Auto-Detect: Data-Driven Error Detection in Tables

Zhipeng Huang*
University of Hong Kong
zphuang@cs.hku.hk

Yeye He
Microsoft Research
yeyehe@microsoft.com

ABSTRACT

Given a single column of values, existing approaches typically employ regex-like rules to detect errors by finding anomalous values inconsistent with others. Such techniques make *local* decisions based only on values in the given input column, without considering a more *global* notion of compatibility that can be inferred from large corpora of clean tables.

We propose AUTO-DETECT, a *statistics-based* technique that leverages co-occurrence statistics from large corpora for error detection, which is a significant departure from existing *rule-based* methods. Our approach can automatically detect incompatible values, by leveraging an ensemble of judiciously selected generalization languages, each of which uses different generalizations and is sensitive to different types of errors. Errors so detected are based on *global* statistics, which is robust and aligns well with human intuition of errors. We test AUTO-DETECT on a large set of public Wikipedia tables, as well as proprietary enterprise Excel files. While both of these test sets are supposed to be of high-quality, AUTO-DETECT makes surprising discoveries of over tens of thousands of errors in both cases, which are manually verified to be of high precision (over 0.98). Our labeled benchmark set on Wikipedia tables is released for future research[1].

ACM Reference Format:
Zhipeng Huang and Yeye He. 2018. Auto-Detect: Data-Driven Error Detection in Tables. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196889

1 INTRODUCTION

Data errors in relational tables are ubiquitous, and are much more common than what most would expect or believe. Some studies report striking numbers such as 90% of large spreadsheets (with more than 150 rows) have errors [28, 47]. Even in professionally-produced spreadsheets (by firms like KPMG), recent studies suggest that as much as 24% spreadsheets have errors [48]. It is estimated that the average cell error rate is between 1% to 5% [47, 49]. Furthermore, not only are errors in spreadsheets common, they have also led to numerous "horror-stories" for enterprises, with embarrassing news headlines and substantial financial losses, as evidenced by the growing list of incidents compiled by EuSpRIG [6].

Commercial offerings. Simple versions of error detection are offered as features in various commercial systems. We discuss a few representatives here; more discussions can be found in Appendix A.

Microsoft Excel [1]. Excel pre-defines a set of 9 simple error checking rules, among which the well-known ones are "Number stored as text", and "Formulas inconsistent with other formulas in the region" (when a contiguous region of cells all share the same formula yet one cell in the region uses a different formula).

Trifacta [8]. Trifacta recognizes around 10 built-in "data types" (IP-address, phone-numbers, email, etc.) based on predefined patterns [9]. Values in a column not conforming to patterns associated with a data-type are flagged as errors. In addition, Trifacta offers a rich set of visual-histograms (e.g., distribution of string lengths) for values in a column, which help users identify potential quality issues. Similar functionalities are also available in systems like Paxata [3] and Talend [7].

OpenRefine/GoogleRefine [2]. OpenRefine does not detect errors directly but provides a text clustering feature that groups together similar values in a column, so that users can see whether similar values may be misspelled variations of canonical values (e.g., "Mississippi" and the misspelled "Missisippi" are clustered together).

Existing research in the literature. While existing commercial offerings in the space of error detection are mostly limited to manually-defined rules (partly to ensure high accuracy), in the literature a variety of techniques have been proposed for automatic error detection that broadly fall into two groups.

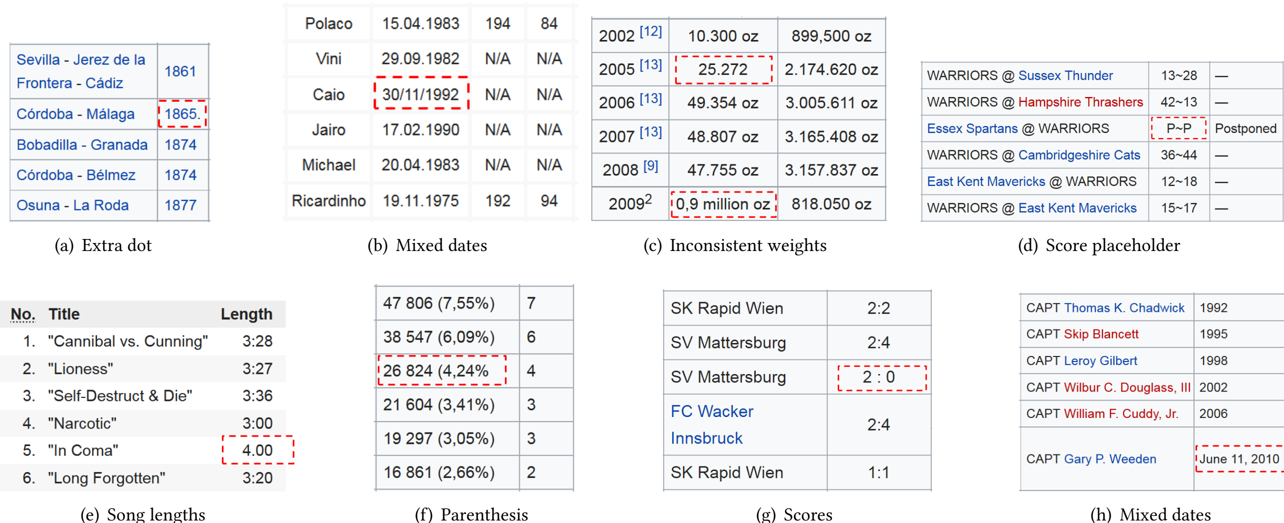
Single-column methods. Single-column approaches detect errors only based on values within an input column. An influential method in this category was proposed in Potter's Wheel [50], which uses minimum description length (MDL) [13] to summarize values in each column into suitable patterns. Data values not conforming to the inferred patterns can be recognized as outliers/errors.

Multi-column methods. When certain multi-column data quality rules (e.g. function-dependencies and other types of first-order logic) are provided (either defined or inferred), such rules can be used to detect non-conforming values. A long and fruitful line of work has been devoted to this area (e.g., [15, 23, 27, 51]).

Design considerations. As general context, this work on error detection is conducted in the context of a *self-service data preparation* project and in collaboration with a commercial data preparation system, which targets the broad audience of end-users such as data scientists and business analysts. The overarching goal is to democratize data preparation by making it "self-service" and accessible to users beyond developers and IT staff. We note that this is in line with a general industry trend reported by Gartner [5].

*Work done at Microsoft Research.
[1] Data is released at https://github.com/zphuangHKUCS/Auto-Detect-released-data.

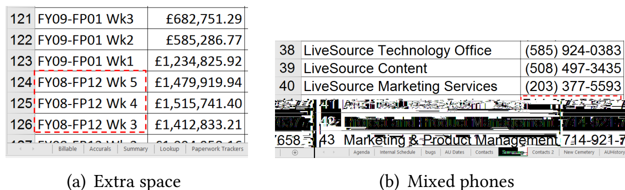
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196889



(a) Extra dot (b) Mixed dates (c) Inconsistent weights (d) Score placeholder

(e) Song lengths (f) Parenthesis (g) Scores (h) Mixed dates

Figure 1: Sample errors (in dashed rectangles) that are detected from real Wikipedia tables (data retrieved on 08/21/2017). There is an estimated 300K such errors in Wikipedia tables, based on a sample of manually labeled tables (Section 4.3).



(a) Extra space (b) Mixed phones

Figure 2: A sample of errors detected from real Excel spreadsheets crawled from a large enterprise.

Since the system in question targets a broad audience of non-technical users (who may not be familiar with functional dependencies and other types of quality rules), there are a few desiderata for error-detection.

First, the detection needs to be *automatic* and works *out-of-box* for end-users, much like standard spell-checkers in *Microsoft Word* or *Google Docs*, as opposed to requiring non-technical users to understand and define rules with first-order logic. We would like it to work like a "spell-checker for data" without needing users to understand or invoke it.

Second, since we aim to automatically detect errors, sometimes without users asking for it, this sets a very high bar in terms of *precision*, as users would quickly lose confidence if the system keeps generating spurious alerts. Therefore, in this work we make high precision (e.g., over 0.95) a primary focus. Note that good recall may also be important in certain scenarios, such as when users want to spend more time on their high-value data sets, and are willing to go down a ranked list to inspect less confident predictions.

Third, although our approach is data-driven and uses large table corpora, the *memory footprint* of our algorithm needs to be modest, as in some cases error-detection will be performed client-only on user machines, which typically have limited memories.

Data errors considered. In this work we focus on single-column error detection, as it is widely applicable, covering a surprisingly large number and variety of real errors. Figure 1 and Figure 2 show a small sample of real errors discovered using Auto-Detect on Wikipedia tables and Excel spreadsheet tables (crawled from a large enterprise), respectively.

It is not hard to see quality problems marked by red rectangles in the context of other values in the columns. Although humans can still understand these tables perfectly fine (Wikipedia tables are after all, mainly intended for human eyeball consumption and are thus "good enough" in that sense), such inconsistencies can post serious challenges for machines. Specifically, programs and queries that process such data are often built with assumptions of how data is expected to be formatted, without accounting for all possible inconsistencies. For example, given the table in Figure 1(b), an aggregate query that groups-by month will try to extract the month field by splitting dates using "." (the expected date format). Inconsistent values (such as the one with "/") can lead to exceptions or even spurious results, polluting downstream pipelines.

Global vs. local reasoning. Given these considerations, the single-column approach from Potter's Wheel [50] seems like a natural choice, as it finds common patterns from values in a column based on MDL, and can then predict values not consistent with patterns as errors. However, we find that this approach falls short in terms of both precision and recall. While we will give a deeper analysis later, the key reason lies in the fact that it only leverages *local* information of values in the input column, without considering a more *global* notion of compatibility.

Consider, for example, a column *Col-1* with values { "0", "1", "2", . . . , "999", "1,000" }. Note that since the last value has a separator "," in an MDL sense the most efficient way to represent these values is to use \d{1-3}, and treat "1,000" as an outlier for separate encoding. The last value would thus be predicted as incompatible and flagged as a potential error, which however is incorrect. Similarly, for a column with mostly integers but a few floating-point numbers such as *Col-2* = { "0", "1", "2", . . . , "99", "1.99" }, the last floating number "1.99" will again be incorrectly treated as an outlier/error based on the MDL principle. We find data columns like these to very common

in the wild (from the web table corpora we find 2.2 million and 1.8 million columns with mixed patterns as described in Col-1 and Col-2, respectively). Since there are many similar scenarios like these, predictions produced based on local MDL would be incorrect more often than acceptable.

The flip side of this is MDL's inability to detect true errors. For instance, given a column *Col-3* with 50%-50% mix of two sets of incompatible date values { "2011-01-01", "2011-01-02", ... } and { "2011/01/01", "2011/01/02", ... }, potentially collected from two data sources. From an MDL's perspective these values are best captured using two patterns \d{4}-\d{2}-\d{2} and \d{4}/\d{2}/\d{2}, and as such a local MDL approach would not find errors. In reality people would intuitively regard the two date formats as incompatible and a possible quality issue.

It is interesting to note that if we change the mixture of the two formats in *Col-3* to 99%-1%, MDL would then predict the values in the 1% bucket to be outliers/errors. This mental exercise underlines the key weakness of the MDL-based approach – conceptually, data compatibility/incompatibility is a *global* notion, irrespective of local value distribution within a specific column. However, MDL is by definition designed to capture *representational efficiency*, which is inherently a *local* notion that changes with intra-column distribution. Although MDL works reasonably well in the absence of *global* information, it does not always align well with humans intuition of data compatibility.

Potter's Wheel is one example *local* approach. There are a large number of local error-detection techniques in the literature [11, 32, 40, 45, 54], many of which measure pattern-distance between values using standard pattern generalizations. Such pattern-level distance is again local and often a poor indicator of compatibility/data-quality, for values like "1,000,000" and "100" can be syntactically dissimilar (and thus predicted as errors by local approaches), but semantically compatible when their co-occurrence in columns is analyzed more globally across all existing tables.

Motivated by these observations, and the fact that large table corpora are now readily available, we propose to predict single-column data quality issues by reasoning about data compatibility more *globally* across large table corpora, denoted by **C**. We formulate this as an optimization problem of selecting a set of generalization languages based on statistics in **C**, whose ensemble can best predict errors, subject to memory and precision requirements.

Consider the *Col-1* above again. Intuitively, using global statistics from **C**, we can observe that numbers with separators "," (i.e., those over 1000) do co-occur often with numbers containing no separators (those under 1000) in same columns, and from that we can conclude that values in *Col-1* are not that unusual/suspicious. Similarly in *Col-2* since we can observe that integers and floating-point numbers also co-occur frequently in same columns, this case is unlikely an error despite a skewed local distribution. Finally, in the case of *Col-3*, because we do not see the pattern \d{4}-\d{2}-\d{2} and \d{4}/\d{2}/\d{2} co-occur often in same columns of **C** (most date columns are either in one format or the other, but are rarely mixed together), we can reliably predict this as a compatibility error.

We evaluate the proposed Auto-Detect approach by testing it on a large number of real Wikipedia tables and enterprise Excel tables. While both are supposed to be of high-quality, our approach detects over *tens of thousands of errors* with over 0.98 precision in both cases, a surprisingly large number considering the fact that these tables are supposed to be clean.

2 PROBLEM STATEMENT

In this section, we will first introduce a few preliminaries before defining our problem.

2.1 Preliminary: Tables and Compatibility

In this work we propose a novel *statistics-based error detection* using large table corpora. Specifically, we use a corpus with over 100M web tables, extracted from Bing's index [18]. Since we focus on single-column error detection, we extract over 350M columns from these tables with some simple pruning. In addition, we obtain Excel spreadsheets from the web and extract around 1.4M spreadsheet columns.

In order to assess the quality of the table corpora, we sample uniformly at random 1000 table columns from Wikipedia tables and general Web tables, respectively, and manually label each column as either "dirty", when *any* quality issue can be identified; or "clean" otherwise. Our evaluation shows that 93.1% of the sampled Web tables are clean, while 97.8% Wikipedia tables are clean. This suggests that the data is reasonably clean and may be leveraged as training corpora for error detection.

Let $\mathbf{C} = \{C_i\}$ be the set of columns extracted from table corpora, where each column $C_i = \{v_j\}$ consists of a set of values. Since **C** is generally clean, the global intra-column co-occurrence of any two given values (v_1, v_2) across all columns in **C** provides reliable statistical evidence of their compatibility. Intuitively, the more v_1 and v_2 co-occur in columns of **C**, the more compatible they should be.

We use a statistical measure called *point-wise mutual information (PMI)* [25] based on information theory to quantify compatibility. Let $c(v) = |\{C \mid C \in \mathbf{C}, v \in C\}|$ be the number of columns with value v, and $c(v_1, v_2) = |\{C \mid C \in \mathbf{C}, v_1 \in C, v_2 \in C\}|$ be the number of columns with both v_1 and v_2. We can define $p(v) = \frac{c(v)}{|\mathbf{C}|}$ be the probability of seeing the value v in a column, and $p(v_1, v_2) = \frac{c(v_1, v_2)}{|\mathbf{C}|}$ be the probability of seeing both v_1 and v_2 in the same column. PMI can be defined as follows:

$$\text{PMI}(v_1, v_2) = \log \frac{p(v_1, v_2)}{p(v_1)p(v_2)} \tag{1}$$

PMI takes the range of $(-\infty, \infty)$. Note that if v_1 and v_2 are co-occurring completely randomly by chances, then $p(v_1, v_2) = p(v_1)p(v_2)$, and thus $\frac{p(v_1, v_2)}{p(v_1)p(v_2)} = 1$, making $\text{PMI}(v_1, v_2) = 0$ for no correlation. If v_1 and v_2 are positively correlated and co-occurring more often, then $\text{PMI}(v_1, v_2) > 0$; otherwise $\text{PMI}(v_1, v_2) < 0$.

PMI can be normalized into $[-1, 1]$ using *Normalized PMI (NPMI)*;

$$\text{NPMI}(v_1, v_2) = \frac{\text{PMI}(v_1, v_2)}{-\log p(v_1, v_2)} \tag{2}$$

Example 1. Let v_1 = "2011", and v_2 = "2012". Suppose we have $|\mathbf{C}| = 100M$ columns, and suppose $c(v_1) = 1M$, $c(v_2) = 2M$, $c(v_1, v_2) = 500K$, respectively. We can compute $p(v_1) = 0.01$, $p(v_2) = 0.02$ and $p(v_1, v_2) = 0.005$, from which we calculate $\text{NPMI}(v_1, v_2) = 0.60 > 0$, indicating a strong statistical co-occurrence. Intuitively, this suggests that the two values are highly compatible in same columns.

Let v_1 = "2011", and v_3 = "January-01", we can run a similar computation to find that $\text{NPMI}(v_1, v_3) = -0.47 < 0$, with

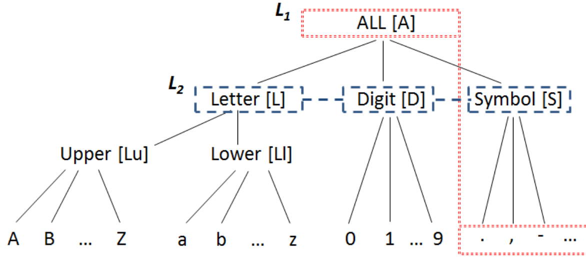


Figure 3: A generalization tree

$c(v_1) = 1M,\ c(v_3) = 2M,$, and $c(v_1, v_3) = 10$. In this case, because v_1, v_3 co-occur highly infrequently relative to their individual frequency/popularity in **C**, the pair can be regarded as incompatible. It would be suspicious if they do co-occur in same columns.

2.2 Generalization For Error Detection

While NPMI is clearly promising for identifying incompatible values, applying it directly on **C** is actually problematic because of data sparsity. For example, given two values "1918-01-01" and "2018-12-31", intuitively they are perfectly compatible since these dates have the same format "\d{4}-\d{2}-\d{2}". However, despite a large **C**, it oftentimes still could not fully capture all possible co-occurrence of compatible values. In this case the two values may never co-occur in the same column in **C**, making NPMI $= -1$, and thus an incorrect prediction that the pair is incompatible.

Our observation is that generalizing values into *patterns* and abstracting away specific values (e.g., "1918" vs. "2018") can overcome data sparsity, because in this example the resulting patterns for both will be the same "\d{4}-\d{2}-\d{2}". Since pattern-level co-occurrence is more reliable in general, our idea is to generalize values into patterns before using Equation 2 to measure the patterns compatibility. As we will see, there exists a large variety of possible generalizations, and a key challenge is to judiciously select the best combination of such generalizations for the best overall prediction.

Generalization. Given an English alphabet $\Sigma = \{\alpha_i\}$, Figure 3 shows one common generalization tree as example[2]. Such generalization trees can be defined as follows.

Definition 1. **Generalization Tree**. A tree H is a generalization tree defined over an alphabet Σ, if each of its leaf nodes corresponds to a character $\alpha \in \Sigma$, and each of its intermediate nodes represents the union of characters in all its children nodes.

Given this one generalization tree H in Figure 3, there are already a variety of ways to generalize a single v using H, because different characters can be generalized into nodes at different levels in H. We define each such generalization as a *generalization language*.

Definition 2. **Generalization Language**. Given a generalization tree H defined over alphabet Σ, a *generalization language* $L : \Sigma \rightarrow H$ maps each character to a tree node.

Given a value $v = \alpha_1\alpha_2 \cdots \alpha_t$ and a generalization language L, we can generalize v by applying the mapping L on each character of v, to produce:

$$L(v) = L(\alpha_1)L(\alpha_2)\cdots L(\alpha_t) \tag{3}$$

[2]We focus on the English alphabet in this work, but generalization trees can be produced similarly for other languages.

Example 2. L_1 and L_2 are two example generalization languages, each of which corresponds to a "cut" of the tree shown in Figure 3.

$$L_1(\alpha) = \begin{cases} \alpha, & \text{if } \alpha \text{ is a symbol} \\ \backslash\text{A}, & \text{otherwise} \end{cases} \tag{4}$$

$$L_2(\alpha) = \begin{cases} \backslash\text{L}, & \text{if } \alpha \in \{a, \cdots, z, A, \cdots, Z\} \\ \backslash\text{D}, & \text{if } \alpha \in \{0, \cdots, 9\} \\ \backslash\text{S}, & \text{if } \alpha \text{ is a symbol} \end{cases} \tag{5}$$

Given two values v_1 ="2011-01-01" and v_2 = "2011.01.02" in the same column, using L_1 we have

$$L_1(v_1) = \text{"\A[4]-\A[2]-\A[2]"}$$
$$L_1(v_2) = \text{"\A[4].\A[2].\A[2]"}$$

where "\A[4]" denotes four consecutive "\A" (the root node of the tree). Computing a PMI-based statistics like in Example 1 but this time at the pattern-level, we can find out that NPMI$(L_1(v_1), L_1(v_2)) = -0.5$, suggesting that the two patterns seldomly co-occur in **C**, and thus likely incompatible.

Using L_2 on the other hand, produces the indistinguishable $L_2(v_1) = L_2(v_2) =$ "\D[4]\S\D[2]\S\D[2]", suggesting that L_2 is ineffective in detecting this incompatibility.

Now consider a different pair of values v_3 ="2014-01" and v_4 ="July-01", using L_1 we have $L_1(v_3) = L_1(v_4) =$ "\A[4]-\A[2]", which would not be able to detect this incompatibility. In comparison, L_2 produces

$$L_2(v_3) = \text{"\D[4]\S\D[2]"}$$
$$L_2(v_4) = \text{"\L[4]\S\D[2]"}$$

Pattern-level co-occurrence reveals that NPMI$(L_2(v_3), L_2(v_4)) = -0.6$, suggesting that they are incompatible. For this particular example, L_2 turns out to be a better choice over L_1.

Note that in this example for simplicity we only analyze one value pair. When given a column C, we can output all incompatible pairs $\{(v_i, v_j) | v_i, v_j \in C, \text{NPMI}(L(v_i), L(v_j)) \leq \theta\}$, or just the most incompatible one for users to inspect.

Example 2 shows that different languages are often complementary in detecting different types of incompatibility, since they are designed to generalize different. This suggests that we need to combine the predictive power of different generalization languages.

It is worth noting that different generalization language from a given H makes trade-offs in a number of dimensions:

(1) *Sensitivity vs. robustness.* Intuitively, the more a language generalizes, the more robust it becomes (to combat data sparsity), but at the cost of becoming less sensitive. For example, consider a trivial language L_{leaf} that does not generalize at all (i.e., keep all characters at the leaf level). L_{leaf} is more "sensitive" in detecting issues, but when data is sparse this can also lead to false-positive predictions (e.g., detecting "1918-01-01" and "2018-12-31" as incompatible like we discussed earlier). On the other extreme, consider a language L_{root} that generalizes everything to the root node. L_{root} is very robust to data sparsity but also becomes "insensitive" to true issues. Naturally, we want to find languages with the right balance of sensitivity and robustness from H, which is partly determined by the amount of training corpus **C** available (the less data we have in **C**, the more we may need to generalize).

(2) *Space consumption.* Apparently, different languages consume different amounts of spaces. L_{leaf}, for example, requires over 100GB

of memory for co-occurrence statistics. The more a language generalizes into higher up nodes in H, the less space it requires. As we discussed earlier, since AUTO-DETECT needs to be client-only, the statistics it relies on need to have a small memory footprint (paging from disks would not be interactive and thus not an option). In practice, AUTO-DETECT is given a memory budget of M that can vary across different scenarios.

The hierarchy H in Figure 3 gives rise to a total of $4^{52}3^{10}3^{33}$ (or 6×10^{51}) possible generalization languages. In practice, we can impose restrictions to require certain classes of characters like [A-Z] to always generalize to the same level. There are still 144 candidate languages with such restrictions. Denote the space of candidate languages by $\mathbf{L}$. As discussed above, individually these languages have different sensitivity/robustness (corresponding to a precision/recall trade-off), and require different amounts of spaces. Furthermore, when multiple languages are used, they can have varying degrees of redundancy/complementarity. A key challenge is to carefully select the best subset of languages in $\mathbf{L}$ for the best error-detection. We formally define the problem as follows.

DEFINITION 3. **Data-driven Single-column Error-Detection**. Given a table corpus $\mathbf{C}$, a generalization tree H, a set of candidate languages $\mathbf{L}$ induced by H, and test columns $\mathbf{T}$ as input; select $\mathbf{L}' \subset \mathbf{L}$ that use corpus statistics derived from $\mathbf{C}$ to detect as many compatibility errors as possible in $\mathbf{T}$, subject to a memory budget of M (e.g. 200MB), and a precision requirement of P (e.g., 0.95).

Note that the problem in Definition 3 is a general statement that can be instantiated differently, as we will see in the next section.

3 AUTO-DETECT

In this section, we study instantiations of the the problem in Definition 3. Note that it leaves two aspects open: (1) how to quantify precision on a test set $\mathbf{T}$ to meet the precision requirement P, and (2) how to aggregate results from a selected set of languages $\mathbf{L}'$.

3.1 Distant-supervision: generate training data

One approach to measure precision is to quantify precision using hand-labeled test columns $\mathbf{T}$, on which the system will run in the future to predict errors. However, it is often difficult if not impossible to anticipate what $\mathbf{T}$ would look like in the future, and even if we have $\mathbf{T}$, labeling it is expensive and do not scale well.

To overcome these issues, instead of relying on *supervised* methods and human labelers to measure precision, we resort to an *unsupervised* approach, known as *distant-supervision* in the literature [46], which has been used in a variety of scenarios such as relation-extraction [46], sentiment analysis [29], and knowledge-base completion [53]. Such techniques build a large amount of training data cheaply, where individual label may not be perfect, but in aggregate the vast amount of training data can often lead to better models compared to a small amount of hand-labeled training data. We apply the general principle here – by automatically building large scale training data with labels, we can estimate precision results of different languages in $\mathbf{L}$ and make informed decisions to select $\mathbf{L}'$.

Specifically, we use *distant-supervision* to automatically construct test case as follows. We select columns $\mathbf{C}^+ \subset \mathbf{C}$ whose values are verified to be statistically compatible using NPMI scores. From these, we sample a column $C_1 \in \mathbf{C}^+$ and take a value $u \in C_1$, and mix u with another column $C_2 \in \mathbf{C}^+$ to produce a synthetic column $C_2' = C_2 \cup \{u\}$. This will, with high likelihood, make C_2' a "dirty" column in which u is the sole value incompatible with others (we detect if u and C_2 are too "similar" and drop such cases). For any $C_2' = C_2 \cup \{u\}$, we can generate a pair of incompatible value $(u, v, -)$, using any $v \in C_2$, where the negative sign indicates the label of the pair. Repeating this process will generate a set of dirty columns $\mathbf{C}^-$, and training examples $\mathbf{T}^- = \{(u, v, -)\}$, all of which are incompatible pairs. Similarly we can generate compatible pairs from $\mathbf{C}^+$ as $\mathbf{T}^+ = \{(v_1, v_2, +) | v_1, v_2 \in C, C \in \mathbf{C}^+\}$. Using $\mathbf{T} = \mathbf{T}^+ \cup \mathbf{T}^-$ as labeled test cases, we can now effectively "run" each language $L \in \mathbf{L}$ on $\mathbf{T}$ and estimate the precision of L using the labels in $\mathbf{T}$.

	$\mathbf{T}^+$					$\mathbf{T}^-$				
	t_1^+	t_2^+	t_3^+	t_4^+	t_5^+	t_6^-	t_7^-	t_8^-	t_9^-	t_{10}^-
L_1	0.5	0.5	-0.7	0.4	0.5	-0.5	0.9	-0.6	-0.7	0.2
L_2	0.5	0.5	0.4	-0.8	0.5	0.9	-0.6	0.2	-0.7	-0.7
L_3	0.4	0.5	0.5	0.6	0.5	-0.6	-0.6	-0.7	-0.5	0.9

Table 1: Generated training examples, where $t_i^+ = (u_i, v_i, +)$, $t_i^- = (u_i, v_i, -)$. Scores are produced based on NPMI after generalization in L_j is performed.

We describe details of this data generation step in Appendix F. We generate a total of over 100M examples in $\mathbf{T}$. Similar to the rationale used in distant-supervision, we hope the large amount of (imperfect) training data would in aggregate enable good language selections.

EXAMPLE 3. Table 1 shows an example $\mathbf{T}$, where $\mathbf{T}^+ = \{t_1^+, \cdots t_5^+\}$ (compatible examples) and $\mathbf{T}^- = \{t_6^-, \cdots t_{10}^-\}$ (incompatible ones). We have $t_i^+ = (u_i, v_i, +)$, and $t_i^- = (u_i, v_i, -)$, corresponding to a pair of values and their label.

From Example 2, the incompatible examples may be generated as t_6^- = ("2011-01-01", "2011.01.02", -), and t_7^- = ("2014-01", "July-01", -), etc., where the value pairs are sampled from different columns in $\mathbf{C}^+$. On the other hand, the compatible examples are generated as t_1^+ = ("2011-01-01", "2012-12-31", +), t_2^+ = ("July-01", "September-20", +), etc., and are sampled from the same columns in $\mathbf{C}^+$.

Given the training corpus $\mathbf{T} = \mathbf{T}^+ \cup \mathbf{T}^-$, we can now estimate precision results for different languages. Specifically, for a given language L_k, and for each example $t = \{u, v, \cdot\} \in \mathbf{T}$, we can generalize values u, v using language L_k, and compute the corresponding NPMI score, henceforth written as $s_k(u, v) = \text{NPMI}(L_k(u), L_k(v))$ for simplicity. Note that this is just like Example 1 but calculated based on patterns generalized using L_k. Table 1 shows example scores for all training examples for three different languages.

For a specific choice of score threshold θ for L_k, all examples with scores below θ can be predicted as possible errors. We can then estimate the precision of L_k on $\mathbf{T}$, by simply computing the ratio of the number of correctly predicted examples to the total number of predictions (those with scores lower than θ). In Table 1, for instance, if we use threshold $\theta = -0.5$ for L_1, then $\{t_3^+, t_6^-, t_8^-, t_9^-\}$ will be predicted as errors. From the ground-truth labels we can see that three such predictions are correct (those with "-") and one is incorrect, thus resulting in a precision of $\frac{3}{4} = 0.75$.

3.2 Aggregate predictions from languages

Suppose we have a set of languages $\mathbf{L}'$, for any $t \in \mathbf{T}$, each language $L_k \in \mathbf{L}'$ produces a different score and a prediction. The next

question is how to aggregate these individual scores in a principled way, to produce an optimized prediction for (u, v).

Because languages are designed to generalize differently, each language will be sensitive to different types of errors. As we illustrated in Example 2 for instance, the L_1 shown in Figure 3 will generalize all non-punctuation to the root node while keeping symbols as is. Therefore it will be sensitive to incompatible punctuation, but not issues in letters and digits. As a result, t_6^- = ("2011-01-01", "2011.01.02", -) in Table 1 can be detected by L_1, but not t_7^- = ("2014-01", "July-01", -). On the other hand, L_2 in Figure 3 can detect t_7^- but not t_6^-.

Naive aggregation such as average-scores ($\text{Avg} s_k(u, v)$) or majority-voting is conceptually inadequate (and indeed lead to sub-optimal experimental results), because for the first case of t_6^- we should completely ignore the prediction of L_2, while in the second case of t_7^- we should ignore the prediction of L_1. Observing the complementarity of L_1 and L_2, intuitively an effective approach is to select both languages from $\mathbf{L}$ (for better coverage). Then to predict a pair of values as incompatible we only need one language to be confident about its incompatibility (with a low $s_k(u, v)$ score), regardless of predictions from other languages (even if they all predict the pair as perfectly compatible). Intuitively this is because each language is designed to be sensitive to different types of errors, and naturally has "blind spots" for certain issues that their generalizations are not designed to detect.

Dynamic-threshold (DT) aggregation. This intuition inspires a first aggregation approach specifically designed for the characteristics of generalization languages that we call *dynamic-threshold.* In this approach, we dynamically determine a separate threshold $\vec{\theta}_k$ for each L_k, and predict all cases below $\vec{\theta}_k$ as incompatible, which can be written as:

$$H_k(\mathbf{T}, \vec{\theta}_k) = \{(u, v) | (u, v) \in \mathbf{T}, s_k(u, v) \le \vec{\theta}_k\}$$

We can define $H_k(\mathbf{T}^+, \vec{\theta}_k)$ and $H_k(\mathbf{T}^-, \vec{\theta}_k)$ similarly. A natural way to aggregate results H_k across all languages is to union their predictions, because as discussed a confident prediction of incompatibility from one language is often sufficient, because other languages may be designed to generalize differently and thus insensitive to this error in question.

This produces a first instantiation of Definition 3, which uses auto-generated $\mathbf{T}$ from distant-supervision, and dynamic-threshold (DT) aggregation. For a given set of $\mathbf{L}'$ and their associated thresholds $\vec{\theta}$, precision and recall can be calculated using the labels in $\mathbf{T}$, defined as:

$$P(\mathbf{L}', \vec{\theta}) = \frac{|\bigcup_{L_k \in \mathbf{L}'} H_k(\mathbf{T}^-, \vec{\theta}_k)|}{|\bigcup_{L_k \in \mathbf{L}} H_k(\mathbf{T}, \vec{\theta}_k)|}$$

$$R(\mathbf{L}', \vec{\theta}) = \frac{|\bigcup_{L_k \in \mathbf{L}'} H_k(\mathbf{T}^-, \vec{\theta}_k)|}{|\mathbf{T}^-|}$$

Definition 4. **Error-Detection with DT aggregation**. Given a corpus of table columns $\mathbf{C}$, a generalization tree H, and a set of candidate languages $\mathbf{L}$ induced by H, select $\mathbf{L}' = \{L_k\} \subset \mathbf{L}$ and their corresponding threshold scores $\vec{\theta}_k$, such that the DT aggregation can detect as many compatibility errors as possible on the training set $\mathbf{T}$, subject to a memory budget of M, and a precision requirement of P. Formally,

$$\begin{aligned} \underset{\substack{\mathbf{L}' \subseteq \mathbf{L} \\ \vec{\theta} \in [-1,1]^{|\mathbf{L}'|}}}{\text{maximize}} \quad & R(\mathbf{L}', \vec{\theta}) \\ s.t. \sum_{L_i \in \mathbf{L}'} & size(L_i) \le M \\ & P(\mathbf{L}', \vec{\theta}) \ge P \end{aligned} \quad (6)$$

It can be shown that this problem is not only NP-hard but also hard to approximate, intuitively because there are too many degrees of freedom (i.e., both $\mathbf{L}'$ and $\vec{\theta}$).

Theorem 1. The problem in Definition 4 is NP-hard and cannot be approximated within a factor of $2^{(\log n)^\delta}$ for some δ, unless $3SAT \in DTIME(2^{n^{\frac{3}{4}+\epsilon}})$.

This hardness result is obtained using a reduction from the densest k-subhypergraph problem (a proof can be found in Appendix C).

With this negative result, we are unlikely to get algorithms with good approximation guarantees. In light of this, we propose a more tractable formulation that optimizes $\mathbf{L}'$ using predetermined $\vec{\theta}$.

Static-threshold (ST) aggregation. Instead of allowing each language $L_k \in \mathbf{L}'$ to pick a separate threshold θ_k, and optimize the union of the predictions in $\mathbf{L}'$ to maximize recall while maintaining a precision P, we can instead require each language $L_k \in \mathbf{L}'$ to be of at least precision P on $\mathbf{T}$. This is equivalent to finding a $\vec{\theta}_k$ such that:

$$P_k(\vec{\theta}_k) = \frac{|H_k(\mathbf{T}^-, \vec{\theta}_k)|}{|H_k(\mathbf{T}, \vec{\theta}_k)|} \ge P \quad (7)$$

Note that because labeled examples in $\mathbf{T}$ is generated in Section 3.1, given a precision requirement P, we can compute the $\vec{\theta}_k$ required for each language L_k as

$$\text{argmax } \vec{\theta}_k, \text{ s.t. } P_k(\theta') \ge P, \ \forall \theta' < \vec{\theta}_k \quad (8)$$

With this, we can uniquely determine $\vec{\theta}_k$ for a fixed P. We write $H_k(\mathbf{T}^-, \vec{\theta}_k)$ as H_k^- for short to denote the set of incompatible examples covered by L_k (and likewise H_k^+), when the context of P and $\vec{\theta}_k$ is clear.

Example 4. Continue with Example 3 and Table 1. Suppose we are given a precision requirement $P = 0.75$. Based on Equation (8), for L_1 we can get $\vec{\theta}_1 = -0.5$, because with this threshold we have $H_1^+ = \{t_3^+\}$ and $H_1^- = \{t_6^-, t_8^-, t_9^-\}$, producing a precision result of $P_1(\vec{\theta}_1) = \frac{|H_1^+|}{|H_1^+|+|H_1^-|} = \frac{3}{4} = 0.75$.

Similarly, from Table 2, we can compute that for L_2, the desired threshold is $\vec{\theta}_2 = -0.6$, for which the $H_2^+ = \{t_4^+\}$, $H_2^- = \{t_7^-, t_9^-, t_{10}^-\}$ and the resulting precision is 0.75. Finally for L_3 the $\vec{\theta}_3$ can be computed as -0.5, where $H_3^+ = \emptyset$, $H_3^- = \{t_6^-, t_7^-, t_8^-, t_9^-\}$ for a precision of 1.

Now that $\vec{\theta}_k$ for each language L_k is individually determined, in the optimization problem we can focus on selecting a subset $\mathbf{L}' \subset \mathbf{L}$ that maximizes the coverage on $\mathbf{T}^-$.

Definition 5. **Error-Detection with ST aggregation**. Given a corpus of table columns $\mathbf{C}$, a generalization tree H, and a set of candidate languages $\mathbf{L}$ induced by H, select $\mathbf{L}' = \{L_k\} \subset \mathbf{L}$, where each L_k has a precision requirement of P, such that the union can

Algorithm 1 Greedy algorithm for Auto-Detect

1: $G \leftarrow \emptyset$, $curr_size \leftarrow 0$, $\mathbf{L}_C \leftarrow \mathbf{L}$
2: **while** $\mathbf{L}_C \neq \emptyset$ **do**
3: $\quad \mathbf{L}'_C \leftarrow \{L | L \in \mathbf{L}_C, size(L) + curr_size \leq M\}$
4: $\quad L^* \leftarrow \text{argmax}_{L_i \in \mathbf{L}'_C} \frac{|\bigcup_{L_j \in G} H_j^- \cup H_i^-| - |\bigcup_{L_j \in G} H_j^-|}{size(L_i)}$
5: $\quad G \leftarrow G \cup \{L^*\}$
6: $\quad curr_size \leftarrow curr_size + size(L^*)$
7: $\quad \mathbf{L}_C \leftarrow \mathbf{L}'_C - L^*$
8: $L_k \leftarrow \text{argmax}_{L_i \in \mathbf{L}, size(L_i) \leq M} |H_i^-|$
9: **if** $|\bigcup_{L_j \in G} H_j^-| \geq |H_k^-|$ **then**
10: $\quad$ **return** G
11: **else**
12: $\quad$ **return** $\{L_k\}$

detect as many compatibility errors as possible on the training set $\mathbf{T}$, subject to a memory budget of M. Formally,

$$\begin{aligned} & \text{maximize}_{\ \mathbf{L}' \subset \mathbf{L}} \ R(\mathbf{L}', \vec{\theta}) \\ s.t. \ & \sum_{L_i \in \mathbf{L}'} size(L_i) \leq M \\ & P_k(\vec{\theta}_k) \geq P \end{aligned} \tag{9}$$

Theorem 2. The problem defined in Definition 5 is NP-hard.

A proof of this can be found in Appendix D.

Approximation Algorithm for ST-aggregation. Despite its hardness, the problem of ST-aggregation is more tractable than DT-aggregation. We use the greedy algorithm in Algorithm 1 inspired by [39], which has a constant approximation ratio. The first part of this algorithm (line 2-7) iteratively find a language L^* from the candidate set $\mathbf{L}_C$, whose addition into the current selected set of candidate language G, will result in the biggest incremental gain, defined as the coverage of new incompatibility cases divided by language size, written as $L^* = \text{argmax}_{L_i \in \mathbf{L}'_C} \frac{|\bigcup_{L_j \in G} H_j^- \cup H_i^-| - |\bigcup_{L_j \in G} H_j^-|}{size(L_i)}$. We iteratively expand the candidate set G using L^*, until no further candidates can be added without violating the memory constraint. At the end of this, we additionally compute the best single language $L_k = \text{argmax}_{L_i \in \mathbf{L}, size(L_i) \leq M} |H_i^-|$ (line 8). Finally, we compare the coverage of L_k and G, and return the better option as the result $\mathbf{L}'$ (line 9-12).

Lemma 3. *The procedure in Algorithm 1 has an approximation ratio of $\frac{1}{2}(1 - \frac{1}{e})$, for the optimization problem described in Definition 5.*

A proof of this can be found in Appendix E, which leverages techniques for the budgeted maximum coverage problem [39].

Example 5. Continue with Example 4 and Table 2, suppose the memory size constraint M = 500MB, precision requirement P = 0.75, we can compute the thresholds $\vec{\theta}_k$ and their coverage H_k^+ as in Example 4.

Using Algorithm 1, we first pick L_1 into G because it has the largest $\frac{|H_k^-|}{size(L_k)}$. In the second round, we can only pick L_2 into G because if L_3 is selected the memory budget would be violated (200 + 400 > 500). We now have $G = \{L_1, L_2\}$ as our first candidate solution, which covers all five cases in $\mathbf{T}^-$. Then, we pick the best singleton language as a second candidate solution, which in this case would be $\{L_3\}$, because it has the best coverage (4 cases in $\mathbf{T}^-$). We can compare the two candidates and use $\{L_1, L_2\}$ as it outperforms $\{L_3\}$. This procedure guarantees a $\frac{1}{2}(1 - \frac{1}{e})$-approximation discussed in Lemma 3.

	size	θ_k	H_k^-	H_k^+	P_k
L_1	200MB	-0.5	$\{t_6^-, t_8^-, t_9^-\}$	$\{t_3^+\}$	0.75
L_2	300MB	-0.6	$\{t_7^-, t_9^-, t_{10}^-\}$	$\{t_4^+\}$	0.75
L_3	400MB	-0.5	$\{t_6^-, t_7^-, t_8^-, t_9^-\}$	$\emptyset$	1.0

Table 2: Example of language selection

Once a set of languages $\mathbf{L}'$ is selected, given a new pair of values (v_1, v_2), we can predict them as incompatible if and only if:

$$\exists L_k \in \mathbf{L}', s_k(v_1, v_2) \leq \vec{\theta}_k$$

3.3 Statistical Smoothing of Rare Events

Recall that we use Equation 2 to compute NPMI as the compatibility of two patterns $L(v_1)$ and $L(v_2)$. Such computation is known to be reliable when we have seen enough data with large occurrence count of $c(L(v_1))$ and $c(L(v_2))$. However, due to data sparsity, in some cases $c(L(v_1))$, $c(L(v_2))$ and $c(L(v_1), L(v_2))$ all $\rightarrow 0$. In such case NPMI scores computed will fluctuate substantially with small changes of $c(L(v_1), L(v_2))$.

One approach is to "smooth out" co-occurrence counts using a technique known as smoothing in the NLP community [20]. We use the Jelinek-Mercer smoothing [58], which computes a weighted sum of the observed $c(L(v_1), L(v_2))$ and its expectation assuming independence $E(L(v_1), L(v_2)) = \frac{c(p_1) \cdot c(p_2)}{N}$, where N is the total number of columns.

$$\hat{c}(L(v_1), L(v_2)) = (1 - f) \cdot c(L(v_1), L(v_2)) + f \cdot E(L(v_1), L(v_2)) \tag{10}$$

where f is the smoothing factor between 0 and 1. Our experiments suggest that smoothed computation of compatibility substantially improves quality results.

3.4 Sketch-based Memory Optimization

As discussed in Section 2, for each language L, in order to compute NPMI between two patterns $L(v_1), L(v_2)$, we need to maintain two types of statistics in memory: (i) the occurrence count of pattern $L(v_1)$ and $L(v_2)$ in $\mathbf{C}$, respectively, and (ii) the co-occurrence count of $L(v_1)$ and $L(v_2)$ in same columns in $\mathbf{C}$. Note that storing co-occurrence counts in (ii) for *all* pairs with non-zero values as dictionary entries $(L(v_1), L(v_2)) \rightarrow Cnt_{12}$ is clearly expensive, because for many candidate languages there exist hundreds of millions of such pairs.

Storing these co-occurrence counts as dictionaries for each language often requires hundreds of MB and multiple GB. We find that a configuration with a good combination of languages for high recall often require a substantial memory budget (e.g., 4 GB), which may still be too heavyweight as this feature is intended for client-side. To further optimize the memory requirement, we use a probabilistic counting method called count-min (CM) sketch [26], initially developed for estimating item frequencies in streaming data using sub-linear space.

Recall that CM sketches maintain a two dimensional array M with w columns and d rows (where wd is substantially smaller than the total number of items for space reduction). Each row $i \in [w]$ is associated with a hash function h_i from a family of pairwise

independent H. When a key-value pair (k, v) arrives, we increment the entry at row i, column position $h_i(k)$, written as $M[i, h_i(k)]$, by v, for all row $i \in [w]$. At query time, the estimated for a key k is

$$\hat{v}(k) = \min_i M[i, h_i(k)]$$

It can be shown that with $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln 1/\delta \rceil$, we can guarantee $\hat{v}(k) \le v(k) + \epsilon N$ with probability $1-\delta$, where $N = \sum_{k \in K} v(k)$ is the total item values. In other words, with high probability $\hat{v}(k)$ will not overestimate its true value $v(k)$ by too much.

We adapt CM sketch to improve space required for storing co-occurrence. Specifically, we show empirically that the co-occurrence counts in real table corpus follows a power-law, which allows a sharper bound of accuracy to be used.

Applying CM sketches to compress co-occurrence dictionaries drastically reduces memory sizes used by language, often by orders of magnitude (e.g., from 4GB to 40MB), without much impact on counting accuracy or precision/recall loss in error detection, as we will verify in our experiments.

4 EXPERIMENTS

4.1 Datasets

We use five different table corpora in our experiments for training and testing. We use training to refer to the process of deriving co-occurrence statistics with that corpus, which is then used to optimize Auto-Detect for selecting languages $\mathbf{L}'$ and caliberating θ_k as described in Section 3; and we use testing to refer to executing $\mathbf{L}'$ on the corpus to predict errors.

This will test how well Auto-Detect generalizes across tables with different characteristics, and more specifically training on one corpus (e.g., public web tables) and testing in a very different corpus (e.g., proprietary enterprise spreadsheets), which is clearly more challenging and useful than training and testing on disjoint subset of the same corpus.

- **WEB** is a set of over 350M table columns from web tables, which are extracted from Bing's index.
- **WIKI** is a subset of WEB within the wikipedia.org domain, with a total of around 30M table columns. As one could expect, WIKI is of high quality since these Wikipedia pages are collaboratively edited by millions of users[3], which often ensures a high standard of quality. We use 100K randomly sampled WIKI columns as a difficult test corpus to see if we can identify quality issues from these tables (they are obviously excluded from WEB when used for training).
- **Pub-XLS** is a corpus of public spreadsheet files (e.g., .xls and .xlsx) crawled from the web, with a total of 1.4M columns.
- **Ent-XLS** is a corpus of proprietary spreadsheet files crawled from a large enterprise, with 3.2M columns. These are sophisticated commercial-grade spreadsheets with complex designs and formulas. Like WIKI we expect it to be of high quality, so we also sample 100K columns as a testing corpus.
- **CSV** is a collection of 26 public available spreadsheet files that we compiled from online sources (e.g., tutorials, demos and online courses) that are known to have quality issues. These were used as demos by various data cleaning systems. We hand-labeled quality problems in these files that result in a total of 441 test columns.

We use WEB and Pub-XLS as training corpora, because (1) they are large corpora with reasonably clean tables, and (2) these tables

[3] https://en.wikipedia.org/wiki/Wikipedia:Who_writes_Wikipedia

	Train		Test		
name	WEB	Pub-XLS	WIKI	Ent-XLS	CSV
#col	350M	1.4M	100K	100K	441

Table 3: Summary of table corpora used.

are publicly available so Auto-Detect can be replicated by others. Furthermore, because these corpora already cover a diverse variety of content, we can reasonably hope that Auto-Detect trained on these corpora can generalize and work well across different testing corpora (WIKI, Ent-XLS and CSV). Table 3 provides a summary of the statistics of these training/testing corpora.

The partially-labeled WIKI data set, as well as raw results of all algorithms compared in this work, are now released and publicly available[4]. A key reason why we choose to release this benchmark data is to address the general lack of large-scale benchmarks for error-detection in the current literature. We hope this data will help reproducibility and facilitate future research.

4.2 Methods Compared

- **Fixed-Regex (F-Regex).** Commercial systems such as Trifacta and PowerBI employ predefined regex to first detect data types of a given column. Values not consistent with known regex will be marked as potential errors. For example, a regex for email may be ".*@.*\..*", and values not consistent with this regex will be predicted as errors. For this approach we use regex patterns of a commercial system. Errors are ranked by confidence to produce precision/recall, where the confidence is defined as the percentage of values in the same column that are actually consistent with known regex.
- **Potter's Wheel (PWheel)** [50]. Potter's Wheel is a pioneering system for interactive data cleaning. One of its novel features is the ability to generate suitable patterns for values in a column using the MDL principal. Values not consistent with the produced patterns can be predicted as outliers/errors. We implement the approach reported in [50]. Similar to F-Regex, errors are ranked by the percentage of values consistent with inferred patterns.
- **dBoost** [45]. dBoost is a type-specific method for known types using predefined expansion rules. For example, observing that date time can be stored as integers, all integer columns can then be interpreted as dates and expanded into fields such as year, month, day, day-of-week, etc. Distributions of these derived fields can then be analyzed to find outliers that may be errors. We use the default setting reported in [45], i.e., with $\theta = 0.8$ and $\epsilon = 0.05$. Predictions are ranked by their levels of deviation for precision/recall.
- **Linear** [11]. Linear is a linear-complexity framework proposed in the data mining community for detecting "exception sets" that are anomalous from input data. The framework can be instantiated with different functions to measure "dissimilarity" for a set of values, and we use a hand-tuned dissimilarity function suggested in the original paper defined over regular expression patterns. This method scans over all values, and iteratively broadens the regex pattern to accommodate values it encounters, while computing dissimilarity scores for each step. Predicted errors are ranked by the dissimilarity score for precision/recall.
- **Linear-Pattern (LinearP).** Observing that Linear performs poorly because its generalization is too coarse-grained, we additionally test a modified method called LinearP that first transforms values

[4] Data is released at https://github.com/zphuangHKUCS/Auto-Detect-released-data.

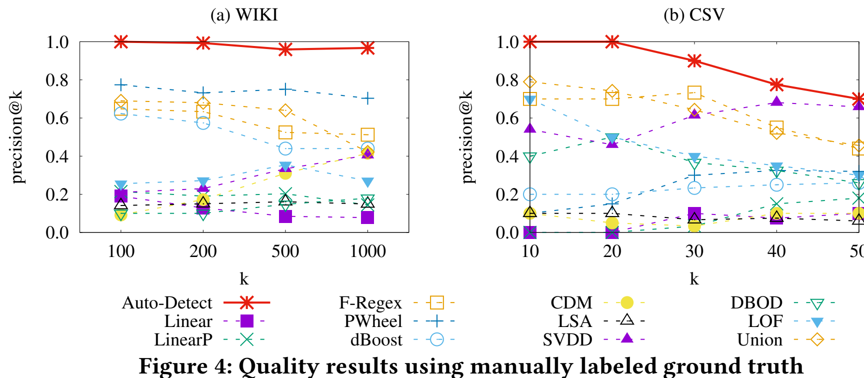


Figure 4: Quality results using manually labeled ground truth

k	v_1	v_2
1	1935/1982/2011	2000
2	2009	27-11-2009
3	1999	2013.
4	1963	1983.
5	2008	2009.
6	1865.	1874
7	1976	198.
8	1,87	5875 CR
9	ITF $50.000	WTA International
10	August 16, 1983	1985

Table 4: Top-10 predictions of incompatible values in WIKI

using our generalization tree (using \D, \L, etc.) before running Linear. This indeed substantially improves the quality of Linear as we will see in our experiments.

• **Compression-based dissimilarity measure (CDM)** [38]. This approach proposes an interesting and parameter-free way to measure distance between two strings by leveraging off-the-shelf compression algorithms. Observing that data size after compression often correlates well with data redundancy, the distance between two strings x and y is defined as $CDM(x,y) = \frac{c(xy)}{c(x)+c(y)}$, where $c(x)$ is the size of x after compression, and $c(xy)$ is the size of x concatenated with y after compression. We use standard generalization to convert values into patterns, and invoke the standard zip compression to compute the *CDM* distance, which is used to produce a ranked list of outliers as proposed in [38].

• **Local search algorithm (LSA)** [32]. Utilizing the observation that entropy often describes the regularity and consistency of a set of values, the LSA approach uses an optimization based formulation for outlier detection, by defining outliers as the subset of values whose removal leads to the largest reduction in entropy. We use the LSA procedure on patterns generated from input values to find the most likely errors, ranked by their reduction in entropy.

• **Support vector data description (SVDD)** [54]. SVDD is an outlier detection method inspired by SVM. Under a suitable definition of distance between values, SVDD assigns a cost for describing values that fall in a "ball" of certain center and radius, as well as a cost for describing values that are outside of the "ball", which are effectively outliers. The ball with minimum description cost is used and values outside of the ball are predicted outliers. We use an alignment-like definition of patterns distance [22], and rank errors by their distance to the ball center.

• **Distance-based outlier detection (DBOD)** [40]. DBOD is an efficient method to find outliers in large databases. A value v is an outlier in DBOD if the distance to its closest neighbor v' is over some threshold D, i.e., $\min_{v' \neq v} dist(v, v') > D$. We use the same pattern distance as in SVDD, and rank the outliers by their corresponding distance.

• **Local outlier factor (LOF)** [16]. This approach defines an *LOF* score for the degree to which a data point v may be classified as an outlier based on the density of data points in v's neighborhood. We rank predictions based on their *LOF* scores.

• **Union.** This is an approach that unions predictions from all ten baseline methods. Predictions are taken from each method at given precision levels (based on ground truth labels) to produce an aggregate prediction.

• **Auto-Detect.** This is the method proposed in this work. By default, we use both WEB and Pub-XLS for training, and smoothing factor $f = 0.1$. As discussed we rank prediction $(v_i, v_j) \in C$ by the maximum estimated precision $\max_{v_i, v_j \in C, L_k \in \mathbf{L}'} P_k(s_k(v_i, v_j))$.

4.3 Manual Evaluation of Prediction Quality

To be sure about the precision of the errors automatically detected by each system, we first ask a human judge to manually inspect top-K results of each method, and provide a true/false/not-sure label for each predicted error. Our quality metric is $Precision@K = \frac{\text{true-errors@K}}{K}$, which is simply the precision at position K.

For the WIKI data set, we run each method on 100K columns sampled from WIKI to produce predicted errors ranked by confidence. For each method, from the top-1000 predictions, we manually label 100 predictions sampled from each of the range: [0-100], [100-200], [200-500] and [500-1000], respectively, and calculate the corresponding result for *Precision@K*.

Figure 4(a) shows the results on WIKI. We can see that Auto-Detect has a very high precision of over 0.98 across the top-1000 results. Most predicted errors indeed make intuitive sense, as was shown in Figure 2. Given that there are a total of 30M columns in WIKI, and our sub-sample of 100K produces at least 980 errors, we can extrapolate using two-tailed t-distribution to infer that if we were to label all predictions from Auto-Detect, we will find $294K \pm 24K$ true errors in WIKI (99% confidence interval). The fact that we can detect around *three-hundred thousand errors* in WIKI is both surprising and encouraging – we initially expect WIKI to be squeaky clean, but the sheer number of errors detected shows the ubiquity of data quality issues and the effectiveness of Auto-Detect.

PWheel performs better than F-Regex and dBoost, because PWheel can generate specific patterns based on input values, which is more flexible that can better handle diverse input data compared to F-Regex and dBoost, both of which require rigid and predefined patterns. Union achieves a good performance overall, but is not as good as PWheel.

Table 4 shows the top-10 predictions by Auto-Detect on WIKI. Most of these value pairs are indeed incompatible, for example an extra dot at the end of numbers are predicted to be highly unlikely (in comparison, a dot in the middle of numbers are often floating-point numbers that frequently co-occur with integer numbers, thus not predicted as errors at all).

For the CSV data set there is a total of 441 test columns, we manually label results produced by each method and report a complete quality evaluation, again using *Precision@K*.

Figure 4(b) shows the results. We can see that Auto-Detect again outperforms alternative methods. F-Regex performs better compared to other methods on this test set, because a large fraction of test columns in CSV can be captured by F-Regex's predefined regex patterns (e.g., date formats).

4.4 Auto-Evaluation of Prediction Quality

Because labeling data manually across all methods is expensive and cannot scale to a large number of analysis we would like to perform (relatively recall, precision at lower K, etc.). In order to facilitate large scale evaluations, we design an automatic evaluation. Specifically, we programmatically generate test cases by sampling a "dirty" value v_d from column C_1 and mixing it with a "clean" column C_2 to produce a test column $C_2 \cup \{v_d\}$, where the goal is to predict v_d as a likely error. We manually design and tune a compatibility score to make sure that v_d is indeed inconsistent with C_2 (more details in Appendix F). We manually inspect a large number of test cases so generated to ensure that they are indeed correct. While such an automatic-evaluation is never perfect, we find the results to correlate well with those using manually labeled ground truth.

For both WIKI and Ent-XLS, we generate 5K dirty test cases this way, and mixing them with clean 5K, 25K, and 50K clean columns (with a dirty/clean ratios of 1:1, 1:5 and 1:10, respectively). We again use *Precision@K* to measure quality. To avoid cluttered plots with a large number of lines to compare with, henceforth we only report results of seven best-performing methods, i.e., Auto-Detect, F-Regex, PWheel, dBoost, SVDD, DBOD and LOF, and omit other baselines.

Figure 5 shows the results on WIKI. As expected, as k increases and as the dirty/clean ratio decreases, the precision of all methods drop. Auto-Detect achieves high precision consistently when varying the dirty/clean ratio. When the ratio is 1:1, the precision of Auto-Detect is close to 100% even with $k \leq 1000$, suggesting that 20% of true errors are discovered. When $k = 5000$, Auto-Detect still maintains a relatively high precision of 0.82, which in this setting translates to a recall of 0.82. We find such results encouraging. Among other methods compared, F-Regex, PWheel and dBoost perform better, but the performance gaps with Auto-Detect become increasingly as the dirty/clean ratio decreases, suggesting that these methods are not as performant for more difficult test cases (with relatively few errors).

Figure 6 shows a similar comparison but using test cases generated from Ent-XLS. Like in the previous experiment using WIKI, Auto-Detect also maintains a very high precision at $k \leq 1000$. For high recall ($k = 5000$), its precision drops faster compared to that on WIKI. We find part of the difference can be attributed to table-extraction errors in parsing .xlsx files, which often involves tables with complex structures such as pivot tables, and thus creates noisy test-cases when evaluated automatically. We observe that dBoost performs better than other methods on this test set, primarily because Ent-XLS contains many numeric columns, and dBoost is better at leveraging numeric statistics (e.g., variance and frequency) to detect potential errors.

4.5 Sensitivity to Training Corpora

To test the effect of using different training corpora, we compare the performance of using WIKI and WEB for training, respectively, and using test cases from Ent-XLS for evaluation (with dirty:clean ratio of 1:10). Results in Figure 8(c) show that using WEB achieves better quality than using WIKI. We believe this is because the WEB corpus is an order of magnitude larger than WIKI (350M columns vs. 30M), thus likely covering more diverse content and producing more robust statistics. We find it interesting that the benefit of using the "bigger" WEB data outweighs the smaller but slightly more clean WIKI data.

4.6 Sensitivity to Memory Budget

We test Auto-Detect with different memory budgets of M = 1MB, 4MB, 16MB, 64MB, 256MB, 1GB and 4GB, respectively. It turns out that when M is between 1MB and 256MB, the same set of languages are selected, so we only report three data points for 1MB, 1GB and 4GB, where two, five and seven languages are selected, respectively.

Figure 7 shows the corresponding quality on Ent-XLS. We can see that as expected, Auto-Detect performs better with more memory since more languages can be selected. Interestingly, with even minimum memory (1 MB), the precision is still over 0.93 with a relative recall of 0.1 ($k = 500$), which is surprisingly good. The quality gap of using different memory becomes more significant for larger K, indicating that in order to achieve both high precision and good recall, a larger amount of memory may be necessary.

4.7 Impact of Count-Min Sketch

We test the impact of using CM sketch on error-detection quality. Specifically, we test the quality of Auto-Detect while requiring CM sketch to compress the co-occurrence data to 1%, 10% and 100% of its original size (the last one corresponds to no sketch).

Figure 8(a) shows the results on Ent-XLS (with a dirty/clean ratio of 1:10). As expected, precision suffers with more substantial compression, but the quality gap is surprisingly small, which is very encouraging and shows the effectiveness of the CM sketch. This means we can effectively maintain the same level of precision with only 1% of the memory budget (35 MB). We further note that compared to Figure 7(c), using 35 MB memory and CM sketches Auto-Detect already achieves better quality than using 1 GB of memory but without sketches.

4.8 Sensitivity to Aggregation Function

We compare Auto-Detect the following aggregation methods, using the same languages selected by Auto-Detect:

- **AvgNPMI** uses the ensemble of average NPMI values from different languages $\mathcal{L}_i$, or $\text{avg}_i\ s_i(v_1, v_2)$, as the confidence score to rank predictions for (v_1, v_2).
- **MinNPMI** instead uses the minimal NPMI values from different languages, or $\min_i\ s_i(v_1, v_2)$.
- **Majority Voting (MV)** simply counts the 0/1 votes from each language $\mathcal{L}_i$. The vote for $\mathcal{L}_i$ is 1 if its NPMI value s_i is over the threshold determined for a certain precision target P (e.g., $P = 0.95$), and 0 otherwise.
- **Weighted Majority Voting (WMV)** is the same as Majority Voting except that it assigns a weight to the vote from $\mathcal{L}_i$ based on the magnitude of the NPMI score.

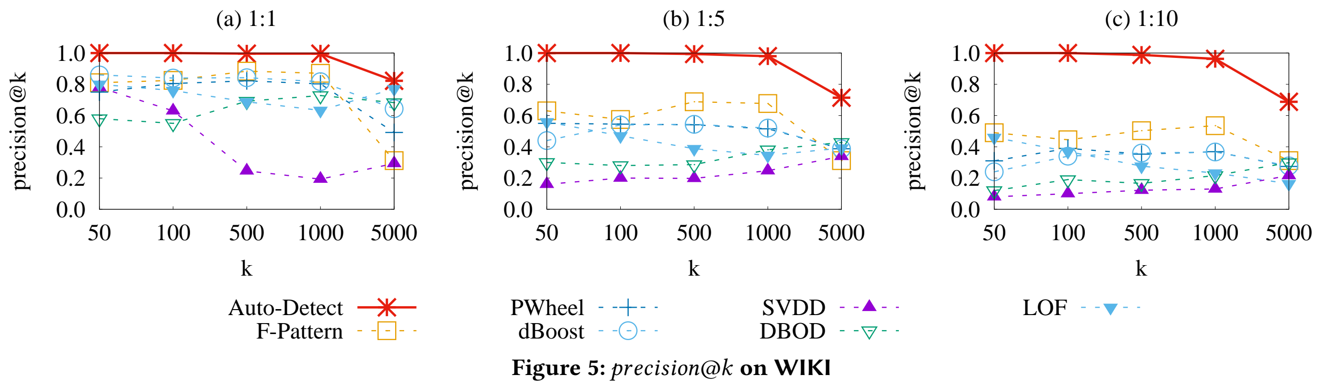


Figure 5: *precision@k* **on WIKI**

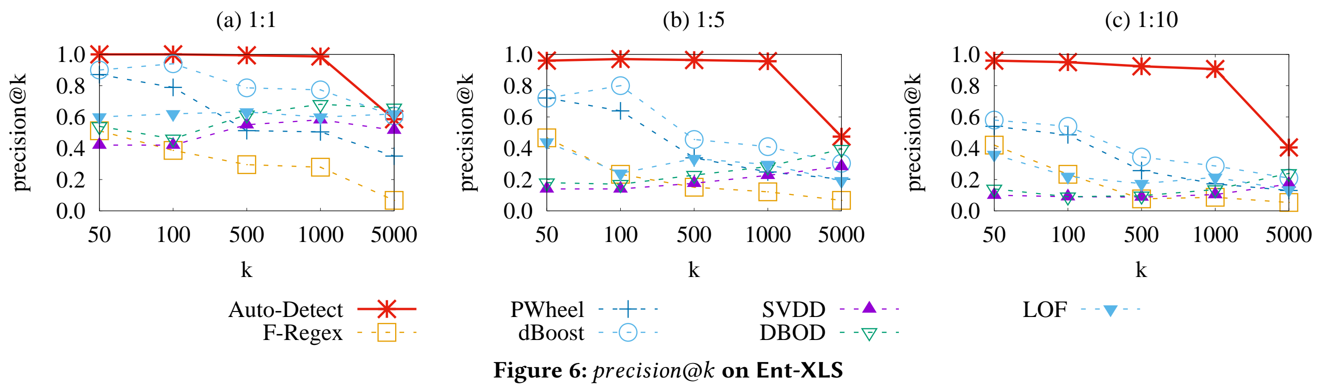


Figure 6: *precision@k* **on Ent-XLS**

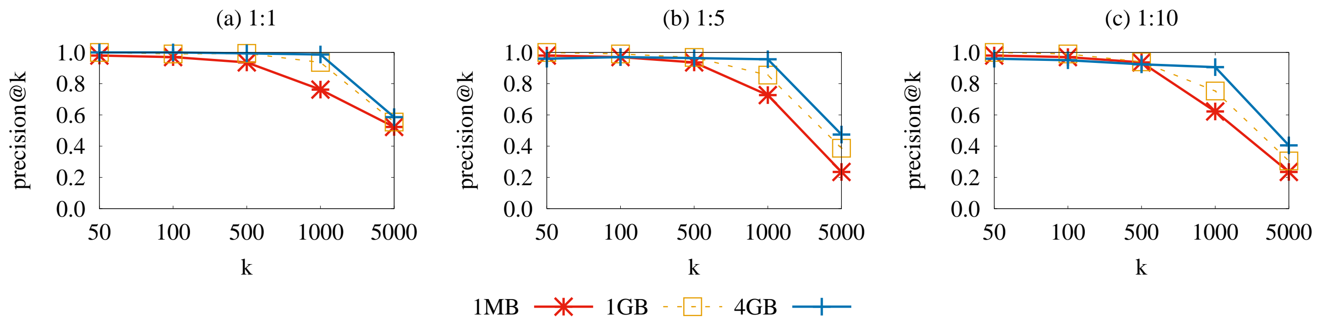
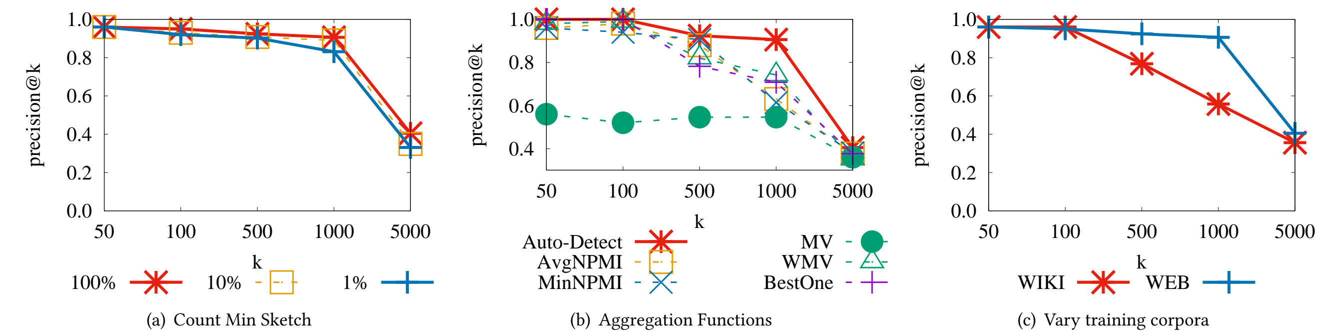


Figure 7: *precision@k* **v.s. memory on Ent-XLS**



(a) Count Min Sketch (b) Aggregation Functions (c) Vary training corpora

Figure 8: *precision@k* **with different configurations on Ent-XLS**

• **BestOne.** In addition to aggregation, We also compare with the best performing single language (which requires over 5Gb memory and actually violates the memory constraint).

Figure 8(b) shows the results on Ent-XLS. We can see that Auto-Detect significantly outperforms all standard aggregation methods, suggesting that our optimization-based formulation with union-aggregation has substantial quality advantages. MV is the least effective aggregation, because it counts each vote equally and not taking into account that languages by design have narrow "focuses" on different types of quality issues. The WMV approach mitigates that issue to some extent. AvgNPMI and MinNPMI also have inferior quality results, because NPMI scores cannot be compared directly between different languages, but should be calibrated with techniques such as distant-supervision as discussed in Section 3.2.

5 RELATED WORKS

Error detection has been extensively studied in the literature. Recent surveys of the area can be found in [10, 19].

Multi-column error detection using rules. Leveraging relationship between multiple columns to detect potential data quality problems is a well-studied topic. Discovering and enforcing various sub-classes of first order logic such as functional dependencies [37, 44, 57], conditional functional dependencies [15], numeric functional dependencies [27], denial constraints [23], as well as holistic inference [51] have been studied. Multi-column error-detection is an important class orthogonal to the focus of this work.

Single-column error detection. Existing systems such as Trifacta [9], Power BI [4] and Talend [7] all have predefined regex-like patterns to recognize common data types (e.g., IP addresses, emails, etc.). When most values in an input column conform to known patterns while a small fraction of values do not, the non-conforming ones can be flagged as errors (Appendix A gives more details).

In addition to using coarse-grained regex patterns, Auto-Type [56] provides an alternative that leverages fine-grained, type-specific logic to ensure more precise type detection and error prediction.

Excel [1] also uses a small set of simple rules (e.g., whether formulas in neighboring cells are consistent) to automatically flag potentially erroneously cells (Appendix A).

Potter's Wheel [50] proposes a novel method to use patterns to summarize values in a column based on minimum description length (MDL). Data values not captured by patterns are likely outliers. As discussed earlier, this method uses only local information and makes no consideration of data compatibility at a global scale. Similar techniques developed in other contexts (e.g., [14, 43]) also suffer from the same limitation.

The authors in [45] propose a dBoost method to detect errors using predefined expansion rules for known types. For example, observing that date time can be stored as integers, all integer columns can then be interpreted as dates and expanded into fields such as year, month, day, day-of-week, etc., over which distributions can be analyzed to find outliers that may be errors.

For categorical values such as popular entities in known domains, approaches like [24, 55] are developed to leverage knowledge bases and mapping tables to discover values not belonging to well-known concepts (e.g., "WA" and "Washington" may be mixed in the same column but actually belong to different knowledge base concepts or mapping relationships). While such techniques are powerful, they would mainly cover popular entities in known concepts, but not the large variety errors as illustrated in Figure 1 and Figure 2. These techniques are thus orthogonal to the focus of this work.

Detect formula errors. Another common type of errors in spreadsheets is formula errors, analogous to "code smells" studied in programming language community. Techniques such as [21, 52] are proposed that exploit contiguity of formulas in neighboring cells to detect formula inconsistency as potential errors.

Numeric error detection. Finding outliers/errors in numerical data has also been studied. For example, Hellerstein [34] proposes to use the metric of MAD (median absolute deviation), which is from robust statistics and analogous to standard deviation to detect outliers in numeric data. Multi-column correlation between numerical attributes have also been considered in [34] to find outliers.

Outlier detection. There is a large body of work on outlier detection [11, 19, 30, 33, 34, 36, 38], which differ in assumptions such as data distribution, data types, and application scenarios.

Our view is that a key difference between outlier detection and error detection studied in this work, is that outlier detection heavily relies on the local data distribution to find data points that deviate from the mass. Outerlies, as such, are only a statement about deviations in the context of a local distribution and not necessarily an indicator of data quality. For example, an CEO's salary may be many standard deviations away from all employees' in a company, which however is not a data error. Errors that are automatically detected, on the other hand, should be a global property of the data in question and not a function of the local data distribution. For instance, the example of *Col-3* discussed in the introduction with two different date formats should be recognized as errors regardless of the exact ratios of the mixtures (99%-1%; or 50%-50%).

For more in-depth discussions on this related topic of outlier detection, we refer readers to surveys like [19, 36].

Application-driven error correction. Recent approaches such as BoostClean [41] and ActiveClean [42] have also explored the interesting area of addressing data errors that have largest impact on specific applications like machine learning.

Corpus-driven applications in data integration. Web tables are known to be a rich source of structured data that have enabled a variety of data-driven approaches in data integration. Example scenarios include schema exploration [17], table segmentation [22], auto-join [31], as well as error-detection studied in this work.

6 CONCLUSION AND FUTURE WORK

We propose Auto-Detect that leverages corpus statistics to perform single-column error detection. Experiments suggest that Auto-Detect can find high-quality errors from supposedly clean Wikipedia tables, and Excel files. Interesting directions of future work include detecting errors in semantic data values as well as other types of multi-column errors.

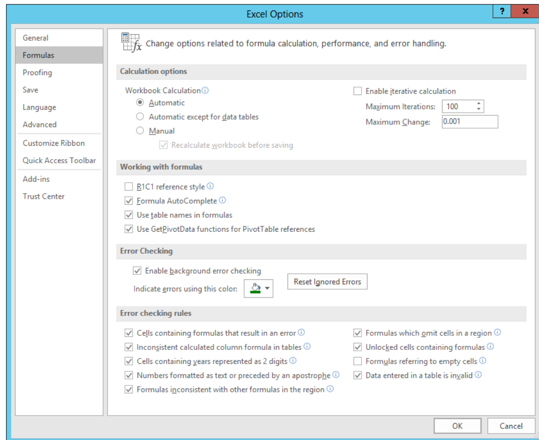


Figure 9: Excel menu with 9 default error detection rules.

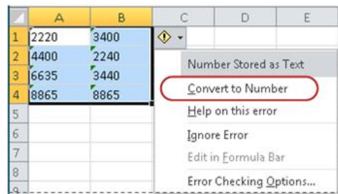


Figure 10: Excel automatically detects one type of error: "number stored as text".

A COMMERCIAL OFFERINGS

A.1 Rule-based Error Detection

Microsoft Excel [1] provides a limited list of 9 simple predefined error checking rules, as shown in Figure 9. The most commonly-known errors among these are "Formulas inconsistent with other formulas in the region" (when formula in a cell is different from its neighboring cells); and "Number stored as text" (when a number is stored as text while others in the column are stored as numbers). The list also contains rules such as "Cells containing years represented as 2 digits".

As can be seen, the number of scenarios covered by these 9 simple rules is quite limited. Manually extending the list to additional rules is expensive and poses manageability issues.

Excel also supports customized data validation rules, which users can define and enforce on individual data columns (e.g., certain columns in a spreadsheet can only have numeric data; Excel should raise errors if non-numeric data are entered into such columns). While such manually-defined rules are customizable which is nice, it does impose a burden on users to declare validation rules, which is analogous to declaring database schemas, but is even more onerous.

A.2 Type-specific Error Detection

Trifacta has a list of around 10 built-in "data types" [9] such as IP addresses and emails that they can recognize using regex-like patterns. The list of supported data types can be seen in Figure 11. Values in a column not conforming to type-specific regex patterns are flagged as errors at the top of each data column, as can be seen in Figure 12. Similar functionality is also available in systems like Power BI [4] and Talend [7].

Trifacta additionally produces a rich set of visualizations of distributions such as numeric value distribution, frequency distribution, and string length distribution, etc., as shown in Figure 13. Such features help users to explore data and discover potential quality issues (e.g., values whose string lengths are significantly larger compared to other values in the same column).

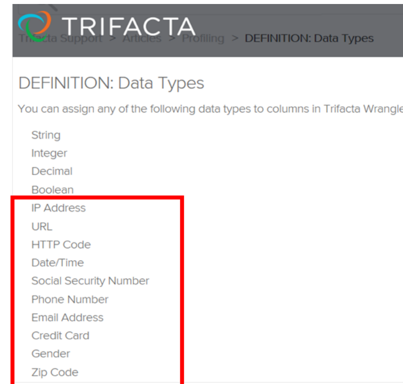


Figure 11: Trifacta's predefined list of types (with corresponding regex detection).

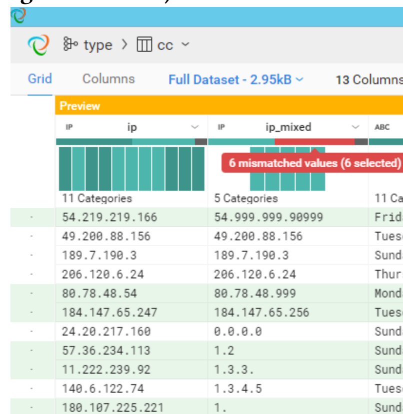


Figure 12: Trifacta detects erroneous values with types.

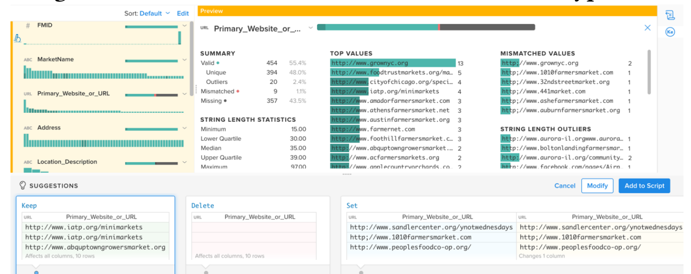


Figure 13: Trifacta visualizes rich column value distribution for users to identify outliers and error.

A.3 OpenRefine/GoogleRefine

OpenRefine does not proactively detect errors but provides a text clustering feature that allows users to group similar values in the same columns together, so that users can see what values are highly similar and decide whether these are near-duplicates due to typos or misspellings. For example, Figure 14 shows results from column clustering, where similar values like "U.S." and "US", "USA" and "U.S.A." are grouped together, so that users can decide whether to collapse these clusters of values into a canonical representation.

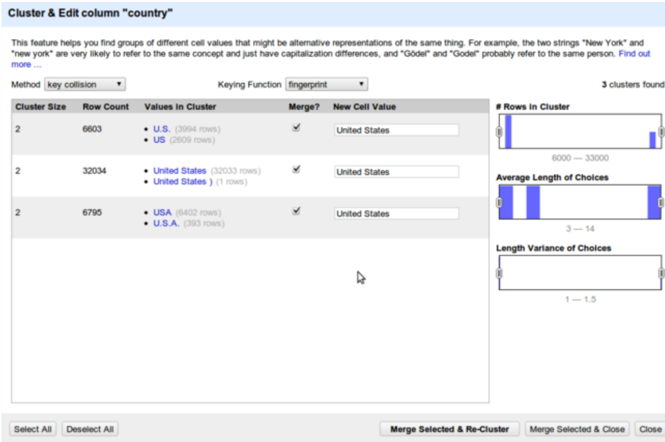


Figure 14: OpenRefine cluster values for duplicates.

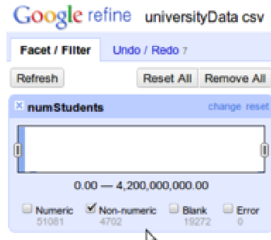


Figure 15: OpenRefine produces visual histograms for value distributions in a column, for easy identification of errors.

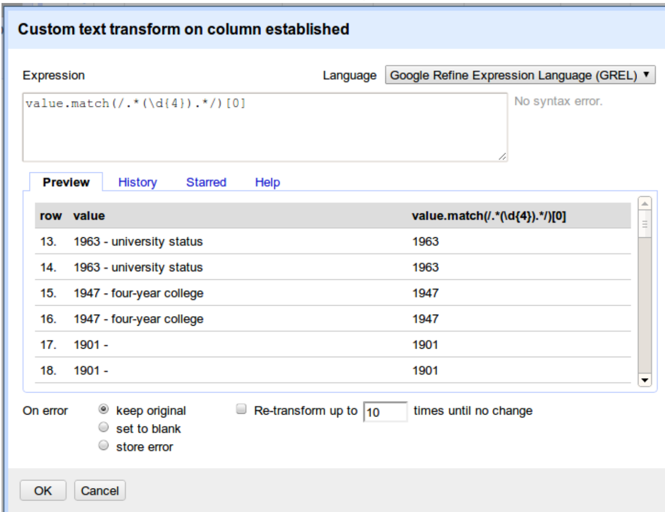


Figure 16: Inconsistent values can be corrected using user-defined transformations in OpenRefine.

Similar to Trifacta, OpenRefine produces visual histograms of data distributions within single columns, to allow users to find data quality issues more easily. In the example shown in the left part of Figure 15, users can explore distribution of numeric values in a column to see if there are outliers. He or she can also filter down to black cells or non-numeric cells to determine if such cells are erroneous.

Once cells with quality problems are identified, users can program transformation rules to correct such problems, as shown in Figure 16.

B COMPATIBILITY SCORE CALCULATION

After selecting a set of languages $\mathbf{L}'$, we can not only predict whether two values are compatible or not, but also compute a score to measure the level of compatibility, considering the NPMI scores $s_k(v_1, v_2)$ given by each language $L_k \in \mathbf{L}'$ jointly.

Recall that during language selection, we have estimated the precision of a language L_k with NPMI threshold θ_k as $P_k(\theta_k)$ by Equation 7. Naturally, we can consider the confidence of predicting (v_1, v_2) as incompatible by language L_k as $P_k(s_k(v_1, v_2))$.

Instead of a straightforward idea, which is to use the average confidence $P_k(s_k(v_1, v_2))$ among all languages $L_k \in \mathbf{L}$ as the final prediction of compatibility, we propose using the max confidence value, i.e.,

$$Q(r_1, r_2) = \max_k P_k(s_k(v_1, v_2)) \tag{11}$$

Using max confidence has two main advantages:

• Trust the most confident language. As discuss in Section 2.2, different languages have different focuses, some on digits and some on symbols. The language gives a strongly negative score only when the two values are highly incompatible and that happens to be that language's focus. Thus using max confidence can put our trust to the one with most confidence.

• No assumption of independence between languages. Compared with average confidence, max confidence does not assume independence between languages. Consider an extreme case where we have selected many languages focusing on digits and only one on symbols. If two values are incompatible w.r.t. symbols, using average confidence cannot detect this.

In our experiments (refer to Figure 8(b)), we show that max confidence outperforms other alternatives, including average confidence and two NPMI-based aggregations.

C PROOF OF THEOREM 1

Proof. We prove the hardness of this problem using a reduction from densest k-subhypergraph (DkH) problem [12]. Recall that in DkH, we are given a hypergraph $H(V, E)$ and the task is to pick a set of k nodes such that the sub-hypergraph induced by the nodes has the maximum weight on hyper-edges. DkH is known to be NP-hard and inapproximable [12].

We show a reduction from DkH. For any given problem in DkH, we construct a problem in DT-aggregation as follows. For each vertex $v \in V$ we construct an incorrect prediction $p^-(v)$. For each hyper-edge $e_i \in E$, we construct a language L_i whose ranked list of predictions has at the top: $p^-(e_i) = \{p^-(u) | u \in V(e_i)\}$, where $V(e_i)$ is the set of vertices incident on e_i. This corresponds to a set of $|V(e_i)|$ incorrect predictions. We then construct a correct prediction $p^+(e_i)$ and place it below $p^-(e_i)$. The "cost" of selecting L_i is now $|p^-(e_i)|$ incorrect predictions, and the "gain" is $|p^+(e_i)|$ correct predictions. For a given DkH problem, we now have a DT-aggregation problem of selecting languages $\mathbf{L}' \subset \mathbf{L}$, with a budget of at most k mis-predictions, or $| \cup_{L_i \in \mathbf{L}'} p^-(e_i)| \leq k$, and a goal of maximizing correct predictions. By giving a unit-size to each language, we can use memory-budget M to control the total number of languages selected. Specifically, we generate a set of decision-version of DT-aggregation with precision threshold $P = \frac{M}{M+k}$ (to force no more than k mis-predictions will be selected). If we were able to solve DT-aggregation, then we would have in turn solved

DkH. Since DkH is NP-hard and inapproximibility as shown in [12], we obtain a similar hardness result for DT-aggregation. □

D PROOF OF THEOREM 2

Proof. We show the hardness of the problem in Definition 5 using a reduction from the budgeted maximum coverage (BMC) problem [35]. Recall that in BMC, we have a universe of elements U, a set of sets $\{S_i | S_i \subseteq U\}$, where each S_i has a cost $c(S_i)$, and a gain function $g(S_i) = \sum_{e \in S_i} g(e)$, where $g(e)$ is the gain of an element $e \in U$. For any instance of the BMC problem, we can generate a ST-aggregation problem as follows. We generate a language L_i for each S_i, and an example $t^- \in \mathbf{T}^-$ for each $e \in U$. We set the incompatibility cases $H_i(\mathbf{T})^-$ covered by L_i to be $\cup t_j^-$ for all $e_j \in S_i$, and naturally set the size of each language $size(L_i) = c(S_i)$. We then have an ST-aggregation problem direct corresponding to BMC. Suppose we can efficiently solve ST-aggregation, this makes BMC solvable which contradicts with its hardness. ST-aggregation is therefore NP-hard. □

E PROOF OF LEMMA 3

Proof. We show that for every problem of ST-aggregation, we can construct a budgeted maximum coverage (BMC) problem. Specifically, we create an element $e \in U$ in BMC for each example $t^- \in \mathbf{T}^-$, and set $g(e) = 1$. Furthermore, we create a set S_i for each language L_i, that consists of elements e_j for each $t_j^- \in H_i^-(\mathbf{T})$. We can now invoke BMC to solve ST-aggregation.

The authors in [39] develop a greedy algorithm for BMC that inspires our algorithm 1. We use a similar argument as their Lemma 3 in [39] to show the $\frac{1}{2}(1 - \frac{1}{e})$ approximation ratio. □

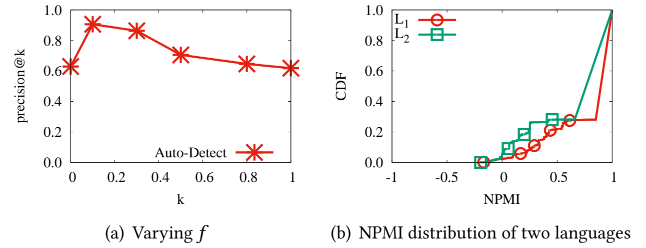
F AUTO-GEN TEST CASES

First, we need to find columns $\mathbf{C}^+ \subset \mathbf{C}$ that are semantically compatible, from which we can take pairs of values in $\mathbf{C}^+$ to generate $\mathbf{T}^+$. In order to find compatible columns, we use a set of crude generalization rules, denoted as $G()$, that generalize characters by classes, into digits, upper-case letters, lower-case letters, respective, while leaving all symbols and punctuation untouched. We can apply $G()$ to all columns in $\mathbf{C}$ to obtain global co-occurrence statistics, from which we can compute a crude measure of compatibility for any pair of values (u, v), based on the NPMI score of their generalized patterns $(G(u), G(v))$.

We then use statistical compatibility of NPMI$(G(u), G(v))$ to find $\mathbf{C}^+$, by ensuring that any pairs of values u, v in the same column has NPMI score over a manually-tuned threshold (0). This threshold is chosen to ensures almost all such $\mathbf{C}^+$ selected are indeed compatible. With $\mathbf{C}^+$, we can then generate positive examples as $\mathbf{T}^+ = \{(v_1, v_2, +) | v_1, v_2 \in C, C \in \mathbf{C}^+\}$.

To generate $\mathbf{C}^-$, we sample a column $C_1 \in \mathbf{C}^+$, take a value $u \in C_1$, and mix u with another column $C_2 \in \mathbf{C}^+$ to produce a synthetic column $C_2' = C_2 \cup \{u\}$. This will, with high likelihood, make C_2' a "dirty" column in which u is the sole value incompatible with others. Occasionally, u may actually be compatible with C_2 by co-incidence (e.g., they share the same pattern). Using the same generalization G, we prune out all cases $C_2' = C_2 \cup \{u\}$, if there exists $v \in C_2$ such that NPMI$(G(v), G(u)) \geq -0.3$. Recall that a high NPMI indicates that the two patterns may statistically be positively-correlated and thus has some chance to be compatible. We manually verify that this heuristic prunes out almost all columns C_2' that are indeed compatible (and thus should be not used as negative examples). We can then generate $\mathbf{T}^-$ as $\{(u, v, -) | u \in C_1, v \in C_2\}$.

G OTHER EXPERIMENTS



(a) Varying f (b) NPMI distribution of two languages

Figure 17: Additional Experiments

G.1 Sensitivity to Smoothing

We test the sensitivity of quality to the smoothing factor f. We vary f from 0 to 1, and plot *Precision@K* with $K = 1000$ on Ent-XLS (results on other data sets are similar). Figure 17(a) shows that using smoothing can indeed improve result quality ($f = 0$ corresponds to no smoothing). We find that quality is the best and relatively stable in the range of [0.1, 0.3].

G.2 NPMI Distribution Analysis

Figure 17(b) shows the cumulative distribution function of NPMI values given by two generalization languages as we are performing NPMI calibration in Section 3.2. We can see that i) around 60% of value pairs have NPMI scores equaling 1.0. This is because many values within the same column have the identical formats, thus leading to the same pattern representation under the same generalization language. ii) L_1 tends to gives larger NPMI scores than L_2, and the NPMI distribution of L_2 is steeper at the range of [-0.2, 0.3]. iii) It is not meaningful to directly perform aggregation based on NPMI scores, given the fact that their distributions are quite different from each other.

G.3 Efficiency Analysis

We also test the efficiency of these methods. All experiments are run on a Windows Server 2012 R2 machine with Intel(R) Xeon CPU E5-2670 2.6GHz and 115GB memory. Since Auto-Detect is used as an interactive feature, we do require it to respond in sub-seconds.

Table 5 shows the execution time comparison, averaged over test cases from the Ent-XLS dataset. We can see that while Linear is the most expensive method, all other approaches are relatively efficient. For high throughput over a set of columns in a user table, parallel execution of error detection on individual columns can be easily achieved.

We note that another mode of invoking Auto-Detect is to execute error-detection tasks in the background, and pop-up notifications when we find quality issues. Such scenarios have a lower requirement on latency and can be executed whenever the client is more idle.

Method	F-Regex	PWheel	dBoost	Linear	Auto-Detect
time(s)	0.11	0.21	0.16	1.67	0.29

Table 5: Average running time per column (in seconds).

REFERENCES

[1] Microsoft excel error checking rules. https://excelribbon.tips.net/T006221_Changing_Error_Checking_Rules.html.

[2] Openrefine (formerly googlerefine). http://openrefine.org/.

[3] Paxata data preparation. https://www.paxata.com/.

[4] Power bi. https://docs.microsoft.com/en-us/power-bi/desktop-data-types.

[5] Self-service data preparation, worldwide, 2016. https://www.gartner.com/doc/3204817/forecast-snapshot-selfservice-data-preparation.

[6] Spreadsheet mistakes - news stories, compiled by european spreadsheet risk interest group EuSpRiG. http://www.eusprig.org/horror-stories.htm.

[7] Talend data services platform studio user guide: Semantic discovery. https://help.talend.com/reader/nAXiZW0j0H~2~YApZIsRFw/_u0D0oqWxesgBDSihDgbYA.

[8] Trifacta. https://www.trifacta.com/.

[9] Trifacta built-in data types. https://docs.trifacta.com/display/PE/Supported+Data+Types.

[10] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016.

[11] A. Arning, R. Agrawal, and P. Raghavan. A linear method for deviation detection in large databases. In *KDD*, pages 164–169, 1996.

[12] A. Arulselvan. A note on the set union knapsack problem. *Discrete Applied Mathematics*, 169:214–218, 2014.

[13] A. Barron, J. Rissanen, and B. Yu. The minimum description length principle in coding and modeling. *IEEE Transactions on Information Theory*, 44(6):2743–2760, 1998.

[14] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtds from xml data. In *Proceedings of the 32nd international conference on Very large data bases*, pages 115–126. VLDB Endowment, 2006.

[15] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 746–755. IEEE, 2007.

[16] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[17] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[18] K. Chakrabarti, S. Chaudhuri, Z. Chen, K. Ganjam, Y. He, and W. Redmond. Data services leveraging bing's data assets. *IEEE Data Eng. Bull.*, 39(3):15–28, 2016.

[19] V. Chandola, A. Banerjee, and V. Kumar. Outlier detection: A survey. *ACM Computing Surveys*, 2007.

[20] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[21] S.-C. Cheung, W. Chen, Y. Liu, and C. Xu. Custodes: automatic spreadsheet cell clustering and smell detection using strong and weak features. In *Proceedings of the 38th International Conference on Software Engineering*, pages 464–475. ACM, 2016.

[22] X. Chu, Y. He, K. Chakrabarti, and K. Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1713–1728. ACM, 2015.

[23] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[24] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. 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*, pages 1247–1261. ACM, 2015.

[25] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Computational linguistics*, 16(1):22–29, 1990.

[26] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[27] G. Fan, W. Fan, and F. Geerts. Detecting errors in numeric attributes. In *International Conference on Web-Age Information Management*, pages 125–137. Springer, 2014.

[28] D. Freeman. How to make spreadsheets error-proof. *Journal of Accountancy*, 181(5):75, 1996.

[29] A. Go, R. Bhayani, and L. Huang. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1(2009):12, 2009.

[30] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014.

[31] Y. He, K. Ganjam, and X. Chu. Sema-join: joining semantically-related tables using big table corpora. *Proceedings of the VLDB Endowment*, 8(12):1358–1369, 2015.

[32] Z. He, S. Deng, and X. Xu. An optimization model for outlier detection in categorical data. In *International Conference on Intelligent Computing*, pages 400–409. Springer, 2005.

[33] Z. He, S. Deng, and X. Xu. An optimization model for outlier detection in categorical data. *Advances in Intelligent Computing*, pages 400–409, 2005.

[34] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[35] D. S. Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In *Approximation algorithms for NP-hard problems*, pages 94–143. PWS Publishing Co., 1996.

[36] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126, 2004.

[37] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658. ACM, 2004.

[38] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 206–215. ACM, 2004.

[39] S. Khuller, A. Moss, and J. S. Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

[40] E. M. Knox and R. T. Ng. Algorithms for mining distance based outliers in large datasets. In *Proceedings of the International Conference on Very Large Data Bases*, pages 392–403. Citeseer, 1998.

[41] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu. Boostclean: Automated error detection and repair for machine learning. Technical report, 2017.

[42] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment*, 9(12):948–959, 2016.

[43] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30. Association for Computational Linguistics, 2008.

[44] S. Lopes, J.-M. Petit, and L. Lakhal. Functional and approximate dependency mining: database and fca points of view. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):93–114, 2002.

[45] Z. Mariet, R. Harding, S. Madden, et al. Outlier detection in heterogeneous datasets using automatic tuple expansion. *MIT Technical Report*, 2016.

[46] M. Mintz, S. Bills, R. Snow, and D. Jurafsky. Distant supervision for relation extraction without labeled data. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 1003–1011. Association for Computational Linguistics, 2009.

[47] R. R. Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 10(2):15–21, 1998.

[48] R. R. Panko. Spreadsheet errors: What we know. what we think we can do. 2008.

[49] S. G. Powell, K. R. Baker, and B. Lawson. Errors in operational spreadsheets: A review of the state of the art. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–8. IEEE, 2009.

[50] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[51] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10(11):1190–1201, 2017.

[52] R. Singh, B. Livshits, and B. Zorn. Melford: Using neural networks to find spreadsheet errors. *MSR technical report*.

[53] M. Surdeanu, D. McClosky, J. Tibshirani, J. Bauer, A. X. Chang, V. I. Spitkovsky, and C. D. Manning. A simple distant supervision approach for the tac-kbp slot filling task. In *TAC*, 2010.

[54] D. M. Tax and R. P. Duin. Support vector data description. *Machine learning*, 54(1):45–66, 2004.

[55] Y. Wang and Y. He. Synthesizing mapping relationships using table corpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1117–1132. ACM, 2017.

[56] C. Yan and Y. He. Synthesizing type-detection logic using open-source code. In *SIGMOD*, 2018.

[57] Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in rdf graphs. In *International Semantic Web Conference*, pages 794–809. Springer, 2011.

[58] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *ACM SIGIR Forum*, volume 51, pages 268–276. ACM, 2017.