MLINSPECT: a Data Distribution Debugger for Machine Learning Pipelines

Stefan Grafberger[†] Shubha Guha[†] Julia Stoyanovich[*] Sebastian Schelter[†]

[†]AIRLab, University of Amsterdam [*]New York University
{s.grafberger,s.guha,s.schelter}@uva.nl stoyanovich@nyu.edu

ABSTRACT

Machine Learning (ML) is increasingly used to automate impactful decisions, and the risks arising from this wide-spread use are garnering attention from policymakers, 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. While bias detection cannot be fully automated, computational tools can help pinpoint particular types of data issues.

We recently proposed `mlinspect`, a library that enables lightweight lineage-based inspection of ML preprocessing pipelines. In this demonstration, we show how `mlinspect` can be used to detect data distribution bugs in a representative pipeline. In contrast to existing work, `mlinspect` operates on declarative abstractions of popular data science libraries like estimator/transformer pipelines, can handle both relational and matrix data, and does not require manual code instrumentation. The library is publicly available at https://github.com/stefan-grafberger/mlinspect.

ACM Reference Format:
Stefan Grafberger[†] Shubha Guha[†] Julia Stoyanovich[*] Sebastian Schelter[†]. 2021. MLINSPECT: a Data Distribution Debugger for Machine Learning Pipelines. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3448016.3452759

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 policymakers, scientists, and the media [9].

Detecting data distribution bugs with `mlinspect`. 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 [2], and technical bias, such as skew introduced during data preparation [7, 8], can heavily impact performance. We refer to these problems collectively as *data distribution bugs*. In this demonstration, our goal is to help data scientists diagnose and mitigate data distribution bugs that arise during preprocessing steps in an ML pipeline, and contribute to technical bias. Such issues are often neglected in fair-ML research, which mostly focuses on learning algorithms applied to static datasets [3].

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 often combines relational operations on tabular data with estimator/transformer pipelines [5], a composable and nestable abstraction for operations on array data, which originates from scikit-learn and has been adopted by popular libraries like SparkML and Tensorflow Transform. Tracing problematic featurised entries back to the pipeline's initial human-readable input is tedious work.

Due to the time pressures in their day-to-day activities, most data scientists will not make an effort to instrument their code manually as required by model management systems [10, 11], and to debug it regularly for data-related issues. In response to this problem, we proposed `mlinspect` [4], a library that offers automated inspection for *code natively written with popular data science libraries* like scikit-learn and pandas. `mlinspect` identifies data distribution bugs in Python-based ML pipelines that combine relational operations on tabular data and estimator/transformer pipelines for feature encoding on matrix data. Our library extracts logical query plans, modeled as directed acyclic graphs (DAGs) of preprocessing operators, and automatically instruments these operators at runtime to detect data distribution bugs.

Demonstration scenario. We provide a web-based interface where attendees can edit ML pipeline code for a representative healthcare use case. They can visualize the pipeline and view intermediate operator outputs based on `mlinspect`'s abstract representation of the pipeline. Attendees will detect data distribution bugs by examining histograms of operator outputs, and rewrite the code interactively to fix the highlighted issues. In summary:

- We showcase the automatic detection of data distribution bugs in native ML pipelines.
- We show how to quickly examine the effects of different data transformations using our inspection techniques.
- We demonstrate how data scientists can interactively use `mlinspect` to find and fix data distribution bugs.

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452759

The `mlinspect` library is publicly available at https://github.com/stefan-grafberger/mlinspect. We are already using the library to teach data science students about the importance of data preprocessing, making data distribution debugging part of their methodological toolkit (see https://dataresponsibly.github.io/rds).

2 OVERVIEW OF MLINSPECT

In the following, we provide a brief overview of `mlinspect` [4], our recently proposed library for the lightweight lineage-based inspection of ML preprocessing pipelines implemented in Python.

Core ideas. `mlinspect` 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. We support code that combines relational operations on dataframes and estimator/transformer pipelines on matrix data for feature encoding. This DAG is used to automatically instrument the code and trace the impact of operators on properties like the distributions of sensitive groups in the data. In this way, `mlinspect` empowers data scientists to automatically check their ML pipeline code for data distribution bugs without requiring changes to the code. Importantly, `mlinspect` implements a library-independent interface to propagate annotations such as tuple lineage 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.

Figure 1 gives an example of this: `mlinspect` identifies a selection operation in the Python code that potentially changes the age distribution in the data. It instruments the operator and computes the distribution change between the inputs and the outputs of the operator, and warns the data scientist if the change exceeds a specified threshold.

Dataflow representation. Data preparation pipelines that use common declarative abstractions, such as pandas data slicing, scikit-learn's `ColumnTransformer` in combination with estimator/transformer pipelines or SparkML pipelines, have a natural DAG representation [6]. In our case, the data sources in this DAG are typically comprised of 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 producing vectors), or standard ML preprocessing operations like normalization or concatenation (consuming vectors and producing vectors).

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 identify relevant function calls. We then construct an intermediate representation (IR) of the pipeline using this information and the abstract syntax tree (AST) as basis. Finally, we build up the DAG using this IR.

Instrumentation. The main idea behind our instrumentation 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

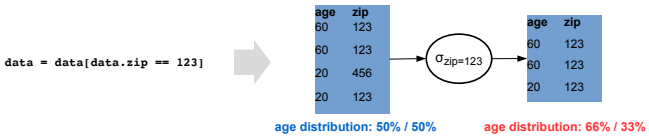


Figure 1: Example: `mlinspect` identifies relational and ML preprocessing operations in Python code, and instruments them with inspections that check for data distribution bugs.

retains a fixed-size state that is invoked once for each DAG operator, and is reset after each 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.

Backends for popular ML libraries. `mlinspect` uses the semantics of preprocessing operations of popular Python frameworks like scikit-learn and pandas. `mlinspect` delegates the captured function calls to the library-specific backend based on module information.

Inspections and checks. Inspections serve as basis for detecting issues 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, 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.

Our lineage-based annotation propagation approach enables different categories of automatic inspections and checks. Fairness is one of them: `mlinspect` can be used to detect operators that *introduce or amplify under-representation issues*. It can also check for *legal restrictions on the usage of demographic features*. ML models may also *perform particularly badly for specific demographic groups* in the data (e.g., higher false positive rates for recidivism predictions for Black defendants [1]). `mlinspect` can assist data scientists in identifying such issues, which 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.

3 DEMONSTRATION SCENARIO

In the demonstration, we will show how the extracted DAG, inspections and checks can be used to inspect pipelines and detect data distribution bugs. We provide a web-based user interface that allows attendees to edit the code of an ML pipeline and view the extracted DAG side-by-side, as shown in Figure 2. Attendees will interact with the DAG to inspect intermediate data and to determine how different operators change the data distribution. `mlinspect` will automatically warn users when a data distribution bug is detected.

We showcase an inspection that can materialise a sample of the output rows for each operator, and a check that uses an inspection that computes histograms of operator outputs for protected groups, and then compares negative changes in relative proportions of minority groups after a filter to an acceptable threshold.

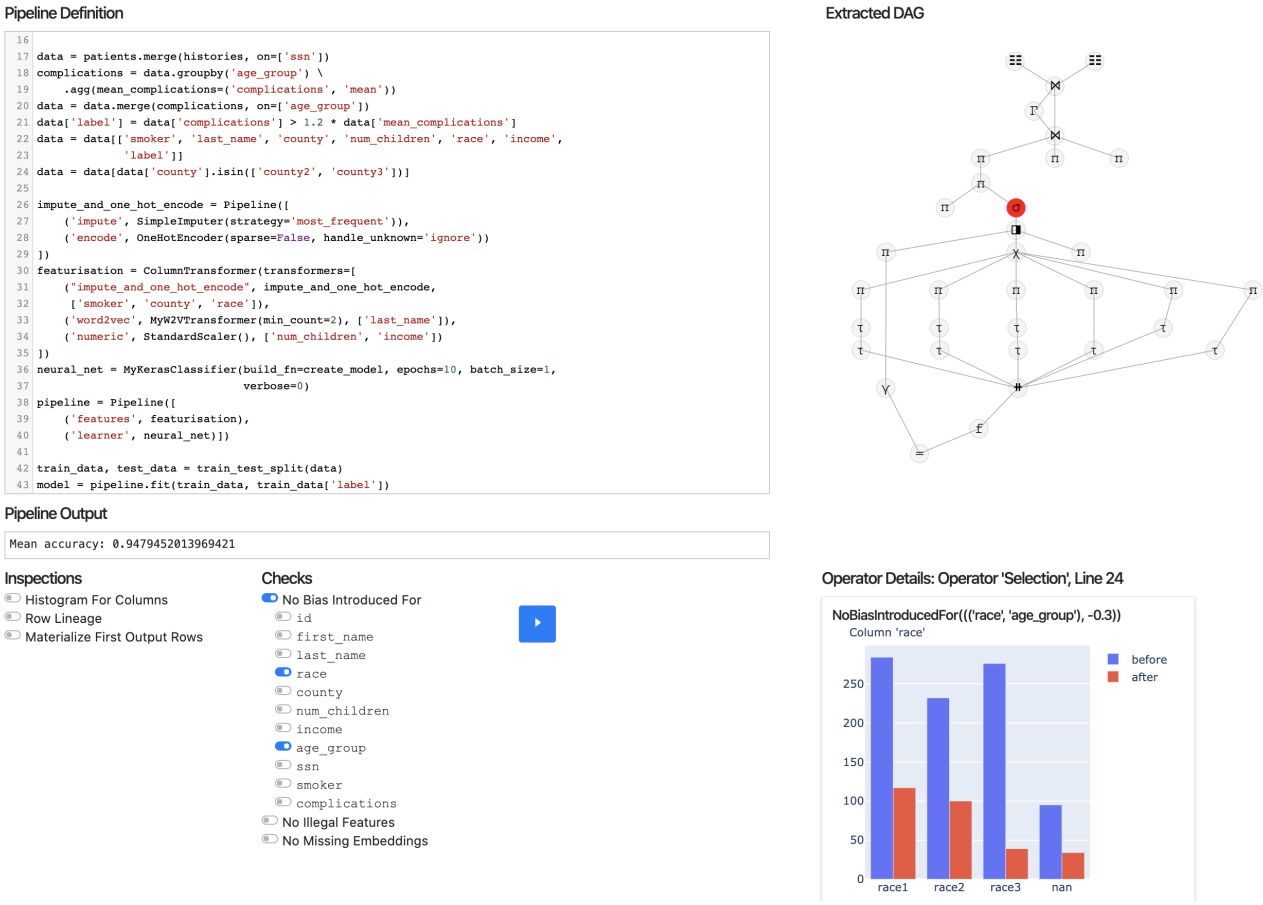


Figure 2: User interface for our demonstration. On the left, attendees can edit the Python code of an ML pipeline and select inspections to run. On the right, we show the corresponding dataflow graph extracted by `mlinspect`. Attendees can interactively click on DAG nodes to highlight the corresponding code snippets and view inspection results for the operators.

3.1 Example Pipeline

For our demonstration, we use an example from the medical domain. 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`.

We provide attendees with an implementation of this pipeline that uses pandas and scikit-learn.[1] The pipeline first reads two CSV files, which contain patient demographics and their clinical histories, respectively, and joins them. 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 others in their age group. The data is then projected to a subset of the attributes, to be used by the classification model.

The data scientist additionally filters the data to contain only records from patients within a given set of counties. This may lead to a data distribution bug if populations of different counties systematically differ in age or race.

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.

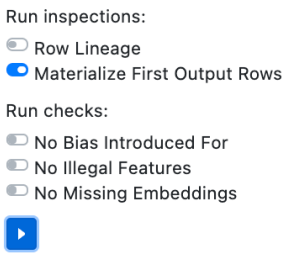
As illustrated by this example, preprocessing can give rise to subtle data distribution bugs that are difficult to identify manually. In our demonstration, attendees will use `mlinspect` to detect some of these potential errors and subsequently fix them in real time.

[1] https://github.com/stefan-grafberger/mlinspect/blob/19ca0d6ae8672249891835190c9e2d9d3c14f28f/example_pipelines/healthcare/healthcare.py

3.2 Pipeline Inspection

Attendees start with the given Python-based ML pipeline, with the data and the initial code for the healthcare example. Attendees can then extend and modify the code and execute it as they would in a computational notebook environment. As they run the pipeline, the DAG on the right side will be updated to reflect the changes they make. The DAG provides an abstract representation of the pipeline, which allows them to analyse complex nested pipelines. In addition, attendees can specify which inspections and checks they want to apply during pipeline execution (Figure 3(a)).

The *No Bias Introduced For* check looks for data distribution changes, while the *No Illegal Features* check determines whether potentially illegal attributes (such as gender) are used by the ML model. We also support inspections that sample intermediate outputs, compute histograms of groups in the intermediate results, and show record-level lineage.



(a) Attendees enable different checks and inspections.

MaterializeFirstOutputRows(5)

county
[1. 0.]
[1. 0.]
[0. 1.]
[1. 0.]
[1. 0.]

(b) Inspection results such as samples of the outputs of each operator.

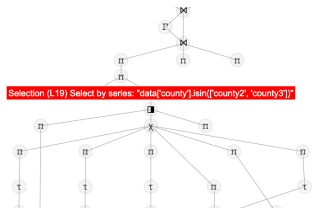
Figure 3: Attendees will run inspections on the Python pipeline and view the corresponding results per operator.

We first focus on the general techniques to inspect the ML pipeline. For that, we showcase the *Materialize First Output Rows* inspection, which allows users to view samples of the outputs of each operator. They enable this inspection, rerun the pipeline, and can then click on the DAG nodes that represent the operators to inspect the operator's output. As a result, attendees will see the first few output rows (Figure 3(b)), and our GUI will also highlight the code corresponding to the operator on the left. We will showcase this for a "one-hot-encoding" operator in a scikit-learn pipeline, which consumes relational data and produces matrix data.

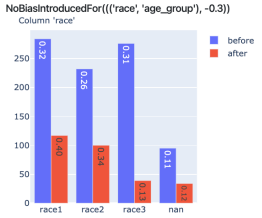
3.3 Data Distribution Debugging

In the next part of the demonstration, we show how our library assists data scientists with the detection of potential data distribution bugs. For that, we focus on operators like selection that may accidentally change the proportion of protected groups in the data. We use the *No Bias Introduced For* inspection, which analyses histograms of the input and output data of operators, and checks whether the change in histograms of protected groups exceeds a specified threshold. When attendees run the pipeline code while enabling the *No Bias Introduced For* check, `mlinspect` will highlight problematic operators directly in the DAG (Figure 4(a)).

Attendees can then click on the DAG node for additional details: `mlinspect` will highlight the source code that introduced the data



(a) Operators that potentially introduce data distribution bugs are highlighted in the DAG.



(b) Attendees can view histograms of the corresponding data distribution changes caused by the operator.

Figure 4: `mlinspect` detects potential data distribution bugs and points users to the corresponding operations.

distribution bug, and will show the data distribution histograms, helping the user investigate the issue (Figure 4(b)).

During the demonstration, we will use a pipeline with a selection that filters for people living in certain counties. In our example dataset, the `county` attribute is correlated with the sensitive attribute `race`. The selection code initially present in the pipeline will lead to a data distribution bug. But if participants discover the correlation, they can include an additional county value in the list of counties used for the filter to fix the bug. To validate their solution, participants can edit the pipeline, rerun the checks, and re-examine the histograms corresponding to the problematic operation.

Interactivity. Our demonstration is highly interactive, as attendees will be able to use our interface to rerun the pipeline, inspect the operators, and check for data distribution bugs. Once they discover a bug with the help of `mlinspect`, they can modify the source code in real time to fix it and re-run the pipeline.

Acknowledgements. This work was supported in part by Ahold Delhaize, and by NSF Grants No. 1926250, 1934464 and 1922658. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] 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
[2] Irene Chen, Fredrik D Johansson, and David Sontag. 2018. Why Is My Classifier Discriminatory? *NeurIPS* (2018), 3539–3550.
[3] Alexandra Chouldechova and Aaron Roth. 2020. A snapshot of the frontiers of fairness in machine learning. *Commun. ACM* 63, 5 (2020), 82–89.
[4] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. 2021. Lightweight Inspection of Data Preprocessing in Native Machine Learning Pipelines. *CIDR* (2021).
[5] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, et al. 2019. Data Science through the looking glass and what we found there. arXiv:1912.09536
[6] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NeurIPS*.
[7] 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* (2019).
[8] Sebastian Schelter and Julia Stoyanovich. 2020. Taming Technical Bias in Machine Learning Pipelines. *IEEE Data Eng. Bull.* 43, 4 (2020).
[9] Julia Stoyanovich, Bill Howe, and H.V. Jagadish. 2020. Responsible Data Management. *VLDB* 13, 12 (2020), 3474–3489.
[10] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng.* 41 (2018), 16–25.
[11] Matei Zaharia, Andrew Chen, Aaron Davidson, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018)