Aggregation Detection in CSV Files

Lan Jiang, Gerardo Vitagliano, Mazhar Hameed, Felix Naumann
Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

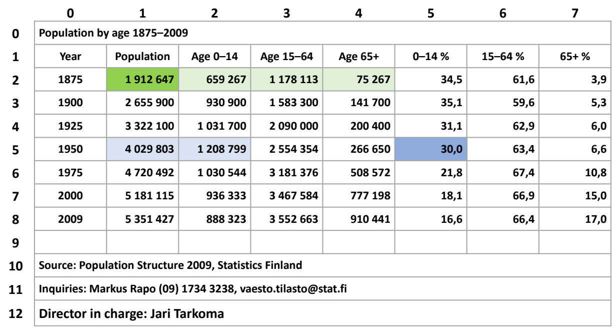
Aggregations are an arithmetic relationship between a single number and a set of numbers. Tables in raw CSV files often include various types of aggregations to summarize data therein. Identifying aggregations in tables can help understand file structures, detect data errors, and normalize tables. However, recognizing aggregations in CSV files is not trivial, as these files often organize information in an ad-hoc manner with aggregations appearing in arbitrary positions and displaying rounding errors.

We propose the three-stage approach AggreCol to recognize aggregations of five types: sum, difference, average, division, and relative change. The first stage detects aggregations of each type individually. The second stage uses a set of pruning rules to remove spurious candidates. The last stage employs rules to allow individual detectors to skip specific parts of the file and retrieve more aggregations. We evaluated our approach with two manually annotated datasets, showing that AggreCol is capable of achieving 0.95 precision and recall for 91.1% and 86.3% of the files, respectively. We obtained similar results on an unseen test dataset, proving the generalizability of our proposed techniques.

1 AGGREGATION DETECTION

An *aggregation* represents an arithmetic relationship between a set of numbers and a single number: the latter (aggregate) can be determined by applying an aggregation function, such as sum or average, on the former (range). As a useful tool to summarize data, aggregations are implemented as operators in many data analytics tools. For example, relational database systems support a collection of aggregation operators based on the underlying relational algebra, such as sum, average, count, and minimum/maximum, to enable data analysis on relational tables. Spreadsheet programs provide an extensive set of formulas to compute aggregations on values of arbitrary cells [25]. Various business intelligence and data analytics platforms, such as Tableau and Trifacta, integrate aggregation calculation into their toolkits. Data scientists and practitioners frequently build pipelines to prepare data or carry out data analytic tasks with the above tools. While doing so, they may obtain insights on how to process data in the next step by looking up various kinds of aggregations amongst numbers.

Numeric data and aggregations calculated from them often serve as intermediate or final results of data analytics pipelines. While some of these results are stored in structured formats, such as relational tables or key-value pairs, many of them are simply saved in data files, such as spreadsheets or plain-text files, where authors can organize content in arbitrary positions. Data files are useful to store data meant for human readability, such as reports of company balance sheets, or student rosters of courses. In this work, we deal with aggregations in a particular type of data files: verbose CSV files [20], whose cell values represent



	0	1	2	3	4	5	6	7
0	Population by age 1875–2009							
1	Year	Population	Age 0–14	Age 15–64	Age 65+	0–14 %	15–64 %	65+ %
2	1875	1 912 647	659 267	1 178 113	75 267	34,5	61,6	3,9
3	1900	2 655 900	930 900	1 583 300	141 700	35,1	59,6	5,3
4	1925	3 322 100	1 031 700	2 090 000	200 400	31,1	62,9	6,0
5	1950	4 029 803	1 208 799	2 554 354	266 650	30,0	63,4	6,6
6	1975	4 720 492	1 030 544	3 181 376	508 572	21,8	67,4	10,8
7	2000	5 181 115	936 333	3 467 584	777 198	18,1	66,9	15,0
8	2009	5 351 427	888 323	3 552 663	910 441	16,6	66,4	17,0
9								
10	Source: Population Structure 2009, Statistics Finland							
11	Inquiries: Markus Rapo (09) 1734 3238, vaesto.tilasto@stat.fi							
12	Director in charge: Jari Tarkoma							

Figure 1: A real-world verbose CSV file that contains two types of aggregations: sum (green) and division (blue).

content of various roles, such as data, header, aggregation, or metadata. A table in a verbose CSV file is a set of data, header, or aggregation cells in any configuration. One table may comprise multiple left and top headers, and contain aggregation rows or columns, while another may simply resemble a relational table. Figure 1 illustrates a real-world verbose CSV file. Besides the title and footnotes at the top and bottom left, the file contains a table that clarifies the overall and per age-group population for several years. The three right-most columns account for the proportion of per-group population against the total.

According to a survey about open portal CSV files, around 9.2% of 107,779 tables in these files have multi-row headers or are correlated to at least one comment line (title, footnote, etc.) in the file [23]. Verbose CSV files include both above cases. Furthermore, our previous work showed that around 5.9% of 1.3M cells in a verbose CSV file dataset are aggregate cells [20].

Locations of aggregations in a file are not always obvious. One reason is the lack of proper metadata to describe them. Because verbose CSV files cannot preserve any metadata, an aggregate appears to be just a normal numeric cell similar to any other numeric data cell in a table. As some verbose CSV files are exported from spreadsheet files, one might refer to the original spreadsheets, if available, which might contain the actual aggregation function as metadata. However, these metadata are not always present, even in spreadsheets, as users copy and paste only the values derived by a formula. Based on our observation on the Troy dataset, 150 of 200 spreadsheet files have at least one aggregation, while none of them include the corresponding formulas.

With the lack of aggregation metadata in verbose CSV files, discovering them is beneficial for several downstream applications:

Enriching files with metadata. Metadata, despite being useful information to help understand data, cannot be embedded into the vanilla CSV file format that is commonly used to store data. Specifically, locations of aggregation cells can reveal the arithmetic relationships between numeric cells. In addition, detected aggregations can be used to improve cell classification algorithms that usually treat "aggregation" as a cell type [15, 20, 21]. In Section 4.6, we show the improvement gain of a state-of-the-art cell classification approach by using the results of our proposed aggregation detection algorithm to fill a binary feature that represents if a cell is an aggregator or not.

Numeric error detection and cleaning. Aggregations in verbose CSV files often include errors: number in the aggregate cell does not precisely calculate the numbers in the range cells. A major reason is that numbers are rounded to preserve a certain amount of decimal digits. The calculated aggregation of a group of rounded numbers may deviate from the intended aggregate value – an effect we have observed in around 29% of the real-world aggregations, as described in Section 4.1. With our advanced approach to detect aggregations with an error, data scientists may realize the numeric data errors and mend them accordingly.

Serving as input for formula smell detection. Many verbose CSV files are exported from spreadsheets where the aggregations have been created via formulas. However, manual mis-operations may cause absence or incorrectness of formulas in spreadsheets. For example, in a row of numeric cells that represent the per-column sum of numbers in other rows, the formula of one cell is missing, which can be identified by various "formula smell" detection approaches [9, 19]. However, these approaches all assume the existence of some surrounding formulas, e.g., the other cells in the row with the sum formula in the above case. Detected aggregations can serve as the input for those formula smell detection approaches to recognize more smelly formulas.

With the usefulness of knowledge about aggregations and the lack of such information in verbose CSV files, being able to detect them is important. Manual work is infeasible, as locations of aggregations are not always apparent: any numeric cell could aggregate any set of other cells. When dealing with a large table, manual checking is error-prone and time-consuming due to many aggregation candidates. The existence of specific keywords in a cell, such as 'total' and 'average', might indicate the presence of aggregates in the same row or column. However, such an approach is unreliable, as keyword dictionaries cannot cover all aggregations. In the table of Figure 1, for example, none of the aggregation cells can be identified by their headers. An investigation into our dataset shows that we could retrieve only around 60.0% of the true sum aggregates by using a set of keywords including 'total', 'all', 'sum', 'subtotal', and 'overall'. In Section 4.4, we elaborate on the results of this keyword-based approach.

In light of these challenges, automated approaches are needed. We propose AggreCol– a three-stage approach to automatically detect various types of aggregations in verbose CSV files. Our approach supports sum, difference, average, division, and relative change, formally defined in Section 2.1. In the first stage, AggreCol detects adjacent aggregations of each aggregation function separately. An aggregation is adjacent if the set of cells in its range are adjacent to its aggregate. The second stage collects the individual detection results and removes the spurious aggregations with a set of pruning rules. In the last stage, our approach aims at recognizing non-adjacent aggregations by skipping the aggregates of detected aggregations. In particular, we make the following contributions:

(i) We formalize the aggregation detection problem for verbose CSV files, and propose the three-stage approach AggreCol to address it for five aggregation functions.
(ii) We annotated two datasets that comprise 466 verbose CSV files of various domains, published on our website[1].
(iii) We conduct a series of comparative experiments to evaluate the effectiveness of AggreCol.

In Section 2, we formalize the definitions of the used terms, and possible aggregation patterns, followed by the formal problem statement. Section 3 describes the technical details of our approach. The experimental results along with a detection error analysis are presented in Section 4. Section 5 briefly discusses the related work. As a conclusion, we summarize this work and envision future work in Section 6.

2 PRELIMINARIES

We first provide definitions of all necessary concepts, including tables in verbose CSV files, the components of an aggregation – aggregate, range, and aggregation function – and the error level. In the next part, we summarize the three observed aggregation patterns and give the formal problem statement.

2.1 Definitions

We adopt the notion of verbose CSV files from Jiang et al. [20]. A verbose CSV file is a two-dimensional plain-text file in which content is organized in cells separated by delimiters. As a generalized version of standard CSV files defined by RFC 4180[2], verbose CSV files do not require a single relational table with the defined schema as content. Instead, every cell may represent an arbitrary type of information, such as data, title, header, aggregate, metadata of a table, or the empty value.

To interpret a file as a verbose CSV file, the corresponding *file dialect* must be recognized first [8, 14, 27]. A file dialect incorporates all utility characters used to interpret its structure, such as field delimiter and quotation character. While comma and double-quote characters are widely used, no mandatory requirements are enforced for their selection. Here, we assume verbose CSV file dialects have been correctly detected, and the input files are delimited accordingly. As verbose CSV files allow a loose layout, tables therein may have unique structures.

Definition 1 (Table). A table in a verbose CSV file is a group of cells. Each cell carries information, such as data, row or column header, group header, aggregate, or empty visual separator. A table contains structured information about entities of a common type.

Stemming from spreadsheets, tables in verbose CSV files often contain sets of numbers and their aggregates. The verbose CSV file in Figure 1 includes a table that spans rows 1 to 8, and columns 0 to 7. This table includes cells that serve as row header, column header, data, or aggregate. In tables of verbose CSV files, users often express the numeric value zero with an empty table cell. We follow this interpretation in this work.

An aggregation describes an arithmetic relationship between a numeric cell and a set of other numeric cells in the same table. We refer to the former numeric cell as the *aggregate* and the latter numeric cell set as the *range*. We call each component in an aggregatee an *range element*. A table may include multiple aggregations.

Definition 2 (Aggregate & Range). An *aggregate* $r = c_{i_r,j_r}$, where $i_r \in [0, M)$ and $j_r \in [0, N)$, is a numeric cell whose value

[1]https://hpi.de/naumann/projects/data-preparation/aggregation-detection-in-verbose-csv-files.html
[2]https://datatracker.ietf.org/doc/html/rfc4180

can be derived by applying a specific arithmetic function on a set of other numeric cells $E = \{c_{i,j} | i = 0, \ldots, M-1, j = 0, \ldots, N-1\}$, referred to as an *range*. We represent the list of row and column indices of the elements in E with i_E and j_E.

Definition 3 (Aggregation function)**.** An aggregation function f is an arithmetic operator that can be applied to the values in an range to determine the value of an aggregate.

While many different aggregation functions can be applied to numeric values, only few appear frequently in verbose CSV files based on our annotation of 385 files. Figure 2 illustrates the occurrence distribution of aggregation functions. In this work, we cover the aggregation functions that appear in more than 5% of the files, i.e., *sum*, *difference*, *average*, *division*, and *relative change*. Table 1 displays the specifications of these aggregation functions. For example, a sum function requires no less than one element in the aggregatee, and is a cumulative function, which means the aggregator derived by applying this function can be further used as aggregatee element in other aggregations. In contrast, average is non-cumulative: averaging numbers that are themselves averages is not arithmetically meaningful. Relative change describes the change from B to C normalized by B, which is commonly used to show changes across a certain time period.

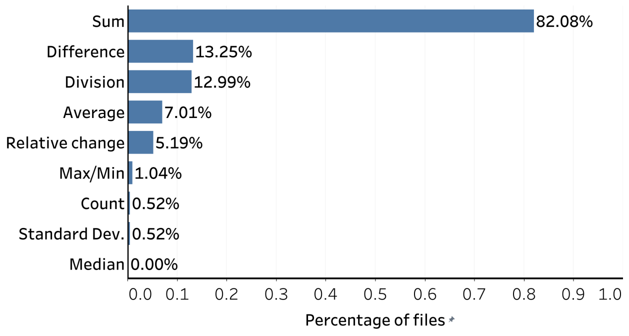


Figure 2: Percentage of 385 files that contain the individual aggregation functions.

We observe that a few real-world aggregations involve multiple aggregation functions. For example, the percentage of population holding at least a university degree is the sum of populations with bachelor, master, and doctor degrees divided by the total population. In this work, we treat only single-function aggregations. Having defined all components of an aggregation, we can formalize the definition of an aggregation as follows.

Definition 4 (Aggregation)**.** An aggregation a is a tuple ($r \leftarrow E$, f), where $r \notin E$ and the value of the cell r can be derived by applying the arithmetic function f on all values in E.

In principle, the value of r should equal precisely the value calculated by applying f on E. However in practice, this is often not true as numbers displayed in cells can be rounded. In fact, we observed that around 30% of all aggregations in our manually annotated files contain such errors. Numbers can be rounded to different significant figures, leading to a wide variety of error margins. As verbose CSV files do not preserve the metadata on how numbers were rounded, we model such a calculation deviation with an *error level* and detect aggregations for different levels.

Table 1: Overview of the supported aggregation functions.

Aggregation function	# Aggregatee elements	Formula	Cumulative
Sum	≥ 1	$A = \sum_{i=1}^{n} B_i$	Yes
Difference	$= 2$	$A = B - C$	Yes
Average	≥ 1	$A = (\sum_{i=1}^{n} B_i)/n$	No
Division	$= 2$	$A = B/C$	No
Relative change	$= 2$	$A = (C - B)/B$	No

Definition 5 (Error level)**.** Given an aggregation a ($r \leftarrow E$, f), where r is the observed aggregate value, and r' the value determined by applying f on E, the error level e of a describes the deviation factor from r to r': $e = |(r' - r)/r|$

We calculate the error level with the normalized absolute difference, because numeric values of aggregates vary in their order of magnitude. Based on this formula, the highest observed error for a true aggregation was 37.5%. The error level is undefined if r = 0. For the few such cases in our dataset, we calculate the error level as the absolute difference between r and r'. We extend the notation of an aggregation to include the error level: a ($r \leftarrow E$, f, e).

While in principle both r and E can be located at any arbitrary position in a verbose CSV file, in practice, most aggregations organize r and E in the same row (row-wise) or column (column-wise). We make this *same-line aggregation assumption*. Discovering non-same line aggregations can be an interesting extension, but is likely to be fraught with many false-positives.

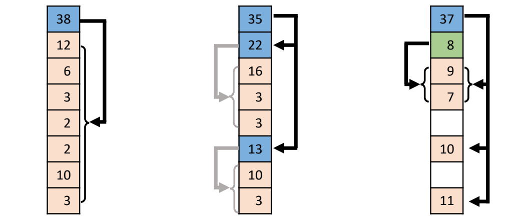
To express a row-wise aggregation, we use a slightly modified notation $(row{:}i, j_r \leftarrow j_E, f, e)$, where i is the row index of all cells; j_r and j_E are column indices of the aggregate and the range, respectively; f is the aggregation function; e is the observed error level. For example, the green-shaded sum aggregation in Figure 1 can be represented as $(row{:}2, 1 \leftarrow \{2, 3, 4\}, Sum, 0)$. We call $j_r \leftarrow j_E$ the *pattern* of the aggregation that indicates the scope of the aggregate and the range. Column-wise aggregations can be represented analogously.

2.2 Aggregation patterns

Given a row-wise or column-wise aggregation, range elements can be any subset of the numeric cells in the same column or same row. However, we observe that range elements are usually not randomly distributed. Authors of verbose CSV files tend to place aggregates close to their corresponding ranges. We summarize three aggregation patterns, shown in Figure 3, based on our examination of our dataset that includes 385 verbose CSV files.

The simplest and the most common pattern is *adjacent*, where the range is right next to the corresponding aggregate. Out of the examined verbose CSV files, 77.9% include adjacent aggregations. The high ratio is likely due to localized reading habits: humans tend to find the summary of a group of numbers in its direct proximity.

The second pattern is *cumulative*. As shown in Figure 3b, the value 35 is the sum of 22 and 13, which is not an adjacent aggregation. However, the latter two numbers are themselves aggregates of two different adjacent aggregations. For example, the global population equals to the sum of populations of northern and southern hemispheres, which, in turn, equals those of a number of countries therein. We have observed such a pattern in around 20% of the files. Note that to obtain a cumulative aggregation, the intermediate aggregates (22 and 13 in the figure) must be of



(a) Adjacent **(b) Cumulative** **(c) Interrupt**

Figure 3: Aggregator-aggregatee patterns of three different aggregations (Blue and green cells are sum and average of the corresponding orange cells).

cumulative nature: for the aggregations considered in this study, only sum and difference can behave cumulatively.

Lastly, 14.8% of our verbose CSV files contained *interrupt* aggregations. The range elements in an interrupt aggregation are scattered in the row or column. Interrupt aggregation happens when, for example, it is blocked by another non-cumulative aggregation (green shaded cell in Figure 3c is the average of the next two cells), an intermediate aggregation is actually not satisfied, or rows/columns are simply not organized in a localized manner.

We can now formally define the problem of *aggregation detection in verbose CSV files*: given a verbose CSV file $V = \{c_{i,j} | i = 0, \dots, M-1, j = 0, \dots, N-1\}$, a set of functions F, a given error level parameter e', find all aggregations in V using F presented in Table 1. Each aggregation satisfies the same-line aggregation assumption, and has the appropriate number of range elements, according to Table 1, i.e., each detected aggregation a ($r \leftarrow E$, f, e) must satisfy the following four requirements:

(i) $f \in F$;
(ii) $e \leq e'$;
(iii) either $\forall c_{i,j} \in E : i = i_r$ or $\forall c_{i,j} \in E, j = j_r$;
(iv) $|E| \geq 1$ if $f \in \{sum, average\}$, or
$|E| = 2$ if $f \in \{difference, division, relative\ change\}$.

3 THE AGGRECOL APPROACH

In this section, we present the methods used by our three-stage algorithm AggreCol to detect aggregations in verbose CSV files. We start by describing the general workflow of our approach, and then elaborate on the individual, collective, and supplemental aggregation detection stages in Sections 3.1-3.3.

Figure 4 illustrates the overview of AggreCol. Given a verbose CSV file, AggreCol first detects aggregations of different aggregation functions separately. Because some results of this step might be incidental, the results are then passed to the collective aggregation detection stage, where we remove spurious aggregations with a set of pruning rules. To specifically handle interrupt aggregations as defined in Section 2.2, we introduce a final stage that constructs new files from the original verbose CSV file, on which we re-apply the individual aggregation function detectors. The results of the third stage are presented as the final output.

We assume same-line aggregations, i.e., aggregate and range are either in the same row or the same column. We handle row-wise and column-wise aggregations equivalently. Without loss of generality, the examples and terminology used in the rest of this section are focused on row-wise aggregations.

3.1 Individual aggregation detection

AggreCol expects a verbose CSV file as input. Each file may use a unique number format. For example, one file may use dot as thousands separator and comma as decimal separator, while another may use space and comma for them, respectively. The interpretation of numbers may affect the results of aggregation detection. Therefore, identifying number format is a necessary prior step. Section 4.2 clarifies our method to recognize the number format and interpret the underlying numbers in the file.

AggreCol detects aggregations of each aggregation function separately. Algorithm 1 outlines the procedure of the individual aggregation detection algorithm. It takes as input a verbose CSV file, the function it detects aggregations of, the maximum tolerable error level, and the minimum coverage of aggregates in a row or column. To help understand the procedure, we use Figure 5 as a working example: it shows a table with four row-wise aggregations whose formulas are shown below the table. Numbers in the table are modified to fit the example. We set e as zero, and cov as 0.7. Details of the algorithm are elaborated below.

The individual aggregation detection algorithm starts by recognizing, within each row, the aggregations whose range is in proximity to its aggregate (lines 4-7). Depending on the mathematical properties of the aggregation functions summarized in Table 1, it employs either an adjacency list strategy, or a sliding window one. The two strategies are explained as follow.

Algorithm 1: Individual aggregation detection (row-wise)

```
Input: Verbose CSV file $V$, aggregation function $f$, error
       level $e$, line aggregation coverage threshold $cov$
Output: A set of row-wise aggregations D
1  $D \leftarrow \{\}$;
2  while $true$ do
3      $m \leftarrow length(V)$;
4      foreach $i$ in $\{0,\dots,m\}$ do
5          $D_m \leftarrow detect\_adjacent\_aggregations(V_m, e)$;
6          $D' \leftarrow D' \cup D_m$
7      end
8      $D' \leftarrow extend\_aggregations(D')$;
9      if $D' = \emptyset$ then
10         break;
11     $D' \leftarrow prune(D', cov)$;
12     $D \leftarrow D \cup D'$;
13     if $f$ is not cumulative then
14         break;
15     $V \leftarrow remove\_columns(V)$;
16 end
17 return $D$
```

Adjacency list strategy. This strategy recognizes an aggregation only if the set of all numeric elements in its range is adjacent to the aggregate, e.g., a_1 and a_2 in Figure 5. We employ this strategy for aggregation functions that are commutative. The commutative property guarantees that changing the order of the elements in an range has no impact on the aggregated result. For the functions supported in this work, sum and average hold such a property. The commutative property allows us to resort to a greedy approach: for each aggregate candidate $c_{i,j}$ and the numeric cells $E = \{c_{i,k} | k > j\}$ in its row, the approach iteratively moves the numeric cell in E that is column-wise closest

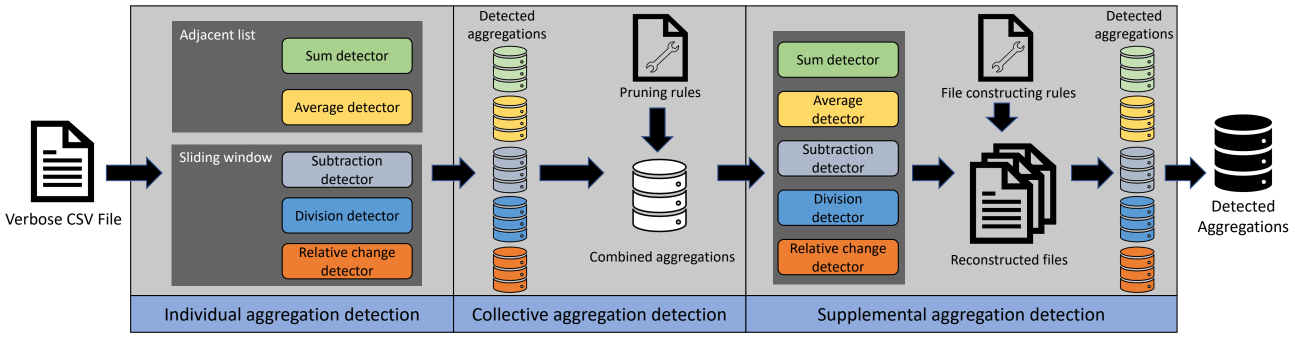
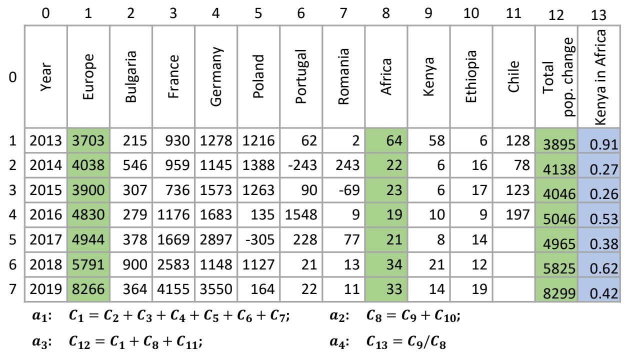


Figure 4: General workflow of our three-stage aggregation detection approach.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	Year	Europe	Bulgaria	France	Germany	Poland	Portugal	Romania	Africa	Kenya	Ethiopia	Chile	Total pop. change	Kenya in Africa
1	2013	3703	215	930	1278	1216	62	2	64	58	6	128	3895	0.91
2	2014	4038	546	959	1145	1388	-243	243	22	6	16	78	4138	0.27
3	2015	3900	307	736	1573	1263	90	-69	23	6	17	123	4046	0.26
4	2016	4830	279	1176	1683	135	1548	9	19	10	9	197	5046	0.53
5	2017	4944	378	1669	2897	-305	228	77	21	8	14		4965	0.38
6	2018	5791	900	2583	1148	1127	21	13	34	21	12		5825	0.62
7	2019	8266	364	4155	3550	164	22	11	33	14	19		8299	0.42

a_1: $C_1 = C_2 + C_3 + C_4 + C_5 + C_6 + C_7$; a_2: $C_8 = C_9 + C_{10}$;

a_3: $C_{12} = C_1 + C_8 + C_{11}$; a_4: $C_{13} = C_9/C_8$

Figure 5: Excerpt of a table that contains three sum (in green) and one division (in blue) aggregations. C_i represents the ith column.

to $c_{i,j}$ into an adjacency list L. After each move, it checks if the error level between $c_{i,j}$ and the value calculated by applying f on L is smaller than the given e. If such a list L can be found, $c_{i,j}$ and L compose a detected aggregation. The same process is applied also to the numeric cells $E = \{c_{i,k} | k < j\}$ to detect possible aggregations at the other side of the aggregate candidate. Although the range of a sum or average aggregation may have only one element, such single range element cases rarely appear in our datasets. Allowing detection of such aggregations would include massive false positives in the results, especially for files with many cells of identical numbers. Therefore, our approach requires at least two range elements for these two functions.

Sliding window strategy. For non-commutative aggregation functions, the order of range elements affects the result. As the greedy adjacency list strategy is not applicable, we use a window for each aggregate candidate. For each aggregate candidate $c_{i,j}$ the approach searches for possible aggregations by traversing all permutations of size n in the set of w numeric cells that are closest to $c_{i,j}$, where n is the number of possible range elements of the function specified in Table 1. Similar to the adjacency list strategy, this process is also applied to both sides of $c_{i,j}$ separately. We employ this strategy when detecting difference, division, and relative change aggregations.

Applying the greedy adjacency list strategy may yield incorrect aggregation candidates in particular situations. For example, $(row{:}2, 1 \leftarrow \{2, 3, 4, 5, 6, 7\}, Sum, 0)$ cannot be retrieved, as a previously detected aggregation with a shorter range $(row{:}2, 1 \leftarrow \{2, 3, 4, 5\}, Sum, 0)$ terminates the search for this aggregate candidate. However, the detected aggregation is in fact not a true one. It occurs because the sum of $c_{2,6}$ and $c_{2,7}$ happens to be zero. We assume that, although such patterns might appear in few rows due to a computational coincidence, they would not repeat across all rows in the table. To discover the above true aggregation and also drop the spurious one, AggreCol extends the detected aggregations by checking whether candidates with the same patterns as the detected aggregations across rows are also valid aggregations (line 8). In the case of Figure 5, as $(row{:}3, 1 \leftarrow \{2, 3, 4, 5, 6, 7\}, Sum, 0)$ is a valid aggregation, we check the candidates with the same pattern in the other rows, and validate the candidates for rows 2 and 5. These two candidates could not be retrieved, because an aggregation with a shorter range in the respective row was detected. However, $(row{:}1, 1 \leftarrow \{2, 3, 4, 5\}, Sum, 0)$ is not valid, even though it shares the same pattern as the detected $(row{:}2, 1 \leftarrow \{2, 3, 4, 5\}, Sum, 0)$. Table 2 lists the detected aggregations with no errors after the extension for the example in Figure 5.

Table 2: Detected row-wise sum aggregations after aggregation extension, grouped by column patterns (e = 0).

	Column pattern	Compliant rows
1	$1 \leftarrow \{2, 3, 4, 5, 6, 7\}$	1, 2, 3, 4, 5, 7
2	$4 \leftarrow \{5, 6\}$	1, 2, 3, 4, 6
3	$8 \leftarrow \{6, 7\}$	1, 2, 6, 7
4	$8 \leftarrow \{9, 10\}$	1, 2, 3, 5, 6, 7

Note that some rows are not compliant to specific column patterns. For example, $(row{:}6, 1 \leftarrow \{2, 3, 4, 5, 6, 7\}, Sum, 0)$ cannot be detected as the numbers in columns 2-7 in row 6 add up to 5792, deviating slightly from the aggregate value “5791” in column 1.

If there are no detected aggregations in any row, the entire process is terminated (line 10). Otherwise, the algorithm removes spurious detected aggregations with a pruning step (line 11).

Before performing the pruning step, all aggregation candidates are first grouped by their patterns. Any group possessing an insufficient number of aggregations is discarded. The sufficiency score is calculated as *the number of candidates in the group* normalized by *the number of numeric cells in the column with the aggregate cell.* Additionally, for all groups sharing the same aggregate, only the one with the highest sufficiency score is preserved. The same process is conducted also for the groups sharing the same range. The surviving groups are organized in a ranked list, where a group with a higher rank has (i) the more detected aggregations; (ii) the smaller average actual error level of the aggregations. After that, AggreCol iterates over the list and prunes the lower-ranked groups whose patterns cannot co-exist with the pattern of the currently inspected group, based on the following three heuristics:

Directional disagreement. Two aggregation candidates may share the same aggregate, yet develop their ranges in the two different directions, e.g., $(row{:}3,4 \leftarrow \{5,6,7\}, Sum, 0)$ and $(row{:}3,4 \leftarrow \{2,3\}, Sum, 0)$. For the same-function aggregation candidates sharing the same aggregate, we allow their ranges to reside only at the same side of the aggregate, reflecting the typical structure of spreadsheets.

Complete inclusion. For two aggregation candidates, a complete inclusion scenario appears if both the aggregate and part of the range elements of one aggregation are contained in the range of the other aggregation. For example, $(row{:}1,4 \leftarrow \{5,6\}, Sum, 0)$ and $(row{:}1,3 \leftarrow \{4,5,6,7\}, Sum, 0)$ form a complete inclusion, because the aggregate and range elements of the former are included in the range of the latter. As all elements in an range should represent entities of the same semantic level, e.g., the average sales of all departments, or the population of all states, any one of them cannot serve as an aggregate of any of its fellows.

Mutual inclusion. Two aggregation candidates are mutually included when the aggregate of each is included in the range of the other. For example, $(row{:}1,4 \leftarrow \{5,6\}, Sum, 0)$ and $(row{:}1,5 \leftarrow \{3,4\}, Sum, 0)$ are mutually inclusive. If one of them, say the former, is true aggregation, then $c_{1,5}$ is part of the range elements that add up to $c_{1,4}$. Then, $c_{1,5}$ should not be the aggregate that sums up numbers including $c_{1,4}$. Two mutually included candidates cause circular calculations, which should not be correct semantically.

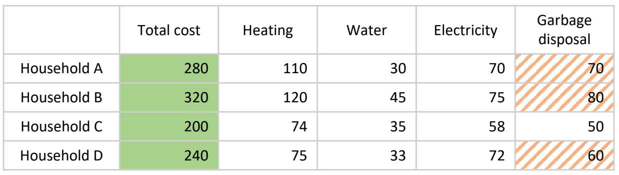
Given these pruning rules, in the example of Table 2, only the aggregations with the 1*st* and 4*th* column patterns are preserved. Six of the seven numeric rows, which is more than 0.7 given as the *cov* parameter, are compliant rows in both cases. Therefore, all aggregations in these rows that have one of the patterns are included in the output of this phase.

Non-cumulative aggregation functions, according to Table 1, need no more iterations, because the detected aggregates cannot be used as ranges any more (line 14 of Algorithm 1). For cumulative aggregation functions, the detected range columns are ignored in the next iteration (line 15), because they cannot be used as aggregates or ranges by any other aggregations that will be detected in the following iterations.

The individual aggregation detection phase outputs a set of row-wise and column-wise aggregation candidates for each aggregation function. The sets of results are consumed by the next phase that removes spurious candidates.

3.2 Collective aggregation detection

The previous phase recognizes aggregations of individual functions with the three pruning rules. However, there might exist a number of false positive results – mathematically correct but coincidental aggregations. Figure 6 demonstrates a fictitious example that has a true aggregation summing up the cost of heating, water, electricity, garbage disposal cost to a total cost. However, this table also includes a spurious average aggregation candidate, which calculates garbage disposal cost by taking the mean cost of the other three individual items. The coverage of this pattern is 3 of 4 (indicated by the orange shaded cells in the "Garbage disposal" column), surpassing the *cov* parameter. Therefore, the aggregation candidates with this pattern could not be removed by the previous phase.



	Total cost	Heating	Water	Electricity	Garbage disposal
Household A	280	110	30	70	70
Household B	320	120	45	75	80
Household C	200	74	35	58	50
Household D	240	75	33	72	60

Figure 6: A fictitious example table with true sum aggregates (green solid fill) and spurious average aggregates (orange diagonal strip fill).

To remove them, we introduce the collective aggregation detection phase. By using pruning rules on the collection of results from all individual detectors of the previous stage, we refine these results: similar to the pruning step in the first phase, aggregation candidates with a particular pattern cannot co-exist with those of another particular pattern in the final results. However, in the first phase each detector refines the detected aggregations of its own function, whereas in this phase, AggreCol performs the refinement across all functions.

Specifically, the algorithm first groups the accepted aggregations from all individual detectors by their column patterns, similar to the process in the first phase, and ranks the aggregation groups by two criteria: (i) the number of range elements in the group of a column pattern; (ii) the number of detected aggregations in a group. As the primary criterion, the more elements an range comprises, the higher the group's priority is, because we believe that an aggregation with fewer range elements is more likely to be a false positive. For the secondary criterion, we favor column patterns induced by a greater number of detected aggregations.

Once AggreCol has re-ordered aggregation groups, it filters out groups contradicting the ones that have been validated, according to the complete inclusion and mutual inclusion rules suggested in Section 3.1. Besides that, AggreCol discards an aggregation group if the aggregate in its pattern is the same as that of a previously validated group, and the ranges of the two groups overlap: if a cell is the aggregate of a particular aggregation function, it should not act as the aggregate of another function, while using (partly) the range of the former one. However, it is valid for a cell to serve as aggregate for two aggregations with disjoint ranges. For example, the yearly net income of a company that equals the difference between the gross income and the expense can simultaneously be the sum of the net income of all quarters.

These rules are not applicable to the division function, because division aggregations can always be included in the final result. For example, given that a_2 in Figure 5 is validated, it is reasonable to justify a_4, even though these two aggregations contradict each other according to the complete inclusion rule. This division records the percentage that a part (column 8) accounts for in the entirety (column 7) – a frequent usage of division in our observations.

3.3 Supplemental aggregation detection

While the collective aggregation detection phase can eliminate many spurious aggregations with the proposed pruning rules, difficult cases, such as interrupt aggregations, still cannot be retrieved. Take the interrupt aggregation shown in Figure 3c as an example: only the average aggregation can be retrieved by applying the first two phases. Because the sum detector in phase one can identify only adjacent aggregations, it cannot recognize the interrupt sum aggregation in this case. We introduce the supplemental aggregation detection phase to detect such interrupt aggregations.

The general idea is to apply individual detectors proposed in Section 3.1 on a set of constructed verbose CSV files derived from the original file. Each of the new files is built by systematically removing certain aggregate cells that have been detected in the previous steps. As such "blocking" aggregates are removed from the file, some interrupt aggregations become detectable using the original individual detectors. Algorithm 2 describes the supplemental aggregation detection approach. It takes the original file, the aggregation functions, the detected aggregations from the previous phase, the error level, and the aggregation row coverage as input, and outputs a set of detected aggregations.

Algorithm 2: Supplemental aggregation detection (row-wise)

```
Input: Verbose CSV file $V$, aggregation functions $F$,
       detected aggregations $A$, error level $e$, line
       aggregation coverage $cov$
Output: A set of row-wise supplemental aggregations D
1  $detectors \leftarrow \{d_f | f \in F\}$;
2  $q \leftarrow detectors$;
3  $D \leftarrow \{\}$;
4  while $q \neq \emptyset$ do
5      $d \leftarrow pop(q)$;
6      $VS \leftarrow construct\_files(V, D)$;
7      $res \leftarrow \{\}$;
8      foreach $V'$ in $VS$ do
9          $res \leftarrow res \cup d(V', e, cov)$;
10     end
11     if $res \neq \emptyset$ then
12         $D \leftarrow D \cup res$;
13         $q \leftarrow \{detectors \setminus d\} \cup q$;
14 end
15 $D \leftarrow prune(D, cov)$;
16 return $D$
```

First, individual detectors of all aggregation functions F are pushed to a queue (line 2), so that each will be executed at least once. The algorithm constructs a set of files based on the original file V by removing specific columns from V (line 6). On the one hand, all aggregate columns of detected non-cumulative aggregations should be excluded from the constructed files when detecting interrupt aggregations, because they cannot be used as range elements. On the other hand, each aggregate column of detected cumulative aggregations can be either excluded from or included in the constructed files, leading to multiple configurations to construct files. The algorithm constructs one file for each configuration, and applies the individual detectors d on all these files to recognize more aggregations (lines 7-10).

The approach runs sequentially: individual detectors are executed one after another. Once a detector discovers new aggregations, the queue reloads detectors of other aggregation functions that have already been processed (line 13). Newly detected aggregations may expose interrupt ones, leading to the necessity to re-check the other functions. Therefore, some detectors might be executed multiple times. The whole process terminates when no aggregation detector reports new aggregations. Finally, the algorithm applies the pruning rules used in the individual aggregation detection phase again to filter out spurious supplemental aggregations (line 15).

To summarize, AggreCol employs a three-stage approach to detect aggregations, where the first stage recognizes simple and cumulative occurrences of individual aggregation functions. The second stage combines individual aggregation results into one collection and aims at removing spurious ones. To extend the ability of AggreCol on discovering interrupt aggregations, we apply the third stage that utilizes the individual detectors from the first phase on a set of reconstructed files.

4 EXPERIMENTAL EVALUATION

This section includes the description of our evaluation datasets and the results of our in-depth qualitative evaluation of AggreCol including a comparison with a baseline and a detailed error analysis.

4.1 Datasets

We collected real-world verbose CSV files from a variety of sources, and constructed two datasets from them to evaluate our approach on detecting aggregations in verbose CSV files. Most tables in these files have fewer than 100 rows and 50 columns, whereas the longest and the widest tables contain 601 rows and 97 columns, respectively.

The first dataset combines files from Troy and EUSES. Troy [24] is a set of 200 verbose CSV files collected between 2009 and 2010 from various international statistic data portals, such as Statistics Finland and The World Bank. The authors evaluated their approach that transforms heterogeneous statistical web tables into relational databases [11]. The original data were stored in HTML and converted by the authors via Excel to verbose CSV files. EUSES [12] includes a collection of 1,352 spreadsheet files from diverse domains, such as data management, education, and finance and is widely used [1, 16, 18]. We randomly sampled 200 files from this dataset, and converted them to CSV format with the Apache POI library[3]. Dropping the files that could not be processed by the library left us with 185 verbose CSV files. As both datasets incorporate files from diverse domains, we merged them into a single dataset referred to as Validation, as we used it to validate the effectiveness of AggreCol while designing our approach.

The second dataset comprises verbose CSV files from three different datasets: the Statistical Abstract of the United States (SAUS), the Criminal In the US (CIUS), and the administrative spreadsheet files on an open data portal of the UK. These datasets include in total 3,053 files, and have been used by related work in classifying lines or cells [15, 20]. We randomly sampled and annotated 100 files and created our Unseen dataset with those 81 files that contain aggregations. This dataset served purely as an unseen test set to assess the generalizability of our approach.

Naturally, verbose CSV files do not contain aggregation information. Thus, we implemented a dedicated annotation tool to manually create aggregation annotations for our datasets. According to Definition 4, an aggregation annotation must include three components: a single cell that acts as the aggregate, a set of other cells as the range, and the aggregation function. Therefore, each annotation includes all the three components. Apart from the typical purely numeric aggregations, we observed that

[3]https://poi.apache.org/

Table 3: Statistics of datasets.

Observations	Dataset	
	Validation	Unseen
Number of files with	385	81
No aggregations	50	0
Aggregations of one type	259	62
Aggregations of two types	71	17
Aggregations of three types	5	2
Aggregations of all types	0	0
Number of aggregations	20,280	5,854
Sum	14,070	4,399
Average	858	33
Division	4,800	1,097
Relative change	552	325
Number of aggregations with	20,280	5,854
error = 0	14,479	4,020
error > 0	5,801	1,834
Min. per-file aggregation count	1	1
Max. per-file aggregation count	1,651	490

verbose CSV files may include aggregations that comprise non-numeric cells. A typical example is the usage of 'x' or '-' in cells to represent zero. Another unusual case represents the number '1.4' with the string '+1.4 Points'. Our annotations cover all cases as such, to better reflect the true aggregations in the datasets. Note that aggregations may be subject to rounding errors. Even when an aggregate could not be precisely derived from its range, we labeled an aggregation based only on its semantics as indicated context, including file and column names, surrounding cell value, and our general understanding of the respective domain.

Table 3 displays some basic statistics of our datasets showing the high complexity of aggregations in data files. Around 20% of the files in both datasets include aggregations of more than one type. Sum is the most frequently used aggregation function, accounting for about 70% aggregations in both datasets. Aggregations in real-world data files often have errors – their aggregate does not precisely aggregate the range. This is the case for about 29% of all aggregations in our datasets.

4.2 Number format transformation

Numbers can be formatted in different ways: the decimal separator may vary depending on the cultural background of file authors, and a thousands separator can differ or be absent entirely. One file may use '12 345,67' and another '12,345.67' to represent the same underlying number '12345.67'. Cell values in verbose CSV files commonly preserve these formats of numbers. As for aggregations, an incorrect interpretation of the numbers in a file might cause incorrect calculation results. For instance, '1.000' may or may not be a correct aggregator for $700 + 300$ if the thousands separator character is treated as the decimal separator.

Before detecting aggregations in verbose CSV files, we first apply a pre-processing step to identify and normalize the number format. An investigation into the Troy dataset indicates five valid number formats, shown in Table 4. We propose a number format transformer that converts values of numeric cells in a file into a normalized format. A normalized format uses no thousands separator and the dot as the decimal separator. We created a regular expression for each valid number format. For each cell in the file, we tried to match it with every regular expression. As a consequence, numeric cells might be matched to one or multiple number formats, while non-numeric cells do not match any. We

Table 4: Overview of valid number formats and their respective occurrence in the 200 files from Troy.

Digit group separator	Decimal separator	Example	Occurrences
Space	Comma	12 345,67	24.5%
Space	Dot	12 345.67	6.0%
Comma	Dot	12,345.67	66.5%
None	Comma	12345,67	1.5%
None	Dot	12345.67	1.5%

selected the number format that matches most cells in the file and performed the appropriate transformation. In case of ties, we chose the number format that has a higher occurrence ratio according to Table 4.

4.3 Quality evaluation

We conducted a series of qualitative experiments to test the ability of AggreCol on recognizing aggregations with a variety of patterns in the datasets. First, we introduce the metrics used in our experiments. Then, we present the evaluation results on the Validation dataset upon both aggregation- and file-level. Our last experiment investigates the generalizability of AggreCol by applying it on the Unseen dataset that remained unseen while designing the approach.

4.3.1 Metrics. According to Definition 4, two aggregations match only if their aggregates, ranges, and aggregation functions all match, respectively. We refer to a detected aggregation as *correct* if it matches some true aggregation in the ground truth, and *incorrect* if not. A true aggregation in the ground truth is *missed* if no detected aggregation matches it. With these interpretations, we apply the commonly used *precision* (P) and *recall* (R) metrics to measure the effectiveness of our approach:

$$P = \frac{|correct|}{|correct + incorrect|} \quad R = \frac{|correct|}{|correct + missed|} \tag{1}$$

Precision counts the number of correctly detected aggregations amongst all detected ones, while recall counts the number of correctly detected aggregations amongst all in the ground truth. The F1-score is the harmonic mean of these two measures. Note that precision is undefined when no result is returned by an approach, similarly for recall if a dataset contains no true aggregations. As usual, we set the score to 1 in both cases.

4.3.2 Aggregation-level effectiveness. The first experiment evaluates the effectiveness of AggreCol at individual aggregation level, considering all aggregations from all verbose CSV files in a dataset equally, regardless of the files they belong to.

Three parameters affect the performance of our approach: (i) the error level e, which specifies how much calculation error the approach tolerates; (ii) the line aggregation coverage cov, which indicates the minimum percentage of numeric cells in a row or column recognized as aggregates of aggregations with the same pattern; (iii) the window size used by the sliding window strategy, which specifies how many numeric cells in the proximity of an aggregate candidate should be checked.

Here, we fix the window size at 10 to cover the majority of the difference, division, and relative change aggregations. Before selecting the best error level for each aggregation function, we first set the coverage as 0.7, as our experiment shows that the average F1-score across aggregation functions is highest when

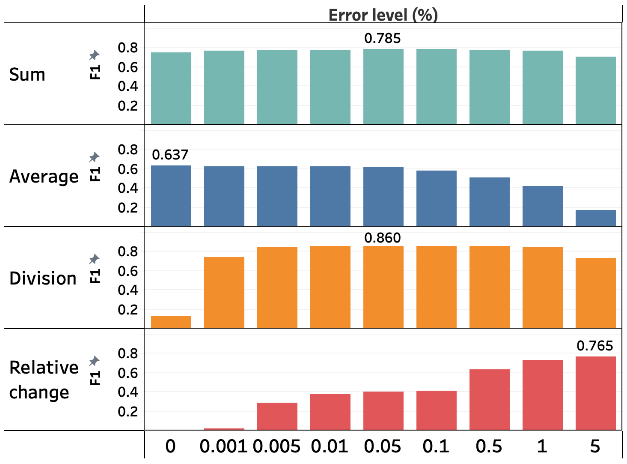


Figure 7: Per-function recall and F1-scores under different error levels. Line aggregation coverage is set as 0.7.

using this coverage value. Also, we select the best error level for each function with the results of individual detectors proposed in Section 3.1. Figure 7 illustrates the recall and F1-scores our approach achieves for individual aggregation functions with coverage value $cov = 0.7$, under different error levels. Note that a difference aggregation can be trivially transformed to a sum aggregation by moving the minuend to the aggregate side. Therefore, we merge the ground truth of sum and difference, and the evaluations on them as well.

We observe a common trend of the change of F1-score against the increasing error level: it first increases as small increments of the error level allow more true positives to be retrieved and are still not large enough to include many spurious cases. As the error level becomes sufficiently large, e.g., 5% for division, F1 starts to decrease, because the algorithm incorporates many occasional false positive cases in the prediction and sharply decreasing the precision.

Overall, the F1-score reaches maximum at different error levels for different aggregation functions. We identify and select the respective optimal error levels for the aggregation functions for the following experiments.

Given the selected values for both per-function error level and line aggregation coverage parameters, we explored the effectiveness of different stages of AggreCol. Figure 8 demonstrates the precision, recall, and F1-scores for each aggregation function. The letters 'I' 'C', and 'S' in the x-axis represent the individual, collective, and supplemental aggregation detection stages, respectively. As each stage depends on the result of the previous stage, the metric score of a particular stage indicates the results obtained by applying AggreCol until this stage.

On the one hand, applying the collective aggregation detection stage increases precision across all aggregation functions, because this stage removes spurious candidates detected in the first stage with a set of pruning rules. Although the pruning could also drop some correct candidates, our experiment shows no or only a very minor drop of recall. On the other hand, adding a supplemental aggregation detection phase to the workflow achieves better recall, as expected. An investigation into the results indeed shows the detection of some interrupt aggregations. Overall, employing all phases yields a better F1-score across all aggregation functions.

4.3.3 File-level effectiveness. Verbose CSV files may have different numbers of aggregations: the file with most aggregations contains 1,651 cases, while the one with fewest only one case in the Validation dataset. The aggregation-level results of the previous section may be biased in favor of long and wide files with many aggregations. To deal with this bias, we conducted an additional file-level evaluation on the Validation dataset. For each verbose CSV file, we compared the entire set of aggregations detected by our approach against the collection of real aggregations in it and determined which percentage of files meets certain minimum precision and recall values. Figure 9 presents the results, where the y-axis represents the percentage of the files on which AggreCol achieves the score in the range given by the x-axis. The score ranges between zero and one is divided into 20 bins, each spans 0.05. As AggreCol achieves medium-range precision or recall (between 0.05 and 0.95) on very few files, we group the bins in this scope into three larger groups, each stretching over 0.3 on the score.

At file-level, AggreCol achieves greater than 0.95 precision and recall for more than 90% of the files with regard to average, division, and relative change. The corresponding scores for aggregation-level results are all below 0.9, and even below 0.6 for the recall of average detection, indicating that most false negative and false positive aggregations are concentrated in few files. Compared to recognizing the above three aggregation functions, correctly detecting sum aggregations in a verbose CSV file is more challenging – our approach achieves more than 0.95 precision and recall for 79.2% and 61.6% of the files, respectively. As around 70% of the aggregations in our ground truth are sum, each file includes on average more sum aggregations than other types, increasing the difficulty to discover all of them correctly. The grey bars demonstrate the overall precision and recall across all aggregation functions. For some cases, it is lower than the precision or recall of any single function, due to undefined precision or recall values that are adjusted to 1.

Overall, the file-level evaluation results indicate that false negative and false positive cases yielded by our approach tend to occur only in few files.

4.3.4 Test on an unseen dataset. Due to ad-hoc shapes and forms of verbose CSV files, a set of 385 files can hardly cover all aggregation patterns. Therefore, we tested the generalizability of our approach with a second dataset after completing the algorithm design. Figure 10 demonstrates the file-level precision and recall achieved by AggreCol on this dataset.

Similar to our approach's performance on the Validation dataset, precision and recall are higher than 0.95 for the majority of the files, and sum detection is the most challenging task on this dataset as well. The similar results on the two datasets indicate that our solution is not tailored for the Validation dataset. A noticeable difference of precision in the results of the two datasets appears in the right-most group ($0.95 < P \leq 1$), where our approach obtains only 0.630 on the Unseen dataset, which is caused by the prevalence of zero-valued cells in the files.

4.4 Comparison to baseline

There is no previous work dedicated to automatically detecting and specifying aggregations in verbose CSV files. Therefore, we compared AggreCol with a baseline approach that eagerly checks all possible range element combinations as possible: for each numeric cell in a verbose CSV file, the baseline traverses the permutations of all numeric cells in the same row or column,

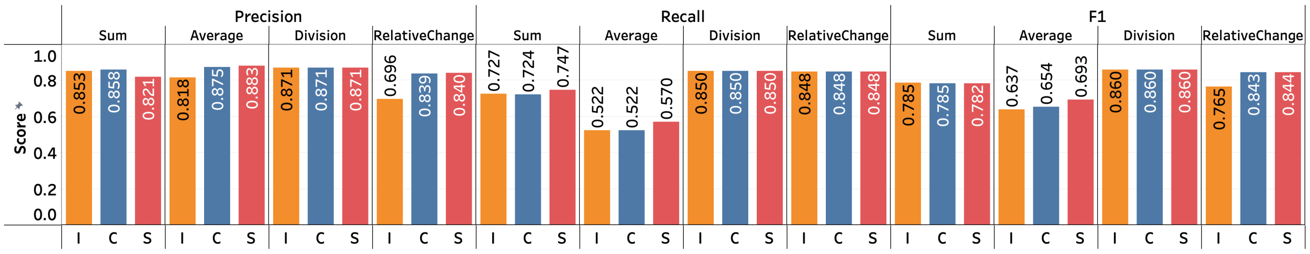


Figure 8: Precision, recall, and F1-scores for each aggregation function obtained by AggreCol at different stages. The individual, collective, and supplemental stages are indicated as "I", "C", and "S" in the x-axis.

treating each permutation as a range candidate. Given a verbose CSV file with n columns, the complexity of the traversal is in $O(n * 2^{n-1})$ for aggregation functions that involve at least two range elements, and $O(n^3)$ for those having only two range elements. The computational burden is thus infeasible in general, and we set a 5 minute timeout for each file. Within that limit, this approach managed to process only 202, 203, 379, and 380 files regarding sum, average, division, and relative change detection, respectively, out of 385 verbose CSV files in the Validation dataset.

The baseline was run on a Mac Pro 2019 using 4 cores. We also tested AggreCol on the same hardware with the same time limit. Our proposed approach is able to finish within 5 minutes per-file for 381 files, which cover all those that can be handled by the baseline. Therefore, only the baseline processable files are used for the following analysis. Note that even with a 20-minute time limit, the baseline still cannot finish all files. In contrast, the longest time taken by AggreCol for any single file is about 599 seconds. Because the individual detectors of AggreCol process each aggregation candidate independently, they can be easily implemented in parallel to improve efficiency. Phase 3 costs on average 85% of the runtime in the entire workflow.

Figure 11 shows the F1-score achieved by the eager baseline approach and AggreCol with the same error level setting for each function. For all functions, AggreCol achieves more than 0.95 on F-1 for more than 60% of the files, whereas the baseline obtains such an F-1 score for only up to 35% of the files. For the majority of the files, the baseline achieves less than 0.05 on F-1, which is caused mainly due to poor precision of the baseline, as it checks every possible range element permutation and predicts numerous false positives. For example, a file with many zeros or ones includes many false positive sum cases. As a consequence, the baseline approach is not an ideal solution to obtain high F1-scores.

Table headers in verbose CSV files could indicate the presence of aggregates in the same row or column. For example, if a header cell includes the word 'total', it might imply that the numeric cells in its row or column might be sum of several other numeric cells. However, an investigation into the Validation dataset shows that only 60.0% of the real sum aggregates use one of the keywords 'total', 'all', 'sum', 'subtotal', and 'overall' in their header. The corresponding ratios for average, division, and relative change are 86.6%, 53.1%, and 92.4%, with a unique keyword dictionary for the respective aggregation functions. What is more, these keywords are often used in rows or columns without aggregates. Our experiments show that the precision scores of detecting aggregates are 0.565, 0.256, 0.458, and 0.038 for sum, average, division, and relative change, respectively.

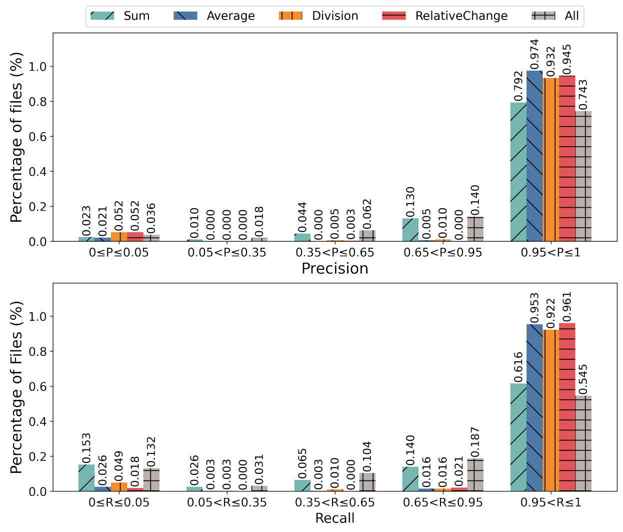


Figure 9: File-level precision and recall obtained by AggreCol on the Validation dataset.

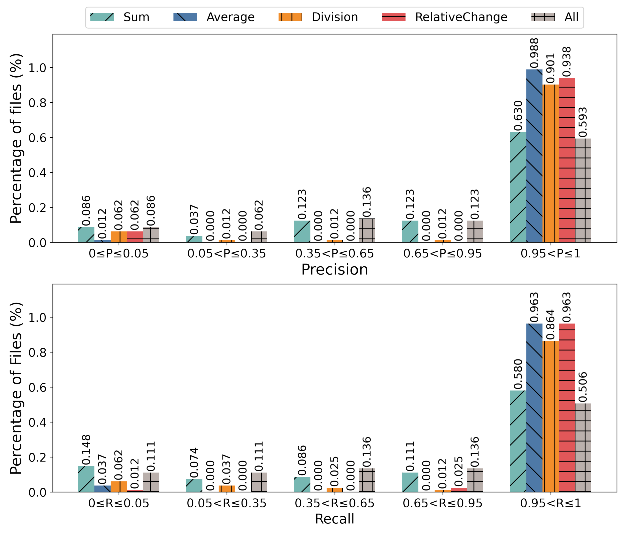


Figure 10: File-level precision and recall obtained by AggreCol on the Unseen dataset.

4.5 Analysis of detection errors

Depending on how the authors organize their data, aggregations in verbose CSV files have all sorts of patterns. Due to this diversity,

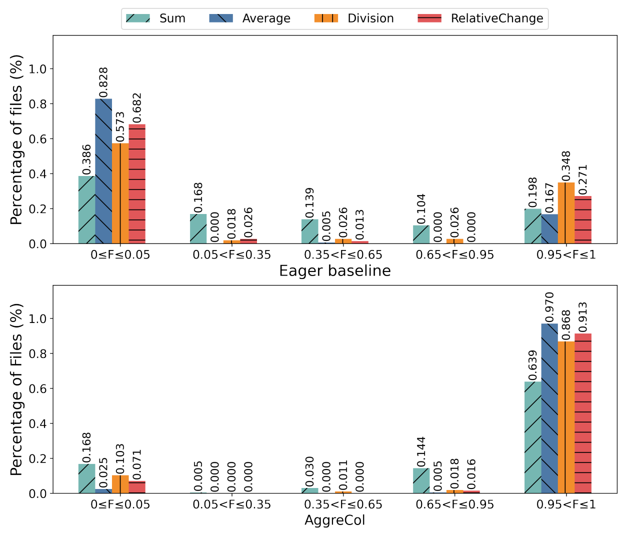


Figure 11: File-level F1-score comparison between the eager baseline approach and AggreCol.

recognizing aggregations in all possible situations is challenging. Here, we analyze the mistakes produced by AggreCol on the Validation dataset, and summarize the major causes that lead to false positive and false negative cases.

A general reason that causes both false positive and false negative cases is the selection of the parameter error level value. As we used a fixed error level for each function determined by the aggregation-level F1-score in our experiment, it may not be resilient to different orders of magnitude of aggregate values. A difference between r and r' normalized by a smaller r is larger than one normalized by a larger r according to Definition 5. Therefore, the fixed error level might be too large for big aggregate numbers and yield incorrect results, or too small for small numbers and causes missed cases.

4.5.1 False positive cases. A particular reason that causes many of the false sum predictions is the small number of different numbers in table cells. For example, when marking absence as '0' and presence as '1', a table for the student roster of a course might include only these two numbers. Therefore, it is very likely that many spurious sum results, such as '$1 = 0 + 1$' are generated. In our experiment, most mistakes involved many '0' valued cells. We make similar observations for the division detection results that involve many '1' valued cells.

Empty or almost empty lines and columns can be misinterpreted as contributing the number zero to the aggregatee. We observe such effects in only very few cases and a corresponding preprocessing might lead to more false negatives.

4.5.2 False negative cases. Ranges of some aggregations in the ground truth are both of the interrupt kind and not in the proximity of their aggregates. Although the supplemental aggregation detection phase retrieves some interrupt aggregations, it cannot deal with cases whose ranges are not interrupted by aggregates of other detected aggregations. For example, consider a table that has columns for trade volume, import volume, and export volume of a country with other countries in a whole year and for every month. The aggregation that sums up the per-month import volume to the yearly one cannot be detected, because the per-month export volume columns, which are not detected aggregates, cannot be removed when constructing files in the third phase. Another reason causing false negative cases is that the selection of a fixed window size cannot cover the whole ground truth. Third, we observed that very few pairs of true aggregations break the 'directional agreement' rule proposed in Section 3.1. However, our approach could retrieve these cases by dismissing this rule at the cost of introducing many false positives into the results. Lastly, ranges whose last cells are '0'-valued could be missed when detecting sum, because our adjacency list strategy stops when encountering zeros. If every sum with the same pattern has the same number of tailing zero cells, an aggregation with fewer range elements is detected, rather than the true one.

We tested our approach with different error levels, yet none of them can completely eliminate detection errors caused by the selection of this parameter value. We did not use particular techniques tailored to address the aforementioned special cases, as they appear in only few files. Resolving them could lead to many more detection errors overall.

Table 5: Per-type F-1 scores of the cell classification approach Strudel by using its original aggregation cell detection approach ($Strudel^O$) or AggreCol ($Strudel^A$).

	SAUS		Troy	
Cell type	$Strudel^O$	$Strudel^A$	$Strudel^O$	$Strudel^A$
metadata	0.987	**0.989**	0.975	**0.976**
header	0.976	**0.978**	0.959	**0.963**
group	0.731	**0.806**	0.638	**0.740**
data	0.967	**0.976**	0.950	**0.962**
aggregation	0.677	**0.786**	0.615	**0.740**
notes	**0.957**	0.956	**0.968**	0.966

4.6 Cell classification improvement

Cell classification aims at classifying each cell in a CSV file as one of several pre-defined semantic types, among which "aggregation" is a common one [15, 20]. AggreCol can contribute to the improvement of overall cell classification performance, as we show for one example: our supervised-learning based cell classification approach Strudel uses a binary feature to represent whether a cell is an aggregate (sum or average) [20]. The authors used an approach similar to the adjacency list strategy of AggreCol to detect aggregates in CSV files. We replaced the values of that feature with the results of AggreCol on the SAUS and Troy datasets[4] used there, and conducted the same cross-validation experiments described in that work. Table 5 shows the per-type F-1 score of the original Strudel algorithm and the version using AggreCol's results.

For both datasets, the F-1 score of the aggregation type increases significantly, as expected. When using AggreCol's output in the feature of Strudel, more cells previously classified as one of the other types are predicted correctly as aggregation, increasing the precision scores, and also the F-1 scores, of most other types.

5 RELATED WORK

Only two previous efforts have been made to directly address aggregation detection in individual verbose CSV files. Long et

[4]Annotations have been revised due to some label errors.

al. introduced a keyword-based approach to recognize only summation in tables of plain-text files [22]. They suggested using two categories of keywords: direct aggregation keywords, such as 'total' and 'sum', and complementary keyword pairs in two rows or columns, such as 'student' and 'non-student'. The Strudel approach discussed in Section 4.6 detects whether a cell is the sum or average of some other numeric cells [20]. This approach can deal with only adjacent aggregations depicted in Figure 3, missing all cumulative and interrupt cases. The result was then encoded in a binary feature to detect classes of lines and cells in verbose CSV files.

While both the aforementioned works exploited keywords, these information are not always reliable in locating aggregations. A limited number of keywords is not general enough to cover all cases. Meanwhile, having keywords in headers does not always imply the existence of aggregations.

Aggregations might reference information, such as weights of a weighted sum, in other tables. Singh et al. suggested a programming-by-example approach to recover aggregations in a table by looking up information across several tables [26]. However, unlike AggreCol, this approach requires users to provide input examples.

Despite the lack of previous work on aggregation detection, there is related work on downstream tasks that require aggregation information as input. In the rest of this section, we discuss related work on these use cases.

5.1 Structure detection

Information in data files, such as spreadsheets and verbose CSV files, are often organized in an ad-hoc manner: authors treat files like a canvas and drop information at arbitrary positions therein, which makes it difficult to process data in these files automatically by a machine. Before we can extract information from these files, it is important to understand their overall structure (not just aggregations) by recognizing the types of content in different regions.

Identifying types of lines or cells in data files is a typical way to tackle the structure detection task. Various previous works have been proposed for this purpose [2, 7, 15, 20, 21]. Common content types include data, header, metadata, aggregation, and so on. Distinguishing aggregation content from data content is usually challenging, as content of both types are often numbers that are similar to each other. An advanced aggregation detection approach helps locate lines or cells of this type more precisely by resolving the above challenge, and therefore can improve the structure detection performance. Detected structures in data files can help extract and transform information from data files [4–6].

5.2 Formula smell detection and repair

Spreadsheet formulas often contain "smells". The term formula smell is inspired by source code smells that indicate violations of programming principles, such as mysterious variable names and overly large classes [13]. Such smells may cause poor code readability and error-prone code refactoring. Similarly, a formula smell in a spreadsheet indicates a wide variety of poor usage of the formula functions, such as inconsistent formulas in a single row or column, missing formulas, and complex formulas that are caused by sloppy cell copy-and-paste, abuse of formula functions, etc.

Many research efforts have been dedicated to detect and repair smelly formulas [3, 9, 10, 17, 19]. However, these approaches all assume that some of the formulas in the files are present. Unfortunately, verbose CSV files do not preserve such information, and not every spreadsheet keeps formulas in its cells. Aggregation detection approaches can supply the necessary input to these approach on formula smell detection and repair tasks.

6 CONCLUSION AND FUTURE WORK

Aggregations represent arithmetic relationships between a set of numbers (range) and a single number (aggregate), and are common in tables of verbose CSV files. A variety of applications depends on the existence of aggregations. Identifying aggregations helps understand the structure of tables and provides insights on how raw data can be extracted from such files. In addition, data errors might be cleaned with the knowledge of the true aggregations.

We formalize the problem of recognizing aggregations in verbose CSV files and recognize three patterns of aggregations in data files. We propose the three-stage approach AggreCol to address this problem. Our approach can detect aggregations of five types in verbose CSV files: sum, difference, average, division, and relative change. We annotated 466 real-world verbose CSV files and conducted a series of qualitative experiments to show the effectiveness of our approach. For the validation dataset encompassing 385 data files, AggreCol achieves on average 0.795 F1-score in the aggregation-level evaluation, and more than 0.95 precision and recall for 91.1% and 86.3% of the files, respectively. A test on an unseen test data shows similar performance. Besides that, we also compared AggreCol with a baseline approach that retrieved eagerly too many spurious aggregations. Our study also reveals that real-world verbose CSV files incorporate surprisingly many errors in aggregations.

AggreCol does have limitations. In this work, we address aggregations whose aggregate and range are in the same row or column, which might not always be the case and could be relaxed. A further observation we made is that some aggregations involve more than one aggregation function in the calculation. For example, students' final scores for a course may be weighted by the importance of different modules (attendance, homework, exams, etc.) – both sum and division calculations are involved. Therefore, further work shall support multi-functional aggregations.

ACKNOWLEDGMENTS

This research was funded partially by the HPI research school on Data Science and Engineering.

REFERENCES

[1] Robin Abraham and Martin Erwig. 2007. GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 251–260.
[2] Marco D Adelfio and Hanan Samet. 2013. Schema extraction for tabular data on the web. *PVLDB* 6, 6 (2013), 421–432.
[3] Daniel W Barowy, Emery D Berger, and Benjamin Zorn. 2018. ExceLint: automatically finding spreadsheet formula errors. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–26.
[4] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.
[5] Zhe Chen and Michael Cafarella. 2013. Automatic web spreadsheet data extraction. In *Proceedings of the 3rd International Workshop on Semantic Search over the Web*. 1–8.
[6] Zhe Chen and Michael Cafarella. 2014. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*. 1126–1135.
[7] Christina Christodoulakis, Eric B Munson, Moshe Gabel, Angela Demke Brown, and Renée J Miller. 2020. Pytheas: pattern-based table discovery in CSV files. *PVLDB* 13, 12 (2020), 2075–2089.

[8] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-hypothesis CSV parsing. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 1–12.

[9] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. 2014. Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 848–858.

[10] Wensheng Dou, Shi Han, Liang Xu, Dongmei Zhang, and Jun Wei. 2018. Expandable group identification in spreadsheets. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*. 498–508.

[11] David W Embley, Mukkai S Krishnamoorthy, George Nagy, and Sharad Seth. 2016. Converting heterogeneous statistical tables on the web to searchable databases. *International Journal on Document Analysis and Recognition (IJDAR)* 19, 2 (2016), 119–138.

[12] Marc Fisher and Gregg Rothermel. 2005. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the First Workshop on End-user Software Engineering*. 1–5.

[13] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[14] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 883–899.

[15] Majid Ghasemi Gol, Jay Pujara, and Pedro Szekely. 2019. Tabular cell classification using pre-trained cell embeddings. In *Proceedings of the International Conference on Data Mining (ICDM)*. 230–239.

[16] Julius Gonsior, Josephine Rehak, Maik Thiele, Elvis Koci, Michael Günther, and Wolfgang Lehner. 2020. Active Learning for Spreadsheet Cell Classification.. In *Proceedings of the Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores*.

[17] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. 2012. Detecting code smells in spreadsheet formulas. In *International Conference on Software Maintenance (ICSM)*. IEEE, 409–418.

[18] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. 2013. On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science)*, Vol. 7793. Springer, 68–82.

[19] Bas Jansen and Felienne Hermans. 2015. Code smells in spreadsheet formulas revisited on an industrial dataset. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 372–380.

[20] Lan Jiang, Gerardo Vitagliano, and Felix Naumann. 2021. Structure Detection in Verbose CSV Files. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 193–204.

[21] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. 2016. Cell classification for layout recognition in spreadsheets. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*. Springer, 78–100.

[22] Vanessa Long. 2010. *An agent-based approach to table recognition and interpretation*. Ph.D. Dissertation. Macquarie University Sydney, NSW, Australia.

[23] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. 2016. Characteristics of open data CSV files. In *International Conference on Open and Big Data (OBD)*. IEEE, 72–79.

[24] George Nagy. 2010. TANGO-DocLab web tables from international statistical sites (Troy_200). http://tc11.cvc.uab.es/datasets/Troy_200_1. [Online; accessed March-2021].

[25] Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya Parameswaran. 2020. Benchmarking Spreadsheet Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1589–1599.

[26] Rishabh Singh and Sumit Gulwani. 2012. Learning semantic string transformations from examples. *PVLDB* 5, 8 (2012), 740–741.

[27] Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.