Responsibility in the Data Science Lifecycle

Responsible Data Science
DS-UA 202 and DS-GA 1017

Compiled by Julia Stoyanovich

This reader contains selected articles on responsibility in the data science lifecycle. For convenience, the readings are organized by course week.

Week 5: Responsible data management, data profiling, data cleaning

DOI:10.1145/3488717

Perspectives on the role and responsibility of the data-management research community in designing, developing, using, and overseeing automated decision systems.

BY JULIA STOYANOVICH, SERGE ABITEBOUL,
BILL HOWE, H.V. JAGADISH, AND SEBASTIAN SCHELTER

Responsible Data Management

INCORPORATING ETHICS AND legal compliance into data-driven algorithmic systems has been attracting significant attention from the computing research community, most notably under the umbrella of fair[8] and interpretable[16] machine learning. While important, much of this work has been limited in scope to the "last mile" of data analysis and has disregarded both the *system's design, development, and use life cycle* (What are we automating and why? Is the system working as intended? Are there any unforeseen consequences post-deployment?) and the *data life cycle* (Where did the data come from? How long is it valid and appropriate?). In this article, we argue two points. First, the decisions we make during data collection and preparation profoundly impact the robustness, fairness, and interpretability of the systems we build. Second, our responsibility for the operation of these systems does not stop when they are deployed.

Example: Automated hiring systems. To make our discussion concrete, consider the use of predictive analytics in hiring. Automated hiring systems are seeing ever broader use and are as varied as the hiring practices themselves, ranging from resume screeners that claim to identify promising applicants[a] to video and voice analysis tools that facilitate the interview process[b] and game-based assessments that promise to surface personality traits indicative of future success.[c] Bogen and Rieke[5] describe the hiring process from the employer's point of view as a series of decisions that forms a funnel, with stages corresponding to

---

[a] https://www.crystalknows.com
[b] https://www.hirevue.com
[c] https://www.pymetrics.ai

IMAGE BY PAVLO PLAKHOTTIA

sourcing, screening, interviewing, and selection. (Figure 1 depicts a slightly reinterpreted version of that funnel.)

The popularity of automated hiring systems is due in no small part to our collective quest for efficiency. In 2019 alone, the global market for artificial intelligence (AI) in recruitment was valued at $580 million.[d] Employers choose to use these systems to source and screen candidates faster, with less paperwork, and, in the post-COVID-19 world, as little in-person contact as is practical. Candidates are promised a more streamlined job-search experience, although they rarely have a say in whether they are screened by a machine.

The flip side of efficiency afforded by automation is that we rarely understand how these systems work and, indeed, whether they work. Is a resumé screener identifying promising candidates or is it picking up irrelevant—or even discriminatory—patterns from historical data, limiting access to essential economic opportunity for entire segments of the population and potentially exposing an employer to legal liability? Is a job seeker participating in a fair competition if she is being systematically screened out, with no opportunity for human intervention and recourse, despite being well-qualified for the job?

If current adoption trends are any indication, automated hiring systems are poised to impact each one of us—as employees, employers, or both. What's

» key insights

- **Responsible data management involves incorporating ethical and legal considerations across the life cycle of data collection, analysis, and use in all data-intensive systems, whether they involve machine learning and AI or not.**
- **Decisions during data collection and preparation profoundly impact the robustness, fairness, and interpretability of data-intensive systems. We must consider these earlier life cycle stages to improve data quality, control for bias, and allow humans to oversee the operation of these systems.**
- **Data alone is insufficient to distinguish between a distorted reflection of a perfect world, a perfect reflection of a distorted world, or a combination of both. The assumed or externally verified nature of the distortions must be explicitly stated to allow us to decide whether and how to mitigate their effects.**

---

[d] https://www.industryarc.com/Report/19231/artificial-intelligence-in-recruitmentmarket.html

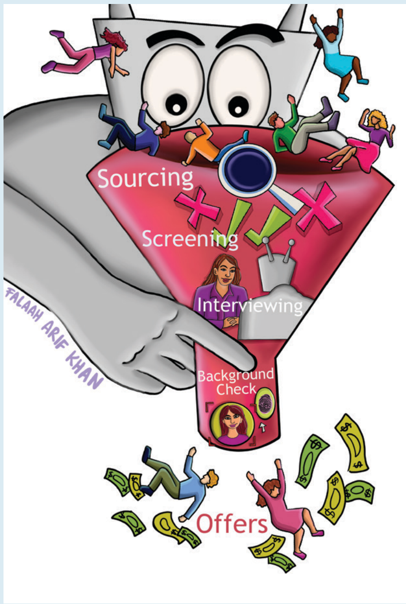


Figure 1. The hiring funnel is an example of an automated decision system—a data-driven, algorithm-assisted process that culminates in job offers to some candidates and rejections to others.

more, many of us will be asked to help design and build such systems. Yet, their widespread use far outpaces our collective ability to understand, verify, and oversee them. This is emblematic of a broader problem: the widespread and often rushed adoption of *automated decision systems* (ADSs) without an appropriate prior evaluation of their effectiveness, legal compliance, and social sustainability.

Defining ADSs. There is currently no consensus as to what an ADS is or is not, though proposed regulation in the European Union (EU), several U.S. states, and other jurisdictions are beginning to converge on some factors to consider: the degree of human discretion in the decision, the level of impact, and the specific technologies involved. As an example of the challenges, Chapter 6 of the New York City ADS Task Force report[e] summarizes a months-long struggle to, somewhat ironically, define its own mandate: to craft a definition that captures the breadth of ethical and legal concerns, yet remains practically useful. Our view is to lean towards breadth, but to tailor operational requirements and oversight mechanisms for an ADS depending on application domain and context of use, level of impact,[34] and relevant legal and regulatory requirements. For example, the use of ADSs in hiring and employment is subject to different concerns than their use in credit and lending. Further, the potential harms will be different depending on whether an ADS is used to advertise employment or financial opportunities or to help make decisions about whom to hire and to whom a loan should be offered.

To define ADS, we may start with some examples. Figure 1's hiring funnel and associated components, such as an automated resume screening tool and a tool that matches job applicants with positions, are natural examples of ADSs. But is a calculator an ADS? No, because it is not qualified with a context of use. Armed with these examples, we propose a pragmatic definition of ADSs:

- They process data about people, some of which may be sensitive or proprietary
- They help make decisions that are consequential to people's lives and livelihoods
- They involve a combination of human and automated decision-making
- They are designed to improve efficiency and, where applicable, promote equitable access to opportunity

In this definition, we deliberately direct our attention toward systems in which the ultimate decision-making responsibility is with a human and away from fully autonomous systems, such as self-driving cars. Advertising systems are ADSs; while they may operate autonomously, the conditions of their operation are specified and reviewed via negotiations between platform providers and advertisers. Further, regulation is compelling ever closer human oversight and involvement in the operations of such systems. Actuarial models, music recommendation systems, and health screening tools are all ADSs as well.

Why responsible data management? The placement of technical components that assist in decision-making—a spreadsheet formula, a matchmaking algorithm, or predictive analytics—within the *life cycle of data collection and analysis* is central to defining an ADS. This, in turn, uniquely positions the data-management community to deliver true practical impact in the responsible design, development, use, and oversight of these systems. Because data-management technology offers a natural, centralized point for enforcing policies, we can develop methodologies to enforce requirements transparently and explicitly through the life cycle of an ADS. Due to the unique blend of theory and systems in our methodological toolkit, we can help inform regulation by studying the feasible tradeoffs between different classes of legal and efficiency requirements. Our pragmatic approach enables us to support compliance by developing standards for effective and efficient auditing and disclosure, and by developing protocols for embedding these standards in systems.

In this article, we assert that the data-management community should play a central role in responsible ADS design, development, use, and oversight. Automated decision systems may or may not use AI, and they may or may not operate with a high degree of autonomy, but they all rely heavily on data. To set the stage for our discussion, we begin by interpreting the term "bias" (Section 2). We then discuss the data management-related challenges of ADS oversight and embedding responsibility into ADS life cycle management, pointing out specific opportunities for novel research contributions. Our focus is on specific issues where there is both a well-articulated need and strong evidence that technical interventions are possible. Fully addressing all the issues we raise requires socio-technical solutions that go beyond the scope of what we can do with technology alone. Although vital, since our focus is on technical data-management interventions, we do not discuss such socio-technical solutions in this article.

Crucially, the data-management problems we seek to address are not purely technical. Rather, they are socio-legal-technical. It is naïve to expect that purely technical solutions will suffice, so we must step outside our engineering comfort zone and start reasoning in terms of values and beliefs, in addition to checking results against known ground truths and optimizing for efficiency objectives. This seems

---

[e] https://www1.nyc.gov/site/adstaskforce/index.page

high-risk, but one of the upsides is being able to explain to our children what we do and why it matters.

All About That Bias

We often hear that an ADS, such as an automated hiring system, operates on "biased data" and results in "biased outcomes." What is the meaning of the term "bias" in this context, how does it exhibit itself through the ADS life cycle, and what does data-management technology have to offer to help mitigate it?

Bias in a general sense refers to systematic and unfair discrimination against certain individuals or groups of individuals in favor of others. In their seminal 1996 paper, Friedman and Nissenbaum identified three types of bias that can arise in computer systems: *preexisting*, *technical*, and *emergent*.[12] We discuss each of these in turn in the remainder of this section, while also drawing on a recent fine-grained taxonomy of bias, with insightful examples that concern social media platforms, from Olteanu et al.[26]

Preexisting bias. This type of bias has its origins in society. In data-science applications, it exhibits itself in the input data. Detecting and mitigating preexisting bias is the subject of much research under the heading of algorithmic fairness.[8] Importantly, the presence or absence of this type of bias cannot be scientifically verified; rather, it must be postulated based on a belief system.[11] Consequently, the effectiveness—or even the validity—of a technical attempt to mitigate preexisting bias is predicated on that belief system. To explain preexisting bias and the limits of technical interventions, such as data debiasing, we find it helpful to use the mirror reflection metaphor, depicted in Figure 2.

The mirror metaphor. Data is a mirror reflection of the world. When we think about preexisting bias in the data, we interrogate this reflection, which is often distorted. One possible reason is that the mirror (the measurement process) introduces distortions. It faithfully represents some portions of the world, while amplifying or diminishing others. Another possibility is that even a perfect mirror can only reflect a distorted world—a world such as it is, and not as it could or should be.

The mirror metaphor helps us make several simple but important observations. First, based on the reflection alone, and without knowledge about the properties of the mirror and of the world it reflects, we cannot know whether the reflection is distorted, and, if so, for what reason. That is, data alone cannot tell us whether it is a distorted reflection of a perfect world, a perfect reflection of a distorted world, or whether these distortions compound. The assumed or externally verified nature of the distortions must be explicitly stated, to allow us to decide whether and how to mitigate their effects. Our second observation is that it is up to people—individuals, groups, and society at large—and not data or algorithms, to come to a consensus about whether the world is how it should be or if it needs to be improved and, if so, how we should go about improving it. The third and final observation is that, if data is used to make important decisions, such as who to hire and what salary to offer, then compensating for distortions is worthwhile. But the mirror metaphor only takes us so far. We must work much harder—usually going far beyond technological solutions—to propagate the changes back into the world and not merely brush up the reflection.[37]

As an example of preexisting bias in hiring, consider the use of an applicant's Scholastic Assessment Test (SAT) score during the screening stage. It has been documented that the mean score of the math section of the SAT, as well as the shape of the score distribution, differs across racial groups.[28] If we believed that standardized test scores were sufficiently impacted by preparation courses and that the score itself says more about socioeconomic conditions than an individual's academic potential, then we would consider the data to be biased. We may then seek to correct for that bias before using the feature, for example, by selecting the top-performing individuals of each racial group, or by using a more sophisticated *fair ranking method* in accordance with our beliefs about the nature of the bias and with our bias mitigation goals.[40] Alternatively, we may disregard this feature altogether.

Technical bias. This type of bias arises due to the operation of the technical system itself, and it can amplify

Figure 2. Data as a mirror reflection of the world,[37] illustrated by Falaah Arif Khan.

(a) Data is an image of the world, its mirror reflection. When we think about bias in the data, we interrogate this reflection. Does the data systematically over-represent or under-represent some parts of the world, or does it distort reality in some other way?

(b) Even if we were able to reflect the world perfectly in the data, it would still be a reflection of the world as it is, not how it could or should be. People—not data or algorithms—must decide what world we want to live in.

(c) Debiasing data may be worthwhile if we are about to base consequential decisions on that data. But we must be careful to propagate the changes back into the world, not merely touch up its reflection.

preexisting bias. Technical bias, particularly when it is due to preprocessing decisions or post-deployment issues in data-intensive pipelines, has been noted as problematic,[23,26,33] but it has so far received limited attention when it comes to diagnostics and mitigation techniques. We now give examples of potential sources of technical bias in several ADS life cycle stages, which are particularly relevant to data management.

Data cleansing. Methods for missing-value imputation that are based on incorrect assumptions about whether data is missing at random may distort protected group proportions. Consider a form that gives job applicants a binary gender choice but also allows gender to be unspecified. Suppose that about half of the applicants identify as men and half as women, but that women are more likely to omit gender. If mode imputation—replacing a missing value with the most frequent value for the feature, a common setting in scikit-learn—is applied, then all (predominantly female) unspecified gender values will be set to male. More generally, multiclass classification for missing-value imputation typically only uses the most frequent classes as target variables,[4] leading to a distortion for small groups, because membership in these groups will not be imputed.

Next, suppose that some individuals identify as non-binary. Because the system only supports male, female, and unspecified as options, these individuals will leave gender unspecified. If mode imputation is used, then their gender will be set to male. A more sophisticated imputation method will still use values from the active domain of the feature, setting the missing values of gender to either male or female. This example illustrates that bias can arise from an incomplete or incorrect choice of data representation. While dealing with null values is known to be difficult and is already considered among the issues in data cleansing, the needs of responsible data management introduce new problems. It has been documented that data-quality issues often disproportionately affect members of historically disadvantaged groups,[20] so we risk compounding technical bias due to data representation with bias due to statistical concerns.

The flip side of efficiency afforded by automation is that we rarely understand how these systems work and, indeed, whether they work.

Other data transformations that can introduce skew include text normalization, such as lowercasing, spell corrections, or stemming. These operations can be seen as a form of aggregation, in effect collapsing terms with different meanings under the same representation. For example, lowercasing "Iris," a person's name, as "iris" will make it indistinguishable from the name of a flower or from the membrane behind the cornea of the eye, while stemming the terms "[tree] leaves" and "[he is] leaving" will represent both as "leav."[26]

Other examples of aggregation that can lead to data distribution changes include "zooming out" spatially or temporally: replacing an attribute value with a coarser geographic or temporal designation or mapping a location to the center of the corresponding geographical bounding box.[26]

Filtering. Selections and joins are commonly used as part of data preprocessing. A selection operation checks each data record against a predicate—for instance, U.S. address ZIP code is 10065 or age is less than 30—and retains only those records that match the predicate. A join combines data from multiple tables—for example, creating a record that contains a patient's demographics and clinical records using the social security number attribute contained in both data sources as the join key. These operations can arbitrarily change the proportion of protected groups (for example, female gender) even if they do not directly use the sensitive attribute (for example, gender) as part of the predicate or the join key. For example, selecting individuals whose mailing address ZIP code is 10065—one of the most affluent locations on Manhattan's Upper East Side—may change the data distribution by race. Similarly, joining patient demographic data with clinical records may introduce skew by age, with fewer young individuals having matching clinical records. These changes in proportion may be unintended but are important to detect, particularly when they occur during one of many preprocessing steps in the ADS pipeline.

Another potential source of technical bias is the use of pretrained word embeddings. For example, a pipeline may replace a textual name feature with the corresponding vector from a word embedding that is missing for rare, non-Western names. If we then filter out records for which no embedding was found, we may disproportionately remove individuals from specific ethnic groups.

Ranking. Technical bias can arise when results are presented in ranked order, such as when a hiring manager is considering potential candidates to invite for in-person interviews. The main reason is inherent position bias—the geometric drop in visibility for items at lower ranks compared to those at higher ranks—which arises because in Western cultures we read from top to bottom and from left to right: Items in the top-left corner of the screen attract more attention.[3] A practical implication is that, even if two candidates are equally suitable for the job, only one of them can be placed above the other, which implies prioritization. Depending on the application's needs and on the decision-maker's level of technical sophistication, this problem can be addressed by suitably randomizing the ranking, showing results with ties, or plotting the score distribution.

Emergent bias. This type of bias arises in the context of use of the technical system. In Web ranking and recommendation in e-commerce, a prominent example is "rich-get-richer": searchers tend to trust systems to show them the most suitable items at the top positions, which in turn shapes a searcher's idea of a satisfactory answer.

This example immediately translates to hiring and employment. If hiring managers trust recommendations from an ADS, and if these recommendations systematically prioritize applicants of a particular demographic profile, then a feedback loop will be created, further diminishing workforce diversity over time. Bogen and Rieke[5] illustrate this problem: "For example, an employer, with the help of a third-party vendor, might select a group of employees who meet some definition of success—for instance, those who 'outperformed' their peers on the job. If the employer's performance evaluations were themselves biased, favoring men, then the resulting model might predict that men are more likely to be high performers than women, or make more errors when evaluating women."

Emergent bias is particularly difficult to detect and mitigate, because it refers to the impacts of an ADS outside the systems' direct control. We will cover this in the "Overseeing ADS" section.

Managing the ADS Data Life Cycle

Automated decision systems critically depend on data and should be seen through the lens of the *data life cycle*.[19] Responsibility concerns, and important decision points, arise in data sharing, annotation, acquisition, curation, cleansing, and integration. Consequently, substantial opportunities for improving data quality and representativeness, controlling for bias, and allowing humans to oversee the process are missed if we do not consider these earlier life cycle stages.

Database systems centralize correctness constraints to simplify application development with the help of schemas, standards, and transaction protocols. As algorithmic fairness and interpretability emerge as first-class requirements, there is a need to develop generalized solutions that embed them as constraints and that work across a range of applications. In what follows, we highlight promising examples of our own recent and ongoing work that is motivated by this need. These examples underscore that tangible technical progress is possible and that much work remains to be done to offer systems support for the responsible management of the ADS life cycle. These examples are not intended to be exhaustive, but merely illustrate technical approaches that apply to different points of the data life cycle. Additional examples, and research directions, are discussed in Stoyanovich et al.[37] Before diving into the details, we recall the previously discussed mirror-reflection metaphor, as a reminder of the limits of technical interventions.

Data acquisition. Consider the use of an ADS for pre-screening employment applications. Historical underrepresentation of women and minorities in the workforce can lead to an underrepresentation of these groups in the training set, which in turn could push the ADS to reject more minority applicants or, more generally, to exhibit disparate predictive accuracy.[7] It is worth noting that the problem here is not only that some minorities are proportionally under-represented, but also that the absolute representation of some groups is low. Having 2% African Americans in the training set is a problem when they constitute 13% of the population. But it is also a problem to have only 0.2% Native Americans in the training set, even if that is representative of their proportion in the population. Such a low number can lead to Native Americans being ignored by the ADS as a small "outlier" group.

To mitigate low absolute representation, Asudeh et al.[2] assess the coverage of a given dataset over multiple categorical features. An important question for an ADS vendor is, then, what can it do about the lack of coverage. The proposed answer is to direct them to acquire more data, in a way that is cognizant of the cost of data acquisition. Asudeh et al.[2] use a threshold to determine an appropriate level of coverage and experimentally demonstrate an improvement in classifier accuracy for minority groups when additional data is acquired.

This work addresses a step in the ADS life cycle upstream from model training and shows how improving data representativeness can improve accuracy and fairness, in the sense of disparate predictive accuracy.[7] There are clear future opportunities to integrate coverage-enhancing interventions more closely into ADS life cycle management, both to help orchestrate the pipelines and, perhaps more importantly, to make data acquisition task-aware, setting coverage objectives based on performance requirements for the specific predictive analytics downstream rather than based on a global threshold.

Data preprocessing. Even when the acquired data satisfies representativeness requirements, it may still be subject to preexisting bias, as discussed in the "Preexisting bias" section. We may thus be interested in developing interventions to mitigate these effects. The algorithmic fairness community has developed dozens of methods for data and model de-biasing, yet the vast majority of these methods take an *associational interpretation of fairness* that is solely based on data, without reference to additional structure or context. In what follows, we present two recent examples of work that take a causal interpretation of fairness: a database repair framework for fair classification by Salimi et al.[29] and a framework for fair ranking that mitigates intersectional discrimination by Yang et al.[38] We focus on examples of causal fairness notions here because they correspond very closely to the methodological toolkit of data management by making explicit the use of structural information and constraints.

Causal fairness approaches—for example, Kilbertus et al.[21] and Kusner et al.[22]—capture background knowledge as causal relationships between variables, usually represented as causal DAGs, or directed acyclic graphs, in which nodes represent variables, and edges represent potential causal relationships. Consider the task of selecting job applicants at a moving company and the corresponding causal model in Figure 3, an example inspired by Datta et al.[10] Applicants are hired based on their qualification score Y, computed from weight-lifting ability X, and affected by gender G and race R, either directly or through X. By representing relationships between features in a causal DAG, we gain an ability to postulate which relationships between features and outcomes are legitimate and which are potentially discriminatory. In our example, the impact of gender (G) on the decision to hire an individual for a position with a moving company (Y) may be considered admissible if it flows through the node representing weight-lifting ability (X). On the other hand, the direct impact of gender on the decision to hire would constitute direct discrimination and would thus be considered inadmissible.

Salimi et al.[29] introduced a measure called *interventional fairness* for classification and showed how to achieve it based on observational data, without requiring the complete causal model. The authors consider the Markov boundary (MB)—parents, children, children's other parents—of a variable Y, which describes whether those nodes can potentially influence Y. Their key result is that the algorithm satisfies interventional fairness if the MB of the outcome is a subset of the MB of the admissible variables—that is, admissible variables "shield" the outcome from the influence of sensitive and inadmissible variables. This condition on the MB is used to design *database repair algorithms*, through a connection between the independence constraints encoding fairness and multivalued dependencies (MVD) that can be checked using the training data. Several repair algorithms are described, and the results show that in addition to satisfying interventional fairness, the classifier trained on repaired data performs well against associational fairness metrics.

The data management problems we are looking to address are not purely technical. Rather, they are socio-legal-technical.

As another example of a data preprocessing method that makes explicit use of structural assumptions, Yang et al.[38] developed a causal framework for *intersectionally fair ranking*. Their motivation is that it is possible to give the appearance of being fair with respect to each sensitive attribute, such as race and gender separately, while being unfair with respect to intersectional subgroups.[9] For example, if fairness is taken to mean proportional representation among the top-k, it is possible to achieve proportionality for each gender subgroup (for instance, men and women) and for each racial subgroup (for example, Black and White), while still having inadequate representation for a subgroup defined by the intersection of both attributes (for example, Black women). The gist of the methods of Yang et al.[38] is to use a causal model to compute model-based *counterfactuals*, answering the question: "What would this person's score be if she had been a Black woman (for example)?" and then ranking on counterfactual scores to achieve intersectional fairness.

Data-distribution debugging. We now return to our discussion of technical bias and consider data-distribution shifts, which may arise during data preprocessing and impact machine learning-model performance downstream. In contrast to important prior work on data-distribution shift detection in deployed models—for instance, Rabanser et al.[27]—our focus is explicitly on data manipulation, a cause of data-distribution shifts that has so far been overlooked. We will illustrate how this type of bias can arise and will suggest an intervention: a data-distribution debugger that helps surface technical bias, allowing a data scientist to mitigate it.[33]

Consider Ann, a data scientist at a job-search platform that matches profiles of job seekers with openings for which they are well-qualified and in which they may be interested. A job seeker's interest in a position is estimated based on several factors, including the salary and benefits being offered. Ann uses applicants' resumes, self-reported demographics, and employment histories as input. Following her company's best practices, she starts by splitting her dataset into training, validation, and test sets. Ann then uses pandas, scikit-learn, and accompanying data transformers to explore the data and implement data preprocessing, model selection, tuning, and validation. Ann starts preprocessing by computing value distributions and correlations for the features in the dataset and identifying missing values. She will use a default imputation method in scikit-learn to fill these in, replacing missing values with the mode value for that feature. Finally, Ann implements model selection and hyperparameter tuning, selecting a classifier that displays sufficient accuracy.

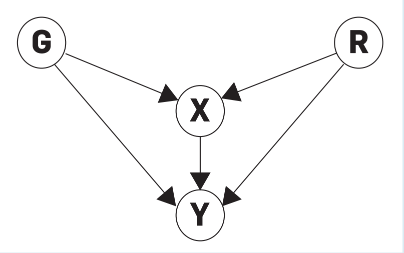
When Ann more closely considers the performance of the classifier, she observes a disparity in predictive accuracy:[7] Accuracy is lower for older job seekers, who are frequently matched with lower-paying positions than they would expect. Ann now needs to understand why this is the case, whether any of her technical choices during pipeline construction contributed to this disparity, and what she can do to mitigate this effect.

It turns out that this issue was the result of a *data-distribution bug*—a shift in the values of a feature that is important for the prediction and that is the result of a technical choice during pre-processing. Here, that feature is the number of years of job experience. The bug was introduced because of Ann's assumption that the values of this feature are *missing at random* and because of her choice to use mode imputation, which is consistent with this assumption. In fact, values were missing more frequently for older job seekers: They would not enter a high value in "years of experience" because they might be afraid of age discrimination. This observation is consistent with the intuition that individuals are more likely to withhold information that may disadvantage them. Taken together, these two factors resulted in imputed years-of-experience values skewing lower, leading to a lower salary-requirement estimate and impacting older applicants more than younger ones.

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, a composable and nestable abstraction for combining operations on array data which originates from scikit-learn and is executed in a hard-to-debug manner with nested function calls.

Grafberger et al. designed and implemented mlinspect,[15] a lightweight data-distribution debugger that supports automated inspection of data-intensive pipelines to detect the accidental introduction of statistical bias and linting for best practices. The mlinspect library extracts logical query plans—modeled as DAGs of preprocessing operators—from pipelines that use popular libraries, such as pandas and scikit-learn, and combines relational operations and estimator/transformer pipelines. The library automatically instruments the code and traces the impact of operators on properties, such as the distribution of sensitive groups in the data. mlinspect is a necessary first step in what we hope will be a long line of work in collectively developing data-science best practices and the tooling to support their broad adoption. Much important work remains to allow us to start treating data as a first-class citizen in software development.

Figure 3. Causal model includes sensitive attributes: *G* (gender), *R* (race), *X* (weight-lifting ability), and *Y* (utility score).



Overseeing ADS

We are in the midst of a global trend to regulate the use of ADSs. In the EU, the General Data Protection Regulation (GDPR) offers individuals protections regarding the collection, processing, and movement of their personal data, and applies broadly to the use of such data by governments and private-sector entities. Regulatory activity in several countries outside of the EU, notably Japan and Brazil, is in close alignment with the GDPR. In the U.S., many major cities, a handful of states, and the Federal government are establishing task forces and issuing guidelines about responsible development and technology use. With its focus on data rights and data-driven decision-making, the GDPR is, without a doubt, the most significant piece of technology regulation to date, serving as a "common denominator" for the oversight of data collection and usage, both in the EU and worldwide. For this reason, we will discuss the GDPR in some depth in the remainder of this section.

The GDPR aims to protect the rights and freedoms of natural persons with regard to how their personal data is processed, moved, and exchanged (Article 1). The GDPR is broad in scope and applies to "the processing of personal data wholly or partly by automated means" (Article 2), both in the private and public sectors. Personal data is broadly construed and refers to any information relating to an identified or identifiable natural person, called the *data subject* (Article 4). The GDPR aims to give data subjects insight into, and control over, the collection and processing of their personal data. Providing such insight, in response to the "right to be informed," requires technical methods for interpretability, discussed in the following section, "Interpretability for a range of stakeholders." We will also highlight, in the upcoming section, "Removing personal data," the right to erasure as a representative example of a regulatory requirement that raises a concrete data-management challenge. Additional details can be found in Abiteboul and Stoyanovich.[1]

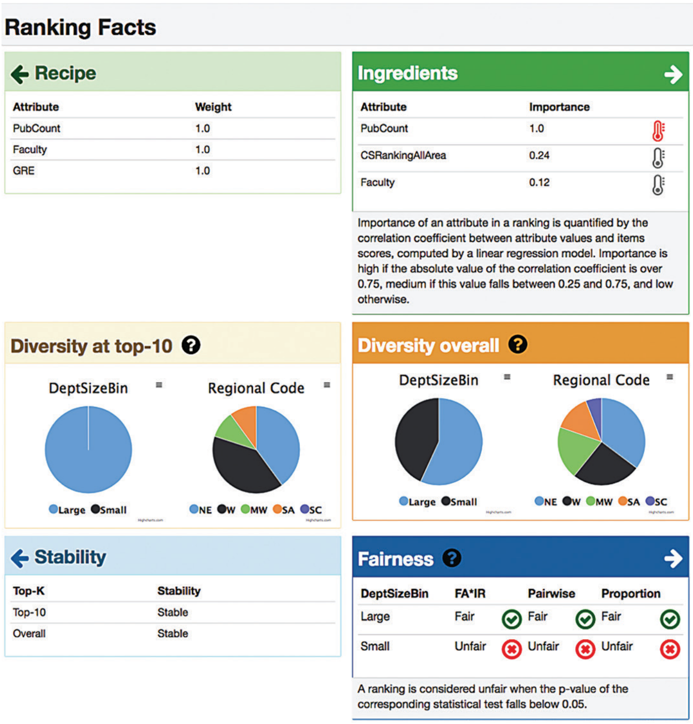
As we have done throughout this article, we highlight specific challenges within the broad topic of ADS oversight and outline promising directions for technical work to address these challenges. It is important to keep in mind that ADS oversight will not admit a purely technical solution. Rather, we hope that technical interventions will be part of a robust distributed infrastructure of accountability, in which multiple stakeholder groups participate in ADS design, development, and oversight.

Interpretability for a range of stakeholders. Interpretability—allowing people to understand the process and decisions of an ADS—is critical to the responsible use of these systems. Interpretability means different things to different stakeholders, yet the common theme is that it allows people, including software developers, decision-makers, auditors, regulators, individuals who are affected by ADS decisions, and members of the public at large, to exercise agency by accepting or challenging algorithmic decisions and, in the case of decision-makers, to take responsibility for these decisions.

Interpretability rests on making explicit the interactions between the computational process and the data on which it acts. Understanding how code and data interact is important both when an ADS is interrogated for bias and discrimination, and when it is asked to explain an algorithmic decision that affects an individual.

To address the interpretability needs of different stakeholders, several recent projects have been developing tools based on the concept of a nutritional label—drawing an analogy to the food industry, where simple, standard labels convey information about ingredients and production processes. Short of setting up a chemistry lab, a food consumer would otherwise have no access to this information. Similarly, consumers of data products or individuals affected by ADS decisions cannot be expected to reproduce the data collection and computational procedures. These projects include the Dataset Nutrition Label,[18] Datasheets for Datasets,[13] Model Cards,[25] and Ranking Facts,[39] which all use specific kinds of metadata to support interpretability. Figure 4 offers an example of a nutritional label; it presents Ranking Facts[39] to explain a ranking of computer science departments.

Figure 4. Ranking Facts for the CS department's dataset.



Ranking Facts

Recipe

Attribute	Weight
PubCount	1.0
Faculty	1.0
GRE	1.0

Ingredients

Attribute	Importance
PubCount	1.0
CSRankingAllArea	0.24
Faculty	0.12

Importance of an attribute in a ranking is quantified by the correlation coefficient between attribute values and items scores, computed by a linear regression model. Importance is high if the absolute value of the correlation coefficient is over 0.75, medium if this value falls between 0.25 and 0.75, and low otherwise.

Diversity at top-10

Diversity overall

Stability

Top-K	Stability
Top-10	Stable
Overall	Stable

Fairness

DeptSizeBin	FA*IR		Pairwise		Proportion	
Large	Fair	✓	Fair	✓	Fair	✓
Small	Unfair	✗	Unfair	✗	Unfair	✗

A ranking is considered unfair when the p-value of the corresponding statistical test falls below 0.05.

In much of this work, nutritional labels are manually constructed, and they describe a single component in the data life cycle, typically a dataset or a model. Yet, to be broadly applicable, and to faithfully represent the computational process and the data on which it acts, nutritional labels should be generated *automatically* or *semiautomatically* as a side effect of the computational process itself, embodying the paradigm of *interpretability by design*.[36] This presents an exciting responsible data-management challenge.

The data-management community has been studying systems and standards for metadata and provenance for decades.[17] This includes work on fine-grained provenance, where the goal is to capture metadata associated with a data product and propagate it through a series of transformations, to explain its origin and history of derivation, and to help answer questions about the robustness of the computational process and the trustworthiness of its results. There is now an opportunity to revisit many of these insights and to extend them to support the interpretability needs of different stakeholders, both technical and non-technical.

Removing personal data. The right to be forgotten is originally motivated by the desire of individuals not to be perpetually stigmatized by something they did in the past. Under pressure from despicable social phenomena such as revenge porn, it was turned into law in 2006 in Argentina, and since then in the EU, as part of the GDPR (Article 17), stating that data subjects have the right to request the timely erasure of their personal data.

An important technical issue of clear relevance to the data-management community is deletion of information in systems that are designed explicitly to accumulate data. Making data-processing systems GDPR-compliant has been identified as one of the data-management community's key research challenges.[35] The requirement of efficient deletion is in stark contrast with the typical requirements for data-management systems, necessitating substantial rethinking and redesign of the primitives, such as enhancing fundamental data structures with efficient delete operations.[30]

Data deletion must be both permanent and deep, in the sense that its effects must propagate through data dependencies. To start, it is difficult to guarantee that all copies of every piece of deleted data have actually been deleted. Further, when some data is deleted, the remaining database may become inconsistent, and may, for example, include dangling pointers. Additionally, production systems typically do not include a strong provenance mechanism, so they have no means of tracking the use of an arbitrary data item (one to be deleted) and reasoning about the dependencies on that data item in derived data products. Although much of the data-management community's attention over the years has been devoted to tracking and reasoning about provenance, primarily in relational contexts and in workflows (see Herschel et al.[17] for a recent survey), there is still important work to be done to make these methods both practically feasible and sufficiently general to accommodate current legal requirements.

An important direction that has only recently come into the academic community's focus concerns ascertaining the effects of a deletion on downstream processes that are not purely relational but include other kinds of data analysis tasks, such as data mining or predictive analytics. Recent research[14,31] argues that it is not sufficient to merely delete personal user data from primary data stores such as databases, but that machine-learning models trained on stored data also fall under the regulation. This view is supported by Recital 75 of the GDPR: "The risk to the rights and freedoms of natural persons … may result from personal data processing … where personal aspects are evaluated, in particular analyzing or predicting aspects concerning performance at work, economic situation, health, personal preferences or interests, reliability or behavior, location or movements." The machine-learning community has been working on this issue under the umbrella of *machine unlearning*.[6,14] Given a model, its training data, and a set of user data to delete/unlearn, the community proposes efficient ways to accelerate the retraining of the model. However, these approaches ignore the constraints imposed by the complexity of production set-ups (such as redeployment costs) and are thereby hard to integrate into real-world ML applications.[32]

We must learn to step outside our engineering comfort zone and to start reasoning in terms of values and beliefs.

Requests for deletion may also conflict with other laws, such as requirements to keep certain transaction data for some period or requirements for fault tolerance and recoverability. Understanding the impact of deletion requests on our ability to offer guarantees on system resilience and performance, and developing appropriate primitives and protocols for practical use, is another call to action for the data-management community.

Conclusion

In this article, we offered a perspective on the role that the data-management research community can play in the responsible design, development, use, and oversight of ADSs. We grounded our discussion in automated hiring tools, a specific use case that gave us ample opportunity to appreciate the potential benefits of data science and AI in an important domain and to get a sense of the ethical and legal risks.

An important point is that we cannot fully automate responsibility. While some of the duties of carrying out the task of, say, legal compliance can in principle be assigned to an algorithm, accountability for the decisions being made by an ADS always rests with a person. This person may be a decision-maker or a regulator, a business leader or a software developer. For this reason, we see our role as researchers in helping build systems that "expose the knobs" or responsibility to people.

Those of us in academia have an

additional responsibility to teach students about the social implications of the technology they build. Typical students are driven to develop technical skills and have an engineer's desire to build useful artifacts, such as a classification algorithm with low error rates. They are also increasingly aware of historical discrimination that can be reinforced, amplified, and legitimized with the help of technical systems. Our students will soon become practicing data scientists, influencing how technology companies impact society. It is our responsibility as educators to equip them with the skills to ask and answer the hard questions about the choice of a dataset, a model, or a metric. It is critical that the students we send out into the world understand responsible data science.

Toward this end, we are developing educational materials and teaching courses on responsible data science. H.V. Jagadish launched the first Data Science Ethics MOOC on the EdX platform in 2015. This course has since been ported to Coursera and FutureLearn, and it has been taken by thousands of students worldwide. Individual videos are licensed under Creative Commons and can be freely incorporated in other courses where appropriate. Julia Stoyanovich teaches highly visible technical courses on Responsible Data Science,[24] with all materials publicly available online. These courses are accompanied by a comic book series, developed under the leadership of Falaah Arif Khan, as supplementary reading.

In a pre-course survey, in response to the prompt, "Briefly state your view of the role of data science and AI in society", one student wrote: "It is something we cannot avoid and therefore shouldn't be afraid of. I'm glad that as a data science researcher, I have more opportunities as well as more responsibility to define and develop this 'monster' under a brighter goal." Another student responded, "Data Science [DS] is a powerful tool and has the capacity to be used in many different contexts. As a responsible citizen, it is important to be aware of the consequences of DS/AI decisions and to appropriately navigate situations that have the risk of harming ourselves or others."

Acknowledgments

This work was supported in part by NSF Grants No. 1934464, 1934565, 1934405, 1926250, 1741022, 1740996, 1916505, by Microsoft, and by Ahold Delhaize. All content represents the opinion of the authors and is not necessarily shared or endorsed by their respective employers or sponsors.

References

1. Abiteboul, S. and Stoyanovich, J. Transparency, fairness, data protection, neutrality: Data management challenges in the face of new regulation. *J. of Data and Information Quality 11*, 3 (2019), 15:1–15:9.
2. Asudeh, A., Jin, Z., and Jagadish, H.V. Assessing and remedying coverage for a given dataset. In *35th IEEE International Conference on Data Engineering* (April 2019), 554–565.
3. Baeza-Yates, R. Bias on the web. *Communications of the ACM 61*, 6 (2018), 54–61.
4. Biessmann, F., Salinas, D., Schelter, S., Schmidt, P., and Lange, D. Deep learning for missing value imputation in tables with non-numerical data. In *Proceedings of the 27th ACM Intern. Conf. on Information and Knowledge Management* (2018), 2017–2025.
5. Bogen, M. and Rieke, A. Help wanted: An examination of hiring algorithms, equity, and bias. *Upturn* (2018).
6. Cauwenberghs, G. and Poggio, T. Incremental and decremental support vector machine learning. *NeurIPS* (2001), 409–415.
7. Chen, I., Johansson, F., and Sontag, D. Why is my classifier discriminatory? S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018*, 3543–3554.
8. Chouldechova, A. and Roth, A. A snapshot of the frontiers of fairness in machine learning. *Communications of the ACM 63*, 5 (2020), 82–89.
9. Crenshaw, K. Demarginalizing the intersection of race and sex: A Black feminist critique of antidiscrimination doctrine, feminist theory and antiracist politics. *University of Chicago Legal Forum 1* (1989), 139–167.
10. Datta, A., Sen, S., and Zick, Y. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *IEEE Symposium on Security and Privacy* (May 2016), 598–617.
11. Friedler, S., Scheidegger, C., and Venkatasubramanian, S. The (im)possibility of fairness: Different value systems require different mechanisms for fair decision making. *Communications of the ACM 64*, 4 (2021), 136–143.
12. Friedman, B. and Nissenbaum, H. Bias in computer systems. *ACM Transactions on Information Systems 14*, 3 (1996), 330–347.
13. Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J., Wallach, H., Daumé III, H., and Crawford, K. Datasheets for datasets. *CoRR* (2018), abs/1803.09010.
14. Ginart, A., Guan, M., Valiant, G., and Zou, J. Making AI forget you: Data deletion in machine learning. In *NeurIPS* (2019), 3513–3526.
15. Grafberger, S., Stoyanovich, J., and Schelter, S. Lightweight inspection of data preprocessing in native machine learning pipelines. In *11th Conf. on Innovative Data Sys. Research, Online Proceedings* (January 2021), http://www.cidrdb.org.
16. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. A survey of methods for explaining black box models. *ACM Computing Surveys 51*, 5 (2019), 93:1–93:42.
17. Herschel, M., Diestelkämper, R., and Ben Lahmar, H. A survey on provenance: What for? What form? What from? *VLDB Journal 26*, 6 (2017), 881–906.
18. Holland, S., Hosny, A., Newman, S., Joseph, J., and Chmielinski, K. The dataset nutrition label: A framework to drive higher data quality standards. *CoRR* (2018), abs/1805.03677.
19. Jagadish, H.V., Gehrke, J., Labrinidis, A., Papakonstantinou, Y., Patel, J., Ramakrishnan, R., and Shahabi, C. Big data and its technical challenges. *Communications of the ACM 57*, 7 (2014), 86–94.
20. Kappelhof, J. Survey research and the quality of survey data among ethnic minorities. In *Total Survey Error in Practice*, Wiley (2017).
21. Kilbertus, N., Carulla, M., Parascandolo, G., Hardt, M., Janzing, D., and Schölkopf, B. Avoiding discrimination through causal reasoning. In *Advances in Neural Information Processing Systems* (2017), 656–666.
22. Kusner, M., Loftus, J., Russell, C., and Silva, R. Counterfactual fairness. I. Guyon, U. von Luxburg, S. Bengio, H.M. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, In *Advances in Neural Information Processing Systems 30:* (2017), 4066–4076.
23. Lehr, D. and Ohm, P. Playing with the data: What legal scholars should learn about machine learning. *UC Davis Law Review 51*, 2 (2017), 653–717.
24. Lewis, A. and Stoyanovich, J. Teaching responsible data science. *Intern. J. of Artificial Intelligence in Education* (2021).
25. Mitchell, M., et al. Model cards for model reporting. In *Proceedings of the Conf. on Fairness, Accountability, and Transparency 2019*, 220–229.
26. Olteanu, A., Castillo, C., Diaz, F., and Kiciman, E. Social data: Biases, methodological pitfalls, and ethical boundaries. *Frontiers Big Data 2*, 13 (2019).
27. Rabanser, S., Günnemann, S., and Lipton, Z. Failing loudly: An empirical study of methods for detecting dataset shift. H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors. In *Advances in Neural Information Processing Systems 32* (December 2019), 1394–1406.
28. Reeves, R. and Halikias, D. Race gaps in SAT scores highlight inequality and hinder upward mobility. Brookings (2017), https://www.brookings.edu/research/race-gaps-in-sat-scores-highlight-inequality-and-hinder-upward-mobility.
29. Salimi, B., Rodriguez, L., Howe, B., and Suciu, D. Interventional fairness: Causal database repair for algorithmic fairness. P.A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors. In *Proceedings of the 2019 Intern. Conf. on Management of Data*, 793–810.
30. Sarkar, S., Papon, T., Staratzis, D., and Athanassoulis, M. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 Intern. Conf. on Management of Data.*
31. Schelter, S. "Amnesia"–a selection of machine learning models that can forget user data very fast. *Conf. on Innovative Data Systems Research*, 2020.
32. Schelter, S., Grafberger, S., and Dunning, T. HedgeCut: Maintaining randomised trees for low-latency machine unlearning. In *Proceedings of the 2021 Intern. Conf. on Management of Data.*
33. Schelter, S. and Stoyanovich, J. Taming technical bias in machine learning pipelines. *IEEE Data Engineering Bulletin 43*, 4 (2020).
34. Selbst, A. Disparate impact in big data policing. *Georgia Law Review 52*, 109 (2017).
35. Shastri, S., Banakar, V., Wasserman, M., Kumar, A., and Chidambaram, V. Understanding and benchmarking the impact of GDPR on database systems. *PVLDB* (2020).
36. Stoyanovich, J. and Howe, B. Nutritional labels for data and models. *IEEE Data Engineering Bulletin 42*, 3 (2019), 13–23.
37. Stoyanovich, J., Howe, B., and Jagadish, H.V. Responsible data management. In *Proceedings of the VLDB Endowment 13*, 12 (2020), 3474–3488.
38. Yang, K., Loftus, J., and Stoyanovich, J. Causal intersectionality and fair ranking. K. Ligett and S. Gupta, editors. In *2nd Symposium on Foundations of Responsible Computing, Volume 192 of LIPICS, Schloss Dagstuhl–Leibniz Center for Informatics* (June 2021), 7:1-7:20.
39. Yang, K., Stoyanovich, J., Asudeh, A., Howe, B., Jagadish, H.V., and Miklau, G. A nutritional label for rankings. G. Das, C. Jermaine, and P. Bernstein, editors. In *Proceedings of the 2018 Intern. Conf. on Management of Data*, 1773–1776.
40. Zehlike, M., Yang, K., and Stoyanovich, J. Fairness in ranking: A survey. *CoRR* (2021), abs/2103.14000.

Julia Stoyanovich (stoyanovich@nyu.edu) is an associate professor at New York University, New York, NY, USA.

Serge Abiteboul is a researcher at Inria & École Normale Supérieure, Paris, France.

Bill Howe is an associate professor at the University of Washington, Seattle, WA, USA.

H.V. Jagadish is a professor at the University of Michigan, Ann Arbor, MI, USA.

Sebastian Schelter is an assistant professor at the University of Amsterdam, Amsterdam, The Netherlands.

To predict and serve?

Predictive policing systems are used increasingly by law enforcement to try to prevent crime before it occurs. But what happens when these systems are trained using biased data? **Kristian Lum** and **William Isaac** consider the evidence – and the social consequences

Main image: Maciej Bledowski/Bigstock.com

Kristian Lum, PhD is the lead statistician at the Human Rights Data Analysis Group

William Isaac, MPP is a doctoral candidate in the Department of Political Science at Michigan State University

In late 2013, Robert McDaniel – a 22-year-old black man who lives on the South Side of Chicago – received an unannounced visit by a Chicago Police Department commander to warn him not to commit any further crimes. The visit took McDaniel by surprise. He had not committed a crime, did not have a violent criminal record, and had had no recent contact with law enforcement. So why did the police come knocking?

It turns out that McDaniel was one of approximately 400 people to have been placed on Chicago Police Department's "heat list". These individuals had all been forecast to be potentially involved in violent crime, based on an analysis of geographic location and arrest data. The heat list is one of a growing suite of predictive "Big Data" systems used in police departments across the USA and in Europe to attempt what was previously thought impossible: to stop crime before it occurs.[1]

This seems like the sort of thing citizens would want their police to be doing. But predictive policing software – and the policing tactics based on it – has raised serious concerns among community activists, legal scholars, and sceptical police chiefs. These concerns include: the apparent conflict with protections against unlawful search and seizure and the concept of reasonable suspicion; the lack of transparency from both police departments and private firms regarding how predictive policing models are built; how departments utilise their data; and whether the programs unnecessarily target specific groups more than others.

But there is also the concern that police-recorded data sets are rife with systematic bias. Predictive policing software is designed to learn and reproduce patterns in data, but if biased data is used to train these predictive models, the models will reproduce and in some cases amplify those same biases. At best, this renders the predictive models ineffective. At worst, it results in discriminatory policing.

Bias in police-recorded data

Decades of criminological research, dating to at least the nineteenth century, have shown that police databases are not a complete census of all criminal offences, nor do they constitute a representative random sample.[2–5] Empirical evidence suggests that police officers – either implicitly or explicitly – consider race and ethnicity in their determination of which persons to detain and search and which neighbourhoods to patrol.[6,7]

If police focus attention on certain ethnic groups and certain neighbourhoods, it is likely that police records will systematically over-represent those groups and neighbourhoods. That is, crimes that occur in locations frequented by police are more likely to appear in the database simply because that is where the police are patrolling.

Bias in police records can also be attributed to levels of community trust in police, and the desired amount of local policing – both of which can be expected to vary according to geographic location and the demographic make-up of communities. These effects manifest as unequal crime reporting rates throughout a precinct. With many of the crimes in police databases being citizen-reported, a major source of

What is predictive policing?

According to the RAND Corporation, predictive policing is defined as "the application of analytical techniques – particularly quantitative techniques – to identify likely targets for police intervention and prevent crime or solve past crimes by making statistical predictions".[13] Much like how Amazon and Facebook use consumer data to serve up relevant ads or products to consumers, police departments across the United States and Europe increasingly utilise software from technology companies, such as PredPol, Palantir, HunchLabs, and IBM to identify future offenders, highlight trends in criminal activity, and even forecast the locations of future crimes.

What is a synthetic population?

A synthetic population is a demographically accurate individual-level representation of a real population – in this case, the residents of the city of Oakland. Here, individuals in the synthetic population are labelled with their sex, household income, age, race, and the geo-coordinates of their home. These characteristics are assigned so that the demographic characteristics in the synthetic population match data from the US Census at the highest geographic resolution possible.

How do we estimate the number of drug users?

In order to combine the NSDUH survey with our synthetic population, we first fit a model to the NSDUH data that predicts an individual's probability of drug use within the past month based on their demographic characteristics (i.e. sex, household income, age, and race). Then, we apply this model to each individual in the synthetic population to obtain an estimated probability of drug use for every synthetic person in Oakland. These estimates are based on the assumption that the relationship between drug use and demographic characteristics is the same at the national level as it is in Oakland. While this is probably not completely true, contextual knowledge about the local culture in Oakland leads us to believe that, if anything, drug use is even more widely and evenly spread than indicated by national-level data. While some highly localised "hotspots" of drug use may be missed by this approach, we have no reason to believe the location of those should correlate with the locations indicated by police data.

bias may actually be community-driven rather than police-driven. How these two factors balance each other is unknown and is likely to vary with the type of crime. Nevertheless, it is clear that police records do not measure crime. They measure some complex interaction between criminality, policing strategy, and community–police relations.

Machine learning algorithms of the kind predictive policing software relies upon are designed to learn and reproduce patterns in the data they are given, regardless of whether the data represents what the model's creators believe or intend. One recent example of intentional machine learning bias is Tay, Microsoft's automated chatbot launched earlier this year. A coordinated effort by the users of 4chan – an online message board with a reputation for crass digital pranks – flooded Tay with misogynistic and otherwise offensive tweets, which then became part of the data corpus used to train Tay's algorithms. Tay's training data quickly became unrepresentative of the type of speech its creators had intended. Within a day, Tay's Twitter account was put on hold because it was generating similarly unsavoury tweets.

A prominent case of unintentionally unrepresentative data can be seen in Google Flu Trends – a near real-time service that purported to infer the intensity and location of influenza outbreaks by applying machine learning models to search volume data. Despite some initial success, the models completely missed the 2009 influenza A–H1N1 pandemic and consistently over-predicted flu cases from 2011 to 2014. Many attribute the failure of Google Flu Trends to internal changes to Google's recommendation systems, which began suggesting flu-related queries to people who did not have flu.[8] In this case, the cause of the biased data was self-induced rather than internet hooliganism. Google's own system had seeded the data with excess flu-related queries, and as a result Google Flu Trends began inferring flu cases where there were none.

In both examples the problem resides with the data, not the algorithm. The algorithms were behaving exactly as expected – they reproduced the patterns in the data used to train them. Much in the same way, even the best machine learning algorithms trained on police data will reproduce the patterns and unknown biases in police data. Because this data is collected as a by-product of police activity, predictions made on the basis of patterns learned from this data do not pertain to future instances of crime on the whole. They pertain to future instances of *crime that becomes known to police*. In this sense, predictive policing (see "What is predictive policing?") is aptly named: it is predicting future policing, not future crime.

To make matters worse, the presence of bias in the initial training data can be further compounded as police departments use biased predictions to make tactical policing decisions. Because these predictions are likely to over-represent areas that were already known to police, officers become increasingly likely to patrol these same areas and observe new criminal acts that confirm their prior beliefs regarding the distributions of criminal activity. The newly observed criminal acts that police document as a result of these targeted patrols then feed into the predictive policing algorithm on subsequent days, generating increasingly biased predictions. This creates a feedback loop where the model becomes increasingly confident that the locations most likely to experience further criminal activity are exactly the locations they had previously believed to be high in crime: selection bias meets confirmation bias.

Predictive policing case study

How biased are police data sets? To answer this, we would need to compare the crimes recorded by police to a complete record of all crimes that occur, whether reported or not. Efforts such as the National Crime Victimization Survey provide national estimates of crimes of various sorts, including unreported crime. But while these surveys offer some insight into how much crime goes unrecorded nationally, it is still difficult to gauge any bias in police data at the local level because there is no "ground truth" data set containing a representative sample of local crimes to which we can compare the police databases.

We needed to overcome this particular hurdle to assess whether our claims about the effects of data bias and feedback in predictive policing were grounded in reality. Our solution was to combine a demographically representative *synthetic population* of Oakland, California (see "What is a synthetic population?") with survey data from the 2011 National Survey on Drug Use and Health (NSDUH). This approach allowed us to obtain high-resolution *estimates* of illicit drug use from a non-criminal justice, population-based data source (see "How do we estimate the number of drug users?") which we could then compare with police records. In doing so, we find that drug crimes known to police are not a representative sample of all drug crimes.

While it is likely that estimates derived from national-level data do not *perfectly* represent drug use at the local level, we still believe these estimates paint a more accurate picture of drug use in Oakland than the arrest data for several reasons. First, the US Bureau of Justice Statistics – the government body responsible for compiling and analysing criminal justice data – has used data from the NSDUH as a more representative measure of drug use than police reports.[2] Second, while arrest data is collected as a by-product of police activity, the NSDUH is a well-funded survey designed using best practices for obtaining a statistically representative sample. And finally, although there is evidence that some drug users do conceal illegal drug use from public health surveys, we believe that any incentives for such concealment apply much more strongly to police records of drug use than to public health surveys, as public health officials are not empowered (nor inclined) to arrest those who admit to illicit drug use. For these reasons, our analysis continues under the assumption that our public health-derived estimates of drug crimes represent a ground truth for the purpose of comparison.

Figure 1(a) shows the number of drug arrests in 2010 based on data obtained from the Oakland Police Department; Figure 1(b) shows the estimated number of drug users by grid square. From comparing these figures, it is clear that police databases and public health-derived estimates tell dramatically different stories about the pattern of drug use in Oakland. In Figure 1(a), we see that drug arrests in the police database appear concentrated in neighbourhoods around West Oakland (1) and International Boulevard (2), two areas with largely non-white and low-income populations. These neighbourhoods experience about 200 times more drug-related arrests than areas outside of these clusters. In contrast, our estimates (in Figure 1(b)) suggest that drug crimes are much more evenly distributed across the city. Variations in our estimated number of drug users are driven primarily by differences in population density, as the estimated rate of drug use is relatively uniform across the city. This suggests that while drug crimes exist everywhere, drug arrests tend to only occur in very specific locations – the police data appear to disproportionately represent crimes committed in areas with higher populations of non-white and low-income residents.

To investigate the effect of police-recorded data on predictive policing models, we apply a recently published predictive policing algorithm to the drug crime records in Oakland.[9] This algorithm was developed by PredPol, one of the largest vendors of predictive policing systems in the USA and one of the few companies to publicly release its algorithm in a peer-reviewed journal. It has been described by its founders

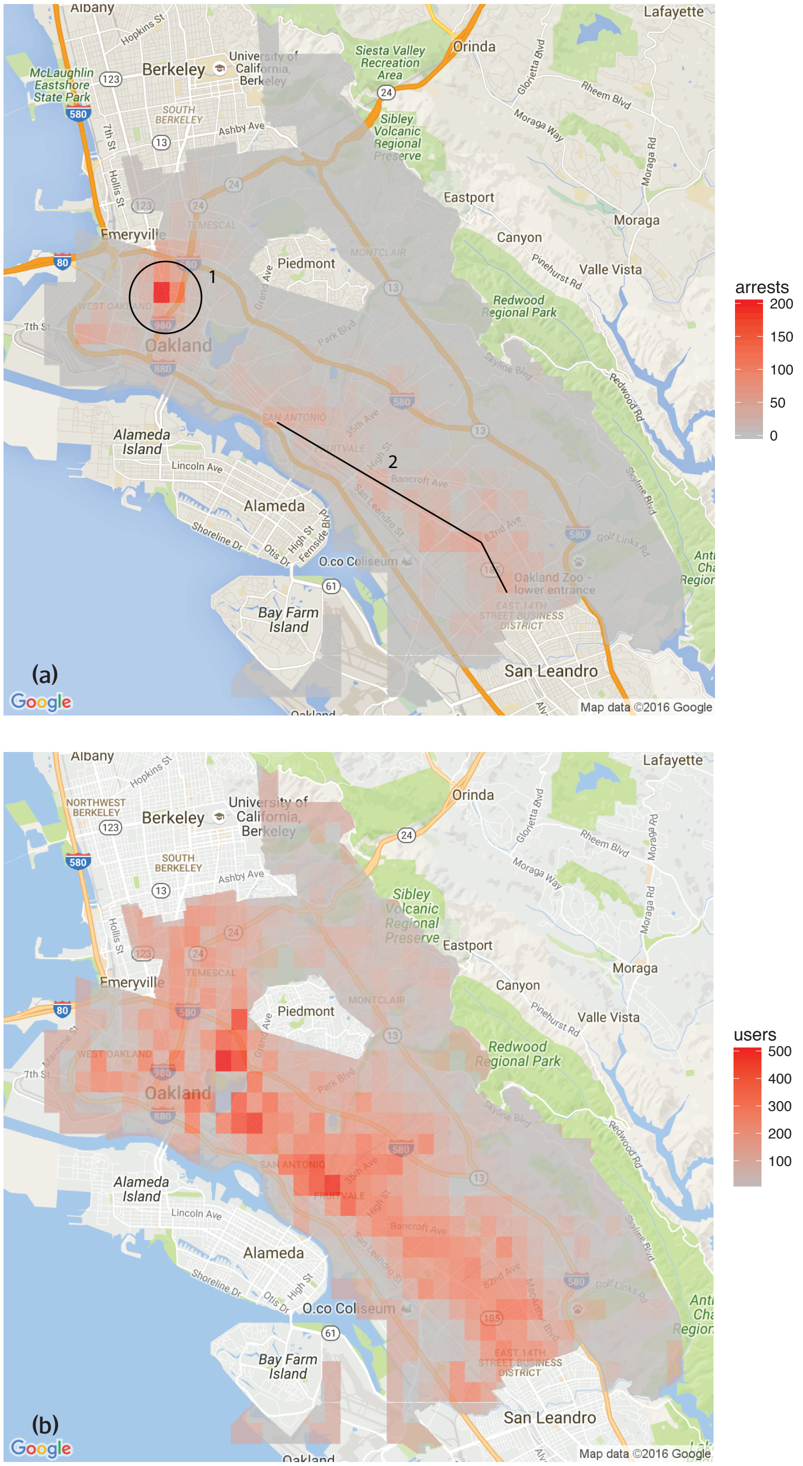


FIGURE 1 (a) Number of drug arrests made by Oakland police department, 2010. (1) West Oakland, (2) International Boulevard. (b) Estimated number of drug users, based on 2011 National Survey on Drug Use and Health

FIGURE 2 (a) Number of days with targeted policing for drug crimes in areas flagged by PredPol analysis of Oakland police data. (b) Targeted policing for drug crimes, by race. (c) Estimated drug use by race

as a parsimonious race-neutral system that uses "only three data points in making predictions: past type of crime, place of crime and time of crime. It uses no personal information about individuals or groups of individuals, eliminating any personal liberties and profiling concerns." While we use the PredPol algorithm in the following demonstration, the broad conclusions we draw are applicable to any predictive policing algorithm that uses unadjusted police records to predict future crime.

The PredPol algorithm, originally based on models of seismographic activity, uses a sliding window approach to produce a one-day-ahead prediction of the crime rate across locations in a city, using only the previously recorded crimes. The areas with the highest predicted crime rates are flagged as "hotspots" and receive additional police attention on the following day. We apply this algorithm to Oakland's police database to obtain a predicted rate of drug crime for every grid square in the city for every day in 2011. We record how many times each grid square would have been flagged by PredPol for targeted policing. This is shown in Figure 2(a).

We find that rather than correcting for the apparent biases in the police data, the model reinforces these biases. The locations that are flagged for targeted policing are those that were, by our estimates, already over-represented in the historical police data. Figure 2(b) shows the percentage of the population experiencing targeted policing for drug crimes broken down by race. Using PredPol in Oakland, black people would be targeted by predictive policing at roughly twice the rate of whites. Individuals classified as a race other than white or black would receive targeted policing at a rate 1.5 times that of whites. This is in contrast to the estimated pattern of drug use by race, shown in Figure 2(c), where drug use is roughly equivalent across racial classifications. We find similar results when analysing the rate of targeted policing by income group, with low-income households experiencing targeted policing at disproportionately high rates. Thus, allowing a predictive policing algorithm to allocate police resources would result in the disproportionate policing of low-income communities and communities of colour.

The results so far rely on one implicit assumption: that the presence of additional policing in a location does not change the number of crimes that are discovered in that location. But what if police officers have incentives to increase their productivity as a result of either internal or external demands? If true, they might seek additional opportunities to make arrests during patrols. It is then plausible that the more time police spend in a location, the more crime they will find in that location.

We can investigate the consequences of this scenario through simulation. For each day of 2011, we assign targeted policing according to the PredPol algorithm. In each location where targeted policing is sent, we increase the number of crimes observed by 20%. These additional simulated crimes then become part of the data set that is fed into PredPol on subsequent days and are factored into future forecasts. We study this phenomenon by considering the ratio of the predicted daily crime rate for targeted locations to that for non-targeted locations. This is shown in Figure 3, where large values indicate that many more crimes are predicted in the targeted locations relative to the non-targeted locations. This is shown separately for the original data (baseline) and the described simulation. If the additional crimes that were found as a result of targeted policing did not affect future predictions, the lines for both scenarios would follow the same trajectory. Instead, we find that this process causes the PredPol algorithm to become increasingly confident that most of the crime is contained in the targeted bins. This illustrates the feedback loop we described previously.

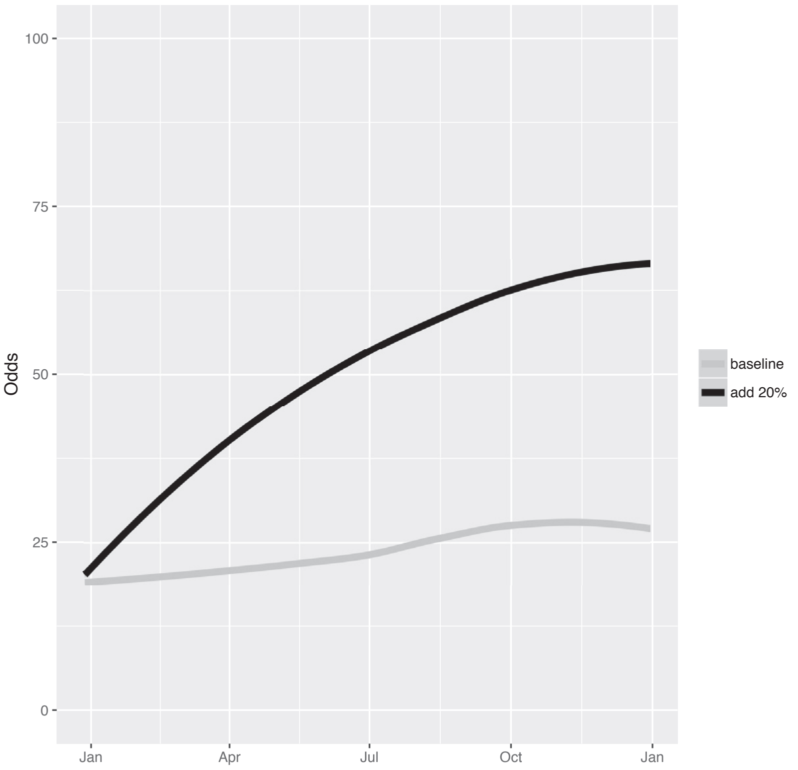


FIGURE 3 Predicted odds of crime in locations targeted by PredPol algorithm, relative to non-targeted locations. 'Baseline' is original Oakland police data. 'Add 20%' simulates the effect of additional crimes being observed in targeted locations

Discussion

We have demonstrated that predictive policing of drug crimes results in increasingly disproportionate policing of historically over-policed communities. Over-policing imposes real costs on these communities. Increased police scrutiny and surveillance have been linked to worsening mental and physical health;[10,11] and, in the extreme, additional police contact will create additional opportunities for police violence in over-policed areas.[12] When the costs of policing are disproportionate to the level of crime, this amounts to discriminatory policy.

In the past, police have relied on human analysts to allocate police resources, often using the same data that would be used to train predictive policing models. In many cases, this has also resulted in unequal or discriminatory policing. Whereas before, a police chief could reasonably be expected to justify policing decisions, using a computer to allocate police attention shifts accountability from departmental decision-makers to black-box machinery that purports to be scientific, evidence-based and race-neutral. Although predictive policing is simply reproducing and magnifying the same biases the police have historically held, filtering this decision-making process through sophisticated software that few people understand lends unwarranted legitimacy to biased policing strategies.

The impact of poor data on analysis and prediction is not a new concern. Every student who has taken a course on statistics or data analysis has heard the old adage "garbage in, garbage out". In an era when an ever-expanding array of statistical and machine learning algorithms are presented as panaceas to large and complex real-world problems, we must not forget this fundamental lesson, especially when doing so can result in significant negative consequences for society. ■

Note

The authors would like to thank Bobbi Isaac, Corwin Smidt, Eric Juenke, James Johndrow, Jim Hawdon, Matt Grossman, Michael Colaresi, Patrick Ball and the members of the HRDAG policing team for insightful conversations on this topic and comments on this article.

References

1. Gorner, J. (2013) Chicago police use "heat list" as strategy to prevent violence. *Chicago Tribune*, 21 August.

2. Langan, P. A. (1995) The racial disparity in U.S. drug arrests. bit.ly/29B2pQu

3. Levitt, S. D. (1998) The relationship between crime reporting and police: Implications for the use of Uniform Crime Reports. *Journal of Quantitative Criminology*, **14**(1), 61–81.

4. Morrison, W. D. (1897) The interpretation of criminal statistics. *Journal of the Royal Statistical Society*, **60**(1), 1–32.

5. Mosher, C. J., Miethe, T. D. and Hart, T. C. (2010) *The Mismeasure of Crime*. Thousand Oaks, CA: Sage Publications.

6. Gelman, A., Fagan, J., & Kiss, A. (2007) An analysis of the New York City Police Department's "stop-and-frisk" policy in the context of claims of racial bias. *Journal of the American Statistical Association*, **102**(479), 813–823.

7. Lange, J. E., Johnson, M. B. and Voas, R. B. (2005) Testing the racial profiling hypothesis for seemingly disparate traffic stops on the New Jersey Turnpike. *Justice Quarterly*, **22**(2), 193–223.

8. Lazer, D., Kennedy, R., King, G. and Vespignani, A. (2014) The parable of Google flu: traps in big data analysis. *Science*, **343**(6176), 1203–1205.

9. Mohler, G. O., Short, M. B., Malinowski, S., Johnson, M., Tita, G. E., Bertozzi, A. L. and Brantingham, P. J. (2015) Randomized controlled field trials of predictive policing. *Journal of the American Statistical Association*, **110**(512), 1399–1411.

10. Sewell, A. A. and Jefferson, K. A. (2016) Collateral Damage: The Health Effects of Invasive Police Encounters in New York City. *Journal of Urban Health*, **93**(1), 42–67.

11. Sewell, A. A., Jefferson, K. A. and Lee, H. (2016) Living under surveillance: gender, psychological distress, and stop-question-and-frisk policing in New York City. *Social Science & Medicine*, **159**, 1–13.

12. Lerman, A. E. and Weaver, V. (2014) Staying out of sight? Concentrated policing and local political action. *Annals of the American Academy of Political and Social Science*, **651**(1), 202–219.

13. Perry, W. L. (2013) *Predictive Policing: The Role of Crime Forecasting in Law Enforcement Operations*. Santa Monica, CA: Rand Corporation.

The VLDB Journal (2015) 24:557–581
DOI 10.1007/s00778-015-0389-y

REGULAR PAPER

Profiling relational data: a survey

Ziawasch Abedjan[1] · **Lukasz Golab**[2] · **Felix Naumann**[3]

Received: 1 August 2014 / Revised: 5 May 2015 / Accepted: 13 May 2015 / Published online: 2 June 2015

Abstract Profiling data to determine metadata about a given dataset is an important and frequent activity of any IT professional and researcher and is necessary for various use-cases. It encompasses a vast array of methods to examine datasets and produce metadata. Among the simpler results are statistics, such as the number of null values and distinct values in a column, its data type, or the most frequent patterns of its data values. Metadata that are more difficult to compute involve multiple columns, namely correlations, unique column combinations, functional dependencies, and inclusion dependencies. Further techniques detect conditional properties of the dataset at hand. This survey provides a classification of data profiling tasks and comprehensively reviews the state of the art for each class. In addition, we review data profiling tools and systems from research and industry. We conclude with an outlook on the future of data profiling beyond traditional profiling tasks and beyond relational databases.

1 Data profiling: finding metadata

Data profiling is the set of activities and processes to determine the metadata about a given dataset. Profiling data is an important and frequent activity of any IT professional and researcher. We can safely assume that any reader of this article has engaged in the activity of data profiling, at least by eye-balling spreadsheets, database tables, XML files, etc. Possibly, more advanced techniques were used, such as keyword searching in datasets, writing structured queries, or even using dedicated data profiling tools.

Johnson gives the following definition: “Data profiling refers to the activity of creating small but informative summaries of a database” [79]. Data profiling encompasses a vast array of methods to examine datasets and produce metadata. Among the simpler results are statistics, such as the number of null values and distinct values in a column, its data type, or the most frequent patterns of its data values. Metadata that are more difficult to compute involve multiple columns, such as inclusion dependencies or functional dependencies. Also of practical interest are approximate versions of these dependencies, in particular because they are typically more efficient to compute. In this survey we preclude these and concentrate on exact methods.

Like many data management tasks, data profiling faces three challenges: *(i)* managing the input, *(ii)* performing the computation, and *(iii)* managing the output. Apart from typical data formatting issues, the first challenge addresses the problem of specifying the expected outcome, i.e., determining which profiling tasks to execute on which parts of the data. In fact, many tools require a precise specification of what to inspect. Other approaches are more open and perform a wider range of tasks, discovering all metadata automatically.

The second challenge is the main focus of this survey and that of most research in the area of data profiling: The com-

---

✉ Felix Naumann
felix.naumann@hpi.de

Ziawasch Abedjan
abedjan@csail.mit.edu

Lukasz Golab
lgolab@uwaterloo.ca

[1] MIT CSAIL, Cambridge, MA, USA

[2] University of Waterloo, Waterloo, Canada

[3] Hasso Plattner Institute, Potsdam, Germany

putational complexity of data profiling algorithms depends on the number or rows, with a sort being a typical operation, but also on the number of columns. Many tasks need to inspect all column combinations, i.e., they are exponential in the number of columns. In addition, the scalability of data profiling methods is important, as the ever-growing data volumes demand disk-based and distributed processing.

The third challenge is arguably the most difficult, namely meaningfully interpreting the data profiling results. Obviously, any discovered metadata refer only to the given data instance and cannot be used to derive schematic/semantic properties with certainty, such as value domains, primary keys, or foreign key relationships. Thus, profiling results need interpretation, which is usually performed by database and domain experts.

Tools and algorithms have tackled these challenges in different ways. First, many rely on the capabilities of the underlying DBMS, as many profiling tasks can be expressed as SQL queries. Second, many have developed innovative ways to handle the individual challenges, for instance using indexing schemes, parallel processing, and reusing intermediate results. Third, several methods have been proposed that deliver only approximate results for various profiling tasks, for instance by profiling samples. Finally, users may be asked to narrow down the discovery process to certain columns or tables. For instance, there are tools that verify inclusion dependencies on user-suggested pairs of columns, but cannot automatically check inclusion between all pairs of columns or column sets.

Systematic data profiling, i.e., profiling beyond the occasional exploratory SQL query or spreadsheet browsing, is usually performed with dedicated tools or components, such as IBM's Information Analyzer, Microsoft's SQL Server Integration Services (SSIS), or Informatica's Data Explorer.[1] These approaches follow the same general procedure: A user specifies the data to be profiled and selects the types of metadata to be generated. Next, the tool computes the metadata in batch mode, using SQL queries and/or specialized algorithms. Depending on the volume of the data and the selected profiling results, this step can last minutes to hours. Results are usually displayed in a vast collection of tabs, tables, charts, and other visualizations to be explored by the user. Typically, discoveries can then be translated to constraints or rules that are then enforced in a subsequent cleansing/integration phase. For instance, after discovering that the most frequent pattern for phone numbers is `(ddd)ddd-dddd`, this pattern can be promoted to a *rule* stating that all phone numbers must be formatted accordingly. Most data cleansing tools can then either transform differently formatted numbers or mark them as improper.

We focus our discussion on relational data, the predominant format of traditional data profiling methods, but we do cover data profiling for other data models in Sect. 7.2.

1.1 Use-cases for data profiling

Data profiling has many traditional use-cases, including the data exploration, data cleansing, and data integration scenarios. Statistics about data are also useful in query optimization. Finally we describe several domain-specific use-cases, such as scientific data management and big data analytics.

Data exploration Database administrators, researchers, and developers are often confronted with new datasets, about which they know nothing. Examples include data files downloaded from the Web, old database dumps, or newly gained access to some DBMS. In many cases, such data have no known schema, no or old documentation, etc. Even if a formal schema is specified, it might be incomplete, for instance specifying only the primary keys but no foreign keys. A natural first step is to understand how the data are structured, what they are about, and how much of them there are.

Such manual data exploration, or data gazing[2], can and should be supported with data profiling techniques. Simple, ad hoc SQL queries can reveal some insight, such as the number of distinct values, but more sophisticated methods are needed to efficiently and systematically discover metadata. Furthermore, we cannot always expect an SQL expert as the explorer, but rather "data enthusiasts" without formal computer science training [68]. Thus, automated data profiling is needed to provide a basis for further analysis. Morton et al. [107] recognize that a key challenge is overcoming the current assumption of data exploration tools that data are "clean and in a well-structured relational format." Often data cannot be analyzed and visualized as is.

Database management A basic form of data profiling is the analysis of individual columns in a given table. Typically, the generated metadata include various counts, such as the number of values, the number of unique values, and the number of non-null values. These metadata are often part of the basic statistics gathered by a DBMS. An optimizer uses them to estimate the selectivity of operators and perform other optimization steps. Mannino et al. [99] give a survey of statistics collection and its relationship to database optimization. More advanced techniques use histograms of value distributions, functional dependencies, and unique column combinations to optimize range queries [118] or for dynamic reoptimization [80].

---

[1] See Sect. 6 for a more comprehensive list of tools.

[2] "Data gazing involves looking at the data and trying to reconstruct a story behind these data. […] Data gazing mostly uses deduction and common sense." [104]

Database reverse engineering Given a "bare" database instance, the task of schema and database reverse engineering is to identify its relations and attributes, as well as domain semantics, such as foreign keys and cardinalities [103, 116]. Hainaut et al. [66] call these metadata "implicit constructs," i.e., those that are not explicitly specified by DDL statements. However, possible sources for reverse engineering are DDL statements, data instances, data dictionaries, etc. The result of reverse engineering might be an entity-relationship model or a logical schema to assist experts in maintaining, integrating, and querying the database.

Data integration Often, the datasets to be integrated are unfamiliar and the integration expert wants to explore the datasets first: How large are they? What data types are needed? What are the semantics of columns and tables? Are there dependencies between tables and among databases?, etc. The vast abundance of (linked) *open data* and the desire and potential to integrate them with local data has amplified this need.

A concrete use-case for data profiling is that of *schema matching*, i.e., finding semantically correct correspondences between elements of two schemata [44]. Many schema matching systems perform data profiling to create attribute features, such as data type, average value length, and patterns, to compare feature vectors and align those attributes with the best matching ones [98, 109].

Scientific data management and integration have created additional motivation for efficient and effective data profiling: When importing raw data, e.g., from scientific experiments or extracted from the Web, into a DBMS, it is often necessary and useful to profile the data and then devise an adequate schema. In many cases, scientific data are produced by non-database experts and without the intention to enable integration. Thus, they often come with no adequate schematic information, such as data types, keys, or foreign keys.

Apart from exploring individual sources, data profiling can also reveal how and how well two datasets can be integrated. For instance, inclusion dependencies across tables from different sources suggest which tables might reasonably be combined with a join operation. Additionally, specialized data profiling techniques can reveal how much two relations overlap in their intent and extent. We discuss these challenges in Sect. 7.1.

Data quality / data cleansing The need to profile a new or unfamiliar set of data arises in many situations, in general to prepare for some subsequent task. A typical use-case is profiling data to prepare a *data cleansing* process. Commercial data profiling tools are usually bundled with corresponding data quality / data cleansing software.

Profiling as a data quality assessment tool reveals data errors, such as inconsistent formatting within a column, missing values, or outliers. Profiling results can also be used to measure and monitor the general quality of a dataset, for instance by determining the number of records that do not conform to previously established constraints [81, 117]. Generated constraints and dependencies also allow for rule-based data imputation.

Big data analytics "Big data," with its high volume, high velocity, and high variety [90], are data that cannot be managed with traditional techniques. Thus, data profiling gains a new importance. Fetching, storing, querying, and integrating big data are expensive, despite many modern technologies: Before exposing an infrastructure to Twitter's firehose, it might be worthwhile to know about properties of the data one is receiving; before downloading significant parts of the linked data cloud, some prior sense of the integration effort is needed; before augmenting a warehouse with text mining results an understanding of its data quality is required. In this context, leading researchers have noted "*If we just have a bunch of datasets in a repository, it is unlikely anyone will ever be able to find, let alone reuse, any of these data. With adequate metadata, there is some hope, but even so, challenges will remain*[…] [7]."

Many big data and related data science scenarios call for data mining and machine learning techniques to explore and mine data. Again, data profiling is an important preparatory task to determine which data to mine, how to import it into the various tools, and how to interpret the results [120].

Further use-cases Knowledge about data types, keys, foreign keys, and other constraints supports data modeling and helps keep data consistent, improves query optimization, and reaps all the other benefits of structured data management. Others have mentioned query formulation and indexing [126] and scientific discovery [75] as further motivation for data profiling. Also, compression techniques internally perform basic data profiling to optimize the compression ratio. Finally, the areas of data governance and data life-cycle management are becoming more and more relevant to businesses trying to adhere to regulations and code. Especially concerned are financial institutions and health care organizations. Again, data profiling can help ascertain which actions to take on which data.

1.2 Article overview and contributions

Data profiling is an important and practical topic that is closely connected to several other data management areas. It is also a timely topic and is becoming increasingly important given the recent trends in data science and big data analytics [108]. While it may not yet be a mainstream term in the database community, there already exists a large body of work that directly and indirectly addresses various aspects of data profiling. The goal of this survey is to classify and describe this body of work and illustrate its relevance to database research and practice. We also show that data profiling is far from a "done deal" and identify several promising directions for future work in this area.

The remainder of this paper is organized as follows. In Sect. 2, we outline and define data profiling based on a new taxonomy of profiling tasks. Sections 3, 4, and 5 survey the state of the art of the three main research areas in data profiling: analysis of individual columns, analysis of multiple columns, and detection of dependencies between columns, respectively. Section 6 surveys data profiling tools from research and industry. We provide an outlook of data profiling challenges in Sect. 7 and conclude this survey in Sect. 8.

2 Profiling tasks

This section presents a classification of data profiling tasks. Figure 1 shows our classification, which includes single-column tasks, multi-column tasks and dependency detection. While dependency detection falls under multi-column profiling, we chose to assign a separate profiling class to this large, complex, and important set of tasks. The classes are discussed in the following subsections. We also highlight additional dimensions of data profiling, such as the type of storage, the approximation of profiling results, as well as the relationship between data profiling and data mining.

Collectively, a set of results of these tasks is called the *data profile* or *database profile*. In general, we assume the dataset itself as our only input, i.e., we cannot rely on query logs, schema, documentation.

2.1 Single-column profiling

A basic form of data profiling is the analysis of individual columns in a given table. Typically, the generated metadata comprise various counts, such as the number of values, the number of unique values, and the number of non-null values. These metadata are often part of the basic statistics gathered by the DBMS. In addition, the maximum and minimum values are discovered and the data type is derived (usually restricted to string versus numeric versus date). More advanced techniques create histograms of value distributions and identify typical patterns in the data values in the form of regular expressions [122]. Data profiling tools display such results and can suggest actions, such as declaring a column with only unique values to be a key candidate or suggesting to enforce the most frequent patterns. As another exemplary use-case, query optimizers in database management systems also make heavy use of such statistics to estimate the cost of an execution plan.

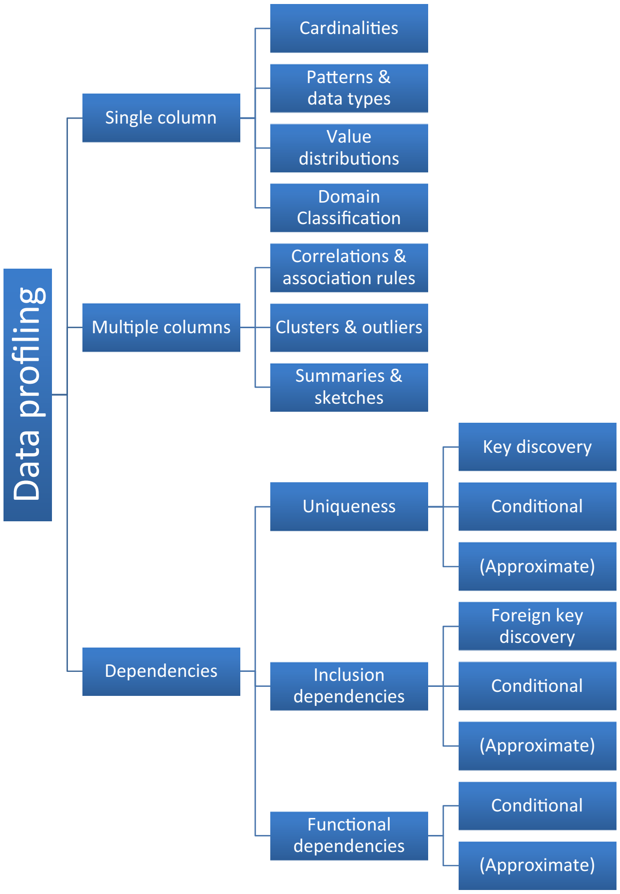


Fig. 1 A classification of traditional data profiling tasks

Table 1 lists the possible and typical metadata as a result of single-column data profiling. Some tasks are self-evident while others deserve more explanation. In Sect. 3, we elaborate on the more interesting tasks, their implementation, and their use.

2.2 Multi-column profiling

The second class of profiling tasks covers multiple columns simultaneously. Multi-column profiling generalizes profiling tasks on single columns to multiple columns and also identifies intervalue dependencies and column similarities. One task is to identify correlations between values through frequent patterns or association rules. Furthermore, clustering approaches that consume values of multiple columns as features allow for the discovery of coherent subsets of data records and outliers. Similarly, generating summaries and sketches of large datasets relates to profiling values across columns.

Table 1 Overview of selected single-column profiling tasks (see Sect. 3 for details)

Category	Task	Description
Cardinalities	num-rows	Number of rows
	value length	Measurements of value lengths (minimum, maximum, median, and average)
	null values	Number or percentage of null values
	distinct	Number of distinct values; sometimes called "cardinality"
	uniqueness	Number of distinct values divided by the number of rows
Value distributions	histogram	Frequency histograms (equi-width, equi-depth, etc.)
	constancy	Frequency of most frequent value divided by number of rows
	quartiles	Three points that divide the (numeric) values into four equal groups
	first digit	Distribution of first digit in numeric values; to check Benford's law
Patterns, data types, and domains	basic type	Generic data type, such as numeric, alphabetic, alphanumeric, date, time
	data type	Concrete DBMS-specific data type, such as varchar, timestamp.
	size	Maximum number of digits in numeric values
	decimals	Maximum number of decimals in numeric values
	patterns	Histogram of value patterns (Aa9…)
	data class	Semantic, generic data type, such as code, indicator, text, date/time, quantity, identifier
	domain	Classification of semantic domain, such as credit card, first name, city, phenotype

Such metadata are useful in many applications, such as data exploration and analytics. Outlier detection is used in data cleansing applications, where outliers may indicate incorrect data values.

Section 4 describes these tasks and techniques in more detail. It comprises multi-column profiling tasks that generate metadata on horizontal partitions of the data, such as values and records, instead vertical partitions, such as columns and column groups. Although the discovery of column dependencies, such as key or functional dependency discovery, also relates to multi-column profiling, we dedicate a separate section to dependency discovery as described next.

2.3 Dependencies

Dependencies are metadata that describe relationships among columns. The difficulties of automatically detecting such dependencies in a given dataset are twofold: First, pairs of columns or larger column sets must be examined, and second, the chance existence of a dependency in the data at hand does not imply that this dependency is meaningful. While much research has been invested in addressing the first challenge and is the focus of this survey, there is less work on semantically interpreting the profiling results.

A common goal of data profiling is to identify suitable keys for a given table. Thus, the discovery of *unique column combinations*, i.e., sets of columns whose values uniquely identify rows, is an important data profiling task [70]. Once unique column combinations have been discovered, a second step is to identify among them the intended primary key of a relation.

A frequent real-world use-case of multi-column profiling is the discovery of foreign keys [96,123] with the help of inclusion dependencies [14,100]. An inclusion dependency states that all values or value combinations from one set of columns also appear in the other set of columns—a prerequisite for a foreign key.

Another form of dependency that is also relevant for data quality is the functional dependency (FD). A functional dependency states that values in one set of columns functionally determine the value of another column. Again, much research has been performed to automatically detect FDs [75,139]. Section 5 surveys dependency discovery algorithms in detail.

Dependencies have many applications: An obvious use-case for functional dependencies is schema normalization. Inclusion dependencies can suggest how to join two relations, possibly across data sources. Their conditional counterparts help explore the data by focusing on certain parts of the dataset.

2.4 Conditional, partial, and approximate solutions

Real datasets usually contain exceptions to rules. To account for this, dependencies and other constraints detected by data profiling can be relaxed. We describe two relaxations below: partial and approximate.

Partial dependencies hold for only a subset of the records, for instance, for 95 % of the records or for all but 10 records. Such dependencies are especially valuable in data cleansing scenarios: They are patterns that hold for almost all records and thus should probably hold for *all* records if the data were clean. Violating records can be extracted and cleansed [129].

Once a partial dependency has been detected, it is interesting to characterize for which records it holds, i.e., if we can find a condition that selects precisely those records. *Conditional dependencies* can specify such conditions. For instance, a conditional unique column combination might state that the column street is unique for all records with city = 'NY.' Conditional inclusion dependencies (CINDs) were proposed by Bravo et al. for data cleaning and contextual schema matching [19]. Conditional functional dependencies (CFDs) were introduced in [46], also for data cleaning.

Approximate dependencies and other constraints are unconditional statements, but are not guaranteed to hold for the entire relation. Such dependencies are often discovered using sampling [76] or other summarization techniques [31]. Their approximate nature is often sufficient for certain tasks, and approximate dependencies can be used as input to the more rigorous task of detecting true dependencies. This survey does not discuss such approximation techniques.

2.5 Types of storage

Data profiling tasks are applicable to a wide range of situations in which data are provided in various forms. For instance, most commercial profiling tools assume that data reside in a relational database, make use of SQL queries and indexes. In other situations, for instance, a csv file is provided and a data profiling method needs to create its own data structures in memory or on disk. And finally, there are situations in which a mixed approach is useful: Data that were originally in the database are read once and processed further outside the database.

The discussion and distinction of such different situations is relevant when evaluating the performance of data profiling algorithms and tools. Can we assume that data are already loaded into main memory? Can we assume the presence of indices? Are profiling results, which can be quite voluminous, written to disk? Fair comparisons need to establish a level playing field with same assumptions about data storage.

2.6 Data profiling versus data mining

A clear, well-defined, and accepted distinction between data profiling and data mining does not exist. Two criteria are conceivable:

1. Distinction by the object of analysis: instance versus schema or columns versus rows
2. Distinction by the goal of the task: description of existing data versus new insights beyond existing data.

Following the first criterion, Rahm and Do distinguish data profiling from data mining by the number of columns that are examined: "Data profiling focuses on the instance analysis of individual attributes. [...] Data mining helps discover specific data patterns in large datasets, e.g., relationships holding between several attributes" [121]. While this distinction is well defined, we believe several tasks, such as IND or FD detection, belong to data profiling, even if they discover relationships between multiple columns.

We believe a different distinction along both criteria is more useful: Data profiling gathers technical metadata to support data management; data mining and data analytics discovers non-obvious results to support business management with new insights. While data profiling focuses mainly on columns, some data mining tasks, such as rule discovery or clustering, may also be used for identifying interesting characteristics of a dataset. Others, such as recommendation or classification, are not related to data profiling.

With this distinction, we concentrate on data profiling and put aside the broad area of data mining, which has already received unifying treatment in numerous textbooks and surveys. However, in Sect. 4, we address the subset of unsupervised mining approaches that can be applied on unknown data to generate metadata and hence serves the purpose of data profiling.

Classifications of data mining tasks include an overview by Chen et al., who distinguish the kinds of databases (relational, OO, temporal, etc.), the kinds of knowledge to be mined (association rules, clustering, deviation analysis, etc.), and the kinds of techniques to be used [130]. We make a similar distinction in this survey. In particular, we distinguish the different classes of data profiling tasks and then examine various techniques to perform them. We discuss profiling non-relational data in Sect. 7.

2.7 Summary

We summarize this section by connecting the various data profiling tasks with the use-cases mentioned in the introduction. Conceivably, any task can be useful for any use-case, depending on the context, the properties of the data at hand,

Table 2 Data profiling tasks and their primary use-cases

	Database management	Data integration	Data cleansing	Database reverse engineering	Data exploration	Data analytics	Scientific data management
Single-column							
Cardinalities	✓				✓	✓	
Patterns and data types		✓	✓	✓			
Value distributions	✓		✓		✓	✓	
Domain classification		✓	✓	✓			✓
Multi-column							
Correlations					✓	✓	✓
Association rules						✓	✓
Clustering		✓			✓	✓	✓
Outliers			✓				✓
Summaries and sketches			✓		✓	✓	
Dependencies							
Unique column combinations	✓		✓	✓			
Inclusion dependencies	✓	✓		✓	✓		✓
Conditional inclusion dependencies		✓			✓	✓	✓
Functional dependencies	✓		✓	✓	✓		
Conditional functional dependencies			✓		✓	✓	

etc. Table 2 lists the profiling tasks and their primary use-cases.

3 Column analysis

The analysis of the values of individual columns is usually a straightforward task. Table 1 lists the typical metadata that can determined for a given column. The following sections describe each category of tasks in more detail, mentioning possible uses of the respective results. In [104], a book addressing practitioners, several of these tasks are discussed in more detail.

3.1 Cardinalities

Cardinalities or counts of values in a column are the most basic form of metadata. The number of rows in a table (num-rows) reflects how many entities (e.g., customers, orders, items) are represented in the data, and it is relevant to data management systems, for instance to estimate query costs or to assign storage space.

Information about the length of values in terms of characters (value length), including the length of the longest and shortest value and the average length, is useful for schema reverse engineering (e.g., to determine tight data type bounds), outlier detection (e.g., single-character first names), and formatting (dates have the same min-, max- and average length).

The number of empty cells, i.e., cells with null values or empty strings (null values), indicates the (in-)completeness of a column. The number of distinct values (distinct) allows query optimizers to estimate selectivity of selection or join operations: The more distinct values there are, the more selective such operations are. To users, this number can indicate a candidate key by comparing it with the number of rows. Alternatively, this number simply illustrates how many different values are present (e.g., how many customers have ordered something or how many cities appear in an address table).

Determining the number of rows, metadata about value lengths, and the number of null values is straightforward and can be performed in a single pass over the data. Determining the number of *distinct values* is more involved: Either hashing or sorting all values is necessary. When hashing, the number of non-empty buckets must be counted, taking into account hash collisions, which further add to the count. When sorting, a pass through the sorted data counts the number of values, where groups of same values are counted only once.

From the number of distinct values the *uniqueness* can be calculated, which is typically defined as the number of unique values divided by the number of rows. Note that the number of distinct values can also be estimated using the minHash technique discussed in Sect. 4.3.

Apart from determining the exact number of distinct values, query optimization is a strong incentive to *estimate* those counts in order to predict query execution plan costs without actually reading the entire data. Because approximate profiling is not the focus of this survey, we give only two exemplary pointers. Haas et al. [65] base their estimation on data samples and describe and empirically compare various estimators from the literature. Other works do scan the entire data but use only a small amount of memory to hash the values and estimate the number of distinct values, an early example being [11].

3.2 Value distribution

Value distributions are more fine-grained cardinalities, namely the cardinalities of groups of values. *Histograms* are among the most common profiling results. A histogram stores frequencies of values within well-defined groups, usually by dividing the ordered set of values into a fixed set of buckets. The buckets of equi-width histograms span value ranges of same length, while the buckets of equi-depth (or equi-height) histograms each represent the same number of value occurrences. A common special case of an equi-depth histogram is dividing the data into four *quartiles*. A more general concept is *biased histograms*, which can adapt their accuracy for different regions[33]. Histograms are used for database optimization as a rough probability distribution to avoid a uniform distribution assumption and thus provide better cardinality estimations [77]. In addition, histograms are interpretable by humans, as their visual representation is easy to comprehend.

The *constancy* of a column is defined as the ratio of the frequency of the most frequent value (possibly a pre-defined default value) and the overall number of values. It thus represents the proportion of some constant value compared with the entire column.

A particularly interesting distribution is the first digit distribution for numeric values. Benford's law [15] states that in naturally occurring numbers the distribution of the first digit d of a number approximately follows $P(d) = \log_{10}(1 + \frac{1}{d})$. Thus, the 1 is expected to be the most frequent leading digit, followed by 2, etc. Benford and others have observed this behavior in many sets of numbers, such as molecular weights, building sizes, and electricity bills. In fact, the law has been used to uncover accounting fraud and other fraudulently created numbers.

Determining the above distributions usually involves a single pass over the column, except for equi-depth histograms (i.e., with fixed bucket sizes) and quartiles, which determine bucket boundaries through sorting. In the same manner or through hashing the most frequent value can be discovered to determine constancy.

Finally, many more things can be counted and aggregated in a column. For instance, some profiling tools and methods determine among others the frequency distribution of soundex code, n-grams, and others, the inverse frequency distribution, i.e., the distribution of the frequency distribution, or the entropy of the frequency distribution of the values in a column [82].

3.3 Types and patterns

The profiling tasks of this section are ordered by increasing semantic richness (see also Table 1). We start with the most simple observable properties, move on to specific patterns of the values of a column, and end with the semantic domain of a column.

Discovering the *basic type* of a column, i.e., classifying it as numeric, alphabetic, alphanumeric, date, or time, is fairly simple: The presence or absence of numeric and non-numeric characters already distinguishes the first three. The latter two can usually be recognized by the presence of numbers only within certain ranges, and by numbers separated in regular patterns by special symbols. Recognizing the actual data type, for instance among the SQL types, is similarly easy. In fact, data of many data types, such as timestamp, boolean, or int, must follow a fixed, sometimes DBMS-specific pattern. When classifying columns into data types, one should choose the most specific data type—in particular avoiding the catchalls char or varchar if possible. For the data types decimal, float, and double, one can additionally extract the maximum number of digits and decimals to determine the metadata size and decimals.

A common and useful data profiling result is the extraction of frequent *patterns* observed in the data of a column. Then, data that do not conform to such a pattern are likely erroneous or ill-formed. For instance, a pattern for phone numbers might be informally encoded as `+dd (ddd) ddd dddd` or as a simple regular expression `\(\d3\)\ − \d3\ − \d4)`.[3] A challenge when determining frequent patterns is to find a good balance between generality and specificity. The regular expression `.*` is the most general and matches any string. On the other hand, the expression `data` allows only that one single string. For the Potter's Wheel tool, Raman and Hellerstein [122] suggest finding the data pattern with the minimal description length (MDL). They model description length as a combination of precision, recall, and conciseness and provide an algorithm to enumerate all possible patterns. The RelIE system was designed

---

[3] A more detailed regular expression, taking into account different formatting options and different restrictions (e.g., phone numbers cannot begin with a 1), can easily reach 200 characters in length.

for information extraction from textual data [92]. It creates regular expressions based on training data with positive and negative examples by systematically, greedily transforming regular expressions. Finally, Fernau [51] provides a good characterization of the problem of learning regular expressions from data and presents a learning algorithm for the task. This work is also a good starting point for further reading

The semantic *domain* of a column describes not the syntax of its values but their meaning. While a regular expression might characterize a column, labeling it as "phone number" provides a concrete domain. Clearly, this task cannot be fully automated, but some work has been done for common-place domains about persons, places, etc. Zhang et al. take a first step by clustering columns that have the same meaning across the tables of a database [144], which they extend to the particularly difficult area of numeric values in [142]. In [133] the authors take the additional step of matching columns to pre-defined semantics from the person domain. Knowledge of the domain is not only of general data profiling interest, but also of particular interest to schema matching, i.e., the task of finding semantic correspondences between elements of different database schemata.

3.4 Data completeness

Explicit missing data are simple to characterize: For each column, we report the number of tuples with a null or a default value. However, datasets may contain *disguised* missing values. For example, Web forms often include fields whose values must be chosen from pull-down lists. The first value from the pull-down list may be pre-populated on the form, and some users may not replace it with a proper or correct value due to lack of time or privacy concerns. Specific examples include entering 99999 as the zip code of an address or leaving "Alabama" as the pre-populated state (in the US, Alabama is alphabetically the first state). Of course, for some records, Alabama may be the true state.

Detecting disguised default values is difficult. One heuristic solution is to examine each column at a time, and, for each possible value, compute the distribution of the other attribute values [74]. For example, if Alabama is indeed a disguised default value, we expect a large subset of tuples with state = `Alabama` (i.e., those whose true state is different) to form an unbiased sample of the whole relation.

Another instance in which profiling missing data is not trivial involves timestamped data, such as measurement or transaction data feeds. In some cases, tuples are expected to arrive regularly, e.g., in datacenter monitoring, every machine may be configured to report its CPU utilization every minute. However, measurements may be lost en route to the database, and overloaded or malfunctioning machines may not report any measurements at all. [60]. In contrast to detecting missing attribute values, here we are interested in estimating the number of missing tuples. Thus, the profiling task may be to single out the columns identified as being of type timestamp, and, for those that appear to be distributed uniformly across a range, infer the expected frequency of the underlying data source and estimate the number of missing tuples. Of course, some timestamp columns correspond to application timestamps with no expectation of regularity, rather than data arrival timestamps. For instance, in an online retailer database, order dates and delivery dates are generally not expected to be scattered uniformly over time.

4 Multi-column analysis

Profiling tasks over a single column can be generalized to projections of multiple columns. For example, there has been work on computing multi-dimensional histograms for query optimization [41,119]. Multi-column profiling also plays an important role in data cleansing, e.g., in assessing and explaining data glitches, which often occur in column combinations [40].

In the remainder of this section, we discuss statistical methods and data mining approaches for generating metadata based on co-occurrences and dependencies of values across attributes. We focus on correlation and rule mining approaches as well as unsupervised clustering and learning approaches; machine learning techniques that require training data or detailed knowledge of the data are beyond the scope of data profiling.

4.1 Correlations and association rules

Correlation analysis reveals related numeric columns, e.g., in an Employees table, age and salary may be correlated. A straightforward way to do this is to compute pairwise correlations among all pairs of columns. In addition to column-level correlations, value-level *associations* may provide useful data profiling information.

Traditionally, a common application of association rules has been to find items that tend to be purchased together based on point-of-sale transaction data. In these datasets, each row is a list of items purchased in a given transaction. An association rule $\{\texttt{bread}\} \rightarrow \{\texttt{butter}\}$, for example, states that if a transaction includes bread, it is also likely to include butter, i.e., customers who buy bread also buy butter. A set of items is referred to as an *itemset*, and an association rule specifies an itemset on the left-hand side and another itemset on the right-hand side.

Algorithms for generating association rules from data decompose the problem into two steps [8]:

1. Discover all frequent itemsets, i.e., those whose frequencies in the dataset (i.e., their *support*) exceed some

threshold. For instance, the itemset {`bread`, `butter`} may appear in 800 out of a total of 50,000 transactions for a support of 1.6 %.

2. For each frequent itemset a, generate association rules of the form $l \rightarrow a - l$ with $l \subset a$, whose *confidence* exceeds some threshold. Confidence is defined as the frequency of a divided by the frequency of l, i.e., the conditional probability of l given $a - l$. For example, if the frequency of {`bread`, `butter`} is 800 and the frequency of {`bread`} alone is 1000, then the confidence of the association rule {`bread`} → {`butter`} is 0.8. That is, if bread is purchased, there is an 80 % chance that butter is also purchased in the same transaction.

In the context of relational data profiling, association rules denote relationships or patterns between attribute values among columns. Consider an Employees table with fields name, employee number, department, position, and allowance. We may find a frequent itemset of the form {department = `finance`, position = `assistant manager`, allowance = `$1000`} and a corresponding association rule of the form {department = `finance`, position = `assistant manager`} → {allowance = `$1000`}. This would be the case if most or all assistant managers in the finance department were assigned an allowance budget of $1000.

While the second step mentioned above is straightforward (generating association rules from frequent itemsets), the first step is computationally expensive due to the large number of possible frequent itemsets (or patterns of values) [72]. Popular algorithms for efficiently discovering frequent patterns include Apriori [8], Eclat [141], and FP-Growth [67].

The Apriori algorithm exploits the observation that all subsets of a frequent itemset must also be frequent. In the first iteration, Apriori finds all frequent itemsets of size one, i.e., those containing one item or one attribute value. In the next iteration, only the frequent itemsets of size one are expanded to find frequent itemsets of size two, and so on.

There are also several optimized versions of Apriori, such as DHP [115] and RARM [35]. FP-Growth discovers frequent itemsets without a candidate generation step. It transforms the database into an extended prefix tree of frequent patterns (FP-tree). The FP-Growth algorithm traverses the tree and generates frequent itemsets by pattern growth in a depth-first manner. Finally, Eclat is based on intersecting transaction-id (TID) sets of associated itemsets and is best suited for dealing with large frequent itemsets. Eclat's strategy for identifying frequent itemsets is similar to Apriori.

Negative correlation rules, i.e., those that identify attribute values that *do not* co-occur with other attribute values, may also be useful in data profiling to find anomalies and outliers [21]. However, discovering negative association rules is more difficult, because *infrequent* itemsets cannot be pruned in the same way as frequent itemsets, and therefore, novel pruning rules are required [135].

Finally, we note that in addition to using existing techniques, such as correlations and association rules for profiling, extensions have been proposed, such as discovering linear dependencies between columns [25].

However, in this approach, the user has to choose the subset of attributes to be analyzed. We discuss dependency discovery in more detail in Sect. 5.

4.2 Clustering and outlier detection

Another useful profiling task is to segment the records into homogeneous groups using a clustering algorithm; furthermore, records that do not fit into any cluster may be flagged as outliers. Cluster analysis can identify groups of similar records in a table, while outliers may indicate data quality problems. For example, Dasu and Johnson [36] cluster numeric columns and identify outliers in the data. Furthermore, based on the assumption that data glitches occur across attributes and not in isolation [16], statistical inference has been applied to measure glitch recovery in [39].

Another example of clustering in the context of data profiling is ProLOD++, which applies k-means clustering to RDF relations [1]. We refer the reader to surveys by Jain et al. [78] and Xu and Wunsch II [137] for more details on clustering algorithms for relational data.

4.3 Summaries and sketches

Besides clustering, another way to describe data is to create summaries or sketches [23]. This can be done by sampling or hashing data values to a smaller domain. Sketches have been widely applied to answering approximate queries, data stream processing and estimating join sizes [37,54,111]. Cormode et al. [31] give an overview of sketching and sampling for approximate query processing.

Another interesting task is to assess the similarity of two columns, which can be done using multi-column hashing techniques. The *Jaccard similarity* of two columns A and B is $|A \cap B|/|A \cup B|$, i.e., the number of distinct values they have in common divided by the total number of distinct values appearing in them. This gives the relative number of values that appear in both A and B. Since semantically similar values may have different formats, we can also compute the Jaccard similarity of the n-gram distributions in A and B. If the distinct value sets of columns A and B are not available, we can estimate the Jaccard similarity using their *MinHash signatures* [38].

Table 3 Dependency discovery algorithms

Dependency	Algorithms
Uniques	HCA [3], GORDIAN [126], DUCC [70], SWAN [5]
Functional dependencies	TANE [75], FUN [110], FD_Mine [139], Dep-Miner [95], FastFDs [136], FDEP [52], DFD[6]
Conditional functional dependencies	[24], [59], CTANE [47], CFUN [42], FACD [91], FastCFD [47]
Inclusion dependencies	[101], [87], SPIDER [14], ZigZag [102]
Conditional inclusion dependencies	[61], CINDERELLA [13], PLI [13]
Foreign keys	[123], [143]
Denial constraints	FastDC [29]
Differential dependencies	[128]
Sequential dependencies	[57]

5 Dependency detection

We now survey various formalisms for detecting dependencies among columns and algorithms for mining them from data, including keys and unique column combinations (Sect. 5.1), functional dependencies (Sect. 5.2), inclusion dependencies (Sect. 5.3), and other types of dependencies that are relevant to data profiling (Sect. 5.4). Table 3 lists the algorithms that are discussed.

We use the following symbols: R and S denote relational schemata, with r and s denoting instances of R and S, respectively. The number of columns in R is $|R|$ and the number of tuples in r is $|r|$. We refer to tuples of r and s as r_i and s_j, respectively. Subsets of columns are denoted by uppercase X, Y, Z (with $|X|$ denoting the number of columns in X) and individual columns by uppercase A, B, C. Furthermore, we define $\pi_X(r)$ and $\pi_A(r)$ as the projection of r on the attribute set X or attribute A, respectively; thus, $|\pi_X(r)|$ denotes the count of district combinations of the values of X appearing in r. Accordingly, $r_i[A]$ indicates the value of the attribute A of tuple r_i and $r_i[X] = \pi_X(r_i)$. We refer to an attribute value of a tuple as a *cell*.

The number of potential dependencies in r can be exponential in the number of attributes $|R|$; see Fig. 2 for an illustration of all possible subsets of the attributes in Table 4. This means that any dependency discovery algorithm has a worst-case exponential time complexity. There are two classes of heuristics that have appeared in the literature. Column-based or top-down approaches start with "small" dependencies (in terms of the number of attributes they reference) and work their way to larger dependencies, pruning candidates along the way whenever possible. Row-based or bottom-up approaches attempt to avoid repeated scanning of the entire relation during candidate generation. While these approaches cannot reduce the worst-case exponential complexity of dependency discovery, experimental studies have shown that column-based approaches work well on tables containing a very large number of rows and row-based approaches work well for wide tables [6,113]. For more details on the computational complexity of various Fd and Ind discovery algorithms, we refer the interested reader to [94].

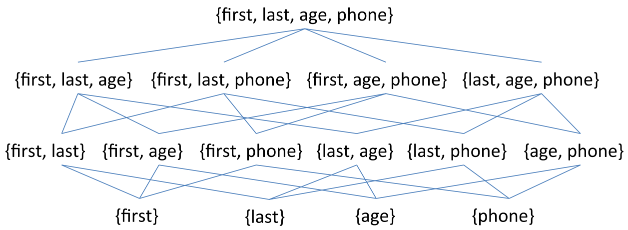


Fig. 2 Powerset lattice for the example Table 4

Table 4 Example dataset

Tuple id	First	Last	Age	Phone
1	Max	Payne	32	1234
2	Eve	Smith	24	5432
3	Eve	Payne	24	3333
4	Max	Payne	24	3333

5.1 Unique column combinations and keys

Given a relation R with instance r, a *unique column combination* (a "unique") is a set of columns $X \subseteq R$ whose projection on r contains only unique value combinations.

Definition 1 (*Unique*) A column combination $X \subseteq R$ is a *unique*, iff $\forall r_i, r_j \in r, i \neq j : r_i[X] \neq r_j[X]$.

Analogously, a set of columns $X \subseteq R$ is a *non-unique column combination* (a "non-unique"), iff its projection on r contains at least one duplicate value combination.

Definition 2 (*Non-unique*) A column combination $X \subseteq R$ is a *non-unique*, iff $\exists r_i, r_j \in r, i \neq j : r_i[X] = r_j[X]$.

Each superset of a unique is also unique while each subset of a non-unique is also a non-unique. Therefore, discovering all uniques and non-uniques can be reduced to the discovery of minimal uniques and maximal non-uniques:

Definition 3 (*Minimal Unique*) A column combination $X \subseteq R$ is a *minimal unique*, iff $\forall X' \subset X : X'$ is a non-unique.

Definition 4 (*Maximal Non-Unique*) A column combination $X \subseteq R$ is a *maximal non-unique*, iff $\forall X' \supset X : X'$ is a unique.

A *primary key* is a unique that was explicitly chosen to be the unique record identifier while designing the table schema. Since the discovered uniqueness constraints are only valid for a relational instance at a specific point of time, we refer to uniques and non-uniques instead of keys and non-keys. A further distinction can be made in terms of possible keys and certain keys when dealing with uncertain data and NULL values [86].

The problem of discovering a minimal unique of size $k \leq n$ is NP-complete [97]. To discover all minimal uniques and maximal non-uniques of a relational instance, in the worst case, one has to visit all subsets of the given relation, no matter the strategy (breadth-first or depth-first) or direction (bottom-up or top-down). Thus, the discovery of all minimal uniques and maximal non-uniques of a relational instance is an NP-hard problem and even the solution set can be exponential [64].

Given $|R|$, there can be $\binom{|R|}{\frac{|R|}{2}} \geq 2^{\frac{|R|}{2}}$ minimal uniques in the worst case, as all combinations of size $\frac{|R|}{2}$ can simultaneously be minimal uniques.

5.1.1 GORDIAN*: row-based discovery*

Row-based algorithms require multiple runs over all column combinations as more and more rows are considered. They benefit from the intuition that non-uniques can be detected without considering every row. A recursive unique discovery algorithm that works this way is GORDIAN [126]. The algorithm consists of three parts: *(i)* Pre-organize the data in form of a prefix tree, *(ii)* find maximal non-uniques by traversing the prefix tree, *(iii)* compute minimal uniques from maximal non-uniques.

The prefix tree is stored in main memory. Each level of the tree represents one column of the table, whereas each branch stands for one distinct tuple. Tuples that have the same values in their prefix share the corresponding branches. For example, all tuples that have the same value in the first column share the same node cells. The time to create the prefix tree depends on the number of rows; therefore, this can be a bottleneck for very large datasets.

The traversal of the tree is based on the cube operator [63], which computes aggregate functions on projected columns. Non-unique discovery is performed by a depth-first traversal of the tree for discovering maximum repeated branches, which constitute maximal non-uniques.

After discovering all maximal non-uniques, GORDIAN computes all minimal uniques by generating minimal combinations that are not covered by any of the maximal non-uniques. In [126] it is stated that this complementation step needs only quadratic time in the number of minimal uniques, but the presented algorithm implies cubic runtime: For each non-unique, the updated set of minimal uniques must be *simplified* by removing redundant uniques. This simplification requires quadratic runtime in the number of uniques. As the number of minimal uniques is bound linearly by the number s of maximal non-uniques, the runtime of the unique generation step is $O(s^3)$.

GORDIAN exploits the intuition that non-uniques can be discovered faster than uniques. Non-unique discovery can be aborted as soon as one repeated value is discovered among the projections. The prefix structure of the data facilitates this analysis. It is stated that the algorithm is polynomial in the number of tuples for data with a Zipfian distribution of values. Nevertheless, in the worst case, GORDIAN has exponential runtime.

The generation of minimal uniques from maximal non-uniques can be a bottleneck if there are many maximal non-uniques. Experiments showed that in most cases the unique generation dominates the runtime [3]. Furthermore, the approach is limited by the available main memory. Although data may be compressed because of the prefix structure of the tree, the amount of processed data may still be too large to fit in main memory.

5.1.2 Column-based traversal of the column lattice

The problem of finding minimal uniques is comparable to the problem of finding frequent itemsets [8]. The well-known Apriori approach is applicable to minimal unique discovery, working bottom-up as well as top-down. With regard to the powerset lattice of a relational schema, the Apriori algorithms generate all relevant column combinations of a certain size and verify those at once. Figure 2 illustrates the powerset lattice for the running example in Table 4. The effectiveness and theoretical background of those algorithms is discussed by Giannela and Wyss [55]. They presented three breadth-first traversal strategies: a bottom-up, a top-down, and a hybrid traversal strategy.

Bottom-up unique discovery traverses the powerset lattice of the schema R from the bottom, beginning with all 1-*combinations* toward the top of the lattice, which is the $|R|$-*combination*. The prefixed number k of *k-combination* indicates the size of the combination. The same notation applies for *k-candidates*, *k-uniques*, and *k-non-uniques*. To generate the set of *2-candidates*, we generate all pairs of

1-non-uniques. *k-candidates* with $k > 2$ are generated by extending the $(k-1)$-*non-uniques* by another non-unique column. After the candidate generation, each candidate is checked for uniqueness. If it is identified as a non-unique, the *k-candidate* is added to the list of *k-non-uniques*.

If the candidate is verified as unique, its minimality has to be checked. The algorithm terminates when $k = |1\text{-}non\text{-}uniques|$. A disadvantage of this candidate generation technique is that redundant uniques and duplicate candidates are generated and tested.

The Apriori idea can also be applied to the top-down approach. Having the set of identified *k-uniques*, one has to verify whether the uniques are minimal. Therefore, for each *k-unique*, all possible $(k-1)$-*subsets* have to be generated and verified. The hybrid approach generates the kth and $(n-k)$th levels simultaneously. Experiments have shown that in most datasets, uniques usually occur in the lower levels of the lattice, which favors bottom-up traversal [3].

HCA is an improved version of the bottom-up Apriori technique [3]. HCA optimizes the candidate generation step, applies statistical pruning and considers functional dependencies that have been inferred on the fly. In terms of candidate generation, HCA applies the optimized join that was introduced for frequent itemset mining [8]. HCA generates candidates by combining only $(k-1)$-*non-uniques* that share the first $k-2$ elements. If no such two non-uniques exist, no candidates are generated and the algorithm terminates before reaching the last level of the powerset lattice. Further pruning can be achieved by considering value histograms and distinct counts that can be retrieved on the fly in previous levels. For example, consider the *1-non-uniques* last and age from Table 4. The column combination {last,age} cannot be a unique based on the value distributions. While the value "`Payne`" occurs three times in last, the column age contains only two distinct values. That means at least two of the rows containing the value "`Payne`" also have a duplicate value in the age column. Using the count distinct values, HCA detects functional dependencies on the fly and leverages them to avoid unnecessary uniqueness checks.

While HCA improves existing bottom-up approaches, it does not perform the early identification of non-uniques in a row-based manner done by GORDIAN. Thus, GORDIAN is faster on datasets with many non-uniques, but HCA works better on datasets with many minimal uniques.

5.1.3 DUCC: traversing the lattice via random walk

While the breadth-first approach for discovering minimal uniques gives the most pruning, a depth-first approach might work well if there are relatively few minimal uniques that are scattered on different levels of the powerset lattice. Depth-first detection of unique column combinations resembles the problem of identifying the most promising paths through the lattice to discover existing minimal uniques and avoid unnecessary uniqueness checks. DUCC is a depth-first approach that traverses the lattice back and forth based on the uniqueness of combinations [70]. Following a random walk principle by randomly adding columns to non-uniques and removing columns from uniques, DUCC traverses the lattice in a manner that resembles the border between uniques and non-uniques in the powerset lattice of the schema.

DUCC starts with a seed set of *2-non-uniques* and picks a seed at random. Each k-combination is checked using the superset/subset relations and pruned if any of them subsumes the current combination. If no previously identified combination subsumes the current combination DUCC performs uniqueness verification. Depending on the verification, DUCC proceeds with an unchecked $(k-1)$-subset or $(k-1)$-superset of the current k-combination. If no seeds are available, it checks whether the set of discovered minimal uniques and maximal non-uniques correctly complement each other. If so, DUCC terminates; otherwise, a new seed set is generated by complementation.

DUCC also optimizes the verification of minimal uniques by using a position list index (PLI) representation of values of a column combination. In this index, each position list contains the tuple ids that correspond to the same value combination. Position lists with only one tuple id can be discarded, so that the position list index of a unique contains no position lists. To obtain the PLI of a column combination, the position lists in PLIs of all contained columns have to be cross-intersected. In fact, DUCC intersects two PLIs in a similar way in which a hash join operator would join two relations. As a result of using PLIs, DUCC can also apply row-based pruning, because the total number of positions decreases with the size of column combinations. Intuitively, combining columns makes the contained combination values more specific and therefore more likely to be distinct.

DUCC has been experimentally compared to HCA, a column-based approach, and GORDIAN, a row-based unique discovery algorithm. Since DUCC combines row-based and column-based pruning, it performs significantly better [70]. Experiments on smaller datasets showed that while HCA outperforms GORDIAN on low-dimensional data with many uniques, GORDIAN outperforms HCA on datasets with many attributes but few uniques [3].

Furthermore the random walk strategy allows a distributed application of DUCC for better scalability.

5.1.4 SWAN: an incremental approach

SWAN maintains a set of indexes to efficiently find the new sets of minimal uniques and maximal non-uniques after inserting or deleting tuples [5]. SWAN builds such indexes based on existing minimal uniques and maximal non-uniques in a way that avoids a full table scan. SWAN consists of two main components: the *Inserts Handler* and the *Deletes Handler*. The Inserts Handler takes as input a set of inserted tuples, checks all minimal uniques for uniqueness, finds the new sets of minimal uniques and maximal non-uniques, and updates the repository of minimal uniques and maximal non-uniques accordingly. Similarly, the Deletes Handler takes as input a set of deleted tuples, searches for duplicates in all maximal non-uniques, finds the new sets of minimal uniques and maximal non-uniques, and updates the repository accordingly.

5.2 Functional dependencies

A *functional dependency* (FD) over R is an expression of the form $X \rightarrow A$, indicating that $\forall r_i, r_j \in r$ if $r_i[X] = r_j[X]$; then, $r_i[A] = r_j[A]$. That is, any two tuples that agree on X must also agree on A. We refer to X as the left-hand side (LHS) and A as the right-hand side (RHS). Given r, we are interested in finding all non-trivial and minimal FDs $X \rightarrow A$ that hold on r, with non-trivial meaning $A \cap X = \emptyset$ and minimal meaning that there must not be any FD $Y \rightarrow A$ for any $Y \subset X$. A naive solution to the FD discovery problem is as follows.

```
For each possible RHS $A$
    For each possible LHS $X \in R \backslash A$
        For each pair of tuples $r_i$ and $r_j$
            If $r_i[X] = r_j[X]$ and $r_i[A] \neq r_j[A]$ Break
    Return $X \rightarrow A$
```

This algorithm is prohibitively expensive: For each of the $|R|$ possibilities for the RHS, it tests $2^{(|R|-1)}$ possibilities for the LHS, each time having to scan r multiple times to compare all pairs of tuples. However, notice that for $X \rightarrow A$ to hold, the number of distinct values of X must be the same as the number of distinct values of XA—otherwise at least one combination of values of X that is associated with more than one value of A, thereby breaking the FD [75]. Thus, if we pre-compute the number of distinct values of each combination of one or more columns, the algorithm simplifies to:

```
For each possible RHS $A$
    For each possible LHS $X \in R \backslash A$
        If $|\pi_X(r)| = |\pi_{XA}(r)|$
            Return $X \rightarrow A$
```

Recall Table 4. We have $|\pi_{\text{phone}}(r)| = |\pi_{\text{age,phone}}(r)| = |\pi_{\text{last,phone}}(r)|$. Thus, phone → age and phone → last hold. Furthermore, $|\pi_{\text{last,age}}(r)| = |\pi_{\text{last,age,phone}}(r)|$, implying $\{\text{last,age}\} \rightarrow \text{phone}$.

The above algorithm is still inefficient due to the need to compute distinct value counts and test all possible column combinations. As was the case with unique discovery, FD discovery algorithms employ row-based (bottom-up) and column-based (top-down) optimizations, as discussed below. Figure 3 lists the algorithms that are discussed, along with their extensions to conditional FD discovery, which are covered in Sect. 5.2.4. An extensive experimental evaluation of various FD discovery algorithms on different datasets, scaling in both the number of rows and the number of columns, is presented in [113].

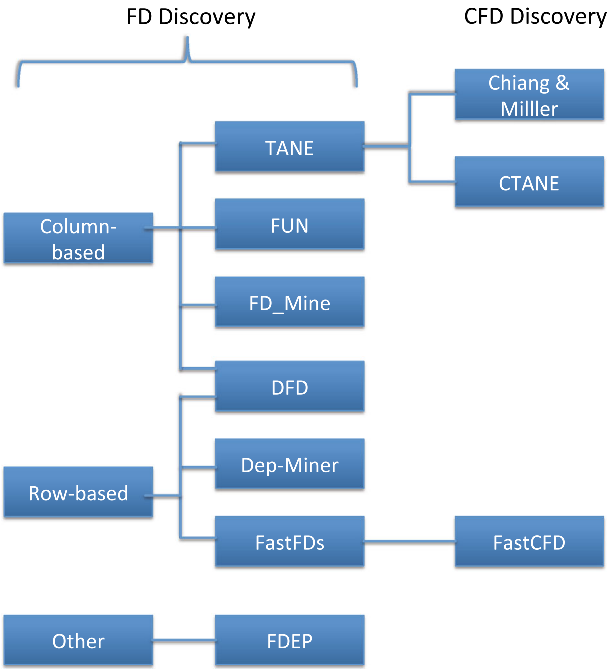


Fig. 3 Classification of algorithms for functional dependency discovery and their extensions to conditional functional dependencies

5.2.1 Column-based algorithms

As was the case with uniques, Apriori-like approaches can help prune the space of FDs that need to be examined, thereby optimizing the first two lines of the above straightforward algorithms. TANE [75], FUN [110], and FD_Mine [139] are three algorithms that follow this strategy, with FUN and FD_Mine introducing additional pruning rules beyond TANE's based on the properties of FDs. They start with sets of single columns in the LHS and work their way up the powerset lattice in a *level-wise* manner. Since only minimal FDs need to be returned, it is not necessary to test possible FDs whose LHS is a superset of an already found FD with the same RHS. For instance, in Table 4, once we find that *phone* → *age* holds, we do not need to consider {first,phone} → age, {last,phone} → age, etc.

Additional pruning rules may be formulated from Armstrong's axioms, i.e., we can prune from consideration those FDs that are logically implied by those we have found so far. For instance, if we find that $A \rightarrow B$ and $B \rightarrow A$, then we can prune all LHS column sets including B, because A

and B are equivalent [139]. Another pruning strategy is to ignore columns sets that have the same number of distinct values as their subsets [110]. Returning to Table 4, observe that phone $\rightarrow$ first does not hold. Since $|\pi_{\text{phone}}(r)| = |\pi_{\text{last,phone}}(r)| = |\pi_{\text{age,phone}}(r)| = |\pi_{\text{last,age,phone}}(r)|$, we know that adding last and/or age to the LHS cannot lead to a valid FD with first on the RHS. To determine these cardinalities the approaches use a so-called partition data structure, which is similar to the PLIs of Sect. 5.1.3.

5.2.2 Row-based algorithms

Row-based algorithms examine pairs of tuples to determine LHS candidates. Dep-Miner [95] and FastFDs [136] are two examples; the FDEP algorithm [52] is also row-based, but the way it ultimately finds FDs that hold is different.

The idea behind row-based algorithms is to compute the so-called difference sets for each pair of tuples, which are the columns on which the two tuples differ. Table 5 enumerates the difference sets in the data from Table 4. Next, we can find candidate LHS's from the difference sets as follows. Pick a candidate RHS, say, phone. The difference sets that include phone, with phone removed are as follows: {first,last,age}, {first,age}, {age}, {last} and {first,last}. This means that there exist pairs of tuples with different values of phone and also with different values of these five difference sets. Next, we find minimal subsets of columns that have a non-empty intersection with each of these difference sets. Such subsets are exactly the LHS's of minimal FDs with phone as the RHS: If two tuples have different values of phone, they are guaranteed to have different values of the columns in the above minimal subsets, and therefore, they do not cause FD violations. Here, there is only one such minimal subset, {last,age}, giving {last,age} $\rightarrow$ phone. If we repeat this process for each possible RHS, and compute minimal subsets corresponding to the LHS's, we obtain the set of minimal FDs. The main difference among row-based FD discovery algorithms is in how they find the minimal subsets.

A recent approach to FD discovery is DFD, which adapts the column-based and row-based pruning of the unique discovery approach DUCC to the problem of FD discovery [6]. DFD decomposes the attribute lattice into $|R|$ lattices, considering each attribute as a possible RHS of an FD. For the remaining $|R| - 1$ attributes, DFD applies a random walk approach by pruning supersets of FD LHS's and subsets of non-FD LHS's.

Table 5 Difference sets computed from Table 4

Tuple ID pair	Difference set
(1,2)	first, last, age, phone
(1,3)	first, age, phone
(1,4)	age, phone
(2,3)	last, phone
(2,4)	first, last, phone
(3,4)	first

DFD has been experimentally compared to TANE, which is a column-based approach, and FastFDs, which is row-based [6]. The experiments confirm that row-based approaches work well on high-dimensional tables with a relatively small number of tuples, while column-based approaches, such as TANE, perform better on low-dimensional tables with a large number of rows. DFD, which benefits from row-based and column-based pruning, performs significantly better than TANE and FastFDs, unless the table has very many tuples and very few columns or vice versa.

5.2.3 Partial and approximate functional dependencies

While FDs were meant for schema design and were enforced by the database management system, there are many instances in which a database may not satisfy some FDs exactly. For example, the application semantics may have changed over time and FD enforcement was disabled, or the database may have been created by integrating conflicting data sources. As a result, it is useful to discover *partial* or *soft* FDs, i.e., those which "almost hold," perhaps with a few exceptions.

A common definition of "almost holding" or "confidence" is the relative size of the largest subset of r on which a given FD holds exactly divided by $|r|$ [58,85]. For example, if we remove tuple 1 from Table 4, the FD $last \rightarrow phone$ holds exactly, and therefore, its confidence is $\frac{3}{4}$. The CORDS system for finding soft FDs uses a slightly different definition: The confidence of $X \rightarrow A$ is $\frac{|\pi_X(r)|}{|\pi_{XA}(r)|}$ [76]. Other definitions involve computing the number of tuples or tuple pairs that do not violate the FD divided by $|r|$ or $|r|^2$, respectively [85].

A related notion is that of *approximate* FD inference, in which partial or exact FDs are generated from a sample of a relation [76,85]. Of course, even if an FD holds exactly on a subset of a relation, it may hold partially on the whole relation. Approximate FD inference is appealing from a computational standpoint as it requires only a sample of the data.

5.2.4 Conditional functional dependencies

Conditional functional dependencies (CFDs), proposed in [46], encode FDs that hold only on well-defined subsets of r. For instance, {first,last} $\rightarrow$ age does not hold on the entire relation in Table 4, but it does hold on a subset of it where first $=$ `Eve`. Formally, a CFD consists of two parts: an embedded FD $X \rightarrow A$ and an accompanying *pattern tuple* with attributes XA. Each cell of a pattern tuple contains a value from the corresponding attribute's domain or a wildcard symbol "_". A pattern tuple identifies a subset of a relation instance in a natural way: A tuple r_i matches a pattern tuple if it agrees on all of its non-wildcard attributes. In the above example, we can formulate a CFD with an embedded FD {first,last} → age and a pattern tuple (`Eve`, _, _), meaning that the embedded FD holds only on tuples which match the pattern, i.e., those with first = `Eve`. We define the *support* of a pattern tuple as the fraction of tuples in r that it matches; for example, the support of (`Eve`, _, _) in Table 4 is $\frac{2}{4}$.

An important special case occurs when the pattern tuple has no wildcards. For example, the following (admittedly accidental) CFD holds on Table 4: age → phone with a pattern tuple (`32, 1234`). In other words, if age = `32`, then phone = `1234`. These special cases, which resemble instance-level association rules (that have 100 % *confidence*), are referred to as *constant* CFDs.

Additionally, as was the case with traditional FDs, we can define approximate CFDs as those that hold on the subset specified by the pattern tableau with some exceptions. For the case of confidence defined as the minimum number of tuples that must be removed to make the CFD hold, [32] gives algorithms for computing summaries that allow the confidence of a CFD to be estimated with guaranteed accuracy.

CFD discovery involves a larger search space than FD discovery: In addition to detecting embedded FDs, we must also find the pattern tuples. CFD discovery algorithms typically extend existing FD discovery algorithms: For example, CTANE [47] and the algorithm from [24] extend TANE, while FastCFD [47] extends FastFDs (see Fig. 3).

Additionally, two simpler problems have been studied. The first is to discover pattern tuples given an embedded FD [59]. The output of this technique is an (approximately) smallest set of pattern tuples, each leading to an approximate CFD with a confidence exceeding a user-supplied confidence threshold, the union of which has a support that exceeds a user-supplied support threshold. The second problem is to report only the constant CFDs. For this problem, CFDMiner has been proposed CFDs [47], which is based on frequent itemset mining, as well as FACD [91], which includes more pruning rules. Also, CFUN, an extension of FUN to generating *frequent* constant CFDs that exceed a given support threshold, has been proposed in [42].

5.3 Inclusion dependencies

An *inclusion dependency* (IND) between column A of relation R and column B of relation S, written $R.A \subseteq S.B$, or $A \subseteq B$ when the relations are clear from context, asserts that each value of A appears in B. Similarly, for two sets of columns X and Y, we write $R.X \subseteq S.Y$, or $X \subseteq Y$, when each distinct combinations of values in X appears in Y. We refer to $R.A$ or $R.X$ as the left-hand side (LHS) and $S.B$ or $S.Y$ as the right-hand side (RHS). INDs with a single-column LHS and RHS are referred to as *unary* and those with multiple columns in the LHS and RHS are called *n-ary*.

A naive solution to IND discovery in relation instances r and s is to try to match each possible LHS with each possible RHS, as shown below.

```
For each column combination $X$ in $R$
  For each column combination $Y$ in $S$
  with $|Y| = |X|$
    If $\forall x \in \pi_X(r)\ \exists y \in \pi_Y(s)$ such that $x = y$
      Return $X \subseteq Y$
```

Note that for any considered X and Y, we can stop as soon as we find a value combination of X that does not appear in Y. Still, this is not an efficient approach as it repeatedly scans r and s when testing the possible LHS and RHS combinations.

5.3.1 Generating unary inclusion dependencies

For the special case of unary INDs, a common approach is to preprocess the data to speed up the subsequent IND discovery. De Marchi et al. [101] propose a technique that scans the database and builds value indices, which are similar to inverted indices. Table 6 shows excerpts of two relations instances, one with columns A and B and the other with columns C and D, and the corresponding value index. The index contains an entry for each value occurring in the database, followed by a list of columns in which this value appears. It is now straightforward to find the INDs: For each possible LHS column, we check if there exists another column that occurs in every row of the value index that contains the LHS column. In Table 6, we have $A \subseteq C$ (whenever A appears in the value index, so does B) and $D \subseteq B$.

The SPIDER algorithm [14] is another example, which preprocesses the data by sorting the values of each column and writing them to disk. Next, each sorted stream, corresponding to the values of one particular attribute, is consumed in parallel in a cursor-like manner, and an IND $A \subseteq B$ can be discarded as soon as we detect a value in A that is not present in B.

Table 6 Excerpts of two relation instances and the corresponding value index

A	B	C	D	Value	Columns
1	3	1	3	1	A, C
1	4	2	3	2	A, C
2	3	4	4	3	B, D
1	5	7	4	4	B, D
				5	B
				7	C

5.3.2 *Generating n-ary inclusion dependencies*

Once all unary INDs have been discovered, De Marchi et al. [101] give a level-wise algorithm, similar to the TANE algorithm for FD discovery, which constructs INDs with i columns from those with $i-1$ columns and prunes INDs that cannot be true. Additionally, hybrid algorithms have been proposed in [87,102] that combine bottom-up and top-down traversal for additional pruning.

The BINDER algorithm uses divide and conquer principles to handle larger datasets than related work [114]. In the divide step, it splits the input dataset horizontally into partitions and vertically into buckets with the goal to fit each partition into main memory. In the conquer step, BINDER then validates the set of all possible inclusion dependency candidates, which are created in the same fashion as in [101], against the partitions. Processing one partition after another, the validation constructs two indexes on each partition, a dense index and an inverted index, and uses them to efficiently prune invalid candidates from the result set.

5.3.3 *Partial and approximate inclusion dependencies*

Similar to partial FDs, partial INDs have been defined as those that almost hold. Using the notion of removing the fewest tuples so that the remainder satisfies the IND exactly, we can define the strength or confidence of a partial IND $X \subseteq Y$ as $\frac{|\pi_X(r)|-|\pi_X(r)/\pi_Y(r)|}{|\pi_X(r)|}$ [96,101]. That is, the confidence is the number of distinct values of X that appear in Y divided by the number of distinct values of X. An equivalent bag-semantics version of this definition is to divide the number of tuples whose X-values appear in Y by the total number of tuples [61]. According to both definitions, the confidence of $B \subseteq D$ in Table 6 is $\frac{3}{4}$. Most of the algorithms discussed above can be extended to discover partial INDs.

5.3.4 *Conditional inclusion dependencies*

Similar to CFDs, *conditional inclusion dependencies* (CINDs) represent INDs that hold only on well-defined subsets of relations [19]. A CIND consists of an embedded standard IND $R.X \subseteq S.Y$ and an accompanying pattern tuple with attributes $R.X_p$ and $S.Y_p$, where $X \cap X_p = \emptyset$ and $Y \cap Y_p = \emptyset$. A CIND specifies that for the subset of R that matches the X_p-values of the pattern tuple, all the X-values must appear in Y, and furthermore, the Y_p values of these tuples in S must match the Y_p-values of the pattern tuple.

For example, suppose a business maintains a Customers table, keyed by cid, and including a column class indicating the class of the customer (e.g., gold or silver). Furthermore, suppose a Services table maintains the services that customers subscribe to, including a service id (sid), a cid and the type of service (e.g., hardware or software). Let Services.cid $\subseteq$ Customers.cid be the embedded IND and let (Services.type $=$ `software`, Customers.class $=$ `gold`) be a pattern tuple. This CIND asserts that the customer ids in the Services table must be drawn from the customer ids in the Customers table, and moreover, gold customers can obtain only software services. On the other hand, a pattern tuple Services.type $=$ `software` implies that only the software services must have customer ids drawn from those in the Customers table (e.g., perhaps hardware services are provided to customers stored in a different table).

Given an embedded IND, the algorithm from [61], which also applies to CFDs, finds pattern tuples that lead to partial CINDs with a confidence satisfying a user-supplied threshold. Similarly, Bauckmann et al. [13] start with a set of approximate INDs and find pattern tuples to turn these into CINDs; however, in contrast to [61], they are not constrained to a single embedded IND. The authors present two algorithms: CINDERELLA, which is based on the Apriori algorithm for association rule mining and employs a breadth-first traversal of the powerset lattice, and PLI, which employs a depth-first traversal instead.

5.3.5 *Generating foreign keys*

IND discovery is a precursor to foreign key detection: A foreign key must satisfy the corresponding inclusion dependency but not all INDs are foreign keys. For example, multiple tables may contain auto-increment columns that serve as surrogate keys, and while inclusion dependencies among them may exist, they are not foreign keys. Once INDs have been discovered, additional heuristics have been proposed, which essentially rank the discovered INDs according to their likelihood of being foreign keys [96,123,143]. A very simple rule may be that if the LHS and RHS have similar names, then A may be a foreign key. It is also useful to examine the set of discovered INDs as a whole: For instance, foreign keys usually are not also primary keys that serve as foreign keys for other tables, and furthermore, a primary key is often referenced by multiple foreign keys in multiple tables, meaning that a primary key should appear in the RHS of multiple INDs, with the LHS's being the foreign keys. More complex rules may reference value distributions; for example, the values in a foreign key column should form a random sample of the values in the corresponding primary key column.

5.4 Other dependencies

Having outlined the algorithms for discovering traditional dependencies and their extensions, we now discuss other types of dependencies related to data profiling. Recently, an extension of FastFDs called FastDC was proposed for discovering *denial constraints*, which are universally quantified first-order logic formulas that subsume FDs, CFDs, INDs and many others [29].

Also, functional dependencies have recently been generalized to *differential dependencies* in [128]. A differential dependency $X \rightarrow Y$ states that if two tuples have “close” values of X (say, the edit distance between them is small), then their A values must also be close.[4] For example, in a financial database, it may be true that if two tuples have similar values of date (e.g., within seven days), then their price values must also be similar (e.g., within 100 dollars). Row- and column-based approaches to discovering differential dependencies were given in [128].

Another interesting class of dependencies involve *order*. For instance, it may be useful to discover that if r is sorted on some attribute A, it is also sorted on B, which gives an order dependency between A and B [56]. This concept was generalized in [57], which proposed *sequential dependencies* (SDs). An SD states that when sorted on A, any two consecutive values of B must be within a predefined range. Given a complete SD, including the attributes A and B as well as the range, [57] gives an algorithm for discovering ranges of values of A in which the SD is approximately satisfied. To the best of our knowledge, the general problem of SD discovery from data is open.

5.5 Summary and discussion

Dependency discovery has been a popular research area in data management. Many of the algorithms and techniques for dependency discovery are based upon classical data mining solutions, such as the Apriori algorithm for efficient generation of association rules. Additional technical challenges arise in the context of conditional dependencies, and novel search space pruning strategies have been developed based on the properties of the given dependencies.

Data profiling results can be not only complex, but also very large. For instance, it is not uncommon to find thousands of functional dependencies in a given dataset. To handle this and focus users on the most important, interesting, or surprising ones, ranking profiling results can help, as Chu et al. [29] show for denial constraints. They suggest two functions, namely succinctness and coverage, to assess their interestingness. Similar interestingness functions for CFDs are given by Chiang and Miller [24]. Additionally, Andritsos et al. [9] show how to rank FDs according to their information content. Furthermore, as we discussed earlier, post-processing methods have been proposed to determine which of the discovered inclusion dependencies are likely to be foreign keys; however, we are not aware of corresponding techniques for uniques and FDs.

6 Profiling tools

Whenever data are too voluminous to fit on a screen or a sheet of paper, data profiling is performed. Even lacking explicit profiling tools, much can already be done with data management tools, such as spreadsheet software, SQL queries, search capabilities of text editors or simply by “eyeballing” the data. Such methods to become acquainted with a new set of data are probably familiar to most readers. The simple method of *sorting* the values of a column can already reveal minimum and maximum values, and scrolling through that sorted data intuits the value distribution, including the number of null values, which are typically sorted to the very beginning or end, and the uniqueness of a column. Finding the median or average values requires additional calculations, whereas it is infeasible to detect dependencies with such simple means.

To allow a more powerful and integrated approach to data profiling, software companies have developed data profiling tools, mostly to profile data residing in relational databases. Most tools discussed in this survey are part of a larger software suite, either for data integration or for data cleansing. We first give an overview of tools that were created in the context of a research project (see Table 7 for a listing). Then, we give a brief glimpse of the vast set of commercial tools with profiling capabilities (see Table 8 for a listing).

6.1 Research tools

In the research literature, data profiling tools are often embedded in data cleaning systems. For example, the Bellman [38] data quality browser supports column analysis (counting the number of rows, distinct values, and NULL values, finding the most frequently occurring values, etc.), and key detection (up to four columns). It also provides a column similarity functionality that finds columns whose value or n-gram distributions are similar; this is helpful for discovering potential foreign keys and join paths. Furthermore, an interesting application of Bellman was to profile the evolution of a database using value distributions and correlations [37]: Which tables change over time and in what ways (insertions, deletions, modifications), and which groups of tables tend to change in the same way. The Potters Wheel tool [122] also supports column analysis, in particular, detecting data types and syntactic structures/patterns.

Data profiling functionality is also included in the MADLib toolkit for scalable in-database analytics [71], including column statistics, such as count, count distinct,

---

[4] Differential dependencies also generalize *matching dependencies* [49] (if two tuples have close values of X, their A values must be exactly the same) and *metric functional dependencies* [89] (if two tuples have the same values of X, their A values must be close).

Table 7 Research tools with data profiling capabilities

Tool	Main goal	Profiling capabilities
Bellman [38]	Data quality browser	Column statistics, column similarity, candidate key discovery
Potters Wheel [122]	Data quality, ETL	Column statistics (including value patterns)
Data Auditor [58]	Rule discovery	CFD and CIND discovery
RuleMiner [28]	Rule discovery	Denial constraint discovery
MADLib [71]	Machine learning	Simple column statistics

Table 8 Commercial data profiling tools/components with their primary capabilities and application areas

Vendor and product	Features → Focus
Attacama DQ Analyzer	Statistics, patterns, uniques → Data exploration, ETL
IBM InfoSphere Information Analyzer	Statistics, patterns, multi-column dependencies → Data exchange, integration, cleansing
Informatica Data Quality	Structure, completeness, anomalies, dependencies → Business rules, cleansing
Microsoft SQL Server Data Profiling Task	Statistics, patterns, dependencies → ETL, cleansing
Oracle Enterprise Data Quality	Statistics, patterns, multi-column dependencies, text profiling → Quality assessment, business rules, cleansing
Paxata Adaptive Data Preparation	Statistics, histograms, semantic data types → Exploration, cleansing, sharing
SAP Information Steward	Statistics, patterns, semantic data types, dependencies → ETL, modeling, cleansing
Splunk Enterprise / Hunk	Patterns, data mining → Search, analytics, visualization
Talend Data Profiler	Statistics, patterns, dependencies → ETL, cleansing
Trifacta	Statistics, patterns → Quality assessment, data transformation

minimum and maximum values, quantiles, and the k most frequently occurring values.

Recent data quality tools are dependency-driven: Classical dependencies, such as FDs and INDs, as well as their conditional extensions, may be used to express the intended data semantics, and dependency violations may indicate possible data quality problems. Most research systems require users to supply data quality rules and dependencies, such as GDR [138], Nadeef [34], Semandaq [45] and StreamClean [84]. These systems focus on languages for specifying rules and generating repairs. However, data quality rules are not always known Apriori in unfamiliar and undocumented datasets, in which case data profiling, and dependency discovery in particular, is an important prerequisite to data cleaning. Notably, many of these systems perform a focused profiling of counting the number of inconsistent tuples with respect to the given rules.

There are at least two research prototype systems that perform rule discovery to some degree: Data Auditor [58] and RuleMiner [28]. Data Auditor requires an FD as input and generates corresponding CFDs from the data. Additionally, Data Auditor considers FDs similar to the one that is provided by the user and generates corresponding CFDs. The idea is to see if a slightly modified FD can generate a more suitable CFD for the given relation instance. On the other hand, RuleMine does not require any rules as input and instead it is designed to generate all reasonable rules from a given dataset. RuleMiner expresses the discovered rules as *denial constraints*, which are universally quantified first-order logic formulas that subsume FDs, CFDs, INDs and many others. Some of the rules it finds are instance-specific and therefore more general than those a typical data profiling tool would find; for example, in a database of income tax records, RuleMiner might find that if one person, A, has a higher salary than another, B, then Person A must have a higher tax rate than Person B.

6.2 Commercial tools

Because data profiling is such an important capability for many data management tasks, there are various commercial data profiling applications. In many cases, they are a part of a data quality / data cleansing tool suite, to support the use-case of profiling for frequent patterns or rules and then cleaning those records that violate them. In addition, most Extract–Transform–Load tools have some profiling capabilities.

Table 8 mentions prominent examples of current commercial tools, together with their capabilities and application focus, based on the respective product documentations. It is beyond the scope of this survey to provide a market overview or compile feature matrices. We also deliberately refrain from providing static URLs for the various products, because commercial Web sites are too fickle.

Finally, and as mentioned before, every database management system collects and maintains base statistics about the tables it manages. However, they do not readily expose those metadata, the metadata are not always up-to-date and sometimes based only on samples, and their scope is usually limited to simple counts and cardinalities.

7 Next generation profiling

Recent trends in data management have added new challenges but also opportunities for data profiling. First, under the *big data* umbrella, industry and research have turned their attention to data that they do not own or have not made use of yet. Data profiling can help assess which data might be useful and reveals the yet unknown characteristics of such new data. Second, much of the data that shall be exploited is of non-traditional type for data profiling, i.e., non-relational, non-structured (textual), and heterogeneous. And it is often truly "big," both in terms of schema and in terms of data. Many existing profiling methods cannot adequately handle that kind of data: Either they do not scale well, or there simply are no methods yet. Third, different and new data management architectures and frameworks have emerged, including distributed systems, key-value stores, multi-core- or main-memory-based servers, column-oriented layouts, streaming input, etc. We discuss some of these trends and their implications toward data profiling. A more elaborate overview of upcoming challenges of data profiling is in [108].

7.1 Profiling for integration

An important use-case of traditional data profiling methods is data integration. Knowledge about the properties of different data sources is important to create correct schema mappings and data transformations, and to correctly standardize and cleanse the data. For instance, knowledge of inclusion dependencies might hint upon ways to join two yet unrelated tables.

However, data profiling can reach beyond such supportive tasks and assess the *integrability* or ease of integration of datasets and thus also indicate the necessary integration effort, which is vital to project planning. Integration effort might be expressed in terms of similarity, but also in terms of manmonths or in terms of which tools are needed.

Like integration projects themselves, integrability has two dimensions, namely schematic fit and data fit. *Schematic fit* is the degree to which two schemata complement and overlap each other and can be determined using schema matching techniques [44]. Smith et al. [127] have recognized that schema matching techniques often play the role of profiling tools: Rather than using them to derive schema mappings and perform data transformation, they might assess project feasibility. Finally, the mere matching of schema elements might not suffice as a profiling-for-integration result: Additional column metadata can provide further details about the integration difficulty.

Data fit is the (estimated) number of real-world objects that are represented in both datasets, or that are represented multiple times in a single dataset and how different they are. Such multiple representations are typically identified using entity matching methods (also known as record linkage, duplicate detection, etc.) [27]. However, estimating the number of matches without actually performing the matching on the entire dataset is an open problem.

7.2 Profiling non-relational data

With the rapid growth of the World Wide Web, semi-structured data, such as XML and RDF data, and non-structured data, such as text document corpora, have become more important. The more flexible structure of non-relational datasets opens new challenges for profiling algorithms. So far, most methods apply only to or were developed for relational data. Below, we give an overview of both existing work that applies traditional profiling algorithms, as well as existing work about data-model-specific profiling approaches, to non-relational data. We focus on the three most relevant non-relational data formats: XML, RDF, and text documents.

7.2.1 XML

XML is the quasi-standard for exchanging data on the Web. Many applications, especially Web services, provide their results as XML documents. Because the XML structure explicitly contains markup and schema information, different profiling approaches have to be considered. Apart from that, Web services themselves are accessible through XML documents, such as WSDL and SOAP files, which are also worth profiling for Web service inspection and categorization.

There has already been a number of research approaches and proposals with a focus on statistical analysis of XML-formatted data. They concentrate either on the DTD structure, the XSD schema structure, or the inherent structure of XML documents. The analysis concentrates on gathering statistics about the number of root elements, attributes, the depth of content models, etc. [26,105,106,124].

Further approaches focus on algorithms that identify traditional relational dependencies in XML data. While Vincent et al. extend the notion of FDs to XML data [132], Yu et al.s [140] present an approach for discovering redundancies based on identified XML FD. There have also been adaptations of unique and key discovery concepts and algorithms to XML data [22]. Due to the more relaxed structure of XML, these approaches identify approximate keys [62] or validate the consistency of the identified keys against XSD definitions [10].

As many XML documents do not refer to a specific schema, a relevant application of profiling approaches is to support the process of schema extraction [17,69]. Additionally, the vast amount of existing documents do not always comply to specified syntactical rules [88], which can be identified via appropriate profiling techniques.

7.2.2 RDF

Although profiling tasks for XML data can easily be adapted to RDF datasets and vice versa, the requirement for RDF data to be machine readable and its important use-case Linked Open Data (LOD) give rise to RDF-specific challenges for data profiling. There are already some tools that generate metadata for a given RDF dataset. For example, LODStats is a stream-based approach for gathering comprehensive statistics about RDF datasets [12].

ProLOD++ provides additional functionalities by applying clustering and rule mining techniques [1]. When profiling RDF data, there are many interesting metadata beyond simple statistics and patterns of RDF statement elements, including synonymously used properties [4], inverse relationships of properties, the conformity and consistence of RDF structured data to the corresponding ontology [2], and the distribution of literals and de-referenceable resource URIs from different namespaces.

Because of the heterogeneity of interlinked sources, it is vital to identify where specific facts come from and how reliable they are. Therefore, another interesting task for profiling RDF data is provenance analysis [18].

7.2.3 Text

Many text analysis approaches and applications can be regarded as text profiling tasks. Statistical methods are used for tasks, such as information extraction [125], part-of-speech tagging [20], and text categorization [83].

Specifically, in the field of author attribution, there has been research on defining interesting features, such as word-length distributions, average number of syllables [73].

Additionally, linguistic metrics, such as distinctiveness, type-token ratio, and Simpson's index have been proposed to measure the style and diversity of text documents. The task of profiling can target single documents, such as a paper or a book, as well as sets of documents, such as Web document corpora, product reviews, or user comments. More sophisticated applications that use metadata generated through profiling include sentiment analysis and opinion mining [93,112].

7.3 Profiling dynamic data

Data profiling describes an instance of a dataset at a particular time. Since many applications work on frequently changing data, it is desirable to re-profile a dataset after a change, such as a deletion, insertion, or update, in order to obtain up-to-date metadata. Simple aggregates are easy to maintain incrementally, but many statistics needed for column analysis, such as distinct value counts, cannot be maintained exactly using limited space and time. For these aggregates, stream sketching techniques [53] may be used to maintain approximate answers. There are also techniques for continuously updating discovered association rules [131] and clusters [43].

Dependency detection may be too time-consuming for repeated execution on the entire dataset. Thus, it is necessary to incrementally update the metadata without processing the complete dataset again. One example is SWAN, an approach for unique discovery on dynamic datasets with insertions and deletions [5] as reported in Sect. 5.1.4. Also, Wang et al. present an approach for maintaining discovered FDs after data deletions [134]. From a data cleaning standpoint, there are solutions for incremental detection of FD and CFD violations [50], and incremental data repairing with respect to FDs and CFDs [30]. In general, incremental solutions for FDs, CFDs, INDs, and CINDs on growing and changing datasets remain challenges for future research.

7.4 Profiling on new architectures

There are at least two database architecture trends that affect profiling. The first is column versus row storage. Column-store systems appear to have a natural computational advantage, at least in terms of the column analysis tasks we discussed in Sect. 3, since they can directly fetch the column of interest and compute statistics on it. However, if all columns are to be profiled, the entire dataset must be read and the only remaining advantage of column stores may be their potential compression. The second trend is that of distributed and cloud data management. This introduces additional profiling challenges, such as combining statistics from multiple nodes into final per-column analysis. There has been some work on detecting FD and CFD violations in a distributed database [48,50], but many other problems in this space, such as efficient dependency detection in distributed data, remain open.

7.5 Visualization

Because data profiling mostly targets human users, effectively visualizing any profiling results is of utmost importance. Only then can users interpret results and react to them. A suggestion for a visual data profiling tool is the Profiler system by Kandel et al. [81]. A strong cooperation between the database community, which produces the data and metadata to be visualized, and the visualization community, which enables users to understand and make use of the data, is needed.

8 Summary

In this article, we provided a comprehensive survey of the state of the art in data profiling: the set of activities and processes to determine metadata about a given database. We discussed single-column profiling tasks such as identifying data types, value distributions and patterns, and multi-column tasks such as detecting various kinds of dependencies. As the amount of data and users who require access to data increase, efficient and effective data profiling will continue to be an important data management problem in research and practice. While many data profiling algorithms have been proposed and implemented in research prototypes and commercial tools, further work is needed, especially in the context of profiling new types of data, supporting and leveraging new data management architectures, and interpreting and visualizing data profiling results.

References

1. Abedjan, Z., Grütze, T., Jentzsch, A., Naumann, F.: Mining and profiling RDF data with ProLOD++. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1198–1201 (2014). Demo
2. Abedjan, Z., Lorey, J., Naumann, F.: Reconciling ontologies and the web of data. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 1532–1536 (2012)
3. Abedjan, Z., Naumann, F.: Advancing the discovery of unique column combinations. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 1565–1570 (2011)
4. Abedjan, Z., Naumann, F.: Synonym analysis for predicate expansion. In: Proceedings of the Extended Semantic Web Conference (ESWC), pp. 140–154 (2013)
5. Abedjan, Z., Quiané-Ruiz, J.-A., Naumann, F.: Detecting unique column combinations on dynamic data. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1036–1047 (2014)
6. Abedjan, Z., Schulze, P., Naumann, F.: DFD: efficient functional dependency discovery. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 949–958 (2014)
7. Agrawal, D., Bernstein, P., Bertino, E., Davidson, S., Dayal, U., Franklin, M., Gehrke, J., Haas, L., Halevy, A., Han, J., Jagadish, H.V., Labrinidis, A., Madden, S., Papakonstantinou, Y., Patel, J.M., Ramakrishnan, R., Ross, K., Shahabi, C., Suciu, D., Vaithyanathan, S., Widom, J.: Challenges and opportunities with Big Data. Technical report, Computing Community Consortium. http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf (2012)
8. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 487–499 (1994)
9. Andritsos, P., Miller, R.J., Tsaparas, P.: Information-theoretic tools for mining database structure from large data sets. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 731–742 (2004)
10. Arenas, M., Daenen, J., Neven, F., Ugarte, M., Van den Bussche, J., Vansummeren, S.: Discovering XSD keys from XML data. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 61–72 (2013)
11. Astrahan, M.M., Schkolnick, M., Kyu-Young, W.: Approximating the number of unique values of an attribute without sorting. Inf. Syst. **12**(1), 11–15 (1987)
12. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats—an extensible framework for high-performance dataset analytics. In: Proceedings of the International Conference on Knowledge Engineering and Knowledge Management (EKAW), pp. 353–362 (2012)
13. Bauckmann, J., Abedjan, Z., Müller, H., Leser, U., Naumann, F.: Discovering conditional inclusion dependencies. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 2094–2098 (2012)
14. Bauckmann, J., Leser, U., Naumann, F., Tietz, V.: Efficiently detecting inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1448–1450 (2007)
15. Benford, F.: The law of anomalous numbers. Proc. Am. Philos. Soc. **78**(4), 551–572 (1938)
16. Berti-Equille, L., Dasu, T., Srivastava, D.: Discovery of complex glitch patterns: a novel approach to quantitative data cleaning. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 733–744 (2011)
17. Bex, G.J., Neven, F., Vansummeren, S.: Inferring XML schema definitions from XML data. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 998–1009 (2007)
18. Böhm, C., Lorey, J., Naumann, F.: Creating void descriptions for web-scale data. J. Web Semant. **9**(3), 339–345 (2011)
19. Bravo, L., Fan, W., Ma, S.: Extending dependencies with conditions. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 243–254 (2007)
20. Brill, E.: Transformation-based error-driven learning and natural language processing: a case study in part-of-speech tagging. Comput. Linguist. **21**(4), 543–565 (1995)
21. Brin, S., Motwani, R., Silverstein, C.: Beyond market baskets: generalizing association rules to correlations. SIGMOD Rec. **26**(2), 265–276 (1997)
22. Buneman, P., Davidson, S.B., Fan, W., Hara, C.S., Tan, W.C.: Reasoning about keys for XML. Inf. Syst. **28**(8), 1037–1063 (2003)
23. Chandola, V., Kumar, V.: Summarization—compressing data into an informative representation. Knowl. Inf. Syst. **12**(3), 355–378 (2007)
24. Chiang, F., Miller, R.J.: Discovering data quality rules. Proc. VLDB Endow. **1**, 1166–1177 (2008)
25. Chiang, R.H.L., Cecil, C.E.H., Lim, E.-P.: Linear correlation discovery in databases: a data mining approach. Data Knowl. Eng. **53**(3), 311–337 (2005)

26. Choi, B.: What are real DTDs like? In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB), pp. 43–48 (2002)
27. Christen, P.: Data Matching. Springer, Berlin (2012)
28. Chu, X., Ilyas, I., Papotti, P., Ye, Y.: RuleMiner: data quality rules discovery. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1222–1225 (2014)
29. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. Proc. VLDB Endow. **6**(13), 1498–1509 (2013)
30. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: consistency and accuracy. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 315–326 (2007)
31. Cormode, G., Garofalakis, M., Haas, P.J., Jermaine, C.: Synopses for massive data: samples, histograms, wavelets, sketches. Found. Trends Databases **4**(13), 1–294 (2011)
32. Cormode, G., Golab, L., Flip, K., McGregor, A., Srivastava, D., Zhang, X.: Estimating the confidence of conditional functional dependencies. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 469–482 (2009)
33. Cormode, G., Korn, F., Muthukrishnan, S., Srivastava, D.: Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In: Proceedings of the Symposium on Principles of Database Systems (PODS), pp. 263–272 (2006)
34. Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A., Ilyas, I.F., Ouzzani, M., Tang, N.: NADEEF: a commodity data cleaning system. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 541–552 (2013)
35. Das, A., Ng, W.-K., Woon, Y.-K.: Rapid association rule mining. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 474–481 (2001)
36. Dasu, T., Johnson, T.: Hunting of the snark: finding data glitches using data mining methods. In: Proceedings of the International Conference on Information Quality (IQ), pp. 89–98 (1999)
37. Dasu, T., Johnson, T., Marathe, A.: Database exploration using database dynamics. IEEE Data Eng. Bull. **29**(2), 43–59 (2006)
38. Dasu, T., Johnson, T., Muthukrishnan, S., Shkapenyuk, V.: Mining database structure; or, how to build a data quality browser. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 240–251 (2002)
39. Dasu, T., Loh, J.M.: Statistical distortion: consequences of data cleaning. Proc. VLDB Endow. **5**(11), 1674–1683 (2012)
40. Dasu, T., Loh, J.M., Srivastava, D.: Empirical glitch explanations. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp. 572–581 (2014)
41. Deshpande, A., Garofalakis, M., Rastogi, R.: Independence is good: dependency-based histogram synopses for high-dimensional data. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 199–210 (2001)
42. Diallo, T., Novelli, N., Petit, J.-M.: Discovering (frequent) constant conditional functional dependencies. Int. J. Data Min. Model. Manag. **4**(3), 205–223 (2012)
43. Ester, M., Kriegel, H.-P., Sander, J., Wimmer, M., Xu, X.: Incremental clustering for mining in a data warehousing environment. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 323–333 (1998)
44. Euzenat, J., Shvaiko, P.: Ontology Matching, 2nd edn. Springer, Berlin (2013)
45. Fan, W., Geerts, F., Jia, X.: Semandaq: a data quality system based on conditional functional dependencies. Proc. VLDB Endow. **1**(2), 1460–1463 (2008)
46. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. **33**(2), 1–48 (2008)
47. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. IEEE Trans. Knowl. Data Eng. **23**(4), 683–698 (2011)
48. Fan, W., Geerts, F., Ma, S., Müller, H.: Detecting inconsistencies in distributed data. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 64–75 (2010)
49. Fan, W., Jia, X., Li, J., Ma, S.: Reasoning about record matching rules. Proc. VLDB Endow. **2**(1), 407–418 (2009)
50. Fan, W., Li, J., Tang, N., Yu, W.: Incremental detection of inconsistencies in distributed data. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 318–329 (2012)
51. Fernau, H.: Algorithms for learning regular expressions from positive data. Inf. Comput. **207**(4), 521–541 (2009)
52. Flach, P.A., Savnik, I.: Database dependency discovery: a machine learning approach. AI Commun. **12**(3), 139–160 (1999)
53. Ganguly, S.: Counting distinct items over update streams. Theor. Comput. Sci. **378**(3), 211–222 (2007)
54. Garofalakis, M., Keren, D., Samoladas, V.: Sketch-based geometric monitoring of distributed stream queries. Proc. VLDB Endow. **6**(10), 937–948 (2013)
55. Giannella, C., Wyss, C.: Finding minimal keys in a relation instance (1999). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.41.7086
56. Ginsburg, S., Hull, R.: Order dependency in the relational model. Theor. Comput. Sci. **26**, 149–195 (1983)
57. Golab, L., Karloff, H., Korn, F., Saha, A., Srivastava, D.: Sequential dependencies. Proc. VLDB Endow. **2**(1), 574–585 (2009)
58. Golab, L., Karloff, H., Korn, F., Srivastava, D.: Data auditor: exploring data quality and semantics using pattern tableaux. Proc. VLDB Endow. **3**(1–2), 1641–1644 (2010)
59. Golab, L., Karloff, H., Korn, F., Srivastava, D., Bei, Y.: On generating near-optimal tableaux for conditional functional dependencies. Proc. VLDB Endow. **1**(1), 376–390 (2008)
60. Golab, L., Korn, F., Srivastava, D.: Discovering pattern tableaux for data quality analysis: a case study. In: Proceedings of the International Workshop on Quality in Databases (QDB), pp. 47–53 (2011)
61. Golab, L., Korn, F., Srivastava, D.: Efficient and effective analysis of data quality using pattern tableaux. IEEE Data Eng. Bull. **34**(3), 26–33 (2011)
62. Grahne, G., Zhu, J.: Discovering approximate keys in XML data. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 453–460 (2002)
63. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Min. Knowl. Discov. **1**(1), 29–53 (1997)
64. Gunopulos, D., Khardon, R., Mannila, H., Sharma, R.S.: Discovering all most specific sentences. ACM Trans. Database Syst. **28**, 140–174 (2003)
65. Haas, P.J., Naughton, J.F., Seshadri, S., Stokes, L.: Sampling-based estimation of the number of distinct values of an attribute. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 311–322 (1995)
66. Hainaut, J.-L., Henrard, J., Englebert, V., Roland, D., Hick, J.-M.: Database reverse engineering. In: Liu, L., Tamer Özsu, M. (eds.) Encyclopedia of Database Systems, pp. 723–728. Springer, Heidelberg (2009)
67. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. **29**(2), 1–12 (2000)
68. Hanrahan, P.: Analytic database technology for a new kind of user—the data enthusiast (keynote). In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 577–578 (2012)
69. Hegewald, J., Naumann, F., Weis, M.: XStruct: efficient schema extraction from multiple and large XML databases. In: Proceed-

ings of the International Workshop on Database Interoperability (InterDB) (2006)

70. Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. Proc. VLDB Endow. **7**(4), 301–312 (2013)
71. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The MADlib analytics library or MAD skills, the SQL. Proc. VLDB Endow. **5**(12), 1700–1711 (2012)
72. Hipp, J., Güntzer, U., Nakhaeizadeh, G.: Algorithms for association rule mining—a general survey and comparison. SIGKDD Explor. **2**(1), 58–64 (2000)
73. Holmes, D.I.: Authorship attribution. Comput. Humanit. **28**, 87–106 (1994)
74. Hua, M., Pei, J.: Cleaning disguised missing data: a heuristic approach. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp. 950–958 (2007)
75. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
76. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 647–658 (2004)
77. Ioannidis, Y.: The history of histograms (abridged). In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 19–30 (2003)
78. Jain, A.K., Narasimha Murty, M., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. **31**(3), 264–323 (1999)
79. Johnson, T.: Encyclopedia of Database Systems, chapter Data Profiling. Springer, Heidelberg (2009)
80. Kache, H., Han, W.-S., Markl, V., Raman, V., Ewen, S.: POP/FED: progressive query optimization for federated queries in DB2. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 1175–1178 (2006)
81. Kandel, S., Parikh, R., Paepcke, A., Hellerstein, J., Heer, J.: Profiler: integrated statistical analysis and visualization for data quality assessment. In: Proceedings of Advanced Visual Interfaces (AVI), pp. 547–554 (2012)
82. Kang, J., Naughton, J.F.: On schema matching with opaque column names and data values. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 205–216 (2003)
83. Keim, D.A., Oelke, D.: Literature fingerprinting: a new method for visual literary analysis. In: Proceedings of Visual Analytics Science and Technology (VAST), pp. 115–122 (2007)
84. Khoussainova, N., Balazinska, M., Suciu, D.: Towards correcting input data errors probabilistically using integrity constraints. In: Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), pp. 43–50 (2006)
85. Kivinen, J., Mannila, H.: Approximate inference of functional dependencies from relations. In: Proceedings of the International Conference on Database Theory (ICDT), pp. 129–149 (1995)
86. Koehler, H., Leck, U., Link, S., Prade, H.: Logical foundations of possibilistic keys. In: Fermé, E., Leite, J. (eds.) Logics in Artificial Intelligence, volume 8761 of Lecture Notes in Computer Science, pp. 181–195. Springer, Heidelberg (2014)
87. Koeller, A., Rundensteiner, E.A.: Heuristic strategies for the discovery of inclusion dependencies and other patterns. J. Data Semant. V. **3870**, 185–210 (2006)
88. Korn, F., Saha, B., Srivastava, D., Ying, S.: On repairing structural problems in semi-structured data. Proc. VLDB Endow. **6**(9), 601–612 (2013)
89. Koudas, N., Saha, A., Srivastava, D., Venkatasubramanian, S.: Metric functional dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1275–1278 (2009)
90. Laney, D.: 3D data management: controlling data volume, velocity and variety. Technical report, Gartner (2001)
91. Li, J., Liu, J., Toivonen, H., Yong, J.: Effective pruning for the discovery of conditional functional dependencies. Comput. J. **56**(3), 378–392 (2013)
92. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H.V.: Regular expression learning for information extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 21–30 (2008)
93. Liu, B.: Sentiment analysis and subjectivity. Handbook of Natural Language Processing, 2nd edn. Chapman and Hall/CRC, London (2010)
94. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data—a review. IEEE Trans. Knowl. Data Eng. **24**(2), 251–264 (2012)
95. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and Armstrong relations. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 350–364 (2000)
96. Lopes, S., Petit, J.-M., Toumani, F.: Discovering interesting inclusion dependencies: application to logical database tuning. Inf. Syst. **27**(1), 1–19 (2002)
97. Lucchesi, C.L., Osborn, S.L.: Candidate keys for relations. J. Comput. Syst. Sci. **17**(2), 270–279 (1978)
98. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with Cupid. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 49–58 (2001)
99. Mannino, M.V., Chu, P., Sager, T.: Statistical profile estimation in database systems. ACM Comput. Surv. **20**(3), 191–221 (1988)
100. De Marchi, F., Lopes, S., Petit, J.-M.: Efficient algorithms for mining inclusion dependencies. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 464–476 (2002)
101. De Marchi, F., Lopes, S., Petit, J.-M.: Unary and n-ary inclusion dependency discovery in relational databases. J. Intell. Inf. Syst. **32**, 53–73 (2009)
102. De Marchi, F. , Petit, J.-M.: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), pp. 27–34 (2003)
103. Markowitz, V.M., Makowsky, J.A.: Identifying extended entity-relationship object structures in relational schemas. IEEE Trans. Softw. Eng. **16**(8), 777–790 (1990)
104. Maydanchik, A.: Data Quality Assessment. Technics Publications, New Jersey (2007)
105. Mignet, L., Barbosa, D., Veltri, P.: The XML web: a first study. In: Proceedings of the International World Wide Web Conference (WWW), pp. 500–510 (2003)
106. Mlynkova, I., Toman, K., Pokorný, J.: Statistical analysis of real XML data collections. In: Proceedings of the International Conference on Management of Data (COMAD), pp. 15–26 (2006)
107. Morton, K., Balazinska, M., Grossman, D., Mackinlay, J.: Support the data enthusiast: challenges for next-generation data-analysis systems. Proc. VLDB Endow. **7**(6), 453–456 (2014)
108. Naumann, F.: Data profiling revisited. SIGMOD Rec. **42**(4), 40–49 (2013)
109. Naumann, F., Ho, C.-T., Tian, X., Haas, L., Megiddo, N.: Attribute classification using feature analysis. In: Proceedings of the International Conference on Data Engineering (ICDE), p 271 (2002)
110. Novelli, N., Cicchetti, R.: FUN: an efficient algorithm for mining functional and embedded dependencies. In: Proceedings of the

International Conference on Database Theory (ICDT), pp. 189–203 (2001)

111. Ntarmos, N., Triantafillou, P., Weikum, G.: Distributed hash sketches: scalable, efficient, and accurate cardinality estimation for distributed multisets. ACM Trans. Comput. Syst. **27**(1), 1–53 (2009)
112. Pang, B., Lee, L.: Opinion mining and sentiment analysis. Found. Trends Inf. Retr. **2**(1–2), 1–135 (2008)
113. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.-P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: an experimental evaluation of seven algorithms. Proc. VLDB Endow. **8**(10) (2015)
114. Papenbrock, T., Kruse, S., Quiané-Ruiz, J.-A., Naumann, F.: Divide & conquer-based inclusion dependency discovery. Proc. VLDB Endow. **8**(7), 774–785 (2015)
115. Park, J.S., Chen, M.-S., Yu, P.S.: Using a hash-based method with transaction trimming for mining association rules. IEEE Trans. Knowl. Data Eng. **9**, 813–825 (1997)
116. Petit, J.-M., Kouloumdjian, J., Boulicaut, J.-F., Toumani, F.: Using queries to improve database reverse engineering. In: Proceedings of the International Conference on Conceptual Modeling (ER), pp. 369–386 (1994)
117. Pipino, L., Lee, Y., Wang, R.: Data quality assessment. Commun. ACM **4**, 211–218 (2002)
118. Poosala, V., Haas, P.J., Ioannidis, Y.E., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 294–305 (1996)
119. Poosala, V., Ioannidis, Y.E.: Selectivity estimation without the attribute value independence assumption. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 486–495 (1997)
120. Pyle, D.: Data Preparation for Data Mining. Morgan Kaufmann, Burlington (1999)
121. Rahm, E., Do, H.-H.: Data cleaning: problems and current approaches. IEEE Data Eng. Bull. **23**(4), 3–13 (2000)
122. Raman, V., Hellerstein, J.M.: Potters wheel: an interactive data cleaning system. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 381–390 (2001)
123. Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U.: A machine learning approach to foreign key discovery. In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB) (2009)
124. Sahuguet, A., Azavant, F.: Building light-weight wrappers for legacy Web data-sources using W4F. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 738–741 (1999)
125. Sarawagi, S.: Information extraction. Found. Trends Databases **1**(3), 261–377 (2008)
126. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: GORDIAN: efficient and scalable discovery of composite keys. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 691–702 (2006)
127. Smith, K.P., Morse, M., Mork, P., Li, M.H., Rosenthal, A., Allen, M.D., Seligman, L.: The role of schema matching in large enterprises. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR) (2009)
128. Song, S., Chen, L.: Differential dependencies: reasoning and discovery. ACM Trans. Database Syst. **36**(3), 16:1–16:41 (2011)
129. Stonebraker, M., Bruckner, D., Ilyas, I.F., Beskales, G., Cherniack, M., Zdonik, S., Pagan, A., Xu, S.: Data curation at scale: the Data Tamer system. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR) (2013)
130. Chen, M., Hun, J., Yu, P.S.: Data mining: an overview from a database perspective. IEEE Trans. Knowl. Data Eng. **8**, 866–883 (1996)
131. Tsai, P.S.M., Lee, C.-C., Chen, A.L.P.: An efficient approach for incremental association rule mining. Methodologies for Knowledge Discovery and Data Mining. volume 1574 of Lecture Notes in Computer Science, pp. 74–83. Springer, Heidelberg (1999)
132. Vincent, M.W., Liu, J., Liu, C.: Strong functional dependencies and their application to normal forms in XML. ACM Trans. Database Syst. **29**(3), 445–462 (2004)
133. Vogel, T., Naumann, F.: Instance-based "one-to-some" assignment of similarity measures to attributes. In: Proceedings of the International Conference on Cooperative Information Systems (CoopIS), pp. 412–420 (2011)
134. Wang, S.-L., Tsou, W.-C., Lin, J.-H., Hong, T.-P.: Maintenance of discovered functional dependencies: incremental deletion. Intelligent Systems Design and Applications, volume 23 of Advances in Soft Computing, pp. 579–588. Springer, Heidelberg (2003)
135. Xindong, W., Zhang, C., Zhang, S.: Efficient mining of both positive and negative association rules. ACM Trans. Inf. Syst. **22**(3), 381–405 (2004)
136. Wyss, C., Giannella, C., Robertson, E.L.: FastFDs: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In: Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK), pp. 101–110 (2001)
137. Xu, R., Wunsch II, D.C.: Survey of clustering algorithms. IEEE Trans. Neural Netw. **16**(3), 645–678 (2005)
138. Yakout, M., Elmagarmid, A.K., Neville, J., Ouzzani, M.: GDR: a system for guided data repair. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1223–1226 (2010)
139. Yao, H., Hamilton, H.J.: Mining functional dependencies from data. Data Min. Knowl. Discov. **16**(2), 197–219 (2008)
140. Yu, C., Jagadish, H.V.: Efficient discovery of XML data redundancies. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 103–114 (2006)
141. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
142. Zhang, M., Chakrabarti, K.: InfoGather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 145–156 (2013)
143. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. Proc. VLDB Endow. **3**(1–2), 805–814 (2010)
144. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: Automatic discovery of attributes in relational databases. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 109–120 (2011)

Week 6: Taming technical bias

Taming Technical Bias in Machine Learning Pipelines [*]

Sebastian Schelter
University of Amsterdam & Ahold Delhaize
Amsterdam, The Netherlands
s.schelter@uva.nl

Julia Stoyanovich
New York University
New York, NY, USA
stoyanovich@nyu.edu

Abstract

Machine Learning (ML) is commonly used to automate decisions in domains as varied as credit and lending, medical diagnosis, and hiring. These decisions are consequential, imploring us to carefully balance the benefits of efficiency with the potential risks. Much of the conversation about the risks centers around bias — a term that is used by the technical community ever more frequently but that is still poorly understood. In this paper we focus on technical bias — a type of bias that has so far received limited attention and that the data engineering community is well-equipped to address. We discuss dimensions of technical bias that can arise through the ML lifecycle, particularly when it's due to preprocessing decisions or post-deployment issues. We present results of our recent work, and discuss future research directions. Our over-all goal is to support the development of systems that expose the knobs of responsibility to data scientists, allowing them to detect instances of technical bias and to mitigate it when possible.

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. Much of this conversation centers around *bias* — a term that is used by the technical community ever more frequently but that is still poorly understood.

In their seminal 1996 paper, Friedman and Nissenbaum identified three types of bias that can arise in computer systems: pre-existing, technical, and emergent [9]. We briefly discuss these in turn, see Stoyanovich et al. [33] for a more comprehensive overview.

- *Pre-existing bias* has its origins in society. In ML applications, this type of bias often exhibits itself in the input data; detecting and mitigating it is the subject of much research under the heading of algorithmic fairness [5]. Importantly, the presence or absence of pre-existing bias cannot be scientifically verified, but rather is postulated based on a belief system [8, 12]. Consequently, the effectiveness — or even the validity — of a technical attempt to mitigate pre-existing bias is predicated on that belief system.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

---

[*] This work was supported in part by NSF Grants No. 1926250, 1934464, and 1922658, and 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.

- *Technical bias* arises due to the operation of the technical system itself, and can amplify pre-existing bias. The bad news is that, as we argue in the remainder of this paper, the risks of introducing technical bias in ML pipelines abound. The good news is that, unlike with pre-existing bias, there is no ambiguity about whether a technical fix should be attempted: if technical systems we develop are introducing bias, then we should be able to instrument these systems to measure it and understand its cause. It may then be possible to mitigate this bias and to check whether the mitigation was effective.

- *Emergent bias* arises in the context of use of the technical system. In Web ranking and recommendation in e-commerce, a prominent example is "rich-get-richer": searchers tend to trust the systems to indeed show them the most suitable items at the top positions, which in turn shapes a searcher's idea of a satisfactory answer.

In this paper, we focus on technical bias, — a type of bias that has so far received limited attention, particularly when it's due to preprocessing decisions or post-deployment issues, and that the data engineering community is well-equipped to address. Our over-all goal is to support the development of systems that *expose the knobs of responsibility to data scientists*, allowing them to detect instances of technical bias, and to mitigate it when possible.

Running example. We illustrate the need for taming technical bias with 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 gender and age groups. This obligation is operationalized as an intersectional fairness criterion, requiring equal false negatives rates for groups of patients identified by a combination of gender and age group.

Consider Ann, a data scientist who is developing this classifier. Following her company's best practices, Ann will start by splitting her dataset into training, validation, and test sets. Ann will then use `pandas`, `scikit-learn` [19], and their accompanying data transformers to explore the data and implement data preprocessing, model selection, tuning, and validation. Ann starts preprocessing by computing value distributions and correlations for the features in her dataset, and by identifying missing values. She will fill these in using a default interpolation method in scikit-learn, replacing missing values with the mode value for that feature. Finally, following the accepted best practices at her company, Ann implements model selection and hyperparameter tuning. As a result of this step, Ann will select a classifier that shows acceptable performance according to her company's standard metrics: it has sufficient accuracy, while also exhibiting sufficiently low variance. When Ann considers the accuracy of her classifier closely, she observes a disparity: accuracy is lower for middle-aged women. Ann is now faced with the challenge of figuring out why this is the case, whether any of her technical choices during pipeline construction contributed to this model bias, and what she can do to mitigate this effect. We will revisit this example, and also discuss issues that may arise after the model is deployed, in the remainder of this paper.

Roadmap. The rest of this paper is organized as follows. In Section 2, we outline the dimensions of technical bias as they relate to two lifecycle views of ML applications: the data lifecycle and the lifecycle of design, development, deployment, and use. Then, in Section 3 we present our recent work on helping data scientists responsibly develop ML pipelines, and validate them post-deployment. We conclude in Section 4 with directions for future research.

2 Dimensions of Technical Bias

There are many different ways in which Ann (or her colleagues who deploy her model) could accidentally introduce technical bias. Some of these relate to the view of ML model development through the lens of the *data lifecycle*. As argued in Stoyanovich et al. [33], responsibility concerns, and important decision points, arise in data sharing, annotation, acquisition, curation, cleaning, and integration. Thus, opportunities for improving data quality and representativeness, controlling for bias, and allowing humans to oversee the process, are missed if we do not consider these earlier data lifecycle stages. We discuss these dimensions of technical bias in Section 2.1. Additional challenges, and opportunities to introduce technical bias, arise after a model is deployed. We discuss these in Section 2.2.

Note that, in contrast to Bower et al. [4] and Dwork et al. [7], who study fairness in ML pipelines in which multiple models are composed, we focus on complex — and typical — pipelines in which bias may arise due to the composition of data preprocessing steps, or to data distribution shifts past deployment.

2.1 Model Development Stage

There are several subtle ways in which data scientists can accidentally introduce data-related bias into their models during the development stage. Our discussion in this section is inspired by the early influential work by Barocas and Selbst [1], and by Lehr and Ohm [15], who highlighted the issues that we will make more concrete.

Data cleaning. Methods for missing value imputation that are based on incorrect assumptions about whether data is missing at random may distort protected group proportions. Consider a form that gives patients a binary choice of gender and also allows to leave gender unspecified. Suppose that about half of the users identify as men and half as women, but that women are more likely to omit gender. Then, if mode imputation (replacing a missing value with the most frequent value for the feature, a common choice in `scikit-learn`) is used, then all (predominantly female) unspecified gender values will be set to male. More generally, multi-class classification for missing value imputation typically only uses the most frequent classes as target variables [3], leading to a distortion for small population groups, because membership in these groups will never be imputed. Next, suppose that some individuals identify as non-binary. Because the system only supports male, female, and unspecified as options, these individuals will leave gender unspecified. If mode imputation is used, then their gender will be set to male. A more sophisticated imputation method will still use values from the active domain of the feature, setting the missing values of gender to either male or female. This example illustrates that bias can arise from an incomplete or incorrect choice of data representation.

Finally, consider a form that has home address as a field. A homeless person will leave this value unspecified, and it is incorrect to attempt to impute it. While dealing with `null` values is known to be difficult and is already considered among the issues in data cleaning, the needs of responsible data management introduce new problems. Further, data quality issues often disproportionately affect members of historically disadvantaged groups [14], and so we risk compounding technical bias due to data representation with pre-existing bias.

Data filtering. Selections and joins can arbitrarily change the proportion of protected groups (e.g., for certain age groups) even if they do not directly use the sensitive attribute (e.g., age) as part of the predicate or of the join key. This change in proportion may be unintended and is important to detect, particularly when this happens during one of many preprocessing steps in the ML pipeline. During model development, Ann might have filtered the data by zip code or county to get a sample that is easier to work with. Demographic attributes such as age and income are highly correlated with places of residency, so such a seemingly innocent filtering operation might have heavily biased the data.

Another potential source of technical bias is the increasingly common usage of pre-trained word embeddings. For example, Ann's code might replace a textual name feature with the corresponding vector from a word embedding that is missing for rare, non-western names (due to lack of data representation in the training corpus). If we then filter out records for which no embedding was found, we may disproportionately remove individuals from specific ethnic groups.

Unsound experimentation. Design and evaluation of ML models is a difficult and tedious undertaking and requires data scientists to strictly follow a set of best practices. During this process, it is unfortunately easy to make subtle mistakes that can heavily impact the quality of the resulting model. In previous research, we found that even expert users violate such best practices in highly cited studies [29]. Common mistakes include hyperparameter selection on the test set instead of the validation set, lack of hyperparameter tuning for baseline learners, lack of proper feature normalisation, or ignoring problematic data subsets during training.

While unsound experimentation is a general issue, ignoring problematic data subsets can specifically affect performance for minority and underrepresented groups, because their data might be prone to data quality issues, as we already discussed under *data filtering* above.

2.2 Model Deployment Stage

After the design of a model is finished, the model is deployed into production and produces predictions on unseen data. We outline a set of circumstances which can introduce technical bias at this stage.

Data errors introduced through integration. In modern information infrastructures, data is stored in different environments (e.g., in relational databases, in 'data lakes' on distributed file systems, or behind REST APIs), and it comes in many different formats. Many such data sources do not support integrity constraints and data quality checks, and often there is not even an accompanying schema available as the data is consumed in a 'schema-on-read' manner, where a particular application takes care of the interpretation. Additionally, there is a growing demand for applications consuming semi-structured data such as text, videos, and images. Due to these circumstances, every real world ML application has to integrate data from multiple sources, and errors in the data sources or during integration may lead to errors in downstream ML models that consume the data.

In our running example in Section 1, it may be the case that patient data is integrated from data sources of different healthcare providers. If one of these providers accidentally changes their schema, or introduces bugs in their data generation procedure, this may negatively impact the predictions for the corresponding patients when their data is used as input to Ann's model.

Distribution shifts. The maintenance of ML applications remains challenging [21], due in large part to unexpected shifts in the distribution of serving data. These shifts originate from changes in the data generating process in the real world, and the problem is exacerbated in situations where different parties are involved in the provision of the data and the training of the model. Many engineering teams, especially in smaller companies, lack ML expert knowledge, and therefore often outsource the training of ML models to data science specialists or cloud ML services. In such cases, the engineering team provides the input data and retrieves predictions, but might not be familiar with details of the model. While ML experts have specialized knowledge to debug models and predictions in such cases [16], there is a lack of automated methods for non-ML expert users to decide whether they can rely on the predictions of an ML model on unseen data. In Ann's case, her final deployed model might work well until new regulations for health care providers change the shape and contents of the patient data that they produce. If her model is not retrained on proper data, its prediction quality may quickly deteriorate.

In the following section we will introduce three software libraries that we developed in recent research to help data scientists like Ann in detecting and mitigating technical bias during model development and deployment.

3 Taming Technical Bias during Model Development and Deployment

In Schelter et al. [29] we described `FairPrep`, a design and evaluation framework for fairness-enhancing interventions in machine learning pipelines that treats data as a first-class citizen. The framework implements a modular data lifecycle, enables re-use of existing implementations of fairness metrics and interventions, and integration of custom feature transformations and data cleaning operations from real world use cases. `FairPrep` pursues the following goals: (i) Expose a developer-centered design throughout the lifecycle, which allows for low effort customization and composition of the framework's components; (ii) Surface discrimination and due process concerns, including disparate error rates, failure of a model to fit the data, and failure of a model to generalize. (iii) Follow software engineering and machine learning best practices to reduce the technical

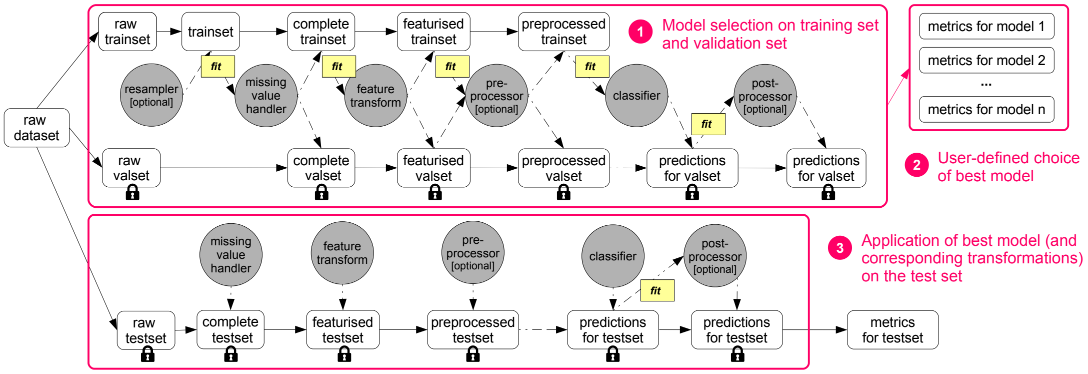


Figure 1: Data life cycle in `FairPrep`, designed to enforce isolation of test data, and to allow for customization through user-provided implementations of different components. An evaluation run consists of three different phases: (1) Learn different models, and their corresponding data transformations, on the training set; (2) Compute performance / accuracy-related metrics of the model on the validation set, and allow the user to select the 'best' model according to their setup; (3) Compute predictions and metrics for the user-selected best model on the held-out test set.

debt of incorporating fairness-enhancing interventions into an already complex development and evaluation scenario [26, 31].

Figure 1 summarizes the architecture of `FairPrep`, which is based on three main principles:

1. Data isolation: to avoid target leakage, user code should only interact with the training set, and never be able to access the held-out test set.
2. Componentization: different data transformations and learning operations should be implementable as single, exchangeable standalone components; the framework should expose simple interfaces to users, supporting low effort customization.
3. Explicit modeling of the data lifecycle: the framework defines an explicit, standardized data lifecycle that applies a sequence of data transformations and model training in a predefined order.

`FairPrep` currently focuses on data cleaning, including different methods for data imputation, and model selection and validation, including hyperparameter tuning, and can be extended to accommodate earlier lifecycle stages, such as data acquisition, integration, and curation. Schelter et al. [29] measured the impact of sound best practices, such as hyperparameter tuning and feature scaling, on the fairness and accuracy of the resulting classifiers, and also showcased how `FairPrep` enables the inclusion of incomplete data into studies and helps analyze the effects.

If Ann wants to ensure that she follows sound experimentation practices during model development, she can use the `FairPrep` library as a runtime platform for experiments, for example to compute various fairness related metrics for the predictions of her classifier. Furthermore, she can leverage the component architecture of `FairPrep` to evaluate different missing value imputation techniques and fairness enhancing interventions to see whether these help with mitigating the low accuracy that she encountered in her model for the predictions for middle-aged women, as discussed in our running example in Section 1.

Source code. A prototype implementation of FairPrep is available at `https://github.com/DataResponsibly/FairPrep`.

3.1 Detecting Data Distribution Bugs Introduced in Preprocessing

In our recent work on the *mlinspect* library [10], we focus on helping data scientists diagnose and mitigate problems to which we collectively refer as *data distribution bugs*. These types of bugs are often introduced during preprocessing, for reasons we outlined in Section 2. For example, preprocessing operations that involve filters or joins can heavily change the distribution of different groups in the training data [35], and missing value imputation can also introduce skew [28]. Recent ML fairness research, which mostly focuses on the use of learning algorithms on static datasets [5] is therefore insufficient, because it cannot address such technical bias originating from the data preparation stage. In addition, we should detect and mitigate such bias as close to its source as possible.

Unfortunately, such data distribution issues 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 [23] often combines relational operations on tabular data with *estimator/transformer pipelines*,[1] a composable and nestable abstraction for combining operations on array data, which originates from `scikit-learn` [19] and has been adopted by popular libraries like `SparkML` [18] and `Tensorflow Transform`. 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 not obvious to the data scientist.

Due to time pressure in their day-to-day activities, most data scientists will not invest the necessary time and effort to manually instrument their code or insert logging statements for tracing as required by model management systems [34, 36]. This calls for the development of tools that support *automated inspection of ML pipelines*, similar to the inspections used by modern IDEs to highlight potentially problematic parts of a program, such as the use of deprecated code or problematic library functions calls. Once data scientists are pointed to such issues, they can use data debuggers like `Dagger` [17] to drill down into the specific intermediate pipeline outputs and explore the root cause of the issue. Furthermore, to be most beneficial, automated inspections need to work with code natively written with popular ML library abstractions.

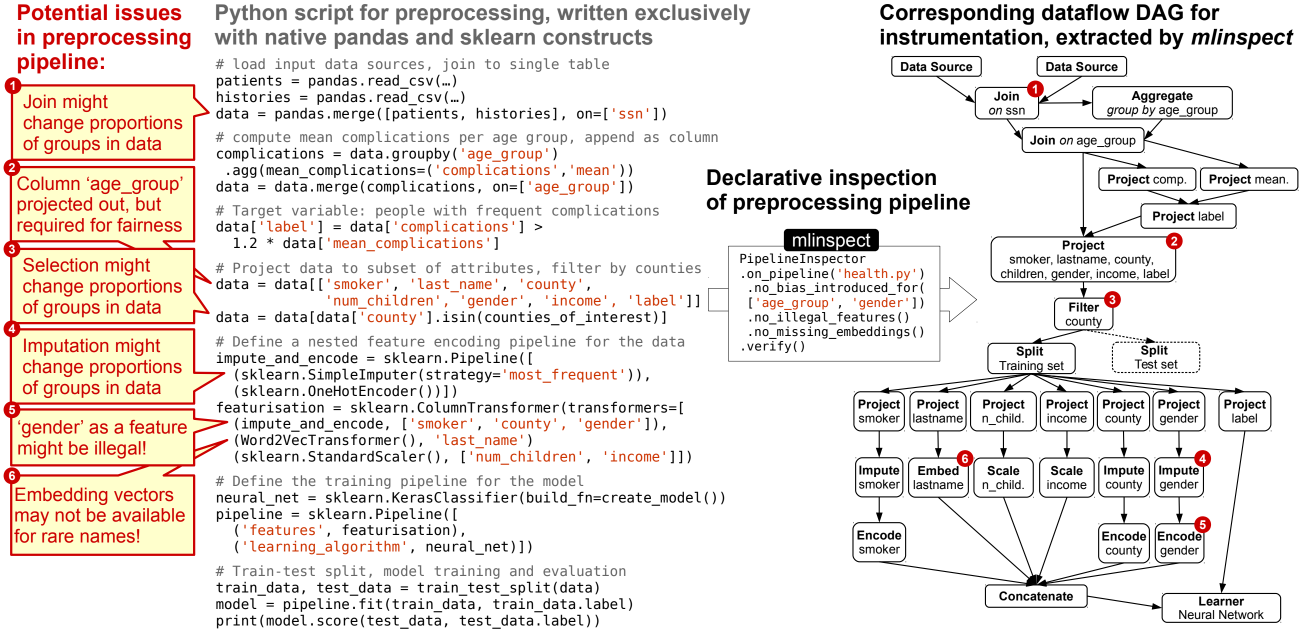
Lightweight inspection with `mlinspect`. To enable lightweight pipeline inspection, we designed and implemented `mlinspect` [10], a library that helps data scientists automatically detect data distribution issues in their ML pipelines, such as the accidental introduction of statistical bias, and provides linting for 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`, and combine relational operations and estimator/transformer pipelines. 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.

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, the library 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 [22], the identification of distortions with respect to protected group membership in the training data [35], or automated sanity checking for ML datasets [13].

Identifying data distribution bugs in our running example. Figure 2 shows a preprocessing pipeline and potential data distribution bugs for our running example from Section 1. The pipeline first reads two CSV files, which contain patient demographics and their clinical histories, respectively. Next, these 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 gender and age group 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

---

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



```
# 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', 'gender', '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', 'gender']),
  (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', 'gender'])
 .no_illegal_features()
 .no_missing_embeddings()
.verify()
```

Figure 2: ML pipeline for our running example 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 gender and age group. On the left, we highlight potential issues identified by `mlinspect`. On the right, we show the corresponding dataflow graph, extracted to instrument the code and pinpoint the issues.

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 data distribution bugs 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 common feature encoders with `ColumnTransformer` from `scikit-learn`, before training a neural network on the features. For the categorical attributes `smoker`, `county`, and `gender`, 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 may introduce statistical bias, as it may associate records with a missing value in the gender attribute with the majority gender in the dataset; ❺ depending on the legal context (i.e., if the disparate treatment doctrine is enforced), it may be forbidden to use `gender` 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, motivating the development of automatic inspection libraries such as `mlinspect`, which will hint the data scientist towards these issues.

Source code. A prototype implementation of `mlinspect`, together with a computational notebook that shows how `mlinspect` can be used to address the issues outlined in the ML pipeline in Figure 2, is available at `https://github.com/stefan-grafberger/mlinspect`.

3.2 Validating Serving Data with Data Unit Tests

Machine learning (ML) techniques are very sensitive to their input data, as the deployed models rely on strong statistical assumptions about their inputs [32], and subtle errors introduced by changes in the data distribution can be hard to detect [20]. At the same time, there is ample evidence that the volume of data available for training is often a decisive factor for a model's performance [11]. How errors in the data affect performance, and fairness of deployed machine learning models is an open and pressing research question, especially in cases where the data describing protected groups has a higher likelihood of containing errors or missing values [29].

Unit tests for data with `Deequ`. As discussed in Section 2.2, accidental errors during data integration can heavily impact the prediction quality of downstream ML models. We therefore postulate that there is a pressing need for increased automation of data validation. To respond to this need, Schelter et al. [30] presented `Deequ`, a data unit testing library. The library centers around the vision that users should be able to write 'unit-tests' for data, analogous to established testing practices in software engineering, and is built on the following principles:

1. Declarativeness: allowing data scientist to spend time on thinking about *what* their data should look like, and not about *how* to implement the quality checks. `Deequ` offers a declarative API that allows users to define checks on their data by composing a variety of available constraints.
2. Flexibility: allowing users to leverage external data and custom code for validation (e.g., call a REST service for some data and write a complex function that compares the result to some statistic computed on the data).
3. Continuous integration: explicitly supporting the incremental computation of quality metrics on growing datasets [27], and allowing users to run anomaly detection algorithms on the resulting historical time series of quality metrics.
4. Scalability: scaling seamlessly to large datasets, by translating the data metrics computations to aggregation queries, which can be efficiently executed at scale with a distributed dataflow engine such as `Apache Spark` [37].

Unit testing serving data in our running example. A prime use case of `Deequ` in ML deployments is to test new data to be sent to the model for prediction. When Ann deploys her model for real world usage, she wants to make sure that it will only consume well-formed data. She can use `Deequ` to write down her assumptions about the data as a declarative data unit test, and have this test integrated into the pipeline that feeds data to the deployed model. If any assumptions are violated, the pipeline will stop processing, the data will be quarantined, and a data engineer will be prompted to investigate the root cause of the failure.

Listing 1 shows what a data unit test may look like. We precompute certain expected statistics for the data such as the number patients to predict for, the valid age groups, and expected distributions by gender and age group. Next, we write down our assumptions about the data, similar to integrity constraints in relational databases. We declare the following checks: we assume that the size of the data corresponds to the expected number of patients, we expect social security numbers (the `ssn` attribute) to be unique, and we expect no missing values for the `lastname`, `county`, and `age_group` attributes. We furthermore assume that the values of the `smoker` attribute are Boolean, while in the `num_children` attribute comprises of integers, and we expect the `age_group` attribute to only contain valid age group values, as defined beforehand. We also expect values of the `num_children` attribute to be non-negative. Finally, we compare the distribution of age groups and gender in serving data to their expected distribution via the `histogramSatisfies` constraint. The user-defined function `notDiverged` compares the categorical distributions of these columns and returns a Boolean value.

```
// Computed in advance
val expectedNumPatients = ...
```

```
val validAgeGroups = ...
val expectedGenderDist = ...
val expectedAgeGroupDist = ...

// Assumptions about data to predict on
val validationResultForTestData = VerificationSuite ()
  .onData(expectedNumPatients)
  .addCheck()
    .hasSize(numPatients)
    .isUnique("ssn")
    .isComplete("lastname", "county", "age_group")
    .hasDataType("smoker", Boolean)
    .hasDataType("num_children", Integral)
    .isNonNegative("num_children")
    .isContainedIn("age_group", validAgeGroups)
    .histogramSatisfies("age_group", { ageGroupDist =>
      notDiverged(ageGroupDist, expectedAgeGroupDist) })
    .histogramSatisfies("gender", { genderDist =>
      notDiverged(genderDist, expectedGenderDist) })
  .run()

if (validationResultForTestData.status != Success) {
  // Abort pipeline, notify data engineers
}
```

Listing 1: Example of a data unit test.

During the execution of the test, `Deequ` identifies the statistics required for evaluating the constraints and generates queries in `SparkSQL` with custom designed aggregation functions to compute them. For performance reasons, it applies multi-query optimization to enable scan-sharing for the aggregation queries, minimizing the number of passes over the input data. Once the data statistics are computed, `Deequ` invokes the validation functions and returns the evaluation results to the user.

Source code. `Deequ` is available under an open source license at `https://github.com/awslabs/deequ`. It for example forms the basis of Amazon's recent `Model Monitor` service[2] for concept drift detection in the `SageMaker` machine learning platform.

4 Conclusions and Future Research Directions

In this paper we discussed dimensions of technical bias that can arise through the lifecycle of machine learning applications, both during model development and after deployment. We outlined several approaches to detect and mitigate such bias based on our recent work, and will now discuss promising directions for future research, where the data engineering community has the potential to make significant impact. We see the overarching goal of this line of research not in mechanically scrubbing data or algorithms of bias, but rather in equipping data scientists with tools that can help them identify technical bias, understand any trade-offs, and thoughtfully enact interventions.

Integrating technical bias detection into general software development tooling. Data science is rapidly becoming an important part of the toolbox of a "general software engineer", and so methods for detection and mitigation of technical bias need to become part of that toolbox as well. The scope of these methods must be extended beyond binary classification, and they must embrace human-in-the-loop elements by providing visualisations and allowing end-users to control experiments with low effort. To achieve practical impact, it is important to integrate these methods into common computational notebooks such as Jupyter, and into general IDE's such as PyCharm.

---

[2] `https://aws.amazon.com/blogs/aws/amazon-sagemaker-model-monitor-fully-managed-automatic-monitoring-for-your-machine-learning-models/`

Automating data quality monitoring. The arising challenge of automating the operation of deployed ML applications is gaining a lot of attention recently, especially with respect to monitoring the quality of their input data [25]. As outlined in Sections 2 and 3, data quality issues and the choice of a data cleaning technique can be a major source of technical bias. Existing approaches [2, 30] for this problem have not yet reached broad adoption, in part because they rely on substantial domain knowledge needed, for example, to define "data unit tests" and the corresponding similarity metrics, and to set thresholds for detecting data distribution shifts. Additionally, it is very challenging to test data during the earlier pipeline stages (e.g., data integration) without explicit knowledge of how an ML model will transform this data at the later stages.

We thus see a dire need for automated or semi-automated approaches to quantify and monitor data quality in ML pipelines. A promising direction is to treat historical data (for which no system failures were recorded and no negative user feedback has been received) as "positive" examples, and to explore anomaly detection-based methods to identify future data that heavily deviates from these examples. It is important to integrate a technical bias perspective into these approaches, for example, by measuring data quality separately for subsets of the data that correspond to historically disadvantaged or minority groups, since these groups tend to be more heavily hit by data quality issues [6].

Integrating technical bias detection into continuous integration systems for ML. Continuous integration is an indispensable step of modern best practices in software engineering to control the quality of deployed software, typically by automatically ensuring that software changes pass a set of unit and integration tests before deployment. There is ongoing work to adapt and reinvent continuous integration for the machine learning engineering process [24], which also exposes a lifecycle similar to the software engineering lifecycle, as discussed in Section 2. We see the need to make detection techniques for technical bias, such as automated inspections and data unit tests, first-class citizen in ML-specific continuous integration systems.

References

[1] Solon Barocas and Andrew D. Selbst. Big data's disparate impact. *California Law Review*, 104(3):671–732, 2016.

[2] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[3] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. Deep learning for missing value imputation in tables with non-numerical data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2017–2025. ACM, 2018.

[4] Amanda Bower, Sarah N. Kitchen, Laura Niss, Martin J. Strauss, Alexander Vargas, and Suresh Venkatasubramanian. Fair pipelines. *CoRR*, abs/1707.00391, 2017.

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

[6] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. Slice finder: Automated data slicing for model validation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1550–1553. IEEE, 2019.

[7] Cynthia Dwork, Christina Ilvento, and Meena Jagadeesan. Individual fairness in pipelines. In Aaron Roth, editor, *1st Symposium on Foundations of Responsible Computing, FORC 2020, June 1-3, 2020, Harvard*

University, Cambridge, MA, USA (virtual conference), volume 156 of *LIPIcs*, pages 7:1–7:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[8] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. On the (im)possibility of fairness. *CoRR*, abs/1609.07236, 2016.

[9] Batya Friedman and Helen Nissenbaum. Bias in computer systems. *ACM Trans. Inf. Syst.*, 14(3):330–347, 1996.

[10] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. Lightweight inspection of data preprocessing in native machine learning pipelines. In *CIDR*, 2021.

[11] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[12] Hoda Heidari, Michele Loi, Krishna P. Gummadi, and Andreas Krause. A moral framework for understanding fair ML through economic models of equality of opportunity. In *Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* 2019, Atlanta, GA, USA, January 29-31, 2019*, pages 181–190. ACM, 2019.

[13] Nick Hynes, D Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. *NIPS MLSys Workshop*, 2017.

[14] Joost Kappelhof. *Total Survey Error in Practice*, chapter Survey Research and the Quality of Survey Data Among Ethnic Minorities. 2017.

[15] David Lehr and Paul Ohm. Playing with the data: What legal scholars should learn about machine learning. *UC Davis Law Review*, 51(2):653–717, 2017.

[16] Zachary Lipton, Yu-Xiang Wang, and Alexander Smola. Detecting and correcting for label shift with black box predictors. In *International Conference on Machine Learning*, pages 3122–3130, 2018.

[17] Samuel Madden, Mourad Ouzzani, Nan Tang, and Michael Stonebraker. Dagger: A data (not code) debugger. *CIDR*, 2020.

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

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

[20] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1723–1726, 2017.

[21] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Record*, 47:17–28, 2018.

[22] Neoklis Polyzotis, Steven Whang, Tim Klas Kraska, and Yeounoh Chung. Slice finder: Automated data slicing for model validation. *ICDE*, 2019.

[23] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, et al. Data science through the looking glass and what we found there, 2019.

[24] Cedric Renggli, Bojan Karlas, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. Continuous integration of machine learning models: A rigorous yet practical treatment. *Proceedings of SysML 2019*, 2019.

[25] Tammo Rukat, Dustin Lange, Sebastian Schelter, and Felix Biessmann. Towards automated ml model monitoring: Measure, improve and quantify data quality. 2020.

[26] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, Gyuri Szarvas, Manasi Vartak, Samuel Madden, Hui Miao, Amol Deshpande, et al. On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.

[27] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. Differential data quality verification on partitioned data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1940–1945. IEEE, 2019.

[28] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. Fairprep: Promoting data to a first-class citizen in studies on fairness-enhancing interventions. *EDBT*, 2019.

[29] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. Fairprep: Promoting data to a first-class citizen in studies on fairness-enhancing interventions. In Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang, editors, *EDBT*, pages 395–398. OpenProceedings.org, 2020.

[30] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, 11(12):1781–1794, 2018.

[31] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015.

[32] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015.

[33] Julia Stoyanovich, Bill Howe, and H. V. Jagadish. Responsible data management. *Proc. VLDB Endow.*, 13(12):3474–3488, 2020.

[34] Manasi Vartak and Samuel Madden. Modeldb: Opportunities and challenges in managing machine learning models. *IEEE Data Eng.*, 41:16–25, 2018.

[35] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. Fairness-aware instrumentation of preprocessing pipelines for machine learning. *HILDA workshop at SIGMOD*, 2020.

[36] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

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

SPECIAL ISSUE PAPER

Data distribution debugging in machine learning pipelines

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

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

Abstract

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

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

1 Introduction

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

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

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

---

Sebastian Schelter
s.schelter@uva.nl

Stefan Grafberger
s.grafberger@uva.nl

Paul Groth
p.t.groth@uva.nl

Julia Stoyanovich
stoyanovich@nyu.edu

[1] University of Amsterdam, Amsterdam, Netherlands

[2] New York University, New York, USA

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

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

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

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

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

In this paper, we make the following contributions:

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

2 Data distribution bugs by example

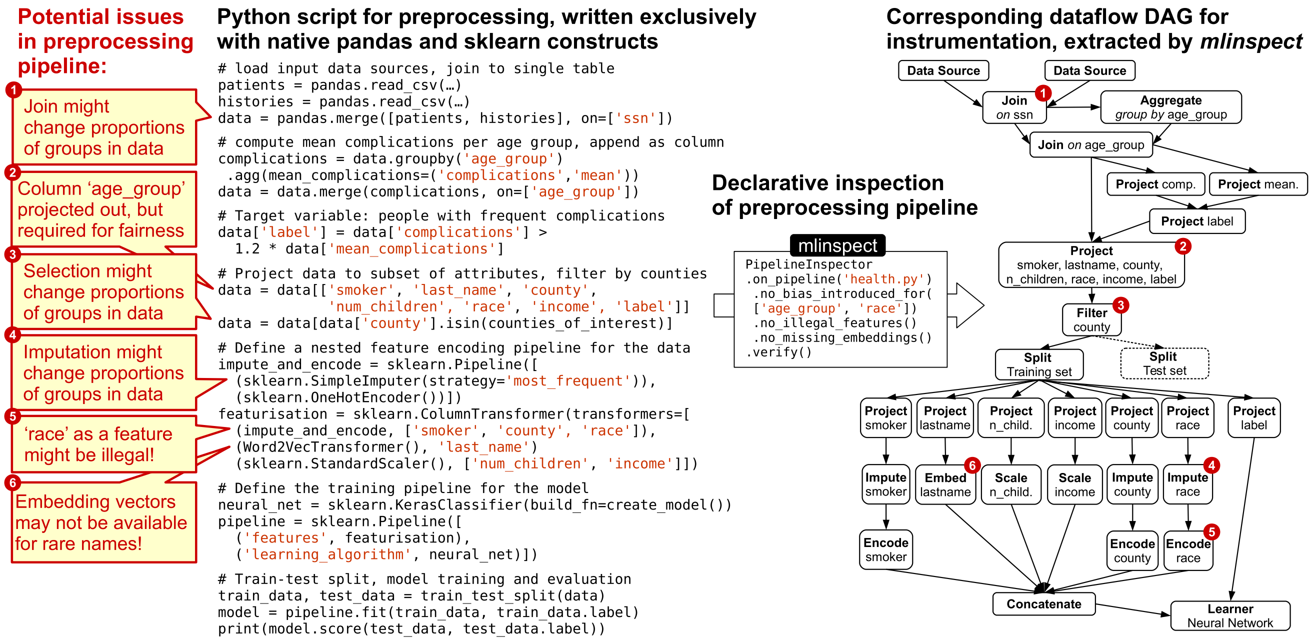
We illustrate the need for assisting data scientists with the inspection of their preprocessing pipelines with an example from the medical domain, shown in Fig. 1. Consider a data scientist who implements a Python pipeline that takes demographic and clinical history data as input, and trains a classifier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

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

---

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

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



```
# 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()
```

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

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

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

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

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

3 Design of mlinspect

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

---

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

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

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

3.1 Overview

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

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

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

3.2 Instrumentation and annotation propagation

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

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

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

---

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

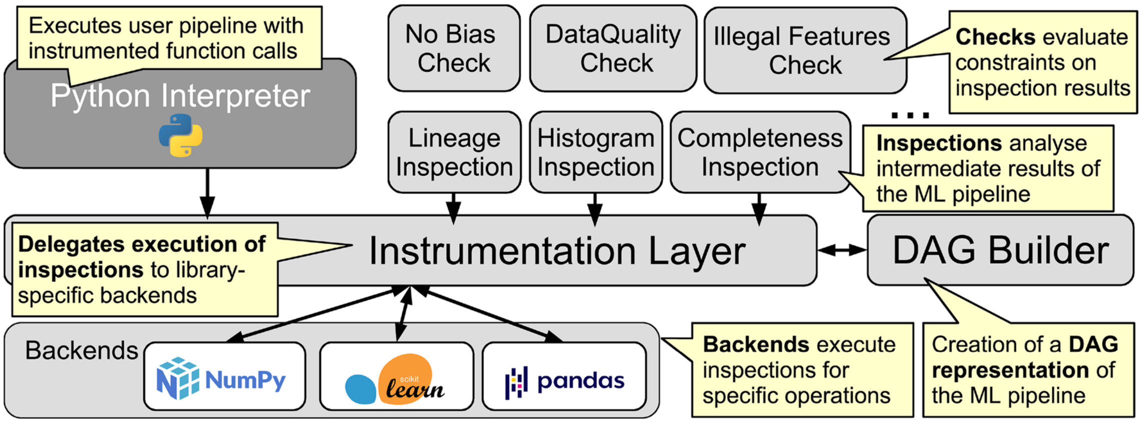


Fig. 2 Architecture of `mlinspect`. We apply checks and inspections to an instrumented ML pipeline written by the user. The instrumentation layer delegates the execution of the inspections to library-specific backends and creates a DAG representation of the pipeline

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Automatic inspections and checks

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

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

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

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

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

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

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

3.4 Algebraic definition of the `mlinspect` dataflow graph

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

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

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

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

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

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

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

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

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

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

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

4 Implementation

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

4.1 Overview

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

4.2 Inspections

Some checks only require the extracted DAG for analysis. An example for this is the `NoIllegalFeatures` check, which inspects the names of projected attributes used as features to ensure that no illegal features, such as gender or race, are used. Other checks only require simple inspections that investigate an operator in isolation. An example is the `NoMissingEmbeddings` check, which simply counts the null values in the outputs of embedding operators. Another example are inspections for data unit testing. Data unit tests typically evaluate constraints based on aggregate statistics of the data such as the completeness (ratio of non-NULL values) of a column or the number of distinct values in a column. Often, these statistics only require a single pass over the data and can therefore be pipelined with the actual execution of an operator. The `Completeness` and `NumDistinctValues` inspections compute these statistics by iterating over the values of a given column and maintaining the counts for NULL/non-NULL values (for completeness) or a hashmap containing the number of occurrences per distinct value.

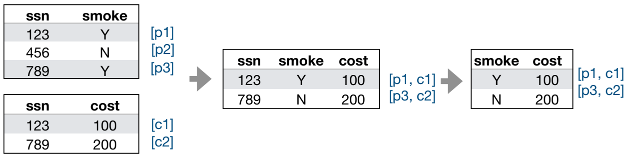


Fig. 3 Lineage tracking by propagating identifier annotations through operators

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

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

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

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

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

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

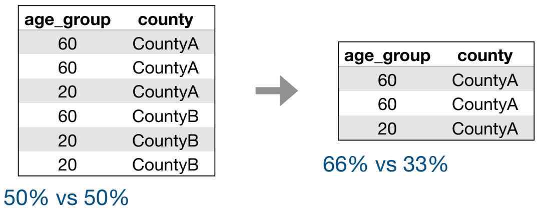
Change detection for proportions of protected groups In our running example (Fig. 1 in Sect. 2), we briefly discussed an inspection to discover the introduction of accidental changes in the proportions of protected groups. This refers to the issues ①, ②, ③ and ④ from the example and requires the histogram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

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

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

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

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



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

50% vs 50%

age_group	county
60	CountyA
60	CountyA
20	CountyA

66% vs 33%

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

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

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

4.3 Execution of inspections, checks, and DAG extraction

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

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

4.3.1 Preparation

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

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

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

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

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

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

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

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

---

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

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

4.3.2 Execution of the instrumented program

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

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

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

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

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

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

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

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

---

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

Table 2 Overview of the internal operator types

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

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

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

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

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

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

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

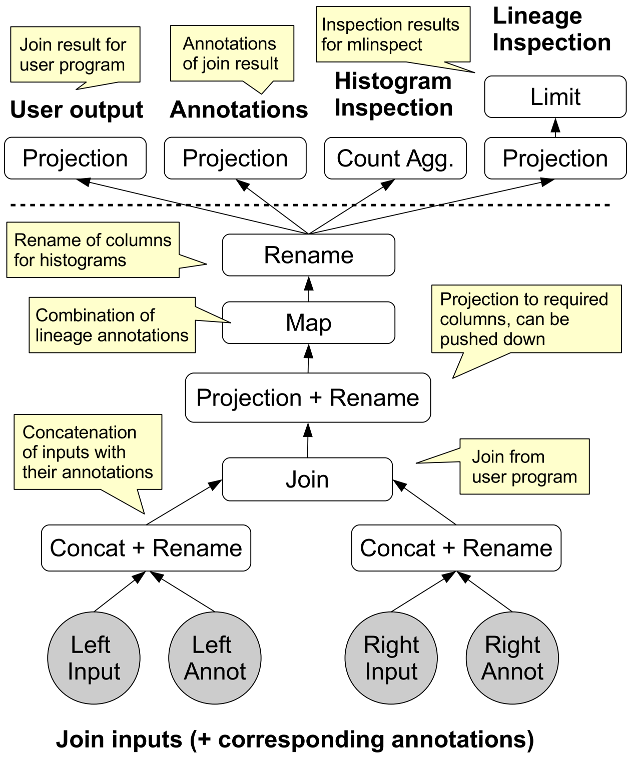


Fig. 5 Example for optimizable inspections: we generate and execute a query plan to apply the histogram and the lineage annotations to a join on two dataframes

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

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

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

4.3.3 Extraction of the dataflow graph and evaluation of checks

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

4.4 Implementation of our example

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

---

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

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

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

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

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

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

5 Experimental evaluation

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

5.1 Runtime overhead

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

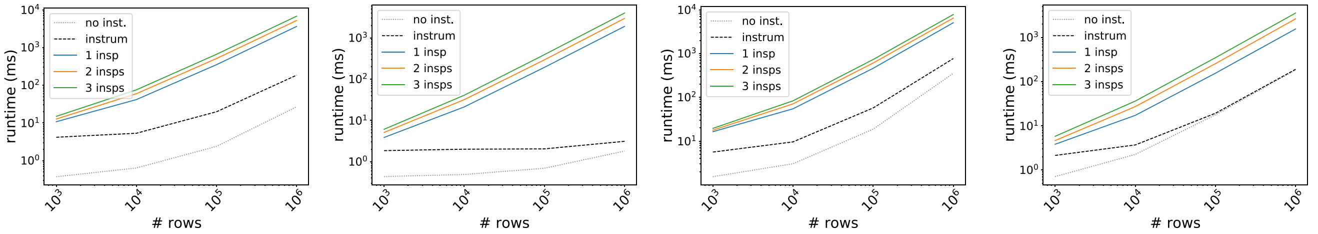
5.1.1 Overhead of python-based operator instrumentation

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

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

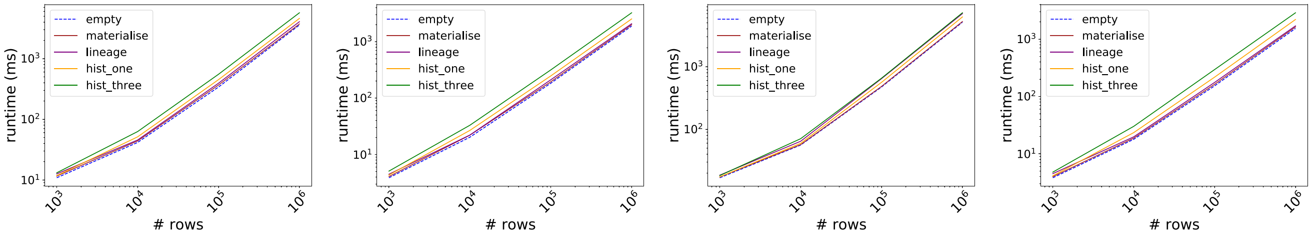
5.1.2 Python-based inspection overhead

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



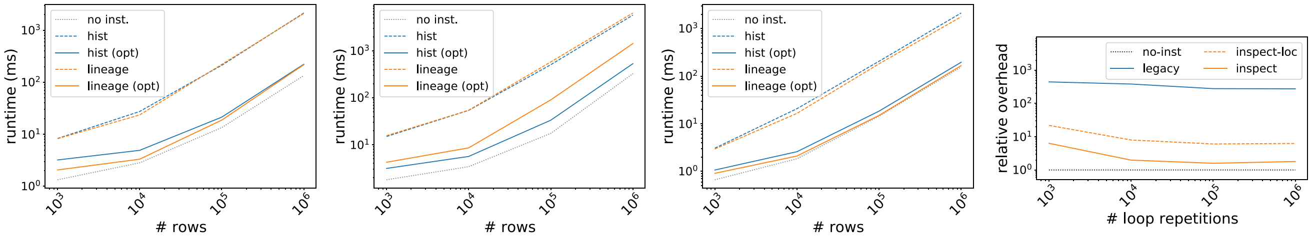
(a) Selection. **(b)** Projection. **(c)** Join. **(d)** One-Hot-Encoder.

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



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

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



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

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

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

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

5.1.3 Optimized execution of inspections

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

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

5.1.4 AST instrumentation overhead

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

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

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

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

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

5.2 Exploratory interview study with experts

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

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

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

---

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

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

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

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

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

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

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

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

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

5.3 Qualitative comparison against experiment tracking and workflow provenance tools

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

5.3.1 MLFlow

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

---

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

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

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

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

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

5.3.2 noWorkflow

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

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

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

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

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

---

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

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

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

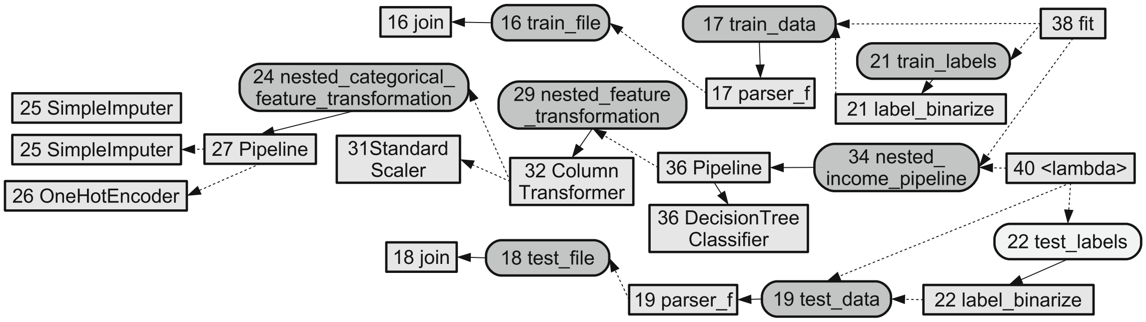


Fig. 9 Simplified illustration of the call graph for our example pipeline produced by noWorkflow. Unfortunately, it is difficult to understand the dataflow of ML pipelines using pandas and scikit-learn. The graph directly reflects each function call in the user code and does not provide an abstract representation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

6 Related work

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

6.1 Provenance for relational workloads

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

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

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

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

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

6.2 Workflow provenance

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

6.3 Experiment tracking and model management

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

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

6.4 Debugging for ML pipelines and data

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

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

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

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

6.5 Fairness-specific analysis of ML pipelines and predictions

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

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

7 Conclusion and future work

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

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

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

References

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

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