Lightweight Inspection of Data Preprocessing in Native Machine Learning Pipelines

Stefan Grafberger
TU Munich
stefangrafberger@gmail.com

Julia Stoyanovich
New York University
stoyanovich@nyu.edu

Sebastian Schelter
University of Amsterdam
s.schelter@uva.nl

ABSTRACT

Machine Learning (ML) is increasingly used to automate impactful decisions, and the risks arising from this wide-spread use are garnering attention from policy makers, scientists, and the media. ML applications are often very brittle with respect to their input data, which leads to concerns about their reliability, accountability, and fairness. In this paper we discuss such hard-to-identify data issues and describe `mlinspect`, a library that enables lightweight lineage-based inspection of ML preprocessing pipelines.

The key idea is to extract a directed acyclic graph representation of the dataflow from ML preprocessing pipelines in Python, and to use this representation to automatically instrument the code with predefined *inspections* based on a lightweight annotation propagation approach. 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` prototype, and give a complex end-to-end example that illustrates its functionality.

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. The risks and opportunities arising from the wide-spread use of predictive analytics are garnering much attention from policy makers, scientists, and the media [Stoyanovich et al. 2020].

The correctness and reliability of ML models critically depend on their training data. Pre-existing bias, such as under- or over-representation of particular groups in the training data [Chen et al. 2018], and technical bias, such as skew introduced during data preparation [Schelter et al. 2019], 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*.

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

Data distribution bugs are often introduced during preprocessing. Input data for ML applications often stems from various data sources, and has to be preprocessed and encoded as features first, which can introduce or amplify representation issues. For example, preprocessing operations that involve filters or joins can heavily change the distribution of different groups represented in the training data [Yang et al. 2020], and missing value imputation can also introduce skew [Schelter et al. 2019].

Recent ML fairness research, which mostly focuses on the use of learning algorithms on static datasets [Chouldechova and Roth 2020], is therefore insufficient because it cannot address such technical bias originating from the data preparation stage. Furthermore, we should detect and mitigate such bias as close to its source as possible.

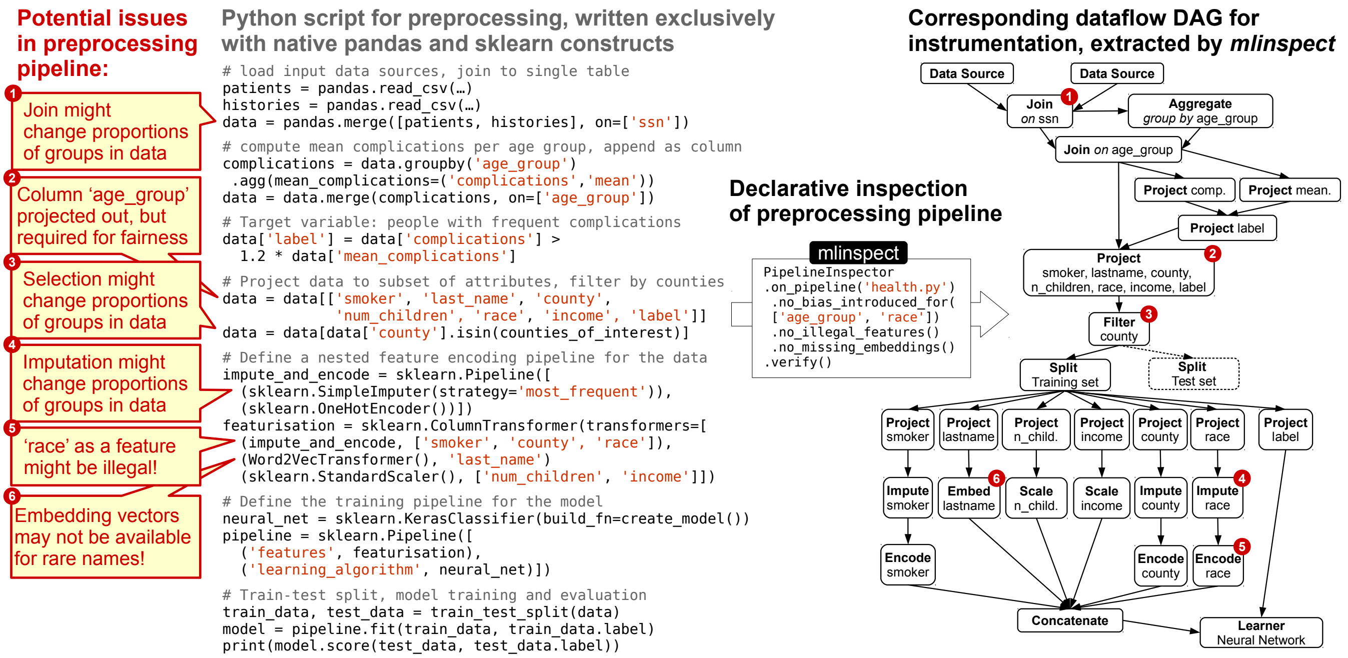
Data distribution bugs are difficult to catch. In part, this is because different pipeline steps are implemented using different libraries and abstractions, and the data representation often changes from relational data to matrices during data preparation. Further, preprocessing in the data science ecosystem [Psallidas et al. 2019] often combines relational operations on tabular data with *estimator/transformer pipelines*,[1] a composable and nestable abstraction for operations on array data, which originates from scikit-learn [Pedregosa et al. 2011] and has been adopted by popular libraries like SparkML [Meng et al. 2016] and Tensorflow Transform.[2]

In such cases, tracing problematic featurised entries back to the pipeline's initial human-readable input is tedious work. Finally, complex estimator/transformer pipelines are hard to inspect because they often result in nested function calls.

We need automated inspection of ML pipelines. Due to time pressure in their day-to-day activities, most data scientists will not spend the time and effort to manually instrument their code or insert logging statements for tracing as required by model management systems [Vartak and Madden 2018, Zaharia et al. 2018]. 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 [Madden et al. 2020] to drill down into the specific intermediate pipeline outputs and explore the root cause of the issue. We furthermore ar-

[1] https://scikit-learn.org/stable/modules/compose.html
[2] https://github.com/tensorflow/transform


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

```
# 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))
```

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

Figure 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).

gue 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 and helps to enforce best-practices.

The `mlinspect` library extracts logical query plans, modeled as directed acyclic graphs (DAGs) of preprocessing operators, from ML pipelines that use popular libraries like pandas and scikit-learn [Pedregosa et al. 2011], and that combine estimator/transformer pipelines and relational operators. These plans are then used to automatically instrument the code and 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` implements 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 us to integrate many issue detection techniques that previously required custom code, such as automated model validation on data slices [Polyzotis et al. 2019], the identification of distortions with respect to protected group membership in the training data [Yang et al. 2020], or automated sanity checking for ML datasets [Hynes et al. 2017].

In summary, we make the following contributions:

- We discuss hard-to-identify issues in ML preprocessing pipelines with respect to the fairness, transparency, and correctness of the resulting ML models (Sections 2 and 3.2).
- We describe the design of `mlinspect`, which enables lightweight lineage-based inspection of ML preprocessing pipelines. The `mlinspect` library bases its analysis on declarative abstractions of existing popular data science libraries, and does not require manual code instrumentation (Section 3).
- We provide a prototype implementation of `mlinspect` at https://github.com/stefan-grafberger/mlinspect.

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 Figure 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 ML model works equally well for patients across different age groups and races. This obligation is operationalized as an intersectional fairness criterion, requiring equal false negatives 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 ❶) if a large percentage of the records of 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. The data is then projected to a subset of the attributes, to be used by the classification model. This leads to the second issue ❷ 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 ❸: 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` is replaced with a corresponding vector from a pretrained word embedding, and the numerical attributes `num_children` and `income` are normalized.

This feature encoding part of the pipeline introduces several potential issues: ❹ the imputation of missing values for the categorical attributes potentially introduces statistical bias, as it might attribute records with a missing value in the race attribute to the majority race in the dataset; ❺ depending on the legal context (i.e., if the disparate treatment doctrine is enforced), it might be forbidden to use `race` as an input to the classifier; ❻ 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. This motivates the development of automatic inspection libraries such as `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 usually not built on top of an algebraic abstraction. Further, these pipelines do not only operate on relational data but also on tensors, when converting the input data to feature matrices. However, popular data science libraries expose a set of 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[3] rely on potentially nested estimator/transformer chains. We therefore focus on data science scripts written using only existing library code (e.g., we do not require manual code instrumentation), but restrict ourselves to code applying combinations of SQL-like operations on dataframes with estimator/transformer pipelines, analogous to our example in Section 2. This covers 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, more than 25% use scikit-learn, and more than 80% of the cell-level code contains little to no control flow [Psallidas et al. 2019].

We propose `mlinspect`, a runtime for lightweight, lineage-based inspection of such preprocessing pipelines, based on the original Python code of the data science script. `mlinspect` extracts a directed, acyclic graph (DAG) representing the dataflow from ML pipelines in Python (using libraries like pandas and scikit-learn) with logical operators like join, selection, projection, column encoders, and missing value imputation. On top of this extraction, `mlinspect` enables the automatic instrumentation of the code with predefined lightweight *inspections* that detect issues in the pipeline and give hints to users. In the following, we discuss our proposed approach. We provide details on the status of our prototype implementation in Section 5.

3.1 Instrumentation and DAG Representation

Data preparation pipelines that use common declarative abstractions such as pandas data slicing, scikit-learn's `ColumnTransformer` and pipelines or SparkML pipelines have a natural directed acyclic graph (DAG) representation [Schelter et al. 2017]. In our case, the data sources in this DAG are typically comprised of tables or files holding relational data. The data flowing through the DAG are 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 outputting vectors), or standard ML preprocessing operations like normalization or concatenation (consuming vectors and producing vectors).

Intermediate representation. In order to support arbitrary Python scripts written with pandas operations and estimator/transformer pipelines, we do not parse the code ourselves, but rely on the AST created by the Python parser itself. However, we cannot directly use the AST, as it does not contain edges between variable definitions and usage, but only contains load and store nodes that provide additional context for variable name nodes. Therefore, we first transform the AST to an intermediate representation (IR), which we later transform into our dataflow DAG. The main difference of our IR to the AST is that we create edges between variable definitions and their usage. To efficiently create the IR, we maintain two mappings while transforming the AST to the IR, one for variable names to IR nodes and another for AST nodes to IR nodes. This allows us to only have to visit each AST node in the AST once for the transformation to the IR.

Instrumentation and DAG extraction at runtime. We extract the DAG for the preprocessing pipeline at runtime during a single execution of the pipeline, and conduct all of the instrumentation necessary for inspection beforehand. At runtime, we use Python's `inspect` module to retrieve the function name and module origin of each function

[3] 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

call and subscript operation (e.g., a `df['column']` function call). We then build up the DAG in two steps: (*i*) We visit each node once and remove all nodes that are not subscripts or function calls. When deleting them, we preserve previous transitive relationships between parents and children by adding the corresponding edges to the IR; (*ii*) Next, we check each node that survived this filter. For each of these call nodes and subscripts, we extract its module information. We determine whether the function call or subscript operation has a corresponding DAG operation, and map it accordingly (e.g., a `('pandas.core.frame', 'merge')` to a join operator).

Lineage-based annotation propagation. The main idea of our lineage-based annotation propagation machinery is to offer a simple, library-independent interface to propagate annotations such as the lineage of tuples across operators from different libraries. We model this with so-called *inspections*. Each inspection retains a fixed-size state that gets 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. It annotates the output tuples, and can optionally annotate the logical operator in the DAG with the computed result, such as a histogram of the outputs.

```
# Abstract base class for all inspections
class Inspection(metaclass=abc.ABCMeta):

  # 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 us to only materialize the state that is required for the actual inspections configured by the user (and to avoid materializing arbitrary information from the pipeline). We use an iterator-to-iterator approach for inspections and chain the computations of all specified inspections immediately after each instrumented DAG operator.

As long as each row annotation by each inspection has a fixed size limit and each inspection only uses a fixed-size state, the overhead of our framework is constant per inspected tuple. Our 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 ensure that the input rows of an operator are correctly mapped to its output rows, and then expose these output rows along with their corresponding annotated inputs to each inspection. This input/output mapping is internally conducted differently depending on the operator semantics.

Operators like projection and transformers are guaranteed to have the same order and the same number of input and output elements, so we do not need to do anything there. For operators like selection, join, and train-test split, we maintain the mapping by generating an identifier column, which we transparently push through the operator (and remove immediately afterwards to hide it from the user code).

Note that for performance reasons, we only track one possible source for operators like deduplications and not all possible sources, as the performance overhead for detailed tracking (e.g., the full semiring provenance framework [Green et al. 2007]) would be too large.

Function call capturing. To allow inspections to access the output of an operator such as a join (and the corresponding input rows and their annotations), we must efficiently capture arguments and return values of function calls and subscripts. For this, we modify the AST from the Python parser with an `ast.NodeTransformer` before compiling and executing it. Our NodeTransformer visits call and subscript AST nodes. It wraps the AST node in a function call that captures and returns the output from the original function call node. We apply the same operation for the object a function is called from, if there is one, and for the arguments and keyword arguments of the function. As an example, the code `obj.a_func("arg0", "arg1", my_arg="arg2")` is transformed to `after_call(before_call_value(obj).a_func( *before_call_args("arg0", "arg1"), **before_call_kwargs( my_arg="arg2")))`.

Backends for popular Python libraries. `mlinspect` is designed to understand the semantics of preprocessing operations of popular Python frameworks from the data science space like scikit-learn and pandas. The instrumentation based on captured function calls described so far is independent of the specific library. As stated before, not all function calls that we need to capture are visible in the AST (e.g., the fit/transform calls of transformers contained in scikit-learn pipeline objects). We therefore introduce library-specific backends in `mlinspect` that understand the semantics of popular libraries like scikit-learn. `mlinspect` delegates the captured function calls to the library-specific backend based on module information. For example, during the invocation of the `after_call` function, our library uses the Python inspection module to find out where the wrapped function comes from, and delegates the call to the corresponding backend.

Execution of inspections. Each backend is responsible for hiding library implementation details from the inspections. The pandas backend, for example, is responsible to call 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. Afterwards, the backend stores the resulting new annotations created by the inspection in an efficient manner that still hides it from the user, (e.g., as attributes of the processed dataframe in the case of pandas). We refer to our backend implementations for details.[4]

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

[4] https://github.com/stefan-grafberger/mlinspect/tree/19ca0d6ae8672249891835190c9e2d9d3c14f28f/mlinspect/backends

3.2 Automatic Inspections and Checks

Inspections serve as our basis for detecting issues in ML pipelines. But they only annotate the extracted DAG with information like computed histograms for different DAG nodes. On top of the extracted and annotated DAG, we provide *checks*, a rule-based approach to verify constraints on the DAG, for example by comparing the change in the histograms to a threshold.

Before execution, `mlinspect` checks which inspections are required by the checks specified by the user. It then instruments the pipeline and executes it using a minimal set of inspections either required by the checks or 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.

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.

Fairness and accountability. In recent years, more and more problems with respect to the fairness and accountability of ML-based decision making systems are uncovered [Stoyanovich et al. 2020]. Such problems are often non-obvious for data scientists without appropriate training and are therefore in the focus of `mlinspect`.

As discussed in our example from Section 2 and outlined in previous work [Yang et al. 2020], operations like join and selection can accidentally filter out records from protected groups and thereby *introduce or amplify under-representation issues*. `mlinspect` provides an inspection that computes histograms of operator outputs based on the protected groups, and alerts the user if the proportions of these groups change drastically after an operator. A related problem is the missing coverage for certain protected groups in the data, identified by certain combinations of attributes [Asudeh et al. 2019]. For that, we need to forward-propagate annotations identifying the groups of interest and materialize the annotated input and final outputs of the complete pipeline.

Furthermore, there are *legal restrictions on the usage of demographic features* such as gender for automated decision making. We can check the operator DAG based on a list of sensitive column name candidates, and alert the user about the places in the code where such an illegal feature is potentially used.

ML models may also *perform particularly bad for specific demographic (sub-)groups* in the data (e.g., higher false positive rates for recidivism predictions about black people [Angwin et al. 2016]). The identification of such groups is in the focus of recent research [Polyzotis et al. 2019]. 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. `mlinspect` allows us to define an inspection that forward-propagates the sensitive column annotations and then materializes the minimum amount of information needed for conducting tests on the performance for different groups: rows only containing the predicted label and the sensitive columns.

Methodology. Additionally, there are lot of methodological errors that unexperienced data scientists might accidentally make, 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 regularisation), or selecting hyperparameters on the test set instead of a validation set. All of these issues can be identified by analyzing our extracted operator DAG.

Performance. In a similar vein, there might be common performance issues in the relational operations applied by the pipeline such as *selections or projections applied after instead of before a join*, which can be identified from the operator graph. Another example here might be *operations that cancel themselves out* and should be removed; and example is up-sampling followed by a duplicate elimination.

Robustness. Furthermore, there might be robustness issues in the pipeline; for example, some scikit-learn transformers cannot handle null values. We can identify such cases from the operator graph, and recommend the user to apply a simple imputation technique. Another problem that can be detected by analysing histograms of operator outputs are *class imbalance issues.* We could analyse the DAG to see if the data scientist already address these with resampling or reweighing and alert her otherwise.

4. RELATED WORK

The challenges of data management for end-to-end ML pipelines [Polyzotis et al. 2018] and the Python-based data science ecosystem [Psallidas et al. 2019, Raasveldt and Mühleisen 2020] 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 [Cheney et al. 2009].

Experiment tracking and model management. Capturing high-level provenance, hyperparameters and evaluation results is in the focus of model management systems such as ModelDB [Vartak and Madden 2018], mlflow [Zaharia et al. 2018], and ExperimentTracker [Schelter et al. 2017], 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 require users to manually instrument their code with system-specific logging statements.

Debugging for ML pipelines and data. Dagger [Madden et al. 2020] is a data-centric debugger that allows users to set data-breakpoints, and store and query intermediate results from Python-based data pipelines. We see `mlinspect` as a complementary solution to Dagger: `mlinspect` can point users to hard-to-identify issues in their pipeline; Dagger will then enables them to drill-down and explore the data and identify the root causes of the issues. Vamsa [Namaki et al. 2020] 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. Vizier [Brachmann et al. 2019] 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. Additional approaches for the validation of ML data are Deequ [Schelter et al. 2018], which enables "unit tests for data", and Mistique [Vartak et al. 2018], a system to store and query intermediates from deep learning models.

Workflow provenance. There exists a large number of approaches for tracking provenance in general data processing workflows [Pimentel et al. 2017, Amsterdamer et al. 2011, Olston and Reed 2011]. However, none of these approaches can leverage the semantics of ML-specific operators such as the components of estimator/transformer pipelines.

Fairness-specific analysis of ML pipelines and predictions. In recent years, a set of specialised analysis tools with respect to the fairness and accountability of ML-based decision making systems has been developed. Examples include SliceFinder [Polyzotis et al. 2019], Coverage [Asudeh et al. 2019], and fairDags [Yang et al. 2020]. `mlinspect` provides a general runtime for implementing and integrating these and similar approaches into a common inspection platform.

5. STATUS OF OUR PROTOTYPE

We provide a prototypical implementation of our proposed approach at https://github.com/stefan-grafberger/mlinspect. In the following, we discuss implementation aspects, revisit our example and present preliminary experiments on measuring the runtime overhead.

5.1 Overview

We implemented the core extraction functionality for the IR and DAG, together with the instrumentation, and initial versions of the backends for pandas and scikit-learn. The instrumentation of complex scripts like our healthcare example from Section 2 already works, but we do not comprehensively cover all API functions yet. We already offer implementations of representative inspections[5], 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, and an inspection that computes histograms of operator outputs for sensitive groups. We also offer implementations of so-called "checks"[6], which evaluate a constraint on the outputs of our inspections, e.g., a threshold comparison of the magnitude of change in the proportions of certain groups in the data after a filter.

5.2 Running Example

We provide an executable implementation of our example[7] from Section 2, along with a jupyter notebook[8] that details and visualises 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()
```

We discuss the inspections required for our example in the following. The expectation about the lack of the introduction of technical bias refers to the issues ❶,❷,❸ and ❹ from our example, and requires the aforementioned histogram inspection 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 ❷).

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 afterwards. To check whether illegal features have been used (issue ❺), 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 ❻) only requires counting the null values in the outputs of the embedding operator.

5.3 Runtime Overhead

We have not yet focused on optimising the performance of our implementation. However, our design requires us to only conduct a single scan over operator inputs and outputs, and to only materialize intermediate results of interest, which requires only a constant overhead per processed row for our discussed inspections. Note that the running time of most end-to-end ML pipelines may, in most cases, be dominated by model training and not by data preprocessing, especially if deep neural networks are used.

We present a set of preliminary experiments to measure the runtime overhead of our `mlinspect` prototype. We expect the absolute overhead to be high in comparison to the non-instrumented operator execution, as both pandas and scikit-learn internally execute operations based on optimised C implementations (for example from numpy). As `mlinspect` operates on the level of the Python script and allows for user-defined inspection functions, it naturally runs in Python, inheriting its overheads. As a consequence, our preliminary experiments focus on the overhead in terms of the number of input rows. We designed our approach with a constant overhead per tuple, and therefore expect the overhead to be linear in the number of rows. In future work, we intend to explore whether we can integrate the execution of our inspections with the execution of the actual library operators via just-in-time compilation techniques for data science libraries (e.g. provided by Weld [Palkar et al. 2017]).

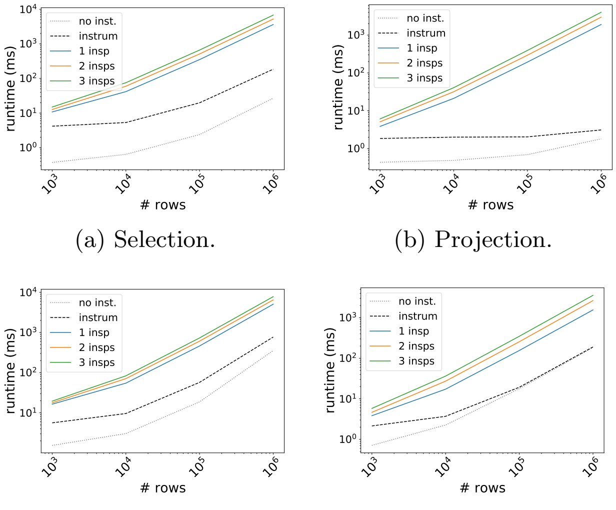
Instrumentation overhead. In our first experiment, we measure the runtime overhead of instrumenting different operators. In particular, we focus on the selection, projection and join operator of pandas, and the ML-specific one-hot-encoder operator from scikit-learn, which turns a categorical string column into a sparse matrix representation. For each operator, we measure the execution time without instrumentation, instrumentation without inspections (in order to see the overhead of our instrumentation approach with callback functions), as well as one to three "empty" inspections that just read the respective inputs and outputs of operators, but do not propagate annotations.

[5] https://github.com/stefan-grafberger/mlinspect/tree/19ca0d6ae8672249891835190c9e2d9d3c14f28f/mlinspect/inspections

[6] https://github.com/stefan-grafberger/mlinspect/tree/19ca0d6ae8672249891835190c9e2d9d3c14f28f/mlinspect/checks

[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



(a) Selection.

(b) Projection.

(c) Join.

(d) One-Hot-Encoder.

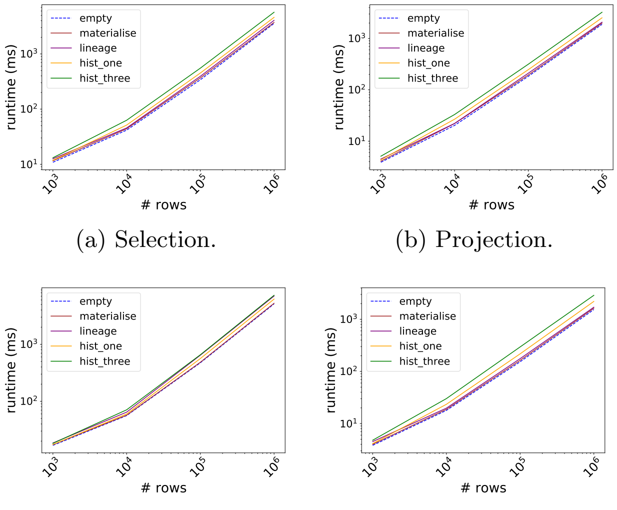
Figure 2: Instrumentation overhead for different operators. We compare the runtime of the execution of a given operator with no instrumentation (`no inst`), instrumentation without inspections (`instrum`), and with one to three empty inspections. We find that the overhead is linear in the number of rows to process.

We report the average runtime from 20 repetitions of the experiment for 1,000, 10,000, 100,000 and 1,000,000 input rows. The results are shown in Figure 5.3. We observe the expected increase in the absolute runtime stemming from our usage of Python. The overhead per tuple is constant however, as the runtime overhead grows linearly with the number of input rows for all operators. The only exception is the projection operator, which is a special case, as the projection operation in pandas is nearly constant due to the underlying columnar data layout.

Inspection overhead. We repeat our experiment with the four previously chosen operators and measure the runtime overhead of our existing inspections. For each instrumented operator, we compare the execution time of the "empty" inspection (which has also been used in the previous experiment) to the execution time of the following inspections (each of which scans all processed rows):

- Materialise a sample of output rows for each operator
- Track the lineage via annotation propagation for a sample of output rows for each operator
- 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 1,000, 10,000, 100,000 and 1,000,000 input rows. The results are shown in Figure 5.3. We again observe an overhead for all inspections that is linear in the number of input 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.



(a) Selection.

(b) Projection.

(c) Join.

(d) One-Hot-Encoder.

Figure 3: 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 materialisation (`materialise`), 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 rows to process.

As a consequence, the overhead for running multiple inspections during one run is low. We achieve this with loop fusion techniques: We implement our inspections with generator-like iterators that yield their elements, and execute the inspections in a way that allows us to avoid multiple scans over the data by exposing each record to all inspections during a single scan over the data.

5.4 Next Steps

We are currently finishing the implementation of our prototype with a focus on the scikit-learn and pandas backends, and aim to implement all outlined inspections soon. A future challenge is to also 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 to them.

Furthermore, we aim to add support for custom functions and control flow like branches and loops, even though recent research indicates that a large fraction of data science scripts contain very little control flow [Psallidas et al. 2019]. We also plan to incorporate more backends for popular ML libraries into `mlinspect`, such as Tensorflow Transform and Apache SparkML [Meng et al. 2016]. For those it will be challenging to find efficient ways to include inspections during the distributed execution of Beam and Spark operators.

As outlined in Section 5.3, we intend to explore just-in-time compilation techniques for pandas and scikit-learn operators from Weld as a means to reduce the runtime overhead induced by Python.

Acknowledgements. The work of Julia Stoyanovich was supported in part by NSF Grants No. 1926250 and 1934464. Stefan Grafberger is a student of the Elite Graduate Program SWE at TU Munich, LMU Munich, and University of Augsburg, and and was supported by a IFI fellowship of the German Academic Exchange Service (DAAD). Further, the research was supported by Ahold Delhaize. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

6. REFERENCES

[Amsterdamer et al. 2011] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, et al. 2011. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*.

[Angwin et al. 2016] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. *Machine bias. (ProPublica)*. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing

[Asudeh et al. 2019] A. Asudeh, Z. Jin, and H. V. Jagadish. 2019. Assessing and Remedying Coverage for a Given Dataset. *ICDE*, 554–565.

[Brachmann et al. 2019] Mike Brachmann, Carlos Bautista, Sonia Castelo, et al. 2019. Data debugging and exploration with vizier. *SIGMOD*, 1877–1880.

[Chen et al. 2018] Irene Chen, Fredrik D Johansson, and David Sontag. 2018. Why Is My Classifier Discriminatory? *NeurIPS*, 3539–3550.

[Cheney et al. 2009] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. *Provenance in databases: Why, how, and where.* Now Publishers Inc.

[Chouldechova and Roth 2020] Alexandra Chouldechova and Aaron Roth. 2020. A snapshot of the frontiers of fairness in machine learning. *Commun. ACM* 63, 5, 82–89.

[Green et al. 2007] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Symposium on Principles of Database Systems*, ACM, 31–40.

[Hynes et al. 2017] Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data sets. *MLSystems workshop at NeurIPS*.

[Madden et al. 2020] Samuel Madden, Mourad Ouzzani, Nan Tang, and Michael Stonebraker. 2020. Dagger: A Data (not code) Debugger. *CIDR*.

[Meng et al. 2016] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1, 1235–1241.

[Namaki et al. 2020] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, and Yinghui Wu. 2020. Vamsa: Tracking Provenance in Data Science Scripts. *KDD*.

[Olston and Reed 2011] Christopher Olston and Benjamin Reed. 2011. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. *SIGMOD*, 1221–1224.

[Palkar et al. 2017] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. *CIDR*. 45.

[Pedregosa et al. 2011] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12, 2825–2830.

[Pimentel et al. 2017] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB* 10, 12.

[Polyzotis et al. 2018] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record* 47 , 17–28.

[Polyzotis et al. 2019] Neoklis Polyzotis, Steven Whang, Tim Klas Kraska, and Yeounoh Chung. 2019. Slice Finder: Automated Data Slicing for Model Validation. *ICDE*.

[Psallidas et al. 2019] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, et al. 2019. Data Science through the looking glass and what we found there. arXiv:1912.09536

[Raasveldt and Mühleisen 2020] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science-Towards Embedded Analytics. *CIDR* (2020).

[Schelter et al. 2017] Sebastian Schelter, Joos-Hendrik Böse, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. *ML Systems workshop at NeurIPS*.

[Schelter et al. 2019] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. 2019. FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions. *EDBT*.

[Schelter et al. 2018] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-scale Data Quality Verification. *PVLDB* 11, 12, 1781–1794.

[Stoyanovich et al. 2020] Julia Stoyanovich, Bill Howe, and H.V. Jagadish. 2020. Responsible Data Management. *VLDB* 13, 12, 3474–3489.

[Vartak and Madden 2018] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng.* 41, 16–25.

[Vartak et al. 2018] Manasi Vartak, Joana Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. *SIGMOD*, 1285–1300.

[Yang et al. 2020] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. 2020. Fairness-Aware Instrumentation of Preprocessing Pipelines for Machine Learning. *HILDA workshop at SIGMOD*.

[Zaharia et al. 2018] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4, 39–45.