Leonardo Hübscher

ExtracTable
Extracting Tables From Plain Text

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Hasso Plattner Institute

Supervisor
Prof. Dr. Felix Naumann

Advisor
Lan Jiang

University of Potsdam

Digital Engineering Faculty

Potsdam, May 2021

Zusammenfassung

Tabellen werden häufig für die Organisation und den Austausch von Informationen verwendet. Um die Verwendung möglichst unabhängig von einem konkreten Anwendungsbezug zu ermöglichen, werden Tabellen gemeinsam mit etwaigen Metadaten in plain text Dateien gespeichert. Die Tabellenstruktur wird dabei mittels CSV (engl. *comma-separated values*) oder visuell strukturiert (ASCII) abgebildet. Obwohl eine Spezifikation für das CSV-Format existiert, entwickeln Nutzer eigene Versionen. Bevor Data Scientists die in den Tabellen enthaltenen Informationen analysieren können, müssen sie daher zunächst das verwendete Format und dessen Konfiguration bestimmen. Dieser Arbeitsschritt ist nicht nur sehr zeitaufwendig, sondern auch repetitiv und somit fehleranfällig.

Wir wollen Data Scientists dabei unterstützen, weniger Zeit mit der Datenaufbereitung zu verbringen. Dazu schlagen wir eine Lösung vor, welche Tabellen aus plain text Dateien automatisiert extrahiert. Während es für Menschen einfach ist, Tabellen anhand deren Semantik in Dateien zu lokalisieren, so stellt es für Maschinen noch eine Schwierigkeit dar, da diese nicht über ein solches Verständnis verfügen.

Bestehende Ansätze sind in der Lage, Tabellen aus CSV Dateien zu extrahieren, in dem sie Heuristiken anwenden oder den strukturellen Aufbau einer Tabelle analysieren. Keine der vorhandenen Lösungen unterstützt jedoch die Extraktion aus Dateien, welche mehrere Tabellen enthalten. In dieser Arbeit stellen wir den ExtracTable Algorithmus vor, welcher Tabellen basierend auf der Datentyp-Konsistenz in Spalten extrahiert. Im Gegensatz zu bestehenden Lösungen ist ExtracTable in der Lage, mit Dateien umzugehen, welche mehrere Tabellen enthalten. Außerdem unterstützt unser Ansatz nicht nur CSV-Tabellen, sondern auch ASCII-Tabellen. Zusätzlich erkennt der Algorithmus CSV-Dialekte, welche stärker von der Spezifikation in RFC 4180 abweichen.

Wir evaluieren ExtracTable anhand eines Datensatzes, welcher ungefähr 1.000 Dateien von verschiedenen Open Data Portalen umfasst. Im Vergleich zu bestehenden Ansätzen erkennt ExtracTable die Tabellengrenzen um 20% besser und gibt den exakten Tabellenbereich für 70% der Tabellen zurück. Beim Vergleich der Parsing-Genauigkeit erkennt der Algorithmus die korrekten Dialekte für 90% der CSV-Tabellen, ähnlich wie bei vorhandenen Ansätzen. ExtracTable ist die einzige Lösung, welche ASCII-Tabellen unterstützt und die richtigen Spalten-Umbruchlinien für 76% zurückgibt. Um den ExtracTable-Algorithmus ausprobieren zu können, haben wir eine Demo Web App entwickelt, welche innerhalb des Universitätsnetzwerks erreichbar ist.

Abstract

Tables are a commonly used way for organizing and exchanging data. To allow generic use, tables are stored in plain text files that might include non-table content such as metadata. Comma-separated values (CSV) and layout features (ASCII) are two prevalent ways to represent tables. Although there is a specification for the CSV format, users develop variations with customized field separators and table layouts. Thus, before data scientists can analyze the data of such tables, they need to configure the parsers for each file manually. This process is not only time-consuming but also repetitive and error-prone.

Given that we want to help data scientists to spend less time with data preparation, we propose a solution that automatically extracts tables from plain text files. While humans can easily locate tables with semantic understanding, machines struggle as they lack such information.

Existing work tackles the table extraction problem for CSV files using heuristics or pattern-based approaches. None of the related work covers a parser that is capable of handling files containing multiple tables. We propose ExtracTable, an algorithm that exploits the data type consistency within columns to extract tables from plain text files. In contrast to existing solutions, ExtracTable works with multi-table files, tables represented in ASCII or CSV, and CSV dialects deviating more from the specification RFC 4180.

We evaluate our algorithm on a data set consisting of about 1,000 files taken from various open data portals. Compared to existing approaches, ExtracTable is 20% better at recognizing the table ranges and returns the accurate table boundaries for 70% of the tables. When comparing the parsing accuracy, the algorithm detects the correct dialects for 90% of the CSV tables similar to existing approaches. ExtracTable is the only solution supporting ASCII tables and returns the proper column boundaries for 76% of them. To demonstrate the capabilities of ExtracTable, we developed a demo web app, available from within the university network.

Contents

List of Figures

1 Tables in unstructured data

The amount of data generated worldwide is overwhelming. According to the IT company IBM, 2.5 million terabytes of data were created daily in 2013[1]. The storage manufacturer Seagate expects the number to multiply a hundredfold by 2025[2]. Companies profit from the data by using it for better decision-making, as it allows them to tailor their products to the customer's needs. Yet, the exact use case of the data in question is unknown during creation and storage. It is difficult and time-consuming to define a suitable data model for such data. Thus, it is stored in an unstructured way, often represented as textual content. According to Seagate, there is a general shift from structured to unstructured formats[2]. Open data portals are one way for governments, companies, and individuals to make data publicly available. They choose to publish data in unstructured ways as it allows a large number of interested users to work with the data. Open data portals are a central place to find plain text files containing tables. The German government even obliges its ministries to publish data about topics of public relevance, such as infrastructure, environment, and finances[3].

Because of the rapid growth of unstructured data, it becomes infeasible to analyze the vast amount of data by hand. Data scientists are experts at analyzing such data by applying appropriate algorithms that process the data automatically. However, before they can start with the actual analysis, they often need to prepare data sets first. Collection, cleanup, and normalization are typical tasks of data preparation. Each task has its challenges: Data scientists need to carefully select the data to avoid adding an unwanted bias to the analysis results. The cleaning and normalization tasks are time-intensive, because data sets can look very different. Data scientists need to handle them individually. 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 with data wrangling, preventing them from tackling the actual problem[4][5][6][7]. It is not only time-consuming but also tedious and error-prone because of its repetitiveness. Still, data preparation is necessary as data quality issues otherwise prevent subsequent algorithms from working well. By the end of 2025, a total of 175 ZB (1 ZB $= 10^9$ TB) of data are expected, emphasizing the need to automate the data preparation task[2]. Time-sensitive information which is not processed fast enough, might loose its value.

One way of sharing data are tables. They are favored because they can hold information in high densities in a structured way. Many companies use relational database systems for their applications, which represent entities and relations as tables. It follows that exports from such systems are also tables. Humans, as well as algorithms, work well with tabular structures. While tables look similar when interpreted, they may look different when persisted as files. Microsoft Excel stores spreadsheets in XSLX-files that are based on XML. This format is well-defined and allows for proper parsing. Plain text files, on the other hand, lack proper instructions on how to interpret them. Still, more than 50% of the files available on open data portals like Mendeley Data are plain text files containing tables[8]. We identified two format families used to persist tables. Figure 1.1 depicts examples showing the structure of those formats. To separate the complex domain knowledge entailed by open data files from the table structures, we show example tables depicting actors from the movie "Extraction"[1].

```
First name,Last name,YOB
Bruce,Willis,1955
Gina,Carano,1982
Kellan,Lutz,1985
```

CSV

```
First name  Last name      YOB
Bruce       Willis        1955
Gina        Carano        1982
Kellan      Lutz          1985
```

ASCII

Figure 1.1: ASCII and CSV tables depicting actors of the movie "Extraction"

The table on the left side uses a character as a field separator. This format is known for its comma-separated values (CSV). It was specified in RFC 4180 in 2005[9]. On the right side, we can see a table separated by layout features. We refer to such tables as ASCII-tables.

IBM first used CSV in 1972, showing a gap of 33 years[10]. The late standardization is the reason companies developed own formats, which deviate from the specification. Unfortunately, the RFC formalization does not account for such variations. This problem forces data scientists to take care of each file individually whenever they want to extract tables.

The examples shown above are rather simple. Yet, there exists a plethora of different formats, shapes, and sizes. Figure 1.2 depicts further examples highlighting the complexity of the table extraction problem.

Many files include a preamble or other text surrounding tables for which Figure 1.2a depicts an example. Existing text typically delivers contextual information depending on the data domain, such as experimental setups or sensor information. Parts of surrounding text might be misinterpreted as a table when

[1] https://www.imdb.com/title/tt4382872/

```
Title: Extraction
Published: 2015
First name,Last name,YOB
Bruce,Willis,1955
             ...
```

(a) (CSV) table with preabmle

```
First name  Last name     YOB
----------------------------
Bruce       Willis       1955
Gina        Carano       1982
Kellan      Lutz         1985
```

(b) ASCII table with styling

```
Full name;mm/dd/yyyy
Bruce Willis;05/19/1955
Gina Carano;04/16/1982
Kellan Lutz;05/15/1985
```

(c) Ambiguous CSV table

```
First and last name       YOB
B. Willis                1955
G. Carano                1982
K. Lutz                  1985
```

(d) Ambiguous ASCII table

Figure 1.2: Examples highlighting the complexity of the table extraction problem

containing table-like structures. Yet, they should not be part of the resulting table. Figure 1.2b shows an ASCII tables that makes use of styling features. While such features can be helpful when parsing, they should be removed from the final result. Figure 1.2c and Figure 1.2d show examples of tables that allow for multiple syntactical interpretations that pose a challenge for computers. While a human can infer the correct interpretations based on its understanding of data, a machine might split the ASCII table between the first and last name when the first line would not be taken into account. It could also consider that the words *last* and *name* represent two individual columns. The CSV table can be interpreted using semicolon, space, or slash as the delimiter. Every option produces a consistent column count across all lines. While these examples emphasize the table variety, other problems aggravate the table extraction, such as files containing multiple tables, different representations of missing values, or sparse tables.

Within the scope of this master thesis, we want to decrease the time data scientists spend with data wrangling by extracting tables from plain text files automatically. While this appears to be trivial to humans, it poses a problem to machines. Humans can easily recognize the proper interpretation of a table file based on their understanding of how data looks. Computers lack this kind of perception.

Existing related work focused mainly on the extraction of CSV tables and paid less attention to ASCII tables. We note that pattern-based approaches, which have been developed for table discovery and parsing, seem to be promising. In contrast to existing approaches, we aim to develop a parsing solution that covers more complex files entailing a wider variety of CSV dialects, files with multiple tables, and tables represented using ASCII.

We developed the ExtracTable algorithm, which processes files in six steps. After reading the file content, the algorithm detects all possible line interpretations. For each alternative, it describes the data types of the cells. The next step requires these when building table candidates, as it groups subsequent lines based on the column count, table format, and a data type-based consistency measure. Finally, the algorithm selects the longest and most consistent tables.

To summarize, the main contributions of this master thesis are:

1. A novel pattern-based approach to solve the table extraction problem for ASCII and CSV tables.
2. The annotation of $\approx 1,000$ plain text files containing more than $1,200$ tables from GitHub, UKdata, and Mendeley Data.
3. The evaluation of ExtracTable to existing solutions concerning line classification, table range selection, and parsing accuracy.

In the next chapter, we present related work dealing with table discovery, table parsing, and other relevant topics. In Chapter 3, we introduce the underlying definitions and technical terms for data tables and the two table formats, which build the foundations for the remaining thesis. We present the ExtracTable algorithm in Chapter 4 and explain its workflow in greater detail. In Chapter 5, we analyze the main characteristics of our ground truth, followed by a series of experiments evaluating the accuracy and run time performance of ExtracTable compared to existing solutions. We conclude the gained insights in Chapter 6.

2 Related Work

Table retrieval is an old problem. Yet, the requirements have changed in the recent years with the rise of open data portals. In the following, we present the characteristics of open data portals as of 2016 based on [11]. We divide the table retrieval problem into table discovery and table parsing and provide related work for both sub-problems per table format. While we did not find existing work covering the fully automatic parsing of ASCII tables, there are two approaches dealing with the parsing of CSV files. Lastly, we present a paper providing an overview of the open data preparation tasks, which benefit from solving the parsing problem. It serves as an outlook for what kind of problems need to be solved afterward.

Mitlöhner et al. analyzed multiple open data portals regarding the shape of CSV tables[11]. They provide an overview of the table file characteristics based on a data set consisting of more than 3.5 million resources taken from 232 different portals. 5% of all documents are labeled as table files and are represented using CSV. The authors developed several heuristics to identify comment lines, header rows, and multiple tables within files and describe their distributions. They interpret the file contents using Pythons Sniffer[1] implementation to analyze the table characteristics, such as the data type distribution. While the work covers only CSV tables, it emphasizes the need for more advanced parsers. Additionally, it provides us with insights, which we can take into account when developing our solution. We include Pythons Sniffer into the baseline for our evaluation.

The paper published by Christodoulakis et al. deals with the table discovery in CSV files using a set of weighted fuzzy rules that exploit column patterns[12]. The weights were trained on a data set collected from open data portals containing governmental data. The paper focuses on the table discovery and row classification tasks rather than file parsing. For the latter, the authors also use the standard Sniffer module of Pythons CSV library in a pre-processing step. After detecting the table boundaries, they classify lines as context, header, sub-header, data, footnote, and other. While the authors optimized their approach for CSV tables, we could imagine that, in theory, Pytheas also translates to tables in ASCII if adapted accordingly. As an interesting side note, the authors state that they did not encounter horizontally stacked tables. Furthermore, they focused on tables, where the attributes are contained in columns. While the line parsing was

[1] `https://docs.python.org/3/library/csv.html#csv.Sniffer`

not the focus of their work, we use the approach as a baseline for the table range detection.

In a paper published by Pyreddy et al. in 1997 the authors propose an approach to detect text lines containing tabular structures represented in ASCII[13]. The approach consists of two steps. The first step deals with table extraction using a so-called Character Alignment Graph, which relies on whitespace alignments in continuous lines. The second step tags potential table lines as one of captions, column headings, or legends based on heuristics. Lastly, it uses the tags and further heuristics to revalidate their previous classification. On the downside, it relies solely on the structural features of tables and does not take the cell content into account. From their work, we learn that whitespace alignments in continuous lines are important for ASCII tables. This observation is backed up by additional related work, such as [14] and [15].

Silva et al. published a survey about table recognition approaches for ASCII tables surrounded by other elements such as text[16]. They separated the table recognition problem into six sub-tasks, namely Location, Segmentation, Functional Analysis, Structural Analysis, and Interpretation. For each sub-task, they do not only state an extensive list of related work but also present an own approach. The *segmentation* deals with the interpretation of table lines and the identification of cells. They propose a solution that works on individual lines and searches for subsequent spaces, for which they apply different rules depending on whether the separated cells belong to numbers or strings. In uncertain situations, the algorithm may leave a flag for the user asking for their attention. While the proposed solution still requires the user to interact with the software, it is one of the few approaches dealing with the segmentation of ASCII tables into cells.

A machine learning approach for the parsing of CSV table files was proposed by Saurav et al.[17]. Their solution consists of two phases: In the first phase, the algorithm finds ways of interpreting the file content, which then is scored in the second phase using logarithmic regression. By heuristics, the number of candidates that need to be rated is reduced, improving the performance. Most of the features used for the logarithmic regression are related to the structure of the resulting table candidates. Yet, the algorithm does not take into account the cell values. Additionally, it does not cover all variants of CSV table representations. In the end, the paper serves as an example for using logarithmic regression to tackle the problem of parsing CSV tables.

The papers [13][16][17] have in common that none of them make use of the cell content when processing. Due to time constraints, we cannot evaluate the parsing accuracy of each solution individually. Thus, we represent the mentioned approaches in our baseline by a single solution, which employs structural features and ignores cell contents.

Döhmen et al. noted that existing CSV parsers make decisions during the file parsing process sequentially, which they suspect to impact the overall quality negatively. They propose a solution that makes decisions about sub-criteria as late as possible, trading run time costs against parsing quality[18]. Besides the CSV parsing, they cover the file encoding detection, table normalization, and table area detection. Each of these steps uses heuristics to produce multiple hypotheses. Afterward, the algorithm passes them down a tree structure to be ranked by a support vector machine. For the ranking, they use several features based on the table structure and cell data types. While they include a stage dealing with the table area detection, they do not support multiple tables to be present within a file. Additionally, they test only a fixed set of CSV variants. They published an implementation of their approach, which we include into our baseline when comparing the parsing accuracy.

In "Wrangling messy CSV files by detecting row and type patterns" Burg et al. introduce a novel data consistency measure to correctly parse CSV files[19]. The consistency measure consists of the row pattern score and the data type score. The row pattern represents the column count per row yielded by an interpretation. 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. As the authors noted, it can become problematic if many of the cell data types are unknown. We use the implementation, which the authors made publicly available, in our baseline.

A summary of typical data preparation problems is provided in the paper published by Nazabal et al.[20]. The authors distinguish between three groups of problems: data organization, data quality, and feature engineering. As outlined in their work, parsing file contents represents the first sub-problem of the data organization. However, many open problems need to be tackled in the field of data preparation, i.e., data integration and data transformation. The quality of the results is of great importance as the following tasks of the data wrangling pipeline rely on them.

3 Foundations

Numerous variations of table definitions make it necessary to recapitulate the underlying table components and concepts. Based on our definition of data tables in Section 3.1, we introduce the table formats CSV and ASCII in Section 3.2. Section 3.3 demonstrates the difficulties of interpreting tables given in such formats like they have been intended by the content creator. Finally, we define the table extraction problem in Section 3.4.

3.1 Data tables

Tables are used to deliver information in a structured way. Data tables depict a set of entities, where all entities are described by the same set of attributes. The smallest component of a table is a *field*, also referred to as cell. Each field represents an attribute value of some entity. Multiple fields are visually structured within a grid, which is represented by matrix C having the dimensions $m \times n$. Let $\alpha = C_t$, the cell value in the i-th row in column j can now be accessed by $\alpha_{i,j}$. To return the fields of the i-th row or j-th column we define $ROW_i = \{\alpha_{i,j} | 0 \leq j < n\}$ and $COL_j = \{\alpha_{i,j} | 0 \leq i < m\}$, respectively.

We divide the table into a *header* and a *data* part. The header part is optional. It covers the first h rows and contains the attribute names. The header can be retrieved using $header : C_t, h_t \rightarrow [ROW_i | 0 \leq i < h_t]$. The remaining $m - h$ rows belong to table's data part. Each row in the data part represents one entity. We refer to these rows as *records*. Similarly to the header, we define the function $data : C_t, h_t \rightarrow [ROW_i | h_t \leq i < m]$ that returns the data part. All m rows have exactly n fields, where n is the number of attributes that are used to describe the entities. The attribute order matters, so that all values within a column belong to the same attribute.

Definition 3.1.1 (Data table)**.** A data table t is defined by its content matrix C_t of $m \times n$ and its number of header rows h_t with $0 \leq h_t < m$ and $m \geq 2 \wedge n \geq 2$.

Table 3.1 depicts a data table that has four columns and four rows. The rows can be further separated into one header row and three records. The entities

ID	Name	Date of birth	Place of birth
246	Willis, Bruce	05/19/1955	Germany
1553725	Lutz, Kellan	05/15/1985	U.S
2442289	Carano, Gina	04/16/1982	U.S

Table 3.1: A data table depicting actors of the movie "Extraction" taken from IMDB[1].

represent actors, for which the id, name, date of birth and place of birth is given.

We expect column values to share the same data type. Consequently, we exclude transposed tables from our definition of data tables. Tables 3.1 and 3.2 show a horizontal ($m = 1$) and vertical ($n = 1$) list, which we do not recognize as tables. In spreadsheets it is common to use cells that span multiple rows or columns. However, we define tables to not include such cells, as they are impractical for data tables.

246	1553725	2442289

Figure 3.1: A horizontal list

246
1553725
2442289

Figure 3.2: A vertical list

3.2 Table formats

We identified two table formats being used for persisting data tables into plain text files. The main difference between both is the way how fields are delimited. Comma-separated-values tables (CSV) use a set of characters to delimit rows and cells. ASCII tables delimit columns by using layout features, such as white-spaces. In the following, we use the data table depicted in Table 3.2 to explain both table formats in detail.

Name	Minute	Quote
Victoria		N\A
Harry	40	"I like complicated. Easy's boring."

Table 3.2: Example table containing quotes from the movie "Extractable"

[1] `https://www.imdb.com/title/tt4382872/`

3.2.1 CSV tables

CSV is one of the most used file formats for sharing tables. It was first supported by IBM in 1972[10]. Surprisingly, its first official specification was published 33 years later in 2005 as part of the request for comments specification (RFC) 4180[9].

The authors of RFC 4180 define CSV files to be text-based, where each line represents a data record. Optionally, the first line might represent the header of the table. A row is transformed into a text line by delimiting fields by a special character, called a *delimiter*. If a cell includes the delimiter character or the newline character, the cell value must be put into quotes using a *quotation* character. Lastly, an *escape* character is required for cases where a quote character appears within quoted field values. The character also escapes itself if it appears within the cell. To interpret a CSV table, the delimiter is always required. Depending on the cell values, the quotation and escape characters might be omitted. The term *dialect* refers to the used delimiter, quotation, and escape characters[9].

Definition 3.2.1 (Dialect)**.** A dialect is used to parse lines belonging to CSV tables. We define a *dialect* as triple $\langle d, q, e \rangle$ representing the strings used as delimiter, quotation, and escape.

The dialect components are typically non-alphanumeric values. Figure 3.3 shows Table 3.2 in CSV format. The dialect uses comma as a delimiter, double quote as a quotation, and backslash as an escape. As we can see, the quotation character can also be present if the cell value does not contain the delimiter.

```
"Name","Minute","Quote"
"Victoria",,N\A
"Harry",40,"\“I like complicated. Easy’s boring.\”"
```

Figure 3.3: Content of Table 3.2 represented in CSV. The dialect is defined by the delimiter (blue), quotation (orange) and escape (violet).

While there exists a suggestion for characters that should be used in a dialect, programs still handle CSV files differently. The RFC document comments: “[...] there is no formal specification in existence, which allows for a wide variety of CSV files.”[9]. Within the specification document, they define CSV by referring to the dialect that “seems to be followed by most implementations”[9]. The proposed dialect is like the one shown in the example, except that it uses the double-quote character as the escape character. For clarity reasons, the dialect recommended by the RFC 4180 is show in Table 3.3[9].

delimiter	,
quotation	"
escape	"

Table 3.3: Recommended dialect in RFC 4180[9]

The existence of different dialects would not be an issue if the proper one would be stored along with the file. However, this is not the case which is problematic when attempting to interpret the text content back to a table. One reason for this could be the time when CSV was first implemented. In 1970 people were satisfied not documenting the used dialect, as they could still infer it by looking at the data.

There have been attempts to overcome this issue by extending the specification. W3C dedicated a working group to the topic "CSV on the Web". They propose the delivery of an additional file in JavaScript Object Notation (JSON), which contains information about the used dialect and the schema[21]. CSVY is a similar development, which stores such information as a YAML (Yet Another Multicolumn Layout) meta block at the beginning of the file[22]. Both proposals would require code changes to existing software. No official numbers are backing their distribution.

3.2.2 ASCII tables

Tables that align columns visually without delimiting characters are called ASCII tables. They are better to read for humans as the information is not as condensed as in CSV files. As with CSV tables, one record per line is assumed.

Fields are separated by white space so that no two values coming from two different columns interfere horizontally. Two columns must therefore be separated by at least one whitespace character. To transform a line of an ASCII table into its fields, we cannot simply divide the line by using an fixed number of whitespace characters as delimiter. This would not work, as empty fields would be lost. Instead, a set of *column boundaries* is required to transform an ASCII table into its fields. Each boundary defines the inclusive start and exclusive end of a column based on the character index. The indexes are chosen to be as tight as possible.

Definition 3.2.2 (Column boundary)**.** To split an ASCII line into its fields, a set of column boundaries B is required. We define a column boundary b as interval $[start, end)$ representing the inclusive start and exclusive end of a column.

```
 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 34 35 36 37
Name       Minute    Quote
====       ======    =====
Victoria             N/A
Harry           40   I like complicated.
```

Figure 3.4: The ASCII table representation of Table 3.2 - columns are aligned by layout

Figure 3.4 shows the ASCII representation of Table 3.2. To transform the ASCII table back into its underlying data table, the following column boundaries must be applied on all lines: $\langle 0, 8 \rangle$, $\langle 10, 16 \rangle$, and $\langle 19, 38 \rangle$.

ASCII tables support the inclusion of styling information, such as borders or horizontal lines. Vertical border lines, i.e., displayed by the pipe symbol (|), are already covered by the CSV table format as the border symbols can be used as the delimiter. Styling rows, used for underlining headers or separating tables, exist to make the table structure clearer to the human reader, but do not deliver content. We therefore distinguish between three line types: solid lines, helper lines and content lines.

We define the class of *solid lines* to include all lines, whose content includes only non-alphanumeric characters excluding any whitespace. These lines are typically used for separating the table header from the data part or to highlight table ends. *Helper lines* are similar to solid lines but include whitespace. These lines are typically used for the underlining of a header and might indicate column widths. The class of *content lines* represents the inverse of solid lines, and covers all lines with at least one alphanumeric character.

line type	example			whitespace	alphanumeric
solid	`==========`	`==========`		no	no
content	`Name`	`Minute`	`Quote`	yes	yes
helper	`====`	`======`	`=====`	yes	no

Table 3.4: Lines are classified into solid lines, content lines, and helper lines.

Table 3.4 shows examples for each line type. The solid line consists only of the non-alphanumeric *equal sign*, repeated 20 times. The content example contains multiple alphanumerical characters and spaces. The helper line is represented by the *equal sign* and *space*. Helper lines and solid lines are omitted as data tables to not support such concept.

3.3 Table extraction challenge

While ASCII tables are easier to read for humans, CSV tables are optimized for machine readability and smaller storage sizes. We refer to the super-group containing dialects and column boundaries as *parsing instructions*. For both table formats, the parsing instructions are not part of the resulting file. In this section, we want to highlight the difficulty resulting from the lack of parsing instructions due to multiple valid ways of interpreting lines.

```
K  Lutz        05/15/1985    U.S
G  Carano      04/16/1982    U.S
Bruce Willis                 Germany
```

Figure 3.5: Modified version of Table 3.1

Figure 3.5 shows an ambiguous text excerpt allowing for multiple ASCII interpretations. When looking at the first two lines, we might split each line into four fields (first name, last name, date of birth DOB, place of birth POB). After seeing the third line, multiple scenarios become possible. The first option is to think of the third row to not be a table row at all. A second option would be to interpret the line to belong to the table above. But, this implies to merge the first two columns. For the record belonging to the actor Bruce Willis, the value for DOB would be empty. Without having access to semantic or context information, the computer needs to consider both scenarios.

Similar effects can be observed for tables stored in the CSV format as the number of dialects depends on the number of non-alphanumeric characters appearing within the content of a line. The text `"Willis, Bruce";05/19/1955;"Germany"` yields seven valid dialects emphasizing how accidental such cases occur. Table 3.5 shows all possible dialects for that record.

delimiter	quotation	escape	column count
␣	n/a	n/a	2
"	n/a	n/a	5
"	;	n/a	5
,	n/a	n/a	2
/	n/a	n/a	3
;	n/a	n/a	3
;	"	n/a	3

Table 3.5: Valid dialects for `"Willis, Bruce";05/19/1955;"Germany"`

3.4 Problem statement

By solving the table extraction problem, we aim to decrease the time data scientists spend with data wrangling. Given a plain text file containing one or more tables, the goals are to extract each table using their correct parsing instructions and store them following RFC 4180. Figure 3.6 illustrates the expected input and desired output for an example.

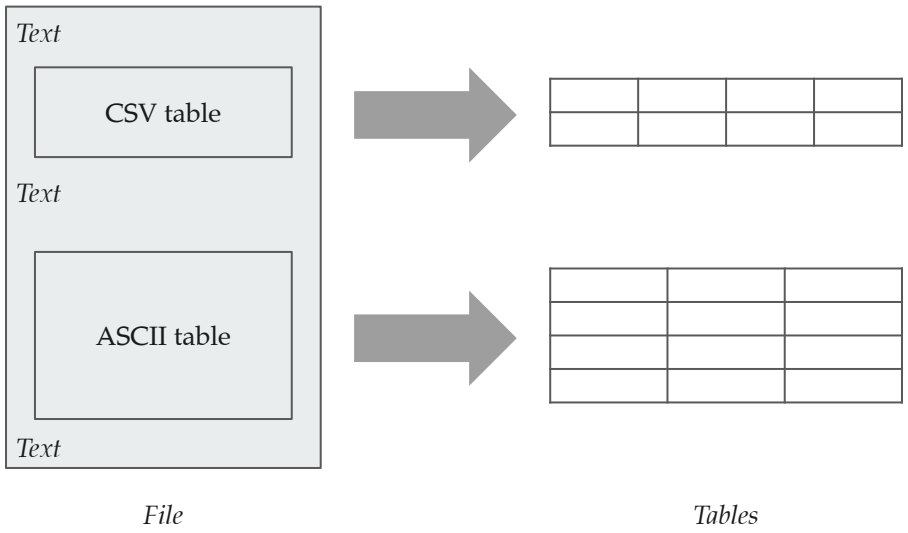

Figure 3.6: Input and output of the table extraction problem

In the following, we define the table extraction problem for plain text files. The content of a plain text file is represented by a set of lines L. A line may belong to one of the tables contained in the file. We denote the set of tables as T. In addition to the data table properties described in Section 3.1, we include the table range range r_t represented as interval $[from, to]$ where *from* and *to* are line indexes.

The function $detect(l)$ returns a set of possible parsing instructions for line l. A parsing instruction *instr* is a triple, including the type *type*, the column boundaries B, and the dialect *dialect*. The type can be 0 (ASCII) or 1 (CSV).

Let $parse(l, instr)$ be the function that applies a given parsing instruction *instr* on line content l. We call the result of the function *interpretation*. An interpretation *INTER* is a vector of strings, where each string represents a field value within a table row. Table 3.6 summarizes the introduced signatures.

Signature	Description	Range/ Format
	file	
L	line-wise file content	string vector
T	tables appearing within the file	set of tables
	table t	
Ct	content matrix	string matrix of $m \times n$
h_t	number of header rows	$0 \leq h_t < m$
r_t	range	$[from_t, to_t]$ with $0 \leq from_t \leq to_t < \|L\|$
	line l	
$detect(l)$	function returning all parsing instructions detected in line l	$l \rightarrow INSTR$
$parse(l, instr)$	function applying parsing instruction $instr$ on line l	$l, instr \rightarrow INTER$
$instr$	parsing instruction	$\langle type, B, dialect \rangle$ with $type \in \{0, 1\}$
$INTER$	interpretation	string vector

Table 3.6: Signatures used for the problem definition

The problem of table extraction can now be described as three sub-problems:

1. For each line l, find all valid parsing instructions for ASCII and CSV tables.
2. For each line l, determine the parsing instruction $instr$ so that $parse(l, instr)$ returns the correct interpretation. An interpretation is correct, if it produces fields exactly like in the underlying data table. Return ϵ if a line l is not part of a table.
3. For each table t, return its line range r_t.

Existing work in this field focused on the extraction of CSV tables, as outlined in Chapter 2. There are approaches handling the dialect detection and table range selection separately, however there is no combined solution. We are interested handling more complex files by including the support for:

- CSV and ASCII tables
- CSV dialects deviating more from RFC 4180
- Files with multiple, vertically stacked tables
- Files containing surrounding text

For simplicity reasons, we assume cells not to contain line breaks. We consider the detection of the file encoding not to be part of the problem and assume UTF-8

as specified in RFC 4180. As per our definition of data tables, we expect tables to have at least two rows and two columns and no spanning cells. As the input files are static, we do not consider scalability to be a big concern. Lastly, we do not expect the algorithm to correct any faults already present in the source document.

4 The ExtracTable Algorithm

We propose an algorithm that exploits one of the main characteristics of data tables, namely the data type consistency within columns, to tackle the problems described in Section 3.4. The algorithm first detects all valid parsing instructions for all lines and selects the parsing instructions that yield the most consistent data type pattern afterwards. Figure 4.1 depicts the six steps of the algorithm's workflow.

Figure 4.1: Workflow of ExtracTable

Firstly, the algorithm reads the file line-wise from disk. For each line, it detect valid dialects for CSV tables and possible column boundaries for ASCII tables. We explain the detection of dialects in Section 4.1.1 and the detection of column boundaries in Section 4.1.2. After applying the found parsing instructions to the lines, the resulting interpretations are passed to the next step. Section 4.2 describes the data type pattern extraction, which is done for every field value of all interpretations. The extracted information is used in the next step, which handles the building of table candidates. In Section 4.3 we illustrate how table candidates are build based on the data type compatibility of interpretations belonging to contiguous rows. In the last step, the algorithm selects a final set of tables and stores them in CSV format following RFC 4180. The details of the selection process are describes in Section 4.4. We discuss the impacts of adding more flexibility to ExtracTable allowing to tackle more complex files in Section 4.5.

4.1 Detecting parsing instructions

After reading a line from the file, the algorithm detects the parsing instructions for both table formats individually. As explained in Section 3.3, many parsing instructions need to be considered. The pruning of invalid detections is important as the number of interpretations affects the run time of the subsequent steps. Before the algorithm detects any dialects or column boundaries, it first pre-processes each line and classifies them as content lines, helper lines, or solid lines

by following the definitions introduced in Section 3.2. The algorithm prunes solid lines, as they neither deliver content nor help detect correct column boundaries of ASCII tables. In Section 4.1.1 we explain how the dialect detection for content lines works. We describe the algorithm for finding column boundaries in content and helper lines in Section 4.1.2. The algorithm applies the detected parsing instructions to extract the resulting interpretations for each line. Leading and trailing white spaces are trimmed from all field values. Only the interpretations of content lines are passed to the following field pattern extraction.

4.1.1 Detecting dialects

As defined in Section 3.2, the components of a dialect are non-alphanumeric values. The most trivial approach would take the content of a line r_l and detect the non-alphanumeric characters first. As a second step, the characters are used to build dialect candidates using different combinations. A CSV parser then validates the resulting dialects.

However, this approach is inefficient, which we demonstrate by the example line:

```
"Willis, Bruce";05/19/1955;"Germany"
```

The line contains five non-alphanumeric characters: double quote, comma, space, semicolon, and slash. Before calculating the number of combinations, we need to recall that the delimiter is essential and must be different from the quotation and escape. The quotation and escape characters can be the same. They can even be absent (ϵ). Following the requirements, this would allow for 125 dialect candidates that need to be checked by a parser. Doing all these checks can be time-consuming, especially as only a minority of dialects are valid. The example has only seven dialects that yield a result. The naive approach does not account for dialects consisting of multi-character components, which aggravates the effect.

We propose a solution that overcomes this issue by detecting valid quotation and escape characters on the fly. By implementing a custom parser, we can avoid re-running all tests on the whole line. We add the option to specify a set of strings P_{del} (delimiter exclusion list) that should be ignored for delimiters. Additionally, a maximum character length P_{mdl} for delimiters, quotation, and escape can be defined to limit the number of potential dialects. The dialect detection consists of two steps: the *detection of delimiters* and the *detection of quotation and escape-characters*.

To detect potential dialects, consecutive alphanumeric characters and excluded characters within l are replaced by a placeholder character first. Afterward, the algorithm splits the resulting string by the placeholder character, yielding a list

of delimiter sequences and empty values. As longer delimiter sequences might hide shorter ones, all substring combinations are built and appended to the list. Values that are empty or whose lengths exceed the maximum length P_{mdl} are removed. Finally, the delimiters are sorted by their character length and value. Table 4.1 provides the intermediate results of the dialect detection for the example line.

Representation	Value
Input	"Willis, Bruce";05/19/1955;"Germany"
Placeholder	"s, s";s/s/s;"s"
Delimiter sequences	" ␣, "; / / ;" "
Expanded and sorted delimiters	␣ " , / ; "; ,␣ ;"

Table 4.1: Detection of possible delimiters (delimiters are marked gray for readability reasons)

For the second step, we explain how we design our custom parser. In general, the parser is based on the grammar stated in RFC 4180[9, p.2]. Our algorithm uses a depth-first search to find all valid dialects and is shown in Algorithm 1.

Algorithm 1: Detecting quotation q and escape e on the fly

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

```
Def parse(l, d, q, e, cursor):
    cursor=0
    tl=trim(l)
    while cursor < |tl| do
        ⟨class, length⟩ = get_class(tl, cursor, d, q, e)
        if class = "error" then
            return ϵ
        if q = ϵ ∧ class = "content" ∧ ¬isalnum(tl_cursor) then
            parse(tl, d, tl_cursor, ϵ, cursor)
        if q ≠ ϵ ∧ e = ϵ ∧ class = "content" ∧ ¬isalnum(tl_cursor) then
            parse(tl, d, q, tl_cursor, cursor)
        update_parser_state(class)
        cursor = cursor + length
    dialects = dialects ∪ ⟨d, q, e⟩
dialects = []                          // square brackets denote list
parse(l, d, ϵ, ϵ, 0)
return dialects \ {ϵ}
```

The parser receives the line content l and a sequence d from the list of possible delimiters as input. At first, it removes any leading or trailing whitespace. It then tries to parse the line using the dialect $dialect_0 = \langle d, \epsilon, \epsilon \rangle$. Therefore, the parser iterates over the character positions of the trimmed line content tl. For each iteration, it checks whether the remaining line starts with a component from the given dialect. If there is a match, the parser acts according to the provided grammar. Otherwise, the algorithm treats the next character as content.

However, at this point, we add a trigger. If the content character at the current position is not alphanumeric, we could interpret the character as a quotation or escape. A second parser instance is started using the remaining line content tl and the updated dialect $dialect_1 = \langle d, q, \epsilon \rangle$ with $q = tl_{cursor}$.

To make it more comprehensible, we take the example line from before using $d =$;. The first parser instance processes the whole string using the dialect $dialect_0 = \langle ;, \epsilon, \epsilon \rangle$. It treats the first character, which is a double quote, as content. As this character is not alphanumeric, it starts a second parser instance using $dialect_1 = \langle ;, ", \epsilon \rangle$. The quotation is set before the escape, as the latter occur only within quoted fields and thus depends on the quotation to be present in the dialect. The first parser that uses $dialect_0$ still continues processing the rest of the line. It starts additional instances when finding other non-alphanumeric content characters. When the second instance $dialect_1$ discovers a non-alphanumeric content character, it treats it as escape and starts a new instance using $dialect_2 = \langle d, q, e \rangle$ with $e = tl_{cursor}$.

In our example, this happens when reaching the eighth character, which is the comma between the words *Willis* and *Bruce*. The new instance tries the dialect $dialect_2 = \langle ;, ", , \rangle$. Though, it stops after the next character, as the escaping of content is invalid. The parser stops immediately if it encounters any of the invalidities listed below:

- Escaping content
- Ending quotes are missing or not followed by a delimiter or newline
- Trailing escape at end of line

As a side note: the above algorithm has been slightly simplified for comprehensibility reasons. In reality, the algorithm does not consider only single characters for quotation and escape. Instead, it also looks at the next P_{mdl} (max dialect length) characters to launch additional instances. While creating new instances, the parser also keeps track of the already started ones to avoid processing the same dialect twice.

If the parser can process the whole line without errors, it found a legitimate dialect. Finally, the parser returns all valid dialects.

4.1.2 Detecting column boundaries

As per our definition of ASCII tables, two columns must be separated by whitespace having at least one space character. By detecting the resulting *vertical lines*, we can infer the boundaries for all columns. In contrast to CSV tables, it is impossible to find vertical lines by looking only at individual lines. The proposed solution is independent of whether the file has been read from the start to the beginning or in reverse. The pseudocode shown in Algorithm 2 introduces the general workflow of the column boundary detection.

Algorithm 2: Column boundaries detection

Input: File content L and file width *width* calculated by Formula 4.1
Output: Column boundaries

```
1  counter={w → 0|0 ≤ w < width}        // curly brackets denote dict
2  tables=[]                            // square brackets denote list
3  boundaries=[]                 // [bit_0, bit_1, ..., bit_{width-1}] with bit_w ∈ {0,1}
4  for k ← 0 to |L| do
5  |  closed=[]
6  |  bitmap=transform(l_k, width)
7  |  if ∃char ∈ l_k : char ∉ WS then                    // WS...whitespace
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 ∃t ∈ |tables| : closed_t = 1 ∨ ∃w ∈ 0..width : counter_w = P_mrc then
16 |  |  |  tables=tables ∪ start_table(counter, k)
17 for t ← 0 to |tables| do
18 |  boundaries=close_table(t)
19 return boundaries
```

In the following, we describe the algorithm that finds vertical lines efficiently. We explain each step using the example depicted in Figure 4.2. The variable k depicts the line index.

Transform line content into bitmap transform(l, *width*)
As a first step, the algorithm 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.

	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 34 35 36 37
$k = 0$	Name Minute Quote
$k = 1$	==== ====== =====
$k = 2$	Victoria N/A
$k = 3$	Harry 40 I like complicated.

Figure 4.2: Example table ($width = 38$)

The number of required characters depends on the tab size. We use a tab size of eight, which is the default used in Pythons `expandtabs`-function. All whitespace characters then are replaced with the number one (*True*) and any other character with zero (*False*). Lines are padded using multiple ones to the width *width* of a file, which is calculated using Formula 4.1.

$$width : L \rightarrow \max(|l_k| : 0 \leq k < |L|) \tag{4.1}$$

Figure 4.3 shows the bitmap representation for the first line ($k = 0$) of Figure 4.2.

	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 34 35 36 37
$k = 0$	Name Minute Quote
$k = 0$	0000111111000000111000001111111111111

Figure 4.3: Bitmap for the first line of Figure 4.2

After transforming the line content into a bitmap, the algorithm searches for *vertical lines*. Subsequent lines, where $bitmap_w = 1$ for the same character position w, form a vertical line at w. Consecutive vertical lines are grouped into *spacers*. We define a spacer as a set of consecutive, ascending sorted *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 ($width - 1 \in indexes$).

Definition 4.1.1 (Spacer)**.** A spacer groups indexes of subsequent vertical lines. Therefore, we define a spacer by its set of character positions *indexes*. Some spacers are significant ($essential = 1$), while others are insignificant ($essential = 0$). Significant spacers are mandatory for tables.

The first line yields three spacers, shown in Table 4.2.

indexes	significant
4, 5, 6, 7, 8, 9	1
16, 17, 18	1
24-37	0 (trailing)

Table 4.2: Spacers of the first and second lines ($width = 38$)

Append discovered tables start_table(*counter*, *l*)
In the next step, the algorithm checks whether it discovered a new table. It found a new table if there is at least one vertical line spanning P_{mrc} (min row count) text lines. The threshold P_{mrc} is specified by the user. The new table is defined by its starting line index and a set of spacers. The starting line is calculated by $k - P_{mrc} + 1$.

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. Figure 4.4 shows the bitmaps for the first two rows of the example table. Table t_0 starts in line $k = 0$ and includes the three spacers shown in Table 4.2.

	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 34 35 36 37
$k = 0$	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 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 1 1 1 1

Figure 4.4: Bitmap for the first two lines of Figure 4.2

Update existing tables 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 a insignificant spacer was discontinued, the algorithm removes it from the set of table spacers. The interruption of all indexes belonging to a significant spacer marks the end of the table regardless whether other significant spacers have been continued.

Close tables close_table(*t*)
If the table that gets closed 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.

To convert a set of spacers to a set of column boundaries, all spacer indexes are unioned first. The resulting indexes are subtracted from the set representing all character positions ($\{0, 1, ..., width\}$). The indexes present in the set difference form multiple blocks of consecutive numbers. The algorithm obtains the set of column boundaries by taking the lowest and highest index for each block. We provide an example at the end of this section.

After processing the third line, all spacers of t_0 still exist, but the first one shrinks. It now covers the indexes 8-9 as shown in Figure 4.5. The algorithm does not find new tables.

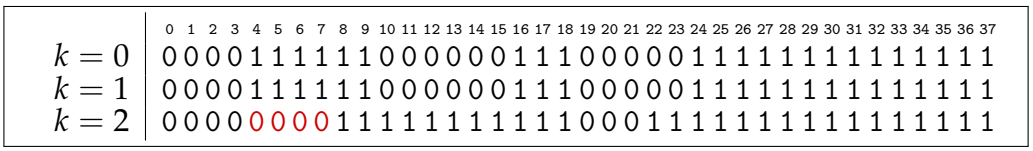

Figure 4.5: Bitmap for the first three lines of Figure 4.2

The algorithm repeats the described procedure for the remaining text lines. The result is shown in Figure 4.6.

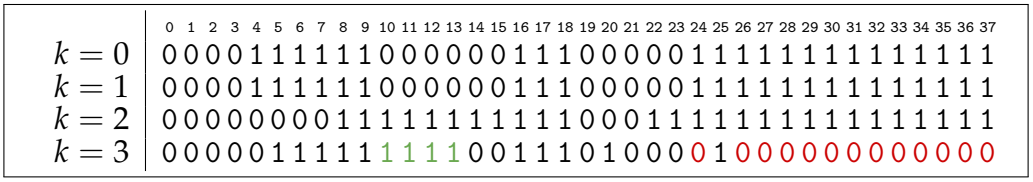

Figure 4.6: Bitmap for Figure 4.2

When reaching the end of our example, the algorithm updates the last spacer of table t_0 by shrinking it into a smaller, insignificant one. Additionally, a vertical lines spanning P_{mrc} rows were found. A new table t_1 is created using the spacers, shown in Table 4.3. We note that the spacers are quite similar to the ones of t_0.

indexes	significant
8, 9, 10, 11, 12, 13	yes
16, 17, 18	yes
25	no (single index)

Table 4.3: Spacers of table t_1

We consider a new table to be *eager* if there exists another table fulfilling the following two requirements: (1) The new table and the existing table share the same number of spacers (ignoring leading and trailing ones). (2) When comparing the spacers of the existing table pairwise to the new table, ordered by the lowest indexes, all spacer pairs have at least one intersecting index. The algorithm drops eagerly detected tables without updating any lines.

After processing the last line of a file, the algorithm closes all tables and stores the column boundaries along the text lines.

In our example, only one table is left. The spacers of table t_0 are converted into column boundaries. When using $P_{msr} = 5$, the insignificant spacer 25 is dropped as the table has only four rows. The remaining two spacers covering the indexes $8-9$ and $16-18$ are subtracted from the set of character positions, yielding the indexes $0-7$, $10-15$ and $19-37$. The column boundaries for table t_0 are: $[0, 8)$, $[10, 16)$, and $[19, 38)$. They are assigned to all four table lines.

4.2 Extracting field patterns

In the previous steps, the algorithm collected all valid parsing instructions for each line and returned the resulting interpretations for them. We want the algorithm to select the interpretations based on their data type consistency. Therefore, it needs to determine the data types for each field first. In this section, we explain how the algorithm describes the data type for a given value.

We use three atomic components to describe the structure of a value: *number* (N), *string* (S), and *other* (O). The *string* class matches sequences of characters from the Latin alphabet (a to z) independent of whether they are in upper case or lower case. The *number* class covers all kinds of numbers, including signed/ unsigned, integers, floats, and numbers in scientific representation. The decimal separator of floating-point numbers and the optional thousands separator are recognized when represented by comma or dot. While we try to allow for a wide range of number formats, we consider values beginning with a sequence of zeroes not to be a single number rather than a series of numbers. The remaining class *other* matches everything that is not a number or string. Thus, it can match only sequences of non-alphanumeric characters. Algorithm 3 shows how to dismantle a value into its components.

We use regular expressions to distinguish among the different component types. The algorithm processes the input value v from left to right. For each iteration, it identifies to what class the beginning of the remaining substring belongs.

Algorithm 3: Field pattern extraction

Input: A string v representing the value of a field
Output: A sequence of atomic components

```
1 cursor=0
2 components=[]                          // square brackets denote list
3 while cursor < |v| do
4 |   ⟨component, length⟩=identify(substr(v, start = cursor, end = |v|))
5 |   components=components ∪ component
6 |   cursor=cursor + length
7 return components
```

The algorithm adds the detected type to a list and increments the cursor by the match length. A match always covers as much text as possible.

The regular expression for strings is simply an arbitrarily long sequence of the characters *a-z* and their uppercase version *A-Z*. The regular expression for numbers consists of three parts as shown in Table 4.4.

Part	Regular expression	Description
1	([-+])?	Signed numbers (optional)
2	([1-9]\d{0,2}([,]\d{3}){2,}([.]\d+)?)	Integer or float having multiple thousand separators (,)
	([1-9]\d{0,2}([.]\d{3}){2,}([,]\d+)?)	Integer or float having multiple thousand separators (.)
	([1-9]\d{0,2}[,]\d{3}[.]\d+)	Float having exactly one thousand separator (,)
	([1-9]\d{0,2}[.]\d{3}[,]\d+)	Float having exactly one thousand separator (,)
	(([1-9]\d*|0)(([.,])\d+)?)	Integer or float without thousand separator
3	([eE]([+-])?\d+)?	Scientific notation (optional)

Table 4.4: The regular expression used for the number component.

The first part captures the leading plus or minus of signed numbers. While the sign is optional, the second part is mandatory. Multiple expressions are required, as various number formats exist. We further distinguish the numbers by the amount of thousand separators so that the expressions are disjoint. The last part covers numbers given in scientific notation. The *other* component is simply the inverse of the string and number classes combined and does not need further specification.

Table 4.5 shows the data type patterns for some example values. The second instance emphasizes that string components do not include spaces. Space characters are treated like any other non-alphanumeric character and belong to the class *other*. The third and fourth rows show how leading zeroes are treated as individual numbers and are not part of the following number. We chose the last two examples to point out the limitations of the proposed solution. The patterns of both rows are different, yet a human may recognize them as file paths.

#	Value	Pattern
1	Gina	S
2	Gina Carano	SOS
3	1553725	N
4	001553725	NNN
5	/imdb/title/extraction.txt	OSOSOSOS
6	/imdb/title/extraction/fullcredits.txt	OSOSOSOSOS

Table 4.5: Data type patterns of example values

Splitting field values into their atomic components is required to compare the column data type consistencies of different table interpretations. However, incorrect interpretations can yield a high evenness, too. Thus, comparing the consistency of data types within columns is not enough. The algorithm needs additional information about how common data looks like to select the proper interpretations. If the algorithm would be able to recognize data types, it would not just be able to prefer known formats to unknowns, but it could also be a solution when dealing with patterns of different lengths, such as file paths or text.

We decided to extend the algorithm by detecting a pre-defined set of data types using regular expressions (REGEX). Before dismantling a value into its atomic components, the algorithm checks whether the field value fully matches any REGEX present in a list of known data types. In the case of a match, the algorithm assigns the index of the first matching expression to the *known* data type (K). If no data type matches the value, Algorithm 3 is applied. Yet, it is unfeasible to have a complete list of regular expressions covering all data types. Therefore, we have to make a choice which data types we want to support.

We built our set of data types from three different sources. First, we started by following a data-driven approach. We used a set of files from Mendeley, which are compliant to RFC 4180. By interpreting the files, we were able to extract the cell values. We applied Algorithm 3 onto each field value and returned the patterns for each file. After comparing the returned patterns across files, we

have been able to derive the most common ones. However, we have mainly detected common formats, like dates or currency numbers. As a second source, we searched for REGEX-libraries, which offer a high number of pre-built and tested regular expressions. The application RegexBuddy[1] includes such library. We picked data types that are not specific to a certain domain. We extracted the REGEX for dates, email addresses and IP addresses. Lastly, we also collected data types while working with the data at hand. We used related work to double-check whether we missed any typical data types. All of the 15 data types that we used are listed in Table .1.

When rating the consistency of tables, empty values (E) must be treated differently as they represent the neutral element within tables. Columns containing values of two different types are usually inconsistent. However, columns of data tables may contain empty values. Thus, the appearance of missing values in combination with another data type in a column has no negative impact on the overall consistency. Therefore, it is crucial to cover as many representations of empty values as possible. The list below includes everything that we consider to be empty. We assume that all variants are case insensitive.

- Empty string ϵ
- N/A or NA
- NaN
- Null
- Unknown
- A sequence of more than one question mark, dash, star, or number sign

Definition 4.2.1 (Pattern)**.** We define a pattern *PAT* as a vector of pattern components. A pattern component *pat* can be one of String, Number, Known, Empty, or Other.

Finally, the algorithm is capable of recognizing common data types and describing the structure of unknown ones. Table 4.6 shows the updated patterns of Table 4.5.

[1] `https://www.regexbuddy.com/`

#	Value	Pattern
1	Gina	K6 (Text)
2	Gina Carano	K6 (Text)
3	1553725	N
4	001553725	NNN
5	/imdb/title/extraction.txt	K12 (File path)
6	/imdb/title/extraction/fullcredits.txt	K12 (File path)

Table 4.6: Updated data type patterns of example values

4.3 Building table candidates

For all lines of a file, the algorithm receives several interpretations. Each interpretation consists of multiple fields for which the algorithm extracted their values and data types. By looking at the pattern representations, we observed that the compatibility of data types of adjacent lines is transitive. Thus, we assume that if two line pairs (l_{k-1}, l_k) and (l_k, l_{k+1}) are consistent, then l_{k-1} and l_{k+1} are also consistent. We introduce a score-based algorithm that exploits the observed transitivity to build *table candidates* in Section 4.3.1. The calculation steps of the used score are explained in Section 4.3.2.

4.3.1 Table candidate generation

Similar to the detection of column boundaries, the algorithm iterates over all lines and builds table candidates on the fly. It groups line interpretations into different bins using the compound of column count n and parsing instruction *instr* as bin key. Afterward, the algorithm compares the list of represented bin keys with the set of existing table candidates TC. A new table candidate tc is started upon the discovery of an unrepresented bin key. 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 the ExtracTable algorithm.

When comparing the bin keys of the current line with the existing tables, the algorithm also needs to consider a special case, which we want to elaborate with the example shown in Figure 4.7.

As we can see, the parsing instruction $instr = \langle 1, \epsilon, \langle \, , \, \epsilon \, , \, \epsilon \, , \rangle \rangle$ can be used for the first and last line. The second line, however, uses quotation for the second field. To correctly interpret the line, the parsing instruction $instr' = \langle 1, \epsilon, \langle \, , \, " \, , \, \epsilon \, , \rangle \rangle$ is needed.

```
246,Bruce Willis,05/19/1955
1553725,"Lutz, Kellan",05/15/1985
2442289, Gina Carano,04/16/1982
```

Figure 4.7: csv table containing lines with and without quotation

Following the strict matching as described would not yield the desired result. To overcome this issue, we extend our algorithm to look for so-called *fallback* table candidates. If no perfect matching bin key could be found, the algorithm searches for an existing table candidate with a *compatible* bin key. A compatible bin key must have the same column count and can only be found for csv tables. Despite starting a new table candidate using the original bin key, the fallback table candidates are updated, too. Table 4.7 shows which dialects are compatible.

Dialect	Compatible dialects
$\langle d, \epsilon, \epsilon \rangle$	none
$\langle d, q, \epsilon \rangle$	$\langle d, \epsilon, \epsilon \rangle$
$\langle d, q, e \rangle$	$\langle d, q, \epsilon \rangle$, $\langle d, \epsilon, \epsilon \rangle$

Table 4.7: Compatible dialects

The algorithm then adds the corresponding interpretations to the table candidates. However, before doing so, the algorithm checks whether the current line and the most recent row of the table candidate are compatible with regard to the *consistency score*. If the data types are consistent, the interpretation are appended. If the lines are incompatible, the algorithm starts a new table candidate. The new table candidate might be created twice: once with and once without using the previous one as a header. The row count of potential headers must not exceed P_{mhr} (max header rows) and they should not include any floats.

4.3.2 Consistency score

To determine whether two interpretations are compatible, we introduce a data type-based *consistency score*. Given two rows, the 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 primary measure for the consistency score. The *value uniformity* within columns is calculated to compare consistent tables.

In the following, we define the consistency score for tables. The score function receives the content of a table C as input. It aggregates the pattern consistencies and value uniformities of all non-sparse columns, as shown in Formula 4.2.

$$score : C_t \rightarrow \begin{cases} -1, & \text{if} |rich| = 0 \\ 0, & \text{if} |cons| < \lfloor \log_2(|rich|) \rfloor \\ \frac{|cons|}{|rich|} * \frac{1}{|rich|} \sum_{COL}^{rich} u(COL) & \text{otherwise} \end{cases} \tag{4.2}$$

The variable *rich* refers to all non-sparse columns. The subset of consistent non-sparse columns are referred by *cons*. The function u returns the *value uniformity* of a column. We explain each of them in greater detail below.

Some tables contain columns where all fields are empty. As such columns should not impact the table consistency, we decided to exclude them before calculating the score. Let $\alpha = C_t$ be the content of table t and $describe(\alpha_{i,j})$ be the function that returns the pattern *PAT* for the cell at (i, j). We then define the set of *rich* columns as a subset of columns that have more than one non-empty field in Formula 4.3.

$$rich = \left\{ COL_j | 0 \leq j < n, (\sum_{i=0}^{m} describe(\alpha_{i,j})) = \text{'E'}) > 1 \right\} \tag{4.3}$$

If all columns of a table are sparse, it is not possible to rate the consistency. In such cases, the *score*-function returns -1.

To determine the pattern-consistent columns the *homogeneity* of every column is calculated. The homogeneity *homo* was introduced by Guo et al. The authors of the paper describe the homogeneity of a column as "[...]the sum of squares of the proportions of each data type [...] present in that column"[23]. Formula 4.4 shows the calculation of the data type homogeneity. However, we modified the formula to exclude all empty fields as they represent the neutral element. The number returned by *homo* is between 0 and 1. Analogous to the consistency score, 0 means that the data types of a column are heterogeneous, while 1 indicates a column where all fields belong to the same data type.

$$homo : COL_j \rightarrow \sum_{PAT}^{\{describe(\alpha_{i,j}) | 0 \leq i < m\} \setminus \{\text{'E'}\}} \left(\frac{|0 \leq i < m, describe(\alpha_{i,j}) = PAT|}{m - |0 \leq i < m, describe(\alpha_{i,j}) = \text{"E"}|} \right)^2 \tag{4.4}$$

The set *cons* includes all columns, whose homogeneity exceeds one of the thresholds P_{mbs} (min block score) or P_{mcs} (min column score). Which threshold is being used depends on the use case. The algorithm uses threshold P_{mbs} when checking the compatibility of two rows, whereas it uses P_{mcs} to calculate the score of a whole table. We distinguish between these two as we want the algorithm to tolerate lower consistencies between two subsequent rows. However, it should be more strict when looking at the whole table. According to our experiments, it works well to require at least $\lfloor \log_2(|rich|) \rfloor$ pattern consistent columns to defer the table's overall consistency. Otherwise, the *score* for that table is 0.

$$cons = \left\{ COL_j | 0 \leq j < n, homo(COL_j) \geq P_{mcs} \right\} \tag{4.5}$$

To compare two consistent tables with each other, the third case of the *score*-function is executed, which takes the tables *value uniformity u* into account.

The value uniformity is defined for three different levels standing in a hierarchical relation: columns, patterns, and pattern components.

On the lowest level, we define the *value uniformity for each pattern component*. To determine the unity of *number* components, we compare how many numbers are integers and how many are floats by calculating the homogeneity based on both classes. A similar calculation is performed for *other* components, where the value homogeneity 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.

From the value uniformity of single pattern components, we can conclude the *pattern uniformity*. The smallest pattern component uniformity represents the uniformity of the whole pattern. We do not want the algorithm to prefer or punish empty fields, which occur in sparse tables. Thus, we ignore them when calculating the value uniformity as they should not have an impact.

On the highest level we aggregate the pattern uniformities for each column. A column may contain values belonging to different patterns, thus the fields of a column are grouped by their patterns first. Each pattern uniformity is then weighted by its occurrence ratio within a column, before returning the highest value as the column uniformity.

We demonstrate the calculation of the consistency score using Table 4.8. The table consists of three rows ($m = 3$) and four columns.

Firstly, the sparse columns are pruned. In the given table, the fourth column is sparse. Thus, all columns but the last one are included in *rich*, as defined in Formula 4.3. As $|rich| > 0$, the consistent columns are determined next.

246.00	Willis , Bruce	05/19/1955		N	K6	K4	E
1553725	Lutz , Kellan	05/15/1985		N	K6	K4	E
unkown	Carano, Gina	Pending	comment: todo	E	K6	K6	K6

Table 4.8: Exemplary table (left) and its data types (right)

The first column consists of two numbers and one empty field. Empty fields are excluded so that the column contains only numbers. The column is perfectly consistent and yields $homo(COL_0) = 1$. The same applies to the second column COL_1. However, the third column consists of two dates and one text field. The distribution translates to 67% of the column are dates and 33% of the column are text. The homogeneity is calculated by $0.67^2 + 0.33^2 = 0.56$. All columns, whose homogeneity exceeds P_{mcs} are members of the *cons* set. When choosing 0.5 as threshold, it follows that $cons = rich$. At least $\lfloor \log_2(|rich|) \rfloor = \lfloor \log_2(3) \rfloor = 1$ columns must be consistent, to proceed with the third case of Formula 4.2, which is true.

To return the final score, we need to determine the value uniformities for all columns. Again, empty values are excluded from the calculation. Therefore we need to consider only the number pattern when looking at the first column. The table has one number given as float (246.00) and one numeric given as integer (1553725). The homogeneity of this is $0.5^2 + 0.5^2 = 0.5$. As the pattern consists only of the number component, it is being used to represent the unity of the first column. The second column is perfectly uniform (1) as all values are known and belong to the type text. The third column consists of the two known patterns K4 and K6 representing dates and text. Known patterns always return a uniformity of 1. As the column consists of two different patterns, the maximum is taken, which is still 1. The resulting uniformities of the columns are $0.5, 1.0, 1.0$. Finally, the consistency score can be calculated using $score = \frac{3}{3} * \frac{0.5+1.0+1.0}{3} = 0.83$.

We have chosen this scoring design since the underlying data type homogeneity described by Guo et al. seems to work very well. The authors Burg et al. also follow a pattern-based approach for dialect detection. Yet, their proposed solution does not take full advantage of the recognized data types. In contrast to their work, we extended our score that accounts for the data type consistency within columns.

To ensure proper functioning of the consistency score, the data types need to be distinct. If two data types overlap, it could be possible that they are punished without reason. As a further development, we could also think of row-based features that might entail advantages when ignoring whole rows disturbing the table structure.

4.4 Selecting table candidates

Previously, we described how the algorithm collects all possible table candidates TC for a given file. In the final step, the algorithm selects a subset of table candidates so that their line ranges do not overlap. In this section, we describe how to map the table candidate selection problem to the *shortest path problem*.

A graph consists of vertexes V and edges E. An edge is defined by $\langle src, dst, dis \rangle$, where src is the source node, dst the destination node and $dis \in \mathbb{R}$ a number describing the distance between src and dst.

In the following, we explain how to transform the set of table candidates to a multi-edged directed acyclic graph (DAG).

The set of vertexes represents the line indexes of a file, so that $V = \{i | 0 \leq i < m\}$. Each table candidate represents one edge, using the first and last line index for src and dst respectively. The distance for a given table candidate is calculated using Formula 4.6. Lower distances represent larger and more consistent tables.

$$\begin{aligned} dist : tc \rightarrow & - score(data(C_{tc}, h_{tc})) * (m_{tc} - h_{tc})^2 \\ & - score(header(C_{tc}, h_{tc})) * (m_{tc}^2 - (m_{tc} - h_{tc})^2) \\ & - 0.0001 * \text{sgn}(h_{tc}) \end{aligned} \tag{4.6}$$

The function calculates the consistency scores for the header and data part. The individual scores are multiplied with the number of rows they cover taken to the power of two. When comparing tables of the same size and consistency, we favor tables having a header. Thus, we subtract a small constant if at least one header row exists.

Before mapping table candidates to edges, the algorithm prunes the ones not fulfilling the minimum size requirements of P_{mrc} and P_{mcc}. If the consistency of the data part of a table is below the threshold P_{mts} (min table score) they are discarded, too.

After the DAG has been filled, adjacent vertexes are linked. Each vertex pair $\langle v, v+1 \rangle$ is connected by an edge having a distance of $dis = 0$. The resulting graph is rather dense, but still acyclic as there are no edges $\langle src, dst \rangle$ with $dst \leq src$.

The following example illustrates the transformation of seven table candidates depicted in Table 4.9.

#	From	To	h_{tc}	$m_{tc} - h_{tc}$	$score(header(C_{tc}, h_{tc}))$	$score(data(C_{tc}, h_{tc}))$
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

Table 4.9: Table candidates of our example

Figure 4.8 shows the resulting graph. The gray circles denote the vertexes. The numbers at the edges represent the distances and the squared boxes are the table candidate indexes.

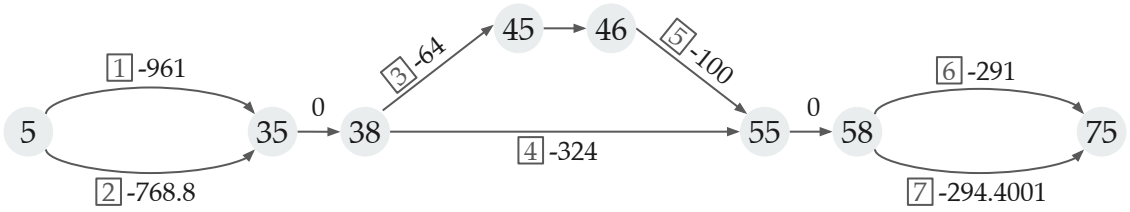

Figure 4.8: Table selection graph

Subsequently, the shortest path algorithm is applied. We decided to use the Bellman-Ford algorithm[24], which is capable of working with negative distances. The algorithm calculates the length of a node path by summing up the distances of the traversed edges. It returns the shortest path as a sequence of vertexes for which we need to retrieve the edges having the shortest distance.

When selecting the edges, multiple candidates may yield the identical *dis*, which can happen if table candidates are equally consistent and share the same dimensions. If such a situation occurs, we pick the best candidate according to our tie-breaking criteria. We prefer candidates having:

1. A higher ratio of recognized fields to total fields
2. A higher number of *rich* columns (as defined in Formula 4.3)
3. Lower patterns lengths
4. Lower number of columns

We briefly explain the impact of this sorting order.

Firstly, we prefer tables where more fields have been recognized as we are more confident in data that looks familiar. We calculate the number of recognized

fields by counting the number of cells having a pattern of length one, excluding the ones consisting only of the *other* component. The ratio is then calculated by dividing that number by the total number of fields. Secondly, we prefer a higher number of *rich* columns as we rather split a column too eagerly. We discourage higher pattern lengths, as we prefer simpler to complex values. Lastly, we prefer a lower number of columns, as we want to select table candidates having less sparse columns. Sparse table candidates might be a sign that the chosen parsing instruction was incorrect. The best table candidate 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.

Using the same example as above, we demonstrate three things when searching for the shortest path. We explain the key insights by traversing the graph from the lowest line index (left) to the highest line index (right). The vertexes representing the lines five and 35 are connected by two edges embodying the table candidates one and two. Both tables candidates have the same size, yet the distance differs as the data part of the first table is more consistent. Therefore, the Bellman-Ford algorithm selects the edge belonging to table candidate one. The second graph segment going from line 38 to 55 demonstrates how bigger tables are favored instead of multiple smaller ones. The table candidates three, four, and five all have the same characteristics as none of them has a header and all of them are equally consistent. Yet, the algorithm prefers the fourth table candidate as it yields a lower distance than the sum of the other two candidates The last segment ranging from 58 to 75 demonstrates, how table candidates containing a header defeat candidates without header, provided that the consistency of the header is comparable. In the end, the shortest path algorithm returns the table candidates one, four and seven as the selection for the given file.

Since we noticed that the table selection might take up to several minutes depending on the number of table candidates, we propose a heuristic: Instead of returning the optimal table selection, all table candidates are sorted by their distance and tie-breaking criteria in ascending order so that the longest and most consistent tables come first. It then takes the first table candidate into its final selection and removes it from the sorted list. All other table candidates whose line range intersect with the selected table candidate are pruned. The algorithm repeats this procedure until all table candidates have been removed from the list. Finally, the heuristic returns the set of selected table candidates. We discuss the performance impacts in Section 5.5.

4.5 Add-on: Supporting complexer tables

One goal of this algorithm was to develop a solution capable of handling more complex files. In the following, we want to discuss what would be necessary to support table candidates produced by different parsing instructions like the one shown in Figure 4.9.

```
246,Bruce Willis,05/19/1955
1553725;"Lutz, Kellan";05/15/1985
2442289, Gina Carano,04/16/1982
```

Figure 4.9: csv table containing lines using comma/ semicolon as delimiter

To extend the algorithm, we need to make three adjustments. The first change affects the building of table candidates. Previously, the algorithm grouped the candidates by parsing instruction and field count. Now, they are grouped only by column count. The format of a parsing instruction can still be used as an indicator for a good fit. Therefore, we propose the second change in the consistency score. The adapted score remains the same for the first and second cases. Only the third case is different when $|cons| \geq \lfloor \log_2(|rich|) \rfloor$. It takes into account the parsing format of all table rows by calculating the homogeneity for it. The result then gets multiplied with the original score.

As the line interpretations are grouped only by column count, multiple interpretations of a row may fit the same table candidate. Figure 4.10 illustrates an example of such scenario, where each line yields multiple interpretations having three columns. The first and third line can be interpreted in two ways, depending whether comma or slash are used as delimiter. The second line yields three interpretations. A table candidate having three columns can be produced by any of the twelve combinations.

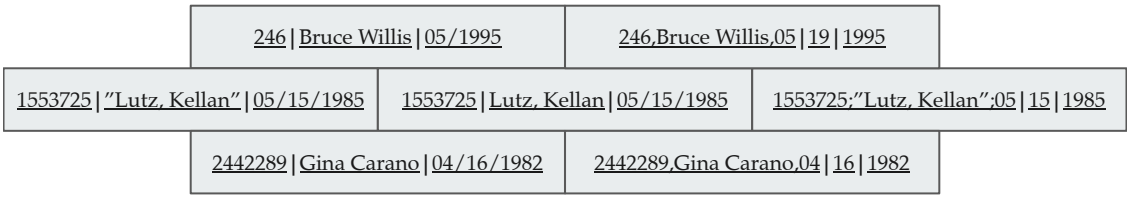

Figure 4.10: Possible interpretations yielding three columns

The algorithm needs to select the interpretations that yield the highest overall consistency. To make the optimal selection, all combinations need to be considered. Even if there are just two possible interpretations per row yielding the same column count, this would require 2^m calculations. For a table having 30 rows

this would result in $2^{30} = 1,073,741,824$. Thus, finding the optimal solution is impractical. It is inevitable to select the interpretations that are locally optimal.

We propose a solution that decides as local as necessary but as global as possible. Therefore, as the last change, when adding the interpretations to a table candidate, we suggest implementing the following *compacting strategy*. Whenever the number of combinations exceeds the threshold P_{csl} (compact size limit), an interpretation is removed from each row of the table candidate unless it has only one left. To select which interpretation should be removed, the consistency score for all combinations is calculated first. Based on the result, the algorithm keeps the best for each row and discards the worst one. By changing the threshold P_{csl}, the user can balance the run time against the algorithm's correctness. However, P_{csl} must be chosen such that the interpretation count of a single row does not exceed it.

We have been able to validate the correctness of the approach using synthetic examples. Yet, we did not find tables using different delimiters in our ground truth. Thus, we are not able to evaluate the accuracy of the described measure. By running the more flexible approach on files of our ground truth, we experienced run-time issues, which we address in Section 5.5. Since our ground truth does not contain such files and the increased run time, we decided to disable this feature per default.

5 Evaluation

In this chapter, we evaluate the performance of the ExtracTable algorithm on files taken from open data portals, and compare it with existing solutions regarding its accuracy and run time performance. First, we describe the creation of the ground truth and provide information about the baseline in Section 5.1. Based on the sub-problems stated in Section 3.4, we evaluate the qualitative performance criteria in a series of experiments dealing with line classification (Section 5.2), table range selection (Section 5.3), and line parsing (Section 5.4). Afterward, we analyze the run time of our approach in Section 5.5. We conclude the evaluation with the parameter selection in Section 5.6.

5.1 Experimental setup

Before we can evaluate the ExtracTable algorithm, we need to prepare a labeled data set. In Section 5.1.1, we explain how we created and annotated our data set consisting of plain text files taken from Mendeley Data[1], GitHub[2], and UKdata[3]. We list the guidelines that we followed during labeling and the tools we used in Section 5.1.2. Based on our annotations, we provide first insights into the data set consisting of nearly 1,000 files in Section 5.1.3. Finally, We present the baseline that we use for comparison in Section 5.1.4. Subsequent experiments have been executed in Python 3.8.5 on a Linux machine. The test system was equipped with an AMD EPYC 7702P CPU having 64 cores operating at 2 GHz and supported by 512 GB memory.

5.1.1 Data sets

Within this thesis, we aim to support ASCII tables and higher CSV table deviations from the specification RFC 4180. Therefore, we need a diverse set of plain text files containing both table formats to evaluate our ExtracTable algorithm against existing approaches. The basis of our test data set are two existing corpora from related work. One corpus was crawled from the open data portal Mendeley

[1] https://data.mendeley.com/
[2] https://github.com/
[3] https://data.gov.uk/

Data, the other one was acquired by web scraping GitHub and UKdata. In the following, we introduce the data sets and their underlying source portals. Afterward, we explain how we selected the files that we have used for labeling.

The open data portal *Mendeley Data* was founded in 2015 and is a website for sharing academic data sets. Researchers working in different domains use the platform to publish experimental data. Scientists can distribute content in various formats, including plain text. Files are grouped by projects. The content creators either store the data on one of Mendeley's servers or upload them to institutional machines[25].

Jiang et al. crawled the Mendeley data corpus in August 2020, and used it to study line and cell classification tasks on verbose CSV files [8]. The corpus includes all projects that contain at least one plain text file and were hosted on Mendeley's servers. The data set consists of 235,471 files distributed over 1,554 different projects.

Within the corpus, we found files belonging to 1,040 different extensions. The majority of files are *.txt* files (25%), followed by *.f* source code files (14%), and *.png* files (7%). csv files are on the sixth place representing 3% of the whole corpus. Not all files are of importance for our evaluation. As we are interested only in plain text files, we keep files with extensions ending in *.txt*, *.dat*, *.csv*, *.md*, and *.out*, resulting in 94,474 files.

The second corpus was provided as part of [19], which the authors used for their evaluation. It consists mainly of csv files taken from GitHub and UKdata.

GitHub is a software development hosting platform used by more than 56 million developers at the time of writing. The number of repositories exceeded 100 million. A repository typically contains a diversity of files required for the development of software. GitHub has no file type restrictions in place, allowing developers and data scientists to upload files of any kind[26].

UKData is a data-sharing platform built by the Government Digital Service of the United Kingdom. The British government uses it to publish data sets from different departments, such as education, economy, or health. As of April 2021, it contained 50,648 data sets from 14 domains[27].

The authors of [19] published a script[4] for downloading the corpus. The original data set consisted of 5,000 files each from GitHub and UKdata, for which the authors annotated a third of them automatically. We make use of the existing annotations to reduce the labeling effort. However, we kept only the manually annotated files to be more confident about the correctness of the provided

[4] https://github.com/alan-turing-institute/CSV_Wrangling/

annotations. We have been able to download[5] 2,577 files from GitHub and 2,539 files from UKdata[19].

The resulting data set contains nearly 100,000 files. Annotating all files would be too time-consuming. Thus, we have made a selection of files for this thesis. To make a fair file selection, we defined three complexity levels for files. The lowest difficulty are *simple-single* files. As the name already indicates, such files contain exactly one table. Surrounding text like notes or metadata is forbidden. Besides the table content, only empty lines may be present. We do not consider preceding or subsequent lines whose interpretation yield an empty row to be part of a table. In contrast, *complex-single* files may contain text lines that are not part of the table. The hardest files are *complex-multi* table files. They contain multiple vertically stacked tables with text in-between and before the first and after the last table. In the following we refer to the group of complex-single and complex-multi as *complex* files.

By looking at the data sets, we have been able to get first insights into the variety and complexity of files. We noticed that the Mendeley corpus provides a large variety of files. As it also represents a substantial part of our data set, we decided to grant the Mendeley data set a higher share of our final evaluation set. Furthermore, we observed that the number of simple-single files is roughly two times the number of complex files for all data sets. Thus, we decided to reflect the ratio of simple to complex files in our evaluation set. Additionally, we did not define individual targets for complex-single and complex-multi files, as the distribution depends on the data set. Instead, we specified a goal for the group of complex files. Table 5.1 shows the number of files that we initially wanted to annotate per data set and complexity level.

	simple-single	complex	total
Mendeley Data	400	200	600
GitHub	100	50	150
UKdata	100	50	150
total	600	300	900

Table 5.1: File selection targets

We selected the files randomly. In multiple iterations, we added a batch of files from a data set and manually classified their complexity. We stopped after reaching the target numbers for both complexity levels. In case that we fulfilled one goal before another, we kept adding encountered files of the fulfilled complexity level until both targets have been achieved.

[5]Last accessed 02.01.2021

We made an additional restriction for the data set from Mendeley Data as we assumed similarity between files belonging to the same project. To increase the variety of the final data set, we limited the number of complex files to one per project. However, we allowed including multiple simple-single table files from the same project, as we assumed they follow similar structures regardless of the project origin.

data set	simple-single	complex-single	complex-multi	total
Mendeley Data	400	147	51	598
GitHub	125	41	10	176
UKdata	100	79	4	183
total	625	267	65	957

Table 5.2: Number of files in the ground truth

Table 5.2 shows the final number of selected files. The numbers slightly differ from the initial targets, as we worked in iterations of different batches and kept adding encountered files.

5.1.2 Annotations

To label the evaluation data set consistently, we defined a set of guidelines. In general, we were interested in annotating all data tables. In contrast, tabular structures mainly used for layout purposes, such as dictionaries, are not of interest to us. Starting with the minimum data table requirements as defined in Section 3.1, we annotated all data type-consistent tables containing at least two columns and two rows. All rows belonging to the same table must have had the same column count. We discovered several cases where a line could have been treated as the table header for semantic reasons. Yet, a mismatching column count of the header and data part of a table led to its exclusion. The definition of data tables allows to include tables having multiple header rows. We did not restrict the maximum number of header rows.

While annotating, we found edge cases whose handling needed further specification.

1. We excluded styling lines at the beginning and end of tables from the annotation, as they do not contribute to the table content.
2. Some files contain dictionaries that are used to depict meta-data. Dictionaries represented by table structures consist of a key column and a value

column. The heterogeneous data type in the value column prevents it from being a data table. We exclude all dictionary-like structures.
3. By default, we treated empty lines as table separators. In cases where the schemas of two separated tables were compatible, we considered them to depict a single table.
4. In cases where a table could be interpreted as ASCII or CSV, we preferred the latter. Favoring the CSV format allows for a better comparison to existing table parsing solutions and makes our annotations more consistent.
5. We treated files containing more than 20 tables differently, as annotating all tables would have been infeasible. Instead, we skimmed the file and annotated one table per schema. We differentiated schemas based on their size, data types, and potential header. We flagged such files as *incomplete* and separated them from the remaining ground truth to make it possible to evaluate these files separately.
6. We had to deal with *ambiguous* text excerpts that allow for multiple valid ways of interpretation like the one shown in Figure 5.1. We admit that whether a file is ambiguous or not is subjective and depends on domain knowledge. If we were unable to find a consensus within the group, we excluded the whole file from the evaluation set.

```
THERMO
   300.000  1000.000  3000.000
AR                      AR  1     298.00   3000.00 1000.00       1 ! Burcat 16
 2.50000000E+00 0.00000000E+00 0.00000000E+00 0.00000000E+00     2
 7.45375000E+02 4.37967491E+00 0.00000000E+00 0.00000000E+00     3
 0.00000000E+00 0.00000000E+00-4.37967491E+00                    4
HE                      HE  1    300.00   3000.00 1000.00       1 ! Burcat 16
 2.50000000E+00 0.00000000E+00 0.00000000E+00 0.00000000E+00     2
 7.45375000E+02 9.28723974E-01 0.00000000E+00 0.00000000E+00     3
 0.00000000E+00 0.00000000E+00-9.28723974E-01                    4


                                      ...
```

Figure 5.1: Excerpt of an ambiguous file

To annotate the data set, we developed a labeling tool based on a web app consisting of two stages. In the first stage, we need to classify whether a file contains any tables and determine its complexity level based on the file content. After classifying the files, we can focus on the table annotation in the second stage. The process of labeling a table starts with the selection of the line range. Afterward, we may specify the parsing instructions for each line belonging to the table.

The tool provides a mechanism similar to the one Excel's text import wizard uses: We can enter the dialect components via text fields and provide the column

boundaries by clicking at the corresponding split positions. In addition to the parsing instruction, we can also specify the row type of a table. While the row type is not used as input for ExtracTable, it allows for a more meaningful evaluation later. The row type can be one of *header*, *data*, or *other*. The type *other* includes all rows that are aggregations or empty lines between header and data. While ExtracTable disregards the differences of aggregation lines, we hope for better insights in case of missing lines. To accelerate the annotation process, it is possible to apply the instructions of the current line to its successor lines. After providing the parsing instructions for all lines, we may generate a table preview to validate its correctness. Another measure to ensure the quality of the annotations is the option to flag more difficult files for peer reviews. Figure 5.2 depicts a screenshot of the annotation view.

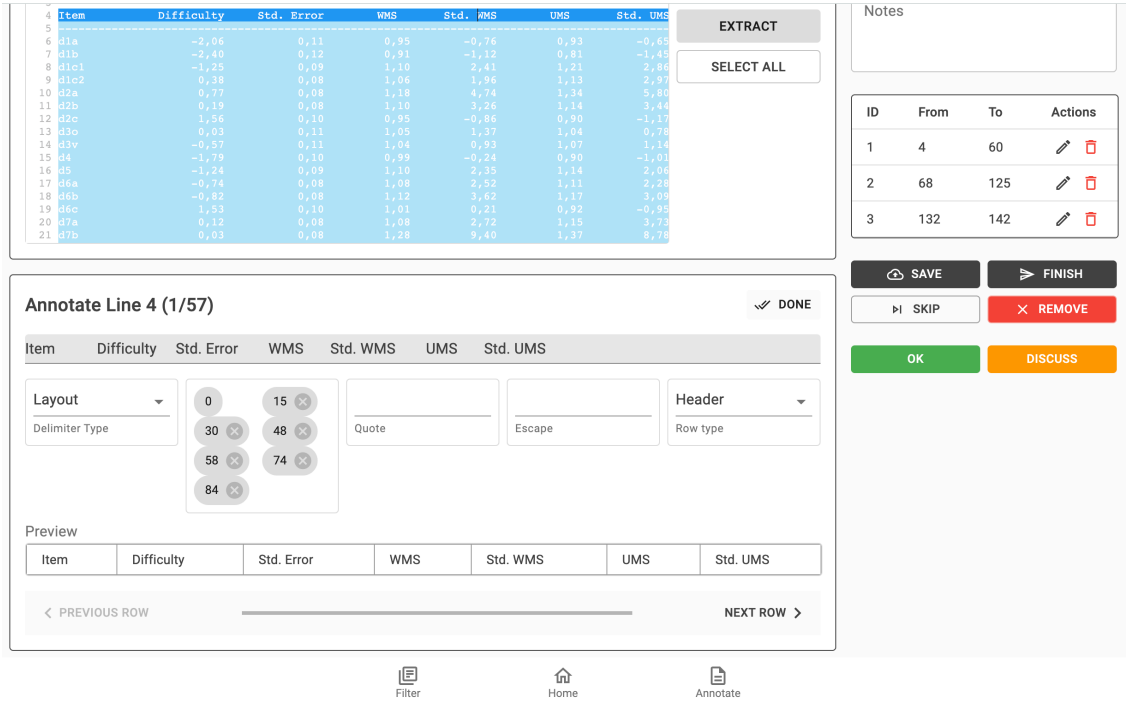

Figure 5.2: Screenshot of the annotation tool

The development of an own annotation tool allowed us to tailor the program's features specifically to our needs without compromises. This way, we quickly implemented additional features, such as the review mode or table preview. Using our annotation tool, we achieved a high degree of flexibility and high annotation quality.

5.1.3 Statistics

Based on our annotated ground truth consisting of 1,208 tables, we can obtain first insights into the data set. While complex-multi files represent only 7% of all files, they account for every fourth table. Such files contain five tables on average. In the following, we examine how a typical table of our data set looks like according to its size, structure, and variety.

A regular table of our ground truth is quite small, having less than 1,000 rows and between two and ten columns. Figure 5.3 depicts the number of tables with regard to their column and row counts that are grouped based on their order of magnitude. The longest table has ≈ 3.4 million rows, and is represented as CSV.

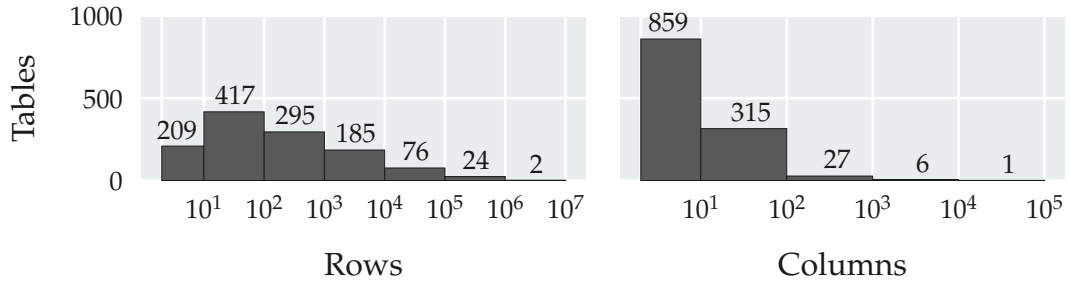

Figure 5.3: Distribution of row count (left) and column count (right)

In Section 3.2.2 we assumed that ASCII tables are primarily used for tables having few rows, as they are intended to be read by humans. The numbers confirm our assumption as ≈ 75% of the 190 ASCII tables have less than 100 rows.

Figure 5.4 shows the distribution of both table types depending on the file complexity. While CSV tables represent the biggest table share, ASCII tables are present with a significant ratio across all file complexities. Tables contained in complex-multi files are represented using ASCII more often than in other complexity classes.

To understand the underlying schema of a table, the header is of great importance. About 65% of all tables have a header which can be valuable for subsequent data preparation tasks. A header typically covers one row. One table consisted of four header rows, which also represents the maximum in our data set.

To learn about the variety of CSV dialects, we counted the number of tables complying with RFC 4180. While others also uncovered the problem that "a considerable amount of files [...] have syntax issues or use different CSV dialects"[18], there are no numbers quantifying the dimension of the problem.

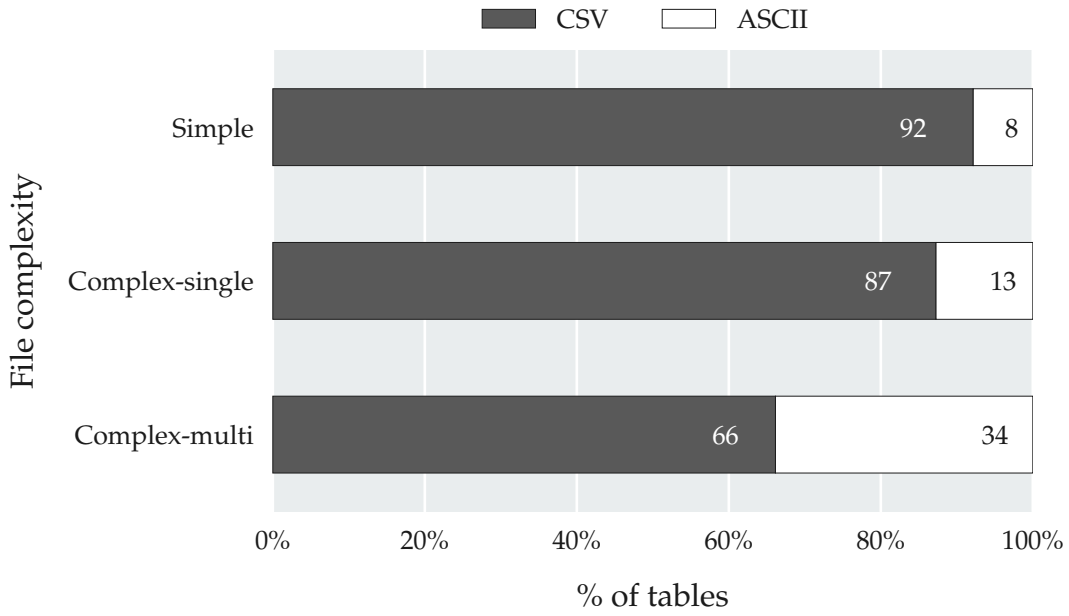

Figure 5.4: Relative number of tables represented as ASCII or CSV

We found out that 47% of the CSV tables follow RFC 4180. 1% of the files contained at least one CSV table using a multi-char delimiter, i.e., a character sequence representing an arrow (->), multiple slashes (//), or multiple tab or space characters.

Furthermore, we use the number of tables conforming to RFC 4180 as an indicator for the complexity of the corpora. Table 5.3 shows the relative number of tables that use the dialect defined in the specification to the subset of tables that uses CSV.

UKdata (%)	GitHub (%)	Mendeley Data (%)
95.4	68.0	22.9

Table 5.3: Relative number of CSV tables conforming to RFC 4180

The numbers are in line with the impression that we have made during the annotation. UKdata follows the specification in nearly all cases, which is probably because these administrative data had been carefully audited and formatted before they were published. Content uploaded to GitHub is more verbose than UKdata as the content is uploaded by many individuals, who tend to be more careless about the file format. We presume similar effects for Mendeley Data, which represents the corpus with the highest variety.

To also support more verbose files, the algorithm uses a set of regular expressions to recognize the data types of cell values, as introduced in Section 4.2. Figure 5.5 illustrates the distribution of the data types listed in Table .1. We note that the majority of fields represent numbers and that only a small portion of 4% cells did not match any of our data types (labeled as *Unknown*).

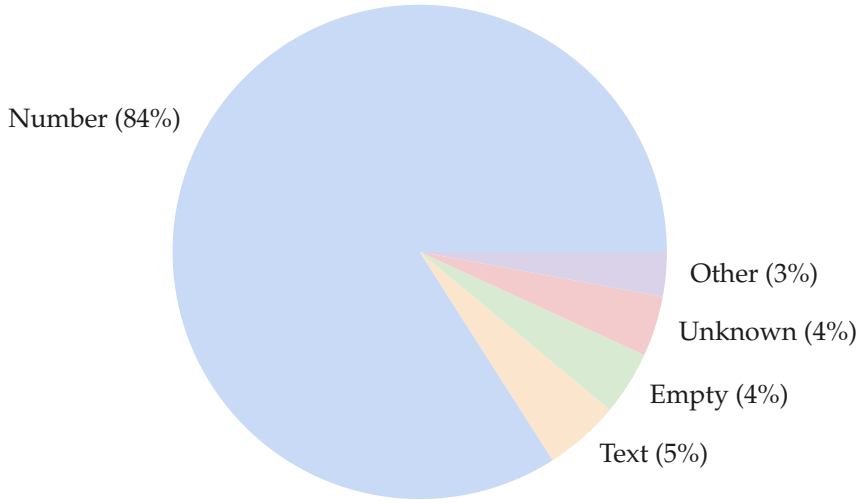

Figure 5.5: Data type distribution across all fields

5.1.4 Baselines

The baseline includes four solutions from related work, which we used to evaluate line classification, table range selection, and parsing accuracy.

The problem of detecting the proper dialects in CSV files has gained importance over the past years. We used Python's built-in CSV Sniffer, Hypoparsr, and CleverCSV to evaluate the performance of the line parsing. Additionally, we included a solution always returning the dialect specified in RFC 4180. To simulate approaches only taking advantage of structural features, we added a modified version of ExtracTable that ignores the cell content.

The *Sniffer* class is part of the CSV package[6] of Python. The method of the algorithm is explained within the source code. According to its documentation[7],

[6] `https://docs.python.org/3/library/csv.html`
[7] `https://github.com/python/cpython/blob/3.8/Lib/csv.py#L280`

Sniffer infers the delimiter by character frequencies across lines. It then selects the character that is the most suitable as a delimiter according to the frequency mode. It processes the file content in chunks and stops as soon as it is confident that it found the correct delimiter. We used the version published along with Python 3.8.5.

Hypoparsr[18] was published by Döhmen et al. in 2017 covering multiple parsing steps such as file encoding detection, dialect detection, and table area detection. While the R package got removed from the Comprehensive R Archive Network by the authors, we used the archived version 0.1.0 from GitHub[8]. By reverse-engineering parts of the source, we extracted the full dialect components, including the delimiter, escape, and quote characters.

The latest software, published in 2019, is *CleverCSV*[19] that proposes a pattern-based approach to infer the dialect of a file. 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[9].

We added a solution that always returns the dialect specified in RFC 4180 (see Table 3.3). By including this baseline when evaluating the parsing results, we were able to gain insights into the complexity of files and the dialect distribution.

Lastly, we simulate the baselines that rely exclusively on the table's structure by including a modified version of ExtracTable that does not exploit the cell contents. Therefore, we introduced a new data type that matches all values. This way, the data type consistency in columns is disabled, and the table selection is restricted to optimize for large tables using the existing tie-breaking criteria. We refer to this approach in the following as *Structure-based*.

We used *Pytheas*[12] for our evaluation related to table discovery. The approach classifies lines belonging to CSV files automatically and infers the table ranges by applying a set of fuzzy rules. We used the Python implementation published by the authors on GitHub[10]. Additionally, Pytheas required a set of weights to infer the table ranges for which we used the weights provided by the authors.

Unfortunately, all other solutions that we have investigated could not be used to evaluate the line classification and table range selection. They either assumed only a single table to be present within a file, or they did not return the explicit table ranges. Therefore, we came up with a *naive* approach that simulates the missing baselines by classifying the complete file content as table lines.

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

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

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

5.2 Line classification

In this section, we focus on the classification of lines as *table lines* or *non-table lines*. We compare our solution to Pytheas and the naive approach in a series of three experiments.

The first measurement is about the binary classification metrics on line-level, where the accuracy and F1-score are of biggest interest. Because of the imbalance between the number of table lines and non-table lines, we calculated the *balanced accuracy* as mentioned in [28], which is the average of true negative rate and recall. The true negative rate is calculated by the number of correctly classified non-table lines divided by the total number of non-table lines. The balanced accuracy equally weights the accuracy between table-lines and non-table lines. It returns a number between zero and one, where higher balanced accuracies mean that table and non-table lines have been similarly well classified.

We assume that the quality of the line classification depends greatly on the file's complexity. We expect all solutions to perform very well on *simple-single* files consisting only of table lines and empty lines. Files belonging to *complex-single* are still mostly csv tables but include a preamble such as meta-data, which might confuse the algorithms. The *complex-multi* files are the most difficult ones, as they can contain more text.

Table 5.4 shows the F1 score and balanced accuracy for ExtracTable, Pytheas, and the naive approach per file complexity level. For each solution, we excluded the files that have not been processed successfully within three minutes. This selection was independent of whether other approaches were able to deal with the files. This decision should be considered when looking at the following numbers. Pytheas finished $\approx$ 79% of all files, whereas ExtracTable processed $\approx$ 87% successfully. The naive approach worked on all files due to its nature. We further elaborate the completion rates in Section 5.5.

	Simple-single		Complex-single		Complex-multi	
	F1	bal. Acc.	F1	bal. Acc.	F1	bal. Acc.
Naive	0.999	0.500	0.927	0.500	0.945	0.500
Pytheas	**1.000**	0.944	0.949	0.909	0.828	0.637
ExtracTable	0.999	**0.999**	**0.997**	**0.987**	**0.973**	**0.796**

Table 5.4: F1 score and balanced accuracy per file complexity (higher is better)

As expected, all three solutions achieved excellent F1 scores on *simple-single* files. Only the balanced accuracies differ. Because of the naive approach's design, the

balanced accuracy was always 0.5 as it detects all table lines but no non-table lines. When looking at *complex-single* files, the numbers of the naive approach show that the numerical margin to improve on is very small. The ExtracTable algorithm performed best, yielding a very high F1 score of 0.997, and worked well on both line classes. The performance differences increased for *complex-multi* files. The ExtracTable algorithm still performed best, yielding an F1 score of 0.973. Yet, its balanced accuracy dropped to 0.796. The score can be divided into the accuracy of 0.997 for table lines and 0.595 for non-table lines. We suspect that the imbalance between table lines and non-table lines affects the parsers in a way that causes them to classify lines as table lines predominantly. The naive approach outperformed Pytheas, which might be because of the following two reasons: (1) The ratio of ASCII tables within complex-multi files is higher and Pytheas is not optimized for this table format. (2) Pytheas requires training of a set of weights. We used the weights provided by the authors, which might not be generic enough.

The second experiment considers the line classification problem on table-level. For each annotated table, we computed the share of lines that got correctly classified as table lines. Figure 5.6 shows the relative number of tables per quota of recognized table lines. The numbers for the approaches are based on all files, including the ones on which the solutions did not finish.

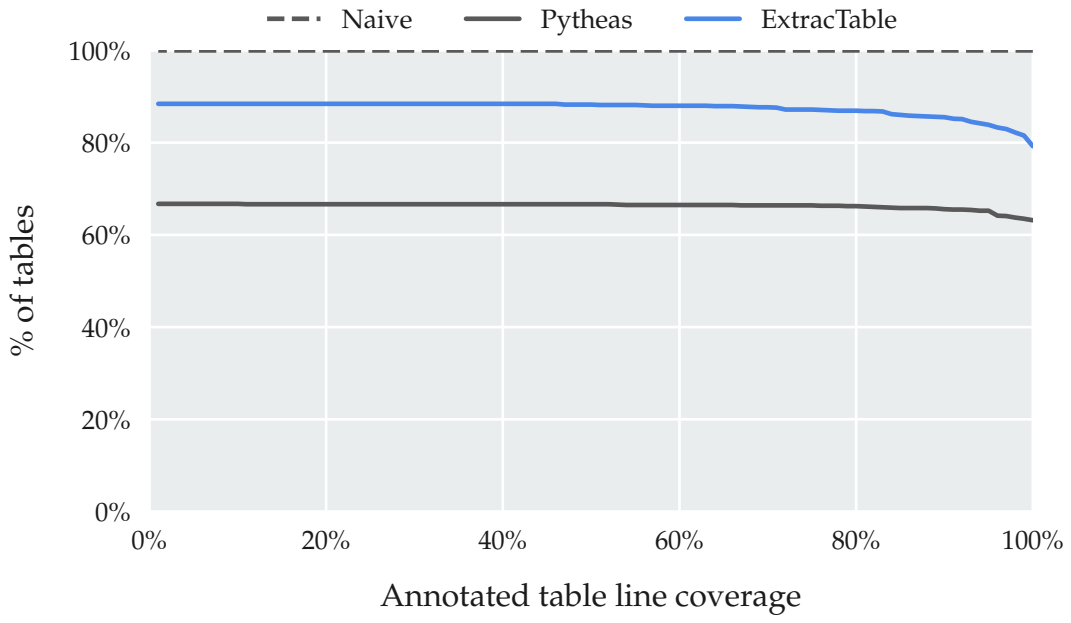

Figure 5.6: Line classification performance (higher is better)

The naive approach classified everything as a table line, which is why it always

returned the best-possible result of 100%. The initial offsets close to 0% coverage for ExtracTable and Pytheas exist as they did not finish on all files. In nearly 80% of the cases, our solution returned all lines of a table. However, the chart does not account for the table size. Therefore, if the algorithm was missing a few lines of a table, it had a stronger impact on short tables than on tables having more rows.

Lastly, we examine what kind of rows have been misclassified as non-table lines by the ExtracTable algorithm. Only 0.02% of the *data* rows have been missing. *Header* rows have been found in 90.1% of the cases, which is important for understanding tables. The algorithm found rows of type *other* less often. Only 78.1% of such rows have been classified correctly as table-line. We presume that this observation mainly relies on empty lines and indicates, that some tables have been over-segmented or that the header was detected to be a table on its own. In general we expected a shift towards data rows, as they typically dominate the table content, which is true especially for tables having a higher number of rows.

We conclude that ExtracTable performs best according to the binary classification metrics. Yet, we recognize the line classification problem to be rather complex. The imbalance between the two classes complicates a more profound evaluation. Pytheas did not perform as well as expected, which we suspect to be caused by the pre-trained weights being to specific and the presence of ASCII tables. The header and data parts of a table have been included in most of the cases. Empty rows and aggregations that we annotated to be part of a table have not been detected as often as the other two types. We investigate the impacts of line classification results on the table range selection in the next section.

5.3 Table range selection

In the following, we evaluate the quality of the table range selection according to the third sub-problem defined in Section 3.4. Similar to the previous chapter, we compare the results of ExtracTable to Pytheas and the naive approach.

We measured the quality of the table range selection by calculating the *Intersection over Union* (IOU) for each pair of detected and annotated tables as explained in [29]. The IOU metric is suitable for this evaluation context because it is a common similarity measure for bounding boxes. In a paper published by Dong et al., 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, a similarity measure for sets. It returns a

number between 0 (no match) and 1 (perfect match). Given a returned table f and an annotated table a, the Jaccard index $jac(f,a)$ is calculated using:

$$jac(f,a) = \frac{|\{k|from_f \leq k \leq to_f\} \cap \{k|from_a \leq k \leq to_a\}|}{|\{k|from_f \leq k \leq to_f\} \cup \{k|from_a \leq k \leq to_a\}|} \tag{5.1}$$

After calculating the Jaccard index for each table pair, we used the maximum Jaccard index to determine the match type. We distinguish between four match types: perfect match, partial match, no match, and eager match.

Annotated tables that returned a Jaccard index of one for some returned table are a *perfect match*. In contrast, if the maximum Jaccard index of an annotated table is zero, *no match* was found. All remaining annotated tables fall into the category of *partial matches*. In these cases, multiple scenarios are possible. The annotated table could be: (1) only partially covered, (2) split into multiple smaller tables, (3) spanning multiple tables, or a combination of these three.

Some of the returned tables have no matching annotated table that they would get mapped to. We refer to such a returned table as *eager match*.

Figure 5.7 shows the match type counts for ExtracTable and Pytheas considering all annotated tables. Again, we took only the files into account that have been finished by the parsers.

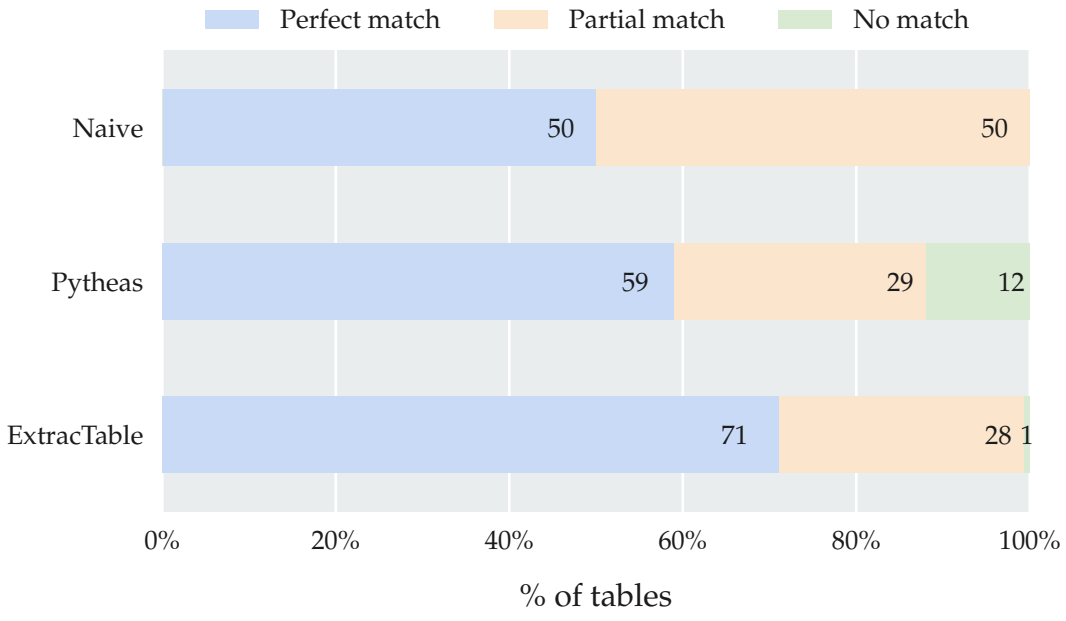

Figure 5.7: Table range selection performance (higher number of perfect matches is better)

The naive approach returned the correct range for exactly 50% of the tables, equal to the number of simple-single files without surrounding empty lines. The remaining half was classified as a partial match, as every file contains at least one table. Due to the nature of the naive approach, there are no missing tables. Pytheas was able to detect 59% of the table ranges correctly, yet the approach missed every ninth annotated table. The tables that have not been recognized are of different sizes and are equally balanced regarding their formats. 6% of all tables returned by Pytheas have been returned eagerly and are not present in the ground truth.

ExtracTable identified the correct table ranges in more than 70% of all tables, which is the highest number compared to Pytheas and the naive approach. It missed seven tables (1%), which we looked at more closely. We have not been able to defer a dominant pattern of why the tables have been missing, yet four tables had several sparse columns. Two of such scattered tables have been matrixes. We show an example of a missing file in Figure 5.8.

```
2 NA 3 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA 4 NA 5 NA NA NA NA NA NA NA NA NA 23 NA NA NA NA NA NA
NA NA 6 NA 7 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA 8 NA 9 NA NA NA NA NA NA NA NA 21 NA NA NA NA NA
NA NA NA NA 10 NA 11 NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA 12 NA 13 NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA NA 14 NA 15 17 NA NA NA NA NA NA 19 NA NA NA NA
NA NA NA NA NA NA NA 16 NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA NA NA 18 NA NA NA NA NA NA NA NA NA NA NA NA NA


                                    ...
```

Figure 5.8: Table that was too sparse to be considered consistent

A flaw of the ExtracTable algorithm is the high number of eager matches that are not depicted in the chart. Nearly one out of every sixth tables returned by ExtracTable is not present in our ground truth. After manually examining a sample of eagerly matched tables, we realized that it found consistent tables within dictionary fragments. Additionally, our parser detected tabular structures within single-column tables, whose content it could interpret as two columns for a subset of rows.

When comparing ExtracTable to Pytheas, we note that the former recognized table ranges more precisely. The latter seems to work more conservative as it rather not recognizes tables instead of returning eager matches.

To gain further insights into partial matches, we used a different representation of the data. Figure 5.9 depicts the ratio of annotated tables exceeding a reference Jaccard index.

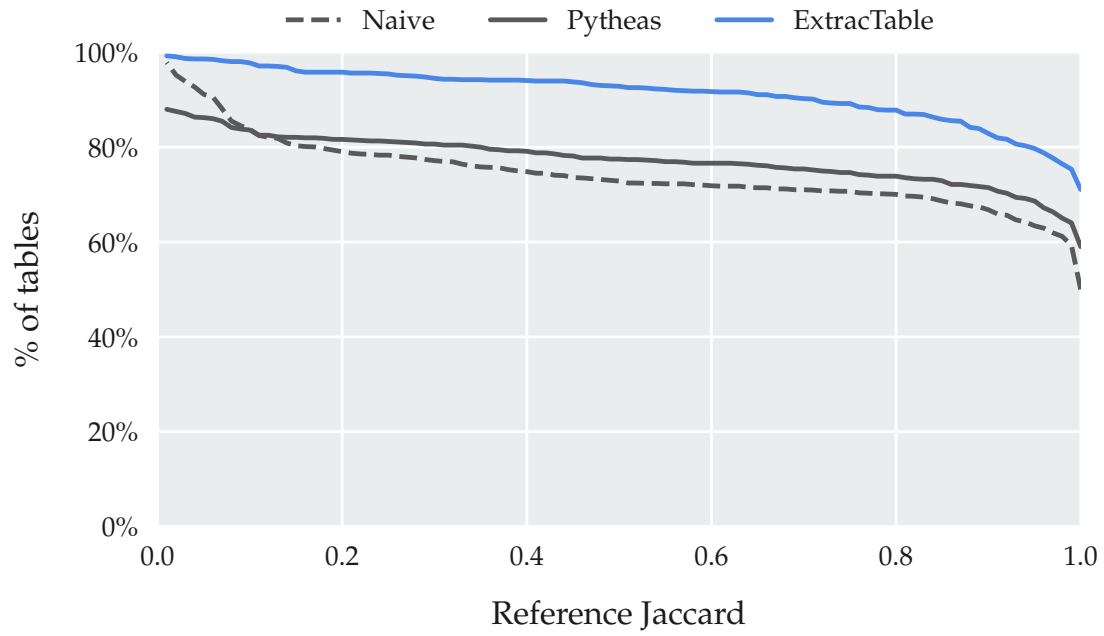

Figure 5.9: Number of tables exceeding a reference Jaccard (higher is better)

Our ExtracTable algorithm returns the most accurate table ranges. We derive that even for the partial matches, the algorithm returns line ranges close to our annotations, as the trend for ExtracTable decreases slowly. We manually looked at files having a low Jaccard index (≤ 0.2). In all of the cases, the tables have either been under-segmented or over-segmented, resulting in a low Jaccard index.

When defining the class of correct table matches, we counted tables having a perfect Jaccard index of one. However, requiring a perfect Jaccard index might be too strict - a table is missed as long as there is at least one incorrect line therein. Using a lower Jaccard index threshold allows to count more correct table matches as correct. For example, when any match with a Jaccard index of more than 0.9 is treated as a correct table match, ExtracTable would return the correct table ranges for 83%. However, determining a fair threshold would be difficult. As the question for the appropriate threshold remains unanswered, we introduce our last measure for the table range selection, which copes with this issue: the mean Average Precision *mAP*.

For a particular Jaccard index threshold, the *average precision* is calculated by determining the area under the precision-recall curve. Typically, the *mAP* is used for ranked results, where each result has a different precision and recall. We have only one data point since we are dealing with an unranked classification.

We interpolate the precision and recall of that point to draw the precision-recall curve. Given a Jaccard threshold *JAC*, the precision *prec* and recall *rec* need to be calculated prior plotting the curve. The x-axis depicts the minimum recall ranging from zero to one. The y-axis represents the yielded precision. For every x smaller than *rec*, y equals to the precision *prec*. For all $x > rec$ y equals to zero.

The *mAP* is derived by calculating the mean of the average precisions for a set of thresholds. The higher the returned score is, the better the algorithm is at detecting the correct table ranges. To calculate the *mAP* for our data set, we use the same detection evaluation metrics as for the COCO[11] data set [31]. This means that our set of thresholds includes $\{0.5, 0.55, ..., 0.95\}$. The area under the curve is calculated by averaging the precision for the recall thresholds $\{0, 0.01, ..., 1]\}$. The results show that the ExtracTable algorithm performs best as it yields a *mAP* of 0.881. Pytheas is second best with 0.725 followed by the naive approach with 0.699.

To conclude, the ExtracTable algorithm provides most precise table ranges among all candidates. For two-thirds of the tables, it detects the exact ranges. When allowing for slight deviations, four out of five table ranges are recognized correctly. However, our solution also returned the highest number of eager matches. As the parsing of files often serves as a pre-processing step within a more complex pipeline, it would be preferable to grasp as many true table ranges as possible at the cost of obtaining many false positive table ranges, which could be filtered out in the downstream processing.

5.4 Line parsing

We dedicate this section to the last series of quality experiments dealing with the accurateness of the line interpretation, which covers the first two sub-problems stated in Section 3.4.

To count the number of correct matches, we compared the results of the parsers line-wise with our annotations. Following the definition of *correct* as introduced in Section 3.4, "[a]n interpretation is correct, if it produces fields exactly like in the underlying data table." It follows that we checked whether the parsed fields were equal to the values in the ground truth and have been provided in the correct order for each line. If an annotated table contains an empty line, which may occur between the header and the data, we accepted all interpretations consisting of a variable number of blank fields. This exception was necessary

[11] `https://cocodataset.org/#detection-eval`

as such rows occurred for both table types: CSV tables always result in a single empty cell, while ASCII interpretations result in multiple blank fields equivalent to the column count of the surrounding table. Additionally, some tables can be interpreted using either ASCII or CSV and still yield the correct interpretation. However, the CSV interpretation might have additional empty columns, where *all* field values are the empty string. In such scenarios, we still aimed to accept both solutions. To make them comparable to our ground truth, we allowed empty columns in an annotated table to be absent.

We compared ExtracTable to five other solutions from our baseline: RFC 4180, Hypoparsr, Structure-based, Sniffer, and CleverCSV. Some aspects of the parser, such as the handling of space characters in-between fields, are implementation-specific. Therefore, we 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 from the parser implementations. Lastly, to reduce the impact of incorrectly classified lines, we took only the parsing results into account, which the algorithms returned for correctly classified table lines. The first experiment is about the parsing correctness per table format. Figure 5.10 shows the ratio of fields that have been correctly parsed belonging to CSV and ASCII tables.

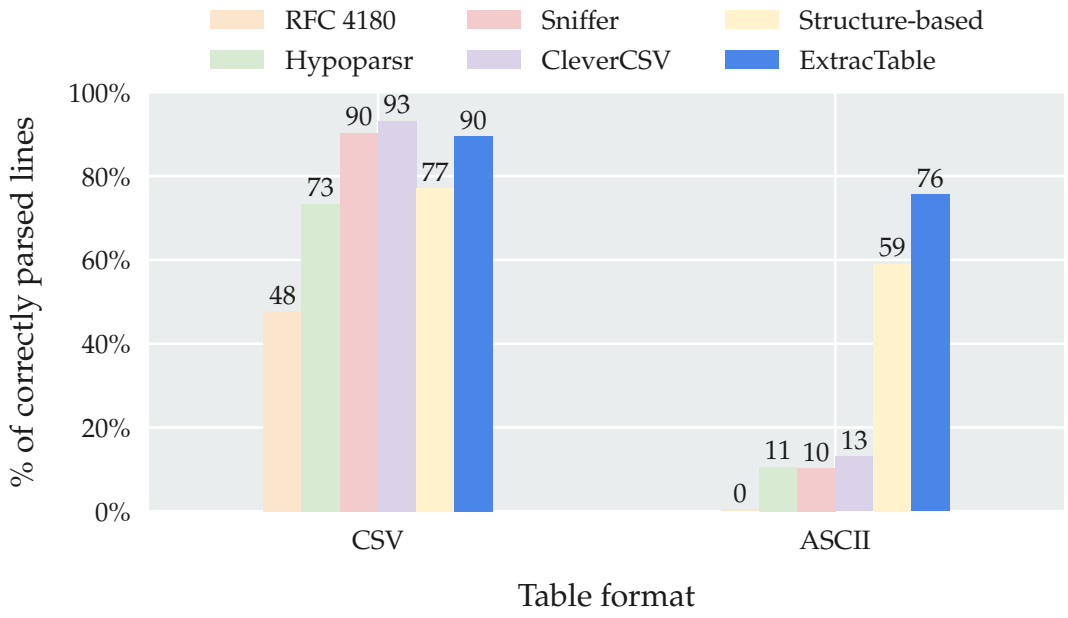

Figure 5.10: Parsing accuracy (higher is better)

When looking at the group of bars representing CSV, we note that Sniffer, CleverCSV, and ExtracTable performed similarly well and detected the correct parsing

instructions in about nine out of every ten cases. We are surprised that Sniffer, which relies on heuristics, accomplished such good results. Yet, the numbers are comparable to the ones stated in the paper published by Döhmen et al.[18]. Regarding CSV tables in complex-multi files, CleverCSV and Sniffer might have benefited in cases where all tables belonging to the same file employ the same dialect, as they assume only one table to be present. ExtracTable achieved lower results, as it interpreted some tables as ASCII instead of CSV. The reduced version Structure-based, which relies only on the tables structure, performs better than Hypoparsr. We presume that this is because Hypoparsr uses a limited set of potential delimiters.

Besides the Structure-based approach, ExtracTable is the only solution optimized for ASCII tables, which is why the remaining solutions recognized merely a small subset of lines. Nevertheless, it is interesting to see that they returned a few correct interpretations. To answer the question of how this is possible, we manually looked into the corresponding files. We identified three reasons that led to the proper representation of single lines. The first reason was empty lines occurring for a small subset of tables between the header and data part of a table. All parsers returned a single blank field regardless of the applied dialects as the field delimiter was absent within the line, which we considered correct. The second reason lies within the nature of ASCII tables as they use a different number of spaces to separate columns. Solutions other than 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*, which is likely to happen for tables having few columns. The last reason leads back to tables that 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. While we aimed at annotating these files using the corresponding dialect, we annotated a handful of files using column boundaries. Independent from these corner cases, we note that ExtracTable could interpret 76% of the lines appearing in ASCII tables.

The differences between the Structure-based approach and ExtracTable are roughly ten percent for CSV and 17% for ASCII tables. We conclude that the cell content is a valuable feature and should be considered when parsing tables. Yet, even when taking the cell contents into account, ExtracTable can be further improved.

In the following, we present the results of our error analysis per table type. In general, the errors made by ExtracTable have been independent of the table format as they led back to the table selection. A line has been interpreted incorrectly mainly due to three reasons: 1) over-segmented and under-segmented tables; 2) short texts surrounding tables; 3) misinterpretation of tables using

tab characters. Tables have been over-segmented, when they contained partially sorted or similar values. The algorithm under-segmented tables if it could find a parsing instruction that could be applied on both tables. As the table selection prefers tables with higher row counts, it merged both tables in such cases. Short texts surrounding the tables, such as table titles, caused ASCII tables to merge the first columns as the algorithm tried to include the header row. ExtracTable misinterpreted CSV tables when it found a dialect applicable to the table and the surrounding text. We traced back both reasons to our design decision to prefer tables having a higher row count; Tables delimited by the tab character sometimes have been wrongly interpreted using ASCII. As we noted that this was true for a higher number of misinterpretations, we decided to further look at this problem in the following experiment.

Within our second experiment, we measured the impact of the ASCII table detection on the parsing accuracy of CSV tables by disabling the feature. Before, the algorithm parsed 89.5% of the CSV table lines correctly. After disabling the column boundary detection, this number increased to 94.2%. On the one hand, this means that our dialect detection, in theory, works better than the ones of CleverCSV and Sniffer. On the other hand, we note that the overall accuracy can be lower when adding more flexibility. While this is not satisfying, it appears logical as additional flexibility can be a case for misinterpretations.

In a smaller experiment, we evaluated the qualitative impact of the data type recognition feature. By disabling the recognition of known data types, the algorithm relies only on the atomic pattern components consisting of *number*, *string*, and *other* as described in Section 4.2. When ExtracTable did not recognize the data type, 78% of the lines were interpreted properly, which is slightly better than the Structure-based approach.

To summarize the results, we acknowledge that CleverCSV yields the highest parsing accuracy for CSV files, followed by ExtracTable and Sniffer. Hypoparsr did not perform that well, yet it outperformed the RFC 4180 baseline. Using ExtracTable's parsing accuracy for CSV tables as starting point, we can confirm the independence of the general scoring method as it also works well on ASCII tables. We assume that the parsing accuracy could be further enhanced by pruning non-table lines, as we see them as a source of errors.

5.5 Run time

In the following, we evaluate how the various flexibility levels affect the performance of the ExtracTable algorithm. We measured the run time of the called

process using Linux's internal system call `getrusage` for *RUSAGE_CHILDREN* within the parent.

In the first experiment, we compared the run time of our approach to the ones of RFC 4180, Sniffer, Hypoparsr, CleverCSV, and Pytheas. To reduce the overall run time of our experiments, we used a timeout of three minutes. For each file, we recorded whether the file was processed successfully and the processing time. To make the run time comparable, we kept only files finished by all parsers. While this could add a bias towards more simple files, it ensures a fair comparison. Figure 5.11 shows the resulting run time per line in milliseconds based on 551 files. We use a logarithmic scale for the y-axis to increase the expressiveness of the chart. To guarantee that all values start at 10^0, we added 1 to all values.

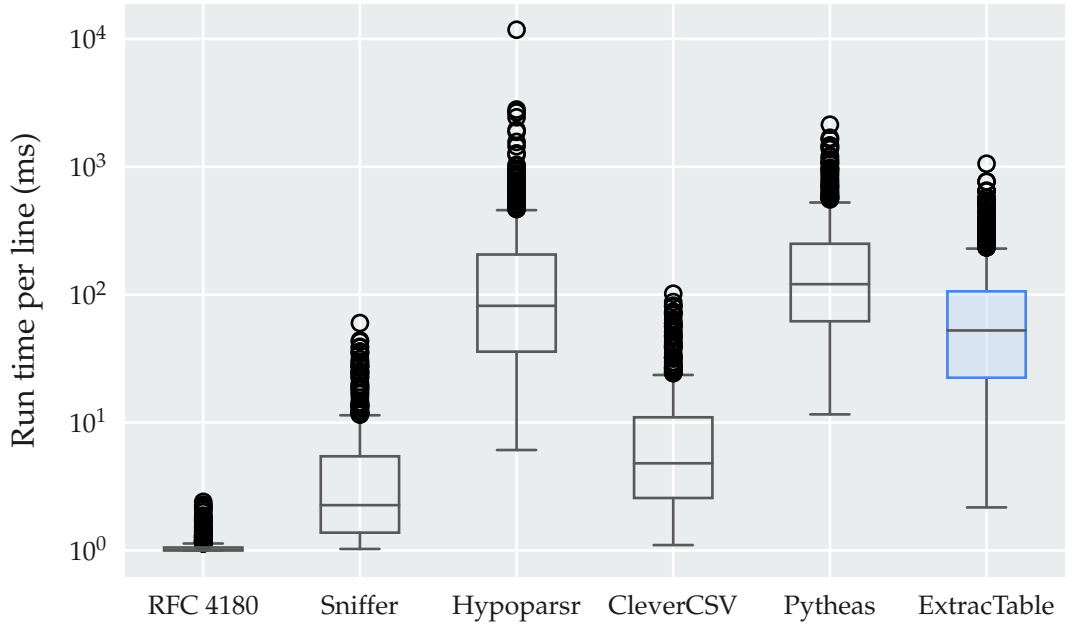

Figure 5.11: Run time comparison using logarithmic scale (lower is better)

The solution that always returns the configuration of RFC 4180 required a constant time as it did not even read the file contents. The observed outliers are probably caused by noise coming from the system. Including the run time for RFC 4180 helps to assess the performance of the other approaches. Sniffer and CleverCSV are both very fast and needed 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 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. Because of these additional features, we accept that ExtracTable is slower than CleverCSV and Sniffer but note that it is faster than Pytheas and Hypoparsr.

Due to the run time differences, the solutions processed different amounts of files. The number of completed files are shown in Table 5.5. Besides the number of complete and timed out files, we also present the number of inconclusive files for which the parsers could not offer a result.

	Success	Timeout	Inconclusive
RFC 4180	969	0	0
Sniffer	899	1	69
Hypoparsr	699	23	247
CleverCSV	946	17	6
Pytheas	692	160	117
ExtracTable	844	125	0

Table 5.5: Number of completed files per parser

The primary purpose of including these numbers is to put the metrics presented in the previous evaluation chapters into perspective. We note the high number of inconclusive returns of Hypoparsr and Pytheas. The former was caused by a preliminary file check done by the developers, which aborts the process if the file size exceeds 400 kB. The latter appeared to be an internal bug within their published code, causing no features to be found within the file content.

There was one file not processed by any of the parsers apart from the RFC 4180 baseline. This file consisted only of 450 lines, but each line had 17,365 characters. Hypoparsr and Sniffer returned no result, while CleverCSV, Pytheas, and ExtracTable timed out.

Our approach could not process 125 (13%) files of the ground truth within the allowed time frame of three minutes. The complexity levels of the missed files reflect the complexity distribution of the ground truth.

The number of files that ExtracTable was able to finish, and therefore its run time, depends on two aspects: One driver for longer run time 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. Because of the different aspects influencing the run time, it was impossible to identify the main driver. Instead, we evaluate the impact of various features on the run time next.

As a second experiment, we evaluate the run time impact of the data type recognition, the ASCII table detection, as well as the complexity add-on described in Section 4.5 and the table selection heuristic described in Section 5.3. For each feature, we ran two measurements: once enabled and once disabled it. We kept all remaining parameters the same. To measure the run time for the complexity plugin, we used $P_{csl} = 50$. Similar to the first experiment, we use a threshold of three minutes for all test runs. To calculate the speed factor, we compared the run time of files that have been processed in both runs. The measurement results are depicted in Table 5.6.

Feature	Speed up factor	Timed out
ASCII disabled	1.28	0
Data type recognition disabled	1.11	1
Table selection heuristic enabled	1.11	0
Complexity add-on enabled	0.82	131

Table 5.6: Speed up when enabling or disabling certain features

From the numbers, we derive that enabling the ASCII detection caused the algorithm to be 28% slower. This slowdown is due to the effort put in detecting column boundaries and considering a higher number of parsing instructions. Disabling the data type recognition had not a crucial effect on the overall run time. However, as shown in Section 5.4, it does affect the parsing accuracy. Using the table selection heuristic affects the run time in the same way as disabling the data type recognition has. Yet, it turned out not to impact the accuracy of ExtracTable significantly. When allowing single tables to be represented by multiple parsing instructions, the general performance suffers a lot. 131 files, which have been processed before, timed out when allowing for more flexibility. The remaining files took nearly 20% more time to finish. In the end, we have not seen any tables requiring multiple parsing instructions. Since the increased run time, we disabled the complexity add-on in practice.

5.6 Parameter selection

In the following, we explain how we selected the parameter set P of the ExtracTable algorithm. We need to specify ten parameters, where half of them are specific to the end-user task.

Parameters, such as the minimum table dimensions, are subjective and depend on the specific task. We do not tweak these parameters as they do no impact the overall correctness. Table 5.7 depicts the values that we chose for them.

Parameter		Value
P_{mrc}	Min row count	2
P_{mcc}	Min column count	2
P_{mdl}	Max dialect component length	4
P_{del}	Delimiter exclusion list	all bracket characters
P_{mhr}	Max header rows	4

Table 5.7: Task-specific parameters

Following the definition of data tables, we require at least two columns and two rows to be present. We restrict the components of CSV dialects to have a max of four characters and forbid them to include any brackets. Based on related work[32], we allow tables to have max 4 header rows.

To find the optimum for the remaining five parameters, we ran a grid search on a subset of our ground truth consisting of 168 files. We created the data set by selecting all complex files from Mendeley Data that require less than three minutes to process. We chose only complex table files as we assume that the ideal parameters for complex tables should also account for simple-single table files. Additionally, we assess that the Mendeley corpus is more heterogeneous than the other data sets. By running the grid search, we want to find the optimal values for P_{msr} and the consistency parameters P_{mbc},P_{mbs},P_{mcs}, and P_{mts}. As the consistency parameters are independent of the parameter P_{msr}, we ran two individual grid searches.

The P_{msr} parameter is an integer, which defines when an insignificant spacer covers enough text lines to be no longer considered to exist coincidental. Thus, it affects only the parsing results of ASCII tables. We test nine different configurations, ranging from the minimum number of rows required for data tables (two), to ten.

The second grid search is about the consistency parameters, which are values between zero and one. They mainly impact what the algorithm considers as a

table and how it selects the table ranges. We tested different configurations for each value ranging from 0.1 to 0.9 in steps of 0.2. We use odd numbers since 0.5 has a special meaning when comparing the consistency of two rows. As the parameters affect each other, we need to test all 625 combinations. In the worst case, an iteration testing a single configuration on all files takes six minutes when parallelizing the grid search using all 128 CPU threads. In practice, it finishes faster as not all files require three minutes of processing time, and subsequent tests can run immediately.

After running both grid search experiments, we analyzed the results to return the *optimal* parameters. We favored configurations that result in a high line classification accuracy, a high Jaccard average, and high parsing correctness. We punished the ones that caused a higher number of missed tables or eager table matches. The five criteria have equally contributed to the parameter selection. Table 5.8 depicts the resulting values which we used for all other experiments.

Parameter		Value
P_{msr}	Min significant rows	4
P_{mbc}	Min block compatibility	0.71
P_{mbs}	Min block score	0.31
P_{mcs}	Min column score	0.51
P_{mts}	Min table score	0.51

Table 5.8: Resulting parameters

6 Conclusion

To properly extract information from table files, it is indispensable to understand the tabular structures of rows and cells. In this work, we identified three unresolved issues regarding the extraction of tables from plain text files taken from open data portals. These issues are caused by the lacking support for:

1. Files containing multiple tables
2. CSV tables deviating more from RFC 4180
3. ASCII tables

Solutions optimized for CSV files, such as Pytheas[12] and CleverCSV[19], solve the table discovery and table parsing problems individually. Yet, no combined solution exists working on files containing multiple tables. We found that 25% of the tables contained in our ground truth come from files having multiple tables. Because of the different implementations of CSV, a wide variety of dialects is used. According to our ground truth, only 47% of the CSV tables follow the specification formulated in RFC 4180. Existing approaches focus on files that use a single character symbol for dialect components. However, we propose a solution that covers files deviating more from RFC 4180 by supporting multi-character dialects. We placed no restrictions on dialect components other than that they need to be non-alphanumerical. Additionally, we discovered tables not represented as CSV, but as ASCII, which represent 16% of all tables in our ground truth. As such tables contain valuable information, too, we added the support for the extraction of such tables to our algorithm. We emphasize that, to our best knowledge, no related work exists that covers the extraction of such ASCII tables.

Figure 6.1: Workflow of ExtracTable

We presented the workflow of our ExtracTable algorithm, which tackles the identified challenges in a series of six steps as depicted in Figure 6.1. After reading the file content, the algorithm uses a custom parser to detect and test possible dialects on the fly. A set of column boundaries is used to interpret ASCII tables, which the algorithm discovers by aligned spaces occurring in subsequent lines. After applying the found parsing instructions onto the line content, the algorithm infers the data type of each field. ExtracTable then builds

table candidates based on the consistency of data type patterns, field count, and parsing instruction. Finally, the algorithm returns the best table candidates by mapping the table selection challenge to the shortest path problem.

To evaluate our algorithm, we annotated a data set consisting of nearly 1,000 files taken from Mendeley Data, GitHub, and UKdata. We analyzed three aspects of our algorithm: (1) the classification of lines into table and non-table lines, (2) the table range selection, and (3) the parsing accuracy. Our evaluation showed that ExtracTable outperformed Pytheas at the line classification and classified all lines for 80% of the tables correctly. ExtracTable was also better at determining the table ranges, as it detected the correct range for more than 70% of the tables. When we compared the parsing results, we found that CleverCSV performed best on CSV files and worked on 93% of the lines. Yet, ExtracTable performed similarly well and parsed 90% correctly. Our solution was the only one capable of parsing a significant amount of ASCII tables and achieved an accuracy of 76%. While the added flexibility allowed to handle more files properly, it showed its downside in the run time analysis as it requires 90 ms per line on average. Yet, it was still faster than Hypoparsr and Pytheas.

To reproduce all results, we published the source code on GitHub[1] including the ExtracTable algorithm, annotation tool, evaluation, and baseline scripts. The annotations are available upon request.

To encourage others to test our algorithm without setting up anything, we developed a demo web app. Interested readers may upload a file to learn about the capabilities and limitations of ExtracTable. Figure 6.2 shows a screenshot of the website. We added the source code of the web app to the GitHub repository.

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

[1] `https://github.com/deeps96/ExtracTable`

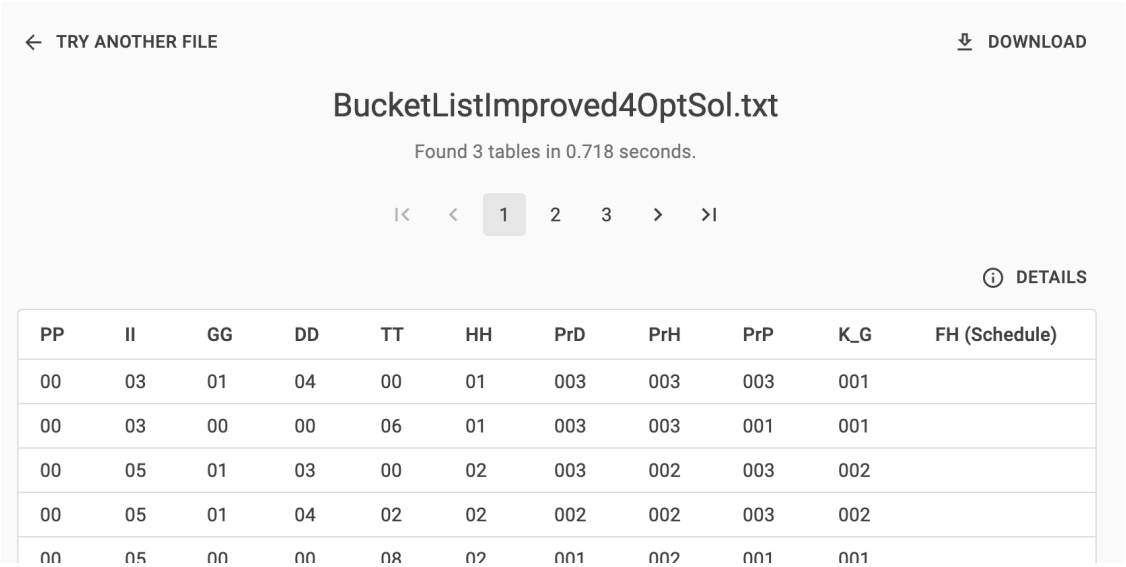

Figure 6.2: Screenshot of the demo web app

Using the ExtracTable algorithm, data scientists can extract tables from a wider variety of plain text files. Thus, they spend less time dealing with data wrangling and instead can focus on their actual tasks. While the evaluation returned good results already, we are still far away from handling files fully automatically. The imminent data preparation problems that need to be tackled are data integration and data transformation, according to [20].

Appendix

References

[1] IBM Corporation, *IBM @ twitter.com*, 2013. [Online]. Available: https://twitter.com/ibm/status/396295485373566976 (visited on 04/22/2021).

[2] D. Reinsel, J. Gantz, and J. Rydning, "Data Age 2025 : Don 't Focus on Big Data; Focus on the Data That's Big," *IDC White Paper*, no. April, pp. 1–25, 2017.

[3] Landesregierung Nordrhein-Westfalen, "Gesetz zur Förderung der elektronischen Verwaltung in Nordrhein-Westfalen (E-Government-Gesetz Nordrhein-Westfalen - EGovG NRW)," p. 15, 2020.

[4] P. Mooney, *2018 Kaggle Machine Learning & Data Science Survey*. [Online]. Available: https://www.kaggle.com/paultimothymooney/2018-kaggle-machine-learning-data-science-survey (visited on 04/21/2021).

[5] Anaconda, "2020 State of Data Science," Tech. Rep., 2020, p. 12.

[6] M. Chessell, F. Scheepers, N. Nguyen, R. van Kessel, and R. van der Starre, "Governing and Managing Big Data for Analytics and Decision Makers," *IBM Redguides for Business Leaders*, p. 28, 2014.

[7] G. Press, "Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says," *Forbes Tech*, pp. 4–5, 2016.

[8] L. Jiang, G. Vitagliano, and F. Naumann, "Structure Detection in Verbose CSV Files," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2021, pp. 193–204.

[9] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC Editor, RFC 4180, 2005.

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

[11] J. Mitlöhner, S. Neumaier, J. Umbrich, and A. Polleres, "Characteristics of Open Data CSV Files," 2016, pp. 72–79.

[12] C. Christodoulakis, E. B. Munson, M. Gabel, A. D. Brown, and R. J. Miller, "Pytheas: Pattern-Based Table Discovery in CSV Files," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2075–2089, 2020.

[13] P. Pyreddy and W. B. Croft, "TINTIN: a system for retrieval in text tables," in *Proceedings of the ACM International Conference on Digital Libraries*, 1997, pp. 193–200.

[14] A. C. e Silva, A. Jorge, and L. Torgo, "Automatic Selection of Table Areas in Documents for Information Extraction," in *Progress in Artificial Intelligence*, 2003, pp. 460–465.

[15] J. Hu, B. Laboratories, L. Technologies, M. Avenue, and M. Hill, "Medium-Independent Table Detection," *Evaluation*, no. January, pp. 1–12, 2000.

[16] A. e Silva, A. Jorge, and L. Torgo, "Design of an end-to-end method to extract information from tables," *Document Analysis and Recognition*, vol. 8, pp. 144–171, 2006.

[17] S. Saurav and P. Schwarz, "A machine-learning approach to automatic detection of delimiters in tabular data files," in *Proceedings - IEEE International Conference on High Performance Computing and Communications, IEEE International Conference on Smart City and IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*, 2017, pp. 1501–1503.

[18] T. Döhmen, H. Mühleisen, and P. Boncz, "Multi-Hypothesis CSV Parsing," in *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2017.

[19] G. J. J. van den Burg, A. Nazábal, and C. Sutton, "Wrangling messy CSV files by detecting row and type patterns," *Data Mining and Knowledge Discovery*, vol. 33, no. 6, pp. 1799–1820, 2019.

[20] A. Nazabal, C. K. Williams, G. Colavizza, C. R. Smith, and A. Williams, "Data engineering for data analytics: A classification of the issues, and case studies," *arXiv*, 2020.

[21] D. Brickley, J. Tennison, and I. Herman, *CSV on the Web Working Group @ Www.W3.Org*, 2016. [Online]. Available: `http://www.w3.org/` (visited on 04/23/2021).

[22] J. Rovegno and M. Fenner, *Index @ csvy.org*, 2018. [Online]. Available: `https://csvy.org/` (visited on 04/23/2021).

[23] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, "Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts," in *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*, 2011, pp. 65–74.

[24] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Algorithmen - Eine Einführung*. De Gruyter, 2017, pp. 663–667.

[25] *Mendeley Data Home*, 2021. [Online]. Available: `https://data.mendeley.com/` (visited on 04/02/2021).

[26] Github, *About @ Github.Com*, 2021. [Online]. Available: `https://github.com/about` (visited on 04/02/2021).

[27] *Search @ data.gov.uk*. [Online]. Available: `https://data.gov.uk/search` (visited on 04/02/2021).

[28] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The balanced accuracy and its posterior distribution," in *Proceedings of the International Conference on Pattern Recognition*, 2010, pp. 3121–3124.

[29] H. Rezatofighi, N. Tsoi, J. Y. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," *arXiv*, pp. 658–666, 2019.

[30] H. Dong, S. Liu, S. Han, Z. Fu, and D. Zhang, "TableSense: Spreadsheet table detection with convolutional neural networks," *AAAI Conference on Artificial Intelligence, AAAI, Innovative Applications of Artificial Intelligence Conference, IAAI and the AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI*, pp. 69–76, 2019.

[31] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8693 LNCS, no. PART 5, pp. 740–755, 2014.

[32] D. W. Embley, M. S. Krishnamoorthy, G. Nagy, and S. Seth, "Converting heterogeneous statistical tables on the web to searchable databases," *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 19, no. 2, pp. 119–138, 2016.

Data types

Name	Description
Boolean	true or false (ignoring case)
Brackets	Anything in-between a pair of round, box, curly, or angle brackets
Currency	Number preceeded or subseeded by a currency character
Date	Date and/ or time in various formats Date d/m/yy, dd/mm/yyyy, m/d/y,mm/dd/yyyy, yy-m-d, yyyy-mm-dd Date-separator can be one of space, slash, dot, minus Time: hh:mm:ss Time-separator can be any non-word character Partially taken from RegexBuddy
Domain name	Taken from RegexBuddy
E-Mail address	Taken from RegexBuddy
Empty	Empty string or values that represent a missing value.
File path	Matching absolute and relative paths using / or \ as delimiter
Hash	A string consisting of upper or lower case characters, digits, and underscore At least one number and English letter required.
IP address	IP address in IPv4 format - taken from RegexBuddy
Number	Signed, unsigned floats and integers. Supports scientific notation
Percentage	Number followed by the percentage sign
String	Sequence of English letters (ignoring case) - minimum length: 1
Text	Multiple strings separated by ␣,.:&%?!–"'/()[]
URL	Created by diegoperini[1]

Table .1: List of data types

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die zusätzlich in digitaler Form eingereichte Version ist mit der vorliegenden gedruckten Arbeit identisch.

Datum

Unterschrift