Structure Detection in Verbose CSV Files

Lan Jiang
lan.jiang@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

Gerardo Vitagliano
gerardo.vitagliano@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

Felix Naumann
felix.naumann@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

ABSTRACT

Numerous data are stored in semi-structured files with ad-hoc layout. Such data are valuable digital assets for various data-driven applications. This work introduces the notion of *verbose CSV files*. Verbose CSV files include content serving different purposes in various positions. They are designed for human visual inspection or statistical report collection. An important preliminary task for extracting information from such files is *structure detection*, in particular classifying lines or cells by their purpose. As manual efforts are infeasible and error-prone for large files or large sets of files, automatic approaches are desirable.

This work addresses both the *line* and the *cell classification* problems on verbose CSV files. Strudel is a supervised learning approach based on a random forest classifier, combined with a set of novel features that fall into three categories: content features, contextual features, and computational features. We annotated five real-world datasets from various domains, on which we tested our approach. Our in-depth experiments show the advantages of Strudel over baseline and state-of-the-art approaches in both line and cell classification tasks.

1 INTRODUCTION

The rapidly growing amount of data promises to be of great value for everyone's day-to-day life, for example, assisting doctors in personalizing healthcare solutions to their patients, scientists conducting open data-based citizen researches, and enterprises making better business decisions. To enable such applications, raw data must be properly processed and analyzed before generating insights. While some data are saved in well-defined formats, such as relational tables or as key-value pairs, that can be readily parsed by dedicated tools, a large quantity of other data are stored in documents with unique structures, for example, CSV files. CSV files are comma-separated values files that provide a great number of data sources for various data-driven tasks, such as data profiling [10, 24], data curation [22, 26, 30], and information extraction [9, 16]. Although there is a standard[1] that stipulates how data shall be stored in CSV files in theory, in practice users and applications do not always conform to it, producing documents with very unique structures.

This work addresses one such type of document: verbose CSV files. Later, we formally define a comma-separated value file as *verbose* if its raw values serve various purposes, such as data, metadata, group headers or notes, and appear in various positions. An example of a real-world verbose CSV file from the "Crime In the US" (CIUS) dataset is given in Figure 1, where groups of cells with different roles are highlighted. This file cannot be directly

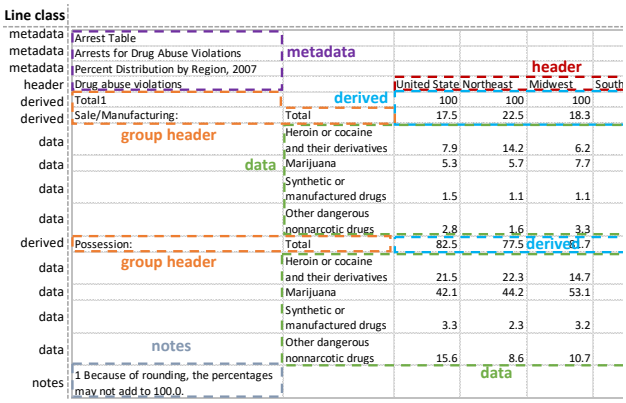

Figure 1: A real-world verbose CSV file with different cell-level and line-level content classes. Here, the line-class is determined by the majority of its cell classes.

ingested by common RDBMS tools, as it contains much additional information, aside from a table with its header and data rows.

While standard CSV files contain data in the form of a structured table, a verbose CSV file is more similar to a spreadsheet, in terms of its flexible content layout. Researchers have suggested that only a minority of spreadsheets (22% of 200 randomly selected spreadsheets) can be directly converted to relational tables [6]. A similar observation by Dong et al. states that less than 3% of spreadsheet tables are "machine-friendly" [11]. Spreadsheets are not the only source for verbose CSV files. We randomly selected 26,140 files from data that we crawled from the Mendeley data portal[2], and manually checked their file types. We grouped all files into three broad categories: (i) application-specific files, such as Microsoft office files; (ii) plain-text files; (iii) multi-media files. Amongst all plain-text files, we are interested in those with at least one table. Out of these files with tabular structures, there are 4,459 files that are verbose, accounting for around 20% of all inspected plain-text files. These verbose plain-text files can be easily transformed to CSV files by applying file-specific delimiters. In our experiments, we tested our approach on verbose CSV files derived from both spreadsheets and arbitrary plain-text files that contain data lines/cells.

Verbose CSV files are prevalent media to store data that aim at aiding human visual inspection or collecting statistical reports. These data often need to be shared amongst communities. Therefore, plain-text files, such as CSV files, are favored due to their generality over those with proprietary file types, such as spreadsheets, that are used by specific applications. However, compared to application-specific files, verbose CSV files are harder to parse, as they do not necessarily follow a specific format. Various open data portals include such files, sometimes labeled as 'ASCII' data[3].

[1] As decribed by RFC 4180: https://tools.ietf.org/html/rfc4180

[2] https://data.mendeley.com

[3] https://data.gov.uk/, https://www.govdata.de/, https://data.europa.eu/euodp/en/data/

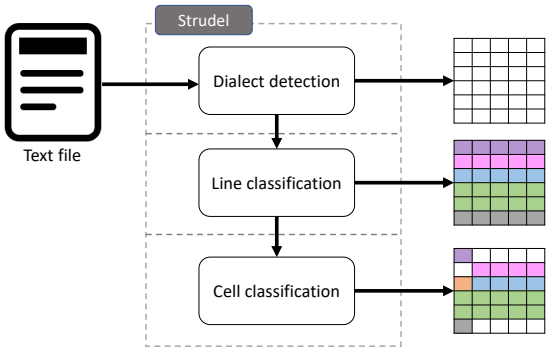

Figure 2: Architecture of the Strudel algorithm.

To obtain insightful information from these files, the first step is to understand their structure, i.e., detect the types of cells, lines, or data blocks. In practice, this is a challenging task due to (i) the wide range of ad-hoc layout variants that can be found and (ii) the lack of rich stylistic features that have been exploited by previous work [2, 11, 18]. Manual inspection to recognize file structure and acquire valuable information is extremely time-consuming for many and large verbose CSV files; automatic methods can support this task.

Information in verbose CSV files is usually organized in a tabular fashion, where each cell is an atomic content unit. File structure is often reflected in the sequence of classes of horizontal lines, as data organized in verbose CSV files usually conforms to the common top-to-bottom data presentation logic. For example, consider the line class labels of the example file in Figure 1. These classes show a natural logic of organizing information: metadata, such as captions, come first, followed by the main body of a table incorporating table headers, derived (aggregated) lines and data lines, and finally a few footnote lines conclude the file. In this work, we aim at *verbose CSV file structure detection* by means of classifying file content in two granularities: *lines* and *cells*.

Obtaining manually labeled datasets with known line and cell annotations to evaluate an approach for the structure detection problem is difficult, as ascertaining these classes in a verbose CSV file is a difficult task even for experienced practitioners. In our study, even by using a tool with a sophisticated graphical interface, practitioners 1) took on average of two minutes to label the lines in a single file, because they needed to spend a lot of time understanding the unique structure of each file; 2) at times disagreed with each other on the annotations of individual lines. These observations highlight the challenge of automatically classifying lines and cells in verbose CSV files.

To address the line/cell classification problem, we propose the Structure Detection in Verbose CSV Files (Strudel) approach, which is grounded on a multi-class random forest classifier. Figure 2 shows the architecture of the approach. It first detects the dialect of a text file, and creates a verbose CSV file from it, based on the dialect. Then Strudel classifies first lines and then cells therein with the proposed feature sets. Cells of different types are distinguished by colors. Sections 4 and 5 describe Strudel for line and cell classification, respectively. We propose sophisticated features to model the individual classes for both classification tasks. The features can be categorized into three groups: 1) *content features* parsing the values of cells or lines, such as cell length and amount of words; 2) *contextual features* comparing the inspected cell or line with its neighbors, such as the similarity of data types between lines/cells; 3) *computational features* seeking to connect lines/cells with each other by inspecting arithmetic correlations between them. The contributions of this work are summarized as follows:

(1) A supervised learning approach with novel features to address the structure detection problem for verbose CSV files.
(2) A dataset with more than 97,000 annotated lines in 226 files, reforged labels of datasets from related work based on our perspective on cell classes, and a dataset with 62 files transformed from plain-text files.
(3) An experimental comparison of Strudel with baseline and state-of-the-art approaches.

In the next section, we introduce the related work on line and cell classification tasks and other relevant areas. In Section 3, we formally define the classification problem and introduce the set of line and cell classes. Section 4 and 5 describe the core idea of Strudel, followed by Section 6, where we present the results and analysis of our experiments. We conclude in Section 7.

2 RELATED WORK

Extracting information from semi-structured documents has been a growing research field in recent years. Relevant research questions include how to locate tabular content in documents such as PDF files [20], and spreadsheets [12], how to distinguish relational tables from non-relational tables [4, 29], and how to extract relational tables from heterogeneous sources [5, 13, 14, 25].

Prior to extracting information from a semi-structured document, understanding its structure is necessary. Some techniques have been proposed to address the structure detection problem on various documents, such as web tables and spreadsheets, which include tabular material and have flexible layout. We summarize these works focusing on structure detection by classifying lines or cells, respectively.

2.1 Line classification

Pinto et al. suggest a conditional random field (CRF) learning approach to predict the label for lines of plain-text documents crawled from a open data portal [23]. For each document, a sequence of features is computed for its lines. The sequences of all documents are used by the CRF classifier to infer the label. This approach was later adopted to infer spreadsheet table schemata [28] and extract relational data from spreadsheets [5]. Moreover, it was extended by Adelfio et al. to recognize line classes in web tables and spreadsheets [2]. The authors suggest feature binning to generalize the training data and show the effectiveness of their approach on recognizing line classes in both HTML tables and spreadsheets crawled from several open data portals. However, the approach assumes the presence of stylistic features, such as font styles, or built-in spreadsheet formula features, which are not available in verbose CSV files.

A recent work has proposed the rule-based approach Pytheas for CSV file line classification [8]. To classify lines in a CSV file, the approach first determines for each line whether it is data or non-data by consulting a set of fuzzy rules, whose weights have been learned beforehand with a training dataset. These binary results are then used to determine the top/bottom borders of tables in the file. Finally, the approach exploits additional class-specific rules on the discovered table/non-table areas to further ascertain the class of each line. The core of this approach is the

design of the fuzzy rule set, which significantly impacts the consequences of table border discovery, and also line classification. However, such a fixed set of rules might fail to generalize to new circumstances in unseen data.

2.2 Cell classification

Finer-grained cell classification in tabular documents has been the subject of academic interest in recent years. Abraham et al. developed the UCheck framework, which includes a component to detect "cell roles", such as header and footer, in spreadsheets using several heuristics [1]. Cell roles are then used by the system to detect spreadsheet errors. The goal of their approach is to correlate cells in a table with their corresponding headers, thus they assume spreadsheets with only table regions as input data.

Koci et al. suggest a supervised learning approach with a post-processing component to repair classification errors [19]. The authors introduce five misclassification patterns and suggest that the occurrence of them in the results hints at a misclassification.

To reduce the amount of manual annotation effort, Chen et al. integrated an active learning technique into their spreadsheet cell classification approach [7]. In their iterative algorithm, a sheet selector presents the most uncertain spreadsheet to human labelers. The sheet is then labeled and included into a training set that is used to train a spreadsheet property classifier.

Ghasemi-Gol et al. suggest a recursive neural network (RNN) architecture on two seperately trained cell embeddings that capture the contextual and the stylistic semantics of cells, respectively [18]. Even though the authors mention the contextual impact on a cell from both neighboring and distant cells, they considered only the former ones in their approach. They built the stylistic features upon those suggested by [19].

In summary, these works all make use of stylistic features of their input. However, no such information can be obtained in verbose CSV files. In this work, we compare our approach with the RNN-based approach of [18], as it was reported to outperform the other approaches. In spite of using stylistics features to solve the task, the authors also reported the performance of their algorithm without their usage, enabling a direct comparison to our approach.

3 DEFINITIONS AND PROBLEM STATEMENT

We first define the notion of verbose CSV files. Then, we present the taxonomy of element classes used to label our datasets and present the detailed definition of each class. Based on these concepts, we formally state the problem addressed by our approach.

3.1 Verbose CSV files

A standard CSV file, according to RFC 4180, contains an optional header line at the beginning of the file, followed by a number of data lines. In contrast to that, a *verbose CSV file* may include elements of heterogeneous classes (which will be introduced in Section 3.2), possibly with empty visual separators. Here, an *element* is either a non-empty cell or a line that includes at least one non-empty cell.

Definition. *A* verbose CSV file *is a comma-separated values file with values including one or more of metadata, header, group, data, derived, and notes at arbitrary positions. Each line of the file may be composed of cells of one or more classes. Empty cells may represent either missing values or serve layout purposes.*

A standard CSV file stores a single table that is machine-readable, whereas a verbose CSV file may include multiple tables, and make use of empty cells and further cell types to improve human readability, leading to various configurations. Information of different kinds may be organized as connected clusters of cells throughout the file: each such cluster may include information of different types, such as data, metadata, or aggregations; a table may be divided by blank visual separators into several table fractions; etc.

3.2 Class taxonomy

Our taxonomy includes six semantic classes and is similar to that of [2], which addressed the line classification problem on web tables and spreadsheets. While in principle content of any class may appear at any location in a verbose CSV file, we enforce a few practical constraints on their possible position, reflecting the usual reading convention: from left to right and from top to bottom, assuming that tables are stacked only vertically. We describe in detail each class in the following list.

- **metadata.** Metadata are the descriptive text *above* a table. Such text may include the title of a table, or additional information on the content of the table. A metadata area may span across one or more lines and columns.
- **group.** In verbose CSV files, tables are often split into several fractions, each including data of a particular group. A group (a.k.a. group header) element serves as the label of such a fraction. We have seen in our datasets the group elements appearing both *above* and *below* header areas. Therefore, we allow both cases in our definition. Group cell may also serve as the leftmost string cells in a derived line, e.g., the 'Sale/Manufacturing:' cell in Figure 1.
- **header.** Headers are the column labels in the top area of a table (or table fraction). Headers may span multiple cells. In our definition, the header elements are located *above* the data area, and *below* any metadata block of the table.
- **data.** Data elements are the content of a table that cannot be derived from any other elements. Because they constitute the main body of a table, data elements of a section of a table are always *below* header and group elements that indicate this section.
- **derived.** A derived cell aggregates the values of some other numeric cells in the same table. In verbose CSV files, derived cells are usually organized as the top- or bottom-most lines, or the left/right-most columns of the data area of a table.
- **notes.** Notes are descriptive text that follow a table. They may give explanations of particular parts of a table, explain the meaning of marks used in the table, or indicate the data source origin.

Any line or cell in a verbose CSV file can be associated with exactly one of the classes introduced above. We address the following problem: *Given a verbose CSV file and the group of possible element classes, how can we determine the classes of all elements?* We consider elements of two natures addressing two sub-problems under the same problem definition: *lines*, reflecting the usual vertical arrangement within a file, and *cells*, as the most fine-grained element of a structured file.

4 LINE CLASSIFICATION

In this section, we describe *Strudel*^{L} – the Strudel approach to deal with the line classification problem. *Strudel*^{L} is based on a

multi-class random forest classifier. Its input includes a set of features extracted from the two-dimensional tabular data.

Various kinds of features have been proposed by previous work to address the line classification problem, including content features, contextual features, spreadsheet formula features, and stylistic features [2]. In our case, formula and stylistic features cannot be applied, as CSV files do not preserve these rich-text information. Instead, we design a set of content features, contextual features, as well as computational features. We build the features of *Strudel*^{L} on top of the applicable features from the previous work [2]. Table 1 lists our complete set of features, divided into the three groups. The context features can include information from both the line above and the line below. Thus, the features marked by a star are applied twice – once for the line above and once for the line below. To distinguish derived cells from data cells, we propose novel *computational* features that check whether the value of a numerical cell can be calculated by applying a specific aggregation function on the numbers in the vicinity, i.e., the cells in the same row or column.

Table 1: Line classification features: '∗' marks contextual features applied to both lines above and below the inspected line; '†' are adapted from [19].

Category	Feature	Value
Content	EmptyCellRatio†	[0.0, 1.0]
	DiscountedCumulativeGain	[0.0, 1.0]
	AggregationWord†	0/1
	WordAmount	[0.0, 1.0]
	NumericalCellRatio†	[0.0, 1.0]
	StringCellRatio†	[0.0, 1.0]
	LinePosition†	[0.0, 1.0]
Contextual	DataTypeMatching*	[0.0, 1.0]
	EmptyNeighboringLines*	[0.0, 1.0]
	CellLengthDifference*	[0.0, 1.0]
Computational	DerivedCoverage	[0.0, 1.0]

Here, we describe and explain only the novel features used in our approach and refer to related work for the others.

- DiscountedCumulativeGain (DCG) calculates the discounted cumulative gain on a vector created from the cells of a line. The vector has the same length as the number of cells in a line. An element is set to '1' if the corresponding cell in the line is non-empty, or '0' if it's empty. This feature is exploited to model the pattern of empty cells. DCG gives more weight to left-more positions than to right-more positions, modeling users laying out data from left to right.
- AggregationWord checks whether a line contains any word that belongs to a pre-made dictionary of terms associated with aggregation in tables (case-insensitive): `total`, `all`, `sum`, `average`, `avg`, `mean`, and `median`. An existence of any keyword gives '1' to this feature, otherwise '0'. Using a dictionary of such kind of keywords proves to be effective [19].
- WordAmount calculates the number of words in all cells of a line. A word is a sequence of alphanumeric characters. The feature values are normalized per file by using a min-max normalization strategy.
- DataTypeMatching calculates the percentage of line cells whose data types match with those of the adjacent line (above or below). Note that some files insert an empty line between every pair of non-empty lines to visually highlight the content. However, comparing the data type of a line with an empty adjacent line does not carry much information. Therefore, an adjacent line refers to the closest non-empty line. Data and derived lines tend to have numerical cells while header lines usually contain alphanumeric values. Other functional lines, such as metadata, notes, and group lines tend to have many empty cells, as they often contain values for only the first cell in a line.
- EmptyNeighboringLines calculates the percentage of empty lines in the five lines above or below the inspected line. Empty lines are often used as visual separators in verbose CSV files. Using such a separator between data lines within a table is uncommon, but placing them between two classes of lines, such as header-data and derived-notes is more common.
- CellLengthDifference calculates the cell value length difference between two adjacent lines by calculating the Bhattacharyya histogram difference on the sequence of cell value length of the two lines. When computing this feature, we compare only a line with its closest non-empty neighboring line, similar to that applied for the DataTypeMatching feature. While data lines tend to have similar cell-wise value lengths, as they usually describe the same property and thus draw values from the same domain and range, non-data lines might have natural language form values that are of arbitrary value lengths.
- DerivedCoverage counts the number of numeric cells that are recognized as derived cells by the derived cell detection Algorithm 2 in the next section. The feature is normalized by the number of numeric cells in this line.

Note that these line classification features are all local features, i.e., describing the characteristics of individual lines. We have tested a few global features that reflect properties of the entire file, namely percentage of empty lines in a file, width and length of a file, and the number of empty line blocks in a file. However, our experiments show no positive impact on the classification problem.

All features are normalized and passed to a random forest classifier that predicts one class for each line. When used as the Line class probability feature in *Strudel*^{C} (Section 5.4), the output is a set of vectors, each of which stands for a probability vector of all classes for a line.

5 CELL CLASSIFICATION

Our cell classification approach *Strudel*^{C} is, like *Strudel*^{L}, based on a multi-class random forest classifier. For the input of this classification task, we have again constructed a set of features that include both the effective ones from previous work, and novel ones. The predictions for line classes are used as a set of features in *Strudel*^{C}. Therefore, the *Strudel*^{L} approach is executed beforehand to obtain the line prediction probabilities that are then transformed into the features of *Strudel*^{C}. We leave the detailed description to Section 5.4.

5.1 Feature extraction

Previous work has proven the effectiveness of content features, stylistic features, spreadsheet formula features, and contextual features [18, 19]. We ignore spreadsheet-specific formula features and stylistic features, as they cannot be constructed from verbose CSV files. Table 2 lists all features involved in our approach, which fall into three groups: content, contextual, and computational features.

Table 2: Cell classification features; '∗' marks contextual features applied to each of the eight surrounding cells of the inspected cell; '†' marks features from related work.

Category	Feature	Value
Content	ValueLength†	[0.0, 1.0]
	DataType†	[0..4]
	HasDerivedKeywords†	0/1
	RowHasDerivedKeywords†	0/1
	ColumnHasDerivedKeywords†	0/1
	RowPosition†	[0.0, 1.0]
	ColumnPosition†	[0.0, 1.0]
	LineClassProbability	$(p_1, ..., p_6)$
Contextual	IsEmptyRowBefore	0/1
	IsEmptyRowAfter	0/1
	IsEmptyColumnLeft	0/1
	IsEmptyColumnRight	0/1
	RowEmptyCellRatio†	[0.0, 1.0]
	ColumnEmptyCellRatio†	[0.0, 1.0]
	BlockSize	[0.0, 1.0]
	NeighborValueLength*	[0.0, 1.0]
	NeighborDataType*	[0..5]
Computational	IsAggregation	0/1

The features marked with '†' in Table 2 are based on those used in [18, 19]. Some of the original features are integrated in our feature set without modification, such as ValueLength and DataType, while others are adapted to a certain extent. For instance, a Boolean feature used to mark the existence of derived cell keywords is extended to a row or a column (RowHasDerivedKeywords and ColumnHasDerivedKeywords), i.e., whether the row or the column that contains the inspected cell contains any derived cell keywords. Features without '†' are new. ValueLength counts the number of characters in the value of the cell. DataType in this work has four possible values, corresponding to four data types: int, float, string, and date. In the next subsections, we explain the intuition and implementation of the four most sophisticated of them.

5.2 Block size

A verbose CSV file may contain multiple tables in various positions, rather than a single relational table. Apart from tables, a verbose CSV files may contain non-data regions composed of aggregation cells, notes, or metadata cells. In our datasets, non-data regions are usually smaller than tables.

To model this phenomenon, we create for each non-empty cell a BlockSize feature, which is calculated as the size of the connected component that contains this cell. A connected component is composed of a group of connected, non-empty cells. Two cells are connected if they are either vertically or horizontally adjacent to each other, or there is at least one connective path between them. Algorithm 1 describes how the value of this feature is produced for each cell in a given verbose CSV file. It takes all non-empty cells in a table as input, and outputs key-value pairs where keys are these non-empty cells and values are their respective block sizes. To obtain the block size for all non-empty cells, the algorithm employs a depth-first strategy to iterate over all of them in a given file. It starts from a single cell block (line 4-7), and continuously adds adjacent cells to expand the block until no more non-empty adjacent cell can be found (line 8-13). The block size is normalized to $[0, 1]$ by the size of the file (line 14). The algorithm terminates once all cells have been touched. Regarding the complexity of this algorithm, assume there are n non-empty cells in a verbose CSV file. On the one hand, each cell will be visited once and only once, resulting a $O(n)$ complexity. On the other hand, the four directions of a cell are checked once the cell is visited, leading to a $O(4n)$ complexity. Therefore, the overall algorithm complexity is $O(n) + O(4n) = O(n)$.

Algorithm 1: Block size calculation

Input: The set of non-empty cells in a table C
Output: The set of key-value pairs BS from cells to block size

1 $BS \leftarrow \{\}$;
2 $V \leftarrow \{\}$ # *visited cells* ;
3 **while** $C - V \neq \emptyset$ **do**
4 $\quad c \leftarrow$ random cell in $C - V$;
5 $\quad bs \leftarrow 1$;
6 $\quad V \leftarrow V \cup c$;
7 $\quad B \leftarrow \{c\}$;
8 $\quad$ **while** *there exist cells in* $C - V$ *adjacent to* B **do**
9 $\quad\quad c_{adj} \leftarrow$ an adjacent cell in C;
10 $\quad\quad bs \leftarrow bs + 1$;
11 $\quad\quad V \leftarrow V \cup c_{adj}$;
12 $\quad\quad B \leftarrow B \cup \{c_{adj}\}$;
13 $\quad$ **end**
14 $\quad bs \leftarrow$ normalize(bs);
15 $\quad$ **foreach** $c \in B$ **do**
16 $\quad\quad BS \leftarrow BS \cup \{c : bs\}$;
17 $\quad$ **end**
18 **end**
19 **return** BS

5.3 Neighbor profile

Cells of some classes may be likely to have particular kinds of neighboring cells. For example, to highlight group header cells, users often separate them from other cells with empty cells, or they place derived cells at the margin of a table, as a way to summarize data. These observations bring our focus on the adjacency context of a cell: for each cell, we gather the data types and value lengths of all eight surrounding cells and present each as a single feature in the feature vector. The neighbor profile of a cell includes all these NeighborValueLength and NeighborDataType features. For the cells on the margins of a file, some adjacent cells do not exist. We set a default value for these non-existent adjacent cells, i.e., -1 for value length and data type.

5.4 Line class probability

Despite the possible flexible layout, verbose CSV files are usually organized in some structurally meaningful way. Lines tend to organize mostly homogeneous types of cells to ease human understanding. For example, a data line contains mostly data cells, while a header line contains mostly header cells. Table 3 displays statistics about the *cell class diversity degree* of all lines in our datasets. The cell class diversity degree of a line is its number of distinct non-empty cell classes. We observe that most lines a have diversity degree of 1: for the cells in these lines, their classes are trivially determined by the class of the line. Therefore, when determining the class of a cell, the class of the line it is located in

is likely a useful feature. In fact, we use this feature alone as one of our baselines.

Table 3: Percentage of lines under different diversity degrees.

	Diversity degree				
Dataset	1	2	3	4	5
SAUS	86.3%	13.7%	0%	0%	0%
CIUS	88.7%	11.2%	0.1%	0%	0%
DeEx	95.3%	4.6%	0.1%	0%	0%

To obtain the line class information, we first run $Strudel^L$ to obtain the prediction for each line. The result of this execution is, however, a probability vector of all classes, instead of a single predicted class. We interpret this probability vector as the classifier's confidence for these classes. Each element of the 6-dimensional vector accounts for a feature for the cell class detection.

5.5 Derived cell detection

If a cell is indeed a derived cell, it should be possible to derive its value by aggregating values of some other neighboring cells. This fact has not been considered by previous work, possibly due to computational cost. We propose a derived cell detection algorithm that seeks to identify derived cells by arithmetically correlating their values with other numeric cells.

We made three observations while investigating the datasets: (i) a derived cell usually aggregates the values of cells from its own row or column; (ii) a derived cell tends to aggregate values close to it; (iii) sum and mean are the two dominant aggregation functions used in verbose CSV files. We integrate these insights into our Algorithm 2. For conciseness, it shows only the approach for detecting derived sum cells.

The algorithm takes as input a table as a two dimensional array, the derived keyword dictionary to look for anchoring cells, an aggregation delta d to give some slack to aggregation results, and a coverage threshold c that controls the generality of aggregation results. Executing the algorithm produces all detected derived cells.

We first determine derived cell candidates, as calculating all aggregation possibilities for all numeric cells is prohibitively expensive. We found that some indicative words usually appear in a cell in the same row or column where there exist many derived cells. For example, for a row with many summing cells, words such as 'Total' are likely to appear in a cell of this row. Therefore, we mark those cells with any of our aggregation keywords (introduced in Section 4) as anchoring cells (line 2). Based on our first observation, only numeric cells in the same row or column as an anchoring cell are treated as derived cell candidates (line 6-8).

For the candidates in the same row as the anchoring cell, the algorithm looks first upwards and then downwards for possible aggregating relationships (line 9-19), whereas for the candidates in the same column as the anchoring cell, the algorithm looks left or right (line 20-30). When looking upwards, the algorithm adds numeric values of a row each time to the sum vector correspondingly, and inspects whether the current sum vector is element-wise close enough (according to d) to the candidates. If the coverage of the close enough elements in the sum vector surpasses c, the candidate is treated as a derived cell (line 14-17). Due to our second observation, a row closer to the row where the candidates are is inspected earlier than a row farther away.

Algorithm 2: Derived cell detection

Input: Table T, keywords K, aggregation delta d, coverage c
Output: All detected derived cell C_D

1 $C_D \leftarrow \{\}$;
2 $A \leftarrow$getAnchoringCells(T, K);
3 **if** *A is empty* **then**
4 **return** C_D;
5 **foreach** *a in A* **do**
6 $i_a, j_a \leftarrow$ row index of a, column index of a;
7 $C_R, cc_{ind} \leftarrow$ the list of numeric cells in row i_a and their column indices;
8 $C_C, rc_{ind} \leftarrow$ the list of numeric cells in column j_a and their row indices;
  /* line 9-19 for upwards detection */
9 $sum \leftarrow (0...0)$;
10 **for** $i = 1$ **to** ∞ **do**
11 **if** $i_a - i < 0$ **then**
12 **break**;
13 **else**
14 $v_o \leftarrow$ numeric values at cc_{ind} in row $(i_a - i)$;
15 $sum \leftarrow sum + v_o$;
16 **if** *coverage of* $(C_R - sum < d) > c$ **then**
17 $C_D \leftarrow C_D \cup C_R$
18 **end**
19 **end**
  /* repeat line 9-19 for downwards detection */
  /* line 20-30 for leftwards detection */
20 $sum \leftarrow (0...0)$;
21 **for** $i = 1$ **to** ∞ **do**
22 **if** $j_a - i < 0$ **then**
23 **break**;
24 **else**
25 $v_o \leftarrow$ numeric values at rc_{ind} in column $(j_a - i)$;
26 $sum \leftarrow sum + v_o$;
27 **if** *coverage of* $(C_C - sum < d) > c$ **then**
28 $C_D \leftarrow C_D \cup C_C$
29 **end**
30 **end**
  /* repeat line 20-30 for rightwards detection */
31 **end**
32 **return** C_D

6 EVALUATION

In this section, we first describe the datasets and all algorithms used in our experiments. After that, we present our experimental evaluation on Strudel, including its comparison with referenced approaches, analysis of feature importance and four advanced features on cell classification, and difficult case study for both line and cell classification tasks.

6.1 Datasets and experimental setup

This section first lists the datasets used in our evaluations, and the preprocessing steps applied thereon. In practice, verbose CSV files may have unique dialects. The dialect of a file specifies the delimiter, quoting character, and escape character, enabling to parse the lines and cells correctly. Therefore, as a general preprocessing, we first applied dialect detection on each file with the approach of van den Burg et al. [27]. This approach takes a text file as input, and produces its detected dialect. The scope of each cell or line is determined by that dialect. The second part of this section describes the list of baseline and competing approaches,

our Strudel approach, and their respective configurations for evaluation.

6.1.1 Datasets. Our datasets with verbose CSV files come from various sources, as summarized in Table 4. Only non-empty lines and cells are counted. The content of our datasets is given in the English language and follows a top-bottom / left-right organisation. Non-Western verbose CSV files might organize the content in a different fashion, which could be an interesting future work.

We created the dataset *GovUK* by crawling all data files in Microsoft Excel format (both in .xls and .xlsx) from an open data portal[4] and transforming them into corresponding CSV format with the Apache POI library[5]. While converting them to CSV files, we omitted both files that contain macros or were not otherwise processable by the library and empty sheets. We randomly selected a subset of 226 files from the dataset, and created a line-level ground truth for them. To perform the actual annotation, we implemented a tool to annotate each line as one of our element classes. Each line of each file in the created dataset was annotated by three human experts. In case of disagreement, which affected only 1% of the annotated lines, we used majority-vote to determine the annotation. For the lines with complete disagreement (fewer than 250 lines in our dataset), we employed an independent fourth annotator to determine which one of the three answers to apply. In the end, we obtained the ground truth for in total more than 110,000 annotated lines. We make all datasets publicly available[6].

Table 4: The dataset overview.

Dataset	# files	# lines	# cells
GovUK	226	97,212	1,382,704
SAUS	223	11,598	157,767
CIUS	269	34,556	367,172
DeEx	444	77,852	784,229
Mendeley	62	195,598	1,359,810
Troy	200	4,348	23,077

Three other datasets *SAUS*, *CIUS*, and *DeEx* were created and annotated by Ghasemi-Go et al. for cell classification [18]. The first two are administrative datasets, while the last one is a business dataset. More detailed descriptions of each dataset can be found in their original paper. The datasets were annotated by the original authors with a slightly different taxonomy. To reconcile their annotations to ours, we partly re-annotate their labels. In summary, they annotated all left-most headers of a table as *attributes*, while we consider them as data of their columns. In the example of Figure 1, they treat the cells that indicate the drug types in the second column as *attributes*, while we annotate them as *data*, as we model them as a data column of the table without a header. We also note that some clearly derived cells were marked as data: understandable errors due to their similarity, which we corrected. In many cases, derived cells form an entire line, with the exception of the leading cell, which is usually textual. This textual cell often includes keywords, such as 'Total', and is neither a derived cell nor a header cell. We treat it as *group* in our system, because a derived line often serves as a section separator in a table. Table 5 presents the class distribution of these three datasets with the reforged annotations.

Table 5: The number of lines or cells per class in the dataset *SAUS*, *CIUS*, and *DeEx* as a whole.

class	# lines	# cells	# cells per line
metadata	2,213	2,479	1.12
header	2,232	19,047	8.53
group	1,767	6,143	3.48
data	114,354	1,202,058	10.51
derived	1,406	76,996	54.76
notes	2,036	2,445	1.20
Overall	124,006	1,309,168	-

Our *Mendeley* data is a set of plain-text files collected from Mendeley's data sharing platform[7] of experimental data. These data are stored in research projects. We crawled all 2,214 projects whose data are stored on Mendeley's own server and that contain at least one plain-text file, i.e., whose MIME type is "text/*". This MIME type corresponds to a wide variety of actual file formats: not only files with table structures and verbose information, such as verbose CSV files, but also programming scripts, HTML pages, etc. We randomly selected 100 projects that include at least one suitable verbose text file, and obtain one such file from each project. Given the intricate dialects of these plain-text files, the dialect detection approach of [27] cannot reliably discover the correct dialect for all files. A file is *parse-able* if the dialect for the table region (including header, data, group, and derived) is correct. For our experiments, we kept the 62 parse-able verbose CSV files. Note that this dataset is used only to verify the performance of our approach on verbose plain-text files and is not part of. the training set. We observe a high line-to-file and cell-to-file ratio, because the files of this dataset are mostly used to store data, e.g., experimental results, rather than presenting statistical tables.

The last dataset, *Troy*, contains 200 CSV files collected from various government websites [17]. Embley et al. used this dataset in their work to convert different statistical tables to relational tuples [15]. We kept the dataset unseen during the design of Strudel to test the out-of-domain generalizability of our approach with this dataset.

In our data preparation process, we cropped each file by removing the marginal empty lines or columns, as some of our features are sensitive to the number of empty cells in the lines, and leading/trailing empty lines are trivial cases. Values of spanning cells in original spreadsheets are copied only to the top-left cell in the CSV file, instead of to all covered cells for two reasons: (i) the top-left is well-defined for all shapes of spanning cells and (ii) copying the values to all covered cells creates too many repeated characters, confusing the models that cause unnecessary over-fittings towards these values. To ease future study on this topic, we will publish all datasets and their annotations.

6.1.2 Setup of experiments. The list below contains all algorithms used in the evaluation, along with their corresponding configurations. All algorithms were implemented in Python with the scikit-learn library[8]. The superscript in the name of an algorithm indicates the type of elements detected by this algorithm, i.e., 'L' and 'C' represent line and cell classification, respectively.

[4] https://data.gov.uk/
[5] http://poi.apache.org/index.html
[6] https://hpi.de/naumann/projects/data-preparation.html
[7] https://data.mendeley.com/, last crawled on 3. August 2020
[8] https://scikit-learn.org/stable/index.html

Table 6: Per-class and overall F-1 score on each dataset for line classification (top) and cell classification (bottom).

		metadata	header	group	data	derived	notes	accuracy	macro-avg
GovUK	CRF^L	**.789**	.379	.898	**.991**	.339	.752	**.979**	.733
	$Pytheas^L$	.446	.444	.172	.986	-	.545	.970	.518
	$Strudel^L$	.670	**.774**	**.919**	.989	**.361**	**.797**	.978	**.751**
	# lines	*878*	*519*	*850*	*93,584*	*665*	*716*	-	-
SAUS	CRF^L	.893	.651	.817	.963	.477	.980	.931	.797
	$Pytheas^L$	.884	.768	.741	.973	-	.814	.944	.836
	$Strudel^L$	**.984**	**.960**	**.882**	**.987**	**.599**	**.984**	**.976**	**.899**
	# lines	*469*	*565*	*289*	*9,346*	*279*	*650*	-	-
CIUS	CRF^L	**.994**	.961	**.992**	**.996**	.749	**.988**	.992	.947
	$Pytheas^L$	.988	.867	.000	.970	-	.637	.943	.692
	$Strudel^L$	**.994**	**.972**	.984	**.996**	**.834**	.978	**.993**	**.960**
	# lines	*1,034*	*435*	*1,074*	*30,890*	*449*	*674*	-	-
DeEx	CRF^L	.753	.373	.027	.970	.244	.480	.942	.475
	$Pytheas^L$	.564	.406	.137	.980	-	.433	.957	.420
	$Strudel^L$	**.797**	**.807**	**.357**	**.989**	**.548**	**.761**	**.976**	**.710**
	# lines	*710*	*1,299*	*407*	*74,116*	*678*	*712*	-	-

		metadata	header	group	data	derived	notes	accuracy	macro-avg
SAUS	$Line^C$	.963	.915	.451	.970	.332	.888	.930	.753
	RNN^C	.977	.925	.466	.956	.345	.902	.919	.762
	$Strudel^C$	**.987**	**.972**	**.752**	**.983**	**.689**	**.957**	**.968**	**.890**
	# cells	*469*	*4,769*	*825*	*142,301*	*8,708*	*695*	-	-
CIUS	$Line^C$	.991	.973	.361	.929	.156	.937	.824	.725
	RNN^C	.987	.976	.679	.904	.443	.963	.850	.825
	$Strudel^C$	**.993**	**.993**	**.916**	**.946**	**.465**	**.989**	**.895**	**.884**
	# cells	*1,035*	*3,838*	*4,228*	*310,354*	*47,043*	*674*	-	-
DeEx	$Line^C$	.630	.625	.155	.981	.258	.520	.955	.528
	RNN^C	.623	.772	.347	.952	.244	.413	.930	.559
	$Strudel^C$	**.689**	**.801**	**.444**	**.988**	**.683**	**.598**	**.977**	**.700**
	# cells	*975*	*10,314*	*1,216*	*749,403*	*21,245*	*1,076*	-	-

- CRF^L is a conditional random field-based learning approach dedicated to line classification from the work of Adelfio et al. [2] as the current state of the art. We applied this approach with the logarithmic binning technique introduced by the authors, as this setting was reported to gain the best performance.
- $Pytheas^L$ is a rule-based approach that discover the locations of tables, and further classifies the lines in CSV files [8]. We use the parameter values introduced in the original paper for our experiments.
- $Strudel^L$ is our proposed approach for line classification. The underlying random forest classifier used the default settings in the scikit-learn library.
- $Line^C$ is a baseline approach for cell classification. This approach simply extends the predicted class of a line from the result of a $Strudel^L$ execution to each non-empty cell in this line.
- RNN^C is based on the state of the art by Ghasemi-Gol et al., which classifies cell types with a recursive neural network using pre-trained cell embeddings [18]. For our experiment, we used the same settings as introduced in the original paper.
- $Strudel^C$ is our approach for cell classification. Again, we used the default settings of the random forest classifier in the scikit-learn library. In our experiment, we do not observe a substantial difference in the result with different values of the aggregation delta d and coverage c. We set them to 0.1 and 0.5, respectively.

Apart from using content and spatial features, both original CRF^L and RNN^C applied stylistic or spreadsheet formula features. Because the input data in our use-case are style-less, verbose CSV files, we remove all stylistic features from the two approaches so as to conduct fair comparisons. Each algorithm is evaluated using 10-fold cross validation. When creating the folds, our process ensures that all elements from a single file appear in either the training or the test set. We repeat the 10-fold cross validation ten times to reduce bias leaning to particular fold splits. The results of all repetitions are averaged to obtain the final score.

We have tested several classification algorithms for Strudel, including Naïve Bayes, KNN, SVM, and random forest. Random forest consistently outperformed the other candidate algorithms on our datasets for both classification tasks. Therefore, we chose it as the backbone supervised learning algorithm of Strudel. The advantage of random forest over the other algorithms is that it reduces the risk of over-fitting by considering the results from multiple base classifiers, which is crucial for unbalanced datasets, such as verbose CSV files.

6.2 Comparative evaluation

This section presents the comparative evaluation results between Strudel and the referenced approaches. We use the F1 measure to evaluate the classification correctness of each approach for both line and cell classification tasks. When comparing the overall result amongst algorithms, we focus on the macro average, which does not weigh the average score with the support of individual classes, thus avoiding the bias from the number of

per-class instances, which is crucial for supervised learning tasks on imbalanced data.

6.2.1 Line classification. We compared *Strudel*^{L} with *CRF*^{L} and *Pytheas*^{L}. *CRF*^{L} uses a set of features, including content features, contextual features, and stylistic features to train a conditional random field based classifier on web tables and spreadsheets. *Pytheas*^{L} uses a number of weighted rules to decide whether a line is data or non-data. The binary results are used to draw the table top/bottom boundaries, on top of which the approach utilizes some additional rules to determine line classes.

Table 6 (top) reports the per-class and macro-average F1 scores, and accuracy for the three approaches. Note that *Pytheas*^{L} can classify a line as one of only five classes that correspond respectively to ours, missing the derived class. Therefore, when calculating the measurements for this approach, we leave out the derived lines from our datasets. Overall, our approach leads on macro-average for all datasets. *Pytheas*^{L} does not perform well in general on the minority classes in all but the *SAUS* datasets, as its proposed rules are not suitable for these datasets: they produce poor results already for the binary data/non-data classification, which disrupts the subsequent table discovery and line classification. Group lines are particularly difficult for *Pytheas*^{L}: the scope of group lines is constrained to lines between data lines and has only the leftmost cell non-empty. While the group lines in *SAUS* mostly follow this definition, those in the other datasets do not. Most header lines of both *SAUS* and *CIUS* are across few lines and with simple structures. Therefore, recognizing those headers correctly is easier. Since the rule used to determine metadata lines is dependent only on the positions of headers, it is also easier to recognize metadata in these two datasets.

For *CRF*^{L} and *Strudel*^{L}, classifying header, group, derived and notes correctly is in general more challenging, compared to metadata and data, according to the per-class scores. Both algorithms perform better on *CIUS* than on the other datasets, because many files in this dataset are essentially the reports from different years on the same themes with the same templates – there are few file structure outliers. Both approaches do not work well on *derived* in *SAUS*, because the dataset has many unanchored derived cells. *GovUK* and *DeEx* are difficult to both approaches, because they both have many heterogeneous files regarding their structures.

In summary, *Strudel*^{L} outperforms *CRF*^{L} without using its stylistic features on our datasets, showing that our approach is more effective when fewer assumptions can be made about the input. *Strudel*^{L} is also more flexible than rule-based approaches such as *Pytheas*^{L} in predicting cases that are not covered by the given rule set.

6.2.2 Cell classification. For the cell classification task, we compare *Strudel*^{C} with two aforementioned algorithms: (i) *Line*^{C} provides a reasonable baseline, as most lines have homogeneous cells; (ii) *RNN*^{C} is based on an advanced deep learning architecture. The authors of *RNN*^{C} evaluated their approach also without stylistic features, which allows a fair comparison to *Strudel*^{C}. Table 6 (bottom) summarizes the comparative result in terms of the per-class and macro-average F1 score, and accuracy. *Strudel*^{C} surpasses its competitors . Meanwhile, the macro-average of *RNN*^{C} shows an advantage against the baseline approach, although the per-class scores of the two approaches are on par with each other. Even though cell classification is a more imbalanced task than line classification, the performance of our *Strudel*^{C} approach is comparable to its line counterpart.

Similar to the line classification problem, metadata, group, derived and notes are the difficult classes in general. *Group* cells are challenging for all approaches, as cells of this class are particularly rare. However, unlike other rare classes, such as metadata and notes, group cells are more likely to co-occur in the same line with data cells. *Line*^{C} reported low F1 score particularly on group and derived cells across datasets. In fact, both group and derived cells often co-occur with other types in the same lines: some tables contain a group cell in a line with several derived cells; other tables have derived columns rather than lines, therefore causing the few derived cells in a same line with multiple data cells. Group and derived cells usually account for a minor amount in these cases.

Line^C applies a majority-take-all strategy to extend the line prediction result of a line to all its non-empty cells, and therefore causes false negative for group and derived cells in the above two cases. *RNN*^{C} shows a low F1 score on the group class, which is not considered in the original paper [18], showing that the approach cannot be directly adapted to it. The set of the reforged derived cells, many of which were misplaced in the original annotations, is also troublesome for *RNN*^{C}, which does not involve value calculation mechanisms to detect them.

6.3 Strudel performance evaluation

In this section, we present experimental results to gain insights on the following questions: (i) When does Strudel mis-classify an instance of a particular class, and which class is most likely to be considered? (ii) Whether our approaches generalize to plain-text files that do not stem from spreadsheets? (iii) How do the features exploited in this work affect performance? (iv) What are the typical reasons that cause these incorrect predictions?

6.3.1 Line classification. Table 6 has introduced the per-class and overall F1 results of *Strudel*^{L}. In this section, we present our analysis of the classification results by using the confusion matrix. Figure 3 (top) shows the confusion matrix on executing *Strudel*^{L} per dataset. Due to space limitations, we leave out the matrix for *SAUS*, which is very similar to that for *GovUK*. To create a confusion matrix with the repeated 10-fold cross validation setting, we concatenate for each line in the files the predictions of all repetitions, and construct an ensemble prediction for it with the majority voting strategy. To resolve possible ties, we stipulate that the fewer instances of a class included in the dataset, the more prior the class is. We normalize the confusion matrix by the number of instances with particular classes.

Correctly identifying derived lines is the most challenging task across all datasets. These lines are mostly misclassified as data. The two main reasons for this are the lack of derived line training instances and the high similarity between derived and data lines, w.r.t. data types and spatial characteristics. Around 11.4% of the derived lines are treated as headers for *GovUK*. We observed this to happen in many tables where derived lines are between header and data areas and separated from these two areas by empty lines. Note that when a line of a minority (non-data) class is misclassified, the wrong prediction tends to be 'data', as the data class has much more instances than any other class. Apart from derived lines, header, group, and notes lines in *DeEx* also incorrectly lean towards the data class, because this dataset contains many tables of complicated structures. We discuss the kinds of mistakes in these categories in Section 6.3.6.

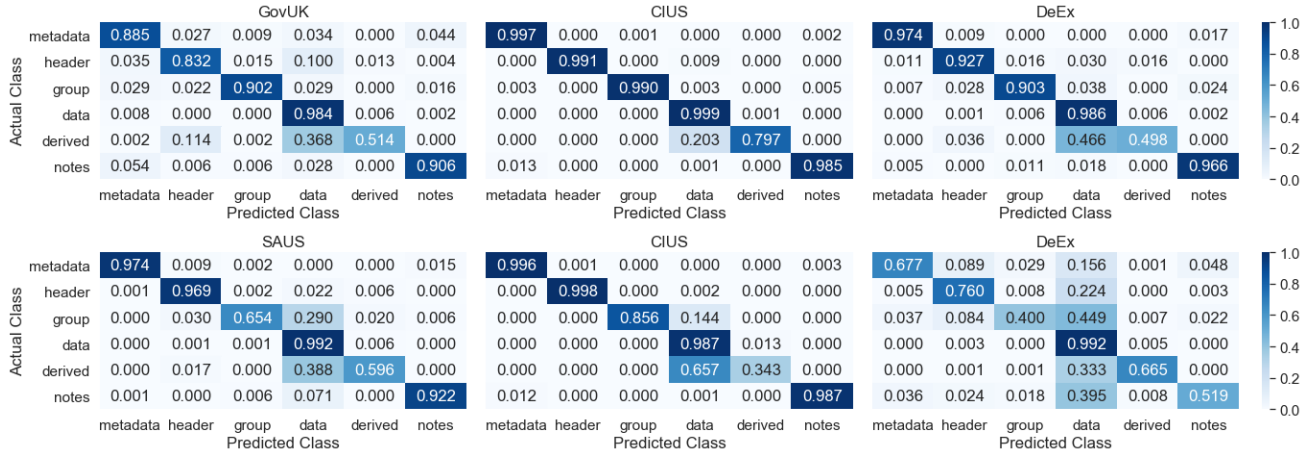

Figure 3: Confusion matrices to describe the pair-wise performance of *Strudel*^{L} (top) and *Strudel*^{C} (bottom) on individual datasets. The numbers are normalized by the amount of instances per class.

6.3.2 Cell classification. Similar to line classification experiments, we performed a set of experiments for evaluating *Strudel*^{C} and show the results in this section. Figure 3 (bottom) depicts the per-dataset confusion matrix on executing *Strudel*^{C}. To create it, we applied on the repeated cross validation results the same procedure used to create the confusion matrix for *Strudel*^{L}.

Compared to the confusion matrix of *Strudel*^{L}, more classes have a higher mis-classification ratio for all three datasets, showing that cell classification is a more challenging task than its line counterpart. On the one hand, the tendency to mark the instances of the minority classes as data is still prevalent. On the other hand, as the complexity of the problem increases, we do not observe many classification errors between two non-data classes, showing the effectiveness of our approach to distinguish between pairs of elements belonging to minority classes. About two-thirds of the derived cells are treated as data for *CIUS*. This is because a number of files share a fixed table schema that uses no keywords to indicate derived columns. Therefore, they are effectively ignored by the derived cell detection component.

6.3.3 Out-of-domain classification performance. To test the out-of-domain classification performance of Strudel, we kept the *Troy* dataset unseen during its design. We used a model trained on the collection of *SAUS*, *CIUS*, *DeEx* datasets to predict the line and cell classes of each file in this dataset.

The results in Table 7 show that group and derived cells are challenging for Strudel. After inspection, we found out that most of the derived cells lay in the lines that do not contain any derived keyword, such as 'total'. These derived cells are excluded from the candidate set, because our derived cell detection algorithm (Algorithm 2) relies on these keywords to anchor the candidates in order to reduce the search space. A typical derived line contains few group cells (usually the left-most) with indicative strings, such as 'total' and a number of numerical derived cells. Many of these derived lines are mis-classified as data, leading to the group cells therein also being mistaken.

6.3.4 Performance on plain-text files. To verify the effects of our Strudel algorithms on more difficult plain-text files that are not converted from spreadsheets, we tested the Strudel algorithm on the *Mendeley* dataset. We trained a model of our algorithm on the collection of *SAUS*, *CIUS*, and *DeEx* datasets, using the whole *Mendeley* dataset for testing.

Table 7: Per-class and overall F1 score on the *Troy* dataset.

	Strudel^L	*# lines*	*Strudel*^{C}	*# cells*
metadata	.935	317	.921	321
header	.798	278	.840	1,341
group	.667	42	.232	294
data	.937	2,898	.936	18,600
derived	.070	239	.216	1,935
notes	.971	575	.952	592
macro-avg	.730	4349	.683	23,083

Table 8 displays the per-class and overall F1 score for this experiment. As mentioned above, files of the *Mendeley* dataset are mostly used to store (tabular) data. Therefore, the minority classes in the *Mendeley* dataset have very few instances.

While the overall F1 scores in this experiment are inferior to the respective ones shown in Table 6, they do show that even for such difficult files our approaches are well able to distinguish data from non-data. The values in *Mendeley*'s plain-text files show properties different from traditional spreadsheets, e.g., length of metadata and notes areas, width of files. The second reason is that no data from *Mendeley* was included in the training phase. Therefore, dataset-specific properties are not learned properly by the classifiers. Last but not least, different areas in a plain-text file might have their own delimiters. As the delimiter of the table areas is used across the file, it is possible to destroy the intrinsic structures of other areas, e.g., when using comma as the delimiter, the value of a note line is split across multiple cells.

Regarding the results of individual classes, our model treats quite a few metadata lines as data, as the delimiter of metadata and data areas are different. At times, the delimiter dilemma also confuses our model of header lines in files where these lines are not split correctly. Out of the few derived cells, most are located in a single file, where the derived cells form a table by themselves, and aggregate on the values from another table, which is not recognizable by our model.

As the *Mendeley* dataset holds the biggest files across all our datasets, we also tested the scalability of our approach. The overall runtime on classifying cells of a file includes that of dialect

detection, feature creation, and cell class prediction. Our experiments show that the overall runtime is linear to the file size. For a file of around 10MB, the whole procedure takes around 256s on a 1.4 GHz MacBook Pro with 16GB RAM. Most of the time is spent on creating the feature vectors, which could be easily parallelized if possible. While we have few big files of such size, most files are only several kilobytes, probably because files with verbose information are usually used to show limited key information, rather than store a big amount of data.

Table 8: Per-class and overall F1 score for *Mendeley*.

	Strudel^L	# lines	*Strudel*^{C}	# cells
metadata	.623	604	.245	2,152
header	.406	86	.629	769
group	.263	27	.303	44
data	.999	194,786	.999	1,356,635
derived	.364	9	.051	99
notes	.448	86	.380	111
macro-avg	.517	195,598	.435	1,359810

6.3.5 Feature analysis. To understand which features exert more influence than others on particular classes, we calculated the feature importance for both *Strudel*^{L} and *Strudel*^{C} models. There are a variety of techniques to calculate feature importance [3, 21]. As many of our features are low-cardinality categorical features, we exploited permutation feature importance, because it does not favor high cardinality features [3]. Permutation feature importance indicates the ability of one feature to distinguish instances of one class from those of another in a binary classification scenario. To adapt this metric to our multiclass classification problem, we trained a model for each class in a one-vs.-rest fashion, and use the permutation feature importance of each such binary classifier to represent the ability of our model to detect instances of the particular class. The permutation of each feature was repeated five times and averaged.

Figure 4 illustrates the per-class feature importance for *Strudel*^{L} (top) and *Strudel*^{C} (bottom) with 100% stacked bars. The models are trained on the collection of *SAUS*, *CIUS*, and *DeEx*. We grouped all neighbor profile features (Section 5.3) into neighbor value length and neighbor data type to reduce the complexity of the figure, as each individual feature has little importance on the cell classification task. Up to five most important features whose proportions are higher than 10% are highlighted.

The line type probability feature is the most crucial feature for notes, metadata, and header. The percentage of empty cells in the row is also quite useful for notes and metadata. The percentage of empty cells in a column is most important to discover group cells: many group cells are in the left-most column (also indicated by the importance of the column position feature) of a file and span multiple rows. Neighbor profile features are most useful on discovering group cells, proving that group cells tend to locate in specific places. The novel feature signifying whether the value of a cell is the aggregation of other cells in the same line or column plays a great role in detecting derived cells, proving its effectiveness. Besides that, the existence of derived keywords, such as 'total' in the same column is also important, indicating that users tend to use these words to mark the derived columns. However, the existence of derived keywords in the same line shows quite limited importance in our experiment, although we expected similar importance of it as its column counterpart.

6.3.6 Analysis of difficult cases. The confusion matrices shed light on which classes are most commonly mis-predicted, either in the line or cell classification task. Here, we identify typical causes of those errors. The list below describes the pairs of common mis-classification cases (with > 10% incorrect classification in the class), e.g., mis-classifying 'derived as data', each followed by an error analysis after manually inspecting the results.

- **Derived as data.** Errors of this type usually happen because derived lines without keywords, such as 'total', in any of the cells are ignored by the derived cell detection algorithm, which uses these words to determine candidates, or because derived lines aggregate values from non-consecutive lines, which are ignored by the detection algorithm.
- **Header as data.** A header line with a number of non-textual values adjacent to a data line may be mis-classified as part of the data area. Examples include numeric headers, such as year and date. Files with multiple vertically-stacked tables may also be affected by this sort of error, as the headers of the tables towards the bottom of the stack have unusual line positions.
- **Notes as data.** Organizing notes as a small table is not uncommon, particularly in *DeEx*. Therefore, these tables of notes are likely to be treated as data. In some cases, authors place notes to the right of a table. Therefore, they are likely to be treated as data areas during cell classification.
- **Group as data.** One reason for this type of error is that some files have multi-level group columns, such as 'country-state-city', to the left of a table, followed by a number of data columns to the right. As most tables have few group columns, the classifier may mis-interpret these rare cases as data. Another reason is that these group cells lay in the same lines as those derived cells, who are not captured by the derived cell detection algorithm, because there is no keyword in the same row or column.
- **Metadata as data.** Tables may have elaborate metadata organized as small tables. Due to the tabular features of these metadata tables, Strudel tends to interpret them as data cells.

In summary, there are three aspects that mostly affect the correctness of our approach: (i) the geographical characteristic of vertically stacked multi-table files; (ii) the arithmetic calculation method for derived lines; (iii) the similarity between numeric header lines and data lines. These facts offer directions for improving our approach in the future work.

7 CONCLUSIONS

Often, valuable data are stored in semi-structured documents and cannot be directly parsed by common data management tools. Prior to extracting information from these files, it is necessary to understand their structure, by means of element classification, at either line or cell level. Previous works have addressed the line or cell classification problem for style-enriched documents, such as web tables or spreadsheets. In this work, we address both tasks on verbose CSV files that, similar to spreadsheets, organize data in a flexible layout, yet lack rich-text features. We addressed the two classification problems separately and designed a set of features for each of them, including content features, contextual features, and computational features.

Based on the experimental evidence, we discovered that with well-designed features, it is possible to reach decent performance of classifying lines and cells in a verbose CSV file and spreadsheets even if the stylistic features are not available. To conduct fair comparison between Strudel and related work, we use only the non-stylistic features. We summarize a handful of reasons that

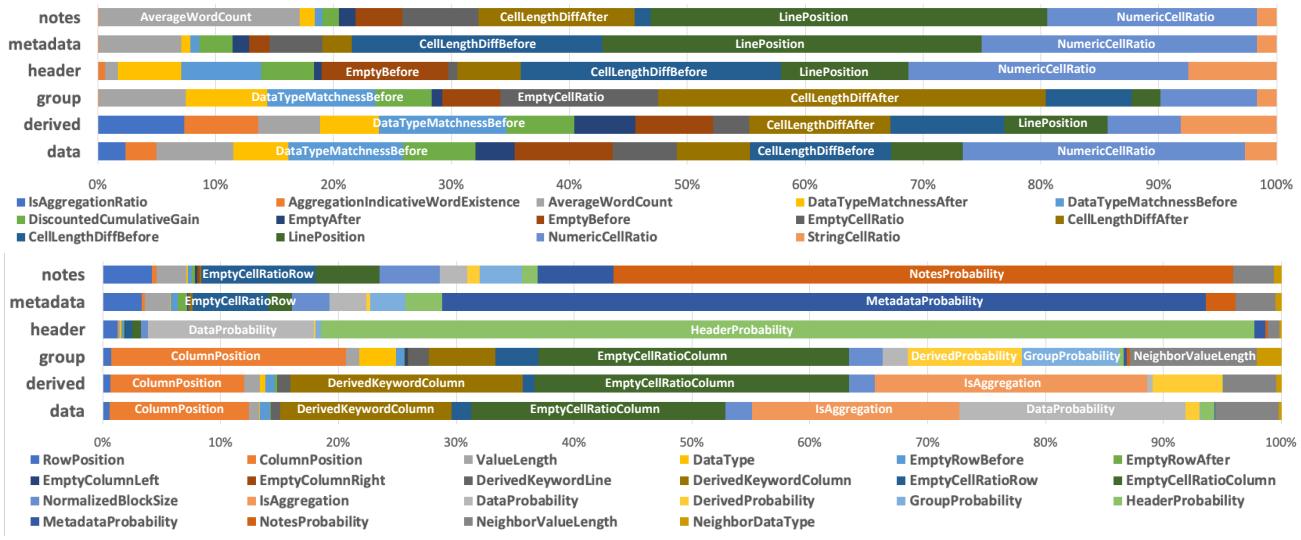

Figure 4: Feature importance of *Strudel*^{L} (top) and *Strudel*^{C} (bottom) trained on the collection of *SAUS*, *CIUS*, and *DeEx*. Several most important features for each class are highlighted.

cause common misclassification cases, and recognize the effectiveness of computational features that are neglected by former studies, drawing key insights for further structure understanding research: (i) how to improve the prediction quality with semantic features; (ii) how can we extend the derived cell detection algorithm by recognizing more aggregation functions; (iii) whether column classification can help boost the classification quality.

REFERENCES

[1] Robin Abraham and Martin Erwig. 2007. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing* 18, 1 (2007), 71–95.

[2] Marco D Adelfio and Hanan Samet. 2013. Schema extraction for tabular data on the web. *PVLDB* 6, 6 (2013), 421–432.

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Michael J Cafarella, Alon Y Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. 2008. Uncovering the Relational Web.. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*.

[5] Zhe Chen and Michael Cafarella. 2013. Automatic web spreadsheet data extraction. In *International Workshop on Semantic Search over the Web*. 1–8.

[6] Zhe Chen, Mike Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. 2013. Senbazuru: A Prototype Spreadsheet Database Management System. *PVLDB* 6, 12 (2013), 1202–1205.

[7] Zhe Chen, Sasha Dadiomov, Richard Wesley, Gang Xiao, Daniel Cory, Michael Cafarella, and Jock Mackinlay. 2017. Spreadsheet property detection with rule-assisted active learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 999–1008.

[8] Christina Christodoulakis, Eric Munson, Moshe Gabel, Angela Demke Brown, and Renée J. Miller. 2020. Pytheas: Pattern-based Table Discovery in CSV Files. *PVLDB* 13, 11 (2020), 2075–2089.

[9] Xu Chu, Yeye He, Kaushik Chakrabarti, and Kris Ganjam. 2015. Tegra: Table extraction by global record alignment. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1713–1728.

[10] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. 2019. Discovering order dependencies through order compatibility. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 409–420.

[11] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. TableSense: Spreadsheet Table Detection with Convolutional Neural Networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 69–76.

[12] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. Tablesense: Spreadsheet table detection with convolutional neural networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Vol. 33. 69–76.

[13] Julian Eberius, Christoper Werner, Maik Thiele, Katrin Braunschweig, Lars Dannecker, and Wolfgang Lehner. 2013. DeExcelerator: a framework for extracting relational data from partially structured documents. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 2477–2480.

[14] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. 2009. Harvesting relational tables from lists on the web. *PVLDB* 2, 1 (2009), 1078–1089.

[15] 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* 19, 2 (2016), 119–138.

[16] Wolfgang Gatterbauer, Paul Bohunsky, Marcus Herzog, Bernhard Krüpl, and Bernhard Pollak. 2007. Towards domain-independent information extraction from web tables. In *Proceedings of the International World Wide Web Conference (WWW)*. 71–80.

[17] George Nagy. 2016. TANGO-DocLab web tables from international statistical sites (Troy_200). http://tc11.cvc.uab.es/datasets/Troy_200_1.

[18] Majid Ghasemi-Gol, Jay Pujara, and Pedro Szekely. 2019. Tabular Cell Classification Using Pre-Trained Cell Embeddings. *Proceedings of the International Conference on Data Mining (ICDM)* (2019).

[19] Elvis Koci, Maik Thiele, Óscar Romero Moral, and Wolfgang Lehner. 2016. A machine learning approach for layout inference in spreadsheets. In *IC3K: Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. SciTePress, 77–88.

[20] Ying Liu, Prasenjit Mitra, and C Lee Giles. 2008. Identifying table boundaries in digital documents via sparse line detection. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 1311–1320.

[21] Wei-Yin Loh. 2011. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1, 1 (2011), 14–23.

[22] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. 2018. Table union search on open data. *PVLDB* 11, 7 (2018), 813–825.

[23] David Pinto, Andrew McCallum, Xing Wei, and W Bruce Croft. 2003. Table extraction using conditional random fields. In *Proceedings of the International Conference on Information retrieval (SIGIR)*. 235–242.

[24] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets.. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 253–264.

[25] Alexey O Shigarov and Andrey A Mikhailov. 2017. Rule-based spreadsheet data transformation from arbitrary to relational tables. *Information Systems (IS)* 71 (2017), 123–136.

[26] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. 2020. Data Curation with Deep Learning.. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 277–286.

[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.

[28] Alexander Wachtel, Michael T Franzen, and Walter F Tichy. 2016. Context Detection in Spreadsheets Based on Automatically Inferred Table Schema. In *International Conference on Human-Computer Interaction*.

[29] Yalin Wang and Jianying Hu. 2002. A machine learning based approach for table detection on the web. In *Proceedings of the International World Wide Web Conference (WWW)*. 242–250.

[30] Meihui Zhang and Kaushik Chakrabarti. 2013. InfoGather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 145–156.