Data Debugging with Shapley Importance over End-to-End Machine Learning Pipelines

Data Importance and Valuation Meet Feature Extractors and Data Provenance

Bojan Karlaš[1], David Dao[1], Matteo Interlandi[2], Bo Li[3], Sebastian Schelter[4], Wentao Wu[2], and Ce Zhang[1]

[1]ETH Zurich, Switzerland
[2]Microsoft, USA
[3]UIUC, USA
[4]University of Amsterdam, Netherlands
{bojan.karlas, david.dao, ce.zhang}@inf.ethz.ch, {matteo.interlandi, wentao.wu}@microsoft.com, lbo@uiuc.edu, s.schelter@uva.nl

Abstract

Developing modern machine learning (ML) applications is *data-centric*, of which one fundamental challenge is to understand the influence of data quality to ML training — *"Which training examples are 'guilty' in making the trained ML model predictions inaccurate or unfair?"* Modeling data influence for ML training has attracted intensive interest over the last decade, and one popular framework is to compute the *Shapley value* of each training example with respect to utilities such as validation accuracy and fairness of the trained ML model. Unfortunately, despite recent intensive interests and research, existing methods only consider a single ML model "in isolation" and do not consider an end-to-end ML pipeline that consists of *data transformations*, *feature extractors*, and *ML training*.

We present `Ease.ML/DataScope`, the first system that efficiently computes Shapley values of training examples over an *end-to-end* ML pipeline, and illustrate its applications in data debugging for ML training. To this end, we first develop a novel algorithmic framework that computes Shapley value over a specific family of ML pipelines that we call *canonical pipelines*: *a positive relational algebra query followed by a K-nearest-neighbor (KNN) classifier*. We show that, for many subfamilies of canonical pipelines, computing Shapley value is in PTIME, contrasting the exponential complexity of computing Shapley value in general. We then put this to practice — given an `sklearn` pipeline, we approximate it with a canonical pipeline to use as a proxy. We conduct extensive experiments illustrating different use cases and utilities. Our results show that `DataScope` is up to four orders of magnitude faster over state-of-the-art Monte Carlo-based methods, while being comparably, and often even more, effective in data debugging.

Code Availability: github.com/easeml/datascope; github.com/schelterlabs/arguseyes/tree/datascope

1 Introduction

Last decade has witnessed the rapid advancement of machine learning (ML), along which comes the advancement of *machine learning systems* [56]. Thanks to these advancements, training a machine learning model has never been easier today for practitioners — distributed learning over hundreds of devices [44, 43, 19, 64, 31], tuning hyper-parameters and selecting the best model [11, 74, 18], all of which become much more systematic and less mysterious. Moreover, all major cloud service providers now support AutoML and other model training and serving services.

```
1  # Data loading
2  train_data = pd.read_csv("...")
3  test_data = pd.read_csv("...")
4  side_data = pd.read_csv("...")
5  # Data integration
6  train_data = train_data.join(side_data, on="item_id")
7  test_data = test_data.join(side_data, on="item_id")
8  # Declaratively defined (nested) feature encoding pipeline
9  pipeline = Pipeline([
10   ('features', ColumnTransformer([
11     (StandardScaler(), ["num_att1", "num_att2"]),
12     (Pipeline([SimpleImputer(), OneHotEncoder()])),
13       ["cat1", "cat2"]),
14     (HashingVectorizer(n_features=100), "text_att1")])),
15   # ML model for learning
16   ('learner', SVC()])
17 # Train and evaluate model
18 pipeline.fit(train_data, train_data.label)
19 print(pipeline.score(test_data, test_data.label))
20
21 # ********************* DATASCOPE *********************
22 # Run data importance computation over pipeline
23 (train_data_with_imp, side_data_with_imp) = \
24   DataScope.debug(pipeline, sklearn.metrics.accuracy)
25 # ******************************************************
```

Listing 1: A simplified illustration of the core functionality enabled by `DataScope`— given an end-to-end ML pipeline (Line 1-19), and a utility (e.g., `sklearn.metrics.accuracy`), `DataScope` computes the Shapley value of each training example as its *importance* with respect to the given utility.

Data-centric Challenges and Opportunities. Despite these great advancements, a new collection of challenges start to emerge in building better machine learning applications. One observation getting great attention recently is that *the quality of a model is often a reflection of the quality of the underlying training data*. As a result, often the most practical and efficient way of improving ML model quality is to improve data quality. As a result, recently, researchers have studied how to conduct data cleaning [38, 34], data debugging [35, 36, 20, 29, 30, 28], and data acquisition [57], specifically for the purpose of improving an ML model.

Data Debugging via Data Importance. In this paper, we focus on the fundamental problem of reasoning about the *importance of training examples with respect to some utility functions (e.g., validation accuracy and fairness) of the trained ML model.* There have been intensive recent interests to develop methods for reasoning about data importance. These efforts can be categorized into two different views. The *Leave-One-Out (LOO)* view of this problem tries to calculate, given a training set $\mathcal{D}$, the importance of a data example $x \in \mathcal{D}$ modeled as the *utility* decrease after removing this data example: $U(\mathcal{D}) - U(\mathcal{D} \backslash x)$. To scale-up this process over a large dataset, researchers have been developing approximation methods such as *influence function* for a diverse set of ML models [35]. On the other hand, the *Expected-Improvement (ExpI)* view of this problem tries to calculate such a utility decrease over *all possible subsets of* $\mathcal{D}$. Intuitively, this line of work models data importance as an "expectation" over all possible subsets/sub-sequences of $\mathcal{D}$, instead of trying to reason about it solely on a single training set. One particularly popular approach is to use Shapley value [20, 29, 30], a concept in game theory that has been applied to data importance and data valuation [28].

Shapley-based Data Importance. In this paper, we do not champion one view over the other (i.e., LOO vs. ExpI). We scope ourselves and only focus on Shapley-based methods since previous work has shown applications that can only use Shapley-based methods because of the favorable properties enforced by the Shapley value. Furthermore, taking expectations can sometimes provide a more reliable importance measure [28] than simply relying on a single dataset. Nevertheless, we believe that it is important for future ML systems to support both and we hope that this paper can inspire future research in data importance for both the LOO and ExpI views.

One key challenge of Shapley-based data importance is its computational complexity — in the worst case, it needs to enumerate *exponentially* many subsets. There have been different ways to *approximate* this computation, either with MCMC [20] and group testing [29] or proxy models such as K-nearest neighbors (KNN) [30]. One surprising result is that Shapley-based data importance can be calculated efficiently (in *polynomial* time) for KNN classifiers [30], and using this as a proxy for other classifiers performs well over a diverse range of tasks [28].

Data Importance over Pipelines. Existing methods for computing Shapley values [20, 29, 30, 28] are designed to directly operate on a single numerical input dataset for an ML model, typically in matrix form. However, in real-world ML applications, this data is typically generated on the fly from multiple data sources with an ML pipeline. Such pipelines often take multiple datasets as input, and transform them into a single numerical input dataset with relational operations (such as joins, filters, and projections) and common feature encoding techniques, often based on nested estimator/transformer pipelines, which are integrated into popular ML libraries such as scikit-learn [51], SparkML [49] or Google TFX [10]. It is an open problem how to apply Shapley-value computation in such a setup.

Listing 1 shows a toy example of such an end-to-end ML pipeline, which includes relational operations from pandas for data preparation (lines 3-9), a nested estimator/transformer pipeline for encoding numerical, categorical, and textual attributes as features (lines 12-16), and an ML model from scikit-learn (line 18). The code loads the data, splits it temporally into training and test datasets, 'fits' the pipeline to train the model, and evaluates the predictive quality on the test dataset. This leads us to the key question we pose in this work:

> *Can we efficiently compute Shapley-based data importance over such an end-to-end ML pipeline with <u>both</u> data processing and ML training?*

Technical Contributions. We present `Ease.ML/DataScope`, the first system that efficiently computes and approximates Shapley value over end-to-end ML pipelines. `DataScope` takes as input an ML pipeline (e.g., a `sklearn` pipeline) and a given utility function, and outputs the importance, measured as the Shapley value, of each input tuple of the ML pipeline. Listing 1 (lines 21-25) gives a simplified illustration of this core functionality provided by `DataScope`. A user points `DataScope` to the pipeline code, and `DataScope` executes the pipeline, extracts the input data, which is annotated with the corresponding Shapley value per input tuple. The user could then, for example, retrieve and inspect the least useful input tuples. We present several use cases of how these importance values can be used, including label denoising and fairness debugging, in section 6. We made the following contributions when developing `DataScope`.

Our first technical contribution is to jointly analyze Shapley-based data importance together with a *feature extraction pipeline*. To our best knowledge, this is the first time that these two concepts are analyzed together. We first show that we can develop a *PTIME algorithm* given a counting oracle relying on data provenance. We then show that, for a collection of "canonical pipelines", which covers many real-world pipelines [55] (see Table 2 in section 6 for examples), this counting oracle itself can be implemented in polynomial time. This provides an efficient algorithm for computing Shapley-based data importance over these "canonical pipelines".

Our second technical contribution is to understand and further adapt our technique in the context of real-world ML pipelines. We identify scenarios from the aforementioned 500K ML pipelines where our techniques cannot be directly applied to have PTIME algorithms. We introduce a set of simple yet effective approximations and optimizations to further improve the performance on these scenarios.

Our third technical contribution is an extensive empirical study of `DataScope`. We show that for a diverse range of ML pipelines, `DataScope` provides effective approximations to support a range of applications to improve the accuracy and fairness of an ML pipeline. Compared with strong state-of-the-art methods based on Monte Carlo sampling, `DataScope` can be up to four orders of magnitude faster while being comparably, and often even more, effective in data debugging.

2 Preliminaries

In this section we describe several concepts from existing research that we use as basis for our contributions. Specifically, (1) we present the definition of machine learning pipelines and their semantics, and (2) we describe decision diagrams as a tool for compact representation of Boolean functions.

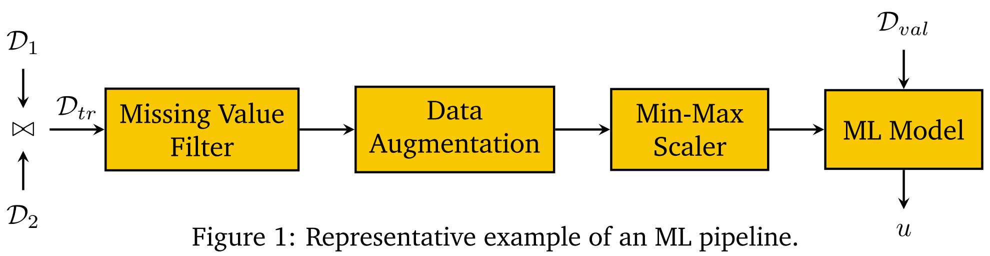

Figure 1: Representative example of an ML pipeline.

2.1 End-to-end ML Pipelines

An end-to-end ML application consists of two components: (1) a feature extraction pipeline, and (2) a downstream ML model. To conduct a joint analysis over one such end-to-end application, we leave the precise definite to subsection 3.2. One important component in our analysis relies on the *provenance* of the feature extraction pipeline, which we will discuss as follows.

Provenance Tracking. Input examples (tuples) in $\mathcal{D}_{tr}$ are transformed by a feature processing pipeline before being turned into a processed training dataset $\mathcal{D}_{tr}^f := f(\mathcal{D}_{tr})$, which is directly used to train the model. To enable ourselves to compute the importance of examples in $\mathcal{D}_{tr}$, it is useful to relate the presence of tuples in $\mathcal{D}_{tr}^f$ in the training dataset with respect to the presence of tuples in $\mathcal{D}_{tr}$. In other words, we need to know the *provenance* of training tuples. In this paper, we rely on the well-established theory of provenance semirings [24] to describe such provenance.

We associate a variable $a_t \in A$ with every tuple t in the training dataset $\mathcal{D}_{tr}$. We define *value assignments* $v : A \to \mathbb{B}$ to describe whether a given tuple t appears in $\mathcal{D}_{tr}$ — by setting $v(a_t) = 0$, we "exclude" t from $\mathcal{D}_{tr}$ and by setting $v(a_t) = 1$, we "include" t in $\mathcal{D}_{tr}$. Let $\mathcal{V}_A$ be the set of all possible such value assignments ($|\mathcal{V}_A| = 2^{|A|}$). We use

$$\mathcal{D}_{tr}[v] = \{t \in \mathcal{D}_{tr} | v(a_t) \neq 0\}$$

to denote a subset of training examples, only containing tuples t whose corresponding variable in a_t is set to 1 according to v.

To describe the association between $\mathcal{D}_{tr}$ and its transformed version $\mathcal{D}_{tr}^f$, we annotate each potential tuple in $\mathcal{D}_{tr}^f$ with an attribute $p : \mathbb{D} \to \mathbb{B}[A]$ containing its *provenance polynomial* [24] which is a logical formula with variables in A and binary coefficients (e.g. $a_1 + a_2 \cdot a_3$) — $p(t)$ is true only if tuple t appears in $\mathcal{D}_{tr}^f$. For such polynomials, an *addition* corresponds to a *union* operator in the ML pipeline, and a *multiplication* corresponds to a *join* operator in the pipeline. Figure 3 illustrates some examples of the association between a_t and $p(t)$.

Given a value assignment $v \in \mathcal{V}_A$, we can define an evaluation function $\text{eval}_v\ \phi$ that returns the *evaluation* of a provenance polynomial ϕ under the assignment v. Given a value assignment v, we can obtain the corresponding *transformed dataset* by evaluating all provenance polynomials of its tuples, as such:

$$\mathcal{D}_{tr}^f[v] := \{t \in \mathcal{D}_{tr}^f \mid \text{eval}_v(p(t)) \neq 0\} \tag{1}$$

Intuitively, $\mathcal{D}_{tr}^f[v]$ corresponds to the result of applying the feature transformation f over a subset of training examples that only contains tuples t whose corresponding variable a_t is set to 1.

Using this approach, given a feature processing pipeline f and a value assignment v, we can obtain the transformed training set $\mathcal{D}_{tr}^f[v] = f(\mathcal{D}_{tr}[v])$.

2.2 Additive Decision Diagrams (ADD's)

Knowledge Compilation. Our approach of computing the Shapley value will rely upon being able to construct functions over Boolean inputs $\phi : \mathcal{V}_A \to \mathcal{E}$, where $\mathcal{E}$ is some finite *value set*. We require an elementary algebra with $+$, $-$, $\cdot$ and $/$ operations to be defined for this value set. Furthermore, we require this value set

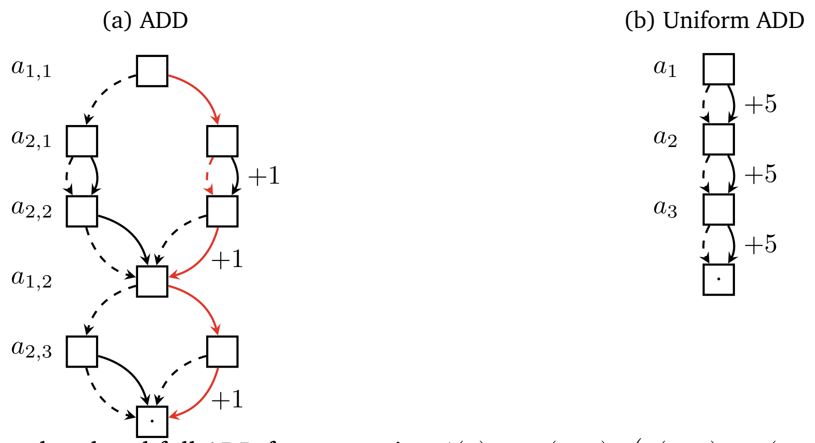

Figure 2: (a) An ordered and full ADD for computing $\phi(v) := v(a_{1,1}) \cdot \big(v(a_{2,1}) + v(a_{2,2})\big) + v(a_{1,2}) \cdot v(a_{2,3})$. (b) A uniform ADD for computing $\phi(v) := 5 \cdot (v(a_1) + v(a_2) + v(a_3))$.

to contain a *zero element* 0, as well as an *invalid element* ∞ representing an undefined result (e.g. a result that is out of bounds). We then need to count the number of value assignments $v \in \mathcal{V}_A$ such that $\phi(v) = e$, for some specific value $e \in \mathcal{E}$. This is referred to as the *model counting* problem, which is #P complete for arbitrary logical formulas [68, 5]. For example, if $A = \{a_1, a_2, a_3\}$, we can define $\mathcal{E} = \{0, 1, 2, 3, \infty\}$ to be a value set and a function $\phi(v) := v(a_1) + v(a_2) + v(a_3)$ corresponding to the number of variables in A that are set to 1 under some value assignment $v \in \mathcal{V}_A$.

Knowledge compilation [14] has been developed as a well-known approach to tackle this model counting problem. It was also successfully applied to various problems in data management [27]. One key result from this line of work is that, if we can construct certain polynomial-size data structures to represent our logical formula, then we can perform model counting in polynomial time. Among the most notable of such data structures are *decision diagrams*, specifically binary decision diagrams [41, 12] and their various derivatives [6, 62, 40]. For our purpose in this paper, we use the *additive decision diagrams* (ADD), as detailed below.

Additive Decision Diagrams (ADD). We define a simplified version of the *affine algebraic decision diagrams* [62]. An ADD is a directed acyclic graph defined over a set of nodes $\mathcal{N}$ and a special *sink node* denoted $\boxdot$. Each node $n \in \mathcal{N}$ is associated with a variable $a(n) \in A$. Each node has two outgoing edges, $c_L(n)$ and $c_H(n)$, that point to its *low* and *high* child nodes, respectively. For some value assignment v, the low/high edge corresponds to $v(a) = 0/v(a) = 1$. Furthermore, each low/high edge is associated with an increment w_L/w_H that maps edges to elements of $\mathcal{E}$.

Note that each node $n \in \mathcal{N}$ represents the root of a subgraph and defines a Boolean function. Given some value assignment $v \in \mathcal{V}_A$ we can evaluate this function by constructing a path starting from n and at each step moving towards the low or high child depending on whether the corresponding variable is assigned a 0 or 1. The value of the function is the result of adding all the edge increments together. Figure 2a presents an example ADD with one path highlighted in red. Formally, we can define the evaluation of the function defined by the node n as follows:

$$\text{eval}_v(n) := \begin{cases} 0, & \text{if } n = \boxdot, \\ w_L(n) + \text{eval}_v(c_L(n)) & \text{if } v(x(n)) = 0, \\ w_H(n) + \text{eval}_v(c_H(n)) & \text{if } v(x(n)) = 1. \end{cases} \tag{2}$$

In our work we focus specifically on ADD's that are *full* and *ordered*. A diagram is full if every path from root to sink encounters every variable in A exactly once. On top of that, an ADD is ordered when on each path from root to sink variables always appear in the same order. For this purpose, we define $\pi : A \to \{1, ..., |A|\}$ to be a permutation of variables that assigns each variable $a \in A$ an index.

Model Counting. We define a model counting operator

$$\operatorname{count}_e(n) := \left|\left\{v \in \mathcal{V}_{A[\leq \pi(a(n))]} \mid \operatorname{eval}_v(n) = e\right\}\right|, \tag{3}$$

where $A[\leq \pi(a(n))]$ is the subset of variables in A that include $x(n)$ and all variables that come before it in the permutation π. For an ordered and full ADD, $\operatorname{count}_e(n)$ satisfies the following recursion:

$$\operatorname{count}_e(n) := \begin{cases} 1, & \text{if } e = 0 \text{ and } n = \boxdot, \\ 0, & \text{if } e = \infty \text{ or } n = \boxdot, \\ \operatorname{count}_{e-w_L(n)}(c_L(n)) + \operatorname{count}_{e-w_H(n)}(c_H(n)), & \text{otherwise.} \end{cases} \tag{4}$$

The above recursion can be implemented as a dynamic program with computational complexity $O(|\mathcal{N}| \cdot |\mathcal{E}|)$.

In special cases when the ADD is structured as a chain with one node per variable, all low increments equal to zero and all high increments equal to some constant $E \in \mathcal{E}$, we can perform model counting in constant time. We call this a *uniform* ADD, and Figure 2b presents an example. The count_e operator for a uniform ADD can be defined as

$$\operatorname{count}_e(n) := \begin{cases} \binom{\pi(a(n))}{e/E}, & \text{if } e \bmod E = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Intuitively, if we observe the uniform ADD shown in Figure 2b, we see that the result of an evaluation must be a multiple of 5. For example, to evaluate to 10, the evaluation path must pass a *high* edge exactly twice. Therefore, in a 3-node ADD with root node n_R, the result of $\operatorname{count}_{10}(n_R)$ will be exactly $\binom{3}{2}$.

Special Operations on ADD's. Given an ADD with node set $\mathcal{N}$, we define two operations that will become useful later on when constructing diagrams for our specific scenario:

1. *Variable restriction*, denoted as $\mathcal{N}[a_i \leftarrow V]$, which restricts the domain of variables A by forcing the variable a_i to be assigned the value V. This operation removes every node $n \in \mathcal{N}$ where $a(n) = a_i$ and rewires all incoming edges to point to the node's high or low child depending on whether $V = 1$ or $V = 0$.

2. *Diagram summation*, denoted as $\mathcal{N}_1 + \mathcal{N}_2$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are two ADD's over the same (ordered) set of variables A. It starts from the respective root nodes n_1 and n_2 and produces a new node $n := n_1 + n_2$. We then apply the same operation to child nodes. Therefore, $c_L(n_1+n_2) := c_L(n_1)+c_L(n_2)$ and $c_H(n_1+n_2) := c_H(n_1)+c_H(n_2)$. Also, for the increments, we can define $w_L(n_1+n_2) := w_L(n_1)+w_L(n_2)$ and $w_H(n_1+n_2) := w_H(n_1) + w_H(n_2)$.

3 Data Importance over ML Pipelines

We first recap the problem of computing data importance for ML pipelines in subsection 3.1, formalise the problem in subsection 3.2, and outline core technical efficiency and scalability issues afterwards. We will describe the `DataScope` approach in section 4 and our theoretical framework in section 5.

3.1 Data Importance for ML Pipelines

In real-world ML, one often encounters data-related problems in the input training set (e.g., wrong labels, outliers, biased samples) that lead to sub-optimal quality of the user's model. As illustrated in previous work [35, 36, 20, 29, 30, 28], many data debugging and understanding problems hinge on the following fundamental question:

Which data examples in the training set are most important for the model utility ?

A common approach is to model this problem as computing the *Shapley value* of each data example as a measure of its importance to a model, which has been applied to a wide range use cases [20, 29, 30, 28]. However, this line of work focused solely on ML model training but ignored the *data pre-processing pipeline* prior to model training, which includes steps such as feature extraction, data augmentation, etc. This significantly limits its applications to real-world scenarios, most of which consist of a non-trivial data processing pipeline [55]. In this paper, we take the first step in applying Shapley values to debug end-to-end ML pipelines.

3.2 Formal Problem Definition

We first formally define the core technical problem.

ML Pipelines. Let $\mathcal{D}_e$ be an input training set for a machine learning task, potentially accompanied by additional relational side datasets $\mathcal{D}_{s_1}, \ldots, \mathcal{D}_{s_k}$. We assume the data to be in a *star* database schema, where each tuple from a side dataset $\mathcal{D}_{s_i}$ (the "dimension" tables) can be joined with multiple tuples from $\mathcal{D}_e$ (the "fact" table). Let f be a feature extraction pipeline that transforms the relational inputs $\mathcal{D}_{tr} = \{\mathcal{D}_e, \mathcal{D}_{s_1}, \ldots, \mathcal{D}_{s_k}\}$ into a set of training tuples $\{t_i = (x_i, y_i)\}_{i \in [m]}$ made up of feature and label pairs that the ML training algorithm $\mathcal{A}$ takes as input. Note that $\mathcal{D}_e$ represents `train_data` in our toy example in Listing 1, $\mathcal{D}_s$ represents `side_data`, while f refers to the data preparation operations from lines 6-14, and the model $\mathcal{A}$ corresponds to the support vector machine `SVC` from line 16.

After feature extraction and training, we obtain an ML model:

$$\mathcal{A} \circ f(\mathcal{D}_{tr}).$$

We can measure the *quality* of this model in various ways, e.g., via validation accuracy and a fairness metric. Let $\mathcal{D}_v$ be a given set of relational validation data with the same schema as $\mathcal{D}_e$. Applying f to $\mathcal{D}_{val} = \{\mathcal{D}_v, \mathcal{D}_{s_1}, \ldots, \mathcal{D}_{s_k}\}$ produces a set of validation tuples $\{t_i = (\tilde{x}_i, \tilde{y}_i)\}_{i \in [p]}$ made up of feature and label pairs, on which we can derive predictions with our trained model $\mathcal{A} \circ f(\mathcal{D}_{tr})$. Based on this, we define a utility function u, which measures the performance of the predictions:

$$u(\mathcal{A} \circ f(\mathcal{D}_{tr}), f(\mathcal{D}_{val})) \mapsto [0, 1].$$

For readability, we use the following notation in cases where the model $\mathcal{A}$ and pipeline f are clear from context:

$$u(\mathcal{D}_{tr}, \mathcal{D}_{val}) := u(\mathcal{A} \circ f(\mathcal{D}_{tr}), f(\mathcal{D}_{val})) \tag{6}$$

Additive Utilities. In this paper, we focus on *additive utilities* that cover the most important set of utility functions in practice (e.g., validation loss, validation accuracy, various fairness metrics, etc.). A utility function u is *additive* if there exists a *tuple-wise* utility u_T such that u can be rewritten as

$$u(\mathcal{D}_{tr}, \mathcal{D}_{val}) = w \cdot \sum_{t_{val} \in f(\mathcal{D}_{val})} u_T\bigg(\Big(\mathcal{A} \circ f(\mathcal{D}_{tr})\Big)(t_{val}), t_{val}\bigg). \tag{7}$$

Here, w is a scaling factor only relying on $\mathcal{D}_{val}$. The tuple-wise utility $u_T : (y_{pred}, t_{val}) \mapsto [0, 1]$ takes a validation tuple $t_{val} \in \mathcal{D}_{val}$ as well as a class label $y_{pred} \in \mathcal{Y}$ predicted by the model for t_{val}. It is easy to see that popular utilities such as validation accuracy are all additive, e.g., the accuracy utility is simply defined by plugging $u_T(y_{pred}, (x_{val}, y_{val})) := \mathbb{1}\{y_{pred} = y_{val}\}$ into Equation 7.

Example: False Negative Rate as an Additive Utility. Apart from accuracy which represents a trivial example of an additive utility, we can show how some more complex utilities happen to be additive and can therefore be decomposed according to Equation 7. As an example, we use *false negative rate (FNR)* which can be defined as such:

$$u(\mathcal{D}_{tr}, \mathcal{D}_{val}) := \frac{\sum_{t_{val} \in f(\mathcal{D}_{val})} \mathbb{1}\{(\mathcal{A} \circ f(\mathcal{D}_{tr}))(t_{val}) = 0\} \mathbb{1}\{y(t_{val}) = 1\}}{|\{t_{val} \in \mathcal{D}_{val} \ : \ y(t_{val}) = 1\}|}. \tag{8}$$

In the above expression we can see that the denominator only depends on $\mathcal{D}_{val}$ which means it can be interpreted as the scaling factor w. We can easily see that the expression in the numerator neatly fits the structure of Equation 7 as long as we we define u_T as $u_T(y_{pred}, (x_{val}, y_{val})) := \mathbb{1}\{y_{pred} = 0\}\mathbb{1}\{y_{val} = 1\}$. Similarly, we are able to easily represent various other utilities, including: false positive rate, true positive rate (i.e. recall), true negative rate (i.e. specificity), etc. We describe an additional example in subsection 4.3.

Shapley Value. The Shapley value, denoting the importance of an input tuple t_i for the ML pipeline, is defined as

$$\varphi_i = \frac{1}{|\mathcal{D}_{tr}|} \sum_{S \subseteq \mathcal{D}_{tr} \backslash \{t_i\}} \binom{n-1}{|S|}^{-1} \left(u(S \cup \{t_i\}, \mathcal{D}_{val}) - u(S, \mathcal{D}_{val}) \right).$$

Intuitively, the *importance* of t_i over a subset $S \subseteq \mathcal{D}_{tr} \backslash \{t_i\}$ is measured as the difference of the utility $u \circ \mathcal{A} \circ f(S \cup \{t_i\})$ *with* t_i to the utility $u \circ \mathcal{A} \circ f(S)$ *without* t_i. The Shapley value takes the average of all such possible subsets $S \subseteq \mathcal{D}_{tr} \backslash \{t_i\}$, which allows it to have a range of desired properties that significantly benefit data debugging tasks, often leading to more effective data debugging mechanisms compared to other leave-one-out methods.

3.3 Prior Work and Challenges

All previous research focuses on the scenario in which there is no ML pipeline f (i.e., one directly works with the vectorised training examples $\{t_i\}$). Even in this case, computing Shapley values is tremendously difficult since its complexity for general ML model is #P-hard. To accommodate this computational challenge, previous work falls into two categories:

1. *Monte Carlo Shapley*: One natural line of efforts tries to estimate Shapley value with Markov Chain Monte Carlo (MCMC) approaches. This includes vanilla Monte Carlo sampling, group testing [29, 73], and truncated Monte Carlo sampling [20].
2. *KNN Shapley*: Even the most efficient Monte Carlo Shapley methods need to train multiple ML models (i.e., evaluate $\mathcal{A}$ multiple times) and thus exhibit long running time for datasets of modest sizes. Another line of research proposes to approximate the model $\mathcal{A}$ using a simpler proxy model. Specifically, previous work shows that Shapley values can be computed over K-nearest neighbors (KNN) classifiers in PTIME [28] and using KNN classifiers as a proxy is very effective in various real-world scenarios [30].

In this work, we face an even harder problem given the presence of an ML pipeline f in addition to the model $\mathcal{A}$. Nevertheless, as a baseline, it is important to realize that all Monte Carlo Shapley approaches [20, 29] can be directly extended to support our scenario. This is because most, if not all, Monte Carlo Shapley approaches operate on *black-box functions* and thus, can be used directly to handle an end-to-end pipeline $\mathcal{A} \circ f$.

Core Technical Problem. Despite the existence of such a Monte Carlo baseline, there remain tremendous challenges with respect to scalability and speed — in our experiments in section 6, it is not uncommon for such a Monte Carlo baseline to take a full hour to compute Shapley values even on a small dataset with only 1,000 examples. To bring data debugging and understanding into practice, we are in dire need for a more efficient and scalable alternative. Without an ML pipeline, using a KNN proxy model has been shown to be orders of magnitude faster than its Monte Carlo counterpart [28] while being equally, if not more, effective on many applications [30].

As a consequence, we focus on the following question: *Can we similarly use a KNN classifier as a proxy when dealing with end-to-end ML pipelines*? Today's KNN Shapley algorithm heavily relies on the structure of the KNN classifier. The presence of an ML pipeline will drastically change the underlying algorithm and time complexity — in fact, for many ML pipelines, computation of Shapley value is #P-hard even for KNN classifiers.

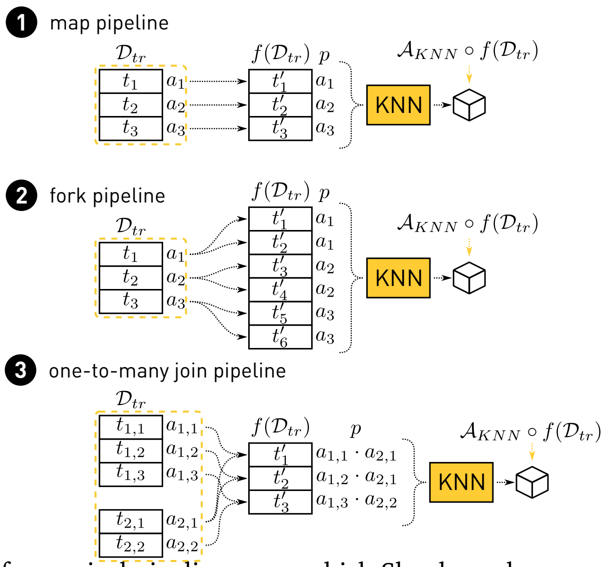

Figure 3: Three types of canonical pipelines over which Shapley values can be computed in PTIME.

4 The DataScope Approach

We summarize our main theoretical contribution in subsection 4.1, followed by the characteristics of ML pipelines to which these results are applicable (subsection 4.2). We further discuss how we can approximate many real-world pipelines as *canonical pipelines* to make them compatible with our algorithmic approach (subsection 4.3). We defer the details of our (non-trivial) theoretical results to section 5.

4.1 Overview

The key technical contribution of this paper is a novel algorithmic framework that covers a large sub-family of ML pipelines whose KNN Shapley can be computed in PTIME. We call these pipelines *canonical pipelines*.

Theorem 4.1. *Let $\mathcal{D}_{tr}$ be a set of n training tuples, f be an ML pipeline over $\mathcal{D}_{tr}$, and $\mathcal{A}_{knn}$ be a K-nearest neighbor classifier. If f can be expressed as an Additive Decision Diagram (ADD) with polynomial size, then computing*

$$\varphi_i = \frac{1}{n} \sum_{S \subseteq \mathcal{D}_{tr} \setminus \{t_i\}} \binom{n-1}{|S|}^{-1} \left(u \circ \mathcal{A}_{knn} \circ f(S \cup \{t_i\}) - u \circ \mathcal{A}_{knn} \circ f(S) \right)$$

is in PTIME for all additive utilities u.

We leave the details about this Theorem to section 5. This theorem provides a sufficient condition under which we can compute Shapley values for KNN classifiers over ML pipelines. We can instantiate this general framework with concrete types of ML pipelines.

4.2 Canonical ML Pipelines

As a prerequisite for an efficient Shapley-value computation over pipelines, we need to understand how the removal of an input tuple t_i from $\mathcal{D}_{tr}$ impacts the featurised training data $f(\mathcal{D}_{tr})$ produced by the pipeline. In particular, we need to be able to reason about the difference between $f(\mathcal{D}_{tr})$ and $f(\mathcal{D}_{tr} \setminus \{t_i\})$, which requires us to understand the *data provenance* [24, 15] of the pipeline f. In the following, we summarise three common types of pipelines (illustrated in Figure 3), to which we refer as *canonical pipelines*. We will formally prove that Shapley values over these pipelines can be computed in PTIME in section 5.

Map pipelines are a family of pipelines that satisfy the condition in Theorem 4.1, in which the feature extraction f has the following property: each input training tuple t_i is transformed into a unique output training example t_i with a tuple-at-a-time transformation function h_f: $t_i \mapsto t'_i = h_f(t_i)$. Map pipelines are the standard case for supervised learning, where each tuple of the input data is encoded as a feature vector for the model's training data. The provenance polynomial for the output t'_i is $p(t_i) = a_i$ in this case, where a_i denotes the presence of t_i in the input to the pipeline f.

Fork pipelines are a superset of Map pipelines, which requires that for each output example t_j, there exists a *unique* input tuple t_i, such that t_j is generated by applying a tuple-at-a-time transformation function h_f over t_i: $t_j = h_f(t_i)$. As illustrated in Figure 3(b), the output examples t_1 and t_2 are both generated from the input example t_1. Fork pipelines also satisfy the condition in Theorem 4.1. Fork pipelines typically originate from data augmentation operations for supervised learning, where multiple variants of a single tuple of the input data are generated (e.g., various rotations of an image in computer vision), and each copy is encoded as a feature vector for the model's training data. The provenance polynomial for an output t_j is again $p(t_j) = a_i$ in this case, where a_i denotes the presence of t_i in the input to the pipeline f.

One-to-Many Join pipelines are a superset of Fork pipelines, which rely on the star-schema structure of the relational inputs. Given the relational inputs $\mathcal{D}_e$ ("fact table") and $\mathcal{D}_s$ ("dimension table"), we require that, for each output example t_k, there exist *unique* input tuples $t_i \in \mathcal{D}_e$ and $t_j \in \mathcal{D}_s$ such that t_k is generated by applying a tuple-at-a-time transformation function h_f over the join pair (t_i, t_j): $t_k = h_f(t_i, t_j)$. One-to-Many Join pipelines also satisfy the condition in Theorem 4.1. Such pipelines occur when we have multiple input datasets in supervised learning, with the "fact" relation holding data for the entities to classify (e.g., emails in a spam detection scenario), and the "dimension" relations holding additional side data for these entities, which might result in additional helpful features. The provenance polynomial for an output t_k is $p(t_k) = a_i \cdot a_j$ in this case, where a_i and a_j denote the presence of t_i and t_j in the input to the pipeline f. Note that the polynomials states that both t_i and t_j must be present in the input at the same time (otherwise no join pair can be formed from them).

Discussion. We note that this classification of pipelines assumes that the relational operations applied by the pipeline are restricted to the positive relational algebra (SPJU: Select, Project, Join, Union), where the pipeline applies no aggregations, and joins the input data according to the star schema. In our experience, this covers a lot of real-world use cases in modern ML infrastructures, where the ML pipeline consumes pre-aggregated input data from so-called "feature stores," which is naturally modeled in a star schema. Furthermore, pipelines in the real-world operate on relational datasets using dataframe semantics [53], where unions and projections do not deduplicate their results, which (together with the absence of aggregations), has the effect that there are no additions present in provenance polynomials of the outputs of our discussed pipeline types. This pipeline model has also been proven helpful for interactive data distribution debugging [22, 23].

4.3 Approximating Real-World ML Pipelines

In practice, an ML pipeline f and its corresponding ML model $\mathcal{A}$ will often not directly give us a canonical pipeline whose Shapley value can be computed in PTIME. The reasons for this are twofold: (i) there might be no technique known to compute the Shapley value in PTIME for the given model; (ii) the estimator/-transformer operations for feature encoding in the pipeline require global aggregations (e.g., to compute the mean of an attribute for normalising it). In such cases, each output depends on the whole input, and the pipeline does not fit into one of the canonical pipeline types that we discussed earlier.

As a consequence, we *approximate* an ML pipeline into a canonical pipeline in two ways.

Approximating $\mathcal{A}$. The first approximation follows various previous efforts summarised as KNN Shapley before, and has been shown to work well, if not better, in a diverse range of scenarios. In this step, we approximate the pipeline's ML model with a KNN classifier

$$\mathcal{A} \mapsto \mathcal{A}_{knn}.$$

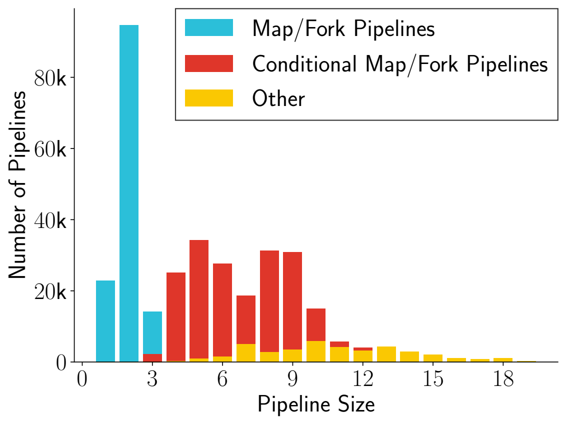

Figure 4: A majority of real-world ML pipelines [55] either already exhibit a canonical map-fork pipeline pattern, or are easily convertible to it using our approximation scheme.

Approximating the estimator/transformer steps in f. In terms of the pipeline operations, we have to deal with the global aggregations applied by the estimators for feature encoding. Common feature encoding and dimensionality reduction techniques often base on a `reduce-map` pattern over the data:

$$op(\mathcal{D}) = \texttt{map}(\texttt{reduce}(\mathcal{D}), \mathcal{D}).$$

During the `reduce` step, the estimator computes some global statistics over the dataset — e.g., the estimator for `MinMaxScaling` computes the minimum and maximum of an attribute, and the `TFIDF` estimator computes the inverse document frequencies of terms. The estimator then generates a transformer, which applies the `map` step to the data, transforming the input dataset based on the computed global statistics, e.g., to normalise each data example based on the computed minimum and maximum values in case of `MinMaxScaling`.

The global aggregation conducted by the `reduce` step is often the key reason that we cannot compute Shapley value in PTIME over a given pipeline — such a global aggregation requires us to enumerate all possible subsets of data examples, each of which corresponds to a potentially different global statistic. Fortunately, we also observe, and will validate empirically later, that the results of these global aggregations are relatively stable given different subsets of the data, especially in cases where what we want to compute is the *difference* when a single example is added or removed. The approximation that we conduct is to reuse the result of the `reduce` step computed over the whole dataset $\mathcal{D}_{tr}$ for a subset $\mathcal{D} \subset \mathcal{D}_{tr}$:

$$op(\mathcal{D}) = \texttt{map}(\texttt{reduce}(\mathcal{D}), \mathcal{D}) \mapsto op^*(\mathcal{D}) = \texttt{map}(\texttt{reduce}(\mathcal{D}_{tr}), \mathcal{D}). \tag{9}$$

In the case of scikit-learn, this means that we reuse the transformer generated by fitting the estimator on the whole dataset. Once all estimators op in an input pipeline are transformed into their approximate variant op^*, a large majority of realistic pipelines become canonical pipelines of a `map` or `fork` pattern.

Statistics of Real-world Pipelines. A natural question is how common these families of pipelines are in practice. Figure 4 illustrates a case study that we conducted over 500K real-world pipelines provided by Microsoft [55]. We divide pipelines into three categories: (1) "pure" map/fork pipelines, based on our definition of canonical pipelines; (2) "conditional" map/fork pipelines, which are comprised of a reduce operator that can be effectively approximated using the scheme we just described; and (3) other pipelines, which contain complex operators that cannot be approximated. We see that a vast majority of pipelines we encountered in our case study fall into the first two categories that we can effectively approximate using our canonical pipelines framework.

Discussion: What if These Two Approximations Fail? Computing Shapley values for a generic pipeline $(\mathcal{A}, f)$ is #P-hard, and by approximating it into $(\mathcal{A}_{knn}, f^*)$, we obtain an efficient PTIME solution. This drastic improvement on complexity also means that *we should expect that there exist scenarios under which this $(\mathcal{A}, f) \mapsto (\mathcal{A}_{knn}, f^*)$ approximation is not a good approximation.*

How often would this failure case happen in practice? When the training set is large, as illustrated in many previous studies focusing on the KNN proxy, we are confident that the $\mathcal{A} \mapsto \mathcal{A}_{knn}$ approximation should work well in many practical scenarios except those relying on some very strong global properties that KNN does not model (e.g., global population balance). As for the $f \mapsto f^*$ approximation, we expect the failure cases to be rare, especially when the training set is large. In our experiments, we have empirically verified these two beliefs, which were also backed up by previous empirical results on KNN Shapley [30].

What should we do when such a failure case happens? Nevertheless, we should expect such a failure case can happen in practice. In such situations, we will resort to the Monte Carlo baseline, which will be orders of magnitude slower but should provide a backup alternative. It is an interesting future direction to further explore the limitations of both approximations and develop more efficient Monte Carlo methods.

Approximating Additive Utilities: Equalized Odds Difference We show how slightly more complex utilities can also be represented as additive, with a little approximation, similar to the one described above. We will demonstrate this using the "equalized odds difference" utility, a measure of (un)fairness commonly used in research [26, 8] that we also use in our experiments. It can be defined as such:

$$u(\mathcal{D}_{tr}, \mathcal{D}_{val}) := \max\{TPR_{\Delta}(\mathcal{D}_{tr}, \mathcal{D}_{val}), FPR_{\Delta}(\mathcal{D}_{tr}, \mathcal{D}_{val})\}. \tag{10}$$

Here, TPR_{Δ} and FPR_{Δ} are *true positive rate difference* and *false positive rate difference* respectively. We assume that each tuple $t_{tr} \in f(\mathcal{D}_{tr})$ and $t_{val} \in f(\mathcal{D}_{val})$ have some sensitive feature g (e.g. ethnicity) with values taken from some finite set $\{G_1, G_2, ...\}$, that allows us to partition the dataset into *sensitive groups*. We can define TPR_{Δ} and FPR_{Δ} respectively as

$$\begin{aligned} TPR_{\Delta}(\mathcal{D}_{tr}, \mathcal{D}_{val}) &:= \max_{G_i \in G} TPR_{G_i}(\mathcal{D}_{tr}, \mathcal{D}_{val}) - \min_{G_j \in G} TPR_{G_j}(\mathcal{D}_{tr}, \mathcal{D}_{val}), \text{ and} \\ FPR_{\Delta}(\mathcal{D}_{tr}, \mathcal{D}_{val}) &:= \max_{G_i \in G} FPR_{G_i}(\mathcal{D}_{tr}, \mathcal{D}_{val}) - \min_{G_j \in G} FPR_{G_j}(\mathcal{D}_{tr}, \mathcal{D}_{val}). \end{aligned} \tag{11}$$

For some sensitive group G_i, we define TPR_{G_i} and FPR_{G_i} respectively as:

$$\begin{aligned} TPR_{G_i}(\mathcal{D}_{tr}, \mathcal{D}_{val}) &:= \frac{\sum_{t_{val} \in f(\mathcal{D}_{val})} \mathbb{1}\{(\mathcal{A} \circ f(\mathcal{D}_{tr}))(t_{val}) = 1\} \mathbb{1}\{y(t_{val}) = 1\} \mathbb{1}\{g(t_{val}) = G_i\}}{|\{t_{val} \in \mathcal{D}_{val} \ : \ y(t_{val}) = 1 \wedge g(t_{val}) = G_i\}|}, \text{ and} \\ FPR_{G_i}(\mathcal{D}_{tr}, \mathcal{D}_{val}) &:= \frac{\sum_{t_{val} \in f(\mathcal{D}_{val})} \mathbb{1}\{(\mathcal{A} \circ f(\mathcal{D}_{tr}))(t_{val}) = 1\} \mathbb{1}\{y(t_{val}) = 0\} \mathbb{1}\{g(t_{val}) = G_i\}}{|\{t_{val} \in \mathcal{D}_{val} \ : \ y(t_{val}) = 0 \wedge g(t_{val}) = G_i\}|} \end{aligned}$$

For a given training dataset $\mathcal{D}_{tr}$, we can determine Equation 10 whether TPR_{Δ} or FPR_{Δ} is going to be the dominant metric. Similarly, given that choice, we can determine a pair of sensitive groups (G_{max}, G_{min}) that would end up be selected as minimal and maximal in Equation 11. Similarly to the conversion shown in Equation 9, we can treat these two steps as a `reduce` operation over the whole dataset. Then, if we assume that this intermediate result will remain stable over subsets of $\mathcal{D}_{tr}$, we can approximatly represent the equalized odds difference utility as an additive utility.

As an example, let us assume that we have determined that TPR_{Δ} dominates over FPR_{Δ}, and similarly that the pair of sensitive groups (G_{max}, G_{min}) will end up being selected in Equation 11. Then, our tuple-wise utility u_T and the scaling factor w become

$$\begin{aligned} u_T(y_{pred}, t_{val}) &:= TPR_{G_{max},T}(y_{pred}, t_{val}) - TPR_{G_{min},T}(y_{pred}, t_{val}), \\ w &:= 1/|\{t_{val} \in \mathcal{D}_{val} \ : \ y(t_{val}) = 1 \wedge g(t_{val}) = G_i\}|, \end{aligned}$$

where

$$TPR_{G_i,T}(y_{pred}, t_{val}) := \mathbb{1}\{y_{pred} = 1\} \mathbb{1}\{y(t_{val}) = 1\} \mathbb{1}\{g(t_{val}) = G_i\}.$$

A similar approach can be taken to define u_T and w for the case when FPR_{Δ} dominates over TPR_{Δ}.

5 Algorithm Framework: KNN Shapley Over Data Provenance

We now provide details for our theoretical results that are mentioned in section 4. We present an algorithmic framework that efficiently computes the Shapley value over the KNN accuracy utility (defined in Equation 6 when $\mathcal{A}$ is the KNN model). Our framework is based on the following key ideas: (1) the computation can be reduced to computing a set of *counting oracles*; (2) we can develop PTIME algorithms to compute such counting oracles for the canonical ML pipelines, by translating their *provenance polynomials* into an Additive Decision Diagram (ADD).

5.1 Counting Oracles

We now unpack Theorem 4.1. Using the notations of data provenance introduced in section 2, we can rewrite the definition of the Shapley value as follows, computing the value of tuple t_i, with the corresponding varible $a_i \in A$:

$$\varphi_i = \frac{1}{|A|} \sum_{v \in \mathcal{V}_{A \setminus \{a_i\}}} \frac{u(\mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]) - u(\mathcal{D}_{tr}^f[v[a_i \leftarrow 0]])}{\binom{|A|-1}{|\text{supp}(v)|}}. \tag{12}$$

Here, $v[a_i \leftarrow X]$ represents the same value assignment as v, except that we enforce $v(a_i) = X$ for some constant X. Moreover, the support $\text{supp}(v)$ of a value assignment v is the subset of variables in A that are assigned value 1 according to v.

Nearest Neighbor Utility. When the downstream classifier is a K-nearest neighbor classifier, we have additional structure of the utility function $u(-)$ that we can take advantage of. Given a data example t_{val} from the validation dataset, the hyperparameter K controlling the size of the neighborhood and the set of class labels $\mathcal{Y}$, we formally define the KNN utility $u_{t_{val},K,\mathcal{Y}}$ as follows. Given the transformed training set $\mathcal{D}_{tr}^f$, let σ be a scoring function that computes, for each tuple $t \in \mathcal{D}_{tr}^f$, its similarity with the validation example t_{val}: $\sigma(t, t_{val})$. In the following, we often write $\sigma(t)$ whenever t_{val} is clear from the context. We also omit σ when the scoring function is clear from the context. Given this scoring function σ, the KNN utility can be defined as follows:

$$u_{t_{val},K,\mathcal{Y}}(\mathcal{D}) := u_T\left(\text{argmax}_{y \in \mathcal{Y}}\left(\text{tally}_{y,\text{top}_K \mathcal{D}_{tr}^f}(\mathcal{D}_{tr}^f)\right), t_{val}\right) \tag{13}$$

where $\text{top}_K \mathcal{D}_{tr}^f$ returns the tuple t which ranks at the K-th spot when all tuples in $\mathcal{D}_{tr}^f$ are ordered by decreasing similarity σ. Given this tuple t and a class label $y \in \mathcal{Y}$, the $\text{tally}_{y,t}$ operator returns the number of tuples with similarity score greater or equal to t that have label y. We assume a standard majority voting scheme where the predicted label is selected to be the one with the greatest tally ($\arg\max_y$). The accuracy is then computed by simply comparing the predicted label with the label of the validation tuple t_{val}.

Plugging the KNN accuracy utility into Equation 12, we can augment the expression for computing φ_i as

$$\begin{aligned}
\varphi_i = \frac{1}{|A|} \sum_{v \in \mathcal{V}_{A \setminus \{a_i\}}} & \sum_{\alpha=1}^{|A|} \mathbb{1}\{\alpha = |\text{supp}(v)|\} \binom{|A|-1}{\alpha}^{-1} \\
& \cdot \sum_{t,t' \in \mathcal{D}_{tr}^f} \mathbb{1}\{t = \text{top}_K \mathcal{D}_{tr}^f[v[a_i \leftarrow 0]]\} \\
& \qquad \cdot \mathbb{1}\{t' = \text{top}_K \mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]\} \\
& \cdot \sum_{\gamma,\gamma' \in \Gamma} \mathbb{1}\{\gamma = \text{tally}_t \mathcal{D}_{tr}^f[v[a_i \leftarrow 0]]\} \\
& \qquad \cdot \mathbb{1}\{\gamma' = \text{tally}_{t'} \mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]\} \\
& \qquad \cdot u_{\Delta}(\gamma, \gamma')
\end{aligned} \tag{14}$$

where $\text{tally}_t\mathcal{D} = (\text{tally}_{c_1,t}\mathcal{D} ... \text{tally}_{y_{|\mathcal{Y}|},t}\mathcal{D})$ returns a tally vector $\gamma \in \Gamma \subset \mathbb{N}^{|\mathcal{Y}|}$ consisting of the tallied occurrences of each class label $y \in \mathcal{Y}$ among tuples with similarity to t_{val} greater than or equal to that of the boundary tuple t. Let Γ be all possible tally vectors (corresponding to all possible label "distributions" over top-K).

Here, the innermost utility gain function is formally defined as $u_\Delta(\gamma, \gamma') := u_\Gamma(\gamma') - u_\Gamma(\gamma)$, where u_Γ is defined as

$$u_\Gamma(\gamma) := u_T(\text{argmax}_{y \in \mathcal{Y}}\gamma, t_{val}).$$

Intuitively, $u_\Delta(\gamma, \gamma')$ measures the utility difference between two different label distributions (i.e., tallies) of top-K examples: γ and γ'. $u_T(y, t_{val})$ is the tuple-wise utility for a KNN prediction (i.e., $\text{argmax}_{y \in \mathcal{Y}}\gamma$) and validation tuple t_{val}, which is the building block of the *additive utility*. The correctness of Equation 14 comes from the observation that for any distinct $v \in \mathcal{V}_{A \setminus \{a_i\}}$, there is a unique solution to all indicator functions $\mathbb{1}$. Namely, there is a single t that is the K-th most similar tuple when $v(a_i) = 0$, and similarly, a single t' when $v(a_i) = 1$. Given those *boundary tuples* t and t', the same goes for the *tally vectors*: given $\mathcal{D}_{tr}^f[v[a_i \leftarrow 0]]$ and $\mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]$, there exists a unique γ and γ'.

We can now define the following *counting oracle* that computes the sum over value assignments, along with all the predicates:

$$\begin{aligned}\omega_{t,t'}(\alpha, \gamma, \gamma') := \sum_{v \in \mathcal{V}_{A \setminus \{a_i\}}} &\cdot \mathbb{1}\{\alpha = |\text{supp}(v)|\} \\ &\cdot \mathbb{1}\{t = \text{top}_K \mathcal{D}_{tr}^f[v[a_i \leftarrow 0]]\} \\ &\cdot \mathbb{1}\{t' = \text{top}_K \mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]\} \\ &\cdot \mathbb{1}\{\gamma = \text{tally}_t \mathcal{D}_{tr}^f[v[a_i \leftarrow 0]]\} \\ &\cdot \mathbb{1}\{\gamma' = \text{tally}_t \mathcal{D}_{tr}^f[v[a_i \leftarrow 1]]\}.\end{aligned} \tag{15}$$

Using counting oracles, we can simplify Equation 14 as:

$$\varphi_i = \frac{1}{N} \sum_{t,t' \in \mathcal{D}_{tr}^f} \sum_{\alpha=1}^{N} \binom{N-1}{\alpha}^{-1} \sum_{\gamma, \gamma' \in \Gamma} u_\Delta(\gamma, \gamma') \omega_{t,t'}(\alpha, \gamma, \gamma'). \tag{16}$$

We see that the computation of φ_i will be in PTIME if we can compute the counting oracles $\omega_{t,t'}$ in PTIME (ref. Theorem 4.1). As we will demonstrate next, this is indeed the case for the canonical pipelines that we focus on in this paper.

5.2 Counting Oracles for Canonical Pipelines

We start by discussing how to compute the counting oracles using ADD's in general. We then study the canonical ML pipelines in particular and develop PTIME algorithms for them.

5.2.1 Counting Oracle using ADD's

We use Additive Decision Diagram (ADD) to compute the counting oracle $\omega_{t,t'}$ (Equation 15). An ADD represents a Boolean function $\phi : \mathcal{V}_A \to \mathcal{E} \cup \{\infty\}$ that maps value assignments $v \in \mathcal{V}_A$ to elements of some set $\mathcal{E}$ or a special invalid element ∞ (see subsection 2.2 for more details). For our purpose, we define $\mathcal{E} := \{1, ..., |A|\} \times \Gamma \times \Gamma$, where Γ is the set of label tally vectors. We then define a function over Boolean inputs $\phi_{t,t'} : \mathcal{V}_A[a_i = 0] \to \mathbb{N}$ as follows:

$$
\begin{aligned}
\phi_{t,t'}(v) &:= \begin{cases} \infty, & \text{if } t \notin \mathcal{D}\big[v[a_i \leftarrow 0]\big], \\ \infty, & \text{if } t' \notin \mathcal{D}\big[v[a_i \leftarrow 1]\big], \\ (\alpha, \gamma, \gamma'), & \text{otherwise,} \end{cases} \\
\alpha &:= |\mathrm{supp}(v)|, \\
\gamma &:= \mathrm{tally}_t \mathcal{D}\big[v[a_i \leftarrow 0]\big], \\
\gamma' &:= \mathrm{tally}_{t'} \mathcal{D}\big[v[a_i \leftarrow 1]\big].
\end{aligned} \tag{17}
$$

If we can construct an ADD with a root node $n_{t,t'}$ that computes $\phi_{t,t'}(v)$, then the following equality holds:

$$\omega_{t,t'}(\alpha, \gamma, \gamma') = \mathrm{count}_{(\alpha,\gamma,\gamma')}(n_{t,t'}). \tag{18}$$

Given that the complexity of model counting is $O(|\mathcal{N}| \cdot |\mathcal{E}|)$ (see Equation 4) and the size of $\mathcal{E}$ is polynomial in the size of data, we have

Theorem 5.1. *If we can represent the $\phi_{t,t'}(v)$ in Equation 17 with an ADD of size polynomial in $|A|$ and $|\mathcal{D}_{tr}^f|$, we can compute the counting oracle $\omega_{t,t'}$ in time polynomial of $|A|$ and $|\mathcal{D}_{tr}^f|$.*

5.2.2 Constructing Polynomial-size ADD's for ML Pipelines

Algorithm 1 presents our main procedure CompileADD that constructs an ADD for a given dataset $\mathcal{D}$ made up of tuples annotated with provenance polynomials. Invoking CompileADD($\mathcal{D}$, A, t) constructs an ADD with node set $\mathcal{N}$ that computes

$$\phi_t(v) := \begin{cases} \infty, & \text{if } t \notin \mathcal{D}[v], \\ \mathrm{tally}_t \mathcal{D}[v], & \text{otherwise.} \end{cases} \tag{19}$$

We provide a more detailed description of Algorithm 1 in subsection C.7.

To construct the function defined in Equation 17, we need to invoke CompileADD once more by passing t' instead of t in order to obtain another diagram $\mathcal{N}'$. The final diagram is obtained by $\mathcal{N}[a_i \leftarrow 0] + \mathcal{N}'[a_i \leftarrow 1]$. The size of the resulting diagram will still be bounded by $O(|\mathcal{D}|)$.

We can now examine different types of canonical pipelines and see how their structures are reflected onto the ADD's. In summary, we can construct an ADD with polynomial size for canonical pipelines and therefore, by Theorem 5.1, the computation of the corresponding counting oracles is in PTIME.

One-to-Many Join Pipeline. In a *star* database schema, this corresponds to a *join* between a *fact* table and a *dimension* table, where each tuple from the dimension table can be joined with multiple tuples from the fact table. It can be represented by an ADD similar to the one in Figure 2a.

Corollary 5.1.1. *For the K-NN accuracy utility and a one-to-many* join *pipeline, which takes as input two datasets, $\mathcal{D}_F$ and $\mathcal{D}_D$, of total size $|\mathcal{D}_F| + |\mathcal{D}_D| = N$ and outputs a joined dataset of size $O(N)$, the Shapley value can be computed in $O(N^4)$ time.*

We present the proof in subsection C.2 in the appendix.

Fork Pipeline. The key characteristic of a pipeline f that contains only *fork* or *map* operators is that the resulting dataset $f(\mathcal{D})$ has provenance polynomials with only a single variable. This is due to the absence of joins, which are the only operator that results in provenance polynomials with a combination of variables.

Corollary 5.1.2. *For the K-NN accuracy utility and a* fork *pipeline, which takes as input a dataset of size N and outputs a dataset of size M, the Shapley value can be computed in $O(M^2 N^2)$ time.*

Algorithm 1 Compiling a provenance-tracked dataset into ADD.

```
function CompileADD
    inputs
        $\mathcal{D}$, provenance-tracked dataset;
        $A$, set of variables;
        $t$, boundary tuple;
    outputs
        $\mathcal{N}$, nodes of the compiled ADD;
begin
    $\mathcal{N} \leftarrow \{\}$
    $\mathcal{P} \leftarrow \{(x_1, x_2) \in A \ : \exists t \in \mathcal{D}, x_1 \in p(t) \ \wedge \ x_2 \in p(t)\}$
    $A_L \leftarrow$ GetLeafVariables($\mathcal{P}$)
    for $A_C \in$ GetConnectedComponents($\mathcal{P}$) do
        $\mathcal{N}' \leftarrow$ ConstructADDTree($A_C \setminus w_L$)
        $A' \leftarrow A_C \setminus w_L$
        $\mathcal{D}' \leftarrow \{t' \in \mathcal{D} \ : \ p(t') \cup A_C \neq \emptyset\}$
        for $v \in \mathcal{V}_{A'}$ do
            $\mathcal{N}_C \leftarrow$ ConstructADDChain($A_C \cap w_L$)
            for $n \in \mathcal{N}_C$ do
                $v' \leftarrow v \cup \{x(n) \rightarrow 1\}$
                $w_H(n) \leftarrow |\{t' \in \mathcal{D}' \ : \ \mathrm{eval}_{v'} p(t') = 1 \ \wedge \ \sigma(t') \geq \sigma(t)\}|$
            end for
            $\mathcal{N}' \leftarrow$ AppendToADDPath($\mathcal{N}'$, $\mathcal{N}_C$, $v$)
        end for
        $\mathcal{N} \leftarrow$ AppendToADDRoot($\mathcal{N}$, $\mathcal{N}'$)
    end for
    for $x' \in p(t)$ do
        for $n \in \mathcal{N}$ where $x(n) = x'$ do
            $w_L(n) \leftarrow \infty$
        end for
    end for
    return $\mathcal{N}$
end function
```

We present the proof in subsection C.3 in the appendix.

Map Pipeline. A *map* pipeline is similar to *fork* pipeline in the sense that every provenance polynomial contains only a single variable. However, each variable now can appear in a provenance polynomial of *at most* one tuple, in contrast to *fork* pipeline where a single variable can be associated with *multiple* tuples. This additional restriction results in the following corollary:

Corollary 5.1.3. *For the K-NN accuracy utility and a* map *pipeline, which takes as input a dataset of size N, the Shapley value can be computed in $O(N^2)$ time.*

We present the proof in subsection C.4 in the appendix.

5.3 Special Case: 1-Nearest-Neighbor Classifiers

We can significantly reduce the time complexity for 1-NN classifiers, an important special case of K-NN classifiers that is commonly used in practice. For each validation tuple t_{val}, there is always *exactly* one tuple that is most similar to t_{val}. Below we illustrate how to leverage this observation to construct the counting oracle. In the following, we assume that a_i is the variable corresponding to the tuple for which we hope to compute Shapley value.

Let ϕ_t represent the event when t is the top-1 tuple:

$$\phi_t := p(t) \wedge \bigwedge_{\substack{t' \in f(\mathcal{D}_{tr}) \\ \sigma(t') > \sigma(t)}} \neg p(t'). \tag{20}$$

For Equation 20 to be *true* (i.e. for tuple t to be the top-1), all tuples t' where $\sigma(t') > \sigma(t)$ need to be *absent* from the pipeline output. Hence, for a given value assignment v, all provenance polynomials that control those tuples, i.e., $p(t')$, need to evaluate to false.

We now construct the event

$$\phi_{t,t'} := \phi_t[a_i/\mathsf{false}] \wedge \phi_{t'}[a_i/\mathsf{true}],$$

where $\phi_t[a_i/\mathsf{false}]$ means to substitue all appearances of a_i in ϕ_t to false. This event happens only if if t is the top-1 tuple when a_i is false and t' is the top-1 tuple when a_i is true. This corresponds to the condition that our counting oracle counts models for. Expanding $\phi_{t,t'}$, we obtain

$$\phi_{t,t'} := \Big(p(t) \wedge \bigwedge_{\substack{t'' \in f(\mathcal{D}_{tr}) \\ \sigma(t'') > \sigma(t)}} \neg p(t'')\Big)[a_i/\mathsf{false}] \wedge \Big(p(t') \wedge \bigwedge_{\substack{t'' \in f(\mathcal{D}_{tr}) \\ \sigma(t'') > \sigma(t')}} \neg p(t'')\Big)[a_i/\mathsf{true}]. \tag{21}$$

Note that $\phi_{t,t'}$ can only be *true* if $p(t')$ is true when a_i is true and $\sigma(t) < \sigma(t')$. As a result, all provenance polynomials corresponding to tuples with a higher similarity score than that of t need to evaluate to false. Therefore, the only polynomials that can be allowed to evaluate to true are those corresponding to tuples with lower similarity score than t. Based on these observations, we can express the counting oracle for different types of ML pipelines.

Map Pipeline. In a *map* pipeline, the provenance polynomial for each tuple $t \in f(\mathcal{D}_{tr})$ is defined by a single distinct variable $a_t \in A$. Furthermore, from the definition of the counting oracle (Equation 15), we can see that each $\omega_{t,t'}$ counts the value assignments that result in support size α and label tally vectors γ and γ'. Given our observation about the provenance polynomials that are allowed to be set to true, we can easily construct an expression for counting valid value assignments. Namely, we have to choose exactly α variables out of the set $\{t'' \in \mathcal{D} \ : \ \sigma(t'') < \sigma(t)\}$, which corresponds to tuples with lower similarity than that of t. This can be constructed using a *binomial coefficient*. Furthermore, when $K = 1$, the label tally γ is entirely determined by the top-1 tuple t. The same observation goes for γ' and t'. To denote this, we define a constant Γ_L parameterized by some label L. It represents a tally vector with all values 0 and only the value corresponding to label L being set to 1. We thus need to fix γ to be equal to $\Gamma_{y(t)}$ (and the same for γ'). Finally, as we observed earlier, when computing $\omega_{t,t'}$ for $K = 1$, the provenance polynomial of the tuple t' must equal a_i. With these notions, we can define the counting oracle as

$$\omega_{t,t'}(\alpha, \gamma, \gamma') = \binom{|\{t'' \in \mathcal{D} \ : \ \sigma(t'') < \sigma(t)\}|}{\alpha} \mathbb{1}\{p(t') = a_i\} \mathbb{1}\{\gamma = \Gamma_{y(t)}\} \mathbb{1}\{\gamma' = \Gamma_{y(t')}\}. \tag{22}$$

Note that we always assume $\binom{a}{b} = 0$ for all $a < b$. Given this, we can prove the following corollary about *map* pipelines:

Corollary 5.1.4. *For the* 1*-NN accuracy utility and a* map *pipeline, which takes as input a dataset of size* N*, the Shapley value can be computed in* $O(N \log N)$ *time.*

We present the proof in subsection C.5 in the appendix.

Fork Pipeline. As we noted, both *map* and *fork* pipelines result in polynomials made up of only one variable. The difference is that in *map* pipeline each variable is associated with at most one polynomial, whereas in *fork* pipelines it can be associated with multiple polynomials. However, for 1-NN classifiers, this difference vanishes when it comes to Shapley value computation:

Corollary 5.1.5. *For the* 1*-NN accuracy utility and a* fork *pipeline, which takes as input a dataset of size* N*, the Shapley value can be computed in* $O(N \log N)$ *time.*

We present the proof in subsection C.6 in the appendix.

Dataset	Modality	# Examples	# Features
UCIAdult [37]	tabular	$49K$	14
Folktables [17]	tabular	$1.6M$	10
FashionMNIST [71]	image	$14K$	(Image) 28×28
20NewsGroups [32]	text	$1.9K$	(Text) $20K$ after TF-IDF
Higgs [7]	tabular	$11M$	28

Table 1: Datasets characteristics

6 Experimental Evaluation

We evaluate the performance of DataScope when applied to data debugging and repair. In this section, we present the empirical study we conducted with the goal of evaluating both quality and speed.

6.1 Experimental Setup

Hardware and Platform. All experiments were conducted on Amazon AWS c5.metal instances with a 96-core Intel(R) Xeon(R) Platinum 8275CL 3.00GHz CPU and 192GB of RAM. We ran each experiment in single-thread mode.

Datasets. We assemble a collection of widely used datasets with diverse modalities (i.e. tabular, textual, and image datasets). Table 1 summarizes the datasets that we used.
(Tabular Datasets) UCIAdult is a tabular dataset from the US census data [37]. We use the binary classification variant where the goal is to predict whether the income of a person is above or below \$50K. One of the features is 'sex,' which we use as a *sensitive attribute* to measure group fairness with respect to male and female subgroups. A very similar dataset is Folktables, which was developed to redesign and extend the original UCIAdult dataset with various aspects interesting to the fairness community [17]. We use the 'income' variant of this dataset, which also has a 'sex' feature and has a binary label corresponding to the \$50K income threshold. Another tabular dataset that we use for large-scale experiments is the Higgs dataset, which has 28 features that represent physical properties of particles in an accelerator [7]. The goal is to predict whether the observed signal produces Higgs bosons or not.
(Non-tabular Datasets) We used two non-tabular datasets. One is FashionMNIST, which contains 28×28 grayscale images of 10 different categories of fashion items [71]. To construct a binary classification task, we take only images of the classes 'shirt' and 'T-shirt.' We also use TwentyNewsGroups, which is a dataset with text obtained from newsgroup posts categorized into 20 topics [32]. To construct a binary classification task, we take only two newsgroup categories, 'sci.med' and 'comp.graphics.' The task is to predict the correct category for a given piece of text.

Feature Processing Pipelines. We obtained a dataset with about $500K$ machine learning workflow instances from internal Microsoft users [55]. Each workflow consists of a dataset, a feature extraction pipeline, and an ML model. We identified a handful of the most representative pipelines and translated them to `sklearn` pipelines. We list the pipelines used in our experiments in Table 2.

As Table 2 shows, we used pipelines of varying complexity. The data modality column indicates which types of datasets we applied each pipeline to. Some pipelines are pure map pipelines, while some implicitly require a reduce operation. Table 2 shows the operators contained by each pipeline. They are combined either using a composition symbol $\circ$, i.e., operators are applied in sequence; or a concatenation symbol $\oplus$, i.e., operators are applied in parallel and their output vectors are concatenated. Some operators are taken directly from `sklearn` (`StandardScaler`, `PCA`, `MissingIndicator`, `KMeans`, `CountVectorizer`, and `TfidfTransformer`), while others require customized implementations: (1) `Log1P`, using the `log1p` function from `numpy`; (2) `GaussBlur`, using the `gaussian_filter` function from `scipy`; (3) `HogTransform`, using the `hog` function from `skimage`; (4) `TextToLower`, using the built-in `tolower` Python function; and (5) `UrlRemover`, using a simple regular expression.

Pipeline	**Dataset Modality**	**w/ Reduce**	**Operators**
Identity	tabular	false	$\emptyset$
StandardScaler	tabular	true	`StandardScaler`
LogarithmicScaler	tabular	true	`Log1P` $\circ$ `StandardScaler`
PCA	tabular	true	`PCA`
MissingIndicator + KMeans	tabular	true	`MissingIndicator` $\oplus$ `KMeans`
GaussianBlur	image	false	`GaussBlur`
HistogramofOrientedGradients	image	false	`HogTransform`
TFIDF	text	true	`CountVectorizer` $\circ$ `TfidfTransformer`
Tolower + URLRemove + TFIDF	text	false	`TextToLower` $\circ$ `UrlRemover` $\circ$`CountVectorizer` $\circ$ `TfidfTransformer`

Table 2: Feature extraction pipelines used in experiments.

Fork Variants: We also create a "fork" version of the above pipelines, by prepending each with a `DataProvider` operator. It simulates distinct data providers that each provides a portion of the data. The original dataset is split into a given number of groups (we set this number to 100 in our experiments). We compute importance for each group, and we conduct data repairs on entire groups all at once.

Models. We use three machine learning models as the downstream ML model following the previous feature extraction pipelines: XGBoost, LogisticRegression, and KNearestNeighbor. We use the `LogisticRegression` and `KNeighborsClassifier` provided by the `sklearn` package. We use the default hyper-parameter values except that we set `max_iter` to 5,000 for LogisticRegression and `n_neighbors` to 1 for the KNearestNeighbor.

Data Debugging Methods. We apply different data debugging methods and compare them based on their effect on model quality and the computation time that they require:

- Random — We measure importance with a random number and thus apply data repairs in random order.
- TMCShapleyx10 and TMCShapleyx100 — We express importance as Shapley values computed using the Truncated Monte-Carlo (TMC) method [20], with 10 and 100 Monte-Carlo iterations, respectively. We then follow the computed importance in ascending order to repair data examples.
- DataScope — This is our K-nearest-neighbor based method for efficiently computing the Shapley value. We then follow the computed importance in ascending order to repair data examples.
- DataScopeInteractive — While the above methods compute importance scores only once at the beginning of the repair, the speed of DataScope allows us to *recompute* the importance after *each* data repair. We call this strategy DataScopeInteractive.

Protocol. In most of our experiments (unless explicitly stated otherwise), we simulate importance-driven data repair scenarios performed on a given *training dataset*. In each experimental run, we select a dataset, pipeline, model, and a data repair method. We compute the importance using the utility defined over a *validation set*. Training data repairs are conducted one unit at a time until all units are examined. The order of units is determined by the specific repair method. We divide the range between 0% data examined and 100% data examined into 100 checkpoints. At each checkpoint we measure the quality of the given model on a separate *test dataset* using some metric (e.g. accuracy). For importance-based repair methods, we also measure the time spent on computing importance. We repeat each experiment 10 times and report the median as well as the 90-th percentile range (either shaded or with error bars).

6.2 Results

Following the protocol of [42, 30], we start by flipping certain amount of labels in the training dataset. We then use a given data debugging method to go through the dataset and repair labels by replacing each label with the correct one. As we progress through the dataset, we measure the model quality on a separate test

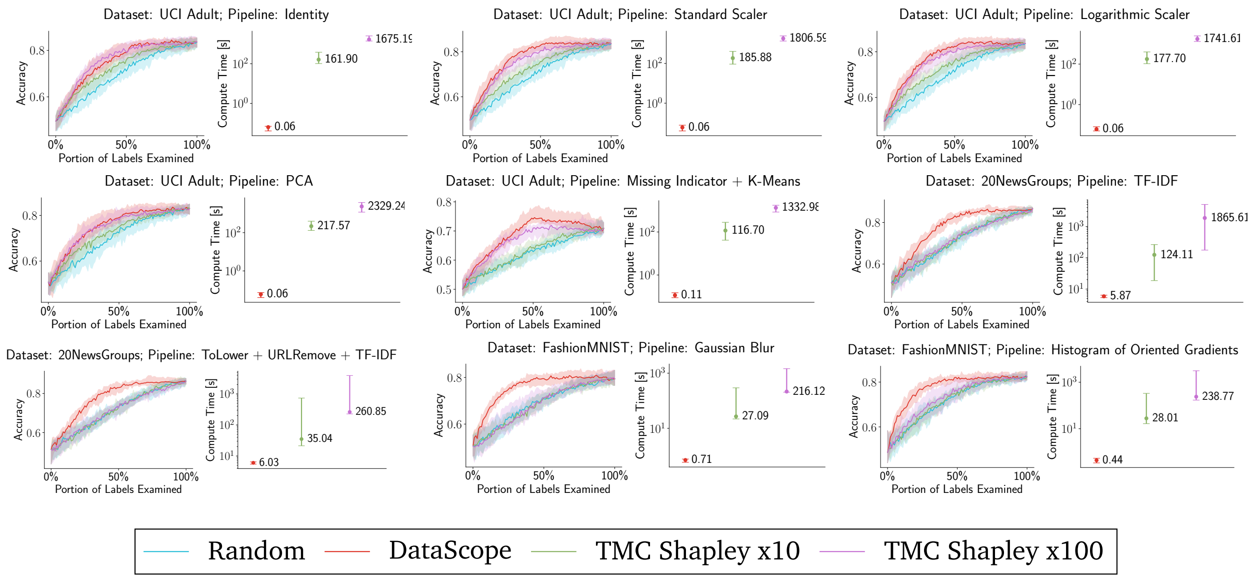
Figure 5: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for accuracy. The model is XGBoost.

dataset using a metric such as accuracy or equalized odds difference (a commonly used fairness metric). Our goal is to achieve the best possible quality while at the same time having to examine the least possible amount of data. Depending on whether the pipeline is an original one or its fork variant, we have slightly different approaches to label corruption and repair. For original pipelines, each label can be flipped with some probability (by default this is 50%). Importance is computed for independent data examples, and repairs are performed independently as well. For fork variants, data examples are divided into groups corresponding to their respective data providers. By default, we set the number of data providers to 100. Each label inside a single group is flipped based on a fixed probability. However, this probability differs across data providers (going from 0% to 100%). Importance is computed for individual providers, and when a provider is selected for repair, all its labels get repaired.

Improving Accuracy with Label Repair. In this set of experiments, we aim to improve the accuracy as much as possible with as least as possible labels examined. We show the case for XGBoost in Figure 5 while leave other scenarios (logistic regression, K-nearest neighbor, original pipelines and fork variants) to Appendix (Figure 8 to Figure 12. Figures differ with respect to the target model used (XGBoost, logistic regression, or K-nearest neighbor) and the type of pipeline (either map or fork). In each figure, we show results for different pairs of dataset and pipeline, and we measure the performance of the target model as well as the time it took to compute the importance scores.

We see that `DataScope` is significantly faster than TMC-based methods. The speed-up is in the order of $100\times$ to $1,000\times$ for models such as logistic regression. For models requiring slightly longer training time (e.g., XGBoost), the speed-up can be up to $10,000\times$.

In terms of quality, we see that `DataScope` is comparable with or better than the TMC-based methods (mostly for the logistic regression model), both outperforming the Random repair method. In certain cases, `DataScope`, despite its orders of magnitude speed-up, also clearly dominates the TMC-based methods, especially when the pipelines produce features of high-dimensional datasets (such as the text-based pipelines used for the 20NewsGroups dataset and the image-based pipelines used for the FashionMNIST dataset).

Improving Accuracy and Fairness. We then explore the relationship between accuracy and fairness when performing label repairs. Figure 6 shows the result for XGBoost over original pipelines and we leave other scenarios to the Appendix (Figure 13 to Figure 17). In these experiments we only use the two tabular

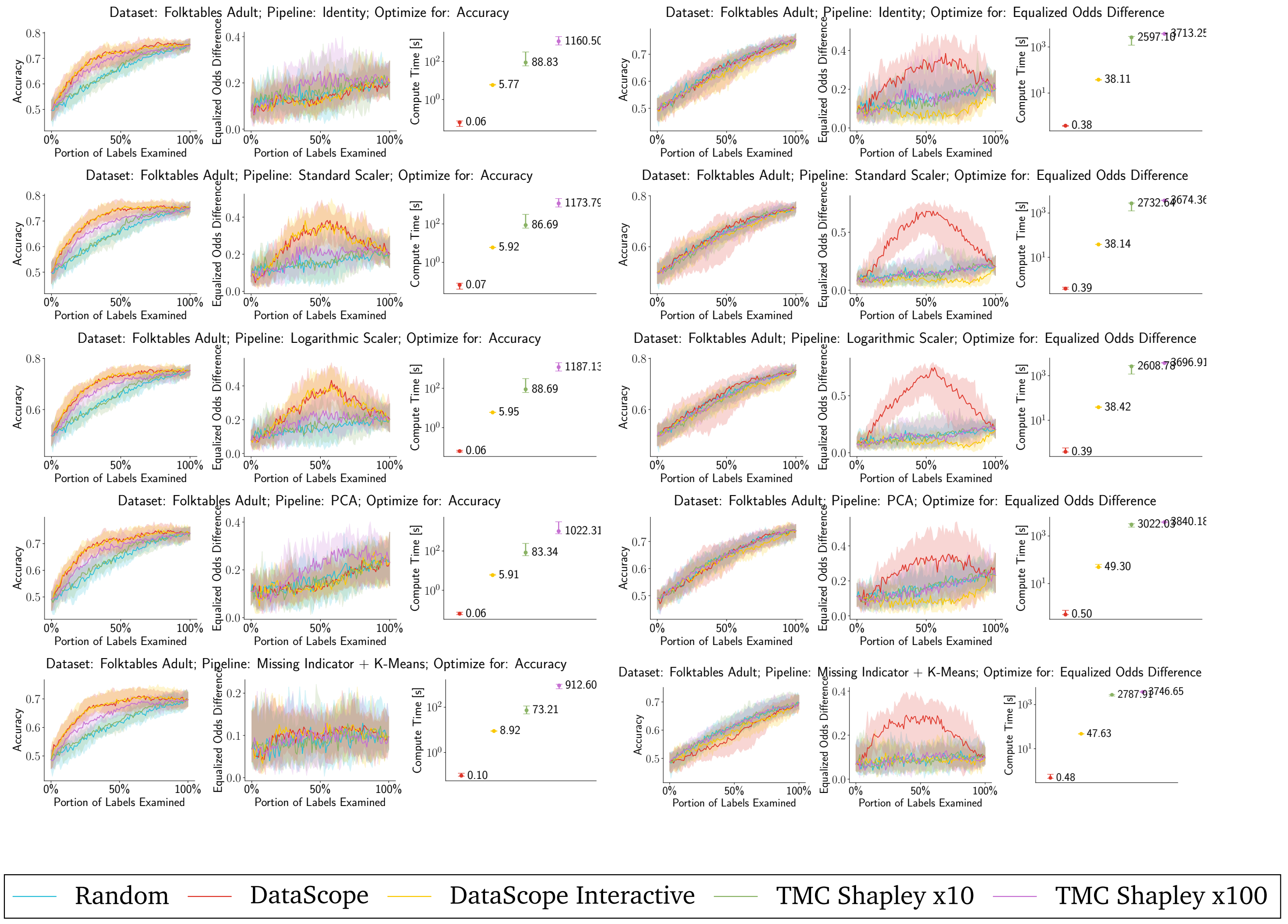

Figure 6: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for fairness. The model is XGBoost.

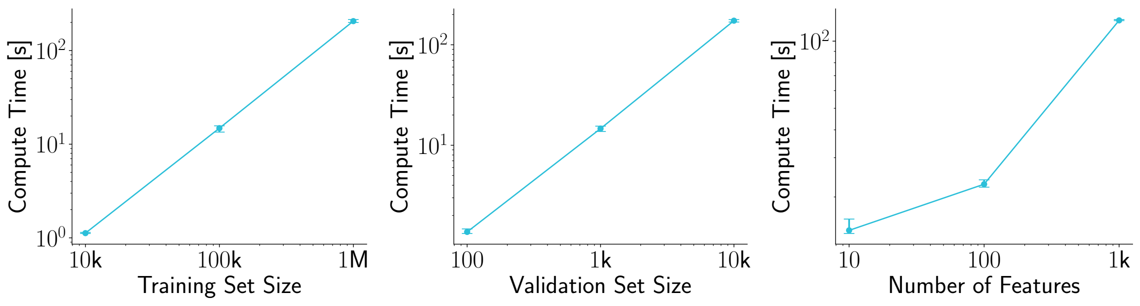

Figure 7: Scalability of `DataScope`

datasets UCIAdult and Folktables, which have a 'sex' feature that we use to compute group fairness using *equalized odds difference*, one of the commonly used fairness metrics [26]. We use equalized odds difference as the utility function for both `DataScope` and TMC-based methods.

We first see that being able to debug specifically for fairness is important — the left panel of Figure 6 illustrates the behavior of optimizing for accuracy whereas the right panel illustrates the behavior of optimizing for fairness. In this example, the 100% clean dataset is unfair. When optimizing for accuracy, we see that the unfairness of model can also increase. On the other hand, when taking fairness into consideration, DataScopeInteractive is able to maintain fairness while improving accuracy significantly — the unfairness increase of DataScopeInteractive only happens at the very end of the cleaning process, where all other "fair" data examples have already been cleaned. This is likely due to the way that the equalized odds difference utility is approximated in DataScope. When computing the utility, we first make a choice on which G_i and G_j to choose in Equation 11, as well as a choice between TPR_Δ and FPR_Δ in Equation 10; only then we compute the Shapley value. We assume that these choices are stable over the entire process of label repair. However, if these choices are ought to change, only DataScopeInteractive is able to make the necessary adjustment because the Shapley value is recomputed after every repair.

In terms of speed, `DataScope` significantly outperforms TMC-based methods — in the order of $100\times$ to $1,000\times$ for models like logistic regression and up to $10,000\times$ for XGBoost. In terms of quality, `DataScope` is comparable to TMC-based methods, while DataScopeInteractive, in certain cases, dramatically outperforms `DataScope` and TMC-based methods. DataScopeInteractive achieves much better fairness (measured by equalized odds difference, lower the better) while maintaining similar, if not better, accuracy compared with other methods. When optimizing for fairness we can observe that sometimes non-interactive methods suffer in pipelines that use standard scalers. It might be possible that importance scores do not remain stable over the course of our data repair process. Because equalized odds difference is a non-trivial measure, even though it may work in the beginning of our process, it might mislead us in the wrong direction after some portion of the labels get repaired. As a result, being able to compute data importance frequently, which is enabled by our efficient algorithm, is crucial to effectively navigate and balance accuracy and fairness.

Scalability. We now evaluate the quality and speed of `DataScope` for larger training datasets. We test the runtime for various sizes of the training set ($10k$-$1M$), the validation set (100-$10k$), and the number of features (100-$1k$). As expected, the impact of training set size and validation set size is roughly linear. Furthermore, we see that even for large datasets, `DataScope` can compute Shapley scores in minutes.

When integrated into an intearctive data repair workflow, this could have a dramatic impact on the productivity of data scientists. We have clearly observed that Monte Carlo approaches do improve the efficiency of importance-based data debugging. However, given their lengthy runtime, one could argue that many users would likely prefer not to wait and consequently end up opting for the random repair approach. What `DataScope` offers is a viable alternative to random which is equally attainable while at the same time offering the significant efficiency gains provided by Shapley-based importance.

7 Related Work

Analyzing ML models. Understanding model predictions and handling problems when they arise has been an important area since the early days. In recent years, this topic has gained even more traction and is better known under the terms explainability and interpretability [1, 25, 21]. Typically, the goal is to understand why a model is making some specific prediction for a data example. Some approaches use surrogate models to produce explanations [58, 39]. In computer vision, saliency maps have gained prominence [72, 66]. Saliency maps can be seen as a type of a *feature importance* approach to explainability, although they also aim at interpreting the internals of a model. Other feature importance frameworks have also been developed [67, 16], and some even focus on Shapley-based feature importance [45].

Another approach to model interpretability can be referred to as *data importance* (i.e. using training data examples to explain predictions). This can have broader applications, including data valuation [52]. One important line of work expresses data importance in terms of *influence fucnctions* [9, 36, 35, 65]. Another line of work expresses data importance using Shapley values. Some apply Monte Carlo methods for efficiently computing it [20] and some take advantage of the KNN model [29, 28]. The KNN model has also been used for computing an entropy-based importance method targeted specifically for the data-cleaning application [34]. In general, all of the mentioned approaches focus on interpreting a single model, without the ability to efficiently analyze it in the context of an end-to-end ML pipeline.

Analyzing Data Processing Pipelines. For decades, the data management community has been studying how to analyze data importance for data-processing pipelines through various forms of query analysis. Some broader approaches include: causal analysis of interventions to queries [47], and reverse data management [46]. Methods also differ in terms of what is the target of their explanation, that is, with respect to what is the query output being explained. Some methods target queried relations [33]. Others target specific predicates that make up a query [60, 61]. Finally, some methods target specific tuples in input relations [48].

A prominent line of work employs *data provenance* as a means to analyze data processing pipelines [13]. *Provenance semiring* represents a theoretical framework for dealing with provenance [24]. This framework gives us as a theoretical foundation to develop algorithms for analyzing ML pipelines. This analysis typically requires us to operate in an exponentially large problem space. This can be made manageable by transforming provenance information to decision diagrams through a process known as *knowledge compilation* [27]. However, this framework is not guaranteed to lead us to tractable solutions. Some types of queries have been shown to be quite difficult [4, 3]. In this work, we demonstrate tractability of a concrete class of pipelines compried of both a set of feature extraction operators, as well as an ML model.

Recent research under the umbrella of "mlinspect" [22, 23, 63] details how to compute data provenance over end-to-end ML pipelines similar to the ones in the focus of this work, based on lightweight code instrumentation.

Joint Analysis of End-to-End Pipelines. Joint analysis of machine learning pipelines is a relatively novel, but nevertheless, an important field [54]. Systems such as Data X-Ray can debug data processing pipelines by finding groups of data errors that might have the same cause [69]. Some work has been done in the area of end-to-end pipeline compilation to tensor operations [50]. A notable piece of work leverages influence functions as a method for analyzing pipelines comprising of a model and a post-processing query [70]. This work also leverages data provenance as a key ingredient. In general, there are indications that data provenance is going to be a key ingredient of future ML systems [2, 63], something that our own system depends upon.

8 Conclusion

We present `Ease.ML/DataScope`, the first system that efficiently computes Shapley values of training examples over an *end-to-end* ML pipeline. Our core contribution is a novel algorithmic framework that computes Shapley value over a specific family of ML pipelines that we call *canonical pipelines*, connecting decades of research on relational data provenance and recent advancement of machine earning. For many subfamilies of canonical pipelines, computing Shapley value is in PTIME, contrasting the exponential complexity of computing Shapley value in general. We conduct extensive experiments illustrating different use cases and utilities. Our results show that `DataScope` is up to four orders of magnitude faster over state-of-the-art Monte Carlo-based methods, while being comparably, and often even more, effective in data debugging.

References

[1] A Adadi and M Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* 6 (2018), 52138–52160.

[2] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, et al. 2020. Cloudy with High Chance of DBMS: A 10-year Prediction for Enterprise-Grade ML. (2020).

[3] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. On the limitations of provenance for queries with difference. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 11)*.

[4] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for Aggregate Queries. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Athens, Greece) *(PODS '11)*. Association for Computing Machinery, New York, NY, USA, 153–164. `https://doi.org/10.1145/1989284.1989302`

[5] Sanjeev Arora and Boaz Barak. 2009. *Computational complexity: a modern approach*. Cambridge University Press.

[6] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebric decision diagrams and their applications. *Formal methods in system design* 10, 2 (1997), 171–206.

[7] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5, 1 (2014), 1–9.

[8] Solon Barocas, Moritz Hardt, and Arvind Narayanan. 2019. *Fairness and Machine Learning*. fairmlbook.org. `http://www.fairmlbook.org`.

[9] Samyadeep Basu, Xuchen You, and Soheil Feizi. 2020. On Second-Order Group Influence Functions for Black-Box Predictions. 119 (2020), 715–724.

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

[11] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. 28, 1 (2013), 115–123.

[12] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100, 8 (1986), 677–691.

[13] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer, 316–330.

[14] Marco Cadoli and Francesco M Donini. 1997. A survey on knowledge compilation. *AI Communications* 10, 3, 4 (1997), 137–150.

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

[16] Ian Covert, Scott Lundberg, and Su-In Lee. 2021. Explaining by removing: A unified framework for model explanation. *Journal of Machine Learning Research* 22, 209 (2021), 1–90.

[17] Frances Ding, Moritz Hardt, John Miller, and Ludwig Schmidt. 2021. Retiring Adult: New Datasets For Fair Machine Learning. *arXiv preprint arXiv:2108.04884* (2021).

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2755–2763.

[19] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Sen Yang, and Ji Liu. 2021. Bagua: scaling up distributed learning with system relaxations. *Proceedings VLDB Endowment* 15, 4 (Dec. 2021), 804–813.

[20] Amirata Ghorbani and James Zou. 2019. Data shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning*. PMLR, 2242–2251.

[21] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, 80–89.

[22] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. 2022. Data distribution debugging in machine learning pipelines. *The VLDB Journal* (2022), 1–24.

[23] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. Mlinspect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*. 2736–2739.

[24] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[25] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.* 51, 5 (Aug. 2018), 1–42.

[26] Moritz Hardt, Eric Price, Eric Price, and Nati Srebro. 2016. Equality of Opportunity in Supervised Learning. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/9d2682367c3935defcb1f9e247a97c0d-Paper.pdf

[27] Abhay Jha and Dan Suciu. 2011. Knowledge compilation meets database theory: Compiling queries to decision diagrams. In *ACM International Conference Proceeding Series*. 162–173. https://doi.org/10.1145/1938551.1938574

[28] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nezihe Merve Gurel, Bo Li, Ce Zhang, Costas J Spanos, and Dawn Song. 2019. Efficient Task-Specific Data Valuation for Nearest Neighbor Algorithms. In *VLDB*.

[29] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nick Hynes, Nezihe Merve Gürel, Bo Li, Ce Zhang, Dawn Song, and Costas J Spanos. 2019. Towards efficient data valuation based on the shapley value. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 1167–1176.

[30] Ruoxi Jia, Xuehui Sun, Jiacen Xu, Ce Zhang, Bo Li, and Dawn Song. 2021. Scalability vs. Utility: Do We Have to Sacrifice One for the Other in Data Importance Quantification? *CVPR* (2021).

[31] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous {GPU/CPU} Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[32] Thorsten Joachims. 1996. *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization.* Technical Report. Carnegie-mellon univ pittsburgh pa dept of computer science.

[33] Bhargav Kanagal, Jian Li, and Amol Deshpande. 2011. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11* (Athens, Greece). ACM Press, New York, New York, USA.

[34] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (nov 2020), 255–267. `https://doi.org/10.14778/3430915.3430917`

[35] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. *International conference on machine learning* 70 (2017), 1885–1894.

[36] Pang Wei W Koh, Kai-Siang Ang, Hubert Teo, and Percy S Liang. 2019. On the accuracy of influence functions for measuring group effects. *Advances in neural information processing systems* 32 (2019).

[37] Ron Kohavi et al. 1996. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid.. In *Kdd*, Vol. 96. 202–207.

[38] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. [n.d.]. ActiveClean: Interactive data cleaning for statistical modeling. `http://www.vldb.org/pvldb/vol9/p948-krishnan.pdf`. Accessed: 2021-4-7.

[39] Sanjay Krishnan and Eugene Wu. 2017. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2Nd workshop on human-in-the-loop data analytics*. 1–6.

[40] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. 1996. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comput.* 45, 2 (1996), 247–255.

[41] C. Y. Lee. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal* 38, 4 (1959), 985–999. `https://doi.org/10.1002/j.1538-7305.1959.tb01585.x`

[42] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *36th IEEE International Conference on Data Engineering (ICDE 2020)(virtual)*.

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings VLDB Endowment* 13, 12 (Aug. 2020), 3005–3018.

[44] Ji Liu, Ce Zhang, and Others. 2020. Distributed Learning Systems with First-Order Methods. *Foundations and Trends® in Databases* 9, 1 (2020), 1–100.

[45] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[46] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. 2011. Reverse Data Management. *Proc. VLDB Endow.* 4, 12 (aug 2011), 1490–1493. `https://doi.org/10.14778/3402755.3402803`

[47] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *Proc. VLDB Endow.* 7, 13 (aug 2014), 1715–1716. `https://doi.org/10.14778/2733004.2733070`

[48] Alexandra Meliou and Dan Suciu. 2012. Tiresias: The Database Oracle for How-to Queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 337–348. `https://doi.org/10.1145/2213836.2213875`

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

[50] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 899–917.

[51] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[52] Jian Pei. 2020. Data Pricing–From Economics to Data Science. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3553–3554.

[53] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. [n.d.]. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment* 13, 11 ([n. d.]).

[54] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1723–1726. `https://doi.org/10.1145/3035918.3054782`

[55] Fotis Psallidas, Yiwen Zhu, Bojan Karlaš, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, et al. 2019. Data Science through the looking glass and what we found there. *arXiv preprint arXiv:1912.09536* (2019).

[56] Alexander Ratner, Dan Alistarh, Gustavo Alonso, David G Andersen, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Eric Chung, Bill Dally, and Others. 2019. SysML: The New Frontier of Machine Learning Systems. (2019).

[57] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proceedings VLDB Endowment* 11, 3 (Nov. 2017), 269–282.

[58] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.

[59] Peter Ross. 1997. Generalized hockey stick identities and n-dimensional blockwalking. *The College Mathematics Journal* 28, 4 (1997), 325.

[60] Sudeepa Roy, Laurel Orr, and Dan Suciu. 2015. Explaining Query Answers with Explanation-Ready Databases. *Proc. VLDB Endow.* 9, 4 (dec 2015), 348–359. `https://doi.org/10.14778/2856318.2856329`

[61] Sudeepa Roy and Dan Suciu. 2014. A Formal Approach to Finding Explanations for Database Queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1579–1590. `https://doi.org/10.1145/2588555.2588578`

[62] Scott Sanner and David McAllester. 2005. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *IJCAI*, Vol. 2005. 1384–1390.

[63] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Olivier Sprangers, Bojan Karlaš, and Ce Zhang. 2022. Screening Native ML Pipelines with "ArgusEyes". *CIDR* (2022).

[64] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. (Feb. 2018). arXiv:1802.05799 [cs.LG]

[65] Boris Sharchilev, Yury Ustinovskiy, Pavel Serdyukov, and Maarten Rijke. 2018. Finding influential training samples for gradient boosted decision trees. In *International Conference on Machine Learning*. PMLR, 4577–4585.

[66] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International conference on machine learning*. PMLR, 3145–3153.

[67] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International conference on machine learning*. PMLR, 3319–3328.

[68] L G Valiant. 1979. The complexity of computing the permanent. *Theor. Comput. Sci.* 8, 2 (Jan. 1979), 189–201.

[69] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1231–1245. `https://doi.org/10.1145/2723372.2750549`

[70] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.

[71] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv:cs.LG/1708.07747 [cs.LG]

[72] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[73] Yingbo Zhou, Utkarsh Porwal, Ce Zhang, Hung Q Ngo, Xuanlong Nguyen, Christopher Ré, and Venu Govindaraju. 2014. Parallel Feature Selection Inspired by Group Testing. In *Advances in Neural Information Processing Systems 27*, Z Ghahramani, M Welling, C Cortes, N D Lawrence, and K Q Weinberger (Eds.). Curran Associates, Inc., 3554–3562.

[74] Barret Zoph and Quoc V Le. 2016. Neural Architecture Search with Reinforcement Learning. (Nov. 2016). arXiv:1611.01578 [cs.LG]

A Implementation

We integrate[1] our Shapley value computation approach into the *ArgusEyes* [63] platform. ArgusEyes leverages the *mlinspect* [22, 23] library to execute and instrument Python-based ML pipelines that combine code from popular data science libraries such as pandas, scikit-learn and keras. During execution, mlinspect extracts a "logical query plan" of the pipeline operations, modeling them as a dataflow computation with relational operations (e.g., originating from pandas) and ML-specific operations (e.g., feature encoders) which are treated as extended projections. Furthermore, ArgusEyes captures and materialises the relational inputs of the pipeline (e.g., CSV files read via pandas) and the numerical outputs of the pipeline (e.g., the labels and feature matrix for the training data).

The existing abstractions in mlinspect map well to our pipeline model from subsection 3.2, where we postulate that a pipeline maps a set of relational inputs $\mathcal{D}_{tr} = \{\mathcal{D}_e, \mathcal{D}_{s_1}, \dots, \mathcal{D}_{s_k}\}$ to vectorised labeled training examples $\{z_i = (x_i, y_i)\}$ for a subsequent ML model. Furthermore, mlinspect has built-in support for computing polynomials for the why-provenance [24] of its outputs. This provenance information allows us to match a pipeline to its canonical counterpart as defined in subsection 4.2 and apply our techniques from section 5 to compute Shapley values for the input data.

Listing 2 depicts a simplified implementation of data valuation for map pipelines in ArgusEyes. As discussed, the ArgusEyes platform executes the pipeline and extracts and materialises its relational `inputs`, its numerical `outputs` and the corresponding `provenance` polynomials. First, we compute Shapley values for the labeled rows $\{z_i = (x_i, y_i)\}$ of the training feature matrix produced by the pipeline, based on our previously published efficient KNN Shapley method [28] (lines 10-13). Next, we retrieve the materialised relational input `fact_table` for $\mathcal{D}_e$ (the "fact table" in cases of multiple inputs in a star schema), as well as the provenance polynomials `provenance_fact_table` for $\mathcal{D}_e$ and `provenance_X_train` for the training samples $\{z_i\}$ (lines 15-17). Finally, we annotate the rows of the `fact_table` with a new column `shapley_value` where we store the computed Shapley value for each input row. We assign the values by matching the provenance polynomials of $\mathcal{D}_e$ and $\{z_i = (x_i, y_i)\}$ (lines 19 24).

```
1  class DataValuation():
2   def compute(self, inputs, outputs, provenance):
3     # Access captured pipeline outputs
4     # (train/test features and labels)
5     X_train = outputs[Output.X_TRAIN]
6     X_test = outputs[Output.X_TEST]
7     y_train = outputs[Output.Y_TRAIN]
8     y_test = outputs[Output.Y_TEST]
9     # Compute Shapley values via KNN approximation
10    shapley_values = self._shapley_knn(
11      X_train, y_train, self.k,
12      X_test[:self.num_test_samples, :],
13      y_test[:self.num_test_samples, :])
14    # Input data and provenance
15    fact_table = inputs[Input.FACT_TABLE]
16    provenance_fact_table = provenance[Input.FACT_TABLE]
17    provenance_X_train = provenance[Output.X_TRAIN]
18    # Annotate input tuples with their Shapley values
19    for polynomial, shapley_value in
20       zip(provenance_X_train, shapley_values):
21      for entry in polynomial:
22        if entry.input_id == fact_table.input_id:
23          row = provenance_fact_table.row_id(entry.tuple_id)
24          fact_table.at[row, 'shapley_value'] = shapley_value
```

Listing 2: Simplified implementation of data valuation for map pipelines in ArgusEyes.

We provide an executable end-to-end example for data valuation over complex pipelines in the form of a jupyter notebook at https://github.com/schelterlabs/arguseyes/blob/datascope/arguseyes/example_pipelines/demo-shapley-pipeline.ipynb, which shows how to leverage ArgusEyes to identify mislabeled samples in a computer vision pipeline[2] on images of fashion products.

[1] https://github.com/schelterlabs/arguseyes/blob/datascope/arguseyes/refinements/_data_valuation.py
[2] https://github.com/schelterlabs/arguseyes/blob/datascope/arguseyes/example_pipelines/product-images.py

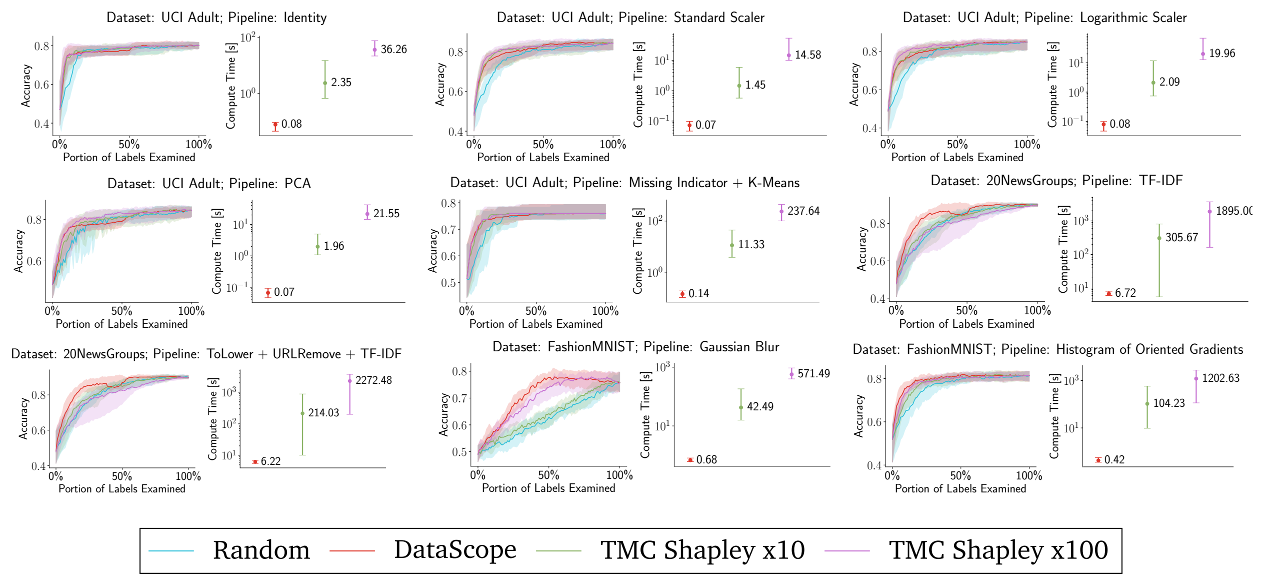

Figure 8: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for accuracy. The model is logistic regression.

B Additional Experiments

Figure 8 to Figure 12; Figure 13 to Figure 17

C Proofs and Details

C.1 Proof of Theorem 5.1

Model Counting for ADD's. We start off by proving that Equation 4 correctly performs model counting.

Lemma C.1. *For a given node $n \in \mathcal{N}$ of an ADD and a given value $e \in \mathcal{E}$, Equation 4 correctly computes* $\text{count}_e(n)$ *which returns the number of assignments $v \in \mathcal{V}_A$ such that* $\text{eval}_v(n) = e$. *Furthermore, when computing* $\text{count}_e(n)$ *for any $n \in \mathcal{N}$, the number of computational steps is bounded by $O(|\mathcal{N}| \cdot |\mathcal{E}|)$.*

Proof. We will prove this by induction on the structure of the recursion.

(Base case.) Based on Equation 2, when $n = \boxdot$ we get $\text{eval}_v(n) = 0$ for all v. Furthermore, when $n = \boxdot$, the set $\mathcal{V}_A[a_{>\pi(a(n))} = 0]$ contains only one value assignment with all variables set to zero. Hence, the model count will equal to 1 only for $e = 0$ and it will be 0 otherwise, which is reflected in the base cases of Equation 4.

(Inductive step.) Because our ADD is ordered and full, both $c_L(n)$ and $c_H(n)$ are associated with the same variable, which is the predecessor of $a(n)$ in the permutation π. Based on this and the induction hypothesis, we can assume that

$$\begin{aligned}
\text{count}_{e-w_L(n)}(c_L(n)) &= \left|\left\{ v \in \mathcal{V}_{A[\leq a(c_L(n))]} \mid \text{eval}_v(c_L(n)) = e - w_L(n) \right\}\right| \\
\text{count}_{e-w_H(n)}(c_H(n)) &= \left|\left\{ v \in \mathcal{V}_{A[\leq a(c_H(n))]} \mid \text{eval}_v(c_H(n)) = e - w_H(n) \right\}\right|
\end{aligned} \tag{23}$$

We would like to compute $\text{count}_e(n)$ as defined in Equation 3. It computes the size of a set defined over possible value assignments to variables in $A[\leq a(n)]$. The set of value assignments can be partitioned into

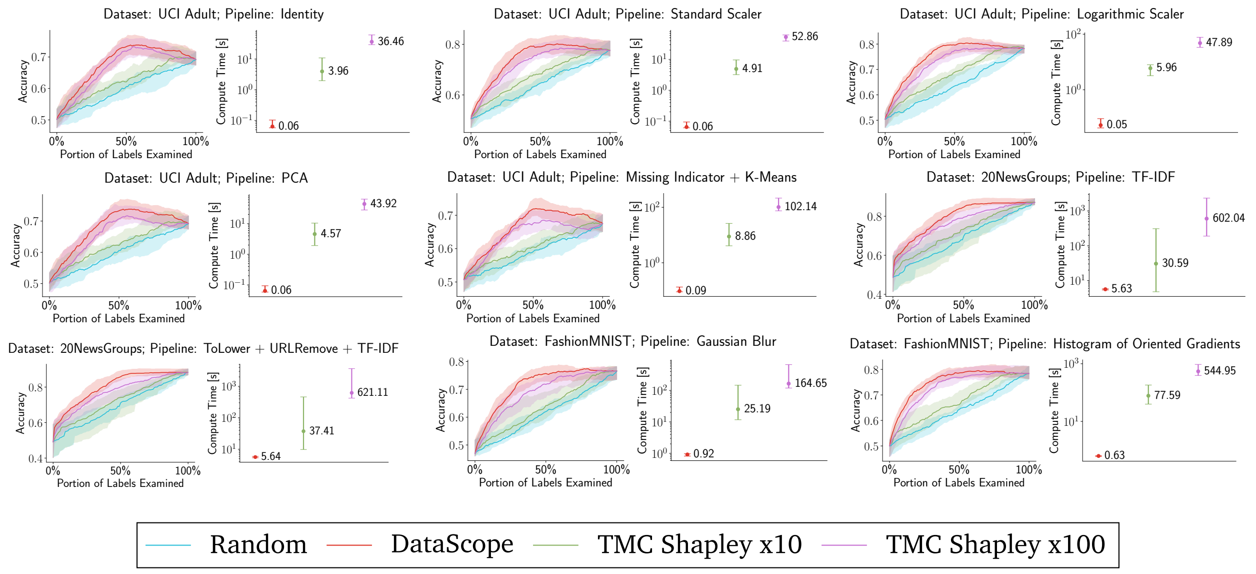

Figure 9: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for accuracy. The model is K-nearest neighbor.

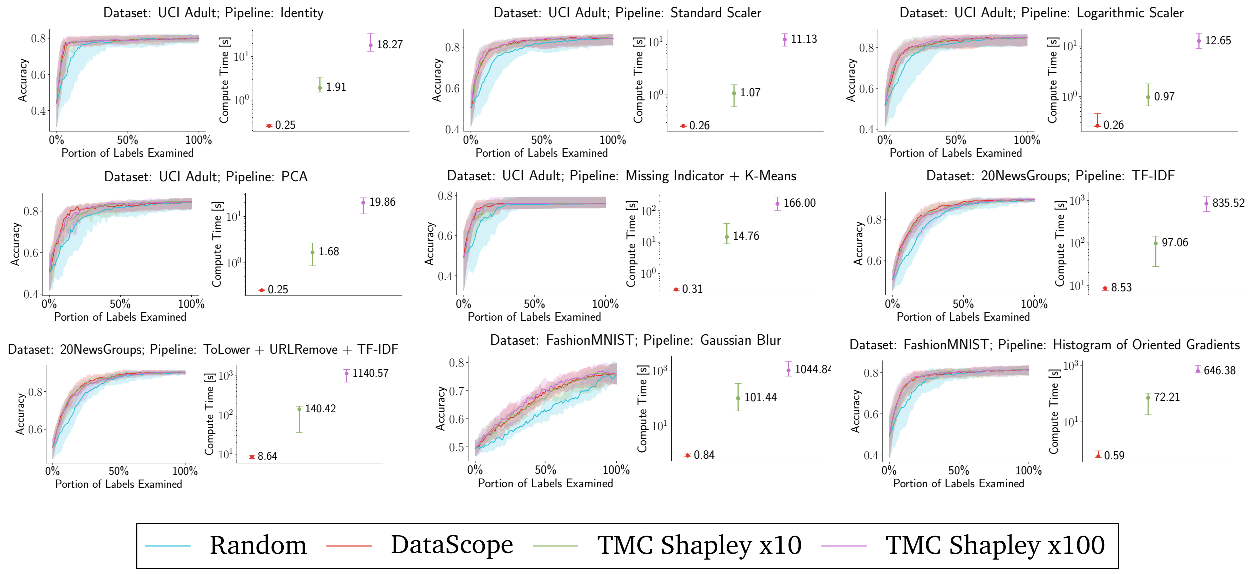

Figure 10: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for accuracy. The model is logistic regression.

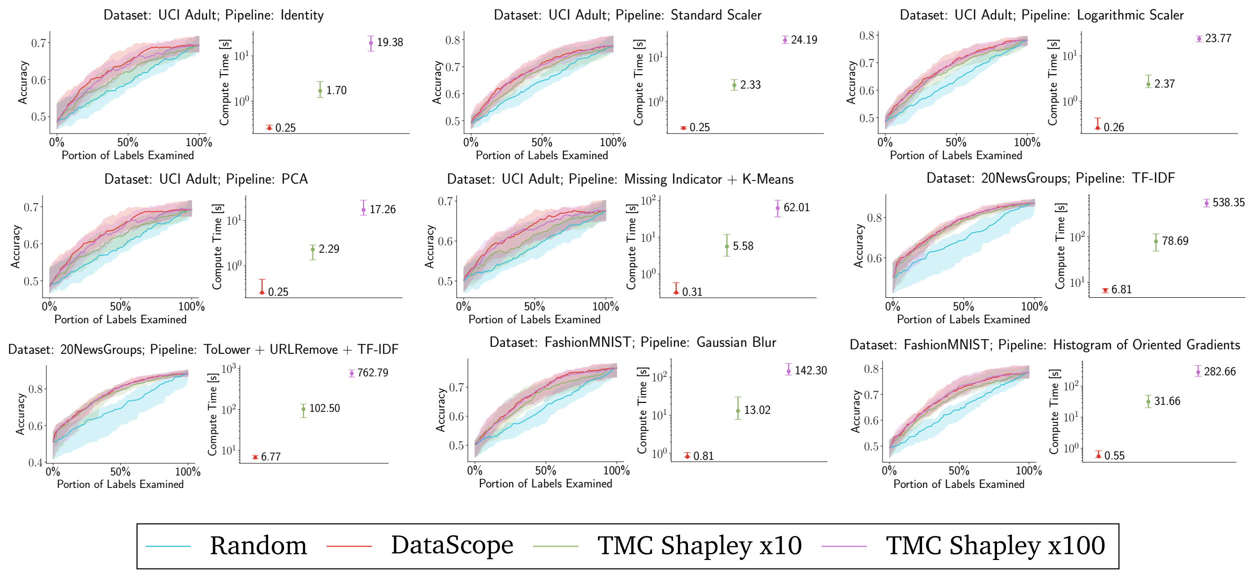
Figure 11: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for accuracy. The model is K-nearest neighbor.

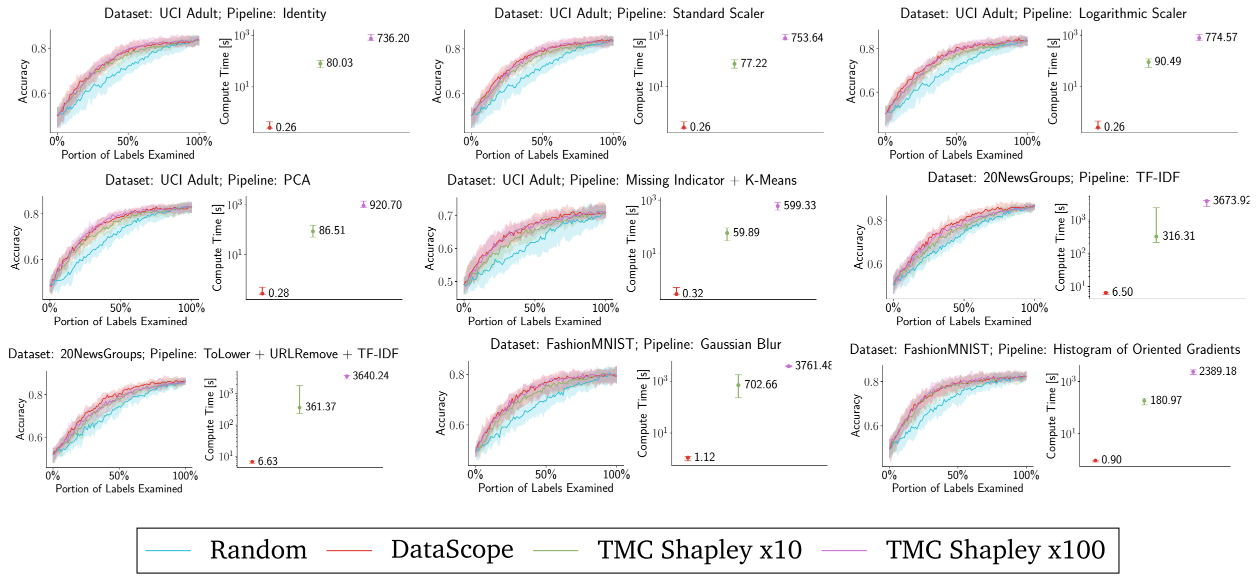
Figure 12: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for accuracy. The model is XGBoost.

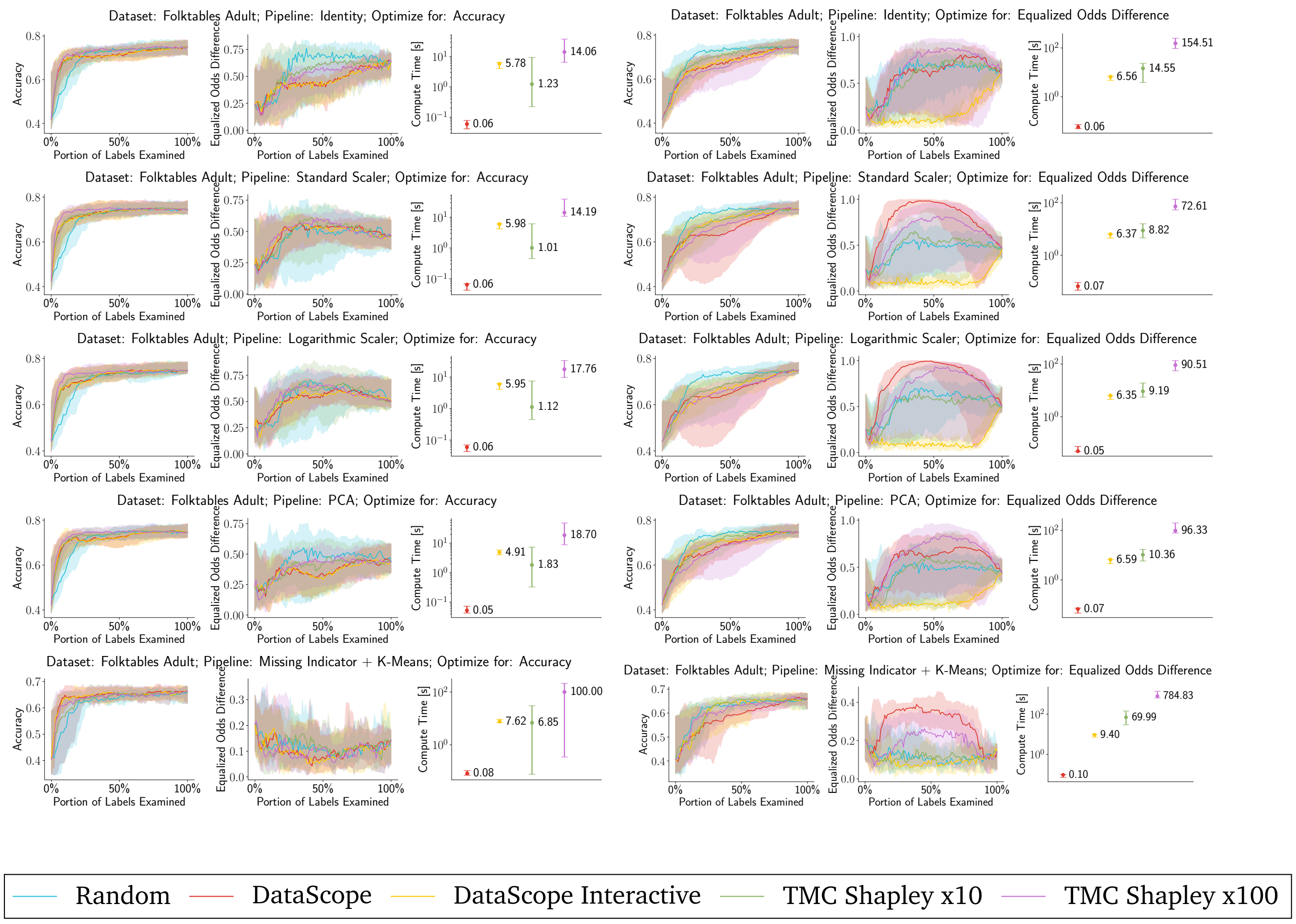

Figure 13: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for fairness. The model is logistic regression.

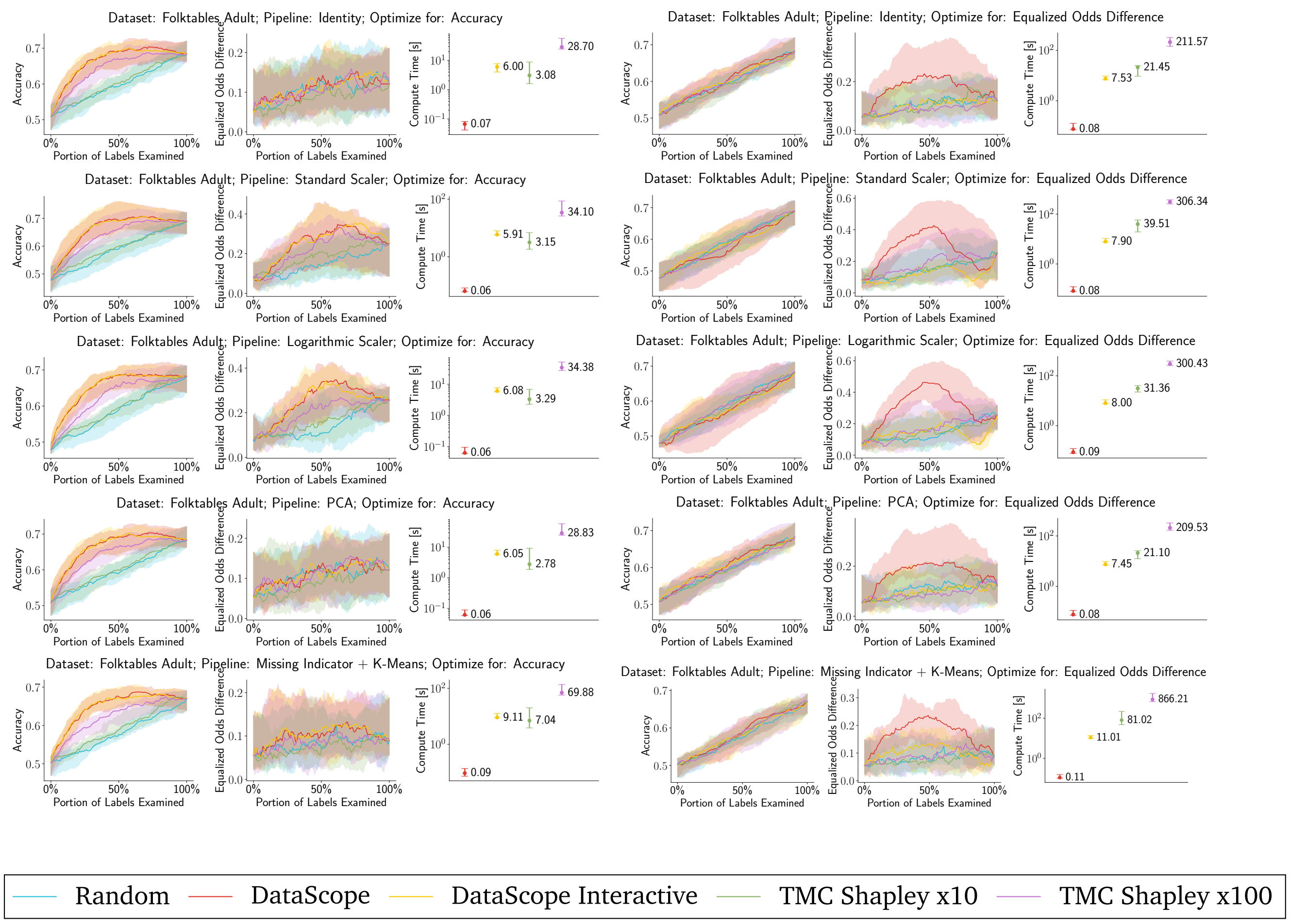

Figure 14: Label Repair experiment results over various combinations of datasets (1k samples) and map pipelines. We optimize for fairness. The model is K-nearest neighbor.

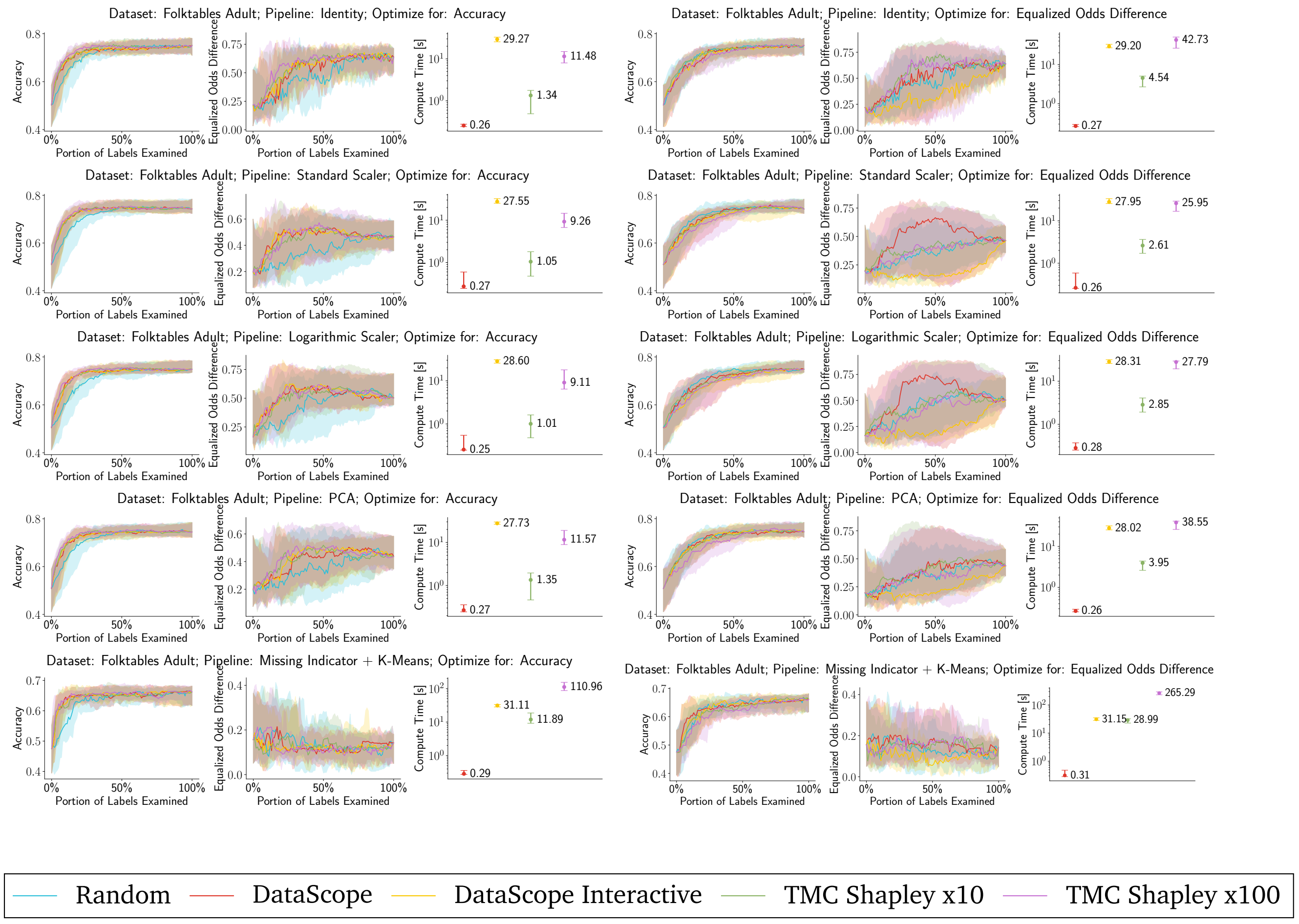

Figure 15: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for fairness. The model is logistic regression.

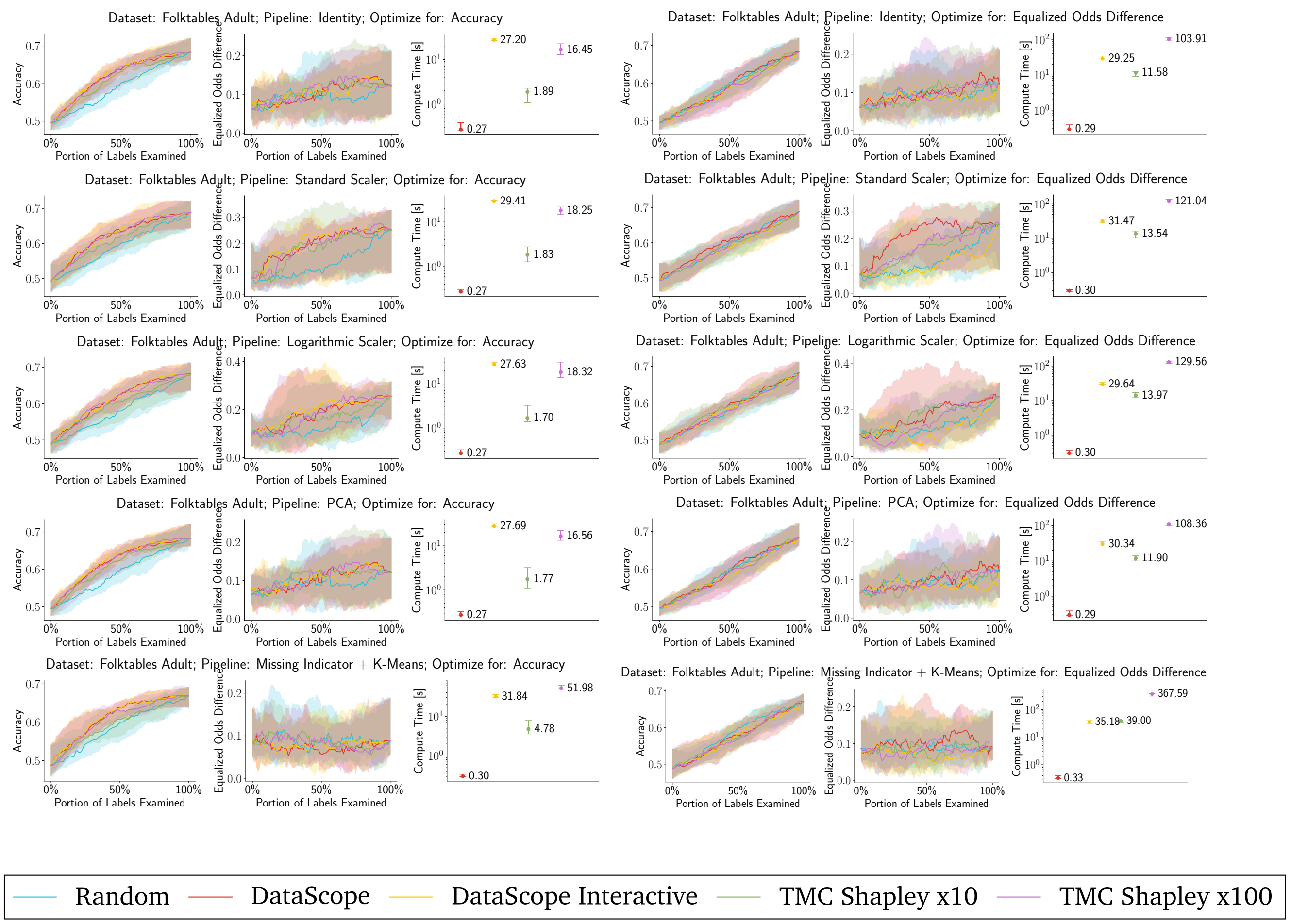

Figure 16: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for fairness. The model is K-nearest neighbor.

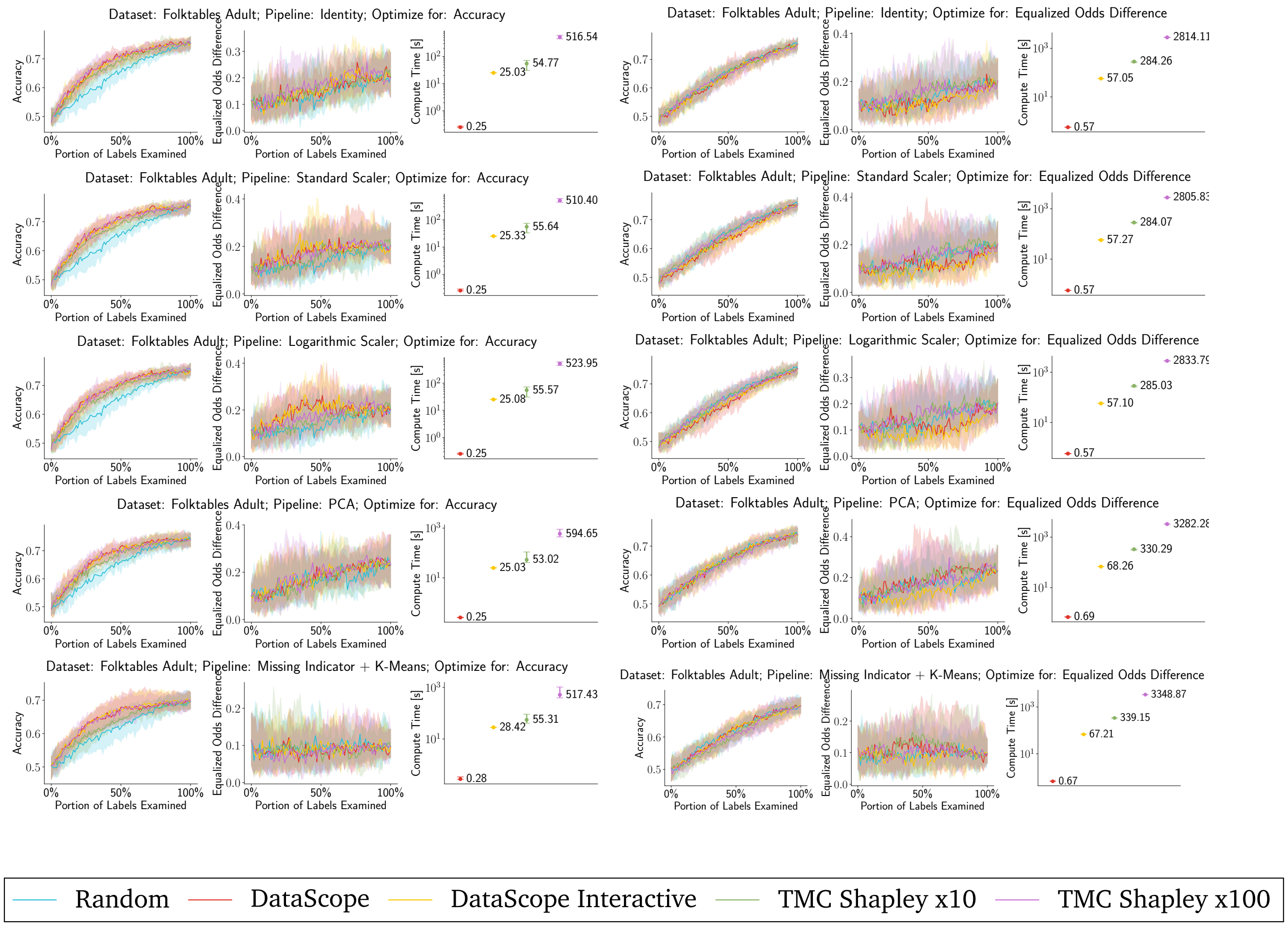

Figure 17: Label Repair experiment results over various combinations of datasets (1k samples) and fork pipelines. We optimize for fairness. The model is XGBoost.

two distinct sets: one where $a(n) \leftarrow 0$ and one where $a(n) \leftarrow 1$. We thus obtain the following expression:

$$\begin{aligned} \text{count}_e(n) :=& \left|\left\{v \in \mathcal{V}_{A[\leq a(n)]}\left[a(n) \leftarrow 0\right] \mid \text{eval}_v(n) = e\right\}\right| \\ &+ \left|\left\{v \in \mathcal{V}_{A[\leq a(n)]}\left[a(n) \leftarrow 1\right] \mid \text{eval}_v(n) = e\right\}\right| \end{aligned} \tag{24}$$

Based on Equation 2, we can transform the $\text{eval}_v(n)$ expressions as such:

$$\begin{aligned} \text{count}_e(n) :=& \left|\left\{v \in \mathcal{V}_{A[\leq a(c_L(n))]} \mid w_L(n) + \text{eval}_v(c_L(n)) = e\right\}\right| \\ &+ \left|\left\{v \in \mathcal{V}_{A[\leq a(c_L(n))]} \mid w_H(n) + \text{eval}_v(c_H(n)) = e\right\}\right| \end{aligned} \tag{25}$$

Finally, we can notice that the set size expressions are equivalent to those in Equation 23. Therefore, we can obtain the following expression:

$$\text{count}_e(n) := \text{count}_{e-w_L(n)}(c_L(n)) + \text{count}_{e-w_H(n)}(c_H(n)) \tag{26}$$

which is exactly the recursive step in Equation 4. This concludes our inductive proof and we move onto proving the complexity bound.

(Complexity.) This is trivially proven by observing that since count has two arguments, we can maintain a table of results obtained for each $n \in \mathcal{N}$ and $e \in \mathcal{E}$. Therefore, we know that we will never need to perform more than $O(|\mathcal{N}| \cdot |\mathcal{E}|)$ invocations of $\text{count}_e(n)$.

□

ADD Construction. Next, we prove that the size of an ADD resulting from *diagram summation* as defined in subsection 2.2 is linear in the number of variables.

The size of the diagram resulting from a sum of two diagrams with node sets $\mathcal{N}_1$ and $\mathcal{N}_2$ can be loosely bounded by $O(|\mathcal{N}_1| \cdot |\mathcal{N}_2|)$ assuming that its nodes come from a combination of every possible pair of operand nodes. However, given the much more narrow assumptions we made in the definition of the node sum operator, we can make this bound considerably tighter. For this we define the *diameter* of an ADD as the maximum number of nodes associated with any single variable. Formally we can write:

$$\text{diam}(\mathcal{N}) := \max_{a_i \in A} \left|\{n \in \mathcal{N} : a(n) = a_i\}\right| \tag{27}$$

We can immediately notice that the size of any ADD with set of nodes $\mathcal{N}$ and variables A is bounded by $O(|A| \cdot \text{diam}(\mathcal{N}))$. We can use this fact to prove a tighter bound on the size of an ADD resulting from a sum operation:

Lemma C.2. *Given two full ordered ADD's with nodes $\mathcal{N}_1$ and $\mathcal{N}_2$, noth defined over the set of variables A, the number of nodes in $\mathcal{N}_1 + \mathcal{N}_2$ is bounded by $O(|A| \cdot \text{diam}(\mathcal{N}_1) \cdot \text{diam}(\mathcal{N}_2))$.*

Proof. It is sufficient to show that $\text{diam}(\mathcal{N}_1 + \mathcal{N}_2) = O(\text{diam}(\mathcal{N}_1) \cdot \text{diam}(\mathcal{N}_2))$. This is a direct consequence of the fact that for full ordered ADD's the node sum operator is defined only for nodes associated with the same variable. Since the only way to produce new nodes is by merging one node in $\mathcal{N}_1$ with one node in $\mathcal{N}_2$, and given that we can merge nodes associated with the same variable, the number of nodes associated with the same variable in the resulting ADD equals the product of the corresponding number of nodes in the constituent ADD's. Since the diameter is simply the upper bound of the number of nodes associated with any single variable, the same upper bound in the resulting ADD cannot be larger than the product of the upper bounds of constituent nodes. □

Computing the Oracle using ADD's. Finally, we prove the correctness of Theorem 5.1.

Lemma C.3. *Given an Additive Decision diagram with root node $n_{t,t'}$ that computes the Boolean function $\phi_{t,t'}(v)$ as defined in Equation 17, the counting oracle $\omega_{t,t'}(\alpha,\gamma,\gamma')$ defined in Equation 15 can be computed as:*

$$\omega_{t,t'}(\alpha,\gamma,\gamma') := \mathrm{count}_{(\alpha,\gamma,\gamma')}(n_{t,t'}) \tag{28}$$

Proof. Let us define $\mathcal{D}[\geq_\sigma t] \subseteq \mathcal{D}$ as a set of tuples with similarity higher or equal than that of t, formally $\mathcal{D}[\geq_\sigma t] := \{t' \in \mathcal{D} : \sigma(t') \geq \sigma(t)\}$. Similarly to $\mathcal{D}$, the semantics of $\mathcal{D}[\geq_\sigma t]$ is also that of a set of possible candidate sets. Given a value assignment v, we can obtain $\mathcal{D}[\geq_\sigma t][v]$ from $\mathcal{D}[v]$. For convenience, we also define $\mathcal{D}[\geq_\sigma t][y]$ as a subset of $\mathcal{D}[\geq_\sigma t]$ with only tuples that have label y. Given these definitions, we can define several equivalences. First, for top_K we have:

$$\Big(t = \mathrm{top}_K \mathcal{D}[v]\Big) \iff \Big(t \in \mathcal{D}[v] \wedge \big|\mathcal{D}[\geq_\sigma t][v]\big| = K\Big) \tag{29}$$

In other words, for t to be the tuple with the K-th highest similarity in $\mathcal{D}[v]$, it needs to be a member of $\mathcal{D}[v]$ and the number of tuples with similarity greater or equal to t has to be exactly K. Similarly, we can define the equivalence for tally_t:

$$\Big(\gamma = \mathrm{tally}_t \mathcal{D}[v]\Big) \iff \Big(\forall y \in \mathcal{Y}, \gamma_y = \big|\mathcal{D}[\geq_\sigma t][y][v]\big|\Big) \tag{30}$$

This is simply an expression that partitions the set $\mathcal{D}[\geq_\sigma t][v]$ based on y and tallies them up. The next step is to define an equivalence for $(t = \mathrm{top}_K \mathcal{D}[v]) \wedge (\gamma = \mathrm{tally}_t \mathcal{D}[v])$. We can notice that since $|\gamma| = K$, if we have $(\forall y \in \mathcal{Y}, \gamma_y = |\mathcal{D}[\geq_\sigma t][y][v]|)$ then we can conclude that $(|\mathcal{D}[\geq_\sigma t][v]| = K)$ is redundant. Hence, we can obtain:

$$\Big(t = \mathrm{top}_K \mathcal{D}[v]\Big) \wedge \Big(\gamma = \mathrm{tally}_t \mathcal{D}[v]\Big) \iff \Big(t \in \mathcal{D}[v]\Big) \wedge \Big(\forall y \in \mathcal{Y}, \gamma_y = \big|\mathcal{D}[\geq_\sigma t][y][v]\big|\Big) \tag{31}$$

According to Equation 30, we can reformulate the right-hand side of the above equivalence as:

$$\Big(t = \mathrm{top}_K \mathcal{D}[v]\Big) \wedge \Big(\gamma = \mathrm{tally}_t \mathcal{D}[v]\Big) \iff \Big(t \in \mathcal{D}[v]\Big) \wedge \Big(\gamma = \mathrm{tally}_t \mathcal{D}[v]\Big) \tag{32}$$

We can construct a similar expression for t' and $v[a_i = 1]$ so we cover four out of five predicates in Equation 15. The remaining one is simply the support of the value assignment v which we will leave intact. This leads us with the following equation for the counting oracle:

$$\begin{aligned} \omega_{t,t'}(\alpha,\gamma,\gamma') := \sum_{v \in \mathcal{V}_A[a_i \leftarrow 0]} & \mathbb{1}\{\alpha = |\mathrm{supp}(v)|\} \\ & \mathbb{1}\{t \in f(\mathcal{D}[v])\}\mathbb{1}\{t' \in f(\mathcal{D}[v[a_i \leftarrow 1]])\} \\ & \mathbb{1}\{\gamma = \mathrm{tally}_t \mathcal{D}[v]\}\mathbb{1}\{\gamma = \mathrm{tally}_t \mathcal{D}[v[a_i \leftarrow 1]]\} \end{aligned} \tag{33}$$

We can use the Boolean function $\phi_{t,t'}(v)$ in Equation 17 to simplify the above equation. Notice that the conditions $t \in f(\mathcal{D}[v])$ and $t' \in f(\mathcal{D}[v[a_i \leftarrow 1]])$ are embedded in the definition of $\phi_{t,t'}(v)$ which will return ∞ if those conditions are not met. When the conditions are met, $\phi_{t,t'}(v)$ returns exactly the same triple (α,γ,γ'). Therefore it is safe to replace the five indicator functions in the above formula with a single one as such:

$$\omega_{t,t'}(\alpha,\gamma,\gamma') := \sum_{v \in \mathcal{V}_A[a_i \leftarrow 0]} \mathbb{1}\{(\alpha,\gamma,\gamma') = \phi_{t,t'}(v)\} \tag{34}$$

Given our assumption that $\phi_{t,t'}(v)$ can be represented by an ADD with a root node $n_{t,t'}$, the above formula is exactly the model counting operation:

$$\omega_{t,t'}(\alpha,\gamma,\gamma') := \mathrm{count}_{(\alpha,\gamma,\gamma')}(n_{t,t'}) \tag{35}$$

□

Theorem C.4. *If we can represent the Boolean function $\phi_{t,t'}(v)$ defined in Equation 17 with an Additive Decision Diagram of size polynomial in $|\mathcal{D}|$ and $|f(\mathcal{D})|$, then we can compute the counting oracle $\omega_{t,t'}$ in time polynomial in $|\mathcal{D}|$ and $|f(\mathcal{D})|$.*

Proof. This theorem follows from the two previously proved lemmas: Theorem C.1 and Theorem C.3. Namely, as a result of Theorem C.3 we claim that model counting of the Boolean function $\phi_{t,t'}(v)$ is equivalent to computing the oracle result. On top of that, as a result of Theorem C.1 we know that we can perform model counting in time linear in the size of the decision diagram. Hence, if our function $\phi_{t,t'}(v)$ can be represented with a decision diagram of size polynomial in the size of data, then we can conclude that computing the oracle result can be done in time polynomial in the size of data. □

C.2 Proof of corollary 5.1.1

Proof. This follows from the observation that in Algorithm 1, each connected component A_C will be made up from one variable corresponding to the dimension table and one or more variables corresponding to the fact table. Since the fact table variables will be categorized as "leaf variables", the expression $A_C \setminus A_L$ in Line 13 will contain only a single element – the dimension table variable. Consequently, the ADD tree in $\mathcal{N}'$ will contain a single node. On the other side, the $A_C \cap A_L$ expression will contain all fact table variables associated with that single dimension table variable. That chain will be added to the ADD tree two times for two outgoing branches of the single tree node. Hence, the ADD segment will be made up of two fact table variable chains stemming from a single dimension table variable node. There will be $O(|\mathcal{D}_D|)$ partitions in total. Given that the fact table variables are partitioned, the cumulative size of their chains will be $O(|\mathcal{D}_F|)$. Therefore, the total size of the ADD with all partitions joined together is bounded by $O(|\mathcal{D}_D|+|\mathcal{D}_F|) = O(N)$.

Given fact and combining it with Theorem 5.1 we know that the counting oracle can be computed in time $O(N)$ time. Finally, given Theorem 4.1 and the structure of Equation 16 we can observe that the counting oracle is invoked $O(N^3)$ times. As a result, we can conclude that the total complexity of computing the Shapley value is $O(N^4)$. □

C.3 Proof of corollary 5.1.2

Proof. The key observation here is that, since all provenance polynomials contain only a single variable, there is no interdependency between them, which means that the connected components returned in Line 12 of Algorithm 1 will each contain a single variable. Therefore, the size of the resulting ADD will be $O(N)$. Consequently, similar to the proof of the previous corollary, the counting oracle can be computed in time $O(N)$ time. In this case, the size of the output dataset is $O(M)$ which means that Equation 16 willinvoke the oracle $O(M^2N)$ times. Therefore, the total time complexity of computing the Shapley value will be $O(M^2N^2)$. □

C.4 Proof of corollary 5.1.3

Proof. There are two arguments we need to make which will result in the reduction of complexity compared to fork pipelines. The first argument is that, given that each variable can appear in the provenance polynomial of at most one tuple, having its value set to 1 can result in either zero or one tuple contributing to the top-K tally. It will be one if that tuple is more similar than the boundary tuple t and it will be zero if it is less similar. Consequently, our ADD will have a chain structure with high-child increments being either 0 or 1. If we partition the ADD into two chains, one with all increments 1 and another with all increments 0, then we end up with two uniform ADD's. As shown in Equation 5, model counting of uniform ADD's can be achieved in constant time. The only difference here is that, since we have to account for the support size each model, computing the oracle $\omega_{t,t'}(\alpha, \gamma, \gamma')$ for a given α will require us to account for different possible ways to split α across the two ADD's. However, since the tuple t needs to be the boundary tuple, which means it is the K-th most similar, there need to be exactly $K-1$ variables from the ADD with increments 1 that can be set to 1. This gives us a single possible distribution of α across two ADD's. Hence, the oracle can be computed in constant time.

As for the second argument, we need to make a simple observation. For map pipelines, given a boundary tuple t and a tally vector γ corresponding to the variable a_i being assigned the value 0, we know that setting this variable to 1 can introduce at most one tuple to the top-K. That could only be the single tuple associated with a_i. If this tuple has a lower similarity score than t, there will be no change in the top-K. On the other side, if it has a higher similarity, then it will become part of the top-K and it will evict exactly t from it. Hence, there is a unique tally vector γ' resulting from a_i being assigned the value 1. This means that instead of computing the counting oracle $\omega_{t,t'}(\alpha, \gamma, \gamma')$, we can compute the oracle $\omega_t(\alpha, \gamma)$. This means that, in Equation 16 we can eliminate the iteration over t' which saves us an order of $O(N)$ in complexity.

As a result, Equation 16 will make $O(N^2)$ invocations to the oracle which can be computed in constant time. Hence, the final complexity of computing the Shapley value will be $O(N^2)$. □

C.5 Proof of corollary 5.1.4

Proof. We start off by plugging in the oracle definition from Equation 22 into the Shapley value computation Equation 16:

$$\begin{aligned}\varphi_i = \frac{1}{N} \sum_{t,t' \in f(\mathcal{D})} \sum_{\alpha=1}^{N} \binom{N-1}{\alpha}^{-1} \sum_{\gamma,\gamma' \in \Gamma} u_\Delta(\gamma, \gamma') &\binom{|\{t'' \in \mathcal{D} \ : \ \sigma(t'') < \sigma(t)\}|}{\alpha} \\ &\mathbb{1}\{p(t') = a_i\} \\ &\mathbb{1}\{\gamma = \Gamma_{y(t)}\}\mathbb{1}\{\gamma' = \Gamma_{y(t')}\}\end{aligned} \tag{36}$$

As we can see, the oracle imposes hard constraints on the tuple t' and tally vectors γ and γ'. We will replace the tally vectors with their respective constants and the tuple t' we will denote as t_i because it is the only tuple associated with a_i. Because of this, we can remove the sums that iterate over them:

$$\varphi_i = \frac{1}{N} \sum_{t \in f(\mathcal{D})} \sum_{\alpha=1}^{N} \binom{N-1}{\alpha}^{-1} u_\Delta(\Gamma_{y(t)}, \Gamma_{y(t_i)}) \binom{|\{t'' \in \mathcal{D} \ : \ \sigma(t'') < \sigma(t)\}|}{\alpha} \tag{37}$$

We could significantly simplify this equation by assuming the tuples in $f(\mathcal{D})$ are sorted by decreasing similarity. We then obtain:

$$\varphi_i = \frac{1}{N} \sum_{j=1}^{N} \sum_{\alpha=1}^{N} \binom{N-1}{\alpha}^{-1} u_\Delta(\Gamma_{y(t)}, \Gamma_{y(t_i)}) \binom{N-j}{\alpha} \tag{38}$$

We shuffle the sums a little by multiplying $\frac{1}{N}$ with $\binom{N-1}{\alpha}^{-1}$ and expanding the u_Δ function according to its definition. We also alter the limit of the innermost sum because $\alpha \leq N - j$. Thus, we obtain:

$$\varphi_i = \sum_{j=1}^{N} \Big(\mathbb{1}\{y(t_i) = y(t_v)\} - \mathbb{1}\{y(t_j) = y(t_v)\} \Big) \sum_{\alpha=1}^{N-j} \binom{N}{\alpha}^{-1} \binom{N-j}{\alpha} \tag{39}$$

The inntermost sum in the above equation can be simplified by applying the so-called Hockey-stick identity [59]. Specifically, $\binom{N}{\alpha}^{-1}\binom{N-j}{\alpha}$ becomes $\binom{N}{j}^{-1}\binom{N-\alpha}{j}$. Then, $\sum_{\alpha=1}^{N-j} \binom{N}{j}^{-1}\binom{N-\alpha}{j}$ becomes $\binom{N}{j}^{-1}\binom{N}{j+1}$. Finally, we obtain the following formula:

$$\varphi_i = \sum_{j=1}^{N} \Big(\mathbb{1}\{y(t_i) = y(t_v)\} - \mathbb{1}\{y(t_j) = y(t_v)\} \Big) \binom{N-j}{j+1} \tag{40}$$

As we can see, the above formula can be computed in $O(N)$ iterations. Therefore, given that we still need to sort the dataset beforehand, the ovarall complexity of the entire Shapley value amounts to $O(N \log N)$. □

C.6 Proof of corollary 5.1.5

Proof. We will prove this by reducing the problem of Shapley value computation in fork pipelines to the one of computing it for map pipelines. Let us have two tuples $t_{j,1}, t_{j,2} \in f(\mathcal{D})$, both associated with some variable $a_j \in A$. That means that $p(t_{j,1}) = p(t_{j,2})$. If we examine Equation 20, we notice that it will surely evaluate to false if either $\sigma(t_{j,1}) > \sigma(t)$ or $\sigma(t_{j,2}) > \sigma(t)$. The same observation holds for Equation 21.

Without loss of generality, assume $\sigma(t_{j,1}) > \sigma(t_{j,2})$. Then, $\sigma(t_{j,1}) > \sigma(t)$ implies $\sigma(t_{j,2}) > \sigma(t)$. As a result, we only ever need to check the former condition without paying attention to the latter. The outcome of this is that for all sets of tuples associated with the same variable, it is safe to ignore all of them except the one with the highest similarity score, and we will nevertheless obtain the same oracle result. Since we transformed the problem to the one where for each variable we have to consider only a single associated tuple, then we have effectively reduced the problem to the one of computing Shapley value for map pipelines. Consequently, we can apply the same algorithm and will end up with the same time complexity. □

C.7 Details of Algorithm 1

In this section we examine the method of compiling a provenance-tracked dataset $f(\mathcal{D})$ that results from a pipeline f. The crux of the method is defined in Algorithm 1 which is an algorithm that takes a dataset $\mathcal{D}$ with provenance tracked over a set of variables $\mathcal{X}$ and a boundary tuple $t \in \mathcal{D}$. The result is an ADD that computes the following function:

$$\phi_t(v) := \begin{cases} \infty, & \text{if } t \notin \mathcal{D}[v], \\ \text{tally}_t \mathcal{D}[v] & \text{otherwise.} \end{cases} \tag{41}$$

Assuming that all provenance polynomials are actually a single conjunction of variables, and that the tally is always a sum over those polynomials, it tries to perform factoring by determining if there are any variables that can be isolated. This is achieved by first extracting variables that appear only once (Line 11) separating the total sum into components that don't share any variables (Line 12). Then for the variables that cannot be isolated (because they appear in polynomials in multiple tuples with multiple different variables) we form a group which will be treated as one binary vector and based on the value of that vector we would take a specific path in the tree. We thus take the group of variables and call the ConstructADDTree function to construct an ADD tree (Line 13).

Every path in this tree corresponds to one value assignment to the variables in that tree. Then, for every path we call the ConstructADDChain to build a chain made up of the isolated variables and call AppendToADDPath to append them to the leaf of that path (Line 22). For each variable in the chain we also define an increment that is defined by the number of tuples that will be more similar than the boundary tuple t and also have their provenance polynomial "supported" by the path. We thus construct a segment of the final ADD made up of different components. We append this segment to the final ADD using the AppendToADDRoot function. We don't explicitly define these functions but we illustrate their functionality in Figure 18.

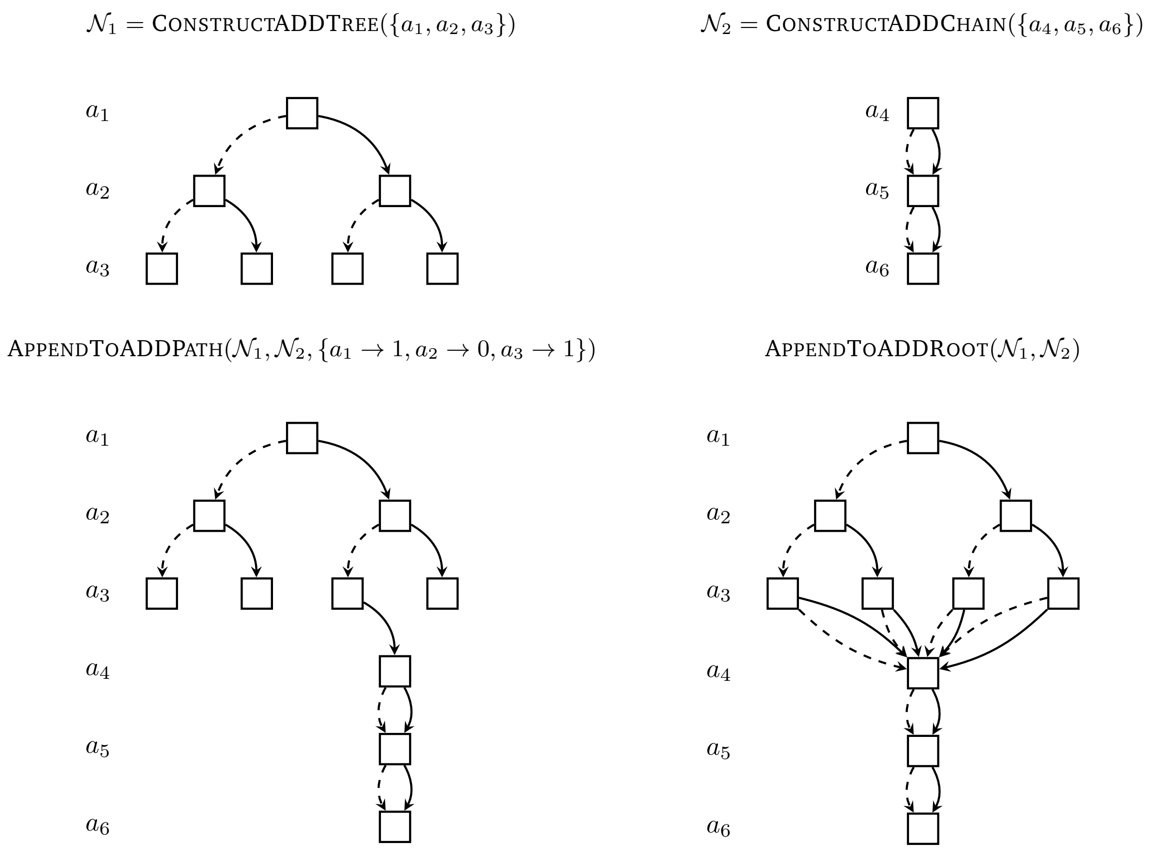
$\mathcal{N}_1 = \textsc{ConstructADDTree}(\{a_1, a_2, a_3\})$
$\mathcal{N}_2 = \textsc{ConstructADDChain}(\{a_4, a_5, a_6\})$
$\textsc{AppendToADDPath}(\mathcal{N}_1, \mathcal{N}_2, \{a_1 \to 1, a_2 \to 0, a_3 \to 1\})$
$\textsc{AppendToADDRoot}(\mathcal{N}_1, \mathcal{N}_2)$
a_1
a_2
a_3
a_4
a_5
a_6

Figure 18: An example of a ADD compilation functions.