Table Autofilling By Example

Taylor Jiang[*]
Cornell University
Ithaca, NY, USA
sj649@cornell.edu

Arya Song
Cornell University
Ithaca, NY, USA
ys434@cornell.edu

ABSTRACT

The expectation for proficient data manipulation and analysis skills has been coevolving with the volume and diversity of data over years. Despite the availability of analytical tools like Excel spreadsheets aimed at simplifying data operations, individuals with the need of data manipulation lack the requisite expertise. Motivated by the challenges individuals face when cautiously repeating manual calculations due to insufficient knowledge of data manipulation methods, this paper proposes a pipeline designed to automate the calculation for values based on only a few user input examples along with some complete columns associated with the column to autofill. Limited work has been done in this direction, possibly due to the varied domain knowledge required for specific autofilling tasks. While Gulwani et al. have focused on string formatting and developed programming by examples (PBE) and domain-specific language (DSL) techniques for autofilling, this paper concentrates on numeric value autofilling (in addition to some basic string formatting). This paper also attempts to offer a framework that from data cleaning to a user-friendly interface, which helps users who are less inclined to data manipulation to achieve their task and can serve as a starting point for people who want to develop their domain knowledge-specific tools.

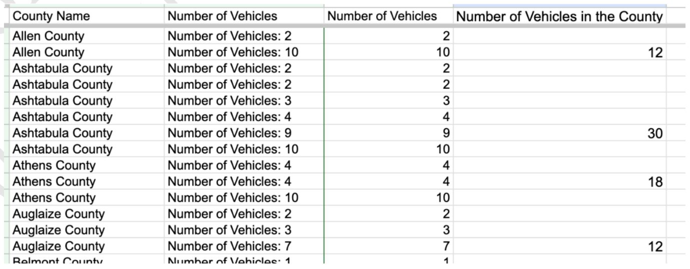
ACM Reference Format:
Taylor Jiang and Arya Song. 2023. Table Autofilling By Example. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In today's professional landscape, data collection and analysis have become integral to a wide range of fields, spanning from data-intensive machine learning endeavors in computer science to what may seem like simpler tasks in Excel and other spreadsheet applications. Contemporary spreadsheets aim to address a common human challenge: simplifying the process of visualizing and comprehending data for a broad audience. However, this "everyone" includes individuals who are well-versed in spreadsheet formulas, which can be intricate to the point that dedicated courses are designed to teach them. To streamline repetitive tasks like manually inputting dates, sequential numbers, and even certain column values that depend on each other, the "autofill" feature was created. For instance, if you input 1 and 2 in the first two rows of a column, the spreadsheet intelligently continues the sequence with ascending integers (e.g., 3, 4, 5, and so on) without requiring a complex formula.

[*]Both authors contributed equally to this research.

Nonetheless, the current autofill functionality is limited. As the complexity of column manipulations increases, the autofill feature's ability to automatically deduce operations falters, necessitating user intervention in the form of formula inputs. For instance, when creating a column such as "Number of Vehicles" as depicted in Figure 1, users are obliged to construct a formula based on the values in the preceding column. The principal achievement of major spreadsheet applications lies in their capacity to reduce the demand for proficiency in a full-fledged programming language, shifting the requirement to a more accessible formulaic language, which helps users translate their data manipulation objectives.



County Name	Number of Vehicles	Number of Vehicles	Number of Vehicles in the County
Allen County	Number of Vehicles: 2	2	
Allen County	Number of Vehicles: 10	10	12
Ashtabula County	Number of Vehicles: 2	2	
Ashtabula County	Number of Vehicles: 2	2	
Ashtabula County	Number of Vehicles: 3	3	
Ashtabula County	Number of Vehicles: 4	4	
Ashtabula County	Number of Vehicles: 9	9	30
Ashtabula County	Number of Vehicles: 10	10	
Athens County	Number of Vehicles: 4	4	
Athens County	Number of Vehicles: 4	4	18
Athens County	Number of Vehicles: 10	10	
Auglaize County	Number of Vehicles: 2	2	
Auglaize County	Number of Vehicles: 3	3	
Auglaize County	Number of Vehicles: 7	7	12
Belmont County	Number of Vehicles: 1	1	

Figure 1: Table of EMS Agencies

Another example task at hand given some data as shown in Figure 1 is to estimate the market size of an ePCR system for ambulances in the U.S. Given the data of EMS agencies with the number of vehicles and their specific location (the county), we wish to sum the number of ambulances in each county. (See Figure 1, the last column is the sum you want to generate.) Existing autofill features also fail to infer the user's intention. Our project is dedicated to investigate the viability of a system that possesses the capability to autonomously populate the target column with relevant data, relying solely on a limited number of examples without necessitating manual formulaic input from the user.

2 WORKS RELATED

Data manipulation by example usually contains two components: a user interactive component which is designed to acquire correct examples from users, and an automation algorithm or inductive synthesizer that modifies or generates values in the data set based on the user inputs. To offer a broader context, we present an overview outlining the present advancements in the following areas: 1) Example-based data manipulation, and 2) User interaction in Data Management.

2.1 Example-Based Data Manipulation

In spreadsheet data manipulation through examples, Sumit Gulwani et al. [2] present a noteworthy approach that leverages domain knowledge and input-output examples. While the implementation and deployment details may surpass our current capabilities, the paper serves as a valuable guide for formulating problems and devising solutions. It advocates for the segregation of string and numerical value manipulation.

Expanding on their prior work in programming by example (PBE) and domain-specific language (DSL), Gulwani et al. [6] integrate probabilities semantics to address challenges related to type of format modifications. This contribution includes examples demonstrating the handling of unstructured, non-uniform, and ambiguous data, thereby enriching the scope of data manipulation possibilities.

Another example of data manipulation based on the examples, the FalshExtract framework [4], offers a comprehensive view through high-level pseudo code for its modular inductive synthesis algorithm. This serves as a valuable reference when implementing core functions for data synthesis. Furthermore, the paper introduces an innovative input format utilizing highlighting, providing an alternative approach by identifying correct and expected examples among all potential values.

Milo et al.'s work on cleaning data with constraints and experts [1] introduces a graph structure designed to aid error detection. This structure updates based on domain expert responses to questions regarding the correctness of values or queries about correct inputs. These insights shed light on potential strategies for cleaning generated data or enhancing the synthesis of data from the outset.

2.2 User Interaction in Data Management

Within the study on better design of user interaction in data management, various perspectives contribute to shaping our approach. The paper "A Spreadsheet Algebra for a Direct data manipulation Query Interface" [5] provides insights that, while diverging from our primary focus, underscore the human preference for hands-on and concrete tasks like providing examples over engaging in abstract or formulated tasks. This inclination serves as a driving force behind our research in example-based inference and data manipulation.

In the context of interactive visual specification for data transformation scripts, "Wrangler" [3] stands out for showcasing the effectiveness and engagement achievable through user interaction and intervention in the data manipulation process. Beyond mere inspiration, the paper introduces practical ideas such as utilizing natural language descriptions to enhance communication with users, enriching our perspective on user interactions.

Looking ahead, "PIClean: A Probabilistic and Interactive Data Cleaning System" [5] seamlessly leads us to the next phase of our project. Acknowledging our strict assumptions and the likelihood of errors in our methods' output, Zhuoran Yu et al.'s work offers valuable guidance on incorporating probabilistic error detection approaches and iterative workflows. This provides a tangible reference for how to effectively clean generated data through user intervention, aligning with our overarching goals in user-centric data management.

3 FORMAL PROBLEM DEFINITION

The difficulty that this paper attempts to address and the scope of this paper can be stated as the following. Given a set of complete lists and a few examples in the incomplete column, the proposed solution finds a valid function (i.e. a numeric relationship or a formatting pattern) and returns all the columns including the autofilled column. To find a valid function, the proposed solution would iterate through all the candidate functions and check them against the examples given by the user.

3.1 Preliminary Assumption

The proposed solution makes the following assumptions:

- All the columns provided by the user are relevant to the values of interest.
- All the examples provided by the users are correct.
- All the tables are vertical, meaning that we will look at the table row by row, and autofill the table row by row, instead of column by column. In most cases, this implies that the table has a very limited number of columns but a substantial number of rows.
- There is only one column that needs to be autofilled. If the user wants to autofill more than one columns, the user can input the table multiple times to our pipeline.
- The input tables are mostly clean. However, it is acceptable to leave a small percentage of entries empty. The empty entries will be filled with 0 for numeric tables. The exception of the empty entries will be addressed in the section 4.3.4.

Definition 1 (Input and Output). Let the input table to the pipeline is T_{in} and the output of the table from the pipeline is T_{out}.

3.2 Problem Statement

Definition 2 (Length of Column). *For the column c, length of column c, len(c), is defined as the number of entries from the first non-empty entries in c to the last non-empty entries in c.*

Note that the length of column counts the empty entries that are between two non-empty entries.

Definition 3 (Completed Column). *Given the table T, it contains n columns, i.e. $C = \{c_1, ..., c_n\}$. For all $i \in C$, Complete(c_i) = TRUE, if there exists $j \in C$ such that len(c_j)< len(c_i). Complete(c_i) = FALSE, otherwise.*

Note that since we allow the empty entries to exist in the completed columns and the empty entries may be in the end of the completed columns, it is not necessary true that every completed column in the table has the same length. But the length of the completed column is guaranteed to be large than that of the filling column.

Definition 4 (Filling Column). *The table T contains n columns, i.e. $C = \{c_1, ..., c_n\}$. For all $i \in C$, Fill(c_i) = TRUE, if len(c_i) > 3 and there not exists $j \in C$ such that len(c_j)< len(c_i). Otherwise, Fill(c_i) = FALSE.*

The column that needs to be filled should be the column of the smallest length in the table. Note that we assume the number of example entries in the c_{fill} is larger than 3, which means we need

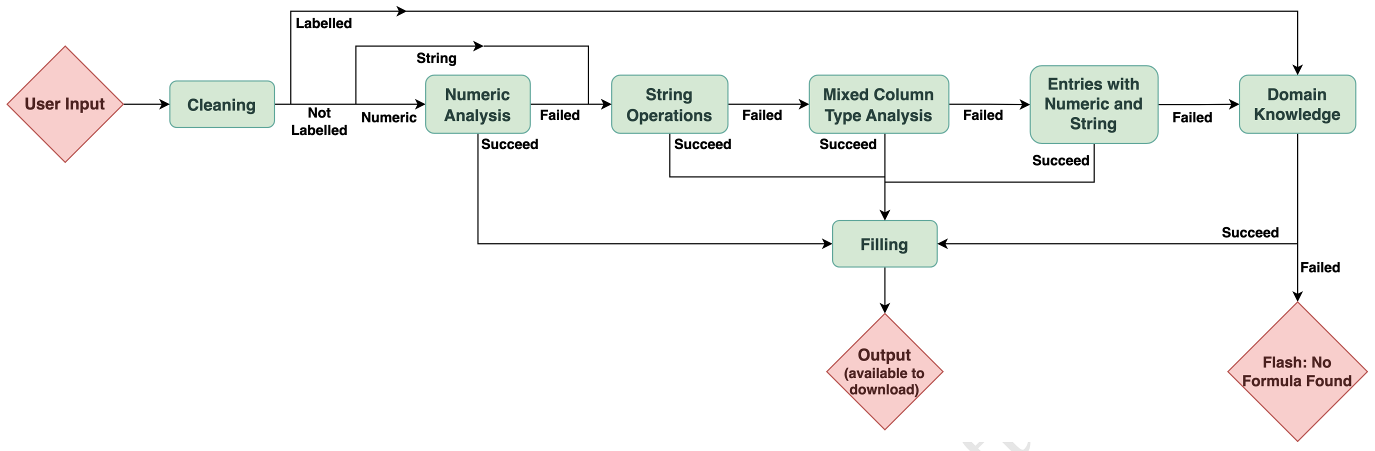


Figure 2: Pipeline

at least 3 examples in the filling column to determine the formula of the table.

Proposition 1 *Complete(c_i) = not Fill(c_i).*

Suppose the column c_i is complete, i.e. Complete(c_i) = TRUE, then there exists another column c_j such that len(c_j) < len(c_i). Thus, c_i can't be the column with smallest length, which implies that Fill(c_i) = FALSE.

Suppose the column c_i is not complete, i.e. Complete(c_i) = FALSE, then there does not exist another column c_j such that len(c_j) < len(c_i). Thus, c_i is the column with smallest length, which implies that Fill(c_i) = TRUE.

Definition 5 (Valid Input Table). *Given the table T, it has columns $C = \{c_1, ..., c_n\}$. $Valid(T)$ = TRUE, if there exists one and only one column c_i such that Fill(c_i) = TRUE, and for all other columns $j \in C$ and $j \neq i$, Complete(c_j) = TRUE. Otherwise, $Valid(T)$ = FALSE.*

As mentioned in 3.1, the valid input table only contains one column that need to be filled, and all other columns are completed.

Problem (Autofill the Table) *Let the table T_{in} to be a valid input table to the pipeline, i.e. Valid(T_{in}) = TRUE. The output table from the pipeline will only fill/modify the column c_i in the table, where Fill(c_i) = TRUE. The entries filled by the pipeline are generated based on the first few example entries in c_i. The pipeline needs to derive the appropriate formula from the example entries to calculate the remaining entries in c_i.*

Note that in the following discussions in the paper, we assume the column that need to be autofilled is the rightmost column.

4 APPROACH

4.1 Pipeline Overview

We constructs a pipeline to structurally find the appropriate formula for the table. For each step in the pipeline, we put the simple operations before the more complex operations so that the pipeline can spend much less time to find simple operations and will take negligible more time to find the complex operations, compared to the pipeline that puts complex operations before the simple ones.

The first step of our pipeline is to perform some basic cleanings on the input data, although we assume the data is relatively clean. If the input data columns are labeled with contextually relevant titles, the pipeline can fast forward to the Domain Knowledge checkpoint to see if any formulas can be employed.

The proposed pipeline in Figure 2 then considers four different scenarios in order, and detailed formal definitions and implementations of each component of the pipeline will be provided in the following.

(I) Numeric Analysis in section 4.3
(II) String Operations in section 4.4
(III) Mixture of Numeric and String Columns in section 4.5
(IV) Entries with Mixture of Numbers and Letters in section 4.6
(V) Domain Knowledge in section 4.7

4.2 User Interaction Design

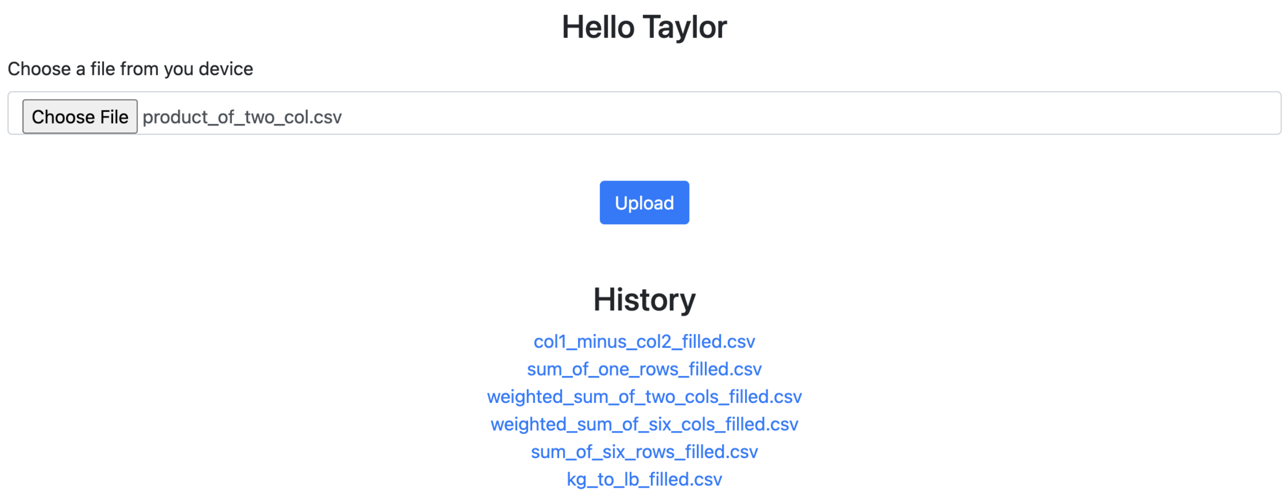
The proposed pipeline is equipped with a user interface as shown in the Figure 3 where users can upload and download the datasets that users wish to fill. Both the frontend layout and the backend data management(SQL) are facilitated by Flask and related packages. For better user experiences, this web application allows users to create independent accounts and keeps track of past datasets that have been uploaded and filled. For simplicity, the only required input is the dataset that the user wishes to fill in. In the case where no valid functions are found, a warning message will flash out on the screen saying there are no valid functions found for the task.

4.3 Numeric Analysis

Definition 6 (Numeric Table). *A numeric table T_N is defined as all the entries in T_N are of the numeric types. In addition, the relationships among entries in T_N are arithmetic.*

Numeric types includes integer, float, and so on. The entries in the numeric table can have different numeric types, such as some entries are of type integer and the others are of type float. It is also acceptable to have some entries containing the string representations of numbers.

Arithmetic relationships refer to the combination of basic operations. Basic arithmetic operations are introduced in Section 4.3.2.



Hello Taylor

Choose a file from you device

History

col1_minus_col2_filled.csv
sum_of_one_rows_filled.csv
weighted_sum_of_two_cols_filled.csv
weighted_sum_of_six_cols_filled.csv
sum_of_six_rows_filled.csv
kg_to_lb_filled.csv

Figure 3: User Interface

Note that sometimes, although the entries of the table are all of numeric type, it may not necessarily be a numeric table. For instance, if the table contains a column of phone numbers and the second column extracts the first three digits of the phone number in the first column, all the entries in the table are of numeric type, but the relationship among entries are extracting one portion of the string, which does not count toward the arithmetic operations. Hence, this table is not a numeric table.

Proposition 2 . $T_{N,in} \implies T_{N,out}$.

The table contains a few completed columns and a few example entries for the filling column. When the input table to the pipeline is numeric, it means the entries in the completed columns are numeric and the example entries in the last column are also numeric. The examples in the last column have strong indications of the remaining entries need to be autofilled. Hence, it is safe to derive that the remaining entries to be autofilled will also be numeric, leading to the output table to be numeric.

4.3.1 Element-Wise Single Column Operation. This can be considered as a simplified version of unit conversion operation, where all the entries in the completed column multiply by a constant to get the entries in the filling column. This implies that all the numbers in the completed columns have the same unit. If there are multiple possible units in the completed column, indicated by the letters after the numbers in each entry, this case will be handled in the section 4.6.

Definition 7 (Element-Wise Single Column Formula). *T_{in} only has one completed column and one filling column. If all the example entries in the filling column can be derived by multiplying one constant to the corresponding entries in the completed column, the table satisfies the element-wise single column formula.*

4.3.2 Basic Arithmetic Operation. The arithmetic operations are basic, so we that we don't add any weights over columns when performing the operations. This means that we will only consider the case such as (entries in col_1 + entries in col_2), and won't consider the case such as (2× entries in col_1 + 3× entries in col_2). These weighted cases will be handled in section 4.3.5.

Also note that the basic arithmetic operation will only look at one row at each time, which means the entries in row i in the last column will only depend on other entries in row i. Multiple rows operations will be handled in section 4.3.3 and section 4.3.4.

Definition 8 (Basic Arithmetic Formula). *For each row that contains the example entry in the filling column, if the value can be derived by performing simple addition, subtraction, multiplication, division, max, min, and averaging, the table satisfies the basic arithmetic formula.*

4.3.3 Multi-row Operation. The pipeline now looks at multiple rows to calculate the entries in the last column. For each entries in the last column, in order to reduce the running time, we set the constraint that we will look at at most 10 rows below the entries row. If there aren't enough rows below the entries, we look at as many rows as possible.

Definition 9 (Multi-row Operation). *For each row that contains the example entry in the filling column, if the value at row i can be derived by performing simple operations over the consecutive entries in the completed columns that are at most 10 consecutive rows below row i, the table satisfies the multi-row formula.*

Note that if the formula needs to look at the first and third rows below the entry, it doesn't look at the consecutive rows. Hence, the formula is not a multi-row formula.

4.3.4 Filter Operation. For the table that satisfies the filter formula, the completed columns of the table can be separated into two categories, filter column and value column. Each completed column belongs to one and only one category.

Definition 10 (Filter Formula). *For each row i that contains the example entry in the filling column, the pipeline looks at the rows*

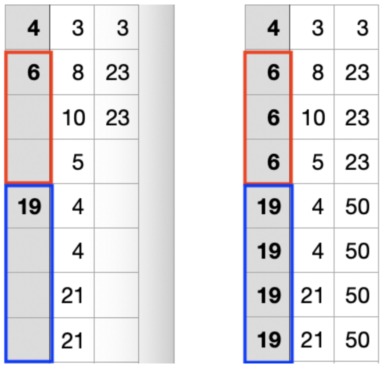
Algorithm 1 Autofill Pipeline

```
Require: $T_{in}$                                              ▷ $|T_{in}| \geq 2$
  if $T_{in}$ is well labeled then
      if Labels and input examples match with some function $f$ in Domain Knowledge then
          return $T_{out}$ autofilled using $f$
      end if
  end if

  if All columns in $T_{in}$ can be converted to numeric values then
      if Input examples match with some numeric operations $f$ then
          return $T_{out}$ autofilled using $f$
      end if
  end if

  Convert all values to strings
  if Input examples match with some string operations $f$ then
          return $T_{out}$ autofilled using $f$
  end if

  if There are both numeric and string columns in $T_{in}$ then
      if Numeric columns match with some numeric operations $f$ based on the string column then
          return $T_{out}$ autofilled using $f$
      end if
  end if

  if Input examples contain numeric values tagged with units then
      Separate the numeric value and the units
      Find unit clusters by calculating Jaccard Similarity
      Perform unit conversion
      if Input examples match with some numeric value operation or some formula in the domain knowledge $f$ then
          return $T_{out}$ autofilled using $f$
      end if
  end if

  return warning message
```

above and below i, if exists. If all the values in the filter columns of the row are the same as those at row i, it continues looking. If any value in the filter column is different, it stops looking. The pipeline then do simple operations on the values in value columns of all the rows it just looked at and have all the same filter column values as those at row i. If the pipeline can find the simple operation such that the result matches the values of example entries, the table satisfies the filter formula.

If the input table satisfies the filter formula, it can have more empty entries in the complete columns than those tables that satisfies other formula.

In Figure 4, the left-hand side is a portion of input table, and the right-hand side is the corresponding portion of output table. The first column is a filter column, the second column is a value column, and the last/rightmost column is the filling column. The pipeline needs to recognize that the empty entries in the completed columns should have the value of the first non-empty entry above it.



4	3	3
6	8	23
	10	23
	5	
19	4	
	4	
	21	
	21	

4	3	3
6	8	23
6	10	23
6	5	23
19	4	50
19	4	50
19	21	50
19	21	50

Figure 4: Empty Entries in Filter Formula Input Table

4.3.5 Weighted Sum Formula. Weighted sum over columns is also a common practice in table. In order to have deterministic weights for n columns, the pipeline needs at least n examples in the filling columns.

Definition 11 (Weighted Sum Formula). *For the table with n completed columns, there exists $w_1, ..., w_n$, where for all $i \in \{1, ..., n\}$, there are at least n example entries in the filling column, and for every example queries v_j in the filling column at row j, $w_1 * c_1 + ... w_n * c_n = v_j$ where c_i is the entry at row j and col i.*

4.4 String Operations

Definition 12 (String Table). *A string table T_S is defined as all the entries in T_S are of String type. In addition, the relationships among entries in T_N are string operations.*

String operations include extractions, concatenation, and refactoring. More details can be found at section 4.4.1, section 4.4.2, and section 4.4.3.

Proposition 3 . $T_{S,in} \implies T_{S,out}$.

Similar argument from Proposition 2 can be applied to the String tables. If the input table to the pipeline is string table, the output table from the pipeline should also be string table.

4.4.1 String Extraction Operation. We assume that there is only one way to perform the extraction on the remaining queries in the filling column based on the example queries. For instance, suppose the table contains one complete column and one filling column. If all the example queries extract the second substring from the corresponding string of the format "XXX, XXX, XXX" in the completed column, it then will have ambiguity if there exists an entry in the remaining row that of the format "XXX, XXX, XXX, XXX", since the pipeline may extract the second or the third substring based on different interpretation. Hence, this table is not accepted, or satisfied, by the string extraction formula.

Definition 13 (String Extraction Formula). *The table has one completed column and one filling column. For all the example entries, if all of them are the substring of the corresponding entries in the completed column, and the characters immediately before and after, if exists, the substring in the whole string are the same, and the characters also* *appear in the remaining entries in the complete column, or all entries in the completed column have the certain pattern and all the example entries appear in the certain position in the corresponding strings' pattern in the completed column, then the table satisfies the string extraction formula.*

4.4.2 String Concatenation Operation. We assume the substring to concatenate is consistent throughout the table. For instance, if the first row concatenates "@cornell.edu" to the entry in the completed row, and the second row concatenates "@gmail.com" to the corresponding entry, the table doesn't satisfy the string concatenation formula, since for the remaining rows, it creates ambuiguity about which substring to concatenate.

Definition 14 (String Concatenation Formula). *The table has one completed column and one filling column. For all the example entries, it contains the string in the corresponding completed column, and concatenate an additional substring to that string. If the additional substring is the same for all the example entries, or the substring is already contained in the example entries, the table satisfies the string concatenation formula.*

Note that it is possible for the entries in the completed column already contain the substring that need to be concatenated, then we keep the original string and not concatenating the substring to the original string.

4.4.3 String Refactoring Operation. We assume all the entries in the completed column can be partitioned into fixed number of segments. If the table contains an entry of pattern "XXX,XXX,XXX" (3 segments) and an entry of pattern "XXX,XXX,XXX,XXX" (4 segments), the table is not satisfied by the string refactoring formula.

Definition 15 (String Refactoring Formula). *The table has one completed column and one filling column. If all the non-empty entries can be partitioned into n number of segments. For all the example entries, if each segment matches the corresponding segment of the entry in the completed column, and the delimiters separating the segments in the example entries are consistent, the table satisfies the string refactoring formula.*

Note that the delimiters separating the segments in one example need not to be the same. For instance, if all the example entries are of the format "XXX, XXX(XXX)". Although the delimiter separating the segments in the entries are different (", " and "(" and ")"). As long as the delimiters are consistent in all the example entries, the table satisfied the string refactoring formula.

4.5 Mixed Column Types

If the table contains one string completed column, a few numeric completed columns, and a numeric filling column, the table can be considered in this pipeline step. Otherwise, the table will be moved to the next step of the pipeline.

This is a special case of the filter operation discussed in the section 4.3.4, where the filter column is string, not numeric.

Definition 16 (Mixed Column Types Formula) *The table has the completed string columns as filter columns, the completed numeric columns as value column, and one filling numeric column. For each row i that contains the example entry in the filling column, the pipeline looks at the rows above and below i, if exists. If all the strings in the filter columns of the row are the same as those at row i, it continues looking. If any string in the filter column is different, it stops looking. The pipeline then do simple operations on the values in numeric value columns of all the rows it just looked at and have all the same filter column strings as those at row i. If the pipeline can find the simple operation such that the result matches the values of example entries, the table satisfies the filter formula.*

4.6 Entries with Numbers and Letters

This is a complicated version of the section 4.3.1. The table contains a completed column and a filling column. For each entry in the completed column, it has some numbers following by string representation of units.

The pipeline first separates the numeric values from their units. The pipeline then tries to cluster the rows based on their units. It uses Jaccard similarity to cluster the rows. We employ Jaccard similarity, instead of directly comparing the string units, is because there are many possible string representations of the same unit. For example, kilograms can be represented as kg, K.G., kilogram or something else. We think Jaccard similarity can better capture the same units with more flexibility.

After clustering, the pipeline then tries to find different unit conversion formulas on the numbers for each cluster by putting the completed and filling column of numbers into the step described in the section 4.3.1. If the pipeline successfully finds the formula, it then applies the formula to the remaining entries in the same cluster in the filling column.

4.7 Domain Knowledge

Domain knowledge generally refers to the specific context needed in performing specific tasks. For instance, the domain knowledge needed in date formatting includes various representations of day, month, and year and punctuations used to separate those representations. However, in this paper, we define *Domain Knowledge* as the set of candidate functions related to specific topics such as personal health indices and financial statistics.

As mentioned in the section 2.1, the paper [2] introduces the importance of domain knowledge in autofilling the table, and we think it is critical for the pipeline to also have the last step of domain knowledge in order to capture more sophisticated calculations on columns, especially numeric columns.

Since this paper introduces the basic idea of how the pipeline should be, and it is impossible for us to fully implement all the domain knowledge for even one domain, we selected a small portion of formula for two domains (personal health and finance) to demonstrate our idea. The user can expand our domain and add more domain-specific formula as needed.

Note that for each formula, it is critical to consider all possible positions of the columns and possible combinations of different units. For instance, when calculating Body Mass Index (BMI), we need the weight and height. It is possible to have the weight column before the height column, or the height column before the weight column. The standard BMI formula uses standard units, i.e. weights in kilogram and heights in meters. However, it is possible for the dataset to use pounds as unit for weight and maybe inches or centimeters for heights. Hence, the BMI formula may imply the several formulas need to consider given different column positions and units. However, it should be very fast to test each formula, since if the formula doesn't work for the table, it is highly likely to be excluded in the first row.

5 EXPERIMENT

5.1 Data Description

For numeric tables, we use the random small number generation to generate the entries in the completed columns and perform the numeric operations on the completed columns to get the example entries in the last column. For the tables containing city, state, and zip code, we extract a subsection of data from this website here. For the tables containing the street addresses, we extract a subsection of data from another website here For the string tables, they are generated separately depend on the domain. For email address, we extract a random subsection of Cornell students' netid. For string extraction, we extract a random subsection of strings from the datasets from other classes.

One of our csv file here is actually real-world data on people's biometry from our friend's laboratory, and we use the data for our domain knowledge on calculating Body Mass Index (BMI) and Basal Metabolic Rate (BMR). Please note that the data should not be publicized.

5.2 Ground Truth and Evaluation Metrics

The pipeline outputs a result table, and we compare it with the expected table. Since there might be some decimals in the filling column, we have the absolute tolerance of 0.01 for the comparison.

5.3 Execution Time Analysis

In the Figure 5, the pipeline doesn't take significantly more time to find the formula as the number of rows it needs to consider for each entry in the filling column increases a little bit. It is because the execution time to look at different number of rows is very small, since the number of rows only ranges from 1 to 10 in the figure. When the number of row goes to a large number, the execution time will gradually increases.

On the other hand, as shown in the Figure 6, as the number of columns the dataset increases a little bit, the execution time of the pipeline increases significantly. This is because as the number of column increases, the number of example entries in the filling column also increases in order to have deterministic weights for the weighted sum formula. Hence, the execution time increases significantly, even though the number of columns increases a little bit.

6 RESULTS

We have examined the code accuracy by conducting tests on limited-scale data intentionally created to assess various facets of our implementation. The code consistently produced accurate results when provided with inputs intentionally generated. However, it is imperative to acknowledge that the code's resilience is not currently sufficient for practical application with real world datasets.

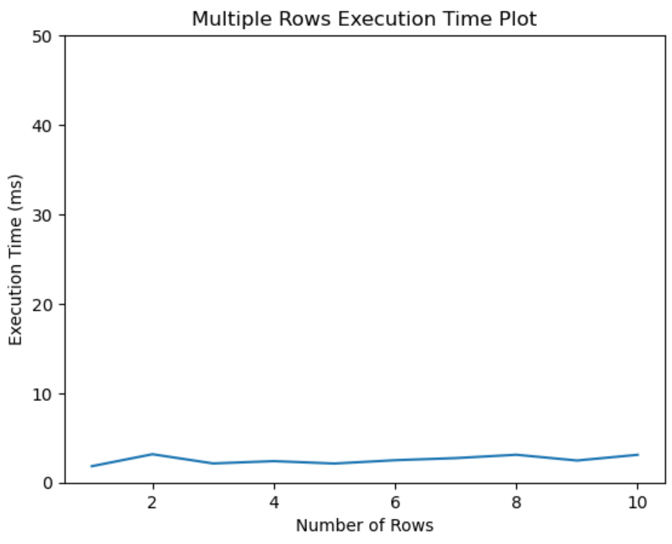


Figure 5: Number of Rows vs. Execution Time in the Multiple Rows Operation

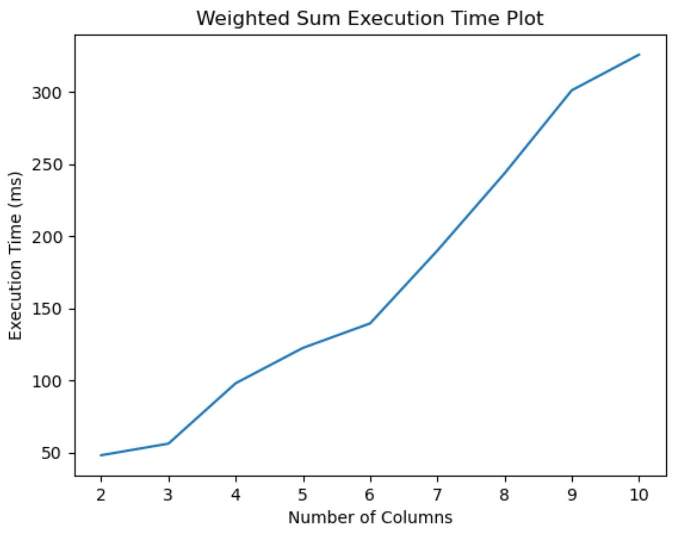


Figure 6: Number of Columns vs. Execution Time in the Weighted Sum Operation

7 DISCUSSION AND FUTURE IMPROVEMENTS

7.1 Limitations

One major concern of the project is the conflict between the generality of users' data manipulation objectives and the specificity required in the coding implementation. Such conflict diminishes the objective of our project. To be more specific, without any user input or hint on their true intention, our proposed solution has to potentially check all the possibilities, which would be computationally expensive or infeasible to implement. That is probably the reason why studies along this direction would focus on a very small problem space (e.g. formatting date and name). Without narrowing down the problem space, the two resulting difficulties are: first, it requires much more time than given to exhaust the commonly desired data manipulations if ever possible; second, we are constantly forced to make strong assumptions to be able to implement effective and efficient autofill functions. For instance, for "refactoring" as defined above, where we want to make the non-punctuation content be wrapped by desired punctuation, we assume that there is the same number of segments in each string of the complete column. (E.g. there are three segments in "(123)-456=789". To argue for our implementation, we would have to prove such an assumption is valid by surveying more real-world data sets. Another solution to mitigate the challenge we are facing is to ask users for more specific inputs to obtain more information about their intention and specific domain knowledge, which does not innovate much from the current state of work.

We are also in doubt about the utilization of math modeling techniques and neural networks since there are only a few required user input examples. It is not feasible to acquire data sets that can potentially be used to train a neural network within a month. In the end, our group is uncertain about the value of our attempt to design a highly automated system to autofill columns for users.

7.2 Algorithm Imrpovements

The pipeline proposed in the paper has a highly linear structure and strict separation of listed scenarios, while the input data files can be a combination of different scenarios. For instance, the proposed algorithm and pipeline lack a cyclic structure between mixed columnn(separated string and numeric value) and mix entry(string containing numeric value an string treated as numeric value with units) since mix entry is technically also a numeric value. Disabling the combination of different scenarios would again limit the types of input can be dealt with using our proposed solution.

Another consequence of the linear structure of the pipeline is the deficient runtime efficiency. As mentioned in the experiment result analysis, the runtime would grow linearly as the number of input columns increase. Additionally, as the valid function comes later in the candidate functions, the runtime would also increase linearly.

7.3 User Interaction

Although one of the subgoals of this project is to achieve minimal user input needed for the automatic calculation, more user inputs prompted in iteration may help with the performance of our approach. For example, in the case where there are multiple valid functions, our approach is deterministic by default since it always returns the value generated by the first valid function. It would be better if users could indicate whether the returned output is correct so that the pipeline can return the values generated by the next valid function.

One assumption made in the paper is that all columns in the input are relevant in generating the value of interest. Nevertheless, this is not usually the case. More often, data files contain columns irrelevant to the calculation of another specific value. It takes some effort for users to carve out columns to be able to upload files. Therefore, another improvement in user interaction is to enable the manual selection of useful columns so that the data separation can be done internally in the pipeline.

8 GIT REPO

You can access our code through this link. Please reach out to Taylor and Arya if you have difficulty accessing it.

9 WORK DISTRIBUTION

In general, Taylor and Arya have made equal contributions to the project, although there was a slight split in their areas of focus. In the context of the final report, Arya primarily concentrated on the formal problem definition and experiment session of the paper which had been previously discussed and agreed upon. Both Arya and Taylor collaborated on the initial experiment description, while Taylor was responsible for completing the remaining sections of the report. In terms of the project itself, Arya took the lead in data generation, while Taylor was more involved in initializing the code structures. Regarding the implementation, both team members have put in an equal amount of effort. More specifically, Arya was more responsible for weighted sums, unit conversions, mixed column (filtering) operation, and domain knowledge implementation, while Taylor was more responsible for implementing the user interface, simple algebraic operations, string formatting, and missed entry analysis.

REFERENCES

[1] Assadi, A., Milo, T., Novgorodov, S.: Cleaning data with constraints and experts. In: Proceedings of the 21st International Workshop on the Web and Databases. WebDB'18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3201463.3201464, https://doi.org/10.1145/3201463.3201464

[2] Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Communications of the ACM **55**, 97–105 (January 2012), https://www.microsoft.com/en-us/research/publication/spreadsheet-data-manipulation-using-examples/, invited to CACM Research Highlights

[3] Kandel, S., Paepcke, A., Hellerstein, J., Heer, J.: Wrangler: Interactive visual specification of data transformation scripts. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 3363–3372. CHI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1978942.1979444, https://doi.org/10.1145/1978942.1979444

[4] Le, V., Gulwani, S.: Flashextract: A framework for data extraction by examples. SIGPLAN Not. **49**(6), 542–553 (jun 2014). https://doi.org/10.1145/2666356.2594333, https://doi.org/10.1145/2666356.2594333

[5] Liu, B., Jagadish, H.V.: A spreadsheet algebra for a direct data manipulation query interface. In: 2009 IEEE 25th International Conference on Data Engineering. pp. 417–428 (2009). https://doi.org/10.1109/ICDE.2009.34

[6] Singh, R., Gulwani, S.: Transforming spreadsheet data types using examples. SIGPLAN Not. **51**(1), 343–356 (jan 2016). https://doi.org/10.1145/2914770.2837668, https://doi.org/10.1145/2914770.2837668