Automated Provenance-Based Screening of ML Data Preparation Pipelines

Sebastian Schelter[1*], Shubha Guha[2] and Stefan Grafberger[1]

[1*]BIFOLD & TU Berlin, Germany.
[2]University of Amsterdam, The Netherlands.

*Corresponding author(s). E-mail(s): schelter@tu-berlin.de;
Contributing authors: s.guha@uva.nl; grafberger@tu-berlin.de;

Abstract

Software systems that learn from data via machine learning (ML) are being deployed in increasing numbers in real world application scenarios. These ML applications contain complex data preparation pipelines, which take several raw inputs, integrate, filter and encode them to produce the input data for model training. This is in stark contrast to academic studies and benchmarks, which typically work with static, already prepared datasets. It is a difficult and tedious task to ensure at development time that the data preparation pipelines for such ML applications adhere to sound experimentation practices and compliance requirements. Identifying potential correctness issues currently requires a high degree of discipline, knowledge, and time from data scientists, and they often only implement one-off solutions, based on specialised frameworks that are incompatible with the rest of the data science ecosystem.

We discuss how to model data preparation pipelines as dataflow computations from relational inputs to matrix outputs, and propose techniques that use record-level provenance to automatically screen these pipelines for many common correctness issues (e.g., data leakage between train and test data). We design a prototypical system to screen such data preparation pipelines and furthermore enable the automatic computation of important metadata such as group fairness metrics. We discuss how to extract the semantics and the data provenance of common artifacts in supervised learning tasks and evaluate our system on several example pipelines with real-world data.

Keywords: Machine learning pipelines, data provenance, responsible data management

1 Introduction

Software systems that learn from data via machine learning (ML) are being deployed in increasing numbers in real world application scenarios. The behavior of such ML applications very much depends on their input data, and they are implemented with systems and libraries from a relatively young data science ecosystem, which is rapidly evolving all the time. Experience shows that it is difficult to ensure that such ML applications are implemented correctly [1, 2, 3, 4], and as a consequence, data scientists building these applications require fundamental system support. These ML applications contain complex data preparation pipelines, which take several raw inputs, integrate, filter and encode them to produce the input data for model training. This is in stark contrast to academic studies and benchmarks, which typically work with static, already

prepared datasets. It is a difficult and tedious task to ensure at development time that the data preparation pipelines for such ML applications adhere to sound experimentation practices and compliance requirements. Identifying potential correctness issues currently requires a high degree of discipline, knowledge and time from data scientists, and they often only implement one-off solutions, based on specialised frameworks that are incompatible with the rest of the data science ecosystem.

Data preparation pipelines. Academic benchmark scenarios in ML often start with a static, single-table input dataset. Real world ML applications on the contrary typically have to consume data from many differents sources to produce a single dataset for the subsequent ML model training. Figure 1 illustrates a common pattern for data preparation workloads observed in many industry scenarios [1, 2, 3]. The ML application consumes several heterogeneous data sources (1) as input (e.g., data originating from logfiles, database tables, web crawls, etc.), and the first stage in the data preparation pipeline is to integrate and clean the data from these data sources (2). In most cases, the data is represented in tabular form, e.g., with so-called dataframes, where individual cells might additionally contain unstructured data such as text or images. The integration and cleaning is typically conducted via relational operations such as joins and filters, often executed with a data processing library like pandas [5] or a dataflow system like Apache Spark [6]. The integration stage produces a single tabular output which must be encoded to matrix data to serve as training and test data for the subsequent model training (3). A common abstraction for conducting such feature encoding steps are so-called estimator/transformer pipelines, which have been popularised by scikit-learn [7] and have also been adapted by dataflow systems for ML like Spark's MLlib [8] or Google's Tensorflow Extended (TFX) platform [9]. The output of the feature encoding stage is trainining data in matrix form, based on which the typical ML model training and evaluation process can begin.

Automation challenges in data preparation pipelines. Many common tasks that arise during the development and training of ML models become more complex and tedious once we have to solve them in light of a data preparation pipeline (and not a static dataset like in many academic scenarios). These include the *violation of sound experimentation practices*, for example when data scientists unintentionally violate the strict isolation of train and test data. Moreover it is often difficult to *assess the group fairness* [10, 4] of a deployed model, which is difficult to conduct in complex data preparation pipelines, as sensitive attributes which identify groups may not directly be used by the model. Furthermore, deployed ML pipelines in the European Union must adhere to the rights defined in the General Data Protection Regulation (GDPR) in Europe, such as the 'right to be forgotten' [11] and the obligation to keep records of processing activities.

Towards automated low-effort screening of data preparation pipelines. Most of these issues are typically addressed manually in an adhoc way, and require high expertise and a large amount of additional code. In many cases there is no system support for detecting particular issues, and typically, data has to be integrated first, as common libraries assume the input to be in a single table. Furthermore, specialised solutions are often incompatible with popular libraries in the ecosystem. For example, AIF360 [12] requires the implementation of custom dataset classes for tracking group fairness, which in turn makes these datasets incompatible with common abstractions from scikit-learn, such as the `ColumnTransformer` for feature transformations on data frames. This situation is in stark contrast to the software engineering world, which has established a comprehensive set of best practices and infrastructure for testing and continuous integration.

Data provenance as foundation. We discuss how to model data preparation pipelines as dataflow computations from relational inputs to matrix outputs, and propose techniques to automatically screen these pipelines for many common correctness issues. We find that we can automate the detection of many such common issues in ML pipelines with access to (*i*) the materialised artifacts of a pipeline (its input relations, and its outputs, e.g., the feature matrices, labels and predictions of a classifier) as well as (*ii*) their why-provenance [13] (e.g., the information which input records were used to compute a particular output). This allows us to design lightweight

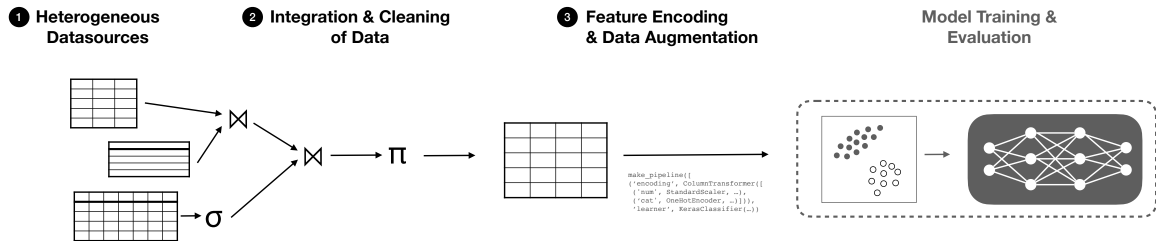

Fig. 1 High-level anatomy of a data preparation pipeline for a supervised learning task.

screening techniques with low invasiveness for natively written data preparation pipelines, which combine code from different libraries from the rapidly evolving data science ecosystem.

Based on these insights, we present a prototype, which operates on a natively written ML pipeline in Python, extracts intermediate results and provenance (in the form of provenance polynomials [14]) with MLINSPECT [15, 16], and infers the semantics of their artifacts based on predefined "templates" (e.g., for a classification task). We extend our initial ideas outlined in a previously published single-page abstract [17] and demonstration paper [18], and provide a prototype, which enables the automatic detection of common issues w.r.t. best practices in ML, and the computation of metadata such as group fairness metrics, record usage by the model, or data valuation with Shapley values. Our prototype handles supervised learning pipelines natively written in pandas/sklearn and keras, stores their artifacts and run data via mlflow [19], and can be easily hooked into continuous integration workflows. It is available at https://github.com/amsterdata/arguseyes.

Related work. Building better tooling for ML pipelines is an active topic of research [20, 21, 22, 23, 24, 25, 26]. In contrast to other recent research that needs to materialise provenance intermediates for every operator [27] or relies on automatically rewriting and re-executing ML pipelines [28, 29], our system only needs to materialise a small number of key artifacts during the execution of the user's original pipeline and implements all functionality as queries on the materialised artifacts [30]. This makes our approach particularly well-suited for integration in CI pipelines and experiment tracking workflows.

In summary, the contributions of this paper are the following:

- We discuss how to model data preparation pipelines for supervised learning as dataflow computations from relational inputs to matrix outputs, and detail how to automate issue screening and metadata computation based on the captured artifacts and record-level data provenance (Section 4).
- We implement a prototype to automatically (*i*) screen for common issues like data leakage between train and test set, covariate shift, label shift or unnormalised features in data preparation pipelines, and (*ii*) compute metadata like group fairness metrics of a classifier (Sections 4).
- We qualitatively evaluate the issue detection and metadata computation capabilities of our system on a set of example pipelines (Section 5.2).

2 Problem Statement

Supervised learning. In the standard scenario for supervised learning, we have a feature space $\mathcal{X} \subset \mathbb{R}^d$ and a label space $\mathcal{Y}$ with $\mathcal{Y} = \{1, 2, \dots, C\}$ for classification problems and $\mathcal{Y} = \mathbb{R}$ for regression problems. We are given a dataset $\mathcal{F}$ of n labeled samples $\{(\mathbf{x}_i, y_i)\}_{i \in [n]}$, where $\mathbf{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, with the goal to learn a map $h : \mathcal{X} \rightarrow \mathcal{Y}$ that generalises to unseen data, based on the labeled dataset $\mathcal{F}$ (with the assumption that all data points in $\mathcal{F}$ sampled i.i.d. from unknown distribution $p(X, Y)$ where X and Y are random variables). In practice, we learn a model $\hat{h} = \text{argmin}_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} L(h(\mathbf{x}_i), y_i)$ by performing empirical risk minimisation on our dataset $\mathcal{F}$ for a task-specific loss function L and hypothesis space $\mathcal{H}$. Note that we split the dataset $\mathcal{F}$ into a disjunct training set $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ for learning model parameters and a test set $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$

for evaluating the generalisation performance of the model on unseen data. Many common techniques for experimental validation (e.g., testing for dataset shift [31] or data debugging [32] are designed to work on the matrix inputs $\mathbf{X}_{\text{train}}$, $\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{train}}, \mathbf{y}_{\text{test}}$.

Incorporating data preparation steps. As discussed in the introduction already, many ML applications in real-world scenarios do not start from a single source of input data $\mathcal{F}$ in matrix form. Instead, there exists a data preparation pipeline that has to compute the matrix data $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ and $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$ from several other input datasets (e.g., log files, databases tables, files in a data lake, ...) $\mathcal{D}_1, \ldots, \mathcal{D}_k$ via a *data preparation pipeline*, often in the form of structured/relational data accompanied by unstructured data such as text and images. We can think of the data preparation pipeline as a map $\mathcal{D}_1, \ldots, \mathcal{D}_k \rightarrow ((\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}}), (\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}}))$ from the raw inputs $\mathcal{D}_1, \ldots, \mathcal{D}_k$ to the desired matrix outputs. In our experience, this map consists of three subsequent high-level stages:

1. Integrating the individual datasets $\mathcal{D}_1, \ldots, \mathcal{D}_k$ into a single tabular dataset; this stage requires the execution of relational joins to combine data.
2. Cleaning and filtering the data to select a subset which will form the train and test data for the model; in its simplest form, this stage requires relational selections and projections to remove tuples and remove or compute attributes
3. Encoding the data into matrix form, this step typically requires specialised feature encoding operations that produce matrix outputs (e.g., one-hot-encoding, feature hashing, binning, etc).

Furthermore, it is important to note that the input datasets are not static in real-world applications, but continously evolve, and both data preparation and model training have to be regularly conducted.

Towards automation for correctness checks and metadata computations in data preparation pipelines. As already discussed, there is a common set of correctness checks and metadata computations that data scientists have to conduct for their ML applications, such as ensuring sound experimentation practices, assessing the group fairness of the model, complying with GDPR legislation and conducting data debugging. These tasks are, at the moment, typically manually executed by data scientists, have to be reimplemented for each application, and require a high amount of expertise, coding, and time. In this paper, we tackle the following research questions to work towards the automation of these tasks in ML applications containing data preparation pipelines:

- How should we model the input data and the data preparation pipeline in order to reason about them?
- How can we automate the outlined common tasks for a wide variety of data preparation pipelines for supervised learning tasks?
- How can we efficiently implement these techniques for the existing data science ecosystem?

3 Example

Before we discuss our approach, we illustrate our outlined problem on a concrete code example:

```python
def load_data(target_categories, verified_only):
  # Load individual input relations
  reviews = pd.read_csv('.../reviews.csv.gz')
  products = pd.read_csv('.../products.csv')
  categories = pd.read_csv('.../categories.csv')
  # Filter categories and join inputs
  categories = categories[categories['category'] \
    .isin(target_categories)]
  if verified_only: # Conditional filtering
    reviews = \
      reviews[reviews['verified_purchase'] == 'Y']
  products = products.merge(categories,
    on='category_id')
  reviews = reviews.merge(products, on='product_id')
  reviews['all_text'] = \
    reviews['title'] + reviews['review'].fillna('')
  return reviews

# Manual train/test splits
def extract_train_data(reviews, split_date):
  train_data = \
    reviews[reviews['review_date'] <= split_date]
  train_labels = \
    label_binarize(train_data['helpful_votes'] > 0)
  return train_data, train_labels

def extract_test_data(reviews, split_date):
  test_data = \
    reviews[reviews['review_date'] > split_date]
  test_labels = \
    label_binarize(test_data['helpful_votes'] > 0)
  return test_data, test_labels

# Generate a nested estimator/transformer pipeline
def feature_encoding(numerical, categorical, text):
  one_hot = Pipeline([
    (SimpleImputer(strategy='most_frequent')),
```

```
    (OneHotEncoder(handle_unknown='ignore'))])

  return ColumnTransformer(transformers=[
    ('num', FunctionTransformer(
      lambda x: log(x)), numerical),
    ('cat', one_hot, categorical),
    ('text', HashingVectorizer(), text)])

# Define layout of neural network classifier
def neural_net():
  nn = Sequential([
    Dense(256, activation='relu'), Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(2, activation='softmax')])
  return nn.compile(
    loss='sparse_categorical_crossentropy')

# Main function to execute the pipeline
reviews = load_data(target_categories=['Games'],
    verified_only=False)
split_date = '2015-07-31'
train_data, train_labels = \
  extract_train_data(reviews, split_date)
test_data, test_labels = \
  extract_test_data(reviews, split_date)
feature_transformation = feature_encoding(
  numerical=['star_rating'], text='all_text',
  categorical=['verified_purchase', 'category_id'])
pipeline = Pipeline([
    ('features', feature_transformation),
    ('learner', KerasClassifier(neural_net()))])
# Model training and evaluation
model = pipeline.fit(train_data, train_labels)
print(model.score(test_data, test_labels))
```

This illustrative example pipeline trains a classifier to identify helpful product reviews in an e-commerce scenario, based on a database of customer reviews consisting of three input datasets. This pipeline applies relational operations on dataframes for preprocessing, and matrix operations for feature encoding and model training. Furthermore, it has its code organised in functions, contains control flow (a conditional selection), applies a manually defined train/test split based on a specified date, and encodes the features a nested estimator/transformer pipeline. Analogously to many ML applications in the real world, our example combines different libraries. The pipeline uses pandas for data preprocessing: it reads several inputs files, filters and joins them, and also derives a new attribute from the data in the `load_data` function. Next, the data is encoded into features via a declaratively specified nested estimator/transformer pipeline from scikit-learn in the `feature_encoding` function, which defines the representation of categorical, numerical and textual attributes of the data. The model to train is a neural network whose layout is specified via the keras API in the `create_neural_net` function.

In real world application scenarios, we would be eager to screen the pipeline for common violations of sound experimentation practices in ML, and to compute additional meta information for its data. In the following, we discuss such example tasks and explain why they are difficult to conduct.

Detecting violations of sound experimentation practices. There is a common set of concerns that have to be checked for every supervised learning scenario: Are train data and test data correctly isolated from each other? Is there covariate or label shift between the train set and the test set? Are there constant features that we could remove? Are the features appropriately normalised? This is difficult for our example pipeline, because the programmer cannot directly access the feature matrices for the train and test data. These matrices are internally built by scikit-learn's `ColumnTransformer` during the `fit` and `score` calls of the nested estimator/transformer pipeline and never explicitly exposed to the user program! In order to apply such checks to the train and feature matrix, a data scientist would have to rewrite the code such that the feature encoders materialise the feature matrices, and would have to subsequently invoke training and inference manually. Furthermore, checking for issues like data leakage between train and test set is also complicated in our example, because each training/test sample for the classifier combines rows from our three input datasets (reviews, products and categories), and the programmer would have to manually reconstruct the relationship between the rows of a feature matrix and the input tuples of the input datasets.

GDPR compliance. For legal compliance, it may be important to know the relationship between the input tuples and the training data of our model in the example. Recall that the pipeline consumes three different input files, joins and filters them (which potentially removes tuples) and only selects a subset of the tuples for training the model. Assume that the reviews written by a user contain personally identifiable data (e.g., about their gender identity). If the corresponding user enforces their "right-to-be-forgotten" from the GDPR and requests the deletion of their data, the model would have to be retrained if the review

text of this particular user was part of the training set. Answering the question which part of the raw input data was used to train a model is difficult in scenarios with complex data preparation pipelines [33], as it again requires a data scientist to manually reconstruct the relationship between the rows of a feature matrix and the input tuples of the input datasets.

Assessing group fairness. There are further challenges when working towards the ethical and legal compliance of ML applications [4]. For example, we may want to investigate whether our classifier works reasonably well for different subsets of the data, in order to avoid bias and discrimination. It could be that the performance of the classifier differs for data from different regions, as identified by the `marketplace` attribute in the data. In order to investigate this, we would for example need to compare the accuracy of the classifier for test data from the US marketplace to its accuracy on reviews from non-US marketplaces. This is difficult for two reasons: (i) it again requires access to the feature matrix for the test data, which is only implicitly formed by the estimator/transformer pipeline during the `score` call, and (ii) it requires us to identify the corresponding `marketplace` value for each row of the feature matrix. This value is not part of the feature matrix, as the `marketplace` column is removed during feature encoding by the `ColumnTransformer`. Solving this issue again requires the data scientist to manually reconstruct the relationship between the rows of a feature matrix and the input tuples of the review dataset, which contains the marketplace information. Note that libraries like AIF360 [12] provide specialised dataset abstractions that retain the group membership information, however these datasets classes are incompatible with the rest of the data science ecosystem (e.g., certain constructs in scikit-learn pipelines).

4 Approach

Modeling data preparation pipelines as a relational dataflow computation. We focus on data preparation pipelines for classification tasks, and look for a lightweight "template" for such tasks, e.g., a minimal set of information that enables us to automatically screen the code for correctness violations and compute metadata such as group fairness metrics.

Dataflow from relational inputs to matrix outputs. We first define a way to reason about the inputs and operations in data preparation pipelines. Therefore, we model a classification pipeline as a dataflow computation on k input relations $D_1, \ldots, D_k$ in a star schema, which produces the following set of matrices: the training feature matrix $\mathbf{X}_{\text{train}} \in \mathbb{R}^{n \times d}$ with the corresponding training labels $\mathbf{y}_{\text{train}} \in \mathbb{R}^n$, and the test feature matrix $\mathbf{X}_{\text{test}} \in \mathbb{R}^{m \times d}$ with the corresponding test labels $\mathbf{y}_{\text{test}} \in \mathbb{R}^m$ and model predictions $\mathbf{y}_{\text{pred}} \in \mathbb{R}^m$.

The assumption of relational inputs in a star schema means that there is one relational input D_e (not necessarily specified explicitly) where each tuple identifies the entities to classify with a primary key (e.g., this input corresponds to the "fact table" in data warehousing scenarios). The remaining $k-1$ relational inputs contain additional side data for the entities, and D_e has a foreign key attribute referring to the primary key of each side input (e.g., they correspond to the "dimensional tables" in data warehousing). This means that the data integration stage can combine the individual datasets into a single table by conducting a series of (left) joins between the entity table D_e and the side tables. The subsequent data cleaning and filtering can then be conducted with relational selections and (extended) projections to remove tuples and remove/recompute attributes. Formally, this means that the first two stages of the data preparation pipeline only require operations from the positive relational algebra (SPJU; selection, projection, join, union) [14]. The final feature encoding stage turns the data into matrix form with operations such as one-hot-encodings. We can also model this stage with the positive relational algebra, if we treat the feature encoders as extended projections that produce vector outputs. Note that the idea of modeling ML workloads with a mixture of relational and feature encoding operations is becoming popular recently; Microsoft's Raven [34] system for example uses a fine-grained intermediate representation such as ONNX [35] to extend relational query processing with ML inference.

Capturing data provenance. Next, we need a principled way to capture the relationship between

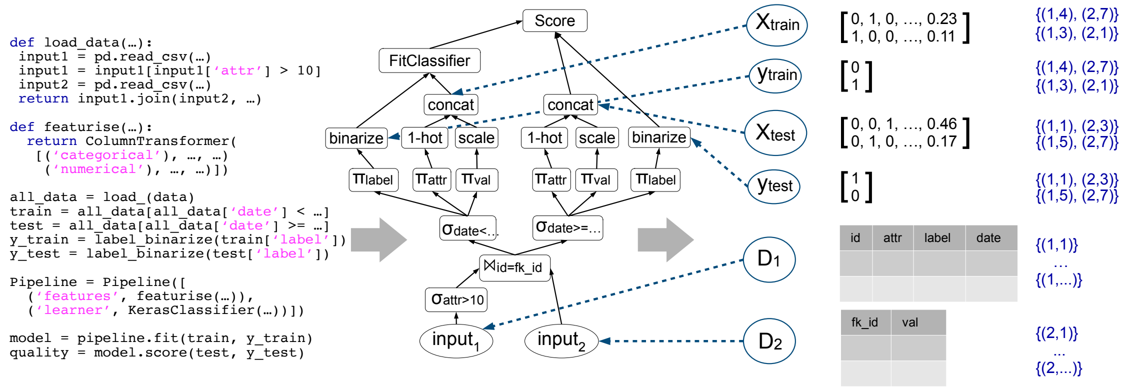

Fig. 2 High-level overview of provenance tracking and artifact capturing for a simplified version of our example pipeline.

the tuples of our relational inputs and the rows of the matrix outputs. For that, we assume that each tuple t_j in an input relation D_i is tagged with a tuple identifier (i, j), and we compute the why-provenance [13, 14] of the pipeline based on the provenance semiring $(\mathcal{P}(I), \cup, \cup, \emptyset, \emptyset)$, where I consists of the ids of the tuples in the input datasets, $\mathcal{P}(I)$ denotes the powerset of I, and polynomials are always combined via set union, e.g., in the case of joins. Specifically, we compute a provenance polynomial per row of each output matrix. These polynomials can be retrieved via the function $L_{\mathbf{M}}(r)$, which maps the r-th row $\mathbf{M}[r]$ of an output matrix $\mathbf{M}$ to its corresponding provenance polynomial from $\mathcal{P}(I)$. In summary, the function L allows us to identify the input tuples from $D_1, \ldots, D_k$ which contributed to the computation of a particular training or test sample.

Figure 2 illustrates the discussed provenance tracking and artifact capturing techniques for a simplified version of our example pipeline.

Correctness checks and metadata computation as queries over materialised artifacts and their data provenance. We detail how to compute correctness checks and metadata based on the outlined artifacts and their provenance.

Data leakage. The test data should be disjunct from the training data to avoid information leakage. Given the pipeline outputs $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$, we can test for this isolation condition by comparing the provenance of the training feature matrix $\mathbf{X}_{\text{train}}$ with the provenance of the test feature matrix $\mathbf{X}_{\text{test}}$, and state that they need to be computed from disjunct sets of input tuples: $\nexists\, 0 \leq p < n, 0 \leq q < m : L_{\mathbf{X}_{\text{train}}}(p) = L_{\mathbf{X}_{\text{test}}}(q)$.

GDPR compliance. For GDPR compliance, we need to answer the question whether a given input tuple has been part of the training data of an ML model produced by a pipeline. If this tuple contains personal information of a user and the user asks for the deletion of their personal data (e.g., enforcing their "right-to-be-forgotten"), we may have to retrain the model without this data [11]. We can determine whether a given tuple t_j from a relation D_i was used as part of the training data of the model by analysing the provenance of the training feature matrix $\mathbf{X}_{\text{train}}$. If the tuple has been used to train the model, then the provenance polynomial of a row $\mathbf{X}_{\text{train}}[k]$ must contain its tuple identifier: $\exists\, 0 \leq k < n : \{(i, j)\} \subseteq L_{\mathbf{X}_{\text{train}}}(k)$.

Group fairness metrics. A crucial question for ethical and legal compliance in many use cases is whether the classifier produced by a pipeline works reasonably well for different groups of persons in the data. Answering this question requires us to compute *group fairness metrics* [36]. Auditors would define a protected group whose records can be identified by a sensitive attribute A of relation D_i having the value v_{prot} (e.g., records with `gender=female`). As basis for the metrics, we need to compute the confusion matrix of the classifier from its ground truth labels $\mathbf{y}_{\text{test}}$ and model predictions $\mathbf{y}_{\text{pred}}$ with respect to the records from the protected group (and analogously for the records from the remainder of the data, the non-protected

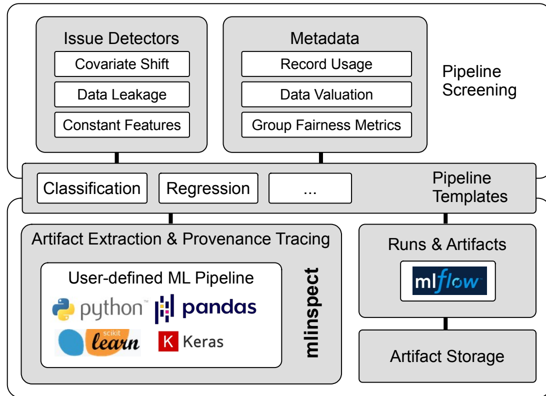

Fig. 3 Architecture of our proposed system for ML pipeline screening.

group). As previously mentioned, these confusion matrices are difficult to compute for ML piplines, as the sensitive attribute A might have been projected out during preprocessing. We use the provenance of each row from the model predictions $\mathbf{y}_{\text{pred}}[k]$ and labels $\mathbf{y}_{\text{test}}[k]$ to identify its corresponding tuple t_j in the input relation D_i, which allows us to test for its group membership analysing the sensitive attribute $t_j[A]$. We can identify all label/prediction pairs required for computing the confusion matrix for the protected group as follows: $\{(\mathbf{y}_{\text{test}}[k], \mathbf{y}_{\text{pred}}[k]) \mid 0 \leq k < m \wedge t_j \in D_i \wedge t_j[A] = v_{\text{prot}} \wedge \{(i,j)\} \subseteq L_{\mathbf{y}_{\text{test}}}(k)\}$. We analogously compute the confusion matrix for the non-protected group by changing the predicate for the sensitive attribute to $t_j[A] \neq v_{\text{prot}}$.

Distribution shift, constant features, etc. Many other issues do not even require the provenance, and can be detected by only examining the matrix outputs of the pipeline, e.g., to detect constant or unnormalised features, we identify the columns of $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$ which have constant values, a non-zero mean or non-unit variance. In order to detect covariate shift, we can train a "domain classifier" [31] which tries to distinguish between records $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$; in order to detect label shift between $\mathbf{y}_{\text{train}}$ and $\mathbf{y}_{\text{test}}$, we can for example apply the "label classifier" approach from [31], which conducts a two-sample χ^2 test between their empirical label distributions. Note that our approach is general and allows us to apply all kinds of shift detection techniques to the pipeline that only require access to $\mathbf{X}_{\text{train}}$, $\mathbf{X}_{\text{test}}$ $\mathbf{y}_{\text{train}}$ and $\mathbf{y}_{\text{test}}$.

Implementation. We extract the artifacts and provenance with our recently published library MLINSPECT [37, 16] which can instrument ML pipelines written natively in Python with dataframe operations from pandas and feature transformation operations from scikit-learn. Based on these artifacts, our system fills a "template" for an ML task, e.g. a classification task, and stores the artifacts together with run information in the experiment tracking system *mlflow* [19]. We implement a variety of so-called "issue detectors" and "metadata" computations that leverage the pipeline template (together with the corresponding artifacts and provenance) to screen the pipeline for correctness issues or to compute metadata like group fairness metrics.

5 Evaluation

We implement three complex example pipelines for classification and regression problems on public datasets to demonstrate how our system can be applied in real world use cases. Section 5 gives an overview of these pipelines and denotes the task type (**type**) of the pipeline, the way in which train and test data are generated (**split**), the number of input datasets (**#inputs**), and whether they contain control flow (**cflow**), missing values in the data (**missing**). Furthermore, we list the number of filter (**#filters**) and join operations (**#joins**) applied. These pipelines use simple off-the-shelf models, as our focus is on handling complex data preparation and encoding operations, and not on the actual learning tasks.

5.1 Experimental Setup

To the best of our knowledge, there exists no other system or benchmark to screen data preparation pipelines for ML applications at this point. This lack of a baseline makes it difficult to quantitatively evaluate our work (e.g., by counting the number of correctly identified issues compared to a potential baseline). We see the main benefit of our approach in its ability to automatically apply techniques to pipelines without code modifications. As a consequence, we decide to qualitatively evaluate our approach based on the introduced complex example pipelines as follows: we artificially introduce correctness issues into our pipelines (or ask for metadata to be computed), detail how this

Name	type	split	#inputs	control flow?	missing?	filters?	#joins
Reviews	classification	temporal	4	yes	yes	4	2
Ratings	regression	temporal	4	yes	yes	4	2
Credit	classification	predefined	1	yes	yes	dyn.	-

Table 1 Overview of our example pipelines for experimentation. We denote the task type of the pipeline (**type**), the way in which train and test data are generated (**split**), the number of input datasets (**#inputs**), and whether they contain control flow (**cflow**) or missing values in the data (**missing**). Furthermore, we list the number of filter (**#filters**) and join operations (**#joins**) applied.

can be achieved automatically by our system, and discuss afterwards how that could have been achieved manually, and what effort this would have required.

We evaluate unsound experimentation issues on the *Reviews* and *Ratings* pipelines and investigate automatic group fairness metrics computation and data usage tracking for GDPR on the *Credit* pipeline. We showcase that our system successfully detects the issues and computes the required metadata without requiring code changes. For comparison, we discuss the difficulties in manually rewriting the pipeline to detect a given issue or compute the required metadata.

Please note that our provenance tracking uses the existing library `mlinspect` [16, 37], thus, we inherit its provenance tracking performance, which has been evaluated in previous publications [21, 16].

5.2 Qualitative Evaluation of the Screening Tasks

Detecting data leakage between train and test set. In our first experiment, we showcase how our system can detect data leakage between train and test data. This is a serious issue, as we might get a wrong estimate of the generalisation capabilities of a model if its evaluation data accidentally contains already seen training examples. In practice such data leakage often happens as a result of programming errors in data preparation code or when the experimentation data is prepared by non-ML experts (e.g., business teams).

Simulating data leakage as a result of an erroneous temporal data split. We create a faulty version of the *Reviews* pipeline. The correct version of the pipeline conducts a temporal split of the data to generate the train set via the predicate `review_date <= split_date` and the test set via `review_date > split_date`. We simulate a programming error by changing the test set predicate to `review_date >= split_date`, which leads to the accidental inclusion of training data from the day specified by `split_date` in the test set.

Automatic detection. We configure our system to run the faulty version of the pipeline and look for data leakage via a declarative config file. Our system automatically detects the data leakage issue from the provenance for the train and test feature matrices, and reports that there are 81 overlapping records between the train and test data, which are exactly the records belonging to the day `2015-07-31` specified in the `split_date` variable. As discussed in Section 4, we determine the overlap solely by examining the provenance polynomials of the captured train and test feature matrices.

Effort and difficulty for manual detection. In contrast to other issues, the data leakage detection would require relatively little code to implement for our example pipeline. A data scientist would have to write code to intersect the pandas dataframes for the train and test data at the correct location in the pipeline (e.g., before the feature encoding) and would have to manually look for duplicate columns. Computing the intersection of the columns on the raw data can become very expensive though as it requires an inner join between the datasets with all existing columns as join keys. Our approach of intersecting the data provenance (sets of integers) of the train and test matrices can be implemented much more efficiently.

Group fairness assessment. As already discussed earlier, a major challenge in production ML is to enforce legal and ethical compliance. In order to determine whether a model works reasonably well for different groups [10], one needs to compute group fairness metrics for different subsets of the data. This is difficult to conduct in

data preparation pipelines, as sensitive attributes which identify groups may not directly be used by the model (or may even be illegal to use). We investigate this task using the *Credit* pipeline, where we intend to compute group fairness metrics based on the demographic features `sex` and `race`, which are not used by the model and projected out during data preparation.

Automatic computation. We declaratively configure ArgusEyes to compute the confusion matrices and group fairness metrics for groups defined by the `race` attribute (white compared to non-white), and `sex` attribute (male compared top non-male) from the data. ArgusEyes computes these metrics and logs them to mlflow. It determines that there is a vast difference in the classifier's false negative rate (FNR) (falsely predicting a person with a high income to have a low income level) for the groups in both cases, violating the fairness notion of *equal opportunity* [36]. The FNR for persons with the value `sex=male` is 59.7%, compared to 66.5% for the rest; the FNR for persons with the attribute `race=white` is 60.1%, while 66.9% for non-white persons.

Effort and difficulty for manual detection. Manually implementing the fairness metrics calculation for our pipeline is very tedious for several reasons: (*i*) we need access to the predictions $\mathbf{y}_{\text{pred}}$, which are not explicitly exposed, as we only call the `score` function; (*ii*) we need to map the predictions back to the input recods, which have different dimensions, as our pipeline filters out records by workclass during data preparation; (*iii*) we need to know the group membership and ground truth label per prediction, which unfortunately rely on sensitive attributes that are removed from the data before feature encoding. A data scientist would either have to completely rewrite the pipelines, e.g., by adding identifiers to records and maintaining a custom mapping of group memberships or would have to switch the code to using a dedicated fairness library like AIF360 [38], whose dataset implementation is unfortunately incompatible with many other popular components from data science libraries such as scikit-learn's estimator/transformer pipelines.

Data usage tracking for GDPR compliance. A related compliance task originating from GDPR that we already discussed is the the obligation to keep records of processing activities (https://gdpr-info.eu/art-30-gdpr/) to enforce data deletion rights such as the 'right to be forgotten' (https://gdpr-info.eu/art-17-gdpr/). For that, we need to know which personal data was used to train a particular model, as the law might require us to retrain the model if a user requests their data to be deleted. In our *Credit* pipeline, not all data is used for model, as some records are filtered out initially, based on a dynamic filter on the `workclass` attribute.

Automatic computation. We configure our system to track input record usage in the *Credit* pipeline, and configure our dynamic filter to remove public employees (with a `workclass` attribute value of 'Federal-gov' or 'State-gov') from the data. Our system captures the input data, and computes an additional attribute based on provenance, which indicates where a particular record was part of the final training data or not. We retrieve the serialised data from mlflow and verify that our system correctly identified that the 960 'Federal-gov ' and 1,298 'State-gov' employees are marked as non-participating.

Effort and difficulty for manual detection. Tracking the record usage is again tedious to manually implement. One would have to generate artificial identifiers for the input records, and manually extract and store the identifiers of the remaining records at the correct location (e.g., before feature encoding).

6 Conclusion

We discussed how to model data preparation pipelines as dataflow computations from relational inputs to matrix outputs, and detailed how to to automatically screen these pipelines for many common correctness issues. We designed a prototypical system to screen such data preparation pipelines and furthermore compute important metadata such as group fairness metrics. Furthermore, we qualitatively evaluated our system on a set of complex example pipelines with real-world data.

References

[1] Sculley, D., Holt, G., Golovin, D. & other. Hidden technical debt in machine learning systems. *NeurIPS* (2015).

[2] Polyzotis, N., Roy, S., Whang, S. E. & Zinkevich, M. Data lifecycle challenges in production machine learning: a survey. *SIGMOD Record* **47** (2018).

[3] Schelter, S. *et al.* On challenges in machine learning model management. *IEEE Data Engineering Bulletin* (2018).

[4] Stoyanovich, J., Howe, B. & Jagadish, H. Responsible data management. *PVLDB* (2020).

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

[6] Zaharia, M. *et al.* Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* 15–28 (2012).

[7] Pedregosa, F., Varoquaux, G. *et al.* Scikit-learn: Machine learning in python. *JMLR* (2011).

[8] Meng, X. *et al.* Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* **17**, 1235–1241 (2016).

[9] Baylor, D. *et al.* Tfx: A tensorflow-based production-scale machine learning platform. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 1387–1395 (2017).

[10] Chen, I., Johansson, F. D. & Sontag, D. Why is my classifier discriminatory? *NeurIPS* (2018).

[11] Schelter, S. "amnesia"-a selection of machine learning models that can forget user data very fast. *CIDR* (2020).

[12] Bellamy, R. K., Dey, K. *et al.* Ai fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv:1810.01943* (2018).

[13] Herschel, M., Diestelkämper, R. & Lahmar, H. B. A survey on provenance: What for? what form? what from? *The VLDB Journal* (2017).

[14] Green, T. J., Karvounarakis, G. & Tannen, V. Provenance semirings. *PODS* (2007).

[15] Grafberger, S., Stoyanovich, J. & Schelter, S. Lightweight inspection of data preprocessing in native machine learning pipelines. *CIDR* (2021).

[16] Grafberger, S., Groth, P., Stoyanovich, J. & Schelter, S. Data distribution debugging for machine learning pipelines. *The International Journal on Very Large Databases (VLDBJ)* (2022).

[17] Schelter, S. *et al.* Screening native machine learning pipelines with arguseyes. *Conference on Innovative Data Systems Research (CIDR)* (2022).

[18] Schelter, S., Grafberger, S., Guha, S., Karlas, B. & Zhang, C. Proactively screening machine learning pipelines with arguseyes. *Companion of the 2023 International Conference on Management of Data* 91–94 (2023).

[19] Zaharia, M., Chen, A., Davidson, A. *et al.* Accelerating the machine learning lifecycle with mlflow. *IEEE Data Engineering Bulletin* (2018).

[20] Klettke, M. & Störl, U. Four generations in data engineering for data science. *Datenbank-Spektrum* **22**, 59–66 (2022). URL https://doi.org/10.1007/s13222-021-00399-3.

[21] Schüle, M. E., Scalerandi, L., Kemper, A. & Neumann, T. Stoyanovich, J. *et al.* (eds) *Blue elephants inspecting pandas: Inspection and execution of machine learning pipelines in SQL.* (eds Stoyanovich, J. *et al.*) *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, 40–52 (OpenProceedings.org, 2023). URL https://doi.org/10.48786/edbt.2023.04.

[22] Neutatz, F., Chen, B., Alkhatib, Y., Ye, J. & Abedjan, Z. Data cleaning and automl: Would an optimizer choose to clean? *Datenbank-Spektrum* **22**, 121–130 (2022). URL https://doi.org/10.1007/s13222-022-00413-2.

[23] Kumar, A., Boehm, M. & Yang, J. Data management in machine learning: Challenges, techniques, and systems. *Proceedings of the 2017 ACM International Conference on Management of Data* 1717–1722 (2017). URL https://doi.org/10.1145/3035918.3054775.

[24] Redyuk, S., Kaoudi, Z., Schelter, S. & Markl, V. Assisted design of data science pipelines. *The VLDB Journal* (2024). URL https://doi.org/10.1007/s00778-024-00835-2.

[25] Grafberger, S., Zhang, Z., Schelter, S. & Zhang, C. Red onions, soft cheese and data: From food safety to data traceability for responsible ai. *IEEE Data Engineering Bulletin* (2024).

[26] Grafberger, S., Groth, P. & Schelter, S. Towards interactively improving ml data preparation code via" shadow pipelines". *DEEM workshop @ SIGMOD* (2022).

[27] Chapman, A., Lauro, L., Missier, P. & Torlone, R. Supporting better insights of data science pipelines with fine-grained provenance. *ACM Trans. Database Syst.* **49** (2024). URL https://doi.org/10.1145/3644385.

[28] Grafberger, S., Groth, P. & Schelter, S. Automating and optimizing data-centric what-if analyses on native machine learning pipelines. *SIGMOD* (2023).

[29] Grafberger, S., Guha, S., Groth, P. & Schelter, S. mlwhatif: What if you could stop re-implementing your machine learning pipeline analyses over and over? *Proc. VLDB Endow.* **16**, 4002–4005 (2023). URL https://doi.org/10.14778/3611540.3611606.

[30] Schelter, S. Reconstructing and querying ml pipeline intermediates. *CIDR* (2023).

[31] Rabanser, S., Günnemann, S. & Lipton, Z. C. Failing loudly: An empirical study of methods for detecting dataset shift. *NeurIPS* (2019).

[32] Jia, R., Dao, D., Wang, B. *et al.* Efficient task-specific data valuation for nearest neighbor algorithms. *PVLDB* **12** (2019).

[33] Namaki, M. H., Floratou, A., Psallidas, F. & other. Vamsa: Automated provenance tracking in data science scripts. *KDD* (2020).

[34] Karanasos, K. *et al.* Extending relational query processing with ML inference. *CIDR* (2021).

[35] Onnx. http://onnx.ai/, (2019).

[36] Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K. & Galstyan, A. A survey on bias and fairness in machine learning. *ACM Computing Surveys* **54** (2021).

[37] Grafberger, S., Guha, S., Stoyanovich, J. & Schelter, S. Mlinspect: A data distribution debugger for machine learning pipelines. *Proceedings of the 2021 International Conference on Management of Data* 2736–2739 (2021).

[38] Bellamy, R. K. E. *et al.* AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias (2018). URL https://arxiv.org/abs/1810.01943.