

DATAVINCI: Learning Syntactic and Semantic String Repairs

Mukul Singh
Microsoft
Delhi, India
singhmukul@microsoft.com

José Cambronero
Microsoft
New Haven, USA
jcambronero@microsoft.com

Sumit Gulwani
Microsoft
Redmond, USA
sumitg@microsoft.com

Vu Le
Microsoft
Redmond, USA
levu@microsoft.com

Carina Negreanu
Microsoft Research
Cambridge, UK
cnegreanu@microsoft.com

Gust Verbruggen
Microsoft
Keerbergen, Belgium
gverbruggen@microsoft.com

ABSTRACT

String data is common in real-world datasets: 67.6% of values in a sample of 1.8 million real Excel spreadsheets from the web were represented as text. Systems that successfully clean such string data can have a significant impact on real users. While prior work has explored errors in string data, proposed approaches have often been limited to error detection or require that the user provide annotations, examples, or constraints to fix the errors. Furthermore, these systems have focused independently on syntactic errors or semantic errors in strings, but ignore that strings often contain both syntactic and semantic substrings. We introduce DATAVINCI, a fully unsupervised string data error detection and repair system. DATAVINCI learns regular-expression-based patterns that cover a majority of values in a column and reports values that do not satisfy such patterns as data errors. DATAVINCI can automatically derive edits to the data error based on the majority patterns and constraints learned over other columns without the need for further user interaction. To handle strings with both syntactic and semantic substrings, DATAVINCI uses an LLM to abstract (and reconcretize) portions of strings that are semantic prior to learning majority patterns and deriving edits. Because not all data can result in majority patterns, DATAVINCI leverages execution information from an existing program (which reads the target data) to identify and correct data repairs that would not otherwise be identified. DATAVINCI outperforms 7 baselines on both error detection and repair when evaluated on 4 existing and new benchmarks.

PVLDB Reference Format:

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. DATAVINCI: Learning Syntactic and Semantic String Repairs. PVLDB, 14(1): XXX-XXX, 2020.

1 INTRODUCTION

Errors in tabular data, such as inconsistent or corrupted values, are common and can result in incorrect computations, invalid conclusions, and downstream data pipeline failures [5]. Such data errors can stem from a variety of sources including manual data entry, data integration, and faulty computations. Prior work [26] reported that even in professional settings, such as financial institutions and

consulting firms, up to 24% of spreadsheets can have mistakes, including data errors. Figure 1 shows some examples of errors found in online Wikipedia tables and public Excel spreadsheets.

In a sample of 1.8 million Excel spreadsheets from the web, we found that 67.6% of values are represented as text (compared to numeric or datetime values), providing a substantial opportunity for string repair systems to help real users. While prior work has introduced approaches relevant to string data, these typically have focused primarily on detecting errors but not repairing them [6, 7, 16, 26], required users to provide (partial) annotations or constraints to drive a semi-supervised detection/repair procedure [11, 12, 18, 19], or have relied on a limited rule-learning approach [2].

Furthermore, effective string cleaning must support errors in columns where values contain both syntactic and semantic substrings. For example, given a column with three values [(NY, Boston), (Miami)] where the pattern is both semantic (city names) and syntactic (parenthesized values), a repair system should report the first entry as a data error and suggest (New York) as the repaired value. Unfortunately, prior rule-based and external-knowledge-based systems can only tackle the syntactic issue or the semantic issue, respectively, but not the combination of these.

We introduce DATAVINCI, a fully automated error detection and resolution system for string columns in tabular data. DATAVINCI is designed to handle qualitative string errors [12] in a column, such as missing string values, inconsistent formats, or misspellings in strings. Critically, in contrast to prior work [7, 12, 26], DATAVINCI does not only detect data errors in strings but also suggests repairs. DATAVINCI is the first system, to our knowledge, to detect and repair errors in strings that consist of both syntactic and semantic substrings. Prior work has been tailored to either category separately. Finally, DATAVINCI can perform detection and repair in a fully unsupervised manner, without requiring user inputs such as providing constraints [6], examples [12], or annotations [18].

Error Detection. To carry out fully unsupervised error detection on a string column, DATAVINCI exploits the regularity in string data, and reports as data errors those values that do not satisfy patterns associated with a (configurable) large fraction of the column’s values. In contrast to existing pattern-based work [2, 7, 16], DATAVINCI uses an LLM to identify and mask semantic substrings, allowing the regular-expression-based pattern learner to capture strings with both syntactic and semantic substrings.

Error Repair. While prior string repair systems require that the user provide examples [11], annotations [18], or constraints [19],

Wikipedia			Excel		
Mixing	Length	Area	Phone	Code	Website
47 (7.55%)	04:34	Birmingham	937-587-3389	US-837	https://www.google.com
38 (6.09%)	05:23	London	419-996-7110	UK-828	https://www.youtube.com
26 (4.24%)	04:38	Wales	440-993-8351	US-219	www.facebook.com
21 (3.42%)	03.45	Hampton	(937) 509 6413	us291	https://www.twitter.com
19 (3.05%)	03:34	Rockford	937-587-3389	POL-921	https://www.instagram.com
16 (2.66%)	04:55	Birminxham	419-908-7099	AUS-463	https://www.linkedin.com
19 (3.05%)	03:22	London	4405764039	USA-392	https://www.reddit.com
16 (2.66%)	04:12	Wales	419-6310-239	UK-389	https://www.amazon.com

Figure 1: Examples of string data errors found in Wikipedia tables and Excel spreadsheets. The data errors in each column are highlighted. There is a mix of both syntactic (like 03.45) and semantic data errors (like Birminxham) found in real tables.

DATAVINCI can suggest data repairs without any additional input by comparing the data error to the regular-expression-based patterns (i.e. significant patterns) that are associated with a large fraction of values. Specifically, DATAVINCI generates candidate repairs by deriving a minimal set of edits to an erroneous value that lead to satisfying a significant pattern. Because these edits may contain elements such as character classes that need to be concretized, DATAVINCI learns relationships between non-error values and significant regular expressions and uses these as constraints when predicting the concrete values. After these edits are applied any semantic mask values remaining are concretized by replacing them with concrete LLM-predicted substrings. DATAVINCI sorts the final set of candidate repairs based on a heuristic ranker.

Execution-Guided Repair. By flagging values that do not satisfy significant patterns as data errors, DATAVINCI can mitigate false positives. However, real data may not always display such significant patterns. For example, consider a column named col1 with values $[c-1, c-2, c3, c4]$. A pattern learner would identify two patterns, one to cover the first two entries and a second to cover the last two entries. Without further information, it is not possible to identify any of these entries as a data error under a majority assumption. However, executing the user-written spreadsheet formula $=SEARCH("-", [@col1])$, which searches for the “-” in a string, on the first two values will yield a valid result, but on the last two values will result in an exception. This execution provides a strong signal that the last two values are data errors with respect to this formula. DATAVINCI can exploit this execution information to provide suggested repairs. First, DATAVINCI flags the last two value as data errors given their exceptional execution. Second, DATAVINCI learns a regular expression exclusively over the two successful input values. Third, DATAVINCI applies the pattern-based repair procedure previously described to the failing inputs, producing the expected repairs $c-3$ and $c-4$ by inserting the missing “-” characters.

We evaluate DATAVINCI on an existing benchmark (Wikipedia web tables [7]) and three new benchmarks (a collection of real-world Excel tables, synthetically corrupted Excel tables, and Excel formulas and their input data). We compare our performance to 7 baselines, which include 5 existing error detection/repair systems, one LLM-based baseline, and one small transformer-based model. In addition, we augment detection-only systems with a GPT-3.5 error

repair module. Jointly these cover a variety of existing approaches from the data management community or reasonable alternatives.

We find that DATAVINCI’s fully unsupervised pattern-based detection and repair approach leads to higher precision detection (+1.6 to +8.7 points than next best), higher recall detection (+3.6 than next best), and higher precision repairs (+1.6 to +12.2 points than next best). We find that removing the LLM-based abstraction and concretization approach that enables repairing mixed syntactic and semantic strings reduces repair precision and recall (-3.8 and -6 points, respectively). Finally, we show that applying DATAVINCI’s execution-guided repair to our benchmark of Excel formulas raises formula-level execution success rates to 54% (single-column inputs) and 47.8% (multi-column inputs) compared to 43.2% and 35.7% for DATAVINCI without execution-guided repair, and substantially outperforming the next best non-DATAVINCI baseline.

To summarize, our contributions are:

- We develop a pattern-based approach to detect *and* repair errors in strings with both syntactic and semantic substrings.
- We implement our approach in DATAVINCI, which can detect data errors and suggest repairs in a fully unsupervised fashion. We also introduce execution-guided repair for data cleaning, where DATAVINCI executes an existing program that depends on the target column, uses the resulting execution success information to learn patterns, and reduces the associated program failures by applying suggested repairs.
- We carry out an extensive evaluation of DATAVINCI on multiple datasets from different domains and against multiple existing systems, which shows that DATAVINCI can outperform existing string error detection and repair approaches.
- We release the URLs and data preparation scripts to produce our three novel Excel-based benchmarks.¹

2 PROBLEM STATEMENT

We take inspiration from the data error detection and repair formulation presented in prior work [7, 19]. First, we introduce our target domain of string data errors.

Definition 2.1 (String Data Error). Let T be a table, consisting of a collection of m columns $\{c_1, \dots, c_m\}$, each with n values. Let c_i be

¹Due to compliance we are not able to release the post-processed data, so we release scripts replicate our benchmarks.

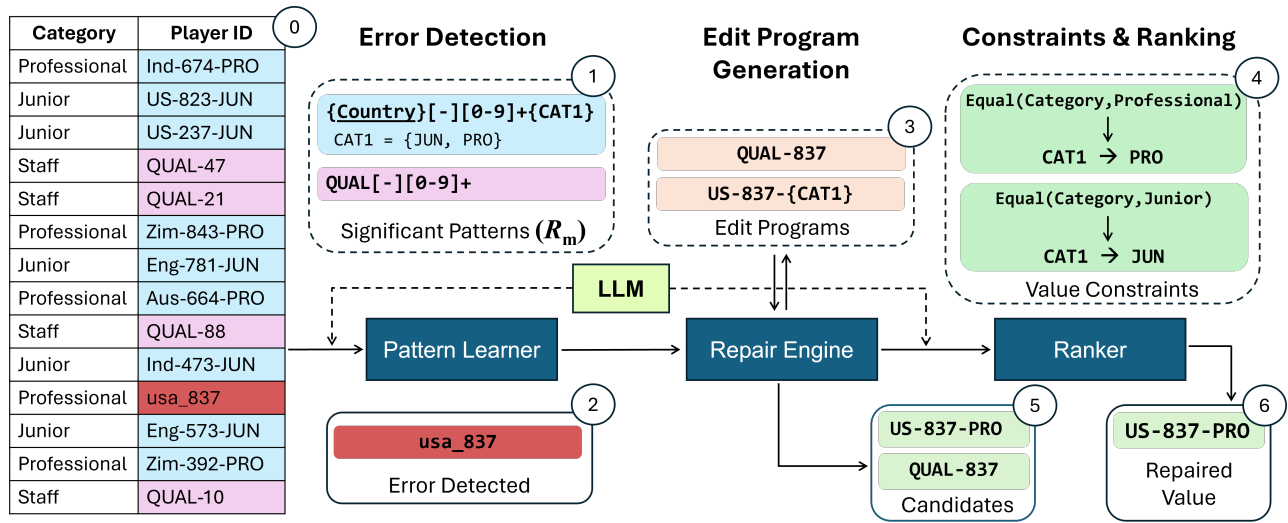


Figure 2: DATAVINCI illustrated with an example from our corpus. ① denotes the input table. ① DATAVINCI learns significant patterns, which account for both syntactic and semantic substrings (using the LLM), for the *Player ID* column. The values satisfying each pattern are shown in the pattern’s color. Underlined pattern elements (e.g. *Country*) correspond to LLM-abstracted semantic substrings. ② significant patterns are used to detect outliers. *usa_837* is detected as an error as it does not satisfy a significant pattern. ③ edit programs for the outlier are learned by deriving edits to the string that will satisfy a significant pattern. ④ constraints are generated to concretize abstraction actions in edit programs. Here, the constraint between {CAT1} and *Category* column is learned. Applying these value constraints produces the candidate repairs ⑤. The heuristic ranker sorts candidates and ⑥ the top ranked candidate is suggested to the user.

a string column consisting of observed values $\{v_1, \dots, v_n\}$, where v_j is a string value. For each v_j , let v_j^* be the latent clean value. If $v_j \neq v_j^*$, we say v_j is a string data error.

The goal of a string repair system is not only to identify such string errors, but also provide repaired values.

Definition 2.2 (String Error Repair). Let v be a string value with a data error. Let \hat{v} be a string value produced by a data repair system, given v and the table T . We say \hat{v} is a candidate string error repair. If $\hat{v} = v^*$, we say it is a successful string error repair.

Because the space of possible errors and repairs described previously can be infinite², we focus our problem on *pattern-based regular error detection and repair* in string values. In Section 3.2 we show that rather than limit our scope, simple adaptations to this framing allows us to perform pattern-based semantic repairs.

Definition 2.3 (Regular string column and regular errors). Let \mathcal{L}_i be the (pre-defined) latent regular language that characterizes latent values in column c_i , such that $\forall v_j \in c_i : v_j^* \in \mathcal{L}_i$. We say a value v_j is a regular error, if $v_j \notin \mathcal{L}_i$.

Definition 2.4 (Regular repair). Let v be a regular string error. We say \hat{v} is a candidate regular string repair, if $\hat{v} \in \mathcal{L}_i$. If $\hat{v} = v^*$, we say it is a successful regular string repair.

The goal of our approach is to automatically learn a representation for \mathcal{L}_i , and use this representation along with our table T to produce repair candidates for every $v \notin \mathcal{L}_i$. Note that a direct

²consider you can always extend the string with new characters

consequence of Definition 2.3 is that a string data error that is in the latent regular language, but is not the right value, will not be detectable in this setting. This is reflected in DATAVINCI’s design and we will show that despite this restriction, our approach achieves high performance on real benchmark tasks. We describe DATAVINCI’s components in greater detail in the following section.

3 DATAVINCI

Figure 2 shows a schematic overview of DATAVINCI. We start with a table that contains a string column to clean (*Player ID*) ①. DATAVINCI performs error detection by learning a set of significant patterns ① using an off-the-shelf pattern learner. Because patterns may need to represent both syntactic and semantic substrings, we use an LLM to replace semantic substrings with a mask. Values that do not satisfy any significant pattern are flagged as data errors ② and are provided to the repair engine. The repair engine produces edit programs by deriving the edits necessary for a data error to satisfy a significant pattern. These edit programs can have abstract edit actions, such as deciding what character in a class or substring in a disjunction (e.g., CAT1) to choose ③. We resolve these choices by learning value constraints over non-error data ④ and applying these to the abstract edit actions to produce full repair candidates ⑤. Because different significant patterns may produce different repair suggestions, we employ a heuristic ranker to return the top suggestion ⑥. The following sections describe these steps in detail.

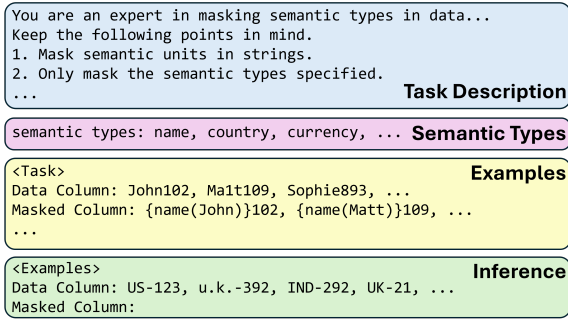


Figure 3: DATAVINCI’s semantic abstraction prompt structure. The different colors show the components of the prompt.

3.1 Detecting Patterns and Errors

DATAVINCI uses a set of patterns to describe the regular language that a column represents. Values that do not match any patterns, and thus not accepted by the language, are marked as errors. Each pattern is described by a regular expression over all characters encountered in our dataset. As is standard in regular expressions, we use the following character classes for simplicity of notation: digits, cased and uncased letters, alphanumeric, spaces, alphanumeric with spaces, and the common recurring character class of $[0, 1]$.

Given a column c , DATAVINCI uses FlashProfile [15] to learn up to k patterns $R = \{r_1, \dots, r_k\}$ such that all values v in c are in the language jointly defined by these patterns $\mathcal{L}_R = \bigcup \mathcal{L}_{r_k}$. FlashProfile supports disjunction (e.g., $(cat|dog)$ matches “cat” and “dog”) and quantification over groups (e.g., $([a-z])_+$ matches one or more repetitions of a letter and a period). FlashProfile balances the number of individual patterns with the generality (number of cells covered) of each pattern. We use the default parameters.

From these patterns, DATAVINCI then selects the subset of patterns $R_m \subseteq R$ that individually cover at least a fraction δ of the values. We refer to these as *significant* patterns. The union of these significant patterns defines the language $\mathcal{L}_{R_m} = \bigcup_{k \in m} \mathcal{L}_{r_k}$. DATAVINCI reports any value $v \notin \mathcal{L}_{R_m}$ as a data error. We can change the confidence required for DATAVINCI to report a value as an error by changing the threshold δ .

3.2 Semantic Abstractions

To allow DATAVINCI’s repairs to perform both syntactic and semantic changes, like `usa_837` \rightarrow `US-837` in Figure 2 (where the substrings `usa` and `US` denote a country), using a syntactic repair engine, we perform a *semantic abstraction* on each value. In such an abstraction, substrings that denote some named concept x , like a city or a color, are replaced with a mask token m_x .

EXAMPLE 1. Consider a column with values [red 1, dark green 2, blue phone 3]. A pattern that matches this column is $([a-z])_+[0-9]$ and we cannot identify that “phone” should be removed. With knowledge about colors, the semantic abstraction of this column is $[m_c 1, m_c 2, m_c 3]$. The significant pattern becomes $m_c [0-9]$ and “ m_c phone 3” is identified as an error.

One way of obtaining the semantic abstraction of a string is to maintain a dictionary of concepts. This has three main drawbacks: some concepts are hard to exhaustively enumerate (like colors), spelling mistakes cannot be repaired without additional fuzzy matching (like “bleu” instead of “blue”) and semantic concepts can be contextual (“red” can also refer to a movie).

We propose to leverage a large language model (LLM) to obtain the semantic abstraction. While abstracting, we also allow the LLM to provide suggestions for replacement strings that can be used to replace the mask in the final string. This allows the LLM to repair spelling mistakes, for example, in Figure 2, it correctly repairs the masked value `usa` to `US`.

To capture the context of semantic concepts, we prompt the model with a whole column at once. Long columns are processed in batches based on the maximum prompt length (4k tokens for GPT-3.5). Due to the repetitive nature of this task, we found that long prompts did not deteriorate the quality of generations.

It is important to mask values at the right level of granularity. For example, values from a column [Q4-2002, Q3-2002, Q32001] are masked entirely as *Quarter* if given to the LLM without further restrictions. This masking would prevent the last value from being repaired (Q32001 \rightarrow Q3-2001).

To mask values with the right granularity, we only mask a set of predefined semantic categories. Sherlock [8], a prior work on semantic type detection, introduced a method to classify a column as one of 78 popular semantic types, such as *Name*, *Country* and *Currency*. We take the 20 most frequently occurring semantic types, which cover 99.2% of values with a detected semantic type, from a sample 25K data columns from our Excel data.

Figure 3 summarizes the prompt to perform semantic abstraction with the LLM. We use few-shot prompting to show the model both (1) to mask the substring s of semantic type t as $\{t(s)\}$ and (2) that it is allowed to repair the masked values (i.e. $\{t(s')\}$ where $s' \neq s$). For example, the masked version of “US-123” is “{country(US)}-123” and “u.k.-392” becomes “{country(UK)}-392”. These are then transformed to “ m_1 -123” and “ m_1 -392” before learning patterns, and m_1 is added to the alphabet for our regular expression learner.

3.3 Repairing Values with Edit Programs

Given a pattern $r_k \in R_m$ and a value $v \notin \mathcal{L}_{r_k}$, DATAVINCI repairs v by learning edit programs e such that $e(v) \in r_k$, where we use $e(v)$ to denote applying program e to value v . Let an edit action be a function that optionally deletes a character and optionally emits a given character. An edit program is then a sequence program over edit actions, which when applied to value v , yields one candidate repair. The edit program is applied to a string by starting from the first character and applying each edit action on the current character and advancing to the next character in the string. An overview of edit actions is shown in Table 1.

EXAMPLE 2. Consider an edit program $[M, S(2), I(.)]$ consisting of three steps. Highlighting the current character being looked at with an underscore, the string `AAA3` is edited as follows.

$$\underline{A}AA3 \xrightarrow{M} \underline{A}AA3 \xrightarrow{S(2)} A\underline{A}AA3 \xrightarrow{I(.)} A2.\underline{A}A3$$

Let e be an edit program that repairs value $v \notin \mathcal{L}_r$ with respect to pattern r . e is minimal if there does not exist an edit program

Table 1: Edit actions over characters.

Action	Shorthand	Delete	Emits	Cost
match()	M			0
insert(c)	I(c)		c	1
delete()	D	✓		1
substitute(c)	S(c)	✓	c	1

$e'(v) \in \mathcal{L}_r$ such that $\text{DIST}(e'(v), v) < \text{DIST}(e(v), v)$, where DIST is the Levenshtein edit distance [20] between strings.

Given $v \notin \mathcal{L}_{r_k}$, DATAVINCI learns minimal edit scripts $e(v) \in \mathcal{L}_{r_k}$ using dynamic programming. The pattern r_k is interpreted as a non-deterministic finite state automaton (NFA) where edges correspond to matching (and consuming) a single character [22]. An example of a pattern and its corresponding NFA is shown in Figure 4. An erroneous value will end in a non-accepting state in the NFA where no further transitions can be taken. By changing the characters of the string as we traverse it, edit actions allow us to follow new edges. These changes come at a cost, however, and these costs are shown in Table 1. Finding a minimal edit script then corresponds to finding the lowest cost path in the NFA, which is done through dynamic programming.

EXAMPLE 3. Consider the example in Figure 4. When processing the highlighted data error, after transitioning on the A edge, there are no more edges that we can follow. For a cost of 1, we can use an edit action I(3) to follow the [0-9] edge.

There are two challenges to finding the lowest cost path: loops due to unbounded quantification and transitioning on character classes and categories. For example, there are ten edit actions I(0) \dots I(9) that allow following an edge [0-9].

To handle loops, we approximate the NFA for a given value v with a directed acyclic graph D_v by unrolling loops up to depth $\lceil \frac{\text{LEN}(v)}{\text{LEN}(\text{cycle})} \rceil$ with the length of a cycle defined as the number of edges in it. We support nested cycles and follow the same unrolling procedure recursively for each nested loop. In practice, we found nested loops to be rare in our learned regular expressions.³ Figure 4 shows the NFA converted into a DAG by unrolling the loop twice and topologically sorting states. The loop was unrolled twice as the length of the cycle is 3 and $\lceil \frac{4}{3} \rceil = 2$.

To match character classes and disjunctions, we first learn *abstract* edit programs, which have edit actions that emit a character class or a value from a disjunction. Two examples of abstract edit actions are S(0-9) (see Figure 4) and I(CAT|PRO) (see Figure 2). After the minimal abstract edit program is obtained, abstract edit actions are concretized by choosing one of the characters in its character class or one of the strings in the category. Concretization is detailed in Section 3.4—it does not influence how minimal (abstract) edit programs are learned.

Let $\text{COST}(i, j)$ be the cost of transitioning on edge j after having consumed i characters in the string and $\text{MOVE}(i, j)$ be the edit action required to do so. In the previous example, we had that $\text{COST}(1, j_2) = 1$, as a result of the necessary insertion action. Since nodes can have

³less than 1% of regular expressions learned over 100,000 Excel columns resulted in nested loops

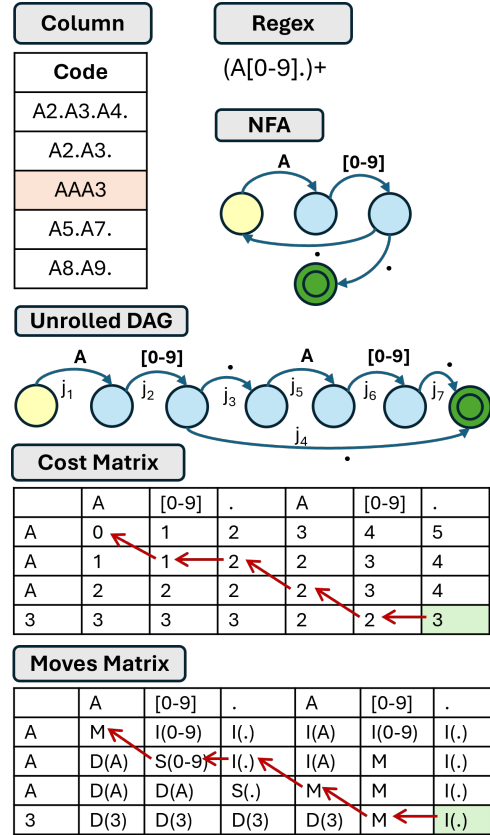


Figure 4: DATAVINCI’s repair engine on an example column. The significant regex learned for the column is $(A[0-9].)^+$ which is used to identify the outlier (AAA3), highlighted in red. The regex is converted to an NFA with the starting and accepting states shown in yellow and green, respectively, and transition symbols over the edges. The NFA is then unrolled to a DAG for the outlier. We show the computed cost and moves matrix and highlight one optimal repair script path in them with red arrows.

multiple incoming edges, we write $p(j)$ to denote incoming edges in the node where edge j starts. The cost of transitioning on each edge while traversing the string is recursively defined as

$$\text{COST}(i, j) = \min \begin{cases} \min_{j' \in p(j)} \text{COST}(i, j') + 1 & (i) \\ \min_{j' \in p(j)} \text{COST}(i-1, j') + [s[i] \neq l(j)] & (M \text{ or } S) \\ \text{COST}(i-1, j) + 1 & (D) \end{cases} \quad (1)$$

with $l(j)$ the label of edge j and $[a \neq b]$ Iverson bracket notation, which evaluates to 1 if $a \neq b$ else 0. The associated moves are shown on the right of each case.

EXAMPLE 4. In Figure 4, we can arrive at $\text{COST}(2, j_2)$ (i.e. traversing j_2 after consuming two characters) through 3 paths. We can move from $\text{COST}(1, j_1)$ to the new state by substituting the second A with a digit (S(0-9) with cost 1), inserting a digit and deleting the second A (I(0-9), D with a cost 2), or delete the second A and insert a digit

($D, I(0-9)$ with a cost 2). Because substitution had the lowest cost (1), $COST(2, j_2) = 1$ and $MOVES(2, j_2) = S(0-9)$.

The correctness of our DP algorithm can be proven through extension of string edit-distance [23]. The time complexity of the algorithm is $O(m^2n)$ with m the number of edges in the DAG and n the number of characters in the erroneous string v . The space complexity is $O(mn)$ as we only need to store the cost and moves matrices.

3.4 Concretizing Edit Programs

DATAVINCI learns decision trees to predict concrete values for each character class (which are just disjunctions over a set of characters) and string disjunction in the pattern r_f that induced our edit program. We refer to these learned rules as concretization constraints. By construction, every abstract edit action corresponds to a character class (or string disjunction) in r_f . We thus learn a decision tree that uses features from rows where value $v \in \mathcal{L}_{r_f}$ to predict the value that allowed transitioning on an edge in the unrolled DAG that has the target character class (or string disjunction).

EXAMPLE 5. Consider value “A2.A3.” in row 2 from Figure 4 and the associated unrolled DAG, which has two [0-9] edges, which match 2 and 3, respectively. Similarity, value “A5.A7.” on row 4 matches 5 and 7. For the first edge, this yields two training examples

$$\text{row 2} \rightarrow 2 \quad \text{row 4} \rightarrow 5$$

for the decision tree.

To learn decision trees over these training examples, DATAVINCI first extracts boolean features from each row. We take inspiration from the CORNET [21] system for conditional formatting in tables and generate predicates over a set of templates to use as features. Table 2 shows all supported predicate templates. To generate candidate string constants s , DATAVINCI considers the set of column values and tokens after splitting (separately) on non-alphanumeric characters, case changes, and switches between contiguous alphabetic and numeric characters. For $\text{length}(v, n)$ we consider the top 5 most frequent cell lengths in the column. In tables with multiple columns, DATAVINCI generates these predicated-based features over every column.

EXAMPLE 6. Consider the first row in Figure 2. For the Player ID column and $\text{TextContain}(c, s)$, we generate four constants for s . The first is the value itself (Ind-674-PRO). Splitting the cell obtains tokens {Ind, 674, PRO, -}. As $\text{TextContain}(\text{Player ID}, -)$ is true for all cells in the column, this is not considered and dropped. We get four features from the first row: $\text{TextContains}(\text{Player ID}, \text{Ind})$, $\text{TextContains}(\text{Player ID}, 674)$, $\text{TextContains}(\text{Player ID}, \text{Pro})$, $\text{TextContains}(\text{Player ID}, \text{Ind-674-Pro})$.

To learn each decision tree, DATAVINCI samples trees with varying number of split nodes and depth, filters down to those with an accuracy of at least α (default 0.8), ranks trees in ascending order of (nodes, depth), and takes the first such tree. This tree can now be applied to a repair that has abstract edits to predict the concretized candidate repair.

Table 2: Supported predicate templates and their arguments. The v argument denotes the column value. For example, $\text{equals}(\text{col1}, \text{“AR”})$ matches the cells in column col1 which are equal to “AR”.

$\text{equals}(v, s)$	$\text{contains}(v, s)$	$\text{startsWith}(v, s)$
$\text{endsWith}(v, s)$	$\text{length}(v, n)$	$\text{hasDigits}(v)$
$\text{isNum}(v)$	$\text{isError}(v)$	$\text{isFormula}(v)$
$\text{isLogical}(v)$	$\text{isNA}(v)$	$\text{isText}(v)$

3.5 Ranking Repair Candidates

Our repair procedure can produce multiple edit programs (since there may be multiple significant patterns). To address this challenge, DATAVINCI uses a heuristic candidate ranker. This heuristic corresponds to a weighted linear combination of edit script properties. The weights are manually set based on qualitative analysis on a small held-out set of 100 columns sampled from our corpus of Excel spreadsheets. The four properties are (1) string edit distance between erroneous value and the repaired value, (2) count of alphanumeric edit operations, (3) string edit distance of repaired value to closest value in column, and (4) fraction of column matching the significant pattern used to generate the repair.

3.6 Execution-Guided Repair

DATAVINCI’s pattern-based detection relies on the assumption that the significant patterns characterize the data distribution well, and that values that do not satisfy such patterns are data errors. However, not all string data will be able to produce a significant patterns nor will significant patterns learned over all values necessarily highlight errors. To address this challenge, DATAVINCI can exploit execution information from programs that operate on columns to further refine its error detection and repair suggestions. This improvement comes from learning a significant pattern set R_m that accounts for different execution outcomes. We now describe this in detail.

Let P be a program that reads a subset of columns in a table, including our target cleaning column c . We say P is a column-transformation program if it can execute over each row tuple independently and produces one or more output values for each row—thus generating one or more output columns.

EXAMPLE 7. Consider a table with two columns $c1 = [x, y, z]$ and $c2 = [a, b, c]$. A program $\text{concat}(c1, c2)$, which produces $[xa, yb, zc]$, is a column-transformation program, while $\text{first}(c1)$, which produces x is not.

DATAVINCI executes the column-transformation program on T and groups executions into successes and failures (as signaled by exceptional values, such as nan, or program exceptions). The non-exception group is then provided to our regular expression learner and all patterns learned are treated as significant patterns R_m . All values v in our target column that were inputs to a failing execution are identified as data errors, and we apply the repair procedure previously described.

Table 3: Benchmark properties and metrics reported. # Rows/# Cols denote the average number of rows/columns in the table.

Dataset	Metrics	# Tables	# Col	# Row
Wikipedia Tables	Precision, Fire Rate	1000	5.1	27.3
Excel	Precision, Fire Rate	200	1.6	523.4
Synthetic Errors	Precision, Recall, F1	1000	4.3	447.5
Excel Formulas	Execution Success	11000	1.4	216.5

4 EVALUATION SETUP

We first describe the hardware specifications used for carrying out experiments, the benchmarks we evaluate over and the baselines to which we compare.⁴

4.1 Hardware Specifications

All experiments were carried out using Python (version 3.8.7) on a machine with an Intel Core i7 processor (base at 1.8 GHz), K80 GPU, 64-bit operating system, and 32 GB RAM.

4.2 Benchmarks

We evaluate on four benchmarks. We use a benchmark of web tables from prior work [7] and we collect and release three new Excel-based benchmarks. We briefly describe the statistics of these benchmarks in Table 3:

- **Wikipedia Tables:** We build on the Wikipedia tables dataset released in the original Auto-Detect [7] paper. Following their approach, we take the sample of 1000 tables on which they manually annotated system predictions and extend this with manual annotations for DATAVINCI and our baselines. Like prior work, given this annotation approach, we only report precision [7].
- **Excel:** We sampled 200 tables present in workbooks drawn from a corpus of 1.8 million publicly available Excel workbooks from the web. Similar to the Wikipedia benchmark, we run all available systems on the sampled tables, manually annotate their suggestions, and report precision.
- **Synthetic Errors:** We sample 1000 Excel tables (disjoint from **Excel** benchmark) from the same corpus previously described. We then synthetically introduce errors with the goal of measuring recall. To introduce errors, we apply the following noise operations: (1) random character insertion, deletion and change, (2) random delimiter insertion deletion and change, (3) random digit swap, (4) random shuffle of characters, (5) random capitalization, (6) random decimal, comma swap in numerics, (7) visually-inspired typos $\{o \rightarrow 0, l \rightarrow 1, e \rightarrow 3, a \rightarrow 4, t \rightarrow 7, s \rightarrow 5\}$. We randomly corrupt cells with 20% probability. For each of the cells to be corrupted, there is a 25% probability of applying 1, 2, 3 or 4 noise operations, sampled without replacement from the set of operations described. Because it is likely there are already real data errors present in the data, systems may detect errors or suggest repairs for cells beyond our

⁴Data will be released for the camera ready to undergo required compliance checks.

Table 4: System comparison overview. Category denotes the task for which the systems were designed. In the case, of T5 and GPT-3.5 category reflects our usage in this work.

System	Category
WMRR	Detection + Repair
HoloClean	Detection + Repair
Raha	Semi-supervised Detection
Auto-Detect	Detection
Potters-Wheel	Interactive Detection+Repair
T5	Detection + Repair
GPT-3.5	Detection + Repair
DATAVINCI	Detection + Repair

synthetically corrupted cells. As a result, we focus our analysis on recall of our synthetic errors, but report precision (which will be naturally deflated) and F1 for completeness.

- **Excel Formulas:** We create a dataset of the form (formula, input columns), where formula is an Excel formula used to define a column (i.e. all rows have the same formula, modulo input values), and input columns correspond to the input values necessary to execute the formula. We restrict ourselves to formulas where input values and output value are part of the same table. The task is to repair any data errors in the input columns such that the formula evaluates without producing any error values. To construct this dataset we sampled 15,000 tables from Excel corpus previously described, extracted 11,000 formulas where at least 1 cell and less than 25% of cells result in an error value. Of these 11,000 formulas, 7,200 have a single column input and 3,800 have multiple column inputs (on average 3.4). We use this dataset to evaluate the impact of execution-guidance in string data cleaning.

4.3 Baselines

We compare against various baselines as summarized in Table 4:

- (1) WMRR [2]: an unsupervised approach to learn weighted data rectifying rules based on functional dependencies. Since the tool is not publicly available, to evaluate against WMRR we reimplement it based on their paper description.
- (2) HoloClean [19]: a popular data repair tool based on probabilistic inference, which can repair qualitative and statistical errors. We run the code released by the authors on GitHub. HoloClean originally requires that users provide denial constraints. To evaluate in a fully unsupervised setting, comparable to DATAVINCI, we use a single vacuous denial constraint (specifically, column 1 = column 1).
- (3) Raha [11]: an ensemble-like system that combines multiple error detection systems and semi-supervision to train an error detection system. As Raha requires the user to annotate examples, in our evaluation we take the first (top-to-bottom) 5 groundtruth errors per column and provide these as examples.

Table 5: Error detection performance across datasets. DATAVINCI outperforms in terms of precision on Wikipedia and Excel, and in terms of recall on our synthetic benchmark. We report Potter’s Wheel and Auto-Detect results using the annotations released with the Auto-Detect paper. For synthetic benchmarks the groundtruth is taken as the original table which can also have inherent data errors which will skew the precision and F1 scores explained in Section 4, hence these metrics are reported with a (*).

System	Wikipedia		Excel		Synthetic		
	Precision	Fire Rate	Precision	Fire Rate	Precision*	Recall	F1 Score*
WMRR	70.0	2.93%	65.8	2.76%	55.3	66.8	60.5
HoloClean	67.0	3.87%	65.2	2.50%	52.1	64.1	57.5
Raha	68.9	4.03%	66.4	3.74%	59.5	68.2	63.6
Potters-Wheel	66.2	-	-	-	-	-	-
Auto-Detect	78.5	-	-	-	-	-	-
T5	60.8	27.47%	53.8	19.02%	40.5	56.3	47.1
GPT-3.5	73.9	10.99%	60.4	11.71%	50.1	69.8	58.3
DATAVINCI	80.1	16.85%	75.1	14.39%	67.4	73.4	70.3

- (4) Auto-Detect[7]: a co-occurrence-based error detection system, which also uses regular-expressions to generalize values. We use the Wikipedia results released with the paper. Unfortunately, we are unable to run the tool on all our benchmarks as it is not available publicly and as a result only report performance on Wikipedia.
- (5) Potter’s Wheel [18]: a seminal data error detection and (semi-supervised) correction system based on functional dependencies. We leverage the Potters Wheel’s error detection annotations on the Wikipedia dataset released in the original Auto-Detect paper. Since we do not run Potter’s Wheel system and rely on the Wikipedia annotations released, we only report Potter’s Wheel results on the Wikipedia benchmark.
- (6) T5[17]: a popular transformer-based encoder-decoder model pretrained on text. We fine-tune T5 for the task of data repair. Since T5 is a text generation model we encode each column as a stringified list of column values separated by a [SEP] token. The model is trained end to end to generate the repaired column, given the potentially noisy column as input. The training data consists of 100K dirty samples (generated by the same approach used in our synthetic benchmarks) and we task it with generating the original columns. Because we run T5 on a single column at a time, it does not consider other columns while repairing.
- (7) GPT-3.5[3]: a state-of-the-art transformer-based decoder-only model. We use the same input structure used to train T5 to include the target column in GPT-3.5’s prompt. We use GPT-3.5 in a fewshot setting, providing three static examples of a dirty column and the cleaned output the model needs to generate. The static examples are from Excel (disjoint from our benchmarks) and are hand annotated. We report results at temperature 0 and top-1 generation. After experimenting with multiple temperatures, we found temperature 0 works best on average based on precision and F1 score across all benchmarks.

To evaluate unsupervised repair when using detection-only systems, such as Raha, Auto-Detect and Potter’s Wheel, we add a call to GPT-3.5 where we include the outlier value and its column header along with 10 sample values selected based on spatial proximity (5 rows above and below and 3 columns to the left and to the right with headers). We ask the model to generate the repaired value. We sample values to fit in the fixed prompt length of 4,000 tokens and make individual repair calls for each outlier detected.

5 RESULTS AND DISCUSSION

We explore the following research questions:

- RQ1. Can DATAVINCI accurately detect string errors?
- RQ2. Can DATAVINCI accurately repair string errors?
- RQ3. Can DATAVINCI use program execution to improve repairs?
- RQ4. How do DATAVINCI’s design decisions impact performance?

5.1 Error Detection (RQ1)

Like prior work [7, 11, 19] we report precision for detection. We leverage existing annotations where possible and otherwise manually annotate systems’ predictions. We report precision, recall, and F1 on our synthetically corrupted dataset, computed with respect to our corruptions, as a result precision/F1 score can be deflated from preexisting data errors.

In addition to standard metrics like precision, we also report each system’s *average fire rate*. We define this as the average fraction of cells in a column that are labeled as data errors.

Table 5 presents error detection results for DATAVINCI and baselines across three benchmarks. We find that DATAVINCI outperforms baselines in terms of precision on both the Wikipedia and the Excel benchmarks, despite having a higher firing rate than all but one baseline (T5). Auto-Detect, which is well-suited to the type of mistakes present in the Wikipedia dataset, performs competitively. Overall, we find that error detection is relatively easier on the Wikipedia benchmark, where tables on average have fewer rows, compared to the Excel benchmark.

Table 6: Error repair performance across datasets. As described in Section 4 for the Wikipedia and Excel benchmarks we report repair precision split into (1) **Certain**: repairs that are certain (based on hand annotation) and, (2) **Possible**: repairs that are reasonable but groundtruth cannot be uniquely determined. For synthetic benchmarks the groundtruth is taken as the original table which can also have inherent data errors which will skew the precision and F1 score hence, these metrics are reported with a (*). Recall is computed as the percentage of cases correctly repaired out of the total errors synthetically introduced.

System	Wikipedia		Excel		Synthetic		
	Precision (Certain)	Precision (Possible)	Precision (Certain)	Precision (Possible)	Precision *	Recall	F1 Score *
WMRR	61.1	57.8	59.2	55.6	43.2	61.1	50.6
HoloClean	58.4	55.6	59.0	54.9	41.3	58.6	48.5
Raha + GPT-3.5	58.6	54.8	56.4	53.5	45.2	62.0	52.3
Potter’s-Wheel + GPT-3.5	56.2	52.0	-	-	-	-	-
Auto-Detect + GPT-3.5	66.9	63.3	-	-	-	-	-
T5	41.0	37.8	37.7	35.2	27.9	47.0	35.0
GPT-3.5	63.9	55.5	52.1	48.9	38.2	63.8	47.8
DATAVINCI	71.3	64.9	71.2	64.6	54.1	68.9	60.6

On our synthetic benchmark, we find DATAVINCI achieves the highest recall followed by GPT-3.5. The learning-based approach taken by Raha also results in a recall rate that is comparable to the more expensive GPT-3.5-based solution.

When performing qualitative inspection of the errors detected, we find that GPT-3.5 can identify errors in semantic substrings well. For example, in the following column of financial quarters, $\{Q1-22, Q4-21, Q5-20, Q2-20, Q1-21\}$ GPT-3.5 correctly identifies the outlier to be $Q5-20$. However, GPT-3.5 fails to detect syntactic errors like $S1.4$ in the column, $\{S.1.2, S.2.3, S1.4, S.1.3, S.2.1\}$, where $S1.4$ is missing a period after S . Neural models like GPT-3.5 struggle at recognizing these patterns whereas DATAVINCI can detect such errors using regular-expression-based patterns.

Other tools, like Auto-Detect, work well on syntactic errors but fails on semantic repairs. For example, consider the column of county and a numeric ID separated by a hyphen $\{Alpine_231, Kings_721, Lake_201, Santa\ Clara_246, Nevad210\}$ the correct repair here is $Nevad210 \rightarrow Nevada_210$. This error involves a combination of syntactic and semantic inconsistency which most baseline systems struggle with. DATAVINCI combines semantic information via masking into its pattern based syntactic repair engine and thus detects this error (and generates the correct repair).

5.2 Error Repair (RQ2)

For repair, we find that often there are cases where various possible data repairs are reasonable and the correct repair cannot be uniquely identified. To account for this, we annotate repair suggestions as *possible* if this is the case, and then annotate the suggestion as correct (i.e. reasonable) or not. In our results, we report repair precision for certain cases and precision for *possible* cases separately for completeness.

Table 6 shows the performance of DATAVINCI and baselines systems for repairing data on our benchmarks. Note that repair metrics combine: (1) detection (as a system must identify an error to fix it), and (2) whether the repair matches the ground-truth repair.

Table 7: Table showing repair precision as the percentage of errors that DATAVINCI and baseline systems can repair correctly out of the errors that were correctly detected by each system. DATAVINCI has the highest repair rate compared to baseline systems.

System	Wikipedia	Excel	Synthetic
WMRR	87.3	89.9	78.2
HoloClean	87.1	90.5	79.3
Raha + GPT-3.5	85.0	85.0	76.0
Potter’s-Wheel + GPT-3.5	84.9	-	-
Auto-Detect + GPT-3.5	85.2	-	-
T5	67.4	70.1	68.8
GPT-3.5	86.5	86.3	76.3
DATAVINCI	89.0	91.2	80.3

We find that DATAVINCI outperforms all baselines in terms of both certain and possible repairs on Wikipedia and Excel benchmarks and has the highest precision, recall and F1 score on the synthetic test set. Raha+GPT-3.5 and Auto-Detect+GPT-3.5 have high precision (Wikipedia), but we find that they have different behaviors. Specifically, Auto-Detect (by design) does not support inter-column dependencies, while Raha struggles to detect intra-column patterns. WMRR and Potter’s-Wheel capture both inter- and intra-column dependencies well but struggle with semantic repairs as they do not detect these issues.

Both GPT-3.5 and T5 perform significantly worse on the synthetic dataset as our noise operations predominantly introduced syntactic errors with minimal semantic content.

Table 7 shows the precision rates when we only consider correctly detected errors for each system as a way to disentangle the detection and correction effectiveness of each system. We find that repair precision is substantially higher across the board, if we only

Category	Player ID	DataVinci	
Professional	Ind-674-PRO	US-837-PRO	✓
Junior	US-823-JUN	GPT-3.5	✗
Junior	US-237-JUN	USA-837-PRO	✗
Staff	QUAL-47	WMRR	✗
Staff	QUAL-21	No Detection	✗
Professional	Zim-843-PRO	Raha	✗
Junior	Eng-781-JUN	US-837-JUN	✗
Professional	Aus-664-PRO	T5	✗
Staff	QUAL-88	usa-837	✗
Junior	Ind-473-JUN	Ground Truth	✗
Professional	usa_837	US-837-PRO	✗
Junior	Eng-573-JUN		
Professional	Zim-392-PRO		
Staff	QUAL-10		

Figure 5: Example from Excel benchmarks where DATAVINCI generates the correct repair while baseline systems fail. The ground truth is US-837-PRO. DATAVINCI combines semantic and syntactic substrings in its pattern and repairs the column correctly. Analysis for this example is presented in Figure 2.

consider correct detections, and DATAVINCI outperforms in all three benchmark sets.

We look at cases where DATAVINCI is able to accurately repair data errors while baselines fail. We find that these are mostly where either (1) DATAVINCI is able to leverage its semantic masking to suppress false positives; or (2) DATAVINCI detects a semantic anomaly using patterns. Figure 5 shows an example from the Excel benchmarks, which contains a tournament table having columns category (Junior or Professional) and Player-ID which has three components (Country code, unique numeric ID, first three letters of category). For non competing players, the ID is QUAL- followed by a unique numeric ID. *usa_837* is an outlier in the Player-ID column and the correct repair should change it to *US-837-PRO*. As highlighted above, DATAVINCI utilizes (1) semantic masking to repair *usa* → *US* and uses (2) patterns, paired with concretization value constraints, to detect that the category substring is *PRO*.

We also look at cases where DATAVINCI failed to generate the correct repair but one of the baseline systems succeeded. We find that these cases were mostly the result of either (1) the column does not have any significant patterns due to irregular data, or (2) the error rate is too high and as a result the outlier is covered by a significant pattern. Figure 6 shows one example for each case from the Excel benchmarks along with the incorrect repair generated by DATAVINCI and the correct repair generated by the baseline. In the first example, DATAVINCI learns two significant patterns $R_M = \{[A-Z]^+, [A-Z]^+0\}$ and hence, does not detect the error. In the second example, since the column contains irregular data, DATAVINCI is unable to learn a significant pattern $R_M = \{\}$ and does not detect any errors.

Action	DataVinci	Option	DataVinci
Cross	No Detection	Beaker	No Detection
Pass0		23.4	
Pass		RD12	
Flash	Pass0 → Pass	Rd13	Rd13 → RD13
Flash0	Flash0 → Flash	RD14	
Cross0	Cross0 → Cross	Flask	
Pass		10.6	
Flash		@Filed	

Figure 6: Example from Excel benchmarks where DATAVINCI fails to generate the correct repair but a baseline system succeeds. Error cells are highlighted. We show all repairs (denoted by →) suggested. DATAVINCI is unable to detect these errors because they either ① satisfy a significant pattern, or ② the column has irregular data and no significant pattern is learned.

Table 8: Execution success rates at formula and cell-level after applying repair suggestions for each system on our Excel Formulas benchmark. We split out formulas that depend on single and multiple columns. We report the *No Repair* (i.e. starting point) success rates first for comparison. We do not report HoloClean as it is expensive to run and did not complete in the 24 hours time limit. Applying DATAVINCI’s execution-guided repairs leads to more successful executions.

Type	Single Column		Multi Column	
	Formula	Cell	Formula	Cell
No Repair	0.0%	85.8%	0.0%	81.4%
WMRR	32.6%	94.4%	29.6%	90.1%
Raha + GPT-3.5	34.5%	92.6%	31.4%	88.3%
T5	11.2%	89.4%	6.4%	86.2%
DATAVINCI Unsupervised	43.2%	94.3%	35.7%	90.9%
DATAVINCI+Execution	54.0%	96.5%	47.8%	94.0%

5.3 Execution-Guidance (RQ3)

We use our Excel Formula benchmark to evaluate the extent to which DATAVINCI can use execution information to provide improved repairs. We report two execution metrics: the fraction of cells that no longer result in an error value, as well as the fraction of columns where no cell results in an error value (i.e., the formula succeeds fully).

To carry out this experiment, we run all baselines, apply repair suggestions *only* on values that are an input into a row that has an error value when the formula is originally executed. We report formula-level and cell-level successful execution rates after applying each systems’ suggestions.

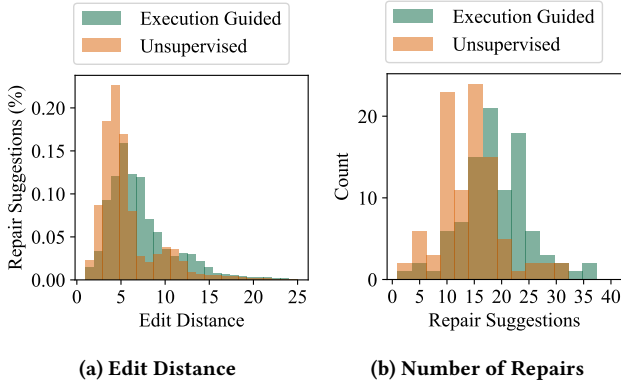


Figure 7: Comparison of unsupervised and execution-guided DATAVINCI. Both the (a) distribution of original value to suggested repair edit distances and (b) number of repairs per column move higher when given execution information. Jointly, this suggests that with execution information DATAVINCI can offer more repairs, with higher complexity (as proxied by string edit distance).

Table 8 summarizes our results. We find that DATAVINCI with execution-guided learning improves over all baselines⁵ (including the fully unsupervised DATAVINCI). While all systems have implicit access to execution information, since we only apply their repairs to inputs associated with erroneous executions, DATAVINCI with execution-guided learning is the only system that also incorporates this information (by affecting the patterns learned: R_m) when learning how to repair the data error.

Figure 7 shows that the distribution of string edit distances from original data error to repaired suggestion, as well as the number of repairs per column. We find that when provided with execution information DATAVINCI can produce more repairs, and these repairs tend to have a higher distance to the original value (implying possibly more complex repairs). For example, Figure 8 shows an example where DATAVINCI with execution learns the correct repair resulting in successful execution of the formula while the unsupervised variant is unable to provide any suggestions because the pattern $C[0-9]\{2\}$ repeats enough times to be considered as a significant pattern.

5.4 Design Decision (RQ4)

We study how the various design decisions impact the performance of DATAVINCI. Namely, we investigate the importance of semantic abstraction/concretization, concretization value constraints, and ranking. We carry out experiments on our synthetically corrupted benchmark. We summarize our results in Table 9.

5.4.1 Semantic Substrings. To evaluate the impact of having semantic information in repairs (see Section 3.2), we evaluate two versions of DATAVINCI. We implement a version of DATAVINCI that treats all strings as purely syntactic (*No semantic abstraction*) and a version that can perform semantic abstraction but is restricted

⁵we exclude HoloClean as it did not scale to this task. We let HoloClean run for 24 hours and it only covered 24% of the formula benchmarks.

Option	Formula
Chrome23	23
Chrome21	21
C30	#ERROR
Chrome19	19
Chrome22	22
C15	#ERROR
C26	#ERROR
Chrome17	17
Chrome20	20
Chrome25	25
Chrome18	18

```
=RIGHT(A48,
LEN(A48)
- SEARCH("Chrome",A48)
- LEN("Chrome")
+1)
```

DataVinci (Unsupervised) ✗

No Detection

DataVinci (Execution) ✓

C30 → Chrome30

C15 → Chrome15

C26 → Chrome26

Figure 8: Example from Excel Formulas benchmark where DATAVINCI with execution-guided repair can provide repair suggestions that lead to successful formula execution but the unsupervised variant cannot because the outlier pattern, $C[0-9]\{2\}$ occurs frequent enough to be considered a significant pattern. Error output values are shown in red font, and data errors in input values are highlighted.

Table 9: Repair precision, recall and F1 score on our synthetically corrupted benchmark for different DATAVINCI ablations. Full DATAVINCI outperforms all ablations. Removing semantic abstraction and removing learned concretization constraints (which concretize abstract edits) have the most impact on F1 score.

Model	Precision	Recall	F1
No semantic abstraction	50.3	62.9	55.9
Limited semantic concretization	52.0	65.6	58.0
No learned concretization	46.3	51.0	48.5
Edit distance ranking	53.2	67.1	69.3
DATAVINCI	54.1	68.9	60.6

to re-use the same substring for concretization, meaning it can not repair errors in semantic substrings (*Limited semantic concretization*). We find that both versions result in a lower performance compared to full DATAVINCI, but removing semantics altogether has a comparatively larger impact on precision and F1.

5.4.2 Concretization and Ranking. To study the effect of learned concretization value constraints and ranking (see Section 3.4 and 3.5), we design two ablated version of DATAVINCI, (1) where DATAVINCI does not learn to concretize abstract edits, instead enumerates all candidates and directly passes them through to the heuristic ranker (*No learned concretization*); and (2) a version where all candidates are ranked just based on the shortest string edit distance with respect to the original data error (*Edit distance ranking*). Table 9 shows that while removing either learned concretization or ranking has a negative impact on performance, removing learned concretization constraints has a larger effect.

Table 10: Comparing time (milliseconds), disk space (MB), and GPU plus CPU memory used (MB), averaged over Wikipedia benchmarks. (*) denotes systems which we did not run and the reported stats are the ones published by Auto-Detect authors for this dataset. Raha/Potter’s Wheel/Auto-Detect only includes the detection time as repair is performed by GPT head in our experiments. GPT-3.5 was used via API and the reported time includes network latency.

System	Time(ms)	Disk(MB)	Memory(MB)
WMRR	247.4	4.6	914.5
HoloClean	1049.3	996.3	1647.2
Raha	321.8	65.3	645.4
Potter’s-Wheel*	110.0	-	-
Auto-Detect*	290.0	-	-
T5	858.3	886.2	1534.2
GPT-3.5	1325.6	-	-
DATAVINCI	261.5	5.6	10.5

5.5 Runtime Performance

Table 10 shows the average time taken, disk space used and RAM + GPU VRAM used to detect (Raha, Potter’s Wheel, Auto-Detect) or detect+repair (WMRR, HoloClean, T5, GPT-3.5, DATAVINCI) errors on the Wikipedia benchmark. We report Potter’s Wheel and Auto-Detect based on the Auto-Detect paper [7] which reports these metrics in a similar environment and on the same benchmarks. We note that GPT-3.5 includes network time. We find that in terms of time and disk space, DATAVINCI is competitive with alternatives such as WMRR, Raha, and Auto-Detect, while using substantially less RAM. Since, T5 and HoloClean also utilize a GPU and don’t run purely on CPU memory we report the sum of GPU and CPU memory usage under Memory. We only report inference resources but it is worth noting that T5 also requires training which took 4 hours on a K80 GPU.

HoloClean and T5 are the most resource intensive systems, as a result of their implementation complexity. DATAVINCI, WMRR and Raha are up to 4 times faster than HoloClean/GPT-3.5 and require fewer resources to run.

6 LIMITATIONS

We describe some limitations of DATAVINCI. We have only evaluated DATAVINCI on English language values and applicability to non-English datasets may be limited. DATAVINCI does not handle inter-table constraints, limiting its effectiveness when data consistency relies on relationships across multiple tables. DATAVINCI relies on identifying recurring patterns to perform error detection/repair, which may not occur in all data. Execution-guided repair may mitigate this limitation but its applicability is limited by the availability of programs that read the target data and the ability to easily execute these programs.

7 RELATED WORK

Data error detection has been the subject of active investigation in the data management community. Prior work has developed systems that can identify qualitative data errors (e.g. incorrect value structure) or quantitative data errors (e.g. unlikely values given a distribution). Often systems further make distinctions between syntactic issues and semantic data errors [11]. The seminal Potter’s Wheel [18] system introduced an interactive data cleaning system where users employ a spreadsheet-like environment to annotate data errors with corrections, which the system then learns to apply throughout the data. More recent work like AutoDetect [7] uses large-scale co-occurrence statistics, along with pattern-based generalization, to achieve high-precision error detection. Raha [12] achieves configuration-free error detection by combining many error strategies, clustering data based on these strategies’ annotations, efficiently gathering user feedback on possible errors from these clusters, and then training a model on this feedback to detect errors throughout the dataset. Commercial platforms like Trifacta [1] and PowerBI [13] typically offer data cleaning based on a fixed set of patterns or common data issues (e.g. leading spaces). In contrast to this line of work, DATAVINCI does not use fixed patterns to detect errors but rather learns them. DATAVINCI does not only detect data errors but also repairs them. In addition, DATAVINCI can detect and repair errors in strings with both syntactic and semantic substrings.

While detecting data errors can help users identify issues in their dataset, correcting these errors can also be costly. As a result, past work has explored not only detecting but also repairing errors identified. HoloClean [19] allows users to (optionally) specify denial constraints, which it combines with error detectors, to build a probabilistic data cleaning program, which unifies these different signals. WMRR [2] presents an unsupervised approach to learning cleaning rules, removing the need for users to specify denial constraints or resolution rules. ActiveClean [9] and Baran [10, 11] are semi-supervised tools to repair data based on few user examples. With the exception of WMRR (which our evaluation shows achieves lower detection/repair performance), this line of work requires a human in the loop. Like this work, DATAVINCI can provide repair suggestions for errors detected. In contrast to these systems, DATAVINCI uses a pattern-based approach to data repair, does not require any user specification in the form of constraints or annotated examples, and can repair strings that have a combination of both syntactic and semantic substrings.

Transformer models [24] have recently gained a lot of popularity and language [17] and code tasks [27]. As part of our evaluation, we employ T5 and GPT-3.5 as baselines to perform data error detection and repair. Prior work [14] has explored using foundation models for data management tasks including data validation and cleaning.

DATAVINCI can repair strings that contain both syntactic and semantic substrings, as described in Section 3.2. FlashGPT [25] showed that LLM-based semantic transformations can be integrated into a programming-by-example synthesizer that learns syntactic string transformations. More recently, SMORE [4] presented a formalization of semantic regular expressions, which generalize traditional regular expressions to include semantic substrings. SMORE can learn such semantic regexes given positive/negative examples.

Potter’s Wheel [18] anticipated the combination of semantic substrings and syntactic substrings for data cleaning, and allowed users to define membership tests associated with semantic types, and used these definitions to guide its transformations. Like this line of work, DATAVINCI combines syntactic and semantic information. Like SMORE, DATAVINCI leverages regular expressions to formally describe string values. In contrast to SMORE, FlashGPT, and Potter’s Wheel, DATAVINCI does not have user examples or definitions but rather learns patterns fully unsupervised. Like SMORE, DATAVINCI uses an LLM to identify semantic substrings, but when learning the regular expression DATAVINCI employs abstraction/concretization, which allows it to use an existing regular expression learner [15].

8 CONCLUSION

In this paper we propose DATAVINCI, a tool for automatic repair of string data errors. DATAVINCI learns majority patterns over columns and uses these to detect string data errors. Because errors may occur in strings with both syntactic and semantic substrings, DATAVINCI employs an LLM to mask semantic substrings before learning a pattern. DATAVINCI suggests repairs for the detected errors by deriving minimal edits to the data error that lead to satisfying a majority pattern. Because majority patterns may not always occur or capture errors, DATAVINCI can address this challenge by incorporating program execution information. We evaluate DATAVINCI against 7 baselines on four existing and new benchmarks and show DATAVINCI achieves higher detection and repair performance. We release scripts to reproduce our novel benchmarks for future data cleaning research.

9 ACKNOWLEDGMENTS

We would like to thank Yair Helman, Einam Schonberg, Tal Kariv, Noa Feiger, Israela Solomon, Irena Berezovsky, Guy Hunkin, David Schwartz, Danielle Maor, Danielle Fainman, Tsofiya Aiello, Jack Williams and Andy Gordon for their feedback on this research.

REFERENCES

- [1] [n.d.]. Trifacta: Data Cleansing Tool. <https://docs.trifacta.com/display/AAC/Data+Cleansing+Tool>. Accessed: August 23, 2023.
- [2] Hiba Abu Ahmad and Hongzhi Wang. 2020. Automatic weighted matching rectifying rule discovery for data repairing: Can we discover effective repairing rules automatically from dirty data? *The VLDB Journal* 29, 6 (2020), 1433–1447.
- [3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>
- [4] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Isil Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. *arXiv preprint arXiv:2305.10401* (2023).
- [5] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*. 2201–2206.
- [6] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.
- [7] Zhipeng Huang and Yeye He. 2018. Auto-detect: Data-driven error detection in tables. In *Proceedings of the 2018 International Conference on Management of Data*. 1377–1392.
- [8] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zraggen, Arvind Satyanarayan, Tim Kraska, Çağatay Demiralp, and César Hidalgo. 2019. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.
- [9] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.
- [10] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective error correction via a unified context representation and transfer learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1948–1961.
- [11] Mohammad Mahdavi and Ziawasch Abedjan. 2021. Semi-Supervised Data Cleaning with Raha and Baran. In *CIDR*.
- [12] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Maden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*. 865–882.
- [13] Microsoft. [n.d.]. *Clean, transform, and load data in Power BI*. <https://learn.microsoft.com/en-us/training/modules/clean-data-power-bi/>. Accessed: August 23, 2023.
- [14] Avnika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *arXiv preprint arXiv:2205.09911* (2022).
- [15] Saswat Padhi, Prateek Jain, Daniel Perelman, Aleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.
- [16] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern functional dependencies for data cleaning. (2020).
- [17] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. <http://jmlr.org/papers/v21/20-074.html>
- [18] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter’s wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.
- [19] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1190–1201. <https://doi.org/10.14778/3137628.3137631>
- [20] Eric Sven Ristad and Peter N. Yianilos. 1996. Learning String-Edit Distance. *IEEE Trans. Pattern Anal. Mach. Intell.* 20 (1996), 522–532.
- [21] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, Mohammad Raza, and Gust Verbruggen. 2022. CORNET: A neurosymbolic approach to learning conditional table formatting rules by example. *arXiv preprint arXiv:2208.06032* (2022).
- [22] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.
- [23] Stanford University. 2016. CS124 Lecture Notes. <https://web.stanford.edu/class/cs124/lec/med.pdf>
- [24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>
- [25] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–25.
- [26] Pei Wang and Yeye He. 2019. Uni-detect: A unified approach to automated error detection in tables. In *Proceedings of the 2019 International Conference on Management of Data*. 811–828.
- [27] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. <https://doi.org/10.18653/v1/2021.emnlp-main.685>