ExtracTable: Extracting Tables from Raw Data Files

Leonardo Hübscher[1] Lan Jiang[2] Felix Naumann[3]

Abstract: Raw data, especially in text-files, comes in many shapes and forms, often tailored toward human readability. They include preambles and footnotes, are formatted visually, and in general do not follow CSV-guidelines. The ability to easily ingest such files into data systems opens up many opportunities for data analysis and processing. With EXTRACTABLE, we present a system that can automatically ingest a large variety of raw data files, including text files and poorly structured CSV-files by detecting row patterns and thus separating their values into coherent columns. We manually annotated 957 files of a wide variety containing 1 208 tables. We show experimentally that ExtracTable can correctly parse 90% of all lines in structured files and 76% of all lines in files with a visual layout only, significantly outperforming state-of-the-art.

1 Table Extraction

As more and more data is created and made accessible, the ability to automatically ingest and analyze them becomes increasingly desirable. Various open data portals are a means for governments, companies, and individuals to make data publicly available. However, to support data creators to easily share their data, these platforms do not enforce specific data formats, and we observe very many home-grown formats that are not amenable to easy parsing and ingesting into a data system.

Data wrangling summarizes the process of transforming raw data into a well-defined format. According to multiple studies from Kaggle, Anaconda, IBM, and Forbes, data scientists spend 26% to 80% of their time on data wrangling, distracting them from tackling the original data processing task [An20; Ch14; Mo18; Pr16]. This effort is not only time-consuming, but also tedious and error-prone. Still, data preparation is necessary, as data quality issues otherwise prevent subsequent algorithms from working well.

Data is often displayed and stored in a tabular format that is suitable for both humans and machines. However, tables may appear quite different when persisted as files. Plain-text files, for instance, lack proper instructions on how to interpret tables therein. Our work regards two table formats: CSV tables and ASCII tables. The widely used CSV (character-separated-values) format was first used by IBM to store tabular data in 1972. However, a formally specified CSV format, which is now known as the RFC 4180 standard, had not been formalized until 33 years later [IB72]. Meanwhile, companies and data practitioners have developed their

[1] Hasso Plattner Institute, University of Potsdam, Germany leonardo.huebscher@student.hpi.de
[2] Hasso Plattner Institute, University of Potsdam, Germany lan.jiang@hpi.de
[3] Hasso Plattner Institute, University of Potsdam, Germany felix.naumann@hpi.de

doi:10.18420/BTW2023-20

```
#   OBIA4RTM config file for setting up Prospect4SAIL
#
#   Typical values (taken from J Gomez-Dans on https://pypi.org/project/prosail/)
#
#   ================================================================================
#   | Parameter    | Description of parameter          | Units        |Typical min | Typical max |
#   |--------------|-----------------------------------|--------------|------------|-------------|
#   |    N         | Leaf structure parameter          | N/A          | 0.8        | 2.5         |
#   |   cab        | Chlorophyll a+b concentration     | ug/cm2       | 0          | 80          |
#   |   caw        | Equivalent water thickiness       | cm           | 0          | 200         |
#   |   car        | Carotenoid concentration          | ug/cm2       | 0          | 20          |
#   |   cbrown     | Brown pigment                     | NA           | 0          | 1           |
#   |   cm         | Dry matter content                | g/cm2        | 0          | 200         |
#   |   lai        | Leaf Area Index                   | N/A          | 0          | 10          |
#   |   lidfa      | Leaf angle distribution           | N/A          | -          | -           |
#   |   lidfb      | Leaf angle distribution           | N/A          | -          | -           |
#   |   psoil      | Dry/Wet soil factor               | N/A          | 0          | 1           |
#   |   rsoil      | Soil brigthness factor            | N/A          | -          | -           |
#   |   hspot      | Hotspot parameter                 | N/A          | -          | -           |
#   |   tts        | Solar zenith angle                | deg          | 0          | 90          |
#   |   tto        | Observer zenith angle             | deg          | 0          | 90          |
#   |   phi        | Relative azimuth angle            | deg          | 0          | 360         |
#   | typelidf     | Leaf angle distribution type      | Integer      | -          | -           |
#   ================================================================================
#
#   You can enter your values below -> make sure not to alter the overall structure of this
#   template -> otherwise bad things might happen
#
#   Further Explainations:
#
#   min: Minimum Value of Parameter
#   max: Maximum Value of Parameter (in case min=max, the parameter will not be retrieved)
#   num: in case min!=max, the number of samples to be drawn for the specific parameter
#   dist: which statistical distribution of values should be used for drawing the samples (igno
#       1: truncated Gaussian (between min and max)
#       2: uniform distribution (between min and max)
#       0: non-applicable
#   mean: mean in case of truncated Gaussian distribution
#   std: in case of truncated Gaussian standard deviation of parameter for drawing the samples
#
#   min max num dist mean std comment
min      max      num      dist     mean     std      comment
1.8      1.8      1        0        1.5      0        N
20       60       40       1        40       15       cab
 0       0        1        0        0        0        car
 0       1        10       2        0        0        cbrown
 0.01    0.01     1        0        0        0         cw
 0.009   0.009    1        0        0        0        cm
 0.2     7        40       1        4        2.5      lai
-0.35    -0.35    1        2        0        0         lidfa
-0.15    -0.15    1        0        0        0        lidfb
 0.5     0.5      1        0        0        0        rsoil
 0.2     0.2      1        0        0        0        psoil
 0.01    0.01     1        0        0        0        hspot
27.947   27.947   1        0        0        0         tts
7.04345  7.04345  1        0        0        0        tto
146.691  146.691  1        0        0        0        psi
1        1        1        0        0        0         typelidf
```

Fig. 1: A real-world plain-text file including two tables in different formats (framed in blue) taken from the Mendeley data portal (doi: 10.17632/vs55cwssyh.2#file-54e4f7c2-0156-4be8-9960-d95b0ba0f940)

own formats that use different utility characters, such as "|" as delimiters, which deviate from the specification. Unfortunately, the RFC formalization does not account for such variations. Our previous work recognizes table regions in CSV files with visual features based on different cell data types [VJN21]. To use this approach, however, one must first identify cells. ASCII tables are another type of plain-text data format used to deposit data. Unlike CSV tables that structure data with particular utility characters, ASCII tables merely store characters, leaving the interpretation of file structures to users. The existence of customized file structures forces data scientists to take care of each file individually.

Figure 1 shows the content of a single real-world file with two tables. While the first table uses special characters, such as "|", "=", and "-" to frame the header row and different columns, the second table uses whitespace regions to separate columns. To facilitate human readability, columns in the two tables visually align their values by using different numbers of utility characters as field separators. There are also texts before or after tables that typically deliver contextual information, such as experimental setups or sensor information. Texts might be misinterpreted as structured data when they contain table-like elements. Due to the ad-hoc shapes of tables, common commercial tools fail to load them correctly [HN20].

We propose the ExtracTable algorithm for automatic table extraction from plain-text files, which takes all the aforementioned file varieties into consideration. Given a file, ExtracTable first detects its structure interpretation and uses it to interpret structures of its lines. Then, the algorithm extracts value patterns of the interpreted lines and builds table candidates with the optimal pattern consistency. Finally, a subset of table candidates are selected as the output tables. Our approach makes the following contributions:

1. A set of 957 annotated raw data files selected from a variety of sources, totaling 1 208 tables across all files.
2. The ExtracTable approach, which detects column and row patterns in data, and ultimately extracts table elements from raw data files.
3. A detailed experimental evaluation, also comparing to multiple CSV parsing tools and the Pytheas system [Ch20]

To encourage further research on this topic, we have published all annotated data and the code[4]. We organize the rest of this paper as follows: Section 2 summarizes related work. We formalize the terms used in this work and the table extraction problem in Section 3. We elaborate on the proposed ExtracTable algorithm in Section 4, and present the results of a series of experiments in Section 5. Finally, we conclude the paper and point out future work in Section 6.

[4] https://github.com/HPI-Information-Systems/ExtracTable

2 Related Work

The Pytheas system addresses the problem of table discovery in CSV files using a set of weighted fuzzy rules that exploit column patterns [Ch20]. The weights were trained on a dataset collected from open data portals containing governmental data. The paper focuses on table discovery and row classification While the authors optimized their approach for CSV tables, Pytheas could also be applied to tables in ASCII files if adapted accordingly. A limitation of this approach is that input files must have been parsed properly, which the authors conduct with the standard Sniffer module of Pythons CSV library in a pre-processing step. In comparison, our approach can parse raw files automatically before detecting tables and classifying rows. We use Pytheas as a baseline for table range detection.

Pyreddy and Croft propose an approach to detect text lines containing tabular structures represented in ASCII [PC97]. A followup work improves the line classification step [Pi03]. From their work, we learn that whitespace alignments in continuous lines are important for ASCII tables. This observation is confirmed by additional related work, such as [SJT03] and [Hu99]. Thus, our approach also makes uses of whitespace alignments. However, we note that line classification is only one aspect toward actually extracting tabular data. The original approach relies solely on the structural features of tables and does not take cell content into account, which we consider in our approach.

Döhmen et al. noted that existing CSV parsers make decisions during the file parsing process sequentially, which they suspect to negatively impact the overall quality [DMB17]. They propose a solution that makes decisions about sub-criteria as late as possible, trading run time for parsing quality. Besides CSV parsing, their heuristics cover file encoding detection, table normalization, and table area detection. While the approach includes a stage dealing with table area detection, it does not handle multi-table files that account for about 7% of the cases in our dataset. Additionally, the authors tested only a limited set of CSV variants. We include their published implementation as a baseline when comparing parsing accuracy.

In [BNS19] the authors introduce a novel data consistency measure to correctly parse CSV files. The consistency measure consists of a row pattern score and a data type score. The row pattern represents the column count per row, depending on the detected CSV dialect. The type score uses regular expressions to detect known data types within cell values and represents the ratio of known cells compared to the total number of cells. Both scores contribute equally to the consistency measure, favoring the pattern score on ties. The pattern-based approach seems to work well according to the provided evaluation. Yet, this solution also does not work with files containing multiple tables that our approach is able to handle. As the authors noted, it can become problematic if many of the cell data types are unknown. We use the publicly available implementation in our experiments.

Ill-formed CSV and ASCII files are not the only opportunity to extract relations from content that is designed to be human-readable. For instance, Chu et al. suggest the Tegra approach to recognize relational tables that appear as lists on web pages with the global record alignment

technique [Ch15]. As our approach is not designed to handle web tables, we do not compare to this approach.

Overall, existing works lack at least one of the aforementioned features: 1) parse input files automatically; 2) take content into account; 3) handle multi-table files. Our approach can handle all these limitations.

3 Table Formats

We recognize two table formats used for persisting data tables in raw plain-text files: character-separated-values tables (CSV) and other (ASCII) formats. We first introduce these two table formats in detail, and then state the table extraction problem.

3.1 CSV and ASCII tables

According to RFC 4180, a CSV file is a line-wise plain-text file that stores a table: each line represents a data record, and the first line optionally represents the table header [Sh05]. The cells of each line are separated by a special character, the *delimiter*. If a cell value includes the delimiter character itself, the value must be put into quotes using *quotation* characters. An *escape* character is used to escape a quote character or the escape character itself, if they appear within quoted field values. A file's *dialect* specifies the used delimiter d, quotation q, and escape characters e, denoted as $\langle d, q, e \rangle$ [Sh05]. Although the RFC document specifies comma as delimiter and double quote as quotation and escape, it acknowledges the usage of a wide variety of characters for each dialect component in real-world data [Sh05]. Because CSV files do not carry metadata, the presence of different dialects within and across files acts as a barrier to the automatic table interpretation and extraction.

A W3C working group for "CSV on the Web" proposes the delivery of an additional JSON file, which contains information about the used dialect and the schema [BTH16]. CSVY is a similar development, which stores such information as a YAML meta block at the beginning of the file (`www.csvy.org`). However, neither standard has been widely adopted.

An **ASCII table** separates columns with white space. To visually align values within each column, ASCII tables fill the column gap between fields with one or more space or tab characters. With their visual alignment, ASCII tables are more suitable for human readability. Fields are separated by white space so that values from different columns do not horizontally interfere each other. Two columns must be separated by at least one whitespace character. Because the characters and their number may vary between different pairs of neighboring fields, we cannot simply delimit lines by using a fixed number of whitespace characters. It is also not possible to accept an arbitrary number of space characters as delimiter, as empty fields would not be recognized properly and field values themselves might include spaces. Instead, a set of *column boundaries* is required: each boundary defines the inclusive start and

exclusive end of a column as the interval $[start, end)$, based on the character index. Figure 2 shows an ASCII table using $[0, 5)$, $[7, 23)$, $[26, 31)$, and $[32, 35)$ as column boundaries.

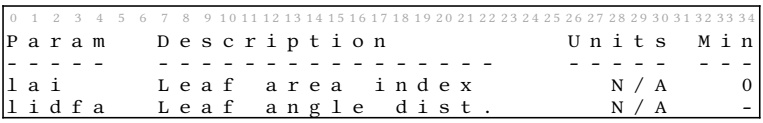

Fig. 2: Exemplary ASCII table – columns aligned by layout

ASCII tables can include style information, such as borders, which can make the table structure clearer to human readers. In particular, horizontal lines are often used for underlining headers or separating tables. We therefore distinguish two line types. We refer to a line as a *helper* line if its content includes only non-alphanumeric characters or whitespace. All lines with at least one alphanumeric character are *content* lines.

3.2 The Table Extraction Problem

Before we can extract tables from plain-text files, we must understand the structures of these files by identifying the dialects and the column boundaries of tables stored in these two respective file formats. We refer to dialects and column boundaries as *parsing instructions* for the two types of files. Here, we highlight the difficulty resulting from the lack of parsing instructions due to multiple valid ways of interpreting lines.

Figure 3 shows a file excerpt allowing for multiple ASCII interpretations. When regarding the first two lines only, we might split each line into five fields, namely `Leaf`, `angle`, `distribution`, `N/A`, and `-`. However, the introduction of the third line yields multiple interpretation possibilities.

```
Leaf angle distribution   N/A   -
Leaf angle distribution   N/A   -
Dry/Wet soil factor              0
```

Fig. 3: ASCII table adapted from Figure 1 emphasizing the ambiguity of some ASCII records

Similar effects can be observed for tables stored in the CSV format. Even if we consider only dialects using single non-alphanumeric characters, we can generate seven valid delimiter candidates for a line with the text `"N, \"Leaf\"structure";N/A;"0.8"`. A simple approach may select the delimiter character based on the candidate frequency across lines. However, this method is sensitive to the content. For example, cells including delimiter-characters could easily fail this approach.

We state the *table extraction problem from plain-text files* as follows: Given a plain-text file containing one or more vertically stacked tables, determine the line structure and the range (the beginning and the end row indexes) of each table and transform every table to

the RFC 4180 standard. For simplicity, we assume that individual cells do not contain line breaks. Moreover, we exclude the detection of the file encoding from our problem and assume UTF-8 as specified in RFC 4180.

4 The ExtracTable algorithm

We propose ExtracTable, an algorithm that exploits data type consistency within columns to tackle the table extraction problem. To interpret fields in a file, ExtracTable first detects per-line valid dialects for CSV tables and possible column boundaries for ASCII tables (Section 4.1). After applying the detected parsing instructions, the resulting interpretations divide each line into several fields, which are passed to the next step to identify data type patterns (Section 4.2). Then our approach generates table candidates with the compatibility score (Section 4.3). Finally, the algorithm selects a subset of the table candidates (Section 4.4).

4.1 Parsing instruction detection

Detecting parsing instructions is modeled as dialect detection for CSV tables and column boundary detection for ASCII tables, respectively. ExtracTable first pre-processes a file by classifying each line as either a helper line or a content line. It prunes all helper lines, as they neither deliver content nor help detect correct column boundaries of ASCII tables, and may be incorrectly treated as part of the header or the data region of a table.

Dialect detection for CSV tables. To recognize a CSV table's dialect, we propose a two-step approach that first detects all delimiter candidates, and then quotation and escape characters for each delimiter candidate. First, ExtracTable replaces consecutive alphanumeric characters and excluded characters within a line l with a placeholder character. It then splits the resulting string by the placeholder character, yielding a list of delimiter sequences and empty values. All substring combinations of each delimiter sequence are appended to the list. Values that are empty or longer than the maximum length are removed.

For each detected delimiter, ExtracTable tries to recognize the quotation and escape characters using a depth-first search method, shown in Algorithm 1. The algorithm receives the line content l and a delimiter sequence d as input. It tries to parse the line using the dialect $dialect_0 = \langle d, \varepsilon, \varepsilon \rangle$ (see line 16). The `parse` method iterates over the character positions of the trimmed line content. For each iteration, get_dialect_component returns the component matching the dialect specified in the method parameters following the RFC 4180 grammar. The component can be one of content, delimiter, quotation, escape, or error (see line 5). In cases where the character at the current position *cursor* violates the RFC 4180 grammar, the get_dialect_component method returns an error and disregards the dialect (line 7). The state machine for parsing the dialect specified in the $parse$ method parameters

is updated in update_parser_state (line 12) based on the returned component. Additionally, the algorithm checks whether the remaining line starts a new component from the given dialect. If the current position was classified as content and is not alphanumeric, we could interpret the content as a quotation or escape. Line 9 starts a new branch of the DFS using the remaining line content and the updated dialect $dialect_1 = \langle d, q, \varepsilon \rangle$, where q is the character sequence at the current position. The same logic is applied to the escape character, as shown in line 11. If the parser can process the whole line without errors, it found a legitimate dialect. Finally, the parser returns all valid dialects.

Algorithm 1: Quotation q and escape e character detection

Input: Line content l and delimiter d
Output: A set of dialects

```
1  Def parse(l, d, q, e, cursor):
2      cursor=0
3      tl=trim(l)
4      while cursor < |tl| do
5          ⟨component, length⟩ = get_dialect_component(tl, cursor, d, q, e)
6          if component="error" then
7              return ε
8          if q = ε ∧ component = "content" ∧ ¬isalnum(tl_cursor) then
9              parse(tl, d, tl_cursor, ε, cursor)
10         if q ≠ ε ∧ e = ε ∧ component = "content" ∧ ¬isalnum(tl_cursor) then
11             parse(tl, d, q, tl_cursor, cursor)
12         update_parser_state(component)
13         cursor = cursor + length
14     dialects = dialects ∪ ⟨d, q, e⟩
15 dialects = []                                          // square brackets denote list
16 parse(l, d, ε, ε, 0)          // start DS using delimiter d, empty quotation and escape
17
18 return dialects \ {ε}
```

Column boundary detection for ASCII tables. Per our definition of ASCII tables, two columns must be separated by a vertical line that has at least one space character. A *vertical line* is a consecutive set of character positions, where all characters in lines are whitespace. To infer the boundaries for all columns, ExtracTable detects vertical lines in-between columns. Algorithm 2 shows the proposed approach to detect vertical lines in an ASCII table. We explain the algorithm using the example depicted in Figure 4. The variable k depicts the line index. The file *width* is the length of the longest line within a file. The set of whitespace characters is represented by the variable *WS*.

Transform line content into bitmap with `transform(`l`,`*width*`)`: The algorithm first transforms the line content l into a bitmap. As lines may contain a combination of tabs and spaces for aligning columns, all tab characters are expanded with the corresponding number of space characters first. We use a tab size of eight, which is the default number in Python's `expandtabs` function. All whitespace characters are then replaced with 1 (*True*) and any

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$k=0$	N	a	m	e							M	i	n	u	t	e				Q	u	o	t	e										
$k=1$	=	=	=	=							=	=	=	=	=	=				=	=	=	=	=										
$k=2$	V	i	c	t	o	r	i	a												N	/	A												
$k=3$	H	a	r	r	y										4	0				I		l	i	k	e		c	o	m	p	l	e	x	.

(a) An example file with four lines where characters are displayed in monospaced font.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$k=0$	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
$k=1$	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
$k=2$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$k=3$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0

(b) A bitmap representation where green and orange shaded areas show vertical lines and the ending of them, respectively. Spacers of two table candidates (shown in red and blue frames) are detected.

Fig. 4: Example for the column boundary detection algorithm.

other character with 0 (*False*). Lines are padded using multiple 1s to the *width* of a file. Figure 4b shows the bitmap representation for the lines in Figure 4a.

After transforming the line content into a bitmap, the algorithm searches for *vertical lines*. Subsequent text lines, where $bitmap_w = 1$ for the same character position w, form a vertical line at w. Consecutive vertical lines are grouped into *spacers*, which are represented as a set of consecutive *indexes*. Each index represents the character positions of a vertical line. Spacers are *significant*, if they contain more than one vertical line ($|indexes| > 1$) and are not leading ($0 \in indexes$) or trailing ($width - 1 \in indexes$). Significant spacers are mandatory for tables. For the first and the second lines, there are three spacers: columns 4-9, 16-18, and 24-33.

Append discovered tables with `start_table(`*counter*`,`*l*`)`: The algorithm identifies a new table if there is at least one vertical line spanning P_{mrc} (min row count) text lines. The new table is defined by its starting line index and a set of spacers. For our example, we choose $P_{mrc} = 2$. Thus, there was no table discovered after processing the first line. However, after proceeding with the second line of the example, multiple vertical lines span the minimum required number of text lines. The first two lines in Figure 4b show a table t_0 that has three spacers shaded in green.

Update existing tables with `update_table(`*t*`,`*bitmap*`)`: While processing subsequent lines, the existing tables are updated based on the continuation of vertical lines. If a subset of vertical lines belonging to a significant spacer is discontinued, the spacer shrinks or is split into smaller ones so that the continued lines are represented. If the vertical line of an insignificant spacer was discontinued, the algorithm removes it from the set of table spacers. The interruption of all indexes of any significant spacer marks the end of the table.

Algorithm 2: Column boundaries detection

Input: File content L, file *width*, white space characters *WS*
Output: Row range of *tables*, table *boundaries*

```
1  counter = {w → 0|0 ≤ w < width}     // number of consecutive lines for each vertical
     index
2  tables = []  // stores tables (indexed by t) with their starting/ending line indexes
3  boundaries = []                                  // stores boundaries of tables t
4  for k ← 0 to |L| do
5      closed = []
6      bitmap = transform(l_k, width)
7      if ∃char ∈ l_k : char ∉ WS then
8          for w ← 0 to width do
9              if bitmap_w = 1 then
10                 counter_w = counter_w + 1
11             else
12                 counter_w = 0
13         for t ← 0 to |tables| do
14             ⟨closed_t, boundaries_t⟩ = update_table(t, bitmap)        // closed_t ∈ {0, 1}
15         if ∃w ∈ 0, ..., width : counter_w = P_mrc ∨ closed_|tables|−1 = 1 then
16             tables = tables ∪ start_table(counter, l_k)
17 for t ← 0 to |tables| do
18     boundaries = close_table(t)
19 return tables, boundaries
```

Close tables with `close_table(t)`: If a closed table covers less than P_{msr} (min significant rows) rows, insignificant spacers are omitted from the final set, which the algorithm uses to compute the column boundaries. If the number of resulting column boundaries exceeds P_{mcc} (min column count), they are stored along the table lines in *boundaries*. The table is finally closed by removing it from the set of running tables. However, if there are still spacers of that table left, the algorithm creates a duplicate of the table. The clone uses the same set of spacers, but without the discontinued ones. The line index $k - 1$ is used as the start for the cloned table.

After processing the third line in the example, all spacers of t_0 still exist, whereas the first one shrinks, because the values included in the indexes 4-7 in the third line are zero (shaded orange). The algorithm does not find new tables. When reaching the last line, it updates the last spacer of table t_0 by shrinking it into a smaller, insignificant one. Additionally, vertical lines spanning P_{mrc} rows were found. A new table t_1 is created using the spacers $[8, 13]$, $[16, 18]$, and 25. The last spacer is insignificant as it contains only one index.

In our example, only the table t_0 is left. The set of column boundaries is the complement of the spacer indexes indicated by the red frames, in all indexes. When using $P_{msr} = 5$, the insignificant spacer at index 25 is dropped, as the table has only four rows. The remaining two spacers cover the indexes 8-9 and 16-18. Therefore, the column boundaries for t_0 are: $[0, 8)$, $[10, 16)$, and $[19, 34)$. They are assigned to all four table lines.

The algorithm applies the detected parsing instructions to obtain the resulting interpretations and the values of every field for each line. Leading and trailing whitespace are trimmed from all fields.

4.2 Field pattern extraction

In the previous steps, ExtracTable collected all valid parsing instructions for each line and returned the resulting interpretations for them. The algorithm generates data types for the values in each interpreted line and uses them in the next step to select the optimal interpretation for the line based on the data type consistency of the field values. Here, we explain how the algorithm determines the data type for a given value.

ExtracTable uses a set of 15 domain-agnostic regular expressions to detect known data types, covering all types mentioned in [BNS19]. Additionally, we include regular expressions for Boolean values, file paths, expressions in brackets, and hash-like values. The algorithm assigns the index of the first matching expression to the *known* data type (K). If no data type matches the value, the algorithm falls back to detect the atomic data type for the value. If a field value cannot be covered by any known data type, we use a sequence of atomic type components to describe the type of this value. We support three atomic types: *number* (N), *string* (S), and *other* (O). The remaining class *other* matches everything that is neither a number nor a string.

Empty values (E) are ignored when calculating the consistency of tables. Therefore, the appearance of missing values in combination with another data type in a column has no negative impact on the overall consistency. We use a list of values to represent various forms of empty values, including empty string ε, N/A, NA, NaN, Null, Unknown, and a sequence of more than one question mark, dash, star, or number sign, respectively.

Finally, we define a pattern as a vector of pattern components. A pattern component can be one of String, Number, Known, Empty, or Other. For the input file, ExtracTable detects several valid interpretations, each of which is applied to obtain a set of fields for each line. The field pattern extraction step assigns a pattern for the value of every field.

4.3 Table candidate generation

With the value pattern for each field, we calculate a consistency score for a set of lines over corresponding fields across these lines. We introduce a score-based approach that exploits this consistency score to build table candidates. Similar to the detection of column boundaries, ExtracTable iterates over all lines and builds table candidates on the fly. It groups line interpretations by two criteria: The primary information is the column count n and the secondary is the parsing instruction *instr*. The algorithm compares the list of represented groups with the set of existing table candidates *TC*. A new table candidate *C*

is started upon the discovery of an unrepresented group. Table candidates are terminated, if they are no longer represented or if the file end has been reached. Terminated table candidates are passed to the final step of ExtracTable.

The algorithm then adds the corresponding interpretations to the table candidates. Before doing so, it checks whether the current line and the lines in a table candidate are compatible with regard to the *consistency score* of corresponding fields across the lines. If the data types are consistent, the interpretations are appended to the table candidate. If the lines are incompatible, the algorithm starts a new table candidate. Based on our observation, we can assume transitivity: If l_k is consistent with both l_{k-1} and l_{k+1}, then l_{k-1} and l_{k+1} are also consistent. Therefore, we compare the current line with only the most recent row of the table candidate. The new table candidate might be created twice: once with and once without using the previous block of compatible lines as a header. The row count of potential headers must not exceed P_{mhr} (max header rows) and the headers should not include any floats.

Given two rows, our data type-based *consistency score* returns a number between 0 (completely inconsistent) and 1 (perfectly consistent). We consider two interpretations to be compatible if the consistency score exceeds the threshold P_{mbc} (min block compatibility). We use the *pattern consistency* as the primary measure for the consistency score. The *value uniformity* within columns is calculated to compare consistent tables:

$$score : C \rightarrow \begin{cases} -1, & \text{if } |rich| = 0 \\ 0, & \text{if } |cons| < \lfloor \log_2(|rich|) \rfloor \\ \frac{|cons|}{|rich|} * \frac{1}{|rich|} \sum_{col}^{rich} u(col) & \text{otherwise} \end{cases} \quad (1)$$

where *rich* is the non-empty subset of columns in C, and *cons* is the homogeneous subset of *rich*. A column is homogeneous if its homogeneity score exceeds one of the thresholds P_{mbs} (min block score) or P_{mcs} (min column score). We calculate the score using the homogeneity metric proposed in [Gu11], which considers the distribution of different data types within one column. Based on our experiments, we require at least $\lfloor \log_2(|rich|) \rfloor$ pattern-consistent columns to compute the table's consistency with the third case of Formula (1). Otherwise, the score for that table is 0. The function u returns the *value uniformity* of a column *col*.

To calculate the uniformity of a column, we first generate the patterns for all values therein and group them by pattern. For each group, we calculate the uniformity using the homogeneity metric for each component in the pattern. For example, "ABC1" and "XYZ0.8" are both mapped to the pattern "SN". Therefore, the uniformity for both "S" and "N" are calculated. For *number* components, we compute the homogeneity of both integers and floats based on their respective counts. A similar calculation is performed for *other* components, where the homogeneity of the values is used. For *known* components of the same type and *string* components, we simply assume that all values are homogeneous and return 1. The value uniformity of *empty* values is undefined. Therefore, the score for the "S" and "N" classes in the above example are 1.0 and 0.5, respectively. Then we denote the

uniformity for this pattern by the maximum uniformity score across all components. Finally, the uniformity of the whole column is the weighted average of the pattern uniformity scores, where the weights are the occurrences of the patterns. The final score of a table is the average uniformity of all *rich* columns.

4.4 Table selection

In the final step, ExtracTable selects a subset of table candidates whose line ranges do not overlap. We model the table candidate selection problem as a *shortest path problem*. We first transform the set of table candidates to a multi-edged directed acyclic graph. The set of vertexes V represents the line indexes of a file. Each table candidate represents one edge, using the first and last line index for the source node *src* and the destination node *dst*, respectively. The distance for a given table candidate is calculated using the following formula. Lower distances represent larger and more consistent tables.

$$\begin{aligned} dist : C \rightarrow & - score(data(C)) \cdot (m_C - h_C)^2 \\ & - score(header(C)) \cdot (m_C^2 - (m_C - h_C)^2) - 0.0001 \cdot \mathrm{sgn}(h_C) \end{aligned}$$

where m_C is the number of lines in the table and h_C is the number of header rows therein. The function calculates the consistency scores $score(header(C))$ and $score(data(C))$ for the header and data parts, respectively. When comparing tables of the same size and consistency, we favor tables with a header by subtracting a small constant from the consistency score if a header exists. Before mapping table candidates to edges, the algorithm prunes the ones that have fewer than P_{mrc} rows and P_{mcc} columns, or have a consistency score of the data part lower than P_{mts} (min table score).

After the DAG has been filled, adjacent vertexes are linked. We connect each vertex pair $\langle v, v+1 \rangle$ by an edge with a distance of $dis = 0$. Figure 5 shows the graph for the seven table candidates shown in Table 1. The numbers at the edges represent the distances, and the squared boxes are the table candidate indexes.

Tab. 1: Example table candidates. 'From' and 'To' fields indicate the beginning and the end indexes of a table candidate. *SH* and *SD* stand for $score(header(C))$ and $score(data(C))$, respectively.

#	From	To	h_C	$m_C - h_C$	*SH*	*SD*
1	5	35	0	31	n/a	1.0
2	5	35	0	31	n/a	0.8
3	38	45	0	8	n/a	1.0
4	38	55	0	18	n/a	1.0
5	46	55	0	10	n/a	1.0
6	58	75	0	18	n/a	0.9
7	58	75	2	16	1.0	0.9

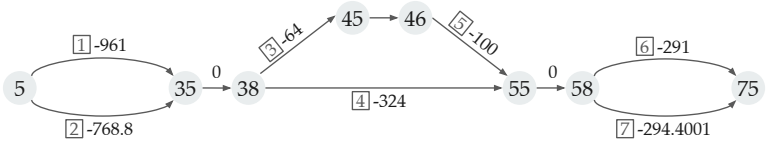

Fig. 5: Table selection graph for example of Table 1

We apply the Bellman-Ford algorithm [Be58] to find the shortest path. In case of a tie, we pick the candidate that has (i) a higher ratio of recognized fields to total fields; (ii) a higher number of *rich* columns; (iii) a lower pattern length; (iv) a lower column count. The best table candidates are found by sorting the edge candidates by their weight and by the criteria above. In the unlikely event that multiple candidates still qualify, we choose the first one.

5 Experimental Evaluation

We evaluated ExtracTable on large sets of files taken from open data portals, and compared it with existing solutions regarding accuracy and runtime. The experiments were executed in Python 3.8.5 on a Linux machine. The test system was equipped with an AMD EPYC 7702P CPU with 64 cores, operating at 2 GHz with 512 GB memory.

5.1 Datasets

The basis of our ground truth are two existing corpora from related work using plain-text files taken from Mendeley Data, GitHub, and UKdata[5]. The Mendeley data corpus was crawled in August 2020 to study line and cell classification tasks on verbose CSV files [JVN21]. This first corpus includes all projects that contain at least one plain-text file and were hosted on Mendeley's servers. It consists of 235 471 files distributed over 1 554 projects. Within the corpus, we found files of 1 040 different extensions. We kept all files with extensions *.txt*, *.dat*, *.csv*, *.md*, and *.out*, resulting in 94 474 files.

The second corpus was provided as part of [BNS19]. It consists mainly of CSV files taken from GitHub and UKdata. A repository hosted on GitHub typically contains a diversity of files required for the development of software. The British government uses UKdata to publish datasets from different departments, such as education, economy, or health. The dataset consists of 5 000 files each from GitHub and UKdata. Using the authors' script for downloading the corpus[6] from the original sources, some files were no longer available, leaving us with 2 577 and 2 539 files from GitHub and UKdata, respectively.

[5] `https://data.mendeley.com/`, `https://github.com/`, `https://data.gov.uk/`
[6] `https://github.com/alan-turing-institute/CSV_Wrangling/`

Annotating all almost 100 000 would be too time-consuming, so we selected a subset. We noticed that the Mendeley data source provides a larger variety of files and decided to grant it a larger share in our final dataset. Ultimately, we randomly selected 598 files from Mendeley Data, 176 files from GitHub, and 183 files from UKdata, resulting in 957 files. All files and annotations are publicly available[7].

We annotated all tables containing at least two columns and two rows. All rows belonging to the same table must have the same column count. Our definition of data tables includes tables with multiple header rows. In our 957 files, we annotated 1 208 tables and obtained first insights into the dataset. A regular table of our ground truth is quite small, with fewer than 1 000 rows and between two and ten columns. Approximately 75% of the 190 ASCII tables have fewer than 100 rows. While files containing a single table are represented using CSV in nine out of ten cases, ASCII tables are used for more than a third of all tables contained in multi-table files. Confirming the general observation of [DMB17], we found that 47% of the CSV tables follow RFC 4180. Also, 1% of the files contained at least one CSV table using a multi-character delimiter, e.g., an arrow (->), multiple slashes (//), or multiple tab or space characters. The majority of fields represent numbers (84%) and only a small portion of cells did not match any of our data types (4%).

5.2 Comparison targets

Our comparative analysis regards a simple baseline and four solutions from related work, which we used to evaluate table range selection and parsing accuracy. The simple baseline approach always returns the dialect specified in RFC 4180. By including this baseline when evaluating the parsing results, we were able to gain insights into the complexity of files and the dialect distribution. The *Sniffer* class is part of the CSV package[8] of Python. Sniffer infers the delimiter by character frequencies across lines. *Hypoparsr* covers multiple parsing steps, such as file encoding detection, dialect detection, and table area detection [DMB17]. While the R package was removed from the Comprehensive R Archive Network by the authors, we used the archived version 0.1.0 from GitHub[9]. Finally, the authors of *CleverCSV* propose a pattern-based approach to infer the dialect of a file [BNS19]. It is capable of handling surrounding text, but does not return the table ranges explicitly. Its command-line tool (version 0.6.7) is available via the Python Package Index[10].

To evaluate the quality of our table range selection, we used the Python implementation of *Pytheas* [Ch20] published by the authors[11] using the weights that the authors suggest. In addition, we use a *naive* approach for this particular evaluation, which simulates the missing baselines by classifying the complete file content as belonging to a single table.

[7] https://owncloud.hpi.de/s/uhHJFzC9mNcdF4i

[8] https://docs.python.org/3/library/csv.html (we used Python 3.8.5)

[9] https://github.com/tdoehmen/hypoparsr

[10] https://pypi.org/project/clevercsv/

[11] https://github.com/cchristodoulaki/Pytheas/tree/d77b82a

Other solutions mentioned in related work could not be applied to the table range selection problem. The authors either assumed only a single table to be present within a file, or their implementations did not return the explicit table ranges.

5.3 Table range selection

ExtracTable can be configured by a set of ten parameters. Half of the parameters, such as the minimum table dimensions, are subjective and depend on specific tasks. For our datasets, we require tables to have at least two columns and two rows. Based on a related work [Em16], we allow tables to have up to four header rows. The length of a dialect component must not exceed four characters, and all bracket characters are not allowed to appear within the delimiters. To find the optimal values for the remaining five parameters, we ran a grid search on a subset of our ground truth. We found the following settings to be optimal: $P_{msr} = 4$; $P_{mbc} = 0.71$; $P_{mbs} = 0.31$; $P_{mcs} = 0.51$; and $P_{mts} = 0.51$.

We measure the quality of the table range selection by calculating the *Intersection over Union* (IOU) for each pair of detected and annotated tables [Re19]. In [Do19] the authors use the IOU metric for evaluating the performance of the table detection in spreadsheets. Since we need to compare only the vertical table boundaries, we use the Jaccard index for the IOU. It returns a number between 0 (no match) and 1 (perfect match).

After calculating the Jaccard index for each table pair, we used the maximum Jaccard index to determine one of four match types: Annotated tables that returned a Jaccard index of 1 for some returned table are a *perfect match*. In contrast, if the maximum Jaccard index of an annotated table is 0, *no match* was found. All remaining annotated tables fall into the category of *partial matches*. We refer to returned tables that have no matching annotated table as *eager match*. Figure 6 shows the match type counts for the naive approach and for Pytheas and ExtracTable (ignoring eager matches). For each individual solution, we excluded those files that were not processed successfully within three minutes. Pytheas finished around 79% of all files, whereas ExtracTable processed around 87% successfully. The naive approach worked on all files due to its nature.

The naive approach returned the correct range for precisely 50% of the tables. The remaining half was classified as a partial match, as every file contains at least one table. Pytheas was able to detect 59% of the table ranges correctly, yet the approach missed every eighth annotated table. The tables that were not recognized are of different sizes and are equally balanced regarding their formats. 6% of all tables returned by Pytheas were not present in the ground truth.

ExtracTable identified the correct table ranges in more than 70% of all tables and missed seven tables (1%). A limitation is the high number of eager matches that are not depicted in the chart. Nearly one out of every six tables returned by ExtracTable is not present in our ground truth, and therefore are false positives. After manually examining a sample of these

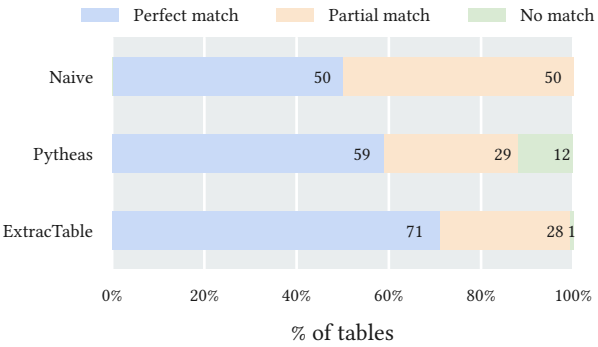

Fig. 6: Table range selection performance (higher number of perfect matches is better).

eagerly matched tables, we realized that it found consistent data tables within unlabeled tabular structures, such as dictionary fragments and single column tables. Pytheas returned fewer false positive tables than ExtracTable. However, we believe that for users, finding missing tables is more difficult than identifying incorrectly recognized tables in ASCII files, which appear in various shapes and forms. Therefore, the number of eagerly matched tables is a secondary metric compared to the number of correctly matched ones.

5.4 Line parsing

To evaluate parsing accuracy, we compared the returned lines of the comparison targets and ExtracTable line-wise with our annotations. A line was parsed correctly if the returned fields corresponded to the values in the ground truth, taking into account the order. We compared ExtracTable to four other solutions: RFC 4180, Hypoparsr, Sniffer, and CleverCSV. Some aspects of CSV parsing, such as the handling of space characters in-between fields, are implementation-specific. Therefore, we first extracted the dialects returned by the candidates. We then interpreted lines by feeding the dialect to the same parser. By doing this, we ensured a fair comparison, independent of the parser implementations.

The first experiment examines parsing correctness per table format. Figure 7 shows the ratio of fields that have been correctly parsed for both table formats. For CSV tables, we note that Sniffer, CleverCSV, and ExtracTable performed similarly well and detected the correct parsing instructions in about 90% of the cases. ExtracTable achieved slightly lower results than CleverCSV, as it interpreted some tables as ASCII instead of CSV. When disabling the ASCII support, ExtracTable parsed 94% of all table lines correctly: a higher generality (the ability to also parse ASCII tables) can be a cause for misinterpretations.

ExtracTable is the only solution optimized for ASCII tables: The remaining solutions recognized merely a small subset of lines correctly. Nevertheless, it is interesting to see that they returned a few correct interpretations. We identified three reasons that led to the

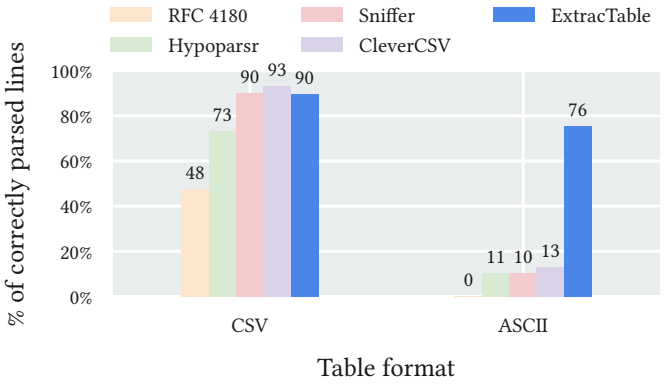

Fig. 7: Parsing accuracy (higher is better).

proper representation of single lines. First, empty lines occurring for a small subset of tables between the header and data part of a table are correct, independent of the used parsing instruction. Second, the nature of ASCII tables lets them use a different number of spaces to separate columns. Solutions besides ExtracTable sometimes chose the single space as the delimiter for interpreting these lines. While this does not result in the correct representation of the *whole* table, it sometimes yields the proper interpretations for a *subset* of lines, which is likely to happen for tables with few columns. Third, some tables can be interpreted using both ASCII and CSV. Such a situation may occur if the same number of spaces is used to separate all columns. Independent of these corner cases, we note that ExtracTable could correctly interpret 76% of the lines appearing in ASCII tables.

In general, the errors made by ExtracTable were independent of the table format but were caused by the table selection, which favors bigger tables. Lines were interpreted incorrectly for three main reasons: (i) over-segmented and under-segmented tables; (ii) short texts surrounding tables; (iii) misinterpretation of tables using tab characters. An annotated table was represented by multiple returned tables that contained partially sorted or similar values (over-segmented). ExtracTable under-segmented annotated tables if it found a parsing instruction that could be applied to neighboring tables of the same schema. As the table selection prefers tables with higher row counts, it merges both tables in such cases. Short texts surrounding the tables, such as table titles, causes ASCII tables to merge the first columns as the algorithm tried to include the header row. ExtracTable misinterprets CSV tables when it finds a dialect applicable to both the table and the surrounding text. We traced both reasons to our design decision to prefer tables having a higher row count. Finally, CSV tables delimited by the tab character were sometimes misinterpreted as ASCII tables.

To summarize the results, ExtracTable performed similarly to CleverCSV and Sniffer on parsing accuracy for CSV tables. Hypoparsr did not perform well, yet it outperformed the RFC 4180 baseline. For ASCII files, only ExtracTable could correctly parse a reasonable

number of lines: our approach is more general across the two file types. We assume that the parsing accuracy could be enhanced by pruning non-table lines – a main source of errors.

5.5 Runtime

We measured the runtime using Linux's internal system call `getrusage`. We compared the runtime of our approach to the ones of RFC 4180, SNIFFER, HYPOPARSR, CLEVERCSV, and PYTHEAS. To reduce the overall runtime of our experiments, we used a timeout of three minutes, which allowed the slowest approach, Pytheas, to finish for more than 70% of the files, covering the majority of the dataset. Only one file fails all approaches with this timeout, which consists of 450 lines, each having 17 365 characters. For each file, we recorded whether the approach was able to process the files within the processing time and returned *some* result. To make the runtime comparable, we kept only files completed by all parsers within the limit. While this could add a bias towards simpler files, it ensures a fair comparison. Figure 8 shows the resulting runtimes per line in milliseconds on a logarithmic scale, based on 551 files.

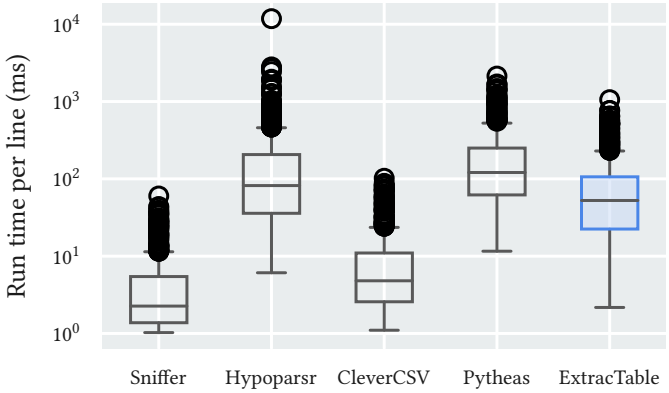

Fig. 8: Runtime comparison using logarithmic scale.

The solution that always returns the configuration of RFC 4180 does not read the file contents and always had a runtime of zero milliseconds. Sniffer and CleverCSV are both very fast, needing less than 10 ms per line on average. Hypoparsr, Pytheas, and EXTRACTABLE were slower and took 190 ms, 217 ms, and 90 ms, respectively. We acknowledge that all solutions cover a different feature set: While Sniffer is a heuristic approach, CleverCSV uses a more advanced, pattern-based dialect detection. Hypoparsr includes multiple stages, such as encoding detection and normalization. Pytheas uses a large set of fuzzy rules to detect table ranges. Our approach includes aspects from different solutions as it covers a wider range of CSV dialects, handles ASCII tables, and is capable of detecting multiple tables within files.

One driver for the longer runtimes is the number of interpretations. This number depends on the chosen configuration, line count, and the actual file content. Lines that are very long or contain many space characters or non-alphanumerical characters take longer to process. The second driver is the number of table candidates. How many table candidates are found depends on the actual content and data type compatibility across lines.

6 Summary and Outlook

Tables are stored in arbitrary shapes and forms in plain-text files. To enable automatic information extraction from these types of files, we must first detect the positions of tables and their structures. We proposed the ExtracTable algorithm, which tackles the table extraction problem. For a given file, the algorithm first detects and tests possible parsing instructions: dialects and column boundaries for CSV and ASCII tables, respectively. After applying the parsing instructions to the line content, ExtracTable infers the data type of each field. It then builds table candidates based on the consistency of data type patterns, field count, and parsing instruction. Finally, the algorithm models the optimal table selection problem as the shortest path problem, and outputs a set of tables for the given file.

To evaluate our algorithm, we annotated a dataset consisting of nearly 1 000 files taken from Mendeley Data, GitHub, and UKdata. We analyzed two aspects of our algorithm: (i) the table range selection; (ii) the parsing accuracy. Our evaluation showed that ExtracTable outperforms the other approaches in determining the table ranges, detecting the correct range for more than 70% of the tables. Comparing the parsing results between ExtracTable and the related approaches, we found that CleverCSV performs best on CSV tables, parsing 93% of the lines correctly. Yet, ExtracTable performs similarly well, yielding correct parsing results for 90% of the lines. Our solution was the only one capable of parsing a significant number of ASCII tables and achieved an accuracy of 76%.

While ExtracTable supports more complex files, we still had to make a few assumptions, whose relaxation could be interesting future work. This includes the support for cells containing line breaks, as well as spanning rows and spanning columns. The main challenge lies in the scoring of different table candidates. Future work may investigate to what extent the algorithm benefits from learning the structure and content of *typical* tables [VHN22]. We hope that by inferring that knowledge during table selection, wrong interpretations yielding high consistencies can be pruned. Additionally, we identified table selection to be misled by text lines preceding or succeeding a table, because we favor tables with higher row counts. This effect could be reduced by filtering non-table lines as a pre-processing step.

By using the ExtracTable algorithm, data scientists can extract tables from a wider variety of plain-text files. Therefore, they spend less time dealing with data wrangling and instead focus on their actual data-driven tasks. While the evaluation returned good results already, we are still far away from handling files fully automatically.

References

[An20] Anaconda: 2020 State of Data Science, tech. rep., 2020, URL: https://know.anaconda.com/rs/387-XNW-688/images/Anaconda-SODS-Report-2020-Final.pdf.

[Be58] Bellman, R.: On a routing problem. Quart. Appl. Math. 16/, pp. 87–90, 1958.

[BNS19] van den Burg, G. J.; Nazábal, A.; Sutton, C.: Wrangling messy CSV files by detecting row and type patterns. Data Mining and Knowledge Discovery 33/6, pp. 1799–1820, 2019.

[BTH16] Brickley, D.; Tennison, J.; Herman, I.: CSV on the Web Working Group @ www.w3.org, tech. rep., 2016, URL: http://www.w3.org/, visited on: 04/23/2021.

[Ch14] Chessell, M.; Scheepers, F.; Nguyen, N.; van Kessel, R.; van der Starre, R.: Governing and Managing Big Data for Analytics and Decision Makers. IBM Redguides for Business Leaders/, p. 28, 2014, ISSN: 0306-0012.

[Ch15] Chu, X.; He, Y.; Chakrabarti, K.; Ganjam, K.: Tegra: Table extraction by global record alignment. In: Proceedings of the International Conference on Management of Data (SIGMOD). Pp. 1713–1728, 2015.

[Ch20] Christodoulakis, C.; Munson, E. B.; Gabel, M.; Brown, A. D.; Miller, R. J.: Pytheas: Pattern-Based Table Discovery in CSV Files. PVLDB 13/12, pp. 2075–2089, 2020, ISSN: 2150-8097, URL: https://doi.org/10.14778/3407790.3407810.

[DMB17] Döhmen, T.; Mühleisen, H.; Boncz, P.: Multi-Hypothesis CSV Parsing. In: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM). New York, NY, USA, pp. 1–12, 2017.

[Do19] Dong, H.; Liu, S.; Han, S.; Fu, Z.; Zhang, D.: TableSense: Spreadsheet table detection with convolutional neural networks. In: Proceedings of the Conference on Artificial Intelligence (AAAI). Pp. 69–76, 2019.

[Em16] Embley, D. W.; Krishnamoorthy, M. S.; Nagy, G.; Seth, S.: Converting heterogeneous statistical tables on the web to searchable databases. International Journal on Document Analysis and Recognition (IJDAR) 19/2, pp. 119–138, 2016.

[Gu11] Guo, P. J.; Kandel, S.; Hellerstein, J. M.; Heer, J.: Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts. In: Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST). Pp. 65–74, 2011.

[HN20] Hameed, M.; Naumann, F.: Data Preparation: A Survey of Commercial Tools. SIGMOD Record 49/3, pp. 18–29, 2020.

[Hu99] Hu, J.; Kashi, R. S.; Lopresti, D. P.; Wilfong, G.: Medium-independent table detection. In: Document Recognition and Retrieval VII. Vol. 3967, International Society for Optics and Photonics, pp. 291–302, 1999.

[IB72] IBM Corporation: IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information./, p. 17, 1972.

[JVN21] Jiang, L.; Vitagliano, G.; Naumann, F.: Structure Detection in Verbose CSV Files. In: Proceedings of the International Conference on Extending Database Technology (EDBT). Pp. 193–204, 2021, ISBN: 9783893180844.

[Mo18] Mooney, P.: Kaggle Machine Learning & Data Science Survey, 2018, URL: https://www.kaggle.com/paultimothymooney/2018-kaggle-machine-learning-data-science-survey, visited on:

[PC97] Pyreddy, P.; Croft, W. B.: TINTIN: a system for retrieval in text tables. In: Proceedings of the ACM International Conference on Digital Libraries (DL). Pp. 193–200, 1997.

[Pi03] Pinto, D.; McCallum, A.; Wei, X.; Croft, W. B.: Table extraction using conditional random fields. In: Proceedings of the International Conference on Information retrieval (SIGIR). Pp. 235–242, 2003.

[Pr16] Press, G.: Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says. Forbes Tech/, pp. 4–5, 2016, URL: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/.

[Re19] Rezatofighi, H.; Tsoi, N.; Gwak, J.; Sadeghian, A.; Reid, I.; Savarese, S.: Generalized intersection over union: A metric and a loss for bounding box regression. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Pp. 658–666, 2019.

[Sh05] Shafranovich, Y.: Common Format and MIME Type for Comma-Separated Values (CSV) Files, RFC 4180, RFC Editor, Aug. 2005, URL: https://www.rfc-editor.org/rfc/rfc4180.txt.

[SJT03] e Silva, A. C.; Jorge, A.; Torgo, L.: Automatic Selection of Table Areas in Documents for Information Extraction. In: Progress in Artificial Intelligence. Berlin, Heidelberg, pp. 460–465, 2003.

[VHN22] Vitagliano, G.; Hameed, M.; Naumann, F.: Structural Embedding of Data Files with MAGRITTE. In: NeurIPS Table Representation Learning workshop (TRL). 2022.

[VJN21] Vitagliano, G.; Jiang, L.; Naumann, F.: Detecting Layout Templates in Complex Multiregion Files. PVLDB 15/3, pp. 646–658, 2021.