21. Hynes, N., Sculley, D., Terry, M. The data linter: lightweight, automated sanity checking for ml data sets. In: MLSystems workshop at NeurIPS (2017)
22. Interlandi, M., Shah, K., et al. Titian: data provenance support in spark. In: VLDB (2015)
23. Jindal, A., Emani, K.V., Daum, M., Poppe, O., et al: Magpie: python at speed and scale using cloud backends. In: CIDR (2021)
24. Logothetis, D., De, S., Yocum, K: Scalable lineage capture for debugging disc analytics. In: SoCC (2013)
25. Lourenço, R., Freire, J., Shasha, D.: A system for debugging computational pipelines. In: SIGMOD, Bugdoc (2020)
26. Madden, S., Ouzzani, M., Tang, N., Stonebraker, M.: Dagger: a data (not code) debugger. In: CIDR (2020)
27. McPhillips, T.M., Song, T., Kolisnik, T., et al.: Yesworkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. In: CoRR, abs/1502.02403 (2015)
28. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D.B., Amde, M., Owen, S., et al.: Mllib: machine learning in apache spark. JMLR **17**(1), 1235–1241 (2016)
29. Miao, H., Li, A., Davis, L.S., Deshpande, A.: Towards unified data and lifecycle management for deep learning. In: ICDE, pp. 571–582 (2017)
30. Miao, H., Deshpande, A.: Provdb: provenance-enabled lifecycle management of collaborative data analysis workflows. IEEE Data Eng. Bull **41** (2018)
31. Moreau, L.: The foundations for provenance on the web. Found. Trends Web Sci. **2**(2–3), 29, 99–241 (2010)
32. Mitchell, M., et al.: Model cards for model reporting. In: FAT* (2019)
33. Murta, L., Braganholo, V., Chirigati, F., Koop, D., Freire, J.: noworkflow: capturing and analyzing provenance of scripts. In: VLDB (2017)
34. Namaki, M.H., Floratou, A., Psallidas, F., Krishnan, S., Agrawal, A., Wu, Y.: Tracking provenance in data science scripts. In: KDD, Vamsa (2020)
35. Olston, C., Reed, B.: Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In: SIGMOD (2011)
36. Ormenisan, A.A., Meister, M., Buso, F., Andersson, R., Haridi, S., Dowling, J.: Time travel and provenance for machine learning pipelines. In: OpML at USENIX (2020)
37. Pedregosa, F., Varoquaux, G., Gramfort, A. et al.: Scikit-learn: Machine learning in python. In: JMLR, vol. 12 (2011)
38. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore. In: SOSP (2017)
39. Petersohn, D., Macke, S., Xin, D., Ma, W., Lee, D., Mo, X., Gonzalez, J.E., Hellerstein, J.M., Joseph, A.D., Parameswaran, A: Towards scalable dataframe systems. In: VLDB (2020)
40. Pimentel, J.F., Murta, L., Braganholo, V. and Freire, J.: noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. In: PVLDB (2017)
41. Polyzotis, N., Roy, S., Whang, S.E., Zinkevich, M.: Data lifecycle challenges in production machine learning: a survey. In: SIGMOD Record (2018)
42. Polyzotis, N., Whang, S., Kraska, T.K. and Chung, Y.: Automated data slicing for model validation. In: ICDE, Slice finder (2019)
43. Psallidas, F., Wu, E.: Smoke: Fine-grained lineage at interactive speed. In: VLDB (2018)
44. Psallidas, F., Zhu, Y., Karlas, B., et al: Data science through the looking glass and what we found there (2019)
45. Raasveldt, M., Mühleisen, H.: Data management for data science-towards embedded analytics. In: CIDR (2020)
46. Schelter, S., Boese, J.H., Kirschnick, J., Klein, T., Seufert, S.: Automatically tracking metadata and provenance of machine learning experiments. In: ML Systems Workshop at NeurIPS (2017)
47. Schelter, S., He, Y., Khilnani, J. and Stoyanovich, J.: Fairprep: Promoting data to a first-class citizen in studies on fairness-enhancing interventions. In: EDBT (2019)
48. Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., Grafberger, A: Automating large-scale data quality verification. In: PVLDB, Meltem Celikel (2018)
49. Sebastian, S.: Stoyanovich, J: Taming technical bias in machine learning pipelines. IEEE Data Eng. Bull. **43**, 39–50 (2020)
50. Sparks, E.R., Venkataraman, S., Kaftan, T., Franklin, M.J., Recht, B.: Keystoneml: Optimizing pipelines for large-scale advanced analytics. In: ICDE (2017)
51. Stoyanovich, J., Howe, B.: Nutritional labels for data and models. IEEE Data Eng. Bull. **42**(3), 13–23 (2019)
52. Stoyanovich, J., Howe, B., Jagadish, H.V.: Responsible data management. In: VLDB (2020)
53. Vartak, M., Madden, S.: Modeldb: opportunities and challenges in managing machine learning models. IEEE Data Eng. Bull. **41**(4), 16–25 (2018)
54. Vartak, M., Joana, Trindade, J.M., Madden, S., Zaharia, M: A system to store and query model intermediates for model diagnosis. In: SIGMOD (2018)
55. Wikipedia. Monkey patch. https://en.wikipedia.org/wiki/Monkey_patch (2021). Accessed 9 Sept 2021
56. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., et al.: The fair guiding principles for scientific data management and stewardship. Sci. Data **3**(1), 1–9 (2016)
57. Yan, Z., Tannen, V., Ives, Z.G.: Fine-grained provenance for linear algebra operators. In: TaPP (2016)
58. Yang, K., Huang, B., Stoyanovich, J., Schelter, S.: Fairness-aware instrumentation of preprocessing pipelines for machine learning. In: HILDA Workshop at SIGMOD (2020)
59. Yang, K., Stoyanovich, J., Asudeh, A., Howe, B., Jagadish, H.V., Miklau, G.: A nutritional label for rankings. In: SIGMOD (2018)
60. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S.A., Konwinski, A., Murching, S., et al.: Accelerating the machine learning lifecycle with MLflow. IEEE Data Eng. Bull. **41**(4), 39–45 (2018)
61. Zhang, Z., Sparks, E.R., Franklin, M.J.: Diagnosing machine learning pipelines with fine-grained lineage. In: HPDC (2017)