Data Cleaning for Machine Learning Systems - A Survey

Amit Rajan [*]

June 28, 2024

Abstract

Data cleaning plays a pivotal role in ensuring the accuracy and reliability of machine learning (ML) systems. The goal of a data cleaning task is to enhance the quality and reliability of datasets by identifying and rectifying errors, inconsistencies, and inaccuracies, ensuring robustness and effectiveness in subsequent data analysis and machine learning tasks. This survey meticulously examines existing data cleaning systems, with a specific focus on three crucial aspects: **1) Integrity constraint violation detection, 2) Identification and handling of outliers, missing values, anomalies, and adversarial examples**, and **3) Deduplication or Entity matching techniques**. Rather than providing a superficial overview of numerous methods, the survey delves into **representative approaches**, offering **in-depth insights into their functionalities and results**. By thoroughly discussing these methods, the survey aims to provide a comprehensive understanding of the landscape of data cleaning techniques tailored for ML systems, aiding researchers and practitioners in selecting and implementing appropriate solutions for their specific use cases.

1 Introduction

There are two dimensions of data quality: **intension** (the structure or schema of the data), and **extension** (data values). A dataset should exhibit **completeness, consistency**, and **accuracy** across both dimensions. **Completeness** quantifies *how well the data defines a real-world object*. *Semantic rules* are constraints that specify the meaning or semantics of the data and ensures that a dataset is complete. These rules define the allowable values, relationships, and constraints within a dataset based on the intended interpretation of the data. **Consistency** encodes the *extent of the violation of semantic rules*. Data consistency can be reinforced through the implementation of *intra-relation* and *inter-relation constraints*, which pertain to individual and multiple attributes of a dataset. **Accuracy** measures the *correctness of the data*. Accuracy can be further categorized into **semantic** and **syntactic accuracy**. **Syntactic accuracy** refers to how closely the values in a column align with their real-world representations, while **semantic accuracy** pertains to the correctness of the value in relation to the represented data point. [36]

Data cleaning is essential for maintaining *data quality*. Data cleaning has two key steps: *error detection* and *error repair*. The process of *error detection* has three key stages: ***"What to Detect"***, ***"How to Detect"***, and ***"Where to Detect"***. *"What to Detect"* stage of error detection constitutes of different ways to detect errors, such as *integrity constraints, functional dependencies, denial constraints, conditional functional dependencies, domain value violations etc.* *"How to Detect"* stage decides whether these methods should be applied automatically or with human intervention. The decision of whether to apply the methods directly to the raw data or after pre-processing determines *"Where to Detect"* stage. Once errors are flagged, the next step is *error repair*. Error repair involves: **"What to Repair"**, **"How to Repair"**, and **"Where to Repair"**. These stages determine repair priorities, the level of automation, and whether changes should be made directly to the original dataset or not. [5]

Data errors can be **qualitative** or **quantitative** in nature. A **qualitative data error** refers to the error in the nature or quality of the data itself rather than the present numerical discrepancies. It includes issues like inconsistencies, inaccuracies, or invalidities in the attributes or values of the dataset, such as missing values, integrity constraint violations, anomalies, and adversarial examples. These errors affect the overall reliability, correctness, and interpretability of the data, often requiring manual inspection or complex algorithms for detection and correction. **Quantitative data error** pertains to numerical discrepancies or anomalies within the dataset. Unlike qualitative errors that affect the quality or nature of the data, quantitative errors involve issues such as

[*]E-mail: amitrajan012@gmail.com.

outliers, incorrect numerical values, or statistical irregularities. These errors can distort the analysis and interpretation of the data, impacting the accuracy and reliability of the results. Detecting and addressing quantitative errors often involves statistical methods or algorithms designed to identify and correct numerical inconsistencies.

With the rise of modern machine learning techniques, particularly Deep Neural Networks (DNNs), using a **noise-aware loss function** can assist trained models in approaching optimal performance even in the presence of noisy data. However, in real-world machine learning deployments, it is common to supplement training techniques with error detection and correction methods to ensure robustness. Ilyas et al., 2022 investigated the issue of mean estimation in the presence of noisy data, where adversaries may introduce errors to hinder accurate estimation by arbitrarily corrupting the data. It was observed that by considering data dependencies and implementing data repairs, more accurate mean estimation was achieved, reaching *information-theoretically optimal results*. This shows that a *two-step meta-algorithm*, involving **data repairs followed by robust learning, outperforms using robust learning alone**.[17] Hence, error detection and correction play a pivotal role in ensuring the reliability and effectiveness of a machine learning pipeline. The presence of data errors can significantly impact the performance and accuracy of ML models and therefore, integrating robust error detection and correction mechanisms into the machine learning pipeline is essential to identify and rectify data anomalies before they propagate through the system.

2 Preliminaries

A usual data cleaning system has various aspects, including the type of data errors it can handle, what objective should data cleaning aim to achieve, with its importance and other critical considerations. Beginning with an exploration of various data errors and their definitions, the section delineates a formal definition of data cleaning with its importance and various nuances that must be considered when undertaking a data cleaning task.

2.1 Data Errors

In addition to being categorized as **qualitative** and **quantitative** based on their characteristics, data errors can also be classified as: **schema-level errors** and **instance-level errors**. **Schema-level errors** in a dataset encompass inconsistencies and inaccuracies within the structure or schema of the data. These errors can manifest in several ways, including missing attributes or columns, incorrect data types, inconsistent attribute names, inconsistencies in defining relationships between tables, and violations of integrity constraints. **Instance-level errors** are the errors in actual data and, usually, are the hardest to flag. *Instance-level errors* can be further categorized as **single-source** and **multi-source**. **Single-source instance-level errors** occur within a single dataset or data source. These errors may include missing values, incorrect data entries, duplicates, outliers, and inconsistencies within the same dataset. On the other hand, **multi-source instance-level errors** involve discrepancies or inconsistencies across multiple data sources. These errors can arise due to data integration or merging processes, where data from different sources are combined. Multi-source instance-level errors may include conflicting information, inconsistencies in attribute values across datasets, and data duplication resulting from merging records from different sources. Multi-source instance-level errors are usually much harder to resolve. [30]

The initial stage of a data cleaning system involves establishing a framework to identify data errors. The data cleaning system usually utilizes various methods tailored to the specific characteristics of the errors for the purpose. Data errors primarily include **integrity constraint violations, missing values, outliers, anomalies, adversarial examples**, and **duplicates**.

A. Integrity Constraint Violations: Integrity constraints serve as a key tool for identifying qualitative data errors in the dataset. Given a database D with relations $R_1, R_2, ..., R_K$, relation R_i having a set of *attributes* $attr(R_i) = \{A_1, A_2, ..., A_N\}$, *integrity constraint* in the form of **functional dependency** over R_i is defined as:

$$\begin{aligned}\Phi_{FD}:\ (R_i : X \rightarrow Y) = \forall t_1, t_2 \in R_i,\\ (t_1[X] = t_2[X]) \implies (t_1[Y] = t_2[Y])\end{aligned} \tag{1}$$

where t_1, t_2 are tuples in R_i, $X \subseteq attr(R_i), Y \subseteq attr(R_i)$ are set of attributes in R_i, and $t_1[X]$ denotes the values for the set of attributes X for tuple t_1. Equation 1 implies that if two tuples in relation R_i have the same values for attributes X, then they must have the same values for attributes Y.

Functional dependency constrained by a condition C is called as **conditional function dependency** and is denoted as:

$$\Phi_{CFD}:\ (R_i : X \rightarrow Y, C) \tag{2}$$

Equation 2 specifies that for a given set of attributes X, the values for the set of attributes Y are uniquely determined, but only when the condition C holds true.

Another form of integrity constraint is denial constraint. **Denial constraints** are defined as:

$$\Phi_{DC}: \neg(P_1 \wedge P_2 \wedge ... \wedge P_M) \quad (3)$$

where P_m is a *predicate* of the form $(t_i[A_n] \circ t_j[A_m])$ or $(t_i[A_n] \circ \alpha)$ with tuples $t_i, t_j \in D$, A_n, A_m being the attributes, α being a constant, and $\circ$ is the comparison operator. [32]

Inclusion dependencies (INDs) ensure that the values in one set of attributes are a subset of the values in the other set. This means that for two attributes A and B, if $dom(A) \subseteq dom(B)$, the values of attribute A should be a subset of values of attribute B, and it is said that attribute A is dependent on attribute B.

Domain value violations are the type of integrity constraint which is used to flag the values of an attribute that is outside its domain. The value of a given tuple t for the attribute A_i exhibits domain value violation if $t[A_i] \notin dom(A_i)$, where $dom(A_i)$ is the domain of A_i. The usual approach to detect domain value violations is by writing *custom error detectors.*

Functional dependencies, conditional functional dependencies, denial constraints, inclusion dependencies and *domain value violations* are tools that address *qualitative data errors* and aid in their identification.

B. Missing Values, Outliers, Anomalies, and Adversarial Examples: Missing value is a qualitative data error which refer to the absence of information for certain attributes in some of the data records. Missing values can be classified as: **missing completely at random (MCAR), missing at random (MAR)**, and **not missing at random (MNAR)**. Given dataset D, a tuple $t \in D$, and ϕ denoting the missing value; for *MCAR*, the probability that a cell value is missing does not depend on any of the attribute values in the tuple, i.e. $Pr(t[A_j] = \phi \mid t[A_i] = v_i, \forall i; D) = p_j$. In *MAR*, the probability that $t[A_j]$ is missing depends on the other observed cells (or attribute values) in t, i.e. $Pr(t[A_j] = \phi \mid t[A_i] = v_i, \forall i; D) = Pr(t[A_j] = \phi \mid t[A_i], \forall i \neq j)$. The missing values that do not follow *MCAR* and *MAR* are *MNAR* (the probability of missing value may depend on the value missed itself). [43]

Anomalies and **outliers** are data points that deviate significantly from the rest of the dataset or exhibit unusual behavior compared to the majority of the data. Anomalies usually arise due to errors in data collection, measurement inaccuracies, or rare events that are not representative of the typical behavior of the data. On the other hand, outliers can occur naturally in data and often fall outside a certain range or threshold of values for a given attribute.

Adversarial examples are data points that are intentionally crafted to cause the machine learning model to make a mistake. They are usually very similar to the legitimate inputs. Missing values, outliers, anomalies and adversarial examples can have a significant impact on statistical analyses and machine learning models if not properly handled.

C. Duplicates: Duplicates refer to records that represent the same real-world entity. Given two relations R_1, R_2, duplicates are identified as the records $a \in R_1$ and $b \in R_2$ which are similar to each other. The notion of similarity is usually defined using a *similarity function* $sim(a, b)$ that compares the record pair (a, b) and gives a *similarity score* between them. Records a and b are termed as a *possible duplicate*, if the similarity score is greater than a threshold:

$$\Phi_{Dup}: ((sim(a,b) > \tau) \implies \text{pair (a, b) is a possible duplicate}) \quad (4)$$

2.2 Formal Definition of Data Cleaning

Given a database D with relations $R_1, R_2, ..., R_K$, the goal of the data cleaning task is to find a **cleaned database** D_{clean} which is as close to the **ground truth database** D_G (usually unknown) as possible. A data cleaning system relies on a **cleaning operation** $C(.)$, which takes the dirty record r as the input and either modifies it to give the clean record $r^{'} = C(r)$ or deletes it $(\phi = C(r))$. Each cleaning operation performs a database edit, which serves as the proxy for closeness to the ground truth database. As the ground truth database D_G is unknown, the system usually tries to minimize the number of database edits (or total cost of database edits) such that the resultant cleaned database D_{clean} satisfies all the given constraints (or any other defined criteria). Each edit $v \rightarrow v^{'}$ (changing value v to $v^{'}$) has an associated cost $cost(v, v^{'})$ with it, which is usually encoded by a **distance function** $dis(v, v^{'})$. The goal of a data cleaning system is to identify the **minimal cost repair** (edits) from the possible set of valid edits (edits which lead to a database that satisfies the given constraints), and is formally defined as:

$$D_{clean} = \underset{edit \in set\ of\ edits}{\arg\min} \left(\sum_{e \in edit} cost_e(v, v^{'}) \right) \quad (5)$$

where $e \in edit$ is one of the edit in the *edit* set, and $cost_e(v, v^{'})$ is the cost incurred for the edit e. This formalization treats all repairs of a database as **equally probable** and **deterministic** and do not offer insights into the **likelihood of specific repairs**. [21] [3] [8] [34]

An alternative approach views data repairing as a **statistical learning and inference problem**, aiming to identify the most probable repair rather than focusing solely on the **concept of minimality**. In the probabilistic model of data cleaning, an **unclean database** D can be viewed as a result of a distortion of the cleaned database D_{clean} due to some noisy model. To clean D, its cleaned version D_{clean} needs to be picked for which $Pr(D_{clean}|D)$ is maximum:

$$\arg\max_{D_{clean}} Pr(D_{clean}|D) = \arg\max_{D_{clean}} Pr(D|D_{clean})Pr(D_{clean}) \quad (6)$$

where $Pr(D_{clean})$ represents the **prior model of clean database** and $Pr(D|D_{clean})$ characterizes the **noisy model**. [34]

2.3 Importance of Data Cleaning

Data cleaning plays a crucial role in ensuring the effectiveness and accuracy of machine learning systems. Clean data is essential for training models that can generalize well and make reliable predictions. Without proper data cleaning, machine learning algorithms may be susceptible to biases, errors, and inaccuracies, leading to sub-optimal performance and unreliable results. With the rise of modern ML systems, the blurring of **abstraction boundaries** (includes weakening of the separation between different components or layers of data sources within a ML system) can result in significant **technical debt** and subsequently higher maintenance costs. Given their inherent complex nature, ML systems often struggle to conform to a specific abstraction concept. An example of this **erosion of abstraction boundaries** is seen in **data dependency**, where ML systems relying on complex data storage systems require consistent and clean input data. The *input signal* serves as a critical component for any ML system, and even slight alterations to it can profoundly affect the system's behavior. [37]

The dataset's quality significantly influences the choice of model in any machine learning system. While conventional ML systems typically prioritize data cleaning for training datasets, the significance of cleaning test datasets cannot be overstated. Northcutt et al., 2021 investigates how label errors in test sets can impact *ML benchmark stability*. The study evaluated the prevalence of labeling errors in commonly used 10 ML benchmark datasets for assessment purposes and examined the practical consequences of these errors, with a particular focus on their impact on model selection. The label errors were algorithmically identified (using *confident learning framework* [28]) and were validated using human reviewers. For the large datasets, a random sample was reviewed, while for the others, all identified errors were checked. Reviewers were presented with *hypothesized errors* and asked whether they observed *the given label, the top algorithmically predicted label, both labels*, or *neither label* in the example. Errors were further categorized as: *correctable* (majority agreed on the algorithmically predicted label), *multi-label* (majority agreed on both labels), *neither* (majority agreed on neither label), and *non-agreement* (if there was no majority). To quantify the effect of correcting label errors in test set, two accuracies are calculated: **original accuracy** and **corrected accuracy**. *Original accuracy* refers to the accuracy of a model's predicted labels computed with respect to the original labels in the dataset. *Corrected accuracy* measures the accuracy of a model's predicted labels over a modified dataset where previously identified erroneous labels have been corrected (when possible) or removed. The experiments show that their exist an estimated least lower-bound of 3.3% errors on average across the 10 selected datasets. Upon closer examination of the models' performance on the *corrected dataset*, the results show that the **models that excel on the original (incorrect) labels perform poorly on the corrected labels**. In most of the cases, lower capacity models fared well on the basis of *corrected accuracy* compared to their more powerful counterparts. Hence, it is recommended that despite training an ML system on a lower-quality training set with noisy data, efforts should be made to invest time and resources in rectifying label errors in the test set. [29]

2.4 Considerations in Data Cleaning

Although data cleaning might appear straightforward at first glance, it involves numerous complexities and challenges. Beyond detecting and correcting obvious errors like missing values or duplicates, data cleaning often involves navigating complex data structures, identifying subtle inconsistencies, and handling noisy or erroneous data points. Moreover, the effectiveness of data cleaning techniques can vary significantly depending on the specific characteristics of the dataset, such as its size, complexity, and domain-specific nuances. Additionally, data cleaning is an iterative process that may require multiple rounds of refinement and human validations to achieve satisfactory results.

Identifying and rectifying **missing values**, seemingly one of the most straightforward data cleaning tasks, can present numerous complexities. When only a small portion of samples contain missing data and these misses are random (MCAR), excluding such samples may not distort subsequent analysis or introduce sampling error. However, missing values categorized as MAR or MNAR may suggest an inherent pattern, such as the absence of a specific attribute for all samples within a particular class, necessitating tailored cleaning approaches. [43]

In addition to this apparent and straightforward observation, there are numerous concealed intricacies that must be taken into account when constructing any data cleaning system. **Freire et al., 2016** used **NYC taxi data** to demonstrate the challenges and considerations while cleaning a **spatial-temporal urban data**. Urban data usually has *limited metadata*, is often derived from *incomplete schema information*, *lacks integrity constraints*, and *requires inference of the information provided*, complicating the entire data cleaning task. The data can be aggregated at different spatial and temporal levels or *resolutions* (e.g., neighborhoods, zip codes, hourly, daily) with the identification of dirty data depending on the chosen resolution. **Data labeled as dirty in a specific time or place may represent a pattern at different resolution**. For example, significant drops in trips on Christmas and New Year's day are recurring yearly patterns, not a dirty data. Apart from this, in urban data, the **outlier and anomalies are not always dirty data**. For example, in *NYC taxi data* , there is a significant absence of taxis on 6^{th} avenue between Midtown and Downtown, which can be easily explained as: during this time frame, 6^{th} avenue was closed for the *annual NYC 5 Boro Bike Tour*. Another example is the *trip drops in August 2011*, which can be linked to weather events like heavy rainfall and Hurricane Irene (after analyzing precipitation and wind data for that year). This suggests that these anomalies may not necessarily indicate erroneous data but instead reveal intriguing phenomena warranting further investigation. Hence, a data cleaning system, designed for urban data cleaning, should **enable users to explore data across different aggregation levels, guiding them to intriguing data subsets automatically**. Despite this guidance, domain experts may still need to scrutinize various spatio-temporal segments to detect patterns, irregularities, and potential errors. Moreover, users must discern whether these events signify data quality concerns or significant features, which may necessitate identification and integration of additional external datasets into the urban data context. [10]

In summary, while data cleaning may seem straightforward, it's fraught with complexities and challenges beyond the obvious errors like missing values or duplicates. It involves navigating intricate data structures, identifying subtle inconsistencies, and handling noisy or erroneous data points. Being an iterative process often requiring multiple rounds of refinement and human validation, the effectiveness of data cleaning techniques varies depending on factors like dataset size, complexity, and domain-specific nuances.

2.5 Components of a Data Cleaning System

A data cleaning system is an essential element within any machine learning framework. Such a system can either function solely as an **Error Detection System** or as an **Integrated Error Detection and Cleaning System**. Based on the type of errors they handle, any data cleaning system can be classified as: Detection/Cleaning of **Integrity constraint violation**; **Outliers, missing values, anomalies and adversarial examples**; and **Duplicates (Entity Matching Systems)**.

In the subsequent sections, existing representative data cleaning systems are outlined, providing detailed insights into their functionalities and outcomes. Section 3 lists error detection and cleaning systems that deal with **integrity constraint violation**. Section 4 provides a thorough overview of systems addressing **missing values, outliers, anomalies, and adversarial examples**. Section 5 discusses representative **entity matching (de-duplication)** systems. Section 6 wraps up the discussion by highlighting the significance of well-informed decision-making and selection framework for identifying an appropriate data cleaning methodology tailored to the specific use-case.

3 Integrity Constraint Violation

Data cleaning systems dealing with **integrity constraint violation** are designed to identify and rectify inconsistencies in datasets that violate predefined integrity constraints. These constraints define the acceptable rules and relationships that data must adhere to. Data cleaning systems targeting integrity constraint violations typically employ techniques such as constraint-based inference and repair algorithms to detect and resolve inconsistencies.

Schelter et al., 2018 introduced an automated data validation framework designed for large-scale data sets. The proposed system offers declarative APIs equipped with both standard quality constraints and custom validations, allowing users to focus on specifying quality checks rather than implementation details. Users can define quality constraints using **internal or external libraries**, or create custom functions as needed. These constraints are then translated into **data quality metrics**, which can be accurately computed or approximated depending on complexity. To handle the growing data volume efficiently, the system is state-aware, facilitating incremental computation of **reformulated data quality metrics**. Additionally, the framework includes a machine learning model for predicting and verifying column values' correctness. It also provides automated constraint suggestions based on heuristic single-column profiling and a straightforward anomaly detection algorithm leveraging historical data quality metrics. Implemented atop Apache Spark, the system utilizes AWS for data storage. [36]

Continuous Data Cleaning uses data, constraints and past repairs as the evidence to suggest most probable and accurate repairs in the future. It is a dynamic data cleaning framework which does **continuous** and **adaptive** data cleaning by performing **data as well as constraint evolution**. The proposed system sees the data cleaning problem as a classification task whose goal is to identify the most probable repairs given a dataset and integrity constraints. The classifier predicts the probable repairs (and their probability) with the help of a set of statistics computed over the dataset and the constraints. The predicted probable repairs are fed to repair algorithm that selects the best possible repairs based on a cost model. The selected repairs are presented to the end-user which then chooses which repair should be used for the resolution. The applied repairs are used to re-train the classifier. The system fixes the violated integrity constraints by: **Data Repair, Constraint Repair** or **Hybrid Repair**. Data repairs are further classified as: **Right Data Repair** and **Left Data Repair**. In right data repair, the constraint violations are fixed by changing the right hand side attribute values. Left data repair changes the left hand side attribute set (X) values for the violated constraints instead (refer Equation 1,2). Constraint repair is implemented by adding more attributes in the left hand side attribute set X. If both the data and constraint repairs are used to repair a violated constraint, the process is termed as hybrid repair. The system maintains a set of statistics, called as **repair statistics**, over dataset and constraints which are then used by the classifier to suggest the repairs. These *repair statistics* should be easy to maintain and compute and should capture the incremental changes needed to identify the constraint violations. A logistic regression based multi-class classifier with a total of 7 classes: *Not Repaired, Completely Repaired by Data/Constraint/Hybrid Repairs, Partially Repaired by Data/Constraint/Hybrid Repairs* with repair statistics and the past repairs as the features, is used for the classification task. Weighted cross-entropy function with weight as the fraction of patterns in the class is used as the objective function and is given as:

$$L = -\frac{1}{|\mathcal{P}|}\sum_{p \in P}\frac{1}{|C_c|}\sum_c P(p=c)\ln Q(p=c) \qquad (7)$$

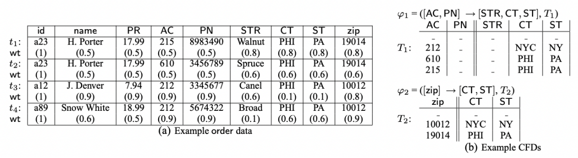
where $\mathcal{P}$ is the set of patterns violating the constraints, Q is the standard *softmax* function, C_c is the set of pattern assigned to class c, and $p = c$ means that the pattern p is assigned to class c. To test the performance of the system on real world dirty datasets a baseline classifier *CL-A* is trained on the initial sets of baseline repairs and repair statistics. *CL-A* predicted repairs are manually evaluated to form a *user validated repair set B* which is then used with repair statistics to train the classifier *CL-B*. The classifier *CL-B* outperforms *CL-A* in terms of accuracy and provides **an average per class classification gain of 11 points**. [42]

Cong et al., 2007 proposed a data cleaning system that guarantees that the suggested repairs satisfy the given integrity constraints and are **accurate above a predefined rate**. They suggested a way to represent condition C in a conditional functional dependency represented by Equation 2 as a table T_p. The updated CFD is given as:

$$\Phi_{CFD}: \ (R_i : X \rightarrow Y, T_p) \qquad (8)$$

T_p follows a **tabular pattern** where for each attribute A in X, Y there will be one column each with values being either *"-"* or some constant from domain of A ($dom(A)$). A typical CFD with sample data is shown in Fig 1. Row 2 of CFD ϕ_2 means that for the *zipcode* 10012, the *city* should be NYC and *state* should be NY. *"-"* means that the attribute A can take any value in the $dom(A)$. CFDs can be transformed into **normal**

form as $(R : X \to A, t_p)$ where the right hand side of FD has a single attribute A and t_p is a single pattern tuple (single row in the table T_p). CFD violations can be identified for a single tuple



	id	name	PR	AC	PN	STR	CT	ST	zip
t_1:	a23	H. Porter	17.99	215	8983490	Walnut	PHI	PA	19014
wt	(1)	(0.5)	(0.5)	(0.5)	(0.5)	(0.8)	(0.8)	(0.8)	(0.8)
t_2:	a23	H. Porter	17.99	610	3456789	Spruce	PHI	PA	19014
wt	(1)	(0.5)	(0.5)	(0.5)	(0.5)	(0.6)	(0.6)	(0.6)	(0.6)
t_3:	a12	J. Denver	7.94	212	3345677	Canel	PHI	PA	10012
wt	(1)	(0.9)	(0.9)	(0.9)	(0.9)	(0.6)	(0.1)	(0.1)	(0.8)
t_4:	a89	Snow White	18.99	212	5674322	Broad	PHI	PA	10012
wt	(1)	(0.6)	(0.5)	(0.9)	(0.9)	(0.1)	(0.6)	(0.6)	(0.9)

(a) Example order data

$\varphi_1 = ([\text{AC}, \text{PN}] \to [\text{STR}, \text{CT}, \text{ST}], T_1)$

	AC	PN	STR	CT	ST
T_1:	–	–	–	–	–
	212	–	–	NYC	NY
	610	–	–	PHI	PA
	215	–	–	PHI	PA

$\varphi_2 = ([\text{zip}] \to [\text{CT}, \text{ST}], T_2)$

	zip	CT	ST
T_2:	–	–	–
	10012	NYC	NY
	19014	PHI	PA

(b) Example CFDs

Figure 1: CFDs (Condition encoded as a Table) [8]

t or a pair of tuples (t, t'). A **single tuple** t **violates CFD** $\phi = (R : X \to A, t_p)$ if $t[X] = t_p[X]$ and $t[A] \neq t_p[A]$. A **pair of tuple** (t, t') **violates CFD** $\phi = (R : X \to A, t_p)$ if $t[X] = t'[X] = t_p[X]$ and $t[A] \neq t'[A]$, given the tuple t satisfies the CFD, i.e. $t[A] = t_p[A]$. To fix the CFD violations, the framework relies on **attribute value modifications**. The value for attribute A can either be changed to some value from $dom(A)$ or to *null* (in the case of uncertainty in the repair). These modifications can either be done to RHS or LHS attributes of CFD ϕ. To delete a tuple, *null* value is set to all of its attributes. To select the optimal repair, the framework relies on weights for each attribute value of a tuple, denoted as $w(t, A)$ and the distance between two values in the same domain, denoted as $dis(v, v')$. The cost of the repair $t, A : v \to v'$ (changing the value of attribute A from v to v') is given as:

$$cost(v, v') = w(t, A) \cdot \frac{dis(v, v')}{max(|v|, |v'|)} \tag{9}$$

The **total cost of modifying tuple** t **to** t' in which a total of R attributes is changed is the sum of $cost(t[A], t'[A]) \forall A \in R$. The weight information and the distance metrics should be provided to the framework. The objective of the system is to find the minimal cost repair for a CFD violation. This problem is **NP-complete**. The heuristic algorithm to find the best repair for a given CFD violation is based on the concept of **equivalence classes** [3]. The key idea behind *equivalence class* is that all the tuples in the same equivalence class will have the same value for the associated attributes. An *equivalence class* E is a list of tuple attribute pair (t, A) whose target value is denoted as $targ(E)$. Hence, $\forall (t, A) \in E; t[A] = targ(E)$. $targ(E)$ can either be a value in $dom(A)$ or *null* or "-" (meaning $targ(E)$ is not yet fixed). *Equivalence class* can be used to repair the violated constraints. For example, for the tuple pair t, t' violating a CFD (i.e. $t[X] = t'[X] = t_p[X]$ and $t[A] \neq t'[A]$), the ideal repair is to move $(t, A), (t', A)$ to the same *equivalence class* E whose target value $targ(E)$ can be decided later. This decouples the two main tasks of deciding which attribute values should be equal and what the value should be. The **assignment of value to a particular *equivalence class*** can be **delayed to mitigate *poor decisions* at early stage**. The system picks the next best (CFD,tuple) pair that takes the least cost to repair by looping over the all possible (CFD,tuple) pairs. The system also supports **incremental data repair**. To find the accuracy of the cleaned database, a part of cleaned tuples is sampled and sent for manual review. If the calculated accuracy is not in the desired range, the user may edit the CFDs and the cleaning process will restart. The experimental results show that the suggested cleaning framework improves data quality in multiple settings, scaling well with database size. **Repair quality may decrease with increasing error in database and sometimes the system may introduce additional errors** during the cleaning process. [8]

Bohannon et al., 2005 proposed a **cost-based data cleaning system** that uses the concept of **equivalence class** to deal with **functional dependencies** (FD) and **inclusion dependencies** (INDs). Given a tuple $t \in R_i$, where R_i is a relation in a database D, and its value for attribute A being denoted as $D(t, A)$, a repair changes the value of $D(t, A)$ from v to v' leading to a clean version D' of database D. The cost of this repair is encoded using a **cost function** $w(t) \cdot dist(D(t, A), D'(t, A))$, where $w(t) \geq 0$ is the weight associated with tuple t. Given a fixed cost to insert any tuple in relation R_i, the repair cost of any tuple t is given as:

$$cost(t) = \begin{cases} inscost(R_i) & \text{if } t \in new(R_i) \\ w(t) \cdot \sum\limits_{A \in attr(R_i)} dis(D(t, A), D'(t, A)) & \text{otherwise} \end{cases} \tag{10}$$

The **total cost of repair** then becomes $cost(D') = \sum_{t \in D'} cost(t)$. The data cleaning problem is to find a minimal cost repair D'. **FD violation** is fixed by changing the right hand attribute. Given an IND $I : R_1[A] \subseteq R_2[B]$ and a violating tuple $t_1 \in R_1$, **IND violation** is repaired by changing $D(t_1, A)$ to any value in $dom(B)$ or $D(t_2, B)$ for some $t_2 \in R_2$ to $D(t_1, A)$. The problem of finding the optimal repair is **NP-complete** and a heuristic approach is used. The algorithm to find the optimal repair is based on the concept of *equivalence class*. Given a **target value** v, the **cost associated** with the **equivalence class** *eq* is given as:

$$cots(eq, v) = \sum_{(t,A)\in eq} w(t) \cdot dis(v, D(t, A)) \quad (11)$$

The target of the data cleaning task is to minimize $cost(eq, v)$ given a set of possible values for v. For some of the repairs, the **equivalence classes may need to be merged**. The cost associated with the merger can be formulated as the difference between the cost of the **merged equivalence class** and the sum of costs associated with individual classes. The proposed repair algorithm puts each **tuple attribute pair in their respective equivalence class and then greedily merges the equivalence classes until all constraints are satisfied**. The experimental results show that the framework performs well on real-world data cleaning problems. The run-time of the algorithm is $O(n^2)$ which can be reduced to $O(n(\ln n)^2)$ if **sub-optimal repairs with respect to cost** are allowed. The optimization allowing the sub-optimal repairs to reduce the run-time doesn't affect the quality of repairs by much. [3]

Llunatic is a **chase-based data cleaning framework** to find minimal cost repair to dirty databases. It formulates integrity constraints using **equality generating dependencies (egds)**. *Egds* takes the form:

$$\begin{aligned} e_1 &: R(\boldsymbol{A}, \boldsymbol{B}, C, D, E), R(\boldsymbol{A}, \boldsymbol{B}, C', D', E') \to C = C' \\ e_2 &: R_1(\boldsymbol{A}, \boldsymbol{B}, C, D, E), R_2(\boldsymbol{A}, \boldsymbol{B}, C') \to C = C' \end{aligned} \quad (12)$$

where e_1 means that in a relation R, if values of attributes A and B match, the value of attribute C should be the same. e_2 extends this constraint to two distinct relations R_1 and R_2. Given a **source database** S (having ground truth information), a **target database** T, and a set of constraints defined in the form of **egds** Σ, **Llunatic** performs the data cleaning task by focusing on **cell groups**. A **cell group** g, is defined by its **justifications** $just(g)$, **occurrence** $occ(g)$, and the **value** v. A **cell group** is repaired by changing all the cells in $occ(g)$ in T to the **value** v which is justified by $just(g)$ from S. A **valid repair** is a set of **cell groups** $\{g_0, g_1, ..., g_k\}$ where each cell in T occurs in at most one cell group (to avoid conflicts). **Each of the cells will either be modified by the repair encoded in the *cell group* or will be unchanged**. The repair algorithm depends on a user-defined **partial order** for each of the attributes. The **partial order** of the attribute A_i, denoted as P_{A_i} is the preferential order in which the value for any cell for the attribute A_i should be picked during the repair process. For example, if in a relation we have two attributes *salary* and *date*, the *partial order* of *salary* can be defined by values in *date* column as *salary* for latest date will be preferred. The concept of **partial ordered** is extended to **cell group** by using the notion of **containment**. Given two cell groups g and g'; if $occ(g) \subseteq occ(g')$, $just(g) \subseteq just(g')$, and $val(g')$ has higher preference (based on *partial ordering*) compared to $val(g)$; then ***cell group*** g' **is said to have higher *partial order* compared to** g. Otherwise, **cell groups** are termed as **incomparable**. The concept of **partial ordering is further extended to repairs**. For two repairs Rep and Rep', if $\forall g \in Rep; \exists g' \in Rep'$ such that g' is higher in *partial ordering* compared to g, and the repair Rep' is preferred (it is said that **repair** Rep' ***upgrades*** Rep). Given $< S, T, \Sigma, \Pi >$, where Π is the *partial order*, a **repair** Rep ***upgrades*** T **if** Rep **satisfies** Σ **with respect to** Π. The **minimal repair** Rep is the repair such that there doesn't exist any other repair which is higher in the order with respect to the *partial order* Π. The *partial ordering* of two repairs can be checked in $O(n + km\log(m))$ time, where n is the total number of cells in T, k is the maximum number of *cell groups*, and m is the maximum size of a *cell group*. To repair any *egd*, the algorithm depends on the notion of **equivalence class**. Whether to proceed with a repair or not, is decided by the **cost manager** that tells whether a proposed repair is well within the expected distance. The experimental results demonstrate that **Llunatic produces repairs of significantly higher quality** on real-world datasets with decent scalability. [11]

BoostClean is a data cleaning system that **selects on ensemble of error detection and repair combination using statistical boosting** for **domain value violations**. It takes training data (X_{train}, Y_{train}), and test data (X_{test}, Y_{test}) as the input where X_{train}, Y_{train}, and X_{test} may have errors, but Y_{test} **should be clean to get an *unbiased measure of accuracy***. A *record* is represented as $r_i = (x_i, y_i) \in (X_{train}, Y_{train})$ and $r_i.y$ denotes the *label* of the record. A *classifier C* gives the correct prediction for the record r_i if $C((x_i, null)).y = y_i$. The overall accuracy of the classifier is given as:

$$acc(C) = \frac{|\{\forall x, y \in (X_{test}, Y_{test}) : C((x, null)).y = y\}|}{|Y_{test}|} \quad (13)$$

A **black-box** function $train(.)$ uses training data (X_{train}, Y_{train}) and returns the classifier C. User-provided sets of **detector generators** $D = \{d_1, d_2, ...\}$ and **repairs** $F = \{f_1, f_2, ...\}$ are used by **BoostClean** to identify **candidate dirty records** and select **appropriate repair** for them. *Detector generators* use **predicates** (a **boolean expression** over any record) to identify *candidate* *dirty records*. While error detection, **predicates** return the set of **referenced attributes** (if evaluated to *true*) or an empty set (Φ), if no error in **referenced attributes** is detected. **Repair functions** do either **data repairs** or **prediction repairs**. **Data repairs** change (or even delete) the attribute values in training data for the **candidate dirty records** before the training process. **Prediction repairs** take the **classifier prediction** and replace it with some **default value**. The generated repairs are denoted as a sequence of *data* and *prediction repairs* $L = (l_1, l_2, ..., l_n)$ where $L \in D \times F$. L is further divided into **sequence of *data repairs* L^d that are applied before ML training and *prediction repairs* L^p that are used to construct the final model C_L** by combining them with classifier C. The problem of selection of **optimal repair sequence** L^* is formulated as:

$$L^* = \underset{L \in D \times F}{\arg\max}\, acc(C_L) \tag{14}$$

that generates a classifier C_{L^*} which maximizes prediction accuracy on (X_{test}, Y_{test}). **BoostClean** uses an **adaptive boosting algorithm** having equal initial weights for all the data points, with increase in the weights of incorrectly classified points in further rounds. *BoostClean* has a **pre-populated library of detector generators** and **repair functions**. It further optimizes the optimal repair sequence selection using **hashing** and **parallelization**. [20]

HoloClean is a data cleaning framework that combines **integrity constraint, external data source based data repair and statistical data repair** together. It treats input data as a **noisy version** of clean data and treats each **repair signal** suggested by the repair algorithms as **evidence**. It then uses **probability theory to combine these evidences** together to come up with the final repair step. Given a dirty dataset D with attributes $A = \{A_1, A_2, ..., A_N\}$, with the n^{th} cell of a tuple $t \in D$ being denoted as $t[A_n]$, where $A_n \in A$. The **unknown true value** of a cell is v_c^* with v_c being the **initial observed value**. A cell is **erroneous** if $v_c^* \neq v_c$. The **estimated true value** for a cell is $\hat{v_c}$ and a repair is correct if $\hat{v_c} = v_c^*$. **HoloClean** takes the dirty dataset D and a set of **integrity constraints** Σ (having **denial constraints** and **matching dependencies**) as input, and returns cleaned version of D as output. *HoloClean* has three main steps: **Error detection, compilation**, and **data repairing**. **Error detection** is treated as a **black-box** step which divides D into D_c (**clean cells**) and D_n (**noisy cells**). The **compilation** step takes the **initial cell values** Ω and **integrity constraints** Σ as input, and identifies each cell value $c \in D$ with a random variable T_c that takes values from $dom(c)$ (domain of cell c). It then uses a **probabilistic graphical model** (factor graph) to encode the **distributions of random variables** T_c. The **data repairing** step uses **empirical risk minimization (ERM)** algorithm to estimate the parameters of the probabilistic model and computes the **marginal probability** $P(T_c = d; \Omega, \Sigma)$. Clean cells of D_c are treated as **labeled examples** to learn the parameters of the model. Each suggested repair by the *HoloClean* framework has an associated probability with it. This probability is inferred as the confidence of the suggested repair and can be used to decide whether the repair can be applied directly or to be sent for manual review. The user-verified repairs can be further used as *labeled example* to retrain the model parameters. *HoloClean* uses **DeepDive** [38], which uses a declarative language **DDlog**, to write **inference rules** and construct factor graphs. It has a compiler that converts *integrity constraints* to a **DDlog program** containing *inference rules*. Scalability of *HoloClean* is affected by two factors: random variables (or attributes) that have large domains and factors that involve multiple tuples. *HoloClean* implements two pruning strategy to make the system efficient. It **prunes the domain of random variables using co-occurrence of other cell values in the tuple containing the cell of interest** c. To prune the factors involving multiple tuples, *HoloClean* **limits the number of tuples to consider for the identification of constraint violations**. It implements a pruning strategy in which tuples are binned into groups such that the **tuples in the same group have high probability of constraint violation**, and hence tuples in the same bin are further considered for constraint violation. The performance of *HoloClean* is evaluated on 4 real-world datasets to validate its accuracy, scalability and effect of *different signals* on data cleaning task. *HoloClean's* approach of **unifying multiple repairing signals** results in 2× **improvement on *F1-score*** over the techniques that *consider isolated signals for data repairing*. Apart from this, the implemented pruning strategies lead to high accurate repairs with scalability. [32]

Holistic Data Cleaning is a data cleaning framework which considers the **interaction among different classes of constraint violations** and takes **holistic view of the violations** to come up with the repair strategy. It compiles violations as a **Conflict Hypergraph (CH)** and uses **novel holistic repairing algorithm** to come up with the repairs with respect to **one unified objective function**. **Integrity constraints** in *holistic data cleaning framework* are represented as unified **denial constraints (DCs)**, which takes the form:

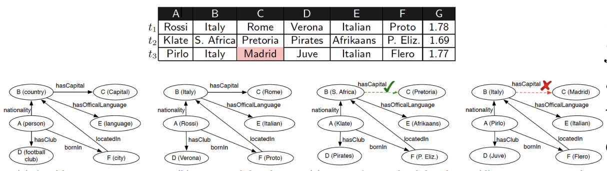
$$\begin{aligned} c_1 : \neg(G(c,r,s), G(c^{'},r',s'), (c = c^{'}), (s = s^{'})) \\ c_2 : \neg(G(c,r,s), G(c^{'},r',s'), (r = r^{'}), \\ (c = \text{"}NYC\text{"}), (c^{'} \neq \text{"}NYC\text{"}), (s^{'} > s)) \end{aligned} \tag{15}$$

DC c_1 means that if two tuples have the same value for attribute c, attribute s should be the same. *DC* c_2 means that if two tuples have the same value for attribute r and different values for attribute c, where one of the tuple has attribute c as "NYC", the value of attribute s for the tuple having $c =$ "NYC" should be greater. Given a set of denial constraints Σ and a database instance I, such that $I \not\models \Sigma$ (I does not satisfy all the DCs in Σ), the **goal of data cleaning is to find the minimal cost repair** I_r, such that $I_r \models \Sigma$, where the cost of the repair is given as:

$$cost(I_r, I) = \sum_{t \in I, t^{'} \in I_r, \forall A} dist_A(t[A], t^{'}[A]) \tag{16}$$

Holistic Data Cleaning framework uses **squared Euclidean distance** (called as **distance cost**) as distance measure for numeric values. The problem of finding minimal *distance cost* repair is **NP-complete**. The framework uses **conflict hypergraph (CH)** to **encode constraint violations** and **repair context (RC)** to **encode violation repairs**. *CH* encodes all violations in a graph where **nodes represent *violating cells* and edges *link the cells involved in the same violation***. At least one of the *violating cells* should be changed to repair the violation. A naive algorithm picks edges one-by-one and repairs the connected cells. This approach will lead to a valid repair but the minimal cost repair is not guaranteed. *Holistic Data Cleaning* uses **minimum vertex cover (MVC)** algorithm to look at the violated constraints holistically and find the minimum cost repair. It first identifies the cells that need to be changed (called as **frontier**) and creates a list of new cell assignments and constraints for the selected *frontier* (called as **repair expression**). The *frontier* and the *repair expression* together is called as **Repair Context (RC)**, which **contains the sufficient and necessary information to repair all cells in a frontier**. Given a *Repair Context*, *Holistic Data Cleaning* finds the minimum cost repair as per the used cost function. After each repair, *CH* is updated as repairing a constraint may lead to other violations. This holistic approach of **considering all constraints on the connected cells** reduces the number of changes leading to efficient and minimal cost repair. Detecting violations and building *CH* is the most expensive step of the algorithm. For a dataset of size D and a *DC* having c attributes, the time-complexity of the naive algorithm is $O(D^c)$. Optimization is done by dividing data into blocks of size B, and the *DC* violation is checked on each of the blocks making the time complexity $O(B^c)$. To repair the violations, the cell value can either be changed to a pre-existing value in domain of the attribute or a **fresh value** is chosen. The performance of *Holistic Data Cleaning* is evaluated on real-world and synthetic dataset. The framework **outperforms the techniques which use constraints in isolation** in all the cases. For random errors, the recall of the algorithm is low. The framework produces low precision results for dataset having errors in numeric attributes as finding accurate values for numeric attribute is a challenging task. [6]

Katara is a data cleaning framework that uses **knowledge base (KB)** and **crowdsourcing** to clean data. It uses **table patterns** to map a table to the available **knowledge base**. Each *table pattern* is a **directed graph**, with node representing column and directed edge between them representing relationship between the nodes. *Katara* uses **crowd to validate the generated table patterns**. Once the table pattern is identified, *Katara* annotates data points as: ***KB validated*** - *correct tuple validated by KB*, ***KB and crowd validated*** - *validated by KB and crowd*, and ***erroneous tuple*** - *identified by KB and crowd*. *Katara* formulates *KB K* as being build of individual **resources** (each real-world entity is associated with a unique resource) and **properties (relationships)** (represents relationship between two resources). *Resource* representing a **set of objects** is called a **class**. Each attribute A_i has certain **type** ($type(A_i)$), and any attribute pair (A_i, A_j) are related through a property P where one of the attributes is a **subject** and another one an **object**. A **table pattern** φ for a table T is a **directed graph** $G(V, E)$, where **each node** $u \in V$ **corresponds to an attribute (possibly typed)** in T and **edge** $(u, v) \in E$ **represents relationship** P **between the attributes** u and v. A tuple $t \in T$ matches a pattern φ if there exist a **one-to-one mapping between attributes of tuple** t **with nodes in** φ, with the types of tuple value match with the types of nodes (or a subclass of types of nodes), and the property (relationship) tagged to each edge should match the relationship between the corresponding tuple attributes. For example, tuple t_1 in Figure 2(b) matches the pattern φ_s in Figure 2(a), as all the attribute values



	A	B	C	D	E	F	G
t_1	Rossi	Italy	Rome	Verona	Italian	Proto	1.78
t_2	Klate	S. Africa	Pretoria	Pirates	Afrikaans	P. Eliz.	1.69
t_3	Pirlo	Italy	Madrid	Juve	Italian	Flero	1.77

(a) A table pattern φ_s (b) t_1: KB validated (c) t_2: KB & crowd validated (d) t_3: erroneous tuple

Figure 2: Katara: Solution Overview [7]

of the tuple t_1 has one-to-one correspondence with pattern nodes with the properties of edges being equal. A tuple and a pattern is a **partial match** if either type of one of the attribute of the tuple or one of the property of an edge does not match (tuple t_2 in Figure 2(c)). To map a table to a *KB*, *Katara* follows a more general **query based approach** instead of relying on attribute names being meaningful. It poses a query to *KB* K to get the matching type for attribute values and to identify the relationship between a pair. Once probable candidates are identified, *Katara* uses a **tf-idf based ranking algorithm** to score them and pick type and property of attributes and attribute pairs. For example, if a column contains *"Apple"* and *"Microsoft"*, *"Apple"* will be tagged with type *"Company"* and *"Fruit"*, while *"Microsoft"* will only be tagged with type *"Company"* and hence taking precedence while deciding the type of the column. To identify the property (relationship), *Katara* relies on **coherence score**. Every relationship in *Katara* has a *subject* and an *object*, and **coherence score** measures **"how likely an entity of type T appears as the subject/object of the relationship P"**. Given a type T and a pattern P, *Katara* gives a **coherence score** for each of the **subject** ($subSC(T, P)$) and **object** $objSC(T, P)$ of P, where T serves as the source. Given $ENT(T)$ (set of entities in K of type T), $subENT(P)$ (set of entities in K that appears as *subject* of P), $objENT(P)$ (set of entities in K that appears as *object* of P), and N as the total number of entities in K; $Pr_{sub}(P) = \frac{|subENT(P)|}{N}$ gives the **probability of an entity appearing as the subject of** P and $Pr(T) = \frac{|ENT(T)|}{N}$ gives the **probability of an entity belonging to** T. *Katara* then computes **pointwise mutual information (PMI) for the subject** as:

$$PMI_{sub}(T, P) = \log \frac{Pr_{sub}(P \cap T)}{Pr_{sub}(P)Pr(T)} \tag{17}$$

where $Pr_{sub}(P \cap T) = \frac{|ENT(T) \cap subENT(P)|}{N}$ is the probability of entity of type T appearing as the *subject* of P. **PMI is further normalized as**:

$$NPMI_{sub}(T, P) = \frac{PMI_{sub}(T, P)}{-Pr_{sub}(P \cap T)} \tag{18}$$

Katara computes the **normalized PMI** for object similarly and gives the **coherence score** of a pattern as the **sum of tf-idf based score for types** and **coherence scores for subjects and objects**. The system then identifies the **top-k pattern based on the computed scores using rank-join algorithm**. *Katara* uses crowd to validate ambiguous patterns. It poses questions of the format **"What is the most accurate type of the highlighted column?"** and **"What is the most accurate relationship for highlighted columns?"**; along with randomly chosen samples from the table to identify the type and relationship for attributes. To reduce worker errors, each question is asked three times, and the majority answer is accepted. For **data annotation**, *Katara* auto-labels tuples which satisfy all the criteria of the identified pattern (**fully-covered tuples** of Figure 2(b)), and asks crowd to label the tuples which does not satisfy any one of the pattern criteria (**partially-covered tuples** of Figure 2(c)). To suggest possible repairs, *Katara* calculates the **repair cost** of a violation as the sum of the cost of changing values in tuple t to align it with the identified pattern and picks the *top-k* repairs on the basis of *repair cost*. It **leaves the final repair selection for crowd**. Empirical evaluation of *Katara* on real-world data cleaning task shows high precision and recall for *pattern validation* and *discovery*. *Katara's data annotation system* achieves $\geq 95\%$ accuracy for all the datasets with on an average of $\sim 65\%$ of annotations done using *KB*. Suggested *possible repairs* of *Katara* have high precision for the cases when *KBs have enough coverage of the input data.* [7]

DBpedia is an **information extraction framework** that converts Wikipedia content into **Resource Description Framework (RDF)** format. Its dataset comprises 103 **million Wikipedia RDF triples**, and when integrated with external data sources, it expands to 2 **billion RDF triples**. *DBpedia* accomplishes this by mapping existing relational database table relationships to RDF and extracting additional information from Wikipedia article texts and infobox templates. Its **infobox extraction algorithm** identifies **infobox templates** and discerns their structure using pattern matching techniques. *Post-processing* methods are then applied to enhance extraction quality. *DBpedia* datasets can serve as the **knowledge base** for data cleaning algorithms and can be accessed through: *Linked Data, SPARQL protocol*, and *downloadable RDF dumps.* [1]

4 Missing Values, Outliers, Anomalies and Adversarial Examples

Missing values, outliers, anomalies, and adversarial examples pose significant challenges in machine learning and data analysis tasks, potentially leading to biased models and inaccurate predictions if left unaddressed. This section explores a diverse range of representative data cleaning techniques, providing detailed insights into their functionalities and outcomes for handling these data errors.

4.1 Missing Values:

DataAssist is an **automated data cleaning and preparation framework** that performs the cleaning task based on user specific requirements and can be integrated seamlessly with existing **autoML tools**. It gives the end-user options to handle **missing values** by either removing them or utilizing the underlying *SVM model* to identify and apply the suitable imputation technique. *DataAssist* supports various **outlier detection algorithm**, such as **IQR, DBSCAN,** and **Isolation Forest**. Once the outlier detection is done, users are prompted to visualize outliers, and take appropriate action as per the results. For duplicate and inconsistency detection, *DataAssist* has a **learnable similarity function**, which is trained to classify pair-wise objects as *similar* or *dissimilar*. It also provides pre-processing options, such as data transformation, feature extraction and prioritization of the cleaning of essential features. [13]

Data Linter is an **automated ML tool designed to analyze summary statistics** of ML training data to detect potential data errors (**duplicates** and **missing values**), termed as **data lints**, and **recommend essential feature transformations** based on the chosen ML model. Users can also incorporate custom data lint detectors tailored to specific use cases. The tool has three main modules: **LintDetectors, DataLinter**, and **LintExplorer**. LintDetectors are model and error-specific. They are applied to the dataset and require summary statistics, data instances, feature names, and metadata to generate **LintResults** which contain suggested issues or transformations with relevant sample data. *LintExplorer* presents the identified issues along with sample data to the end user. The system includes pre-built data lints for common issues like **duplicates** and **missing values**, as well as generic feature transformations. The tool was evaluated for the usability in real world ML development cycle, and its performance was further evaluated on 600 Kaggle datasets. In the ML development cycle, *Data Linter* suggested a feature transformation which increased the precision of the model from 0.48 to 0.59, and flagged duplicate training data which was missed in the usual run. On the Kaggle datasets, *Data Linter* **identified on an average of** 4 **data lints per dataset, with no false negatives found in manual analyses** of randomly selected instances. [16]

Query-Oriented Data Cleaning (QOCO) is a novel **query-oriented** system that interacts with domain experts (modeled as oracles) to remove or add incorrect or **missing values** from an unclean database based on the results of a query. For an **unclean database** D and the **ground truth database** D_G, **oracle will have the knowledge of** D_G. For any query Q and tuple t, there will be three types of answers - **True Answer**: if $t \in Q(D)$ and $t \in Q(D_G)$; **Missing Answer**: if $t \in (Q(D_G) - Q(D))$; **Wrong Answer**: if $t \in (Q(D) - Q(D_G))$, where $Q(.)$ is the result of query Q on the concerned database and $t \in Q(D)$ means that tuple t exists in the response of query Q over D. Whenever the **oracle says that an answer is wrong or missing, the data base D is cleaned to** $D^{'}$, such that $t \notin Q(D^{'})$ for the wrong tuple and $t \in Q(D^{'})$ for the missing tuple. To get to the database $D^{'}$, a sequence of edits are performed based on the responses from the oracle. The edits can be of two types - **Insertion Edit**: $R(a)^{+}$, if a tuple a is added to the relation R; **Deletion Edit**: $R(a)^{-}$, if a tuple a is deleted from the relation R. The update in any tuple can be done by a **sequence of *deletion* and *insertion* edits**. The updated database after the edit e is represented as $D \oplus e$. The problem of **finding the optimal sequence of edits based on a query** Q, also called as **edit generation problem**, can be formulated as finding the edits $e_1, e_2, ..., e_k$ by having the minimal interaction with the oracle to change the database D to $D^{'}$ such that $Q(D^{'}) = Q(D_G)$, where $D^{'} = D \oplus e_1 \oplus e_2 \oplus ... \oplus e_k$. The notation $Q(D^{'}) = Q(D_G)$ doesn't mean that $D^{'} = D_G$. Instead, it means that **with respect to the query Q, the databases $D^{'}$ is equivalent to** D_G, though $D^{'}$ may still be dirty. Every edit e takes the dirty database D closer to D_G; and any **desired target action via edits** can be achieved by using **oracle's response to a finite number of questions**. The task of removing a wrong answer (finding a deletion edit) and the task of adding a missing tuple (finding an insertion edit) are **NP-hard** and can be reduced to **Hitting Set Problem** [27] and **One-3SAT Problem** [27] respectively. QOCO uses greedy algorithms to find the optimal solutions, and employs binary YES/NO queries to find deletion edits and straightforward questions to identify insertion edits. [2]

Wu et al., 2020 proposed **AimNet**, an **attention-based model tailored for imputing missing values in mixed data sets comprising both discrete and continuous variables**. *AimNet* leverages a novel version of the **dot product attention mechanism** to learn structural properties of the data distribution at the schema level. It directly processes raw tabular data, requiring minimal pre-processing (**mapping discrete values to trainable embeddings, and z-score normalization for continuous values**). AimNet is trained using **self-supervised gradient descent-based end-to-end learning**. Given a dataset D having schema R, and a **ground truth database** D^*, the goal of **missing data imputation** is to find the imputed version $\tilde{D}$ of D, such that for every tuple $\tilde{t_i} \in \tilde{D}$, all the cell values should be same as the corresponding tuple $t_i^* \in D^*$. The set of attributes $A \subseteq R$ for which missing values are there are called **target attributes** and the set of remaining attributes $A^{'} \in R \setminus \{A\}$ are called **context attributes**. *AimNet* uses an **autoencoder architecture** tailored for mixed data types, employing a **combination of projections for continuous data** and **contextual embeddings** for discrete data. It leverages a novel variation of the *dot product attention mechanism* to capture structural dependencies within the input data. Attention weights are employed to merge representations of various data coordinates into a cohesive context representation for a target attribute, followed by a non-linear transformation for imputation. Training incorporates a mixed loss function to accommodate diverse data types. Continuous attributes are projected onto a k-dimensional vector space, while discrete attributes undergo learning of a contextual k-dimensional embedding. Continuous attribute A_i with value $x \in \mathbb{R}^{n_i}$ $(n_i \leq k)$ is **standardized across each dimension** to zero mean and unit variance as: $\bar{x}_j = \frac{x_j - \mu_{ij}}{\sigma_{ij}} \forall j = 1, 2, ..., n_i$. A **linear layer** followed by a **ReLU layer** is then applied to the transformed vector to get the k-dimensional embedding:

$$z = \mathbf{B}\sigma(\mathbf{A}\bar{x} + c) + d \tag{19}$$

where σ is the $ReLU$ and $\mathbf{A}, \mathbf{B}, c, d$ are **learnable parameters**. For the embedding of discrete attributes, a **lookup table of discrete context embedding** is learned. The output of the attention layer is a **context vector** of dimension k which **contains the necessary information to perform imputation on the target attribute**. The context vector of continuous target attribute A_j is projected to its dimension d_j by passing the **context vector through a fully-connected ReLU layer** of dimension $k \times k$, followed by a linear transformation to the attribute's dimension d_j. The resulting predicted value for the cell is compared to the actual continuous value using the mean squared loss. For discrete attribute, the inner product between the context vector from the attention layer and the discrete target's vector embeddings for the cell's domain values is computed. A softmax function is then applied to the inner products to generate prediction probabilities for each domain value. **HoloClean framework** with **AimNet** is evaluated on 14 real-world datasets and its performance is compared to state-of-the-art baseline methods. Missing values are injected with a probability of 0.2 (following *MCAR* principle) for each of the attributes in the datasets. **Accuracy** and **normalized root mean square error** are used as evaluation metrics for discrete and continuous attributes. **AimNet surpasses state-of-the-art systems by up to** 43% **in accuracy for discrete attributes and up to** 26.7% **in *normalized-RMS* error for continuous attributes**. Additionally, *AimNet* achieves significantly faster run times, up to 54% lower, compared to other baselines on datasets with large discrete domains. The ablation study by removing attention layer shows that as the number of classes increases, the attention mechanism contributes to over 50% of the prediction accuracy. [43]

4.2 Outliers and Anomalies:

Breck et al., 2019 proposed a novel approach for **anomaly detection** that **uses expert domain knowledge to codify data characteristics in the schema** which are further used to detect anomaly in any batch of ingested data. The system supports **single** and **inter-batch validation**. The data characteristics tagged to the schema is used for single-batch validation. **Inter-batch validation** flags any significant deviation in statistical measures across multiple data batches or between training and serving data. It uses **feature skew, distribution skew** and **scoring/serving skew** to detect inter-batch anomalies. **Feature** and **distribution skew** encode deviation in categorical and continuous features respectively of a new batch of data. **Scoring/serving skew** occurs when out of all the predicted or scored data points by a ML model, only a subset is used for further processing (for example, for a recommender system if out of a total of 100 recommended products only top 10 products are shown to the end-user every time, and further used for re-training based on user's action). This strategy further amplifies the model's poor performance on the discarded recommended data points. Techniques which **quantify the distance between distributions** (KL divergence, cosine similarity etc.), or **goodness of fit statistical tests** (chi-square test etc.) are primarily used to detect these skews. The system's performance was evaluated across over 700 production ML pipelines, revealing significant schema evolution over time. Nearly 90% **of cases saw up to five schema revisions**, indicating stabilization of input data properties after a few iterations. The system demonstrated a **fixing rate of over** 50% **for anomalies in nine out of ten suggested categories**. Notably, the Google Play team utilized the system to uncover a feature skew, resulting in a 2% **increase in the app install rate** on the main landing page of the app store after correction. [4]

ActiveClean is an **iterative data cleaning system that adapts already existing data subset cleaning techniques for a statistical modeling framework with the guarantee of convergence** for **outlier removal** and **attribute transformation**. The system approaches the data cleaning challenge as the issue of the machine learning model being affected by the dirty dataset it's trained on. Consequently, it takes in the model, feature functions, and the dirty data to seek out the **global clean model**. The system guarantees **global convergence for convex-loss models** like SVMs, Linear Regression, and Logistic Regression. The system treats the cleaning operation $C(.)$ as a black-box that takes the dirty record r as the input and either modifies it to give the clean record $r' = C(r)$ or delete the dirty record $\phi = C(r)$. The clean relation R_{clean} can then be denoted as

$$R_{clean} = \cup_{i=1}^{N} C_i(r_i \in R) \tag{20}$$

where R is the dirty data set. The proposed system consists of following modules: **Sampler, Cleaner, Updater, Detector** and **Estimator**. Let $\theta^{(d)}$ be the model trained on unclean relation R, $\theta^{(d_c)}$ the **optimal clean model** and $\theta^{(t)}$ the **current best model at iteration** t. **Sampler** selects a sample of dirty data S from R randomly and passes it to the *cleaner*. **Cleaner** is a user defined module which takes each of the dirty record and gives a clean version of it. **Updater** updates the model using gradient descent and clean batch of data. The process continues until the system terminates as no dirty data is left or some early stopping criteria such as number of iterations is met. **Detector** helps *sampler* in identifying the most likely dirty records and hence helping in fast convergence of the algorithm. Figure 3 shows how *ActiveClean* works. As seen in figure 3, even if a usual model trained on the dirty data achieves global optimum (red star), its performance (shown as red solid dot on blue curve) on ideal clean data will be sub-optimal. This sub-optimal model needs to be further updated to reach the actual global optimum (optimum point on clean data, shown as yellow star on the blue curve). The update in the dirty model is achieved as:

$$\theta^{new} \leftarrow \theta^{(d)} - \gamma \cdot \nabla\phi(\theta^{(d)}) \tag{21}$$

where ϕ is the **convex loss function** and $\nabla\phi$ is its **gradient**. This gradient should be calculated on the entire clean data, which is usually not available, and hence *ActiveClean* approximates it from a sample of cleaned data. *ActiveClean* can also be applied to non-convex loss models but instead of converging to global optima, it will converge to a local optimum point. The performance

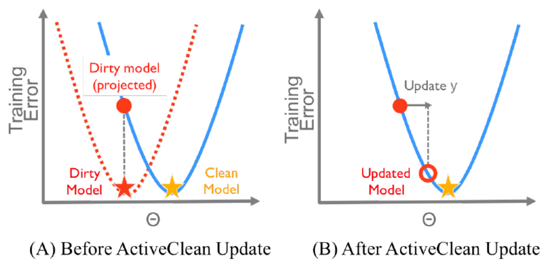


Figure 3: ActiveClean: How it works? [21]

of *ActiveClean* is reported with respect to **model error** (distance between the *ActiveClean* trained model and true model) and **test error** (accuracy of model on test data). The experimental results show that the **ActiveClean converges faster compared to other data cleaning methods** as it uses adaptive detector to exploit the systematic error present in the real-world datasets. [21]

Potter's Wheel is an **interactive data cleaning system** where users can **gradually build transformations by composing and debugging transforms, one step at a time, on a spreadsheet like interface** to flag **anomalies**. It shows the effects of transforms in real-time and if they are not **desirable**, can be **undone**. **Discrepancy detection** is automated and runs in background (even on the newly transformed data). The system suggests the transformations based on the **desired results on the sample data** provided by the end-user. The primary components of *Potter's Wheel* are: **Data Source,**

Transformation Engine, Online Reorderer, and **Automatic Discrepancy Detector**. **Data Source** takes dataset and integrates it with the *Potter's Wheel* system. As the dataset integration starts, a *spreadsheet like interface* appears, which allows users to iteratively sort and reorder data values on sample dataset. **Transformation engine** applies user-specified transforms in real-time on the data loaded on the screen of the *spreadsheet interface* giving a notion of instantaneous transformation. **Automatic discrepancy detector** runs in background that consumes the raw data or its transformed value, and passes them to the suitable **sub-components** (depending on the **inferred structure** of the attribute). Each *sub-component* has appropriate algorithm tagged to it, which is then used to identify the discrepancy in data value. **Tagging data values to a sub-component depends on the domain of data values**. For example, a data value *19 January 06:45* has the structure $\langle number\rangle\langle word\rangle\langle time\rangle$ and needs to be tagged to the *sub-component* associated with domains: $\langle number\rangle, \langle word\rangle$, and $\langle time\rangle$. To define *domains*, *Porter's Wheel* provides a programmable interface, which uses a **user-implemented function named *"match"*** to tag data values to the **domain**. The system comes with a set of **pre-implemented ready-to-use domains**. The *domain* associated with a data value depends on its structure. Given a set of values $v_1, v_2, ..., n_n$ in a column, and a set of *domains* $d_1, d_2, ..., d_m$, **structure extraction** means **choosing the best structure that fits given set of values**. Two extreme *structures* that fit the value *19 January 06:45* are: $\langle \xi^* \rangle$ (representing ASCII string of arbitrary length) and $\langle 19\ January\ 06:45\rangle$. $\langle \xi^* \rangle$ is *concise* and have higher *recall* but the second one is *precise* and will not encode any other value in the column. Hence, the **task of structure extraction** is to find a balance between the properties: **recall, precision**, and **conciseness**. **Description length (DL)** is used as the metric to encode structure quality. **Better structure will have smaller *description length***. *DL* of the structure used to encode a column value is the **length of the structure definition** plus the **length required to encode the values given the structure**. **Conciseness** is directly captured by **length of the structure definition**. The **length required to encode the values given the structure** captures the **precision** (for the values that match the structure). The values in a column that do not match the structure are encoded *explicitly* as themselves capturing a structures' **recall**. The task of choosing the best structure means enumerating all the structures matching the values in the column and choose the one with the lowest *DL* score. Instead of matching the structure on the entire set of column values, *Potter's Wheel* takes a sample of column values for the purpose. The number of structures that are considered for the matching is **pruned by removing redundant structure** like $\langle word\rangle\langle word\rangle$ (this is equivalent to $\langle word\rangle$). Once the *domain* (based on the derived *structure*) of a column is identified, the algorithm tagged to the identified *domain* is used to detect discrepancy in column values. For example, for domain $\langle Integer\rangle$, a typical algorithm that can be used to flag potential errors is identifying the values that are 2 standard deviation away from the mean. **Transformations** in *Potter's Wheel* are applied on an interactive fashion. Users can **construct transformations gradually**, and can **adjust and undo them based on feedback**. *Potter's Wheel* supports inbuilt as well as user-defined *transforms*, which can be defined through examples by *performing them on selected sample values*, and the system picks up the suitable transformations based on these examples. [31]

DBSCAN is a **density based clustering algorithm** which can be used to detect **outliers** and **anomalies** in a dataset. Figure 4 shows a sample density based clustering approach where each probable cluster has a **typical density of points** which is **significantly higher than outside of the cluster**. The idea of *density based*

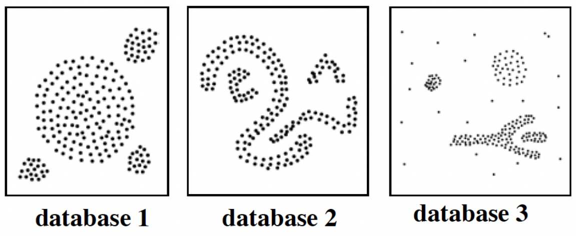


Figure 4: Density Based Clustering: Sample Datasets [9]

clustering is based on the fact that **for each point in a cluster, the neighborhood within a specified radius must have a minimum number of points**. **Distance function** (denoted as $dist(p, q)$ for the points p and q) decides the shape of the neighbourhood. For **Manhattan distance** in $2D$ space, the shape of the neighborhood is rectangular. *DBSCAN* uses a concept of **density-reachable** points to decide the clustering results. For a dataset D, the **Eps-neighborhood** of a point p is defined as $N_{Eps}(p) = \{q \in D | dist(p, q) \leq Eps\}$. A point

p is **directly density-reachable** from point q if p is in *Eps-neighborhood* of q ($p \in N_{Eps}(q)$) and q is a **core point** ($|N_{Eps}(q)| \geq MinPts$). A point p is **density-reachable** from point q, if there exists a chain of points $p_1, p_2, ..., p_n; p_1 = q; p_n = q$ such that p_{i+1} is **directly density-reachable** from p_i. This is an extension of *directly density-reachability* and forms a chain of *density-reachable* points. The **border points** of a cluster may not be *directly density-reachable* from each other but there will always exist a **core point** from which these points will be *density-reachable*. This is termed as **density-connectivity**. Two points p and q are **density-connected** if there exist a point o such that both the points p and q are *density-reachable* from o. For a database D, a **cluster** C with respect to $Eps, MinPts$ is a **non-empty subset** of D, such that

1. $\forall p, q$, if $p \in C$ and q is *density-reachable* from p, then $q \in C$. (*Maximality*)
2. $\forall p, q \in C$, p is *density-connected* to q. (*Connectivity*)

Noise is the set of points in D which are not in any of the identified clusters. Given Eps and $MinPts$, a *cluster* is discovered by taking an arbitrary point (satisfying the *core point* condition) as the *seed* and placing all the points *density-reachable* from the *seed* into the *cluster*. The *clusters* are further merged based on distance between them. The **distance between two set of points S_1 and S_2** is defined as $dist(S_1, S_2) = \min\{dist(p, q)|p \in S_1, q \in S_2\}$. **Two clusters are merged if distance between them is less than** Eps. The time-complexity of the $DBSCAN$ algorithm is $O(n \log n)$. The optimal values of Eps and $MinPts$ are determined using the concept of **d-neighborhood** and by locating the **first point in the first valley of the sorted k-dist graph**. $DBSCAN$ identifies the clusters of arbitrary shape more efficiently and accurately and can successfully isolate the noisy data points. [9]

HoloDetect is a **few-shot learning framework for error detection** (**anomalies**) that uses weak supervision, leveraging less precise or higher-level signals to train **high-quality ML-based error detection system**. The proposed system does not need explicit feature engineering and addresses the extreme data imbalance and diversity in a cohesive manner. It uses a **representation learning framework** to eliminate the need for feature engineering in error detection. The system utilizes a **template ML-model** to learn a comprehensive representation, encompassing **attribute, tuple, and dataset-level features**. To tackle the challenges of heterogeneity and imbalance, the system uses a data augmentation methodology which exploits weakly supervised methods, and **learns data transformations and augmentation policies from a small set of labeled data**. Given a dataset D with C_D being the set of all the cells contained in D, a cell $c \in C_D$ is **erroneous if its unknown true value v^*_c is different from its observed value** v_c, i.e. $v_c \neq v^*_c$. A **training dataset** $T = \{(c, v_c, v^*_c)\}_{c \in C_T}$, where $C_T \subset C_D$ is a set of tuples whose true and observed values are known. For each tuple $c \in C_D$, an indicator $E_c = \{-1, 1\}$ is stored, where -1 means that the cell is erroneous and 1 means otherwise, with e^*_c its **unknown true assignment**. The goal of the error detection system is to find the **most probable assignment** $\hat{e_c}$ for each cell $c \in C_D \setminus C_T$ such that $\hat{e_c} = e^*_c$. **HoloDetect models the distribution of correct and erroneous data**, enabling it to discriminate between valid and erroneous data values. Let I^* represents the clean data distribution (characterized by *attribute, tuple, and dataset-level features*), i.e. $P(v^*_c) \sim I^*$, where erroneous cell values have low probability. Error is added through a conditional probability distribution $R^* \sim P(v_c|v^*_c)$. The goal of *HoloDetect* is to learn I^* and R^*. *HoloDetect* learns a **representation model** jointly with a classifier, and a **generative model** H to approximate I^* and R^* respectively. The data augmentation module of *HoloDetect* uses a *set of transformations* Φ and a policy $\Pi \sim P(\Phi|v_c)$, such that each transformation $\phi \in \Phi$ transforms the original value of a cell c as $\phi(v_c) = v^{'}_c$. The **cell values are treated as strings** and the **transformation function** ϕ introduces errors by **adding, removing, or exchanging characters** as $v = \phi(v^*)$. To select the transformation, the channel samples the transformation from $\phi \sim P(\Phi|v^*) = \Pi(v^*)$ and applies it to v^*. The data augmentation module of *HoloDetect* learns the **noisy channel distribution** R^*, which is specified by Φ and a *policy* Π, using the given example training data set. It uses a **pattern matching based algorithm**, which returns a set of **valid transformations** Φ_e, containing **specialized** (applied at a particular location in a string) and **generic transformations** (can be applied to any position to any input). R^* is then used to **generate additional training examples T_H by transforming some of the labeled set**. The empirical evaluation of *HoloDetect* shows that it consistently outperforms other error detection methods, showing improvements of up to *20 F1 points* in some cases. *HoloDetect* averages a precision of 92% and a recall of 96% across evaluated datasets. [15]

Isolation Forest (iForest) is an **anomaly detection** algorithm, which is built upon anomalies being **"few and different"**. The algorithm uses **Isolation Tree or iTree**, in which **anomalies are isolated closer to the root of the tree, while normal points are isolated at the deeper end of the tree**. *iForest* efficiently identifies anomalies without constructing the full isolation tree for normal points, reducing computational overhead. It doesn't rely on distance or density calculations, eliminating major computational costs. *iForest* creates partitions in dataset to isolate data points. Partitions in *iForest* are created by randomly selecting attributes and split values. The path length from the root to a terminating node indicates the number of partitions required to isolate a point. **Normal points typically have longer path lengths compared to anomalies**. With each tree using different partitions, averaging path lengths across multiple trees yields the expected path length. Increasing the number of trees improves the accuracy of the average path length estimation. A tree node T can either be an **external-node** (no child) or an **internal-node** (with exactly two **daughter nodes** (T_l, T_r) and a *test*). The partition test takes the form $q < p$ where q is an attribute and p is the **split value**. Attribute and **split value** for a *test* is selected randomly from all the set of attributes and a value in between the maximum and minimum value of the selected attribute respectively. Given a sample data $X = \{x_1, x_2, ..., x_n\}$, the algorithm iteratively selects q and p and divides X until the tree reaches a height limit, or no data point is left $|X| = 1$, or all the data points have the same value. An *iTree* is a **binary tree** and a fully grown tree will have a total of n **external nodes** and $n-1$ **internal nodes**. Data points are sorted based on their path length and the **ones with the shorter path lengths are flagged as anomalies**. *Path length* $h(x)$ is the number of edges that need to be traversed to reach a data point. The estimation of **average path length for external node** is same as the **unsuccessful search in BST**, and is given as:

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n} \tag{22}$$

where $H(i) = \ln(i) + \epsilon$ is the **harmonic number**, with $\epsilon = 0.5772156649$ being the **Euler's constant**. The **anomaly score** of a data point x is defined as:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{23}$$

where $E(h(x))$ is the expected value of the *path lengths* $h(x)$ for all the *iTrees*. The *anomaly score* s is monotonically decreasing with respect to *average path length* $h(x)$ and can be used to flag anomalies instead. Value of s closer to 1 means anomaly. Value of s much smaller than 0.5 means data point is normal. If all the data points have *anomaly score s* closer to 0.5, this means that the sample does not have anomalies. *iForest* doesn't need to isolate every normal instance, as it can effectively work with a partial model, especially when dealing with large datasets. *Sub-sampling*, achieved through *random selection of instances without replacement*, helps *iForest* perform well by preventing **swamping** and **masking** effects. **Swamping** occurs when normal instances outnumber anomalies, while **masking** happens when anomalies are concealed by their own abundance. *iForest's* unique approach allows it to build a partial model through sub-sampling, mitigating the challenges posed by swamping and masking. For a *sub-sampling* size of ψ, the the *tree height limit* l can be set as $l = \lceil \ln_2(\psi) \rceil$, as anomalies will more likely to have a path length less than the average tree height. **Increasing sub-sampling size ψ increases processing time and memory requirement without much gain in detection capability**. Setting $\psi = 256$ provides enough samples to perform anomaly detection in most of the cases. **Number of tree** t controls the **ensemble size**. $t = 100$ leads to convergence in most of the cases. To adjust the *average path length* for partially built tree, an *adjustment factor proportional to the size of the tree* is added in the *average path length.* To identify the top m anomalies, the data is sorted in descending order based on the *anomaly score s*. The first m instances in the sorted list represent the top m anomalies. **Kurtosis test** can be used in high-dimensional data setting to enhance the performance of *iForest.* With linear time complexity, low memory requirements, and scalability to high-dimensional large datasets, *iForest* is well-suited for real-world applications. [24]

Raha is a **semi-supervised error detection method** (**anomalies** and **outliers**) that limits the user involvement by auto-configuring underlying error detection algorithms. It **assigns a feature vector** to each data cell, with each **component representing the binary output of a specific error detection algorithm configuration**. The feature vector captures outputs from four main traditional error detection techniques: **outlier detection, pattern violation detection, rule violation detection**, and **knowledge base violation detection algorithms**. Each technique is **automatically configured with a limited set of parameters**, such as thresholds for outlier detection or patterns/rules for violation detection. The exponential complexity of configurations is managed by **setting boundaries and discretizing parameters**. The results from these configurations are aggregated to create a feature vector for each data cell. The cells in each data column are clustered based on these feature vectors, and **users are only required to label one data point from a cluster at a time**. This labeled data is then propagated to all other values within the same cluster, yielding additional noisy training data. Given a dataset D, a set of available error detection algorithms $B = \{b_1, b_2, ..., b_{|B|}\}$, and a **labeling budget** θ_{labels}, the goal of *Raha* is to identify all data errors in D. Individual error detection algorithms can have numerous possible configurations, and each combination of algorithm and configuration is treated as unique **error detection strategy**. For an error detection algorithm b, let the space of configuration be $G_b = \{g_1, g_2, ..., g_{|G_b|}\}$, the subset $S \subseteq B \times G_b$ is the **subset of all possible error detection strategies**. The possible configurations for each each error detection algorithm is selected systematically, generating error detection strategies that mark data cells as errors. *Raha* collects the output of these strategies to create a **feature vector for each data cell**, where each element represents whether a strategy flags the cell as an error or not. *Raha* further post-processes the generated features to **remove non-informative features for each attribute** (column). Individual clusters are generated for each attribute, and cells with similar feature vectors are grouped as same cluster. Users label a sample tuple from a cluster, and *Raha* propagates these labels to other cells in the same cluster (called as **noisy labels**). To manage labeling at the cluster level, the number of clusters k per column should be controlled as per the labeling budget. Smaller k values result in larger clusters, which may contain a mix of dirty and clean cells. Conversely, larger k values create more clusters, demanding more labels from the user. *Raha* uses **hierarchical agglomerative clustering** and let the user decide the number of samples in a cluster as per the labeling budget. In the case, when same data point is labelled by multiple users, **conflict is resolved using majority voting**. Using these labeled data, *Raha* trains **individual classification models for each of the attributes**, which are then used to predict labels for remaining unlabeled cells. Though individual classifiers are trained for each attribute, attribute dependency is captured through features generated by rule and knowledge base based violations. *Raha* uses historical data to decide the important features upfront by utilizing the fact that the similar error detection strategies perform similarly on comparable data domains. For instance, for a column like *"City"*, error detection strategies that were effective on the column *"Capital"* in past datasets would perform better. For this purpose, *Raha* maintains **column profiles** which capture **syntactic** (based on similarity of data distribution) and **semantic similarity** (by overlap of data values) between data columns. Empirical evaluation of *Raha* shows that it surpasses standalone error detection tools across all tested datasets, **achieving an F1 score improvement ranging from** 12% **to** 42%. Its effectiveness decreases marginally with higher user labeling errors. However, the conflict resolution function, which relies on majority voting, helps alleviate this issue to some extent. [26]

4.3 Adversarial Examples:

Grosse et al., 2017 proposed a **statistical tests based adversary detection system** which leverages the distinctiveness of adversarial examples from the expected data distribution. Additionally, they recommend that the ML models can be enhanced by incorporating an extra output dedicated to adversarial examples, effectively training the model to classify them separately. This approach enables the model to recognize and handle adversarial inputs more effectively, enhancing its robustness. In a classification task, the ML system aims to learn the function $f(x) \mapsto y$, where $x \in X$ is the input sample and $y \in Y$ is the corresponding predicted class label. Input x comes from an unknown distribution for each class, denoted as $D^{C_i}_{real}$. The training objective of the ML system is to **approximate these class-wise distributions** (learned as $D^{C_i}_{train}$) as accurately as possible. Adversary attacks the trained ML model $f(\_, \theta)$, by generating the *adversarial sample* $x^{'}$ as close as possible to the original sample x such that the predictions for x and $x^{'}$ are different, i.e. $f(x^{'}, \theta) \neq f(x, \theta)$, where $x^{'} = x + \delta$ with minimum δ. The goal of the adversary is to find a sample from $D^{C_i}_{real}$ that does not follow $D^{C_i}_{train}$. The samples generated for class C_i by an adversary will follows $D^{C_i}_{adv}$ instead such that $D^{C_i}_{adv}$ *is consistent with* $D^{C_i}_{real}$ but $D^{C_i}_{adv} \neq D^{C_i}_{train}$. **Statistical tests that compares two distributions can detect adversarial examples once a sizable batch of adversarial inputs is collected**. An alternative approach is to integrate an extra outlier class, C_{out}, into the learning model. This allows the ML model to classify adversarial examples as C_{out}, as they differ from the learned training distribution, $D^{C_i}_{train}$. Grosse et al., 2017 conducted experiments to demonstrate the efficacy of detecting adversarial examples using statistical tests and incorporating an adversarial class into ML models. Statistical tests reliably identify the benign data with approximately 95% confidence regardless of sample size. For most datasets and models, just 50 adversarial examples suffice for the statistical test to reject the null hypothesis. Individual tests on class-wise separated inputs prove to be more effective, requiring a smaller minimum sample size for confident detection compared to the general statistical test. To study the efficacy of ML models trained to detect adversarial samples, two separate models, one on clean data and one on adversarial examples infused data are trained. While the accuracy of the second model on benign data slightly decreases compared to the first model, it effectively detects adversarial examples. Moreover, the model demonstrate the ability to generalize to various attacker strategies, detecting adversarial inputs crafted using different algorithms than those used to generate the adversarial training samples. [14]

MLClean is a unified framework that deals with **data cleaning, data sanitization**, and **unfairness mitigation** in ML systems. It can be used to detect **adversarial examples** in a dataset. *MLClean* exploits the inter-dependencies of these three processes and integrates them to produce a clean and unbiased dataset. On empirical evaluation, *MLClean* shows *similar accuracy* compared to existing *data cleaning* and *sanitization* methods with significantly better run-time. [41]

Picket is a **self-supervised learning based adversarial sample detection framework** designed to protect against data corruptions in both **training and deployment of machine learning models on tabular data**. During training, it filters out corrupted examples from the training data, while during deployment, it identifies and flags erroneous query data points to a pre-trained ML model. Picket uses **PicketNet**, a **novel deep learning framework tailored for mixed-type tabular data**, which adeptly handles numerical, categorical, and short text entries, aiming to understand the distribution traits of non-corrupted data. *Picket* does not need **access to clean data to learn non-corrupted data distribution**. In self-supervised learning, a prevalent approach involves masking a portion of the input and prompting the model to reconstruct it using the remaining unmasked information. Models utilizing multi-head self-attention mechanisms acquire representations for structured inputs, like tuples or text sequences, by capturing inter-dependencies among different segments of the inputs. This enables various segments to display diverse levels of attention towards each other within the same structured input. *Picket* trains a **self-supervised PicketNet model**, M, to capture clean data feature distributions. During training, *Picket* records **reconstruction losses** across epochs for all dataset points, D. After training, it analyzes *reconstruction losses* of early epochs to identify corrupted points, and constructs data set C by removing them from D. M is then trained on C. *PicketNet* uses a novel **two-stream multi-head self-attention model**, which grasps the distribution of tabular data. Each stream, representing a distinct perspective of the input data, focuses on learning specific aspects. The **schema stream** identifies schema-level dependencies among data attributes, while the **value stream** discerns dependencies among individual data values. *Schema stream* represents **positional encoding** of each attribute. To generate *value stream*, each attribute value is encoded separately. *Categorical attributes* are encoded using a learnable lookup table, which is trained alongside other PicketNet components. *Numerical attributes* are encoded using zero-padded raw value. *Text attributes* are encoded using word embeddings. During training, each data point in D undergoes **attribute masking**, where one attribute is masked at a time and reconstructed using the remaining attributes in the tuple. *Reconstruction loss* specific to attribute types is used: **mean squared error** for *numeric attributes* and **cosine similarity-based cross-entropy loss** for *categorical* and *text attributes*. A **loss-based filtering mechanism**, which **removes samples with high loss** to address random or systematic corruptions and samples with unusually **low loss** to mitigate poisoning attacks, is used to detect and exclude corrupted data. After removal of corrupted data, dataset C is constructed and *PicketNet* is retrained on it. During inference, *Picket* operates in **offline** and **online phases**. Having access to a *classifier* f, data set C and model M, *Picket* builds a **class-wise victim-sample detector** (with the feature space of original features concatenated with *reconstruction loss*) for the given prediction task. A **logistic regression based binary classifier**, one for each class (g_y for class y) is trained as **victim-sample detector**. For the sample x (with prediction $f(x)$), *victim-sample detector* $g_{f(x)}$ is used to mark x as corrupt or non-corrupt. Victim-sample detectors are trained using a dataset containing artificially corrupted data points. Initially, the trained classifier f is applied to all points in C, resulting in a subset C_{cor} where $f(x) = y$, indicating correct predictions. C_{cor} is then partitioned into C^y_{cor} , one for each class y. Artificial victim samples and noisy points are constructed from C^{y}_{cor}, by adding noise to sample x to generate $x^{'}$. If $f(x^{'}) = f(x) = y$, $x^{'}$ is tagged as noisy sample; otherwise, it's labeled as a victim sample. The empirical assessment of *Picket* involves six real-world datasets where various types of noise are deliberately introduced using customized methods. For downstream modeling tasks, *logistic regression, SVM*, and a *fully connected neural network* are used. Across all datasets and noise types (random, systematic, and adversarial), *Picket* consistently outperforms existing methods. [25]

Roth et al., 2019 proposed a method that detects **adversarial examples** irrespective of their origin, as long as they **introduce recognizable patterns in the feature representations of a neural network**. Given a *multi-class* setting with (x^*, y^*) as the input-output pair, $x^* \in R^D$ and $y^* \in \{1, 2, ..., K\}$. Adversarial perturbation applied on input x^* generates $x = x^* + \Delta x$, such that $F(x) \neq y^* = F(x^*)$, where F is the learned classifier. For a **probabilistic classifier with a logit layer for probability scores**, the probability of data point x being classified as class y is given as $f_y(x) = \langle w_y, \phi(x) \rangle$, with w_y being the **class specific weight** and ϕ is the learned feature map. Final prediction is then given as: $F(x) = \arg\max_y f_y(x)$. **Pairwise log-odds between class y and z given input** x is given as:

$$f_{y,z}(x) = f_z(x) - f_y(x) = \langle w_z - w_y, \phi(x) \rangle \quad (24)$$

A **defense strategy against adversarial perturbation** is to induce noise on input samples. For a data point x, a noise component η is added such that $Pr\{F(x + \eta) = y^*\}$ is as large as possible. The **noise-perturbed log-odds** is used to calculate **pairwise log-odds** for class pair (y, z):

$$g_{y,z}(x, \eta) := f_{y,z}(x + \eta) - f_{y,z}(x) \quad (25)$$

The system uses a **z-score standardized version** of pairwise log-odds $(\bar{g}_{y,z}(x))$ as the unstandardized distribution depends on the class-pairs. The perturbations Δx overfits the data, i.e. x; and the effect of perturbation can be undone by adding noise to the sample. For perturbed sample $x^* + \Delta x$, the model prediction will be $F(x^* + \Delta x) = y \neq y^*$. **The added noise η partially counteracts the adversarial manipulation, directing the log-odds and hence the prediction towards the true class** y^*, i.e. $F(x^* + \Delta x + \eta) \rightarrow y^*$. Figure 5 shows how adding noise (darker the color, higher the added noise) to the perturbed adversarial sample (light red) moves it towards the original data point x^* (shown

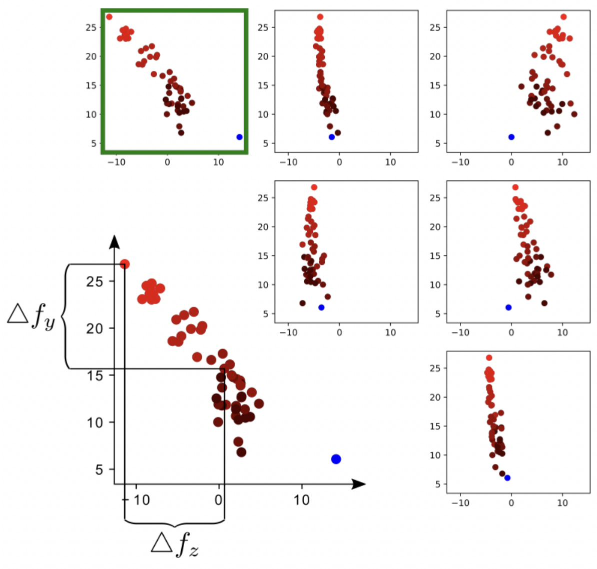


Figure 5: Effect of adding noise to adversarially perturbed sample on logit score [33]

in blue). The **standardized version of pairwise log-odds** $\bar{g}_{y,z}(x)$ can be used as a measure of **whether x classified as y is a manipulated example of class** z. The data point is flagged manipulated if

$$\max_{z \neq y}\{\bar{g}_{y,z}(x) - \tau_{y,z}\} \geq 0 \quad (26)$$

where $\tau_{y,z}$ is a constant. A new simple classifier G which can be used to correct the erroneous classification output is defined as:

$$G(x) = \arg\max_{z}\{\bar{g}_{y,z}(x) - \tau_{y,z}\} \quad (27)$$

with $\tau_{y,y} = \bar{g}_{y,z} = 0$. The selected class z (green box) for the example shown in Figure 5 is as per the classification output of $G(x)$. On empirical evaluation, the proposed method showcases the **adversarial examples detection rate of** $\sim$ 100% **with a false positive rate of** $\sim$ 1%. The proposed correction method **successfully reclassifies almost all detected adversarial samples to their original class**. [33]

5 Entity Matching

Entity matching plays a pivotal role in data cleaning by eliminating redundant or duplicate records within a dataset. The presence of duplicate entries can skew analysis results, compromise data integrity, and lead to inaccurate insights and decisions.

Data Tamer is a scalable **entity matching system** which **groups data source, called as**

***sites*, into classes**, where each class has data source referring to the same real-world entity. **Sites** are identified manually by users categorized as **Data Tamer administrator (DTA)**. *Data Tamer* system uses *dictionaries*, called as **authoritative tables, which have correct information** for the cleaning purpose. All these configurations can be setup using **DTA management console**. The console has options for the *DTA* to specify **actions** such as: **ingestion of new data source, attribute identification**, and **entity consolidation**. The system has the option to configure one more level of human interference called as **Domain Experts (DE)**. *DEs* can be consulted for assistance at any stage of data curation. For **schema integration**, *Data Tamer* system has 4 **in-built experts** (algorithms): **fuzzy string comparison between attribute names, TF-IDF cosine similarity between tokenized data values for attributes, ratio of intersection and union of data values for two attributes**, and **Welch's t-test for pair of attributes having numeric values**; each giving a score between 0 and 1 for the pairwise comparison of attributes. The final score is the weighted average of these scores, which serves as the basis for *schema integration*. The system learns the **de-duplication rules** by presenting the identified probable duplicate tuple pairs for human review in decreasing order of similarity score so that human reviewers can **stop labeling below a certain similarity threshold**. The labeled set of duplicate and non-duplicate pairs are denoted as T_P and T_N. *Data Tamer* has a Naive Bayes classifier based learning module that learns *de-duplication rules* from T_P and T_N. The *de-duplication rules* cane be: rules based on **cut-off threshold on attribute similarities**; rules based on **probability distribution of attribute similarities for duplicate and non-duplicate pairs**. A typical rule takes the form: the probability of the first name and last name having similar values is almost equal to 1 for duplicate tuples. The system uses a **correlation clustering algorithm** to generate consistent results. There can be a case when for a set of three tuples t_1, t_2, t_3, (t_1, t_2) and (t_2, t_3) are marked as duplicates, but (t_1, t_3) as non-duplicate. This inconsistency is resolved using a correlation based clustering algorithm to form clusters in a graph where each node represents a tuple and an edge between two tuples represents duplicates. The algorithm considers all nodes as **singleton clusters** and keep on merging them if the **connection strength** (quantified as the number of existing edges between two clusters divided by the total number of possible edges) is above certain threshold. *Data Tamer* has a separate module, named as **Data Tamer Exchange (DTX)**, which manages the involvement of *DEs* in data curation task at **attribute identification** and **entity consolidation** phases. The system maintains **confidence based ratings** for each *DE* across all the available domains and motivates them to give high quality response with managing the workload for them. On empirical evaluation, *Data Tamer* achieved 90% success rate for *attribute mapping*, with a *modest training* ($\sim$ *50 records)*. For *duplicate identification, Data Tamer* achieved 100% precision with a recall rate of 98.9% for one of the tasks. To evaluate the usefulness of *DTX*, 33 *domain experts* were contacted to beta-test the system of which 18 participated. These experts were asked to rate the system on a scale of 1 to 3, in which the *DTX* received the average score of 2.6. [40]

Corleone is a **hands-off crowdsourcing (HOC) based entity matching (EM)** workflow that uses crowd in all the *EM* steps. Give two relations A and B, *entity matching* is finding two records $a \in A$ and $b \in B$ that *refers to the same real-world entity*. Any *EM* workflow consists of the following steps: **blocking, matching, accuracy estimation,** and **reiteration**. **Blocking** identifies probable match candidates based on defined *heuristic rules*. **Matching** uses a learned ML model or rule-based *matcher* to predict matches from probable match candidates. The next step is the **estimation of match accuracy**. The final step in the *EM* workflow is the identification of **difficult pairs, revising the matcher, and then matching again**. *Corleone* is a *HOC* system that uses crowd for all the *entity matching steps*. It supplies crowd with a **short textual instruction on what it means for two tuples to match**, and *four examples (two positive and two negative)*. Using the instruction and the provided examples, crowd performs the entire task of *entity matching* on two relations A and B. The architecture of *Corleone* is shown in Figure 6. It consists of: **Blocker, Matcher, Accuracy Estimator**, and **Difficult Pairs' Locator**. *Blocker*

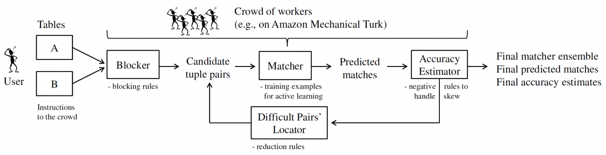


Figure 6: The Corleone Architecture [12]

uses blocking rules to identify probable match candidate which are then used by *Matcher* to train a **random forest model using active learning**.

Accuracy Estimator quantifies *matcher's* performance. *Difficult Pairs' Locator* finds the incorrect matches which are then used to re-train the ML model used in the *matcher*. Different components of *Corleone* can be used in isolation. *Blocking* is applied only when $|A \times B| > t_B$, where t_B is the **blocking threshold**. *Blocker* takes a sample $S \subset A \times B$, such that S has sufficient samples from both the classes (probable matches and non-matches). It then builds initial random forest F using given examples (two positive and two negative), which is then used to find the **informative samples** in S to be labelled by crowd and used to improve F. Finally, **blocking rules** are extracted from the **random forest classifier** naturally by taking the branches that lead to negative class (non-matches). A total of k *blocking rules* are selected for human evaluation based on rule's **precision** and **coverage**. For each of the selected rule, a sample of b **pairs from the covered data points by it are labelled as matched and non-matched** (negative with count n_-) and added to a set X (total sample count n). Based on the labelled examples, the precision of the rule can be estimated as $P = \frac{n_-}{n}$. Rule R is selected if estimated precision $P \geq P_{min}$ and is within the pre-defined bound. The selected rules are added in a set, from which a subset of rules (R) is selected greedily such that the set of pairs obtained (denoted as Z_R) by applying the rules to $A \times B$ is the largest and $|Z_R| \leq t_B$. Given C, the selected candidates by the *blocker*, and initial trained random forest based classifier M, the *matcher* selects a set of **most informative samples** of size q from C to be labelled by the crowd. *Informative samples* are selected based on the *entropy*. For a sample e, *entropy* measures the *disagreement* of different classifiers for the classification task, and is given as:

$$entropy(e) = -[P_+(e)\ln(P_+(e)) + P_-(e)\ln(P_-(e))] \quad (28)$$

where $P_+(e)$ and $P_-(e)$ are the fraction of decision trees labelling sample e as positive and negative respectively. **The higher the entropy, stronger the disagreement, and the more informative the sample is**. The training stops once the confidence of the pre-selected *monitoring set* V converges. The confidence of the *monitoring set* is defined as:

$$conf(V) = \frac{\sum_{e \in V}(1 - entropy(e))}{|V|} \quad (29)$$

The training process stops once $conf(V)$ does not change significantly over a window of training iterations. The empirical results show that *Corleone* achieves comparable (slightly better) accuracy than traditional solutions, requiring no developer in the loop and at a reasonable crowd cost. The components of *Corleone* are modular and each of them can be used in isolation. [12]

Sarawagi et al., 2002 proposed an **active learning based entity matching system (ALIAS)** that **discovers challenging training pairs iteratively**, producing a two-fold reduction in the numbers of required labeled training examples to achieve a desirable level of accuracy. Given a database D, *ALIAS* uses a set of n_f predefined similarity functions F, where each **similarity function** takes record pair (r_1, r_2) as input and gives a *similarity score* between them as output. Given a record pair, **mapper** applies the set of *similarity functions* F on it to produce a n_f **dimensional feature vector**. For the **initial set of labeled training record pairs** $L \times L$, *mapper* generates a **mapped training dataset** L_p, which is used to train the initial learner. The trained initial learner is then used to select a set S of cardinality n out of D_p (mapped pair of records in $D \times D$). Record pairs in S are selected based on the predicted label for the pairs in D_p on the basis of the criteria that the **selected records will produce most information gain when labeled and used for retraining**. Human reviewers are presented with the set of chosen samples S along with their predicted labels, allowing them to correct the incorrect predictions if any. The initial training set is then augmented with the newly-labeled record pairs and used for re-training the classifier. The process continues till a desired level of accuracy is achieved. The output of the *ALIAS system* is a **deduplication function** I, which when given a list of records A, finds the duplicates in the set $A \times A$. For a large dataset D, the number of similarity scores to be calculated will be of the order $O(n_f|D|^2)$. This complexity is reduced by implementing a **grouping strategy** that divides the dataset into smaller groups (based on certain attribute criteria, such as records with same last name in the same group) and forming the pairs within the group. The **learning component** of *ALIAS* is selected based on: **accuracy, interpretability, indexability**, and **training efficiency**. Sarawagi et al., 2002 assessed the performance of three classification methods— *Decision tree, naive Bayes,* and *SVMs*. Among the chosen classifiers, *Decision tree* classifiers produced more interpretable classification rules compared to others. The decision tree's predicates, which involve simple *conjuncts* and *disjuncts* on individual similarity functions, make it more indexable than classifiers like SVMs and naive Bayes. Active learning leads to faster convergence of the used training algorithm. The concept of active learning can be explained using a simple example shown in Figure 7. It showcases a scenario where points on a line need classifying as positive or negative. With a labeled negative sample (r) and positive sample (b), points left of r and right of b are confidently classified. The region between them, labeled the **region of uncertainty**, is where future training points should be selected. Labeling point m, situated in the middle, halves the size of this uncertain region when used for training. *ALIAS* uses a **classifier-independent**

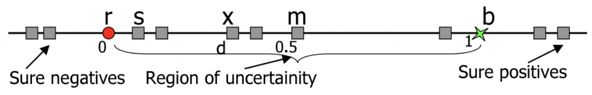


Figure 7: Active Leaning [35]

approach to measure *uncertainty* in prediction, which encodes **disagreement among predictions from a *committee* of N classifiers**. This committee, comprised of slightly varied yet similarly accurate classifiers, offers diverse classification perspectives. Certain data instances receive consistent predictions, while uncertain instances will get different labels representing the *uncertainty*. **Randomization of model parameters, partitioning of training data,** and **attribute partition** can be used to form *committees*. Empirical evaluation of *ALIAS* shows that the active learning based selection of training set reduces the number of labeled training samples on an average by 40× compared to random selection, to achieve the same accuracy level. [35]

Ditto is an **entity matching** system which **frames EM as a sequence-pair classification task and utilizes pre-trained language models** for the purpose. **Ditto EM's pipeline** takes two collections D and $D^{'}$ as input and returns $M \subset D \times D^{'}$ as output, where each entity pair $(e, e^{'}) \in M$ represents the same real-world entity. Pre-trained language models (LMs) used by *Ditto* have simplified architecture tailored for EM, and capture both basic lexical meanings and deeper syntactic and semantic nuances. Pre-training exposes LMs to vast text data, allowing them to develop rich language semantics. The pre-trained LM is then fine tuned for EM using a labeled dataset containing positive (matching) and negative (non-matching) entity pairs. Fine tuning involves: adding task-specific layers to the LM(a **fully connected layer and softmax output for binary classification**); initializing the modified network with the pre-trained LM parameters; training it on the dataset until convergence. Entity pair $(e, e^{'})$, where entity $e = \{(attr_i, val_i)\}_{1 \leq i \leq k}$, is serialized as:

$$\begin{aligned} &serialize(e, e^{'}) ::= [CLS]serialize(e)[SEP]serialize(e^{'})[SEP] \\ &serialize(e) ::= [COL]attr_1[VAL]val_1 ... [COL]attr_k[VAL]val_k \end{aligned} \tag{30}$$

where $[COL]$ and $[VAL]$ are **special tokens** indicating start of attributes and values respectively, and $[SEP]$ is a *special token* separating the two entities. **Ditto's serialization method** doesn't demand uniform schema adherence or attribute matching before executing the matcher. The system can **incorporate domain knowledge** through the pre-processing of input sequences. *Ditto* uses a **recognizer** to identify **span** of a text v, which can be tagged as a specific type, aiding in entity matching process. For example, tagging spans as product IDs or street numbers can guide the system to make more accurate matches and avoid pairing unrelated entities. Given the input text v, **recognizer gives the start and end of a span** with its **span type** in the text: $recognizer(v) = \{(s_i, t_i, type_i)\}_{i \geq 1}$. Once the *span* is identified, the original text can be augmented with tokens representing span and aligned accordingly. For example, *a phone number "(866) 123-4567" may be replaced with "(866) 123- [LAST]4567[/LAST]"*, indicating the last 4 digits of a phone number. To overcome the limit of sequence length, *Ditto* uses **TF-IDF based summarizing technique** to just retain the non-stop words. *Ditto* does training data augmentation using an **augmentation operator** o on a serialized pair s, such that $o(s) = s^{'}$ have the same label l (matching or non-matching) as s. The used operators are **adding or deleting spans, attributes, and swapping the order of entities** in the entity pair. Empirical evaluation of *Ditto* shows that it excels with noisy data and small training sets, achieving state-of-the-art results with just half the labeled data. Pre-trained LMs contribute significantly to its performance, emphasizing language understanding as its strength. Ditto's optimization techniques are also impactful, maintaining competitive training and prediction times despite using comparatively deeper models. [23]

Fusion is a novel **entity matching** system that employs **ordinal regression to model pairwise similarity between records**. It assigns discrete ordinal match levels to record pairs. This approach allows to **generate multiple clusters at different match levels while incurring the full cost of entity matching only once**. *Fusion* handles the **"bad-triplet"** problem (where a graph between three records includes *two positive edges (indicating similarity)* and one *negative edge (indicating dissimilarity)*) by handling the possible reasons - **conflicting in-**

formation and **systematically missing data**, separately. It treats the scenario of **conflicting information as disagreeable-triplet** and **systematically missing data as agreeable-triplet**). *Fusion* follows a standard workflow similar to other entity matching systems, comprising **pre-processing, blocking, pairwise comparison/classification, computing connected components**, and **clustering**. In *pre-processing*, all records are normalized and invalid values are removed. *Blocking* groups records using multiple *blocking indices* to reduce the matching space. *Pairwise comparison* employs an *ordinal regression model* to assign discrete ordered labels to candidate pairs, indicating match likelihood. *Clustering* groups record pairs into connected components, then separates them into final clusters using hierarchical clustering. The resulting hierarchy of disjoint clusters is associated with ordinal match thresholds, allowing flexibility in cluster creation based on business needs. *Fusion* finally **assigns a persistent entity identifier** to each entity. This identifier remains in the system as long as the entity exists, even if the cluster composition changes. When all associated records are deleted, the identifier is removed, and a new one is created for new entities. Given a record pair encoded as feature vector $x^{(i)}$ and the corresponding *target match level* $y^{(i)}$, the goal of the ordinal regression is to predict a match level $z(x) = w^T x$, by learning the weight w and **ordinal threshold levels** $\theta_1, \theta_2, ..., \theta_T$, such that the $loss(z(x), y)$ is minimized. The predicted match level is k if $\theta_k < z(x) < \theta_{k+1}$. In ordinal regression, there is a specific order to the labels and hence the **goal is to minimize the number of crossed thresholds**. The loss (cost) of a single sample is given by aggregating the loss across all the ordinal levels as:

$$C(x,y) = \sum_{l=1}^{T} loss\Big(s(l,y)(\theta_l - z(x))\Big) \tag{31}$$

where

$$s(i,j) = \begin{cases} -1, & \text{if } i < j \\ 0, & \text{if } i = j \\ 1, & \text{if } i > j \end{cases} \tag{32}$$

Fusion uses logistic loss. The **augmented loss function** with **L2 regularization** is

$$J(w,\theta) = \tfrac{1}{2}\sum_{i=1}^{N}\sum_{l=1}^{T} loss\Big(s(l^{(i)}, y^{(i)})(\theta_l - w^T x^{(i)})\Big) + \tfrac{\lambda}{2}||w||^2 \tag{33}$$

Fusion has 5 ordinal levels for match: **hard-conflict, non-conflict, weak-match, moderate-match** and **strong-match**. The *cost function* can be extended to include **cost-sensitive prediction error** by introducing weights for each miss-classification:

$$C(x,y) = \sum_{l=1}^{T} loss\Big(\gamma_{l,y} s(l,y)(\theta_l - z(x))\Big) \tag{34}$$

For example, by assigning higher weight to $\gamma_{weak\text{-}match,strong\text{-}match}$, the precision of *strong-match* can be increased. A cluster C is defined as a set of records $r_1, r_2, ..., r_n$; where each record r_i has a total of M attributes denoted as $(a_1, a_2, ..., a_M)$, with the j^{th} attribute of r_i being $r_i[a_j]$. Attribute j at cluster level is then defined as:

$$A_j = \cup r_i[a_j],\ \forall r_i \in C \tag{35}$$

The similarity between two values for attribute j is calculated using function j as $S_j(a, a^{'})$. For cluster level attribute A_j, **weakest similarity** between any two values in A_j is defined as:

$$S_{min}(A_j) = \min_{a_p, a_q \in A_j} S_j(a_p, a_q) \tag{36}$$

Weakest similarity $S_{min}(A_j)$ can be used to identify and resolve *hard-conflict.* If for any cluster attribute A_j, $S_{min}(A_j) < t_j$ where t_j is a threshold, it can be said that for attribute j in cluster C, there are some values which are dissimilar. *Fusion* uses a variant of **hierarchical clustering-based algorithm** to partition connected components to clusters. Given cluster C and $C^{'}$ having cluster level attributes $\mathbf{A}$ and $\mathbf{A}^{'}$, where $\mathbf{A} = (A_1, A_2, ..., A_M)$, A_j is a cluster level attribute; the *hard-conflict* criteria of $S_{min}(A_j) < t_j$ can be defined using ordinal regressor output as $M(r, r^{'}) < \theta$, where θ is the *ordinal threshold* (i.e. **do not merge cluster if there exists any record pair which violates the merge threshold condition**). Each merge operation evaluates the measure $M(\mathbf{A}, \mathbf{A}^{'})$ associated with the cluster pair C and $C^{'}$. If it falls below the classifier threshold for all cluster pairs, the algorithm stops. This ensures that **no hard conflict exists in the final clusters**, preserving connections between records as far as the classifier permits. *Fusion* retains same identifier for similar cluster by finding the number of overlapping records between two clusters and doing the **optimal cluster mapping** by cluster assignment which **maximizes the total number of intersecting records between** two clustering outcomes. *Fusion* uses a greedy algorithm to find an efficient sub-optimal solution for *optimal cluster mapping.* Empirical evaluation of *Fusion* demonstrates that ordinal regression outperforms logistic regression in predicting fixed ordinal match levels, with a 50% reduction in error when trained on identical feature representations. [44]

Kasai et al., 2019 proposed a **low-resource deep entity matching** system that uses **transfer** and **active learning** at its core. The system uses existing learned entity resolution model on pre-labelled data and then employs *active learning* on the *target dataset* to select *informative examples*, subsequently refining the model through fine-tuning. If a high-resource dataset is unavailable, transfer learning can be skipped, and active learning can be used directly, and vice versa. Active learning targets **high-confidence** and **uncertain** examples, enhancing the precision and recall of the transferred model for the target dataset. The proposed system uses **attribute and record level tf-idf** and **jaccard similarity** based **blocking algorithm** to reduce the number of record pairs for *matching*. *Matching phase* has a **sequence of steps that computes attribute representations, attribute similarity**, and **finally the record similarity**, which is then used by a binary classifier to classify the record pair as {match, non-match}. Each entity record pair is tokenized using existing word embedding techniques. A **bidirectional RNN** processes the tokenized representation of words, and generates **attribute vectors**. These attribute representations are then compared across record pairs by computing the **element-wise absolute difference** to construct attribute similarity vectors (sim_1 and sim_2). These vectors are added to find the overall similarity between the entity record pair. This approach ensures a final *similarity vector* of consistent dimensionality, regardless of the number of attributes. A **multi-layer perceptron (MLP)** is fed with the *similarity vector*, whose output is **normalized using a softmax function** to get final probability distribution. The network is trained to **minimize the negative log-likelihood loss**. The system uses *adversarial transfer learning* to make the *network invariant to idiosyncratic properties of datasets*. To make the network dataset agnostic, a *dataset classifier*, having identical architecture as the *matching classifier*, is utilized to forecast the dataset origin of the input pair. The training objective shifts to the **combined negative log-likelihood losses from both classifiers**. By integrating a *gradient reversal layer* between the *similarity vector* and the *dataset classifier*, the parameters within the *dataset classifier* are trained to discern the dataset, while concurrently training the rest of the network to deceive it. An **iterative active learning algorithm** is used to further fine-tune the network for the dataset of interest. The algorithm identifies *high-confidence* and *uncertain* record pairs from unlabeled data in each iteration and uses it for further training. Given a unlabeled dataset $D^U = \{x_i\}_{i=1}^N$, with $p(x_i)$ being the probability that the record pair x_i is match, the entropy $H(x_i)$ (defined in Equation 37) is used to flag *high-confidence* and *uncertain* record pairs.

$$H(x_i) = -p(x_i)\log p(x_i) - (1 - p(x_i))\log(1 - p(x_i)) \qquad (37)$$

High-confidence record pairs will have $p(x_i) \approx 1$, and hence low entropy. **Uncertain record pairs** will have $p(x_i) \approx 0.5$, and hence high entropy. The naive approach for the selection of the training examples for the next iteration would be to select bottom K record pairs with low entropy as *high-confidence record pairs* and top K record pairs with high entropy as *uncertain record pairs*. This approach may unintentionally favor a specific direction in selecting samples, leading to inconsistent performance. Instead, the algorithm partitions D^U into two subsets: D^U_M (having samples the model predicted as match) and D^U_N (having samples the model predicted as non-match); and picks top/bottom $\frac{K}{2}$ samples based on entropy from each subset. The selected *uncertain examples* will now have **balanced *likely false positives* and *likely false negatives***. Selected *uncertain record pairs* are hand-labeled while the predicted labels for *high-confidence examples* are used for training. Experimental assessment of the system demonstrates that initializing network parameters via transfer learning and employing active learning with a sample selection size of $K = 20$ results in superior performance, particularly in low-resource environments, even when tagged data availability is limited (less than 6% of training data for all datasets). [18]

Magellan is an **entity matching** system, which offers **step-by-step guides for various EM scenarios and provides comprehensive tools to cover the entire EM pipeline** by leveraging Python's data analysis and Big Data stacks for efficient and easy implementation. *Magellan* separates the resolution of EM scenarios into two phases: **development** and **production**. In the development phase, users craft an effective EM workflow, guided step by step for accuracy. In the production phase, the focus shifts to implementation and scaling of the workflow across the entire dataset. Development stage tools leverage an open-source data analysis stack, maximizing available resources, while production stage tools are built on top of a Big Data stack, prioritize scalability. *Magellan* automates each step wherever feasible and provide detailed guidance when automation is not possible. For blocking, *Magellan* suggest users to try increasingly complex blockers and cease when the remaining tuple pairs are sufficiently reduced in number. It **automatically suggests blocking rules and offers users with the option to debug blockers by verifying their output**. Once the blocking output (set of tuple pairs) C is available, *Magellan* suggests a method to select a sample $S \subset C$ for labeling as *match* or *non-match* to train the *matcher*. *Magellan* suggests an **iterative approach for sampling and labeling**. If the user requires a sample S of size n, they select and label a random sample of size k (denoted as S_1) initially. If S_1 contains sufficient matches, the user can infer that the density of matches in C is high, and proceed to randomly sample rest of the pairs from C. However, if the density of matches in S_1 is low, the user must **reconsider the blocking step**, potentially by devising new blocking rules to eliminate more non-matching tuple pairs in C. Users then take the selected labelled data S and create a set of features, which are to be used to train the available set of **learning based matchers**. The data is divided into development and evaluation set and matchers accuracy are evaluated on evaluation set. If a matcher achieves slightly lower accuracy but generates results that are easier to explain, *Magellan* also *highlights that matcher for the user's consideration. Magellan* has the provision to debug certain class of matchers. Its debugger highlights problem with the data, labels, features etc. based on received *false positive* and *false negative* results. Users can also add **rule-based matchers** to further improve accuracy. [19]

Singh et al., 2017 proposed **rule-based entity matching** system grounded on **General Boolean Formulas GBFs**, which offers enhanced interpretability, achieves comparable performance to probabilistic approaches, generates succinct and understandable rules, and can learn from restricted training instances. GBFs use **attribute matching combined by conjunctions, disjunctions**, and **negations** for rule definition. Given two relations R and S having aligned attributes $\{A_1, A_2, ..., A_n\}$ and $\{A_1^{'}, A_2^{'}, ..., A_n^{'}\}$, with records $r \in R$ and $s \in S$, record-level matching is measured by a **boolean predicate** $f(r[A_i], s[A_i]^{'}) \geq \theta$ (*true* means match), where f is a *similarity function*. These attribute matching rules are called **atoms**. **Boolean matching rules** are the *atoms* combined by *conjunctions, disjunctions*, and *negations*. For a given attribute, the proposed system automatically identifies the similarity function and threshold to be used, and under what logic they should be combined. The **optimization metric** that is used to identify the parameters can be selected from a range of options including *F-measure, precision, recall*, and *accuracy*. Alternatively, users have the flexibility to define their own metric using native Python code within the tool. The system features a **user-friendly interface designed to facilitate various tasks** within the entity matching pipeline, including **dataset manipulation, schema matching, customization of entity matching rules**, and **monitoring the progress of ongoing experiments**. The empirical evaluation of the proposed system demonstrates a **comparable F-measure** compared to other existing methods across chosen datasets. [39]

6 Conclusion

The importance of data cleaning is paramount in ensuring the quality and reliability of datasets for machine learning systems. Through an in-depth exploration of existing data cleaning techniques, particularly focusing on integrity constraint violation, outliers, missing values, anomalies, adversarial examples, and entity matching, it becomes evident that **no single data cleaning approach is universally applicable across all cleaning tasks**. Instead, the selection of an appropriate data cleaning technique must be backed by a nuanced understanding of various factors, including the **nature of errors, dataset characteristics, employed error detection and repair techniques, chosen machine learning models, and specific cleaning scenarios**. Experimental studies play a crucial role in guiding this decision-making process, allowing practitioners to assess the effectiveness and suitability of different cleaning methods for the selected use-case. **CleanML** is an initiative to design a framework for a **systematic study on the impact of data errors and cleaning methods on downstream ML models**. It establishes a **thorough and structured approach for assessing the combined task of data cleaning and ML modeling**. Such methods are essential because evaluating individual ML algorithms alone is inadequate for determining the effectiveness of data cleaning on ML outcomes. Some ML algorithms may inherently handle noise better, potentially reducing the need for extensive data cleaning. This approach addresses the challenge of identifying statistically significant results amidst diverse datasets, cleaning techniques, and ML models. Several factors can impact the process of *data cleaning* for a ML system. These factors are: the **dataset** to be cleaned, **error type** in the dataset,

cleaning method (detection and repair algorithms) to be used, used **ML algorithm**, and the **stage** (cleaning of training or test dataset) in the pipeline where cleaning process is integrated. *CleanML* represents these attributes affecting a ML system with respect to *data cleaning* process in the form of a **relational schema**. **Each tuple in the schema represents a unique hypothesis to be tested**. *CleanML relational schema* is shown in Figure 8. *Dataset* represents the in-

R1 (Vanilla)

Dataset	Error Type	Detection	Repair	ML Model	Scenario	Flag

R2 (With Model Section)

Dataset	Error Type	Detection	Repair	Scenario	Flag

R3 (With Model Selection and Cleaning Method Selection)

Dataset	Error Type	Scenario	Flag

Figure 8: CleanML Relational Schema [22]

put data to the ML system. *Error type* attribute represents the type of error to be tested. *Detection* and *Repair* represents the data cleaning methods to be tested for error detection and error repair. *ML Model* represents the *ML algorithm* used in the experiment. *Scenario* represents whether the data cleaning is applied on training or test data. *Flag* represents the result of the experiment (*"P (positive)"*, *"N (negative)"*, and *"S (insignificant)"*). *Schema R1* represents: **"how does cleaning some type of error using a detection method and a repair method affect a ML model for a given dataset?"**. *Schema R2* represents: **"how does cleaning some type of error using a detection method and a repair method affect the best ML model for a given dataset?"**. *Schema R3* represents: **"how does the best cleaning method affect the predictive performance of the best model for a given dataset?"**. Resultant relation *R* after conducting a set of experiments can be queried to reach to a conclusion. For example, if for the *error type "outliers"*, *Flag P* dominates, this means that cleaning *outliers* improves the performance of ML system. If for the *error type "outliers"* and *ML Model "decision tree"*, *Flag S* dominates, this means that cleaning *outliers* does not improve the performance of ML system when *decision tree* is the used *ML Model*. With respect to *scenario* there can be a total of 4 encoding: **A: Model trained and tested on dirty training and test set; B: Model trained on dirty training set and tested on clean test set; C: Model trained on clean training set and tested on dirty test set**; and **D: Model trained and tested on clean training and test set**. Out of a total of 6 possible combinations of these 4 scenarios, only two: **BD** (shows the effect of cleaning the training set on the performance of ML system on the clean test set) and **CD** (how the evaluation of ML model on clean vs dirty test set affects the overall performance of ML system) makes sense. *CleanML* compares these two combinations in each of the conducted experiment. An example experiment is shown in Figure 9. The

s_2

Dataset	Error Type	Detection	Repair	Scenario
EEG	Outliers	IQR	Mean Imputation	BD

	Train on Dirty Training Set		Train on Clean Training Set	
Model	Validation Accuracy	Clean Test Accuracy	Validation Accuracy	Clean Test Accuracy
AdaBoost	0.763205	0.711393	0.718193	0.715176
Decision Tree	0.822621	0.754784	0.796487	0.810414
KNN	0.895481	0.821095	0.948312	0.956386
Logistic Regression	0.638849	0.634179	0.673467	0.668892
Naive Bayes	0.453365	0.457276	0.634745	0.638407
Random Forest	0.918556	0.854695	0.903680	0.907210
XGBoost	0.932098	0.862706	0.920369	0.922786
Metric Pair: (0.862706, 0.956386)				

Train on Dirty Training Set			Train on Clean Training Set	
Random Search Seed	Validation Accuracy of the Best Model	Clean Test Accuracy of the Best Model	Validation Accuracy of the Best Model	Clean Test Accuracy of the Best Model
8006	0.932098	0.862706	0.948312	0.956386
6130	0.930381	0.868046	0.948312	0.956386
5824	0.932098	0.862706	0.920369	0.922786
3659	0.930381	0.868046	0.948312	0.956386
3239	0.932098	0.862706	0.948312	0.956386
Metric Pair: (0.862706, 0.956386)				

Figure 9: CleanML: Sample Experiment with Result [22]

goal of experiment s_2 is to study: **how does the cleaning of "training dataset" for "outliers" through "IQR" detection and "Mean Imputation" affect the accuracy of the ML system on "EEG dataset", irrespective of the *ML Model* used?**. The experiment fits in the *R2 semantics*. The reported results in Figure 9 will have **randomness due to *hyper-parameter tuning* and *train-test split***. To handle the randomness due to **hyper-parameter tuning**, for each of the ML Model five experiments with different random seed for hyper-parameter search is conducted. Out of the conducted 5 experiments, the one with the best result is selected. The results and selection procedure for *XGBoost* for scenario *training on dirty training set* and for *KNN* for scenario *training on clean training set* is shown in the third table of Figure 9. To handle the randomness due to **train-test split**, each experiment is repeated for a set of 20 train-test splits and the results are used to test the **statistical significance** of the test. **paired sample t-test** (two-tailed t-test, upper-tailed t-test, and lower-tailed t-test) is used to generate the final result (value of the attribute *Flag*) of the experiment. The idea behind running three paired tests is the fact that the value of one-tailed t-test is used for reporting only if the two-tailed t-test is significant. The case when two-tailed t-test is insignificant is considered as *flag "S"*. Using proposed bench-marking by the application of *CleanML*, it is found that **data cleaning may not necessarily improve the performance of downstream ML Models**. The analysis of results indicates that applying cleaning methods without careful consideration could detrimentally affect model performance. The impact of cleaning duplicates is ambiguous. Cleaning outliers results in either no change or slight improvement in the performance of ML systems, though the effectiveness largely depends on the methods used for detection and repair. Imputation of missing values, if distant from ground truth, may introduce bias in the dataset and diminish the performance of the ML system. [22]

There are multiple methods available for addressing different types of data errors. The complexity and user-friendliness of these data cleaning systems differ, making them important factors to evaluate when choosing the suitable system for a specific use-case. The effectiveness of the selected technique further depends on the specific characteristics of the dataset in question, the chosen error detection and repair techniques, the type of machine learning (ML) model to be employed, and whether the cleaning process pertains to the training or test data. It's evident that no one-size-fits-all approach exists when it comes to data cleaning, as the **optimal choice of cleaning tool** depends on a careful consideration of these diverse factors. Hence, researchers and practitioners must **conduct thorough assessments and experimental studies** to determine the most suitable cleaning technique for their specific scenario, enhancing the reliability and performance of the overall ML system.

References

[1] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[2] Moria Bergman, Tova Milo, Slava Novgorodov, and Wang-Chiew Tan. Qoco: A query oriented data cleaning system with oracles. *Proc. VLDB Endow.*, 8(12):1900–1903, aug 2015.

[3] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 143–154, New York, NY, USA, 2005. Association for Computing Machinery.

[4] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. Data validation for machine learning. In *Proceedings of SysML*, 2019.

[5] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2201–2206, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 458–469, 2013.

[7] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1247–1261, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 315–326. VLDB Endowment, 2007.

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[10] Juliana Freire, A. Bessa, Fernando Chirigati, H. Vo, and K. Zhao. Exploring what not to clean in urban data: A study using new york city taxi trips. *IEEE Data Engineering Bulletin*, 39(2):63–77, 2016.

[11] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The llunatic

data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9):625–636, 2013. 39th International Conference on Very Large Data Bases, VLDB 2012 ; Conference date: 26-08-2013 Through 30-08-2013.

[12] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. Corleone: hands-off crowdsourcing for entity matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 601–612, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Kartikay Goyle, Quin Xie, and Vakul Goyle. Dataassist: A machine learning approach to data cleaning and preparation, 2023.

[14] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples, 2017.

[15] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 829–846, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Nick Hynes, D. Sculley, and Michael Terry. The data linter: Lightweight automated sanity checking for ml data sets. In *""*, 2017.

[17] Ihab F. Ilyas and Theodoros Rekatsinas. Machine learning and data cleaning: Which serves the other? *J. Data and Information Quality*, 14(3), jul 2022.

[18] Jungo Kasai, Kun Qian, Sairam Gurajada, Yunyao Li, and Lucian Popa. Low-resource deep entity resolution with transfer and active learning. *ArXiv*, abs/1906.08042, 2019.

[19] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. Magellan: toward building entity matching management systems. *Proc. VLDB Endow.*, 9(12):1197–1208, aug 2016.

[20] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning, 2017.

[21] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, aug 2016.

[22] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. Cleanml: A benchmark for joint data cleaning and machine learning [experiments and analysis]. *CoRR*, abs/1904.09483, 2019.

[23] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. Deep entity matching with pre-trained language models. *Proc. VLDB Endow.*, 14(1):50–60, sep 2020.

[24] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[25] Zifan Liu, Zhechun Zhou, and Theodoros Rekatsinas. Picket: guarding against corrupted data in tabular data during learning and inference. *The VLDB Journal*, 31(5):927–955, oct 2021.

[26] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 865–882, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Adam Marcus, David Karger, Samuel Madden, Robert Miller, and Sewoong Oh. Counting with the crowd. *Proc. VLDB Endow.*, 6(2):109–120, dec 2012.

[28] Curtis Northcutt, Lu Jiang, and Isaac Chuang. Confident learning: Estimating uncertainty in dataset labels. *J. Artif. Int. Res.*, 70:1373–1411, may 2021.

[29] Curtis G Northcutt, Anish Athalye, and Jonas Mueller. Pervasive label errors in test sets destabilize machine learning benchmarks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[30] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[31] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 381–390. Morgan Kaufmann, 2001.

[32] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, aug 2017.

[33] Kevin Roth, Yannic Kilcher, and Thomas Hofmann. The odds are odd: A statistical test for detecting adversarial examples. *CoRR*, abs/1902.04818, 2019.

[34] Christopher De Sa, Ihab F. Ilyas, Benny Kimelfeld, Christopher Ré, and Theodoros Rekatsinas. A formal framework for probabilistic unclean databases. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[35] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, page 269–278, New York, NY, USA, 2002. Association for Computing Machinery.

[36] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proc. VLDB Endow.*, 11(12):1781–1794, aug 2018.

[37] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[38] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. *Proc. VLDB Endow.*, 8(11):1310–1321, jul 2015.

[39] Rohit Singh, Vamsi Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Generating concise entity matching rules. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1635–1638, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. Data curation at scale: The data tamer system. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[41] Ki Hyun Tae, Yuji Roh, Young Hun Oh, Hyunsu Kim, and Steven Euijong Whang. Data cleaning for accurate, fair, and robust models: A big data - ai integration approach. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, DEEM'19, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. Continuous data cleaning. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 244–255. IEEE Computer Society, 2014.

[43] Richard Wu, Aoqian Zhang, Ihab Ilyas, and Theodoros Rekatsinas. Attention-based learning for missing data imputation in holoclean. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[44] Yan Yan, Stephen Meyles, Aria Haghighi, and Dan Suciu. Entity matching in the wild: A consistent and versatile framework to unify data in industrial applications. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2287–2301, New York, NY, USA, 2020. Association for Computing Machinery.